Un modèle générique pour la gestion des informations complexes et dynamiques :

Sahar Jarwah

THESE

présentée par

Sahar JARWAH

pour obtenir le titre de **docteur**

de l'Université Joseph Fourier - Grenoble I

(arrêté ministériel du 5 juillet 1984)

spécialité **INFORMATIQUE**

Un Modèle Générique pour la Gestion des Informations Complexes et Dynamiques

Gestionnaire d'Objets du Prototype ELEN Dédié au Génie Logiciel

date de la soutenance : 28 avril 1992

COMPOSITION DU JURY :

Président :	M. Jacques MOSSIERE
Rapporteurs:	M. Patrick BOSC
	M. Jean-Claude DERNIAME
Examinateurs:	M. Michel ADIBA
	Mme. Marie-France BRUANDET
	M. Yves CHIARAMELLA

Thèse préparée au sein du Laboratoire de Génie Informatique
à l'Université Joseph Fourier-Grenoble I

Je tiens à remercier,

Monsieur Jacques Mossière, Professeur à l'INPG, Directeur de l'ENSIMAG, de m'avoir fait l'honneur de présider le jury de cette thèse.

Monsieur Patrick Bosc, Professeur à l'ENSSAT (Lannion), qui a bien voulu rapporter sur ce travail. Qu'il trouve ici toute ma reconnaissance pour sa lecture soigneuse de ce document et pour ses remarques qui m'ont été fort utiles.

Monsieur Jean-Claude Derniame, Professeur à l'Université de Nancy, qui a accepté également de rapporter sur ce document et qui, par sa connaissance du domaine du génie logiciel et par ses remarques, a contribué à sa qualité.

Madame Marie-France Bruandet, Maître de Conférences à l'Université Pierre Mendès-France, qui a dirigé ce travail, et qui a accompagné mes premiers pas dans la recherche. Elle a contribué efficacement, par sa disponibilité, ses relectures et ses remarques pertinentes, à l'aboutissement de ce travail. Je la remercie vivement pour son appui moral dans les moments difficiles et pour les encouragements qu'elle m'a prodigués.

Monsieur Michel Adiba, Professeur à l'Université Joseph Fourier, pour l'intérêt qu'il a manifesté pour ce travail. Qu'il trouve ici toute ma gratitude pour avoir participé à ce jury.

Monsieur Yves Chiaramella, Professeur à l'Université Joseph Fourier, Directeur du Laboratoire de Génie Informatique, qui m'a fait l'honneur de m'accueillir dans sa communauté de recherche. Je lui suis reconnaissante de s'être intéressé à ce travail et d'avoir participé à ce jury.

Je ne saurais oublier Christine Collet et sa participation chaleureuse et précieuse à la lecture de ce document et à ses remarques intéressantes.

Je n'oublierai pas non plus ni l'aide de Catherine, ni le soutien de Jean-Pierre, ni l'amitié de Patrick, ni la bonne humeur et la gentillesse de l'ensemble des collègues du laboratoire...

RESUME

Ce travail a pour objectif d'apporter une aide au développement et à la maintenance de logiciels de grande taille dans les environnements de génie logiciel (EGL). Il s'agit de définir un noyau générique de stockage et de gestion permettant l'intégration de l'ensemble des informations génie logiciel et offrant des outils généraux et déclaratifs. Le gestionnaire d'un projet peut ainsi spécifier la stratégie ou la méthode de développement souhaitées.

Nous avons défini un modèle de données pour les systèmes hypertexte capable de gérer la spécificité des informations génie logiciel et d'offrir une interrogation bien adaptée à ce type d'environnement (navigation, "browser", etc). Le modèle est sémantique orienté-objet et intègre aussi bien les aspects statiques que les aspects dynamiques des informations génie logiciel. Nous avons exploité la richesse des modèles sémantiques pour la représentation des différentes abstractions nécessaires : objets complexes, sémantique des liens (composition, références simples), versions multiples et généralisation de la notion d'objet composite à celle d'objet générique composite. Via la notion de *document*, qui représente, en plus du contenu textuel des documents, leur contenu sémantique et leur présentation visuelle, le modèle comprend les éléments de base nécessaires pour l'intégration des fonctionnalités d'un système hypertexte, et, dans une étape ultérieure, de la recherche par le contenu sémantique des informations.

Concernant la dynamique, nous utilisons le couplage entre données et traitements existant dans les modèles orientés-objet et donnons aux *méthodes* une définition déclarative sous forme de règles similaires à celles des Evénement-Condition-Action. Les méthodes, dans ce formalisme, sont des déclencheurs. Elles n'expriment pas seulement le comportement des objets, mais aussi, les situations de leur déclenchement et les contrôles à effectuer sur le comportement des objets.

Mots-Clés

Environnement Génie Logiciel, Base de Données, Modèle Orienté-Objet, Objets Composites, Dynamique, Déclencheurs, Système Hypertexte.

TABLE DES MATIERES

INTRODUCTION

CHAPITRE 1 : ETAT DE L'ART DES ENVIRONNEMENTS GENIE LOGICIEL

CHAPITRE 2 : LES SYSTEMES DE GESTION DE BASES DE DONNEES POUR LES APPLICATIONS NON-CONVENTIONNELLES

CHAPITRE 3 : LA GESTION DES INFORMATIONS DANS LES ENVIRONNEMENTS GENIE LOGICIEL

CHAPITRE 4 : MODELE DE DONNEES, ASPECTS STATIQUES

CHAPITRE 5 : MODELES DE DONNES, ASPECTS DYNAMIQUES

CHAPITRE 6 : REALISATION, LE PROTOTYPE

CONCLUSIONS ET PERSPECTIVES

INTRODUCTION

1. INTRODUCTION

La programmation globale est le domaine de développement et de maintenance des logiciels de grande taille [Der 76]. Par opposition à la programmation locale, elle ne s'intéresse pas aux détails de réalisation (algorithmes, procédures, etc), mais à la structure générale des programmes composés de modules.

La programmation globale induit la manipulation de grandes quantités d'informations. Ces informations sont issues d'une séquence de phases qui représente le cycle de vie d'un logiciel [Boe 82, Boe 88] : la phase de définition des besoins, la phase des spécifications fonctionnelles, la phase de conception, la phase de codage, la phase de test et d'intégration, la phase de mise en opération et de maintenance.

Au cours de ces différentes phases sont produites des informations de nature hétérogène dont la plupart se trouvent dans les documents produits durant le cycle de vie. Un logiciel peut être vu à chaque étape au travers de l'ensemble de ces documents. On parlera dans la suite des *composants logiciel*. Ce sont par exemple : les documents de définition des besoins (cahiers des charges), les documents des spécifications fonctionnelles, les documents de conception, le code source et objet, les documents de test et les manuels d'utilisation [Blu 88, Huf 88, Pre 87, GEC 86]. Ces documents sont complexes de par leur structure et de par les opérations que l'on peut leur appliquer. Ils sont fortement reliés entre eux; les liens existant entre les différents composants montrent le flot d'informations et les relations sémantiques existant entre les différentes phases du cycle de vie d'un logiciel.

D'autres informations d'une granularité plus fine sont mises en jeu lors de la production et de la maintenance de logiciels. Celles-ci concernent, par exemple, les personnes, les ressources, les délais, les droits, etc.

Après la première livraison d'un logiciel au client débute la phase de la maintenance qui dure toute la vie du logiciel. La maintenance d'un logiciel consiste à le suivre en

effectuant la correction des anomalies découvertes durant son utilisation et en le complétant selon les besoins des utilisateurs ou suivant l'évolution dans le domaine.

Le problème essentiel de la maintenance d'un logiciel est l'insuffisance de la documentation associée au code source. Quand elle existe, celle-ci est constituée d'un ensemble de documents existant en plusieurs versions, généralement non structurés, d'accès et de maintenance difficiles et, par conséquence, non mis-à-jour par rapport au code source.

Dans de telles situations, un grand nombre d'outils génie logiciel performants sont utilisés pour supporter les activités de développement et de maintenance. Nous citons ici les éditeurs, les formateurs de texte, les compilateurs, les éditeurs de liens, etc, et des outils plus sophistiqués tels que les gestionnaires de versions et de configurations, les outils d'aide à la spécification et à la documentation, etc. Néanmoins, l'utilisation de ces outils ne résout pas tous les problèmes de maintenance.

Les recherches dans le domaine génie logiciel [Sca 88, Hab 86, Gal 86, Pen 85, Onu 85, Hor 84, Kra 82, Was 81] ont fait apparaître le besoin d'une base d'informations commune pour supporter l'intégration de l'ensemble des outils génie logiciel, et dont l'objectif est d'améliorer la qualité des composants logiciel et de les rendre facilement accessibles par les outils de maintenance. Ces recherches ont abouti au développement des Environnements Génie Logiciel (EGL).

Un EGL est un cadre de développement de logiciels, il regroupe l'ensemble des outils nécessaires et des informations produites tout au long de la vie des logiciels. Il assiste les utilisateurs en maintenant la structure des composants logiciel, en contrôlant leur évolution et en fournissant une base d'activités et de procédés représentant les différents aspects du développement (spécification, codification, etc). Ces environnements sont classiquement statiques; ils ne tiennent pas compte de l'intégrité et de la cohérence des données manipulées.

Dans la variété des EGLs proposés [SEE 89, God 92], dont certains sont décrits au chapitre 1, on remarque qu'aucun à l'heure actuelle ne possède un gestionnaire d'informations capable de couvrir tous les besoins pour pouvoir assurer un meilleur suivi des logiciels. D'une part, ils ne couvrent pas tout le cycle de vie des logiciels et sont dédiés à une activité particulière. D'autre part, ils n'intègrent pas tous les aspects nécessaires pour la gestion d'informations.

Notre objectif dans ce travail est d'étudier les différents besoins pour la gestion d'informations dans un environnement génie logiciel, et, de proposer un outil d'aide

dans le processus de maintenance de logiciels de grande taille. Pour cela, nous prenons en considération plusieurs points essentiels :

- le réflexe habituel lors de la recherche d'une fonction particulière ou pour comprendre une fonction, est de consulter la documentation associée. Cela suppose que l'environnement soit capable de gérer en plus du code source toute la documentation associée, ainsi que les liens existant entre les deux. Dans ce cadre, un accès direct et facile aux différents composants logiciel, en se servant des liens existant entre eux, permet une meilleure efficacité pour consulter la documentation associée au code source et inversement.
- la difficulté de la maintenance vient du fait que la documentation, lorsqu'elle existe, ne suit généralement pas l'évolution du code, et peut se trouver dans un état incohérent par rapport au code qu'elle documente, en particulier si celui-ci a subi des changements significatifs. Il faut être capable d'assurer l'évolution de la documentation de manière cohérente avec celle du code.
- la difficulté de la maintenance est liée également à l'existence de chaque composant logiciel en versions multiples dont il faut assurer le contrôle et la cohérence. Dans ce contexte la gestion des liens entre les différentes versions devient problématique, du fait qu'ils sont souvent non maîtrisés et non automatiquement modifiés.
- pour pouvoir localiser les parties de code à l'origine d'un dysfonctionnement ainsi que celles qui doivent être modifiées lors d'une évolution ou d'une adaptation à de nouveaux besoins, il faut fournir des outils pour permettre de comprendre le plus rapidement possible le logiciel. Une nécessité dans ce cadre est de pouvoir accéder au code source et à la documentation via une description sémantique de leur contenu. L'environnement doit donc être capable de gérer en plus du contenu textuel, voir graphique, des composants logiciel, leur contenu sémantique sous forme de termes d'indexation de structure plus ou moins complexe (mots-clés, graphes conceptuels, etc).

Le reste du chapitre est consacré à la description du contexte de notre travail. Dans le paragraphe 2, nous présentons les activités de développement et de maintenance en génie logiciel et leur problématique. Ensuite, nous donnons une description générale du projet ELEN supportant notre travail, suivie d'une présentation plus spécifique du travail effectué dans le cadre de cette thèse.

2. PROBLEMATIQUE DE DEVELOPPEMENT, DE MAINTENANCE ET DE REUTILISATION DE LOGICIELS

2.1. L'ACTIVITE DE LA MAINTENANCE

Par le mot logiciel, nous faisons référence aux composants logiciel nécessaires au développement d'un projet. La maintenance d'un logiciel est un travail complexe et coûteux, on estime qu'il représente plus de la moitié du coût du logiciel. L'étape de maintenance [Oma 90, Cha 88, Cal 88] est la dernière étape du cycle de vie du logiciel. Elle démarre dès la mise en œuvre du logiciel afin de corriger les erreurs, d'améliorer la performance et de l'adapter à de nouveaux besoins et/ou à de nouveaux environnements. Plus précisément, les changements dans un logiciel sont classés généralement suivant plusieurs axes:

- La correction des erreurs survenues dans le code ainsi qu'éventuellement dans sa documentation (maintenance corrective).
- L'ajout de nouveaux aspects et d'améliorations pour assurer un meilleur comportement (maintenance préventive) ou une meilleure performance (maintenance perfective).
- L'évolution du logiciel pour s'adapter à un nouvel environnement, tel que le changement de matériels (maintenance adaptative).
- L'évolution du logiciel pour satisfaire aux nouveaux besoins des clients (maintenance évolutive).

Il est important de remarquer aussi que l'étape de maintenance inclut la totalité des autres étapes du cycle de vie. En effet, en plus des activités propres à la maintenance des logiciels (tels que la localisation du problème, l'analyse des solutions possibles pour prendre une décision, etc), toutes les activités de planification, de conception, de codage et de test sont incluses. Les personnes chargées de la maintenance sont obligées de produire des documents, d'éditer des programmes, de compiler, de tester et d'évaluer le code modifié. Elles ont besoin de comprendre le code existant en le consultant ainsi que la documentation associée. Bien que nous nous soyons fixé au départ comme objectif l'étude des besoins de la maintenance, nous constatons, en fait, que nous sommes également obligé de nous intéresser aux problèmes de conception et de développement.

Un autre aspect que l'on ne peut pas ignorer dans une activité de maintenance est la réutilisation de composants logiciel [Pri 87]. En effet, lors d'une modification d'un logiciel dans l'objectif d'une maintenance corrective ou adaptative, le programmeur a besoin de connaître le code réutilisable.

La tâche de maintenance est une tâche très difficile du fait de la difficulté de comprendre un logiciel, d'autant plus que le personnel de maintenance est la plupart du temps différent du personnel de développement. Le problème se pose d'une manière analogue lorsque l'on veut réutiliser des logiciels écrits par d'autres programmeurs. Nous pensons qu'une grande partie de ces difficultés est due à l'absence ou à la mauvaise utilisation de la documentation produite, et aussi, à la complexité du logiciel et à la difficulté d'identifier sa structure lorsqu'il a été soumis à plusieurs modifications successives.

Dans le paragraphe suivant nous énumérons les informations qui doivent être gérées dans un environnement génie logiciel.

2.2. NATURE DES INFORMATIONS MANIPULEES LORS DU DEVELOPPEMENT ET DE LA MAINTENANCE

Le travail de développement et de maintenance implique la gestion d'une quantité importante d'informations [Pen 89, God 92]. La plupart de ces informations sont représentées pour chaque logiciel sous forme de documents produits à l'issue de chaque phase de son cycle de vie [Huf 88, Wei 84]. Ces documents représentent le code source (programmes) et la documentation du logiciel en développement ou en maintenance :

- document de spécification des besoins : il décrit les besoins opérationnels que doit satisfaire le système en cours de développement. Il précise également les contraintes liées à l'environnement physique, à la performance, aux interfaces, aux facteurs humains, etc. Le document de spécification des besoins est écrit en langue naturelle en collaboration avec les clients.
- document des spécifications fonctionnelles : ce document détaille les fonctions à remplir par le système. Il est écrit ou bien en langue naturelle, ou bien dans un langage formel de spécifications (Hood, etc).
- document de conception non-détaillée : il donne l'architecture générale du système tout en décrivant les interfaces entre les différents modules.
- document de conception détaillée : il décrit précisément les algorithmes de réalisation pour chaque module du système, tout en respectant les spécifications des interfaces données dans le document de conception non-détaillée.
- code source : c'est le code qui implante les algorithmes spécifiés dans le document de conception détaillée.

- code exécutable : c'est le code dérivé lors de la compilation et de l'édition de lien du code source.
- document de tests : dans ce document sont spécifiés les plans des tests pour valider les spécifications opérationnelles du système.
- manuels d'utilisation : ces documents décrivent les commandes, leurs entrées et sorties, les messages d'erreurs et donnent des exemples d'utilisation du système.
- guides de maintenance : dans ces documents sont précisées les anomalies du système et les façons de l'améliorer du point de vue des fonctionnalités et de la performance.

Ces documents sont volumineux, de nature multimédia (texte, code source, graphique, etc), de tailles importantes, très structurés et de types différents (i.e., documents des spécifications, documents de conception, manuels d'utilisation, etc). De plus, ils sont "composites", dans le sens où ils sont composés d'autres objets. Par exemple, un document est composé de plusieurs chapitres, eux mêmes composés de sections, etc. L'interdépendance entre les documents est grande : il existe des relations entre les différentes parties de la documentation et entre la documentation et le code source. Les documents existent souvent en plusieurs versions, et par conséquence le maintien de la cohérence est une opération complexe. Les outils (tels que, compilateurs, processus d'indexation) agissent sur les documents pour produire dans certains cas (i.e., code source, texte) d'autres informations dérivées (i.e., code objet, termes d'indexation). Nous reviendrons sur ces différents aspects dans le chapitre 3.

Les informations manipulées lors de la production et de la maintenance de logiciels incluent aussi des informations de taille et de granularité différentes de celles des documents. Il s'agit des informations concernant les développeurs, les ressources, les budgets, etc.

Comme nous le verrons par la suite, les EGLs existants n'offrent que des solutions partielles aux problèmes de gestion de ces informations : certains ne gèrent que les programmes [Bel 88], d'autres ne gèrent que la documentation. Ceux qui gèrent à la fois les programmes et la documentation [Big 87, Gar 90], sont soit limités à un modèle de logiciels spécifique ou à un langage de programmation précis, soit pas assez généraux pour proposer une solution complète aux divers besoins pour la représentation et pour la gestion simultanée d'informations de complexité et de granularité très variables.

2.3. ASPECTS DYNAMIQUES DU DEVELOPPEMENT ET DE LA MAINTENANCE

Le mot "dynamique" est très général et a été utilisé dans la littérature pour désigner des notions différentes. Dans cette étude, la notion de dynamicité est associée à l'évolution, à l'intégrité et à la cohérence de la base d'informations. Elle est liée à la sémantique attribuée au comportement des informations.

Le développement et la maintenance des logiciels mettent en jeu une base importante d'informations complexes et partageables, dont l'interdépendance est très forte. Cette base est caractérisée par ses aspects dynamiques et évolutifs qui demande une gestion efficace de la sémantique liée aux informations et aux situations critiques.

Les aspects dynamiques du développement et de la maintenance (que nous allons étudier avec plus de précision dans le chapitre 3) sont dus aux caractéristiques suivantes des informations manipulées :

- Les composants logiciel sont forcément partagés entre plusieurs usagers, il est donc essentiel de les protéger. Ainsi, lors de l'accès à un composant il faut vérifier que l'usager a le *droit* d'effectuer l'opération en cours sur le composant en question.
- Certaines modifications sont effectuées localement aux composants logiciel, mais dans la plupart des cas les modifications effectuées sur un composant logiciel nécessitent la création de nouvelles *versions*. Le contrôle de la création et de la manipulation de versions multiples des composants logiciel est un des problèmes clés de la maintenance.
- Dans le cas de logiciels composés de plusieurs centaines de modules, chacun pouvant exister en plusieurs versions, le choix d'une liste cohérente de versions permettant de construire une version complète du logiciel devient problématique. La gestion de la création de ces listes, appelées *configurations*, est spécifique aux environnements de développement et de maintenance.
- La manipulation d'une base d'informations de volume important pose le problème de la gestion de l'intégrité de la base. Il est essentiel de gérer les *contraintes d'intégrité* complexes qui expriment les états valides des informations modifiées lors de la maintenance.
- Les modifications effectuées sur un composant logiciel peuvent rendre invalide un composant ou plusieurs autres composants qui en dépendent. Les *effets* des modifications doivent être analysés pour informer l'utilisateur des

composants atteints (portion de texte, procédures, etc) ou, dans certains cas, les propager automatiquement pour assurer la cohérence de la base.

Pour résoudre ces problèmes, certains EGLs [Bel 90, Dit 85, Kai 88, etc] étendent leurs fonctionnalités en supportant plusieurs mécanismes pour la gestion de l'intégrité et de la cohérence.

3. LE SYSTEME ELEN

3.1. PRESENTATION GENERALE DU SYSTEME

ELEN (GeniE LogicieL et RecherchE d'InformatioN) est un système conçu et réalisé dans le cadre du projet ARISTOTE [Ari 91] dont les partenaires sont le Laboratoire de Génie Informatique (LGI), et le centre de recherche Bull. Le projet Aristote concerne l'étude et le développement d'un Système de Gestion d'Objets Complexes Multimédia, dont les travaux de recherche s'inscrivent dans le cadre de l'approche *objet*. Les deux thèmes majeurs sur lesquels sont centrés les activités d'Aristote sont :

- Environnement de conception et de développement d'applications persistantes.
- Etude d'applications complexes : génie logiciel et application médicale.

Le système ELEN dans ce contexte concerne l'étude de l'application génie logiciel, où nous nous intéressons à la gestion des informations produites pendant le cycle de vie d'un projet. Il s'agit essentiellement de gérer d'une manière cohérente la documentation associée à un logiciel (documents des spécifications, manuels d'utilisation, etc). Le but final est d'apporter une aide aux personnes chargées de la maintenance des logiciels selon deux points de vue.

Le premier vise à assurer la cohérence de la base en ce qui concerne, essentiellement, les liens existant entre les logiciels et leur documentation, de manière à ce que la documentation suive l'évolution des programmes qu'elle décrit. Il est évident que l'utilisation d'une approche base de données où ces problèmes ont déjà été étudiés peut s'avérer bénéfique [Pen 89, Per 88]. Afin d'assurer l'intégrité et la cohérence des informations, les Systèmes de Gestion de Bases de Données (SGBD) ont développé plusieurs techniques, parmi elles : la définition de contraintes d'intégrité qui assurent l'intégrité des données en caractérisant les états valides; la définition de vues, notion correspondant aux droits de différents usagers; et la définition de transactions permettant de restaurer la base dans un état de cohérence. Contrairement aux SGBDs classiques qui gèrent des informations atomiques de longueur fixe pour lesquelles peu de types sont définis, mais ayant de très

nombreuses instances, les EGLs manipulent des informations souvent structurées et complexes, pour lesquelles on a de très nombreux types n'ayant-que peu d'instances.

La prise en considération des nouveaux besoins nécessite l'adaptation des modèles de données existants, ou la définition de nouveaux modèles de données qui prennent en compte la spécificité de l'application génie logiciel. Cette dernière approche vise à intégrer, au niveau du modèle de données, la gestion des objets volumineux, complexes et structurés d'une part, et la gestion de la cohérence et de la dynamique d'autre part [Jar 90a].

Le second point de vue concerne l'aide aux personnes chargées de la maintenance pour retrouver les informations recherchées dans le but de les modifier ou de les réutiliser. En effet, l'accès aux informations au travers de requêtes basées sur la structure des informations et sur leur description externe, comme c'est le cas dans les SGBDs, exige une connaissance parfaite de la structure des informations. Vu le grand volume et la complexité des informations manipulées dans un environnement de maintenance, ce type de recherche n'est pas suffisant et même n'est pas toujours possible. D'autres types de recherches sont souhaitables.

Une recherche par le contenu des documents est nécessaire dans ce cas. La recherche par le contenu tient compte du contenu sémantique des documents ainsi que de leur description externe. Dans ELEN, ce travail fait l'objet d'une autre thèse [Che 91]. L'interrogation porte sur la fonctionnalité que remplit un composant logiciel. En effet, dans le cadre de la réutilisation, cette interrogation permet de rechercher la partie du code réutilisable satisfaisant le mieux une fonctionnalité à implanter; dans le cadre de la maintenance elle permet de localiser une partie du code défaillant à partir de la description de son comportement. Cette recherche est fondée sur l'association d'un descripteur, exprimé dans un formalisme issu du modèle des graphes conceptuels, à chaque partie du code source. La requête est exprimée dans le même formalisme et le processus de recherche s'appuie sur des mécanismes de correspondance de graphes. La recherche par le contenu sémantique des documents représente une fonctionnalité essentielle des Systèmes de Recherche d'Informations (SRI). Cette étude met en évidence la complémentarité des deux domaines : les SGBDs et les SRIs. C'est pour cela que dans ELEN nous proposons une extension d'un SGBD offrant de réelles possibilités d'intégrer des fonctionnalités des SRIs.

Un troisième type de recherche nous semble très important dans une base contenant un grand volume d'informations complexes et corrélées, c'est la recherche par navigation sur l'ensemble des informations. Les utilisateurs ne sont pas forcément capables de formuler des questions portant sur la structure ou sur le contenu des

informations. Ils ont souvent besoin de regarder la structure des documents ou de retrouver simplement les parties de documentation liées à une partie de code source qu'ils sont en train de modifier. Ce type de recherche simple et efficace est la fonctionnalité principale des Systèmes Hypertexte (SHs) [Jar 90b].

Les systèmes classiques dans chacun de ces trois domaines, la recherche d'informations, les bases de données et les systèmes hypertexte, ne prennent pas en compte leur complémentarité. Notre principale apport dans ces domaines est d'avoir pris en compte cette complémentarité (dans [Jar 89] on trouve une première approche de solution à ce problème).

3.2. LE TRAVAIL REALISE DANS LE CADRE DE LA THESE

Dans cette thèse nous nous sommes intéressé à la définition et à l'implantation d'un noyau générique de stockage et de gestion d'informations pour les EGLs qui intègre les programmes et leur documentation tout en considérant une approche hypertexte. Le terme générique signifie ici que notre proposition ne tient pas compte d'une méthode particulière ou d'une stratégie de développement spécifique comme c'est le cas dans la plupart des EGLs : le noyau est ouvert et capable de gérer les informations génie logiciel quelle que soit la stratégie utilisée. Pour cela, il offre des outils généraux et déclaratifs. Le gestionnaire d'un projet est capable ainsi de spécifier le modèle de logiciels qu'il souhaite utiliser.

Dans ce travail , nous portons notre attention à la définition d'un modèle de données pour un système hypertexte qui prend en considération l'intégration des aspects suivants :

- Le modèle proposé s'intéresse à la spécificité des objets manipulés dans les environnements génie logiciel. Dans ce modèle sont représentées, en plus des documents et des logiciels en tant qu'objets composites, leurs relations sémantiques, ainsi que leurs versions multiples. D'autre part, on s'est intéressé à l'étude des éléments de base à introduire au niveau du modèle de données pour enrichir un SGBD avec une fonctionnalité de recherche par le contenu de type SRIs, et une fonctionnalité d'accès direct de type SHs. Cette partie du modèle sera désignée dans la suite de ce document par le terme *aspects statiques.*

- Une attention particulière a été portée à la représentation du comportement des composants logiciel et à l'assurance de l'intégrité de la base et de la cohérence entre ces composants, et en particulier entre les programmes et leur documentation. Notre objectif dans ce cadre est de pouvoir exprimer ces aspects, désignés par le terme *aspects dynamiques*, d'une façon uniforme

et intégrée au niveau du modèle de données. Pour cela nous avons utilisé des déclencheurs prenant la forme de règles événement-condition-action. Ces déclencheurs expriment d'une manière déclarative, d'une part le comportement des objets, et d'autre part, les situations de déclenchement et les contrôles à effectuer sur le comportement des objets.

Notre travail se situe dans le cadre des modèles sémantiques et orientés-objet. La richesse des modèles sémantiques pour supporter la sémantique des données nous a permis de représenter les mécanismes d'abstractions nécessaires (objets composites, versions multiples, etc). Nous bénéficions du couplage entre les données et les traitements de l'approche objet via les méthodes. Classiquement, les méthodes sont utilisées pour exprimer le comportement des objets. Dans notre proposition, elles sont utilisées pour exprimer le comportement des objets, mais aussi et d'une façon déclarative les contrôles nécessaires sur ce comportement.

Le modèle proposé servira de base pour la construction d'un gestionnaire d'objets pour le système ELEN. Le prototype, réalisé en Loops sur une machine Xerox, représente un système hypertexte qui intègre les programmes et leur documentation et implante les différentes abstractions et fonctionnalités mentionnées ci-dessus.

4. PLAN DU DOCUMENT

L'objectif de ce document est de présenter le travail réalisé pour définir et implanter un modèle de données tenant compte de la spécificité des informations des EGLs. La première partie concerne la présentation du contexte du travail et des principaux travaux effectués dans ce domaine. La deuxième partie porte sur notre proposition et sur le prototype réalisé.

Ce document est organisé de la façon suivante :

- Le chapitre 1, étudie les principales propositions faites dans le domaine du génie logiciel pour la gestion d'objets dans les EGLs.
- Le chapitre 2 présente les recherches faites dans le domaine des bases de données pour remédier à l'insuffisance des propositions faites en génie logiciel et pour satisfaire aux besoins des applications non-conventionnelles. Nous faisons le point sur des propositions comme l'extension des modèles relationnels, ou comme l'utilisation de l'approche objet.
- Le chapitre 3 synthétise la problématique de la gestion d'informations pour le domaine du génie logiciel. Nous avons classifié les besoins à satisfaire selon trois aspects: les aspects statiques, les aspects dynamiques et l'accès aux informations. Une présentation plus détaillée de chacun de ces trois aspects

est effectuée en soulignant les avantages et les inconvénients des solutions proposées par les systèmes existants, soit dans le domaine du génie logiciel, soit dans le domaine des bases de données.

- Dans le chapitre 4 nous abordons la définition du modèle de données retenu pour un noyau générique de gestion d'objets dans les EGLs. Ce chapitre étudie en particulier les aspects statiques du modèle, c'est-à-dire tout ce qui est lié à la représentation d'objets composites, à la description des types d'objets, à la modélisation des relations entre les objets et du contenu sémantique des documents, et à la représentation des versions.
- Ensuite, dans le chapitre 5, nous complétons la définition du modèle proposé dans le chapitre précédent en intégrant les aspects dynamiques. Un mécanisme d'événement-action qui supporte le comportement des objets et leur réaction à toute modification dans la base est proposé.
- Le chapitre 6 est une description du prototype réalisé. Ce prototype consiste en un système hypertexte dont la base supporte les différents aspects du modèle discuté dans les deux chapitres précédents. Les exemples présentés dans ce chapitre sont tirés de l'application définie dans l'annexe 1.
- Enfin, dans notre conclusion, nous replaçons notre proposition par rapport aux approches plus classiques proposées dans les deux domaines de recherche : le génie logiciel et les bases de données. Nous mettons en évidence les apports de notre approche qui vise à intégrer les différents besoins au niveau du modèle de données.

CHAPITRE 1

ETAT DE L'ART DES ENVIRONNEMENTS GENIE LOGICIEL

1. INTRODUCTION

Dans le milieu des années soixante les systèmes logiciels ont commencé à atteindre une taille et un niveau de complexité qui ont rendu leur développement et leur maintenance problématique. On parle dans ce cas de la programmation globale "programming in the large" où les logiciels sont développés par un grand nombre de programmeurs. Dans ce cadre, le code source des logiciels ainsi que la documentation sont constitués d'une collection de composants, chaque composant existant en plusieurs versions.

Les outils isolés, apparus pendant les années soixante-dix, tels que SCCS, RCS et MAKE, traitent le problème de stockage de versions de code source et de construction de programmes de taille importante. Ils sont limités à des environnements spécifiques, dépendant du système d'exploitation et ne pouvant pas s'appliquer à la production de logiciels de taille importante et de structure complexe.

Le concept d'Environnement Génie Logiciel (EGL) est apparu au milieu des années soixante-dix avec l'objectif de combiner les techniques, les outils et les méthodes, pour supporter la programmation globale. Un EGL est ainsi un cadre de développement et de production de logiciels dans lequel le processus de développement est de plus en plus automatisé. Il doit proposer un ensemble d'outils, de langages et de méthodes, afin de faciliter la gestion et la production des logiciels.

Dans les premiers environnements assistant les programmeurs dans les phases de conception et de codage, les outils étaient peu supportés. Ces environnements sont dépendants du système d'exploitation utilisé. Les environnements plus récents ont une vision globale du cycle de vie des logiciels. Ils ont validé une large variété de concepts importants pour la programmation globale (programmation structurée, intégration d'outils, etc).

Un des points clés des EGLs supportant la programmation globale est la gestion de la base d'informations dont ils disposent. Cette base contient tous les composants logiciels produits pendant le cycle de vie (programmes, documentation), leurs relations et leur processus de production. Elle constitue aussi le moyen de communication entre les outils. Dans cet objectif, les recherches menées par les spécialistes en génie logiciel consistent au développement d'un EGL dont le noyau est un gestionnaire d'informations supportant la spécificité des informations génie logiciel.

Ce chapitre présente une étude détaillée des principales propositions faites dans le domaine du génie logiciel. Dans la section suivante nous présentons tout d'abord l'architecture générale des EGLs classiques. Pour pouvoir comparer les différents EGLs existants, nous avons essayé, dans la section 3, de les classifier selon les approches choisies. Ensuite, nous exposons selon cette classification quelques uns de ces systèmes.

2. ARCHITECTURE GENERALE DES EGLS

Il n'existe pas d'architecture unifiée suivant laquelle les EGLs sont conçus. Nous pouvons trouver autant d'architectures qu'il y a d'environnements. Néanmoins, certaines fonctionnalités sont communes à tous les environnements.

Du point de vue conceptuel, nous pouvons identifier plusieurs couches essentielles (voir Figure 1) :

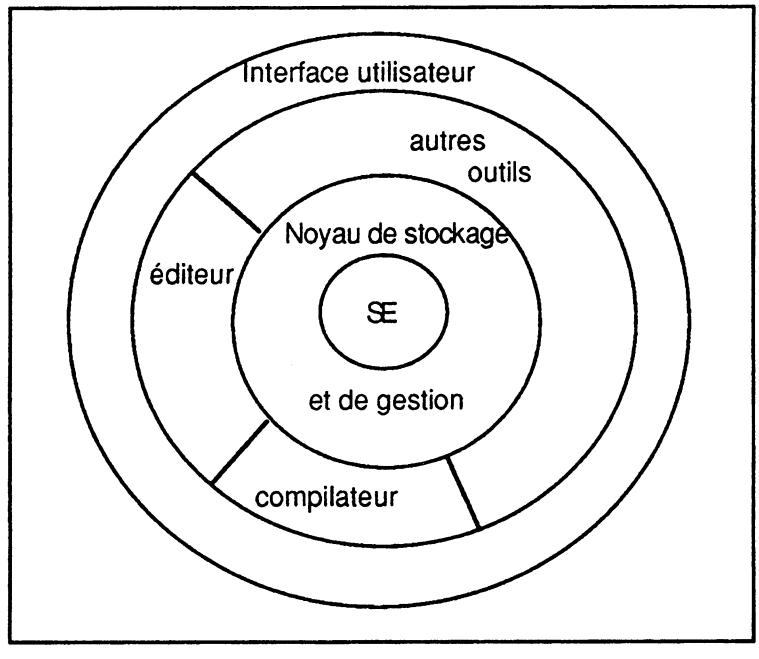


Figure-1

- le niveau logiciel et matériel de base (système d'exploitation et configuration de machine).
- le gestionnaire d'informations responsable du stockage, de la gestion d'informations persistantes et du contrôle d'activités.
- l'interface utilisateur.

Selon l'environnement, les limites entre les différentes couches peuvent varier. Par exemple, le gestionnaire d'informations peut disposer d'une interface permettant l'interaction avec l'utilisateur. Cependant, ce gestionnaire, en assurant les fonctionnalités de base d'un EGL, détermine l'efficacité et la performance de celui-ci. Les fonctionnalités principales que doit supporter ce gestionnaire se résument par :

- le stockage et la gestion des informations persistantes : l'ensemble des composants logiciels sont stockés dans une base d'informations et le gestionnaire doit proposer un ensemble d'opérations de manipulation.
- la gestion de versions et de configurations : le développement et la maintenance nécessitent la gestion de versions multiples des composants logiciels. Le contrôle de la création et de la manipulation des versions multiples est une des fonctionnalités principales du gestionnaire d'informations. Celui-ci doit intégrer également un gestionnaire de configuration pour aider à reconstruire un logiciel à partir de ses composants en choisissant les versions appropriées.
- la gestion de projet en ce qui concerne les droits d'accès et les accès concurrents aux objets : les composants logiciel sont généralement partagés entre plusieurs utilisateurs. Le gestionnaire doit proposer des mécanismes nécessaires pour les protéger et assurer leur cohérence.
- l'intégration d'outils génie logiciel: l'activation des outils génie logiciel ne se fait pas directement par l'appel de ces outils. Le gestionnaire d'informations fournit des abstractions d'outils ou de combinaisons d'outils. Par exemple, pour compiler un module, le gestionnaire est chargé d'appeler le compilateur correspondant au langage du module et de recompiler également les autres modules nécessaires à sa compilation.
- la gestion d'activités et l'adaptation aux environnements spécifiques: le gestionnaire propose une représentation du processus utilisé pour le développement et la maintenance des logiciels sous forme de tâches à

réaliser. Il permet aussi de définir des stratégies particulières pour l'adaptation aux environnements spécifiques. Certains gestionnaires vont plus loin en proposant une automatisation de certaines parties du processus de développement, ou en analysant les effets de changements pour déclencher certaines actions et maintenir ainsi la cohérence de la base.

L'évaluation d'un EGL se fait selon plusieurs critères représentant les caractéristiques désirées. Les plus importants sont les suivants :

- utilité : le rang des fonctionnalités offertes par le gestionnaire d'informations et le support qu'offre celui-ci aux différentes phases du développement d'un logiciel.
- adaptabilité et extensibilité : la possibilité d'adapter l'environnement à des stratégies de conception et de développement différentes et d'étendre ses fonctionnalités pour couvrir de nouvelles techniques de développement et de maintenance.
- niveau d'automatisation : le degré d'assistance automatisée offerte par le gestionnaire pour l'exécution des activités de développement de logiciels.
- niveau d'intégrité : l'intégrité veut dire ici l'existence d'un formalisme uniforme pour la représentation et la gestion de composants logiciels, le partage de données, l'activation et la combinaison d'outils, le contrôle de la consistance et de la cohérence de données.
- facilité d'utilisation : l'adaptabilité de l'interface et la facilité d'utilisation et d'apprentissage offerte aux utilisateurs.

3. ENVIRONNEMENTS GENIE LOGICIEL EXISTANTS

Nous avons vu que le composant central d'un EGL est son gestionnaire d'informations. On peut classifier les EGLs selon l'approche choisie pour implanter ces gestionnaires; on distingue les quatre types suivants:

- Les environnements classiques : ils gèrent des systèmes de fichiers traditionnels tout en supportant une collection incomplète d'outils. Bien que ces systèmes, dont les exemples sont DSEE [Leb 84], CEDAR [Tei 84], GANDALF [Hab 86], aient validé certains concepts de la programmation globale (utilisation de la programmation structurée, contrôle de versions), ils restent limités et élémentaires au niveau du contrôle de leur base d'informations.

- Les environnements basés sur un SGBD conventionnel : ils sont souvent conçus par les spécialistes en génie logiciel. Pour la gestion de leur base d'information, ils utilisent des systèmes de gestion de bases de données conventionnels existant sur le marché et souvent relationnels. Les exemples de ces environnements sont nombreux : SODOS [Hor 86], PMDB [Pen 85], ALMA [Lam 88], etc.

- Les environnements basés sur un SGBD spécialisé : c'est l'approche suivie et élaborée par les chercheurs de génie logiciel en collaboration souvent avec des chercheurs en bases de données. Pour satisfaire les besoins en base de données pour les EGLs, des modèles dédiés ont été proposés et implantés. C'est le cas de CAIS [Obe 88], PCTE+ [Tho 89, Oqu 89], NOMADE [Bel 92, Bel 88], CACTIS [Hud 88], EPOS [Con 90a, Con 90b], DAMOKLES [Dit 87], MARVEL [Kai 88], etc.

- Les environnements à base d'hypertexte : l'hypertexte est une approche pour la gestion d'informations dans laquelle les données sont stockées dans les nœuds d'un réseau connectés par des liens. Les systèmes hypertexte (SHs) ont trouvé largement leur application dans le domaine du génie logiciel. Plusieurs SHs pour la gestion d'informations dans les EGLs ont vu le jour dont les plus connus sont Neptune [Del 87, Del 87] et DIF [Gar 90, Gar 87].

3.1. LES ENVIRONNEMENTS CLASSIQUES (A BASE DE FICHIERS)

C'est la première génération des EGLs supportant la programmation globale et basés sur la technologie des systèmes d'exploitation. Le but de ces systèmes est de supporter les différentes phases du cycle de vie des logiciels. Les objets ici sont les modules qui sont décrits par des informations (attributs) dérivées du système d'exploitation. Ces environnements utilisent un système de fichiers classique pour le stockage et la gestion des informations. Parmi ces environnements nous distinguons les environnements supportant la programmation structurée dont l'exemple est Gandalf, et ceux basés sur un langage et représentant ainsi l'extension de ce langage pour l'intégration des outils génie logiciel. Les exemples de ces derniers incluent Interlisp-D pour Lisp [Moo 76], Cedar pour Mesa [Tei 84], etc.

3.1.1. GANDALF

Gandalf [Hab 86] est un générateur d'environnements de développement et de programmation, conçu pour le langage GC (extension du langage C). Il est composé essentiellement de deux parties : un gestionnaire de versions et de configurations, et

un gestionnaire de projet supportant la gestion de droits d'accès et l'accès concurrent aux données.

Le gestionnaire de versions permet de décrire et de manipuler les composants logiciels, leurs versions et les relations entre eux. Les logiciels sont structurés en modules, et chaque module comporte plusieurs *versions* réalisant une même interface en utilisant des méthodes différentes. Chaque version consiste en plusieurs *révisions* montrant ses implantations successives. Par défaut, les versions des modules sélectionnées pour la construction d'un logiciel sont celles dites "standards".

Le gestionnaire de versions de Gandalf fait la distinction entre deux notions : les *implantations* et les *compositions*. L'implantation d'un module est le code source définissant les réalisations des ressources spécifiées par son interface. Une *composition* est la liste des modules ou des versions de modules qui composent un logiciel, c'est la notion correspondant aux configurations.

Les informations manipulées par le gestionnaire de versions sont organisées sous forme d'une forêt. Le sommet de chaque arbre est une *boîte* contenant un ensemble de modules et pouvant inclure d'autres *boîtes*. A chaque *boîte* est associée une "liste d'accès" désignant les personnes autorisées à manipuler les modules de la *boite* et réparties en trois classes : administrateurs, programmeurs et autres usagers.

Le gestionnaire de projet a pour rôle de maintenir l'intégrité des informations, de gérer les droits d'accès décrits par les listes d'accès associées aux *boîtes*, et de protéger les informations lors d'accès concurrents par la disposition de plusieurs commandes telles que : *reserve*, *release* et *deposit*.

Un mécanisme d'attachement procédural a été développé par Gandalf : il consiste en l'exécution de certaines "actions routines" attachées aux *boîtes* et aux modules, dès qu'un composant est modifié. Le but est de contrôler les versions, de propager les changements et de recréer les configurations atteintes.

3.1.2. DSEE

DSEE (DOMAIN Software Engineering Environment) [Leb 84] est un environnement de développement réparti développé sur les ordinateurs Apollo. Il est composé de cinq outils : un gestionnaire de versions, un gestionnaire de configurations, un gestionnaire de tâches, un gestionnaire d'instructions et un gestionnaire de dépendance.

DSEE utilise un système de gestion de bases de données réparti pour stocker les informations structurées hiérarchiquement; un système classique de fichiers pour stocker les deltas (différences en contenu entre deux versions successives) et les tâches; et un serveur de processus.

Le gestionnaire de versions contrôle le code source dont les unités de base sont appelées *éléments* et maintient les versions d'éléments et leur historique. Le système considère la version la plus récente comme étant la version par défaut.

Le gestionnaire de configurations décrit les composants d'un logiciel et leur dépendance en utilisant la notion de *modèle de système*. Il spécifie aussi les contraintes de composition à appliquer pour la construction d'une configuration et détermine ainsi la version de chaque composant à utiliser dans une configuration.

Le gestionnaire de tâches permet de décrire la liste de sous-tâches nécessaires à la réalisation d'une activité de développement ou de maintenance appelée *tâche*. Une tâche est une structure utilisée pour planifier et enregistrer les étapes à exécuter pour la réalisation d'une activité d'un niveau supérieur. Les tâches sont enregistrées et affichées par le gestionnaire d'événements sur le poste de l'usager qui est responsable de leur exécution. Une fois réalisées, les tâches sont archivées.

Le gestionnaire d'instructions est une aide pour la gestion de tâches similaires et communes à plusieurs usagers. Il supporte des modèles de tâches à partir desquels les nouvelles tâches sont créées.

Le gestionnaire de dépendance permet aux utilisateurs de définir des relations de dépendance entre les différents éléments. Pour informer d'une éventuelle incohérence, le gestionnaire de dépendance permet de déclencher des alarmes avant et après la modification d'un élément lié à d'autres éléments par une relation de dépendance. Les services qui sont donc offerts sont : indiquer le danger d'une modification, informer d'une modification à introduire dans un autre objet, indiquer les tâches à réaliser après une modification ou lancer une séquence de commandes comme par exemple la reconstruction d'une configuration. Les relations de dépendance maintenues par le gestionnaire d'événements dans DSEE sont les suivantes : programme-document, module commun à deux systèmes et les relations de dépendance entre deux modules. Pour limiter la propagation des changements et éviter de lancer une cascade de recompilation dès qu'un module est modifié, une dépendance entre deux unités de compilation peut être déclarée non critique, ainsi le changement d'une unité ne cause pas la recompilation de celle qui en dépend.

3.1.3. Discussion

Les EGLs classiques sont simples, performants, efficaces et gèrent un volume important d'informations.

Bien que ces systèmes aient validé certains concepts de la programmation globale (utilisation de la programmation structurée, contrôle de versions et de configurations), ils restent limités et élémentaires au niveau description et contrôle de leur base d'informations. De plus, le modèle considéré est figé : ils sont généralement dédiés à un langage de programmation donné, à un système d'exploitation et à une configuration de machine spécifique. Les seuls objets pris en compte sont les modules (i.e. la documentation n'est souvent pas prise en considération), la description des objets est limitée aux informations dérivées du système d'exploitation, les contraintes sont prédéfinies, etc.

Ces environnements ne disposent pas d'opérations pour la représentation et la gestion de la sémantique des objets composites et pour la manipulation de la structure logique des programmes et de la documentation. Ils utilisent le système de fichiers classique dont dispose le système d'exploitation pour la gestion des objets volumineux tel que le code source, ce qui rend l'accès à ces informations et leur interrogation très difficile .

Les aspects dynamiques sont souvent non abordés par ces environnements, ou bien ils sont gérés par des mécanismes élémentaires et rigides. Dans le cas de mécanismes performants, ils sont dépendants du système d'exploitation; les gestionnaires de versions et de configurations sont basés souvent sur des outils Unix (SCCS, RCS). Dans le cas de DSEE qui dispose d'un gestionnaire d'événements, le mécanisme est limité à certains types d'événements (i.e. modification d'un module) et à certaines relations prédéfinies (relation de dépendance entre modules).

3.2. LES ENVIRONNEMENTS BASES SUR UN SGBD CONVENTIONNEL

Pour résoudre certains problèmes posés par les environnements classiques, tels que : le partage de données entre les utilisateurs et la concurrence d'accès, l'intégrité de données, la facilité d'accès et d'interrogation, etc, la deuxième génération des EGLs a utilisé la technologie des bases de données. Ces environnements sont conçus souvent par des spécialistes en génie logiciel dans le but d'intégrer les composants logiciels (programmes, documentation) dans un noyau supportant leur structuration et organisation basé sur une approche conventionnelle (relationnelle). Certains utilisent un SGBD existant sur le marché pour gérer leur noyau d'informations (Ingres pour PMDB), d'autres, tel que SODOS, définissent un système spécifique de

gestion de données. L'accès aux informations se fait dans ces cas, via un langage de manipulation proposé par le SGBD utilisé.

3.2.1. PMDB

Le PMDB (Project Master Data Base) [Pen 86, Pen 85] est un noyau de gestion d'informations pour les EGLs automatisés. Son objectif est la modélisation de tous les composants d'un logiciel et les relations existant entre eux indépendamment de toute stratégie de développement.

Le contenu de la base de données est décrit en utilisant le modèle de données Entité-Association-Attribut, qui est un modèle Entité-association étendu. Les notions de base du modèle sont : les objets, les attributs et les associations. Les *objets* représentent les composants d'un schéma PMDB, ils décrivent les types de données essentielles dans un projet (sources, produits, documentation, etc). Les objets peuvent avoir des instances. Un ensemble d'attributs est associé à chaque objet pour représenter ses propriétés. Les instances contiennent donc les valeurs de ces attributs. Les relations entre les objets sont modélisés par des *associations* binaires entre eux. Le modèle de PMDB ne supporte pas le concept d'objets composites ni celui de versions d'objets.

Le schéma résultant en modélisant l'environnement dans PMDB est constitué de 31 objets, 220 attributs et 170 associations. Le choix des objets et des relations est basé sur leur importance de façon à supporter dans ce noyau les informations communes à la plupart des projets logiciels. Toutefois, il est possible d'étendre ce schéma par l'ajout d'autres objets.

Le prototype qui implante les concepts de PMDB utilise Ingres et le système de fichiers d'Unix comme supports de stockage pour les informations manipulées. Ceci consiste à définir un schéma relationnel d'Ingres en traduisant les objets, les attributs, et les associations par des relations et des domaines Ingres tout en stockant les données textuelles dans les fichiers du système Unix.

