89 | 354

Sc N 89/382 B

Université de NANCY I
Faculté des Sciences
UER de Mathématiques
Département d'Informatique

Centre de Recherche en Informatique de Nancy
C.R.I.N.
B.P. 239
54506 Vandœuvre Cedex

Assistance et pilotage dans le développement de logiciel :

Vers un modèle de description.

Khalid BENALI

UNIVERSITE * BIBLIOTHEQUE-SCIENCES * NANCY

Doctorat de l'Université de NANCY I
en Informatique

Composition du jury

M. Jean-Paul Haton	INRIA-Lorraine	Président du jury
M. Alfred Strohmeier	Ecole Polytechnique de LAUSANNE	Rapporteur externe
Mme Odile Foucaut	Université de NANCY II	Rapporteur interne
M. Jean-Claude Derniame	Université de NANCY I	Directeur de thèse
M. Amaury Legait	GIE EMERAUDE	Examinateur
M. Claude Godart	Université de NANCY I	Invité

Doctorat soutenu publiquement **le 24 novembre 1989**

Université de NANCY I
Faculté des Sciences
UER de Mathématiques
Département d'Informatique

Centre de Recherche en Informatique de Nancy
C.R.I.N.
B.P. 239
54506 Vandœuvre Cedex

Assistance et pilotage dans le développement de logiciel :

Vers un modèle de description.

UNIVERSITE * NANCY * BIBLIOTHEQUE-SCIENCES

Khalid BENALI

Doctorat de l'Université de NANCY I
en Informatique

Composition du jury

M. Jean-Paul Haton	INRIA-Lorraine	Président du jury
M. Alfred Strohmeier	Ecole Polytechnique de LAUSANNE	Rapporteur externe
Mme Odile Foucaut	Université de NANCY II	Rapporteur interne
M. Jean-Claude Derniame	Université de NANCY I	Directeur de thèse
M. Amaury Legait	GIE EMERAUDE	Examinateur
M. Claude Godart	Université de NANCY I	Invité

Doctorat soutenu publiquement le **24 novembre 1989**

A mes parents,
sans l'éducation et l'amour desquels rien n'aurait été possible.

Avant-Propos

Cette thèse est le résultat de quatre années d'activité 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 puis dans l'équipe du projet ALF, pour l'encadrement et le suivi de mon travail, pour l'ambiance de travail et de coopération qu'il sait faire régner au sein de son équipe et enfin pour ses qualités humaines qui font que *"le chef"* est plus qu'un chef.

Je tiens à remercier Monsieur le professeur Jean Paul Haton, professeur à l'université de Nancy I et directeur de recherches à l'INRIA, d'avoir accepté de présider le jury de cette thèse.

Je tiens à exprimer ma gratitude à Madame le professeur Odile Foucaut, professeur à l'université de Nancy II, 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 remercie Monsieur le professeur Alfred Strohmeier, professeur à l'Ecole Polytechnique Fédérale de Lausanne, 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 définitive de celui-ci.

Tous mes remerciements à Monsieur Amaury Legait, responsable du projet ALF, de s'être intéressé à mon travail.

Merci aussi à Monsieur Claude Godart qui m'a toujours prodigué conseils et encadrements bénéfiques tout au long de cette thèse.

Ce travail a été réalisé en partie dans le cadre du projet ESPRIT ALF [1] et est donc partiellement financé par la CEC que je tiens à remercier ici, ainsi que les tous les collègues

1. We are indebted to all the members of the ALF Consortium, for fruitful discussions and advice. The partners involved in the project ALF are GIE Emeraude (France), CIG-INTERSYS (Belgium), Computer Technologies Co.-CTC (Greece), Grupo de Mecánica del Vuelo, S.A. (Spain), International Computers Limited (United Kingdom), University of Nancy-CRIN (France), University of Dortmund-Informatik X (Germany), Cerilor (France), Université Catholique de Louvain (Belgium) and University of Dijon-CRID (France). This work is partially sponsored by the Commission of the European Communities under the ESPRIT programme (Project Ref. No. 1520).

avec lesquels j'ai travaillé dans le cadre de ce projet et avec lesquels j'espère poursuivre le travail dans la même ambiance de sérieux et de cordialité.

Je tiens en outre à remercier toutes les personnes du CRIN et en particulier de l'équipe qui ont, d'une manière ou d'une autre, contribué à l'aboutissement de ce travail. Je pense en particulier à Nacer Boudjlida, à François Charoy, à tous les autres "génies du logiciel" et à Mme Marchand secrétaire efficace de l'équipe.

Merci aussi à Monsieur Hervé Coilland, chef du département informatique de l'IUT de NANCY II, et aux autres enseignants de l'IUT, qui m'ont accueilli parmi eux pendant quelques temps.

Merci enfin à Catherine Tisseron pour sa relecture attentive et pour bien d'autres choses encore.

Table des matières

Avant-Propos 3

Introduction 11

I LA PROBLEMATIQUE 17

1 Vers des ateliers logiciels supportant des modèles de processus de développement 19

1.1 Introduction 19

1.2 Pourquoi des structures d'accueil ? 21

1.2.1 Nécessité des environnements de programmation 21

1.2.2 La fabrication des PSE nécessite des investissements importants . . 23

1.2.3 Les structures d'accueil : la solution économique pour la fabrication des IPSE ou AIGL 24

1.3 La gestion des objets dans une structure d'accueil 25

1.3.1 L'apport des bases de données 25

1.3.2 Les structures d'accueil à gestion d'objets 30

1.4 Etat de l'art 36

1.4.1 Les SGBD pour le génie logiciel 36

1.4.2 Les AIGLS 36

1.4.3 Les structures d'accueil 36

1.5 La prise en compte des modèles de processus de développement dans un atelier de logiciel 38

1.6 Conclusion 40

II GESTION DE L'INFORMATION EN GENIE LOGICIEL : PREMIERE ETAPE VERS LES ATELIERS DE GENIE LOGICIEL DE TROISIEME GENERATION 43

1 Systèmes de Gestion de l'information associée au développement de logiciel 45

1.1 Introduction . . . 45
1.2 Le projet SKB . . . 45
1.2.1 Introduction . . . 45
1.2.2 Les objectifs du système SKB . . . 46
1.2.3 Présentation du modèle SKB . . . 47
1.2.4 Implantation du modèle SKB . . . 48
1.2.5 Conclusion . . . 49
1.3 Le projet PMDB . . . 49
1.3.1 Introduction . . . 49
1.3.2 Le cahier des charges de PMDB . . . 50
1.3.3 Le Modèle . . . 50
1.3.4 Le prototypage . . . 51
1.3.5 Conclusion . . . 54
1.4 Conclusion . . . 54

2 Atelier Intégré de Génie Logiciel : un exemple de réalisation 57

2.1 Concerto en 1986 . . . 58
2.1.1 Présentation générale . . . 58
2.1.2 Le système de communication . . . 59
2.1.3 Le système opérant . . . 61
2.1.4 Le système d'information . . . 68
2.1.5 Le système de pilotage . . . 73
2.2 Concerto aujourd'hui . . . 74
2.3 Conclusion . . . 75

3 Les Structures d'accueil actuelles 79

3.1 Introduction . . . 79
3.2 PCTE . . . 80
3.2.1 Introduction . . . 80

3.2.2 Les mécanismes de base . . . 80
3.2.3 Conclusion . . . 90
3.3 PACT . . . 91
3.3.1 Introduction . . . 91
3.3.2 Les outils de base de PACT . . . 91
3.3.3 Conclusion . . . 92
3.4 CAIS . . . 93
3.4.1 Introduction . . . 93
3.4.2 Le modèle nodal . . . 93
3.4.3 Les entrées/sorties . . . 97
3.4.4 Les utilitaires . . . 98
3.4.5 La structure de travail . . . 98
3.4.6 Conclusion . . . 98
3.5 Conclusion . . . 98

4 Tableaux comparatifs des systèmes étudiés 99

4.1 Projets . . . 99
4.2 Gestion d'Objets . . . 100
4.3 Gestion d'objets complexes ou composites . . . 100
4.4 Objets de taille importante . . . 100
4.5 Définition de nouveaux types de données . . . 101
4.6 Langage de requête . . . 101
4.7 Espaces de travail, vues et schémas . . . 101
4.8 Expression de contraintes sémantiques . . . 102
4.9 Mécanismes de triggering . . . 102
4.10 Gestion des événements en tant qu'objets . . . 102
4.11 Gestion de données historiques . . . 103
4.12 Gestion de Version/Configuration . . . 103
4.13 Gestion d'activités . . . 103
4.14 Contrôle d'accès . . . 104
4.15 Gestion de communication . . . 104
4.16 Conclusion . . . 105

III PROPOSITION D'UN MODELE POUR LE PROCESSUS DE PRODUCTION DE LOGICIELS (SOFTWARE PROCESS) : DEUXIEME ETAPE VERS LES ATELIERS DE GENIE LOGICIEL DE TROISIEME GENERATION 107

1 La modélisation du processus de production de logiciels 109

1.1 Introduction . . . 109
1.2 Modèles utilisés dans les systèmes de gestion de l'information des projets présentés . . . 110
1.3 Prise en compte de la dynamique dans les projets étudiés . . . 112
1.4 Objectifs du contrôle et de l'assistance en développement de projet dans ALF 113
1.5 Assistance dans la production de logiciels . . . 114
1.5.1 Développement de logiciels . . . 114
1.5.2 Gestion de projet . . . 116
1.6 Les motivations pour la modélisation des processus de production . . . 117
1.7 Différentes approches pour une modélisation du processus de développement 120
1.7.1 La modélisation de la dynamique dans les systèmes d'information . . 120
1.7.2 La programmation de processus . . . 123
1.7.3 Le projet ISTAR . . . 127
1.7.4 Le projet MARVEL . . . 131
1.8 Conclusion . . . 134

2 Le modèle que nous proposons : le MASP 135

2.1 Introduction . . . 135
2.2 Présentation des Composants du modèle . . . 136
2.3 Les composants du MASP . . . 139
2.3.1 Le Modèle d'objets . . . 139
2.3.2 Le Modèle d'opérateurs . . . 143
2.3.3 Les Expressions . . . 151
2.3.4 Les règles . . . 154
2.3.5 Ordonnancement des opérateurs . . . 156
2.3.6 La caractéristique . . . 159
2.4 Interaction entre les composants . . . 160
2.5 Relation entre le modèle (MASP) et le processus modélisé (ASP) . . . 162

2.5.1 Instanciation d'un MASP . . . 163
2.6 Conclusion . . . 178

3 Un exemple et une réalisation 179

3.1 Présentation de l'exemple . . . 179
3.2 le MASP PRODUCTION DE LOGICIELS . . . 180
3.3 Le MASP-fils GESTION . . . 182
3.4 Le MASP-fils DEVELOPPEMENT . . . 187
3.5 La mise en œuvre de l'exemple . . . 195
3.6 conclusion . . . 200

Bilan 203

A Bibliographie 209

B Chronologie des publications à des congrès dans le cadre de ce travail 223

Introduction

Le domaine général de cette étude est le Génie Logiciel. Elle a été réalisée au Centre de Recherche en Informatique de Nancy (CRIN) au sein de l'équipe Génie Logiciel de Monsieur le professeur Jean-Claude Derniame, et une partie de ces travaux a été financée par la Commission des Communautés Européennes dans le cadre du projet Esprit ALF [2].

Pour situer en quelques mots le projet ALF, disons que son objectif est de proposer une nouvelle génération d'interfaces compatibles avec PCTE [PCT86], en vue de permettre l'implantation de l'assistance et de la prise d'initiative dans les Ateliers Intégrés de Génie Logiciel (AIGL). Le consortium ALF regroupe à la fois des industriels et des universitaires :

- le GIE Emeraude regroupant Bull, Eurosoft et Syseca (Paris, France),
- CIG-INTERSYS S.A. (Bruxelles, Belgique),
- Computer Technologies Co.-CTC (Athènes, Grèce),
- Grupo de Mecánica del Vuelo, S.A. (Madrid, Espagne),
- International Computers Limited (Reading, Grande Bretagne),
- L'université de Nancy-CRIN (Nancy, France),
- L'université de Dortmund-Informatik X (Dortmund, RFA),
- Cerilor (Nancy, France),
- L'université Catholique de Louvain (Louvain-la-Neuve, Belgique),
- L'université de Dijon-CRID (Dijon, France).

Des chercheurs des différents partenaires participent donc à ce projet commun. En particulier, ALF est un projet sur lequel des membres de l'équipe Génie Logiciel du CRIN ont travaillé ou travaillent encore. Des travaux ont été réalisé en DEA sur les idées proposées dans le cadre du projet :

2. This work is partially sponsored by the Commission of the European Communities under the ESPRIT programme (Project Ref. No. 1520).

- Une maquette basée sur les concepts de pré- et post-condition d'un opérateur à été réalisée dans [GER87].

- une étude des réflexes (attachement procédural) dans les langages à base de frames (Systèmes de Représentation des Connaissances) pour la vérification et la propagation des Contraintes d'Intégrité dans les Systèmes d'Information des Ateliers de Génie Logiciel a été menée dans [MAR88].

D'autre part, des travaux sont actuellement en cours concernant l'implantation d'une couche objet au-dessus de PCTE [CHA90], et l'étude d'un modèle pour le processus de production de logiciels [DER89].

Après cette brève description du contexte de travail, venons-en à la thèse proprement dite.

Pour bien comprendre la structuration de celle-ci et ses objectifs, nous allons commencer par présenter la démarche que nous avons suivie.

La difficulté qu'il y a à développer un logiciel ne fait de doute pour personne. Il suffit pour s'en convaincre de constater les retards chroniques à la livraison d'un logiciel et la part prise par la maintenance dans son prix de revient. Le plus souvent cela n'est pas dû à une mauvaise organisation mais à la grande complexité que représente la gestion de la production d'un logiciel. En effet, un logiciel est un vaste ensemble d'objets de types différents et issus de toutes les étapes du cycle de vie d'un logiciel : document de spécification, de conception, module, interface, programme source, code objet et auteur en sont quelques exemples. Le nombre de composants manipulés augmentant de façon exponentielle au fur et à mesure que l'on progresse dans le cycle de vie, il est très difficile d'avoir une vue cohérente de tous les objets contribuant à la réalisation d'une étape du cycle de vie et quasiment impossible d'avoir une vue complète de l'ensemble des objets composant le logiciel et de leurs relations.

Il est donc naturel que la production de logiciels passe par la structuration des objets en une Base de Données. Les objets de cette base sont gérés (décrits, instanciés et modifiés) par une collection d'opérations qui constitue le Système d'Information (SI) associé à la production de logiciels. Une telle base est un puissant outil de développement et de suivi de projet.

Cette gestion de l'information associée à la production de logiciels est, à notre avis, essentielle. Nous nous sommes donc intéressés à quelques études représentatives ayant cet objectif. C'est la raison d'être de la partie II de cette thèse : focaliser sur une fonctionnalité commune et incontournable des environnements de production de logiciels, à savoir le stockage et la manipulation des informations liées à la production de logiciels. Cette fonctionnalité sera étudiée en détails dans la deuxième partie de cette thèse. La partie II subira donc un "éclairage système d'information".

Cette gestion de l'information a été envisagée et réalisée de différentes façons :

- Des projets comme SKB et PMDB (cf chapitre 1 de la partie II) ont essayé de réaliser cette gestion en couplant un SI avec un système de développement. La conclusion tirée des études de ces projets a été que le couplage d'un tel SI avec un système de développement ne pouvait pas être une solution convenable pour le Génie Logiciel (redondance d'information, incohérence,...).

- Une autre approche étudiée est l'intégration au sein d'un atelier des fonctionnalités de gestion et de développement de logiciels. Ainsi nous nous intéressons à l'Atelier de Génie Logiciel paramétré Concerto, regroupant à la fois la gestion de l'information et le développement de logiciel. Celui-ci nous est apparu comme le plus représentatif de sa génération (et aussi comme le plus accessible vu son développement en FRANCE par le CNET). Nous l'avons donc étudié en détails et en particulier le SI incorporé en son sein (cf. chapitre 2 de la partie II). Cette étude, bien qu'ayant mis l'accent sur les avantages d'une telle approche d'atelier paramétré par rapport au simple couplage SI/système de développement, ne nous a pas moins montré ses limites et le coût prohibitif en résultant (nécessité de développer au moins un type d'atelier par type d'application, couplage SI/reste de l'atelier plus ou moins lâche,...).

- Les deux approches précédentes tendaient à nous montrer que la solution aux problèmes du développement de logiciel n'était ni un système de développement couplé à un SI, ni un atelier paramétré, mais l'utilisation d'une structure d'accueil. Nous avons donc étudié les deux structures d'accueil actuelles à savoir PCTE et CAIS (cf. chapitre 3 de la partie II). Cette dernière étude montre qu'effectivement un système de gestion d'objets au sein d'une structure d'accueil est, à notre avis, le noyau de base idéal pour le développement d'AIGLs, et que PCTE était le mieux placé par rapport à son concurrent américain dans ce domaine : les structures d'accueil représentent de manière tout à fait satisfaisante l'aspect statique du développement de logiciel.

Toute la deuxième partie de la thèse a été rédigée en respectant la démarche initiale, à savoir l'amélioration de la production de logiciels par le biais d'une structuration des données, et d'une manipulation de celles-ci de manière adéquate. C'est la raison pour laquelle cet aspect Système d'Information a été hypertrophié dans la présentation d'ateliers tels que Concerto et que l'aspect Système de Gestion d'Objets (SGO) est au centre de la présentation de PCTE et CAIS.

Nous avons tiré deux conclusions de cette première étude des Systèmes d'Information pour le Génie Logiciel :

- Un SGO au sein d'une structure d'accueil est la solution pour la gestion de l'information au sein d'un atelier de développement de logiciels. Ceci est mis en évidence par

l'évaluation comparée des capacités de représentation de l'aspect statique des différents systèmes étudiés (cf. chapitre 4 de la partie II).

- La prise en compte de l'aspect statique de la production de logiciels est assurée de manière satisfaisante dans une structure d'accueil, mais l'aspect dynamique de cette production n'est pas suffisamment traité.

Cet aspect dynamique de la production de logiciels commence à être ressenti comme crucial par un grand nombre de chercheurs. Ce centre d'intérêt est communément appelé "Software Process" [3] mais c'est une activité assez neuve, par conséquent une quantité d'autres appellations sont aussi usitées pour désigner les mêmes concepts (processus de production de logiciels, processus de développement de logiciels, ou encore processus de génie logiciel).

Nous nous sommes intéressés aux concepts développés dans un certain nombre de projets se préoccupant de la dynamique dans le développement de logiciel (Istar, Marvel, Arcadia,...), aux concepts mis à jour dans un certain nombre d'études du projet ALF [BEN89a] [BEN89b] [DER88] [GOD88] [GOD89a] et aux concepts développés pour modéliser la dynamique dans les SI comme Remora et Merise pour proposer un modèle pour le processus de production de logiciels.

Cet aspect dynamique fera l'objet de la partie III de cette thèse. Les études réalisées dans ce cadre se feront dans l'optique "processus logiciel". Nous ne reviendrons plus sur l'aspect statique, même si certains projets comme Arcadia intègrent cette fonctionnalité. Nous n'avons pas voulu donner le même éclairage aux projets étudiés pour l'aspect statique et aux projets étudiés pour leur aspect dynamique car il est difficile de trouver des critères d'évaluation significatifs pour les deux domaines. D'autre part le SGO de PCTE peut difficilement être comparé au SGO embryonnaire d'un projet s'intéressant à l'aspect processus logiciel, comme il est difficile de comparer la description sommaire de PCTE d'un outil ("static context") à la définition de Marvel décrivant les activités en termes de préconditions et de postconditions.

Après la présentation de quelques projets significatifs dans le domaine de la modélisation de processus logiciels, nous présentons le MASP (Model for Assisted Software Process), modèle que nous sommes en train d'élaborer dans le cadre du projet ALF. Cette présentation se veut succincte et cohérente. Que le lecteur ne s'attende pas à trouver la description complète du langage de modélisation du processus de production : d'une part celui-ci n'est pas encore complétement figé, et d'autre part ce langage a pour noyau de base le langage de description et de manipulation de SDS de PCTE+ qui est, lui même, en cours d'extension avec des concepts nouveaux comme, par exemple, les attributs multivalués ou les domaines énumérés comme types d'attributs. Par conséquent, nous préférons

3. "Processus logiciel"

baser notre description sur un "langage pédagogique" qui tout en permettant de représenter l'essentiel des concepts présentés sera assez éloigné du langage conçu dans le cadre de ALF [BEN89c]. Nous terminerons enfin cette troisième partie par la description d'une maquette dont le but est de proposer un support pour l'étude du comportement d'un système mettant en œuvre quelques-uns des concepts décrits dans cette thèse.

La thèse s'articulera de la manière suivante :

PARTIE I Cette partie présente de manière plus détaillée que ne le fait cette courte introduction la problématique à laquelle nous nous sommes confrontés.

PARTIE II Une deuxième partie est axée sur les techniques de construction d'Ateliers de Génie logiciel en utilisant les deux grandes voies possibles en dehors d'un couplage "ad hoc" entre un système d'exploitation et un système d'information :

- le développement d'ateliers paramétrés,
- et le développement de Structures d'Accueil.

Quel que soit l'approche retenue, la gestion de l'information statique en est la partie centrale. Nous avons donc insisté sur cet aspect de gestion de l'information en génie logiciel. Les différentes approches que nous avons donc traitées sont :

- couplage plus ou moins fin d'un système de gestion de données au système de développement de logiciels *(e.g. SKB, PMDB)*;
- développement d'ateliers complets paramétrés *(e.g. CONCERTO)*;
- développement de Structures d'Accueil regroupant les fonctionnalités de base permettant la construction d'AIGLs *(e.g. CAIS, PCTE)*.

PARTIE III Une troisième partie est centrée sur la proposition d'un modèle de production de logiciel, en s'appuyant sur des projets existant et sur différentes études menées dans le cadre de ALF.

Les objectifs de cette thèse, après une étude bibliographique, ne sont pas d'arriver à la solution en ce qui concerne la modélisation du processus de production de logiciels mais à une solution acceptable, et même seulement à un début de solution. Notons que, dans le cadre du projet ALF, deux ans sont encore prévus pour parvenir à une version figée du modèle que nous ébauchons dans cette thèse.

PARTIE I

LA PROBLEMATIQUE

1

Vers des ateliers logiciels supportant des modèles de processus de développement

1.1 Introduction

Développer un logiciel important nécessite, pour chaque activité intervenant dans le développement, l'utilisation d'outils et de méthodes de travail nombreux et variés. Citons-en quelques-uns pour montrer la réelle diversité des alternatives :

- modélisation de systèmes parallèles par les réseaux de Petri et utilisation de leurs propriétés formelles pour un certain nombre de vérifications ou de simulations qui peuvent être réalisées par des outils spécifiques;
- spécification du problème à résoudre en termes de types abstraits [DER79];
- programmation avec tel ou tel langage de programmation "classique" (PASCAL, ADA, C,...), de programmation fonctionnelle (essentiellement Lisp [WIN84] et ses dialectes (le_lisp [CHA86], Common lisp,...), de programmation logique (Prolog [COL82],...), de programmation objet (Smalltalk [GOL83],...) ou même de programmation par acteurs.

Ces outils communiquent entre eux : ils s'échangent, au moins, leurs résultats. Mener un développement de logiciel du début à la fin ne peut consister en l'utilisation de tel outil sur telle machine pour la conception, de tel autre sur une autre machine pour le codage et enfin d'un troisième outil sur une tierce machine pour la gestion des informations pertinentes du projet.

De même qu'un atelier de construction de voitures centralise les fonctions d'assemblage, de peinture et de test au même endroit, nous appellerons Atelier de Génie Logiciel (AGL ou PSE -Programming Support Environment-) ou Atelier Intégré de Génie Logiciel (AIGL

ou IPSE -Integrated Programming Support Environment-) l'intégration sur le même site de différents outils pour le développement de logiciel. Comme il existe différents outils et différents concepts à chacune des étapes de développement de logiciels, nous verrons des AIGLs de différentes natures, suivant les différents choix faits dans leur construction.

Les ateliers de la première génération -les premiers à être apparus sur le marché- sont des "boîtes à outils" style Unix, affectant à chacune des fonctionnalités requises dans le développement de logiciel un outil indépendant qui la réalise.

Ceux de la seconde génération visent à accroitre l'intégration des outils et leur interactivité, mais la démarche suivie dans la construction de ces ateliers étant ascendante -on part d'outils définis indépendamment et on essaye de les intégrer au maximum- le résultat est souvent décevant.

Nous voyons actuellement apparaître une troisième génération d'ateliers de logiciels, à savoir celle des ateliers qui visent la prise en charge du travail en équipe et de la notion de projet, en se basant sur une gestion intégrée de l'ensemble des objets intervenant dans tout le cycle de vie d'un logiciel.

Cependant, il est un fait que la construction d'un atelier logiciel est toujours le résultat d'un nombre considérable de choix, comme par exemple, le choix d'un langage, d'un système support, de méthodes de travail pour chacune des étapes du cycle de vie couvertes par l'atelier. La solution classique, consistant à développer un atelier spécifique pour chaque "n-uplet" de choix, devient économiquement non-viable. D'autre part, la paramétrisation complète d'un modèle d'atelier par ces différents choix (un langage, un système opératoire, des méthodes) ne semble pas encore, à ce jour, techniquement réalisable, même si la paramétrisation du langage peut l'être plus facilement [DON75] [GIP87] [CRU86] [CRU88] [COL89].

De ces constatations vient l'idée [PCT86] [CAI85] d'identifier puis d'**intégrer** au sein d'une **structure d'accueil**, les fonctionnalités communes aux divers ateliers : interfaces avec l'utilisateur, gestion des activités, gestion des objets. Cette dernière fonctionnalité est, à notre sens, l'élément essentiel. Elle concerne la prise en charge de l'ensemble des informations liées à un projet : non seulement les textes sources, les textes objets, les fichiers, les documentations, etc..., mais aussi leurs liens et leurs contraintes (un module **réalise** un interface, un projet **a des contraintes** financières, une équipe **a une organisation**, etc...). Cela revient à considérer l'ensemble du système de développement comme un système d'informations et à le représenter par une base de données [PCT86] [GOD86] [CON86]. Il ne s'agit pas d'une base de données classique mais d'une base intégrant textes et données, à schéma dynamique, et qui doit être partie intégrante de la structure d'accueil au sens où elle doit être, à tout instant, le reflet fidèle de l'état du projet.

Le système de gestion d'objets peut alors devenir le support fondamental pour prendre en compte le comportement des utilisateurs, l'évaluer, le situer relativement à des comportements types, et partant, pour guider l'utilisateur : en un mot, pour permettre de décrire et d'interpréter des méthodes de production de logiciels.

Dans ce chapitre, nous développerons cette idée de structure d'accueil en présentant ses avantages par rapport aux autres approches pour le développement d'AIGL. Nous conclurons ensuite en présentant l'étape suivante de notre démarche à savoir la prise en compte de modèles de processus de développement.

1.2 Pourquoi des structures d'accueil ?

1.2.1 Nécessité des environnements de programmation

Le support de base classique pour un environnement de programmation est un système opératoire et une collection d'outils élémentaires : un éditeur adéquat, un compilateur, un chargeur, un exécutif et, quelquefois, un gestionnaire de versions et de configurations (e.g. Unix avec les commandes SCCS et Make). Ces outils permettent la création et la transformation de fichiers. Ils ne permettent pas la structuration ni la conservation de toutes les informations nécessaires au développement. En particulier, ils ne fournissent aucun moyen pour structurer, conserver, gérer les données spécifiques d'un projet, tels que les délais de réalisation, les contraintes financières, la structuration et les responsabilités des équipes... En conséquence, comme dit Lehmann [LEH87] " *ces supports de base qui n'utilisent que le concept de fichier sont insuffisants pour garantir un développement cohérent de logiciel; il est nécessaire de les compléter par une couche de gestion des informations spécifiques pour construire des environnements de programmation* " (que nous notons indifféremment AGL pour Atelier de Génie Logiciel ou PSE, pour Programming Support Environment) .

De tels environnements sont basés sur un dictionnaire de données, ou mieux, une base de données, qui structure et conserve tous les objets créés par les différents outils du PSE. De plus, comme les outils coopèrent dans la fabrication de logiciels, il semble clair que toute donnée fabriquée par un outil devra être présentée sous une forme convenable pour en faciliter l'utilisation par les outils qui lui seront appliqués ultérieurement. Ainsi, **un AGL peut être vu comme un ensemble d'outils permettant de capturer, représenter, contrôler, raffiner et transformer les informations relatives à un projet.**

Lorsque les outils auront été conçus pour pouvoir coopérer autour d'une telle base de données, et couvrir toutes les étapes du cycle de vie, de la conception à la maintenance, on parlera aussi d'atelier intégré (IPSE ou AIGL).

Ateliers intégrés de génie logiciel : une solution au développement de logiciel

Un Atelier Intégré de Génie Logiciel (**AIGL**) est un atelier de génie logiciel (**AGL**) cohérent. C'est une collection d'outils pour capturer, représenter, contrôler, raffiner, transformer,... les informations relatives à un projet, à un logiciel. Ces outils doivent permettre la conception et le développement cohérents de programmes (logiciels) et leur maintenance.

Nous opposons les Ateliers Intégrés de Génie Logiciel aux environnements de codification classiquement constitués d'un éditeur, d'un compilateur, d'un éditeur de liens et d'un système exécutif : un AIGL doit couvrir tout le cycle de vie du logiciel et permettre la gestion du projet en cours de développement.

La notion d'intégration est une notion progressive et le degré d'intégration d'un AIGL dépend des propriétés vérifiées par cet AIGL. Les ateliers de génie logiciel actuels ne sont que faiblement intégrés. Ils constituent des boîtes à outils, lesquels outils évoluent indépendamment les uns des autres.

Propriétés d'un Atelier Intégré de Génie Logiciel Ces propriétés sont celles qui sont actuellement considérées comme nécessaires pour qu'une collection d'outils constitue un atelier de génie logiciel totalement intégré.

Structuration de la collection d'outils La collection d'outils est **structurée**. Ainsi, une sous-collection d'outils peut être **associée** à une méthode [1], à un utilisateur, à un type particulier d'informations,... Un outil peut appartenir à une ou plusieurs sous-collections. Tout outil ou toute sous-collection d'outils doit être **identifiable**.

Complétude La collection d'outils doit permettre l'utilisation d'au moins une méthode assurant le développement d'un logiciel de A à Z (Couverture du cycle de vie du logiciel et possibilité de gestion du projet en cours de réalisation)

Cohérence Tout outil doit laisser les informations qu'il transforme ou génère dans une forme utilisable par n'importe quel outil qui lui succède dans un quelconque séquencement logique décrit par une méthode.

Conservation de l'information Toutes les **données** générées pendant le développement et la maintenance d'un logiciel doivent être **conservées** durant la vie du logiciel. Cela signifie , en théorie que les données ne sont accessibles qu'en lecture, en pratique que toute destruction doit être légalisée. En particulier, ces informations incluent

1. Une méthode est un ensemble d'outils, de modèles et de règles . Ces modèles et ces règles définissent l'ensemble des séquences logiques possibles d'opérations transformant les informations relatives à un logiciel d'un état donné cohérent à un autre état cohérent. En pratique, la notion de méthode est une notion récursive : une méthode est une combinaison logique (non une juxtaposition) de méthodes s'intéressant chacune à un problème spécifique. Par exemple, la méthode de développement d'un logiciel est une combinaison des méthodes s'intéressant aux différentes phases du cycle de vie du logiciel

les noms des outils et l'ordre de leur invocation ayant abouti à l'état d'avancement du logiciel : un logiciel doit être reconstructible.

Structuration des données Les données doivent être structurées. Ces données peuvent être organisées, en même temps, par projet, par génération, par auteur, par phase du développement,... On doit pouvoir regarder la collection de données de différentes façons; la collection de données doit être structurée pour **intégrer ces différentes vues**.

Extensibilité Il devrait être possible d'ajouter ou de changer un outil de l'atelier. Il devrait également être possible d'ajouter ou de changer une collection d'outils (par exemple, pour changer une méthode).

Distribution C'est une des réalités actuelles de l'architecture des Ateliers Intégrés de Génie Logiciel; ceci car le besoin de cohérence des informations gérées par un AIGL rend nécessaire leur partage par les différents postes de travail de l'équipe. Les AIGLs sont alors constitués par un ensemble de postes de travail et de machines spécialisées reliés par un réseau et coopérant à la production d'un logiciel.

Portabilité L'atelier devrait être le plus indépendant possible de la structure d'accueil sur laquelle il est développé. Un AIGL doit se conformer autant que possible aux standards existant. La définition partielle d'interfaces normalisées pour les structures d'accueil [PCT86] [CAIS85] rend possible leur indépendance vis à vis des matériels support.

Il est évident que si cette description de propriétés [GOD87] [LEH87] [REE87] a le mérite de poser les problèmes à résoudre, il n'existe pas encore d'atelier ayant toutes ces propriétés. En particulier, les propriétés d'extensibilité et de portabilité semblent encore difficiles à atteindre. Cependant, des progrès certains se font dans ce sens. Nous montrerons également les problèmes de mise en œuvre en faisant référence à des prototypes existant.

1.2.2 La fabrication des PSE nécessite des investissements importants

Actuellement, aucun PSE n'a les caractéristiques d'un IPSE. Nous ne pouvons pratiquement que distinguer différents degrés d'intégration (ou de non-intégration). On peut distinguer les "boîtes à outils" qui rassemblent des outils hétérogènes, des PSE de seconde génération qui tentent de se rapprocher des IPSE. Si les PSE de seconde génération regroupent un ensemble cohérent d'outils, ils ne couvrent jamais tout le cycle de vie : on connaît de nombreux environnements de programmation et des environnements de spécification, mais peu (pas ?) d'environnements intégrés de spécification et de programmation.

D'autre part, les PSE ne sont jamais, à la fois, multi-méthodes, multi-langages, et multi-systèmes. Enfin, la plupart des PSE sont spécifiques à un domaine d'application, il n'y a pas de PSE général, ce qui est évident : on ne développe certainement pas un système temps réel et une base de données de gestion avec les mêmes outils, ni avec les mêmes méthodes. Ceci signifie sans doute, que la solution la plus simple serait de construire chaque IPSE dans un objectif d'applicabilité bien particulier. Chaque IPSE est alors le résultat du choix d'un domaine d'application, d'un langage, d'un système support, de méthodes à respecter à chaque étape. **Sa réalisation nécessite des investissements considérables et constitue un risque financier réel certain.** Une autre solution consiste à faire un certain nombre de choix concernant, par exemple, le système support, le type d'application concerné tout en paramétrant certains aspects de l'atelier. En particulier, des ateliers comme Concerto (cf chapitre 2 de la partie II) sont paramétrés par les langages de programmation supportés. Cependant cette paramétrisation réduite ne résoud pas tous les problèmes des investissements pour la construction des ateliers. Changer de système support ou de type d'application, qui sont fixés une fois pour toute dans ce genre d'atelier paramétré, implique le changement d'atelier.

1.2.3 Les structures d'accueil : la solution économique pour la fabrication des IPSE ou AIGL

L'économie dans le développement des ateliers logiciels intégrés passe nécessairement par la **normalisation** : identification des fonctionnalités communes à implanter dans un noyau (une structure d'accueil) sur lequel seront développés ou adaptés les outils de l'atelier et **normalisation de l'interface de ces structures d'accueil**. C'est la voie suivie par le D.o.D. et la Communauté Européenne. On trouvera dans [CAI85a], les recommandations suivies dans le projet CAIS [CAI85] et dans [PCT86] une présentation de PCTE (Programming Common Tool Environment) qui est une proposition, par le Programme Esprit, de **normes** pour l'interface des structures d'accueil.

Les fonctionnalités communes sont essentiellement celles qui concernent :

- la gestion des objets qui interviennent dans l'atelier,
- la gestion des connaissances relatives aux objets, aux outils, à la structure de l'atelier, aux règles d'évolution des objets,...
- la communication entre les diverses activités (l'atelier est probablement distribué)
- la génération des interfaces avec l'utilisateur.

Dans les deux sections suivantes, nous nous intéressons plus particulièrement aux deux premiers points. Nous montrons dans chacun des cas, quels types d'information et de mécanismes sont nécessaires pour leur prise en compte.

1.3 La gestion des objets dans une structure d'accueil

1.3.1 L'apport des bases de données

Motivations

Les données d'un système d'exploitation, la structure d'accueil de base, sont déjà très structurées et intègrent les différentes vues des opérations système. Dans ce sens, on peut dire qu'un système d'exploitation a déjà sa propre base de données, mais celle-ci ne peut être adaptée aux besoins spécifiques d'un utilisateur ("customization").

Exemple1 : les structures de données de Task Control Block, Interupt Vector, File Control Block... sont utilisées pour différents besoins du système d'exploitation GCOS (système d'exploitation des DPS de Bull) : allocation de l'unité centrale, gestion des interruptions, gestion de mémoire et de fichiers,...

Exemple2 : la commande MAKE d'UNIX permet d'apposer et de vérifier une contrainte d'intégrité portant sur des fichiers en relation de dépendance (x dépend de y si la construction de x nécessite l'utilisation de y). La contrainte porte plus précisément sur un attribut du descripteur de fichier, à savoir, l'attribut dernière_date_de_mise_à_jour du type fichier_UNIX et la contrainte d'intégrité est la suivante :

si un fichier x dépend d'un fichier y alors

date_de_dernière_mise_à_jour de x > date_de_dernière_mise_à_jour de y.

Notons, cependant, que l'activité des systèmes d'exploitation et celle de gestion de projets concernent souvent les même objets, même si cette dernière concerne davantage la description logique des objets alors que les systèmes d'exploitation s'intéressent à la fois à la description et au contenu des objets. Malgré cela, il n'y a pas de véritable partage des informations concernant ces objets. En effet, pour des besoins de gestion de projet, on peut être amené à gérer un ensemble d'informations, décrivant des objets d'un projet, indépendamment de celles gérées par le système d'exploitation, ce qui, outre la redondance, présente un risque évident d'incohérence entre les deux ensembles d'informations. Par ailleurs, la stricte utilisation des données du système d'exploitation à des fins de gestion des ressources fait que des vérités aussi simples que "on n'édite pas un fichier binaire" ne sont pas considérées. Il n'est pas possible de se donner des règles simples d'utilisation autre que celles, figées, du système.

Une solution à ces quelques préoccupations, consiste à intégrer les informations du système d'exploitation et celles descriptives d'un projet au sein d'une organisation de données unique. Ainsi, que l'on commence par énoncer les propriétés attendues d'un PSE [GOD87] [LEH87] [REE87], puis en tirer les conséquences pour les structures d'accueil [GOD86], ou que l'on cherche à identifier les connaissances nécessaires à la représentation

d'un logiciel [BEN87], on est conduit aux mêmes conclusions : il est indispensable d'utiliser une approche "base de données" pour représenter et gérer tous les objets d'un projet. La base de données sert alors de référentiel commun aux différents "acteurs" de l'atelier, qu'ils soient acteurs logiciels (les outils utilisés dans l'atelier) ou acteurs humains (gestionnaires de projets, développeurs,...). Un modèle de données, par exemple de type ENTITE-ASSOCIATION [CHE76], permet de définir **une vision et une syntaxe communes aux différents outils, en particulier à deux outils se succédant logiquement au sein d'un modèle de processus de développement.** (Une étude faite par TRW a mis en évidence l'existence de 31 types d'entités et de 170 types d'associations pour leur modèle de processus de développement de logiciel [PEN85] [PEN86].)

Le point de vue statique : quelle base de données dans une structure d'accueil ?

La base de données doit comporter des informations relatives aux différents aspects du développement de projets : aspects techniques, aspects financiers, aspects temporels,... De ce fait, trois classes de données semblent nécessaires [GOD87] :

1. **Des données sémantiquement dépendantes d'un projet particulier :** elles représentent les objets qui sont créés et modifiés par les outils de l'atelier. Elles décrivent le logiciel en cours de développement. L'exemple suivant illustre cette classe de données en décrivant la structure d'un projet, certes simplifié, mais intégrant différents aspects.

   Exemple3 :

   Nous reviendrons par la suite sur cet exemple mais donnons cependant un énoncé simplifié d'un exemple de projet [ALF87] :

   - Le logiciel est développé par des personnes qui ont des rôles différents.
   - Certaines personnes ont la responsabilité du développement d'un module.
   - Chaque module doit être visé par le responsable de ce module pour qu'il soit utilisable dans une tâche.
   - Tout module est spécifié par un document.
   - La prise de retard d'un module provoque un risque temps pour les tâches qui ont besoin de ce module.
   - Une contrainte financière représente une enveloppe budgétaire.
   - Une tâche est soumise à une contrainte financière.
   - Une tâche qui a utilisé 80% de ressource financière devient critique de ce point de vue.

La figure 1.1 intègre une partie de cette description dans un diagramme de type entité-association où un rectangle symbolise un type d'entité et une ellipse un type d'association.

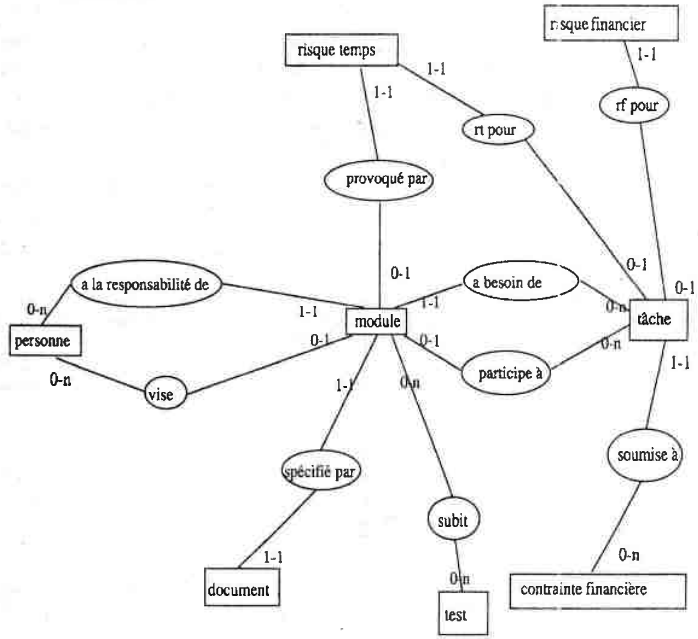


Figure 1.1. Un exemple de projet simplifié

2. **des données décrivant l'atelier lui-même**, indépendamment de chaque projet particulier.

   Un atelier regroupe un ensemble d'outils et cet ensemble doit être structuré selon les méthodes, les sources, les auteurs, etc... Cette structuration est la base de l'intégration des outils et de leur évolution (par exemple, lors du remplacement d'un outil par un autre ou lors de l'ajout d'un nouvel outil). Du point de vue de l'organisation, ces outils ne sont que des objets que l'on peut représenter comme des entités. Ces objets de type outil peuvent également être liés avec n'importe quel objet d'un type quelconque : c'est ce qui permettra de décrire des modèles d'enchaînement d'outils (morceaux de modèle de processus de développement).

La figure 1.2 décrit un modèle d'enchaînement d'outils en termes du modèle Entité-Association. Une activité y représente une séquence d'opérations. Nous conseillons au lecteur intéressé de se référer à [TAR86] où il trouvera une expertise plus complète d'un Atelier de Génie Logiciel particulier en termes d'entités et d'associations. La figure 1.2 peut être interprétée de la manière suivante : une phase (instance de type de phase) est composée de n activités (instance de type d'activité). Chaque activité produit un résultat (instance de type de résultat) dépendant des conditions ou paramètre (instance de type de condition) dans lesquelles s'exécute cette activité. Cette définition de phase définit donc un ensemble d'activités réalisant cette phase. D'autre part, un regroupement (agrégation) de m phases peut s'avérer utile pour réaliser une tâche composée de $m \times n$ activités. En outre, chaque phase peut être ordonnée par rapport aux autres phases par une contrainte de succession.

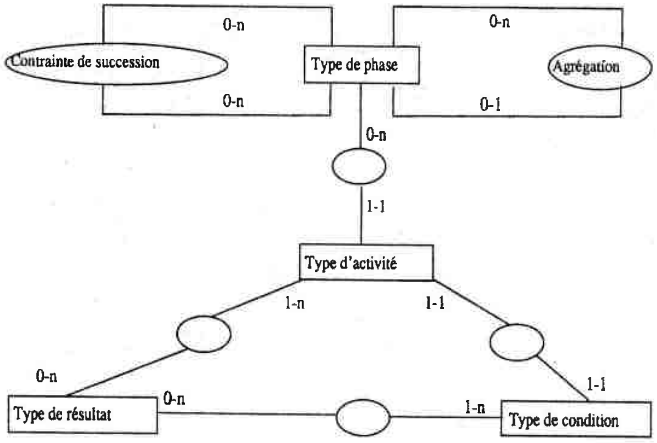


Figure 1.2. Un modèle d'enchaînement d'outils

3. **Des données historiques** : Elles conservent l'histoire du projet. Elles sont utiles pour le "Reverse Engineering" [2], et sont souvent utilisées pour expliquer l'état du logiciel lors des opérations de maintenance et comprendre le pourquoi de certaines modifications ou corrections du logiciel. Elles sont également utiles pour l'assistance en cours de développement, comme nous essaierons de le montrer dans les sections

2. Le "reverse engineering" consiste à travailler à l'envers par rapport au cycle normal de développement d'un logiciel.

suivantes.

Il faut conserver des générations d'objets et d'associations entre objets. Les études menées sur **les bases de données historiques** [KEB84] [TIG85] apportent des éléments de réponse à la gestion de l'histoire d'un système d'information. Notons également qu'un mécanisme de datation couplé à un mécanisme de propagation de contraintes temporelles pourrait, au moins partiellement, permettre une reconstruction automatique et donc plus sûre d'un logiciel plus sain.

Le point de vue opérationnel : la juxtaposition d'un Système d'Exploitation et d'un SGBD est une solution insuffisante

La pratique actuelle consistant à juxtaposer le S.G.B.D. gérant la base de données et le Système d'Exploitation, avec une connexion très faible, pour ne pas dire inexistante, entre les deux systèmes semble peu viable. En effet, il y a une séparation entre le développement de logiciel et la gestion de projet : les développeurs et les gestionnaires de projet sont souvent des personnes différentes qui utilisent des outils différents, mais ils se partagent des informations. La gestion de projet nécessite cependant que la plupart des actions entreprises par les développeurs, telles qu'édition d'un diagramme de flux de données, compilation, modification d'un source, etc...laissent une trace dans la base de données du projet. Cet effort d'information est actuellement laissé à la charge des développeurs qui doivent accompagner les opérations de développement d'actions de mise à jour de la base de données. Cette situation conduit inévitablement à des oublis et à des incohérences. Il est nécessaire de saisir "au vol" les informations de gestion de projet, c'est-à-dire **d'intégrer les outils des développeurs et le schéma de la base de données**. En d'autres termes, l'effet de toute action entreprise par un développeur sur un ou des objets doit automatiquement mettre à jour la base de données, par un "effet de bord" des opérations réalisant les intentions du développeur.

Sur un autre plan, une des caractéristiques essentielles du développement de logiciel est de manipuler des objets complexes comme des textes de programmes, de documentation, de spécification, des codes objets, des graphiques,..., faisant ainsi ressortir la nécessité d'intégrer des données "relationnelles" (celles que savent gérer les S.G.B.D.) et des données textuelles. Or, les S.G.B.D. actuels n'offrent pas encore de moyens efficaces pour gérer de tels objets hétérogènes. Enfin, la faiblesse des S.G.B.D. pour la prise en compte de contraintes d'intégrité semble aussi un handicap pour leur utilisation en tant que tels au sein d'une structure d'accueil. Il ne suffit pas, en effet, de détecter des violations de contraintes d'intégrité; il faut aussi "réparer", c'est-à-dire entreprendre une action constructive et non pas se contenter de rejeter l'opération violant la contrainte d'intégrité.

Conclusion

Nous ne voulons pas, à travers ce qui précède, exclure totalement l'utilisation d'un S.G.B.D. dans le développement d'un IPSE. Notre but est d'insister sur la nécessité d'intégrer les fonctionnalités des S.G.B.D. et celles des systèmes d'exploitation. Notre proposition fait surgir un certain nombre de problèmes dont ceux cités ou rappelés ci-dessous ne sont pas les moindres :

- intégrer données simples (nombres, courtes chaînes de caractères,...) et données complexes (textes, programmes,...)
- intégrer la description des outils de développement au schéma de la base de représentation du SGBD afin d'assurer une mise à jour automatique ("par effet de bord") lorsqu'un outil modifie l'état d'un objet,
- disposer de mécanismes efficaces de préservation de l'intégrité de la base et des objets,
- offrir des mécanismes permettant de faire évoluer le schéma de la base afin que chaque organisation de projet puisse y être décrite.

Notons cependant que des travaux intéressants, tels que Software Knowledge Base [MEY85] [MEY85b], TRW-PMDB [PEN85] [PEN86] ou [GOD86] qui interface le SGBD Ingres avec Unix, explorent cette voie et mettent en évidence les caractéristiques d'un **système de gestion d'objets** intégrant les fonctions d'un S.G.B.D et celles d'un système d'exploitation.

1.3.2 Les structures d'accueil à gestion d'objets

L'apport d'une approche "orientée objet" aux structures d'accueil

Nous définissons une structure d'accueil à gestion d'objets par opposition aux systèmes à gestion de fichiers. Une structure d'accueil est un noyau permettant la génération d'environnements dans lesquels on manipule des objets, c'est-à-dire des sources, des documents, des programmeurs,... et non plus des fichiers et seulement des fichiers. Plus précisément, dans ces environnements, on édite des textes (on peut y éditer la documentation d'un source x donné), on compile des programmes,... par opposition aux systèmes actuels où l'on édite, on compile,... des fichiers et seulement des fichiers, avec, par exemple, la conséquence vécue par chacun qui est le "plantage" du terminal à l'édition malencontreuse d'un fichier contenant de l'exécutable.