En ce qui concerne la sémantique des données, certains types de contraintes référentielles concernant la dépendance entre les objets sont maintenus. Toutefois, aucun mécanisme n'est offert pour exprimer et gérer des contraintes définies par l'utilisateur.

3.2.2. SODOS

SODOS [Hor 86, Hor 84] est un EGL conçu au dessus d'un SGBD relationnel, son intérêt est d'utiliser les techniques des bases de données pour supporter la structure logique des informations complexes.

La base de données de SODOS modélise le cycle de vie des logiciels en utilisant une approche "objet". Les documents issus du cycle de vie sont représentés sous forme d'objets structurés hiérarchiquement dont les propriétés sont modélisées par un ensemble d'attributs. Les objets ayant des propriétés similaires sont regroupés dans des ensembles appelés classes. Chaque classe définit une structure suivant laquelle les objets sont créés, et un ensemble d'associations modélisant les relations les liant à d'autres objets.

Pour définir les documents, SODOS considère la classe *document*. Ainsi chaque document du cycle de vie est une instance de cette classe ou d'une de ses sous-classes. Chaque sous-classe de *document* (*user manual*, *fonctional requirement document*, etc) définit la structure indiquant le type d'informations contenues dans les documents de cette classe.

Du point de vue représentation interne, SODOS utilise un SGBD relationnel conçu comme extension de l'environnement Smalltalk-80. Un ensemble de relations est utilisé pour la représentation des documents. Ainsi les relations : *document; structure; texte; figure* et *keyword*, représentent respectivement: les propriétés, la structure, le contenu textuel, le contenu graphique et les mots-clés de chaque document. Les associations entre les documents sont représentées par la relation *document interface relation.*

Pour définir le comportement des objets, chaque classe définit, en plus de la structure de ses objets, un ensemble d'opérations pouvant être exécutées sur ces objets (*modify*, *delete*, *copy*, *print*). Des opérations sont également définies pour la manipulation de la structure hiérarchique des objets et pour la définition de concepts de complétude, de consistance et d'équivalence d'objets structurés.

Néanmoins, nous pouvons qualifier SODOS d'un environnement statique puisque très peu d'attention est accordée à la gestion d'activités et à la cohérence d'informations.

3.2.3. Discussion

L'utilisation des techniques des SGBDs classiques pour la gestion d'informations dans les EGLs a abouti à résoudre certains problèmes posés par les environnements

classiques, tels que : le partage de données entre les utilisateurs et la concurrence d'accès, l'intégrité de données, la facilité d'accès et d'interrogation. Néanmoins, d'autres aspects, liés à la complexité, à la structuration, à l'existence en versions multiples et à la dynamique des informations, restent problématiques.

Nous venons de voir que, pour le stockage des données textuelles, PMDB utilise un système classique de fichiers. Outre les problèmes de performance que pose une telle approche, l'utilisation d'un système de fichiers pour le stockage rend impossible d'assurer la cohérence de la base de données avec le monde réel. En effet, il est impossible d'empêcher les utilisateurs d'agir sur le système de fichiers sans avertir la base de données. Certains systèmes, tel que SODOS, étendent leur capacité pour gérer le contenu textuel des documents, ainsi, l'utilisation du système de fichiers, même si elle a lieu, est cachée par rapport à l'utilisateur.

La gestion de la structure logique des objets est plus complexe et souvent écartée dans les environnements qui disposent d'un SGBD conventionnel. SODOS permet une représentation artificielle des objets structurés sous formes de n-uplets représentés par une ou plusieurs relations formant logiquement les composants d'un même objet. Cette approche manque de clarté et de simplicité d'expression.

Du point de vue dynamique, le contrôle de l'intégrité et de la cohérence dans les modèles relationnels est basé sur le mécanisme de vérification de contraintes d'intégrité, mécanisme ayant comme conséquence une dégradation notable en performance dans le cas d'un gros volume de données. C'est pourquoi la plupart des EGLs qui utilisent un SGBD relationnel ne disposent que d'un sous ensemble très réduit des capacités de ce mécanisme. C'est la raison pour laquelle ces environnements sont qualifiés de statiques plutôt que de dynamiques.

3.3. LES ENVIRONNEMENTS BASES SUR UN SGBD SPECIFIQUE

Cette approche a été suivie et élaborée par les chercheurs du génie logiciel en collaboration souvent avec des chercheurs en bases de données. Les approches utilisant des SGBDs conventionnels ont montré que seul un modèle dédié peut satisfaire les différents besoins en bases de données pour les EGLs. Des modèles spécifiques ont été proposés et implantés, certains sont figés et plus ou moins pragmatiques, tels que le noyau de gestion NOMADE. Mais la plupart de ces modèles sont génériques, dans le sens où il est possible d'utiliser leur terminologie pour définir des modèles de logiciels et des stratégies de développement spécifiques. C'est le cas des structures d'accueil PCTE+ et CAIS, et des systèmes ALF, EPOS, DAMOKLES, MARVEL, etc.

3.3.1. NOMADE

Nomade [Bel 92, Bel 88] est un gestionnaire d'objets permettant de gérer les logiciels, leurs versions et leurs configurations. L'élément central du gestionnaire Nomade est une base de données définie pour les besoins spécifiques de la programmation globale. Elle vise à supporter la construction modulaire des logiciels de grande taille. Nomade est composé principalement d'un gestionnaire d'objets et de versions, d'un gestionnaire de configurations et d'un gestionnaire d'activités. Un nombre d'outils périphériques (interface graphique, générateur de Make, etc) sont supportés.

Le gestionnaire d'objets représente une base spécialisée pour le génie logiciel basée sur une approche pragmatique s'inspirant du modèle entité-association. Toute entité de la base d'objets a un type descendant du type *objet* qui est un type prédéfini de Nomade. A chaque objet, on peut associer des attributs, des relations et des activités décrites dans la définition de son type. Deux types structurés sont supportés par le gestionnaire d'objets Nomade: *Famille* qui regroupe les versions d'un module structurées sous forme de listes de *révisions*, et *Usager* qui permet de modéliser l'organisation des équipes de développement. Chaque type structuré est composé de deux parties : une partie descriptive représentée par les entités *document*, *attributs*, *relations* et *droits d'accès*, et une partie structurelle correspondant à la décomposition des entités. La notion de *partition* dans Nomade permet de définir des sous-ensembles d'objets ayant des caractéristiques communes. Les partitions sont organisées en arbre dont *projet* est la partition racine, chaque partition hérite des définitions de sa partition ancêtre. A chaque objet est associé un *manuel* qui contient sa description sous forme de clauses (*attributes-clause, relations-clause, rights-clause, action-clause*). Ce manuel définit à chaque instant l'état de l'objet (en indiquant les valeurs associées aux attributs, aux relations, etc).

Pour gérer la dynamique des données, Nomade dispose de plusieurs mécanismes supportant les fonctions suivantes :

1) Gestion de droits d'accès : les droits d'accès sont exprimés pour chaque usager au moyen de couples (opération, domaine) où *domaine* définit les types d'objets sur lesquels l'opération peut porter. Les contrôles des droits d'accès sont mis en œuvre par la commande de vérification de droits, *checkright*, appelée explicitement lors de l'accès à un objet.

2) Contrôle de versions : Nomade dispose d'une fonction de contrôle de versions dont les objectifs sont : la création d'une nouvelle version à chaque modification d'un objet, l'optimisation du stockage du contenu des versions basé sur le

mécanisme de delta de RCS d'Unix, l'enregistrement automatique de l'historique des actions exécutées sur chaque version et le contrôle de l'accès aux versions par le mécanisme de réservation/libération.

3) Gestion de configurations : la gestion de configurations dans Nomade est axée sur la fonction de construction de configurations. Nomade offre trois modes de définition de configuration. Un mode *manuel* où l'usager fournit la liste des versions qui composent une configuration. Un mode *automatique* mais *implicite* basé sur la sélection, soit des dernières révisions pour chaque composant, soit des révisions prises par défaut. Et finalement, un mode *automatique* mais *explicite* en utilisant des règles de sélection basées sur la description des révisions.

4) Gestion d'activités (mécanisme événement-action) : pour décrire les tâches susceptibles d'être effectuées sur les objets (compilation, création d'une version, etc) et pour gérer les contraintes de cohérence complexe lors de l'exécution de ces commandes, Nomade utilise un gestionnaire d'activités. Ce gestionnaire est basé sur un mécanisme événement-action dont l'approche est inspirée des déclencheurs des bases de données (voir chapitre 2) et adaptée au contexte génie logiciel. La dynamique dans Nomade est donc contrôlée par l'association entre les objets et les règles événement-action, appelées *actions* dans la terminologie Nomade. Ce mécanisme permet de réaliser plusieurs fonctionnalités telles que : la vérification des conditions nécessaires à l'exécution d'une commande lancée par l'utilisateur, la détection de violation de contraintes d'intégrité après l'exécution d'une commande, l'exécution des commandes proprement dite (on appelle *commande* toute opération primitive prédéfinie dans Nomade), etc.

Le mécanisme est caractérisé par deux notions essentielles : *événement* et *action*, dont l'association constitue un *déclencheur*. Les déclencheurs sont associés aux objets et décrits dans leurs manuels de description.

Un ***événement*** est créé à chaque fois qu'une commande est exécutée sur un objet. L'événement est défini par cette commande, ses paramètres et ses options.

Les ***actions*** sont, soit primitives, appelées *commandes*, soit définies par l'usager. Les actions usager sont des programmes de haut niveau contenant des commandes base de données ou des appels à des outils externes. Elles peuvent être paramétrées par les informations définissant le contexte de l'événement. Les paramètres sont substitués par leurs valeurs effectives du contexte de l'événement avant l'exécution de l'action.

L'association entre l'événement et l'action est définie comme suit :

TypeObject <typeName>
<DefAttributs>
Trigger
<Mode> **On** <Event> **Do** <Action>

Les actions ne sont pas forcément exécutées immédiatement lors de l'arrivée des événements correspondants, il est possible de spécifier le moment de l'exécution de l'action. Ainsi, plusieurs modes de contrôle sont définis :

- Le mode PRE indique que l'action doit être exécutée immédiatement avant l'exécution de la commande qui a provoqué l'événement. Ce mode permet de contrôler l'état du système avant l'exécution des commandes.
- Le mode POST indique que l'action doit être exécutée avant la validation de la commande qui a provoqué l'événement. Ce mode permet de valider ou de provoquer l'abandon de la commande. Ainsi, dans le cas de violation de certaines contraintes d'intégrité, la commande est défaite.
- Le mode AFTER indique qu'il faut exécuter l'action juste après la validation de la commande provoquant l'événement, dans le but de restaurer la cohérence des objets affectés par cette commande et de propager les effets de bord.
- Le mode FAIL indique que l'action est exécutée en cas d'échec ou d'annulation de la commande provoquant l'événement. Ce mode permet de programmer des opérations à exécuter en cas d'échec ou, tout simplement, d'avertir l'usager.

Les déclencheurs peuvent être définis sur les relations existant entre les objets et qui sont utilisées pour propager les effets de modifications sur les objets sources de ces relations. L'événement se produit lors de la demande d'exécution de la commande associée sur l'objet destination de la relation. Cet événement est propagé explicitement, en utilisant la primitive *propagate*, depuis la destination vers la source de la relation.

Certaines propositions sont faites pour contrôler le déclenchement des déclencheurs, tels que : l'utilisation d'un ordre de priorité pour contrôler la propagation de plusieurs événements sur une relation unique, l'utilisation d'un ordre de priorité associé aux relations pour contrôler la propagation de plusieurs événements sur plusieurs relations, la détection des boucles sur les actions.

3.3.2. PCTE+ , ALF

PCTE+ [God 92, Oqu 89] est une évolution de PCTE (A Basis for Portable Common Tool Environment) qui représente une structure d'accueil pour les EGLs. L'objectif de PCTE+ est de fournir l'interface public d'un ensemble de mécanismes de base permettant la construction d'environnements portables.

Le système de gestion d'objets SGO de PCTE+ est basé sur un modèle de données entité-association intégrant des notions de l'approche objet. Le modèle est fondé sur quatre concepts : les objets ayant des identités uniques, les liens binaires orientés entre les objets, les associations dont chacune est un couple de liens (l'un étant l'inverse de l'autre) et les attributs caractérisant les objets et les liens. Une base d'objets PCTE+ est un graphe ayant un unique point d'entrée qui est un objet particulier appelé common-root.

Pour modéliser des objets composites et exprimer la sémantique des liens, PCTE+ supporte plusieurs catégories de liens : les liens de la catégorie *existence* dont les objets cibles sont dépendants de l'existence de ces liens, les liens de la catégorie *composition* qui modélisent une relation composant/composé pour laquelle les composant peuvent être partageables ou exclusifs, ces liens donnent lieu à la notion d'objets composites, les liens de la catégorie *implicite* qui sont créés comme références inverses et les liens de *destination* qui représentent de simples designations entre les objets.

Les objets, les liens et les attributs sont typés. Les types d'objets constituent un treillis ayant le type prédéfini *Object* pour racine et permettant ainsi l'héritage multiple. Certains types d'objets sont prédéfinis, tels que, *file*, *message-queue*, etc.

Les définitions de types sont organisées dans des schémas appelés SDS (Schema Definition Sets). Chaque schéma définit une sorte de vue sur la base d'objets. Il est possible d'importer des définitions d'un schéma à un autre. Plusieurs schémas sont prédéfinis dans l'environnement initial de PCTE+, ils contiennent une description minimum de la base et font partie de tout schéma de travail.

PCTE+ fournit des mécanismes pour la gestion de versions. L'ensemble des versions d'un objet peut être représenté par un objet lié à toutes les versions. Il est possible d'avoir un arbre de versions à l'aide des liens successeur/prédecesseur et des opérateurs de manipulation de ces liens.

L'accès concurrent aux objets est géré de façon à préserver l'intégrité de la base et à assurer sa cohérence. Pour cela, les notions d'*activité* et de *verrou* sont utilisées. L'activité est le contexte dans lequel se déroulent un ou plusieurs processus. Une

activité peut être soit protégée contre la concurrence d'accès des autres activités, soit non protégée.

Le modèle de PCTE+ n'intègre pas les aspects dynamiques de donnée. Le contrôle de l'intégrité se limite à assurer l'intégrité de la base par rapport au schéma, au contrôle de conflit de nommage et au contrôle de règles de création et de suppression d'objets. Cependant, le contrôle de la dynamique est supporté par ALF [Der 90] qui constitue une couche au-dessus de PCTE+.

ALF (Accueil de Logiciel Futur) est un projet Esprit dont l'objectif est de décrire des modèles de procédés et d'assister les utilisateurs dans le développement de logiciels, un procédé étant un ensemble d'activités exécutées pendant le développement d'un logiciel.

Un modèle de procédé MASP dans ALF, décrit par le langage de description de MASPs (MASP/LD), comprend : un modèle d'objets (celui de PCTE), un modèle d'opérateurs qui représente une liste de types d'opérateurs, des expressions décrivant des prés et des post-conditions, des ordonnancements décrivant des contraintes sur l'ordre d'exécution des opérateurs et un ensemble de caractéristiques qui sont les contraintes à satisfaire pendant l'activation d'un MASP. Chaque type d'opérateurs définit une condition qui doit être satisfaite avant son exécution, une condition supposée vérifiée après son exécution, le type d'objets sur lesquels le type d'opérateur est permis ainsi que le type d'objets produits.

L'activation d'un MASP se fait en lui fournissant les informations nécessaires pour son exécution (instanciation) ce qui peut être fait explicitement.

ALF comporte aussi un interprète à base de règles offrant une assistance aux utilisateurs sous forme d'actions exécutées automatiquement dans des situations particulières. Chaque règle est définie par une association entre une expression et un type d'opérateurs.

3.3.3. MARVEL

MARVEL [Kai 88, Kai 87] est un environnement de programmation à base de connaissances destiné à assister les utilisateurs durant le développement, le test et la maintenance d'un projet. Il gère l'ensemble des entités produites pendant le développement d'un projet (modules, fonctions, etc), les relations entre les entités et l'activation d'outils génie logiciel (compilateurs, éditeur, etc), en utilisant une approche intelligence artificielle.

Le composant de base dans MARVEL est une base de données générique. Elle décrit la structure d'un projet en terme de classes d'objets ayant des attributs typés représentant la description de l'objet. Un type d'attributs peut être soit un type de base (entier, réel, booléen) soit une classe d'objets. Le concept d'objets composites est supporté dans MARVEL par les attributs ayant un objet comme valeur. Les classes sont organisées dans une hiérarchie permettant de définir un héritage simple entre les objets. Trois constructeurs de types sont définis : ensemble, séquence et union. Les relations entre objets sont modélisées par des associations binaires ne faisant pas partie des objets.

La gestion de la dynamique dans MARVEL se fait en simulant un système à règles de production. Les règles ici ont une double fonctionnalité : d'une part, elles modélisent le processus de développement en définissant les conditions nécessaires pour l'activation d'un outil, et d'autre part, elles automatisent l'activation de certains outils dans certaines situations.

Les règles de MARVEL sont similaires à celles des systèmes de production, elles en diffèrent par le fait qu'elles séparent l'action exécutée de ses effets sur la base des objets. En effet, Marvel sépare au niveau du modèle les entités réelles (fichiers) et les objets de la base représentant une abstraction du monde réel. L'action dans les règles correspond à l'activation d'un outil agissant sur un système de fichiers et non pas directement sur la base des objets. Elle est donc suivie par des post-actions qui effectuent les changements correspondant dans la base des objets.

Contrairement aux règles de production constituées de deux parties (condition/action), les règles de MARVEL sont donc composées de trois parties :

La première partie est une *précondition* qui doit être vérifiée avant l'exécution d'une certaine activité de programmation.

La deuxième partie est l'action appelée *activité* . Elle est définie par un nom d'outil, une opération et une liste d'arguments nécessaires pour l'exécution de l'opération. La distinction entre outil et opération est faite, car quelques outils mettent en œuvre plusieurs opérations. L'*activité* est externe par rapport à la base d'objets de MARVEL, elle agit sur un système de fichiers et non directement sur la base d'objets.

Une troisième partie est donc nécessaire pour refléter le résultat de l'activité sur la base d'objets. Celle-ci est constituée d'un ensemble de *post-conditions* dont seulement une devient vraie après l'exécution de l'*activité*. Généralement, une des post-conditions correspond à une terminaison réussie de l'activité, et chacune des

autres post-conditions représente un type d'erreur détecté par l'outil. Les préconditions et les post-conditions sont exprimées par des formules de la logique du premier ordre.

Pour provoquer le déclenchement du mécanisme de déduction, MARVEL considère deux types d'événements : les événements *internes* et les événements *externes*.

Les événements *internes* correspondent à des changements dans l'état de la base d'objets à la suite de la création d'un objet ou à la modification d'un objet existant (modification d'une valeur d'attribut ou d'une relation). La production d'un événement interne provoque l'évaluation des préconditions des règles. Le contexte de l'évaluation des règles étant l'état de la base au moment de la production de l'événement. Si la précondition d'une règle est satisfaite, l'activité correspondante s'enchaîne pour l'exécution, ce qui peut éventuellement changer l'état de la base et entraîner l'exécution d'autres règles. Ce mécanisme correspond au chaînage avant des règles dans les systèmes de production.

Les événements *externes* sont provoqués par la demande explicite de l'utilisateur pour exécuter une certaine *activité*. La règle correspondant à l'activité est déclenchée et sa précondition est évaluée. Le contexte de l'évaluation de la règle est l'état de la base lors de la production de l'événement et les paramètres effectifs de l'activité à exécuter. Si la précondition n'est pas vérifiée, un mécanisme de chaînage arrière essaye de satisfaire la précondition de la règle en enchaînant toutes les règles qui peuvent satisfaire une partie de cette précondition. Chacune de ces règles dont la précondition n'est pas vérifiée conduit à d'autres enchaînements. Le chaînage arrière s'arrête lorsque la précondition de l'activité demandée par l'utilisateur est satisfaite.

Par ailleurs, pour définir des environnements spécifiques de développement, MARVEL utilise la notion de *stratégie*. Une *stratégie* spécifie un ensemble de classes d'objets, d'associations, d'outils et de règles utilisés pour modéliser le processus de développement d'un ou d'un ensemble de logiciels.

3.3.4. DAMOKLES

DAMOKLES [Dit 87] est un noyau de gestion d'informations pour les EGLs, développé à l'Université Karlsruhe. Il est défini pour être général et indépendant de la stratégie de conception et de programmation utilisée. Le modèle de données de DAMOKLES appelé DODM (Design Object Data Model) essaie d'intégrer les différents aspects nécessaires à la gestion des informations en génie logiciel et

d'exprimer leur sémantique. Il est basé sur une approche entité-association étendue par la notion d'objets structurés et de versions.

DODM distingue les objets *simples* des objets *structurés* (ou *composites*). Un objet simple se compose de plusieurs attributs composant la partie descriptive de l'objet. Un ou plusieurs attributs sont désignés comme étant la clé de l'objet, auquel cas ces attributs obéissent à la règle de l'unicité de valeur. Un objet structuré consiste également en une partie structurelle qui désigne les objets composants pouvant à leur tour être simples ou structurés.

Les opérations de manipulation d'objets exprimant la sémantique d'objets composites sont supportées par une interface opérationnelle construite au dessus du gestionnaire d'informations.

Une instance d'un objet structuré peut être créée de deux façons différentes : les objets composants sont, soit créés automatiquement lors de la création d'une instance de l'objet père, soit existant déjà et, donc, insérés dynamiquement comme composants du père.

Les relations entre les objets sont représentées par des associations n-aires bidirectionnelles. D'une façon similaire à la description des objets, DODM permet aussi la description de types de relations, chaque relation est définie par les objets de ses extrémités, les cardinalités minimum et maximum et un ensemble d'attributs décrivant la relation.

DODM supporte plusieurs domaines de valeurs prédéfinies (*integer*, *boolean*, *string*) et un ensemble de constructeurs de valeurs (*subrange*, *enumeration*, *array*). En plus, le domaine LongField est défini pour représenter le contenu textuel des documents.

DODM supporte aussi la notion de versions. Chaque version est associée à un objet appelé l'*objet générique*. Le type de la version est celui de son objet générique. Bien que les valeurs d'attributs et la structure de l'objet générique soient communes à toutes ses versions, chaque version peut avoir ses propres valeurs d'attributs et sa propre structure. Pour structurer l'ensemble de versions pour un objet générique, une relation implicite de succession-précédence entre les versions est maintenue. Cette relation peut être linéaire, arborescence, ou un graphe acyclique.

Il est possible dans DAMOKLES de créer des bases de données multiples, les objets sont créés dans une base et peuvent être transférés d'une base à l'autres. Un objet et tous ses composants doivent résider dans une seule et même base, tandis que les

relations peuvent être créées entre objets de bases différentes. Les attributs de ces relations sont confiés à une de ces bases.

La notion de bases de données multiples alliée à celles des utilisateurs, groupes d'utilisateurs, bases privées, semi-publiques et publiques, procurent des mécanismes nécessaires de protection d'accès concurrent aux objets.

Pour supporter les aspects dynamiques de la gestion d'informations, DAMOKLES implante un mécanisme de déclencheurs approprié pour gérer des transactions longues. Ce mécanisme est supporté par le système et non intégré au niveau du modèle. Un déclencheur est un doublet (événement, action) où *événement* est un identificateur indiquant une situation spécifique et *action* est la réaction à cette situation. Les contraintes sont exprimées par des déclencheurs spécifiques dans lesquels les actions sont des conditions à vérifier. Plusieurs opérations de manipulation de déclencheurs, telles que, la définition, l'activation ou la désactivation sont définies. Les événements sont provoqués explicitement en utilisant l'opération *raise*.

3.3.5. Discussion

Nous venons de voir que les environnements à SGBD spécifique offrent déjà plusieurs mécanismes nécessaires pour la gestion d'informations en génie logiciel. Ces environnements utilisent largement les modèles sémantiques, et souvent un modèle Entité-Association étendu, c'est le cas par exemple de PCTE. Cette tendance vient du fait que ces modèles par leur richesse sémantique, favorisent la représentation des relations sémantiques entre les objets, et supportent plusieurs mécanismes d'abstraction nécessaires pour la gestion d'informations complexes.

Ainsi, des atouts importants sont mis en évidence surtout dans le cadre de PCTE+ et DAMOKLES (i.e., la représentation des objets composites, la notion de versions). Néanmoins, les mécanismes offerts par ces environnements ne sont pas toujours satisfaisants. Pour gérer les objets complexes (code source, texte), MARVEL par exemple sépare les entités du monde réel de ses abstractions dans la base de données. Cette approche a comme conséquence la dégradation des performances du système et des problèmes de cohérence entre la base de données et le monde réel (fichiers de code source et de texte). D'autre part, la gestion de versions d'objets se limite souvent à des objets simples et ne se généralise pas à des objets composites. Face à ces critères, NOMADE souffre d'une approche pragmatique relativement figée, ce qui n'est pas le cas des deux autres propositions.

Les modèles utilisés par ces environnements souffrent d'une pauvreté concernant la description de la dynamique des informations. Les mécanismes puissants offerts par ces environnements pour la gestion de la dynamique ne sont pas intégrés au niveau du modèle. C'est le cas par exemple de ALF qui procure d'un mécanisme fournissant un support actif aux développeurs de logiciels en ajoutant une couche au dessus de PCTE+.

En plus, les mécanismes de déclencheurs offerts par les EGLs ne traitent pas uniformément tous les problèmes liés à la dynamique. NOMADE, par exemple, dispose de plusieurs autres mécanismes pour la gestion de versions et de configurations ainsi que pour le contrôle de droits d'accès.

MARVEL utilise une approche inspirée des règles de production pour gérer les aspects dynamiques du système. L'intérêt de ce choix réside dans sa base théorique (logique du premier ordre). Mais son inconvénient est dû au fait que le chaînage se fait d'une manière implicite suite à un changement dans l'état de la base. Ainsi, à chaque modification, une évaluation de toutes les règles de la base est effectuée. Vu le nombre important de règles et la nature des opérations dans une application génie logiciel, ceci pose un problème d'efficacité et de contrôle. La complexité de cette proposition est une conséquence aussi du fait que MARVEL sépare les entités du monde réel (code source des programmes qui est géré par un système de fichiers) de ses abstractions représentées par la base. Ceci fait intervenir les post-actions pour refléter le résultat des activités dans la base des objets.

3.4. LES EGLS A BASE D'HYPERTEXTE

L'intérêt des systèmes hypertexte dans le contexte des EGLs [Big 88] réside dans le fait qu'ils procurent des mécanismes et des outils pour la représentation des informations multimédia (texte, graphique, code source, etc), pour la représentation des relations existant entre elles et pour l'accès direct aux informations.

Les Systèmes Hypertexte (SHs) sont basés sur la représentation non-linéaire du texte. Cette idée n'est pas récente, une première proposition a été faite par Buch en 1945 [Buc 45]. Elle n'a été développée qu'en 1963 par Engelbart dans un système quasi-hypertexte NLS (oN Line System) [Eng63] où se trouve la possibilité de lier entre eux des segments de fichiers. Ted Nelson a enrichi le terme hypertexte par la description de son système Xanadu [Nel 81]. Dans Xanadu, l'unité fondamentale est le document. Des opérations de création, de stockage et de recherche sont définies sur les portions de texte (fenêtres) et sur les liens existant entre ces portions.

Plusieurs SHs ont été développés à Brown University, en particulier : "Electronic Document System" qui est un système de documentation orienté graphique; Intermedia [Smi 87], un système hypertexte permettant à l'auteur de créer des liens entre documents de média variés (texte, graphique, image, vidéo et musique). Plusieurs autres systèmes ont été développés parmi lesquels: Xerox PARC's NoteCards [Hal 88]; Electronic Encyclopedia; CMU's ZOG [Rob 81] et Neptune [Del 87, Del 86]. Pour un survey détaillé sur les SHs voir [Con 87, Gar 88a, Dan 90, Sav 89].

L'intérêt que l'on porte aux SHs a été grandissant ces dernières années. En conséquence, les développements et les recherches dans le domaine ont explosé, et leur champ d'application s'est élargi pour atteindre plusieurs domaines, parmi lesquels le génie logiciel [Gar 90, Big 88, Blu 88, Fle 88, Big 87, Del 86]. Bien que ces développements soient assez largement différenciés, ils implantent malgré tout un certain nombre de concepts de base communs.

Comme les lecteurs ne sont pas obligatoirement familiarisés avec les systèmes hypertexte, nous commençons par une présentation générale de ces systèmes. Ensuite nous explorons deux exemples de SHs développés pour la gestion d'informations dans les EGLs : Neptune et DIF.

3.4.1. Définition des Systèmes Hypertexte

Pour illustrer l'idée essentielle des SHs, nous allons considérer la définition donnée dans [Con 87] et la figure 2 expliquant cette définition : "Le concept d'hypertexte est très simple : des fenêtres sur l'écran associées à des objets dans une base de données avec des liens définis entre ces objets. Les liens sont représentés sur les fenêtres graphiquement par des icônes, et dans la base de données, par des pointeurs entre les objets de la base".

Le modèle de la base de données d'un hypertexte est très simple : un graphe orienté dont les nœuds sont les portions de texte et les arcs (liens) modélisent les relations entre les différentes portions. L'ensemble des nœuds constituant un document est appelé *hyperdocument*. Les nœuds et les liens orientés d'un hyperdocument représentent un graphe orienté dit *hypergraphe*. Des paires (attribut/valeur) sont associés aux nœuds et aux liens de l'hypertexte pour les identifier et pour étendre les capacités du système.

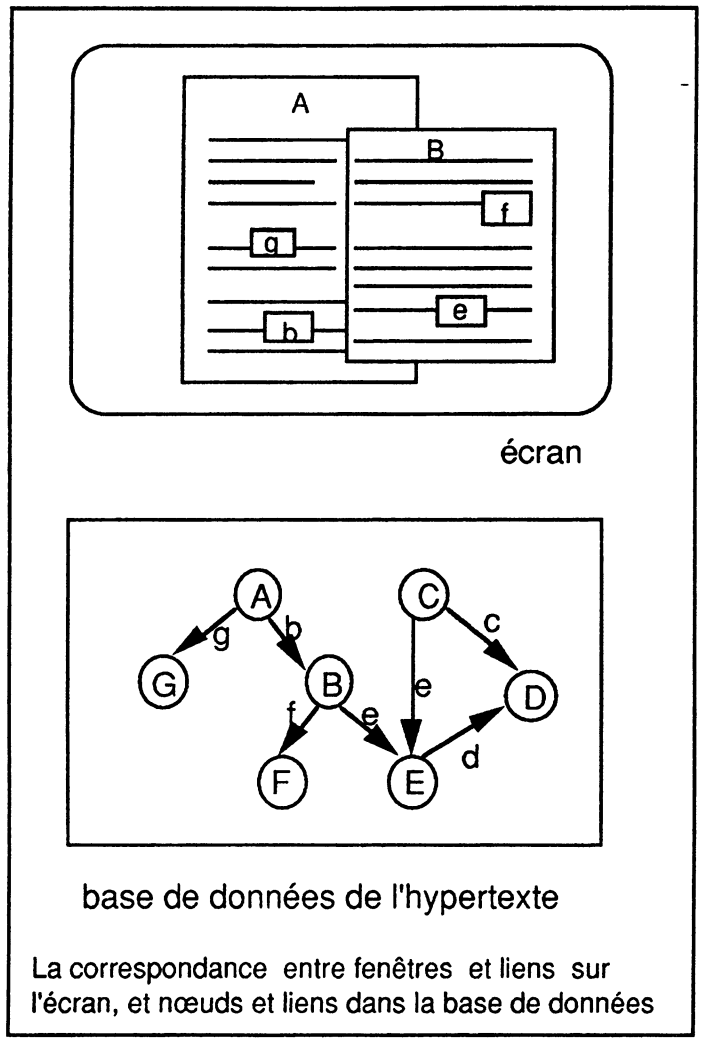


Figure-2

Les nœuds sont les unités de base, ils représentent les grains d'informations. Dans le cas général (hypermédia), le contenu d'un nœud peut être non seulement une portion de texte, mais aussi des éléments graphiques, des images, des données digitales, etc. En effet, un nœud peut être utilisé pour exprimer une entité sémantique (un concept ou une idée), ce qui consiste à modéliser le texte suivant les idées et à identifier les entités sémantiques par des entités syntaxiques que sont les nœuds de l'hypertexte. Des attributs peuvent être associés à chaque nœud pour l'identifier (le nom de l'auteur, la date de création, etc) aussi que le type de son contenu (texte, graphe, etc).

Certaines propositions introduisent la notion de nœuds composés (*webs* dans Intermedia, *tocs* dans textnet, *fileBoxes* dans NoteCards). Ce type spécial de nœuds permet d'organiser et de classifier des ensembles de nœuds en traduisant la structure logique des documents. Ils servent ainsi à établir des plans, des tables de matières, des vues partielles de l'hypergraphe, etc.

Les liens représentent les relations entre les nœuds de l'hypertexte. Certains systèmes proposent deux types de liens : les liens *référentiels* qui servent pour connecter un document aux documents référencés, aux annotations et (ou) aux commentaires, et les liens *organisationnels* reliant les nœuds composés aux nœuds fils pour représenter la structure logique des documents. Les liens peuvent avoir un ensemble de paires (attribut/valeur) associés pour les identifier et désigner leur rôle. Des opérations sont définies sur ces liens : la création d'un lien, la suppression d'un lien, la modification des noms ou des valeurs des attributs, etc.

Un des principes importants des SHs est l'accès direct à leur base d'objets, un principe appelé le dynamisme : l'hypergraphe peut être continuellement modifié, l'on peut créer, détruire, modifier des nœuds et des liens, restructurer des ensembles de nœuds tout en conservant la cohérence des informations. Détruire un nœud entraîne la destruction de tous les liens qui y sont attachés, et déplacer une partie du contenu à laquelle est attaché un lien peut entraîner le déplacement du lien. Le modèle de l'hypertexte doit tenir compte de son dynamisme.

Pour accéder aux informations de la base de données d'un hypertexte, deux modes essentiels d'interrogation sont définis:

- L'accès direct par navigation qui consiste à traverser les liens de l'hyperdocument et à accéder ainsi successivement aux nœuds [Nie 90]. Une autre façon de naviguer est de consulter une représentation du réseau ou d'une de ses parties "browser" et désigner le nœud à afficher.
- L'interrogation de la base par des requêtes utilisant des prédicats basés sur les paires attribut/valeur associés aux nœuds et aux liens de l'hypertexte.

3.4.2. Neptune et son Application DynamicDesign

Le système Neptune a été conçu en 1986 [Del 86, Del 87]. C'est un système à deux couches : la couche de haut niveau qui constitue une interface utilisateur, et la couche de base appelé Hypertexte Abstract Machine (HAM). Le HAM [Cam 87] est un modèle générique d'hypertexte basé sur cinq notions : *graphe, contexte, nœud, lien et attribut.* Il gère un historique complet de versions pour chacun des nœuds de façon à pouvoir faire référence à une certaine version ou à la version courante.

Neptune utilise le concept de *contexte* pour définir des partitions de documents. Construire un contexte consiste à définir un schéma selon lequel on rassemble un ensemble de nœuds et de liens suivant certains critères.

L'accès aux informations dans HAM se fait de deux façons différentes : en suivant les liens attachés à un nœud pour arriver aux nœuds suivants, ou en interrogeant à l'aide de prédicats basés sur les paires attribut/valeurs pour accéder directement à un ensemble de nœuds et de liens.

Une des applications de Neptune est DynamicDesign [Big 87], un EGL développé par Tektronix. Dans DynamicDesign tous les composants du projet : cahiers des charges, documents de spécifications, documents de conceptions, code source, code objet, résultats de tests et manuels d'utilisation, sont stockés dans les nœuds de l'hyperdocument. Les liens entre les nœuds représentent les relations qui relient les différents documents. Les attributs avec leurs valeurs associées décrivent les types et les caractéristiques des nœuds et des liens. Par exemple, la valeur de l'attribut *ProjectComponent* associé à chaque nœud identifie le composant du projet représenté par le nœud. Les valeurs possibles de cet attribut sont: *requirement, specification, designNote, designAssumption, comment, source, object, symbolTable, documentation, report.* L'attribut *RelatesTo* associé aux liens est utilisé pour désigner le type de la relation représentée par le lien. Les valeurs possibles pour cet attribut sont : *leadsTo, comments, refersTo, callsProcedure, followsFrom, implements, isDefinedBy.*

Dans DynamicDesign, toutes les versions du contenu pour un nœud peuvent être archivées et retrouvées suivant la demande.

Comme on l'avait vu, les *contextes* regroupent des nœuds et des liens ayant des caractéristiques communes. La notion de *contexte* est utilisée par DynamicDesign pour satisfaire plusieurs objectifs liés aux aspects dynamiques des informations : la gestion de configurations, la protection des informations par la création des espaces de travail et la définition de catégories de projet.

DynamicDesign présente une aide à la maintenance. En effet, les liens relient directement les documents de spécifications et de conceptions et les notes d'implantation aux portions de code correspondant. Ainsi, lorsque des modifications sont effectuées sur les portions de code le système indique directement les portions correspondantes de texte à modifier. De même tout changement dans les besoins ou les spécifications d'un logiciel peut impliquer des modifications à effectuer dans le code associé. Or, le système n'est pas capable d'exécuter des actions pour restaurer la cohérence de la base.

3.4.3. DIF (Documents Integration Facility)

Le système DIF [Gar 90, Gar 88b, Gar 87] constitue une aide pour intégrer et gérer les documents produits et utilisés durant le cycle de vie d'un logiciel. Les informations sont considérées comme des objets stockés dans les nœuds d'un hypertexte.

Deux types de nœuds sont considérés par DIF : les nœuds de base (appelés *Basic Templates* BT), ils représentent les unités de base de l'hypertexte contenant les portions d'informations. Les nœuds d'organisation (appelés *Forms*) qui définissent les structures hiérarchiques des documents et assurent ainsi une standardisation des documents utilisés par tous les projets.

En correspondance avec les types d'utilisateurs accédant à la base d'informations, DIF permet deux modes d'opérations : le mode *superUser* et le mode *generalUser*. En mode *superUser* il est possible de définir la structure du projet, les structures de documents (forms) héritées par tous les projets, les BTs, la nature des informations dans les BTs et les responsabilités des utilisateurs. En mode *generalUser* on peut modifier et naviguer sur les informations dans la base de l'hypertexte, on peut aussi créer des instances des BTs définis par le superUser ou définir des BTs additionnels.

Pour chaque BT, l'utilisateur peut définir un ensemble de mots-clés décrivant la sémantique des informations contenues dans ce nœud.

Des liens peuvent être créés par les utilisateurs entre les BTs pour définir les relations entre les différents documents. Les liens dans DIF peuvent êtres *opérationnels* : visiter ces liens consiste à l'exécution d'une certaine opération. Par exemple, le code source peut être lié au code exécutable correspondant par un lien opérationnel, traverser ce lien implique l'exécution du code correspondant.

Deux types de liens sont supportés par DIF : les liens nœud-à-nœud qui lient deux nœuds de l'hypertexte, et les liens point-à-point qui lient un point d'un nœud à un point d'un autre nœud.

Dans DIF il est possible de définir des *configurations*. Une *configuration* est similaire à une *Form*, elle en diffère par le fait qu'elle n'est pas héritée par tous les projets, seul l'utilisateur qui l'a créée peut l'utiliser. Les compositions sont utilisées pour désigner un ensemble de BTs ayant des caractéristiques en commun.

Les informations textuelles et graphiques de DIF sont stockées dans le système de fichiers d'Unix et utilisent ainsi RCS pour le contrôle de leurs versions multiples.

Le SGBD Ingres est aussi utilisé pour stocker les informations concernant le niveau structurel.

DIF est un environnement passif dans le sens qu'il ne participe pas au processus de développement. Il est en plus incapable de réagir suite à une éventuelle incohérence ou non-intégrité dans la base d'informations.

3.4.4. Discussion

Les hypertextes peuvent constituer un support adéquat pour les gestionnaires d'objets des EGLs. En particulier, ils procurent des mécanismes et des outils pour la représentation des informations multimédia (texte, graphique, code source, etc), pour la représentation des relations existant entre elles pour la gestion et la création de versions et de configurations, et pour l'accès direct aux informations.

Les EGLs à base d'hypertexte existants sont centrés sur les aspects interface utilisateur et sur la représentation des liens entre les portions arbitraires d'informations. Bien que ce soient des aspects très importants, ils ne sont pas cependant suffisants au niveau modélisation de données des EGLs. Ces systèmes souffrent du manque au niveau de la représentation de la sémantique liée aux données et de l'absence de mécanismes d'abstraction.

Neptune ne supporte pas la notion de composition et de typage. DIF utilise les formes pour représenter la structure des documents et pour imposer une structure commune à tous les documents d'un projet. D'une part, cette notion de structuration a besoin de s'enrichir sémantiquement pour pouvoir manipuler un ensemble de nœuds et de liens comme étant une unité abstraite et non seulement une collection arbitraire de nœuds et pour supporter la différence sémantique entre les liens de référence et ceux de composition. D'autre part, cette notion doit être généralisée pour définir un mécanisme de typage dans le sens des SGBDs.

Neptune et Dif, comme les SHs en général, peuvent être qualifiés de passifs. Il manque de moyen pour associer des opérateurs aux nœuds. Contrairement aux EGLs à SGBD spécifique, ces systèmes ne participent pas au processus de développement des logiciels ni au maintien de la cohérence référentielle et de l'intégrité des informations contenues dans les nœuds. Ils ne sont pas capables de réagir et d'exécuter automatiquement certaines actions dans le but de restaurer la cohérence et l'intégrité de l'hypertexte.

4. CONCLUSION GENERALE

L'état de l'art présenté dans ce chapitre fait ressortir les éléments essentiels pour évaluer les approches utilisées pour la gestion des informations dans le domaine génie logiciel.

Les gestionnaires d'objets des EGLs classiques ont identifié rapidement le besoin d'intégrer les techniques des bases de données pour résoudre les problèmes dus à la gestion d'un volume important d'informations, à la persistance, au partage et à l'intégrité de données. L'utilisation des SGBDs comme composant central des EGLs a montré l'incapacité des modèles conventionnels pour la représentation et la gestion des informations multimédia, d'une sémantique riche, d'une structure complexe et dont la dynamique joue un rôle important. En effet, à la différence des bases de données classiques qui stockent peu de types de données avec de très nombreuses instances et favorisent les opérateurs ensemblistes, les gestionnaires d'informations génie logiciel doivent manipuler de nombreux types de données n'ayant que peu d'instances.

Pour pallier cette insuffisance, les chercheurs en génie logiciel ont proposé des modèles dédiés supportant les techniques des SGBDs et les caractéristiques des EGLs. A cet égard des noyaux figés ou génériques ont été implantés. Les propositions faites pour supporter les différents besoins (gestion d'objets complexes, gestion de versions, gestion de configurations, gestion de tâches, transaction longue, etc) sont pragmatiques, nécessitant une intégration des différents mécanismes au niveau du modèle de données et une généralité pour gérer les différents aspects dynamiques d'une façon uniforme et homogène.

Par ailleurs, les systèmes hypertexte ont pris place dans le domaine de génie logiciel. Les utilisateurs des EGLs à base d'hypertexte ont profité d'une utilisation interactive et d'un accès direct aux informations qui remplace les modes d'accès ensemblistes peu appropriés dans ce type d'application. Toutefois, ces systèmes ont montré le besoin d'un modèle plus formel supportant les techniques des SGBDs et prenant en considération les mécanismes d'abstraction et le maintien de l'intégrité et de la cohérence de données.

En effet, ces derniers aspects ont commencé à être largement étudiés par les chercheurs en bases de données. Il est reconnu, depuis plusieurs années, que les SGBDs basés sur les modèles conventionnels sont mal adaptés aux besoins des applications non-conventionnelles telles que la Conception Assistée par Ordinateurs (CAO), la bureautique et le génie logiciel. Le support des mécanismes d'abstraction, de la sémantique des données structurées et de la dynamique représentent les voies

essentielles explorées par les recherches actuelles en bases de données. Ces recherches ont montré des résultats prometteurs pour satisfaire aux besoins des applications non-conventionnelles. Dans le chapitre suivant, nous faisons une étude de quelques propositions faites dans ce cadre pour identifier leur apport possible dans le domaine du génie logiciel.

CHAPITRE 2

LES SYSTEMES DE GESTION DE BASES DE DONNEES POUR LES APPLICATIONS NON-CONVENTIONNELLES

1. INTRODUCTION

Les systèmes de gestion de bases de données ont témoigné ces dernières années d'un élargissement de leur champ d'application à des domaines très variés et non-conventionnels tels que le CAO, la bureautique et le génie logiciel. Ces applications posent des problèmes nouveaux pour les bases de données dont la prise en considération nécessite l'exploration de voies nouvelles. Essentiellement, deux catégories de problèmes ont été identifiées par les recherches actuelles en bases de données :

- La représentation des objets structurés, de nature multimédia et en versions multiples. Ces différents critères sont regroupés généralement sous le terme *aspects statiques* de données.
- Les problèmes liés aux opérateurs de manipulation, au comportement, au maintien de la l'intégrité et de la cohérence des données, appelés généralement *aspects dynamiques* des données.

Il est largement reconnu maintenant que les SGBDs basés sur les modèles de données classiques sont mal adaptés aux besoins des applications non-conventionnelles. Bien que ces systèmes, et surtout ceux basés sur le modèle relationnel, soient parfaitement adaptés à certains problèmes des applications conventionnelles (certaines formes de typage, accès ensembliste à un volume important d'informations, intégrité), ils ne supportent pas les mécanismes de structuration, la nature textuelle, la taille importante, la gestion de versions multiples d'objets et les traitements et l'accès individuel aux objets. Pour résoudre ces problèmes, deux axes de recherche ont été élaborés :

- L'adaptation des modèles de données existants en leur apportant des extensions simples et puissantes. C'est notamment le cas du modèle relationnel étendu : POSTGRES [Sto 89], ETM [Dit 86], ARIEL [Han 89].

- La définition de nouveaux modèles de données à sémantique plus riche. C'est le cas de nombreux modèles ayant cette ambition : modèles sémantiques, logiques, fonctionnels ou orientés-objet. Parmi les propositions utilisant une approche sémantique nous citons TIGRE [Vel 84] et FAKIR [Col 87a] dans le domaine de la bureautique, ESTRELLA [Dam 88] pour les applications géographiques, et ETIC [Fau 88] pour les applications CAO.

  L'approche la plus en vogue actuellement est l'approche objet dont les réalisations commencent à être nombreuses : ORION [Kim 89], HiPAC [Day 88], O2 [Lec 87], IRIS [Ris 89], et des SGBDs extensibles tel que EXODUS [Car 88].

Dans les paragraphes suivants nous allons explorer l'apport de ces deux axes de recherche des deux points de vue : statique et dynamique.

2. EXTENSION AU MODELE RELATIONNEL

Le modèle relationnel a été conçu pour gérer les données statiques, volumineuses, constituées d'entités très simples (n-uplets). Ce modèle privilégie les traitements associatifs. Pour s'adapter aux applications complexes, le modèle relationnel a été étendu avec des concepts et des fonctionnalités appropriés. Parmi les exemples de ces extensions nous citons :

- le système POSTGRES [Sto 89], successeur d'Ingres, réalisé à l'Université de Californie pour étendre les modèles relationnels en vue de supporter le concept d'objet complexe et le mécanisme de *trigger*.
- ETM (Event/Trigger Mechanism) [Dit 86], qui étend le mécanisme de *trigger* proposé par les modèles relationnels pour les associer à des événements arbitraires. Il est développé au Forschungszentrum Informatik à Karlsruhe.
- ARIEL [Han 89], un SGBD relationnel étendu par un système de règles pour construire une base de données active.

2.1. ASPECTS STATIQUES

Pour gérer les données multimédia (image, texte), les extensions aux systèmes relationnels ont abordé deux voies essentielles [Adi 89, Val 87, Abi 87]. La première consiste à gérer les données de grande taille comme des attributs de type 'long-string' stockés dans un système de fichiers classiques. Le SGBD sert à retrouver ces données mais leur interprétation est dévolue à des programmes spécifiques interagissant avec le SGBD. Cette solution impose une forte dépendance

données/programmes et oblige à une duplication possible des traitements d'où une certaine inefficacité. La deuxième consiste à augmenter le SGBD par des nouveaux types de données dont l'exemple est le type *text* dans POSTGRES. Or, les problèmes liés à la manipulation des objets multimédia restent ouverts du fait que le modèle relationnel ne permet pas de décrire en même temps que ces nouveaux types les opérateurs spécifiques pour les manipuler. L'utilisation des Types Abstraits de Données [Gut 77] offre une solution élégante à ce problème en englobant les types de données et leurs opérations d'accès.

Quant à la gestion des objets structurés, elle est plus complexe et souvent non abordée par ces extensions du fait de la structure plate imposée par les n-uplets relationnels. Certaines propositions permettent de représenter artificiellement des objets structurés sous forme de n-uplets se trouvant dans plusieurs relations, et formant logiquement les composants d'un même objet. Dans ce cas la sémantique des liens doit être explicitée par des contraintes d'intégrité dont le contrôle est assez difficile. Cependant, le modèle relationnel étendu N1NF (Non First Normal Form) [Ozs 87] a essayé de décrire des données structurées et complexes. Il a généralisé la notion de relation du modèle relationnel en utilisant les constructeurs ensemble et n-uplet et en autorisant que les attributs d'un n-uplet soient une relation ou un autre n-uplet. Néanmoins, ces extensions restent descriptives et aucun modèle ne supporte la sémantique de composition d'objets.