L'apport de cette informatique des objets est une intégration des données opérationnelles habituelles, qui constituent le logiciel en cours de développement, avec les données organisationnelles, qui décrivent l'organisation du projet en garantissant leur cohérence. L'enjeu en est une meilleure organisation des projets et un pas vers l'automatisation du développement de logiciels, en particulier si l'on intègre le concept de modèle de processus de développement.

Sur un autre plan, dans le domaine des bases de données, on a longtemps favorisé l'aspect structuration des objets [SMI82] par rapport à l'aspect comportement de ces objets (identification des opérations qui leur sont applicables). Du point de vue conceptuel, l'aspect comportemental n'était alors représenté que partiellement par la proposition de mécanismes de "triggering" permettant de préserver les propriétés invariantes d'une base. L'aspect comportemental fut véritablement abordé dans les travaux synthétisant les bases de données et les types abstraits [CHA82], [BOU84], [DOS82], dans ceux intégrant données et dynamique des systèmes d'information, en particulier REMORA [FOU82] [ROL88] et MERISE [TAR83] et enfin dans ceux synthétisant les bases de données et les langages orientés objets [BAR81]. Pour une bibliographie complète sur le domaine, le lecteur peut se référencer à [GOD89b]. Nous verrons par la suite, après une description succinte de REMORA et MERISE pourquoi de telles méthodes, tout en prenant en compte la dynamique des systèmes d'informations, sont insuffisantes pour la représentation et la prise en compte des processus de développement de logiciels. Les langages orientés objets offrent des mécanismes comme ceux de classification et d'héritage de propriétés et d'opérations qui semblent intéressants à intégrer. L'héritage permet de définir un concept à partir d'un ou plusieurs autres. Cela est fondamental dans l'application qui nous intéresse. En effet, le noyau d'une structure d'accueil est classiquement constitué d'un ensemble d'objets de base (et d'opérations sur ces objets) et plus fondamentalement d'un schéma minimum qui structure ces objets et décrit l'organisation de l'application "structure d'accueil". Tout atelier bâti sur cette structure doit alors spécialiser cette organisation de base. La figure 1.3 montre une telle spécialisation du type source en module. Plus généralement, le problème est celui de **l'extensibilité** d'un schéma de base de données. L'utilisation des techniques d'héritage semble être une voie prometteuse pour résoudre ce problème de "schémas dynamiques".

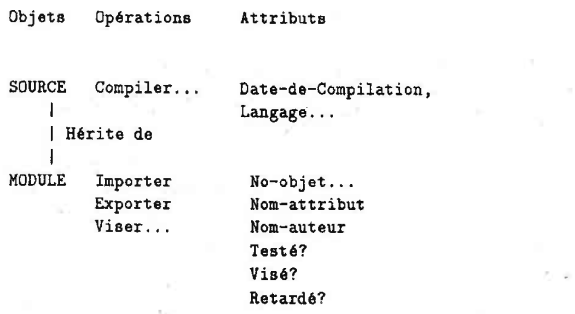


Figure 1.3. Module est une spécialisation de Source

Par ailleurs, l'héritage permet la réutilisation de programmes (héritage d'opérations) et la simplification de l'interface utilisateur. Il est alors nécessaire de généraliser les mécanismes d'héritage simple et multiple [GOL83], [COL86] entre objets à l'héritage des associations et des contraintes [CAZ86]. [BOZ86] propose une formalisation, basée sur une typologie des généralisations, qui tente de résoudre ces problèmes. [REI84] montre comment la transitivité des généralisations (liens IS_A) peut être vue comme une conséquence directe de l'implication logique.

Constituants d'une structure d'accueil à gestion d'objets

Du point de vue opérationnel, une telle structure d'accueil peut être vue comme la donnée :

- d'un ensemble de types prédéfinis, (la figure 1.4 en est un sous-ensemble dans lequel les queues sont des fichiers de communication inter-processus. Pipe décrit le mécanisme UNIX de même nom, Multiple permet l'échange de données entre un processus et plusieurs autres).
- d'un ensemble d'objets prédéfinis : ce sont les objets systèmes habituels. Ce sont des instances d'objets tels que ceux décrits dans la figure 1.5 .
- d'une organisation prédéfinie de ces objets : la figure 1.5 en est un exemple.
- d'un ensemble d'opérations : ce sont surtout les opérations des types.

Les opérations originales et fondamentales sont celles qui permettent d'étendre l'organisation de base, c'est-à-dire de **définir de nouveaux types par spécialisation ou généralisation de ceux existant, de créer de nouveaux objets et de créer de nouvelles**

associations entre ces objets, éventuellement entre un nouvel objet et un objet de base. La figure 1.4 ne dit pas comment créer des associations; notons cependant que l'on peut physiquement considérer ces associations comme des objets particuliers ou même comme des propriétés d'objets. [CAI85] propose une implantation des associations comme des propriétés d'objets appelés NODE.

Dans ce cadre, la définition d'un atelier sur une structure d'accueil revient essentiellement à :

- "affiner" les types prédéfinis pour tenir compte des spécificités de l'atelier,
- intégrer le schéma de l'atelier à celui de la structure d'accueil (par exemple, intégrer le schéma de la figure 1.1 à celui de la figure 1.5 dans lequel les cardinalités non précisées sont de type n-m. Un "outil privé" est de type exécutable. Les objets et associations au-dessus de la barre horizontale sont ceux conservés d'une session à une autre. Ceux sous cette barre ont pour durée de vie la session).

La figure 1.4 définit une hiérarchie de types. Par exemple, le type terminal est un sous-type du type fichier. On peut lui appliquer les opérations applicables à tout fichier (ouvrir, fermer,...), mais aussi des opérations spécifiques à son type (rafraichir,...)

La figure 1.5 définit une organisation possible des types définis. Un système est composé de terminaux, d'outils de base, de projets et d'utilisateurs. Chaque utilisateur possède un certain nombre d'outils de base, de sources et de textes. Le système, par ailleurs, supporte un processus courant qui est un des processus exécutant l'un des outils de base ou l'un des outils privés de l'utilisateur. Les processus communiquent entre eux par l'intermédiaire de queues qui sont soit de type pipe, soit de type multiple.

Figure 1.4. Un exemple de types de base

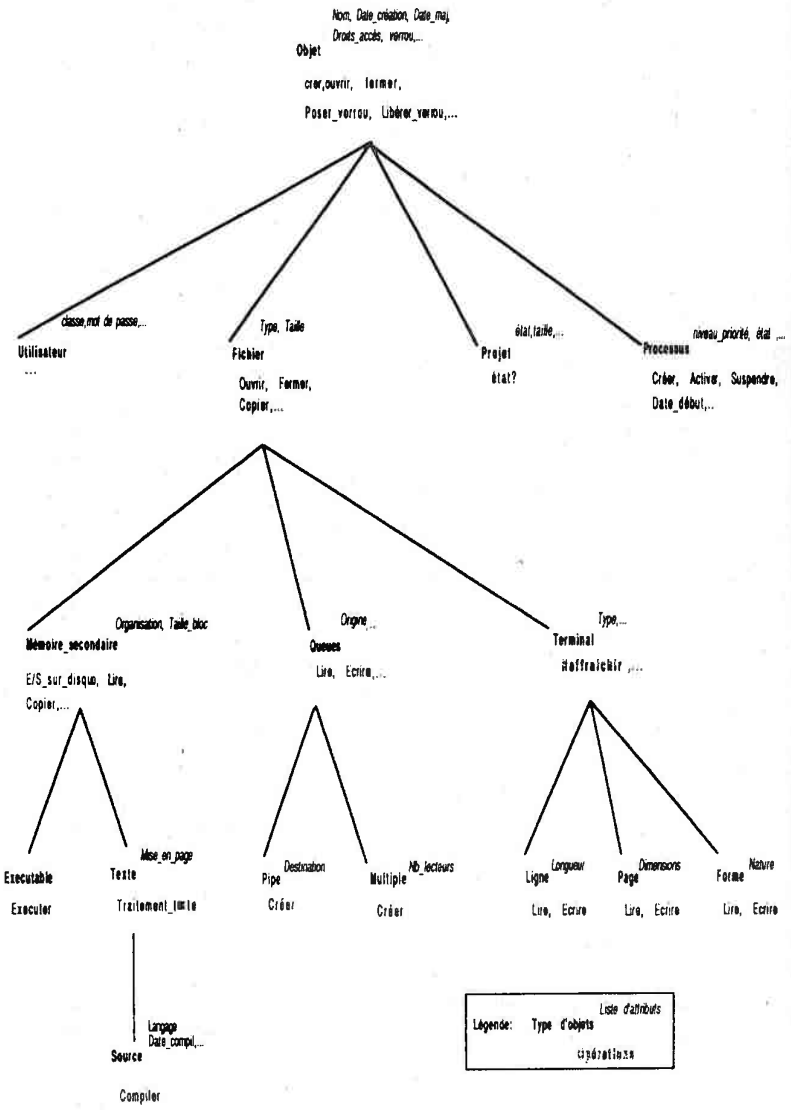
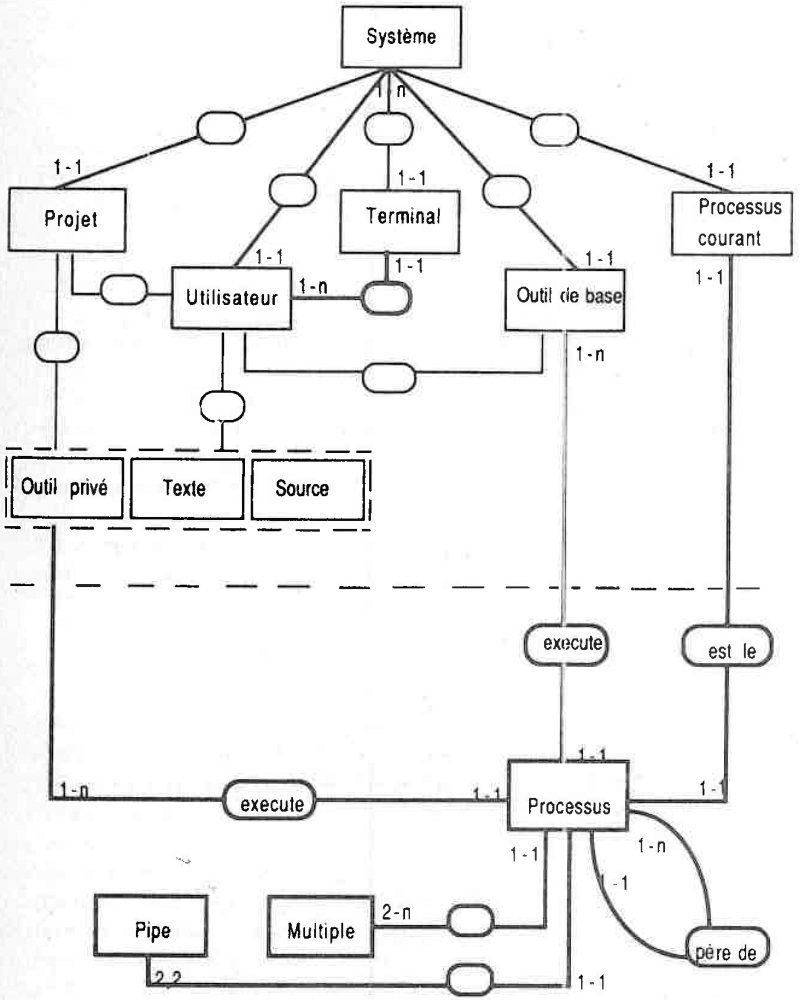


Figure 1.5. Un exemple d'organisation prédéfinie



1.4 Etat de l'art

Une énergie importante est dépensée dans le domaine et les résultats différeront selon l'approche suivie pour résoudre les besoins en développement de logiciels. Nous classerons ces résultats en trois catégories suivant les choix faits (couplage SGBD/Système d'Exploitation, développement d'ateliers, développement de structures et de mécanismes de base permettant le développement d'ateliers).

1.4.1 Les SGBD pour le génie logiciel

Pour être complet il est nécessaire de référencer les travaux effectués autour de cet axe. Citons l'atelier SEMA-METRA (modélisé dans [TAR86]), le SGBD belge ALMA [LAN86], le SGBD SYBASE [HOW86] les projets américains PMDB [PEN85] [PEN86] et SKB [MEY85] [MEY85b].

1.4.2 Les AIGLS

Le projet CONCERTO, développé au CNET est celui qui intégre le plus grand nombre de concepts décrits et nécessaires à un AIGL. La structuration d'un projet se fait dans le langage F1 qui permet la description des objets et de leurs associations et surtout de contraintes sémantiques complexes. La faiblesse du système de gestion réside dans l'absence d'un mécanisme de "triggering", ce qui ne permet pas la vérification implicite des contraintes sur plus d'un type d'objets, la vérification de telles contraintes devant être, bien qu'écrites en F1, demandée explicitement. Nous reviendrons sur Concerto dans le chapitre 2 de la partie II. Citons pour mémoire les projets ISTAR [DOW86] et IDE [WAS85]. On peut trouver dans la littérature de nombreuses descriptions d'ateliers. Pour une synthèse, on pourra se référer à [GL1] et [GL13].

1.4.3 Les structures d'accueil

PCTE

PCTE (Portable Common Tool Environment, Projet ESPRIT) définit les interfaces d'une structure d'accueil à gestion d'objets et à architecture distribuée. Le concept d'objet est réduit à l'aspect structurel (on n'attache pas d'opérations aux objets). La description des objets et de leurs associations se fonde sur le modèle E-A. Un autre aspect intéressant de PCTE, bien que non directement lié à nos propos, y est la spécification d'interfaces génératrices d'interfaces utilisateurs [PCT86]. L'intérêt de telles interfaces est la définition d'un standard en matière de structures d'accueils. L'intérêt en est la possibilité de porter des collections d'outils indépendamment des matériels utilisés. EMERAUDE [EME84] est la première implantation commercialisée de PCTE bâtie autour d'UNIX, d'autres sont en

cours (cf § 3.2.3 du chapitre 3 de la partie II). PACT (PCTE Added Common Tools, Projet ESPRIT) est l'un des nombreux projets Esprit bâtis sur PCTE. Il a pour but d'enrichir la norme PCTE d'outils de base : outils d'interrogation relationnelle de la base d'objets, de documentation, de communication, de gestion de configurations,...

CAIS

CAIS (Common APSE Interface Set, APSE : ADA Programming Support Environment) est un projet du D.o.D américain qui définit les interfaces d'une structure d'accueil pour le développement d'APSEs [CAI85] [CAR87]. Un seul concept permet la définition des objets et des liens, celui de NODE. On peut assimiler un NODE particulier à une entité; les associations sont représentées par des liens binaires qui sont propriétés du NODE source de la liaison. [CAI85] spécifie ces interfaces, [CAR87] est une implantation particulière partielle de ces interfaces au dessus d'UNIX. Toutes les interfaces CAIS sont des interfaces ADA, ce qui permet l'unification des concepts systèmes avec n'importe quelle application écrite en ADA : l'utilisateur ne connaît plus qu'un seul langage. Par exemple,

```
La procédure OPEN  (NODE: in out NODE_TYPE;
                    NAME:in NAME-STRING;
                    INTENT: in INTENTION:= (1=> READ);
                    TIME-LIMIT: DURATION:= NO_DELAY);
```

permet l'accès à un NODE donné; INTENT permet la pose de verrous d'intention sur ce NODE, TIME-LIMIT précise le temps d'attente maximum.

```
La procédure LINK  (NODE: in NODE_TYPE;
                    NEW_BASE: in NODE_TYPE;
                    NEW_KEY: in RELATIONSHIP_KEY;
                    NEW_RELATION: in RELATION_NAME := DEFAULT_RELATION);
```

permet à un utilisateur de lier deux NODEs. Chaque lien reçoit un numéro qui l'identifie. On peut regrouper des liens semblables d'un point de vue sémantique dans une même RELATION. Une telle relation est alors une fonction d'accès qui permet d'atteindre plusieurs NODEs à partir d'un seul.

CAIS propose donc la structuration des objets en réseau. Le modèle conceptuel est relativement pauvre. En particulier, il n'existe pas de véritable schéma de la base d'objets. Cependant, la "philosophie objet" est bien présente. L'une des raisons de cette pauvreté est certainement le souci constant de performances des spécificateurs de CAIS. Un autre objectif prioritaire est la transportabilité de logiciel : le respect des interfaces CAIS assure le transfert, non seulement d'un outil, mais surtout d'un ensemble d'outils (se partageant un ensemble de NODEs structurés en commun).

1.5 La prise en compte des modèles de processus de développement dans un atelier de logiciel

Avant-propos : Nous adoptons dans le paragraphe suivant une attitude évidemment réductionniste. Les notions d'organisation et de modèle de processus de développement sont en effet suffisamment complexes pour que leur définition nécessite de plus amples développements. La dernière partie de cette thèse s'y attachera.

Un environnement de production de logiciel est une organisation, c'est-à-dire qu'elle a la propriété de relier, de maintenir et de produire des objets. **Un modèle de processus de production de logiciel est un ensemble d'outils, de modèles de représentation de l'information, de modèles d'enchaînement d'outils et de règles de gestion. Il décrit un schéma de fonctionnement de l'atelier.**

Un modèle de processus de développement n'est pas un programme : il ne définit pas une démarche unique pour atteindre un état particulier. Il peut y avoir plusieurs chemins pour atteindre un état particulier du logiciel : le modèle de processus de développement est là pour **guider** le développement et assurer que l'état atteint à la fin d'une activité a un sens.

Ainsi, la définition d'un modèle de processus de développement comportera :

- la description des types d'objets et de leurs liens (cf figure 1.1),
- la spécification syntaxique (profil des opérations) et sémantique (que fait l'opération) des opérations. D'autre part, comme certaines opérations sont réalisées par des outils de l'atelier (e.g. : un compilateur ADA implante l'opération compiler), la définition d'un modèle de processus de développement devra connaître le lien entre une opération et l'outil qui la réalise.
- la description des contraintes à respecter pour le développement d'un projet. Cette dernière composante d'un modèle de processus de développement définit les états acceptables, que nous appellerons aussi stables, des objets du système. Des exemples de contraintes extraites de l'exemple 3 sont : " un module est dit testé s'il a subi un ensemble déterminé de tests et s'il n'a obtenu aucune note inférieure à 15 à chacun de ces tests", "un module visé est forcément testé" ou encore "un module ne peut participer à la réalisation d'une tâche que s'il est visé".
- un ensemble de règles de production décrivant les opérations à entreprendre en cas de non respect d'une contrainte, et ceci en vue d'atteindre un état stable dans le développement d'un logiciel.
- un ensemble d'enchaînements possibles des opérations. Cet ensemble d'enchaînements possibles définissant un ordre partiel (ordonnancement) sur l'ensemble des opérateurs.

Les états stables sont des états où les contraintes définies sont vérifiées. Le système de gestion d'objets et le système chargé de la mise en œuvre d'un modèle de processus de développement ne devront autoriser que des opérations qui assurent une transition d'état stable à état stable. Les types de contraintes ci-dessus, ainsi que l'ordre (ordonnancement) sur les opérations -que nous évoquerons plus loin-, sont à la base du fonctionnement du système de pilotage. En effet, les contraintes peuvent être exploitées, d'un point de vue opérationnel, de diverses façons :

1. **Les contraintes peuvent être assimilées aux contraintes d'intégrité** des bases de données traditionnelles : si un usager veut délivrer un visa à un module, son opération est refusée si aucune information dans la base n'indique que ce module est testé. La contrainte est alors une propriété qui doit toujours être vraie.
2. **Les contraintes peuvent aider à décider quelles opérations pourront être exécutées** sur une instance d'un type d'objet dans un état donné : si le système "constate" qu'un module est testé, il pourrait, par exemple, proposer à l'utilisateur de le viser puis de le diffuser auprès d'une liste de destinataires. Cette manière d'utiliser les contraintes correspond à une utilité possible des règles de production.
3. Les contraintes peuvent permettre de décider **quelles opérations entreprendre pour éviter de rejeter une demande** : si l'on veut viser un module non testé, le système pourra demander (ou décider) de le tester d'abord. Cette manière d'utiliser les contraintes correspond à une autre utilité possible des règles de production.

Les techniques nécessaires à la résolution des problèmes soulevés dans ce paragraphe font évidemment penser à celles utilisées en intelligence artificielle [NIL82] [RIC83] ou encore dans le domaine "bases de données et intelligence artificielle" [GAL78] [GAL81] [GAL82] [BOU85] [FIR87]. Si les travaux concernant la vérification de contraintes d'intégrité décrites par la logique n'ont pas encore totalement abouti, un système tel que PROLOG-KB [BOC86] semble cependant prometteur.

D'autre part, les mécanismes de base nécessaires pour concrétiser les utilisations 2 et 3 s'apparentent beaucoup aux mécanismes de raisonnement en avant et en arrière [NIL71], [RIC83] pour décider dynamiquement quelle opération ou quelle séquence d'opérations est possible dans un état donné du projet en cours de développement. Il est bien évident que le raisonnement en avant et/ou en arrière ne se limite pas au seul niveau des opérations précédant ou suivant immédiatement celle invoquée par l'utilisateur. De plus le choix de suggérer ou de décider de l'exécution des opérations relève des stratégies du système. Nous

pensons que la structure d'accueil doit les mettre toutes à la disposition du rédacteur de méthodes.

D'autre part, l'ordre partiel (ordonnancement) défini sur les opérations permet quant à lui d'exprimer un autre aspect des méthodes de travail : l'interdépendance intrinsèque des opérations. Par exemple, sur une station de travail, le fait qu'un outil graphique ne puisse être utilisé que sous l'environnement graphique de la station pourra s'exprimer par la précédence de l'opération "lancer l'environnement graphique" sur l'opération "lancer outil graphique". Toute combinaison d'opérations dans laquelle "lancer outil graphique" précédera "lancer l'environnement graphique" devra être rejetée à priori.

Ainsi, la base d'objets augmentée des règles [3] décrivant un modèle de processus de développement peut être vu comme **une base de connaissances** et le système d'assistance comme un **système à base de connaissances**. Réaliser une structure d'accueil à gestion d'objets et "paramétrable par les modèles de processus de développement" consiste alors en l'intégration des fonctionnalités d'un système à base de connaissances et d'un système d'exploitation.

1.6 Conclusion

Ce chapitre montre les principaux concepts et mécanismes que nous estimons indispensables pour le développement de structures d'accueil d'ateliers de logiciels. Nous avons essayé d'insister sur le fait que de telles structures devaient nécessairement englober une base de données contenant une représentation unique des différents objets intervenant dans le développement et la gestion de projets et donc priviligier l'approche structure d'accueil. Cette base sert, à la fois :

- au système d'exploitation qui y trouvera les informations utiles à son fonctionnement (identification des opérations, des outils, des objets...) et qui y inscrira, "par effet de bord", toutes les transformations qu'il fait subir aux objets,
- aux gestionnaires de projets qui y trouveront la description fidèle (on l'espère) de l'état courant du projet et des étapes qui ont conduit à cet état,
- aux développeurs qui seront guidés et aidés (voire contrôlés ?) dans leur activité de développement.

Ce dernier point relève de la définition et de la prise en compte de modèles de processus de production de logiciels. La dernière section de ce chapitre a essayé de donner un aperçu de ce que l'on entendait par modèle de processus de développement et du bénéfice que l'on

3. Le terme règles est utilisé ici avec son sens commun : il regroupe à la fois l'ordre (ordonnancement), les contraintes et leurs règles associées

pouvait en tirer, notamment par le biais de mécanismes plus ou moins "intelligents" de pilotage et d'assistance.

La prise en compte des modèles de processus de développement dans une structure d'accueil fera l'objet de développements détaillés dans la dernière partie de cette thèse.

PARTIE II

GESTION DE L'INFORMATION EN GENIE LOGICIEL : PREMIERE ETAPE VERS LES ATELIERS DE GENIE LOGICIEL DE TROISIEME GENERATION

1

Systèmes de Gestion de l'information associée au développement de logiciel

1.1 Introduction

Comme nous l'avons vu dans le chapitre 1 de la partie I, la juxtaposition d'un SGBD et d'un système d'exploitation est insuffisante pour pouvoir réellement parler d'un AIGL. Cependant de nombreuses propriétés attendues d'un AIGL sont typiquement des propriétés pour lesquelles ont été définis les systèmes d'information. Par exemple la *structuration des données* manipulées dans un atelier et la *conservation de l'information* sont des propriétés pouvant être remplies par un système d'information. La juxtaposition d'un SGBD à un système d'exploitation peut donc être considérée comme un bon point de départ (même s'il nous apparaîtra insuffisant ultérieurement) pour étudier la construction d'AIGLs.

Dans ce chapitre nous allons donc présenter deux approches assez similaires pour la représentation et la gestion de l'information associée au développement de logiciels, à savoir le projet SKB et le projet TRW-PMDB. Ces deux projets, bien que n'étant pas à proprement parler des Ateliers de Génie Logiciel, sont cependant inscrits dans le cadre d'un tel développement et sont résolument orientés vers le génie logiciel.

1.2 Le projet SKB

1.2.1 Introduction

Le projet SKB est *"un système pour conserver les informations utiles d'un projet de logiciel"* [MEY85b] développé à partir de 1985 à l'université de Santa Barbara (Californie) par l'équipe de B. Meyer et est à l'heure actuelle à l'état de prototype. Le concept de Software Knowledge Base (SKB qu'on peut traduire par "Base de Connaissances Logicielles") a été introduit par B. Meyer pour pallier aux difficultés des gestionnaires et développeurs

de projets qui étaient confrontés à la difficulté d'avoir à tout moment une vision claire et juste de l'état du projet et de son état d'avancement [MEY85] [MEY85b].

Le SKB a pour rôle de conserver les propriétés des composants d'un logiciel et leurs différentes relations. Les composants de SKB sont tout ce que l'on peut rencontrer dans un projet de logiciel (modules, documentations, spécifications,...) et les relations les mettant en cause sont, bien sur, de plusieurs types (relation de documentation entre un module et sa documentation, relation d'implantation entre une spécification et un module). Le système chargé de stocker et de manipuler l'ensemble des informations formant le SKB est appelé système du SKB (on utilisera le terme système pour le désigner s'il n'existe aucune ambiguïté).

Ce système tout en se voulant simple et adaptable a pour objectifs :

- l'indépendance du système, de la méthode et du langage de développement
- la couverture de tout le cycle de vie
- la prise en compte de la sémantique propre de chaque système de développement
- une description formelle de tout système de développement
- l'indépendance entre les objets et leur représentation.

Ces différents objectifs seront explicités dans le paragraphe suivant.

1.2.2 Les objectifs du système SKB

Le système doit être **simple de compréhension et d'utilisation.** Dans ce but le système devra être de conception simple et uniforme.

Il doit en outre être considéré comme une aide aux gestionnaires et développeurs et ne pas changer les habitudes de ceux-ci en introduisant une nouvelle manière de travailler. Il ne doit pas sous-entendre l'utilisation d'une méthode, d'un langage ou d'un système particulier : le système SKB doit être **indépendant.**

Cette indépendance ne doit cependant pas en faire un système inutilisable par les développeurs utilisant un certain langage ou par les gestionnaires utilisant une certaine méthode. Il doit être **adaptable** : on doit pouvoir le paramétrer par une méthode, un langage ou un système.

Le système SKB, pour espérer jouer un rôle significatif en développement de logiciel, doit couvrir tout le **cycle de vie d'un logiciel.**

Par ailleurs, le système SKB devra prendre en compte la **sémantique** d'un système de développement sans pour autant imposer une sémantique propre. Il devra être capable

de manipuler des attributs caractéristiques de tel ou tel projet et de prendre en compte des propriétés caractéristiques de projets telles que les relations entre objets.

Le projet SKB essaie de faire une analyse fine des différents besoins et de **formaliser** ceux-ci avant toute tentative d'implantation.

Et enfin, l'un des aspects primordiaux de SKB, qui en fait sa force ou sa faiblesse, est **l'indépendance entre les objets et leur représentation.** Une base de connaissances logicielles est un modèle conceptuellement et physiquement différent du projet de logiciel qu'elle modélise. Les représentations des composants d'un logiciel sont appelées atomes et possèdent un certain nombre d'attributs décrivant l'objet représenté. Cette séparation entraîne un risque d'inconsistance et d'incohérence entre les objets et leur représentation. Cependant un tel choix rend l'écriture d'un système pour le SKB assez simple, et le système peut être vu comme un outil indépendant de l'atelier de développement, pouvant même être implanté sur une machine autre que celle de développement.

1.2.3 Présentation du modèle SKB

Les concepts

Le modèle du SKB est très classique et repose sur les concepts d'**atomes**, d'**attributs**, de **relations**, de **contraintes** et d'**actions**.

Les **atomes** sont la **représentation** dans le SKB des objets réels intervenant dans le développement de logiciels. Ils ne sont concernés que par les opérations du système SKB et leurs propriétés sont celles que leur définit le système SKB.

Ces atomes peuvent avoir des **attributs** dont les seuls valeurs autorisées sont celles des **types prédéfinis** entier, chaîne, date et fichier. La liste des attributs d'un atome est définie par l'utilisateur qui a, en outre, à sa disposition un certain nombre d'atomes dont le "type", représenté par un attribut particulier de type chaîne, est prédéfini.

Les attributs ne pouvant prendre comme valeurs qu'un nombre réduit et prédéfini d'attributs, d'autres propriétés doivent pouvoir être exprimées différemment. Certaines de ces caractéristiques d'un projet sont représentées par des **relations** entre atomes. Ces relations expriment les propriétés structurelles du système modélisé. Le système SKB utilise pour exprimer ces propriétés structurelles des **relations binaires.** Rappelons que toute relation n-aire peut se ramener à un certain nombre de relations binaires [BRA76], et qu'ainsi le système SKB peut exprimer toute sémantique d'un système modélisable par des objets et des relations entre objets.

Les atomes et leurs relations ne suffisent pas à exprimer des propriétés fondamentales, telles que la propriété de dépendance entre deux objets (un source et son code par exemple). Un nouveau concept est introduit dans ce but : les **contraintes** associées à un SKB. Elles

définissent un ensemble de propriétés plus fines que celles définies par les attributs des atomes et leurs relations.

Le dernier concept est celui d'**action**. Une action est associée à une contrainte et cette action spécifie les opérations à effectuer si la contrainte est invalidée. Dans l'exemple de la contrainte de dépendance du source et de son code objet, l'action associée à une violation de cette contrainte sera la recompilation du source. On peut noter que dans ce cas précis le mécanisme du Make d'Unix est largement suffisant dans la mesure où l'expression de cette contrainte ne met en cause que les attributs de dernière mise à jour d'un fichier (présents dans le système de gestion de fichiers d'Unix) manipulés par le Make. Le concept d'action n'est pas un concept purement interne au SKB. Il doit être possible de réaliser des opérations dans le système de développement. En effet dans l'exemple précité, la recompilation du source ne doit en aucun cas être limitée à un quelconque changement d'attributs de l'atome représentant le source, mais bien être l'opération de compilation du système de développement. Il ne faut pas oublier que le SKB n'est qu'un modèle du système réel.

Mise en œuvre de ces concepts pour le génie logiciel

Un des objectifs de SKB est de fournir une description formelle des différentes relations entre composants d'un projet de logiciel. Un ensemble de définitions formelles d'atomes, de relations et de contraintes sont données dans [MEY85b]. Nous en énoncerons quelques-unes.

Les atomes peuvent être "typés" par un attribut type d'atome qui leur est associé : *spécification, module, coût, etc...* sont ainsi prédéfinis.

Les relations sont définies de manière plus stricte. Meyer définit un certain nombre de relations et les contraintes sémantiques qui leur sont associées en développement de logiciels. Ainsi, par exemple, les relations partie_de, comprend, et complément_de sont définies entre atomes du même type. Ces relations doivent respecter la contrainte suivante : si a est partie_de b et b comprend c alors il existe une relation complément_de entre a et c. Cette contrainte peut alors être exprimée en CRAC (Calcul des Relations Attributs et Contraintes) de la manière suivante : "partie_de ; comprend $\leq$ complément_de". Pour de plus amples informations sur les propriétés définies, se rapporter à [MEY85b].

1.2.4 Implantation du modèle SKB

Le système SKB fournit les primitives classiques de tout SGBD, à savoir les primitives de création, d'accès et de mise à jour des atomes, attributs, relations et contraintes. Ces primitives sont exprimées par le langage des prédicats **CRAC**. L'implantation du langage CRAC se fait de manière à fournir une interface uniforme utilisable à la fois de manière

interactive ou non-interactive par un utilisateur, et par tout programme. Afin d'aider à la phase de conception, pendant laquelle les concepts sont plus facilement exprimables sous forme de schémas, une implantation graphique de CRAC est réalisée.

Le second volet de cette implantation est un essai d'implantation efficace du mécanisme de vérification de contraintes. En effet une vérification globale de toutes les contraintes du système entrainerait un temps de réponse beaucoup trop long dans la mesure où la taille d'une telle base de connaissances logicielles est proportionnelle à la taille très importante d'un projet de logiciel (cf. étude de TRW). Le concept de **démon** a été introduit dans le but de diriger le système dans sa recherche de contraintes invalidées. On associe un démon à chaque contrainte et celui-là sert de garde à l'action associée à la contrainte. Lors d'une mise à jour de la base, les démons concernés vérifient leur contrainte respective, et si celle-ci est violée, ils **déclenchent l'action** associée qui peut avoir un effet sur le système de développement lui-même.

1.2.5 Conclusion

L'expertise proposée dans SKB peut être très utile pour l'implantation d'un environnement de génie logiciel, mais la séparation du modèle et de la réalité rend presque caduque le développement d'un système SKB. Pourquoi, dans ce cas, ne pas envisager l'utilisation d'un SGBD classique et de l'étendre en lui rajoutant la prise en compte des contraintes (certains SGBD gérent déjà, par exemple, des contraintes telles que celle de référence). En outre l'état embryonnaire et stationnaire du projet ne permet pas d'autres conclusions quant à ses qualités ou défauts.

1.3 Le projet PMDB

1.3.1 Introduction

Le projet **PMDB** (Project Master Data Base) développé par la société TRW (Redondo beach, Californie) a pour objectif la conception et la réalisation d'une "Base de Données projet" comme partie intégrante des environnements de développement de logiciels et est à l'heure actuelle à l'état de prototype. Les résultats de cette étude sont dans [PEN85][PEN86]. En effet Penedo considère la base de données projet (PMDB) comme le cœur des environnements de développement. Ces bases de données dans lesquelles sont stockés les produits du développement ou les méthodes de développement et de gestion sont encore trop souvent gérées manuellement et l'objectif du projet PMDB est le passage à des bases de données actives et automatisables. Les composants d'une telle base devront être intégrés et celle-ci doit permettre un accès et une manipulation simples des données stockées.

Les différentes étapes dans la réalisation de cet objectif sont :

- l'identification du modèle pour une telle base,
- l'identification des fonctions associées à une telle base,
- l'identification des vues utilisateurs d'une telle base,
- l'implantation des notions précédentes.

1.3.2 Le cahier des charges de PMDB

L'identification du modèle passe d'abord par l'identification des objets à modéliser et les différentes interactions entre ces objets. Les objets en développement de logiciels comprennent des produits (résultats de projets) comme un programme ou un manuel, des ressources (consommables par le projet) comme de l'équipement ou de la main d'œuvre et des politiques d'affectation des ressources aux produits à développer.

L'identification des fonctions associées à une telle base consiste à recenser les manipulations effectuées sur les données d'un projet en vue d'en automatiser un certain nombre. Cette identification doit recenser les procédures de manipulation de la base de données, mais aussi associer à celles-ci ses *"composants actifs"*. Ces composants actifs sont les mécanismes permettant de mettre en œuvre ces fonctions.

L'identification des vues utilisateurs doit permettre de définir des environnements de travail adaptés aux besoins et aux prérogatives de chaque type d'utilisateur. Dans ce but, la base projet devra supporter différents niveaux d'abstraction, qui pourront être obtenus par la manipulation d'objets composites ou par différents degrés de granularité.

L'identification de ces différents aspects d'une base projet n'a cependant pas été faite de manière descendante et formelle. Penedo a choisi une stratégie de **prototypage** (modèle en spirale). C'est-à-dire que chaque prototype réalisé doit amener de nouvelles spécifications pour le PMDB. Le prototype suivant intègre ces nouvelles spécifications et en introduit de nouvelles, et ainsi de suite. Par ailleurs, tous les prototypes développés permettent de choisir une implantation finale efficace en fonction des résultats et des problèmes produits par les prototypes successifs.

1.3.3 Le Modèle

Le modèle de données adopté dans le PMDB est le modèle E-A de Chen [CHE76].Les trois composants de base du modèle retenu sont les objets, leurs attributs et les associations existant entre eux.

Les **objets** caractérisent le type des données du PMDB. Ceux-là sont instanciés et chaque instance représente ainsi une occurrence différente du même type d'objet. Les

attributs décrivent les caractéristiques des objets. Ces attributs prendront des valeurs spécifiques à chaque instanciation d'objet auquel ils sont associés. Les **associations** représentent, elles, les différentes connexions existant entre les objets.

31 types d'entité, 220 types d'attribut et 170 types d'association ont été retenus pour pouvoir modéliser le développement d'un logiciel -de la phase de faisabilité à la phase de maintenance-. Parmi les 31 types d'entité dénombrés, on peut citer les types d'entité personnes, composant logiciel, document, procédure de test, rapport d'anomalie, outil, etc....

Citons aussi quelques-uns des types d'attributs d'un rapport d'anomalie : numéro d'identification, intitulé, sévérité, solution proposée, etc.... Et enfin quelques relations : une personne en gère d'autres, une personne gère des composants logiciels, un rapport d'anomalie affecte un document, etc ....

1.3.4 Le prototypage

Complétion du cahier des charges

Le prototype a été réalisé en implantant le modèle du PMDB grâce au SGBD Ingres et au système Unix, en identifiant tous les types de données et en implantant les mécanismes du PMDB. Ce prototype a ensuite été validé en utilisant un scénario de développement et en vérifiant l'adéquation du PMDB à un tel scénario.

Le **modèle du PMDB** étant un modèle entité-association, il a fallu proposer un schéma de traduction des entités, de leurs attributs et de leurs associations en relations du modèle relationnel utilisé pour Ingres, et en fichiers du système Unix. En effet, il a été décidé de ne pas se limiter aux relations Ingres pour représenter tout composant du modèle. Ceci en vue d'autoriser la manipulation de données textuelles de grande taille, impossible à réaliser en relationnel pur.

La traduction du modèle E-A au modèle relationnel étendu a été faite en respectant les règles suivantes :

- Pour chaque objet, une relation au moins est créée et contient l'identificateur de l'objet et quelques-uns de ses attributs.
- Certains attributs deviennent soit des attributs calculés, soit des relations.
- Dans la plupart des cas, une association a été traduite par une relation, mais dans certains cas, n associations participent à la création d'une seule relation appelée alors relation "quelconque". Par exemple toutes les différentes associations aboutissant à un même type d'objet sont parfois regroupées en une relation unique dont l'un des attributs spécifie le type de relation source (représentant un objet).

L'identification de tous les **types de données** et des différents **mécanismes**, à rajouter aux mécanismes simples de Ingres, permettant de gérer ces données a été la deuxième étape dans la réalisation du prototype. Cependant ces mécanismes n'ont pas été implantés. Seules les possibilités offertes par Ingres ont été utilisées, en particulier tous les mécanismes permettant de déclencher des actions comme les triggers n'ont pas été utilisés car ils ne sont pas implantés par Ingres.

La liste de ces types de données et de mécanismes mis à jour, dans [PEN86], comprend :

- le type application implantant un algorithme,
- le type calculé permettant l'obtention d'une valeur d'attribut par calcul,
- un mécanisme de cohérence maintenant la consistance de la base,
- un mécanisme démon (ou garde, ou trigger) surveillant la survenance d'un événement et exécutant une action dans ce cas,
- le type document représentant une information textuelle,
- le type externe représentant toute information pouvant être obtenue d'une autre base que le PMDB,
- le type textuel représentant une information textuelle non bornée (représenté par un fichier Unix),
- le type structure permettant la composition de plusieurs types -ou même structures- en une seule structure,
- le type objet correspondant à un objet réel existant,
- le type requête,
- le type "nouveau type" permettant l'extension de la sémantique du PMDB,
- le type relation,
- le type domaine.

Cette étape d'identification est faite en vue d'une implantation finale éventuelle, mais n'intéresse que faiblement la réalisation du prototype. Seuls certains de ces types et mécanismes seront mis en œuvre.

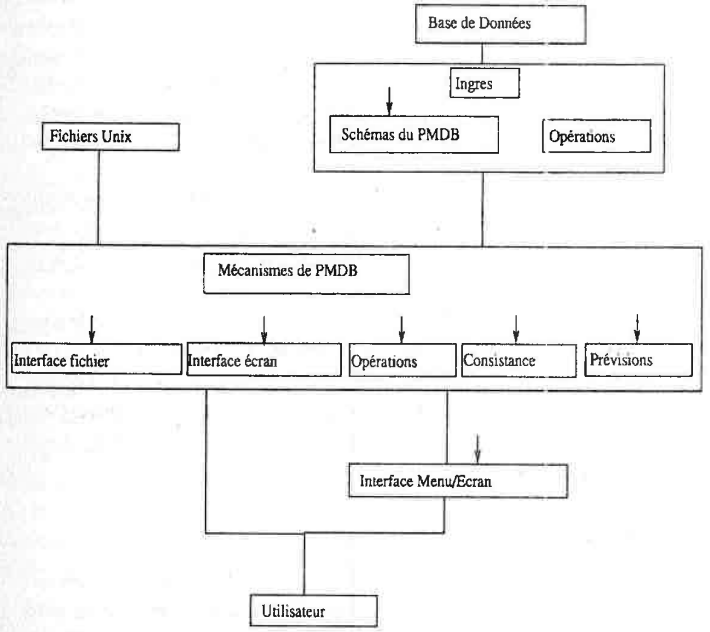


Figure 1.1. Architecture générale du prototype PMDB

Implantation de quelques mécanismes du cahier des charges

L'architecture générale du prototype PMDB est représentée par la figure 1.1 [PEN86] les composants développés par TRW étant les boîtes fléchées.

La **cohérence** ou **consistance** de la base doit être préservée. Cette cohérence regroupe deux propriétés disjointes :

- la consistance intrinsèque du schéma PMDB manipulé, représentant les propriétés sémantiques du schéma ;
- le maintien de la cohérence, celle-ci risquant d'être violée à cause de la redondance d'informations dans le modèle support (relationnel) lui-même.

L'option choisie consiste à résoudre quelques aspects du deuxième problème en implantant un mécanisme de maintien de la contrainte de référence, qu'elle mette en jeu des valeurs de domaines ou des valeurs de relations. Ces contraintes de référence, classiques dans le modèle relationnel [DAT83], expriment le fait qu'une valeur référencée doit exister dans la base. Par exemple si l'attribut a de la relation R est l'identifiant i de la relation R', il doit toujours exister une instance de la relation R' dont l'identifiant est i. Ou encore, en termes plus parlant, on ne peut affecter une ressource à une tâche que si cette ressource existe. Ce mécanisme a été implanté en créant et maintenant dynamiquement des dépendances entre les valeurs se référençant. Une **opération** du PMDB sera donc d'abord filtrée par ce mécanisme de maintien de la consistance avant d'être traduite en opération Ingres.

L'interface écran réalisée pour le PMDB est extraite de l'environnement de production de logiciel de TRW et fournit une interface conviviale à l'utilisateur final.

L'interface fichier est le mécanisme permettant de rendre transparent à l'utilisateur le stockage des données dans la base de données Ingres ou dans le système de fichier Unix. Les données textuelles non bornées sont détectées par le PMDB et stockées en conséquence dans le système de fichier Unix.

1.3.5 Conclusion

La conception et le prototypage de PMDB ont plus servi à identifier les problèmes qu'à les résoudre. Un problème spécifique à l'implantation d'un modèle par un autre (dans ce cas le modèle E-A par le modèle relationnel) a été l'apparition d'effets de bord indésirables comme la redondance des données et la difficulté à automatiser un tel passage (indépendamment de PMDB, des travaux sur ce problème ont été présentés dans [KIM86] et [KIM86b]). Par ailleurs, le système Ingres choisi -représentatif des SGBDs actuels- montre assez vite la limite de ses capacités en ce qui concerne les mécanismes de cohérence et de déclenchements d'actions. Et enfin, l'introduction d'un grand nombre de relations (50 relations pour représenter le sous-ensemble des objets, attributs et relations du modèle PMDB ayant servi de base à l'expérimentation) rend l'utilisation du langage de requêtes relationnel lourd et inapproprié. Une interface conviviale avec des aides à la manipulation de la base est souhaitée.

1.4 Conclusion

La conclusion de ce travail de TRW est, de même que pour le projet SKB, une ouverture sur la suite.

Le travail d'étude réalisé par Penedo pour TRW-PMDB est intéressant par l'introduction et le typage des différents objets et relations intervenant dans le développement de logiciels. Ce recensement est plus complet que celui proposé par Meyer pour SKB, mais les deux projets sont très proches par leurs objectifs ainsi que par les résultats obtenus. A savoir, un recensement des besoins en données et en système de gestion de ces données pour le génie logiciel.

Nous avons vu dans ce chapitre comment répondre aux besoins de gestion en développement de logiciels en couplant un SGBD avec un atelier ou avec des outils de développement existant. Cette solution peut être perçue comme intéressante si le but déclaré est l'indépendance entre les objets du développement et leur représentation (c'est le cas pour SKB, par exemple). C'est-à-dire modéliser un projet de développement de logiciels comme une entreprise quelconque sans tenir compte de la spécificité du génie logiciel.

Cette gestion se fait alors de manière indépendante du développement de logiciel et le gestionnaire a à sa disposition des informations censées représenter les informations du système de développement.

Cela peut correspondre à une duplication de l'information si le gestionnaire manipule la date de dernière modification d'un fichier et que le développement du logiciel se fait sous Unix. En effet, Unix gère lui-même une telle information. Cette duplication pourra donc être source d'incohérence. Le gestionnaire se basera sur la date à sa disposition pour prendre des décisions qui pourront donc être hors de propos.

D'autre part, comment considérer une décision prise par le CRAC de SKB de déclencher une opération dans le système de développement ? Est-ce vu uniquement comme un conseil au développeur, ou alors comme une opération réellement entreprise dans le développement de logiciel ? Dans ce cas, comment savoir si l'opération déclenchée est effectivement prise en compte dans le développement, vu l'indépendance de la gestion par rapport au développement.

Nous voyons donc clairement apparaître les limites d'une telle approche même si elle permet de faire avancer la réflexion sur le sujet. En effet la séparation du problème de la gestion de projet du développement permet de se concentrer sur certains aspects : en particulier le recensement et l'étude des différents types d'objets et des relations entre eux a certainement été possible grâce à cette abstraction.

2
Atelier Intégré de Génie Logiciel : un exemple de réalisation

L'atelier Concerto [AND86] développé par le CNET est un bon exemple d'atelier paramétré. Nous désignons par atelier paramétré un atelier qui ne soit pas totalement figé. Un atelier figé est un atelier dans lequel tous tous les choix quant à son utilisation ont déjà été faits lors de sa construction et tous ces choix sont immuables :

- choix d'un langage de développement,
- choix d'une "méthodologie" de développement,
- choix d'une organisation du projet de développement,
- choix des outils à la disposition de l'utilisateur de l'atelier,
- ...

Un atelier paramétré est donc un atelier permettant de choisir, à posteriori, les paramètres effectifs de développement que sont les langages de développement utilisés, les méthodes de développement, ... Le projet GIPE (Generation of Interactive Programming Environments) [GIP87] et le projet Concerto [AND86] sont des exemples de tels ateliers paramétrés.

Nous commencerons par décrire l'atelier Concerto tel qu'il a été présenté à la fin théorique du projet en février 1986, puis tel qu'il apparaît à l'heure actuelle et enfin nous finirons notre étude par quelques remarques quant à son utilisation.

Nous avons vu les limites et les avantages de la première approche -à savoir couplage d'un SGBD à un atelier de développement existant ou à un ensemble d'outils- dans le chapitre précédent.

L'approche suivie par les développeurs de Concerto représente le deuxième choix possible pour le développement d'environnements de développement et de gestion de projets : développer un certain nombre d'outils capables de coopérer entre eux et fournir à

l'utilisateur un AIGL complet comprenant à la fois des outils de développement et des outils de gestion.

Nous avons choisi Concerto comme représentant des ateliers intégrés de génie logiciel pour plusieurs raisons :

- il a été conçu en tant que tel et n'est donc pas une simple boîte à outils. Bien que ses composants aient pu être développés par différentes équipes, ceux-la s'intègrent ensemble et de ce fait forment un AIGL.
- il donne un bon aperçu de l'état d'avancement en matière d'ateliers,
- et enfin, il est paramétré. Bien qu'il ait été conçu comme un atelier avec ses outils et ses méthodes spécifiques, une volonté d'évolutivité et d'ouverture a été prise en compte, en particulier par la paramétrisation du langage de développement de l'atelier grâce à PPML et Metal, et par la paramétrisation de l'organisation du projet logiciel grâce à F1.

2.1 Concerto en 1986

2.1.1 Présentation générale

Les deux mots clés du projet Concerto sont *"intégration et flexibilité"* [AND86]. Toute la démarche des concepteurs sera axée autour de la recherche d'une solution présentant un atelier flexible dont tous les outils coopéreront et présenteront une interface unique à l'utilisateur.

L'architecture générale (cf. figure 2.1) proposée repose sur trois composants (système opérant, système d'information et système de pilotage) coopérant entre eux grâce à un quatrième (système de communication).

L'atelier Concerto se propose donc d'étudier et de développer chacun des quatre constituants pris séparément puis d'en effectuer l'intégration.