Par ailleurs, d'autres fonctions sont désirables afin d'accepter de nouvelles applications dont l'exemple est la gestion de versions multiples de données et de leur historique.

2.2. ASPECTS DYNAMIQUES

Le contrôle de la dynamique a toujours été un des problèmes les plus complexes. Dans le modèle relationnel, la gestion de la dynamique correspond à la définition et à la vérification de contraintes d'intégrité. Les contraintes d'intégrité expriment les états valides de données, notion très statique adaptée aux SGBDs classiques, de plus elle est difficile à mettre en œuvre du fait de gros volumes de données gérées.

La notion de la dynamicité dans les SGBDs commence à être largement étudiée [Han 89, Dal 88, Col 87a, Dit 86, etc]. On la retrouve sous le terme SGBD actifs [Ber 90], ces systèmes permettent de déclencher certaines actions dans des situations, appelées *événements*, liées à la base. De manière générale, l'expression de la dynamique des données se fait par l'intermédiaire de règles de la forme événement-condition-action appelées déclencheurs, et gérées par un mécanisme de déclencheurs.

Dans les extensions au modèle relationnel, ce mécanisme est souvent appelé *trigger*. Il est utilisé dans la plupart des cas pour résoudre les problèmes d'intégrité de données, dans ce cas, la partie action d'une règle exprime les traitements à effectuer lors d'une violation de contraintes. Il est utilisé aussi pour introduire une fonctionnalité de déduction au SGBD relationnel, ceci revient à ajouter au SGBD des fonctionnalités d'un système de production.

Les *triggers* permettent de spécifier les contrôles à effectuer lorsque des opérations susceptibles d'altérer la cohérence de la base s'exécutent. Un *trigger* est composé de trois parties : un événement qui est à l'origine de l'activation du trigger, une condition qui permet une exécution conditionnelle de l'action qui constitue la troisième partie du *trigger*.

Les seuls ***événements*** pris en compte sont les événements, appelés internes, liés aux opérations de mise-à-jour de relations (ajout de n-uplet, modification de n-uplets, suppression de n-uplets). Les événements sont donc liés au changement d'état des données de la base. Par exemple, dans la norme ANSI et ISO de SQL3 [ANS 89], la déclaration suivante crée un *trigger* avec comme événement déclencheur l'opération de mise à jour de l'attribut SALAIRE de la relation Emp décrivant les employés :

```
Create Trigger T-Salemp
Before Update Of Salaire On Emp
Referencing Old AncEmpl New NouvEmpl
When ...
```

Dans cet exemple on peut remarquer que l'événement est défini non seulement par l'identification de l'opération de mise à jour, mais aussi par l'indication de l'instant où s'exécute l'opération à laquelle l'événement sera signalé (Before ou After).

Dans POSTGRES qui simule un système de production, l'événement est toujours lié à une opération de mise à jour de n-uplets de relations, cependant, le mode d'expression et le mécanisme d'activation associés sont différents. Les règles étiquetées par Always sont considérées comme des déclencheurs qui permettent d'exécuter une action à chaque fois qu'un événement de mise à jour d'une relation a lieu. L'exemple suivant définit un *trigger* qui détruira les n-uplets de la relation *Dept* correspondant aux départements qui n'ont pas d'employés. L'événement se produit lorsqu'il y a une mise à jour de la relation *Emp* :

```
Delete Always Dept
Where COUNT (Emp.Nom By Dept.DNom Where Emp.Dept=Dept.DNom)=0
```

Comme dans POSTGRES, le mécanisme de *triggers* d'ARIEL correspond à un système de production, et les événements pris en compte sont les opérations de manipulation de données (append, delete, replace , retrieve). Les *triggers* sont exprimés ainsi :

```
Define Rule NomRegle
[Priority ValPriorite]
[On Event ]
[If Condition]
Then Action
```

La clause ON étant optionnelle, ARIEL permet de définir une règle avec ou sans événement déclencheur. Si la règle est définie sans événement, elle est activée lors de la satisfaction de sa condition.

Une extension est faite par ETM en définissant l'*event/trigger* mécanisme. Ce mécanisme consiste à associer des actions à des événements arbitraires et non seulement à des états prédéfinis de la base de données. Dans cette proposition, un événement est un indicateur qui signale qu'une situation spécifique a eu lieu et pour laquelle des réactions peuvent être nécessaires. Un type d'événement est défini par un identificateur unique et une liste de paramètres formels qui servent pour passer à l'action des informations concernant le contexte de l'événement. Un type d'événement peut avoir un nombre arbitraire d'occurrences créées par l'opération *raise*. On trouve ici la notion d'événements externes signalés de l'extérieur du système.

Les *triggers* permettent de placer une ***condition*** à l'exécution de l'action associée à un événement. Cette condition se place comme un affinement de l'événement survenu afin de restreindre son contexte. Dans la norme SQL la condition est une expression similaire à celle d'une clause *where* d'une requête SQL. Dans l'exemple suivant d'un *trigger* exprimé dans SQL3, la clause *When* représente la condition à évaluer avant l'exécution de l'action :

```
Create Trigger GrosSal
After Update Of Salaire On Emp
When Salaire > 30000
Insert Into ...
```

Le contexte de l'évaluation de la condition est l'état de la relation sur laquelle a été défini le *trigger*. En effet, ce contexte n'est pas forcément limité à cette relation, on peut avoir dans la clause condition des requêtes faisant intervenir d'autres relations de la base. C'est le cas de la règle suivante de POSTGRES faisant intervenir la relation *Emp* en plus de *Dept* dans la clause condition :

```
Delete Always Dept
Where COUNT (Emp.Nom By Dept.DNom Where Emp.Dept=Dept.DNom)=0
```

Le contexte de l'évaluation de la condition peut aussi accéder à l'état avant et à l'état après la mise-à-jours effectuée sur la base. Par exemple, dans le cas de SQL3, si on veut repérer l'augmentation de salaires on définit la règle suivant :

```
Create Trigger GrosSal
Before Update Of Salaire On Emp
Referencing Old As AncEmp New As NouvEmp
When NouvEmp.Salaire > (AncEmp.salaire*1.05) ...
```

Dans d'autres approches on trouve des définitions plus formelles du contexte d'évaluation des conditions basées sur la notion de ΔRelation [Ros 89] représentant des couples d'objets avant et après modifications.

L'évaluation de la condition consiste à trouver les n-uplets des relations qui vérifient la formule représentant cette condition. Cette opération est souvent effectuée par un calcul algébrique dans les SGBDs relationnels.

La condition est parfois omise, si elle existe elle se trouve alors noyée dans le code de l'action. C'est le cas de ETM où les *triggers* sont l'association entre un événement et une action T=(E, A).

L'action définit les traitements associés à l'événement survenant dans certain contexte. Elle est spécifiée dans un langage de haut niveau de type SQL. Elle représente donc un ordre ou une séquence d'ordres SQL de mise à jour de la base. Le contexte de l'évaluation de l'action est celui créé par la condition.

Dans SQL3 par exemple, l'action est une requête SQL de modification de relation. L'exemple suivant montre l'utilisation d'une requête d'insertion d'un n-uplet dans la relation GrosContribuable:

```
Create Trigger GrosSal
After Update Of Salaire On Emp
When Salaire>30000
Insert Into GrosContribuable Values (Nom, Prénom, Salaire) For
Each Row
```

L'action dans POSTGRES est spécifiée en tête de la règle après le préfixe Always. Par exemple, pour assurer que les deux employés Mike et Bill touchent le même salaire, on définie la règle suivante:

```
Always Replace Emp (salaire =E.salaire)
Using E In Emp
Where Emp.nom = "Mike" And E.nom = "Bill"
```

D'autres approches proposent une vision plus large de la notion d'action. L'action dans ETM, par exemple, fait l'objet d'un module séparé. C'est un programme défini dans un langage de haut niveau qui comporte les opérations de base de données. L'action est définie par un identificateur unique, les paramètres formels et le corps de l'action. Les actions peuvent être exécutées explicitement par la primitive *exec*, ou implicitement lors de la production de l'événement associé par les *triggers*.

Certains systèmes relationnels disposent d'actions particulières de contrôle permettant d'annuler la cause de l'événement : *refuse* dans POSTGRES et *cancel* dans ETM. Ces actions permettent d'annuler les effets de l'opération origine de l'activation du *trigger*.

Dans les systèmes relationnels, l'action spécifiée est obligatoirement exécutée immédiatement lors de l'arrivée de l'événement. Il n'est pas possible de contrôler d'une façon asynchrone le moment où la partie action va être exécutée. Une des rares exceptions est ETM, où il est possible d'exécuter les actions d'une manière synchrone ou asynchrone.

Généralement dans les systèmes relationnels (SQL3, POSTGRES), les *triggers* sont pris en compte dès leur déclaration. Il n'est pas possible de mettre en œuvre un *trigger* seulement au moment nécessaire. Les modèles plus avancés comme ETM permettent de séparer la définition des *triggers* de leur utilisation, ce qui permet d'augmenter l'efficacité du système et de simplifier l'expression des conditions. Ainsi, les *triggers* peuvent être activés et désactivés dynamiquement en utilisant les primitives *activate* et *deactivate*. Un *trigger* désactivé ne cause pas de réaction si son événement est survenu.

3. APPROCHE ORIENTEE-OBJET

Face à l'insuffisance du modèle relationnel et de ses extensions pour la manipulation des données multimédia, la représentation de la structure complexe, la représentation des versions multiples de données et la gestion de la dynamique, l'approche objet présente des solutions plus convenables [Adi 90, Ban 87a, Ban 87b, Smi 87].

L'idée de base de l'orientation objet est l'encapsulation, dans la notion d'objet, des données et des opérations permises sur ces données. Bien que chaque système orienté-objet possède des caractéristiques différentes, nous pouvons dégager certains concepts communs à l'approche objet :

- *Objet* : donnée identifiée d'une manière unique. A chaque objet est attachée une structure et un ensemble d'opérations appelées *méthodes* permettant de le manipuler.
- *Classe* : ensemble d'objets ayant une même structure et sur lesquels il est possible d'appliquer les mêmes opérations.
- *type* : description associée aux classes d'objets définissant la structure et les opérations permises sur les objets de ces classes.
- *héritage* : notion permettant de raffiner les définitions des classes en introduisant les concepts de généralisation et de spécialisation de classes.

Certains SGBDs, appelés spécialisés, sont conçus pour s'adapter à un domaine d'application donné. Nous citons par exemple :

- le système ETIC [Fau 88] développé à Grenoble au LGI. Ce système est dédié aux applications CAO, il est enrichi par un modèle de versions permettant la gestion d'un projet CAO.
- le système ESTRELLA [Dam 88] pour les applications géographiques qui utilise une approche objet pour la gestion des informations multimédia.

D'autres SGBDs sont ouverts dans le sens qu'ils ne sont pas spécifiques à une application particulière. C'est le cas du :

- système ORION [Kim 89, Kim 88c, Kim 87] : un système conçu à MCC, ouvert aux applications bureautiques, CAO, etc. Il offre une définition précise pour les objets composites et pour les liens d'existence et de référence entre les objets.
- système EXODUS [Car 88] : un système orienté-objet extensible permettant la définition d'objets complexes. Il est conçu à l'Université de Wisconsin, Madison.
- système HiPAC [Day 88] : un SGBD actif, développé à Computer Corporation of America, utilisant une approche objet pour la modélisation de déclencheurs.
- système IRIS [Ris 89] : un système développé à Hewlett-Packard Laboratories, Paolo-Alto (USA). Il utilise une approche fonctionnelle pour la gestion des attributs d'objets dérivés ou stockés. Il offre la possibilité de gérer la dynamique en supportant un mécanisme appelé *moniteurs*.

Dans les deux paragraphes suivants nous allons citer les principales caractéristiques de l'approche objet des deux points de vue statique et dynamique à la lumière des exemples cités précédemment.

3.1. ASPECTS STATIQUES

Le couplage données-traitement proposé par l'approche objet permet entre autres la prise en compte de la particularité des applications multimédia. Il est en effet possible d'encapsuler les nouveaux types de données (texte, image, etc) et leurs opérations de manipulation tout en cachant les détails de l'implantation.

Grâce à la notion d'encapsulation, les modèles orientés-objet favorisent aussi l'expression de la sémantique de données et de leur structure complexe.

Dans ORION, un objet composite est défini comme étant une hiérarchie d'objets exclusifs. Chaque objet est composant d'au plus un objet composite, et son existence dépend de l'existence de l'objet dont il est composant. On prend donc en compte des liens existentiels entre un objet composite et ses composants. Par exemple, le *Body* d'un *Vehicle* est la propriété exclusive de ce *Vehicle* et ne peut pas appartenir à un autre *Vehicle*. Lors de la destruction du *Vehicle* on détruit également son *Body* et son *Drivetrain*. Bien qu'un objet dépendant ne puisse pas être référé plus d'une fois par des liens de composition, il peut être référé par d'autres liens que ceux de composition.

Dans ORION, on définit également la notion de propagation de valeurs. Des valeurs par défaut peuvent être propagées d'un objet composite à tous ses composants pour simplifier la définition des objets composants. Ceci constitue une sorte de partage de valeurs d'attributs entre les objets. La propagation des valeurs n'est pas automatique, elle doit être spécifiée dans le schéma de l'objet composite.

Pour manipuler et gérer des objets composites, le modèle de données EXTRA d'EXODUS dispose de plusieurs types de base (integer, boolean, character, string, etc) et de plusieurs constructeurs de types (n-uplet, tableau avec longueur fixe ou variable, ensemble). Exemple:

```
Type Personne
   n-uplet numss : Int4,
           nom : Char[],
           rue: Char[20],
           ville : Char[10],
           anniversaire : Date,
```

En plus des constructeurs définis précédemment, EXTRA permet trois modes d'attributs (*own, ref, own ref*) pour associer des sémantiques différentes aux

attributs et, ainsi, modéliser les objets composites et les références entre les objets. Exemple:

```
Type Employee
   n-uplet travail: Char[20],
           dep : ref Departement,
           directeur : ref Employe,
           salaire : Int[4],
Type Departement
   n-uplet nom: Char[],
           nbEmploye : Int[4],
           employes : {ref Employe},
```

Un attribut du mode *own* représente une valeur atomique associée à l'objet, ce sont les attributs de bases de données classiques (nom, date, etc), ils ne représentent pas d'objets référés. Un attribut du mode *ref* représente simplement une référence à un autre objet indépendant. Ce qui signifie que l'existence de l'objet référé n'est pas liée à celui de l'objet qui lui fait référence. Par contre, les attributs *own ref* constituent des références avec la contrainte additionnelle de possession de l'objet référé, ceci est équivalent à la sémantique associée aux liens de composition dans ORION. Les objets composites définis dans EXODUS ne permettent pas le partage de composants entre plusieurs objets composites.

Les modèles orientés-objets ont accordé une attention particulière au problème de versions multiples d'objets. Dans ORION [Kim 88a], chaque objet est une instance d'une classe *à versions* ou pas. Dans une classe *à versions,* un objet peut avoir en général plusieurs versions structurées sous forme d'une hiérarchie de dérivation. Le terme *instance de version* est utilisé pour désigner une instance particulière de la hiérarchie. L'objet (la hiérarchie de versions) est appelé instance *générique.* Les références entre les objets peuvent porter soit sur les instances de versions soit sur les instances génériques, dans ce dernier cas un rattachement par défaut est prévu.

Dans ETIC, une version est un état de l'objet que le système ou l'utilisateur veut conserver. ETIC considère la notion de version d'objet composite, décrite par l'ensemble des objets qui la composent, chacun pour une version donnée. Deux critères d'évolution sont pris en considération, ce qui donne lieu aux versions *dérivées* et versions *alternatives* structurées sous forme d'un arbre. Chaque version est caractérisée par sa situation dans l'arbre de dérivation, son contexte de création, sa classe d'équivalence et son état. Le contexte de création d'une version représente l'ensemble des informations déduites automatiquement par le système à la création de la version : son ancêtre, la date de création, l'auteur, la date de dernière modification, le type de la version. Les classes d'équivalence de versions permettent

de caractériser les versions d'un objet suivant un ensemble de critères exprimés par des règles données par le concepteur; les versions appartenant à la même classe d'équivalence vérifient toutes les règles associées à cette classe. L'état d'une version la caractérise en ce qui concerne sa cohérence et sa complétude.

3.2. ASPECTS DYNAMIQUES

Dans l'approche objet, la dynamique a été gérée de manières très variées. Certains systèmes ont proposé des mécanismes de déclencheurs, plus ou moins intégrés au niveau du modèle, mais dans la plupart des cas le contrôle de la dynamique est noyé dans le code des méthodes.

Dans ORION, des invariants, exprimés sous forme de règles, assurent la validité des mises-à-jour en refusant certaines opérations qui rendent la base incohérente. Pour assurer la cohérence, ORION propose un système de versions d'objet et de schéma couplé à un système de filtre agissant avant l'exécution des méthodes.

ETIC propose un système de contrôle de versions qui laisse à l'usager la responsabilité de décider si tel état de l'objet doit être ou non considéré comme une version. Pour manipuler les versions plusieurs opérations sont définies : *gel*, *dégel*, *création*, *dérivation*, *abandon*. Les modifications de certaines versions, considérées *gelées*, sont interdites, les seules opérations possibles sont le dégel et la dérivation. Le système de contrôle de versions a pour objectif également de contrôler le nombre de versions : limiter l'accroissement trop important des versions, obliger un nombre minimum de versions, et donc définir des règles pour la génération de versions. En plus, il contrôle la propagation de mises-à-jour d'une version aux versions qui l'utilisent.

Alors que dans les approches relationnelles l'introduction du mécanisme de déclencheurs apparaît comme une extension au modèle, les approches orientées-objet visent à intégrer cette notion dans le modèle (voir [Ber 90]). Celles-ci par leur richesse sémantique et par la possibilité d'encapsulation ont franchi un pas dans l'expression et la manipulation du mécanisme. En plus de certaines fonctionnalités externes, le mécanisme est utilisé ici d'une façon uniforme pour implanter plusieurs fonctionnalités du SGBD : contrôle d'intégrité, contrôle d'accès, manipulation des données dérivées, définition et application de l'héritage, notification de changements, mesure de performance, inférence, gestion de configurations.

Dans les approches orientées-objet la notion d'***événement*** n'est pas liée seulement aux opérations standards de manipulation de données, mais aussi à des opérations définies par l'usager (méthode, fonction). Ces événements correspondent à des

événements internes signalés par le système. Dans Iris, qui représente une approche fonctionnelle, toute fonction *Fn* est considérée comme un événement dès qu'il est déclaré *moniteur* en utilisant la primitive *DefineMonitor*(Fn). De plus, la définition d'un événement se précise : un événement est lié au début ou à la fin d'une opération.

Le champ des événements s'élargit dans les approches orientées-objets pour considérer d'autres types d'événements tels que les événements externes et les événements temporels. Les événements externes sont ceux signalés par les usagers ou les programmes ou ceux produits suite à la réception de messages en provenance de l'extérieur du système. Dans HiPAC par exemple, l'événement externe Flight-Airborne (Flight-No, Destination, Takeoff-Time, Aircraft,Wind-Speed, Wind-Direction) peut être signalé par l'usager quand un avion décolle.

Les événements temporels sont considérés par les systèmes qui gèrent le temps. Ils permettent de lancer périodiquement une opération, de coordonner des activités, etc. HiPAC propose une expression de temps soit d'une manière absolue (9:00:00 a.m., April 10,1988) soit d'une manière relative (30 sec après la production d'un événement E), et offre ainsi une grande richesse d'expression des événements temporels. Il permet de construire des types d'événements sur le temps: point de temps, intervalle de temps, durée, etc.

Pour enrichir la sémantique des événements, les approches objet proposent des opérations explicites sur les événements. Par exemple, il est possible en utilisant l'opération *signal* de signaler un événement dans HiPAC. D'autres opérations permettent de séparer la définition d'un événement de son utilisation. Il est donc possible de mettre en œuvre un événement aux seuls moments nécessaires. Cette possibilité est offerte par IRIS en utilisant les opérations *ActivateMonitor* et *DesactivateMonitor*, et par HiPAC en utilisant les opérations *enable* et *disable*. Cette possibilité permet d'augmenter l'efficacité du système et de simplifier l'expression des déclencheurs.

Certaines modèles dont HiPAC proposent des constructeurs d'événements (disjonction, séquence, etc) qui permettent de définir un événement comme une composition d'autres événements et de faciliter ainsi l'écriture des règles.

La ***condition*** est une requête, exprimée dans le langage de manipulation de données et pouvant faire référence à des arguments en provenance de l'événement. Elle est placée avant l'action pour permettre une exécution conditionnelle de cette action et pour passer son résultat comme contexte d'exécution de l'action. Une condition est considérée satisfaite si elle donne un ensemble non-vide d'objets.

En plus de la condition, une mode de couplage peut être défini pour indiquer le moment de l'évaluation de la condition par rapport à la production de l'événement et pour permettre ainsi une sorte d'asynchronisation. Dans HiPAC trois modes de couplage sont possibles : *immédiat, différé, détaché*. Dans le cas d'un mode *immédiat* ou *différé*, la transaction de l'évaluation de la condition est un sous-transaction de celle de l'événement qui a causé le déclenchement de la règle. Pour un couplage *immédiat*, la transaction de l'évaluation de la condition est créée et exécutée au moment de la production de l'événement, tandis que, pour un mode *différé*, cette transaction est créée juste après l'exécution de son événement origine. Dans le cas d'un couplage *détaché*, l'évaluation de la condition se fait dans une transaction séparée et concurrente à celle de l'événement.

La notion d'action est très générale dans les approches objet. Par exemple, dans HiPAC une action est une séquence d'instructions pouvant être des opérations sur la base de données ou des requêtes externes passées aux programmes d'applications. Des actions particulières sont définies pour contrôler les règles : rendre une règle inactive, ou faire passer une règle de l'état actif à l'état inactif.

L'aspect important des déclencheurs est la propriété d'asynchronisation : l'action n'est pas obligatoirement exécutée immédiatement lors de l'arrivée de l'événement. Dans Iris l'action associée à un événement peut être exécutée de façon asynchrone à la fin de la transaction de l'événement responsable de l'activation du déclencheur. Dans HiPAC, grâce au mode de couplage *immédiat, différé, séparé*, l'utilisateur peut contrôler le moment de déclenchement de l'action par rapport à l'évaluation de la condition. Des mécanismes de transaction assurent la cohérence de cette proposition.

Dans certaines propositions et notamment HiPAC un déclencheur est considéré comme un objet au même titre que les autres objets de la base. Ceci permet une manipulation des déclencheurs similaires à celle des autres objets (création, destruction, etc). De plus, d'autres opérations spécifiques aux déclencheurs peuvent être définies pour influencer le processus de déclenchement et d'exécution des règles. Par exemple, dans HiPAC un déclencheur est activé automatiquement une fois que son événement s'est produit. Mais il est possible d'activer explicitement un déclencheur en utilisant l'opération *fire* qui cause l'exécution de son action, si sa condition est satisfaite.

4. CONCLUSION GENERALE

Dans ce chapitre nous venons d'effectuer un état de l'art des recherches faites dans le domaine des bases de données pour satisfaire aux besoins des applications non-conventionnelles.

Le modèle relationnel est sémantiquement très pauvre pour répondre aux besoins des nouvelles applications telle que le génie logiciel. Cette pauvreté se manifeste par : la difficulté de représenter la structure logique complexe, la rigidité des types face aux nouveaux types de données (texte, image, etc) nécessitant d'exprimer des opérateurs spécifiques de manipulation, la difficulté de représenter des versions multiples et la complexité de vérification des contraintes d'intégrité et de contrôle de cohérence. Dans ce contexte, le problème a été abordé en envisageant deux extensions possibles au modèle relationnel : l'introduction de types abstraits de données et l'extension de la notion de relation avec des attributs à structure complexe. Cependant, d'autres fonctions importantes, telles que le support de modes d'accès autres qu'ensemblistes, du partage de composant et des transactions longues, de la gestion de versions multiples et du contrôle de l'évolution et de la dynamique, sont désirables afin d'accepter de nouvelles applications.

Face à l'insuffisance du modèle relationnel et ses extensions, les liens entre les modèles sémantiques et orientés-objet d'un côté et les nouvelles applications de l'autre côté se sont établis. Les modèles sémantiques par les concepts d'agrégation, de classification et de généralisation ont favorisé le support de mécanismes d'abstraction. Les modèles orientés-objet ont profité de la notion de l'identité d'objet et du couplage données-traitement pour intégrer le support de la dynamique des données à leurs aspects statiques.

La plupart des modèles orientés-objet prennent en considération la particularité des applications multimédia en permettant l'ajout de nouveaux types et la définition des opérateurs de manipulation de ces types. Ceci fait que ces modèles intègrent, avec le même support, la représentation de données de granularité variable allant d'un document de type texte à une variable. Les modèles orientés-objet favorisent également la représentation de la sémantique des données et de leur structure complexe. ORION et EXODUS présentent des solutions convenables à cet égard en définissant des sémantiques assez précises pour les associations entre objets. Or, dans ORION, tout est objet, une approche souhaitable dans certains cas, mais dans une application, telle que les EGLs, où la description et les propriétés jouent un rôle important et veulent se différencier des objets, il est préférable de considérer une approche par les objets et les valeurs. Quand à EXODUS, comme c'est le cas des

premières propositions d'ORION, il ne permet pas le partage des composants entre plusieurs objets composites. Le partage veut dire ici qu'un objet peut être composant de plusieurs objets différents (c'est le cas d'un paragraphe de texte existant dans deux documents différents, etc). On désire donc associer aux objets composites une sémantique plus large de façon à permettre à un objet de constituer une partie de plusieurs objets à la fois. Les problèmes liés à la représentation de versions multiples d'une manière intégrée au niveau du modèle sont loin d'être résolus aussi. En ce qui concerne les applications spécifiques, il est nécessaire de prendre en considération certains aspects particuliers tels que l'accès concurrent et le travail coopératif. Ces aspects imposent l'utilisation de la notion de vues ou d'environnements de travail.

Dans le cas des applications complexes, les contraintes structurelles et référentielles s'ajoutent aux problèmes d'intégrité et de cohérence de données. L'analyse des différentes approches utilisées pour la gestion de ces aspects dynamiques montre que la notion de déclencheur s'est affinée au cours du temps en s'adaptant au modèle de données. Dans les approches relationnelles la notion est apparue comme une extension au modèle, tandis que dans les approches objet, qui prennent en compte la dynamique des objets, on cherche à intégrer cette notion au modèle. Certaines propositions vont loin dans leur formalisme : les déclencheurs sont des objets comme les autres objets de la base. Les approches objet permettent une évolution de la notion de déclencheurs aussi bien dans l'aspect utilisation que dans l'aspect représentation. Avec la prise en compte des événements temporels et des événements externes, le champ d'application des déclencheurs se généralise. Une description très précise des événements, qui ne sont plus restreints à certaines opérations de mise à jour de données, devient possible. Contrairement aux approches relationnelles, dans les approches objet, la définition d'un déclencheur peut être différenciée de son utilisation. Par ailleurs, une évaluation asynchrone des trois parties d'un déclencheur devient possible, ce qui n'est pas le cas dans les autres approches où l'exécution de l'action est synchronisée avec la production de l'événement.

CHAPITRE 3

LA GESTION DES INFORMATIONS DANS LES ENVIRONNEMENTS GENIE LOGICIEL

1. INTRODUCTION

Dans les deux chapitres précédents nous avons présenté différentes propositions existantes pour gérer les informations dans une application génie logiciel. Dans le chapitre 1, l'accent a été mis sur les recherches faites dans le domaine génie logiciel. Dans le chapitre 2, nous avons abordé les solutions proposées dans le domaine des bases de données pour la gestion d'informations complexes et dynamiques. L'état de l'art fait ressortir les caractéristiques de l'application génie logiciel et la classe parmi les applications non-conventionnelles des bases de données, et révèle de nombreux besoins nouveaux.

Dans ce chapitre, nous exposons, comme conclusion des chapitres précédents, la problématique de la gestion d'informations dans les EGLs et les besoins du domaine d'application, qu'est le génie logiciel. Dans la section suivante nous donnons une classification des besoins à satisfaire selon trois catégories. Les sections 3, 4 et 5, présentent chacune de ces catégories tout en soulignant les problèmes des systèmes existants, soit dans le domaine du génie logiciel, soit, plus généralement, dans le domaine des bases de données. Dans la section 6, nous concluons en présentant l'objectif de notre travail et l'approche choisie pour le réaliser.

2. LES BESOINS POUR LA GESTION D'INFORMATIONS EN GENIE LOGICIEL

L'ensemble des composants logiciels constitue une base importante d'objets. Par le mot *objet* nous faisons référence à une entité du monde réel (des programmes, des documents, des usagers, etc).

Les besoins concernant la gestion de bases d'objets pour les applications non-conventionnelles ont été largement étudiés [EUR 87 , CAI 85, Per 88, Rud 86, Kat 85, Onu 85]. Dans les EGLs ces besoins peuvent être classés selon trois aspects :

- les aspects statiques de la base d'objets : lors de la représentation des objets manipulés, il faut tenir compte de la nature des objets, de leur type, de leur description, de leur structure et des diverses relations qui les associent et ainsi que de la structure des multiples versions existant pour chaque objet.
- les aspects dynamiques de la base d'objets : du fait de l'intégration des outils génie logiciel, du partage des objets entre plusieurs utilisateurs et de la nature évolutive de ces objets, plusieurs besoins s'imposent. Par exemple, il faut pouvoir activer les outils de l'environnement, contrôler les droits d'accès, contrôler les contraintes d'intégrité, propager les effets de bord de modifications, etc.
- L'interface utilisateur pour l'accès à la base d'objets : vue la complexité des informations manipulées, l'utilisateur n'a pas toujours une vision exacte des objets manipulés. Plusieurs modes d'accès doivent être supportées par un tel système. En plus de la recherche classique dans les SGBDs, la recherche par le contenu des informations devient une nécessité. Par exemple, l'utilisateur a besoin de retrouver les portions de documentation spécifiant une certaine fonctionnalité ou la procédure réalisant une certaine action. Un troisième type de recherche nous semble très important dans un noyau contenant un grand volume d'informations complexes et corrélées, il s'agit de la recherche par navigation sur l'ensemble des informations. Les utilisateurs ont besoin souvent de regarder la structure des programmes ou de la documentation, ou de chercher simplement les parties de la documentation liées à une partie de code source qu'ils sont en train de modifier.

Dans la suite nous exposons ces trois catégories de besoins, et nous présentons pour chacun les solutions proposées, soit par les gestionnaires d'objets des EGLs existants, soit par les SGBDs.

2.1. ASPECTS STATIQUES DES INFORMATIONS

La documentation et le code source des logiciels sont des objets complexes, de grande taille, partageables, fortement structurés et interdépendants. Ils se présentent pourtant sous un aspect linéaire (fichier de code, texte de documentation) ce qui rend leur accès difficile. Nous analysons, dans ce paragraphe, ce que devrait être le gestionnaire d'objets d'un EGL du point de vue statique.

2.1.1. Les Différents Aspects Statiques

2.1.1.1. La Granularité d'Objets

En génie logiciel, les objets manipulés sont de granularité variable allant de celle des objets complexes (i.e., documents) à celle des objets représentant les personnes, les droits, etc. Le gestionnaire d'objets doit être donc capable de stocker des objets de taille et de granularités différentes et permettre des types de traitement adaptés aux différentes granularités. En effet, la gestion d'objets complexes de grande taille nécessite d'inclure des techniques d'accès individuel (i.e., par le nom) aux objets remplaçant les modes d'accès ensemblistes.

2.1.1.2. Structure Hiérarchique des Documents, Objets Composites

En génie logiciel, il est souvent nécessaire de manipuler un ensemble d'objets comme un tout. Par exemple, chaque programme est composé de plusieurs modules, et chaque module peut lui même être composé d'une interface et d'une réalisation (comme c'est le cas dans ADA et Modula2). L'interface d'un module est la partie contenant les spécifications de ses ressources (procédures, fonctions, variables). La réalisation est la partie qui réalise les ressources du module, elle est composée donc des réalisations de tous les procédures et des fonctions définies par l'interface. Le niveau de granularité des programmes peut être davantage affiné et aller jusqu'au niveau des fonctions, procédures et variables, et ne pas rester au niveau général des interfaces et des réalisations.

Comme les programmes, les autres documents (documents de spécifications, documents de conceptions, manuels d'utilisation, etc) sont aussi des objets structurés et composites. Les différents composants (tels que chapitres, sections, etc) forment une hiérarchie dont les feuilles sont les portions de contenu du texte. On peut considérer comme critère de décomposition des documents les concepts sémantiques représentés par chaque composant. Ceci permet aux utilisateurs des documents de regarder la structure, pour chercher puis retrouver les chapitres ou les sections satisfaisant leur requête. L'utilisation de tables de matières des manuels d'utilisation et des autres documents montre cette nécessité. Par exemple, chaque document de spécification est constitué, en général, des différents paragraphes donnés dans la figure 1. Cette structure peut être représentée par un arbre dont la racine représente le document de spécification en entier et les autres nœuds de l'arbre représentent les différents chapitres et sections du document.

SPECIFICATIONS FONCTIONNELLES

1 INTRODUCTION (but du document)
2 DESCRIPTION GENERALE DU LOGICIEL.
  2.1 Définition du logiciel.
  2.2 Schéma fonctionnel.
  2.3 Contraintes.
3 DESCRIPTION DES INFORMATIONS.
  3.1 Flot d'informations.
  3.2 Représentation des structures d'information.
  3.3 Description des interfaces.
4 DESCRIPTIONS FONCTIONNELLES.
  4.1 Introduction.
  4.2 Description des fonctions :
  4.3 pour chaque fonction du système:
    • Description des entrées.
    • Description des sorties.
    • Description de fonctionnalité.
    • Restrictions et limitations.
5 DESCRIPTIONS NON-FONCTIONNELLES.
  5-1 Spécification du système.
  5-2 Spécification de l'ordinateur.
  5-3 Spécification des interfaces.
  5-4 Contraintes de conception et critères de performances et de qualité.
6 ANNEXES.
7 BIBLIOGRAPHIE.

Figure-1

Les objets sont donc composites et structurés hiérarchiquement, dans lesquels chaque niveau représente une abstraction du niveau se situant en dessous dans la hiérarchie.

2.1.1.3. Partage des Objets

Il est indispensable dans ce contexte d'envisager le partage des objets. En effet, les portions de contenu peuvent être partagées entre plusieurs documents. Les exemples de partage sont nombreux dans un environnement de génie logiciel : une introduction générale décrivant les concepts du projet peut exister dans le document de spécifications et le document de conception, un paragraphe de texte décrivant les entrées et les sorties d'un module peut exister à la fois dans le document de conception et dans les commentaires du code ou le manuel d'utilisation; le document de spécifications et le manuel d'utilisation peuvent utiliser tous les deux la figure représentant le schéma général du système, plusieurs programmes peuvent utiliser une même portion de code source (procédure, fonction, module), etc. *La notion de partage d'objets représente un aspect important pour la gestion d'informations dans un environnement de génie logiciel.* Il est donc souhaitable de pouvoir partager un paragraphe, une figure ou même un sous-arbre de la hiérarchie des documents entre un nombre quelconque de documents. Sans un tel mécanisme de partage le volume de la base va augmenter considérablement, et la gestion des copies différentes du même composant devient très difficile. De ce fait, la *structure des documents n'est pas exclusivement hiérarchique, un graphe de composition de documents peut donc exister.*

2.1.1.4. Relations entre les Objets

Outre les relations de composition entre les objets, il existe de nombreuses relations décrivant des propriétés importantes entre les différents composants d'un projet (i.e. les relations entre des portions de documents, éventuellement différentes, entre différents documents, entre la documentation et les programmes, entre les documents et les utilisateurs, etc).

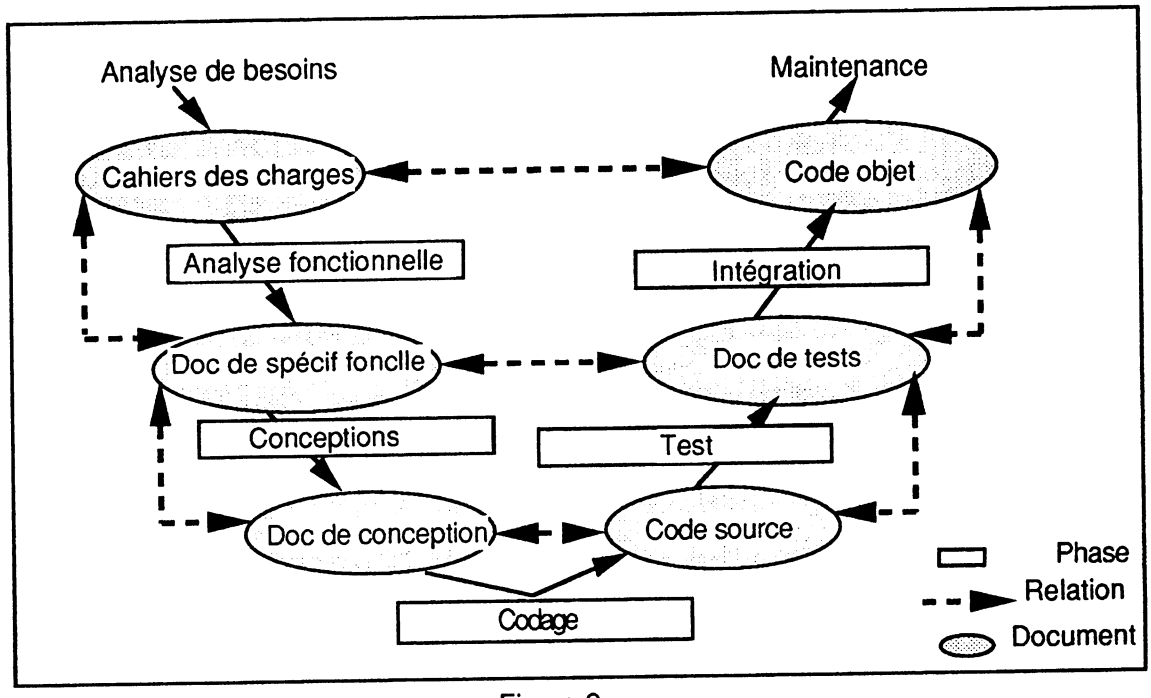


Figure-2

Il faut noter ici que les relations pouvant exister à des niveaux différents de granularité, c'est-à-dire entre les objets composants et non seulement au niveau des objets de plus haut niveau de la hiérarchie de composition. Par exemple, pour une fonctionnalité donnée du logiciel, il existe des liens qui relient la partie du document de spécifications à la partie correspondant dans le document de conception, et aussi, à la partie code qui implante cette fonctionnalité. Un module est lié à un autre par une relation de dépendance, s'il utilise des ressources définies dans celui-ci. La figure 2 montre une partie des relations existantes entre les différents documents. Ces relations qui modélisent des propriétés importantes des objets, ont des sémantiques précises et doivent donc être supportées par le gestionnaire d'objets de l'environnement.

2.1.1.5. Informations Typées

Dans une application telle que le génie logiciel, les objets manipulés ont des types très variés. Le gestionnaire d'objets doit permettre la manipulation d'un nombre important de types. La notion de type permet de gérer la structure des objets

manipulés et leur interaction. Du côté document, il est souhaitable pour des raisons de standardisation et d'efficacité au niveau des méthodes d'accès et de stockage que les différents documents d'un projet (ou de tous les projets) aient une structure semblable. En particulier, les documents doivent être créés suivant des structures prédéfinies correspondant aux normes du projet, et le système doit être capable de vérifier que les informations correspondent à la définition de leur type. Ainsi, plusieurs types d'objets sont définis de façon à pouvoir associer à chaque objet les propriétés qui le décrivent, des informations qui facilitent le contrôle de son état, sa structure de composition et des informations identifiant ses relations avec les autres objets de la base. Une description similaire d'un ensemble d'objets peut déterminer un mode de manipulation et de contrôle semblables.

2.1.1.6. Versions Multiples

Dans un environnement de développement et de maintenance, les logiciels (programmes et documentation) évoluent constamment pour des raisons différentes : développement, simple correction d'erreurs, adaptation à des environnements différents tels que changement de configuration de machine, adaptation aux besoins des clients, etc.

Dans la plupart des cas les modifications effectuées sur un composant logiciel nécessitent la création de nouvelles *versions* de celui-ci. Il est intéressant de conserver les différentes versions de ces composants pour pouvoir retracer l'historique du projet et pour permettre de revenir à une version antérieure de l'objet.

En pratique, deux types d'évolutions ont été identifiés : les révisions qui correspondent à des raffinements successifs et éventuellement à des corrections d'erreurs; les alternatives qui correspondent à des développements parallèles et indépendants des objets.

Du point de vue statique, la gestion des versions multiples pour chaque objet nécessite la définition d'un modèle de versions, c'est-à-dire la définition des relations déterminant la structure de ces versions et leur identification. Nous retrouvons globalement deux structures d'évolution : un arbre de révisions dont les branches sont les alternatives [Fau 91, Kat 90, Tic 88, Dit 85, Tic 85], ou des suites parallèles de révisions dont chacune représente une alternative [Bel 88].

Dans le cadre de la gestion d'objets composites, le concept de versions se généralise : il ne reste plus limité aux versions des objets primitifs, il s'étend pour

s'appliquer aux versions des objets composites, d'où le concept désigné souvent par le terme *configuration* [Tic 88, Buc 85, Ber 84].

2.1.2. La Gestion des Aspects Statiques

Les propositions faites pour supporter les différents aspects statiques des informations peuvent être classifiées selon deux axes de recherches :

Le premier est celui des recherches menées principalement par les spécialistes en génie logiciel, en collaboration dans certains cas avec des chercheurs dans le domaine des bases de données. Il consiste au développement d'un EGL dont le noyau est un gestionnaire d'informations supportant la spécificité des informations génie logiciel.

Le deuxième est celui des chercheurs en bases de données, ayant pour objectif de concevoir un modèle de données capable de décrire et de gérer les objets manipulés dans les applications non-conventionnelles.

L'étude des différentes propositions faites par les spécialistes en génie logiciel (voir chapitre 1) montre que seuls les EGLs à base de SGBDs dédiés commencent à satisfaire les différents besoins. Les gestionnaires d'objets supportant des modèles spécifiques (tels que PCTE+, NOMADE, DAMOKLES, EPOS, etc) commencent à fournir des mécanismes puissants pour gérer les objets composites et les versions d'objets. Toutefois, ces systèmes ne supportent pas de mécanismes puissants pour la manipulation d'objets de granularité variée. Quant à la gestion de versions, ils se limitent souvent à gérer des versions d'objets simples et ne s'intéressent pas à représenter les versions d'objets composites. Dans le cas de solutions convenables, il s'agit de mécanismes supportés par le système et non intégrés au niveau des modèles de données.

Quant aux recherches faites dans le domaine des bases de données (cf chapitre 2), le support des nouveaux besoins pour la représentation d'informations dans les applications non conventionnelles a abouti à l'utilisation de nouveaux modèles de données, dits sémantiques, pour la définition de modèles spécialisés. Dans ce cadre, le modèle Orienté-Objet, dérivé des modèles sémantiques et des langages orientés-objet, a été largement utilisé en raison de sa richesse sémantique pour la représentation des structures complexes et des liens entre les objets de granularités différentes. Parmi les propositions faites dans le domaine nous intéressons plus particulièrement à : ORION et EXODUS. L'intérêt de ces deux approches réside dans les propositions faites au niveau de la définition de la sémantique des objets

complexes et des différents liens entre objets. D'autres propositions ont été centrées sur le modèle de versions qu'ils ont adopté [Fau 91, Kat 90].

Il faut noter cependant que la richesse sémantique de l'approche objet n'est pas forcément offerte par toutes les propositions actuelles. En particulier, la possibilité de partage d'un composant entre plusieurs objets n'est pas toujours prise en considération. Les problèmes liés à la représentation de versions multiples d'une manière intégrée au niveau du modèle, et surtout en ce qui concerne la généralisation de la notion d'objet composite pour tenir en compte des versions d'objets composites et de configurations est loin d'être résolue.

2.2. ASPECTS DYNAMIQUES DES INFORMATIONS

Le développement et la maintenance des logiciels mettent en jeu une base de grand volume d'informations complexes et partageables. Cette base est caractérisée par ses aspects dynamiques et évolutifs demandant une gestion efficace de la sémantique liée aux informations et aux situations critiques. Nous détaillons par la suite certains de ces aspects dynamiques qui nécessitent une assistance permettant de contrôler la cohérence de la base.

2.2.1. Les Différents Aspects Dynamiques

2.2.1.1. Comportement des Objets et Activation d'Outils Génie Logiciel

Gérer les documents et les logiciels dans un EGL revient à définir un ensemble d'opérations de manipulation : création et mise-à-jour. En plus de ces opérations de manipulation il est nécessaire de gérer l'activation des outils génie logiciel (éditeurs de texte, compilateurs, éditeurs de lien, etc) à partir de la base d'objets dans le but d'intégrer ceux-ci dans l'environnement. L'intégration des outils est un aspect essentiel ayant pour vocation de supporter le processus de développement du logiciel. L'activation des outils génie logiciel se fait en définissant des opérations primitives (compiler, éditer, etc) faisant les liens entre la base d'objets et ces outils.

L'activation des outils génie logiciel agit sur les objets de la base et peut avoir comme effet la création automatique (ou manuelle) de certains objets dérivés à partir des objets sources existants. Les objets dérivés sont, par exemple, le code objet et le code exécutable associés au code source, les termes d'indexation décrivant le contenu sémantique des documents, etc.

2.2.1.2. Droits d'Accès

Les informations manipulées lors du développement et de la maintenance de logiciels sont partagées entre plusieurs usagers, il faut donc pouvoir les protéger. Les usagers n'ont pas les mêmes droits sur tous les objets. Ainsi, lors de l'accès à un objet, il faut vérifier que l'usager a le droit d'effectuer l'opération en cours sur cet objet. Une protection minimum peut être dérivée du système d'exploitation. Certains systèmes gèrent des listes d'accès associées à chaque objet pour spécifier les droits possibles pour chaque usager. D'autres systèmes associent à chaque usager une liste de privilèges spécifiant l'ensemble des objets accessibles ainsi que les opérations permises.

2.2.1.3. Contrôle de Versions et de Configurations

Nous avons vu que le développement et la maintenance nécessitent la gestion de versions multiples des logiciels. Le contrôle de la création et de la manipulation des versions multiples est un aspect important dans un EGL. Un des problèmes importants ici, est d'assurer la cohérence entre les différentes versions des composants d'un projet, c'est-à-dire déterminer les effets de la création d'une version d'un objet sur les autres objets qui en dépendent. Il faut aussi contrôler la création de versions (accès concurrents, fusion de versions, nombre de versions, cohérence de point de vue sémantique, etc). Ces fonctionnalités sont souvent le rôle des gestionnaires de versions incorporés dans les EGLs.

Dans le cas de la gestion des objets composites le problème se généralise : il n'est plus suffisant de contrôler les versions des objets primitifs, il faut aller plus loin pour contrôler les versions des objets composites souvent appelés *configurations*. Par exemple, un programme de taille importante peut être constitué de plusieurs centaines de modules. Chaque module peut évoluer séparément en plusieurs versions selon des critères prédéterminés (langages différents, algorithmes de développement différents, etc). Une configuration du programme est un choix cohérent d'une liste de versions, une pour chaque module composant le logiciel. Or, le choix de cette liste par l'usager est une opération complexe souvent génératrice d'erreurs. C'est pourquoi il est nécessaire de fournir un mécanisme permettant la création automatique de ces configurations. Des gestionnaires de configurations assez performants sont spécifiés par les environnements existants, la plupart offrent un support basé sur l'outil Make d'Unix.

Le même problème se pose aussi pour la documentation existant en plusieurs versions car chaque document est composé de plusieurs chapitres, chaque chapitre de plusieurs paragraphes. On constate que les gestionnaires de configurations ne

sont pas assez généraux pour prendre en compte la documentation, et dans le cas où elle est considérée, on propose des mécanismes différents et ad-hoc.

2.2.1.4. Contraintes d'Intégrité

La manipulation d'une base importante d'informations pose un problème de gestion de cohérence. Les contraintes d'intégrité permettent d'exprimer les états valides des informations stockées dans la base et leur sémantique. Dans l'état actuel de la plupart des environnements, ces contraintes sont figées et définies d'une façon implicite dans les outils qui font la vérification de l'intégrité des informations ou dans le code des commandes. Des approches alternatives utilisant les techniques explorées par les bases de données tendent à rendre ces contraintes plus explicites. Elles définissent des règles qui expriment la sémantique attachée aux objets et aux relations et les actions à entreprendre dans les cas de violation de contraintes. Cependant les solutions proposées sont souvent limitées aux problèmes de l'intégrité du code source et de la recompilation automatique.

2.2.1.5. Effets de Modification

Durant le développement d'un logiciel et sa maintenance, les modifications d'un objet peuvent effectuer des changements "remarquables" dans son état qui rendent invalides un ou plusieurs autres objets. Par exemple, des modifications dans le code source peuvent influencer les documents de conception, mais pas les spécifications et les plans de tests, cependant, l'évolution de code source pour satisfaire de nouveaux besoins des clients peut avoir comme conséquence la modification des spécifications. Les évolutions des objets ainsi que leurs effets doivent être analysés pour informer l'utilisateur des objets (portions de texte, procédures, etc) touchés par cette évolution. Dans certains cas des actions (recompilation, réindexation) sont à exécuter et des outils de l'environnement doivent être activés automatiquement pour assurer la cohérence de la base.

Le but d'un gestionnaire d'objets dans un EGL est donc d'assurer la cohérence de la base après des modifications effectuées sur certains objets, c'est-à-dire propager les effets de modifications, ou, tout au moins, pouvoir avertir l'utilisateur de l'incohérence ou de l'incomplétude lors de l'accès à un objet, donc maintenir son état.

La plupart des environnements classiques qui détectent les effets de modifications utilisent implicitement la relation de dépendance entre les modules pour effectuer une recompilation automatique des modules dépendant d'autres modules ou interfaces modifiés. Les mécanismes d'analyse des effets des changements sont plus

généraux et prennent en compte d'autres relations sémantiques entre les objets. Ils utilisent soit des contraintes explicites de gestion de logiciels soit des mécanismes de déclenchement de type événement-action pour propager les effets de bord de modifications.

2.2.2. La Gestion des Aspects Dynamiques

Les recherches menées à ce sujet peuvent être également différenciées suivant deux axes de recherches : les recherches effectuées dans le domaine génie logiciel dont le but est de rendre les gestionnaires d'objets des EGLs plus dynamiques, et les recherches menées dans un cadre des SGBDs actifs, qui commencent à proposer des solutions plus homogènes et plus intégrales au problème.

La tendance actuelle des environnements de développement et de maintenance en génie logiciel est d'introduire une plus grande dynamicité en modélisant les processus de création, de développement et d'évolution des logiciels. Ceci correspond à la volonté d'avoir un mécanisme puissant qui permet de gérer efficacement l'évolution et la sémantique liée aux informations de la base d'une manière uniforme. Cette évolution tend à introduire la connaissance nécessaire sous forme de règles modélisant les différentes activités en génie logiciel et l'évolution de la base.

On distingue essentiellement deux approches différentes de gestionnaires de règles : une approche déductive (MARVEL) intégrant des techniques utilisées en Intelligence Artificielle, et une approche inspirée des gestionnaires d'exceptions des langages de programmation ou des déclencheurs supportés par les SGBDs dits actifs (NOMADE, ELF). Nous trouvons qu'une approche déductive n'est pas adaptable à notre contexte. Vus le grand volume d'informations manipulées et la nature d'information, il n'est pas possible de déclencher le système de règles à chaque modification effectuée dans la base. En effet, cette approche pose des problèmes au niveau de l'efficacité et la performance du système.

L'étude des différentes propositions pour la deuxième approche montre qu'elles sont souvent pragmatiques et consistent à définir une couche supplémentaire au noyau de la gestion d'informations. Le mécanisme qui gère la dynamique n'est pas intégré au niveau du modèle de données utilisé, et pose ainsi un problème d'homogénéité d'expression et d'efficacité. De plus, ce mécanisme n'est pas utilisé uniformément dans les EGLs pour résoudre tous les problèmes liés à la dynamique. Ceci est souvent dû au manque en puissance d'expression et de généralité. Dans NOMADE par exemple, un gestionnaire de configurations séparé du mécanisme général est défini pour contrôler la création et la manipulation de configurations.

Les recherches dans le domaine des SGBDs proposent un mécanisme général pour gérer uniformément les différents aspects dynamiques et tendent à l'intégrer au niveau du modèle de données. L'arrivée des modèles Orientés-Objets représente un terrain d'implantation privilégié de cette généralisation et intégration.

Néanmoins les différentes propositions ne tiennent pas en compte de la particularité de l'application génie logiciel : les opérations sont longues et coûteuses. Il n'est pas possible de *jeter* un ensemble d'opérations lorsqu'elles conduisent à une incohérence, il faut distinguer ici des états d'incohérence tolérée des états d'incohérence non-tolérée. En effet, comme nous allons voir par la suite, l'utilisation de la notion de transactions imbriquées constitue une approche satisfaisante à ce niveau.

2.3. ACCES AUX INFORMATIONS ET INTERFACE UTILISATEUR

Vu le grand volume d'informations manipulées et leur complexité, l'accès aux informations à travers des requêtes basées sur la structure des objets et leur description externe, comme c'est le cas dans les SGBDs, n'est pas suffisant et même n'est pas toujours possible. En effet, une recherche par le contenu des documents est exigée dans ces cas. La recherche par le contenu tient compte de la sémantique représentée par les documents et non seulement de leur description. Pour pouvoir interroger les documents par le contenu, ceux-ci doivent être représentés dans un formalisme qui est censé exprimer leur sémantique. Cet aspect représente la fonctionnalité principale des Systèmes de Recherche d'Informations (SRI) appelée indexation.

Un autre type de recherche nous semble très important, il s'agit de la recherche par navigation sur l'ensemble des informations. La programmation et la documentation sont des activités interactives, elles consistent à éditer, modifier et à interroger. Les utilisateurs ont besoin souvent d'accéder directement aux parties de documentation ou de code source. Il faut disposer donc d'une interface flexible et facile à utiliser, qui permette, entre autres, l'interrogation de la base graphiquement par la navigation sur les objets. Ce type de recherche simple et efficace est la fonctionnalité principale des Systèmes Hypertexte (SH).

Nous avons remarqué qu'aucun des noyaux de gestion d'informations des EGLs n'intègre de telles fonctionnalités. L'accès aux informations se fait suivant l'approche utilisée : dans le cas des systèmes disposant d'une base hypertexte, l'accès se fait par navigation sur l'ensemble des objets; dans le cas des systèmes disposant d'un SGBD, l'accès aux informations se fait d'une manière déterministe par des

requêtes basées sur la structure de données, et l'information est extraite selon des critères de correspondance exacte entre la requête et la réponse.

Dans les deux paragraphes suivants, nous présentons plus précisément les deux fonctionnalités à introduire dans les EGLs : l'accès aux informations par le contenu via l'utilisation des fonctionnalités d'un SRI, et l'accès direct par navigation via l'introduction des fonctionnalités des SHs. Et nous soulignons leur problématique pour une application aussi complexe : le génie logiciel.

2.3.1. L'Accès par le Contenu, Systèmes de Recherche d'Informations

Pour accéder aux informations dans un EGL il faut tenir compte de la nature de ces informations. Les EGLs manipulent un volume important d'informations ayant un contenu textuel et une corrélation importante. Nous avons vu que la gestion de ces informations exige l'introduction des fonctionnalités des SGBDs qui ne sont pas supportées par les SRIs classiques. Or, puisque l'utilisateur n'a pas toujours connaissance de la structure des informations manipulées dans la base, l'accès à travers des requêtes basées sur cette structure n'est pas toujours possible. Une recherche par le contenu sémantique des documents est exigée. Par exemple, il est nécessaire de pouvoir formuler des requêtes sur la fonctionnalité que remplit un composant logiciel, ou de localiser une partie du code défaillant à partir de la description de son comportement. Une combinaison de ce type de recherche supporté par les SRIs avec la recherche classique aux SGBDs nous semble exigée dans un EGL qui supporte les activités de maintenance.

Pour pouvoir enrichir un SGBD par les fonctionnalités des SRIs nous avons besoin d'élaborer plusieurs catégories de problèmes qui se posent :

- le modèle de données utilisé doit être capable de représenter non seulement le contenu textuel (voire multimédia) des documents, mais aussi, de représenter leur contenu sémantique. Le contenu sémantique des documents étant une forme réduite contenant les éléments jugés à priori les plus caractéristiques du contenu des documents. Il peut avoir une structure plus ou moins complexe : des simples mots-clés, des groupes nominaux, des graphes conceptuels [Che 91], etc.
- le modèle de données doit permettre d'élaborer les mécanismes nécessaires à la gestion du contenu sémantique des documents et des liens existence entre le contenu propre aux documents et leur contenu sémantique. Par exemple, il faut pouvoir supporter un processus permettant, à partir du contenu

propre au document, de retrouver son contenu sémantique. Ce processus est appelé classiquement l'*indexation*. Suivant l'application considérée, l'indexation des documents peut être faite manuellement ou automatiquement. Les liens existants entre le contenu des documents et leur contenu sémantique sont appelés *relations d'indexation*, ils modélisent le processus utilisé pour l'*indexation*.

- dans le contexte génie logiciel, les problèmes de la recherche par le contenu sémantique des documents sont essentiellement dus à l'évolution du corpus : les programmes, ainsi que leur documentation, évoluent constamment. Dans la plupart des SRIs classiques, du fait de l'étude de corpus stable, les relations d'indexation sont établies une fois pour toutes. En effet, étant sensible à l'évolution du corpus, l'indexation du contenu des documents par certains termes peut être remise en cause. Il se peut que certains concepts deviennent peu importants, voire éliminés, dans une nouvelle version d'un document ce qui nécessite une réévaluation de la relation d'indexation entre le document et la représentation sémantique correspondante.

  Par conséquent, pour effectuer la recherche par le contenu en génie logiciel, l'on doit assurer la cohérence entre les versions différentes d'un document et leur représentation sémantique, c'est-à-dire, entre les versions différentes d'un document et les relations d'indexation associées. Cet aspect s'ajoute en effet aux aspects dynamiques des informations en génie logiciel, et nécessite ainsi un mécanisme puissant capable de couvrir ces besoins.

L'étude des différentes propositions des gestionnaires d'objets pour les EGLs et des SGBDs montre qu'aucune ne supporte les différents aspects mentionnés ci-dessus.

2.3.2. L'Accès Direct et la Recherche par Navigation, Système Hypertexte

Comme nous l'avons vu dans le chapitre 1, les SHs ont trouvé leur application dans plusieurs domaines parmi lesquels le génie logiciel. Les SHs servent pour gérer des réseaux de documents en réunissant deux aspects : le premier consiste en la représentation et la manipulation des informations, le deuxième est un aspect communication avec l'extérieur d'une manière directe et interactive avec l'utilisateur.

En conclusion, les hypertextes peuvent constituer un support adéquat pour les gestionnaires d'objets des EGLs. En particulier, ils procurent des mécanismes et des outils pour représenter des informations complexes (texte, graphique, code source,

etc), les corrélations existant entre les différents documents et entre les documents et les programmes, pour la gestion et la création de versions et de configurations, et pour l'accès direct aux informations.

Cependant, les SHs existants concentrent l'intérêt aux aspects interface utilisateur et à la représentation des liens entre les portions arbitraires d'informations. Bien que ce soient des aspects très importants, ils ne sont pas suffisants au niveau modélisation de données des EGLs.

La faiblesse majeure que présentent ces systèmes est le manque de richesse au niveau de la représentation de la sémantique de données liée aux différentes abstractions nécessaires et aux aspects dynamiques des informations. La problématique des SHs pour le génie logiciel se résume par les points suivants :

- bien que certains SHs explorent la notion de composition, cette notion a besoin de s'enrichir sémantiquement pour pouvoir manipuler un ensemble de nœuds et de liens comme une unité abstraite et non seulement une collection arbitraire de nœuds. Ce mécanisme revient à prendre en compte la différence sémantique entre les liens de référence et les liens de composition. Contrairement aux liens de référence qui sont des liens descriptifs, l'existence d'un lien de composition implique que les opérations (création, suppression) appliquées sur le nœud abstrait d'origine ont des conséquences sur ses composants aussi.

- un autre domaine qui n'a pas été suffisamment exploré par les SHs est la manipulation d'information typée. Dans une application telle que le génie logiciel, il est souhaitable pour des raisons de standardisation et d'efficacité au niveau des méthodes d'accès et de stockage que les différents documents d'un projet aient une structure similaire, c'est-à-dire que les documents doivent être créés suivant des structures prédéfinies et que le système soit capable d'assurer l'intégrité des informations avec la définition de leur type. L'utilisation des *formes* dans DIF est une étape vers la définition d'information typée, elle peut être généralisée pour définir un mécanisme de typage dans le sens des langages de programmation.

- Un des grands principes des systèmes hypertexte est que les informations sont continuellement modifiées. La cohérence doit naturellement être conservée : détruire un nœud entraîne la destruction de tous les liens qui y sont attachés, déplacer une partie du contenu d'un nœud à laquelle est attaché un lien peut entraîner comme conséquence le déplacement du lien. Dans une application génie logiciel le dynamisme du système ne doit pas s'arrêter là.

Par exemple, une action exécutée sur un nœud peut avoir des effets sur l'intégrité des informations contenues dans le nœud ou sur d'autres nœuds liés de l'hyperdocument. Le système doit être actif, c'est-à-dire, capable de réagir et d'exécuter automatiquement certaines actions dans le but de maintenir la cohérence et l'intégrité de l'hypertexte, aspect non supporté par les SHs existants.

- l'accès par navigation sur les informations de l'hypertexte est très convenable dans une application génie logiciel, mais il est problématique car le risque de se perdre dans un tel réseau volumineux d'informations augmente considérablement. La solution dans ce cas est d'élargir le mécanisme d'interrogation de la base pour intégrer la recherche par la structure supportée par les SGBDs et la recherche par le contenu des informations classique aux SRIs.

Il est clair que la faiblesse des SHs pour supporter ces différents aspects est due essentiellement à la simplicité du modèle de base de ces systèmes : graphe orienté. Le support de ces aspects nécessite la définition d'un modèle plus formel pour la description d'informations dans la base de l'hypertexte pour l'adapter aux applications génie logiciel. Ce modèle doit être capable de supporter les différentes abstractions nécessaires et d'introduire des techniques de SGBDs et de SRIs. Certaines propositions ont été faites dans ces sens [Bru 90, Fur 90, Hal 90, Lan 90, Sav 91].

3. CONCLUSION GENERALE, NOTRE APPROCHE

Les SGBDs présentent des solutions plus convenables et plus générales pour supporter ces différents aspects. L'utilisation des modèles sémantiques et orientés-objet a montré des résultats prometteurs pour tenir compte de la complexité des nouvelles applications. Il faut noter cependant que la richesse en sémantique de l'approche objet n'est pas forcément offerte ou n'est pas toujours exploitée au mieux par les propositions actuelles. Si l'approche objet permet le couplage statique-dynamique, l'expression de la dynamique est souvent noyée dans le code des méthodes d'une façon procédurale. Le peu de propositions faites pour exprimer la dynamique d'une manière déclarative manquent d'une capacité d'expression et de clarté.

L'intégration de fonctions d'accès par le contenu sémantique des documents et d'accès direct par la navigation est nécessaire dans ces cas. L'introduction de ces

deux types de recherches consiste à introduire les fonctionnalités principales des SRIs et des SHs aux gestionnaires d'objets des EGLs.

En conclusion, nous pensons que la combinaison des fonctionnalités des SGBDs, des SRIs et des SHs est une approche très intéressante pour résoudre les problèmes dus à la particularité des gestionnaires d'objets des EGLs. Vu la complexité d'une telle combinaison, le travail effectué dans cette thèse n'a pas pour but de réaliser un système intégrant toutes ces fonctionnalités. Notre objectif est de concevoir un modèle de données qui tienne compte de cette combinaison.

Il s'agit de définir un modèle de base pour les systèmes hypertexte permettant d'intégrer les aspects suivants :

1) la représentation et l'expression de la sémantique des objets multimédia, structurés et des liens sémantiques entre eux, y compris les liens de composition.

2) la représentation de versions multiples d'objets, y compris les objets composites.

3) le support des éléments de base pour intégrer les aspects interface et l'accès direct aux informations.

4) l'intégration des éléments de base permettant d'effectuer la recherche par le contenu sémantique des documents, tels que la représentation du contenu sémantique des documents et la gestion du processus d'indexation. Il faut noter ici que l'étude du processus même de l'indexation sort du cadre de cette thèse, et que nous utilisons un processus (manuel ou automatique) existant.

5) la représentation et la gestion de la dynamique des informations pour supporter l'intégrité des données, leur cohérence, la cohérence des liens entre les documents et leur représentation sémantique et la propagation des effets de modification.

Notre approche a bénéficié de la richesse des modèles de données sémantiques pour la représentation des mécanismes d'abstraction, et notamment pour la représentation de la structure complexe et de la sémantique des liens entre les données. Elle a bénéficié également du couplage entre les données et les traitements des approches objet pour la représentation de la dynamique des informations.

Plus précisément, pour résoudre les deux premiers objectifs cités ci-dessus, nous avons procédé à une synthèse des approches entreprises par les propositions

existantes, notre objectif étant d'introduire les notions nécessaires pour satisfaire les différents besoins.

Quant au troisième et quatrième objectifs, des aspects qui n'ont pas été abordés par les systèmes existants, nous avons augmenté la représentation des documents par une abstraction modélisant en plus du contenu textuel, leur contenu sémantique et leur présentation visuelle.

Concernant les aspects dynamiques, notre idée est d'utiliser les méthodes, modélisant le comportement des objets, pour exprimer d'une façon déclarative les contrôles nécessaires au maintien de l'intégrité et de la cohérence. Les méthodes expriment, indépendamment du langage de programmation, les conditions nécessaires à leur déclenchement et les contrôles à entreprendre suite à leur exécution. Elles ne sont pas seulement déclenchées suite à l'envoi explicite de messages aux objets concernés, mais aussi d'une manière implicite suite à la détection de certaines situations dans la base.

Pour réaliser notre objectif, nous nous sommes fixés les critères suivants:

- le modèle doit tenir compte des caractéristiques spécifiques des informations mises en jeu dans les EGLs; cependant, il doit être générique et non pas spécifique à une méthode ou stratégie de programmation précise.
- les différents aspects statiques et dynamiques doivent être intégrés d'une manière homogène au niveau du modèle de données et non pas supportés artificiellement par le système.
- la dynamique doit être supportée par un mécanisme ayant une puissance d'expression permettant d'exprimer tous les aspects dynamiques des données d'une manière homogène et uniforme. Le système doit garder une vision globale de la sémantique liée à la cohérence des données et des traitements.
- l'expression de l'intégrité et de la cohérence doit se faire d'une manière déclarative, simple et claire. Le contrôle des données et des traitements ne doit plus être noyé dans le code des méthodes d'une manière procédurale. Le langage d'expression de la cohérence doit être simple et accessible par l'utilisateur.
- le contrôle de la cohérence ne doit pas être d'un coût prohibitif de manière à mettre en cause l'efficacité du système et à pénaliser ainsi l'utilisateur.

Les deux chapitres suivants décrivent le modèle de données proposé. Les différentes notions liées aux aspects statiques du modèle sont définies dans le chapitre 4. Ensuite, le chapitre 5 précise les aspects dynamiques du modèle.

CHAPITRE 4

MODELE DE DONNEES, ASPECTS STATIQUES

1. INTRODUCTION

Dans le chapitre précédent, nous avons donné les grandes lignes de notre travail : il s'agit de définir un système hypertexte pour la gestion d'informations dans les environnements de développement et de maintenance en génie logiciel. Nous avons vu dans le chapitre 2 que les systèmes hypertexte souffrent d'une insuffisance au niveau des mécanismes d'abstraction qu'ils supportent. Ils sont passifs dans le sens qu'ils ne participent pas au processus de développement et de maintenance des logiciels et qu'ils ne sont pas capables de réagir automatiquement dans certaines situations pour le maintien de l'intégrité et de l'incohérence de données. Cette insuffisance est due essentiellement à la simplicité du modèle de base de ces systèmes : Graphe Orienté.

Dans ce chapitre et le chapitre suivant, nous présentons un modèle de données sémantique orienté-objet qui intègre les deux aspects statique et dynamique nécessaires à la modélisation des informations mises en jeu dans les EGLs. Le modèle s'intéresse à la représentation des différentes abstractions nécessaires pour la gestion d'objets complexes, composites, typés, ayant des liens sémantiques entre eux et existant en versions multiples.

Bien que le domaine d'application que nous avons considéré pour valider nos propositions soit celui du génie logiciel, il est important de noter que le modèle proposé ici pourrait résoudre les problèmes de gestion d'objets complexes dans un cadre plus général que celui du génie logiciel. Par exemple, nos propositions pourraient s'intégrer dans un environnement de type CAO. Cependant, notre souci a toujours été de présenter les problèmes dans le contexte du génie logiciel, ce qui était le point de départ de notre étude.

Dans le modèle proposé, une attention particulière est portée sur la modélisation des documents dans l'objectif de supporter au niveau du modèle, d'une part, les aspects hypertexte, et d'autre part, les outils de base pour pouvoir effectuer la recherche documentaire par le contenu sémantique.

La représentation et la gestion de la dynamique des données sont des aspects très importants supportés également par le modèle. Pour cela, nous bénéficions du couplage entre les données et les traitements offert par les modèles orientés-objet.

Nous n'avions pas pour objectif de proposer un nouveau modèle de données. Nous avons essayé au mieux d'utiliser des notions que nous jugeons utiles existantes dans d'autres modèles [Kim 89, Fau 91, Car 88, etc]. Celles-ci étant insuffisantes dans notre contexte, nous avons augmenté l'existant par un raffinement de certaines définitions (objets composites, configurations, etc) ou par des notions propres à notre proposition (document, projet, déclencheur, etc).

Ce chapitre est consacré à la description des aspects statiques du modèle. Dans le paragraphe 2, nous présentons les notions de base du modèle, telles que: valeur, objet composite, type. Dans les paragraphes 3 et 4 nous présentons les notions de *document* et de *projet* définies pour supporter la gestion de documents dans un environnement génie logiciel. Le paragraphe 5 est une présentation du modèle de versions et de sa généralisation aux objets composites. Nous donnons pour chaque concept proposé une définition informelle puis une définition formelle.

2. MODELE DE DONNEES MINIMAL

Les objets en génie logiciel (programmes, documentation) peuvent faire **référence** à d'autres objets. Les références entre les objets peuvent représenter des contraintes précises comme celles existant entre un objet et ses **composants.**

Par exemple, le module WindManips (montré dans la figure 1) qui implante un gestionnaire de fenêtres, est composé de plusieurs fonctions : OpenWind, CloseWind, etc. De plus, il fait référence au document de conception ConcWindManips (et éventuellement à d'autres documents). Les différents composants sont des objets dépendant du module de point de vue existence. Les autres objets référés tels que ConcWindManips sont considérés comme des objets indépendants ayant des liens simples avec ce module puisqu'ils ont besoin d'être en évolution constante avec lui.

De plus, un objet peut entrer dans la composition de plusieurs autres objets dits *ses parents*, son existence est donc dépendante de celle de ses parents.

Les objets peuvent avoir des **propriétés** ne correspondant pas à des entités ayant une existence propre dans le monde réel. Les propriétés sont attachées aux objets pour décrire leurs caractéristiques. Par exemple, la date de création (05.05.89) et le nom du programmeur (Dupont) sont des propriétés attachées au module WindManips (voir Figure 1).

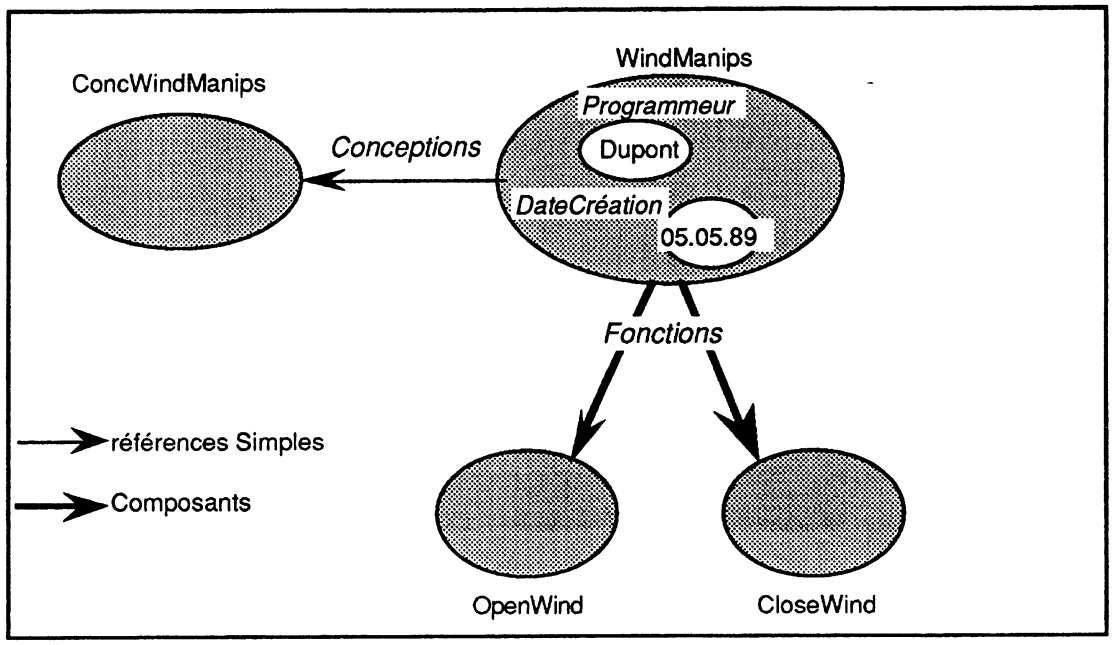


Figure-1

En génie logiciel, les objets doivent être créés suivant des **structures** correspondant à des normes prédéfinies. Cette structure décrit un ensemble d'objets en précisant leurs propriétés, leurs composants, et les références qu'ils ont avec les autres objets. Par exemple, il est souhaitable que les documents aient une structure correspondant à un certain standard, et que les programmes soient connectés aux documents d'une manière prédéfinie par l'administrateur du projet en développement. Le système doit être capable de vérifier que les objets créés correspondent bien à la structure prédéfinie.

Ces différents aspects sont plus ou moins supportés par des modèles de données existants qui traitent des objets complexes, structurés et typés [Kim 89, Car 88, Lec 87]. Le modèle de base que nous considérons intègre des concepts proposés par ces modèles et jugés utiles dans notre contexte.

Contrairement aux modèles où "tout est objet", notre proposition (comme celle d'EXODUS et O2) différencie deux sortes d'entités conceptuelles, les objets et les valeurs. Les *objets* sont les entités conceptuelles du modèle correspondant aux entités du monde réel. Les *valeurs* représentent des propriétés attachées aux objets, et ne correspondant pas à des entités ayant une existence propre dans le monde réel. Pour la représentation des sémantiques des liens existant entre les objets nous utilisons les notions d'attributs de composition et d'attribut de référence. Cela nous amène au concept d'objet composite défini originellement dans ORION [Kim 89]. Pour définir une structure et un comportement communs à un ensemble d'objets, la notion de *type* est supportée aussi par le modèle.

Notre objectif dans ce paragraphe est de donner des définitions et un formalisme à ces différentes notions comme nous l'utilisons dans notre étude.

2.1. NOTATIONS UTILISEES POUR LE FORMALISME

Pour donner des définitions formelles aux différents concepts du modèle, nous considérons les ensembles suivants:

- un ensemble fini de domaines $\mathfrak{D}_1,\ldots, \mathfrak{D}_n$, $n\geq 1$, dont l'union est désigné par $\mathfrak{D}$. Chaque domaine désigne un ensemble de valeurs (i.e., l'ensemble de tous les entiers est un domaine).
- un ensemble infini $\mathfrak{I}$ de symboles dont les éléments sont utilisés pour identifier d'une façon unique les objets et dont l'intersection avec $\mathfrak{D}$ est vide.
- un ensemble infini $\mathfrak{N}$ de symboles dont les éléments sont utilisés pour nommer les objets.
- un ensemble infini $\mathfrak{A}$ de symboles désignant les noms des attributs. Intuitivement, les attributs sont les éléments utilisés pour la construction des objets (comme nous allons voir dans la suite).

2.2. VALEURS

Notre proposition différencie donc deux sortes d'entités conceptuelles, les objets et les valeurs. Les **valeurs** représentent des propriétés attachées aux objets, elles ne correspondent pas à des entités ayant une existence propre dans le monde réel.

Par exemple, la date de rédaction d'un document; le nom de son auteur; le nom du programmeur d'un module, etc, sont des **valeurs** associées aux objets et ne pouvant pas être partagées par plusieurs objets.

On distingue deux catégories de valeurs : celles dites **valeurs de base**, ce sont des valeurs prédéfinies (les valeurs entières, les valeurs booléennes, portions de texte, etc), et celles dites **construites**, elles sont définies d'une façon récursive à partir des valeurs de base en utilisant les constructeurs *N-uplet* et *Seq*. Les constructeurs sont *des opérateurs de structuration de données*, le *N-uplet* est utilisé habituellement par les modèles de données pour construire des agrégations d'attributs. Le constructeur *Seq* est utilisé pour former *des listes ordonnées de longueur variable de valeurs*. Nous désignons dans la suite l'ensemble des listes ordonnées d'un ensemble homogène de valeurs $\mathfrak{X}$ par $Seq(\mathfrak{X})$.

Soit $\mathfrak{V}$ l'ensemble de toutes les valeurs. Cet ensemble est défini récursivement de la manière suivante :

- le symbole *nil* est une valeur dite *valeur de base*, nil$\in \mathcal{V}$.
- chaque élément d de $\mathcal{D}$ est une valeur dite *valeur de base*, d$\in \mathcal{V}$.
- chaque nuplet d'attributs de la forme $<a_1:v_1,\ldots, a_n:v_n>$ tels que n>=1 et $a_i \neq a_j \in \mathcal{A}$ et $v_i \in \mathcal{V} \cup Seq(\mathcal{V})$ est une *valeur construite* $\in \mathcal{V}$.

2.3. OBJETS

Les **objets** sont les entités conceptuelles du modèle correspondant aux entités du monde réel. Ainsi, un document de spécification, un programme, un module, etc, sont modélisés sous forme d'objets. Chaque objet est créé suivant une structure prédéterminée par son **type**; le type d'un objet définit aussi la manière dont cet objet peut se comporter (la définition des types est donnée dans le paragraphe 2.5).

Par opposition aux **valeurs**, les **objets** sont identifiés. Chaque objet est identifié par deux sortes d'identificateurs permettant l'accès à l'objet : un **identificateur externe** et un **identificateur interne**. Un identificateur externe est un **nom** donné à l'objet par l'utilisateur. Le nom de l'objet l'identifie par rapport à tous les objets du même type et des sous-types du treillis des types (voir §2.5.5). Les **noms** sont le moyen dont disposent les utilisateurs pour accéder aux objets sans ambiguïté. L'identificateur interne est unique et est inaccessible à l'utilisateur.

Nous pouvons différencier deux catégories d'objets: les objets **atomiques** et les objets **complexes**. Un objet atomique est un objet n'ayant pas de référence à d'autres objets dans la base. Il représente en effet, une valeur à laquelle est associé un identificateur externe et un identificateur interne. Un objet complexe est un objet construit récursivement à partir des **objets atomiques** et des **valeurs** en utilisant les constructeurs *Nuplet* et *Seq*.

Pour donner une définition formelle au concept d'objet, nous considérons l'ensemble $\mathcal{O}$ formé de tous les objets de la base. Chaque objet O$\in \mathcal{O}$ est défini par un triplet (I, N, E) tels que : I$\in \mathcal{I}$ est un identificateur interne, N$\in \mathcal{N}$ est le nom de l'objet, E est, soit une valeur de base, soit, un nuplet d'attributs de la forme $<a_1:e_1, a_2:e2, \ldots, a_n:e_n>$ tels que $a_i \neq a_j \in \mathcal{A}$ et $e_i \in \mathcal{V} \cup Seq(\mathcal{V}) \cup \mathcal{I} \cup$ Seq($\mathcal{I}$).

Cette définition montre bien que l'élément E du triplet n'est pas forcément une valeur, il peut être un nuplet d'attributs qui peuvent désigner des identificateurs ou des séquences d'identificateurs d'objets. Souvent cet élément est appelé la *valeur* de l'objet, ce qui peut ici entraîner une confusion avec la notion de **valeur** définie dans le paragraphe précédent. Nous allons désigner cet élément par l'**état** de l'objet. En effet, l'ensemble $\mathcal{V}$ des **valeurs** représente un sous ensemble des **états** possibles des objets.

Les **objets atomiques** sont représentés par des triplets (I, N, E) tels que E∈ $\mathcal{V}$ est une valeur. Tandis que les **objets complexes** sont ceux tels que E est un nuplet d'attributs dont l'un au moins désigne un identificateur ou une séquence d'identificateurs d'objets, c'est-à-dire:

$E=<a_1:e_1,..., a_n:e_n>$ tel que $\exists i \in \{1, ..., n\}, e_i \in \mathcal{I} \cup Seq(\mathcal{I})$.

Pour retrouver l'état d'un objet O=(I, N, E) nous considérons l'opérateur *OState* :

OState(O)=E

2.4. ATTRIBUTS

Comme nous venons de le voir, les attributs sont les éléments de base qui servent pour la construction des valeurs et des objets. Ils servent comme éléments de base pour la construction des valeurs construites. Dans le cas des objets, ils modélisent les caractéristiques des objets en représentant leurs **propriétés**, leurs **composants** et les autres objets de la base auxquels ils font **référence**.

Nous considérons l'opérateur *OAttributes*, qui s'applique aux objets pour donner les attributs participant à sa construction :

OAttributes(O= (I, N, $<a_1:e_1, a_2:e_2, ..., a_n:e_n>$))={$a_i$ tels que i∈ [1...n]}

Chaque attribut A d'un objet O est défini par un doublet (a, e) où a∈ $\mathcal{A}$ est le nom de l'attribut A, et e∈ $\mathcal{V} \cup Seq(\mathcal{V}) \cup \mathcal{I} \cup Seq(\mathcal{I})$ est dit l'**état** de l'attribut A pour O (nous dirons aussi que A désigne e).

Nous considérons l'opérateur *AState* qui permet d'obtenir l'état d'un attribut A=(a, e) pour un objet O :

AState(O, A)=*e*.

Les définitions données aux valeurs et aux objets montrent que l'état d'un attribut participant à la construction d'une valeur peut être une valeur ou une séquence de valeurs. Tandis que l'état d'un attribut participant à la construction d'un objet peut être : une valeur, une séquence de valeurs, un identificateur ou une séquence d'identificateurs d'objets.

Dans le cas où l'état d'un attribut est un identificateur (une séquence d'identificateurs), l'attribut modélise une relation 1-1 (1-n) entre l'objet auquel est attaché l'attribut et l'objet (les objets) dont l'identificateur (les identificateurs) représente(nt) l'état de l'attribut. Les relations entre les objets peuvent représenter des contraintes précises comme celles existant entre les objets et leurs composants.

Reprenons l'exemple du module WindManips (Figure 1). Dans notre modèle, ce module est représenté par un objet de nom WindManips. Les propriétés d'un tel objet sont des *valeurs* (Dupont, 05.05.89). Les fonctions (OpenWind et CloseWind)

qui représentent les composants du module ont besoin d'être considérées comme des *objets* identifiés (de noms OpenWind et CloseWind) dont le module lui même représente l'abstraction. Ils sont dépendants, du point de vue existence, de l'objet représentant le module. Les autres objets référencés tels que ConcWindManips sont considérés comme des *objets* indépendants ayant des liens simples avec ce module.

Pour modéliser les différentes sortes de liens entre les entités du modèle, nous considérons trois **modes d'attributs** : les **attributs de propriété** modélisant les valeurs associées à l'objet comme étant ses propriétés (date de création, programmeur, etc), les **attributs de référence** représentant des relations simples entre les objets et les **attributs de composition** qui représentent les relations entre les objets et leurs composants. Ces derniers constituent des références à des objets identifiés ayant des contraintes supplémentaires d'existence. Pour désigner le **mode** d'un attribut, nous utilisons l'opérateur *Mode* défini ainsi :

Pour un attribut A d'un objet O, si A est un attribut de propriété de O alors *Mode*(O, A)=Prop, si A est un attribut de référence de O alors *Mode*(O, A)=Ref, et si A est un attribut de composition de O alors *Mode*(O, A)=Comp.

La sémantique associée à chaque mode d'attributs est définie dans les paragraphes suivants.

2.4.1. Attributs de Propriété

Ce que nous appelons **attributs de propriété** est la notion classique des attributs trouvée dans la plupart des modèles de données connus. Ils désignent des valeurs et non pas des identificateurs.

Chaque *attribut de propriété* est un doublet (a, e) tels que $e \in \mathcal{V} \cup Seq(\mathcal{V})$.

Dans le cas où un **attribut de propriété** est associé à un objet, cet attribut désigne une valeur appartenant à l'objet en propre. Cette valeur ne peut être accédée qu'à travers l'objet considéré et elle est créée après la création de l'objet auquel elle appartient et détruite en même temps que lui.

Par exemple, la date de création et les noms de programmeurs du modules WindManips sont modélisés par des attributs de propriété (DateCréation, Programmeurs) associés à l'objet WindManips. Ils représentent des valeurs non identifiées appartenant à cet objet. Si WindManips est supprimé, ces valeurs sont supprimées également.

En considérant les définitions des objets atomiques données dans le paragraphe 2.3, nous constatons que les objets atomiques sont constitués exclusivement d'**attributs de propriété**.

2.4.2. Attributs de Composition

Les **attributs de composition** sont utilisés pour modéliser les relations entre un objet et ses composants. Les objets ayant de tels attributs sont appelés **objets composites**. La notion d'objet composite permet de rendre compte de la structure hiérarchique d'un objet en composants, chacun des composants peut être simple ou composé lui même d'autres objets. Définir un objet comme étant composite représente une abstraction permettant de manipuler un ensemble d'objets comme étant un seul objet tout en gardant la possibilité d'accéder individuellement à chacun de ces objets. Cette notion est semblable à celle des liens de la catégorie *composition* de PCTE+ et à celle des références de *composition* dites *partagées* et *dépendantes* d'ORION.

Un attribut de composition désigne un ou plusieurs objets comme composants de l'objet considéré. Les composants ont une contrainte additionnelle de dépendance de leurs parents. Les composants sont des objets identifiés, ils peuvent être référés par les autres objets de la base et on peut y accéder directement sans passer par leurs parents.

Par exemple, les fonctions (OpenWind, CloseWind) constituant les composants du module WindManips sont désignées par un attribut de composition associé à l'objet WindManips. Ces fonctions dépendent du module WindManips du point de vue existence, mais ils peuvent être référencés par d'autres objets de la base.

Contrairement à la sémantique des liens exclusifs donnée dans Orion [Kim 87], nous permettons qu'un objet soit le composant de plusieurs objets composites. Cette possibilité constitue un support au partage de composants entre plusieurs objets. Un objet composant peut être ainsi référé par plusieurs objets composites au moyen de liens de composition, et l'existence de cet objet est dépendante de l'existence de tous les objets qui l'utilisent comme composant. Cela veut dire que *la suppression d'un objet composite n'implique pas forcément la suppression de tous ses composants, il implique seulement la suppression de ses composants non référencés par d'autres objets au moyen de liens de composition.*

Soit A=(a, e) un attribut de composition de l'objet O (Mode(O, A)=Comp) alors :

- $e \in \mathcal{I} \cup Seq(\mathcal{I})$, c'est-à-dire l'état d'un attribut de composition est un identificateur ou une séquence d'identificateurs d'objets.

- la suppression de O implique la suppression des objets dont les identificateurs sont désignés par A et qui ne sont pas désignés par d'autres attributs de composition.

L'ensemble des objets composites noté par $\mathcal{O}_{comp}$ est donc défini comme l'ensemble des objets ayant au moins un attribut de composition :

$\mathcal{O}_{comp}$={O tel que O$\in \mathcal{O}$ et O=(I, (A_1,..., A_n)) et $\exists$ i$\in${1, ..., n}tel que *Mode*(A_i)=Comp}

Pour chaque objet composite O nous pouvons former l'ensemble de ses composants ainsi :

Children(O)={O' tels que O' est désigné par un *attribut de composition* de O}

Inversement, pour chaque objet O$\in \mathcal{O}$ nous pouvons former l'ensemble de ses parents :

O $\in$ *Parents*(O') $\Leftrightarrow$ O'$\in$ *Children*(O)

La suppression d'un objet est exprimée par l'opérateur *Delete* défini ainsi :

Delete(O) implique *Delete*(O') si et seulement si O'$\in$ *Children*(O) et *Parents*(O')={O}

2.4.3. Attributs de Référence

Les **attributs de référence** sont utilisés pour modéliser les relations simples entre les objets. L'*attribut de référence* désigne donc un ou plusieurs objets identifiés et indépendants de celui possédant l'attribut, il n'implique aucune contrainte d'existence entre les objets.

Par exemple, le lien de référence existant entre le module WindManips et le document de conception ConcWindManips est modélisé par un attribut de référence *Conception* associé à l'objet WindManips. ConcWindManips est un objet indépendant de WindManips du point de vue de son existence.

Un *attribut de référence* est un doublet (a, e) tel que $e \in \mathcal{I} \cup Seq(\mathcal{I})$, c'est-à-dire l'*état* d'un *attribut de référence* peut être un identificateur d'objet ou une séquence d'identificateurs.

Pour chaque objet O nous pouvons former l'ensemble des objets auxquels il fait référence ainsi :

References (O) $\in$ {O' tels que O' est désigné par un *attribut de référence* de O}

Aucune contrainte de suppression n'est appliquée aux objets référés:

$\forall$ O' $\in$ *References* (O) : *Delete*(O) $\neq>$ *Delete*(O')

2.4.4. Attributs Opérationnels

Les attributs de propriétés, de références et de composition peuvent être **opérationnels**. Contrairement aux attributs classiques où il s'agit d'une information stockée dans une structure de données, l'*état* d'un attribut opérationnel est déduit par un programme lors de l'accès à cet attribut. A chaque attribut opérationnel est donc associé un **opérateur** dont le but est de définir son **état**.

Pour exprimer qu'un attribut A de l'objet O est opérationnel, nous utilisons l'opérateur *Operational* défini ainsi:

Operational(O, A)=T, si A est opérationnel
Operational(O, A)=F, si A est non-opérationnel.

Dans un environnement de programmation, les attributs opérationnels sont particulièrement importants. Ils permettent par exemple de gérer les objets dérivés dont les exemples sont nombreux : code objet dérivé du code source, termes d'indexation dérivés du texte des documents, etc.

2.5. TYPES

La notion de type représente la partie intentionnelle du modèle. En effet, les valeurs et les objets sont créés suivant des structures prédéterminées. Un type *spécifie une structure commune et un comportement commun à plusieurs objets*. Dans un contexte génie logiciel, la notion de type permet de considérer des ensembles de documents "similaires" ou "standardisés", ayant la même structure et le même comportement, c'est-à-dire, le même type.

Par la structure, nous désignons la partie statique du type, c'est-à-dire la description des entités de ce type. Par le comportement, nous désignons la partie dynamique du type définissant les opérations de manipulation et les contraintes de cohérence des objets de ce type. Cette deuxième partie est exprimée en utilisant des méthodes dont la définition sera donnée dans le chapitre 5.

2.5.1. Types de Base et Types Construits

Les *types de base* correspondent aux domaines que nous considérons de base pour la construction des valeurs et des objets. En plus des types conventionnels : entier, réel, booléen, chaîne de caractères, etc, nous considérons plusieurs autres *types de base* qui nous permettent de représenter et de manipuler le contenu des documents et des programmes. Chaque type de base met en œuvre les opérateurs de manipulation des entités ayant ce type (édition de texte, compilation de code source, etc). Nous désignons l'ensemble de types de base par $\mathfrak{B}$t. En particulier nous considérons les *types de base* suivants:

- *Text, Graphic* pour représenter et gérer les portions du contenu textuel ou graphique des documents.
- *SourceCode* pour représenter et gérer les portions de contenu des programmes.
- *ObjectCode* pour représenter et gérer le code objet associé au code source.

Les noms des *types de base* constituent un ensemble fini $\mathfrak{B}n$ de symboles, qui consiste en un symbole d_i pour chaque domaine $\mathfrak{D}_i$ (parmi lesquels nous retrouvons Text, SourceCode, ObjectCode, etc).

A partir des types de base, on peut définir d'une façon récursive des **types construits** en utilisant les constructeurs *Nuplet* et *Seq*. Pour donner une définition aux types construits nous considérons les deux ensembles suivants :

- $\mathfrak{C}n$, un ensemble infini de symboles désignant les noms des types construits.
- $\mathfrak{M}$, un ensemble dont les éléments sont appelés méthodes (la définition des méthodes sera donnée dans le chapitre 5)

Considérons l'ensemble de tous les types construits noté par $\mathfrak{C}t$, chaque type construit T de $\mathfrak{C}t$ est un triplet (t, S, M) tel que, $t \in \mathfrak{C}n$ est le nom du type, S est la *partie structurelle* du type, sa définition sera donnée dans le paragraphe suivant, et M est un sous ensemble de $\mathfrak{M}$ désignant les méthodes du type. M est la *partie dynamique* du type.

L'ensemble de tous les types est alors $\mathfrak{B}t \cup \mathfrak{C}t$ que nous désignons par $\mathfrak{T}$.

Nous appelons **extension** d'un type l'ensemble d'objets ayant une structure commune et un comportement commun décrits par ce type. Nous utilisons l'opérateur *Extension*(T) pour désigner cet ensemble, et l'opérateur *OType* pour retrouver le type d'un objet : $OType(O)=T \Leftrightarrow O \in Extention(T)$.

2.5.2. Partie Structurelle d'un Type Construit

La partie structurelle d'un type décrit la structure des données (valeurs, objets) de ce type.

Nous considérons l'opérateur *Structure*. Cet opérateur appliqué à un type T=(t, S, M), donne sa structure S. La structure d'un type construit est un *Nuplet* d'attributs :

$$Structure(T)=(a_1: Da_1, a_2: Da_2, \ldots, a_n: Da_n) \text{ tels que } a_i \in \mathfrak{A} \text{ et } a_i \neq a_j \text{ pour } i \neq j$$

La définition de chaque attribut comporte plusieurs informations qui servent pour structurer les données désignées par cet attribut et définir leur sémantique :

- le **mode** de l'attribut qui désigne si c'est un attribut de propriété, un attribut de référence ou un attribut de composition.
- le **type** des entités (valeurs, objets) qui peuvent être désignées par l'attribut.
- un **constructeur** qui spécifie si l'attribut va désigner une entité (valeur, identificateur), ou une séquence d'entités (séquence de valeurs, séquence d'identificateurs). Dans le premier cas, le constructeur est **Tuple** et dans le second cas, le constructeur est **Seq**.
- une **cardinalité** qui est un entier déterminant le nombre minimum d'entités pouvant être désignées par l'attribut. Une cardinalité 0 associée à un des attributs du type T signifie que cet attribut est *optionnel* pour les objets (ou les valeurs) ayant ce type (c'est-à-dire qu'il peut ne pas exister pour l'un de ces objets). Dans le cas où la cardinalité est un nombre positif (n>0), cela signifie que l'attribut désigne *au moins n entités* pour chaque objet de type T.
- une valeur (T ou F) qui détermine si l'attribut est **opérationnel** ou non (la définition des attributs opérationnels a été donnée dans §2.4.4).

Ainsi la description d'un attribut consiste en un quadruplet désignant les éléments mentionnés ci-dessus:

$\forall\ i \in [1, \ldots, n]$ alors $Da_i=(Mode_i, Cons_i, Card_i, t_i, Oper_i)$

tels que : $Mode_i \in \{Prop, Ref, Comp\}$ est le mode de l'attribut, $Cons_i \in \{Tuple, Seq\}$ est le constructeur de l'attribut, $Card_i$ est un entier ≥ 0 désignant la cardinalité de l'attribut, $t \in \mathfrak{C}n \cup \mathfrak{B}n$ désigne le type de l'attribut, $Oper \in \{T, F\}$ désigne si l'attribut est opérationnel ou non.

Nous considérons en particulier l'opérateur TAttributes qui s'applique à un type pour retrouver tous ses attributs, et les opérateurs *Mode* et *AType* qui s'appliquent à un type T et un attribut A_i pour désigner le mode et le type de A_i dans la structure de T (l'opérateur *Mode* utilisé ici est l'extension de l'opérateur *Mode* défini précédemment sur l'ensemble des objets):

$TAttributs(T)=\{a_i$ tels que $i \in [1...n]\}$, $Mode(T, A_i)=Mode_i$, $AType(T, A_i)=t_i$

Chaque attribut appartenant à la structure d'un type peut être un attribut de propriété, un attribut de composition ou un attribut de référence. Cela donne lieu aux opérateurs *Props*, *Refs* et *Comps* qui, appliqués à l'ensemble de types construits, donnent respectivement les attributs de propriété du type, ses attributs de référence et ses attributs de composition :

si T est un type tel que *Structure*(T)= (a_1: Da_1, ..., a_n: Da_n) alors
1) *Props*(T)={a_i tels que *Mode*(T, a_i)=Prop}
2) *Comps*(T)={a_i tels que *Mode*(T, a_i)=Comp}
3) *Refs*(T)={a_i tels que *Mode*(T, a_i)=Ref}

Puisque les objets sont classés en plusieurs catégories: atomiques, complexes et composites, les types sont également classés en plusieurs catégories :

- les **types atomiques** sont les types selon lesquels sont créés les objets atomiques. Les attributs appartenant à la structure de tels types sont des *attributs de propriété exclusivement* :
  T est un type atomique ⇔ Comps(T)=Ø et Refs(T)=Ø

- les **types complexes** possèdent dans leur structure *au moins un attribut de composition ou un attribut de référence* :
  T est un type complexe ⇔ ou bien Comps(T)≠Ø ou bien Refs(T)≠Ø

- les **types composites** sont les types selon lesquels sont créés les objets composites. Ils possèdent dans leur structure *au moins un attribut de composition* :
  T est un type composite ⇔ Comps(T)≠Ø

2.5.3. Partie Dynamique d'un Type Construit, Méthodes

La partie dynamique d'un type décrit le comportement des objets ayant ce type, elle est représentée par un ensemble de méthodes. Les méthodes dans notre proposition ne sont pas des simples fonctions exécutées lors de l'envoi de messages, comme c'est le cas dans les approches objet classiques. Les méthodes jouent le rôle de déclencheurs associés aux objets, elles expriment d'une façon déclarative les conditions nécessaires à leur exécution aussi que les actions à entreprendre suite à cette exécution. Elles ne sont pas seulement déclenchées à la suite de l'envoi explicite de messages aux objets, mais aussi d'une manière implicite dans des situations particulières décrites dans ces méthodes. Cet aspect du modèle sera décrit en détail dans le chapitre suivant.

2.5.4. Types des Valeurs

Les valeurs construites, comme les objets, sont créées suivant des types prédéfinis. Puisque les valeurs ne peuvent pas faire référence à des objets, les types de valeurs ne peuvent être qu'*atomiques*. C'est-à-dire, la structure d'un type de valeurs ne peut comporter ni attribut de référence ni attribut de composition, elle ne comporte que des attributs de propriété.