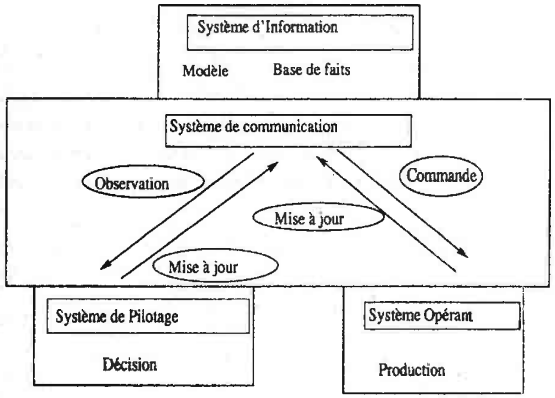


Figure 2.1. Architecture générale de l'atelier Concerto

2.1.2 Le système de communication

Ulysse, un langage d'acteurs

Le système de communication est basé sur le langage d'acteurs Ulysse [AUT85]. Du point de vue d'Ulysse, tous les outils de base composant Concerto sont des acteurs qui s'activent et communiquent entre eux par envoi de messages de différents types. Nous citerons juste les types de messages sans entrer dans le détail :

- envoi simple
- envoi avec attente de réponse
- envoi avec attente de terminaison

auxquels on ajoutera un nouveau type d'envoi de message permettant de faire coopérer différents sites de l'atelier (Concerto est alors indifféremment considéré comme un ensemble de sites ou machines ou postes de travail) :

- envoi distant (sur un autre site)

Les primitives de communications étant établies, l'environnement ISA [ROU86] met en œuvre le système de communication de l'atelier.

ISA, l'environnement de communication

Cet environnement constitue l'élément essentiel d'intégration de l'atelier.

L'ISA permet la définition logique d'un poste de travail grâce aux tâches de ce poste. Ces tâches définissent, d'une part, un mode de structuration du poste de travail, et, d'autre part, une structuration de tout l'atelier grâce à la notion de sous-traitance. La sous-traitance est la soumission d'une demande de travail -réalisation d'une tâche- d'un agent à un autre. Un agent étant défini par l'attribution d'une tâche racine qui lui est affectée à chaque session.

La notion de tâche est définie comme étant l'élaboration d'objets définis par un agent défini.

Les fonctions principales de l'environnement ISA sont de :

- Représenter sur un poste de travail les tâches qui doivent s'y dérouler;
- Gérer ces tâches, c'est-à-dire fournir les opérateurs élémentaires permettant de connaître celles-ci, de les affecter et de les accepter ou de les refuser.

L'environnement ISA permet donc de manipuler la notion de tâche, mais cela serait insuffisant s'il ne permettait pas, par ailleurs, d'aider à la réalisation de cette tâche c'est-à-dire à l'élaboration d'objets. Cette aide à la réalisation d'une tâche se fera par l'association d'un domaine à chaque tâche.

Un domaine regroupe et structure toutes les informations nécessaires à la réalisation de la tâche associée. Ces informations sont l'ensemble des objets en entrée de la tâche, l'ensemble des outils nécessaires à sa réalisation, et enfin tous les objets produits que l'agent désire conserver. Cet ensemble d'objets produits englobe, bien sûr, les objets résultats attendus de la tâche.

Donc tout agent peut définir une tâche par son domaine, et la soumettre, par sous-traitance, à un autre agent pour que celui-ci la réalise s'il l'accepte, et rende un résultat à l'agent demandeur.

Toute la description précédente correspond à la description de l'environnement de communications d'un atelier décentralisé où toutes les décisions peuvent se prendre à un niveau local -celui du poste de travail-. Pour prendre en compte la globalité de l'architecture (figure 2.1), il faut rajouter une nouvelle fonction à l'ISA :

- Assurer la jonction entre le système d'information décrivant l'organisation de l'atelier et les tâches devant intervenir dans cette organisation.

Cette jonction permet au système d'information de centraliser toute la connaissance relative à l'affectation et l'état des tâches et permet ainsi au système de pilotage d'en déduire, à partir des règles définissant la méthode de travail, de nouvelles tâches à soumettre. On voit ici apparaître le rôle intégrateur de l'environnement de communications.

2.1.3 Le système opérant

Le poste de travail étant l'élément permettant l'accès au système opérant, nous commencerons par décrire celui-là, puis nous décrirons différents environnements de travail sur le système opérant par l'intermédiaire de ce poste de travail.

Le poste de travail

Nous pouvons scinder le poste de travail en deux couches complémentaires :

- l'interface utilisateur
- la manipulation des objets de Concerto et leur représentation interne

1. L'interface utilisateur.

   Cette interface est caractérisée par des mécanismes d'affichage, d'interaction parallèle et de manipulation structurelle.

   Le système d'affichage est réalisé par le système Apotre [CAN86] qui réalise les fonctions d'accès au poste de travail. Apotre réalise des fonctions, devenues classiques maintenant, telles que la création d'une fenêtre, sa désignation à partir d'un clavier ou d'une souris, la gestion de menus... Apotre gère donc la fonction d'affichage et communique avec les autres composants (outils) du poste de travail grâce à un ensemble d'acteurs écrits en Ulysse (appelés Dialogueur).

   L'interaction parallèle dans Concerto revient au multifenêtrage dans un système multi-tâches.

   La manipulation structurelle est une composante importante du poste de travail. En effet, on manipule des objets qui ont une structure précise définie par une grammaire ou un formalisme (langages de programmation, formalismes de spécifications, conventions de documentations...).

   Le but de cette manipulation structurelle est de manipuler non plus une suite de caractères dans un éditeur mais de manipuler un objet structuré. Cette notion de structure est intimement liée à la représentation interne adoptée par Concerto, l'interface utilisateur permettant d'accéder grâce à cette manipulation structurelle à une vue proche de la représentation interne. Cette manipulation structurelle est réalisée par un éditeur syntaxique générique manipulant la syntaxe du formalisme décrivant chaque type d'objet.

2. La représentation interne (RI).

Tous les objets Concerto ont une représentation interne unique arborescente. Cette représentation interne reprend celle du système Mentor [DON75] et est réalisée sous forme d'arbres étiquetés décrits par une syntaxe abstraite.

Chaque formalisme utilisé dans un atelier de génie logiciel ayant une syntaxe abstraite différente, deux solutions pour la manipulation d'arbres abstraits sont envisageables :

- Fixer à priori les différents formalismes de l'atelier et décrire pour chacun d'eux la structure des arbres abstraits associés et les fonctions de manipulation de ceux-ci.
- Définir un méta-formalisme permettant de décrire les formalismes et associer des fonctions génériques de manipulation d'arbres à ce méta-formalisme.

Concerto adopte la deuxième solution en utilisant le méta-formalisme Metal [KAH83] de Mentor pour décrire la syntaxe abstraite du formalisme utilisé. Le compilateur associé à Metal permet le passage de tout texte donné dans un formalisme lui-même décrit en Metal à sa représentation sous forme d'arbre abstrait.

La syntaxe abstraite d'un objet est la référence commune pour tous les outils manipulant des objets de ce type. Un outil manipulant un objet peut modifier l'arbre abstrait qu'il faut pouvoir revisualiser et manipuler de la même manière qu'un objet directement saisi sous forme textuelle. Concerto propose donc un outil de passage d'un arbre abstrait au texte correspondant. PPML [MOR86] est le méta-formalisme de cet outil de décompilation.

Nous résumons dans la figure 2.2 une vue du poste de travail Concerto tel que nous l'avons décrit.

La définition de ces fonctions génériques pour le poste de travail permet de construire des environnements de travail particuliers mais dont l'interface utilisateur est la même que sur n'importe quel autre poste de travail. Nous ne reviendrons plus sur cet aspect des choses, mais il faut noter que les fonctionnalités décrites ci-dessus sont utilisées comme constantes sur tous les postes de travail, qu'ils soit dédiés à la production, à la documentation, au pilotage ou à la gestion. En particulier les différents environnements que nous allons décrire instancient tous un poste de travail particulier.

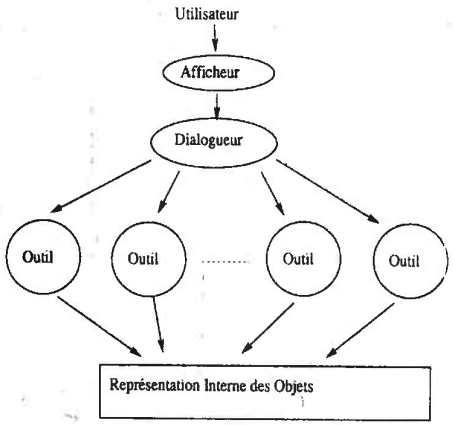


Figure 2.2. Poste de travail Concerto

Les différents environnements

Définir un poste de travail revient à définir le formalisme utilisé et à définir un certain nombre d'outils. La définition du formalisme est caractérisée par ses descriptions en Metal et PPML. Cette première étape fournit déjà en sortie tous les outils générés pour l'atelier Concerto, à savoir :

- un éditeur caractère
- un éditeur structurel
- un environnement multi-tâches
- un dialogueur pour le traitement des commandes dont l'interface est unique quel que soit l'outil.

Dans ces conditions, un poste de travail minimal est fourni à l'utilisateur dès la première étape. Mais le domaine du génie logiciel étant vaste, les outils manipulés à différents stades du cycle de vie du logiciel ne sont pas tous des instances d'outils génériques propres au génie logiciel. C'est pourquoi, une seconde étape de création des outils spécifiques à certaines étapes du cycle de vie du logiciel devra être entreprise.

Cependant, pour ce faire, les développeurs de tels outils pourront s'appuyer sur les outils de bases de Concerto tels que le dialogueur ou l'environnement multi-tâches. Nous décrivons succinctement quelques-uns des différents environnements que le CNET a développés pour mettre en œuvre les idées proposées dans l'élaboration du poste de travail.

1. L'environnement de programmation.

   L'équipe de programmation s'est d'abord tournée vers des environnements Pascal et Chill, Chill étant le langage de programmation normalisé par le CCITT et utilisé par le CNET pour la programmation des systèmes de télécommunications. Leur but étant de fournir trois outils d'aide à la programmation dans chacun de ces langages. Ces outils sont :

   - un contrôleur sémantique
   - un interpréteur / metteur au point
   - un compilateur multi-machines

   (a) Le contrôleur sémantique.

   L'atelier Concerto fonctionnant avec un éditeur structurel et possédant un compilateur Metal et un décompilateur PPML, l'analyse syntaxique est complètement ignorée au niveau du contôleur. Ceci permet de réaliser un outil cohérent ne s'intéressant qu'à un seul type de problèmes, les problèmes de sémantique. Le contrôleur travaille sur l'arbre de Représentation Interne (RI) d'un programme syntaxiquement correct mais éventuellement incomplet, l'arbre pouvant comporter des sous-arbres non encore développés.

   Le contrôleur sémantique associe à chacun des nœuds de la RI d'un programme Pascal ou Chill un attribut. Cet attribut a pour rôle de repérer des erreurs éventuelles, de les typer, d'évaluer statiquement toute partie du programme qui peut l'être et d'établir des liens entre la déclaration et l'utilisation des objets d'un programme. Pour ce dernier point, le contrôleur réalise la gestion des noms en tenant compte des règles de portée des identificateurs dans chacun des langages et vérifie l'adéquation des différents types intervenant dans le programme.

   L'arbre étiqueté ainsi produit, permettra au contrôleur de détecter un certain nombre d'erreurs et de préparer le travail des deux outils suivants, à savoir l'interpréteur/metteur au point et le générateur de code multi-cibles.

   (b) L'interpréteur / metteur au point.

   L'interpréteur est chargé de simuler l'exécution d'un programme éventuellement incomplet. Il réalise donc l'allocation des adresses mémoires de chaque variable,

l'évaluation d'expressions et le traitement des différentes instructions. Il doit réaliser tout ceci de façon interactive.

C'est le rôle du metteur au point de fournir une telle possibilité. Celui-ci est constitué de trois ensembles de fonctions avec une aide en ligne pour chacune d'entre elles.

Le premier ensemble de fonctions permet de contrôler l'exécution de l'interpréteur. Cette exécution peut être arrêtée, reprise au même endroit ou pas, effectuée suivant différents modes (pas à pas, transparent ou automatique). En outre des points d'arrêt peuvent être insérés à différents endroits du programme en cours d'exécution.

Le deuxième ensemble de fonctions regroupe des primitives de visualisation des objets du programme.

Et enfin, le troisième ensemble permet d'intervenir sur les variables du programme lors de l'exécution. En particulier, la trace de certaines variables pourra être obtenue, une fenêtre dédiée montrant les valeurs successives de ces variables. En outre, une protection en écriture de variables pourra être faite. Elle aura pour effet l'arrêt de l'exécution lorsqu'une instruction du programme tentera de modifier une de ces variables et l'affichage de la raison de cet arrêt.

L'interpréteur/metteur au point permet d'arriver à un programme "validé" dont on peut avoir une idée raisonnable de son exécution grâce à l'interpréteur. Cependant une dernière étape est nécessaire pour le produit final : un compilateur.

(c) Le compilateur multi-machines.

Cet outil appelé Gemme [SAN86] reçoit en entrée un programme validé sous la forme d'un arbre et réalise son codage en un programme assembleur destiné à une machine cible particulière. Celle-ci est spécifiée grâce à un langage de description qui permet à Gemme de connaître toutes les caractéristiques de la dite machine en vue de la génération du code assembleur adéquat.

Utilisant cette description de la machine cible, un codeur génère donc un programme assembleur qu'un optimiseur de code transforme, en dernier lieu, en un code performant.

L'optimisation est faite en deux étapes. La première consiste à optimiser le code indépendamment de la machine cible, c'est-à-dire que les optimisations portent principalement sur la suppression d'instructions inaccessibles, la factorisation des séquences identiques, la suppression des branchements à l'instruction suivante, la transformation des branchements en cascade en branchements directs et enfin l'inversion de certaines conditions de branchements.

La deuxième phase de l'optimisation consiste à choisir les instructions de branchements les plus efficaces de la machine cible pour réaliser tous les branchements afin d'accélérer l'exécution finale.

Ce code assembleur généré par Gemme est alors prêt à être fourni à l'assembleur de la machine cible.

Ces deux environnements de programmation ont été développés pour montrer l'intérêt d'une telle approche paramétrée. Il est clair qu'un nouvel environnement de programmation pour tout autre langage pourra être développé, le poste de travail minimal garanti étant celui fourni de manière générique grâce à Metal et PPML.

2. L'environnement de documentation.

L'environnement de documentation, décrit ici, s'intéresse bien sûr à une documentation technique, donc fortement structurée et dont la manipulation diffère fortement de la manipulation d'un roman. En effet, la manipulation d'une telle documentation relève plus d'une activité de parcours global avec la lecture de parties de textes restreintes et fortement enchevêtrées que d'une lecture séquentielle. Par ailleurs, l'outil informatique crée de nouveaux besoins et de nouvelles commodités pour la manipulation de documents. L'environnement devra donc gérer les deux aspects. Cet environnement repose sur les mêmes bases que les "systèmes hypertextes" permettant le traitement de documents complexes. Citons comme exemples des systèmes hypertextes NoteCards [HAL87], IBIS [RIT73], Hypercard [GOO88] et enfin Hyperbook [KEM88] [KEM89] qui est réalisé autour du SGO de PCTE [PCT86]. Notons en outre la proposition d'une interface relationnelle de type hypertexte faite dans [ARD89].

Les outils spécifiques à cet environnement de documentation de Concerto réalisent donc deux types de fonctions :

- des fonctions que réalise tout lecteur d'un document technique sur papier
- des fonctions spécifiques à l'édition automatique.

Dans la première catégorie, il est permis à l'utilisateur d'annoter son document, c'est-à-dire qu'il n'est pas entièrement restreint par le formalisme document. Cette annotation se traduit par une décoration de l'arbre de RI.

D'autre part, il est rare qu'un lecteur fasse une lecture séquentielle d'un document en commençant à la page un et en finissant à la dernière page. Dans cette optique l'environnement minimal du poste de travail permet déjà, grâce à l'édition structurelle, de voir le document à différents degrés de raffinement. Ne visualiser, par exemple,

que les titres des différentes sections du document ne nécessitent pas de nouveaux développements au niveau de l'environnement de documentation Concerto. Par contre, passer d'un paragraphe à celui du chapitre précédent traitant du même sujet, à l'annexe référencée, à une note en bas de page ou à une référence bibliographique sont des activités courantes dans le parcours d'une documentation technique. Pour permettre cette activité, l'environnement de documentation Concerto permet de créer et de manipuler des liens entre différentes parties d'un même document, c'est-à-dire entre différents sous-arbres de l'arbre de la RI. L'utilisateur peut donc créer ou détruire un lien d'un sous-arbre à un autre, parcourir les liens partant d'un endroit donné ou encore parcourir "à l'envers" les liens arrivant à un endroit donné.

Par ailleurs les possibilités qu'offre l'édition automatique de Concerto peuvent être résumées comme suit :

- une fonction documentaire
- une fonction de manipulation de schéma

La fonction documentaire [CHI86] est basée sur un thésaurus et une base d'indexation. Le thésaurus représente les concepts associés aux textes et la base d'indexation les relations de pertinence entre ce thésaurus et des éléments de textes (sous-arbres de la RI). La réalisation de ces deux ensembles de données est donc la première étape dans la réalisation de cette fonction documentaire. Un exemple d'une telle réalisation est donné dans [FER86].

La fonction de manipulation de schémas, quant à elle, constitue un sous-environnement spécifique aux schémas [FER86]. Un schéma est écrit en SL4 qui est un langage textuel de description de schémas. SL4 est bien sûr manipulé grâce à l'éditeur structurel général qui permet de construire la RI d'un schéma à partir de sa description textuelle en SL4. Cependant la manipulation de schémas ne saurait se limiter à l'acquisition et à la manipulation de schémas sous forme textuelle. Le but de l'environnement de manipulation de schémas (non encore achevé en 1986) est de permettre la manipulation de schémas sous forme graphique. Pour ce faire, un traducteur d'arbre transforme la RI en une RISchéma spécifique aux schémas et adaptée à la manipulation graphique de schémas.

Cette RIS est alors utilisée par un générateur et un éditeur de schémas. Ceux-ci servent d'interface entre la RIS et un afficheur de schémas qui permet une représentation graphique de schémas. La RIS permet de stocker -grâce aux facettes associées aux différents nœuds de cette représentation- :

- le nom, la forme et la taille de chaque objet

- la position relative des différents objets
- les différentes connexions entre objets.

Le sous-environnement schéma sus-décrit s'intègre dans l'environnement de documentation par le fait qu'un schéma est considéré comme un élément terminal du formalisme document de la même manière qu'un pavé de textes. C'est le formalisme document qui permet à l'utilisateur de manipuler les pavés (feuilles de la RI) de manière globale, mais pour une manipulation spécifique d'un schéma, telle que l'affichage sous forme graphique, l'environnement de documentation fait appel à l'environnement schéma.

2.1.4 Le système d'information

F1 [CAZ86] est le formalisme du système d'information. Il est à la fois le langage de définition de données (LDD) et le langage de manipulation de données (LMD). F1 est écrit en Prolisp [ZAN84] et réalise les fonctions du système d'informations.

F1 permet la description :

- d'entités et de relations (modèle de données),
- de lois et d'actions pour la consultation et la mise à jour d'une base de faits -ou base de données- avec la possibilité d'effectuer du calcul logique pour déduire de nouveaux faits ou pour vérifier des contraintes sur des faits déjà présents.

Le langage de définition de données

Le langage de définition de données est basé sur trois concepts génériques (les domaines de valeurs, les types d'entité et les relations). Le LDD de F1 est une extension du modèle Entité-Relation de CHEN [CHE76]. Les extensions étant :

- l'héritage multiple pour les domaines et les types dérivés.
- l'expression de contraintes par
  - des clés d'unicité
  - des contraintes sur les attributs
  - des contraintes de cardinalité
- la possibilité de polymorphisme, le polymorphisme permettant de définir une nouvelle loi (voir plus loin) en réutilisant un nom déjà affecté pour un autre élément (loi, fonction, action,...).

1. Les domaines de valeur.

   Ils sont de différentes sortes (les italiques représentent les mots-clés de F1) :

   - Prédéfinis (*entier*, *reel*, *symbole* et *chaine*)
   - Définis par énumération (*discret*)

   ```
   ex: ( domaine booleen
         ( discret vrai faux))
   ```

   - Dérivés de domaines
     - par héritage simple ou multiple (*sorte_de*)
     - et/ou par contraintes (*respecte*)

   ```
   ex: ( domain date ( sorte_de entier)
         ( respecte ((date *d)
         (et ( evalt ( >  '*d 1987)) ( evalt ( <  '*d 1992))))))
   ```

   N.B. : En F1 les variables commencent par une étoile (*d est une variable date).

2. Les types d'entité.

   Ils sont définis par :

   - un ensemble d'attributs, chaque attribut prenant sa valeur dans un domaine
   - une clé d'unicité (*cle*)
   - des contraintes sur leurs attributs (*respecte*)

   ```
   ex: ( type fichier
         ( structure ( entier id)
                 ( symbole nom_auteur)
                 (date date_de_dernière_maj))
          ( cle id)
          ( respecte ((fichier *f) ( symbole *nom))
                 ( existe (*nom) ( et (nom_auteur *f *nom )
                                      ( diff *nom benali)))))
   ```

   Cet exemple traduit le fait qu'un fichier est complétement identifié par id (2 fichiers différents ne peuvent avoir le même id) et que benali n'a pas le droit d'écrire de fichiers!

3. Les relations.

   Les relations sont :

- définies entre des types d'entité
- composées de deux sortes d'attributs
  - des attributs-objets ou rôles
  - des attributs-valeurs

et elles ont :

- une clé d'unicité (*cle*)
- des contraintes sur leurs attributs (*respecte*)
- des cardinalités (minimum et maximum) portant sur les rôles

```
ex: ( relation subit_test
      ( structure
            (module &module)
                  (test &test)
                  ( entier note))
      ( cardinalite (&module 0 10)
                  (&test 0 6)))
```

subit_test est une relation entre le type d'entité module (à travers le rôle -ou attribut-objet- &module) et le type d'entité test. Les cardinalités traduisent le fait qu'un module doit être soumis à t tests ($0 < t < 10$) et qu'un test est imposé à m modules ($0 < m < 6$).

4. Les types dérivés.

Comme pour les domaines, il est possible de définir un type par dérivation et héritage (l'héritage peut être multiple s'il existe un ancêtre commun et qu'il n'y a pas d'ambiguïté sur les attributs). Définir un type par dérivation-héritage (*sorte_de*) signifie pour ce type dérivé que :

- il hérite des types pères
  - les attributs
  - les contraintes
  - les relations et lois (nous définissons les lois par la suite)
- il peut être étendu en ajoutant
  - de nouveaux attributs
  - de nouvelles contraintes


```
ex: ( type module ( sorte_de fichier)
      ( structure (booleen compilé)...)
      ( respecte  (( entier *id) )
            ( existe (*id) ( et  (id *id) ( evalt
                  ( >  '*id 100)))))
```

Le langage de manipulation de données

Le langage de manipulation de données, quant à lui, est basé sur deux concepts :

- la sélection et la consultation
- la mise à jour

1. Sélection et consultation.

Les expressions de sélection sont écrites dans un langage logique typé du premier ordre en termes d'atomes liés par des opérateurs logiques. Un atome est un prédicat appliqué à une liste de termes. Un terme est une constante, une variable typée, ou un appel de fonction. Les expressions sont résolues par un mécanisme but orienté, après un contrôle de type.

Comme F1 est un langage logique du premier ordre, il possède des variables. Elles sont de quatre sortes (et commencent toutes par *) :

- variable-valeur $\in$ domain
- variable-objet $\in$ object type
- variable-tuple $\in$ relation
- variable-liste $\in$ l' interface le_lisp (F1 étant écrit en prolisp)

Les expressions sont écrites dans une notation préfixée parenthésée et leur résultats sont des listes (éventuellement vides).

Comme les expressions de sélection sont des expressions logiques, nous devons définir :

- les opérateurs (*et, ou, impl, ou_ex, cond, non*)
- les quantificateurs (*exist, exist_un, exist_un_seul, qqsoit*)
- les atomes qui sont des prédicats appliqués à des termes :
  - prédicats sur type d'entité e.g. : (personne *id benali *gr)
  - prédicats sur relations e.g. : (initial *pers *mod)

- fonctions d'accès à un attribut e.g. : (id *pers *id)
- prédicats d'appels de lois e.g. : (assertion2)
- prédicats prédéfinis *id*, *diff* et *positionnel* (*positionnel* associe un objet à tous ses attributs)

F1 n'est pas un langage complétement fermé : il autorise des échappemment vers le_lisp grâce à deux prédicats (*evalt* et *eval*). Les expressions logiques peuvent être nommées et sont alors appelées *loi* en F1. F1 permet aussi la définition de *fonction*.

Les expressions de sélection sont écrites comme suit :

```
( < selecteur >   < list_de_variables>   < expression_logique >)
```

avec pour sélecteurs :

- *les* pour toutes les solutions;
- *un* pour la première solution ;
- *le* pour la solution unique (sinon résultat nul)).

Une requête de consultation est alors décrite par :

```
( fedit   < partie_déclaration >
          < expression_de_sélection > )
```

2. Les mises à jour.

Les mises à jour permettent d'instancier, de modifier et d'effacer des faits dans la base de faits -ou base de données- qui est une instanciation de la base modèle -ou base de description-.

Les mises à jour élémentaires sont *ins_objet* , *modif_objet* , *sup_objet* , *ins_tuple* , *modif_tuple* , *sup_tuple*.

C'est lors des mises à jour que sont vérifiées les contraintes sur types qui sont définies dans le modèle.

Seules les contraintes locales sont vérifiées :

- vérification des domaines de valeurs
- vérification des clauses *respecte*
- vérification de l'unicité des clés
- vérification des contraintes de cardinalités

Toute violation de ces contraintes annule l'opération de mise à jour tentée et aucune modification n'altère la base de données.

Les mises à jour complexes sont décrites comme suit :

```
( fmaj  < partie_déclaration >
        ( pour  < sélection >
                < seq_ou_gran >   <mises_à_jour > ))
```

ou < seq_ou_gran > est soit *sequence* (plusieurs mises à jour sont traitées séquentiellement) soit *granule* (les contrôles sur les contraintes locales sont effectuées seulement à la fin du granule et le granule est accepté ou refusé dans sa totalité).

Les actions sont des mises à jour nommées avec précondition.

```
( action nom  < partie_déclaration >
              ( precondition  < expression_logique> )
              ( corps_action  < mises à jour > ))
```

Les mises à jour sont effectuées seulement si la précondition est respectée.

2.1.5 Le système de pilotage

Le pilotage d'un projet consiste à décrire une méthode de travail et une méthode de gestion de projet et à faire coincider le déroulement du projet à cette description. Ecrire une méthode de travail et de gestion sur Concerto revient à écrire en F1 un ensemble de lois et d'actions que le pilote utilise pour décrire le comportement attendu du projet, les manipulations permises et celles qui ne le sont pas. Cette description de méthodes en F1 doit être manipulable par le pilote grâce à l'éditeur structurel et aux autres outils générique du poste de travail. Le projet Concerto ne fait pas, à priori, de choix quant à ces méthodes. Cependant, pour valider les options choisies dans le projet dans le cadre d'une méthode particulière, une implantion de la méthode Prysme [MOT86] a été réalisée pour instancier le système de pilotage.

L'étape intermédiaire, consistant en l'écriture et la manipulation d'un modèle arborescent par le pilote, a été d'abord entreprise indépendamment du modèle Prysme lui-même. Cette étape a abouti au système Patre [ALL86] qui réalise l'interface entre une représentation externe du système de pilotage arborescente (la RI du poste de travail) et une représentation interne du système d'information basée sur le modèle entité-relation avec une déduction logique. Patre permet donc l'échange d'informations entre le système de pilotage et le système d'information ainsi que l'utilisation par le pilote de déductions effectuées par F1.

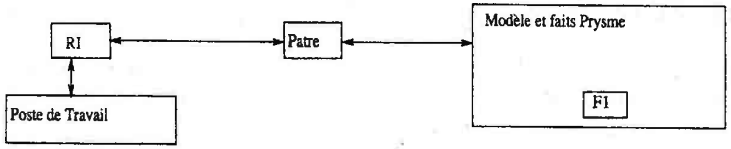


Figure 2.3. Architecture du système de pilotage

L'architecture du système de pilotage peut être résumée par la figure 2.3.

Prysme est une méthode de planification et de suivi de projet basée sur cinq activités qui sont la structuration, l'estimation, la planification, le suivi et le bilan. Prysme ne définit pas cependant de méthode de développement. D'autre part, Prysme manipule dans chacune des activités précitées des objets dont la structure est arborescente.

La définition dans Prysme de la méthode de développement désirée se fait par la description des différentes phases -ou tâches- d'un projet. Ces phases sont alors structurées en un arbre de phases, chaque phase étant décomposée en sous-phases. Le modèle de la dynamique de ces tâches est ensuite représenté grâce à un graphe de précédence sur l'ensemble des tâches.

La structuration du projet résulte de l'instanciation de la méthode de développement décrite par l'arbre du produit que l'on se propose de développer.

L'estimation et le calcul du planning se font par une décoration des feuilles (phases terminales) de l'arbre des phases et par l'exploitation de cette décoration des feuilles pour déduire des informations au niveau de n'importe quel sous-arbre de Prysme (phases non terminales).

Le suivi se fait en mesurant la part de travail non commencé, en cours, terminé par rapport au travail total à effectuer. Cette mesure se fait de la même manière que l'estimation en exploitant les informations attachées à chaque feuille.

Il en est de même pour l'établissement d'un bilan par la méthode Prysme.

2.2 Concerto aujourd'hui

L'étude et la réalisation du projet Concerto se terminant fin 1985, les différentes équipes qui ont développé Concerto se sont séparées et chacune d'entre elles a continué pour son propre compte le développement d'un des aspects de Concerto.

En ce qui concerne le système d'information et son formalisme F1, celui-ci a gardé les mêmes spécifications mais ses performances ont été multipliées -il s'appelle maintenant Turbo-F1- et il a complétement été réécrit en le_lisp.

Les environnements de programmation originels sont dans le même état qu'en 1985, par contre le poste de travail a été entiérement instancié avec un certain nombre d'outils pour le langage le_lisp.

Une part importante a, par ailleurs, été réservée à l'environnement de documentation.

Une architecture fondamentalement différente est apparue, et l'environnement Concerto tend de plus en plus vers un poste de travail intelligent sur une architecture décentralisée. Le système d'information et le système de pilotage centralisé ont disparu : l'environnement Concerto devient un réseau de postes de travail communiquant par l'intermédiaire de l'environnement ISA. Chacun des postes peut alors être utilisé indifféremment comme poste de production -développement de programmes sous le_lisp ou production de documents-, comme un poste de communication -en développant une requête sous l'environnement ISA- ou enfin comme un poste de décision -en décidant de soumettre une tâche par l'intermédiaire d'ISA-. Cette architecture peut être résumée dans la figure figure 2.4.

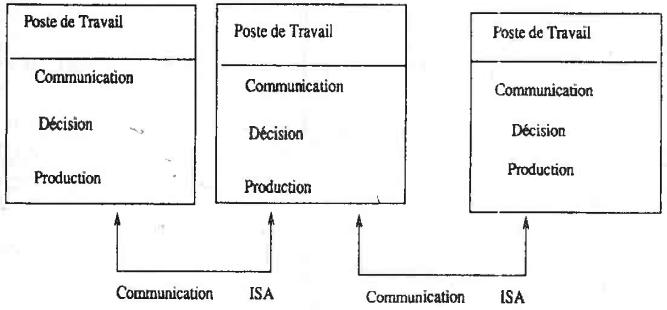


Figure 2.4. Architecture actuelle de Concerto

2.3 Conclusion

En conclusion sur le poste de travail Concerto, les qualités qu'il présente au niveau de son interface utilisateur, unique et générique, sont réellement très intéressantes et toute l'édition structurelle associée paraît être la voie du futur, voie ouverte par [DON75]. Des

études [COL89] [CRU88] sont faites sur de tels environnements dirigés par la syntaxe. Par ailleurs le Projet GIPE (Generation of Interactive Programming Environments) [GIP87] construit toute sa démarche autour de la syntaxe, exprimée en Centaur, des langages utilisés.

L'environnement de communication, quant à lui, semble l'élément essentiel d'intégration de l'atelier.

En ce qui concerne le système d'information et le système de pilotage, nous reviendrons par la suite sur certains des choix de Concerto notamment en ce qui concerne le formalisme F1, son applicabilité à un projet réel et son utilité pour la description d'un système de pilotage.

Une évaluation qualitative de Concerto peut se faire à travers la vérification de la satisfaction des propriétés attendues d'un AIGL présentées dans le § 1.2.1 du premier chapitre de la partie I par Concerto.

La structuration de la collection d'outils est réalisée de manière satisfaisante du fait que les outils sont tous sensés utiliser la même représentation interne des objets manipulés. Cependant cette RI est souvent seulement un moyen de communication entre outils, chacun des outils d'une classe travaillant sur sa représentation propre. F1 par exemple n'est pas du tout concerné par cette RI, Patre devant prendre le relais entre cette RI et la représentation de F1.

La complétude est réalisable assez aisément, d'après l'exemple d'implantation de Prysme sur Concerto.

La cohérence est assurée par l'existence de cette représentation interne unique et donc repose sur l'accès des outils à cette RI.

La conservation de l'information est entièrement laissée à la charge de l'utilisateur ou du chef de projet qui doit répercuter entièrement les effets des opérations de l'atelier sur les entités et relations conservées dans le système d'information, comme dans SKB ou PMDB.

La structuration des données n'est pas réalisée, ce sont leurs images dans le système d'information qui sont structurées.

Par contre l'extensibilité est une propriété forte de l'atelier Concerto. Il est "facile" de rajouter un outil qui s'intégrera parfaitement dans l'atelier. Il "suffit" pour cela de réaliser l'interface de cet outil en utilisant Metal et PPML.

La distribution est également un point fort de Concerto grâce au système ISA.

La portabilité n'est pas un atout majeur de Concerto -version fin 1985-. En effet, celui-ci est en partie écrit en Ulysse qui est lui-même écrit en le_lisp, donc ne serait portable que dans la mesure ou le_lisp le serait (avec l'environnement le_lisp de programmation objet

qu'est Alcyone [HUL85]), or le_lisp n'est pas un langage de programmation particulièrement portable. En outre F1 est écrit en Prolisp ! Le problème spécifique de portabilité de F1 est en voie de résolution vu que Turbo-F1 est entièrement réécrit en le_lisp. Tous les éléments de Concerto seront alors au même faible niveau de portabilité. Enfin, en ce qui concerne l'interface visuelle, un changement de machine entraîne la modification de la partie afficheur mais uniquement celle-ci car le "dialogueur", lui, est indépendant du matériel (dépend d'une machine virtuelle).

Concluons en rappelant que Concerto est un prototype d'atelier très intéressant par les concepts qu'il met en œuvre mais que, pour une exploitation industrielle et pour le passage de l'état de prototype à celui de produit, de nombreux efforts restent encore à fournir. La Sema-Metra s'intéresse à l'industrialisation de ce produit et a fait un certain nombre de choix restrictifs dans les concepts à garder et entreprend de développer autour de ceux-ci des outils performants.

L'option Sema-Metra tend vers la réalisation d'un poste de travail performant en abandonnant la partie système d'information. Par contre le Cert s'intéresse, lui, tout particulièrement à ce point.

3
Les Structures d'accueil actuelles

3.1 Introduction

Ce chapitre présente les deux structures d'accueil existantes. Les structures d'accueil, comme nous l'avons présenté dans le chapitre 1 de la première partie, représentent le choix le plus économique à long terme pour le développement d'AIGLs, et la garantie d'une intégration réelle. Quelques problèmes d'incohérence dûs au simple couplage SGBD/Système d'exploitation sont peut-être levés par la deuxième approche développant directement un AIGL, cependant celle-ci nécessite un investissement important et cet investissement est difficilement réutilisable. Développer un AIGL nécessite de choisir le domaine d'application concerné par cet AIGL et un certain nombre d'outils. Changer de domaine d'application entraîne presque automatiquement l'abandon de cet atelier pour un autre. Le développement d'un AIGL coûte moins cher que le développement d'une structure d'accueil mais celui-ci coûte cependant moins cher que le développement de plusieurs AIGLs.

Le développement d'une structure d'accueil consiste donc à caractériser les fonctionnalités communes à tout atelier de développement de logiciels et à fournir un ensemble de mécanismes minimaux implantant ces caractéristiques communes. Ces caractéristiques étant implantées une fois pour toutes, tout développeur d'atelier utilise ces mécanismes de base pour écrire ses outils de développement et de gestion.

L'élément de base de cette notion de structure d'accueil est le système de gestion d'objets. La notion d'objet complexe manipulée lors du développement de logiciel est fondamentale. La richesse et les facilités proposées pour la gestion et la manipulation des objets informatiques sera l'un des critères essentiels d'évaluation des structures d'accueil.

Nous étudierons dans ce chapitre deux "normes" de structures d'accueil, à savoir l'européenne PCTE et l'américaine CAIS.

3.2 PCTE

3.2.1 Introduction

PCTE [GAL86b] [PCT86] est une spécification de structure d'accueil conçue pour servir de base à la construction d'Ateliers Intégrés de Génie Logiciel.

PCTE introduit une séparation nette entre la structure d'accueil elle-même et les outils construits autour de cette structure. La construction de tels outils est possible grâce à la définition par PCTE des différentes primitives systèmes que peuvent invoquer les outils. PCTE est défini par la spécification d'un ensemble d'interfaces décrivant des primitives systèmes et des mécanismes considérés comme indispensables à une structure d'accueil. Ces primitives se scindent en trois groupes :

- les primitives des mécanismes de base
- les primitives de l'interface utilisateur
- les primitives de distribution.

Les mécanismes de base sont les différentes fonctionnalités de manipulation des différents objets intervenant dans un environnement de développement de logiciels. Ces objets sont essentiellement les objets exécutables (outils, programmes chargés...) et les autres (textes, objets physiques de type terminal par exemple...).

Les primitives de l'interface utilisateur fournissent l'ensemble des fonctionnalités d'aide au développement de l'interface que présentera un outil à l'utilisateur final. Ces aides concernent, par exemple, la forme du dialogue ou encore une forme d'interaction graphique.

Et enfin, les primitives de distributions permettront aux concepteurs d'AIGLs distribués, autour de la structure d'accueil PCTE, de garantir à l'utilisateur final la transparence d'une telle distribution.

3.2.2 Les mécanismes de base

Les mécanismes de bases sont divisés en cinq catégories :

- Les mécanismes d'exécution
- Les mécanismes de communication
- Les mécanismes de communication inter processus
- Le système de gestion des objets
- Les activités

Mécanismes d'exécution

Ils concernent les objets actifs c'est-à-dire des programmes exécutables et des processus. Ces mécanismes permettent l'activation et le contrôle d'un processus s'exécutant dans un contexte particulier. Il s'agit là de notions que l'on retrouve dans la plupart des systèmes d'exploitation (UNIX en particulier) car il ne faut pas perdre de vue qu'un AIGL, construit autour d'une structure d'accueil ou pas, est avant tout un système opérant.

PCTE définit un programme comme la description d'une activité potentielle et un processus comme, et c'est la définition classique [CRO75] [KRA85], l'exécution d'une telle activité. Les informations caractérisant un programme indépendamment de son exécution sont ce que PCTE définit comme le contexte statique. Tout processus s'exécute alors dans ce contexte.

Le contexte statique L'élément principal du contexte statique d'un programme est l'indication de la manière dont ce programme devra être exécuté : si c'est un programme interprétable, quel interpréteur devra être utilisé et sinon dans quelle classe devra être exécuté le code compilé du programme. La classe d'exécution est un objet caractérisant le type de processeur à utiliser et les sites sur lesquels un tel processeur est présent.

Par ailleurs, le contexte statique détermine les différents droits du processus lors de son exécution.

Le contexte dynamique d'un programme et le contrôle de l'exécution Un programme est exécuté par un processus qui a un certain nombre de propriétés. Parmi celles-ci, nous pouvons noter toutes les variables d'identification du propriétaire de ce processus, le contexte statique associé, ou encore l'ensemble des fichiers ouverts par le processus (propriétés similaires à celles d'Unix). Par contre, des caractéristiques spécifiques à PCTE telles que le schéma de travail à travers lequel le processus voit la base d'objets (cf. § 3.2.2) font de ce contexte dynamique un ensemble complet et cohérent d'informations.

Le contrôle de l'exécution d'un programme se fait par l'intermédiaire de primitives de lancement d'exécutions sous différents modes (lancement avec suspension de l'appelant, lancement en parallèle avec l'appelant), de suspension et de reprise d'exécutions et enfin par la terminaison d'exécutions.

Nous ne nous étendrons pas sur les aspects qui sont largement utilisés dans les systèmes d'exploitation classiques, par contre nous reviendrons dans la § 3.2.2 sur le schéma de travail d'un processus.

Les contrôles d'accès Ils sont dérivés de ceux d'Unix : à chaque objet PCTE sont associés des droits d'accès définissant les opérations permises au propriétaire de l'objet, aux membres de son groupe et enfin à tous les autres utilisateurs du système. Un processus ne peut entreprendre une opération sur un objet que si le propriétaire du processus en a le droit. Une différence essentielle avec les contrôles d'accès d'Unix est la possibilité de poser des verrous ("locks") lors "d'activités". Des contrôles d'accès plus complets sont développés dans PCTE+ [PCT88] [THO88b] avec les notions de "Mandatory access control" et "Discretionary access control".

Il existe également des primitives de manipulation de l'image d'un processus permettant de contrôler de manière fine son exécution.

Mécanismes de communication

PCTE fournit des mécanismes de **communication entre un programme et les objets extérieurs** non structurés de type fichier (le contenu des objets) et des mécanismes de **communication interprocessus**. Les fonctions du premier type permettent d'agir sur le contenu d'un objet, de l'ouvrir en lecture, en écriture. Ces fonctions peuvent être bloquantes ou pas pour le programme appelant. Les objets concernés par le premier type de fonctions sont soit des fichiers, soit des périphériques.

Les **mécanismes de communications interprocessus** permettent la coopération entre des processus d'une même famille. Ils peuvent être perçus comme des mécanismes d'exécution. Ils sont généraux et englobent les mécanismes d'Unix (pipe,...). Ils doivent permettre, de manière efficace et en supportant éventuellement la distribution des processus, une coopération entre processus en termes de synchronisation et d'échange d'informations. Ces mécanismes sont dérivés de ceux d'Unix.

PCTE permet donc de créer des tubes -pipes- comme objets nommés du Système de Gestion d'Objets ou anonymes qui vont permettre la communication, l'émission de signaux, l'envoi et la gestion de messages.

Les tubes permettent, comme dans Unix, la coopération entre des processus qui n'ont pas été spécifiquement écrits pour communiquer entre eux. Une autre possibilité pour la communication inter-processus tirée d'Unix est l'envoi de signaux à un processus qui y répond par l'activation d'une procédure attachée à ce signal.

Et enfin, les queues de messages sont la pierre angulaire de tout ce système de communication inter-processus. Ces queues sont des objets manipulables par l'intermédiaire du système de gestion d'objets de PCTE. Les messages sont typés et l'envoi ou la réception de messages peuvent être bloquant ou pas. L'accès à un message dans une queue de messages peut se faire en fonction du type ou d'un autre champ du message. La particularité des queues de messages PCTE par rapport à celles d'Unix est le fait que ce sont des objets

PCTE, donc désignables par leur nom et donc utilisables par n'importe quel processus via le système de gestion d'objets de PCTE.

Le Système de gestion d'objets

La base de tout environnement étant la manipulation des objets de l'environnement, la base de PCTE est son Système de Gestion d'Objets (SGO) [GAL86b]. Un objet est une entité sur laquelle peuvent travailler les utilisateurs et les programmes.

Ce système de gestion d'objets va se substituer au système de gestion de fichiers (SGF) traditionnel d'un système d'exploitation en ayant les fonctionnalités du dit SGF et en y ajoutant de nouvelles.

Le modèle sous-jacent au SGO de PCTE est dérivé du modèle Entité-Association de Chen [CHE76]. Le modèle permet donc la définition de types d'objets (ou entité pour le modèle E-A), de types de liens entre objets (qui peuvent représenter l'association du modèle E-A) et de types d'attributs.

Un schéma est un ensemble complet de définitions de types d'objets, de liens et d'attributs, et de relations entre ces types. Un schéma (SDS)[1] définit une vue sur la base d'objets.

Un schéma de travail (WS)[2] est défini en fusionnant un ou plusieurs SDS. (cf § 3.2.2)

Objets et types d'objets Les objets sont les entités identifiables de l'environnement. Les objets sont typés, et donc caractérisés par ce type. Un type d'objet est défini par des attributs (propriétés "internes" de l'objet), un ensemble de liens sources et destinations (propriétés "globales" de l'objet vis-à-vis des autres objets), et éventuellement un contenu.

Le typage des objets est organisé hiérarchiquement. Un type d'objet est défini comme fils d'un autre et hérite de toutes les propriétés du père. Tout type d'objet est de ce fait un descendant du type racine OBJECT qui est caractérisé par l'ensemble des propriétés communes à tous les objets. Un type est donc soit le type racine OBJECT lui-même, soit un de ses descendants.

Un type d'objet est défini par :

- son nom
- le nom du SDS où il est défini
- le type du père
- un ensemble de types d'attributs

1. Schema Definition Set
2. Working Schema

- un ensemble de types de liens pouvant partir de ce type d'objet
- un ensemble de types de liens pouvant arriver à ce type d'objet.

Attributs et types d'attributs Un type d'attribut est défini par son nom, un type de valeur (entier, chaîne de caractères,...), et une valeur initiale.

Associations et types d'associations La définition des associations et types d'associations se fait grâce aux liens unidirectionnels définis par PCTE.

1. Liens et types de liens.

   Un lien est une association unidirectionnelle entre un objet origine et un objet destination. Ces liens sont typés et un type de liens est défini par un nom, une cardinalité, une catégorie, une propriété de stabilité, un ensemble de types d'objets sources, un ensemble de types d'objets destinations, un ou plusieurs attributs clés pour les liens n-aires, un ensemble d'attributs.

   La **cardinalité** d'un lien dépend des attributs clés. Si un type de lien a des attributs clés, il a une cardinalité de *plusieurs* (many), sinon la cardinalité est *un*.

   Un objet pointé par un lien ayant la propriété de **stabilité** ne peut être mis à jour.

   Les liens sont de trois catégories, ils sont de **composition**, de **référence** ou **inverses**.

   Définition et usage des liens de composition : La création d'un objet nécessite forcément la création d'un lien de composition partant d'un objet source. L'objet destination ainsi créé est alors appelé composant de l'objet source. Par ailleurs, un lien de composition peut être appliqué à un objet pré-existant. Ces liens de composition ont pour rôle de définir au moins un chemin d'accès à n'importe quel objet de la base (ils supportent la propriété d'existence des objets) : un ensemble d'objets ne pourra jamais être déconnecté de la base d'objets. Ils sont à la base de la gestion des objets par le SGO de PCTE. Un objet n'existe que par le dernier lien de composition qui y mène : la destruction effective de ce lien (autorisé sous certaines conditions : droits sur le lien, aucun lien de composition ne partant de l'objet destination, aucun lien de référence sur cet objet...) est équivalente à la destruction de l'objet. Ces contraintes permettent au SGO d'assurer l'accès à n'importe quel objet à partir de la racine.

   Définition et usage des liens de référence : Les liens de référence permettent de référencer un objet existant, mais peuvent être détruits sans effet immédiat sur l'objet destination. Cependant l'existence d'un lien de référence sur un objet

empêche la destruction du dernier lien de composition y conduisant et donc la destruction de l'objet lui-même. Ces liens permettent, en outre, d'améliorer les temps d'accès à un objet.

Définition et usage des liens inverses : Enfin, on peut associer aux liens un lien inverse implicite qui est créé et détruit en même temps que le lien explicite (de composition ou de référence) auquel on l'associe.

La définition d'un ensemble de types **d'objets sources et destinations** se fait en référençant différents types d'objets définis par ailleurs et qui constituent l'ensemble des sources et des destinations possibles d'un lien de tel type.

2. Associations et types d'associations.

   Une association est alors définie comme étant une paire de liens inverses (source de l'un, destination de l'autre et vice versa). Un type d'association consiste donc en la définition de deux types de liens inverses. Ces associations ont les mêmes propriétés que les autres liens.

Définitions de schémas et schémas de travail

1. Schémas.

   Ce qu'on appelle, dans PCTE, un schéma est un ensemble de définitions de types d'objets, de types d'attributs, de types de liens et de types d'associations. Les schémas vont permettre de structurer les différentes définitions entre elles et de s'assurer de la complétude de ces définitions. En effet chaque schéma de définition, appelé SDS (Schema Definition Set), doit être complet et cohérent et est identifié par un nom. PCTE définit et utilise des schémas E-A typés.

   Une définition de type est identifiée de manière unique pour toute la base d'objets par un numéro de référence.

   Un type peut être utilisé dans plusieurs SDS, à condition d'être importé de son SDS d'origine, dans le nouveau SDS, ceci pour permettre la correspondance entre les numéros de référence. L'importation ne concerne que la déclaration du type et non sa définition complète. Ainsi, lorsqu'un type d'objet est importé, les attributs et les liens qui le caractérisent ne sont pas importés. Seul son ancêtre dans la hiérarchie des types est importé implicitement. En ce qui concerne l'importation des liens, seuls les attributs clés sont importés implicitement.

   Un SDS est un ensemble structuré de définitions évoluant dans le temps. A tout moment de la vie d'un SDS, une définition présente peut être étendue ou une nouvelle définition peut être créée. Cependant cette évolution est soumise à certaines

contraintes : en règle générale cette évolution doit demeurer interne au SDS en question et ne pas affecter les autres SDS. En particulier, de nouvelles définitions de types ne sont autorisées qu'à condition que le nouveau type soit sous-type d'un type local; on ne peut appliquer en extension d'un SDS que des liens ou attributs locaux à des objets locaux. Seuls les ensembles des objets destinations des liens locaux peuvent être étendus, et de plus uniquement avec des types d'objets locaux.

Nous avons vu comment créer et étendre un ensemble de définitions. Par ailleurs, tout utilisateur sera amené à définir et utiliser un certain nombre de définitions de base telles que fichier, device, etc... Toutes ces définitions sont donc prédéfinies et regroupées dans un SDS prédéfini appelé **sys** (system). Ces définitions concernent, en particulier, des types d'attributs de contrôle d'accès (owner, group, others, mode...) et des types d'attributs utiles au gestionnaire des objets tels que le nombre de liens arrivant ou partant d'un objet (utile pour la suppression d'un objet si son dernier lien est supprimé), la taille d'un objet, la date des différentes manipulations sur cet objet, etc...

Ces SDS vont permettre de garantir la cohérence des opérations de manipulation des objets par rapport aux typages des différents composants du schéma. Ils permettront d'effectuer l'initialisation des attributs d'un lien ou d'un objet à la création, de garantir l'intégrité des données stockées, et de décrire l'organisation de celles-ci.

2. Schémas de travail.

Ils vont permettre d'adapter la base d'objets à différents modèles d'organisation. Chaque utilisateur pourra avoir une vue propre correspondant à ses besoins. Les schémas de travail vont également permettre de développer des outils dans des contextes particuliers.

Un schéma de travail, ou vue de la base d'objets, est nécessaire à tout processus dans PCTE. Cette vue est obtenue grâce à la composition de plusieurs SDS dans une liste qui définit le schéma de travail, appelé WS (Working Schema).

Les définitions d'un WS sont alors constituées de l'union des définitions des différents SDS de la liste. Les types de liens appliqués à un objet du WS sont l'union de tous les types de liens appliqués à cet objet dans chacun des SDS. Les types d'objets destinations d'un lien sont eux aussi l'union de tous les types d'objets destinations dans chacun des SDS. Il en est de même pour les attributs d'un type d'objet ou d'un type de lien. Tout conflit entre les différentes définitions des SDS est résolu en accordant une propriété décroissante du premier au dernier élément de la liste des SDS définissant le WS.

Ce WS doit, comme tout SDS, être complet et cohérent et se suffire à lui-même. Pour être certain d'avoir à sa disposition les définitions minima, le SDS **sys** est

implicitement considéré comme dernier élément de toute liste de SDS définissant un WS.

Tout processus possède donc un WS qui fait partie de son contexte statique. Par défaut ce WS est celui hérité du processus père, mais le processus peut dynamiquement changer pour un autre WS.

Manipulation des objets du SGO PCTE fournit des primitives de manipulation des types d'objets et de leurs instances qui sont :

- Création ou destruction d'un SDS
  - Création ou destruction d'un type d'objet, de lien ou d'attribut.
  - Importation d'un type d'objet, de lien ou d'attribut.
  - Extension d'un type par les types qu'on peut lui appliquer.
  - Restriction des types qu'on peut appliquer à un autre type.
- Sélection d'un WS
  - Manipulation des types définis dans le WS
  - Création, destruction et manipulation des instances de tous ces types.

Activités

PCTE, devant fournir une structure d'accueil pour un environnement multi-utilisateurs, doit donc gérer les problèmes d'accès et de concurrence d'accès aux objets de la base. En effet il est hors de question d'avoir un accès aux objets du type de celui d'Unix, qui autorise plusieurs accès parallèles à une même version de fichier sans assurer la cohérence du fichier en fonction des différentes opérations réalisées par les différents utilisateurs.

Le contrôle d'accès concurrents, dans PCTE, propose de garantir la cohérence d'un objet par rapport aux différentes opérations parallèles qu'on lui applique.

Il y a trois types d'activité :

1. Les activités non protégées (unprotected activities) pour lesquelles aucun contrôle n'est fait.
2. Les activités protégées, où le contrôle de la concurrence d'accès est assuré.
3. Les transactions : une transaction est une série d'opérations considérée comme atomique : cette série est soit entièrement réalisée, soit n'a aucun effet sur la base. En outre, plusieurs transactions activées en parallèle doivent avoir le même résultat que si elles avaient été exécutées en exclusion mutuelle.

La propriété de cohérence des activités est réalisée grâce à un mécanisme de verrous sur les objets. Ceux-là ont différents niveaux et permettent l'interdiction de lecture ou d'écriture de tout objet de la base. Ces poses de verrous pouvant conduire à des situations d'interblocage, PCTE contrôle et résoud de telles situations. La propriété d'atomicité des transactions est réalisée par l'intermédiaire de trois primitives (startact, endact et abortact) qui travaillent sur une duplication des objets manipulés, les objets eux-mêmes n'étant effectivement modifiés qu'à la fin de la transaction (endact).

Distribution

PCTE est prévu pour fonctionner sur un réseau de stations de travail compatibles. Il propose donc un modèle pour la gestion de la distribution. La distribution sur PCTE comprend la distribution de tous les "concepts" manipulés par PCTE. PCTE gère donc la distribution de l'exécution de processus, la distribution des objets, des activités et des communications inter-processus.

La distribution de l'exécution se fait de deux manières. L'utilisateur peut ignorer une telle distribution et demander une "exécution neutre" (non attribuée à un site particulier) d'un processus. Le système de gestion de la distribution choisira, alors, le site d'exécution en fonction des sites sur lesquels sont ouverts les fichiers utilisés, des sites contenant les objets manipulés et des sites où s'exécute des processus avec lesquels le processus devra communiquer.

L'autre façon de gérer la distribution de l'exécution pour l'utilisateur est de désigner explicitement le site d'exécution désiré. Cela permet, par exemple, de spécialiser les stations de travail, comme c'est fait dans Concerto, pour le traitement de la documentation, pour les compilations, ou pour la communication, etc...

La distribution des objets se fait en créant et gérant les objets sur des volumes spécifiques (disques logiques) montés sur telle ou telle station de travail. Cette distribution ne doit cependant pas handicaper l'utilisateur dans son utilisation de la base d'objets. Il peut désirer travailler sans avoir à gérer cette distribution et surtout même si le réseau local supportant cette distribution est partiellement défaillant. PCTE fournit un mécanisme de duplication d'objets permettant à un utilisateur de travailler sur sa station de travail sans se préoccuper du reste du réseau. Le système de distribution se charge de maintenir cohérentes entre elles les différentes copies d'un objet.

L'interface utilisateur

L'interface utilisateur de PCTE fournit à l'utilisateur un ensemble de mécanismes de base afin que celui-ci puisse communiquer avec différentes applications s'exécutant en parallèle et réalisant toutes des entrées/sorties. Ces mécanismes vont permettre d'unifier la

communication entre l'utilisateur et les applications. Ils vont permettre la définition de terminaux virtuels, la gestion de fenêtres (iconisation, déplacement, activation...) et les opérations d'édition de base (désignation d'un élément, activation d'un processus d'édition...). Cet ensemble de fonctionnalités sera rempli par **l'agent utilisateur** qui servira d'interface entre les applications (ou outils) et les organes d'entrée/sortie. Cet agent utilisateur a pour rôle de permettre la séparation de l'écriture des outils et la gestion des entrées/sorties sur une station de travail spécifique. Il offre en outre la possibilité d'utiliser des outils préexistants en créant une fenêtre émulant un terminal standard. L'interface PCTE débarrasse le programmeur de tout l'aspect externe de son application. Le programmeur réalise ses entrées/sorties en utilisant les primitives de base fournies par l'agent utilisateur.

Ces primitives de base permettent la manipulation de :

- **Cadre** (Frame) : entité virtuelle structurée (point par point, graphique ou textuelle) d'affichage.
- **Fenêtre** : zone sur l'écran dans laquelle tout ou partie d'un cadre peut être visualisé (la partie affichable est décrite dans un "viewport"[port de visualisation?]), différentes fenêtres peuvent être visibles simultanément, se recouvrir partiellement.
- **Icones, polices de caractères, curseur, mécanisme de désignation**...sont aussi gérés par l'agent utilisateur.
- **Application** : Processus s'exécutant.
- **Application courante** : application recevant une entrée de la part de l'utilisateur.

L'agent utilisateur va donc servir d'interface entre les applications et les différents périphériques (souris, bitmap, clavier). Il gérera :

- les fenêtres : visualisation, mise à jour, recouvrement;
- les entrées : souris, clavier;
- les commandes : interprétation et échange des commandes systèmes, distribution des messages, vérification de la syntaxe des commandes.
- les menus : visualisation et transformation de la commande choisie en commande compréhensible pour l'application;
- gestion des sélections;
- gestion des commandes génériques;

3.2.3 Conclusion

PCTE fournit une proposition de norme assez complète de structure d'accueil pour construire un Atelier Intégré de Génie Logiciel (Nous entendons par norme une interface normalisée). Cependant des lacunes existent; certaines d'entre elles sont en train d'être résolues dans les développements actuels de la norme PCTE+ [PCT88] [THO88b] et PCTE fourth year, alors que d'autres sont résolues dans des structures d'accueil comme CAIS. Contrairement à une structure comme CAIS (cf. § 3.4), les processus ne sont pas matérialisés par des objets. Cette option aurait uniformisé le modèle. D'autre part, le langage de manipulation de données PCTE est assez restreint et est loin d'offrir le confort d'utilisation de celui des SGBD classiques. La consultation de la base d'objet est limitée à un parcours des liens (navigation) et à une exploration simple des différents objets (accès aux liens et aux attributs d'un objet particulier). Un langage évolué devra être disponible, pour permettre une manipulation et donc une gestion efficace de la base d'objets.

Les interfaces PCTE devant évoluer pour répondre aux critiques et suggestions faites par les utilisateurs, de nombreux projets ont vu le jour. Le projet Sfinx a deux objectifs principaux : collecter les requêtes de modifications émanant des autres projets Esprit et intégrer les résultats de ces projets dans un IPSE basé sur PCTE. Le projet VIP (VDM Interfaces for PCTE) produit une spécification formelle des interfaces PCTE pour éviter les ambiguïtés et les omissions existant dans une spécification en langage naturel.

Le "Independent European Procurement Group" de l'OTAN a créé le projet PCTE+ pour étudier des évolutions concernant plus particulièrement la sécurité, l'intégrité, l'indépendance vis-à-vis d'Unix et une spécification formelle.

Le but du projet ALF (sur lequel nous reviendrons plus spécialement) est d'améliorer les interfaces PCTE, en vue de permettre l'implantation de l'assistance et de la prise d'initiative dans les IPSE.

Toutes ces évolutions ou améliorations sont sous le contrôle du "PCTE Interface Management Board (PIMB)", qui regroupe des représentants des concepteurs initiaux et des différents utilisateurs. Le PIMB s'appuie pour les problèmes techniques sur le "PCTE Interface Control Group (PICG)".

Quelques implantations de cette norme commencent à voir le jour, l'implantation la plus mûre étant Emeraude [EME84] développée par le GIE Emeraude. Le résultat actuel est le produit Emeraude tournant sur Bull DPX2000. Cette implantation Emeraude des interfaces PCTE est en cours de portage sur d'autres machines Unix (Sun 3, DEC Vax, IBM PC/AT, HP9300, stations de travail Appolo) dans le cadre du projet ESPRIT Sapphire. Le projet PAVE (PCTE And VMS Environment) quant à lui cherche à permettre l'utilisation des interfaces PCTE -et donc des outils développés sur PCTE- sur des environnements VAX/VMS. Le projet Aphrodite est un autre projet Esprit dont le but est

de définir un environnement distribué comprenant à la fois la machine hôte et la machine cible.

Cependant PCTE n'est que le noyau sur lequel on peut construire des environnements. Il est loin d'être outillé en compilateurs, éditeurs, etc... Que serait Unix sans cc, dbx, lex, yacc,etc...? Le projet PACT [PAC86] a été défini dans ce sens (cf §3.3).

PACT (PCTE Added Common Tools) est un projet Esprit construisant un premier ensemble d'outils intégrés et de services communs (gestion de versions, gestion de dialogues, manipulation de données,...). Les résultats du projet seront disponibles fin 1989.

Eclipse est un projet Alvey réalisant un IPSE sur PCTE. Entreprise 2 est un projet de la Délégation Générale de l'Armement conçu pour développer un environnement supportant le développement de logiciels en Ada, C et LTR3. Athmosphere est un projet Esprit et EAST est un projet Eureka dont le but est de produire des IPSEs spécifiques à un certain nombre de domaines d'application.

3.3 PACT

3.3.1 Introduction

Le but du projet PACT (Pcte Added Common Tools) [PAC86] [PAC88] [THO88] réalisé par le consortium PACT est de fournir un ensemble intégré et cohérent d'outils et de services communs (gestion de versions, gestion de dialogues, manipulation de données,...) qui, ajoutés à PCTE permettront la construction d'Ateliers Intégrés de Génie Logiciel. Les résultats du projet sont annoncés pour fin 1989.

L'ensemble des outils actuellement en développement dans PACT est composé de :

- un gestionnaire de dialogues
- un gestionnaire de données
- un gestionnaire de versions
- un gestionnaire de documents
- un gestionnaire de composition d'outils.

Ces outils vont être sommairement présentés dans le paragraphe suivant.

3.3.2 Les outils de base de PACT

Le gestionnaire de dialogues (DMCS pour Dialogue Management Common Service) fournit un support logiciel pour un style constant d'interfaces utilisateur. Cependant le DMCS n'étant pas complètement directif, et afin d'éviter une discordance entre interfaces

d'outils différents, le DMCS est complété par un guide ("méthode") d'utilisation. Ce guide décrit les aspects du style PACT, style sous-jacent et utilisé dans la communauté PACT, qui ne sont pas supportés par le DMCS.

Le gestionnaire de données (DQMS pour Data Query and Manipulation Common Service) fournit quelques primitives de base de données afin de manipuler la base d'objets PCTE. L'existence de ce gestionnaire de données est significative de la philosophie PCTE. En effet PCTE ne fournit que les mécanismes de base pour la gestion des objets, ces mécanismes étant -"one object at a time"- et n'incluant pas de primitives de manipulation d'un ensemble d'objets -"one set at a time"- absolument nécessaire à la formulation d'une requête complexe. Cet outil montre en outre une des limites du "modèle E-A" de PCTE : peut-on parler d'un modèle E-A sans aucune primitive de manipulation ensembliste ? Le DQMCS fournit les résultats des requêtes sous forme de tables qui peuvent être utilisées comme moyen d'échange de données entre outils.

Le gestionnaire de versions (VMCS pour Version Management Common Service) fournit des fonctions de base pour la gestion de différentes versions d'un objet particulier ou d'un groupe d'objets liés entre eux. Il est à noter, par ailleurs, que certaines primitives pour la gestion de versions jugées primordiales ont été introduites dans la définition de PCTE+ [PCT88] [THO88b].

Le gestionnaire de documents (DSMCS pour Document Structure Management Common Service) fournit les fonctionnalités de base pour la création et la manipulation de documents structurés. Le DSMCS gère la représentation de documents dans la base d'objets et fournit ainsi le noyau de tous les outils qui créent ou utilisent des documents.

Le gestionnaire de composition d'outils (TCCS pour Tool Composition Common Service) permet la création d'outils complexes à partir de différents outils qui n'ont, à priori, pas été conçus pour être intégrés. Le TCCS autorise une plus grande liberté dans l'écriture d'outils.

3.3.3 Conclusion

Nous n'avons fait, dans cette présentation de PACT, que citer les différents outils de base de PACT, le but n'étant pas une présentation complète de ceux-ci mais juste de montrer qu'il est possible d'utiliser la structure d'accueil PCTE. Les outils décrits ne sont pas les seuls, mais présentent un large éventail de possibilités et serviront de base aux outils futurs.

3.4 CAIS

3.4.1 Introduction

Le projet CAIS (Common APSEs Interface Set) du Département de la Défense américain (DoD/USA) se propose de définir un ensemble d'interfaces communes aux environnements de programmation Ada (APSEs pour Ada Programming Support Environments). Le projet en est toujours, en 1989, à l'étape des spécifications. Les outils écrits en Ada, et utilisant cet ensemble d'interfaces, seront ainsi aisément, **transportables** d'un APSE à l'autre. Le projet CAIS a pour but d'arriver à un standard en matière de développement de programmes Ada. Dans le but de s'imposer à une large communauté d'utilisateurs d'Ada, CAIS se doit donc d'être simple, uniforme et flexible. CAIS a été spécifié dans [CAI85]. Cette spécification a été justifiée et commentée dans le "CAIS Rationale" [CAI85a] [CAI86]. Les critiques et nouveautés dans le modèle ont mené à la spécification de CAIS-A [CAI88].

L'ensemble des interfaces définies par CAIS se subdivise en quatre sous-ensembles d'interfaces concernant :

-le modèle nodal

-les processus

-les entrées/sorties

-les utilitaires.

Le modèle nodal définit le modèle sous-jacent de CAIS des nœuds, de leur contenu, de leurs relations et de leurs attributs (les objets manipulés par CAIS sont appelés nœuds par analogie avec les nœuds d'un graphe). Ce sous-ensemble contient également les interfaces pour le contrôle d'accès et la synchronisation des accès.

L'interface des processus se charge de l'invocation des programmes et du contrôle de leur exécution.

L'interface E/S s'occupe de la gestion des protocoles de communication interprocessus et de la gestion des différents types de terminaux.

L'interface utilitaire fournit, elle, les opérations de manipulation des paramètres d'un APSE et des valeurs d'attributs.

3.4.2 Le modèle nodal

Comme toute structure d'accueil -ou tout bon AIGL- une des tâches majeures de CAIS est la gestion des informations caractéristiques des données stockées. CAIS résoud cette tâche en fournissant un support pour une représentation commune des caractéristiques des

données et de leurs interconnexions. Cette représentation commune a pour rôle d'éviter toute incohérence dûe à une multiplicité des représentations. Le modèle conceptuel de représentation des informations de CAIS est un graphe orienté. Cela ne veut pas dire que le modèle sera implanté sous forme de graphe, le modèle de graphe étant le modèle de CAIS et non son implantation. Tous les objets manipulés par CAIS sont décrits par des **nœuds** liés par des **relations**, les nœuds et les relations possédant des **attributs** les décrivant.

Les nœuds

Les nœuds sont la représentation des informations sur les données de l'APSE (les fichiers). Cette représentation est centralisée dans le modèle nodal. Trois sortes de nœuds sont manipulées par CAIS :

- les nœuds fichier,
- les nœuds processus,
- les nœuds structure.

Un nœud peut avoir un contenu, des relations et des attributs. Le type de contenu d'un nœud est fonction du type de nœud. Les nœuds fichier ont pour contenu un fichier, les nœuds processus la représentation de l'exécution d'un programme Ada mais les nœuds structure n'ont pas de contenu. Les nœuds peuvent être créés, renommés, détruits et bien sûr accédés.

Les nœuds fichier Les nœuds fichier décrivent un fichier et ont donc pour contenu un fichier. Le type de ce fichier définit le type du nœud fichier. CAIS distingue 4 types de nœuds fichier prédéfinis :

- mémoire secondaire
- queue
- terminal
- cassette magnétique.

1. Les nœuds fichier de type terminal

   Les **nœuds fichier** de type **terminal** permettent de rendre transparente à l'utilisateur de l'APSE la nature physique réelle du terminal sur lequel s'effectue l'entrée/sortie. L'utilisateur manipulera uniquement un type de nœud et réalisera ses opérations

d'E/S en utilisant la description du canal d'E/S. Cependant CAIS a été amené à distinguer trois sous-types de terminaux en fonction des facilités offertes à l'utilisateur. Les **terminaux scroll** sont les plus simples, ils permettent à l'utilisateur de travailler séquentiellement sur le terminal. Une position adressée une première fois ne peut plus l'être à nouveau. Les **terminaux page** permettent l'adressage de tout point de l'écran à tout moment (éventuellement plusieurs fois). Les **terminaux format** permettent l'acquisition de données sous forme de grille, le contenu de cette grille étant local au terminal et la prise en compte de ces données par l'UC n'intervenant que sur demande de l'utilisateur.

2. Les nœuds fichier de type queue

   Les **nœuds fichier** de type **queue** permettent la communication interprocessus. Les queues sont gérées de manières différentes suivant leur type. Elles peuvent être :

   - simples gérées en FIFO (file d'attente analogue au pipe Unix) (**solo queue**)
   - couplées avec un nœud fichier (copy queue et mimic queue).

   Les **copy queues** ont le même contenu à la création que le nœud fichier correspondant. Elles permettent à différents processus de lire alternativement des données successives sur un même fichier de données. Le processus i traitant la donnée n et le processus j traitant à sa suite la donnée n+1, par exemple.

   Les **mimic queues** ont le même contenu à la création que le nœud fichier correspondant et les écritures effectuées sur ce type de queues sont répercutées sur le fichier associé. Elles permettent, entre autres, à différents utilisateurs d'écrire sur le même fichier sans gérer les accès concurrents à ce fichier. CAIS gérant l'accès et l'écriture de plusieurs processus sur la même queue gérera par la même occasion l'accès au fichier associé.

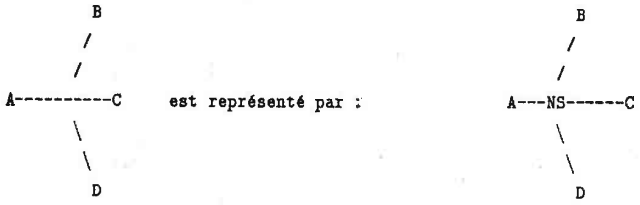
3. Les nœuds fichier de type bande magnétique

   Les **nœuds fichier** de type **bande magnétique** permettent le transfert de fichier entre différents APSEs.

4. Les nœuds fichier de type mémoire secondaire

Les nœuds processus Les nœuds processus représentent l'exécution d'un programme ADA. Le contenu d'un tel nœud est l'image mémoire de l'exécution d'un programme (image exécutable).

Les nœuds structure Les nœuds structure sont introduits dans un souci de clarté, mais ils ne sont pas indispensables. Un nœud fichier classique englobe les propriétés d'un tel nœud. Les nœuds structure n'ont pas de contenu et servent à l'implantation des relations n-aires. Un nœud structure joue le rôle des directories dans les systèmes hiérarchiques classiques (cf. Unix). Le besoin de ces nœuds structure est dû au fait que les relations implantées directement dans CAIS sont les relations binaires. La représentation d'une relation n-aires se fera donc par l'utilisation d'un nœud intermédiaire (nœud structure) et de n relations binaires. Les nœud structures servent donc uniquement à regrouper de manière physique des attributs et des relations binaires correspondant à une relation logique n-aire.



A, B, C et D étant des nœuds fichiers et NS un nœud structure.

Les relations

Les relations servent à représenter les liens existant entre les différentes informations stockées dans les nœuds CAIS (Typiquement, il y a un lien entre le nœud contenant un programme ADA et le nœud contenant le résultat de la compilation de ce programme code). Ces relations sont binaires et unidirectionnelles : elles partent d'un nœud source et arrivent sur un nœud destination. Comme les nœuds, les relations possèdent des attributs les caractérisant. Les relations du modèle nodal de CAIS sont de deux types :

- relations primaires
- relations secondaires

Les relations peuvent être "typées" par le biais d'un groupement des relations de même sémantique, et ce "typage" est soit prédéfini, soit défini par l'utilisateur. Les relations, instances d'un type de relation, sont identifiées par une clé.

Relations primaires L'ensemble de ces relations primaires forme un **arbre** dont la racine est un **nœud system** (system node). Cet arbre est construit et maintenu grâce aux caractéristiques suivantes : une relation primaire est créée à la création d'un nœud

entre le nœud courant et le nouveau nœud. Cette relation peut toutefois être "déplacée" de manière à émaner d'un autre nœud que le nœud courant. Les suppressions d'un nœud et de la relation primaire y aboutissant sont équivalentes. Cependant un nœud (ou la relation primaire y aboutissant) ne peut être supprimé que s'il n'en émane aucune relation primaire. Cette contrainte empêche la création d'arbres déconnectés de la racine (system node). Chaque nœud est référencé de manière absolue et peut être obtenu par une séquence unique de relations primaires émanant de la racine et y aboutissant.

Relations secondaires L'ensemble de ces relations secondaires forme un **graphe orienté**. Elles peuvent être créées et détruites sans contrôle de la part du système. Elles permettent d'accéder directement à tout nœud de l'arbre réalisé par les relations primaires à l'exception de la racine (system node) qui n'est pas accessible à l'utilisateur. La souplesse d'utilisation de ces relations secondaires -pas de restriction du système quant à leur création, leur destruction ou la destruction d'un objet désigné- a comme contre-partie la possibilité de garder un lien secondaire sur un objet n'existant plus dans la base.

Les attributs

Les relations et les nœuds possèdent des attributs décrivant leurs caractéristiques. Les attributs sont définis par leur nom. Ils sont multivalués, c'est-à-dire qu'à chaque attribut est affectée une liste de valeurs.

Les processus

CAIS entend par processus l'exécution d'un programme Ada (ou d'un autre langage dans la suite du projet CAIS), et de toutes les tâches associées à ce programme. Le contenu d'un nœud processus est l'image d'exécution du processus actif.

Le modèle des processus CAIS permet la hiérarchisation des processus d'une manière similaire à celle utilisée dans Unix : un processus père créant des sous-processus qui lui sont rattachés pour la réalisation de tâches particulières. Les fonctions de suspension, reprise et destruction d'un processus sont alors des fonctions appliquées à tout le sous arbre dont le processus visé est racine.

3.4.3 Les entrées/sorties

Les entrées/sorties de CAIS ont été réalisées en vue d'une compatibilité, aussi grande que possible, avec les entrées/sorties du langage Ada (En fait les entrées/sortie de CAIS sont celles d'Ada). Ces entrées/sorties doivent en outre permettre une gestion logique des entrées/sorties physiques standard (terminal, imprimante,...).

3.4.4 Les utilitaires

CAIS fournit un certain nombre de facilités de gestion de liste, ces facilités étant regroupées dans le "package utilities". Nous ne pensons pas que la présentation de ces facilités soit intéressante pour le lecteur, même si pour des raisons de structuration du chapitre nous avons gardé ce paragraphe.

3.4.5 La structure de travail

Après avoir présenté les éléments du modèle sous-jacent à CAIS, reste à en apprécier la structure globale : l'environnement CAIS est perçu comme un système hiérarchique de fichiers, ce système comprenant en outre des arbres de processus. L'environnement permet, grâce à un système unifié de nommage, de créer et de manipuler des arbres mixtes comprenant à la fois des fichiers et des processus, le but étant de faciliter la production d'outils composés de processus communiquant, créant et manipulant des fichiers et des répertoires.

3.4.6 Conclusion

De même que PCTE, CAIS fournit une proposition d'interfaces pour construire des ateliers de génie logiciel. Cependant CAIS n'est pas standardisé, alors que PCTE l'est.

Aucune implantation complète de CAIS n'est opérationnelle, même s'il existe quelques prototypes comme [CAR87].

3.5 Conclusion

Une des restrictions de PCTE, à savoir une représentation homogène des processus, est levée dans CAIS. Cependant, la lacune la plus importante de CAIS est la non-existence de schémas de travail préalables comme dans PCTE. En effet, le seul schéma manipulé est un schéma dynamique. Cette limite est levée dans la suite du projet, nommée CAIS-A.

Une autre limitation de CAIS par rapport à PCTE est la stricte orientation de la version actuelle de CAIS vers Ada. PCTE ne fait, lui, aucune hypothèse restrictive sur les langages utilisés dans les IPSEs supportés.

Le langage de manipulation de données CAIS est aussi restreint que celui de PCTE. La consultation des nœuds est limitée à un parcours des liens et à une exploration simple des différents nœuds (accès aux liens et aux attributs d'un objet particulier).

4

Tableaux comparatifs des systèmes étudiés

En vue de synthétiser les avantages et les inconvénients de chacun des systèmes décrits, nous allons en dresser un tableau récapitulatif dans lequel nous inclurons Unix comme élément de comparaison. Ce tableau récapitulatif ne se veut pas complet ou exhaustif des avantages ou inconvénients de chacun des systèmes étudiés. Nous avons essayé de comparer les systèmes sur des caractéristiques qui peuvent être communes à plusieurs d'entre eux, c'est à dire essentiellement des caractéristiques de systèmes d'information et des caractéristiques de systèmes d'exploitation. Ces résultats sont extraits de [STU88] et [CHA88].

4.1 Projets

Présentation des maîtres d'œuvre des projets et le type de résultat de chacun d'eux

Système	Maître d'œuvre	Résultat
Unix	Université de Berkeley, AT&T	Système fiable largement utilisé
SKB	Université de Santa Barbara (Californie)	Prototype
PMDB	Société TRW (Redondo Beach, Californie)	Prototype
Concerto	CNET (Lannion, FRANCE)	Prototype en cours de transformation en produit par la SEMA-METRA
PCTE/PCTE+	European Standard Tool Interface	Spécifications
Emeraude	Groupement d'Intérêt Economique Emeraude	Réalisation de PCTE (produit industriel)
PACT	PACT Project	Outils (seront diffusés fin 1989)
CAIS	Département de la Défense (D.o.D/USA)	Spécifications

4.2 Gestion d'Objets

Capacité de gestion d'objets individuels de différents types.

Système	Gestion d'Objets
Unix	Gestion de fichiers et de directories
SKB	Gestion d'objets (`atoms`), d'attributs et de relations
PMDB	31 types d'objets, 220 types d'attributs et 170 types de relations
Concerto	Oui
PCTE/PCTE+	Oui
CAIS	Oui

4.3 Gestion d'objets complexes ou composites

Gestion et description d'objets composés d'autres objets.

Système	Gestion d'objets complexes ou composites
Unix	Non
SKB	Non
PMDB	PMDB ne gère pas réellement des objets composites, mais certains attributs sont des éléments composites et peuvent être vus comme des objets
Concerto	Non
PCTE/PCTE+	Pas dans PCTE mais dans PCTE+
CAIS	Non

4.4 Objets de taille importante

Capacité du système à gérer des objets de taille variable et pouvant être importante.

Système	Gestion d'objets complexes ou composites
Unix	Les fichiers sont de taille variable (très grande taille possible)
SKB	Oui, le type d'un attribut peut être fichier (`file`)
PMDB	PMDB fournit des types d'attributs prédéfinis (comme `textual data` et d'autres éléments composites) qui peuvent être de taille variable
Concerto	Les outils autres que le Système d'Information (F1) peuvent manipuler une Représentation Interne (RI) d'un objet de grande taille mais pas F1
PCTE/PCTE+	Oui, car les Objets peuvent avoir un contenu de type fichier et donc de la taille d'un fichier
CAIS	Oui, car les Nodes ont un contenu de type fichier

4.5 Définition de nouveaux types de données

Possibilité de définir de nouveaux types de données, ou types d'objets

Système	Définition de nouveaux types de données
Unix	Non
SKB	Oui
PMDB	Non
Concerto	Oui
PCTE/PCTE+	Oui, pour les associations, attributs et types d'Objets, mais pas pour les types domaine
CAIS	Non

4.6 Langage de requête

Capacité du système à fournir des moyens de formuler des requêtes complexes sur la base sans avoir à les exprimer de manière navigationnelle

Système	Langage de requête
Unix	Non
SKB	Oui
PMDB	Oui
Concerto	Oui
PCTE/PCTE+	Non, mais facilité offerte dans PACT
CAIS	Non

4.7 Espaces de travail, vues et schémas

Capacité du système à fournir à l'utilisateur les moyens de définir sa propre vue, de travailler dans son espace de travail et de définir des schémas de base de données

Système	Espaces de travail, vues et schémas
Unix	Non
SKB	Non
PMDB	Oui
Concerto	Concerto permet la définition de schémas grâce à F1
PCTE/PCTE+	Oui, PCTE fournit des schémas de définition et d'intégration (`SDS`). La vue est alors le schéma de travail (`WS`) PCTE gère aussi un espace de travail par utilisateur
CAIS	Non

4.8 Expression de contraintes sémantiques

Possibilité d'exprimer des contraintes sémantiques sur la base d'objets manipulés.

Système	Expression de contraintes sémantiques
Unix	Non
SKB	Oui, avec le CRAC
PMDB	Non
Concerto	Oui, avec les clauses de F1
PCTE/PCTE+	Non
CAIS	Non

4.9 Mécanismes de triggering

Possibilité de déclenchement automatique d'actions par le système en réaction à des situations prédéfinies.

Système	Mécanismes de triggering
Unix	Non
SKB	Oui, avec le concept de **daemon**
PMDB	Non
Concerto	Non
PCTE/PCTE+	Non, mais en cours dans ALF
CAIS	Non

4.10 Gestion des événements en tant qu'objets

Possibilité de gérer les événements intervenant dans le système.

Système	Gestion des événements en tant qu'objets
Unix	Non
SKB	Les événements sont gérés grâce aux **daemons** mais pas comme des objets
PMDB	Non
Concerto	Non
PCTE/PCTE+	Les événements sont représentés par des messages aux queues de messages qui sont des objets
CAIS	Les événements sont représentés par des messages aux queues de messages qui sont des objets

4.11 Gestion de données historiques

Capacité du système à garder des traces des opérations sur la base.

Système	Gestion de données historiques
Unix	Non
SKB	Non
PMDB	Non mais tous les "jalons" et les programmes intermédiaires sont gardés dans l'objet `Milestone`
Concerto	Non
PCTE/PCTE+	Oui, PCTE+ possèdera des mécanismes d'audit
CAIS	Non

4.12 Gestion de Version/Configuration

Capacité de gestion de versions et de configuration d'un groupe d'objets.

Système	Gestion de Version/Configuration
Unix	Non, mais il existe des outils sur Unix le permettant (Make, SCCS)
SKB	Non
PMDB	Oui, l'objet `Software Configuration` caractérise les différentes versions des composants logiciels d'une même configuration
Concerto	Non
PCTE/PCTE+	Pas dans PCTE, mais PCTE+ possède quelques mécanismes de base pour la gestion de version et PACT fournit un gestionnaire de Version avec le VMCS
CAIS	Non

4.13 Gestion d'activités

Capacité du système à permettre l'exécution d'activités (processus ou ensemble de processus.

Système	Gestion d'activités
Unix	Oui, Unix gère des processus
SKB	Non, les processus sont gérés par le système hôte
PMDB	Non, les processus sont gérés par l'environnement de développement de TRW auquel PMDB est associé
Concerto	Oui
PCTE/PCTE+	Oui, PCTE gère les processus. Il permet aussi de gérer des `activités` qui sont l'équivalent des transactions des SI
CAIS	Oui, CAIS gère les processus

4.14 Contrôle d'accès

Capacité du système à contrôler les droits d'accès aux objets pour les utilisateurs.

Système	Contrôle d'accès
Unix	Oui
SKB	Non
PMDB	Oui
Concerto	Oui
PCTE/PCTE+	Oui dans PCTE+, par l'intermédiaire des `discretionary and mandatory access control`
CAIS	Oui, par l'intermédiaire des `discretionary and mandatory access control`

4.15 Gestion de communication

Possibilité pour le système de communiquer avec d'autres systèmes.

Système	Gestion de communication
Unix	Oui
SKB	Non actuellement, mais à terme SKB devrait être capable de communiquer avec le système hôte
PMDB	Non
Concerto	Non
PCTE/PCTE+	Oui
CAIS	Oui

4.16 Conclusion

Système	Conclusion
Unix	Unix a seulement été cité comme élément de comparaison. C'est le premier système (d'information et d'exploitation) pour le génie logiciel.
SKB	SKB est un prototype de base de données qui présente comme intérêt ses contraintes sémantiques et les actions associées à ces contraintes par les `daemon`.
PMDB	Le but de l'étude PMDB par TRW était de jeter les fondations pour l'implantation d'une base de données pour le génie logiciel, mais le produit lui-même est juste une base de données spécifique.
Concerto	Concerto est un atelier intégré paramétré intéressant par les concepts qu'il met en œuvre mais se devant de passer de l'état de prototype à celui de produit.
PCTE/PCTE+	PCTE et PCTE+, sont des structures d'accueil supports pour des IPSES. Par leur mécanisme de définition de schémas (`SDS`) et de gestion d'objets, elles sont à l'heure actuelle les meilleures.
CAIS	CAIS est aussi une structure d'accueil mais le désavantage essentiel par rapport à PCTE ou PCTE+ est l'inexistence de schéma de travail et la complète orientation de CAIS vers le langage ADA. Cependant la nouvelle version de CAIS (CAIS-A) devrait apporter une meilleure réponse à ces deux problèmes.

PARTIE III

PROPOSITION D'UN MODELE POUR LE PROCESSUS DE PRODUCTION DE LOGICIELS (SOFTWARE PROCESS) : DEUXIEME ETAPE VERS LES ATELIERS DE GENIE LOGICIEL DE TROISIEME GENERATION

1
La modélisation du processus de production de logiciels

1.1 Introduction

L'étude faite dans la deuxième partie montre la position centrale de la fonction de gestion des informations dans la production de logiciels. Comme nous l'avons vu à travers les différentes études faites, cette fonction est réalisée de différentes façons :

- couplage d'un Système d'information (SKB, PMDB) à un système de développement;
- développement d'une fonction spécifique de gestion de l'information (F1) à l'intérieur d'un atelier (Concerto);
- développpement de structures d'accueil centrées autour d'un Système de Gestion d'Objets (CAIS, PCTE).

Cette gestion de l'information répond aux besoins de stockage et de recherche d'informations associées à la production de logiciels. Parmi ces différents choix pour la réalisation de la fonction de manipulation et de stockage de l'information, l'approche structure d'accueil centrée autour d'un Système de Gestion d'Objets sera sans conteste la première étape vers les ateliers de génie logiciel de troisième génération.

Ces ateliers de troisième génération auront pour première caractéristique une gestion cohérente des informations liées à la production de logiciels. Cette gestion cohérente des informations permettra de répondre aux propriétés de *cohérence* et de *structuration des données* présentées dans le chapitre 1 de la partie I. Une première étape vers la propriété de *cohérence* est atteinte de la sorte : tout outil transforme ou génère des objets du SGO et donc laisse les informations qu'il transforme ou génère dans une forme utilisable par tout autre outil. La *structuration des données* se fait, elle, grâce à ce système de gestion d'objets qui structure les données manipulées soit sous la forme d'un schéma E-A typé (PCTE), soit sous forme d'un graphe orienté (CAIS).

Nous allons succintement reprendre dans le paragraphe suivant les caractéristiques essentielles des modèles des différents systèmes de gestion de l'information étudiés.

1.2 Modèles utilisés dans les systèmes de gestion de l'information des projets présentés

Les études réalisées sur les projets tels que CAIS, PCTE, SKB et PMDB montrent une convergence des vues vers une modélisation basée sur le modèle Entité Association. Ce choix du modèle ne fait pas pour autant converger les projets vers un modèle de représentation et de gestion de l'information unique. En effet de nombreuses différences subsistent à tous les niveaux :

- degré d'intégration du modèle dans l'atelier correspondant (AGL, AIGL ou Structure d'accueil),
- degré de proximité du modèle final au modèle E-A,
- choix d'implantation des modèles.

Enfin tous ces projets ne sont pas dans le même état d'avancement et le produit correspondant varie d'une spécification papier à une implantation industrielle (comme Emeraude, par exemple).

Degré d'intégration du modèle dans l'atelier : Les différences intervenant au niveau de l'intégration des modèles de représentation des objets et des modèles manipulés dans l'atelier sont -à notre avis- essentielles dans les critères d'évaluation des différents produits étudiés. Le degré d'intégration du modèle dans l'atelier indique si le modèle est réellement interconnecté aux autres composants de l'atelier ou s'il est simplement juxtaposé à celui-ci.

Si F1, le système d'information de Concerto (cf. chapitre 2 de la partie II) propose bien un modèle E-A, l'intégration de F1 à Concerto est effectuée de manière "ad hoc". La représentation des objets manipulés est indépendante des objets physiques. F1 gère une base d'information indépendamment de la gestion des objets de Concerto.

PMDB étant uniquement -à l'état actuel- une étude des besoins en gestion d'informations dans un atelier de génie logiciel, le modèle est complètement déconnecté de l'atelier. Le degré d'intégration dépendra des choix faits, par la suite, par TRW pour l'implantation du système de gestion d'informations au sein de leur atelier.

La représentation d'objets (*atoms*) par SKB et de leurs relations est indépendante du système support des objets. Cependant un couplage sera éventuellement réalisé dans la suite du projet, mais l'atelier final obtenu sera plus proche d'une boîte à outils que d'un atelier intégré.

Tous ces systèmes restent des systèmes "ad hoc" même si un effort est fait au niveau de la structuration de la partie statique en développement de projets, cet effort porte uniquement sur une représentation plus ou moins fine des objets et non pas sur les objets eux-mêmes. La meilleure intégration entre le modèle de représentation de la partie statique et la partie statique elle-même d'un atelier est réalisée par les structures d'accueil CAIS et PCTE. Dans celles-ci, aucune distance n'existe entre l'objet de la structure et sa représentation. Ceci n'est pas sans effet sur le degré de proximité du modèle final au modèle E-A.

Le modèle nodal de CAIS définit le modèle de gestion des nœuds, de leur contenu, de leurs relations et de leurs attributs. Le SGO de PCTE, tout en permettant de modéliser la partie statique, se substitue totalement au système de gestion de fichier et les objets manipulés sont toujours les mêmes quel que soit l'outil. Il n'y a plus de séparation entre les objets du système de développement et les informations associées pour la gestion de projet ou pour le plan qualité ou toute autre activité de la production de logiciels. Tous les composants de la structure utilisent les mêmes objets même s'ils s'intéressent plus à tel aspect de l'objet qu'à tel autre.

Degré de proximité du modèle conceptuel au modèle E-A : Le modèle de F1 reprend bien tous les concepts du modèle E-A et l'étend même avec les concepts de clef, et de contraintes.

SKB lui étend le modèle E-A avec les concepts de démons et le concept de CRAC.

PMDB utilise le modèle E-A sans chercher à introduire de nouveaux concepts ni à l'adapter à ses besoins.

Si, du fait de leur non intégration dans un atelier, les modèles de F1, PMDB et SKB ne sont pas soumis aux contraintes de l'atelier, les modèles des structures d'accueil étant intimement liés à l'atelier, de nombreuses limites leur sont imposées.

Ainsi dans PCTE, la notion d'association est une notion tout-à-fait fictive. Le modèle est basé sur des objets et des liens unidirectionnels entre objets. On appelle alors association entre deux objets, deux liens unidirectionnels inverses. Cependant, rien n'interdit de modéliser complétement une application sans utiliser du tout d'associations. Ce choix de liens unidirectionnels a été guidé par le choix de conception énoncé dans [PCT86] à savoir que *"tout accès à un objet est le résultat d'une requête navigationnelle ou d'un chemin d'accès dont la syntaxe est un sur-ensemble de celle d'Unix"*. Par ailleurs PCTE ne fournit pas de primitives de manipulation d'un ensemble d'objets ou de liens -"a set at a time"-, mais uniquement des primitives de manipulation d'un objet (ou lien) unique -"an object at a time"-. Le modèle du DQMCS (Data Query and Manipulation Common Services) de PACT résoud en partie ce problème.

Dans CAIS, le modèle est encore plus éloigné du modèle Entité Association classique. Outre la manipulation de liens unidirectionnels pour représenter des associations, la notion de schéma n'existe pas. Le type d'une entité ou d'une association n'est défini que comme un ensemble d'occurrences.

Choix d'implantation des modèles : F1 propose une implantation réalisée de manière "ad hoc" en le_lisp. Cette implantation aboutit à un prototype mono-utilisateur sans gestion de concurrences d'accès ni de recouvrements d'erreurs. SKB et PMDB se proposent de réaliser une implantation de leur modèle sur INGRES [STO76], système de gestion de bases de données relationnelles. PCTE a été implanté autour d'Unix par le GIE Emeraude. CAIS ne supporte pour l'instant aucune implantation (sauf quelques prototypes expérimentaux).

1.3 Prise en compte de la dynamique dans les projets étudiés

Les outils comme F1 et SKB proposent quelques éléments de solutions à la modélisation du comportement du système de production.

F1 par l'intermédiaire de la clause *respecte* permet la description du comportement attendu lors de l'évolution des objets de la base, évolution réalisée au travers des opérations de création, mise à jour et suppression. Par ailleurs F1 permet de définir des pré-conditions associées aux opérations. Celles-ci décrivent alors le comportement individuel des opérateurs. L'opération n'est alors réalisée que si la précondition est vérifiée. Ces préconditions permettent le refus d'opérations inappropriées dans un contexte donné. Nous avons réalisé une étude plus complète dans [BEN88b].

SKB propose un calcul des prédicats *CRAC* permettant de décrire des contraintes sur les objets manipulés, et permet d'associer à ces contraintes des actions à entreprendre par le système lorsque celles-ci sont violées. Le mécanisme de *démon* permet de vérifier les contraintes exprimées en termes de CRAC et de déclencher les opérations associées. Cependant ces mécanismes ne sont pas installés sur le prototype construit autour de INGRES.

Les structures d'accueil actuelles, si elles prennent correctement en compte la représentation de la partie statique de la production de logiciels (le WS de PCTE ou les relations de CAIS) ne représentent la partie dynamique de celle-ci que de manière superficielle. Ceci étant dû à la non prise en compte par les structures d'accueil des méthodes de travail et du comportement attendu du système de production. La prise en compte de la partie statique dans une structure d'accueil se faisant grâce à la modélisation de celle-ci, la modélisation de la partie dynamique apparaît comme son complément pour la prise en compte de la

dynamique dans la production de logiciels. Cette prise en compte de la dynamique ayant pour objectif de fournir assistance et contrôle lors de la production de logiciels.

1.4 Objectifs du contrôle et de l'assistance en développement de projet dans ALF

Dans cette section, nous allons définir les objectifs que nous nous sommes assignés dans le cadre du projet ALF en vue de fournir les mécanismes de base permettant l'assistance et le contrôle en production de logiciels. Ces objectifs sont ébauchés dans l'annexe technique du projet [ALF87], sont résumés dans l'ALFRAC [LEG88] et établissent le cahier des charges du projet. Nous nous proposons de construire un système capable d'initiatives afin d'assister et de contrôler l'utilisateur dans son travail.

Contrôle et initiative : Le système devra garantir que les initiatives de l'utilisateur sont acceptables. Initiative acceptable ne signifie pas forcément acceptable telle quelle ou rejetable, le but de ALF étant de forcer, si possible, la conformité de l'initiative à la méthode de travail utilisée. Pour ce faire, le système pourra prendre des initiatives.

Les initiatives du système pourront être utiles pour l'observation du système ou pour le contrôle de l'utilisateur. L'observation de l'utilisateur, l'assistance à l'utilisateur et le contrôle des activités d'un utilisateur requièrent un système à capacités d'initiatives.

Conseil et assistance : Le système fournira une assistance pour la prise de décision et la planification en gestion de projet. Il fournira en outre à l'utilisateur un certain nombre de conseils :

- comment invoquer un opérateur ?
- quels sont les opérateurs accessibles dans le contexte actuel ?
- quel est l'état d'avancement de mon travail dans le projet ?
- quelles sont les tâches essentielles à effectuer à ce niveau du projet ?

Observation du système et des utilisateurs : Le système devra fournir les mécanismes de base permettant de définir les mesures pertinentes à effectuer, et les conditions de déclenchement de ces mesures. La gestion de ces mesures et des résultats délivrés ne devant être bien sûr accessibles qu'au gestionnaire de projet. Ces mesures se rapporteront aussi bien à des mesures de qualité de produit qu'à des mesures de performance des intervenants, ces mesures de performance permettant de moduler le profil des utilisateurs. ALF doit supporter la gestion des profils utilisateurs, ceux-ci étant soumis à changement.

Le système doit être capable de moduler le profil d'un utilisateur en fonction des mesures effectuées. ALF pourra, en outre, fournir au gestionnaire de projet l'état d'avancement d'un projet et éventuellement les tâches critiques, les délais prévisibles, les variations de coût. L'observation du système nécessite par ailleurs l'observation par ALF des événements survenant dans la vie d'un projet.

En se basant sur une observation fine du système, ALF a donc pour but de fournir assistance et conseil à l'utilisateur, de contrôler le développement de projet tout en prenant un certain nombre d'initiatives permettant de guider ce développement.

1.5 Assistance dans la production de logiciels

Cette partie a pour but de présenter un ensemble représentatif des différentes fonctionnalités d'assistance qu'un IPSE de troisième génération doit fournir. Ces fonctionnalités doivent s'ajouter à celles déjà existantes ou déjà répertoriées comme la conservation de l'information, la structuration des données, etc...caractérisant les ateliers de deuxième génération. Nous nous intéresserons dans un premier temps aux fonctionnalités concernant le développement de logiciels, puis nous évoquerons celles concernant la gestion du projet.

1.5.1 Développement de logiciels

Un développeur de logiciels attend de son IPSE qu'il possède les fonctions d'aides suivantes [GOD89a] :

- **Réaliser un travail :** un IPSE doit au moins être capable d'exécuter un processus et de manipuler les objets apparaissant lors de la production de logiciels : un IPSE doit au moins réaliser les fonctions de base de tout système d'exploitation.
- **Prendre des initiatives :** c'est le souhait de base de toute industrie de production de logiciels. Un IPSE doit être capable de prendre des initiatives, c'est-à-dire d'exécuter et de combiner l'exécution d'opérations sans l'intervention de l'utilisateur. Cette capacité d'initiatives pouvant décharger l'utilisateur de toutes les tâches fastidieuses que l'ordinateur fait très bien. Par exemple l'édition de liens de tout module testé avec l'ensemble des autres modules testés d'un projet. Le système peut très bien réaliser cette tâche en prenant l'initiative de celle-ci dès qu'un module est testé. L'ordinateur peut prendre l'initiative d'une tâche et la réaliser et ceci pour une quantité de tâches qualifiées d'automatiques (contrôle syntaxique, réalisation de statistiques, compilation, etc). Il est cependant clair que certaines opérations nécessitent l'intervention de l'utilisateur (édition de texte, création de programmes). Mais même pour celles-ci, la prise d'initiative par l'ordinateur peut se traduire par le lancement de la tâche à réaliser et par la mise à la disposition de l'utilisateur de tous les outils logiciels nécessaires à sa bonne réalisation.