2.5.5. Spécialisation de Types

Sur l'ensemble de types nous définissons une relation d'ordre partiel **IsA**. Cette relation peut classer les types suivant un treillis de types. Chaque type peut être considéré ainsi comme étant un sous-type d'un ou de plusieurs autres types, on dit alors qu'il constitue une spécialisation de ces types.

La spécialisation d'un type revient à affiner sa définition pour obtenir une structure et un comportement plus précis. Spécialiser la structure d'un type revient à rajouter des attributs ou à affiner la définition de certains attributs en spécialisant leurs domaines. De même on peut spécialiser le comportement en définissant des méthodes supplémentaires ou en spécialisant la définition des méthodes existantes, c'est un aspect sur lequel nous reviendrons dans le chapitre suivant.

Pour définir la relation d'ordre partiel **IsA**, nous supposons l'existence d'un symbole Super$\in \mathfrak{C}n$, le type de nom Super constitue la racine du treillis des types, la relation **IsA** est définie ainsi :

1) pour chaque type T on a : T **IsA** Super.

2) soient les deux types T1=(t1, S1, M1), T2=(t2, S2, M2), on dit que T1 **IsA** T2 si et seulement si S1 $\mathbf{IsA_S}$ S2 et M1 $\mathbf{IsA_m}$ M2 tels que :

- $\mathbf{IsA_S}$ est une relation d'ordre partiel définie sur les structures : soient les deux structures S1, S2, on dit que S1 $\mathbf{IsA_S}$ S2 si et seulement si pour chaque attribut a_i de type t_{i2} appartenant à la définition de S2, a_i appartient à la définition de S1 avec un type t_{i1} tel que T_{i1} **IsA** T_{i2}.
- $\mathbf{IsA_m}$ est une relation d'ordre partiel définie sur les ensembles de méthodes : soit deux ensemble de méthodes M1, M2, on dit que M1 $\mathbf{IsA_m}$ M2 si et seulement si M1 est inclu dans M2.

3) si T1 **IsA** T2 alors chaque objet appartenant à l'extension de T1, appartient à l'extension de T2.

Cela signifie que lorsqu'un type T1 est défini comme étant une spécialisation de T2, T1 hérite de la structure et du comportement de T2. L'héritage utilisé ici est similaire à l'héritage multiple défini dans les modèles orientés-objet. Les conflits d'héritage sont résolus d'une manière similaire à celui proposé par Loops. En effet, Loops utilise un mécanisme effectuant l'héritage en profondeur d'abord, de gauche à droite, jusqu'à ce que les branches de la hiérarchie de types se rejoignent. Par exemple, si T est une spécialisation de deux types désignés par la liste (T1 T2), et si

T1 et T2 ont un attribut de nom a. Les objets de T utilisent l'attribut a spécifié par T1.

2.5.6. Aspects Syntaxiques

La syntaxe de définition d'un type est la suivante :

(DefineType <NomType>
IsA <ListeSuperTypes>
Props <ListeDéfinitionAttributs>
Comps <ListeDéfinitionAttributs>
Refs <ListeDéfinitionAttributs>
[**Methods** <ListeMéthodes>])

Tels que : <NomType> est un symbole de $\mathfrak{T}n$ désignant le nom du type, <ListeSuperTypes> est une liste de noms de types de $\mathfrak{B}n \cup \mathfrak{T}n$ désignant les noms des super-types, <ListeDéfinitionAttributs> est un ensemble de définitions d'attributs, il est donné par la grammaire suivante:

<ListeDéfinitionAttributs> ::= (<DéfinitionAttribut>$^+$)| **NIL**
<DéfinitionAttribut> ::= (Nom Const Card Type Oper)

tels que : Nom$\in \mathfrak{A}$, Card est un entier$\geq$0, Const$\in$ {Tuple, Seq}, Type$\in \mathfrak{B}n \cup \mathfrak{T}n$, Oper$\in$ {T, nil}, <ListeMéthodes> est un ensemble de méthodes.

Dans la Figure 2, nous donnons des exemples de définitions de types. Dans cet exemple, les attributs de propriété sont désignés par **Props**, les attributs de composition par **Comps** et ceux de référence par **Refs**.

```
( DefineType Module
  IsA      Source
  Comps    ((Fonctions Seq 1 Fonction))
  Refs     ((ModDeps Seq 0 Module)
            (Conceptions Tuple 0 ConcMod)
            (Spécifs Tuple 0 SpecMod)))

( DefineType Fonction
  IsA      Source
  Refs     ((appelle Seq 0 Fonction)
            (UtiliséPar Seq 0 Module)
            (Conceptions Tuple 0 ConcFonc)))

( DefineType Source
  Props    ((DateCréation Tuple 1 Date T)
            (Programmeur Tuple 1 String)))
```

Figure-2

2.6. MANIPULATION DES OBJETS

Dans les paragraphes précédents, nous avons défini plusieurs opérateurs de manipulation d'objets tels que: *Delete*, *AState*, *OType*, qui aident respectivement à supprimer un objet, à retrouver l'état d'un attribut, à retrouver le type d'un objet. Nous allons, dans les paragraphes suivants donner les autres opérateurs de manipulation d'objets.

2.6.1. Création d'un Objet

La création des objets se fait selon des structures prédéterminées par leur type. Pour créer un objet d'un certain type T nous utilisons l'opérateur NewObject, NewObject crée un objet et initialise ses attributs à nil.

$\forall$ T tel que *Structure*(T)= (a_1: Da_1, ..., a_n: Da_n), *NewObject* (T) =O

alors O=(I, N, (a_1: nil, ..., a_n: nil)) et $\forall$ O'=(I', N', E')$\in \mathcal{O}$, I'$\neq$I

2.6.2. Affectation d'Attributs de Propriété

L'affectation de l'attribut de propriétés A=(a, Cons, Card, t, Oper) d'un objet O=(I, N, E) de type T à une valeur V se fait en utilisant l'opérateur *PutProp*.

PutProp vérifie tout d'abord que A est un attribut de propriété de O : *Mode*(O, a)=Prop, et que V est du type t donné dans la définition de A : *VType*(V)=t (*VType* est un opérateur permettant de retrouver le type d'une valeur). Ensuite, si le constructeur de l'attribut est une séquence (Cons=Seq), E devient (..., a={V},...), et si Cons=Tuple, E devient (..., A=V,...).

Il faut rappeler ici que les valeurs ne peuvent exister qu'associées aux objets. La création d'une valeur se fait lors de l'affectation d'un attribut de propriété d'un objet.

2.6.3. Affectation d'Attributs de Composition

L'affectation de l'attribut de composition A=(a, Cons, Card, t, Oper) d'un objet O1=(I1, N1, E1) par un objet O2=(I2, N2, E2) revient à créer un lien de composition entre O1 et O2, cela se fait en utilisant l'opérateur *PutComp*.

PutComp vérifie tout d'abord que A est un attribut de composition de O: *Mode*(O, A)=Comp, et que O2 est d'un type correspondant à celui donné dans la définition de A (O2∈*Extension*(t)). Ensuite, si le constructeur de l'attribut est une séquence (Cons=Seq), E1 devient (..., a={I2},...) et si Cons=Tuple, E1 devient (..., A=I2,...). *PutComp* rajoute O2 à l'ensemble de composants de O1 et O1 à l'ensemble de parents de O2. Ainsi, O2∈*Children*(O1) et O1∈*Parents*(O2).

2.6.4. Affectation d'Attributs de Référence

L'affectation d'un attribut de référence A=(a, Cons, Card, t, Oper) d'un objet O1=(I1, N1, E1) par l'objet O2=(I2, N2, E2) se fait en utilisant l'opérateur *PutRef*.

PutRef vérifie tout d'abord que A est un attribut de référence de O1: Mode(O, A)= Ref, et que O2 est d'un type correspondant à celui donné dans la définition de A (O2∈*Extension*(t)). Ensuite, si le constructeur de l'attribut est une séquence (Cons=Seq), E1 devient (..., a={I2},...) et si Cons=Tuple, E1 devient (..., A=I2,...). *PutRef* rajoute O2 à l'ensemble de références de O1, Ainsi, O2∈*References*(O1).

2.6.5. Ajout d'Eléments aux Séquences

Plusieurs opérateurs sont définis pour manipuler les attributs ayant la séquence comme constructeur.

L'opérateur *AppendProp* appliqué à un objet O, un attribut A de O et une valeur V, a pour effet de vérifier que V est du type donné à A dans la structure du type de O, ensuite, il ajoute la valeur V à la fin de la liste des valeurs représentant l'état de l'attribut A.

L'opérateur *AttachProp*, est similaire à *AppendProp*, sauf qu'il ajoute V au début de la liste représentant l'état de A.

De même plusieurs autres opérateurs sont définis tels que : *AppendComp*, *AppendRef*, *AttachComp*, *AttachRef*.

3. DOCUMENTS

Un de nos objectifs premiers est de supporter les aspects hypertexte et la recherche par le contenu des documents. Le modèle de données doit donc contenir les éléments nécessaires pour de telles extensions. Par exemple, il doit supporter la représentation visuelle des documents à l'écran et la représentation de leur contenu sémantique au moyen de termes d'indexation qui peuvent avoir des structures plus ou moins complexes (simples mots-clés, graphes conceptuels, etc). Une attention toute particulière est donc donnée à la modélisation des documents.

Nous définissons le concept de **Document**. Les **Documents** sont des objets structurés possédant un contenu textuel, voire graphique, une représentation de leur contenu sémantique aussi qu'une représentation visuelle à l'écran. Les **Documents** représentent un sous ensemble $\mathfrak{Doc}$ de $\mathfrak{O}$, ils ont des sous-types du type prédéfini **Document** :

$\mathfrak{Doc}$={D tel que *OType*(D) **IsA** Document}

Le type **Document** est un type construit ayant en particulier deux attributs de composition : Content et Descriptor, représentant respectivement le contenu textuel et le contenu sémantique des documents, et un attribut de propriété Nœud qui désigne les nœuds de l'hypertexte correspondant aux documents.

Document$\in \mathfrak{C}$n tel que
Content, Descriptor, Nœud$\in$ *TAttributes*(Document) et
AType(Document, Content)$\in$ {Text, SourceCode) et
Mode(Document, Content)=
Mode(Document, Descriptor)=Comp et
Mode(Document, Nœud)=Prop et
Operational(Document, Descriptor) et
Extension(Document)=$\mathfrak{Doc}$

Le type de l'attribut Descriptor désigne en effet la structure des termes d'indexation utilisés pour décrire le contenu sémantique des documents. Le processus associé à cet attribut opérationnel représente le processus d'indexation utilisé pour trouver le contenu sémantique des documents, c'est-à-dire, la valeur de l'attribut Descriptor. Le processus peut être soit manuel soit automatique, il est à spécifier pour chaque type de documents. Par défaut, lorsque l'attribut Descriptor n'est pas spécifié pour un type de documents, nous considérons que les documents de ce type sont à indexer manuellement par des mot-clés.

L'étude des techniques d'indexation pour la définition d'un processus automatique d'indexation appliqué à ce type de documents est encore peu abordé. Dans ce travail

nous ne nous sommes pas intéressés à ce type de problèmes. Nous offrons seulement la possibilité de gérer un tel processus et les termes d'indexation résultants.

Lors de la création d'un *Document*, une fenêtre va être créée automatiquement, et sera désignée par l'attribut Nœud. Cette fenêtre représente un nœud de l'hypertexte. Pour afficher sur l'écran la fenêtre représentant un document, l'opérateur *Display* défini sur l'ensemble de documents est utilisé :

$\forall$ D $\in$ $\mathfrak{Doc}$: Display (D) affiche la fenêtre AState(D, Nœud) sur l'écran.

En effet, l'affichage de documents complexes, qui constitue un problème en soi, n'est pas abordé dans ce travail. Nous utilisons simplement les fonctionnalités offertes par les systèmes hypertexte.

Un type de documents est défini donc par son super type, le type du contenu des documents correspondants, le type des descripteurs associés et la liste des descriptions des autres attributs de ce type. La syntaxe utilisée est la suivante :

(**DefineDocType** <NomType>
IsA <ListeSuperTypes>
Content <TypeContenu>
Descriptor <TypeDescripteur>
Props <ListeDéfinitionsAttributs>
Comps <ListeDéfinitionsAttributs>
Refs <ListeDéfinitionsAttributs>
[**Methods** <ListeMéthodes>])

où <ListeSuperTypes> est une liste de nom de types de $\mathfrak{B}$ n$\cup$ $\mathfrak{C}$ n, <TypeContenu>$\in$ {Text, SourceCode}, <TypeDescripteur>$\in$ $\mathfrak{C}$n est le type des descripteurs associés à ce type de documents.

Dans la figure 3, nous donnons quelque exemples de définitions de types de documents. Remarquons que dans le cas où la clause Descriptor prend la valeur NIL, le type des descripteurs considéré est une séquence de chaînes de caractères représentant une liste de mot-clés, et l'indexation sera effectuée manuellement.

```
( DefineDocType SpecDoc
  IsA        Documentation
  Content    Text
  Descriptor NIL
  Comps      ((SpecFonc Tuple 1 DoSpecFonc)
              (SpecNonFonc Tuple DocSpecNonFonc)))

( DefineDocType DocSpecFonc
  IsA        Documentation
  Content    Text
  Descriptor NIL
  Comps      ((SpecMods Seq 0 DocSpecMod))
  Refs       ((Conceptions Tuple 0 DocConc)))

( DefineDocType Documentation
  Props      ((DateCréation Tuple 1 Date)
              (Auteur Seq 1 String)))
```

Figure-3

4. PROJETS

La notion de *Projet* est utilisée pour définir des environnements de programmation spécifiques dans le but de s'adapter à des stratégies de développement précises. Les projets définissent ainsi des *sous-bases* (une sorte de vues) de la *base de données*.Cette notion est semblable à celle de SDS (Schema Definition Set) de PCTE.

Pour donner une définition à la notion de *projet* nous allons tout d'abord définir la notion de *base*. Une *base*, notée $\mathfrak{B}$, est définie par l'ensemble de tous les types et l'ensemble des objets représentant l'union des extensions de ces types :

$\mathfrak{B}=(\mathfrak{T}, \mathfrak{O})$ tel que $\mathfrak{O}=\cup$ *Extension*(t) pour $t \in \mathfrak{T}$

Un projet, noté par p, définit un ensemble de types $\mathfrak{T}p$ pouvant être utilisés par les objets du projet, un administrateur, utilisateur privilégié, capable de définir les types du projet, et les autres utilisateurs ayant le droit de créer et de manipuler les objets de ce projet. A chaque projet est ainsi associée une base $\mathfrak{B}p$ telle que :

$\mathfrak{B}p=(\mathfrak{T}p, \mathfrak{O}p)$, $\mathfrak{O}p=\cup$ *Extension*(t) pour $t \in \mathfrak{T}p$.

Pour intégrer la notion de projet dans le modèle nous considérons le type **Project**. Tous les projets sont des objets de ce type. Soit $\mathfrak{P}roj$ l'ensemble de tous les projets :

Project$\in \mathfrak{C}n$ tel que *Attributes*(Project)={SuperUsers, GeneralUsers, ImportedTypes, DefinedTypes} et *Extension*(Project)=$\mathfrak{P}roj$

La syntaxe utilisée pour la définition d'un projet est la suivante :

(**DefineProject** <NomProj>
SuperUsers <NomDesAdminist>
GeneralUsers <NomDesUtil>
ImportedTypes <ListeTypeImportes>)

où <NomDesAdminist> et <NomDesUtil> sont deux listes de chaînes de caractères désignant les noms des utilisateurs, <ListeTypeImportes> est un ensemble de doublets désignant, d'une part, un nom de type et, d'autre part, le nom du projet dans lequel il est défini.

Les types peuvent être définis dans un projet en utilisant l'opérateur *DefineType* considéré pour l'ensemble $\mathfrak{P}$roj :

DefineType(p, tn)=T alors tn$\in \mathfrak{C}$n, T$\in \mathfrak{C}$, tn est le nom de T,
tn$\in$ *AState* (p, DefinedTypes)

5. MODELE DE VERSIONS

Prendre en considération l'évolution d'un objet pendant le processus de développement revient à considérer un ensemble de versions représentant cet objet. Une version de l'objet est *un état de cet objet que l'on a voulu conserver durant son processus de développement* [Fau 88]. Dans un environnement de génie logiciel il est admis classiquement trois types d'évolutions possibles pour chaque objet [Bel 89, Dit 85, Kat 84].

Le premier correspond à la coexistence de plusieurs représentations d'un objet. Des représentations différentes d'un objet constituent des développements parallèles et indépendants de celui-ci.

Par exemple, un logiciel peut être implanté en plusieurs langages de programmation, ce qui donne lieu à plusieurs programmes représentant le même logiciel. Un logiciel peut être également implanté en utilisant différents algorithmes de conception, ce qui donne lieu à plusieurs programmes et à plusieurs documents de conception représentant le logiciel.

Les représentations différentes d'un objet ont la même structure et peuvent avoir des propriétés en commun qu'on appellera *invariants*. Dans l'exemple précédent, les différents programmes qui implantent le même logiciel sont équivalents du point de vue fonctionnel, dans le sens où ils réalisent les mêmes fonctionnalités, et, donc, sont liés au même document de spécification.

Le deuxième type d'évolution constitue l'évolution des objets selon le temps, il correspond aux raffinements successifs et aux corrections des erreurs dans une

représentation d'un document, donc à des variations légères dans le contenu durant le processus de construction d'un document. Cet aspect amène à considérer une succession de révisions de l'objet. Chaque révision peut remplacer celle qui la précède du point de vue sémantique, puisqu'elle est plus complète ou plus correcte "sémantiquement". Les révisions successives d'un objet *ont la même structure.*

Le troisième type d'évolution vient du fait de l'insuffisance de la notion de révisions successives d'un objet. En effet, le processus de construction de documents est rarement linéaire. Par exemple, plusieurs tentatives de solutions peuvent être envisagées pour corriger des erreurs dans un document, plusieurs utilisateurs peuvent aussi mener des développements parallèles partant du même document.

En effet, le modèle de versions de base utilisé pour supporter l'évolution des objets que nous proposons intègre les propositions faites dans le domaine [Kat 90, Dit 85, Kim 87, Fau 88, Fau 91]. L'originalité de nos propositions réside dans l'extension des concepts existant pour supporter la notion d'objets composites définie dans le paragraphe 2.4.2.

5.1. MODELE DE BASE

5.1.1. Structure de Versions

Pour faciliter l'accès aux versions d'un objet et représenter les différents types d'évolution possibles, il est souhaitable de structurer l'ensemble des versions. En effet, une version d'un objet peut être *tirée* du néant, ce qui correspond à une représentation différente de l'objet (premier type d'évolution décrit précédemment), ou *dérivée* d'une autre version, ce qui correspond aux révisions successives et parallèles d'une certaine représentation de l'objet (deuxième et troisième type d'évolution). L'ensemble des versions d'un objet est structuré suivant l'origine de chaque version lors de sa création :

- La **dérivation** d'une version constitue la notion d'évolution d'un objet dans le temps. Elle correspond aux raffinements successifs et aux corrections des erreurs durant le développement de l'objet.

  Dans la réalité, plusieurs versions peuvent être dérivées d'une même origine. Elles correspondent à des évolutions parallèles de l'objet. Les dérivations successives et parallèles partant d'une version origine conduisent à considérer un arbre de versions dérivées dont la racine est l'objet version considéré, et les nœuds sont les différentes versions différentes de cet objet. Cet arbre est appelé **arbre de dérivation** [Kat 85, Fau 88]. Chaque version

O de Ⓞv est associée à un ensemble de versions par l'opérateur *Derived* dont l'opérateur inverse est *Origin*:

Derived(O)={Oi / Oi est une version dérivée de O}.
Origin(O)= O' : O' est l'origine de O dans l'arbre de dérivation.
Origin(O)=nil : O est une racine d'un arbre de dérivation.

La dérivation d'une version consiste à faire une copie de la version originale et à lui associer un identificateur externe dérivé de l'identificateur de son origine et de celui de sa voisine (voir Figure 4).

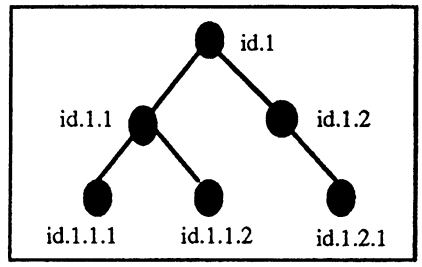


Figure-4

Deux versions sœurs dans l'arbre de dérivation ne peuvent pas être permutées, dans le sens où l'une ne peut pas remplacer l'autre du point de vue sémantique, tandis que toutes les deux peuvent remplacer leur origine. La relation de dérivation de versions définit donc une relation d'ordre partiel sur l'ensemble de versions d'un objet.

- La **Création** d'une représentation différente d'un objet correspond à considérer une évolution parallèle et indépendante de cet objet (voir Figure 5).

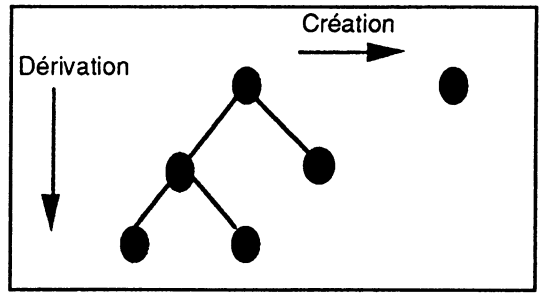


Figure-5

Plusieurs représentations différentes, que nous appelons par la suite **alternatives**, peuvent coexister pour un objet générique (i.e. implantations en plusieurs langages de programmation, des algorithmes de conception différents pour la même fonctionnalité d'un logiciel).

Chaque alternative peut évoluer suivant les deux axes de dérivation, successive et parallèle, décrits précédemment et constituer, ainsi, son propre

arbre de dérivation. Donc, pour chaque objet on peut considérer plusieurs arbres de dérivation (voir Figure 6).

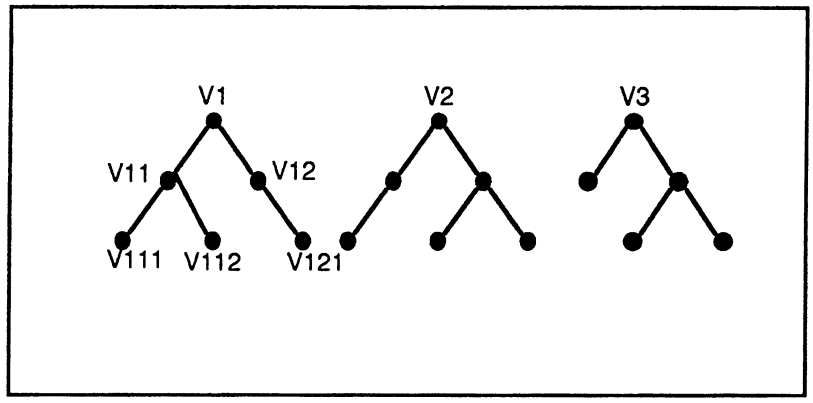


Figure-6

La décision de créer ou de dériver une version pour un objet est prise par l'utilisateur, toutefois, il faut assurer un minimum de contrôle sur ces décisions. Ces contrôles peuvent être exprimés au niveau des applications en utilisant le mécanisme de déclencheurs que nous présentons dans le chapitre 5.

5.1.2. Objets Génériques

Pour regrouper les versions résultant de l'évolution d'un objet, désigner leur structure et désigner leurs propriétés communes et leurs invariants, nous utilisons la notion d'*objet générique*. Un *objet générique* correspond à une abstraction selon laquelle un ensemble de versions peut être considéré comme un tout.

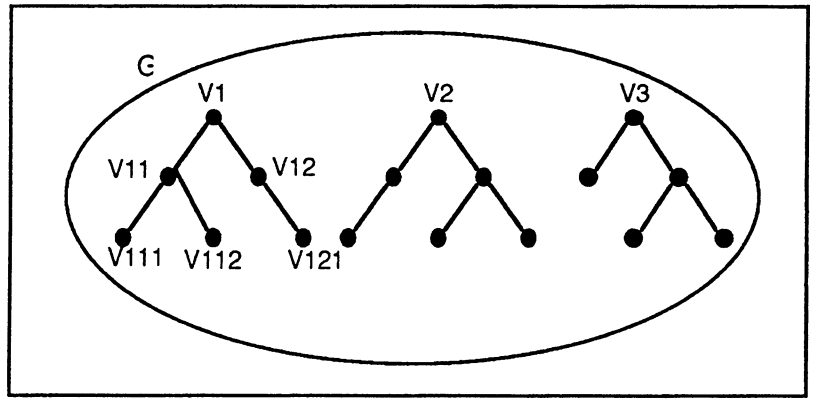


Figure-7

Nous considérons l'opérateur *Alternatives* qui s'applique aux objets génériques pour désigner toutes les alternatives, et l'opérateur *Generic* qui s'applique à une version pour désigner l'objet générique auquel elle appartient.

Chaque objet générique indique parmi ses versions la *version par défaut* et la *version courante*. La version courante est celle dont la date de création est la plus

récente. La version par défaut est la version prise par défaut lors de l'accès à l'objet générique, elle peut être désignée explicitement par l'utilisateur, dans le cas où aucune version n'est indiquée, la version courante est prise par défaut. Les opérateurs *Default* et *Current* sont utilisés pour retrouver respectivement la version par défaut et la version courante d'un objet générique.

Par ailleurs, nous considérons que toutes les versions d'un objet générique ont le même type : cela signifie que dans cette étude nous ne prenons pas en considération l'évolution de la structure d'un objet durant son développement.

5.1.3. Attributs Invariants

Certains attributs doivent conserver toujours le même **état** lors de l'évolution de l'objet, ces attributs sont désignés comme étant *invariants*. Par exemple, un module peut exister en plusieurs versions correspondant à des algorithmes différents réalisant les mêmes fonctionnalités. Ces versions différentes du module doivent être liées au même document de spécifications fonctionnelles. L'attribut désignant ce document doit être donc défini comme étant invariant.

Les attributs invariants sont spécifiés dans la structure de leur type. Un attribut est dit **invariant** si son état est le même pour toutes les versions d'un objet générique. Les attributs **invariants** sont associés aux objets génériques et leur *état* est propagé à toutes les versions de cet objet.

5.2. INTEGRATION AU NIVEAU DU MODELE DE DONNEES

Les versions sont des objets identifiés (identificateurs externes et internes), elles représentent un sous-ensemble, désigné par $\mathcal{O}v$, de l'ensemble d'objets $\mathcal{O}$. De même, les objets génériques sont des objets identifiés, ils représentent un sous-ensemble $\mathcal{O}g$ de $\mathcal{O}$.

Nous considérons deux types : VObject, GObject. Le type VObject spécifie les propriétés communes à toutes les versions et nécessaires pour enregistrer leur historique (date de création, la version origine et ses versions dérivées, etc), son extension représente l'ensemble de toutes les versions : *Extension*(VObject)=$\mathcal{O}v$. Parallèlement, le type *GObject* est utilisé pour désigner les propriétés communes à tous les objets génériques (nombre d'alternatives, alternatives, version courante, version prise par défaut lors de l'accès à cet objet, etc), son extension est l'ensemble de tous les objets génériques : *Extension*(GObject)=$\mathcal{O}g$.

Dans ce contexte, la création d'un type d'objets susceptibles d'avoir des versions consiste en deux étapes : la création d'un type correspondant sous-type de *GObject*, et la création d'un type correspondant sous-type de *VObject*.

5.3. GENERALISATION DU MODELE DE VERSIONS AUX OBJETS COMPOSITES

5.3.1. Objets Génériques Composites

Nous avons vu que les objets peuvent être composites, donc être composés d'autres objets. Si l'objet composite est susceptible d'avoir de versions, lui et ses composants ont des versions (voir Figure 8).

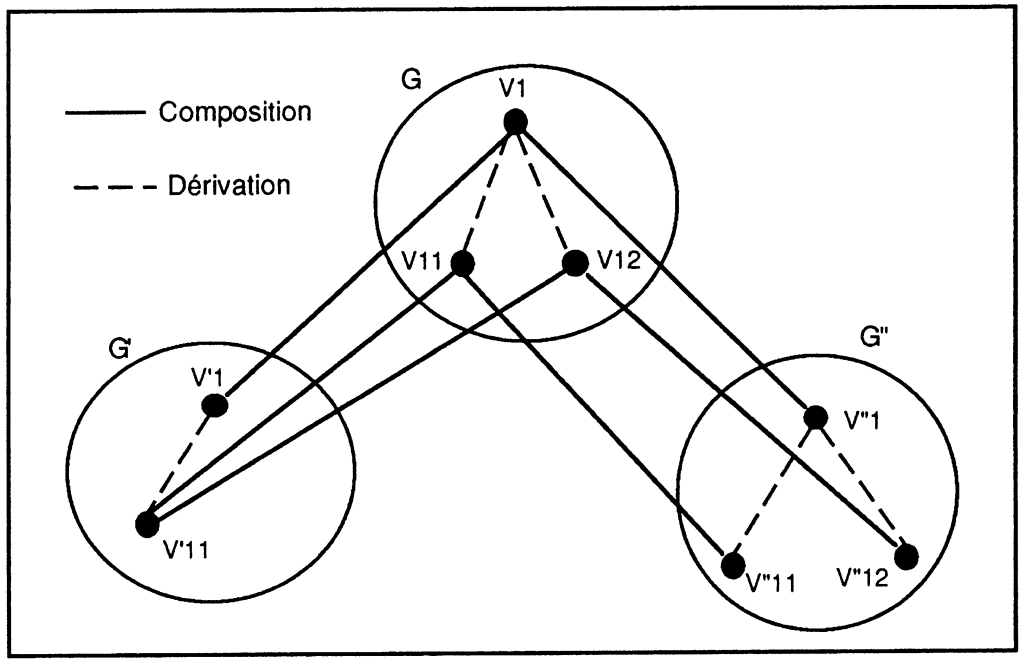


Figure-8

L'existence de liens de composition entre les versions implique l'existence de liens de composition entre les objets génériques de ces versions. Nous avons donc deux niveaux de structuration pour les objets correspondant à des granularités différentes, donc à des niveaux d'abstractions différents (voir Figure 9).

Les liens de composition s'appliquent alors aux objets génériques, ce qui donne des **objets génériques composites.** Un lien de composition existant entre deux objets génériques G et G' veut dire que les composants des versions de G sont des versions de G'.

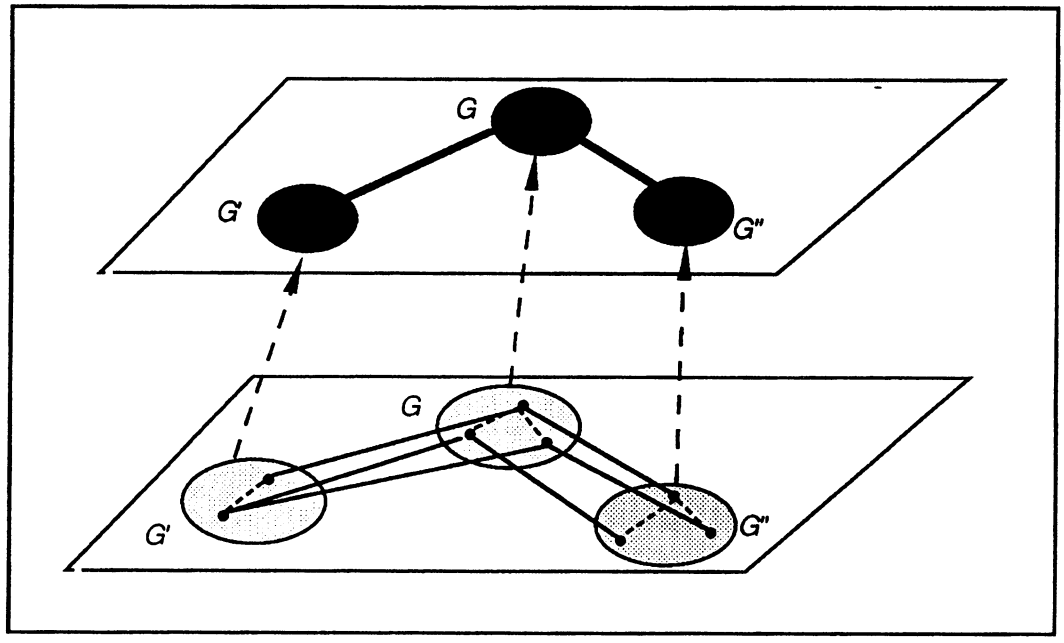


Figure-9

La gestion de ces niveaux de granularité différents impose de définir un ensemble de règles pour exprimer la sémantique de liens de composition pouvant exister entre les versions, entre les objets génériques et entre les versions et les objets génériques :

- L'ensemble des liens de composition existant entre deux objets versions est vu comme étant un lien de composition entre les objets génériques des deux versions (voir Figure 9). Cette règle généralise la notion d'objet générique à celui d'**objet générique composite**. Nous appelons l'ensemble des objets génériques composites $\mathcal{G}$comp :

  1) $\forall$ G $\in$ $\mathcal{G}$comp alors $\exists$ V tel que G=*Generic*(V) et V$\in$ $\mathcal{O}$comp
  2) si O'$\in$ *Children*(O) et *Generic*(O)=G et *Generic*(O')=G'
     alors G'$\in$ *Children*(G)

  Les objets génériques composites obéissent à la contrainte de composition : supprimer un objet générique composite implique la suppression de tous ses composants s'ils ne sont pas des composants d'autres objets, et, si le composant est un objet générique, sa suppression implique à son tour la suppression de toutes ses versions :

  3) $\forall$ G $\in$ $\mathcal{G}$comp, *Delete*(G) implique $\forall$ G'$\in$ *Children*(G),
     si *Parents*(G')={G} alors *delete*(G') et $\forall$ O$\in$ *Versions*(G), *Delete*(O)

- L'existence d'un lien de composition (ou de référence) entre deux objets génériques G et G' suivant un attribut de composition (ou de référence) A

signifie que les composants des versions de G, suivant l'attribut A, sont forcément des versions de G' (voir Figure 10):

4) G∈ 9comp et A∈ *Comps*(G) et *Astate*(G, A)=G' et G'∈ ⊙g
alors ∀ O∈ *Versions*(G), *AState*(O, A)∈ *Versions*(G')

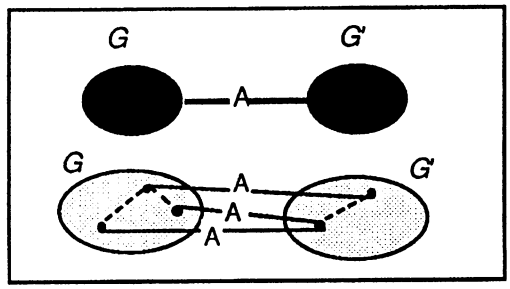


Figure-10

Si A est un invariant, la version par défaut de G' représente le composant de toutes les versions de G selon cet attribut (voir Figure 11).

5) si G ∈ 9comp et A∈ *Comps*(G) et *Invariant*(G, A) et *AState*(G,A)=G' et G'∈ ⊙g alors ∀ O∈ *Versions*(G), *AState*(O,A) =*Default*(G')

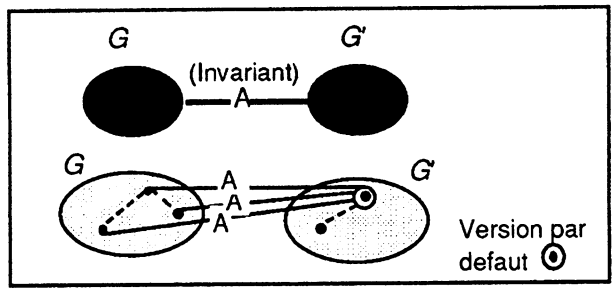


Figure-11

- Un lien correspondant à un attribut de composition (ou de référence) A existant entre un objet générique G et une version O' implique que toutes les versions de l'objet générique G désigne cette version par l'attribut A:

6) si G ∈ ⊙g et A∈ *Comp*(G) et *AState*(G, A)=O' et O'∈ ⊙v
alors ∀ O∈ *Versions*(G), *AState*(O,A) = O'

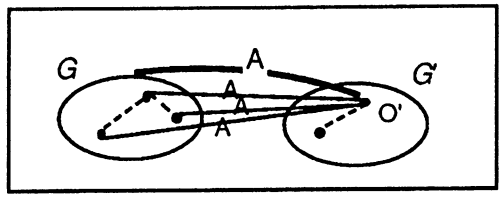


Figure-12

- Un lien correspondant à un attribut de composition (ou de référence) A et existant entre une version O et un objet générique G implique que la version fait référence à la version prise par défaut de l'objet générique (voir Figure 13):

7) si O ∈ ⊙v et A=*Comps*(O) et *AState*(O, A)=G tel que G∈ ⊙g
alors *AState*(O, A)=*Default*(G)

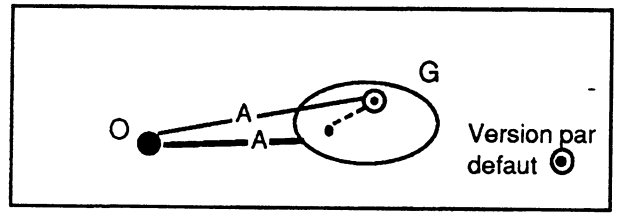


Figure-13

- Les composants d'une version sont, ou bien, les mêmes que ceux de son origine, ou bien, des versions dérivées des composants de son origine (voir Figure 14):

  8) ∀ O, O1, O' ∈ ℧v tels que *Origin*(O1)=O et A∈ Comps(O) et AState(O, A)=O' alors
  AState(O1, A)=O' ou AState(O1,A)∈ *Derived*(O')

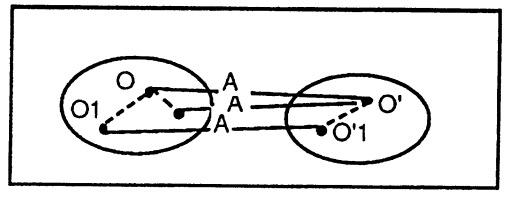


Figure-14

5.3.2. Configurations

Le résultat de la combinaison des deux concepts *version* et *objet générique composite* est la notion de version d'objet composite, appelée **configuration.**

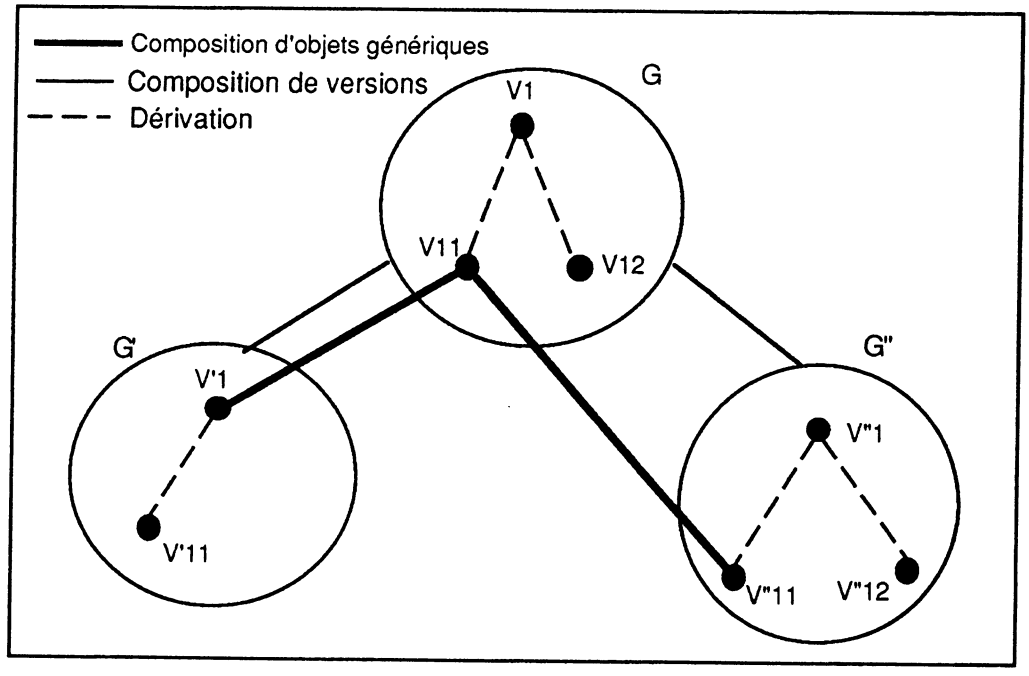


Figure-15

Une configuration d'un objet générique composite est une version *complète* de cet objet vérifiant un ensemble de critères. Par version complète, on sous-entend

qu'elle désigne une version de chaque composant de l'objet générique et ainsi de suite jusqu'aux feuilles de la hiérarchie de composition de l'objet. Dans la figure 15, les versions (V11, V'1, V"11) constituent une configuration.

Par exemple, un programme est composé généralement de plusieurs modules, et chaque module est composé de plusieurs fonctions. Chacun des composants peut exister en plusieurs versions correspondant à des langages de programmation différents. Il est intéressant de pouvoir retrouver le programme écrit en Langage C. Ceci revient à la création d'une configuration de l'objet générique représentant le programme en choisissant les modules écrits en C, et pour chacun les fonctions écrites en C.

Pour construire une configuration d'un objet générique G (noté par Conf(G)) vérifiant un ensemble de critères décrit par une expression Exp. Nous considérons les définitions suivantes :

1) on dit qu'une version O *vérifie une expression* Exp, s'il est possible d'évaluer Exp pour O et si Exp(O)=vrai.

2) si Children(G)≠Ø alors $Conf(G)=\{V : V \in Versions(G) \text{ et } V \text{ vérifie Exp}\} \cup_{G_i \in Children(G)} Conf(G_i)$.

Les expressions que nous considérons pour la création de configurations sont des expressions logiques ne pouvant avoir que l'une des formes suivantes :

- Exp= DefaultVersion : cette expression est évaluable pour toutes les versions; elle est évaluée à vraie si la version est la version par défaut de l'objet générique considéré. Cette expression permet de construire une configuration en choisissant pour chaque objet générique la version par défaut.
- Exp=CurrentVersion : cette expression est évaluable pour toute les versions; elle est évaluée à vrai seulement pour la version courante de l'objet générique considéré. Cette expression permet de construire une configuration en choisissant pour chaque objet générique la version courante.
- Exp peut être une expression logique de premier ordre portant sur les *états* des versions, c'est-à-dire, sur les valeurs ou les objets désignés par leurs attributs. Par exemple : Langage="C" et Programmeur="Dupont" est une expression vérifiée par les versions ayant deux attributs Langage et Programmeurs évalués respectivement à "C" et "Dupont".

5.4. MANIPULATION DES VERSIONS ET DES OBJETS GENERIQUES

5.4.1. Création d'une Version

La création d'une version d'un objet générique se fait en utilisant l'opérateur *NewVersion*. Cet opérateur, appliqué sur un objet G, crée une version O de cet objet :

NewVersion(G)=O alors O∈ *Alternatives*(G)

5.4.2. Dérivation de Versions

La dérivation d'une version à partir d'une version origine se fait en utilisant l'opérateur *DeriveVersion*. Cet opérateur, appliqué à une version O, crée une autre version O' de la façon suivante :

DeriveVersion(O)=O' alors O'∈ *Derived*(O) et
∀ A∈ *Props*(O), *AState*(O, A)=*AState*(O', A) et
∀ A∈ *Comps*(O)∪*Refs*(O), *AState*(O, A)=nil et
si O∈ 𝔇oc, *AState*(O,Content)=*AState*(O',Content)

Remarquons que la version dérivée a, pour les mêmes valeurs pour les attributs de propriétés que son origine, son contenu est identique à celui de son origine et a la valeur nil pour les autres attributs.

5.4.3. Interrogation d'Attributs

Pour obtenir l'état d'un attribut *a* associé à un objet O, on utilise l'opérateur *GetAttribute*. Cet opérateur respecte les règles définissant la sémantique des objets génériques et des versions définies dans le paragraphe 5.3.1.

5.4.4. Suppression d'une Version

La suppression d'une version a pour effet de supprimer successivement tous ses composants qui ne sont pas des composants d'autres versions et de reconstruire son sous arbre de dérivation.

si *Delete*(O) alors ∀ O'∈ *Children*(O),si *Parents*(O')={O} alors *Delete*(O')
et ∀ O"∈ *Derived*(O), *Origin*(O")=*Origin*(O)

5.4.5. Suppression d'un Objet Générique

La suppression d'un objet générique G consiste en la suppression de cet objet avec toutes ses versions et la suppression successive de tous les objets génériques représentant ses composants qui ne sont pas référés par d'autres objets comme composants.

delete(G) alors ∀ O'∈ *Children*(G), si *Parents*(O')={G} alors *delete*(O')
et ∀O"∈ *Versions*(O), *Delete*(O)

6. CONCLUSION

Dans ce chapitre, nous avons présenté la partie liée aux aspects statiques d'un modèle de données qui supporte la particularité des informations génie logiciel. Le modèle défini est un modèle sémantique, utilisant une approche orientée valeur et objet. L'orientation valeur est supportée par les attributs de propriété qui désignent des valeurs associées aux objets. Parallèlement, l'orientation objet est supportée en définissant les attributs de référence et de composition qui représentent les associations entre les objets. Contrairement aux attributs de référence, qui désignent les liens simples entre les objets, les attributs de composition désignent des liens avec des contraintes additionnelles de dépendance supportant la sémantique des objets composites. Les composants peuvent être partagés entre plusieurs objets.

La notion de *document* a été introduite pour pouvoir représenter, en plus du contenu textuel d'un document, son contenu sémantique et son présentation visuelle. Notre objectif étant d'augmenter le modèle par les éléments de base pour supporter la recherche des informations par leur contenu sémantique et les fonctionnalités d'un système hypertexte.

Pour pouvoir définir des environnements de programmation spécifiques et s'adapter à des stratégies de développement précises, nous avons défini la notion de *projet*. Un projet définit une vue de la base permettant ainsi la coexistence de plusieurs bases de données.

Par ailleurs, les notions d'*objet générique composite* et de *configuration* permettent d'avoir deux niveaux de structuration des objets correspondant à des niveaux d'abstraction différents.

CHAPITRE 5

MODELE DE DONNEES, ASPECTS DYNAMIQUES

1. INTRODUCTION

Dans le chapitre précédent, nous avons présenté la partie du modèle de données concernant les aspects statiques. Les aspects dynamiques du modèle, liés au comportement des objets, à l'intégrité et à la cohérence des données sont traités dans ce chapitre.

L'introduction de la dynamique dans les gestionnaires d'objets des EGLs devient une nécessité pour plusieurs raisons. D'une part, les informations génie logiciel sont sujettes à un traitement continu soit pour la saisie et la mise-à-jour, soit pour l'activation d'outils génie logiciel. D'autre part, elles subissent une évolution continue nécessitant la vérification de leur intégrité et la prise en compte des effets de bord des changements effectués.

Reprenons l'exemple de module structuré en fonctions présenté dans le chapitre précédent. Outre les fonctions composant le module, chaque module peut aussi utiliser des fonctions définies par d'autres modules. Après la modification d'une fonction, tous les modules utilisant cette fonction doivent être compilés. La modification de la fonction peut aussi entraîner des modifications dans la partie du document de conception correspondant à cette fonction. La compilation des modules utilisant la fonction modifiée doit être faite automatiquement par le gestionnaire de la base pour maintenir sa cohérence. Si la compilation de l'un de ces modules ne réussit pas, les modifications dans la fonction ne doivent pas être prises en compte et l'utilisateur doit de nouveau être amené à modifier cette fonction. La partie du document de conception correspondant à la fonction modifiée doit être localisée et affichée sur l'écran pour que l'utilisateur puisse effectuer des modifications si nécessaires.

Pour gérer de telles situations, un mécanisme puissant doit être défini. Ce mécanisme doit tenir compte de la particularité d'une application génie logiciel où

les opérations (compilation, édition, etc), représentant le comportement des objets, peuvent être très longues et coûteuses.

Dans le paragraphe 2 nous commençons par préciser l'approche utilisée pour représenter la dynamique des objets. Le paragraphe 3 donne une présentation sommaire de la notion de transaction utilisée par notre étude. Le paragraphe 4 est consacré à la définition des méthodes et à la description de leurs composants. Dans le paragraphe 5, nous présentons le mécanisme de déclencheur utilisé pour la gestion des méthodes. Des exemples d'utilisation de ce mécanisme, soit au niveau des applications soit au niveau de la gestion d'aspects sémantiques du modèle de données, sont présentés dans le paragraphe 6.

2. L'APPROCHE UTILISEE

Dans les modèles Orientés-Objet le comportement d'un objet est défini par un ensemble de procédures ou de fonctions (opérations), appelées méthodes, attachées à son type. Une méthode est définie par un nom (ou sélecteur), un ensemble de paramètres et le corps (le code). L'envoi de message est l'unique moyen de communiquer avec les objets.

Un message est défini par : l'objet receveur du message, le sélecteur de la méthode à déclencher, et la liste des paramètres effectifs. L'envoi d'un message s'interprète comme un appel de la méthode de nom sélecteur associée à l'objet receveur et appliquée aux arguments effectifs donnés par le message.

Le contrôle de la cohérence de la base et des effets de bord des opérations est noyé dans le code des méthodes d'une façon procédurale ou effectué explicitement par les utilisateurs ou les applications. L'inconvénient d'une telle approche est lié à son aspect procédural. Il est souhaitable de pouvoir exprimer d'une façon déclarative les règles de cohérence de la base et les effets de bord de déclenchement des méthodes.