- **Contrôler :** un IPSE doit protéger l'utilisateur de ses propres initiatives si elles sont intempestives. En effet une initiative de l'utilisateur peut être inadéquate au regard de l'état d'avancement du projet, l'utilisateur devant respecter le modèle de production de logiciels proposé. En outre, un IPSE doit garantir aux utilisateurs des droits différents (un utilisateur a un rôle dans la production de logiciels et doit se limiter aux droits inhérents à son rôle)
- **Fournir de l'assistance spécifique :** à tout moment l'IPSE doit être capable d'assister un utilisateur dans sa tâche. Cette assistance spécifique a les différentes facettes suivantes :
  - **Guidage :** un IPSE doit guider l'utilisateur dans son travail. Ce guidage peut revêtir plusieurs aspects:
    * *Que puis-je faire ?:* un IPSE doit être capable de proposer à l'utilisateur les différentes opérations que celui-ci peut réaliser. L'applicabilité ou non d'une opération dépend de l'état d'avancement du logiciel en cours de développement et pas seulement de droits d'accès comme c'est trop souvent le cas dans les systèmes d'exploitation courants. En effet, rien n'interdit actuellement (sous Unix par exemple) la compilation d'un module pendant que celui-ci est en cours d'édition et donc de modification.
    * *Comment ça marche ?:* un IPSE doit non seulement être capable d'expliquer comment un outil marche (aide classique dans les systèmes d'exploitation, man d'Unix par exemple) mais doit aussi être capable d'expliquer comment générer un objet d'un certain type. Il doit être capable de proposer des combinaisons d'opérations pouvant conduire à la production d'un objet de ce type.
    * *Comment le faire?:* un IPSE doit être capable de proposer au moins une combinaison d'opérations en vue de produire un objet spécifique d'un type spécifique.

    La différence entre le *Comment ça marche ?* et le *Comment le faire?* est introduite par la prise en compte du contexte dans le deuxième cas. Le *Comment le faire?* est "contexte sensitif". Il concerne un but particulier à atteindre dans un contexte particulier, alors que le *Comment ça marche ?* est plus général.
  - **Simuler :**
    * *Quel est son effet ?:* cette fonctionnalité permet à l'utilisateur de découvrir les effets qui découleront de l'exécution de telle ou telle opération. Toute

initiative a des effets de bord : par exemple la modification d'un module a pour effet de nécessiter une nouvelle édition de lien de tous les modules l'utilisant. Savoir, à priori, l'effet d'une telle opération peut permettre d'éviter des catastrophes ou des pertes de temps considérables. Si le *Quel est son effet ?* appliqué à la modification du module montre la nécessité de faire l'édition de liens de tous les autres modules d'un projet et de lancer une procédure de contrôle complexe, cette demande de modification peut être annulée si elle concernait par exemple une écriture à l'écran en inverse vidéo ou tout autre modification mineure.

- **Expliquer :**
  - Quand un IPSE ne peut pas réaliser ou terminer une tâche il doit être capable d'*expliquer* à l'utilisateur les raisons de son échec et de lui proposer, si possible, un moyen de l'éviter ou de corriger cet échec.
- **Rapporter:**
  - *Qu'est ce qui a été fait ?:* un IPSE doit pouvoir indiquer à l'utilisateur les tâches qui ont déjà été réalisées et donner toutes les indications nécessaires sur l'état du projet en cours de développement.

- **Permettre à l'utilisateur d'apprendre:** les fonctionnalités *Que puis-je faire*, *Comment ça marche*, *Comment le faire* et *Expliquer* peuvent s'utiliser et s'enchaîner de manière à permettre à l'utilisateur d'apprendre à développer du logiciel et à utiliser l'environnement de développement.
- **Communiquer:** le développement d'un gros logiciel nécessite l'intervention d'un grand nombre de personnes. La fonctionnalité de communication leur permet de se comprendre et de coopérer.

1.5.2 Gestion de projet

D'un point de vue général, la gestion de projet est aussi une activité qui consiste à appliquer des opérations à des objets. Les fonctionnalités d'assistance que nous avons proposées peuvent donc s'appliquer à la gestion de projet.

Cependant les opérations du gestionnaire concernent essentiellement la mesure du travail effectué et la planification du travail à fournir en fonction des objectifs et des délais impartis. Aussi de nombreuses observations doivent être réalisées lors du développement de logiciels, ces observations étant ensuite analysées et traitées par le gestionnaire de projet.

Outre les fonctionnalités précédemment citées, un IPSE de troisième génération doit fournir **des capacités d'observation et de mesures.**

1.6 Les motivations pour la modélisation des processus de production

Nous verrons dans ce paragraphe différentes approches ayant conduit d'abord à la modélisation du processus de production de logiciels [1] et ensuite à des tentatives de traitement automatique de cette modélisation. Les travaux sur la modélisation des processus de production et des ateliers est encore à l'état embryonnaire [DER88] [DER89] [OST87] [PER88] [KAI88] [BEN89b] et les différents projets s'apparentent plus à des ambitions qu'à des produits ou des spécifications complètes.

La première question à se poser pour bien cerner le problème est :

"Qu'est-ce qu'un processus de développement?"

Quelques réponses reflétant quelques-unes des opinions exprimées dans [3SPW] et [4SPW] peuvent être extraites de [RID86] [ROB88] et [HUM88] :

Pour Riddle et Williams [RID86], *"le processus de développement de logiciels est la séquence d'activités effectuées durant la création et l'évolution d'un système logiciel."* [2]

Pour Roberts [ROB88], *"les modèles de processus représentent l'organisation des activités réalisées soit par les utilisateurs soit par les systèmes, et la coopération requise par ceux-ci."* [3]

Pour Humphrey [HUM88], *"le processus de génie logiciel est l'ensemble de toutes les activités de génie logiciel nécessaires à la transformation des besoins de l'utilisateur en logiciel."* [4]

Osterweil [OST87], quant à lui, voit le processus comme étant *"l'approche systématique de la création d'un produit ou de la réalisation d'une tâche."* [5]

Taylor et cie [TAY88] ont basé leur projet ARCADIA sur la notion de *programme de processus*[6] dont ils donnent la définition suivante : *"un programme de processus indique comment les différents outils et objets logiciels devraient être coordonnés pour supporter un processus."* [7]

Considérant ces vues d'un processus de production, il apparaît évident qu'il existe

1. Nous utiliserons les dénominations processus de production de logiciels, processus de développement de logiciels, ou encore processus logiciel pour traduire l'expression Software Process.
2. "The software process is the sequence of activities performed during the creation and evolution of a software system."
3. "Process models capture the organisation of activities performed by both users and machines, and the required cooperation between them."
4. "The software engineering process is the total set of software engineering activities needed to transform a user's requirements into software."
5. "We shall take as our elementary notion of a process that it is a systematic approach to the creation of a product or the accomplishment of some task."
6. process program
7. "A process program indicates how the various software tools and objects would be coordinated to support a process."

un processus de production pour chaque système logiciel développé. Cependant plusieurs processus de développement suivent le même paradigme, c'est-à-dire le même modèle de processus de production. Nous employons le terme paradigme dans ce contexte tel que défini dans [DAV88].

Un modèle pour le processus de production de logiciels décrit les caractéristiques générales d'une classe de processus de développement mais pas les caractéristiques qui sont spécifiques à chaque processus. En adaptant le modèle aux besoins spécifiques à une société ou à un projet et en raffinant le modèle, un modèle de développement de logiciels adapté au problème considéré peut être fourni.

Les raisons en faveur d'une description des caractéristiques générales des processus de développement plutôt qu'une description d'un processus spécifique peuvent être trouvées dans [CHR88] et [OST87]. Osterweil mentionne explicitement que *"les humains préfèrent créer les spécifications d'une solution généralisée et construire des instances plutôt que résoudre directement et de manière répétitive des problèmes"* [8].

Deux façons distinctes de décrire des modèles de développement de logiciels peuvent apparaître :

- une description formelle du modèle en termes de notation formelle
- une description narrative ou informelle [9]

Une description narrative est malheureusement trop souvent imprécise, elle se résume parfois en conseils en tous genres : "commencez le travail par un cahier des charges" ou "testez soigneusement tout module produit". Elle est source d'incompréhension et d'erreurs. Une description narrative du processus de développement de logiciels est désavantageuse (même si une telle description est préférable à une absence totale de description), et une description formelle trouve son intérêt pour les raisons suivantes :

Complexité croissante des systèmes logiciels

La complexité des systèmes logiciels actuels fait apparaître le besoin de modèles plus élaborés pour le développement de logiciels et de mécanismes de contrôle du processus de développement plus efficaces. Plus le nombre de personnes impliquées dans le développement d'un système logiciel est grand, plus la tâche de coordination sera complexe. Tant qu'un ou deux programmeurs seulement développent un logiciel, il est suffisant de coordonner leur travail par des conseils généraux du type "Faites le cahier des charges avant les spécifications" ou "testez soigneusement tout module produit". Mais comme le nombre d'intervenants croit en fonction de la complexité du système logiciel, de tels

8. "Humans prefer to create generalized solution specifications and build instances rather than directly and repetitively solving problems"

9. "Narrative descriptions have been employed by organizations to record their standard operating procedures - a form of process description" [KEL88]

conseils sont complétement dépassés et inefficaces pour de gros logiciels. Une description formelle est nécessaire pour intégrer le développement de logiciels, la gestion de projets et toutes les activités relatives à la production de logiciels.

Compréhension/communication

Le développement de gros systèmes logiciels ne peut pas être réellement compris dans son ensemble sans une définition explicite et formelle du processus de développement. Une description formelle du processus permet l'identification de l'état courant de celui-ci et diminue le danger de ne plus savoir où l'on en est dans le développement ni quel est le but de ce développement. En outre "le gestionnaire de projet peut communiquer avec les développeurs, clients et d'autres gestionnaires" [10] à travers un "process program" [OST87] (qui n'est rien d'autre qu'une description explicite d'un processus de développement). Il est clair que des descriptions explicites du processus de développement forment une meilleure base pour la communication qu'une simple collection de conseils.

Structuration du développement de logiciels

Seule une description explicite et formelle du développement de logiciels permet une structuration complète et cohérente du développement. Une description informelle du développement de logiciels ne peut être qu'informellement structurée, ceci impliquant une non-structuration du développement. Pour des développements de logiciels complexes, un mécanisme formel de structuration est nécessaire afin de pouvoir travailler sur des composants du logiciel clairement définis.

Réutilisation de processus de développement

Si l'information à propos du processus de développement est dans la tête des gestionnaires ou écrite informellement, cette connaissance est difficilement exploitable pour des développements futurs. Des descriptions explicites et formelles du processus de développement sont des conditions nécessaires à la réutilisation de processus de développement. Ce besoin de réutilisation n'est pas spécifique au processus de développement. Il est le prolongement naturel du besoin de réutilisation des programmes ou des structures de données en programmation classique [GEN89].

Toutes ces raisons montrant clairement l'intérêt d'un modèle pour le processus de développement de logiciels, nous allons voir dans le paragraphe suivant quelques approches de modélisation du processus de développement de logiciels.

10. "the manager of a project can communicate to workers, customers and other managers..."

1.7 Différentes approches pour une modélisation du processus de développement

Ces approches concernent toutes la modélisation de la dynamique, soit dans les systèmes d'information comme REMORA, IDA ou MERISE, soit dans les systèmes de développement de logiciels. Ces approches relèvent de projets dont le but premier n'est pas le stockage et la restitution d'informations, ce premier aspect de la production de logiciels ayant déjà été illustré avec F1, SKB, PMDB, PCTE et CAIS.

1.7.1 La modélisation de la dynamique dans les systèmes d'information

Les travaux effectués pour une modélisation de la dynamique des systèmes d'information fournissent une expertise non négligeable pour la modélisation des processus de développement. Nous allons voir quelques-uns des plus intéressants et indiquer pourquoi une telle modélisation de la dynamique des systèmes d'information n'est pas suffisante pour modéliser le processus de développement de logiciels. Nous nous inspirerons dans ce paragraphe de l'étude faite dans [FOU82].

La première modélisation de la dynamique dans les systèmes d'information est la prise en compte des contraintes d'intégrité. Celles-ci sont statiques si elles concernent l'état courant de la base, et dynamiques si elles concernent un changement d'état de celle-ci. Un exemple de mise en œuvre de ces contraintes d'intégrité dans la modélisation de la dynamique des SIs peut être trouvé dans Sybase [HOW86]. Elles sont appelées règles et spécifient les valeurs valides pour un type d'attribut [11]. Des critiques à l'encontre de cette façon de modéliser la dynamique se trouvent dans [FOU82] : *"... pas de concepts précis permettant d'appréhender l'ensemble des phénomènes réels à décrire par les contraintes,... collection de contraintes juxtaposées qui ne donnent pas une vue d'ensemble de la dynamique du système représenté... rend difficile les contrôles nécessaires concernant la complétude ou la cohérence de la représentation effectuée..."*.

Une vraie modélisation de la dynamique des SI a été faite, ensuite, dans des projets comme REMORA [FOU82] [ROL88] ou MERISE [TAR83]. Ce type de modélisation s'appuie sur des concepts moins primaires que les contraintes d'intégrité et permet une expression de la dynamique des traitements dans les systèmes d'information prenant en considération les stimuli auxquels doit réagir le système d'information et la façon dont celui-ci doit réagir.

MERISE est une méthode de conception et de réalisation de système d'information réalisée en 1978-1979 par la mission à l'informatique du ministère de l'industrie. MERISE modélise la dynamique des traitements par :

11. "A rule specify the domain of valid values for a column"

- des **événements** représentant les faits nouveaux pour le système d'information.
- des **opérations** représentant un ensemble de traitements effectués dans le système d'information.
- des **résultats** produits par une opération.
- des **synchronisations** représentant les préconditions pour l'activation d'une opération à partir de plusieurs événements. Cette synchronisation se fait en termes de et, ou, ou exclusif,...

Cette dynamique peut être schématisée par la figure 1.1.

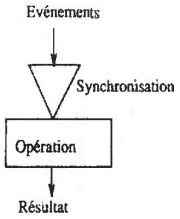


Figure 1.1. Représentation de la dynamique dans MERISE

L'objectif de REMORA [FOU82] [ROL88] est la définition d'un système de pilotage pour l'analyse et la conception des systèmes d'informations. L'analyse du monde réel conduit à la définition d'un modèle descriptif répondant à la question *"que représenter?"*. Ce modèle descriptif de REMORA correspond à une approche causale d'un système, et peut être schématisé dans la figure 1.2.

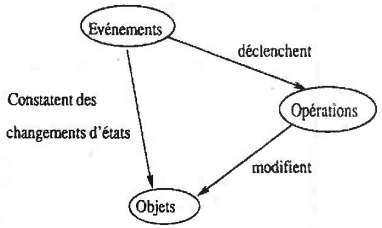


Figure 1.2. Modèle descriptif de REMORA

A ce modèle descriptif est associé un modèle conceptuel permettant une représentation normalisée et intégrée des aspects statiques et dynamiques du système d'information. Les

concepts de ce modèle sont :

- les **c-classes** qui représentent les classes d'objets réels
- les **c-objets** qui représentent des aspects temporels des classes d'objets
- les **c-opérations** qui représentent des **actions** élémentaires ayant pour effet la modification de l'état d'un unique c-objet
- les **c-événements** qui représentent des événements élémentaires et qui déclenchent des c-opérations suivant certaines conditions.

REMORA et MERISE fondent leur modélisation du traitement de la dynamique sur le concept d'événement et sur la prise en compte de celui-ci par le déclenchement des opérations associées. Citons aussi IDA [IDA86a] [IDA86b] dont les concepts pour la modélisation de la dynamique sont voisins de ceux de MERISE.

Les Systèmes d'Information ont été introduits pour traiter automatiquement les informations que peut manipuler un système quelconque du monde réel. L'entreprise est le cas le plus courant de système réel. Réaliser un SI revient à choisir un modèle extérieur pour représenter le système réel ou Système Opérant. Toute information du monde réel doit être représentée par une information du monde de représentation. Les modèles de la dynamique des SI sont donc basés sur la manipulation des duplications ou copies de l'information du système opérant.

Cette approche bien adaptée aux problèmes de l'entreprise ne convient pas au génie logiciel. En effet, en génie logiciel le monde réel ou système opérant est lui-même un ordinateur. Cet ordinateur de développement manipule déjà un certain nombre d'informations et met en œuvre la dynamique du développement. Il faut absolument éviter d'avoir une modélisation de la dynamique (en utilisant REMORA ou MERISE) d'une part, et d'autre part une dynamique effective différente du système de développement. Par ailleurs, dans les systèmes d'informations, l'ensemble des événements pouvant intervenir est clairement identifié à priori et seuls ceux-ci peuvent survenir au cours de la vie du système. Tous les autres événements intervenant dans le monde réel sont ignorés. Lors de la modélisation de processus de développement, tous les événements et toutes les modifications des objets interviennent directement dans la base et ne peuvent de ce fait être ignorés. Qu'il y ait ou non un déclenchement d'opérations ou une quelconque réaction du système, ces événements sont traités ne serait-ce que dans la mesure où ils modifient la base.
Une autre spécificité du génie logiciel rendant difficile l'utilisation des modèles de la dynamique comme Remora ou Merise est la durée des opérations effectuées. Dans les systèmes d'information, une opération de mise à jour dure quelques secondes alors que lors du développement de logiciels une opération de spécification détaillée peut durer des mois.

1.7.2 La programmation de processus

"Process Programming"

Une des premières approches vers une modélisation du processus de développement a été décrite par Osterweil [OST87]. C'est l'approche **"Process Programming"** largement discutée et commentée dans [3SPW]. L'idée principale est de considérer le développement de logiciels comme étant lui-même un logiciel [OST88]. Dans l'approche "Process Programming", le processus complet de développement de logiciels est décrit en termes de langage de programmation en vue de clarifier ce qui doit arriver durant le développement de logiciels. Décrire le logiciel comme un programme étant l'idée essentielle de l'approche d'Osterweil, il reste à choisir le type de langage de programmation à utiliser. Ce choix est à l'heure actuelle encore ouvert : même si les exemples dans [OST87] sont décrits de manière impérative, il est explicitement dit que les langages déclaratifs [12] sont plus appropriés à la description de certains types de processus de développement. L'utilisation d'un langage de programmation pour la description du processus de développement de logiciels est l'idée de base à retenir et non le type de langage utilisé.

L'approche "Process Programming" est critiquée pour diverses raisons. Lehmann [LEH87b] fait remarquer qu'il sera difficilement possible de décrire "à priori" un processus créatif tel que le processus de développement de logiciels, c'est-à-dire avant même le démarrage du développement lui-même. Il est par ailleurs proposé de combiner [CON88] les idées du "Process Programming" avec une certaine modélisation dynamique, une telle combinaison devant permettre de modéliser et "d'exécuter" le processus de développement de logiciels de manière entrelacée. Il nous semble cependant possible de considérer le process programming comme une façon non exhaustive d'indiquer des règles à respecter (cf notre proposition de MASP).

Le projet ARCADIA

L'approche Le projet **ARCADIA** [TAY87] [TAY88] qui se propose de développer un environnement de programmation a été largement influencé par l'approche "Process Programming". Dans ce projet développé par le California University Department of Information and Computer Science, le Massachusets University Department of Computer and Information Science, le Colorado University Department of Computer Science et les sociétés TRW (Redondo Beach, California) et AT&T Bell Laboratories (Murray Hill, New Jersey), le besoin de faire une distinction claire entre une partie fixe[13] (ou statique) et une partie variable[14] (ou dynamique) dans les environnements de génie logiciel a été res-

12. rule-based languages
13. *"fixed part"*
14. *"variant part"*

senti et longuement présenté, la partie variable décrivant le modèle pour le processus de développement de logiciels. Par ailleurs, la nécessité de combiner des concepts de langages procéduraux et des concepts de langages déclaratifs a été présentée et expliquée.

L'architecture globale d'ARCADIA L'un des objectifs d'ARCADIA est le *"développement d'une architecture d'environnement orientée vers un maximum de* **fléxibilité/extensibilité** *et d'***intégration***"* [TAY88].

Cette volonté de **flexibilité/extensibilité** s'est traduite par la notion de programmation de processus. L'activité de production de logiciels étant loin d'être aisée et cette activité n'étant pas toujours identique d'un projet à l'autre, l'idée de base est que des programmes de processus, écrits dans un langage de programmation de processus (PPL pour Process Programming Language), décriront de manière adéquate les différents processus que l'utilisateur veut appliquer dans le développement et la maintenance d'un logiciel. Les outils réalisant une tâche seront des opérateurs ou des fonctions de ce langage et les opérandes seront les objets créés par les outils ou les utilisateurs. Cependant, vu la complexité des processus de production de logiciels, un PPL devra avoir au moins les fonctionnalités d'un langage de programmation classique, plus de nouvelles fonctionnalités dont le but sera de satisfaire les besoins de la programmation de processus de programmation.

Cette idée de fournir une fléxibilité/extensibilité au travers d'un PPL ne doit cependant pas nuire au deuxième objectif affiché, à savoir l'intégration. Cette **intégration** sera faite, et c'est la tendance actuelle [PCT86] [CAI85], grâce à un système de gestion d'objets (SGO). Celui-ci centralise toutes les informations qui seront partagées entre les différents outils. Le projet ARCADIA s'articulera donc autour d'un SGO, d'un interpréteur de programmes de processus et d'une interface utilisateur accentuant l'aspect intégration de l'environnement, cette interface permettant une interaction uniforme entre l'utilisateur et le système. Nous pouvons représenter l'architecture par le schéma de la figure 1.3).

L'interpréteur de programmes de processus est responsable du traitement des instructions des programmes de processus. Il communique avec les utilisateurs à travers l'interface utilisateur et accède aux objets logiciels via le système de gestion d'objets. En outre, ces trois composants de base interagissent avec la machine support pour la réalisation de primitives systèmes de base.

L'architecture décrite dans la figure 1.3 est appelée partie fixe. Elle ne comprend ni les programmes de processus, ni les outils, ni les instances d'objets : elle ne comprend aucun des objets qui sont amenés à changer souvent et qui composent ce que les concepteurs d'ARCADIA appellent la partie variable. Cette claire séparation entre la partie fixe et la partie variable est présentée comme un des résultats essentiels d'ARCADIA.

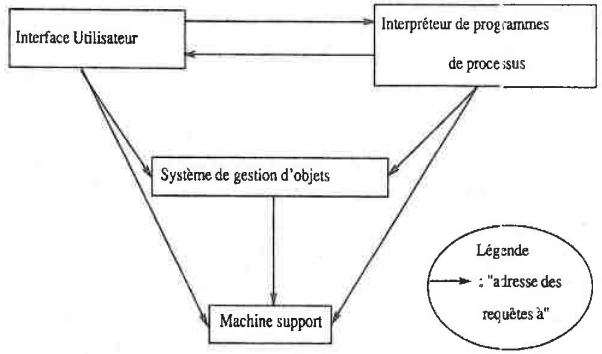


Figure 1.3. Architecture de l'environnement ARCADIA

Les programmes de processus logiciels Afin de développer un langage de programmation de processus réaliste, l'équipe d'ARCADIA base toute son approche de conception du PPL sur l'existant en matière de langages de programmation. Elle essaye juste de généraliser les concepts de ceux-ci et de les étendre aux besoins des programmes de processus. Les langages de contrôle des systèmes d'exploitation sont les *"ancêtres primitifs des langages de programmation de processus"*. Les fichiers de commande et les "shell script" sont les premiers programmes de processus. Cependant les possibilités des langages de contrôle sont, en général, très réduites :

- pas de typage des objets (les fichiers et les variables shell étant les seuls objets manipulés),
- pas de séquences de contrôle complexes (même si dans des langages comme csh sous Unix, la possibilité d'écrire des boucles et des conditionnelles en fait des langages d'un bon niveau par rapport à un langage comme le DOS),
- granularité "grossière" limitée à des opérandes de la taille de fichiers et à des opérateurs de la taille de programmes exécutables,
- ...

Le typage des objets manipulés, des structures de contrôles non triviales et une hiérarchisation des processus sont les premiers pas vers une modélisation du processus de production de logiciels. Les concepteurs d'ARCADIA estiment que la création d'un langage de programmation de processus nécessite forcément une expérience acquise lors de l'écriture de programmes de processus. Cette écriture devant être faite en termes de

langages de programmation de processus, ils se trouvent confrontés à ce qu'ils appellent un "research deadlock". Ils résolvent ce deadlock en se basant sur des langages de programmation existant, même si ceux-ci n'ont pas été conçus en vue de la programmation de processus, afin d'acquérir l'expertise souhaitée. Cette expertise mène à la découverte des besoins spécifiques à la programmation de processus, et à l'adjonction de nouveaux concepts aux langages de programmation d'origine afin de satisfaire ces besoins. C'est ainsi qu'Ada a été utilisé comme l'un des langages de départ pour écrire des programmes de processus, ou plus exactement Appl/A sur-ensemble d'Ada permettant la définition de relations entre les objets logiciels. En effet, la représentation explicite des interconnexions entre objets logiciels est l'un des premiers besoins lors de la description de processus logiciels. Appl/A a servi à écrire des programmes de processus décrivant le modèle de développement en spirale de Boehm [BOE88], puis des programmes de processus décrivant la gestion de configuration et la gestion de projet dans le cadre de PMDB+ (suite de PMDB intégrant l'aspect description de processus). D'autres programmes de processus ont été écrits pour la spécification des besoins, la conception, le test et la maintenance dans le cadre de Bopeep, autre prototype de travail sur la programmation de processus de ARCADIA.

La première conclusion de ces différentes études d'ARCADIA, à travers différents prototypes, a été la découverte d'un besoin essentiel en programmation de processus : la nécessaire **hiérarchisation des types**. Cette hiérarchisation des types permet à des types différents de partager certains aspects communs, ces aspects communs appartenant à un ancêtre commun et étant partagés par les différents sous-types de cet ancêtre.

En outre, un certain **dynamisme** est souvent nécessaire. La programmation des processus ne peut pas toujours se faire à priori; la programmation d'un nouveau sous-programme de processus peut s'avérer nécessaire pendant l'exécution du programme original. Cette programmation doit pouvoir être faite "au vol"[15]. Ce besoin de dynamisme dans la programmation des processus logiciels prône un langage dans lequel les données et le code peuvent être dynamiquement et librement modifiés. Bien que ne répondant pas à d'autres contraintes pour un langage de programmation de processus, Lisp et ses dialectes possèdent cette propriété.

L'utilisation d'Ada a aussi montré les limites des **langages procéduraux** en matière de programmation de processus. Bien que certains processus peuvent être naturellement décrits de manière impérative, le besoin de mécanismes de **description déclarative** (ou à base de règles) se fait ressentir. Le Make [FEL79] peut être considéré comme l'ancêtre de ce type de langage déclaratif pour la programmation de processus.

15. "on the fly"

En conclusion, **pour ARCADIA le langage de programmation de processus idéal serait un langage combinant les paradigmes de programmation impérative (ou procédurale) et de programmation déclarative (ou à base de règles).**

Quelques remarques sur ARCADIA Le projet peut être considéré comme très bénéfique si on le considère comme étude des besoins pour la conception d'un langage de programmation de processus. Cependant aucun langage de ce type n'est fourni et cela est assez normal dans la mesure où le domaine de recherche est tout-à-fait récent et qu'une quantité de travail importante doit être fournie avant d'atteindre un langage de programmation de processus satisfaisant.

PML, un autre langage de programmation de processus

Parmi les langages pour la description de processus de développement de logiciels, nous pouvons citer **PML** (Process Modeling Language), un langage pour la modélisation de processus [ROB88]. Celui-ci est basé sur le cahier des charges établi par **RML** (Requirements Modeling Language) [BOR88] et propose une modélisation centrée objet du processus de production de logiciels. Celui-ci est décrit dans [GRE84] en termes d'entités, d'activités et d'assertions. PML est développé dans le cadre du programme ALVEY (Programme financé par le gouvernement de Grande Bretagne) au sein du projet IPSE 2.5 [WAR89] qui a débuté en septembre 1985 et doit se terminer fin 1989. Nous pouvons voir que les concepteurs des nouveaux langages pour la modélisation (ou programmation) des processus de production de logiciels essayent de tirer profit des différentes facettes des langages de programmation existant, de la programmation procédurale la plus classique à la programmation centrée objet en passant par la programmation déclarative.

1.7.3 Le projet ISTAR

L'approche contractuelle

Une autre approche importante dans le domaine de la modélisation du processus de production de logiciels est l'approche du projet **ISTAR** [DOW86]. ISTAR est un environnement de programmation intégré (IPSE), indépendant de tout langage, développé par IST (Imperial Software Technology). Il a pour but de supporter *"l'approche contractuelle"* [16]. Celle-ci s'apparente aux mécanismes supportés par l'environnement de communications ISA de Concerto (cf chapitre 2 de la partie II). Cette approche du développement de logiciels est assez générale. Elle repose sur l'assertion suivante : "Toute activité dans le développement de logiciels a le caractère d'un contrat." Cela est traduit par le fait que

16. "contractual approach"

l'on peut associer à chaque activité un contractant, un client, au moins un objet produit ("délivrable") et des critères d'acceptation de ce délivrable (cf figure 1.4).

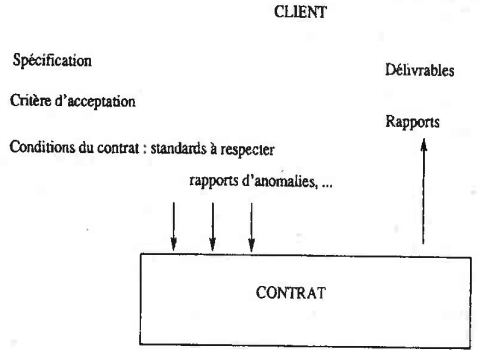


Figure 1.4. Un contrat d'ISTAR

Chaque activité est donc définie en termes de contrat et peut être traitée en étant décomposée elle-même en sous-contrats correspondant à des sous-activités traitées par des sous-contractants. La collection des tâches composant un projet logiciel forment une **hiérarchie de contrats** (cf. figure 1.5). La racine de cette hiérarchie est le contrat pour le projet global; les feuilles sont, elles, des contrats ne nécessitant pas de sous-contrats.

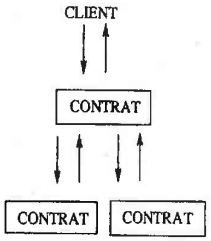


Figure 1.5. Une hiérarchie de contrats sous ISTAR

Considérer le processus de développement comme une hiérarchie de contrats n'impose pas le choix d'un cycle de vie particulier ou d'une organisation particulière du projet. Différentes organisations de projets peuvent être traitées par différentes hiérarchies de contrats. Cependant cette approche impose une stricte division du projet en tâches spécifiques. Le développement de projet, sous ISTAR ne peut se faire si cette division n'est pas clairement établie. Cette division clairement établie, le contractant (personne responsable de la réalisation d'un contrat pour un client donné) doit avoir un certain nombre d'outils à sa disposition pour l'accomplissement de sa tâche. Dans ce but, ISTAR manipule, d'une manière duale à sa hiérarchie de contrats, des ensembles d'outils.

Le noyau de base d'ISTAR

Le noyau de base d'ISTAR supporte l'approche contractuelle à travers la gestion et la manipulation d'une Base de Données (BD) propre à chaque contrat d'un projet. Après la création d'une BD racine à l'initialisation d'un projet, tout nouveau sous-contrat créé voit une BD propre créée pour ses besoins. Une hiérarchie de BDs est donc manipulée en parallèle avec la hiérarchie de contrats.

La majorité des tâches réalisées par un contractant est réalisée sur la BD propre à son contrat. Cette BD sert alors de support et de référence à toutes ces tâches. Cependant, quelques tâches moins nombreuses, telles que la création de nouveaux sous-contrats et donc de nouvelles Bases de Données ou telles que la récupération des résultats d'un sous-contrat nécessitent la coopération entre plusieurs contrats (et par conséquent plusieurs Bases de Données). Le noyau de base d'ISTAR pour chaque contrat est composé de :

- un Système de Gestion de Base de Données incluant la Base de Données pour le contrat et une interface pour celle-ci,
- une interface utilisateur,
- un système de communication permettant l'échange d'informations avec d'autres couples "contrat-BD",
- et enfin, un ensemble d'outils.

Les outils

Les outils manipulés dans ISTAR sont regroupés en "ateliers" -ensemble coordonné d'outils, selon la définition d'ISTAR-. Un utilisateur d'ISTAR sélectionnera un atelier pour réaliser un contrat particulier. A partir de là et jusqu'à la fin de la session, tous les outils faisant partie de cet "atelier" pourront être appliqués au contenu de la Base de Données de ce contrat. "L'atelier" pourra être étendu avec de nouveaux outils, si nécessaire. ISTAR

fournit un certain nombre d'outils utilisables de suite permettant d'utiliser l'approche contractuelle proposée. Ces outils essayent de couvrir les activités classiques de développement et de gestion d'un projet. Cependant et afin de ne pas imposer à l'utilisateur une méthode de travail particulière, celui-ci n'est pas obligé d'utiliser les "ateliers" préexistants et peut développer ses propres outils à condition de les "envelopper" dans une enveloppe ISTAR. Cette enveloppe devra assurer l'interaction de l'utilisateur avec l'outil à travers l'interface utilisateur d'ISTAR, et elle devra assurer la manipulation de la Base de Données du contrat.

Les outils de base fournis par ISTAR sont de six classes générales :

Gestion du projet et des ressources : Ces outils de gestion de projet et des ressources sont essentiels pour ISTAR. Ce sont eux qui permettent la mise en œuvre de l'approche contractuelle décrite précédemment. En particulier, l'outil de définition de tâche [17] faisant partie des outils de gestion de projet est nécessaire au partage d'un contrat en sous-contrats.

Gestion des données et gestion de Configurations : La gestion des données est partie intégrante de ISTAR dans la mesure où à chaque contrat est associée une Base de Données. Lors de la réalisation d'un contrat, les données sont créées et modifiées par ISTAR dans un espace de travail puis "exportées" vers la Base de données du contrat qui est alors sous le contrôle du gestionnaire de configurations d'ISTAR. Ce gestionnaire de configuration permet la manipulation de *versions* successives ou parallèles d'un module. Ces versions sont liées entre elles et servent à la construction de "versions délivrables" [18] de logiciels. Ces "versions délivrables" sont alors exportées de l'espace de travail à la Base de Données du contrat. Elles passent alors sous le contrôle du gestionnaire de configuration.

Développement : Moins intimement liés à ISTAR, à son approche contractuelle et à sa hiérarchie de Base de Données, les outils de développement permettent la réalisation des contrats et la vérification des délivrables des sous-contrats.

Gestion de la qualité : La gestion de la qualité peut être faite sous ISTAR grâce au mécanisme de "vérification d'une liste d'Assurance Qualité" [19]. Ces listes de qualités peuvent être utilisées comme spécification d'un contrat, comme qualité requise d'un délivrable, ou être vérifiées par un contrat s'occupant uniquement de l'aspect qualité dans le projet.

Administration du système : ISTAR devant supporter toutes les activités du génie logiciel, un ensemble d'outils administratifs est fourni. Cet ensemble comprend des

17. Task definition tool
18. Release
19. Quality Assurance check list

primitives telles que l'ajout d'un utilisateur, l'utilisation d'un réseau et l'utilisation du courrier électronique.

Quelques remarques sur ISTAR

La caractéristique essentielle de ISTAR est la hiérarchisation des tâches à effectuer. Cette hiérarchisation se traduit sous forme d'une hiérarchie de contrats qui peut être vue comme une modélisation du processus de production. Cependant ISTAR ne fait aucune différence entre la modélisation du processus et le processus lui-même. Il n'existe pas de description à priori de la hiérarchie de contrats représentant le projet. Cette hiérarchie est construite au fur et à mesure de l'avancement du projet. Une tâche est divisée en sous-tâches uniquement quand le besoin s'en fait sentir. Ce genre de "modélisation" permet une souplesse et un ajustement permanent de la méthode de travail mais est loin de fournir une modélisation satisfaisante du processus de production de logiciels.

1.7.4 Le projet MARVEL

L'approche

Une autre approche est réalisée dans le système **MARVEL** [KAI87], [KAI88], [KAI88b], [KAI88c]. C'est un système à base de règles pour la modélisation et l'exécution de processus de production développé par le Columbia University Department of Computer Science (New York USA) et le Carnegie-Mellon University Software Engineering Institute (Pittsburg, USA). Chaque modèle pour le développement de logiciels est décrit par un ensemble de stratégies et par un ensemble d'activités. Le mot clé de cette approche est *"développement opportuniste"* [20]. Ce type de développement est basé sur la description des activités en termes de préconditions et postconditions. Ces préconditions et postconditions décrivent la précondition nécessaire à l'exécution d'une activité et le résultat assumé de cette exécution.

MARVEL utilise cette description pour produire des règles d'inférence pour un moteur d'inférence à chaînage avant dont le but est de faire réaliser par MARVEL le maximum de tâches possibles. Cependant il est clair que MARVEL ne pourra pas tout faire tout seul. L'utilisateur peut donc demander l'exécution d'une activité ou être sollicité par le système pour en exécuter certaines que le système lui indique. Dans le cas où l'utilisateur demande l'exécution d'une activité dont la précondition n'est pas vérifiée, MARVEL lance un moteur d'inférence à chaînage arrière pour déterminer puis déclencher une séquence d'activités rendant possible l'exécution de l'activité demandée.

En marge des préconditions et postconditions attachées aux activités, MARVEL dis-

20. "opportunistic processing"

pose d'une "méta-connaissance" décrite par des règles ou "indications" [21]. Ces règles contiennent une partie condition et une partie activité et leur rôle est d'indiquer à MARVEL, lors d'un chaînage avant, le moment le plus opportun pour déclencher l'activité sélectionnée. Un ensemble de règles est appelé stratégie et il est possible d'activer et de désactiver des stratégies différentes pour le même ensemble d'activités.

Les activités et leur description

MARVEL s'attache à la description et à la manipulation d'activités d'une taille homogène. Les activités de haut-niveau, regroupant plusieurs tâches et nécessitant éventuellemt la coopération de plusieurs utilisateurs pour leur réalisation sont écartées. MARVEL écarte également les activités de trop bas niveau, comme par exemple l'activité consistant, lors d'une édition d'un fichier, à rechercher une chaîne de caractères (ou toute autre activité du même type). MARVEL ne se préoccupe d'une **activité** que si elle est de **taille moyenne** et qu'on peut lui associer un **outil la réalisant**. Chaque outil réalisant une activité est classé suivant le fait qu'il nécessite une interaction avec l'utilisateur ou pas.

Une activité est décrite de la manière suivante :
$pré_i$;*activité*;$post_i$.

$pré_i$ et $post_i$ sont des formules bien formées du calcul des prédicats du premier ordre, représentant respectivement la précondition et la postcondition de l'activité *activité*$_i$.
Une précondition, associée à une activité, est une expression booléenne devant être vraie pour que l'activité en question puisse être entreprise.
Une postcondition, associée à une activité, est une assertion devenant vraie quand l'activité en question est terminée. Une postcondition peut être une disjonction d'assertions, chacune d'entre celles-ci reflétant un des résultats de l'activité réalisée.

Des exemples d'activités de compilation et d'édition extraits de [KAI88b] :

```
notcompiled(module) and
for all components c such that in(module,component c):analysed(component c)
{compile module}
compiled(module) | errors(module) ;

in(module, component c)
{edit component c}
notanalysed(component) and notcompiled(module);
```

Une activité ne peut être exécutée que si sa précondition est vraie.

L'activité de compilation d'un module (compile module) ne peut être effectuée que si sa précondition est vérifiée, à savoir :
que ce module m n'est pas déjà compilé (notcompiled(module))

21. "hints"

et que l'analyse sémantique des composants c de ce module s'est bien passée
(for all components c such that in(module, component c) : analysed(component c)).

L'activité d'édition d'un composant c d'un module (edit component c) ne peut se faire que si c est un composant d'un module in(module, component c).

La réalisation d'une activité rend vraie sa postcondition.

La compilation d'un module se traduit soit par le fait que le module est compilé (compiled(module)) soit par le fait que des erreurs de compilation subsistent (errors(module)).

L'édition d'un composant c d'un module rend obsolète l'analyse sémantique de celui-là (notanalysed(component)) et la compilation de ce dernier (notcompiled(module)).

L'assistance de MARVEL dans la production de logiciels

La description des activités est l'élément central dans MARVEL, et cette description permet à MARVEL de fournir une assistance à l'utilisateur.

Une première assistance est fournie à l'utilisateur désirant effectuer une activité. Si la précondition de cette activité est vérifiée, MARVEL lance l'exécution de celle-ci et en déduit sa postcondition. On voit apparaître ici une capacité de **contrôle** de MARVEL. Par contre, si la précondition de l'activité requise n'est pas vérifiée, MARVEL lance un **moteur d'inférence à chaînage arrière** sur les règles d'inférence $pré_i \rightarrow post_i$ (déduites de la description d'activités $pré_i$;*activité*;$post_i$) pour déterminer puis déclencher une séquence d'activités rendant vraie la précondition et autorisant donc l'exécution de l'activité demandée. MARVEL cherchera une séquence d'activités ne nécessitant pas d'intervention humaine et qu'il pourra donc réaliser directement.

En dehors d'une requête utilisateur, MARVEL utilise un **moteur d'inférence à chaînage avant** pour faire un développement opportuniste. Le développement opportuniste consiste à utiliser un moteur d'inférence en avant ayant comme règles d'inférences des règles du type $pré_i \rightarrow post_i$ déduites de la description d'activités : $pré_i$;*activité*;$post_i$. Toute règle de ce type, dont la précondition est vérifiée, est sélectionnée par le moteur d'inférence en avant en vue de réaliser l'activité correspondant à ces précondition et postcondition. MARVEL construit alors une liste d'activités à déclencher. Tant que le système est capable de déclencher et de réaliser des activités, son système de chaînage avant est mis en œuvre [22]. Le système choisit, parmi les activités sélectionnées, de déclencher celles apparaissant dans la partie activité d'une "indication".

22. NB: Notons que dans l'exemple cité, le moteur d'inférence en avant doit boucler ! En effet, malgré les explications fournies dans [KAI87] et dans [KAI88b], la précondition de compiler demeure vraie si le résultat est errors(module) et donc compiler peut être activé indéfiniment tant que la compilation n'aboutit pas. La solution à cette boucle se trouve réalisée de manière "ad hoc" dans [KAI88d] : elle consiste en un estampillage du module et de ses composants par la date à laquelle l'opération d'édition ou de compilation a été faite et en l'utilisation de ces dates dans la précondition d'un manière similaire à celle du Make.

Par exemple, la stratégie consistant à ne compiler que les modules non réservés par un utilisateur, ou bien alors réservés mais dont l'utilisateur ne s'occupe pas est traduite par "l'indication" :

```
notreserved(module) or
< reserved(module,userid) and not equals(module,focus(userid)) >
{compile module}
```

Dans l'exemple développé, une opération de compilation ne sera donc entreprise par MARVEL en chaînage avant que si le module n'est pas déjà compilé, tous ses composants ont bien été analysés (précondition) et qu'il n'est pas réservé ou bien s'il l'est, l'utilisateur n'est pas en train de le modifier (indication). A la fin d'une telle opération de compilation, le module est compilé ou bien il existe des erreurs (postcondition).

Quelques remarques sur MARVEL

MARVEL ne propose pas à proprement parler de modèle pour la production de logiciels. En effet, toute l'approche du projet est basée sur la description "à plat" des activités en termes de préconditions et postconditions. Ces activités n'étant pas hiérarchisées comme dans ISTAR, il est impossible de décrire des activités de développement complexes telles que "conception détaillée" ou "codage" et les relations entre ces deux activités.

Par ailleurs, MARVEL ne considère dans son assistance que les postconditions décrites des activités. Il ne tient aucun compte des effets réels des outils associés aux activités. Si un outil fait plus que ce qui est décrit dans la postcondition, cet effet supplémentaire ne sera pas pris en compte pour la déduction par MARVEL.

Malgré les critiques que l'on peut faire de MARVEL, celui-ci présente l'intérêt d'être opérationnel et de servir à ses concepteurs de base d'expérimentation.

1.8 Conclusion

Nous avons essayé de tirer avantage des spécificités de chaque approche. La synthèse de toutes ces idées sera utilisée dans le MASP et peut être considérée comme le résultat de notre démarche. Nous présentons dans le chapitre suivant le modèle auquel nous avons abouti (MASP) en regard des modèles existant en vue de répondre aux objectifs du projet ALF.

2

Le modèle que nous proposons : le MASP

2.1 Introduction

Un produit logiciel est le résultat d'une somme d'activités (ou réalisations de tâches) qui nécessitent des compétences et des connaissances variées. Des efforts sont faits pour faire en sorte que ces connaissances soient utilisées pour fournir aux différents intervenants lors de la production de logiciels des mécanismes d'assistance [ALF87] [AMB88] [DER88] [KAI87] [KAI88]. Ces mécanismes d'assistance, fournis aussi bien aux gestionnaires de projets qu'aux développeurs, ont pour buts principaux le contrôle et l'assistance durant la production de logiciels. Garantir l'adéquation d'une activité, prendre l'initiative de la réalisation de certaines tâches pour décharger l'utilisateur de tâches fastidieuses sont des exemples d'utilisation de tels mécanismes d'assistance. Une première étape dans cette direction est de fournir une description de nos connaissances sur les activités de production de logiciels [OST87] [PER88] et sur les mécanismes à mettre en œuvre pour exploiter ces connaissances.

La description devra concerner :

- Le but de chaque activité qui est la réalisation d'une tâche précise,
- Les conditions requises pour le déroulement d'une activité,
- Les objets concernés par une activité (données et résultats d'une activité),
- Les propriétés des objets ou les contraintes que ceux-ci doivent respecter dans les différents stades du développement,
- La description de la réaction du système quand l'utilisateur essaye de déclencher une activité dans des circonstances inadéquates,
- La description de la réaction du système quand une activité se termine anormalement,

• ...

Nous appellerons cette description MASP [DER88] [DER89] [BEN89b] (MASP pour *Model for Assisted Software Process*). Un MASP décrit la façon dont les activités de production de logiciels (software process) doivent être exécutées et l'organisation des objets manipulés par ces activités. Ainsi, le MASP est une description statique du comportement attendu du système. Nous verrons par la suite que ce n'est pas forcément le comportement effectif.

2.2 Présentation des Composants du modèle

Un processus logiciel assisté est un ensemble d'activités respectant un certain nombre de contraintes. Les contraintes à respecter représentent l'expertise en développement de logiciel. Cette expertise est multiple :

- expertise fournie par la description des **opérateurs** (modèle d'opérateurs) et la description des **objets** sur lesquels ils sont appliqués (modèle d'objets),
- expertise sur **l'ordonnancement** des opérateurs,
- expertise sur le comportement imposé au système, dans certains états particuliers.

Les états particuliers du système sont identifiés et décrits par les **expressions**. Un sous-ensemble de ces expressions est appelé **caractéristique** et cette caractéristique doit être garantie aux points de contrôle du système. Ces points de contrôle sont l'activation, l'arrêt, la reprise d'exécution et la terminaison d'un opérateur (cf § 2.3.3).

Les caractéristiques définissent les états imposés du système.

Des réactions imposées du système sont décrites par des **règles** définissant ce que doit faire le système en certaines situations. Ce que doit faire le système est décrit par des opérations à exécuter, et les situations par des expressions. En d'autres termes, des expressions identifient des états du système dans lequel certaines opérations doivent être exécutées.

Ainsi nous proposons de modéliser le processus logiciel par un (MASP) -Model for Assisted Software Process- qui comprend six composants **(Om, Op, Ex, Ru, Or, Ch)**.

Nous essayerons de clarifier la description de la composante statique des composants en termes d'un méta-modèle E-A [CHE76] (Un travail similaire a été fait pour l'atelier de la SEMA METRA dans [TAR86]). Les entités sont représentées par des rectangles, les associations par des ellipses et les cardinalités montrent la contribution minimum et maximum des entités à une association.

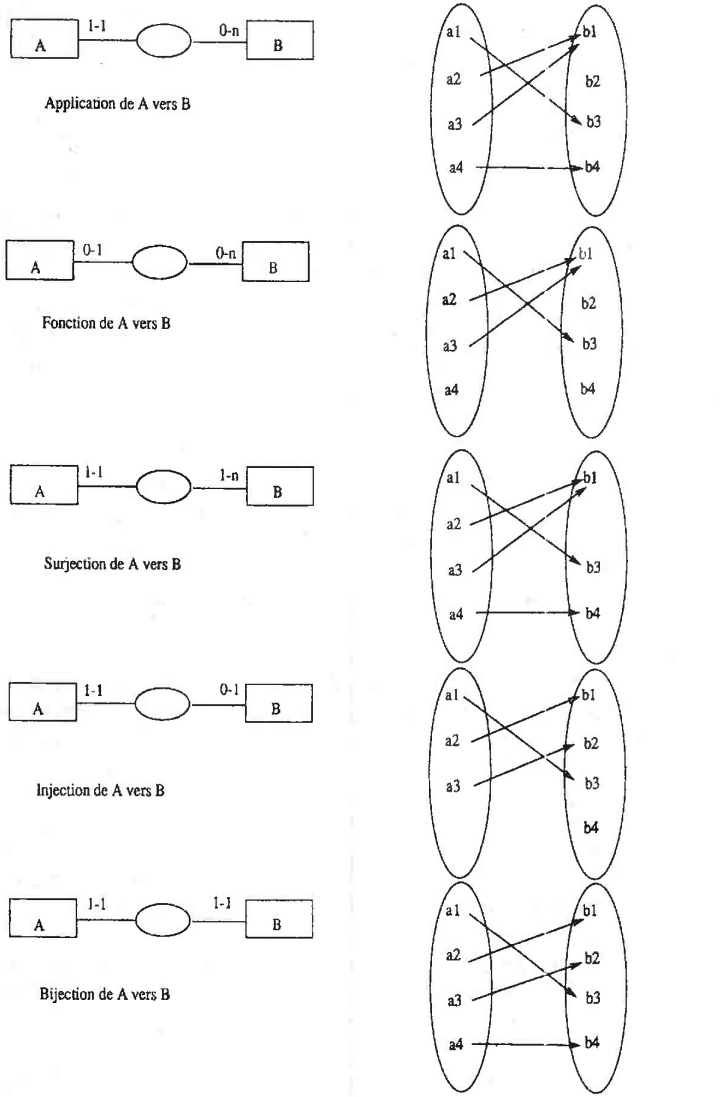


Figure 2.1. Signification ensembliste des schémas E-A utilisés

Les cardinalités maxima et minima sont celles utilisées dans la méthode Merise [TAR83] et nous donnons dans la figure 2.1 la signification des schémas E-A (ou plutôt Merise) et des cardinalités utilisées en termes ensemblistes.