L'approche que nous considérons consiste à intégrer la dynamique au niveau du modèle de données. Pour cela, nous définissons un formalisme pour les méthodes permettant d'exprimer d'une façon déclarative, les situations de leur déclenchement, les conditions à vérifier avant l'exécution de leur code et les effets de leur exécution. Les méthodes représentent donc des déclencheurs, elles ne sont pas exécutées seulement lors de l'envoi explicite de messages aux objets, mais elles peuvent être déclenchées implicitement dans des situations particulières.

Dans un objectif de performance, nous avons offert la possibilité que les méthodes spécifient d'une façon déclarative les objets qui doivent réagir à la suite de leur

exécution. Ainsi, nous offrons une certaine limite de la propagation des effets de bord résultant de l'exécution d'une méthode.

Un mécanisme de déclencheurs est utilisé pour la gestion des méthodes ainsi définies. Comme nous allons voir par la suite, ce mécanisme est basé sur la notion de transactions imbriquées.

Cette vision des méthodes permet d'exprimer non seulement le comportement classique des objets, mais également, les contraintes à satisfaire pour l'exécution d'une méthode, les effets de bord de l'exécution d'une méthode, les droits d'accès des utilisateurs aux objets, la gestion d'attributs opérationnels, le contrôle de versions, la gestion de configurations, etc.

3. NOTION DE TRANSACTIONS

3.1. INTRODUCTION

Classiquement, une transaction est définie comme étant *une séquence d'opérations transformant la base de données d'un état cohérent en un autre état cohérent* [Del 82].

Les principales propriétés d'une transaction sont l'atomicité, l'intégrité et la persistance. L'intégrité d'une transaction découle de sa définition puisqu'on passe d'un état cohérent à un autre état cohérent. Les règles de cohérence peuvent être violées pendant la transaction mais elles doivent être respectées à la fin de la transaction. L'atomicité signifie que l'ensemble des opérations constituant la transaction est considéré comme un tout, soit toutes les opérations sont exécutées et la transaction est dite valide (commit ou validate), soit, au contraire, ces opérations sont ignorées et la transaction est dite annulée (abort). L'état de la base après la validation d'une transaction est persistant.

Le gestionnaire de transactions dispose généralement de trois primitives permettant de maintenir la cohérence de la base de données en assurant les propriétés des transactions: **beginT**, **validate**, **abort**. La primitive **beginT** permet de débuter une transaction. La primitive **validate** permet de valider la transaction en rendant persistant l'état de la base après son exécution. La primitive **abort** permet d'annuler toute modification effectuée sur la base à partir du début de la transaction. Les approches utilisées classiquement supposent que les transactions sont *atomiques, de courte durée et ne concernent que peu d'objets de la base*, elles correspondent à ce qu'on appelle des transactions simples.

Dans le cadre d'un SGBD orienté-objet, les utilisateurs manipulent la base de données via des programmes d'applications contenant des messages envoyés aux objets. Ces messages sont regroupés dans des transactions qui assurent les propriétés d'intégrité, d'atomicité et de persistance.

Par exemple, dans [Mar 91], le modèle de transactions pris en compte définit une transaction comme étant *un ensemble de messages envoyés à la base de données durant un programme d'application. Cet ensemble possède les propriétés suivantes : intégrité, atomicité, isolation et persistance.* Le début et la fin de la transaction sont réalisés par les commandes **Begin Transaction** et **End Transaction.** Le système de contrôle considéré permet de décider si une transaction doit être validée ou non. Pour assurer l'intégrité de la base, il est possible de spécifier des contrôles à effectuer lors de la réception d'un message par un objet, après l'exécution des méthodes associées à ce message et en fin de transaction, la figure 1 (extraite de [Mar 91]) montre le schéma général des contrôles effectués.

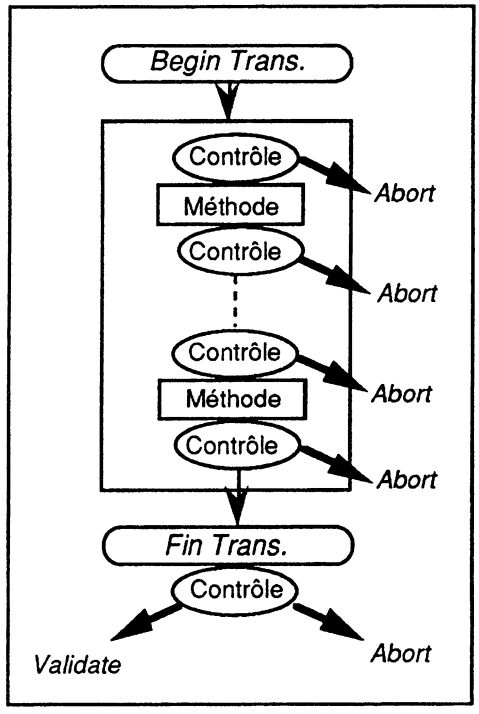


Figure-1

La vision classique des transactions se révèle insuffisante et inadaptée dans le cadre d'une application génie logiciel. En effet, les opérations en génie logiciel sont de longue durée, coûteuses et manipulent un nombre important d'objets. Pour cela, la notion d'intégrité d'une transaction est remise en cause et l'incohérence des données peut être dans certains cas tolérée et gérée. Il n'est pas possible de rejeter un ensemble d'opérations coûteuses si elles produisent un objet incohérent, comme c'est

le cas dans l'exemple précédent. Il faut différencier ici les cas d'incohérence tolérée des cas d'incohérence non-tolérée de façon à pouvoir disposer, à la fin d'une transaction, d'états d'incohérence tolérés.

Pour ces raisons, des modèles plus puissants doivent être utilisés tels que les transactions imbriquées.

3.2. MODELE DE TRANSACTIONS CONSIDERE

Dans le modèle de transactions imbriquées [Eas 82, Mos 82], on manipule deux types de transactions, les transactions de haut niveau et les sous-transactions ou transactions filles. Une transaction de haut niveau peut être composée de plusieurs sous-transactions, elles mêmes peuvent être composées de sous-transactions. Une transaction de haut niveau obéit aux propriétés d'atomicité et de persistance. Par contre, les effets d'une transaction fille ne peuvent pas être considérés comme persistants avant sa validation ainsi que la validation de toutes ses transactions ancêtres. Lors de l'annulation d'une transaction, ses effets et les effets de toutes ses filles sont ignorés.

Les transactions imbriquées offrent la possibilité de distribuer le travail entre les sous-transactions, de n'abandonner qu'un nombre limité de sous-transactions et d'exécuter les sous-transactions en parallèle dans la mesure du possible.

Le modèle de transactions imbriquées constitue la base pour l'exécution du mécanisme de déclencheurs que nous proposons.

La création d'une transaction dans notre proposition correspond à l'*activation* d'une méthode. Comme nous allons voir par la suite, l'activation d'une méthode peut impliquer l'activation d'une autre méthode pour effectuer certains contrôles. La transaction correspondant à l'activation de la dernière peut être une sous-transaction de celle correspondant à l'activation de la première. Les contrôles de l'intégrité et de la cohérence peuvent être effectués avant et après l'exécution des méthodes. Ils peuvent aussi être effectués après la validation de la transaction dans des transactions indépendantes. Le schéma général des transactions est donné par la figure 2.

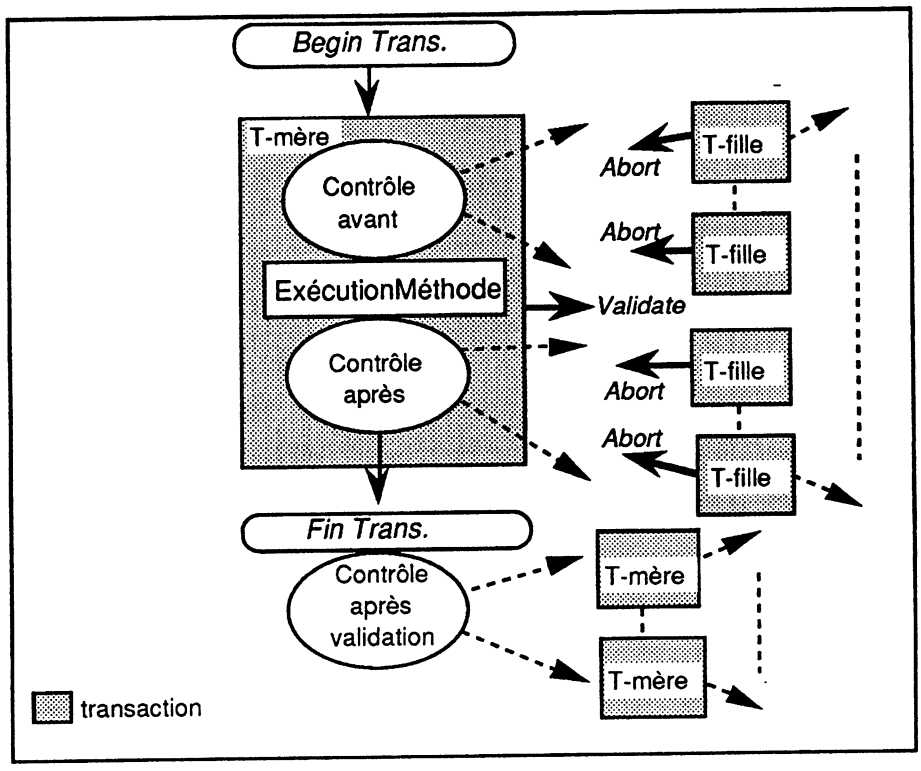


Figure-2

Il est clair dans cette figure que les effets d'une sous-transaction ne peuvent pas être considérés comme persistants qu'après la validation des transactions ancêtres (validate), et que chacune des sous-transactions peut annuler les effets de la transaction mère (abort).

Chaque transaction prend ce que nous appelons une *valeur d'état* liée au résultat de la méthode. Si la méthode n'a pas été exécutée pour des raisons liées aux contrôles avant, la valeur d'état de la transaction devient *Refused*, si l'exécution de la méthode réussit, la transaction prend *Succeeded* comme valeur d'état. Dans le cas contraire, son état prend la valeur *Failed*. Les notions de réussite, d'échec et de refus d'une méthode vont être éclairées dans les paragraphes 5.2 et 5.3.

Pour gérer les transactions nous définissons plusieurs primitives :

BeginTrans : est utilisée pour créer une transaction lors de l'activation d'une méthode.

Abort : est utilisée pour annuler une transaction dont l'état est *Refused* ou *Failed*. L'annulation d'une transaction a comme résultat d'ignorer tous les effets de cette transaction et d'annuler aussi toutes les transactions ancêtres.

Validate : est utilisée pour terminer une transaction de haut niveau dont l'état est *Succeeded* et pour rendre persistant les effets de cette transaction. Elle termine aussi toutes ses filles.

4. LES METHODES

D'une manière générale, le mécanisme de déclencheurs est utilisé pour gérer le comportement des objets, mais aussi pour contrôler ce comportement. Les méthodes sont ici des règles (événement condition-action) qui jouent le rôle de déclencheurs. Avec ces déclencheurs, le concepteur (administrateur d'un projet) a la possibilité de spécifier, d'une façon déclarative, les situations auxquelles les objets doivent réagir, le comportement des objets dans ces situations et les contrôles sur ce comportement. Lors du déclenchement d'une méthode, les contrôles possibles sont ceux effectués avant son exécution, juste après son exécution mais dans le contexte de la transaction correspondant à cette exécution et également après la validation de cette transaction.

Les situations auxquelles un objet doit réagir par le déclenchement d'une méthode sont exprimées par ce qu'on appelle le **contexte de déclenchement** de la méthode (voir Figure 3). En effet, ces situations, appelées aussi **événements**, peuvent correspondre, ou bien, à un envoi explicite d'un message, ou bien, à des situations particulières détectées implicitement et indiquées par le système. Ces dernières sont liées à l'activation des méthodes, ce qui veut dire, à la création de transactions. La détection de telles situations par le système peut impliquer le déclenchement d'une ou plusieurs méthodes.

Les méthodes déclenchées ne sont pas forcément activées tout de suite. Elles peuvent être mises en attente pour l'activation. Le moment de l'activation d'une méthode par rapport à son déclenchement est déterminé par son **contexte d'activation**. Celui-ci contient des informations concernant le moment de l'activation de la méthode, soit par rapport à la transaction déclencheur, soit par rapport aux autres méthodes déclenchées en même temps.

Les contrôles à effectuer avant l'exécution d'une méthode représentent ce que nous appelons le **contexte de l'exécution** de la méthode. Ils ont pour objectif de vérifier la validité du message reçu par l'objet et la validité de l'état de la base pour l'exécution de la méthode.

L'exécution d'une méthode consiste à évaluer sa partie **Action**. Cette partie détermine le comportement de l'objet proprement dit dans la situation responsable de son déclenchement.

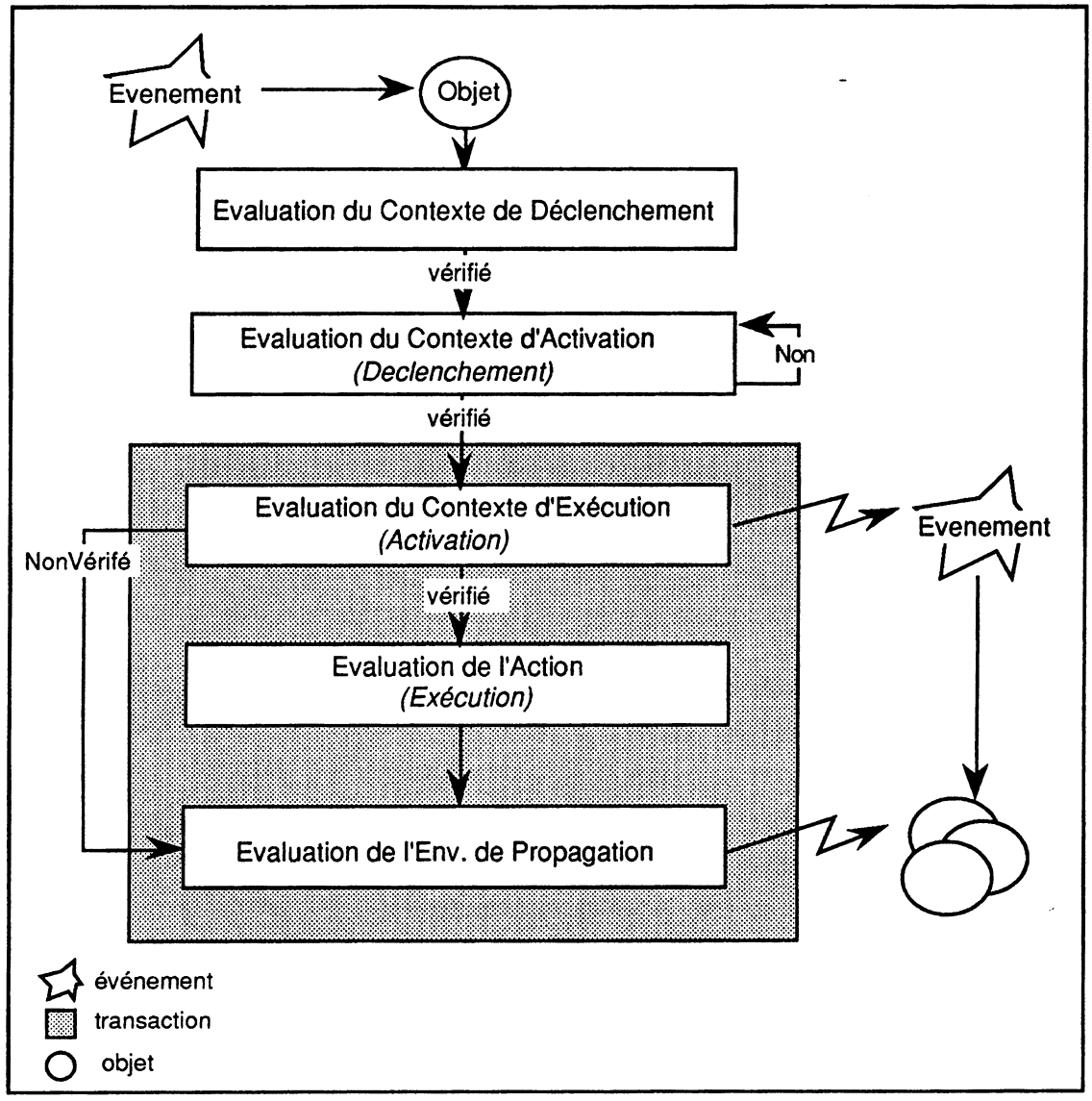


Figure-3

Après l'exécution de la méthode d'autres contrôles peuvent être effectués sur l'objet même ou sur d'autre objets. Le but est, d'une part, de vérifier la validité de l'état de la base après l'exécution de la méthode, et, d'autre part, d'exécuter les actions nécessaires pour maintenir l'intégrité et la cohérence de la base, ainsi que, pour propager les effets de bord de la méthode. Cela correspond à l'exécution d'autres méthodes associées à l'objet lui-même ou à d'autres objets dans la base. Les objets responsables des contrôles après l'exécution d'une méthode représentent l'**environnement de propagation** de la méthode et sont spécifiés par la méthode elle-même d'une façon déclarative. Ainsi, la propagation des effets de l'exécution d'une méthode est limitée, et ne concerne pas toute la base. Vu le grand nombre d'objets manipulés et la nature des opérations exécutées dans un environnement

génie logiciel, ceci a des conséquences très importantes du point de vue efficacité et performance.

4.1. DEFINITION DES METHODES

Les méthodes dans notre formalisme sont donc définies par les différents éléments décrits dans la section précédente : **contexte de déclenchement, contexte d'activation, contexte d'exécution, Action** et **environnement de propagation.**

Si nous considérons un ensemble $\mathcal{S}$ fini de symboles servant pour nommer les méthodes et l'ensemble $\mathcal{M}$ des méthodes, une méthode m$\in \mathcal{M}$ est définie ainsi:

m = ($\mathcal{I}$dent, $\mathcal{C}$déclenchement, $\mathcal{C}$activation, $\mathcal{C}$exécution, $\mathcal{A}$ction, $\mathcal{E}$propagation) tels que :

- $\mathcal{I}$dent est un identificateur défini par un sélecteur$\in \mathcal{S}$ et le type auquel est associée la méthode.
- $\mathcal{C}$déclenchement, est le **contexte de déclenchement** de la méthode.
- $\mathcal{C}$activation, est le **contexte d'activation** de la méthode.
- $\mathcal{C}$exécution, est le **contexte d'exécution** de la méthode.
- $\mathcal{A}$ction, est l'**Action** de la méthode.
- $\mathcal{E}$propagation, est l'**environnement de propagation** de la méthode.

La syntaxe de la déclaration d'une méthode est :

((Method (<Sel> <Type>))
<$\mathcal{C}$déclenchement>
<$\mathcal{C}$activation>
<$\mathcal{C}$exécution>
<$\mathcal{A}$ction>
<$\mathcal{E}$propagation>)

Nous donnons brièvement la sémantique associée aux méthodes, cette sémantique va être explicitée dans les paragraphes suivants :

Une méthode est déclenchée lorsque l'on se trouve dans une *situation* analogue à celle décrite dans son *contexte de déclenchement.* Les méthodes qui seront ainsi sélectionnées sont activées à des moments déterminés par leur *contexte d'activation.* Pour chaque méthode activée, une transaction est créée, dans laquelle le contexte d'exécution est évalué. Si le *contexte d'exécution* est vérifié, l'action de la méthode est exécutée. L'exécution de la méthode consiste à évaluer son action. A son tour,

l'activation de la méthode met la base dans une autre situation à laquelle les objets spécifiés dans l'*environnement de propagation* doivent réagir en déclenchant d'autres méthodes.

Pour contrôler les méthodes, les opérateurs *Trigger*, *Activate*, *Execute* sont définis sur l'ensemble de méthodes $\mathfrak{M}$:

• *Trigger* correspond au déclenchement d'une méthode. Le déclenchement d'une méthode se fait lors de la vérification de son *contexte de déclenchement.* Plusieurs méthodes peuvent être déclenchées et sont alors candidates à l'activation. Les contextes d'activation des méthodes déclenchées sont évalués, et les méthodes dont le contexte d'activation est verifié sont activées.

• *Activate* correspond à l'activation d'une méthode. L'activation d'une méthode se fait si son *contexte d'activation* est vérifié. Elle se déroule dans une transaction créée à cet effet et dans laquelle sera évalué le contexte d'exécution. Une seule méthode peut être activée à la fois, sauf si différentes méthodes peuvent être activées en parallèle.

• *Execute* correspond à l'exécution de la méthode, c'est-à-dire, à l'évaluation de son *Action* dans la transaction correspondant à l'activation de la méthode. L'exécution d'une méthode ne se fait que si son contexte d'exécution est satisfait.

Dans les paragraphes suivants, nous présentons les différents éléments qui servent pour la définition et le contrôle des méthodes.

4.2. CONTEXTE DE DECLENCHEMENT

Une méthode peut être déclenchée sur un objet de deux façons différentes : soit par l'envoi direct d'un message à l'objet, soit lors de l'apparition de certaines situations spécifiées dans le contexte de déclenchement de la méthode.

L'envoi de message correspond à ce que nous appelons des **événements externes.** Ces événements externes sont le seul moyen de communication entre le monde extérieur (utilisateur et applications) et la base. Par contre, l'ensemble de situations ou d'indicateurs spécifiés dans le contexte de déclenchement d'une méthode sont appelés **événements internes.** Ces événements internes sont crées par le système et signalés aux objets concernés.

4.2.1. Evénements Externes

L'envoi de messages est le seul moyen de communication entre les objets. Nous les appelons **événements externes** parce qu'ils ne sont pas créés par le système.

L'envoi d'un message à un objet est un indicateur qui nécessite la réaction de l'objet par le déclenchement de la méthode désignée par ce message.

L'envoi d'un message à un objet de nom *Obj* se fait en utilisant la primitive **SendMessage** :

(**SendMessage** *Obj Sel ListeParamEffs*)

où :

- *Sel* ∈ $\mathcal{S}$, correspond au nom de la méthode.
- *Obj* ∈ $\mathcal{O}$, est l'objet auquel est envoyé le message ($\mathcal{O}$ est l'ensemble d'objets défini dans le chapitre 4).
- *ListeParamEffs* est la liste de paramètres effectifs à passer à la méthode lors de son activation.

L'envoi de ce message a pour effet de déclencher la méthode de nom *Sel.*

Par exemple, pour éditer la fonction OpenWind, l'utilisateur envoie le message suivant :

(**SendMessage** 'OpenWind 'Editer)

L'envoi de ce message a pour effet de déclencher la méthode Editer sur l'objet OpenWind.

4.2.2. Evénements Internes

La création d'un **événement interne** est provoquée par *l'activation d'une méthode* ou (en d'autres termes) par *la création d'une transaction.* Nous appelons la méthode et l'objet sur lequel cette méthode a été activée les origines de l'événement. Ces événements sont appelés internes puisqu'ils sont créés par le système et ne viennent pas de l'extérieur.

Un **événement interne** est défini par le sélecteur de la méthode, le nom de l'objet origine de l'événement, et la liste des paramètres effectifs utilisée par la méthode. On désigne par $\mathcal{E}$vent l'ensemble de tous les événements pouvant être créés par le système. Chaque événement E de cet ensemble peut être défini ainsi :

E = (Sel, Obj, ListeParamEffs)

où Sel∈ $\mathcal{S}$ et Objet∈ $\mathcal{O}$, ListeParamEffs est une liste de paramètres effectifs utilisés lors de l'activation de la méthode Sel sur l'objet Obj.

Par exemple, l'activation de la méthode Editer par l'envoi du message précédent a pour effet la création d'un événement interne :

(Editer, OpenWind)

Les événements internes sont créés par le système et sont signalés aux objets spécifiés par l'**environnement de propagation** de la méthode origine de l'événement (voir 4.7).

4.2.3. Définition du Contexte de Déclenchement d'une Méthode

Les événements capables de déclencher une méthode sont décrits dans son contexte de déclenchement. Le contexte de déclenchement d'une méthode est défini par un sélecteur de méthode, le type d'objets sur lequel cette méthode est définie, et une liste de paramètres formels à remplacer par les paramètres effectifs utilisés lors de l'activation de la méthode.

Le **contexte de déclenchement** d'une méthode est donc défini de la façon suivante:

$\mathfrak{C}$déclenchement = (Sel, Type, ListeParamFormels)

où Sel désigne la méthode dont l'activation est à l'origine de l'événement, Sel$\in \mathfrak{S}$, Type désigne le type de l'objet sur lequel Sel est activée, Type$\in \mathfrak{T}$ n, et ListeParamFormels est une liste de symboles désignant les paramètres formels utilisés par la méthode Sel.

Le doublet (Sel, Type) définit un ensemble d'événements internes correspondant aux occurrences d'activation de la méthode Sel sur tous les objets de type Type. (Sel, Type) définit ainsi un **type d'événements**.

Le contexte de déclenchement d'une méthode est spécifié, au niveau de la définition de la méthode en utilisant la syntaxe suivante :

<$\mathfrak{C}$déclenchement> ::= **(On** <TypeEvent>)
(WithArgs <ListeParamFormels>)
<TypeEvent> ::= (Sel Type)
<ListeParamFormels> ::= **(self origin** (<Paramètre>*)

où **self** est un paramètre formel spécial utilisé pour désigner l'objet sur lequel la méthode est déclenchée, **origin** est un paramètre formel spécial utilisé pour désigner l'objet origine de l'événement déclencheur et <Paramètre> est un paramètre au sens langage de programmation.

Les méthodes ainsi définies sont déclenchées par un événement de la forme (Sel Obj) où Obj est un objet de type Type. Par exemple, la méthode Compiler suivante définie sur les objets de type Module est déclenchée par l'événement (Editer OpenWind) produit à la suite de l'activation de la méthode Editer :

((**Method** (Compiler Module)
(**On** (Editer Fonction)) (**WithArgs** (self origin))...

4.3. CONTEXTE D'ACTIVATION D'UNE METHODE

Les méthodes déclenchées ne sont pas forcément activées immédiatement, le moment de l'activation d'une méthode est déterminé par le **contexte d'activation**. Le contexte d'activation d'une méthode est défini par deux informations : un mode d'activation et une priorité d'activation. Nous pouvons définir le contexte d'activation ainsi :

$\mathfrak{C}$activation= (Mode, Priorité)

le mode d'activation détermine le moment de l'activation de la méthode par rapport à la transaction origine de l'événement déclencheur, et la priorité définit l'ordre de déclenchement de la méthode par rapport aux autres méthodes ayant le même mode d'activation.

La syntaxe utilisée pour exprimer le contexte de déclenchement est la suivante :

<$\mathfrak{C}$activation>::= **(OfMode** <Mode>**)(WithPriority** <Priorité>**)**

Nous allons définir, plus précisément, dans les paragraphes suivants les modes d'activation et la notion de priorité.

4.3.1. Modes d'Activation

Les méthodes déclenchées ne sont pas forcément activées immédiatement. Le moment d'activation d'une méthode, par rapport à la transaction origine, dépend de la nature des contrôles voulus par l'exécution de la méthode (voir Figure 4).

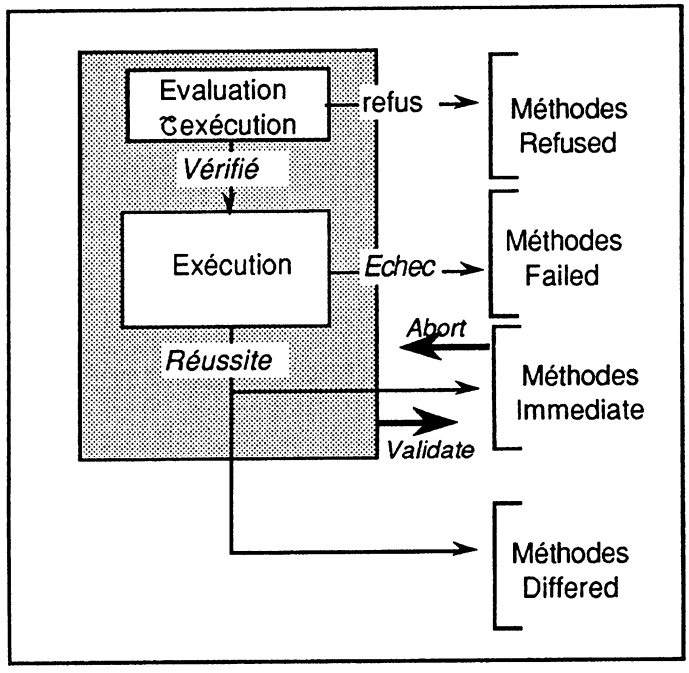


Figure-4

En effet, les méthodes déclenchées peuvent être activées dans une transaction fille de la transaction correspondant à l'événement déclencheur et juste avant sa validation, elles sont dites méthodes à activation **Immédiate.** Elles peuvent aussi être activées après la validation de la transaction de l'événement déclencheur et dans une transaction séparée, ce sont les méthodes à activation **Différée.** Certaines méthodes, dites les méthodes **Refused,** sont activées lors d'un refus d'exécution de la méthode origine de l'événement déclencheur. Ceci correspond à la notion d'exception dans les langages de programmation. D'autres méthodes sont activées lors d'un échec dans l'exécution de cette méthode, ces dernières sont les méthodes **Failed** (le refus et l'échec de l'exécution d'une méthode seront présentés dans §5.2 et §5.3) .

Ainsi, nous distinguons quatre modes d'activation possibles :

Mode $\in$ {**Immediate, Deferred, Refused, Failed**}.

Chacun de ces différents modes réalise des contrôles appropriés que nous allons détailler.

4.3.1.1. Les méthodes à activation immédiate

Les contrôles exprimés par des méthodes dont le mode est **Immediate** ont pour objectif d'examiner l'état de la base à la fin d'une transaction et d'analyser ses conséquences. Ces contrôles permettent d'empêcher les cas d'incohérence non-tolérée, voire d'annuler la transaction origine de l'événement.

L'activation des méthodes **Immédiates** se fait après l'exécution de la méthode origine, à la fin de la transaction correspondant et avant la validation de cette transaction. La réussite de l'exécution de ces méthodes conditionne la validation de la transaction.

Exemple:

```
((Method (Compiler Module))
 (On (Editer Fonction)) (WithArgs (self origin))
 (OfMode Immediate) ...
```

La méthode Compiler peut être activée à la fin de la transaction correspondant à l'événement (Editer, OpenWind) et avant sa validation. La validation des modifications effectuées sur la fonction OpenWind par "Editer" est dépendant de la réussite de la compilation des modules qui l'utilisent.

4.3.1.2. Les méthodes à activation différée

Les contrôles exprimés par des méthodes **Deferred** ont pour objectif d'exécuter certaines actions ou de modifier certains objets dans le but de maintenir la base dans un état cohérent. Ils sont aussi utilisés pour avertir l'utilisateur de certains cas d'incohérence dans la base considérés comme tolérés. Toutefois, ils ne sont pas capables d'annuler les effets de la transaction correspondante.

Les méthodes **Deferred** sont activées à la fin de la transaction origine de l'événement déclencheur mais après la validation de cette transaction. La réussite ou l'échec de telles méthodes n'a pas de conséquences sur la transaction. En effet, un échec dans l'exécution d'une méthode différée laisse la base dans un état d'incohérence tolérée.

Exemple :

```
((Method (ModifConc DocConcFonc))
 (On (Editer Fonction)) (WithArgs (self origin))
 (OfMode Deferred) ...
```

La méthode ModifConc est définie par le type DocConcFonc qui représente les documents de conception. Cette méthode peut être déclenchée par l'événement (Editer, OpenWind). Elle est activée, après la validation de la transaction correspondant à l'édition de OpenWind, dans une transaction indépendante. La réussite, l'échec ou le refus de ModifConc n'a pas donc de conséquence sur la transaction de Editer.

4.3.1.3. Les méthodes Failed

Les contrôles sont exprimés par les méthodes **Failed** sont effectués à la suite de l'échec d'une transaction et dans des transactions indépendantes de celle-ci. Les méthodes de ce type sont utilisées, par exemple, pour avertir l'utilisateur de l'échec d'une transaction.

Exemple :

```
((Method (EchecComp Module))
 (On (Compiler Module)) (WithArgs (self origin))
 (OfMode Failed) ...
```

La méthode EchecComp peut être déclenchée si la compilation d'un module échoue.

4.3.1.4. Les méthodes Refused

Dans le cas d'un contexte d'exécution non vérifié pour une méthode activée, l'exécution de la méthode est refusée. Certains contrôles doivent alors être

effectués. Ces contrôles sont exprimés par les méthodes **Refused** activées dans des transactions indépendantes de la transaction déclencheur. Ces méthodes peuvent être utilisées, par exemple, pour avertir l'utilisateur de la cause d'un refus d'exécution d'une méthode.

Exemple :

```
((Method (RefuserModifConc DocConcFonc))
 (On (ModifConc DocConcFonc)) (WithArgs (self origin))
 (OfMode Refused) ...
```

La méthode RefuserModifConc est activée si les modifications dans un document de conception ne sont pas effectuées à cause d'un refus (lorsque le contexte d'exécution de la méthode ModifConc est non vérifié).

4.3.2. Priorité d'Activation

Lors de la production d'un événement plusieurs méthodes de même mode peuvent être déclenchées. Ces méthodes activées dans un certain ordre défini par leur priorité.

La *priorité* d'une méthode est un entier compris entre 1 et 100 donné lors de sa définition. Si la priorité pour une méthode n'est pas donnée, par défaut elle est considérée de 1, ce qui veut dire, qu'elle est la première à être traitée. Dans le cas où plusieurs méthodes ayant la même priorité sont déclenchées, ces méthodes sont exécutées dans un ordre quelconque, voire en parallèle.

4.4. CONTEXTE D'EXECUTION D'UNE METHODE

Nous définissons le *contexte d'exécution* d'une méthode comme étant *une expression représentant une condition d'exécution dont l'évaluation à une valeur non nil autorise l'exécution de la partie action de la méthode.* L'évaluation de la condition se fait en utilisant les paramètres effectifs venant de l'évaluation du contexte de déclenchement.

D'une manière générale, le contexte d'exécution permet une exécution conditionnelle du corps de la méthode. Il vérifie la validité de l'état de la base avant l'exécution de la méthode en définissant les situations dans lesquelles l'exécution de la méthode est autorisée. Il restreint ainsi le contexte de déclenchement de la méthode. Lorsqu'une méthode est activée, si le contexte d'exécution est vérifié la méthode est exécutée. Dans le cas contraire, le contexte d'exécution est dit non vérifié et l'exécution de la méthode est *refusée.*

Le contexte d'exécution d'une méthode permet d'exprimer d'une manière déclarative des contrôles à effectuer lors de son activation et avant son exécution. Ces contrôles sont de natures différentes :

- Contrôles sur les paramètres effectifs passés à la méthode. Dans le cas où l'activation de la méthode est effectuée à la suite d'un envoi de message à l'objet, les paramètres passés par le message doivent vérifier certaines conditions. Dans le cas où ces conditions ne sont pas vérifiées le message, c'est-à-dire l'événement, est rejeté et, par conséquence l'exécution de la méthode est refusée et la transaction correspondante est alors annulée.
- Contrôles sur l'état de l'objet avant l'exécution de la méthode. L'état de l'objet doit vérifier certaines contraintes dont la validation autorise l'exécution de la méthode. Ces contraintes portent sur les valeurs des attributs de l'objet lors de la réception de l'événement qui déclenche la méthode.
- Contrôles sur l'état de la base avant l'exécution de la méthode. La base doit être aussi dans un état permettant l'exécution de la méthode. Cet état peut être exprimé par des contraintes qui portent sur les états des autres objets de la base.
- Déclenchement d'autres méthodes. L'exécution de la méthode peut nécessiter l'exécution d'autres méthodes sur l'objet même ou sur d'autres objets de la base dont la réussite conditionne l'exécution de la méthode en question. Ceci est fait en envoyant des messages pour l'exécution de ces méthodes.
- Raffinement du contexte de déclenchement. L'évaluation de la condition d'exécution peut avoir comme effet l'évaluation de certaines variables pour les passer à la méthode, lors de son exécution, en plus des paramètres effectifs de l'événement.

Le contexte d'exécution est exprimé au niveau de la définition d'une méthode dans la clause **if** de la façon suivante:

<ℭactivation> ::= **(If** <ExprCondition>**)**

<ExprCondition> est une expression évaluée lors de l'activation de la méthode, elle est considérée comme satisfaite si elle est évaluée à une valeur non nil. La grammaire correspondant à l'expression de la condition est la suivante :

<ExprCondition> ::= <Forme> | (<OpBin> <Forme><Forme>) | (OpUn <Forme>)

<Forme> ::= <FormeLoops> | <Message>
<Message> ::= **(SendMessage** Obj Sel <ListeParamEffs>)

où Obj est le nom d'un objet, Sélecteur est un élément de $\mathcal{S}$. <OpBin> et <OpUn> sont des opérateurs utilisés pour définir des prédicats. Pour des détails concernant les expressions (Formes) et les opérateurs utilisés en Loops voir [Loo 88, Xer 85].

4.5. ACTION

L'**action** d'une méthode représente *la partie code à exécuter si le contexte d'exécution est vérifié.* Elle constitue le corps de la méthode et spécifie alors les traitements associés à un événement. L'action est une expression dont l'évaluation retourne une valeur ou un identificateur d'objet. L'évaluation de l'action se fait en utilisant les paramètres effectifs portés par l'événement et, éventuellement, les variables évaluées lors de l'évaluation du contexte d'exécution.

Les traitements pouvant être exprimés par l'action peuvent inclure:

- des envois de messages à l'utilisateur pour signaler un état d'incohérence ou pour demander la réaction de l'utilisateur.
- des envois de messages pour l'exécution d'autres méthodes dans le but de contrôler la cohérence de la base et de propager les effets de bord de l'événement qui a activé la méthode. Ceci est fait en utilisant la primitive **SendMessage**.
- une demande d'enregistrement de l'événement qui déclenche la méthode pour un traitement ultérieur en utilisant la primitive **Record**. Cette primitive est utile pour pouvoir dans certains cas tolérer temporairement des états d'incohérence en retardant le traitement d'un événement. Les événements enregistrés sur un objet sont traités à la demande de l'utilisateur qui utilise pour cela la primitive **Make**.
- la demande d'annulation de la transaction qui a produit l'événement. Ceci est fait en utilisant la primitive **Undo**. Ainsi chaque objet atteint par les effets de bord d'une méthode est capable de la défaire et d'annuler ainsi ses effets.
- des traitements liés aux données (ajout, mise-à-jour, etc).

L'action d'une méthode est exprimée au niveau de sa définition dans la clause **do**:

<action> ::= **(Do** <ExprAction>)

L'expression de l'action est spécifiée en utilisant une extension du langage Loops par les primitives que nous définissons pour le contrôle du mécanisme. La grammaire correspondant à l'expression de l'action est la suivante :

<ExprAction> ::= <Forme> | (OpProg <Forme>+)
<Forme) := <FormeLoops> | <Message> | **Undo** | **Record**

où <OpProg> définit des opérateurs utilisés pour la définition de fonctions tels que PROG, PROG1, etc (voir [Xer 85]); les <FormesLoops> sont des expressions Loops (voir [Loo 88]); Undo, Record sont les primitives définies ci-dessus.

Par exemple, la méthode Editer peut être définie de la façon suivante:

```
((Method (Editer Fonction))
 (WithArgs (self origin))
 (OfMode Immediate)
 (Do (DF self))...
```

ce qui consiste à éditer la fonction recevant le message Editer.

4.6. ENVIRONNEMENT DE PROPAGATION

L'exécution d'une méthode peut avoir des conséquences sur l'objet auquel est attachée la méthode, ou sur d'autres objets de la base. Ces objets doivent réagir, soit en exécutant des actions pour maintenir la cohérence de la base, soit en annulant les effets de la méthode. Pour cela, l'événement produit lors de l'activation de la méthode est signalé aux objets de la base pour qu'ils réagissent. Vu le nombre important d'objets, et la nature des opérations effectuées dans un environnement génie logiciel, la réaction de tous les objets de la base pose un problème de performance et d'efficacité. Nous avons donc intérêt à *limiter la propagation des événements sur une partie de la base* en limitant le nombre d'objets informés par la production d'un événement, voire empêcher cette propagation.

L'**environnement de propagation** d'une méthode spécifie donc *la liste des objets à informer par la production de l'événement* correspondant à l'activation de la méthode. Ainsi, cet événement n'est signalé qu'aux objets spécifiés par l'**environnement de propagation**, ce qui limite la partie de la base capable de réagir après l'exécution de la méthode. Dans le cas où l'environnement de propagation est inexistant pour une méthode, l'événement produit par la méthode est rejeté et aucune réaction n'est entreprise par le système. C'est le cas des méthodes dont l'exécution n'ayant pas des effets de bord.

A travers l'environnement de propagation, plusieurs contrôles peuvent être effectués après l'évaluation de l'action d'une méthode:

- Des contrôles à effectuer sur l'objet responsable de la production de l'événement comme, par exemple, la validité de son état après l'exécution de la méthode. Cet objet est capable d'exécuter d'autres méthodes et, éventuellement de défaire la transaction provoquant l'événement. Ceci peut être fait en utilisant le mot-clé **self** dans l'environnement de propagation.
- Des contrôles sur les objets parents de l'objet en question. Pour cela le mot-clé **Parents** est utilisé dans l'environnement de propagation, dans ce cas, les parents de l'objet peuvent réagir à l'événement.
- Des contrôles sur des objets liés, par des relations, à l'objet responsable de la production de l'événement. Nous utilisons pour cela le nom de la relation dont le destinataire va réagir. Ainsi tous les objets liés par la relation dont le nom est donné par l'environnement de propagation peuvent réagir et même annuler la transaction ayant provoqué l'événement.
- Des contrôles sur tous les objets liés à l'objet responsable de la production de l'événement. Cela se fait en utilisant le mot-clé **AllRelated**, ce qui veut dire que l'événement doit être signalé à tous les objets liés par une relation de façon à ce qu'ils puissent réagir.
- Des contrôles sur tous les objets liés à l'objet responsable de la production de l'événement par une relation de composition. Cela se fait en utilisant le mot-clé **AllComponents**, ce qui veut dire que l'événement doit être signalé à tous les composants de l'objet pour qu'ils puissent réagir à l'événement produit.
- Des contrôles sur des objets non liés à l'objet responsable de la production de l'événement. Ces objets sont spécifiés dans l'environnement de propagation par leur nom et sont donc notifiés par la production de l'événement.
- Des contrôles sur toute la base. Cela se fait en utilisant le mot-clé **All,** qui indique que tous les objets de la base doivent être notifiés par la production de l'événement. Ainsi tous les objets ayant des méthodes correspondant à cet événement peuvent réagir.

La syntaxe de l'environnement de propagation d'une méthode est définie par les règles suivantes :

<êpropagation> ::= (**Notify** <ListeSpécObjets>)
<ListeSpécObjets> ::= (<SpécObjet>$^{+}$)

<SpécObjet> ::= **self** | **All** | **AllRelated** | **AllComponents** | **Parents** | Relation | Objet

où Objet est un élément de $\mathcal{O}$, Relation est un attribut de référence ou de composition associé à l'objet sur lequel la méthode est exécutée.

Par exemple, la méthode Editer peut être complétée de la façon suivante:

```
((Method (Editer Fonction))
 (WithArgs (self origin))
 (OfMode Immediate)
 (Do (DF self))
 (Notify (Parents UtiliséPar Conception))...
```

ce qui consiste à notifier les parents de la fonction éditée, les modules liés par une relation UtiliséPar et le document de conception de la fonction lié par une relation Conception.

4.7. SYNTAXE GENERALE DES METHODES

La syntaxe de la déclaration d'une méthode devient donc la suivante:

```
((Method (<Sel> <Type>))
 [(On (<Sel> <Type>))] (WithArgs (self origin <ListeParamFormels>))
 (OfMode <Mode>) (WithPriority <Priorité>)
 [(If <ExprCondition>)]
 (Do <ExprAction>)
 [(Notify <SpécObjs>)])
```

Par défaut, nous considérons que les méthodes ont un mode **immediate** et une priorité 1.

5. LE MECANISME DE DECLENCHEURS

Le mécanisme de déclencheurs est responsable de la gestion du déclenchement, de l'activation, de l'exécution et de la propagation des effets des méthodes. Il pilote aussi la création, l'annulation et la validation des transactions. Il existe deux façons pour enclencher ce mécanisme : l'envoi explicite de messages aux objets (événements externes), et l'activation des méthodes donnant lieu à la création d'événements internes. Dans la suite nous allons décrire le fonctionnement du mécanisme en utilisant l'exemple suivant :

```
((Method (Editer Fonction)) (WithArgs (self origin))
 (Do (DF self))
 (Notify (Parents UtiliséPar Conception)))
```

```
((Method (Compiler Module))
 (On (Editer Fonction)) (WithArgs (self origin))
 (OfMode Immediate) (WithPriority 1)
 (Do (COMPILE self)))

((Method (ModifConc DocConcFonc))
 (On (Editer Fonction)) (WithArgs (self origin))
 (OfMode Deferred)
 (If (EQUAL (ASKUSER NIL NIL "Voulez-vous modifier le document de
                Conception ?") 'Y))
 (Do (<- self Display)))
```

Les trois méthodes définies ci-dessus modélisent l'exemple présenté dans l'introduction de ce chapitre : à la suite de la modification d'une fonction (édition), tous les modules utilisant cette fonction doivent être compilés. La réussite de la compilation de ces modules conditionne la prise en compte des modifications effectuées sur la fonction. Par ailleurs, le document de conception associé à la fonction doit être modifié si l'utilisateur le décide. Effectuée ou pas, la modification de ce document ne doit pas avoir de conséquence par rapport aux modifications effectuées sur la fonction.

En considérant cet exemple, la figure 5 montre le déroulement du mécanisme de déclenchement que nous allons expliquer dans les paragraphes qui suivent.

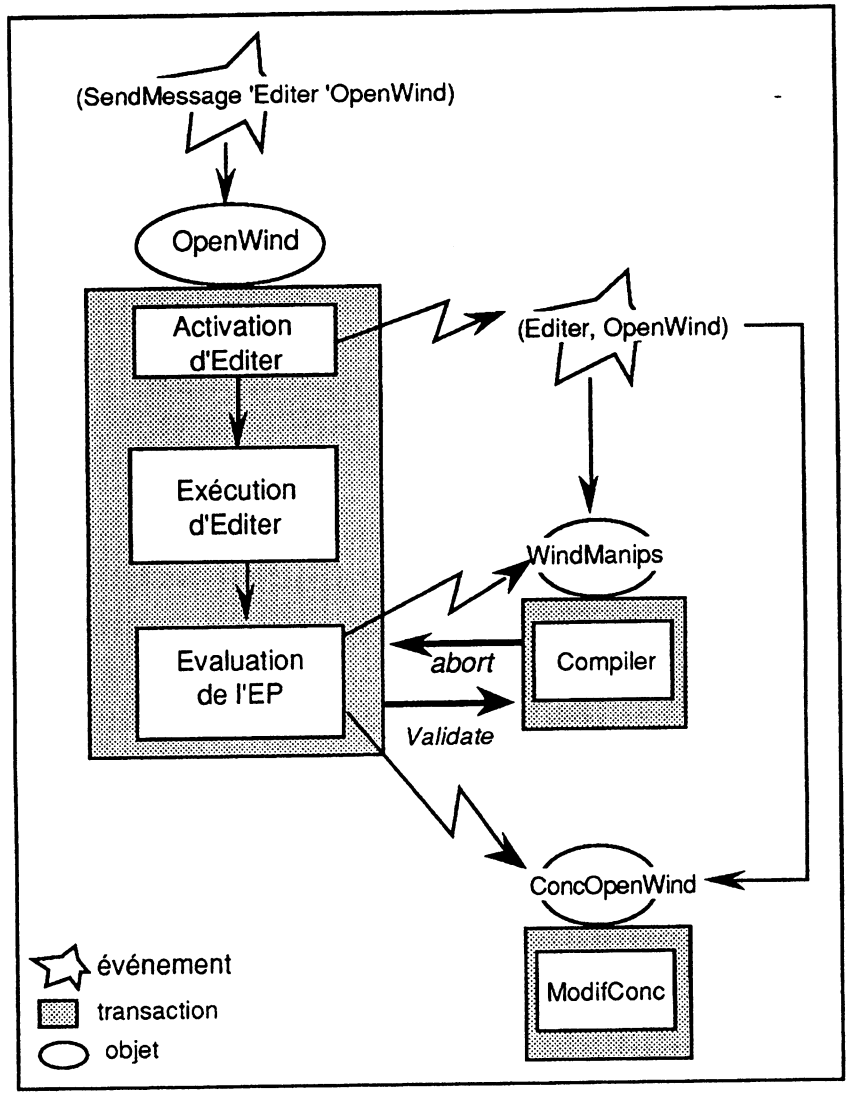


Figure-5

5.1. TRAITEMENT DES MESSAGES

Le déclenchement de la méthode de nom *Sel* attachée à l'objet *Obj* se fait par l'envoi du message suivant à cet objet :

(**SendMessage** Obj Sel ListeParamEffs)

En effet, ce message joue le rôle d'un signal d'événement externe envoyé à Obj.

Lorsqu'une méthode est déclenchée à la suite d'un événement externe, son contexte d'activation est considéré vérifié et elle est immédiatement activée en utilisant la liste de paramètres effectifs venant du message. Par exemple, le message suivant :

(**SendMessage** 'Editer 'OpenWind)

active la méthode Editer sur la fonction OpenWind.

5.2. ACTIVATION DES METHODES, CREATION DE TRANSACTIONS ET D'EVENEMENTS INTERNES

Lorsqu'une méthode Sel est activée sur un objet Obj, une transaction est créée, provoquant la création d'un événement interne. Cet événement est défini par le nom de la méthode activée Sel, le nom de l'objet Obj et les paramètres effectifs passés à la méthode:

(Sel, Obj, ListeParamEffs)

Par exemple, l'activation de la méthode Editer sur la fonction OpenWind crée l'événement interne (voir Figure 5) :

(Editer, OpenWind)

Après l'activation de la méthode Editer sur la fonction OpenWind, une transaction est créée; le contexte d'exécution qui représente les conditions d'exécution de la méthode est évalué dans cette transaction. Dans l'exemple, la méthode Editer n'a pas de contexte d'exécution; ce contexte est donc considéré comme vérifié.