- **Le modèle d'objets Om** définit la structure des composants logiciels en termes d'objets, de propriétés d'objets et de relations entre objets.
- **Le modèle d'opérateurs Op** décrit les types d'opérateurs qui peuvent être exécutés pour réaliser une activité. La description des types d'opérateurs comprend la condition -l'état des composants logiciels- sous laquelle un opérateur de ce type est exécutable, la condition déduite de l'application de l'opérateur -le nouvel état des composants logiciels-, les types d'objets sur lesquels peut être appliqué l'opérateur et les types d'objets changés ou produits par l'opérateur. Nous pouvons distinguer deux catégories de types d'opérateurs :
  - les types d'opérateurs simples : le modèle que nous proposons se limitera à une description externe de ceux-ci.
  - les types d'opérateurs complexes qui seront, eux, décrits en termes du modèle et raffinés jusqu'à leur description en termes d'opérateurs simples.
- **Les expressions Ex** décrivent des états particuliers du système.
- **L'ordonnancement Or** est un ordre total ou partiel sur les opérateurs. L'ordonnancement des opérateurs -ou des MASPs si ces opérateurs sont eux-mêmes des MASPs- est une façon d'exprimer explicitement des politiques d'application des opérateurs durant le développement de projet. **Or** permet l'ordonnancement d'opérateurs tels que le séquencement, la simultanéité, la répétition, etc...
- **Les règles Ru** -Ru pour rules- est un ensemble de règles qui associent une expression appelée partie condition à un opérateur appelée partie action. **Ru** décrit ce que le système doit faire -grâce à la partie action- quand une situation spécifiée est détectée -grâce à la partie condition-.
- **La caractéristique Ch** est un sous-ensemble de Ex; ce sous-ensemble décrit les états sains du logiciel en développement. ALF doit assurer que **Ch** est vérifiée aux points de contrôle du système.

Tous ces composants jouent des *rôles complémentaires* dans la description du processus de développement d'un logiciel.

2.3 Les composants du MASP

Bien que les différents participants au modèle soient de natures différentes, nous essayerons de suivre une approche homogène pour les décrire ainsi que leur rôle en nous basant sur un méta-modèle E-A.

Nous nous efforcerons tout au long de cette présentation de nous appuyer sur un exemple illustrant chacun des concepts introduits. Cet exemple se veut simple et n'est là que pour illustrer nos propos. La syntaxe utilisée est en outre éloignée de la syntaxe qui sera choisie en définitive dans le projet ALF.

L'exemple sera raffiné au fur et à mesure de l'introduction des concepts. L'exemple concerne la modélisation du processus d'organisation d'une conférence que nous appellerons O-ORGANISER_CONFERENCE. Cette modélisation concernera un certain nombre de types d'objets (lettre de convocation, participants,...) et des types d'opérateurs sur ces types d'objets (rédaction d'une lettre, envoi d'un courrier,...).

2.3.1 Le Modèle d'objets

Le modèle d'objets est le modèle le plus classique du MASP. En effet, même dans des projets comme ceux présentés (F1, PMDB, etc...), la nécessité d'un modèle d'objets a été ressentie et ces modèles sont tous plus ou moins proches du modèle E-A. Nous manipulerons un modèle E-A typé. Le typage sera défini de manière hiérarchique. Un type d'objet ou d'association pourra être sous-type d'un autre type de même nature.

En ce qui concerne les types d'objets, ils seront définis par la liste des types d'attributs (ou types de propriétés) spécifiques au type en cours de définition. Chaque type pourra être défini comme sous-type d'un type père et héritera des types d'attributs de son père. Nous manipulerons un héritage simple dans lequel un type est sous-type d'un seul autre type.

Les types d'associations seront définis, eux aussi, de manière hiérarchique. Un type d'association est défini par la définition de ses rôles, c'est-à-dire des types d'objets intervenant dans l'association en cours de définition, et par la liste de ses types d'attributs spécifiques. Nous nous limiterons dans le modèle décrit aux types d'associations binaires. Nous définissons les types d'associations binaires comme faisant intervenir deux types d'objets seulement. Un cas particulier de ces associations binaires est celui des associations faisant intervenir un seul type d'objet jouant deux rôles différents. Par exemple l'association COUPLE mettrait en jeu le même type d'objet PERSONNE avec le rôle *mari* et le rôle *femme*. Une facilité de sous-typage est offerte. Un type d'association peut être défini comme sous-type d'un autre. Le sous-type hérite alors des rôles et des types d'attributs de son père. Ce sous-type peut être alors étendu par de nouveaux types

d'attributs spécifiques mais pas par de nouveaux rôles.

Le modèle d'objets : Le modèle d'objets est composé de types d'objets, de types d'associations entre objets et de leurs types d'attributs. Ces types d'attributs prennent leurs valeurs dans les domaines prédéfinis *entier, réel, chaîne et date* et sont des propriétés des types d'objets ou des types d'associations. Les types d'associations relient entre eux deux types d'objets (ou le même type avec deux rôles différents). Un type d'objets peut être un sous-type d'un type d'objet père, hériter des types d'attributs de celui-ci et avoir de nouveaux types d'attributs spécifiques. Un type d'association peut être un sous-type d'un type d'association père, hériter des rôles et des types d'attributs de celui-ci et avoir de nouveaux types d'attributs spécifiques. Un type d'association sous-type ne peut se définir de nouveaux rôles.

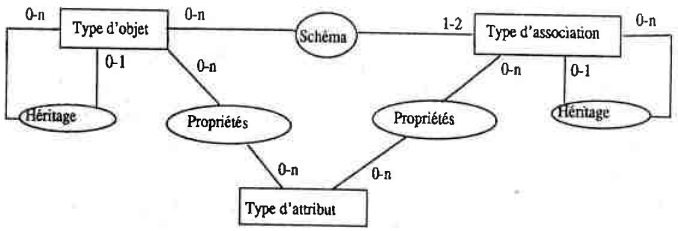


Figure 2.2. Méta-modèle du Modèle d'Objets

En ce qui concerne les cardinalités pour la relation d'héritage, la cardinalité 0-1 signifie qu'un type d'objet (ou type d'association) est sous-type de 0 ou 1 type d'objet (ou type d'association) père; la cardinalité 0-n signifie qu'un type d'objet (ou type d'association) peut être père de 0 ou plusieurs sous-types.

Nous utiliserons pour la description du modèle d'objet la grammaire décrite ci-dessous sous forme BNF.


```
om
  ::= 'MODELE D'OBJETS' modèle_d'objets

modèle_d'objets
  ::= type_d'objet modèle_d'objets
    | type_d'association modèle_d'objets
    |^

type_d'objet
  ::= identificateur 'sous type de' identificateur
      liste_de_propriétés ';'
    |  identificateur  liste_de_propriétés ';'

type_d'association
  ::= identificateur 'sous type de' identificateur
      liste_de_propriétés ';'
    |  identificateur ',' liste_de_rôles  liste_de_propriétés ';'

liste_de_rôles
  ::= identificateur ':' nom_de_type_d'objet ','
      identificateur ':' nom_de_type_d'objet

liste_de_propriétés
  ::= ',' identificateur ':' domaine  liste_de_propriétés
    | ^

domaine
  ::= 'entier' | 'reel' | 'chaine' | 'date'
```

Afin de faciliter la lecture de notre exemple, les identificateurs des types d'associations commenceront systématiquement par "R-". Le modèle d'objets de notre exemple comprend :

- un type d'objet PERSONNE représentant les caractéristiques communes à toute personne,
- un sous-type de PERSONNE décrivant les ORGANISATEURs,
- un sous-type de PERSONNE décrivant les PARTICIPANTs-POTENTIELs à un meeting,
- un type MEETING décrivant les meetings par leur thème général et la date de leur tenue,
- un type d'association R-RESPONSABLE dont les instances indiqueront l'ORGANISATEUR responsable d'un MEETING,
- un type d'association R-LETTRE dont les instances indiqueront l'ORGANISATEUR rédacteur de la lettre d'invitation (contenu) d'un MEETING.

- un type d'association R-MAIL dont les instances indiqueront les PARTICIPANTs-POTENTIELs auxquels un ORGANISATEUR a envoyé une lettre d'invitation,

Ce modèle d'objet est décrit ci-dessous en utilisant la grammaire présentée et dans la figure 2.3 sous forme d'un schéma E-A en utilisant l'association ISA pour décrire les liens entre un type d'objet et ses sous-types.

```
MODELE D'OBJETS

PERSONNE ,
    nom : chaine ;

ORGANISATEUR sous type de PERSONNE ,
  rôle_dans_l'organisation_du_meeting : chaine ;

PARTICIPANT-POTENTIEL sous type de PERSONNE ,
  adresse : chaine ;

MEETING ,
  nom : chaine  ,
  thème_général : chaine ,
  date_début : date ,
  date_fin : date ;

R-RESPONSABLE ,
  responsable : ORGANISATEUR ,
  meeting : MEETING ;

R-LETTRE ,
 rédacteur : ORGANISATEUR ,
 meeting : MEETING ,
 contenu : chaine ;

R-MAIL ,
 émetteur : ORGANISATEUR ,
 récepteur : PARTICIPANT_POTENTIEL ,
 date : date ;
```

Le modèle d'objet permet la description des "objets" intervenant dans la production de logiciel. Cette description permet d'avoir, à tout moment, une vue sur l'état courant de ces "objets". La structuration de cette description statique se fait par la définition des différents types d'objets, d'attributs et d'associations.

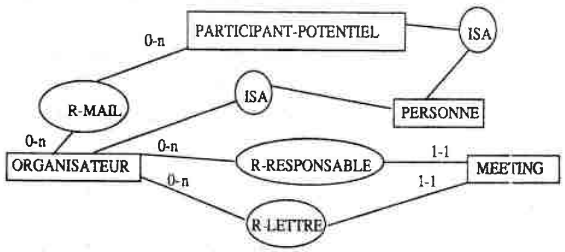


Figure 2.3. Modèle d'Objet du MASP O-ORGANISER_CONFERENCE

2.3.2 Le Modèle d'opérateurs

Le modèle d'opérateurs décrit les opérateurs intervenant dans la production de logiciels. L'approche choisie pour cette description est une approche externe. Nous ne nous occupons pas d'écrire ou de réaliser des opérateurs triviaux, mais uniquement de modéliser leur comportement et de décrire les opérateurs complexes.

Le modèle d'opérateurs : Les types d'opérateurs décrivent des activités réalisées par des opérateurs de ce type. Un type d'opérateur est décrit par :

IN : Les types des objets et les types des associations qu'il utilise en paramètres d'entrée. Un opérateur ne modifie pas les objets et les associations qu'il utilise en entrée. Ces types d'objets et d'associations appartiennent au modèle d'objets.

INOUT : Les types d'objets et les types d'associations qu'il utilise en entrée et qu'il peut éventuellement modifier. Ces types d'objets et d'associations appartiennent au modèle d'objets.

OUT : Les types d'objets et les types d'association qu'il crée en sortie. Ces types d'objets et d'associations appartiennent au modèle d'objets du processus de production.

PRE : Une expression logique appelée précondition décrivant une condition nécessaire à l'activation d'un opérateur du type décrit lorsqu'un tel opérateur est invoqué.

POST : Une expression logique appelée postcondition décrivant le résultat attendu, l'effet escompté d'un tel opérateur. Tout opérateur de ce type devra produire un résultat vérifiant cette postcondition.

SEMANTIQUE : Un opérateur du type en cours de description utilise les objets passés en paramètres d'entrée (in et inout) pour produire les objets passés en paramètres de sortie (inout et out). A la fin de l'exécution d'un tel opérateur, sa postcondition est vérifiée. Nous venons de décrire en quelque sorte ce que réalisait un opérateur mais nous n'avons pas encore dit comment cet opérateur réalisait cette tâche. La sémantique de l'opérateur joue ce rôle. Deux cas distincts sont à considérer :

- l'opérateur spécifié est simple, la sémantique de l'opérateur sera alors réalisée par un outil. L'opérateur est alors lié à cet outil. Ces types d'opérateurs terminaux devront être indifféremment définis en termes de langages procéduraux ou déclaratifs ou être associés à un outil existant. Cependant nous ne nous intéressons pas aux outils en tant que tels. La modélisation s'arrête au niveau descriptif. La réalisation même de l'outil ou la réutilisation d'un outil existant n'intéresse pas notre propos. Notre niveau de granularité est au niveau de l'outil et non pas au niveau des instructions d'un langage donné.

- l'opérateur spécifié est trop complexe et il n'existe pas d'outil le réalisant. Cet opérateur est alors décrit lui-même en termes de MASP. Il sera donc composé par un modèle d'objets, un modèle d'opérateurs, un ensemble d'expressions, un ensemble de règles, un ordonnancement et une caractéristique. Cette façon de spécifier un opérateur en termes de MASP permet une description hiérarchique du processus de production. Cette démarche s'apparente à la méthode de spécification par raffinement successif ou top-down qui permet la décomposition du problème complexe à traiter en une suite d'ensembles de problèmes de plus en plus simples et donc plus facilement appréhendables par le spécifieur. Cette hiérarchisation de la description des opérateurs ne peut se faire indépendamment des autres composants du modèle. Un opérateur porte sur un ensemble de types d'associations et de type d'objets. Cet ensemble définit le plus petit modèle d'objets de cet opérateur. Le modèle d'objet final de cet opérateur est alors une extension de ce modèle d'objets. Le modèle d'opérateurs de cet opérateur est, lui, composé d'au moins un opérateur de fin que nous appellerons O-FIN. Cet opérateur O-FIN existe pour chaque niveau du modèle et permet de terminer tout opérateur complexe. Son profil sera le même pour tous les types d'opérateurs. Nous reviendrons, par la suite, sur les autres composants du modèle.

Le modèle d'opérateurs : Le modèle d'opérateurs décrit les types d'opérateurs manipulés lors de la production de logiciels. Cette description est composée

- des paramètres en entrée d'un type d'opérateur (in et inout) des paramètres en sortie d'un type d'opérateur (inout et out),
- de la précondition nécessaire à l'activation d'un opérateur de ce type,
- de l'effet de l'exécution de l'opération (postcondition).

Cette description du modèle d'opérateur est par ailleurs hiérarchique, chaque type d'opérateur pouvant lui-même être décrit en termes de MASP.

La description réalisée dans la figure 2.4 est une description statique du modèle d'opérateurs. Nous avons utilisé les associations ISA, AGREGATION et GROUPEMENT qui étendent le modèle de CHEN pour une meilleure capture et une meilleure représentation de la sémantique décrite par le modèle E-A [SMI82]. Nous avons appelé ISA la relation de généralisation de Smith & Smith, nous avons gardé le nom AGREGATION et nous utilisons l'association GROUPEMENT pour dénoter qu'une entité est soit d'un type, soit d'un autre.

La grammaire du modèle d'opérateurs est décrite ci-dessous.

```
op
  ::= 'MODELE D'OPERATEURS'  modèle_d'opérateurs

modèle_d'opérateurs
  ::= type_d'opérateur ';' modèle_d'opérateurs | ^

type_d'opérateur
  ::= identificateur
      'in' ':' liste_de_paramètres ';'
      'inout' ':' liste_de_paramètres ';'
      'out' ':' liste_de_paramètres ';'
      'pré' ':' expression_logique ';'
      'post' ':' expression_logique ';'
      'sémantique' ':' description ';'

liste_de_paramètres
  ::=   identificateur suite_paramètres | ^

suite_paramètres
  ::= ',' liste_de_paramètres

description
  ::= 'par' 'MASP' identificateur  | 'par' 'OUTIL' identificateur
```

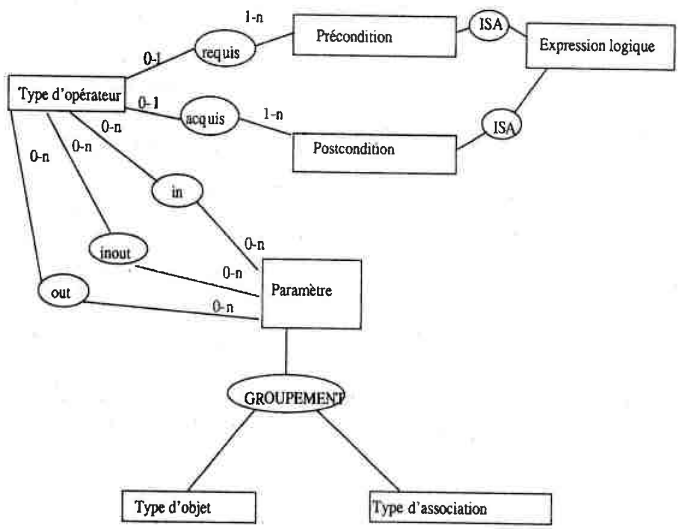


Figure 2.4. Méta-modèle du Modèle d'Opérateurs

Toujours pour faciliter la lecture de notre exemple, les identificateurs des types d'opérateurs commenceront systématiquement par "O-". Si nous utilisons un pseudo-langage naturel pour décrire de manière intuitive les expressions logiques, le modèle d'opérateurs de notre exemple est composé de :

- O-CREER_MEETING qui consiste à choisir le thème général de la conférence. Ce thème général doit être choisi parmi une liste bien déterminée de thèmes (postcondition), par un ORGANISATEUR responsable (précondition).
- O-FAIRE_R-LETTRE permet à un secrétaire parmi les ORGANISATEURs de faire une lettre de présentation du meeting en utilisant les informations contenues dans MEETING, c'est-à-dire de créer une relation de type R-LETTRE entre l'instance d'ORGANISATEUR le représentant et le MEETING concerné.
- O-FAIRE_R-MAIL consiste pour un secrétaire à envoyer une lettre d'invitation à un PARTICIPANT-POTENTIEL.


```
MODELE D'OPERATEURS

O-FIN
 in :  ;
 inout : ;
 out:  ;
 pré: ;
 post: ;
 sémantique : par OUTIL $fin ;

O-CREER_MEETING
 in : ORGANISATEUR ;
 inout : ;
 out: MEETING, R-RESPONSABLE ;
 pré : Le rôle_dans_l'organisation_du_meeting de l'ORGANISATEUR
est responsable ;
 post: thème_général (MEETING) ∈ (Systèmes d'informations,
Génie Logiciel, Ateliers de Génie Logiciel) ;
 sémantique : par MASP O-CREER_MEETING ;

O-FAIRE_R-LETTRE
 in : ORGANISATEUR, MEETING ;
 inout : ;
 out: R-LETTRE ;
 pré : Le rôle_dans_l'organisation_du_meeting de l'ORGANISATEUR
est secrétaire ;
 post: ;
 sémantique : par MASP O-FAIRE_R-LETTRE ;

O-FAIRE_R-MAIL
 in : ORGANISATEUR, R-LETTRE, PARTICIPANT-POTENTIEL ;
 inout : ;
 out: R-MAIL ;
 pré: Le rôle_dans_l'organisation_du_meeting de l'ORGANISATEUR
est secrétaire ;
 post: ;
 sémantique : par MASP O-FAIRE_R-MAIL  ;
```

Certains des types d'opérateurs apparaissant à ce niveau sont complexes. La sémantique de ceux-ci peut être donnée par la description des types d'objets intervenant dans leur réalisation, des types d'opérateurs dont la combinaison peut réaliser cette sémantique, de l'ordonnancement de ces types d'opérateurs, des contraintes de développement à respecter,... Autrement dit, cette sémantique peut être décrite en termes de MASP.

Continuons le développement de l'exemple sur O-CREER_MEETING en décrivant sa sémantique en termes de MASP.

Le modèle d'objets de ce MASP-fils (O-CREER_MEETING) correspond à une extension du modèle d'objets du MASP père O-ORGANISER_CONFERENCE. Cette extension du modèle d'objets, par analogie avec les langages de programmation, peut être vue comme la définition des variables locales de cet opérateur. Les premiers éléments du modèle d'objets de ce MASP fils sont les types d'objets et d'associations passés en paramètres à l'opérateur. Nous avons donc pour ce MASP O-CREER_MEETING les types d'objets MEETING, ORGANISATEUR et le type d'association R-RESPONSABLE entre eux qui sont donc déjà définis.
Nous étendons ce modèle d'objets avec :

- un type d'objet THEME qui représente l'ensemble des thèmes pouvant être abordés lors d'une conférence,
- un type d'objet ADRESSE recensant les différentes localisations possibles pour une conférence,
- le type d'association R-ADRESSE-MEETING décrivant la localisation d'une conférence,
- le type d'association R-AGENDA-MEETING décrivant les thèmes abordés à une conférence et la date à laquelle ils sont abordés.

```
MODELE D'OBJETS

THEME ,
   sujet : chaine ;

ADRESSE ,
  societe  : chaine ,
  ville    : chaine ,
  adresse : chaine ,
  salle_de_réunion  : entier ;

R-ADRESSE_MEETING ,
  adresse : ADRESSE ,
  meeting : MEETING ;

R-AGENDA_MEETING ,
  thème : THEME  ,
  meeting : MEETING ,
  date : date ;
```

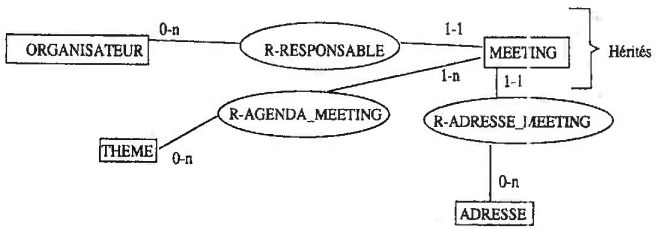



Figure 2.5. Modèle d'Objets du MASP O-CREER_MEETING

Les opérateurs décomposant l'opérateur O-CREER_MEETING sont :

- O-FIN qui clot l'exécution de O-CREER_MEETING,
- O-COMPLETER_AGENDA qui consiste à choisir un thème particulier parmi l'ensemble des thèmes disponibles et à décider d'une date à laquelle ce thème sera traité. Ceci sera traduit par la création d'une association du type R-AGENDA_MEETING entre le thème choisi et le meeting concerné, cette association aura en outre un attribut indiquant la date à laquelle le thème sera traité.
- O-CHOISIR_DATE modifie une instance du type MEETING en mettant à jour ses attributs date_début et date_fin.
- O-CHOISIR_ADRESSE consiste à choisir l'adresse où aura lieu la conférence avec comme contrainte exprimée par la postcondition que la ville choisie soit PARIS, LONDRES, BONN, MADRID, ou BERN.

```
MODELE D'OPERATEURS

O-FIN
 in :  ;
 inout : ;
 out:  ;
 pré: ;
 post: ;
 sémantique : par OUTIL $fin ;

O-COMPLETER_AGENDA
 in : ORGANISATEUR , MEETING, liste de THEMEs ;
 inout : ;
 out: R-AGENDA-MEETING ;
 pré:  Le THEME n'est pas déjà traité (c'est-à-dire qu'il n'existe
pas déjà d'association R-AGENDA_MEETING entre ce THEME et ce MEETING) ;
 post: date(R-AGENDA_MEETING) ∈ [date_début (MEETING), date_fin (MEETING)] ;
 sémantique : par MASP O-COMPLETER_AGENDA ;

O-CHOSIR_DATE
 in : ORGANISATEUR ;
 inout: MEETING ;
 out :  ;
 pré:  ;
 post:  ;
 sémantique : par OUTIL $date_opérateur ;

O-CHOISIR_ADRESSE
 in : ORGANISATEUR,  MEETING, liste de ADRESSEs ;
 inout : ;
 out: R-ADRESSE_MEETING ;
 pré:  ;
 post: R-ADRESSE_MEETING (adresse, meeting)
       et ville (adresse) ∈ (PARIS, LONDRES, BONN, MADRID, BERN) ;
 sémantique : par OUTIL $adresse_opérateur ;
```

La définition du modèle d'objets et du modèle d'opérateurs est la première étape dans la description d'un MASP. Cette description est d'ailleurs la plus classique. Même si cette description n'est pas toujours explicite, tout atelier ou système d'exploitation met à la disposition des utilisateurs le type fichier et un ensemble de primitives à appliquer à des instances du type fichier. Cela représente, en quelque sorte, un mini-modèle d'objets et un mini-modèle d'opérateurs du système d'exploitation. Cette première étape permet la modélisation des différents "objets" du système et des différentes opérations que l'utilisateur ou le système peut réaliser.

2.3.3 Les Expressions

Comme nous l'avons évoqué auparavant, les expressions décrivent des états (ou des transitions d'états) particuliers du système. La première étape pour déterminer les résultats attendus du langage de description des expressions est d'identifier les différents éléments qui fournissent des informations sur l'état du système.

Tout système peut être perçu de deux façons différentes : une perception statique ou une perception dynamique.

Dans la perception statique, le système est décrit par l'état de ses composants. Ces composants sont décrits dans le modèle d'objets. Les expressions devront inclure la désignation et la manipulation des instances des types d'objets et des types d'associations. Cette désignation et manipulation se fera à travers une expression logique.

Dans la perception dynamique, le système est perçu à travers son comportement. Ce comportement peut lui-même être perçu à travers les événements survenant dans le système. Donc les expressions devront inclure la désignation et la manipulation d'événements.

Les concepts de base dans la description des états du système sont donc les événements et les composants du modèle d'objets. Nous considérerons donc une expression, dont le méta-modèle est décrit dans la figure 2.6 comme formée de deux parties :

- une partie événement
- une partie logique

Les deux cardinalités 0-1 du schéma 2.6 expriment le fait qu'une expression peut avoir ou non une partie logique, et peut avoir ou non une partie événement. Cependant nous ne pouvons exprimer comme nous aurions aimé le faire, grâce aux cardinalités, le fait qu'une expression a forcément une de ses deux parties non vide.

Nous dirons qu'une expression est dite vérifiée dans les cas suivants :

- sa partie événement est vide, et sa partie logique n'est pas vide et celle-ci prend la valeur vrai,
- sa partie événement n'est pas vide et l'événement décrit est survenu, et sa partie logique, qui n'est pas vide, prend la valeur vrai,
- sa partie événement n'est pas vide, et l'événement décrit est survenu, et sa partie logique est vide.

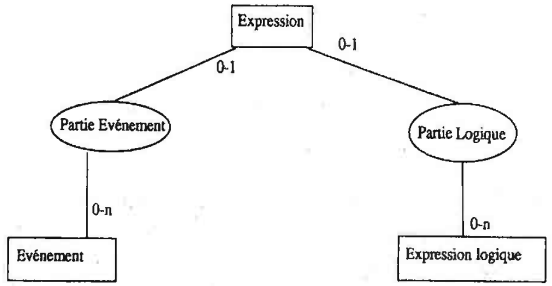


Figure 2.6. Méta-modèle des expressions

Nous devons être capable d'exprimer les états du système d'une manière "formelle" compréhensible et facilement utilisable par les concepteurs modélisant le processus de production. Cependant pour l'exemple en cours de développement, nous utiliserons le langage naturel pour la description d'expressions.

Les événements sur opérateurs peuvent être déduits des concepts des systèmes d'exploitation [CRO75] [KRA85]. Classiquement les événements de base concernant un opérateur sont : l'activation, l'arrêt , la reprise d'exécution et la terminaison.

- L'événement d'activation est le résultat de la transformation d'un opérateur en une opération.
- L'événement d'arrêt est le résultat de la mise en attente d'un opérateur actif (opération).
- La reprise d'exécution redémarre une opération arrêtée.
- La terminaison achève toute opération.

Les événements sur objets peuvent être déduits des concepts de base de données [DAT83]. Ils sont de deux types : événements sur des objets en tant que tout, et événements sur leurs attributs. Les événements sur attributs sont de deux types : consultation, et mise à jour.

- L'événement de consultation apparaît à chaque fois qu'un attribut est accédé sans être modifié.
- L'événement de mise à jour comprend l'accès et la modification d'un attribut.

Les événements sur objets sont de quatre types : création, consultation, mise à jour, et suppression.

- L'événement de création apparaît, une et une seule fois, pour chaque objet dans la base d'objets. C'est le résultat de la première opération sur un objet.
- L'événement de consultation apparaît à chaque fois qu'un objet est accédé sans être modifié.
- L'événement de mise à jour comprend l'accès et la modification d'un objet.
- L'événement de suppression apparaît, une et une seule fois, pour chaque objet dans la base d'objets. C'est le résultat de la dernière opération possible sur un objet.

Il est clair que la consultation et la mise à jour d'un objet sont seulement une vue globale de la consultation et de la mise à jour de ses attributs. Les expressions décrites ci-dessous ne sont pas des composants du modèle au même titre que les autres composants. Elles sont utilisées par les autres composants pour décrire des états du système.

Une **expression** est composée d'une partie événement et d'une partie logique. Une expression décrit un état du système et est utilisée par les autres composants du modèle lorsque ceux-ci ont besoin de décrire un état spécifique du système.

Nous avons vu que les opérateurs ont une précondition et une postcondition. Celles-ci sont des expressions sans partie événement, c'est-à-dire des expressions logiques. Nous verrons, par la suite, comment les autres composants du modèle utilisent les expressions. Un tel exemple d'utilisation est l'utilisation d'une expression comme partie gauche d'une règle (cf § 2.3.4).

Décrites sous forme BNF, les expressions sont composées d'une partie événement et d'une partie logique séparées par une ',' :

```
expression
 ::= partie_événement ',' partie_logique
   | partie_événement
   | partie_logique

partie_événement
  ::= événement_sur_objet
    | événement_sur_opérateur

événement_sur_objet
  ::= type_d'événement_sur_objet '(' identificateur ')'

type_d'événement_sur_objet
  ::= 'création' | 'consultation' | 'mise_à_jour' | 'suppression'

événement_sur_opérateur
  ::= type_d'événement_sur_opérateur '(' identificateur ')'

type_d'événement_sur_opérateur
  ::= 'activation' | 'terminaison' | 'suspension' | 'reprise'

partie_logique
  ::= expression_logique
```

Dans notre exemple, *"terminaison (O-FAIRE_R-LETTRE)"* est l'expression composée uniquement d'une partie événement caractérisant la fin de l'opération O-FAIRE_R-LETTRE.

2.3.4 Les règles

La modélisation de la production de logiciels en termes de modèle d'objets et de modèle d'opérateurs est, à notre avis, la pierre de base de toute modélisation de la production de logiciels. C'est en ce sens que les systèmes d'informations pour le génie logiciel s'inscrivent dans le cadre de cette modélisation. Ils sont, non seulement, le noyau central des ateliers de deuxième génération, mais aussi la base de départ des ateliers de troisième génération. Cependant la modélisation en termes de modèle d'objets et de modèle d'opérateurs est loin d'être satisfaisante ou complète. Cette modélisation permet la prise en compte de règles de production de logiciels[1] telles que *"Il est interdit à un responsable de faire une lettre"* grâce, par exemple, à la précondition de O-FAIRE_LETTRE. Cependant une telle modélisation ne permet pas d'exprimer des règles de production de logiciel telles que *"Dès qu'une lettre est faite, envoyer un premier courrier"*. De telles règles ne peuvent donc pas être prises en compte par le modèle réduit au modèle d'objets et au modèle d'opérateurs.

1. N.B: règle est ici prise dans son sens commun, c'est-à-dire règle de travail ou règle à respecter.

Nous avons vu, dans le § 2.3.3, que nous pouvions exprimer la première partie de cette règle de production de logiciels grâce à *"terminaison (O-FAIRE_R-LETTRE)"*. Par ailleurs, nous avons déjà modélisé l'opérateur O-FAIRE_R-MAIL réalisant l'envoi d'un courrier dans le § 2.3.2. Il reste donc à faire le lien entre les deux. C'est le rôle des règles dans notre "Model for Asssisted Software Process". Le méta-modèle des règles est donné dans la figure 2.7.

Figure 2.7. Méta-modèle des règles

Une **règle** associe une expression à un opérateur. Cet opérateur spécifie l'opération que le système doit déclencher lorsque survient la situation décrite par l'expression.

Nous voyons apparaître le système dans un rôle actif. C'est le système qui prend l'initiative de déclencher une opération. L'utilisateur est assisté dans sa tâche par le système qui le décharge d'une partie de son travail. L'utilisateur n'a plus besoin de se rappeler de la règle *"Dès qu'une lettre est faite, envoyer un premier courrier"* : le système s'en charge pour lui et fait en sorte de déclencher l'opération au moment voulu. Un exemple d'utilisation simple des règles pourrait être :
dans UNIX le fait de créer un fichier initialise la date_de_création de ce fichier; cette caractéristique pourrait être représentée dans notre modèle par une règle :
SI création(FICHIER) ALORS O-METTRE_à_JOUR (FICHIER,date_de_création).

Nous utiliserons les notations suivantes pour décrire les règles :

```
ru
  ::= 'REGLES'  règles

règles
  ::= règle ';' règles
    |^

règle
  ::=  'SI' expression 'ALORS' type_d'opérateur
```

Pour le MASP du premier niveau de notre exemple (O-ORGANISER_CONFERENCE) les règles sont :

- SI un meeting est créé ALORS faire la lettre de présentation de celui-ci,

- SI la réalisation d'une lettre de présentation du meeting est terminée ALORS faire un courrier,

Les règles sont alors écrites en utilisant notre grammaire de la manière suivante :

```
REGLES
SI terminaison (O-CREER_MEETING) ALORS O-FAIRE_R-LETTRE  ;
SI terminaison (O-FAIRE_R-LETTRE) ALORS O-FAIRE_R-MAIL ;
```

2.3.5 Ordonnancement des opérateurs

Une certaine expertise peut être décrite en termes d'ordonnancement d'opérateurs : tout développeur sait qu'avant de faire l'édition de liens et le chargement d'un exécutable, il faut compiler les différents modules intervenant dans le programme. Cette expertise existe mais elle est trop souvent stockée dans la tête de l'utilisateur et il peut très bien ne pas appliquer ces contraintes. La modélisation du processus de développement gagnera donc à la représentation de cette expertise. C'est le rôle de l'ordonnancement des opérateurs dans le MASP dont le méta-modèle est donné par la figure 2.8. La relation d'ordre entre deux types d'opérateurs exprime le fait qu'ils interviennent tous les deux dans l'ordonnancement. Cette relation d'ordre exprime une contrainte temporelle de succession, de concurrence,... Les types d'opérateurs intervenant dans l'ordonnancement sont ceux pour lesquels existe une contrainte d'ordre. Les autres peuvent être activés à n'importe quel moment, mais toujours en respectant les autres contraintes imposées par le modèle (par exemple, la précondition doit être vérifiée, en tout état de cause, avant l'activation d'un opérateur).

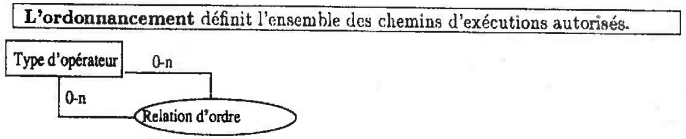


Figure 2.8. Méta-modèle de l'ordonnancement

Cependant il ne faut pas perdre de vue que nous sommes au niveau du modèle de processus et non pas au niveau des processus eux-mêmes : il convient de bien différencier l'ordonnancement effectif de l'exécution des opérateurs et l'ensemble de

tous les ordonnancements possibles fournis par **Or**. A un ordonnancement **Or** correspondent plusieurs ordonnancements au niveau de l'exécution (ASP pour Assisted Software Process). L'ordonnancement décrit au niveau du MASP doit être le moins restrictif possible. Par exemple, il est raisonnable de dire que l'exécution d'un opérateur peut être répétée. Ceci n'implique pas de contraintes quant au nombre exact et effectif de répétitions de l'opérateur. Cela permet un degré de flexibilité dans le développement d'un projet. L'ordonnancement au niveau du MASP décrit un ensemble d'ordonnancements concrets. Il y a différentes sortes d'opérateurs d'ordonnancements :

- séquencement
- simultanéité
- concurrence
- alternative (non déterministe)
- option
- répétition
- groupement

Nous utiliserons les notations suivantes pour l'ordonnancement :

```
or
  ::= 'ORDRE' ordre

ordre
  ::= identificateur
    | ordre ';' ordre          /* pour exécution séquentielle  */
    |  '{' ordre '}'           /* pour exécution simultanée  */
    | ordre '||' ordre         /* pour exécution concurrente  */
    | ordre '|' ordre          /* pour une alternative d'exécution  */
    |  '(*' ordre '*)'         /* pour exécution optionnelle  */
    |  '[' ordre ']'           /* pour exécution répétée */
    | '(' ordre ')'            /* pour groupement d'opérateurs  */
```

Le séquencement

Le séquencement de deux opérateurs A et B (exprimé par A ; B) décrit une contrainte de succession entre ces deux opérateurs. Cela permet de définir des séquences obligatoires d'exécutions d'opérateurs. Une contrainte sous Unix sur SUN comme *"l'outil d'édition graphique touchup ne tourne que sous l'environnement graphique suntools (lancé par suntools)"* se traduit aisément dans notre modèle par : suntools ; touchup.

La simultanéité

La simultanéité pour un opérateur A (exprimée par { A }) fait état de la possibilité pour un opérateur A d'être exécuté plus d'une fois simultanément. { sh } traduit la possibilité sous Unix, de lancer simultanément plusieurs *"shells"* en même temps.

La concurrence

La concurrence entre deux opérateurs A et B (exprimée par $A||B$) décrit la possibilité pour deux opérateurs différents d'être exécutés concurremment. Le fait de pouvoir sous Unix lancer à la fois un *"bourne shell"* -sh- et un *"C shell"* -csh- se traduit par : $sh||csh$.

L'alternative

L'alternative entre deux opérateurs A et B (exprimée par $A \mid B$) propose à l'utilisateur le choix entre l'exécution de A ou de B. L'utilisateur d'un SUN a le choix entre l'environnement *xwindows* d'une part et l'environnement *suntools* d'autre part, et un tel choix (non déterministe) sera traduit par : $xwindows \mid suntools$.

L'option

L'option permet de proposer à l'utilisateur un opérateur potentiel mais pas obligatoire. En C, il est d'usage de faire une vérification sémantique des programmes plus précise, grâce à *lint*, que celle effectuée par le compilateur *cc*, cependant rien n'oblige à effectuer cette vérification supplémentaire. Nous traduirions cette contrainte par : cc ; (*lint*).

La répétition

La répétition permet l'itération d'un opérateur. [A] exprime que A pourra être exécuté plusieurs fois de suite. Par exemple, l'utilisateur pourra faire n opérations d'éditions successives : [emacs].

Le groupement

Le groupement permet de manipuler un ordonnancement d'opérateurs comme un opérateur unique. Il permet en particulier de définir les priorités des différents autres opérateurs intervenant dans l'ordonnancement.
(cc ; lint) | emacs qui exprime l'alternative pour l'utilisateur d'une séquence cc ; lint et d'une édition -emacs- est différent de
cc ;(lint | emacs) qui exprime l'alternative pour l'utilisateur d une compilation -cc- et d'une séquence lint ; emacs.

L'ordonnancement de notre exemple est alors décrit de la manière suivante :
Pour le MASP du premier niveau, O-ORGANISER_CONFERENCE :

```
ORDRE
O-CREER_MEETING ;  O-FAIRE_R-LETTRE ; [O-FAIRE_R-MAIL] ; O-FIN
```

Pour le MASP fils O-CREER_MEETING :

```
ORDRE
((O-CHOISIR_DATE ; O-CHOISIR_ADRESSE) | (O-CHOISIR_ADRESSE; O-CHOISIR_DATE)) ;
    [O-COMPLETER_AGENDA]  ; O-FIN
```

Un tel ordonnancement n'impose pas une seule façon d'exécuter les opérations associées aux types d'opérateurs concernés, mais décrit l'ensemble de tous les schémas d'exécution. Il est, par exemple, autorisé de débuter avec O-CHOISIR_ADRESSE ou bien avec O-CHOISIR_DATE.

2.3.6 La caractéristique

La caractéristique est un sous-ensemble des expressions (cf. figure 2.9), elle ne nécessite donc pas de formalisme nouveau.

Figure 2.9. Méta-modèle de la caractéristique

La caractéristique est une conjonction de toutes les expressions la composant. Le rôle de la caractéristique est toutefois très différent de celui des expressions : si les expressions ne sont qu'un moyen commun que partagent les règles et la caractéristique, cette dernière a pour rôle de décrire des états "corrects" du processus. Le système ALF doit garantir l'exactitude des expressions appartenant à la caractéristique. Cette garantie peut être assurée de différentes façons :

- considérer la caractéristique comme un ensemble de contraintes d'intégrité et refuser toute opération violant ces contraintes,
- considérer la caractéristique comme représentant les "états à atteindre" et utiliser les autres composants décrivant le modèle pour atteindre cet état.

La caractéristique caractérise l'ensemble des états stables par lesquels le processus de production peut passer.

Continuant à utiliser le langage naturel pour les expressions, la caractéristique de notre exemple pourrait être :

Pour le MASP au premier niveau, O-ORGANISER_CONFERENCE :

```
CARACTERISTIQUE
 Il y a un seul responsable dans le groupe des ORGANISATEURs. ;
```

Pour le MASP fils O-CREER_MEETING :

```
CARACTERISTIQUE
 La date_début d'un MEETING n'est jamais un dimanche.
```

2.4 Interaction entre les composants

Durant le développement du logiciel, des objets sont créés, mis à jour et détruits. L'état du processus de production est défini par l'ensemble des objets et leurs propriétés à un moment donné. Cet état doit toujours respecter un certain nombre de contraintes : elles sont exprimées par la structure des objets et par la caractéristique.

Les contraintes que doivent respecter les objets manipulés lors de la production de logiciels sont exprimées à travers le modèle d'objet et la caractéristique.

Le comportement attendu d'un projet en cours de développement est assuré par les opérateurs, les règles et l'ordonnancement décrits dans le MASP. Nous allons maintenant décrire l'interaction entre ces différents composants pendant le processus de production.

Tout d'abord, les opérateurs sont les moyens de réaliser des activités. L'invocation d'un opérateur est le seul moyen de passer d'un état à un autre. Cette invocation est soit du fait de l'utilisateur, soit du fait du système.

Un opérateur est **activé** après son **invocation** si et seulement si :
il respecte l'ordonnancement, et sa précondition est vérifiée.

Dans le nouvel état atteint après l'exécution d'un opérateur, si celle-ci est autorisée, la caractéristique est vraie ainsi que la postcondition de l'opérateur.

La transition d'états peut rendre vraie la partie condition de certaines règles, ou la partie précondition de certains opérateurs. Elle peut aussi conduire à une nouvelle étape de l'ordonnancement défini.

Ainsi dans ce nouvel état, le système peut déduire :

- Un premier ensemble d'opérateurs qui **doivent** être réalisés. Ce sont ceux issus de la partie action des règles ayant une partie condition vraie.
- Un deuxième ensemble d'opérateurs qui **peuvent** être réalisés. Ce sont les opérateurs ayant une précondition vraie.
- Un troisième ensemble d'opérateurs qui **peuvent** être réalisés. Ce sont les opérateurs autorisés par l'ordonnancement.

Ces ensembles doivent aider l'utilisateur ou le système à décider de l'action à entreprendre. L'intersection des deuxième et troisième ensembles définit l'ensemble des opérateurs qui **peuvent** être exécutés dans l'état actuel du projet. Un de ces opérateurs peut être invoqué par l'utilisateur ou le système.

Le premier ensemble définit l'ensemble des opérateurs -ce n'est probablement pas un singleton- qui doivent être exécutés, c'est-à-dire les opérateurs que le système doit invoquer. A ce niveau, une stratégie définissant les priorités des invocations d'opérateur(s) doit être établie. Cette stratégie doit être soit connue du système, soit fournie -quand c'est nécessaire- au système par l'utilisateur. Cet ensemble n'est pas forcément inclus dans l'intersection des deux autres. Par exemple, il peut survenir qu'une règle impose une action alors que cette action est interdite par l'ordonnancement.

Pour résoudre ce problème, trois approches peuvent être suivies :

- fournir des outils et des méthodes pour vérifier la cohérence entre les alternatives proposées par les différents ensembles, ainsi que des outils vérifiant qu'il n'existe pas de "dead-lock" (un exemple de "dead-lock" pourrait être :
  un ordonnancement A ; B définissant une séquence
  et une règle IF terminaison (B) THEN A définissant la séquence contraire);
- définir des priorités entre ces alternatives, pour être capable de décider de l'action à entreprendre en cas de conflits;
- décider qu'un opérateur doit être invoqué par le système seulement s'il appartient à l'intersection des trois ensembles, ou définir des méta-règles de choix.

Dans cette section nous avons présenté le modèle. Ce modèle permettra, aussi bien, une **prise d'initiatives** et un **contrôle** par le système. qu'une **assistance** et des **conseils** à l'utilisateur. Par exemple aider l'utilisateur à atteindre un état donné ou lui indiquer les actions à entreprendre pour pouvoir utiliser un opérateur non autorisé dans l'état courant [GOD89a] [BOU88b] [KAI88] sont deux possibilités d'assistance. Dans le dernier cas, le système peut expliquer *pourquoi* l'opérateur ne peut pas être invoqué, par exemple quelle partie de l'expression formant la précondition est fausse, ou quelle est la partie de l'ordonnancement violée. Puis le

système peut utiliser l'ordonnancement, les préconditions et les postconditions pour déduire un ou plusieurs séquencements d'opérateurs à exécuter, pour atteindre l'état désiré [KAI88].

2.5 Relation entre le modèle (MASP) et le processus modélisé (ASP)

Le MASP comme indiqué par son nom (Model...) est un modèle abstrait. Ce modèle abstrait supporte plusieurs réalisations ou instanciations. Ces réalisations sont les effets du processus logiciel. Ce processus logiciel concerne autant le processus de gestion que de développement dudit logiciel. C'est la raison pour laquelle nous parlons de processus de production de logiciels. Supposons défini le type d'objet programmeur au niveau du modèle d'objet, instancier par Khalid, François et Luc ce type d'objet est un début de réalisation du processus logiciel. Cette action est assurée par le gestionnaire de projet. Lancer la compilation d'un module est une réalisation de l'opérateur de compilation. Cette action assurée par le développeur, aussi bien que celle assurée par le gestionnaire doit respecter le modèle introduit par le MASP. Nous devons introduire à ce niveau une précision au niveau de la réalisation du modèle. Cette réalisation n'est pas seulement l'instanciation des modèles d'objets et d'opérateurs, mais aussi l'exécution des opérations associées à ce modèle d'opérateurs. En effet, il ne faut pas perdre de vue que le MASP représente un modèle du processus logiciel, c'est-à-dire qu'il modélise un monde dynamique et que cette modélisation doit prendre en compte cette dynamique. Il ne suffit pas de représenter les objets statiques du modèle d'objet, il faut représenter ou instancier toutes les composantes du modèle.

Nous appellerons instanciation les réalisations du gestionnaire à savoir l'instanciation des types d'objets et des types d'opérateurs qui seront utilisés par le développeur; et nous appellerons interprétation les réalisations du développeur. Nous parlerons alors d'IMASP (pour Instanciated MASP) et d'ASP (pour Assisted Software Process).

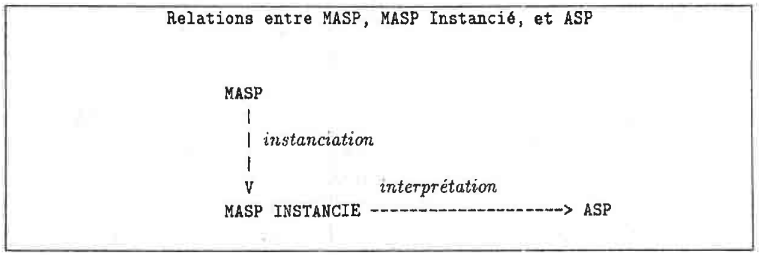
En instanciant un MASP, différentes descriptions statiques peuvent être obtenues. Cette description statique (appelée IMASP) est obtenue en instanciant les types d'objets et d'opérateurs intervenant dans le MASP et en remplaçant les paramètres formels existant au niveau des MASPs par des paramètres effectifs.

L'exemple donné ne comprend pas de paramètres formels mais un tel paramètre formel aurait pû exister si au lieu de la caractéristique suivante : La date_début d'un MEETING n'est jamais un dimanche
nous avions choisi :
La date_début d'un MEETING n'est jamais un $SPECIAL_JOUR

En instanciant tout ce qui peut l'être dans un MASP, nous obtenons un IMASP qui correspond à un modèle de processus de développement propre à une société. **Au lancement du processus de production, l'IMASP est le contexte minimal d'interprétation, ensuite l'IMASP est le dépositaire des objets et des opérateurs manipulés.**

Cependant cet IMASP ne spécifie toujours pas un comportement unique du projet de développement de logiciel. Son interprétation pourra déboucher sur plusieurs ASPs. En effet, un ASP dépend de l'intervention de l'utilisateur qui peut différer tout en se référant et en respectant le même IMASP.

En résumé, un MASP décrit une classe d'IMASPs. Chaque MASP instancié (IMASP) sert alors de modèle à une classe d'ASPs (cf figure ci-dessous).



2.5.1 Instanciation d'un MASP

L'instanciation du MASP O-ORGANISER_CONFERENCE et de son MASP fils O-CREER_MEETING peut être décrite de la manière suivante :

Types d'objets

Nous devons distinguer les deux situations suivantes :

(a) Instanciation des types d'objets du MASP racine (dans l'exemple, le MASP O-ORGANISER_CONFERENCE est le MASP racine) : dans ce cas, c'est cette première instanciation qui crée le contexte de travail (en ce qui concerne les objets). Il faut instancier les types d'objets nécessaires au début de l'interprétation de ce MASP instancié. Dans notre exemple il est raisonnable d'identifier les ORGANISATEURs et les PARTICIPANT-POTENTIELs avant l'interprétation du MASP instancié, c'est-à-dire avant le début de l'ASP.