L'évaluation du contexte d'exécution se fait en utilisant les paramètres effectifs venant de l'événement déclencheur. Si le contexte d'exécution est satisfait, la méthode est exécutée dans la transaction créée lors de son activation. Si le contexte d'exécution n'est pas satisfait, cette transaction prend **Refused** comme valeur d'état.

La satisfaction du contexte d'exécution $\mathfrak{C}$exécution d'une méthode *m* en utilisant ListeParamEffs s'exprime de la façon suivante :

$\mathfrak{C}$exécution *est satisfait pour* ListeParamEffs si et seulement si $\mathfrak{C}$exécution(ListeParamEffs)≠nil

5.3. EXECUTION DES METHODES

L'exécution d'une méthode consiste en l'évaluation de sa partie action. Elle est effectuée dans la transaction correspondant à l'activation de la méthode et en utilisant les paramètres effectifs venant de l'événement déclencheur. Si l'exécution d'une méthode échoue, la transaction prend **Failed** comme valeur d'état. Dans le cas où cette exécution réussit, l'état de la transaction devient **Succeeded**.

L'échec et la réussite de l'exécution de l'action d'une méthode *m* peuvent être exprimés ainsi :

> L'exécution de $\mathfrak{A}$ction en utilisant ListeParamEffs est dite *succeeded* si et seulement si $\mathfrak{A}$ction(ListeParamEffs)$\in \mathfrak{V} \cup \mathfrak{I}$, dans le cas contraire l'exécution de $\mathfrak{A}$ction est dite *failed.*

Dans l'exemple, l'exécution de la méthode Editer correspond à exécuter (DF 'OpenWind) qui permet d'éditer la fonction OpenWind.

Après l'exécution d'une méthode, l'événement interne correspondant à son activation est signalé à tous les objets spécifiés par son **environnement de propagation.** Si aucun objet n'est spécifié, l'événement est rejeté et sa transaction est validée ou annulée suivant son état. La validation et l'annulation des transactions se fait respectivement en utilisant les primitives **Validate** ou **Abort** présentées dans le paragraphe 3.2.

5.4. TRAITEMENT DES EVENEMENTS INTERNES

Dans le cas où un événement interne n'est pas rejeté, il doit être signalé aux objets spécifiés par l'environnement de propagation. L'information envoyée à chaque objet de l'environnement de propagation doit inclure en plus de l'événement déclencheur, des informations concernant la transaction origine. C'est pourquoi nous utilisons la notion de **signal d'événement**. Le signal indique en plus de l'événement, l'état de la transaction correspondante :

(Sel, Obj, ListeParamEffs, EtatTrans)

Le signal est envoyé à chaque objet Obj' de l'environnement de propagation en utilisant la primitive **Notify** :

(**Notify** Obj' Sel Obj ListeParamEffs EtatTrans)

Les **contextes de déclenchement** des méthodes associées à Obj' sont évalués, et celles dont le contexte de déclenchement est vérifié sont déclenchées. En effet, les méthodes ayant dans leur contexte de déclenchement le doublet (Sel, Type), tel que Type est le type de l'objet Obj origine de l'événement, sont déclenchées.

Ceci peut être exprimé pour une méthode *m* avec comme contexte de déclenchement $\mathfrak{C}$ déclenchement=(Sel', Type, ListeParamEffs) et un signal (Sel, Obj, ListeParamEffs, EtatTrans) ainsi :

$\mathfrak{C}$déclenchement=(Sel', Type, ListeParamEffs)*est vérifié par* le signal (Sel, Obj, ListeParamEffs, EtatTrans) si et seulement si Sel=Sel' et Obj$\in$*Extension*(Type)

Revenons à notre exemple, les objets désignés par l'environnement de propagation de la méthode Editer sont : les parents de la fonction OpenWind (les modules dont elle est composant), les modules liés à la fonction OpenWind par une relation *UtiliséPar* et le document de conception de cette fonction; soient le module WindManips et document de conception ConcOpenWind. Ces objets sont informés par l'événement (Editer, OpenWind) en utilisant :

(**Notify** WindManips Editer OpenWind NIL Succeeded)
(**Notify** ConcOpenWind Editer OpenWind NIL Succeeded)

Cela a pour effet de déclencher la méthode Compiler sur l'objet WindManips et la méthode ModifConc sur l'objet ConcOpenWind. En effet, le contexte de déclenchement de ces deux méthodes (Editer Fonction) est vérifié par le signal envoyé aux objets WindManips et ConcOpenWind.

5.5. DECLENCHEMENT DES METHODES

Le déclenchement d'une méthode consiste en l'évaluation de son contexte d'activation, les méthodes dont le contexte d'activation est vérifié sont activées d'abord.

A ce niveau , nous pouvons distinguer trois cas possibles suivant l'état de la transaction transmis par le signal :

- l'état de la transaction est **Refused**, seulement les méthodes dont le mode d'activation est **Refused** sont candidates à l'activation. Elles sont activées selon leur ordre de priorité, chacune dans une transaction indépendante de la transaction origine. Ensuite la transaction origine est annulée en utilisant la primitive **Abort**.
- l'état de la transaction est **Failed**, seules les méthodes dont le mode d'activation est Failed sont candidates à l'activation. Elles sont également activées selon leur ordre de priorité, chacune dans une transaction indépendante de la transaction origine. Ensuite la transaction origine est annulée en utilisant la primitive **Abort**.
- l'état de la transaction est *Succeeded*, les méthodes dont le mode est **Immediate** ou **Deferred** sont candidates à l'activation. Les méthodes de mode **Immediate** sont ordonnées pour leur activation selon leur ordre de priorité. Les méthodes de mode **Deferred** sont ensuite traitées.

Dans l'exemple, la méthode Compiler est d'abord activée, ensuite la méthode ModifConc.

L'activation des méthodes de mode **Immediate** se déroule dans des sous-transactions de celle correspondant à l'événement déclencheur (transactions imbriquées), provoquant ainsi la création d'autres événements internes. Lors de son activation, les paramètres formels de la méthode sont remplacés par les paramètres effectifs venant du signal de l'événement. La réussite de toutes les transactions filles (correspondant aux méthodes de mode *Immediate*) conditionne la validation de la

transaction mère, ce qui est fait en utilisant la primitive **Validate**. L'échec ou le refus d'une seule transaction fille implique l'échec de ses ancêtres, et défait ainsi ses effets.

Si la transaction mère est validée, les méthodes ayant **Deferred** comme mode deviennent candidates à l'activation. Elles sont ordonnées selon leur priorité et exécutées chacune dans une transaction indépendante de celle correspondant à l'événement déclencheur. Le refus, la réussite ou l'échec de l'exécution de ces méthodes n'a aucun effet sur la transaction de l'événement déclencheur.

Par exemple, si l'édition de la fonction OpenWind réussit (supposons que c'est le cas), la méthode Compiler est d'abord activée dans une sous-transaction de la transaction correspondant à l'édition de la fonction openWind (voir Figure 5). Les modifications apportées par l'édition d'une fonction ne sont validées que si la compilation des modules qui utilisent cette fonction réussit, dans le cas contraire, les modifications sont rejetées.

Après la validation de la transaction de l'édition, la méthode ModifConc est activée sur l'objet ConcOpenWind dans une transaction indépendant de celle de l'édition. La modification ou non modification du document de conception, c'est-à-dire, le refus, la réussite ou l'échec de ConcModif n'a pas de conséquence sur la transaction de Editer.

6. UTILISATION DU MECANISME

Le mécanisme de déclencheur présenté ici peut être utilisé pour résoudre uniformément tous les problèmes liés à la dynamique dans un environnement génie logiciel. Les méthodes peuvent être utilisées pour exprimer, non seulement le comportement classique des objets, mais aussi, tous les contrôles imposés par les applications : les conditions permettent d'exprimer des contraintes à satisfaire avant l'exécution d'une méthode, les méthodes **Immediate** permettent d'exprimer les contraintes d'intégrité à satisfaire et les actions à entreprendre pour empêcher les cas d'incohérence non-tolérée par le système, la propagation des effets de l'exécution des méthodes peut se faire en utilisant les méthodes *Deferred*, la vérification des droits d'accès des utilisateurs aux objets représente aussi un exemple d'utilisation de ce mécanisme.

Le mécanisme peut être également utilisé pour gérer certains aspects sémantiques du modèle de données proposé dans le chapitre 4, tels que, la gestion d'attributs opérationnels, d'objets composites, de versions et de configurations, etc. Dans la

suite de ce chapitre nous donnons quelques exemples d'utilisation du mécanisme pour la gestion des aspects sémantiques du modèle.

6.1. CONTROLE DES DROITS D'ACCES

Avant d'exécuter une méthode sur un objet par un utilisateur, les droits d'accès doivent être contrôlés : si l'utilisateur n'a pas le droit d'exécuter la méthode, l'envoi du message doit être annulé, et l'utilisateur doit être averti. Cela peut être exprimé en utilisant les deux méthodes ci-dessous déclarées au niveau du type Super la racine du treillis de types, elles sont donc valables pour tous les objets de la base :

```
((Method ( CheckRights Super))
 (On (SendMessage Super))
 (WithArgs (self origin (Sel Args)))
 (If (HasRight USERNAME origin Sel))
 (Do (<- origin Send Sel Args))
 (Notify (self)))

((Method (AccessRefused Super))
 (OfMode Refused)
 (On (CheckRights Super))
 (WithArgs (self origin (Sel Args)))
 (Do (PRINTOUT T "Vous n'avez pas le droit d'exécuter" Sel " sur
       "origin" !!")))
```

La méthode CkeckRights est déclenchée à la suite d'un envoi de message à un objet pour exécuter une méthode. CheckRights signifie qu'à l'envoi d'un message par un utilisateur (dont le nom est indiqué par la variable USERNAME) à l'objet Obj pour exécuter la méthode de nom Sel, les droits d'accès doivent être vérifiés (en utilisant la fonction HasRights). Si l'utilisateur a le droit d'activer la méthode sur l'objet Obj, elle est déclenchée. Dans le cas contraire la transaction correspondant à CheckRights prend la valeur d'état **Refused**, ce qui a pour effet de déclencher la méthode AccessRefused. Cette dernière envoie à l'utilisateur un message pour l'avertir que son message est non valable.

Remarquons que, dans cet exemple, les variables *self* et *origin* sont toutes les deux remplacées par l'objet auquel est envoyé le message.

6.2. GESTION D'ATTRIBUTS OPERATIONNELS (OBJETS DERIVES)

Les attributs décrits dans le chapitre précédent peuvent être opérationnels. Contrairement aux attributs classiques où l'état de l'attribut est stocké dans une structure de données, un attribut opérationnel déduit son état par un programme exécuté lors de l'accès à cet attribut. A chaque attribut opérationnel est donc associé

un opérateur dont le but est de définir à chaque moment son état. Cela peut être défini en utilisant les méthodes suivantes :

```
((Method ( GetAttribute Super))
 (WithArgs (self origin (nomAtt)))
 (If (NOT (<- self Operational nomAtt)))
 (Do (<- self GetAtt nomAtt))
 (Notify (self)))

((Method ( GetOpAtt Super))
 (On (GetAttribute Super)) (WithArgs (self origin (nomAtt)))
 (OfMode Refused)
 (Do (<- origin nomAtt)))
```

Dans le cas où l'attribut n'est pas opérationnel, l'exécution de l'action de GetAttribute permet de trouver la valeur stockée de l'attribut. Si l'attribut est opérationnel, la méthode GetOpAtt est déclenchée. Elle permet d'exécuter l'opérateur nomAtt sur l'objet *origine*.

6.3. GESTION D'OBJETS COMPOSITES

Comme nous avons vu dans le chapitre précédent, la sémantique des objets composites peut être exprimée par la règle suivante : la suppression d'un objet composite implique la suppression de tous ses composants qui ne sont pas utilisés en tant que composants d'autres objets. Cette règle est exprimée en utilisant les méthodes suivantes :

```
((Method ( Delete Super))
 (WithArgs (self origin))
 (Do (<- self DelRels))
 (Notify ( AllComponents self)))

((Method ( DelIfNotComp Super))
 (On (Delete ElenObject))
 (WithArgs (self origin))
 (OfMode Immediate)
 (Do (COND ((EQUAL (GetValue self 'ChildOf) (LIST origin))
            (SendMessage self 'Delete))
           (T (<- self ExtractParent origin)))))

((Method ( DeleteOnlyRoot ElenObject))
 (On (Delete ElenObject))
 (WithArgs (self origin))
 (OfMode Immediate)
 (Do (<- self Destroy)))
```

L'envoi d'un message pour activer la méthode Delete sur un objet a pour effet de supprimer toutes les relations dont il est destinataire (en utilisant DelRels). Les composants de l'objet sont informés de la suppression de leur père. Ils réagissent en déclenchant la méthode DelIfNotComp. Cette dernière, a pour effet de supprimer chaque composant qui n'est pas composant d'autres objets dans la base. Pour les composants qui ne sont pas supprimés, elle enlève la relation de composition entre le composant et son père qui doit être supprimé. Enfin, l'objet auquel est envoyé le message initialement réagit de nouveau en déclenchant la méthode DeleteOnlyRoot qui le détruit effectivement.

6.4. CONTROLE D'OBJETS GENERIQUES ET DE VERSIONS

L'ensemble des règles définies dans le paragraphe 5.3.1 du chapitre 4 expriment la sémantique associée aux objets génériques et aux objets versions. Nous allons donner, à titre d'exemple, les méthodes nécessaires pour exprimer la règle suivante : l'envoi d'un message à un objet générique revient à envoyer ce message à la version par défaut. Cette règle est exprimée par la méthode suivante :

```
((Method ( SendToDefIfGen Elen))
 (On (SendMessage Super))
 (WithArgs (self origin (Sel Args)))
 (Do (COND ((<- origin InstOf! ($ GObject)) (<- Send (GetValue self
                'DefaultVersion) Sel Args))
           (T (<- Send self Sel Args))))
```

La méthode SendToDefIfGen est déclenchée à la suite d'un envoi d'un message pour exécuter la méthode Sel sur un objet Obj. Si Obj est un objet générique, elle déclenche la méthode Sel sur la version *par défaut* de cet objet. Dans le cas contraire, elle déclenche la méthode Sel sur l'objet auquel est envoyé le message initialement.

7. CONCLUSION

Dans ce chapitre, nous avons abordé les aspects dynamiques du modèle de données proposé pour la gestion d'un noyau générique des environnements génie logiciel.

En résumé, notre proposition consiste à définir un mécanisme de déclencheur ayant la particularité d'être, d'une part, général pour pouvoir traiter d'une manière uniforme tous les aspects dynamiques (propagation des effets de bord, vérification des contraintes d'intégrité, contrôle des droits d'accès, gestion des objets dérivés et gestion des objets complexes), et, d'autre part, spécifique aux applications génie logiciel où les problèmes de performance s'imposent.

Dans cette proposition, le comportement classique des objets et leurs réactions dans le cas d'incohérence ou de violation de contraintes d'intégrités sont exprimés d'une façon uniforme par les méthodes. Les méthodes sont ainsi des règles (déclencheurs) exprimées d'une façon déclarative, indépendamment du langage de programmation. Elles sont exécutées lors de la production d'événements associés, ou bien, à la demande explicite par l'envoi de message, ou bien, implicitement lors de la création d'événements internes par le système.

Chaque méthode spécifie aussi les objets pouvant réagir suite à son exécution. Il n'est pas possible de laisser toute la base réagir à la production d'événements étant donné le grand nombre des objets et la nature des opérations longues et coûteuses qui peuvent être déclenchées inutilement. Limiter le nombre d'objets répondant à une action, en utilisant la notion d'**environnement de propagation** des méthodes, permet une flexibilité du mécanisme et une efficacité.

Les méthodes déclenchées à la suite de la production d'un événement sont traitées soit dans la transaction de l'événement (méthodes **Immédiate**) soit dans une autre transaction indépendante (méthodes **Deferred**). Cela permet de tolérer certains cas d'incohérence, aspect très important dans un environnement génie logiciel.

En utilisant le mécanisme de déclenchement ainsi défini, nous pouvons décrire aussi bien des contrôles au niveau des applications, que des contrôles au niveau de l'expression des aspects sémantiques du modèle utilisé (gestion des objets composites, etc).

CHAPITRE 6

LA REALISATION, PROTOTYPE ELEN

1. INTRODUCTION

Dans ce chapitre nous présentons le prototype Elen, implantant le modèle de données et les opérations associées présentées dans les chapitres 4 et 5. Cette version du prototype réalise les fonctionnalités de base aussi bien sur les aspects statiques que sur les aspects dynamiques.

Le prototype est actuellement opérationnel sur Machine Xerox 1186, sous l'environnement Interlisp-D. Il est implanté en Loops qui est un langage orienté-objet.

L'utilisation d'un langage orienté-objet permet de bénéficier des mécanismes de base de l'approche objet (classe, objet, héritage, etc). Nous avons dû enrichir ces notions pour réaliser les fonctionnalités de notre proposition (vérification de types, gestion d'objets composites et de versions, gestion de la dynamique, etc).

Pour réaliser les fonctionnalités des systèmes hypertexte, nous avons utilisé NoteCards. NoteCards est un système d'hypertexte complètement intégré à l'environnement de programmation Interlisp. Dans l'objectif de s'adapter aux applications génie logiciel, le système Elen ajoute à NoteCards les mécanismes d'abstraction nécessaires pour la gestion d'objets composites, pour la représentation des versions et pour la gestion de la cohérence et de la dynamique des données. Aussi, il augmente les capacités de recherche de NoteCards en ajoutant les outils de base pour effectuer la recherche par le contenu sémantique des documents.

Pour expérimenter le prototype, nous avons défini une application permettant de gérer le prototype lui même. Notre application consiste en la définition d'un environnement de programmation orienté-objet permettant de programmer en Loops sur la machine Xerox. Pour valider cet environnement, celui-ci a été utilisé pour gérer le code source du prototype et la documentation associée. Ainsi toute utilisation ou modification de ces objets est assurée par l'environnement de façon à ce que le prototype reste dans un état cohérent.

Nous commençons ce chapitre par une brève présentation de l'environnement de programmation utilisé : le langage Loops et l'hypertexte NoteCards. Ensuite, dans la section 4, nous présentons le prototype réalisé.

2. PRESENTATION DU LANGAGE LOOPS

Loops [Loo 88] est un langage Orienté-Objet développé à Xerox PARC sur l'environnement Interlisp. Il intègre la programmation procédurale, la programmation Orientée-Objet et la programmation déclarative par règles. Loops est intégré dans Interlisp, et donne ainsi la possibilité d'accéder à toutes les fonctions standards de Lisp et de profiter des facilités que présente le système Interlisp-D (Structure Editor, Lisp Windows, etc).

Trois types d'objets sont supportés par Loops : les classes, les instances, les méta-classes. Le comportement des objets est décrit par des méthodes qui sont déclenchées par des messages envoyés aux objets. Loops supporte également la notion de *valeurs actives* qui provoquent l'exécution de certaines procédures lors de l'accès aux objets. Les règles dans LOOPS sont des règles de production, elles constituent un moyen pour représenter des bases de connaissances.

Nous présentons par la suite les notions de Loops que nous utilisons dans le prototype et certains aspects syntaxiques. En effet, dans l'implantation, nous avons choisi de nous limiter à l'utilisation des concepts de Loops présents dans la plupart des langages orientés-objet : instance, classes, méta-classes et méthodes. Ceci rend le prototype facile à comprendre et lui assure une certaine indépendance par rapport au langage utilisé.

2.2. INSTANCES, CLASSES ET META-CLASSES

Une instance est une entité représentée par un identificateur unique (UID) et un ensemble de valeurs appelés *Instance Variables*, elle peut avoir un nom. Chaque instance est créée dans une classe en lui envoyant le message **New**. Par exemple le message suivant :

```
(<- ($ Window) New ' Window1)
```

crée une instance de nom Window1 de la classe Window.

Une classe est la description d'un ensemble d'objets appelés ses instances. Les classes sont rangées dans un treillis permettant l'héritage multiple de structure. Les classes sont elles-mêmes instances de certaines classes appelées méta-classes. La notion de méta-classe est utilisée pour définir une description et un comportement unique à un ensemble de classes. Un exemple de méta-classe est *Class* qui regroupe toutes les classes.

La création d'une classe se fait en envoyant le message **New** à la méta-classe Class :

```
(<- ($ Class) New 'Window)
```

Chaque classe indique sa méta-classe, une liste de variables dont la valeur est commune à toutes les instances de la classe, une liste de variables appelées *Instances Variables* et dont les valeurs sont spécifiques à chaque instance de la classe, une liste représentant les super-classes de la classe et un ensemble de méthodes décrivant le comportement des instances de la classe. Par exemple, la classe Window créée précédemment peut être spécifiée ainsi :

SEdit Window Package INTERLISP

```
((MetaClass Class
 (Supers Object)
 (ClassVariables
  (TitleItems NIL doc "title items of window"))
 (InstanceVariables
  (Left NIL doc "left position of window")
  (Bottom NIL doc "bottom position of window"))
 (MethodFns  Window.Open Window.Close))
```

2.3. METHODES ET MESSAGES

Le comportement des instances d'une classe est exprimé par des méthodes associées à cette classe. Les méthodes sont des fonctions Interlisp associées aux classes. Chaque méthode est identifiée par un nom appelé sélecteur.

Les méthodes sont définies en utilisant la fonction **DefineMethod**. L'exemple suivant montre comment associer une méthode de nom Open à la classe Window :

```
(DefineMethod ($ Window) 'Open Args Expr)
```

Args est une liste d'arguments et *Expr* est la définition d'une fonction Interlisp qui constitue le corps de la méthode.

Les méthodes sont exécutées en réponse à des messages envoyés aux objets et dont la syntaxe est la suivante :

```
(<- self sel arg1 .. argn)
```

self désigne l'objet recevant le message, *sel* est le nom (sélecteur) de la méthode et *arg1... argn* sont les paramètres effectifs passés à la méthode. Par exemple, pour ouvrir la fenêtre Window1, le message suivant est envoyé :

```
(<- ($ Window1) Open)
```

3. PRESENTATION DE NOTECARDS

NoteCards [Hal 88] est un système d'hypertexte développé à Xerox PARC intégré dans l'environnement Interlisp. Il est conçu pour rassembler, structurer et analyser des informations textuelles et graphiques.

Les objets de base dans NoteCards sont les NoteCards ou plus simplement les *cards*, que nous désignons dans la suite par le terme *cartes*. Pour manipuler des informations de natures différentes, trois types de *cartes* sont proposés: *Text* pour représenter les unités textuelles, *Sketch* pour représenter des éléments graphiques et *Graph* pour représenter des structures de graphes. Les *cartes* peuvent être décrites par une liste de propriétés spécifiant, par exemple, l'auteur, la date de création, etc.

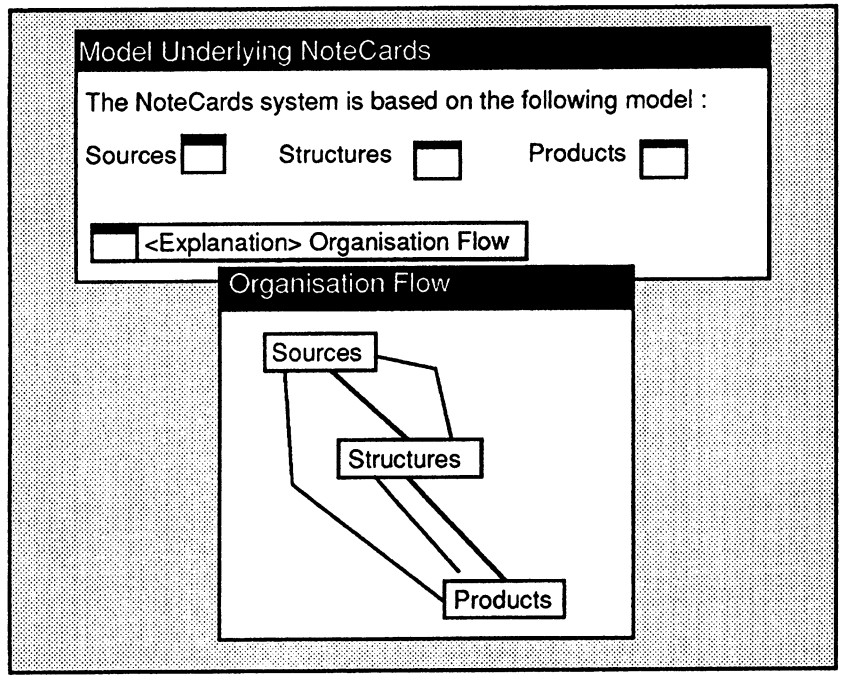


Figure-1

Les *liens* sont utilisés pour connecter les *cartes*, ils représentent les associations entre les informations. Ils sont identifiés sur l'écran par des icônes qui peuvent se trouver dans le texte ou les graphiques d'une *carte*. Les liens sont typés; le type d'un lien spécifie la nature de l'association entre les *cartes*. La notion de type de lien sert seulement pour nommer les liens, aucune vérification de type n'est effectuée par le système. L'utilisateur a la possibilité de définir les types de liens qu'il souhaite avoir. L'utilisateur, en utilisant la souris, peut retrouver et visualiser la destination d'un lien. Par exemple, dans la figure 1, la *carte* intitulée "Model Underlying NoteCards" a plusieurs liens vers d'autres *cartes* expliquant les différents composants du modèle. Le destinataire d'un de ces liens est la *carte* intitulée

"Organisation Flow". Le type est Explanation, ce lien a été traversé pour montrer la *carte* destinataire.

Les *cartes* sont organisées en utilisant les *FileBoxes*. Les *FileBoxes* sont des *cartes* spécifiques qui regroupent un ensemble de *cartes* et de *FileBoxes* constituant ainsi une hiérarchie de *FileBoxes*. La figure 2 montre un exemple d'un FileBoxe contenant, entre autres, les *cartes* illustrées par la figure 1.

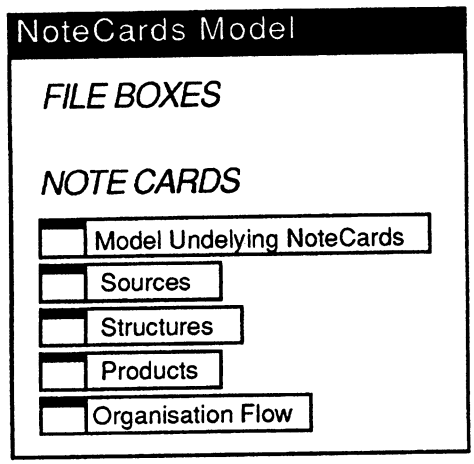


Figure-2

Aussi, il est possible d'avoir une représentation graphique d'un ensemble de *cartes* et de liens à l'aide des *browsers*. Les *browsers* sont des *cartes* spécifiques, ils représentent des réseaux constitués de nœuds et de liens : les nœuds représentent des *cartes* et les liens représentent les liens entre ces *cartes*.

Les objets (cartes, liens, FileBoxes, etc) sont stockés dans des fichiers NoteFiles, chacun correspond à une application ou à un sujet précis.

La navigation est le principal moyen pour accéder aux informations dans NoteCards. Toutefois, on dispose d'une facilité limitée d'interrogation qui permet de retrouver les *cartes* répondant à des requêtes. Les requêtes peuvent être spécifiées en utilisant les valeurs des propriétés associées aux *cartes*. Par exemple la requête suivante :

(NCP.PropSearch *NoteFile* (CreationDate 10/12/91) (Author Dupon))

retrouve toutes les *cartes* contenues dans le fichier de nom *NoteFile* ayant pour date de création et nom d'auteur les valeurs données.

4. LE PROTOTYPE

Notations utilisées : Pour désigner les types Elen, les méthodes Elen, etc, nous allons utiliser la notation Etype, Eméthode, etc. Les méthodes Loops sont

référées par : méthodes. Le mot *schéma* est utilisé pour désigner un ensemble de définitions de types et des méthodes associées. Un Eschéma désigne donc un ensemble de définitions de types et de méthodes Elen.

Afin de représenter les différents concepts du modèle (tels que : types, objets, documents, attributs de référence et de composition, méthodes, etc), nous avons défini un schéma Loops formant un treillis de classes (voir Figure 3). Les méthodes attachées à ces classes expriment la sémantique associée aux différents concepts (composition, versions, transactions, etc). L'annexe donne les spécifications du schéma Loops prédéfini.

Pour spécifier un Eschéma, c'est-à-dire pour intégrer des déclarations de Etypes et de Eméthodes, nous avons choisi d'utiliser un langage déclaratif de définition. Le prototype joue le rôle d'un générateur qui traduit le langage de spécification utilisé pour enrichir le schéma Loops prédéfini. Ainsi, la déclaration d'un Etype est traduite par la création d'une ou de plusieurs classes Loops, et chaque définition d'une Eméthode est traduite aussi par la création d'une ou de plusieurs méthodes Loops dont l'appel sera fait automatiquement.

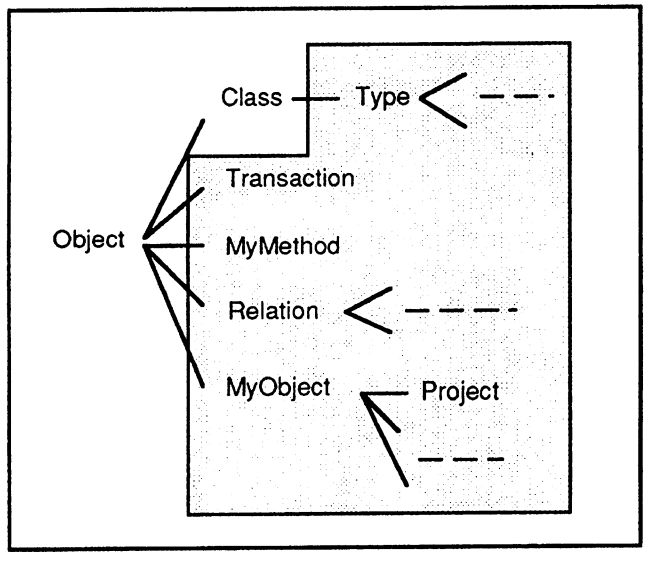


Figure-3

Les différents aspects de l'hypertexte sont intégrés à l'aide de NoteCards. Les objets Loops réalisent la liaison avec les cartes de NoteCards. Ceci nous a permis de disposer très rapidement des facilités que présente NoteCards de point de vue navigation, *browsering*, etc.

Deux types d'utilisateurs ont la possibilité d'interagir avec le système : l'administrateur d'un projet et l'utilisateur général (programmeur ou rédacteur d'un document). La définition d'un Eschéma se fait par le super-utilisateur qui utilise une interface de menus et le langage de définition pour spécifier les Etypes et les

Eméthodes. L'interaction avec la base se fait par l'envoi de Emessages, ce qui déclenche les Eméthodes définies par l'administrateur. Dans le cas de documents l'utilisateur peut accéder directement aux objets via leur interface hypertexte.

La figure 4 donne une idée générale du fonctionnement du prototype et de son interaction avec les différents composants de l'environnement.

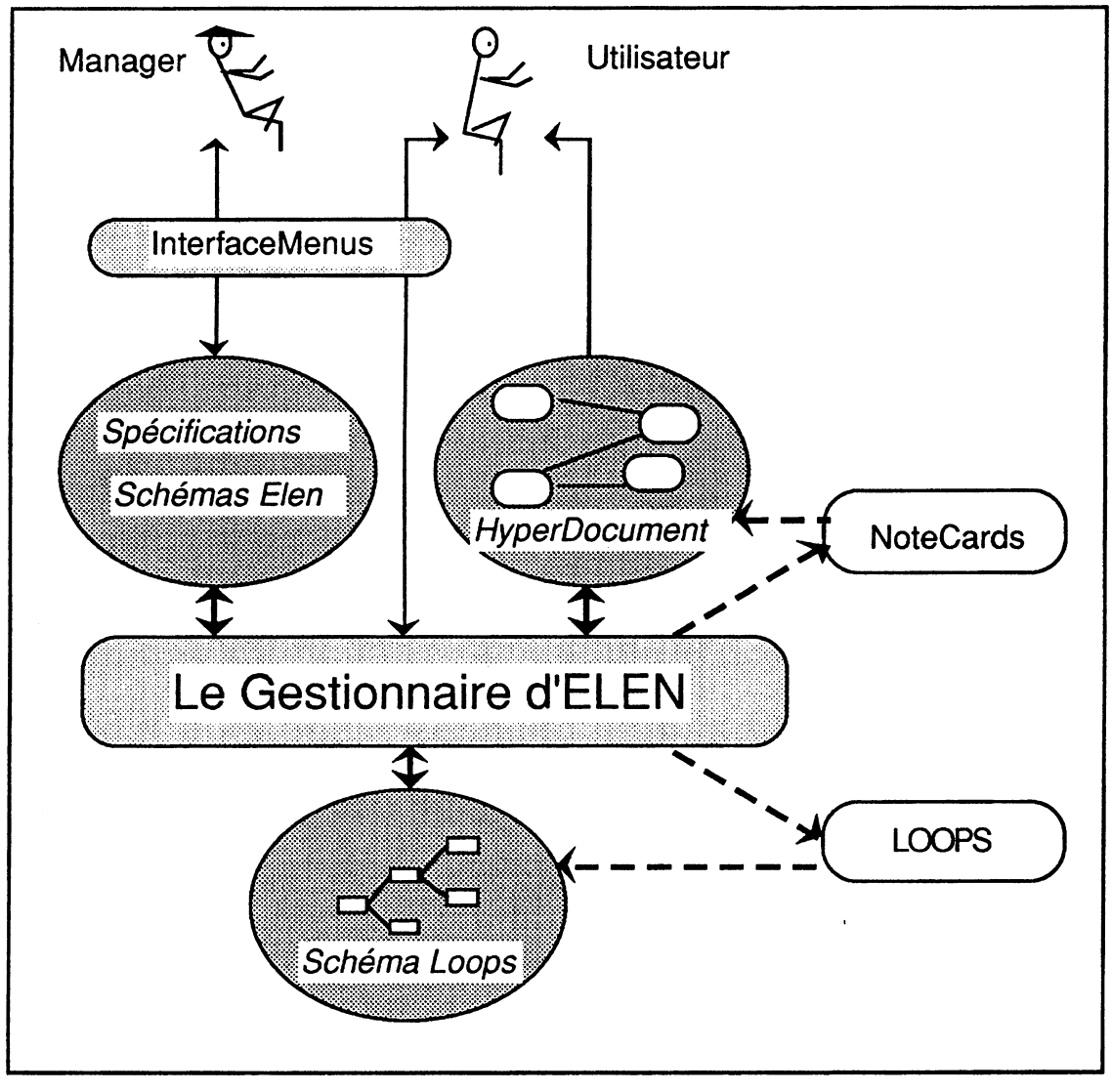


Figure-4

Si on exclut les aspects d'interface utilisateur et d'interface avec NoteCards, le prototype est constitué uniquement d'un ensemble de classes et de méthodes associées. Ceci rend difficile la possibilité de donner une architecture conceptuelle au prototype. Toutefois, du point de vue fonctionnel nous pouvons identifier les composants suivants.

- L'*interface utilisateur* : il est responsable de la gestion des menus permettant à l'utilisateur d'accéder aux différentes fonctionnalités implantées par le prototype.
- Le *gestionnaire des projets* : il permet la définition et la manipulation des projets, c'est-à-dire des sous-bases.
- Le *gestionnaire des types* : il est responsable de la saisie et du stockage des définitions de types. Il génère aussi le schéma Loops correspondant (création des classes nécessaires).
- Le *gestionnaire des objets et des versions* : il assure la création et la manipulation des objets composites et de leurs versions (dans le cas d'objets à versions). Il exprime ainsi la sémantique associées aux objets composites et aux versions. Pour les documents, ce gestionnaire implante les aspects hypertexte en assurant les liens entre la base d'objets et NoteCards.
- Le *gestionnaire des méthodes* : qui est responsable de la définition des méthodes, de traitement des messages et du déclenchement automatique des méthodes. Il implante ainsi les aspects dynamiques de la base.

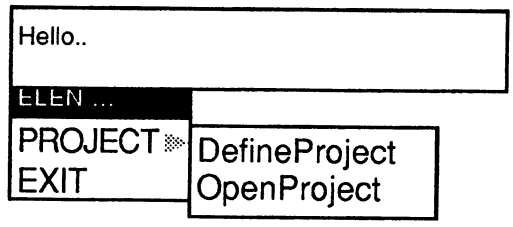
Par la suite nous présentons chacun de ces composants des deux points de vues : du point de vue utilisation du système et du point de vue implantation.

Remarque : Nous désignons par les mots en caractères gras les noms de méthodes et de classes Loops prédéfinies par le prototype. Les mots en italique désignent les arguments de méthodes.

Les exemples sont tirés de l'application définie dans l'annexe 1.

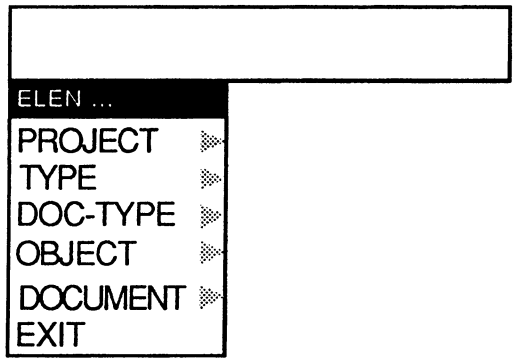
4.1. INTERFACE UTILISATEUR

L'utilisateur peut utiliser le système via des menus disposant de toutes les fonctionnalités du prototype et dont le choix des options (éléments des menus) se fait en utilisant la souris. Le menu principal ci- dessous permet de disposer des opérations nécessaires pour manipuler les bases d'objets :



Il est possible de quitter Elen en choisissant l'option EXIT de ce menu. Ceci a comme conséquence de sauvegarder le projet ouvert et de quitter l'application.

Lorsqu'on choisit PROJECT dans le menu principal, un sous-menu apparait sur l'écran (la flèche a côté d'une option du menu indique l'existence d'un sous-menu) dont les options sont : DefineProject et OpenProject, ils représentent deux fonctionnalités du gestionnaire de projets (voir le paragraphe suivant). En choisissant OpenProject, un deuxième menu (que nous appelons par la suite le deuxième menu Elen) apparaît sur l'écran, il représente les fonctionnalités des gestionnaires de types, de méthodes et d'objets :



Dans ce menu, l'option PROJECT permet de définir un autre projet ou de fermer le projet ouvert. TYPE et DOC-TYPE donnent la possibilité d'accéder à certaines fonctionnalités des gestionnaires de types et de méthodes (§4.2 et §4.4). OBJECT et DOCUMENT permettent de manipuler les objets de la base tout en utilisant le gestionnaire d'objets (§4.3). EXIT est le moyen de quitter l'application en sauvegardant la sous-base ouverte.

A chaque menu est associée une zone de dialogue permettant d'interroger l'utilisateur et de saisir ses réponses.

4.2. GESTIONNAIRE DES PROJETS

Le gestionnaire des projets permet de définir et de manipuler des sous-bases qu'on appelle projets. Chaque projet contient un ensemble de définition de types et peut utiliser des types définis par d'autres projets et appelés types importés. Pour un projet donné, les objets sont créés conformément aux types définis par ce projet ou importés d'autres projets.

Du point de vue implantation, chaque projet est représenté par un objet Loops qui est une instance de la classe prédéfinie **Project**. Les opérateurs définis par le gestionnaire de projets permettent de définir des projets, d'ouvrir un projet ou de fermer un projet ouvert.

4.2.1. Définition d'un Projet

4.2.1.1. Interface utilisateur

Le manager d'un projet a la possibilité de définir une sous base pour un projet en choisissant DefineProject dans le menu principal, il est donc demandé d'introduire, dans la zone de dialogue, le nom du projet à définir *nomProjet* . De point de vue interne le système génère le message Loops suivant :

(<- ($ **Project**) **Define** *nomProjet*)

Define étant une méthode de la classe **Project** prédéfinie par le système. Une "fenêtre guide" apparaît sur l'écran permettant d'éditer la définition du projet en utilisant un éditeur syntaxique "SEdit". La définition du projet doit spécifier le super-utilisateur, les utilisateurs généraux, et les types à importer des autres projets :

```
SEdit Elen Package: INTERLISP
( DefineProject Elen
  SuperUsers -ListOfUserNames-
  GeneralUsers -ListOfUserNames-
  ImportedTypes -ListeOfImportedTypesWithProjectNames-)
```

4.2.1.2. Représentation interne

A la suite d'une définition d'un projet, le système génère une instance de la classe **Project** de nom *nomProjet* . Cette classe sert à stocker cette définition. Aussi, deux fichiers sont créés : le premier sert à stocker le schéma Loops généré par la définition des types de ce projet, le deuxième est un NoteFile pour stocker les objets NoteCards (*cartes* et liens) correspondants à l'hyperdocument de ce projet. Par exemple, considérons la définition suivante du projet nommé Elen:

```
SEdit Elen Package: INTERLISP
( DefineProject Elen
  SuperUsers (JARWA)
  GeneralUsers (BRUANDET CHEVALLET)
  ImportedTypes NIL)
```

Ceci a comme conséquence de créer : une instance de la classe **Project** de nom Elen pour stocker la définition du projet, un fichier de nom ELENOBJF pour stocker le schéma Loops qui va être généré par le système et qui correspond aux définitions des types de ce projet, et un fichier ELEN.NOTEFILE qui servira pour le stockage des objets NoteCards correspondant à ce projet.

4.2.2. Ouverture d'un Projet

La création des types et des objets ne peut être effectuée que dans un certain projet. L'utilisateur a la possibilité d'ouvrir un projet en choisissant OpenProject dans le menu principal. Il doit donner le nom du projet à ouvrir *nomProjet*. Le système génère le message **Open** envoyé à l'objet de nom *nomProjet* :

(<- ($ *nomProjet*) **Open**)

De point de vue interne cela a pour conséquence d'initialiser certaines variables globales (ELENOPENEDPROJECT, ELENOBJECTFILE, ELENNOTEFILE), de charger le fichier contenant le schéma Loops concernant ce projet et d'ouvrir le NoteFile correspondant. Par exemple, le message suivant :

(<- ($ Elen) **Open**)

initialise les variables ELENOPENEDPROJECT, ELENOBJECTFILE et ELENNOTEFILE à Elen, ELENOBJF et ELEN.NOTEFILE respectivement, charge le fichier ELENOBJF et ouvre le NoteFile ELEN.NOTEFILE. Le système est alors prêt à recevoir des définitions de types ou des créations d'objets des types prédéfinis. Pour cela le deuxième menu apparaît sur l'écran (voir 4.1).

Dans l'état actuel du prototype un seul projet (sous-base) peut être ouvert à la fois, ce qui donne lieu à la notion du projet courant. Il est intéressant d'avoir la possibilité d'ouvrir plusieurs projets à la fois pour supporter le partage de tâches entre plusieurs équipes et pour pouvoir gérer l'accès concurrent à la base.

4.2.3. Fermeture d'un Projet

Pour fermer un projet l'utilisateur peut choisir CloseProject du deuxième menu Elen figurant sur l'écran. Le système envoi donc le message **Close** au projet considéré :

(<- ($ *nomProjet*) **Close**)

L'envoi de ce message a pour conséquence de sauvegarder les fichiers désignés par les variables globales concernant le projet courant. En envoyant le message (<- ($ Elen) **Close**), le fichier ELENOBJF est sauvegardé en utilisant la commande Interlisp **MAKEFILE**, le fichier ELEN.NOTEFILE est sauvegardé et fermé en utilisant la commande NoteCards **NCP.CloseNoteFile**.

4.3. GESTIONNAIRE DES TYPES

Chaque Etype est représenté d'une façon interne par un objet Loops instance de la classe prédéfinie **Type**. Plusieurs sous-classes de **Type** sont définies pour représenter les différentes catégories de types : composites, simples, atomiques, etc. Leur schéma est montré par la figure 5. La fonctionnalité du gestionnaire de types

est la saisie et le stockage les définitions des types d'objets et de documents ainsi que la gestion du schéma Loops correspondant en vue de supporter les futurs objets de ces types.

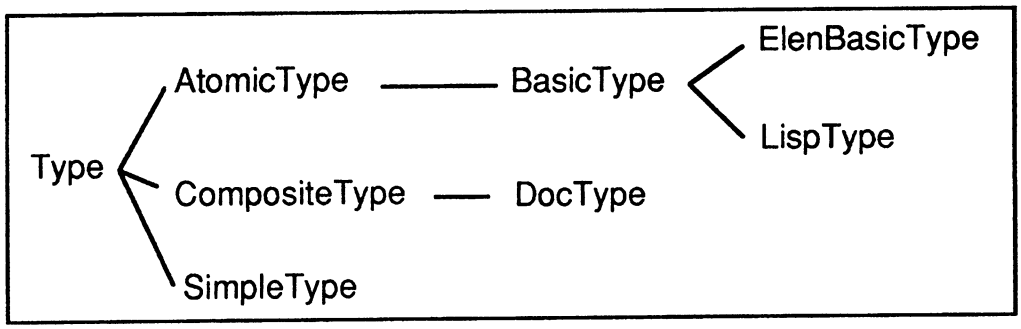


Figure-5

4.3.1. Définition des Types

4.3.1.1. Interface utilisateur

Le super-utilisateur du projet ouvert peut définir un type en choisissant DefineType dans le deuxième menu. Il est donc amené à introduire le nom du type *nomType* à définir. Ceci est traduit par le système en envoyant le message **Define** à l'objet prédéfini **Type**:

(<- ($ **Type**) **Define** *nomType*)

Une fenêtre d'édition est alors affichée sur l'écran pour aider l'utilisateur à donner la déclaration du type à définir. Par exemple, pour définir le type Logiciel, le message (<- ($ Type) **Define** Logiciel) est envoyé, et la fenêtre suivante est affichée sur l'écran :

SEdit Logiciel Package: INTERLISP

(**DefineType** Logiciel
Versioned -TorNIL-
IsA -listOfSupersOrNIL-
Props -listOfPropertiesOrNIL-
Comps -listOfComponentsOrNIL-
Refs -listOfReferencesOrNIL-)

L'utilisateur a la possibilité (en utilisant l'éditeur syntaxique SEdit) de remplir les différentes clauses définissant le type. Soit la déclaration suivant du type Logiciel :

```
SEdit Logiciel Package: INTERLISP

( DefineType Logiciel
  Versioned NIL
  IsA NIL
  Props ((Droits Seq 1 droit))
  Comps ((Modules Seq 1 Module)
        (Manuel Tuple 0 DocUsager)
            (Specifs Tuple 0 DocSpecifs)
            (Conception Tuple 0 DocConception))
  Refs NIL)
```

4.3.1.2. Représentation interne

Une vérification syntaxique est effectuée par le système sur la déclaration donnée. Ensuite, le système crée plusieurs objets dont l'objectif est de stocker la déclaration et de supporter les objets de ce type.

Par exemple la déclaration précédente du type Logiciel a pour effet de créer un objet de nom Logiciel, instance de la classe **CompositeType** (voir Figure 5). De plus, une classe OLogiciel est créée pour supporter les objets de type Logiciel. La classe OLogiciel est considérée comme une sous-classe de **MyObject** qui représente la racine du treillis des classes supportant les objets Elen (voir Figure-3). En effet, **MyObject** modélise les caractéristiques et le comportement communs aux objets composites. Elle possède pour cela plusieurs *instances variables* désignant, pour chaque objet, sa date de création, ses relations de référence et de composition avec les autres objets, les objets qui lui font référence, ses parents, etc. (*instance variable* est la notation utilisée dans Loops pour désigner les attributs attachés aux instances des classes).

Les attributs de propriétés associés à un type sont représentés par des *Instances Variables* attachées à la classe représentant ce type. Pour chaque attribut de référence ou de composition une classe est créée pour regrouper toutes les relations de référence ou de composition correspondantes. Les classes utilisées pour stocker les relations sont des sous-classes de **Relation** qui modélisent les informations spécifiques à chaque relation (type du destinataire, cardinalité, constructeur, objet source, objet destinataire, etc). Deux sous-classes de **Relation** sont considérées : **Reference** et **Composition** (voir Figure-6). Les méthodes prédéfinies dans ces classes représentent la sémantique associée aux références simples et aux références de composition.

Dans l'exemple précédent, l'attribut de propriété Droits est représenté par une *instance variable* associées à la classe OLogiciel. La déclaration de l'attribut de

composition Modules a pour effet de générer une classe Logiciel.Modules qui est une sous-classe de **Composition.**

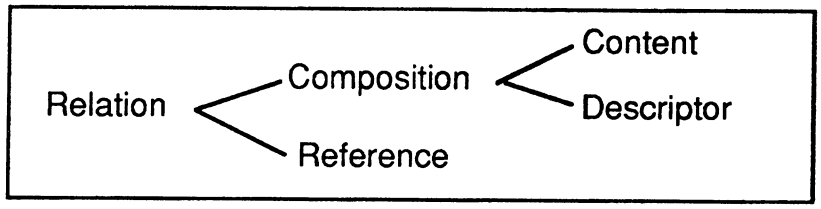


Figure-6

4.3.2. Treillis des Types, Héritage

Nous avons vu dans le chapitre 4 que les Etypes peuvent constituer un treillis dont Super est à la racine. Chaque type hérite de la structure des types de plus haut niveau. Pour réaliser l'héritage nous avons bénéficié de l'héritage défini dans Loops. Dans l'exemple précédent, le type Logiciel est considéré comme une spécialisation du type Source en utilisant la clause **IsA**. La classe associée OLogiciel est créée comme étant une sous-classe de OSource. Cela permet l'utilisation du mécanisme d'héritage de Loops pour réaliser l'héritage d'attributs de propriétés. Mais l'héritage d'attributs de composition, d'attributs de référence et des méthodes doit être de la responsabilité du prototype.