Une instanciation possible serait :

```
ORGANISATEUR :  Mme S.Smith, responsable
                Mr J.Paco, secrétaire
                (deux objets du type ORGANISATEUR)
PARTICIPANT-POTENTIEL : Mr D.Dupont, France
                        Mme M.Mueller, RFA
                (deux objets du type  PARTICIPANT-POTENTIEL)
```

(b) Instanciation des types d'objets d'un MASP qui n'est pas le MASP racine (dans l'exemple, le MASP O-CREER_MEETING est un tel MASP) : ce type d'instanciation est différent du précédent. En effet dans ce cas, le MASP père de ce MASP a déjà été instancié, c'est-à-dire qu'il existe déjà des objets qui doivent être connus. Dans l'exemple, O-CREER_MEETING doit pour son instanciation avoir connaissance de l'instanciation faite dans O-ORGANISER_CONFERENCE des paramètres de O-CREER_MEETING. Par ailleurs, des objets spécifiques à ce MASP-fils doivent être instanciés afin de créer le contexte minimum d'interprétation de cet ASP. Une instanciation des THEMEs et des ADRESSEs nous sera utile :

```
THEME : BDOO (pour Base de Données Orientées Objet)
        BDD (pour Base de Données Déductives)
        BDR (pour Base de Données Réparties)
                        (trois objets du type THEME)
ADRESSE : CRIN, NANCY, Campus Scientifique, 11
          GIE, PARIS, Bureaux de la colline, 22
          GMV, MADRID, Cristobal Bordiu, 33
          UNIDO, DORTMUND, Universitat Dortmund, 44
                        (quatre objets du type ADRESSE)
```

Pour résumer, l'instanciation du Modèle d'Objets commence par l'identification de tous les objets devant être connus avant l'interprétation d'un IMASP. Ces objets peuvent être soit hérités, soit créés avant le début de l'interprétation. Ensuite, tout au long de l'interprétation, l'IMASP est le dépositaire des objets manipulés (nous verrons qu'il est aussi le dépositaire des opérateurs manipulés).

Types d'opérateurs

Pour les types d'objets, nous avons vu que l'IMASP initial était le contexte d'exécution minimal. Un type d'opérateur élémentaire est instancié en le liant à un outil ayant la même sémantique que celle décrite dans le type d'opérateur.

Un type d'opérateur complexe (MASP) est instancié par l'instanciation de tous les types d'opérateurs introduits dans ses raffinement successifs.

Pour l'exemple, nous instancions les types d'opérateurs introduits dans l'exemple:

$fin : mail -s "C'est fini" 'whoami'

$date_opérateur : Sun3/60#date

$adresse_opérateur : Vax780#adresse

$fin, $date_opérateur et $adresse_opérateur définissaient la sémantique de l'opérateur correspondant dans le MASP. Mail -s "C'est fini" 'whoami', Sun3/60#date et Vax780#adresse, sont supposés être des outils répondant à la sémantique attendue des opérateurs.

l'IMASP est aussi le dépositaire des outils associés aux types d'opérateurs du MASP.

Ordonnancement / Règles / Caractéristique

Dans ces composants, la seule instanciation possible est le passage des paramètres formels aux paramètres effectifs (remplacement de $SPECIAL_JOUR par dimanche, par exemple).

En instanciant comme décrit ci-dessus tous les composants d'un MASP, nous générons un IMASP.

Reprenons l'ensemble de l'exemple :

MASP O-ORGANISER_CONFERENCE

MODELE D'OBJETS

```
PERSONNE ,
    nom : chaine ;

ORGANISATEUR sous type de PERSONNE ,
  rôle_dans_l'organisation_du_meeting : chaine ;


PARTICIPANT-POTENTIEL sous type de PERSONNE ,
  adresse : chaine ;

MEETING ,
  nom : chaine  ,
  thème_général : chaine ,
  date_début : date ,
  date_fin : date ;

R-RESPONSABLE ,
  responsable : ORGANISATEUR ,
  meeting : MEETING ;

R-LETTRE ,
 rédacteur : ORGANISATEUR ,
 meeting : MEETING ,
 contenu : chaine ;

R-MAIL ,
 émetteur : ORGANISATEUR ,
 récepteur : PARTICIPANT_POTENTIEL ,
 date : date ;
```


MODELE D'OPERATEURS

```
O-FIN
 in :  ;
 inout : ;
 out:  ;
 pré: ;
 post: ;
 sémantique : par OUTIL $fin ;

O-CREER_MEETING
 in : ORGANISATEUR ;
 inout : ;
 out: MEETING, R-RESPONSABLE ;
 pré : Le rôle_dans_l'organisation_du_meeting de l'ORGANISATEUR
est responsable ;
 post: thème_général (MEETING) ∈ (Systèmes d'informations,
Génie Logiciel, Ateliers de Génie Logiciel) ;
 sémantique : par MASP O-CREER_MEETING ;

O-FAIRE_R-LETTRE
 in : ORGANISATEUR, MEETING ;
 inout : ;
 out: R-LETTRE ;
 pré : Le rôle_dans_l'organisation_du_meeting de l'ORGANISATEUR
est secrétaire ;
 post: ;
 sémantique : par MASP O-FAIRE_R-LETTRE ;

O-FAIRE_R-MAIL
 in : ORGANISATEUR, R-LETTRE, PARTICIPANT-POTENTIEL ;
 inout : ;
 out: R-MAIL ;
 pré: Le rôle_dans_l'organisation_du_meeting de l'ORGANISATEUR
est secrétaire ;
 post: ;
 sémantique : par MASP O-FAIRE_R-MAIL  ;
```

REGLES

SI terminaison (O-CREER_MEETING) ALORS O-FAIRE_R-LETTRE ;
SI terminaison (O-FAIRE_R-LETTRE) ALORS O-FAIRE_R-MAIL ;

ORDRE

O-CREER_MEETING ; O-FAIRE_R-LETTRE ; [O-FAIRE_R-MAIL] ; O-FIN

CARACTERISTIQUE

Il y a un seul responsable dans le groupe des ORGANISATEURs. ;

FIN MASP O-ORGANISER_CONFERENCE ;

MASP O-CREER_MEETING

MODELE D'OBJETS

THEME ,
  sujet : chaine ;

ADRESSE ,
  société : chaine ,
  ville : chaine ,
  adresse : chaine ,
  salle_de_réunion : entier ;

R-ADRESSE_MEETING ,
  adresse : ADRESSE ,
  meeting : MEETING ;

R-AGENDA_MEETING ,
  thème : THEME ,
  meeting : MEETING ,
  date : date ;

MODELE D'OPERATEURS

O-FIN
 in : ;
 inout : ;
 out: ;
 pré: ;
 post: ;
 sémantique : par OUTIL $fin ;

O-COMPLETER_AGENDA
 in : ORGANISATEUR , MEETING, liste de THEMEs ;
 inout : ;
 out: R-AGENDA-MEETING ;
 pré: Le THEME n'est pas déjà traité (c'est-à-dire qu'il n'existe
pas déjà d'association R-AGENDA_MEETING entre ce THEME et ce MEETING) ;
 post: date(R-AGENDA_MEETING) ∈ [date_début (MEETING), date_fin (MEETING)] ;
 sémantique : par MASP O-COMPLETER_AGENDA ;

O-CHOSIR_DATE
 in : ORGANISATEUR ;
 inout: MEETING ;
 out : ;
 pré: ;
 post: ;
 sémantique : par OUTIL $date_opérateur ;

O-CHOISIR_ADRESSE
 in : ORGANISATEUR, MEETING, liste de ADRESSEs ;
 inout : ;
 out: R-ADRESSE_MEETING ;
 pré: ;
 post: R-ADRESSE_MEETING (adresse, meeting)
 et ville (adresse) ∈ (PARIS, LONDRES, BONN, MADRID, BERN) ;
 sémantique : par OUTIL $adresse_opérateur ;

REGLES

ORDRE

```
((O-CHOISIR_DATE ; O-CHOISIR_ADRESSE) | (O-CHOISIR_ADRESSE; O-CHOISIR_DATE)) ;
   [O-COMPLETER_AGENDA]  ; O-FIN
```

CARACTERISTIQUE

```
La date_début d'un MEETING n'est jamais un dimanche.
```

FIN MASP O-CREER_MEETING ;

IMASP initial pour O-ORGANISER_CONFERENCE

```
  ORGANISATEUR :  Mme S.Smith, responsable
                  Mr J.Paco, secrétaire
                              (deux objets du type ORGANISATEUR)

  PARTICIPANT-POTENTIEL : Mr D.Dupont, France
                          Mme M.Mueller, RFA
                             (deux objets du type  PARTICIPANT-POTENTIEL)

$fin : mail -s "C'est fini" `whoami`
```

FIN IMASP O-ORGANISER_CONFERENCE

IMASP initial pour O-CREER_MEETING

```
  THEME : BDOO (pour Base de Données Orientées Objet)
          BDD (pour Base de Données Déductives)
          BDR (pour Base de Données Réparties)
                              (trois objets du type THEME)

  ADRESSE : CRIN, NANCY, Campus Scientifique, 11
            GIE, PARIS, Bureaux de la colline, 22
            GMV, MADRID, Cristobal Bordiu, 33
            UNIDO, DORTMUND, Universitat Dortmund, 44
                              (quatre objets du type ADRESSE)

$adresse_opérateur : Vax780#adresse

$date_opérateur : Sun3/60#date
```

FIN IMASP O-CREER_MEETING

Un exemple d'ASP interprétant l'instanciation présentée du MASP est donné ci-dessous. Cette interprétation n'est pas bien sûr la seule possible. Si par exemple, des requêtes utilisateur différentes interviennent, alors un ASP différent est obtenu. L'ASP décrit ci-dessous est l'un des nombreux ASPs pouvant résulter de l'interprétation du même IMASP choisi.

Les questions émanant du système sont suivies de " ?". Les réactions du système commencent par "->". Le système indique à quel niveau du MASP il se trouve : c'est le niveau 1 si nous sommes dans le MASP O-ORGANISER_CONFERENCE et le niveau 2 si nous sommes dans le MASP MASP O-CREER_MEETING. "niveau i op ?" signifie que le système attend une invocation d'un opérateur du niveau i.

```
niveau 1 op ?  O-CREER_MEETING
               IN ? Paco
                  --> REJET (la précondition de O-CREER_MEETING est fausse:
                             Paco n'est pas un responsable)

niveau 1 op ? O-FAIRE_R-LETTRE
           --> REJET (l'ordonnancement n'est pas respecté :
                      O-CREER_MEETING devrait être fait avant)

niveau 1 op ? O-CHOISIR_ADRESSE
           --> REJET (opérateur indéfini à ce niveau)

niveau 1 op ?  O-CREER_MEETING
               IN ? Smith
               INOUT ?
               OUT ? $meeting , R-RESPONSABLE (Smith, $Meeting)
                  --> BIENVENU dans le contexte de  O-CREER_MEETING
                  niveau 2 op ? O-FIN
                             --> REJET (l'ordonnancement n'est pas respecté :
((O-CHOISIR_DATE ; O-CHOISIR_ADRESSE) | (O-CHOISIR_ADRESSE; O-CHOISIR_DATE)) ;
   [O-COMPLETER_AGENDA]
                                        devrait être fait avant)

                  niveau 2 op ? O-CHOISIR_ADRESSE
                               IN ? Paco, $meeting, (CRIN, GMV, GIE)
                                 --> REJET ( ORGANISATEUR inconnu: choisir Smith)

                  niveau  op ? O-FAIRE_R-MAIL
```



```
                  --> REJET (opérateur indéfini à ce niveau)

niveau 2 op ? O-CHOISIR_ADRESSE
               IN ? Smith, $meeting, (CRIN, GMV, GIE)
               INOUT ?
               OUT ? R-ADRESSE_MEETING ($adresse, $meeting)
                   |
                   | Appel de l'outil instanciant
                   | O-CHOISIR_ADRESSE (Vax780#adresse)
                   |
                   V
          OUT :
          $adresse = CRIN, NANCY, Campus Scientifique, 11
          R-ADRESSE_MEETING ($adresse, $meeting)

               --> REJET (la  postcondition de
                 O-CHOISIR_ADRESSE n'est pas respectée :
                 NANCY ∉ (PARIS, LONDRES, BONN, MADRID, BERN)

niveau 2 op ? O-CHOISIR_ADRESSE
               IN ? Smith, $meeting, (CRIN, GMV, GIE)
               INOUT ?
               OUT ? R-ADRESSE_MEETING ($adresse, $meeting)
                   |
                   | Appel de l'outil instanciant
                   | O-CHOISIR_ADRESSE (Vax780#adresse)
                   |
                   V
          OUT :
          $adresse = GIE, PARIS, Bureaux de la colline, 22
          R-ADRESSE_MEETING ($adresse, $meeting)
               --> O.K. (la postcondition est bien respectée)

               --> O.K. (caractéristique respectée)


niveau 2 op ? O-CHOISIR_DATE
               IN ? Smith
```

```
                INOUT ? $meeting
                OUT ?
                    |
                    | Appel de l'outil instanciant
                    | O-CHOISIR_DATE (Sun3/60#date)
                    |
                    V
                 INOUT :
                 $meeting = SI'89,
                 Systèmes d'informations,
                 Dimanche 2/7/89,
                 Mercredi 5/7/89
                --> REJET  (caractéristique violée :
                    le meeting ne peut commencer un dimanche)

niveau 2 op ? O-CHOISIR_DATE
                IN ? Smith
                INOUT ? $meeting
                OUT ?
                    |
                    | Appel de l'outil instanciant
                    | O-CHOISIR_DATE (Sun3/60#date)
                    |
                    V
                 INOUT :
                 $meeting = SI'89,
                            Systèmes d'informations,
                            Lundi 3/7/89,
                            Mercredi 5/7/89

                --> O.K.  (caractéristique respectée)

niveau 2 op ? O-COMPLETER_AGENDA
                IN ? Smith, $meeting, (BDOO, BDD, BDR)
                INOUT ?
                OUT ? R-AGENDA_MEETING ($thème,$meeting,$date)
                    |
                    | Interprétation de l'instanciation  du MASP
```



```
                    | O-COMPLETER_AGENDA (non développé)
                    |
                    V
                OUT :
                $thème = BDOO
                R-AGENDA_MEETING ($thème,$meeting,Mardi 4/7/89)
                --> O.K. (la postcondition est bien respectée:
                          4/7 ∈ [3/7, 5/7])

                --> O.K. (caractéristique respectée)

niveau 2 op ? O-COMPLETER_AGENDA
                IN ? Smith, $meeting, (BDOO, BDD, BDR)
                INOUT ?
                OUT ? R-AGENDA_MEETING ($thème,$meeting,$date)
                    |
                    | Interprétation de l'instanciation  du MASP
                    | O-COMPLETER_AGENDA (non développé)
                    |
                    V
                OUT :
                $thème = BDD
                R-AGENDA_MEETING ($thème,$meeting, Lundi 3/7/89)
                --> O.K. (la postcondition est bien respectée:
                          3/7 ∈ [3/7, 5/7])

                --> O.K. (caractéristique respectée)

niveau 2 op ? O-CHOISIR_ADRESSE
                --> REJET (L'ordonnancement n'est pas respecté:
                O-CHOISIR_ADRESSE est autorisé une seule fois)

niveau 2 op ? O-COMPLETER_AGENDA
                IN ? Smith, $meeting, (BDOO, BDD, BDR)
                INOUT ?
                OUT ? R-AGENDA_MEETING ($thème,$meeting,$date)
                    |
                    | Interprétation de l'instanciation  du MASP
```

```
                    | O-COMPLETER_AGENDA (non développé)
                    |
                    V
                   OUT :
                   $thème = BDR
                   R-AGENDA_MEETING ($thème,$meeting, Lundi 3/7/89)
                   --> O.K. (la postcondition est bien respectée:
                            5/7 ∈ [3/7, 5/7])

                   --> O.K. (caractéristique respectée)

        niveau 2 op ? O-FIN
                    |
                    | Appel de l'outil instanciant
                    | O-FIN (mail -s "C'est fini" `whoami`)
                    |
                    V
                   --> O.K. C'est fini
      --> FIN du contexte de O-CREER_MEETING
   OUT :
   $meeting, R-RESPONSABLE (Smith,$meeting)
    --> O.K.

A ce niveau O-CREER_MEETING a été exécuté, aussi la règle
SI terminaison (O-CREER_MEETING) ALORS O-FAIRE_R-LETTRE
déclenche l'opérateur O-FAIRE_R-LETTRE

--> niveau 1 op  O-FAIRE_R-LETTRE déclenché par le système
   IN ? Smith, $meeting;
       --> REJET (la précondition de O-FAIRE_R-LETTRE est fausse :
              Smith n'est pas secrétaire)

   IN ? Paco, $meeting;
        |
        |  Interprétation de l'instanciation  du MASP
        |  O-FAIRE_R-LETTRE (non développé)
        |
        V
```



```
                   OUT :
                   $r-lettre = R-LETTRE (Paco, $meeting, 'Madame, Monsieur, ...")
                    --> O.K.

A ce niveau O-FAIRE_R-LETTRE  a été exécuté, aussi la règle
SI terminaison (O-FAIRE_R-LETTRE) ALORS   O-FAIRE_R-MAIL
déclenche l'opérateur O-FAIRE_R-MAIL

--> niveau 1 op  O-FAIRE_R-MAIL déclenché par le système
   IN ? Paco, $r-lettre , (Dupont, Mueller) ;
        |
        |  Interprétation de l'instanciation  du MASP
        |  O-FAIRE_R-MAIL (non développé)
        |
        V
   OUT :
   $r-mail = R-MAIL (Paco, Mueller, `date`)
    --> O.K.

niveau 1 op ?  O-CREER_MEETING
    --> REJET (l'ordonnancement n'est pas respecté O-CREER_MEETING
   est autorisé une seule fois, il est interdit de répéter
   l'exécution de O-CREER_MEETING)

niveau 1 op ? O-FAIRE_R-MAIL invoqué cette fois par l'utilisateur
   IN ? Paco, $r-lettre , (Dupont, Mueller) ;
        |
        |  Interprétation de l'instanciation  du MASP
        |  O-FAIRE_R-MAIL (non développé)
        |
        V
   OUT :
   $r-mail = R-MAIL (Paco, Dupont, `date`)
    --> O.K.

niveau 1 op ? O-FIN
        |
        | Appel de l'outil instanciant
```

```
                | O-FIN (mail -s "C'est fini" `whoami`)
                |
                V
--> O.K. C'est fini
```

2.6 Conclusion

Dans ce chapitre, nous avons présenté notre modèle et un exemple montrant son utilisation. La syntaxe de ce modèle n'est pas à l'heure actuelle complètement formalisée et figée. Nous travaillons actuellement dans le cadre du projet ALF à la définition d'une syntaxe précise pour chacun des composants de ce modèle. Ce n'est qu'une fois cette syntaxe définie et complétement développée qu'il sera possible de présenter un exemple concret et complet d'utilisation du modèle basée sur cette syntaxe. Cependant, après avoir vu sur l'exemple simple développé l'intérêt de chacun des composants du modèle, nous présenterons une maquette montrant la faisabilité et l'intérêt d'une telle modélisation sur un exemple lié au développement de logiciel (voir chapitre 3).

3
Un exemple et une réalisation

3.1 Présentation de l'exemple

Nous avons réalisé une maquette mettant en œuvre la création d'un IMASP correspondant à un MASP particulier (MASP PRODUCTION DE LOGICIEL) et l'interprétation de cet IMASP.

Le but de cette maquette n'est en aucune façon de proposer une implantation complète de notre modèle, mais plutôt d'avoir un support pour étudier le comportement d'un système mettant en œuvre quelques-uns des concepts décrits dans cette thèse. Nous avons mis l'accent sur l'aspect contrôle de l'utilisateur par le système et sur l'aspect prise d'initiatives du système. L'aspect contrôle est pris en compte par l'utilisation faite des préconditions et des postconditions des opérateurs et par la vérification de la caractéristique. L'aspect prise d'initiatives est traité grâce aux règles et au déclenchement automatique des actions décrites dans la partie droite des règles lorsque les situations décrites en partie gauche sont vérifiées. Cette prise d'initiatives est réalisée grâce à un moteur d'inférence qui a conduit à nommer la maquette **PIRE** (Piloting using an Infering Rule Engine). La maquette que nous avons développée repose sur une représentation sous forme de schéma Entité-Association des données opérationnelles basée sur le modèle d'objets présenté dans le chapitre 2. Nous définissons un schéma E-A en termes de types d'objets, de types d'associations et de types d'attributs. La maquette fournit toutes les fonctions de création, d'accès et de modification associées à chacun de ces types. Chaque opération y est spécifiée, grâce aux différentes fonctions précitées, en termes d'effets sur la base de données. Ces effets représentent l'exécution de l'opération. Les opérations sont donc évaluées comme des fonctions de mise à jour de la base des données opérationnelles.

Définissons en quelques mots notre modèle de processus de production de logiciels. Deux types d'activités peuvent être réalisés lors de la production de logiciels : une activité de GESTION ou bien une activité de DEVELOPPEMENT.

L'activité de gestion permet la manipulation d'informations concernant les personnes

impliquées dans l'activité de développement, la manipulation des tâches à réaliser ainsi que les contraintes financières associées à ces tâches (ces contraintes exprimant le coût maximum autorisé pour une tâche donnée).

L'activité de développement permet la manipulation de modules. Cette manipulation peut se traduire en création, édition, compilation, application de tests aux modules, visa appliqué à un module,etc.

Nous présenterons au fur et à mesure de ce chapitre toutes les spécificités de ce modèle de PRODUCTION DE LOGICIELS.

3.2 le MASP PRODUCTION DE LOGICIELS

Le modèle d'objets de ce MASP comprend les types d'objets PERSONNE, TEST, TACHE et RISQUE. Par ailleurs, toute PERSONNE participant à une TACHE est reliée à cette TACHE par une association du type TRAVAILLE DANS. Le modèle d'objets est le suivant :

- Le type d'objet PERSONNE est caractérisé par le *nom* de la personne et par son *grade*, entier censé représenter la position hiérarchique de la personne dans l'entreprise.
- Le type d'objet TEST représente les tests que l'on peut faire subir à un module et est caractérisé par son *nom*. Un programme permettant de tester la portabilité d'un module est un exemple de TEST.
- Le type d'objet TACHE est caractérisé par son *nom* et par le *coût* représentant la somme d'argent déjà dépensée par cette tâche.
- Le type d'objet RISQUE représente un risque dont la gravité est exprimée par l'entier *sévérité*.
- Le type d'association TRAVAILLE DANS associe toute PERSONNE aux TACHEs dans lesquelles elle est impliquée.


```
MODELE D'OBJETS

PERSONNE ,
 nom : chaine ,
 grade : entier ;

TACHE ,
 nom : chaine ,
 coût : réel ;

RISQUE ,
 sévérité : entier ;

TEST ,
  nom : chaine ;

TRAVAILLE DANS ,
  personne : PERSONNE ,
  tâche : TACHE ;
```

Le modèle d'objets de ce MASP PRODUCTION DE LOGICIELS est schématisé dans la figure 3.1.

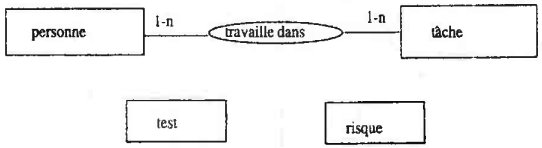


Figure 3.1. Le modèle d'objets du MASP PRODUCTION DE LOGICIELS

Le modèle d'opérateurs comprend deux types d'opérateurs permettant de réaliser une tâche de gestion (GESTION) ou une tâche de développement (DEVELOPPEMENT). Ces deux types d'opérateurs sont complexes et sont donc décrits en termes de MASP.

```
MODELE D'OPERATEURS

GESTION
 in :  ;
 inout : PERSONNE, TEST, TACHE, RISQUE, TRAVAILLE DANS ;
 out : ;
 pré :  ;
 post:  ;
 sémantique: par MASP GESTION ;

DEVELOPPEMENT ;
 in : PERSONNE, TEST, TRAVAILLE DANS;
 inout : TACHE, RISQUE ;
 out : ;
 pré :  ;
 post:  ;
 sémantique: par MASP DEVELOPPEMENT ;
```

Nous ne définissons pas de règle ni de caractéristique au niveau de ce MASP.

L'ordonnancement des opérateurs est défini de la sorte :

```
[ GESTION | DEVELOPPEMENT ]
```

Cet ordonnancement traduit que le choix est offert à l'utilisateur entre une activité de développement ou une activité de gestion. L'utilisateur peut réitérer son choix c'est-à-dire qu'il peut soit réaliser à nouveau l'opération choisie précédemment, soit décider de passer à l'autre opération.

3.3 Le MASP-fils GESTION

Celui-ci doit permettre de réaliser le type d'opérateur GESTION défini au niveau du MASP père PRODUCTION DE LOGICIELS.

Le modèle d'objets de ce MASP-fils GESTION permet de manipuler les types d'objets PERSONNE, TACHE, RISQUE et TEST hérités du MASP père car ils sont passés en paramètres de GESTION (cf chapitre 2 de cette partie), les types d'objets CONTRAINTE FINANCIERE, RISQUE FINANCIER, et les types d'associations entre ces types d'objets, en particulier l'association SOUMISE A qui associe une CONTRAINTE FINANCIERE à une TACHE et l'association RF POUR qui associe un RISQUE FINANCIER à une TACHE.

Le modèle d'objets de ce MASP est décrit ci-dessous :

- Le type d'objet CONTRAINTE FINANCIERE représente une enveloppe budgétaire, elle-même caractérisée par son *nom* et son *montant*.

- Le type d'objet RISQUE FINANCIER est un sous-type de RISQUE (hérité du MASP père) et représente un risque de dépassement de budget.
- Le type d'association SOUMISE A associe une TACHE à l'enveloppe budgétaire qui lui est attribuée.
- Le type d'association RF POUR associe un RISQUE FINANCIER à une TACHE, si celle-ci risque de dépasser son budget. Le type d'objet TACHE n'a pas à être redéfini à ce niveau car il est hérité du modèle d'objets du MASP père (PRODUCTION DE LOGICIELS).

```
MODELE D'OBJETS

CONTRAINTE FINANCIERE ,
 nom : chaine ,
 montant : réel ;

RISQUE FINANCIER sous type de RISQUE ;

RF POUR ,
 risque financier : RISQUE FINANCIER,
 tâche : TACHE ; /*hérité du MASP père*/

SOUMISE A ,
 tâche : TACHE , /*hérité du MASP père*/
 contrainte financière : CONTRAINTE FINANCIERE ;
```

Le modèle d'objets de ce MASP GESTION est schématisé dans la figure 3.2.

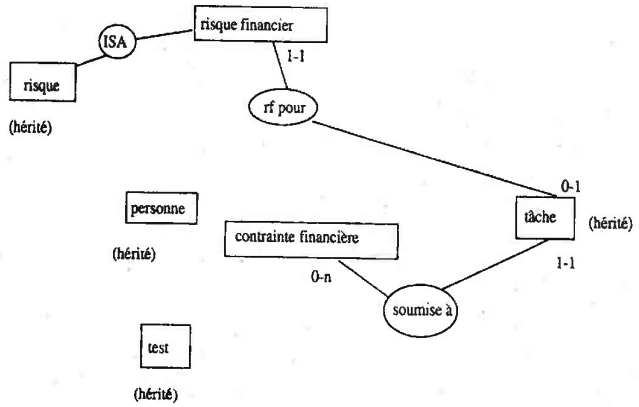


Figure 3.2. Le modèle d'objets du MASP GESTION

Au niveau de ce MASP, le gestionnaire peut créer des personnes -ou tout au moins leur représentation- (CREER-PERSONNE), mettre à jour les informations concernant ces personnes (MAJ-PERSONNE), créer des tests (CREER-TEST), créer des tâches (CREER-TACHE), mettre à jour les informations concernant ces tâches (MAJ-TACHE), contraindre ces tâches (CONTRAINDRE-TACHE) c'est-à-dire créer une CONTRAINTE FINANCIERE et l'associer à travers SOUMISE A à une tâche. En outre, comme pour tout MASP, il existe une opération permettant de terminer l'interprétation (FIN). Le modèle d'opérateurs du MASP-fils GESTION est donc le suivant :

- Le type d'opérateur CREER-PERSONNE permet la création d'une instance du type d'objet PERSONNE. C'est un type d'opérateur simple réalisé grâce à l'outil créer-personne qui doit satisfaire, à la fin de son exécution, le fait que le *grade* de toute PERSONNE est compris entre 1 et 5.
- Le type d'opérateur MAJ-PERSONNE permet la modification des informations concernant une PERSONNE. Il est réalisé par l'outil maj-personne qui ne doit pas, lui non plus, changer le *grade* d'une PERSONNE en dehors de la fourchette de valeurs autorisée (de 1 à 5).
- Le type d'opérateur CREER-TACHE permet la création d'un objet de type TACHE et ce type d'opérateur est réalisé par l'outil créer-tâche.
- Le type d'opérateur CONTRAINDRE-TACHE permet d'associer une TACHE à une CONTRAINTE FINANCIERE à travers l'association SOUMISE A, c'est-à-dire imposer une limite financière à la TACHE grâce à l'outil contraindre-tâche.
- Le type d'opérateur MAJ-TACHE permet de modifier les attributs caractérisant une TACHE : il permet de modifier le *coût* de celle-ci (montant déjà consommé).
- En dernier lieu, le type d'opérateur FIN permet de terminer une session de gestion.


```
MODELE D'OPERATEURS

CREER-PERSONNE
  in :  ;
  inout : ;
  out: PERSONNE ;
  pré :  ;
  post:  1 ≤ grade (PERSONNE) ≤ 5   ;
  sémantique: par OUTIL créer-personne ;

MAJ-PERSONNE
  in : ;
  inout : PERSONNE ;
  out:  ;
  pré :  ;
  post:  1 ≤ grade (PERSONNE) ≤ 5   ;
  sémantique: par OUTIL maj-personne ;

CREER-TEST
  in : ;
  inout : ;
  out: TEST ;
  pré :  ;
  post:  ;
  sémantique: par OUTIL créer-test ;

CREER-TACHE
  in :  ;
  inout : ;
  out: TACHE ;
  pré :  ;
  post:  ;
  sémantique: par OUTIL créer-tâche ;

CONTRAINDRE-TACHE
  in : TACHE, CONTRAINTE FINANCIERE ;
  inout : ;
  out: SOUMISE A (TACHE, CONTRAINTE FINANCIERE) ;
  pré :  ;
  post:  ;
  sémantique: par OUTIL contraindre-tâche ;
```

```
MAJ-TACHE
  in : ;
  inout : TACHE ;
  out:  ;
  pré :  ;
  post:  ;
  sémantique: par OUTIL maj-tâche ;

FIN
 in :  ;
 inout : ;
 out:  ;
 pré: ;
 post: ;
 sémantique : par OUTIL fin-tâche ;
```

Les règles de ce MASP-fils sont :

```
SI le nombre de PERSONNEs est égal à deux ou six
  ALORS prévenir que le nombre de PERSONNEs a atteint un extremum
SI SOUMISE A (tâche, contrainte financière)
  et coût (tâche) ≥ 0,8 * montant(contrainte financière)
  ALORS créer un RISQUE FINANCIER
   et l'associer à cette TACHE à travers RF POUR (RISQUE FINANCIER, TACHE)
```

Ces règles décrivent les initiatives que le système doit prendre (partie droite des règles) dans les situations définies dans la partie gauche des règles.

L'ordonnancement utilisé est le suivant :

```
CREER-TEST
| ( (CREER-TACHE ; (*[MAJ-TACHE]*) ; (*CONTRAINDRE-TACHE*) ; (*[MAJ-TACHE]*))
   | (CREER-PERSONNE ; (*[MAJ-PERSONNE]*)) )
```

Cet ordonnancement offre à l'utilisateur la possibilité de choisir de travailler sur les tests, les tâches ou les personnes. Un objet du type TEST ne peut être créé qu'une seule fois. Un objet du type TACHE ne peut être créé qu'une seule fois, ne peut être contraint qu'une seule fois au plus, et seulement après sa création. Une modification d'un objet du type PERSONNE ou du type TACHE ne peut être faite qu'après la création de cet objet.

La caractéristique est :

```
2 ≤ card (PERSONNE) ≤ 6
```

Cette caractéristique limite le nombre d'intervenants dans un projet (il y a toujours entre 2 et 6 PERSONNEs dans un projet).

A ce stade, le MASP-fils GESTION est entièrement décrit : en effet, tous les types d'opérateurs intervenant dans celui-ci sont des outils atomiques ne nécessitant plus de raffinements. Nous avons vu dans le § 2.5 du chapitre 2 de cette partie que l'affectation d'outils exécutables à ces outils du modèle permet de réaliser une partie de l'instanciation d'un IMASP correspondant à ce MASP et permettra l'interprétation de cet IMASP, c'est-à-dire qu'un ASP sera créé. Ces outils effectifs seront fournis dans la réalisation que nous avons faite sous forme de fonctions le_lisp.

3.4 Le MASP-fils DEVELOPPEMENT

Le MASP fils DEVELOPPEMENT définit le modèle dans lequel sera plongé le développeur et est construit autour du type d'objet MODULE. Le type d'objet MODULE, ainsi que le type d'objet DOCUMENT, sont des sous-types du type d'objet FICHIER. Un objet du type MODULE est SPECIFIE PAR un DOCUMENT et une PERSONNE est RESPONSABLE DE ce MODULE. Une PERSONNE VISE un MODULE. Une PERSONNE est AUTEUR d'un MODULE. Un MODULE SUBIT des TESTs et PARTICIPE A une TACHE. Une TACHE A BESOIN de MODULEs. Un RISQUE TEMPS peut être PROVOQUE PAR un MODULE, et devenir un RT POUR (risque temps pour) une TACHE. Une PERSONNE CONTROLE une TACHE. Le modèle d'objets de ce MASP-fils DEVELOPPEMENT peut être décrit de la manière suivante :

- Le type d'objet FICHIER est caractérisé par son *nom*, sa *date-de-création* et sa *date-de-dernière-mise-à-jour*.
- Le type d'objet MODULE est sous-type de FICHIER; il est donc caractérisé par les attributs de celui-ci auxquels viennent s'ajouter les booléens *compilé?*, *testé?* et *retardé?* (indiquant respectivement si un module est compilé, testé et/ou s'il a pris du retard), une *date-de-compilation* indiquant pour un module compilé sa date de compilation et les entiers *nombre-de-tests-prévus* pour ce module et *nombre-de-tests-réalisés* effectivement.
- Le type d'objet DOCUMENT sous-type de FICHIER n'est caractérisé, lui, que par les attributs hérités de son type d'objet père.
- Le type d'objet RISQUE TEMPS sous-type du type d'objet RISQUE hérité du MASP-père exprime un risque de dépassement de délai.
- Le type d'association RT POUR associe un RISQUE TEMPS à une TACHE et signifie que l'exécution de celle-ci risque d'être retardée. Le retard envisagé est fonction de la *sévérité* du RISQUE TEMPS associé.

- Le type d'association PROVOQUE PAR relie un MODULE qui prend du retard à un RISQUE TEMPS.
- Le type d'association SPECIFIE PAR relie un MODULE à un DOCUMENT constituant sa spécification.
- Le type d'association A LA RESPONSABILITE DE relie un MODULE à la PERSONNE responsable de ce module.
- Le type d'association AUTEUR relie un MODULE à son AUTEUR
- Le type d'association VISE relie tout MODULE visé à la PERSONNE l'ayant visé.
- Le type d'association CONTROLE affecte une TACHE à la PERSONNE responsable de cette tâche.
- Le type d'association A BESOIN DE associe à une TACHE tous les MODULEs dont celle-ci a besoin.
- Le type d'association PARTICIPE A associe à une TACHE tous les modules terminés (nous verrons qu'un module est terminé s'il est visé) dont celle-ci a besoin.
- Le type d'association SUBIT associe un MODULE à l'ensemble des TESTs qu'il a subi.

```
MODELE D'OBJETS

FICHIER ,
 nom : chaine ,
 date-de-création : date ,
 date-de-dernière-mise-à-jour : date ;

MODULE sous type de FICHIER  ,
  compilé? : booleen ,
  testé? : booleen ,
  retardé?: booleen ,
  date-de-compilation : date ,
  nombre-de-tests-prévus : entier ,
  nombre-de-tests-réalisés :entier ;

DOCUMENT sous type de FICHIER;

RISQUE TEMPS sous type de RISQUE ; /*hérité du MASP père*/

RT POUR ,
 risque temps : RISQUE TEMPS,
 tâche : TACHE ;
```



```
PROVOQUE PAR,
 risque temps : RISQUE TEMPS,
 module : MODULE ;

SPECIFIE PAR,
 module : MODULE,
 document : DOCUMENT;

A LA RESPONSABILITE DE,
  personne : PERSONNE ,/*hérité du MASP père*/
  module : MODULE ;

AUTEUR,
  personne : PERSONNE , /*hérité du MASP père*/
  module : MODULE ;

VISE,
  personne : PERSONNE , /*hérité du MASP père*/
  module : MODULE ;

CONTROLE,
  personne : PERSONNE , /*hérité du MASP père*/
  tâche : TACHE ;

A BESOIN DE,
  tâche : TACHE, /*hérité du MASP père*/
  module : MODULE,

PARTICIPE A ,
  module : MODULE,
  tâche : TACHE ; /*hérité du MASP père*/

SUBIT ,
  module : MODULE,
  test : TEST , /*hérité du MASP père*/
  note : entier ;
```

Le modèle d'objet de ce MASP DEVELOPPEMENT est schématisé dans la figure 3.3.

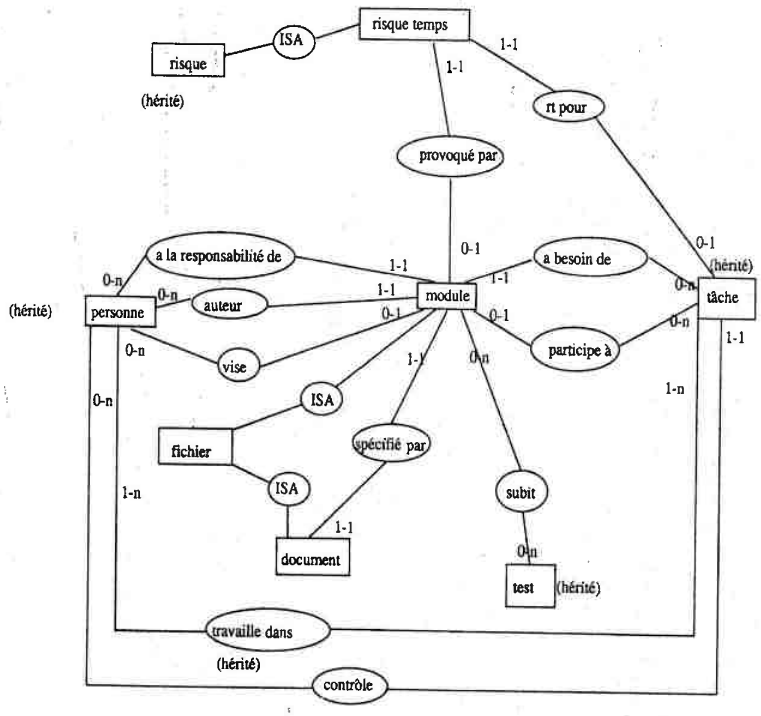


Figure 3.3. Le modèle d'objets du MASP DEVELOPPEMENT

Des instances du type MODULE peuvent être créées (CREER-MODULE), éditées (EDITER-MODULE), documentées (DOCUMENTER-MODULE), compilées (COMPILER-MODULE), testées (TESTER-MODULE) par leur auteur, visées (VISER-MODULE) par leur responsable, exportées (EXPORTER-MODULE) par leur auteur vers les tâches ayant besoin d'elles, ou encore prendre du retard (RETARD-ENVISAGE). Le modèle d'opérateurs du MASP-fils DEVELOPPEMENT est alors :

- Le type d'opérateur CREER-MODULE permet la création d'un objet du type MODULE, du DOCUMENT lui servant de spécification et il permet l'association du module créé avec son AUTEUR et avec la TACHE à laquelle il devra participer quand il sera terminé. L'outil créer-module de ce type devra faire en sorte que le MODULE créé ne peut être considéré comme déjà testé (*testé?* doit avoir la valeur false, le nombre-de-tests-réalisés la valeur 0 et le *nombre-de-tests-prévus* doit avoir une valeur supérieure ou égale à 1.

- Le type d'opérateur EDITER-MODULE permet l'édition d'un MODULE. En réalité nous nous sommes bornés à simuler cette édition par le changement de l'attribut *date-de-dernière-mise-à-jour*. L'opérateur editer-module ne peut être activé que par l'auteur du MODULE (précondition).

- Le type d'opérateur COMPILER-MODULE permet la compilation d'un MODULE (en fait changement des attributs *compilé?* et *date-de-compilation* suivant que la compilation est considérée comme correcte ou pas).

- Le type d'opérateur TESTER-MODULE permet de simuler le test d'un module. L'opérateur tester-module ne peut être activé que par l'auteur du MODULE (précondition) et son effet est d'associer le MODULE à un TEST grâce au type d'association SUBIT et d'affecter une note correspondant à ce test (attribut *note* de SUBIT).

- Le type d'opérateur VISER-MODULE est la création d'une association du type VISE entre une PERSONNE et un MODULE. La précondition impose qu'un module ne peut être visé que par la personne RESPONSABLE DE celui-ci.

- Le type d'opérateur EXPORTER-MODULE permet à la personne responsable du module (précondition) de créer une association PARTICIPE A entre un module et la TACHE qui A BESOIN DE celui-ci.

- Le type d'opérateur DOCUMENTER-MODULE permet à l'auteur d'un module et uniquement à lui (précondition) de modifier le DOCUMENT spécifiant le module (modification simulée par la mise à jour de l'attribut *date-de-dernière-mise-à-jour* du DOCUMENT associé au module par le type d'association SPECIFIE PAR).

- Le type d'opérateur RETARD-ENVISAGE permet au responsable d'un MODULE et uniquement à celui-ci (précondition) de modifier l'attribut *retardé?* de ce module, s'il estime qu'il a pris du retard.

```
MODELE D'OPERATEURS

CREER-MODULE
  in : PERSONNE, TACHE ;
  inout : ;
  out: MODULE, DOCUMENT, SPECIFIE PAR, A LA RESPONSABILITE DE, A BESOIN DE
  pré : ;
  post:  testé? (MODULE) = false
   et nombre-de-tests-prévus  (MODULE) ≥ 1
   et nombre-de-tests-réalisés (MODULE) = 0 ;
  sémantique: par OUTIL créer-module ;



EDITER-MODULE
  in : PERSONNE, TRAVAILLE DANS, TACHE ;
  inout : MODULE;
  out: ;
  pré:  TRAVAILLE DANS (personne1, tâche)
        et AUTEUR (personne2, module)
        et personne1 =  personne2 ;
  post : ;
  sémantique: par OUTIL editer-module ;



COMPILER-MODULE
  in : PERSONNE, TRAVAILLE DANS /*hérité de MASP PRODUCTION DE LOGICIELS*/;
  inout: MODULE ;
  out ;
  pré:  TRAVAILLE DANS (personne1, tâche)
        et AUTEUR (personne2, module)
        et personne1 =  personne2 ;
  post :  ;
  sémantique: par OUTIL compiler-module ;



TESTER-MODULE
  in : PERSONNE, TRAVAILLE DANS, TEST ;
  inout: MODULE ;
  out: SUBIT ;
  pré:  TRAVAILLE DANS (personne1, tâche)
        et AUTEUR (personne2, module)
        et personne1 =  personne2 ;
  post : ;
  sémantique: par OUTIL tester-module ;
```



```
VISER-MODULE
  in : PERSONNE, TRAVAILLE DANS , A LA RESPONSABILITE DE ;
  inout: MODULE ;
  out: VISE ;
  pré:  TRAVAILLE DANS (personne1, tâche)
   et A LA RESPONSABILITE DE (personne2, module)
   et personne1  = personne2 ;
  post :  ;
  sémantique: par OUTIL viser-module ;

EXPORTER-MODULE
  in : PERSONNE, TRAVAILLE DANS, MODULE, A LA RESPONSABILITE DE, A BESOIN DE ;
  inout ;
  out: PARTICIPE A ;
  pré:  TRAVAILLE DANS (personne1, tâche)
   et A LA RESPONSABILITE DE (personne2, module)
   et personne1  = personne2 ;
  post :  ;
  sémantique: par OUTIL exporter-module ;

DOCUMENTER-MODULE
  in : PERSONNE, TRAVAILLE DANS, MODULE, SPECIFIE PAR ;
  inout :  DOCUMENT ;
  out: ;
  pré:  TRAVAILLE DANS (personne1, tâche)
        et AUTEUR (personne2, module)
        et personne1 =  personne2 ;
  post :  ;
  sémantique: par OUTIL documenter-module ;

RETARD-ENVISAGE
  in : PERSONNE, TRAVAILLE DANS, TACHE;
  inout: MODULE;
  out :
  pré:  TRAVAILLE DANS (personne1, tâche)
   et A LA RESPONSABILITE DE (personne2, module)
   et personne1  = personne2 ;
  post :  ;
  sémantique: par OUTIL exporter-module ;

FIN
 in :  ;
 inout : ;
 out:  ;
 pré: ;
 post: ;
 sémantique : par OUTIL fin-tâche ;
```

Les règles sont :

```
SI nombre-de-tests-réalisés (MODULE) ≥ nombre-de-tests-prévus (MODULE)
  et SUBIT (MODULE, TEST, note)
  et toutes  note (s) ≥ 15
  ALORS mettre testé? (module) à true
SI testé? (module) = true
  ALORS soumettre le module à son responsable pour un visa
SI A BESOIN DE (tâche, module) et vise?(module) = true
  ALORS EXPORTER (module) [i.e. créer une association PARTICIPE A]
SI PROVOQUE PAR (risque temps, module) et A BESOIN DE (tâche, module)
  ALORS créer, si elle n'existe pas déjà,
   association RT POUR (risque temps, tâche)
SI retardé?(module) = true
  ALORS créer, si elle n'existe pas déjà,
   association PROVOQUE PAR (créer(risque temps),  module)
```

Les règles permettent à l'auteur d'un module de ne pas s'occuper de la gestion globale des opérations qu'il réalise sur un module et de leur effet de bord. Le développeur peut s'occuper uniquement de son module et toutes les répercussions de ses actions seront prises en compte et réalisées par le système. D'autre part, tout module complétement testé est présenté au responsable pour un visa. Le responsable peut ainsi "oublier" un module tant que celui-ci n'est pas testé.

L'ordonnancement est :

```
CREER-MODULE ;
 [ RETARD-ENVISAGE ]
  [ DOCUMENTER-MODULE |  EDITER-MODULE ] ;
   [ EDITER ] ; [DOCUMENTER-MODULE] ;
    [ COMPILER-MODULE ; [ EDITER-MODULE | DOCUMENTER-MODULE ] ] ;
     [ COMPILER] ; [ [ TESTER-MODULE ] | VISER-MODULE ] ;
       VISER-MODULE ; EXPORTER-MODULE
```

Cet ordonnancement permet de documenter ou d'éditer un module créé, ou encore d'envisager un retard pour celui-ci, de compiler un module édité, de tester ou de viser un module compilé et d'exporter un module visé.

La caractéristique est :

```
SPECIFIE PAR (module, document)
 ⇒ date-de-dernière-mise-à-jour(document) ≥ date-de-dernière-mise-à-jour(module)
```

Cette caractéristique impose qu'un module soit toujours documenté après sa dernière édition.

3.5 La mise en œuvre de l'exemple

La mise en œuvre de cet exemple a été réalisée en le_lisp sur Sun 3/60. Nous avons réalisé le prototype **PIRE** (Piloting using an Infering Rule Engine) dont l'objectif est de montrer l'intérêt d'une modélisation en termes de MASP. Le système réagit aux requêtes de l'utilisateur comme pourrait réagir le système ALF. Le but de cette mise en œuvre n'était pas de proposer un système basé sur PCTE (but du projet ALF lui-même), mais de montrer sur un exemple simple, mais propre au génie logiciel le comportement d'un système prenant en compte une méthode de travail à travers une modélisation en termes de MASP.

Nous avons défini un certain nombre de fonctions génériques permettant de construire notre schéma E-A de travail. En particulier, la définition d'un type d'entité ou d'un type d'association génère toutes les fonctions de création, mise à jour, extraction d'informations des instances de ce type d'entité ou d'association. Nous avons structuré notre schéma E-A en une hiérarchie de types, chaque type héritant de ses types ancêtres l'ensemble de leurs attributs (pour ce faire, nous avons utilisé les facilités objet de la version 15.2 de le_lisp). Par exemple MODULE et DOCUMENT sont deux sous-types du type FICHIER : ils partagent en effet, comme nous l'avons décrit dans notre modèle, quelques mêmes types d'attributs spécifiques au type FICHIER.

```
;création de la classe ancêtre entity pour toutes les entités.
;Cette classe ne correspond pas à un élément du modèle présenté et
;n'est qu'un moyen "d'ancrage" pour l'implantation de notre
;hiérarchie  de type
(defstruct entity)
(setq entity.p (#:entity:make))

(def-entite 'fichier ;nom
            '(nom date-de-creation date-de-derniere-mise-a-jour) ; attributs
            'entity) ; père

(def-entite 'document
            '()       ; pas de nouveaux attributs autres que ceux
            'fichier) ; du type père fichier

(def-entite 'module
            '(compile?
              teste?
              retarde?
              date-de-compilation
              nombre-de-tests-prevus
              nombre-de-tests-realises) ; nouvelle liste d'attributs
            'fichier)                   ; à rajouter à ceux de fichier
```

Ces définitions de types d'entités génèrent les fonctions le_lisp (fichier ...), (module

...) et (document ...) qui permettent de créer et de mettre à jour des entités de type FICHIER, MODULE et DOCUMENT; et les fonctions (date-de-compilation ...), (compile? ...),... qui permettent d'accéder aux valeurs des attributs date-de-compilation, compilé?,... des instances du type module.

Pour les associations, nous avons utilisé le même procédé en y rajoutant la possibilité de définir des rôles.

```
;création de la classe ancêtre association pour toutes les associations
(defstruct association)
(setq association.p (#:association:make))

(def-associer 'specifie_par
              '((&module module ) ; &module est le rôle que joue l'entité
                (&document document)) ; module dans specifie_par
              'association)
```

Ces définitions de type d'associations génèrent la fonction (specifie_par ...) qui permet de créer et de mettre à jour des associations de type SPECIFIE PAR, entre des instances de MODULE et des instances de DOCUMENT. Notons qu'une association peut, elle aussi, avoir des attributs (comme c'est le cas pour l'association SUBIT avec l'attribut note).

Toutes ces fonctions génériques nous fournissent alors un environnement de base pour réaliser les autres composants d'un MASP.

Les expressions pour exprimer les préconditions, les postconditions, les parties gauches des règles et la caractéristique ont, elles, été définies comme une conjonction de s-expressions le_lisp, les termes de base étant essentiellement les fonctions génériques présentées.

Nous avons associé à chaque outil défini dans le MASP une fonction dont le corps réalise l'effet de bord attendu par cet outil.

Les opérateurs sont décrits de la manière suivante :

```
(operateur '(nom          ;nom de l'opérateur
             expression   ;exprimant la précondition
             body (fonction le_lisp) ; corps de l'opérateur
             post expression ;exprimant la postcondition
             fin))
```

Les règles ont été exprimées sous la forme.
(si expression alors fonction)

Nous avons écrit un petit moteur d'inférence en profondeur d'abord d'ordre 0+ permettant la manipulation de variables dans les règles de productions (ces variables sont en fait gérées par le_lisp d'où l'ordre 0+ et non pas 1). Ce moteur utilise une

base de règles composée de l'ensemble des règles de notre modèle et une base de fait composée de l'ensemble des objets de la base d'objets. Nous utilisons donc les règles de notre modèle comme règles d'inférence.

L'algorithme pour la prise d'initiatives

base-de-fait ← L'ensemble des objets de la base d'objets
base-de-règles ← L'ensemble des règles
prise-d'initiatives (base-de-fait)
Début
une-règle ← choix-règle (base-de-règles)
Si une-règle ≠ nil
Alors
déclencher-action(membre-droit(une-règle)) qui modifiera base-de-fait
prise-d'initiatives (base-de-fait)
Fin-si
Fin
choix-règle (base-de-règles)
Début
règle ← première-règle (base-de-règles)
Tant qu'il existe des règles et évaluer(membre-gauche (une-règle) = faux
Faire
règle ← règle-suivante (base-de-règles)
Fait
S'il n'existe plus de règle
Alors retourner(nil)
Sinon retourner(règle)
Fin-si
Fin

L'ordonnancement a, lui, été représenté de manière "ad hoc" comme une précondition spéciale à chaque opérateur. Nous avons traduit l'ordonnancement, exprimant des contraintes à respecter quant aux chemins d'exécutions autorisés, sous forme de préconditions pour chaque opérateur. Chacune de ces préconditions exprime une condition supplémentaire à vérifier avant d'autoriser l'exécution d'une opération.

Nous pourrions résumer le comportement de la maquette à travers les deux schémas suivants (voir figures 3.4 et 3.5).

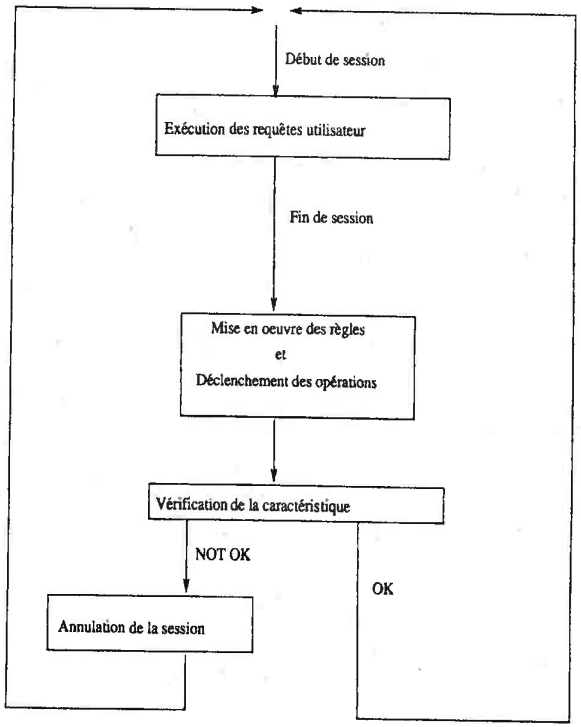


Figure 3.4. La représentation du PILOTE

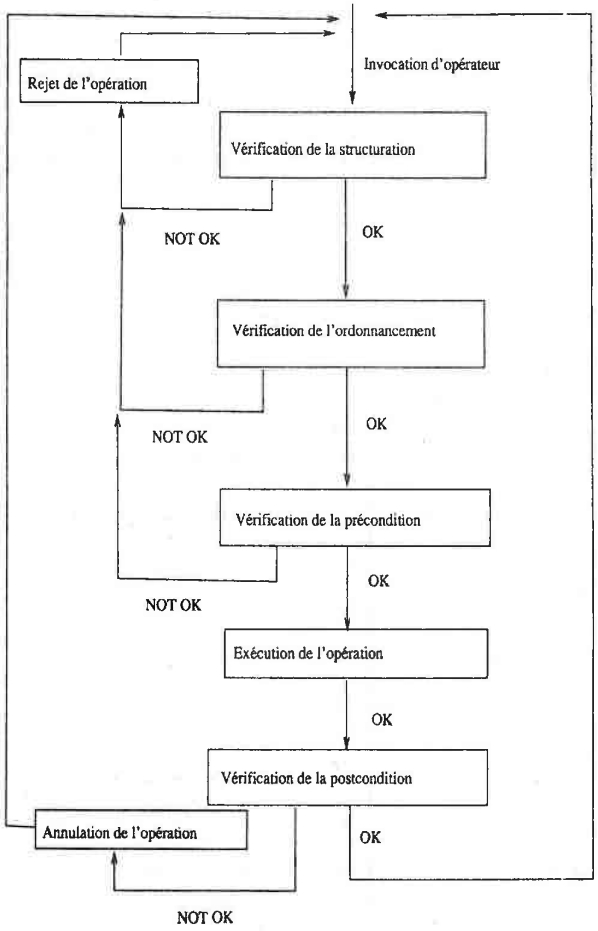


Figure 3.5. Le traitement des OPERATEURS

3.6 conclusion

Nous avons utilisé pour décrire nos exemples de MASP la grammaire la plus simple possible. D'autre part, nous n'avons pas donné dans cette thèse une syntaxe complète du langage de description de MASP. En effet, nous sommes encore, au sein du projet ALF, dans la phase de définition de cette syntaxe. Nous pouvons cependant dire en quelques phrases, à quoi ressemblera la syntaxe de chacun des composants. Le lecteur intéressé se rapportera à [BEN89c], version actuelle de la syntaxe du langage de description de MASP. Toutes les descriptions de la grammaire données ci-dessous sont tirées de [BEN89c].

Le **modèle d'objets** d'un MASP sera une liste de définition, d'utilisation ou d'extension de SDSs de PCTE. Il nous a en effet semblé plus sage dans un premier temps de ramener au moins l'un des composants à un modèle existant et permettre ainsi un travail sur des bases stables pour les autres composants.

```
<s_more>
::= ',' <sds_name> [ <s_more> ]
  | ';' <sds_definition> [';' <sds_indication_list> ]
  | ';' <sds_extension> [';' <sds_indication_list> ]
```

Pour les **expressions**, nous avons séparé la partie événement de la partie logique représentant une connaissance statique de la manière suivante :

```
<expression>
::= <identifier> ':' [ ON <event_indication_list> ]
    [ EVALUATE <expression_definition> ] ';'
```

Les **événements** pris en compte sont les événements pouvant être détectés lors d'une session de travail sur PCTE comme "read_situation", "lock_situation",etc... La partie **logique** est, elle, exprimée dans un langage de la logique des prédicats du premier ordre dont les termes sont les objets de base du système de gestion d'objets de PCTE.

Le langage logique défini pour les expressions est utilisé pour exprimer les préconditions et postconditions des **opérateurs**. Nous n'autorisons pas d'événement dans les pré et les post des opérateurs. Les IN et OUT des opérateurs sont, eux, décrits en termes de types de base de l'OMS de PCTE.

Les **règles** sont décrites sous forme d'une association entre une expression (comprenant partie logique et partie événement) et la désignation d'un opérateur.

L'ordonnancement est, lui, décrit en utilisant les notations introduites précédemment dans nos exemples ([] pour une répétition, (* *) pour une option,etc...).

La caractéristique est, elle, décrite comme une liste d'expressions.

Bilan

Nous avons adopté dans cette thèse une démarche progressive qui nous a paru refléter le mieux la façon dont nous avons travaillé. Nous avons traité successivement dans cette thèse les deux aspects qui nous ont semblé essentiels en vue d'une progression significative vers une industrialisation de la production de logiciels.

Quitter l'aspect artisanal de cette production pour produire des logiciels de manière industrielle en permettant à des intervenants différents et nombreux de coopérer implique de permettre le partage des **informations communes et cohérentes** sur les objets manipulés, que ce soit pour les consulter ou les produire.

Gestion de l'information en Génie Logiciel

Couplage Système d'Information / Système de développement

Le partage d'informations communes entre un grand nombre d'utilisateurs est la spécificité principale des Systèmes d'Information (SI).

Nous avons vu dans le chapitre 1 de la partie II ce que pouvait apporter à la production de logiciels, le couplage d'un SI à un système d'exploitation : *"un système pour conserver les informations utiles d'un projet de logiciel"* [MEY85b] et *"permettre un accès et une manipulation simples des données stockées"* [PEN86]. Cet accès et cette manipulation simples des données passent par une modélisation des données stockées. Nous avons vu le choix des projets SKB et PMDB pour une modélisation basée sur le modèle Entité-Association, s'appuyant sur une **"représentation"** des données réellement manipulées lors de la production de logiciels.

C'est cette représentation, dissociée de la représentation effective des objets logiciels (fichiers de documentation, fichiers de code exécutable, modules,...) qui nous a amené à critiquer ce couplage entre un système de développement et un système de gestion des informations associées à ce développement (cf. § 1.4 du chapitre 1 de la partie II). Ce couplage est source d'incohérence et est loin de réaliser le partage d'informations communes et cohérentes entre les intervenants dans la production

de logiciels. Ce n'est plus un partage d'informations si les informations manipulées pour la gestion d'un projet sont propres aux gestionnaires et celles utilisées pour le développement sont propres aux développeurs. Ce couplage, intéressant au point de vue fonctionnalités d'interrogations et de manipulations des informations relatives à un projet, pose donc le problème de la cohérence entre les informations réelles du système de développement et leur représentation dans le SI.

Intégration des fonctionnalités de gestion et de développement au sein d'un même atelier

La manipulation d'informations, le développement et l'exécution de programmes nécessitant des outils, nous avons vu dans le chapitre 2 de la partie II comment la construction d'un atelier paramétré pouvait répondre à ce besoin. L'outil primordial de Concerto pour la gestion de l'information est le système d'information construit autour du langage de définition et de manipulation de données F1 (cf. § 2.1.4 du chapitre 2 de la partie II). Les outils pour le développement de logiciels sont, eux, centrés autour d'une représentation interne (RI) arborescente des objets manipulés, d'un méta-formalisme (PPML et Metal) permettant de décrire les différents formalismes utilisés dans l'atelier et des fonctions génériques de manipulation associées à ce méta-formalisme (cf. § 2.1.3 du chapitre 2 de la partie II). Le méta-formalisme et le système d'information permettent d'arriver à un atelier paramétré par les langages de développement et par l'organisation du projet. Le couplage entre les informations réelles et leur représentation dans le SI reste lâche et l'atelier Concerto tout en étant intégré par sa RI unique et son méta-formalisme ne propose pas de solution satisfaisante, à nos yeux, pour la gestion des informations associées au développement de logiciels.

En outre, le coût d'un tel type d'Atelier, et la nécessité d'en développer un par type d'application nous a conduit à considérer l'utilisation d'une structure d'accueil, basée sur un système de gestion d'objets, commune à tous les types d'application comme la meilleure solution à la production de logiciel (cf. chapitre 3 de la partie II).

Les Structures d'accueil et leur Système de Gestion d'Objets

L'un des buts des Systèmes d'information est d'automatiser au maximum le traitement des données. Pour ce faire, il faut d'abord représenter ces données sur un support informatique qui permettra cette automatisation. S'il est concevable de gérer des informations sur un support informatique (Système d'Information) différent du support effectif de ces informations dans le monde réel, cela ne l'est plus dès que le support des informations du monde réel est déjà informatique. En effet, lors de la

production de logiciels, la quasi-totalité des données manipulées (fichiers) sont déjà sur un support informatique et possèdent donc déjà une représentation permettant une automatisation des traitements qui leur sont appliqués.

Il faut donc écarter toute approche qui séparerait les objets informatiques et leur représentation, source d'incohérence et de redondance.

Cependant, jusqu'à présent les traitements en vue du développement de logiciels ont été priviligiés par rapport aux traitements en vue de la gestion. Compiler un programme est possible simplement sur tous les systèmes mais retrouver le dernier programme compilé nécessite de la part de l'utilisateur une séquence d'opérations non triviale. Par ailleurs, tous les systèmes d'exploitation actuels (DOS, UNIX, Gcos6, Multics, etc) utilisent une structuration hiérarchique des fichiers censée résoudre tous les problèmes de structuration des informations.

La pauvreté de cette représentation hiérarchique est l'une des raisons de la difficulté à effectuer des traitements de gestion sur les objets informatiques. Les raisons de cette difficulté ne manquent pas :
la liste des informations associées à un fichier est toujours figée (*st_uid*, *st_gid*,... sous Unix) et l'utilisateur n'a aucune latitude pour décider qu'il serait intéressant d'associer à un fichier contenant un programme un attribut décrivant le langage dans lequel est écrit ce programme ;
pour accéder à la valeur d'un attribut, il faut écrire un programme,....

Toutes ces considérations ont amené à se poser le problème de la représentation des objets informatiques permettant à la fois le développement et la gestion. Le Système de Gestion d'Objets de la structure d'accueil PCTE a été la solution à ce problème. La structuration des objets ne se fait plus seulement de manière hiérarchique, mais grâce à un modèle Entité-Association (cf. § 3.2 du chapitre 3 de la partie II). En ce qui concerne CAIS, son SGO est basé sur un modèle nodal qui n'est pas très éloigné de la représentation hiérarchique des fichiers mais dont la caractéristique essentielle est la manipulation d'arbres mixtes comprenant à la fois des fichiers et des processus (cf. § 3.4 du chapitre 3 de la partie II). Ces Systèmes de Gestion d'Objets sont la pierre angulaire des structures d'accueil, même si celles-ci proposent d'autres composants comme la distribution ou une interface normalisée. Ces structures d'accueil permettront le développement d'AIGLs à moindre coût, étant donné que les mécanismes de base au développement et à la gestion sont intégrés à la structure d'accueil. Ces AIGLs coûteront moins et seront meilleurs que les actuels grâce à la représentation unique des informations servant soit à la gestion soit au développement.

Nous avons vu à travers toute la partie II différentes approches pour la prise en

compte des informations statiques associées au développement de logiciels. Nous avons synthétisé toutes nos remarques sur ces approches dans les tableaux du chapitre 4. La conclusion générale à tirer de cette deuxième partie est d'une part la nécessité d'une gestion complète et cohérente des informations statiques de la production de logiciels, et d'autre part notre intime conviction (fondée sur une comparaison détaillée) que la solution en ce qui concerne cette gestion passe par le SGO des structures d'accueil et plus particulièrement par celui de PCTE.

Cependant dans tous les projets que nous avons étudiés, les seules informations traitées de manière satisfaisante sont les informations statiques. Une modélisation de celles-ci est réalisée et leur utilisation doit se faire à travers ce modèle. Nous pouvons comparer cette modélisation et son utilisation à la représentation des objets logiciels sous la forme de photographies instantanées, chaque photographie représentant un état du système logiciel. Or le processus de production de logiciels est dynamique et sa représentation ne peut se limiter à la représentation des différents états que traverse le logiciel. La représentation d'un processus doit s'apparenter à un film et non pas à une série de photographies. Il faut pouvoir représenter la dynamique du processus de production de logiciels.

Processus de production de logiciels

Nous avons dans la partie III, en raison des lacunes des structures d'accueil en matière de modélisation de la dynamique (cf. chapitre 1 de la partie III), proposé dans le chapitre 2 de cette partie une solution pour la représentation et donc pour la prise en compte de cet aspect dynamique. Cette représentation est basée sur un certain nombre d'études de projets s'intéressant à la représentation de cette dynamique (cf. chapitre 1 de la partie III). Nous avons délibérément choisi de faire cette étude de la dynamique sur d'autres projets que ceux de la partie II. En effet, nous n'avons pas trouvé de projets traitant de manière satisfaisante à la fois la représentation des aspects statiques et la représentation des aspects dynamique en production de logiciels. Il est à noter cependant que des projets comme PMDB (cf. chapitre 1 de la partie II) commencent à s'intéresser à cet aspect et que PMDB+, suite envisagée de PMDB, se pose le problème de cette représentation dynamique.

Etat de l'art

Les projets étudiés dans le chapitre 1 de cette troisième partie sont de trois types :

- des projets comme Remora et Merise s'intéressant à la dynamique dans les systèmes d'information (cf. § 1.7.1) qui nous ont permis, de par leur ancienneté relative, de nous conforter dans notre conviction du besoin réel d'une modélisation du processus de production de logiciels, même si ce type d'approche ne convient pas tout-à-fait au Génie Logiciel;
- des projets comme Arcadia se basant sur l'utilisation de langages de programmation pour programmer des processus de production de logiciels (cf. § 1.7.2) avec les contraintes que cela implique (programmation dynamique souhaitable, mélange programmation dynamique et procédurale,...);
- et enfin des projets comme Istar (cf. § 1.7.3) ou Marvel (cf. § 1.7.4) cherchant à "modéliser" les processus de production de logiciels, même si cette modélisation est incomplète (pas de structuration des activités dans Marvel, pas de modélisation "à priori" mais seulement au fur et à mesure dans Istar,...).

Le modèle

Nous nous sommes inspirés des avantages de chacun de ces projets pour proposer un modèle pour le processus de production de logiciels (cf. chapitre 2 de la partie III).

Le modèle proposé (**MASP** : Model for Assisted Software Processes) est inspiré du modèle que nous développons actuellement dans le cadre de ALF. Ce modèle est, comme nous l'avons vu, composé de six éléments :

- un modèle d'objets,
- un modèle d'opérateurs,
- un ensemble d'expressions,
- un ordonnancement,
- un ensemble de règles,
- une caractéristique.

Nous avons ensuite présenté une maquette développée en le_lisp et basée sur les concepts présentés (cf. chapitre 3 de la partie III). Celle-ci a permis de se rendre compte "de visu" de certains aspects du comportement du système ALF et a permis de mettre à jour un certain nombre de problèmes qu'il faudra résoudre lors de l'implantation finale du système ALF ("bouclage" possible des règles, cohérence entre les différentes connaissances associées aux 6 composants du modèle,...).

Si les premières étapes de recensement des besoins en modélisation de la dynamique dans les systèmes de développement, et la conception de la solution MASP ont été menées à leur terme, il reste à concrétiser cette solution par une implantation sur PCTE qui permettrait d'évaluer et d'améliorer le modèle. Cette implantation se fera lors des deux prochaines années dans le cadre du projet ALF.

Conclusion

Nous sommes conscient que nous venons juste d'initialiser l'itération permettant d'arriver à un modèle satisfaisant pour les processus de production de logiciels. Le MASP est au modèle de processus de production de logiciels "parfait" ce que Fortran est au langage de programmation "idéal" : une étape essentielle par laquelle il faut passer pour apprendre mais à laquelle il ne faut surtout pas s'arrêter. La synthèse faite par Dowson dans [DOW86b] est tout à fait pertinente et s'applique à la perfection à la démarche que nous avons suivie et que nous comptons poursuivre dans les années à venir. *L'expérimentation de programmes de processus, ou d'autres formes de représentation exécutable du processus de production de logiciels sera bénéfique à court terme. Nous commençons à mieux comprendre le processus logiciel comme un tout, et cette expérimentation accélérera cette compréhension car elle rend le processus logiciel explicite et testable.* [1]

1. "Experimentation with process programs, or other forms of executable representation of the software process will be a fruitful area in the short term. We are beginning to arrive at a much better understanding of the software process as a whole, and this work is likely to accelerate the development of that understanding by providing a means of making it explicit and testable."

Appendice A
Bibliographie

3SPW: M Dowson (éditeur)
"Proceedings of the third International Software Process Workshop"
Breckenridge, Colorado, US, édité par M.Dowson, Novembre 1986

4SPW: ACM/IEEE (éditeur)
"Proceedings of the fourth International Software Process Workshop"
Moretonhampstead, Devon, UK, May 1988

ALF87: ALF Consortium.
"Advanced Software Engineering Logistics Framework"
Technical Annex, 1987.

ALL86: F. Allez, L. Boi, M. Bosco, S. Benoit, Y. de la Motte Collas
"Le poste de pilotage de l'atelier Concerto"
Concerto, Présentation technique, Perros-Guirec, Février 86

AND86: André E., Moreau B., Rougeot B.
"Vers un atelier flexible et intégré de logiciel : le projet CONCERTO"
Concerto, Présentation technique, Perros-Guirec, Février 86

AMB88: J. Ambras and V. O'Day.
"Microscope : a knowledge-based programming environment"
IEEE Software, May 1988.

ARD89: J.N. Ardot, G. Pujolle, A. Test
"Une interface relationnelle de type hypertexte"
Actes du congrès Inforsid , Nancy, mai 1989

AUT85: Y. Autret
"Ulysse, une infrastructure logicielle pour la communication"
Thèse d'état, Rennes I, 1985

BAR81: A. J. Baroody Jr, D. J. Dewitt.
"An object oriented approach to DBMS implementation."
ACM TODS Vol 6. No 4. Dec 81.

BAR86: Barbuceanu M.
"Knowledge based development and reusable software"
6èmes journées internationales sytèmes experts et applications, Avignon Avril 8

BEL86: Belkhatir N.; Estublier J.
"Coopération et protection dans un environnement de Génie Logiciel"
3ème Colloque de Génie Logiciel, Versailles, Mai 86

BEN87: Benali K., Derniame J.C., Godart C.
"Système d'information et génie logiciel."
Congrès Inforsid, Lyon, Juin 87

BEN88: K. Benali
"Piloting with infering rules expressions : PIRE"
Rapport ALF-NCY-KB-WPP-?-1.A, janvier 1988

BEN88b: K. Benali
"Presentation of F1, F1 and ALF"
Rapport ALF-NCY-KB-WP1-1-1-D1-1.1, avril 1988.

BEN89a: K. Benali, N. Boudjlida, F. Charoy, JC.Derniame
"A Model for Assisted Software Processes"
International Conference on Computing and Information
Canada Scholar Press, Laurantian, (Ontario, Canada), may 1989

BEN89b: K. Benali, N. Boudjlida, F. Charoy, JC.Derniame, C. Godart
P. Griffiths, V. Gruhn, P. Jamart, A. Legait, D.E. Oldfield, F. Oquendo
"Presentation of the ALF Project"
International Conference on System Development Environments and
Factories, (SDE&f), Berlin, RFA, May 1989

BEN89c: K. Benali, N. Boudjlida, F. Charoy, V. Gruhn, F. Oquendo
"Syntax of the MASP Description Langage"
Rapport ALF-VG-UDO-WP2-1-4-D1-2.2, september 1989.

BOC86: J. Bocca, H. Decker, J-M. Nicolas, L. Veille, M. Wallace
"Some steps towards a DBMS based KBMS"
Proceedings of the 10th World Computer Congress IFIP, Dublin - September 86

BOE82: Barry W. Boehm
"Les facteurs du coût du logiciel"
T.S.I. Vol. 1, no 1, 1982

BOE88: B. Boehm, F. Belz
"Applying process programming to the spiral model"
in [4SPW]

BOR88: A.Borgida, S.Greenspan, J.Mylopoulos
"Knowledge Representation as the Basis for Requirements Specification"
Computer, 1985

BOU84: N. Boudjlida.
"Un système expérimental d'aide à la spécification et à l'utilisation de bases de données déductives"
Thèse Docteur ingénieur INPL Nancy , Mai 84.

BOU85: N. Boudjlida, J.J. Chabrier
"Bases de données déductives et types abstraits de données : le système expérimental SINDBAD"
Actes du congrès Matériels et logiciels de 5ème génération
Paris - Mars 1985

BOU87: N. Boudjlida, C. Godart, J.C. Derniame
"Database requirements for environment kernels"
Rapport CRIN 87-R-095

BOU88: N. Boudjlida, O. Gervaise, C. Godart, J.C. Derniame
"Méthode de développement et pilotage : des concepts et une maquette"
Rapport CRIN

BOU88b: N. Boudjlida and O. Gervaise.
"Concepts and experiments in intelligently-assisted software development"
Rapport CRIN, Septembre 1988.

BOU88c: N. Boudjlida, C.Godart, JC.Derniame, K. Benali, O. Gervaise.
"Vers des ateliers de logiciel supportant des méthodes"
Quatrièmes Journées Internatinales des Sciences Informatiques
(JISI '88), Tunis, avril 1988

BOZ86: M. Bouzeghoub
"Les contraintes d'héritage dans les hiérarchies de généralisation"
Actes des journées FIRTECH. Systèmes et télématique.
Bases de données et intelligence artificielle. PARIS 9-10 april 1987.

BRA76: G. Bracci, P. Padini, and G. Pelagatti
"Binary logical associations in Data modeling"
Modeling in Data Base Management Systems,
IFIP working conference in modeling on DBMS's, ed. Nijssen 1976.

CAI85: CAIS
"Kit/Kitia, Military Standard Common APSE Interface Set (CAIS)"
Proposed MIL-STD-CAIS, US DoD, January 1985

CAI85a: CAIS
"Rationale for the DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)"
Institute for Defense Analyses, US DoD 1801 North Beauregard Street,
Alexandria, Virginia 22311, 13 September 1985

CAI86: CAIS
"CAIS Rationale"
Institute for Defense Analyses, US DoD 1801 North Beauregard Street, Alexandria, Virginia 22311, 11 April, 1986

CAI88: CAIS-A
"Proposed Military Standard - Common ADA Programming Support Environment (APSE) Interface Set (Revision A)"
SofTech, Revision 15, Volumes I - II,and III, 12 January 1988

CAN86: G. Cany
"Apotre : un système d'accès poste de travail sous UNIX"
Concerto, Présentation technique, Perros-Guirec, Février 86

CAR87: P. Carr and al.
"Implementation of a prototype CAIS environment"
Ada letters, vol 7 no 2 Mars 1987

CAZ86: Cazin J.; Jacquart R.; Michel P.
"Manuel de référence F1 version6"
Rapport 2/3239/DERI, Janvier 86

CHA82: J.J. Chabrier.
"Spécification et construction de systèmes orientés bases de données: techniques et langages basés sur le concept de type abstrait."
Thèse d'état Université de NANCY I

CHA86: J. Chailloux, M. Devin, F. Dupont, JM. Hullot, B. Serpette, J. Vuillemi
"Manuel de référence Le'lisp"
I.N.R.I.A., Mai 1986

CHA88: F. Charoy, C. Godart, R. Martz, W. Stulken
"Classification Properties of SEE Information Systems"
Rapport ALF/NCY-FC/WP-1/1/7.D3, octobre, 1988

CHA89: F. Charoy, C. Godart, J.C. Derniame
"Des Réflexes dans un Système à Gestion d'Objets"
Congrès INFORSID'89 Nancy, Mai 1989

CHA90: F. Charoy
"Vers une structure d'accueil orientée objet"
Thèse de L'université de Nancy I, à paraître en Janvier 1990

CHE76: Chen P.P.
"The entity-relationship model : Toward an unified view of data"
ACM-TODS 1,1 Mars 1976, pp 9-36

CHI86: Y. Chiaramella, D Kerkouba, M.F. Bruandet
"Intégration d'une fonction documentaire dans un atelier de logiciel"
Concerto, Présentation technique, Perros-Guirec, Février 86

CHR88: G. Chroust
"Models and Instances"
Software Engineering Notes vol. 13, July 88, no 3

COL82: A. Colmerauer
"PROLOG II : Manuel de référence et modèle théorique"
Groupe I.A., Marseille LUMINY, mars 1982

COL86: Colnet D.; Masini G.; Noiret Y.; Tombre K.
"Les langages orientés objets"
Rapport interne CRIN 86

COL89: Colnet D.
"Prototypage, programmation objet : application à l'édition structurelle"
Thèse de l'Université de Nancy I, Février 1989

CON86: Projet CONCERTO.
"Presentation technique"
4-5-6 fevrier 1986. CNET Lannion.

CON88: Conradi, M.Lacroix, E.Adhami, V.Ambriola et al.
"Initial Design of SESE, a Kernel Environment for Integrated Configuration and Process Management"
Proposal for an ESPRIT project, April 1988

CRO75: CROCUS
"Systèmes d'exploitation des ordinateurs"
DUNOD informatique, 1975

CRU86: S. Cruzlara Silva, J.C. Derniame
"Une approche pour la construction d'environnement de programmation orientés vers un langage particulier"
2ième Journée internationnales de l'informatique. Tunis - 1986

CRU88: S. Cruzlara Silva
"GEODE : Un système pour la génération d'environnements de programmation intégrés"
Thèse de l'Institut National Polytechnique de Lorraine. novembre 1988

DAH66: Dahl O.J., Nygaard K.
"SIMULA : An ALGOL based simulation language."
Communication of the ACM, Vol. 9, No 9, pp. 671-678, 1966

DAT83: Date J.C.
"An introduction to Database Systems"
Addison-Wesley , VOL 2, 3th edition 1983,

DAV88: A.M.Davis, E.H.Bersoff, E.R.Comer
"Strategy for comparing alternative software development life cycle models"
IEEE Transaction on Software Engineering, Vol14, No10, October 1988

DHA86: Dhar V.; Jarice M.
"Using the logical design knowledge for large systems development and maintenance"
6èmes journées internationales Sytèmes experts et applications,
Avignon, Avril 86

DEL82: C. Delobel, M. Adiba
"Bases de données et systèmes relationnels"
Dunod, 1982

DER79: J.C. Derniame et J.P. Finance.
"Types abstraits de données: spécification, utilisation et réalisation"
Ecole d'été de l'AFCET. 79-E-57.

DER88: JC Derniame, B. Ayoub, K. Benali, N. Boudjlida, C. Godart, V. Gruhn
"Toward assisted software process"
Proceedings CASE88, Boston - MA, July 1988.

DER89: J.C. Derniame and all,
"The MASP concept in the ALF project"
Rapport Esprit ALF/NCY-JCD/WP-2-1-2/D2 issue 2.0

DON75: V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J.J Levy
"A structure-oriented program editor : a first step towards computer-assisted programming"
Rapport de recherche Inria No 114, France, avril 1975

DOS82: W. Dosc, G. Mascari, M. Wirsing
"On the algebraic specification of databases"
8th VLDB - Mexico - September 1982

DOW86: M. Dowson.
"ISTAR: An integrated Project Support Environment"
SIGPLAN notices Vol. 22 NO. 1 pp 27-33 ACM.

DOW86b: M. Dowson.
"Workshop introduction and overview"
in [3SPW]

EME84: G.I.E. EMERAUDE
"Rapport intermédiaire"
Avril 84, Publication du G.I.E. Emeraude

ENT86: ENTREPRISE
"Atelier de logiciel intégré"
Manuel d'utilisation, juin 86, Publication du Groupe Syseca

FAI85: R. Fairley
"Software engineering concepts"
McGraw-Hill book company, 1985

FAR87: H. Farreny, M Ghallab
"Eléments d'intelligence artificielle"
Editions HERMES, 1987

FEL79: S. I. Feldman
"Make - a program for maintaining computer programs"
Sofware Practice & experience Volume 9, No 4, April 1979

FER86: C. Fernstrom
"Un environnement de schémas structurés sur le poste de travail Concerto."
Concerto, Présentation technique, Perros-Guirec, Février 86

FIR87: Actes des journées FIRTECH. Systèmes et télématique.
"Bases de données et intelligence artificielle."
PARIS 9-10 avril 1987.

FOR82: Forgy C.
"RETE : a fast algorihm for the Many Pattern/Many Object Match Problem"
Artificial Intelligence, No 19 pp17-37, 1982

FOU82: O. Foucaut
"Méthode et outil pour la conception des systèmes d'information dans les organisations : projet REMORA"
Thèse de doctorat d'état, Nancy, 1982

GAL78: H. Gallaire, J. Minker, J.M. Nicolas (éditeurs)
"Logic and databases"
Plenum Publishing Corporation - New York - 1978

GAL81: H. Gallaire, J. Minker, J.M. Nicolas (éditeurs)
"Advances in database theory I"
Plenum Publishing Corporation - New York - 1981

GAL82: H. Gallaire, J. Minker, J.M. Nicolas (éditeurs)
"Advances in database theory II"
Plenum Publishing Corporation - New York - 1982

GAL86: Gallaire M.
"Evolution du logiciel. Tendances et exemples"
3ème Colloque de Génie Logiciel, Versailles, Mai 86

GAL86b: F. Gallo, R. Minot, I. Thomas.
"The object Management System of PCTE as a Software Engineering Database Management System".
ACM SIGPLAN Notices, Vol.22 No 1 pp 12 à 15

GEN89: C. Genillard, N. Ebel, A. Strohmeier
"Rationale for the design of reusable abstract data types implemented in ADA"
Ada LETTERS, SIGAda, ACM press, march 1989

GER87: O. Gervaise
"Méthode de développement et pilotage : des concepts et une maquette"
Rapport de DEA - CRINancy - Sept 1987

GIP87: GIPE consortium
"Second annual review report"
Esprit Project 348, january 1987

GL1: Génie Logiciel
"Ateliers de génie logiciel"
Publication Génie Logiciel No 1, décembre 1984

GL13: Génie Logiciel et systèmes experts
"Les ateliers"
Publication Génie Logiciel et systèmes experts No 13, décembre 1988

GOD86: C.Godart, K. Benali, JC.Derniame.
"Propositions pour un système de gestion d'objets."
3ème colloque de génie logiciel. Versailles. 27-30 Mai 86.

GOD87: C.Godart, K. Benali, N. Boudjlida, F. Charoy, JC.Derniame.
"Les bases de données sur le chemin du génie logiciel".
Journées CERAM Des bases de données aux bases de connaissances.
Sophia Antipolis. 28-29 Sept. 87.

GOD87b: C.Godart, K. Benali, JC.Derniame.
"Proposition for a third generation environment kernel"
Rapport CRIN 87-R-033.

GOD88: C. Godart, P. Jamart
"ALF functionnalities"
Rapport ALF-NCY-CG-WP1-1-7, Janvier 1988

GOD88b: C. Godart, J.M. Bretnacher, P. Jamart, P. Griffiths, S. Psonis
"Working Group 1 Report"
Rapport ALF/NCY-CG/TC-WG1/1.0 - mars 1988

GOD89a: C. Godart, F. Charoy, and J.C. Derniame.
"Computer Assisted Software Engineering : Characterisation and Modeling."
Proceedings of International Conference on Computing and Information,
Canada Scholar Press, Ontario, CANADA, May 89.

GOD89b: C. Godart, F. Charoy, and J.C. Derniame.
"For an information system specific to software engineering"
Proceedings of Computer Aided software engineering'89, London, July 89.

GOO88: D. Goodman
"Hypercard et hypertalk : guide d'utilisation et de programmation"
Edition Dunod, 1988

GOL83: A.Goldberg. R.Robson.
"SMALLTALK80, the language and its implementation."
Addisson-Wesley. Reading. Massachusets 83.

GRE84: S.Greenspan
"Requirements Modeling :
A Knowledge Representation Approach to Software Requirements Definition"
Technical Report CSRG-155, University of Toronto, March 1984

HAL87: F.G. Halsz, T.P. Moran, T.H. Trigg
"NoteCards in a nutshell"
ACM conference on human factors in computing system, Toronto, Canada, avril 1987

HOW86: I. Howe
"Sybase data integrity for on-line applications"
Sybase Inc. 1986

HUL85: J.M. Hullot.
"ALCYONE, la boite à outils objets"
Rapport de Recherche no. 60, INRIA Rocquencourt, 1985.

HUM88: W.S Humphrey
"The software engineering process : definition and scope"
in [4SPW]

IDA86a: IDA
"Atelier logiciel IDA d'aide à la conception de systèmes d'informations"
Manuel de référence : DSL-SPEC, release 2.0R0, version française, mars 1986

IDA86b: IDA
"Atelier logiciel IDA d'aide à la conception de systèmes d'informations"
Manuel de référence : DSL, release 2.0R0, version française, mars 1986

JAC83: M. Jackson.
"System Development"
Series in Computer Science, Prentice-Hall International, 1983.

KAH83: G. Kahn, B. Lang, B. Melese, E. Morcos
"Metal : A formalism to specify formalisms"
Rapport de recherche INRIA, 1983

KAI87: G.E. Kaiser and P.H. Feiler.
"An architecture for Intelligent Assistance in Software Development"
Proceedings of 9th International Conference on Software Engineering,
Monterey CA, April 1987

KAI88: Gail E. Kaiser.
"Constructing enactable models"
Representing and enacting the software process
in [4SPW]

KAI88b: G.E. Kaiser, P.H. Feiler, and S.S. Popovitch.
"Intelligent Assistance for software development and maintenance".
IEEE Software, Vol. 5, No 3, May 1988

KAI88c: G.E.Kaiser
"Rule-Based Modelling of the Software Development Process"
in [4SPW]

KAI88d: G.E. Kaiser, N.S. Barghouti, P.H. Feiler, and R.W. Schwanke.
"Database support for knowledge-based engineering environment"
IEEE expert 1988.

KEB84: A. Kebaili
"Une contribution à l'étude du problème de l'évolution de structure dans les systèmes d'informations historiques Projet REMORA"
Thèse de troisième cycle, Nancy 1984

KEL88: M.I.Kellner
"Representation Formalisms for Software Process Modelling"
in [4SPW]

KEM88: M. KEMP
"PCTE"
Ecole Polytechnique fédérale de Lausanne
Rapport du travail pratique de diplôme, 23 décembre 1988

KEM89: M. KEMP
"Hyperbook : an experiment with PCTE"
ACM SIGSOFT Software Engineering Notes vol 14 no 5 july 1989 pp 55-61

KIM86: Kim M.J.; Tounsi N; Derniame J.C.
"Cohérence d'un schéma de base de données de type Entité-Association"
INFOCOM'86, Paris, 1986

KIM86b: Kim M.J.
"Etudes sur l'approche base de données pour la gestion d'objets issus du developpement de logiciel"
Thèse de troisième cycle, Nancy I, mai 1986

KRA85: S. Krakowiak
"Principes des systèmes d'exploitation des ordinateurs"
DUNOD informatique, Paris, 1985

LAN86: A.V Lansweerde, M. Buyse, B. Delcourt, E. Delor, M. Ervier, M.C Schayes
"The kernel of a generic software development environment."
SIGPLAN Notices, Vol.22 No 1 pp 208-227 ACM.

LEG88: A. Legait
"Rationale For the System Requirements"
Rapport ALF/EMR-LGT/WP-1/3/1.D2, novembre 1988

LEH87: M.M. Lehman, W.M. Turski.
"Essential properties of IPSEs."
ACM SIGSOFT.Software Engineering Notes. Vol 12 No 1. Janv 1987. p52.

LEH87b: M.M.Lehman
"Process Models, Process Programs, Programming Support"
Proceedings of 9th International Conference on Software Engineering,
Monterey CA, April 1987

MAR88: R. Martz.
"PILOTAGE DANS LES ATELIERS DE GENIE LOGICIEL -L'Apport des Langages à Base de Frames-"
Rapport de DEA - CRINancy - septembre 1988

MEY85: B. Meyer
"Etapes sur le chemin du génie logiciel"
Thèse d'état, Nancy I, 9 septembre 1985

MEY85b: Meyer B.
"The software knowledge base"
8th international conference of software ingineering, pp 158-165, London 85

MOR86: H. Morcos, M.J. Brossard, A. Conchon
"Affichage interactif d'arbres abstraits"
Concerto, Présentation technique, Perros-Guirec, Février 86

MOT86: Y. de la Motte Collas
"Prysme méthode et modèle de gestion de projets"
Concerto, Présentation technique, Perros-Guirec, Février 86

NEU86: P. L. Neumann
"Ulysse, système et langage d'implémentation de l'atelier Concerto."
Concerto, Présentation technique, Perros-Guirec, Février 86

NIL71: N.J. Nilsson
"Problem solving methods in artificial intelligence"
Mac Graw Hills comp. ser. 1971

NIL82: N.J. Nilsson
"Principle of artificial intelligence"
Springer Verlag - 1982

OST87: L. Osterweil.
"Software Processes are Software Too"
Proceedings of 9th International Conference on Software Engineering, Monterey CA, April 1987, pages 2--13,

OST88: L.Osterweil
"Automated Support for the Enactment of Rigorously Described Software Processes"
in [4SPW]

PAC86: PACT Consortium
"PACT general description"
Programme ESPRIT, Septembre 86

PAC88: PACT Consortium
"The PACT Tool Writer's Guide"
Programme ESPRIT, 1988

PCT86: PCTE Consortium
"A basis for a portable common tool environnement : Functionnal Specifications"
Programme ESPRIT, 4ème edition, 1986

PCT88: PCTE Consortium
"PCTE+ functional specification, C binding"
Issue 2, juillet 88

PEN85: M.H Penedo, E.D. Stuckle.
"A Project Master Database for Software Engineering environments"
8th international conference of software ingineering, London 85

PEN86: M.H Penedo.
"Prototyping a Project Master Database for Software Engineering Development"
SIGPLAN Notices Vol.22 No 1 pp 1-11 ACM.

PER88: D.E. Perry and G.E. Kaiser.
"Models of software development environments."
Proceedings of the 10th Int'l Conf. on Soft. Eng., Singapoor, March 1988.

REE87: K. Reed.
"Practical Software Engineering Environments: Report of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium."
ACM SIGSOFT.Software Engineering Notes. Vol 12 No 1. Janv 1987. p56.

REI84: R. Reiter
"Towards a logical reconstruction of relational theory"
On conceptual modelling - Brodie & al editors - North Holland Publishing - 198

RIC83: E. Rich
"Artificial Intelligence"
MacGraw Hills Series in Artifical Intelligence 1983

RID86: W.E.Riddle, L.G.Williams
"Modeling Software Development in the Large"
in [3SPW]

RIT73: H Rittel, W. Webber
"Dilemmas in a general theory of planing"
Policy sciences, vol 4, 1973

ROB88: C.Roberts
"Describing and Acting Process Models with PML"
in [4SPW]

ROL88: C. Rolland, O. Foucaut, G. Benci
"Conception des systèmes d'information : La méthode REMORA"
Editions Eyrolles, 1988

ROU86: B. Rougeot
"ISA : un environnement Concerto de gestion de l'espace de travail"
Concerto, Présentation technique, Perros-Guirec, Février 86

SAN86: M. Santana, J.C. Hochain
"Gemme : compilation multimachine"
Concerto, Présentation technique, Perros-Guirec, Février 86

SMI82: Smith J.M & D.C.P.
"Principles of database conceptual design"
Lecture Notes in Computer Sciences No 132, 1982

SOM85: I. Sommerville
"Software Engineering"
International Computer Science Series, Addison-Wesley, Second edition, 1985

STO76: M. Stonebraker & E. Wong & P. Kreps & G. Held,
"The Design and the Implementation of INGRES"
ACM Transactions on Database Systems, Septembre 1976

STU88: W. Stulken, F. Charoy, K. Benali
"Evaluation of Database Concepts"
Rapport ALF/DTD-WS/WP-1/1/7-D4 Octobre, 1988

TAR83: H. Tardieu, A. Rochfeld et R. Colletti.
"La méthode Merise, principes et outils".
Les éditions d'organisation - Paris - 1983

TAR86: H. Tardieu.
"Contribution of the entity relationship approach to object management in an information system design workbench." Tutorial 3,
5 th int. conference on entity relationship approach. Nov 1986. DIJON.
(une autre version de ce tutorial a pour auteurs H. Tardieu et M. Franckson)

TAY87: R.Taylor et al
"Next Generation Software Environments : Principles, Problems, and Research Directions"
Technical Report, University of California, Irvine, 1987

TAY88: R.Taylor, F.C.Belz, L.A.Clarke, L.Osterweil and al
"Foundations for the Arcadia Environment Architecture"
in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, appeared in Software Engineering Notes, Vol.13, No.5

THO88: I. Thomas
"Writing Tools for PCTE and PACT : The HOW-TO-DO-IT Guide"
Rapport Bull, 1988

THO88b: I. Thomas, G. Boudier, F. Gallo, R. Minot
An overview of PCTE and PCTE+
Proceedings of the ACM SIGSOFT/SIGPLAN Software Egineering Symposium on Practical Software Development Environments, November 1988

TIG85: TIGRE
"Gestion des historiques pour la base de données généralisée TIGRE"
Rapport TIGRE No 29 - 1985

WAR89: B. Warboys
"The IPSE 2.5 project : Process modelling as the basis for a Support Environment"
International Conference on System Development Environment & Factories Berlin, Mai 89

WAS82: A.I. Wasserman
"The User Software Engineering Methodology : an Overview"
Conference CRIS IFIP82

WAS85: A.I. Wasserman.
"Extending State Transaction Diagram for the specification of Human Computer Interaction".
IEEE Trans. Software Engineering Vol. SE-11 pp 699-713 - (Aug. 1985).

WIN84: P.H Winston, B.K.P Horn
"LISP"
Addison-Wesley 1984

ZAN84: G. Zanon
"Prolisp, manuel d'utilisation"
Rapport CERT/DERI, Septembre 1984
Concerto, Présentation technique, Perros-Guirec, Février 86

Appendice B

Chronologie des publications à des congrès dans le cadre de ce travail

GOD86: C.Godart, K. Benali, JC.Derniame.
"Propositions pour un système de gestion d'objets."
3ème colloque de génie logiciel. Versailles. 27-30 Mai 86.

BEN87: Benali K., Derniame J.C., Godart C.
"Système d'information et génie logiciel."
Congrès Inforsid, Lyon, Juin 87

GOD87: C.Godart, K. Benali, N. Boudjlida, F. Charoy, JC.Derniame.
"Les bases de données sur le chemin du génie logiciel".
Journées CERAM Des bases de données aux bases de connaissances. Sophia Antipolis. 28-29 Sept. 87.

BOU88: N. Boudjlida, C.Godart, JC.Derniame, K. Benali, O. Gervaise.
"Vers des ateliers de logiciel supportant des méthodes"
Quatrièmes Journées Internatinales des Sciences Informatiques (JISI '88), Tunis, avril 1988

DER88: JC Derniame, B. Ayoub, K. Benali, N. Boudjlida, C. Godart, V. Gruhn
"Toward assisted software process"
Publication au Second International Workshop on Computer-Aided Software Enginneering
(CASE '88), Cambridge (Boston, Massachusetts), Juillet 1988

BEN89a: K. Benali, N. Boudjlida, F. Charoy, JC.Derniame
"A Model for Assisted Software Processes"
International Conference on Computing and Information (ICCI '89) Canada Scholar Press, Laurantian, (Ontario, Canada), may 1989

BEN89b: K. Benali, N. Boudjlida, F. Charoy, JC.Derniame, C. Godart P. Griffiths, V. Gruhn, P. Jamart, A. Legait, D.E. Oldfield, F. Oquendo
"Presentation of the ALF Project"
International Conference on System Development Environments and Factories, (SDE&f '89), Berlin, RFA, May 1989

NOM DE L'ETUDIANT : **BENALI Khalid**

NATURE DE LA THESE : **Doctorat de l'Université de NANCY I en Informatique**

SCD NANCY

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 23 NOV. 1989 n° 2 297

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITE NANCY I
LE PRESIDENT
M. BOULANGE

Résumé

Cette thèse s'intéresse à la construction d'ateliers de développement de logiciels et en particulier à l'assistance et au pilotage dans ce développement. Pour atteindre cet objectif, nous nous sommes concentrés sur deux aspects qui nous semblent essentiels, à savoir le partage d'informations communes et cohérentes sur les objets manipulés et la modélisation du processus de production de logiciels.

Après une présentation de la problématique dans la première partie, la deuxième décrit les différentes façons de gérer et de partager les informations relatives au développement dans un atelier (en couplant un Système d'Information à un Système de Développement, en intégrant la fonction de gestion de l'information au sein d'un atelier intégré paramétré, ou encore par le biais d'un Système de Gestion d'Objets au cœur d'une structure d'accueil). La conclusion de cette deuxième partie est, d'une part la nécessité d'une gestion complète et cohérente des informations statiques de la production de logiciels, et d'autre part la conviction (fondée sur une comparaison détaillée) que la meilleure solution est la définition d'un Système de Gestion d'Objets au cœur d'une structure d'accueil du type PCTE.

La troisième partie montre les lacunes des structures d'accueil actuelles en matière de modélisation de la dynamique de la production de logiciels, puis propose une solution pour la représentation et la prise en compte des aspects dynamiques dans le processus de production de logiciels. Un certain nombre d'études de systèmes décrivant le processus de production nous a amené à la proposition d'un modèle qui nous est propre pour le processus de production assistée de logiciels : le MASP (Model for Assisted Software Processes). Ce modèle est composé de six éléments : un modèle d'objets, un modèle d'opérateurs, un ensemble d'expressions, un ordonnancement, un ensemble de règles et une caractéristique.

Pour terminer, une maquette est présentée. Elle est développée en le_lisp et est basée sur les concepts cités ci-dessus. Son objectif est de permettre de se rendre compte "de visu" de certains aspects du comportement d'un système basé sur un tel modèle.