4.3.3. Définition des Types de Documents

Les documents sont des Eobjets ayant un contenu (texte, code source), un contenu sémantique et sont représentés par des fenêtres sur l'écran (*cartes* de NoteCards). Les liens entre les documents sont représentés sur l'écran par des liens entre les fenêtres correspondantes. L'ensemble des documents de la base représente ce qu'on appelle un hyperdocument.

Pour modéliser les documents, nous considérons la classe prédéfinie **DocType** qui est une sous-classe de **CompositeType** (voir Figure 5) et la classe **Doc** qui est une sous-classe de **MyObject** (voir Figure 7). Les instances de **DocType** sont des objets représentant les types de documents. **Doc** modélise les liens entre les objets de la base et l'interface hypertexte. Elle désigne, par exemple, la *carte* représentant chaque document et spécialise le comportement des objets pour réaliser les fonctionnalités de l'hypertexte.

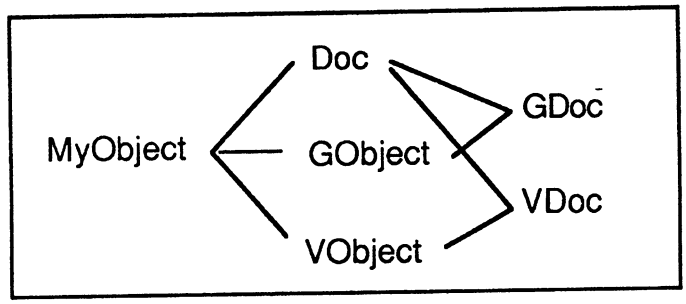


Figure-7

4.3.3.1. Interface utilisateur

Pour définir un type de documents le super-utilisateur choisit DefineDocType du deuxième menu Elen et introduit le nom du type à créer *nomDocType*. Le système crée ensuite le message Loops **Define** envoyé à l'Objet **DocType** ainsi :

(<- ($ **DocType**) **Define** *nomDocType*)

Une fenêtre d'édition apparaît sur l'écran pour éditer la déclaration du type. A la suite d'une déclaration d'un type de documents le système crée une instance de **DocType** représentant ce type, et une sous-classe de **Doc** supportant les objets de ce type. Par exemple, soit la déclaration suivante du type DocSpecFonlle :

```
SEdit DocSpecFonlle Package: INTERLISP

( DefineDocType DocSpecFonlle
  Versioned NIL
  IsA (Documentation)
  Content Text
  Descriptor NIL
  Props NIL
  Comps ((SpecModules Seq 1 DocSpecMod))
  Refs ((Conception Tuple 0 DocConception)))
```

4.3.3.2. Représentation interne

Le système crée un objet DocSpecFonlle instance de **DocType** pour stocker la définition de ce type et une classe ODocSpecFonlle, sous-classe de **Doc**. Cette dernière va contenir tous les documents du type DocSpecFonlle.

Si la déclaration d'un type de documents a un contenu (clause contenu non Nil), un attribut de composition de nom Content est créé. Pour représenter cet attribut de composition une sous-classe de **Content** doit être créée. **Content** spécifie les caractéristiques communes à toutes les relations de contenu (voir Figure 6). Pour l'exemple précédent, une classe de nom "DocSpecFonlle.Content" est créée comme étant une sous-classe de **Content,** "DocSpecFonlle.Content" va représenter toutes les relations de contenu associées aux documents du type DocSpecFonlle.

4.3.4. Les Types d'Objets ayant des Versions

La déclaration d'un type ayant des objets susceptibles d'avoir dēs versions engendre, en plus de l'objet représentant le type, deux classes ayant pour objectif de supporter d'une part les objets génériques de ce type et d'autre part les versions de ce type. Les classes représentant des classes d'objets génériques ou d'objets versions sont respectivement des sous-classes de **GObject** ou de **VObject**. Ces deux dernières sont à leur tour des sous-classes de **MyObject** (voir Figure 7), elles définissent les informations et le comportement spécifiques aux objets génériques et versions. **GObject** spécifie pour chacune de ses instances (objet générique) la version par défaut, la dernière version créée et les différentes versions alternatives. **VObject** spécifie pour chaque version la version origine, l'objet générique et les différentes versions dérivées.

Supposons que, dans la définition précédente du type Logiciel, la clause **Versionned** prend la valeur true "T", Le système crée dans ce cas, en plus de l'objet Logiciel, les deux classes VLogiciel et GLogiciel sous-classes de **VObject** et **GObject** respectivement.

Pour représenter les documents ayant des versions, deux classes sont définies : **VDoc** et **GDoc**. Comme leurs noms l'indiquent, elles représentent respectivement les versions des documents et les objets génériques des documents. **VDoc (GDoc)** est une sous-classe de **Doc** et de **VObject (GObject)** (voir Figure 7).

4.4. GESTIONNAIRE DES OBJETS ET DES VERSIONS

Le gestionnaire d'objets est responsable de la création, de la modification et de l'interrogation d'objets. Dans le cas d'objets ayant des versions, il assure la gestion des objets génériques et des versions. Nous présentons par la suite les fonctionnalités offertes aux utilisateurs et les effets internes qu'elles ont sur le schéma Loops.

4.4.1. Création d'un Objet sans Versions

La création d'un objet se fait forcément conformément à un certain type. L'utilisateur a la possibilité de créer un Objet en choisissant dans le second menu CreateObject (sous choix d'OBJECT). Il est demandé donc d'introduire le nom du type à considérer *Type* et le nom de l'objet à créer *NomObj*. Le système crée le message **NewO** envoyé à l'objet représentant ce type :

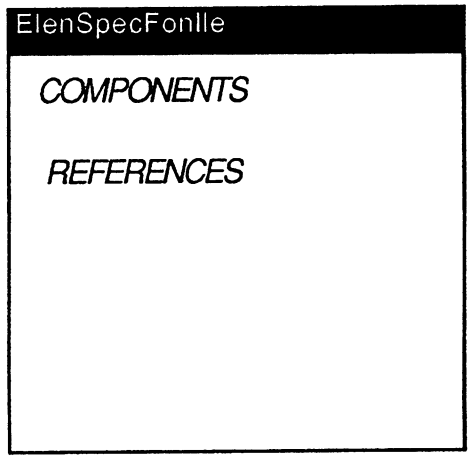
(<- ($ *Type*) **NewO** *NomObj*)

Du point de vue interne, si le nom donnée à l'objet est unique dans le type, cela entraîne la création d'une instance de nom *NomObj* de la classe correspondant à *Type*.

Si le type considéré est un type de documents, une *carte* de NoteCards va être créée et affichée sur l'écran pour représenter visuellement le document. Par exemple, à la suite de l'envoi du message Loops Suivant :

(<- ($ DocSpecFonlle) **NewO** MonSpecFonlle)

Le système crée une instance de la classe ODocSpecFonclle de nom ElenSpecFonlle et une *carte* qu'il affiche sur l'écran :



Les propriétés de cette *carte* sont les attributs de propriétés de l'objet ElenSpecFonlle.

4.4.2. Création d'un Objet d'un Type à Versions

Pour les types d'objets ayant des versions, deux opérateurs de création d'objets sont définis : la création des objets génériques et la création des versions. Toutefois, la création des objets génériques et la distinction entre ces objets et les versions peuvent être cachées à l'utilisateur final (programmeur) par le super utilisateur du projet.

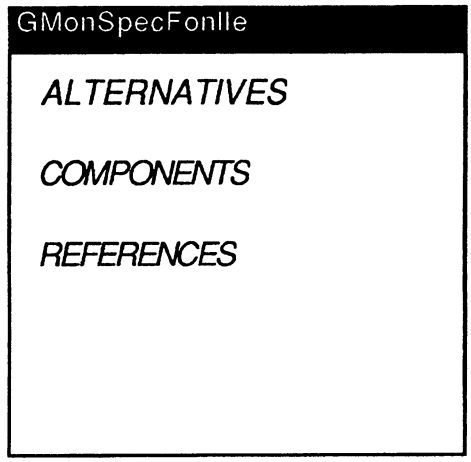
La création d'un objet générique d'un certain type se fait en choisissant CreateGenericObject du menu (sous choix d'OBJECT). L'utilisateur doit donner le nom du type et le nom de l'objet générique à créer *NomGObj*. Ceci entraîne l'envoi du message **NewGO** au type *Type* :

(<- ($*Type*) **NewGO** *NomGObj*)

Du point de vue interne le système crée une instance de la classe modélisant les objets génériques de *Type*. Dans le cas où *Type* est un type de documents, une *carte* représentant le document générique est créée et illustrée sur l'écran. A titre d'exemple, supposons que le type DocSpecFonlle est "versionnable" (clause "versionnable" est T dans la définition du type). L'envoi du message :

(<- ($ DocSpecFonlle) **NewGO** GMonSpecFonlle)

a pour effet de générer une instance de la classe GODocSpecFonlle de nom GMonSpecFonlle et de créer la *carte* suivante :

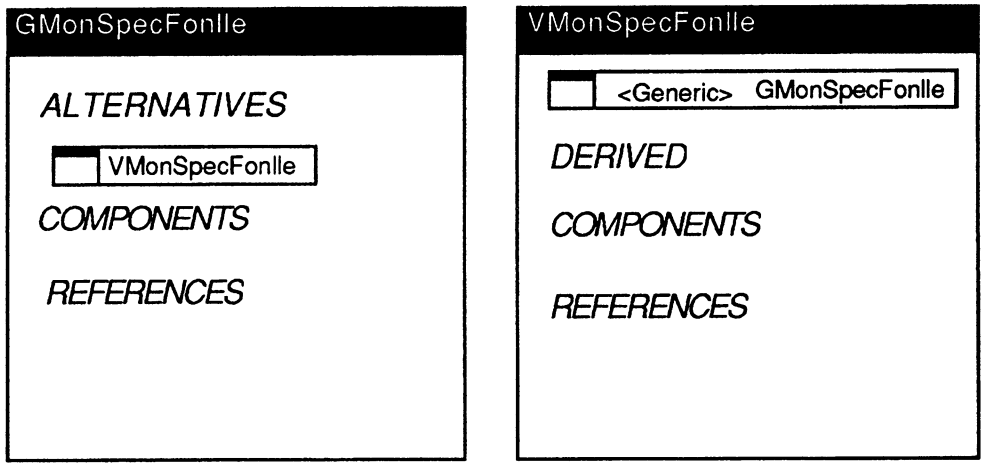


La création d'une version se fait en choisissant l'option CreateVersion du second menu (sous-choix d'OBJECT) et en donnant le nom de l'objet générique *NomGen* auquel appartient la version et le nom de la version *NomVer* à créer. Ceci engendre le message **NewV** envoyé à l'objet générique *NomGen* :

(<- ($ *NomGen*) **NewV** *NomVer*)

Ce message crée une instance de la classe *VOType* qui modélise les objets versions de *Type* (*Type* est le type de *NomGen*). *NomGen* va désigner la version *NomVer* comme étant l'une de ces alternatives. Si *Type* est un type de documents, une *carte* de type VDoc correspondant à cette version est créée et affichée sur l'écran. Des liens vont être créés entre *NomGen* et *NomVer*:

La création d'une version de nom VMonSpecFonlle de l'objet générique GMonSpecFonlle créé dans l'exemple précédent entraîne la création de la *carte* VMonSpecFonlle. Des liens sont créés entre les deux *cartes* VMonSpecFonlle et GMonSpecFonlle :



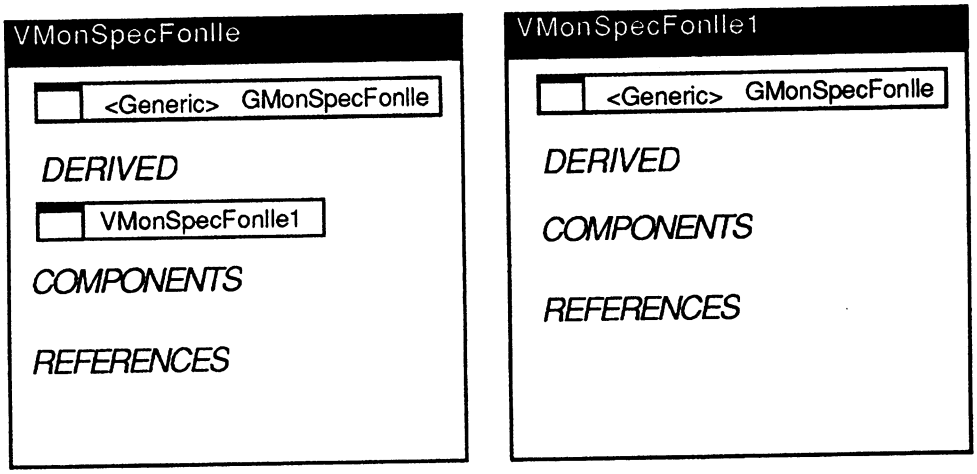
4.4.3. Dérivation d'une Version

La dérivation d'une version se fait en choisissant DeriveVersion dans le second menu Elen (sous choix d'OBJECT). L'utilisateur indique le nom de la version origine *VerOrigin* et le nom de la version dérivée *VerDer*. Un message Derive est généré par le système :

(<- ($ *VerOrigin*) **Derive** *VerDer*)

Du point de vue interne, il y a copie de l'objet origine et création des liens entre l'objet origine et la version dérivée. Pour les documents, la dérivation d'une version consiste en générer en plus la copie du contenu du document origine et en créant un lien de dérivation entre les *cartes* correspondant aux deux versions.

Nous montrons ci-dessous la création des *cartes* correspondant aux deux versions VMonSpecFonlle et VMonSpecFonlle1 à la suite de la dérivation de VMonSpecFonlle1 à partir de VMonSpecFonlle.



4.4.4. Evaluation des Attributs

L'évaluation des attributs se fait en choisissant PutObjectAtt dans le second menu. L'utilisateur doit introduire le nom de l'objet *Obj*, l'attribut à évaluer *Att* et la valeur *Val*. Le système engendre le message Loops suivant :

(<- ($ *Obj*) **PutAtt** *Att Val*)

Avant toute évaluation d'attribut Att une vérification de type doit être effectuée. *Val* doit avoir le type d'*Att* spécifié dans la déclaration du type de l'objet *Obj*.

Si l'attribut *Att* est un attribut de propriété, ceci représente une simple affectation de l'*instance variable* de nom *Att* associée à *Obj* par la valeur *Val*.

Dans le cas où *Att* est un attribut de référence ou de composition, *Val* doit être le nom d'un objet existant dans la base. PutAtt crée alors une relation de nom *Obj.Att* .

Soit *Type* le type de *Obj*, la relation *Obj.Att* est considérée comme étant une instance de la classe *Type.Att.* (remarquons que cette dernière a été créée lors de la définition de *Type* (voir 4.3.1.2)). Les *instances variables* de *Obj.Att* (représentant l'origine, la destination, etc) sont alors évaluées par les valeurs correspondantes (*Obj*, *Val*, etc). Si le constructeur de type associé à *Att* est une séquence alors *Val* est convertie automatiquement en une liste.

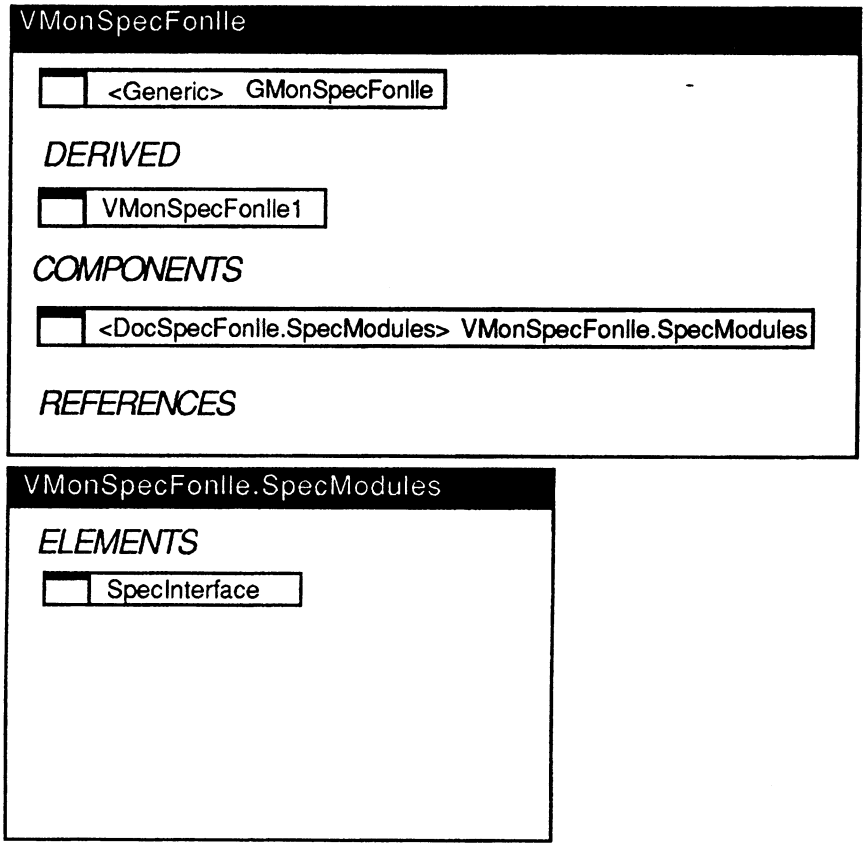
Nous pouvons distinguer plusieurs cas possibles pour *Obj* : il peut être un objet non versionnable, une version, un document (versionnable ou non). Pour prendre en considération les différents cas, la méthode **PutAtt** attachée à MyObject a été redéfinie (voir Figure 7).

Par exemple, si *Obj* est une version et Att est un attribut invariant, *Val* doit être associée à l'objet générique d'*Obj*. Dans le cas où *Obj* est un document, des liens sont créés entre les *cartes* d'*Obj* et de *Val* pour mettre à jour l'hyperdocument.

A titre d'exemple, pour évaluer l'attribut de composition SpecModules du document VMonSpecFonlle, le message suivant est utilisé :

```
(<- ($ VMonSpecFonlle) PutAtt 'SpecModules 'SpecInterface)
```

où SpecInterface est le nom d'un objet existant dans la base. Du point de vue interne, une relation de nom VMonSpecFonlle.SpecModules est créée. Cette relation est stockée comme instance de la classe DocSpecFonlle.SpecModules. Elle désigne VMonSpecFonlle comme origine et une liste contenant SpecInterface comme destinataire. VMonSpecFonlle est alors rajouté à la liste représentant les parents de SpecInterface. Un lien de composition est créé pour lier la *carte* de VMonSpecFonlle à la *carte* représentant la liste de ses composants dont le titre est VMonSpecFonlle.SpecModules. La carte représentant la liste des composants est liée à la *carte* de SpecInterface pour le désigner comme l'un de ses éléments :



Plusieurs autres opérateurs d'évaluation d'attributs sont mis à la disposition de l'utilisateur. L'objectif est surtout de manipuler les attributs ayant la séquence comme constructeur de type, par exemple :

AppendAtt a pour effet d'ajouter un élément à la fin de la liste des valeurs d'un attribut.

AttachAtt a pour effet d'ajouter un élément au début de la liste des valeurs d'un attribut.

4.4.5. Suppression d'un Objet

La suppression d'un objet se fait en choisissant Delete (sous choix d'OBJECT) dans le second menu et en indiquant le nom de l'objet à supprimer. Ce qui est traduit par le système par l'envoi d'un message **Delete** à cet objet :

(<- ($ *Objet*) **Delete**)

Dans le cas général, la suppression d'un objet conformément à la sémantique d'objets composites revient à supprimer tous les composants non composants d'autres objets, et ensuite à détruire effectivement l'objet. Tous les objets faisant référence à l'objet détruit doivent être informés de ce fait et les relations correspondantes sont alors détruites. La méthode **Delete** est définie sur la classe

MyObject, elle est redéfinie pour les sous-classes de MyObject (voir Figure 7) pour prendre en considération le cas des versions et de documents.

4.4.5.1. Cas d'une version

Avant de détruire une version avec ses composants non partagés, la version doit être enlevée de l'arbre de dérivation correspondant. Si cette version est celle par défaut, la version la plus récente va être considérée comme étant la version par défaut.

4.4.5.2. Cas d'un objet générique

Chaque objet générique désigne plusieurs versions. La suppression d'un objet générique consiste, non seulement à supprimer tous les composants non partageables, mais aussi, à supprimer toutes ses versions.

4.4.5.3. Cas d'un document

Quand un objet document est supprimé, la *carte* correspondante et tous les liens associés doivent être supprimés.

4.4.6. Interrogation des Attributs

Dans l'état actuel du prototype, à part la navigation, c'est la seule fonctionnalité offerte pour l'interrogation de la base. Pour obtenir la valeur d'un attribut, l'option GetObjectAtt peut être choisie dans le second menu (sous-choix d'OBJECT). Le système demande à l'utilisateur d'introduire le nom de l'objet *Obj* et l'attribut *Att* en question. Il envoie ensuite le message **GetAtt** à l'objet de nom *Obj* :

(<- ($ *Obj*) **GetAtt** *Att*)

La méthode **GetAtt** prend en considération les différentes règles définies dans le chapitre 4. Pour cela plusieurs cas sont envisageables : *Obj* peut avoir de versions ou non, *Obj* est un objet générique ou *Obj* est un document.

4.4.6.1. Cas d'un objet sans versions

Si *Att* est un attribut de propriété, la valeur de cet attribut stockée dans l'*instance variable Att* est retournée.

Pour les attributs de composition ou de référence, **GetAtt** retourne l'identificateur de l'objet (ou la liste des identificateurs d'objets) qui est lié à *Obj* via *Att*. Pour cela l'objet *Obj.Att* représentant le lien est consulté.

4.4.6.2. Cas d'une version

Dans le cas où la version fait référence via Att à un objet générique, **GetAtt** retourne sa version par défaut. Pour les attributs spécifiés comme étant invariant, la valeur retournée par **GetAtt** est obtenu de l'objet générique de la version.

4.4.6.3. Cas d'un objet générique

Un objet générique peut faire référence soit à une version soit à un autre objet générique. Le premier cas signifie que toutes les versions de cet objet générique font référence à cette version. l'identificateur de la version est alors retourné par **GetAtt**.

Le deuxième cas dépend de la spécification de l'attribut considéré : si cet attribut représente un invariant, toutes les versions de l'objet générique font référence à la version par défaut qui est alors retournée par **GetAtt**.

Si l'attribut n'est pas un invariant, le lien entre les deux objets génériques signifie que les versions du premier ne peuvent faire référence qu'à des versions du deuxième via l'attribut considéré et **GetAtt** retourne dans ce cas l'objet générique.

4.4.6.4. Cas d'un document

GetAtt fonctionne comme mentionné précédemment, il est en plus responsable de l'affichage des *cartes* des documents retournés en utilisant le message **Display**.

4.5. GESTIONNAIRE DES METHODES

Le gestionnaire de méthodes est responsable de la définition des Eméthodes, de la gestion des transactions, du traitement des Emessages envoyés, du déclenchement, de l'activation et de l'exécution des méthodes ainsi que de la propagation des effets de bord. En d'autres termes, il contrôle la *dynamique dans la base*.

4.5.1. Définition des Méthodes

Une Eméthode est définie, comme nous avons vu dans le chapitre 5, par son contexte de déclenchement, son contexte d'activation, son contexte d'exécution, l'action et l'environnement de propagation.

La définition des Eméthodes se fait par le super-utilisateur en utilisant l'option AddMethodToType du second menu. Le système génère le message Loops suivant :

(<- ($ *Type*) **AddMethod** *NomMeth*)

Envoyé à un type de nom *Type*, le message **AddMethod** permet de déclarer une méthode Elen en envoyant sur l'écran une fenêtre d'édition. Par exemple, pour

associer une méthode de nom Compiler au type Module, la fenêtre d'édition suivante est affichée sur l'écran :

```
SEdit Module.Compiler Package: INTERLISP

( (Method (Compiler Module))
  (OfMode -ImmediateDifferedRefused-
  (WithPriority -PriorityNb-
  (On (-Selector- -ObjectType-))
  (WithArgs  (self origin -OtherArgs-))
  (If -ConditionSpec-)
  (Do -ActionSpec-)
  (Notify -ListOfObjectSpec-))
```

L'utilisateur a la possibilité de déclarer la méthode en remplissant les différentes clauses correspondantes. On dispose pour cela de l'éditeur syntaxique SEdit. A la suite d'une déclaration d'une Eméthode, le système crée un ensemble d'objets et de méthodes Loops pour stocker la déclaration et effectuer les contrôles nécessaires. Nous allons montrer dans les paragraphes qui suivent les effets internes de la déclaration de la méthode Compiler suivante :

```
SEdit Module.Compiler Package: INTERLISP

( (Method (Compiler Module))
  (OfMode Immediate)
  (WithPriority 1)
  (On (Editer Fonction))
  (WithArgs  (self origin))
  (If (EQUAL (<- self GetProp 'CompileFlg) 'T))
  (Do (COMPILE (GetObjectName self))))
```

4.5.1.1. Représentation interne de la condition

L'implantation de la condition se fait par la génération d'une méthode Loops de nom Compiler.Cond. Cette méthode est définie sur la classe VOModule qui modélise les versions de type Module. Si le type Module est défini comme un type sans versions, Compiler.Cond sera définie sur la classe OModule. Les arguments de la méthode Compiler.Cond sont les mêmes que ceux de la méthode Compiler, exprimés par la clause **WithArgs**. Son corps est l'expression de la condition de Compiler exprimée par la clause **If**. La syntaxe Loops de la déclaration de Compiler.Cond est donc la suivante:

```
(DefineMethod ($ VOModule) 'Compiler.Cond '(self origin) '(EQUAL
                    (<- self GetProp 'CompileFlg) 'T))
```

self et *origin* représentent les paramètres formels de la méthode Compile.Condition: *self* désigne l'objet recevant le message Loops qui déclenche la méthode, et origin désigne l'objet origine de l'événement cause du déclenchement.

Remarquons que VOModule est remplacé par OModule dans le cas d'un type Module non versionnable.

En effet, l'activation de la méthode Compiler.Cond est faite automatiquement par le système lors de l'activation de la méthode Compiler. Comme nous allons voir dans §4.5.2, Compiler.Cond retourne une valeur booléenne qui va autoriser ou non l'exécution de l'action.

4.5.1.2. Représentation interne de l'action

Pour représenter l'action de la méthode Compiler, une méthode Loops de nom Compiler.Action est générée. Elle est associée également à la classe VOModule (ou OModule dans le cas d'un objet non versionnable). Les arguments de Compiler.Action sont également ceux exprimés par la clause **WithArgs**, et son corps est l'action spécifiée par la clause **Do**. La syntaxe Loops de la déclaration de Compiler.Action est la suivante :

```
(DefineMethod 'VOModule 'Compiler.Action '(self origin)
                        '(COMPILE (GetObjectName self)))
```

Comme nous allons voir dans §4.5.2, l'exécution de cette méthode est faite automatiquement par le système si l'exécution de Compile.Cond retourne une valeur non nil.

4.5.1.3. Représentation interne des autres clauses de la déclaration

Les clauses OfMode, WithPriority, Notify données par la déclaration de la méthode Compile sont stockées dans un objet de nom Module.Compile instance de la classe prédéfinie **MyMethod**. Cet objet est créé par le système à la suite de la déclaration de Compiler. Ses différents éléments sont utilisés pour le contrôle du mécanisme. Il sert aussi à recevoir les appels d'activation générés automatiquement par le système comme nous allons le voir par la suite.

4.5.2. Contrôle des Méthodes

4.5.2.1. Représentation et gestion de transactions

Nous considérons la classe Loops **Transaction** ayant pour objectif de décrire les transactions créées. Chaque transaction est représentée comme une instance de la classe **Transaction**. La création d'une transaction se fait lors de l'activation d'une méthode en utilisant la primitive **BeginTrans. BeginTrans** a pour effet de créer

une instance de la classe **Transaction** et d'initialiser ses *instances variables* pour désigner la méthode et l'objet origines de la transaction, la liste des arguments utilisée lors de l'activation de cette méthode, la transaction mère.

En utilisant la notion de processus d'Interlisp-D supporté par Loops, nous associons à chaque transaction un processus. Le mécanisme de processus procure un environnement dans lequel plusieurs processus peuvent être exécutés en parallèle.

L'état de la transaction est représenté par une *instance variable* spécifiée par la classe *Transaction*. Elle est évaluée à la fin du traitement de la transaction.

Deux méthodes (**Abort** et **Validate**) sont définies sur la classe **Transaction** pour supporter les primitives nécessaires pour la gestion des transactions imbriquées. **Abort** a pour effet d'annuler la transaction et toutes les transactions ancêtres. **Validate** est utilisée pour terminer une transaction de haut niveau, elle valide aussi toutes ses descendantes.

Nous définissons aussi la méthode **Notify** sur la classe **Transaction**. Notify informe l'objet donné en argument par l'événement responsable de la création d'une transaction.

4.5.2.2. Envoi de messages

Les sélecteurs des Eméthodes définies sur un type de documents apparaissent dans un menu associé aux *cartes* des documents de ce type. L'envoi d'un message Elen à un objet se fait en choisissant son sélecteur dans le menu de l'objet. Ceci peut être fait aussi en utilisant la primitive **SendMessage** prédéfinie par le prototype :

```
(SendMessage NomObjet Sélecteur Args)
```

NomObjet est le nom de l'objet recevant le message, *Sélecteur* est le nom de la méthode et *Args* est la liste d'arguments effectifs à utiliser pour l'exécution du corps de la méthode.

Par exemple, la méthode Compiler déclarée précédemment peut être déclenchée sur un module de nom Interface en envoyant le message Elen suivant :

```
(SendMessage 'Interface 'Compiler)
```

A la suite de l'envoi de ce message, le système crée un événement interne de nom Interface.Compiler dont le type est Module.Compiler et une transaction, instance de la classe Transaction, dans laquelle la méthode Compiler sera activée.

Ensuite la méthode Compiler est activée et l'événement créé est signalé aux objets spécifiés dans la clause Notify.

4.5.2.3. Activation d'une méthode

Si le contexte de l'évaluation d'une méthode (mode, priorité) est vérifiée (voir 4.5.2.4), la méthode est activée. L'activation d'une méthode consiste en l'évaluation de son contexte d'exécution, c'est-à-dire, sa condition. Ceci se fait par le système en envoyant le message Loops prédéfini **Activate** à l'objet représentant la méthode (Module.Compiler dans notre exemple) :

```
(<- ($ Module.Compiler) Activate)
```

Si la méthode activée possède une condition d'exécution, ce qui est le cas dans l'exemple, **Activate** engendre un appel automatique de la méthode Loops correspondante : Compile.Cond, en envoyant le message suivant :

```
(<- ($ Interface) Compiler.Cond)
```

Les paramètres effectifs utilisés lors de l'évaluation de la condition sont portés par l'événement qui active la méthode. Ils sont indiqués par la transaction correspondant à cet événement. Dans le cas d'une activation à la suite d'un envoi de message, ces arguments sont tirés du message envoyé. L'argument *origin* (voir 4.5.1.1) prend la valeur NIL.

Suivant la valeur retournée par l'exécution de Compiler.Cond, nous pouvons distinguer deux cas possibles :

- Compiler.Cond retourne NIL : la transaction Interface.Compiler prend **Refused** comme valeur d'état.
- Compiler.Cond retourne une valeur non nil : la transaction Interface.Comipler prend la valeur **Accepted** comme valeur d'état et la méthode Compiler est alors exécutée.

Ensuite, l'événement interne, Interface.Compiler, résultant de l'activation de la méthode est signalé à tous les objets spécifiés dans la clause Notify en utilisant la primitive **Notify** (voir 4.5.2.5).

4.5.2.4. Exécution d'une méthode

Si le contexte d'exécution d'une méthode est vérifié, la méthode est exécutée. L'exécution d'une méthode se fait en exécutant la méthode représentant son action. Les paramètres utilisés lors de l'évaluation de l'action sont ceux déterminés par le contexte d'exécution. Pour l'exemple précédent, la méthode représentant l'action est Compiler.Action associée aux objets de type Module, elle est appelée par le message:

```
(<- ($ Interface) Compiler.Action)
```

Cette méthode est exécutée dans la transaction Interface.Compiler. Si l'exécution de cette méthode réussit, la transaction prend comme valeur d'état **Succeeded**, dans le cas contraire elle prend la valeur d'état **Failed**.

4.5.2.5. Propagation des événements et déclenchement des méthodes

A la suite de l'exécution d'une méthode, l'événement correspondant est signalé à tous les objets spécifiés dans la clause **Notify**. Ceci est la responsabilité de la transaction, c'est-à-dire, il est fait par la méthode **Notify** définie sur la classe **Transaction**. Prenons l'exemple de la méthode Editer suivante définie sur les objets de type Fonction :

```
SEdit Fonction.Modifier Package: INTERLISP

((Method (Editer Fonction))
 (OfMode Immediate)
 (WithPriority 1)
 (WithArgs (self)
 (Do (DF (GetObjectName self)))
 (Notify (Parents UtilisePar Conception)))
```

L'activation de la méthode Modifier sur la fonction OpenWind crée un événement OpenWind.Modifier et une transaction correspondante. Après l'exécution de cette méthode les objets spécifiés par la clause Notify sont informés; il s'agit des modules ayant OpenWind comme composant, les modules liés à OpenWind par la relation UtiliséPar et des documents de conception liés à OpenWind par la relation Conception, soient, WindManips et ConcOpenWind respectivement.

Pour chacun des objets à informer (WindManips, ConcOpenWind), les méthodes dont le contexte de déclenchement est vérifié (ayant comme contexte de déclenchement le type de l'événement produit) sont déclenchées. Le déclenchement d'une méthode par le système se fait en envoyant le message **trigger** à l'objet sur lequel la méthode est déclenchée:

(<- *Obj* **trigger** *Sel Args TransMère*)

où *Obj* est l'objet sur lequel la méthode est déclenchée, *Sel* est le nom de la méthode déclenchée, *Args* est la liste des paramètres effectifs passés à la méthode lors de son activation, *TransMère* est la transaction de l'événement responsable du déclenchement.

Revenons à notre exemple, et considérons la méthode Compiler définie précédemment sur les objets de type Module et la méthode ModifConc suivante définie sur les documents de conception :

```
SEdit DocConcFonc.ModifConc Package: INTERLISP

( (Method (ModifConc DocConcFonc))
  (OfMode Differed)
  (WithPriority 1)
  (On (Modifier Fonction)
  (WithArgs  (self origin))
  (If (EQUAL (ASKUSER NIL NIL "Voulez-vous modifier
  le document de concepton ?" )'y ))
  (Do (<- self Display)))
```

En effet, le contexte d'activation de ces deux méthodes est vérifié par l'événement produit. Elles sont alors déclenchées ainsi :

(<- ($ WindManips) **trigger** 'Compiler NIL 'OpenWind.Editer)
(<- ($ ConcOpenWind) **trigger** 'ModifConc NIL NIL)

la méthode **trigger** évalue le contexte d'activation des méthodes déclenchées, pour cela elle prend en considération deux cas possibles :

- la transaction a pour état **Refused** : pour chacun des objets informés, les méthodes ayant comme mode **Refuse** sont activées selon leur ordre de priorité. Enfin, la transaction de l'événement déclenchant ces méthodes est abandonné en envoyant le message **Abort** :

  (<- *trans* **Abort**).

  Abort est responsable de la destruction de la transaction et de toutes les transactions filles.

- la transaction a pour état **Succeeded** : pour chacun des objets informés, les méthodes ayant comme mode **Immediate** ou **Deferred** sont activées. Les méthodes de mode **Immédiate** sont activées, selon leur ordre de priorité, dans des transactions filles de la transaction de l'événement responsable de leur activation, c'est le cas de la méthode Compiler qui est activée dans la transaction ConcOpenWind.Modifier, fille de OpenWind.Editer. Si l'une des transactions filles est refusée (condition non vérifiée) ou a échouée, la transaction OpenWind.Editer est abandonnée avec toutes ces filles en lui envoyant le message **Abort**. Dans le cas contraire, elle est validée en lui envoyant le message **Validate**. La validation d'une transaction revient à valider toutes ses filles. Une fois la transaction validée, les méthodes de mode Deferred sont activées dans leur ordre de priorité. L'activation des méthodes **Deferred** se déroule dans des transactions séparées de celle de l'événement origine, et leur abandon ou validation n'a pas de conséquences sur cette transaction. C'est le cas de la méthode ModifConc activée sur ConcOpenWind.

Remarquons que

- si plusieurs méthodes Refused, Immediate ou Deferred sont déclenchées à la fois, elles sont activées selon leur ordre de priorité. Pour les méthodes ayant la même priorité, elles sont activées en parallèle. Cela est réalisé en utilisant la notion de processus d'Interlisp-D qui procure un environnement dans lequel plusieurs processus peuvent être exécutés en parallèle. La création d'un processus en Loops se fait en utilisant (**<-Process** obj sel arg1 ... argn) qui envoie un message tout en créant un processus pour l'exécution de la méthode correspondante.
- Dans l'état actuel du prototype, la méthode **Abort** en abandonnant la transaction, ne permet pas d'annuler toutes les modifications effectuées sur la base à partir du début de la transaction (une telle opération n'est pas possible en Loops!!). Elle envoie simplement un message à l'utilisateur pour l'avertir de l'état d'incohérence.

5. CONCLUSION

Dans ce chapitre, nous avons décrit les fonctionnalités offertes par le prototype réalisé des deux points de vue : utilisation et réalisation. Ces fonctionnalités concernent les gestionnaires de projets, de types, d'objets et de méthodes et l'interface d'utilisation.

Dans son état actuel, le prototype réalise la plupart des fonctionnalités proposées par le modèle de données défini dans les chapitres 4 et 5. Certains aspects proposés du modèle et du langage ne sont pas encore implantés (distinction entre deux types d'utilisateurs : administrateur et utilisateur générale, certaines primitives dans l'expression des conditions et des corps des méthodes tel que recordEvent, etc).

En considérant les limites imposées par la machine utilisée (Xerox) et son environnement (les fichiers sont longs à charger en mémoire centrale, etc), nous pouvons assurer que notre but est atteint. En effet, le prototype, dans son état actuel, montre la faisabilité et l'intérêt de nos propositions. Toutefois, vu l'environnement utilisé, une étude objective de performance ne peut être fournie.

Par ailleurs, comme nous avons montré dans le chapitre 5, le mécanisme de déclencheurs proposé est assez puissant, il est possible, en utilisant ce mécanisme, de gérer les aspects sémantiques du modèle de données (Objets composites, attributs opérationnels, versions, etc). Or, cette version du prototype ne bénéficie pas de cet avantage, et l'utilisation du mécanisme de déclencheurs implanté se limite à l'expression de la dynamique dans les applications.

CONCLUSIONS ET PERSPECTIVES

1. INTRODUCTION

Notre objectif dans ce travail a été d'apporter une aide à la maintenance de logiciels de grande taille en fournissant une base d'objets commune ainsi que des outils d'interrogation. C'est pour cela que nous avons défini un modèle de données pour un système hypertexte capable de gérer simultanément le code source et la documentation associée.

La proposition d'un modèle de données a été motivée par les caractéristiques des informations manipulées en génie logiciel, notamment le volume important et la structure complexe des objets manipulés, l'existence de liens sémantiques entre les différents objets et la nature évolutive et dynamique des objets manipulés, et, aussi, l'insuffisance des propositions faites aussi bien dans le domaine génie logiciel que dans le domaine bases de données.

En effet, dans le domaine du génie logiciel, les solutions proposées pour la gestion de ces informations sont souvent pragmatiques et n'apportent que des solutions partielles. Dans le cas de solutions performantes, il s'agit de mécanismes ad hoc supportés par le système et non intégrés au niveau du modèle de données.

Quant aux systèmes de gestion de bases de données, ils présentent des solutions plus convenables aux problèmes de la cohérence et de l'intégrité de données. Cependant, les différentes propositions ne tiennent pas assez compte de la particularité de l'application génie logiciel. En particulier, les propositions de gestion de la dynamique, bien que généralement satisfaisantes, sont difficilement applicables en génie logiciel à cause des impératifs de performance liés à ce domaine. De plus, l'accès aux données via des requêtes exige une parfaite connaissance de la structure des informations, et dans le cadre d'un grand volume d'informations se révèle insuffisant. C'est pourquoi, nous avons prévu d'autres types d'interrogation, tels que la recherche par le contenu sémantique des documents et la recherche par navigation sur l'ensemble des informations.

Nous pensons que les systèmes hypertexte constituent un support adéquat pour les gestionnaires d'objets dans les EGLs. Ils procurent des mécanismes nécessaires pour la représentation des informations multimédia, et des liens entre les portions arbitraires d'informations ainsi qu'une recherche par navigation. Ils ne sont cependant pas satisfaisants au niveau représentation de la sémantique des données et des mécanismes d'abstractions nécessaires aux objets du génie logiciel. De plus, ils ne sont pas actifs dans le sens qu'ils ne participent pas au processus de maintenance, ni à l'assurance de la cohérence et de l'intégrité des informations manipulées. Ces insuffisances caractérisent les systèmes hypertexte en général et ceux construits particulièrement pour supporter l'application génie logiciel tels que DIF, Neptune, etc. Remédier à ces insuffisances a été un des points centraux de notre travail.

2. CONTRIBUTION DE NOTRE TRAVAIL

Pour cela, nous avons défini un modèle de données pour les systèmes hypertexte qui tienne compte de la particularité des informations manipulées dans un environnement génie logiciel.

Le modèle proposé est un modèle sémantique orienté-objet. Il procure des abstractions nécessaires pour la représentation des objets composites, partageables, typés et liés entre eux. A ce niveau, le modèle de base proposé intègre des notions déjà proposées par d'autres travaux (PCTE+, ORION, EXODUS, ETIC, etc), en l'adaptant à notre contexte. L'intégration de ces différentes abstractions n'est pas supportée par les gestionnaires d'objets les environnements génie logiciel tels que NOMADE, MARVEL, etc.

D'autres aspects sont spécifiques à notre modèle et contribuent à son originalité. En particulier, le concept de *document* représente une abstraction permettant de représenter les objets ayant un contenu textuel, voir graphique. Ce concept tient compte de la représentation du contenu propre aux documents et de leur contenu sémantique (termes d'indexation de structure plus ou moins complexe), notre but étant de pouvoir supporter ultérieurement la recherche par le contenu sémantique des documents.

Le concept de document tient compte aussi de la représentation visuelle des documents et constitue ainsi un support aux aspects hypertexte au niveau du modèle de données.

Le modèle permet la modélisation de versions multiples, non seulement pour les objets atomiques, mais également, à la différence de la plupart des autres propositions, pour les objets composites. Cela donne lieu au concept d'*objets génériques composites* permettant de manipuler les objets composites à des niveaux

de granularité différents. Le modèle fixe un ensemble de règles permettant d'exprimer la sémantique des liens existant entre des objets appartenant à des niveaux différents de granularité.

Une attention toute particulière a été portée sur les aspects dynamiques des informations gérées. Nous avons défini un mécanisme de déclencheurs supporté au niveau du modèle de données. L'utilisation d'une approche objet nous a permis de profiter du couplage entre les données et les traitements.

Contrairement aux autres propositions, telles que NOMADE ou MARVEL, la gestion de la dynamique ici est complètement intégrée dans un modèle orienté-objet. Nous avons utilisé le concept de méthode pour définir non seulement le comportement des objets, mais également, les contrôles nécessaires sur ce comportement. Les méthodes jouent ici le rôle de déclencheurs définis d'une façon déclarative indépendamment du langage de programmation. Elles sont des règles événement-condition-action qui expriment les situations de leur déclenchement, le moment de leur activation, les conditions de l'exécution de l'action associée et l'effet de leur exécution.

En utilisant les méthodes définies de cette façon, il est possible d'exprimer des contrôles de natures différentes représentant les aspects dynamiques des informations manipulées, notamment, les droits d'accès, l'intégrité des informations, la cohérence référentielle, la propagation des effets de bord, etc. Il est également possible d'exprimer les aspects sémantiques du modèle de données lui-même (sémantique d'objets composites, contrôle de versions, sémantique de liens entre les versions, et entre les versions et les objets génériques, etc).

Le mécanisme de déclencheurs proposé repose sur la notion de transactions imbriquées. En effet, la notion de transactions simples utilisée dans [Mar 90] se révèle insuffisante dans le cadre d'une application génie logiciel où les opérations sont longues et coûteuses. Il n'est pas possible de rejeter un ensemble d'opérations coûteuses si elles produisent un objet incohérent. L'incohérence de données à la fin d'une transaction peut être tolérée et gérée dans certains cas.

Nous pensons que l'utilisation d'une approche déductive pour la gestion des déclencheurs, comme c'est le cas dans MARVEL, est difficilement adaptable dans le contexte d'une application génie logiciel. Vu le grand volume de la base et la nature des opérations, une telle approche a des conséquences notables en terme de dégradation de performance.

Par opposition aux autres propositions tels que DAMOKLES et MARVEL, il est possible dans notre proposition, pour des raisons de performance, de limiter la

propagation des effets de bord des déclencheurs à un sous-ensemble d'objets (la notion d'environnement de propagation).

L'originalité du modèle proposé réside aussi dans le fait qu'il est générique dans le sens qu'il n'est pas spécifique à un environnement de programmation particulier ou à une certaine stratégie de développement. Il est possible en utilisant les notions du modèle de définir des schémas spécifiques à un projet ou à un ensemble de projets. Cela est supporté par la notion de *projet* dont l'utilisation permet de spécifier un ensemble de types auxquels sont associées des méthodes précisant la stratégie de développement considéré. Toutefois, la réutilisation des définitions de types est tout à fait possible en donnant la possibilité d'importer des définitions supportées par d'autres projets. Cependant, le modèle tient compte de l'application génie logiciel en fournissant des éléments de base pour gérer du code source, des documents, etc.

3. PERSPECTIVES

Les travaux futurs envisagés peuvent être classés selon deux axes:

- des extensions au travail lui même, soit au niveau du modèle, soit au niveau du prototype.
- des extensions en vue de l'ouverture à d'autres travaux de recherche en cours.

3.1. EXTENSIONS A APPORTER AU MODELE ET AU PROTOTYPE

Notre contribution essentielle dans ce travail est l'intégration au niveau du modèle de données des différents besoins pour une gestion simultanée des programmes et de la documentation associée dans les EGLs. Néanmoins, certains problèmes ne sont pas traités complètement :

• des extensions doivent être apportées au modèle de données, afin de pouvoir supporter la coopération entre plusieurs équipes de travail, le partage d'un projet en sous-projets, la concurrence d'accès aux objets, etc. Nous pensons que la définition de la notion de projet telle qu'elle est donnée est insuffisante. Il faut l'étendre pour supporter les différents aspects mentionnés ci-dessus. Une voie possible est d'étudier l'adaptation des propositions faites par [Fau 88] dans le cadre de la gestion de projets CAO (verrouillage, bases publiques et semi-publiques, vues, etc). Toutefois, en supportant des extensions dans ce sens, il ne faut pas mettre en cause la généricité de nos propositions.

• nous n'avons pas pris en considération les problèmes liés à l'évolution du schéma de la base et de la structure des objets. Or, c'est un aspect important dans un

contexte évolutif. Les travaux effectués dans ORION [Kim 88a] peuvent être bénéfiques à ce niveau.

• dans la définition des méthodes (déclencheurs), le contexte de déclenchement se limite à l'expression d'événements simples. L'expression d'événements complexes (comme c'est le cas par exemple dans HiPAC), ce qui revient à la définition d'opérateurs (conjonction, disjonction, etc) sur les événements, n'a pas été étudiée.

• au niveau de la spécialisation de types, les définitions données ne permettent pas de supporter la notion de spécialisation de méthodes. Ceci étant un aspect très important dans le cadre de la réutilisation en génie logiciel. En effet, on doit pouvoir réutiliser les définitions des méthodes existant tout en les spécialisant. Cela suppose des sémantiques précises données à la spécialisation du contexte de déclenchement, du contexte d'activation, du contexte d'exécution et de l'action d'une méthode.

• quant au contrôle du mécanisme de déclencheurs, ce travail a laissé plusieurs points inexplorés. A l'heure actuelle aucune étude n'a été fournie pour l'optimisation du traitement de déclencheurs en vue d'améliorer la performance du mécanisme proposé. Aussi, le contrôle de la cohérence des déclencheurs et le contrôle de la propagation en détectant les cycles n'ont pas été abordés. Pour ce dernier problème plusieurs types de solutions peuvent être envisagés : une horloge de garde, un catalogue qui stocke et contrôle les événements produits, etc.

• Le mécanisme de déclencheurs proposé est basé sur la notion de transactions imbriquées. Bien que très utile, cette notion n'est pas suffisante dans le cadre de l'application génie logiciel. Les activités développement de logiciels et ainsi les transactions peuvent être très longues (plusieurs heurs, plusieurs jours, etc). Il est intéressant d'étudier la possibilité d'introduire des transactions longues au mécanisme de déclencheurs proposé.

• par rapport au prototype, nous avons montré dans le chapitre 5, qu'en utilisant le mécanisme de déclencheurs, il est possible de gérer les aspects sémantiques de modèle de données. Or, dans la réalisation du prototype cet aspect n'a pas été pris en compte. Ceci est dû à l'historique du développement du prototype; la réalisation des gestionnaires de types et d'objets qui implantent les différents aspects sémantiques a été fait avant d'avoir spécifié complètement le mécanisme de déclencheurs. Sa définition actuelle nous permettrait de réécrire le prototype d'une manière plus déclarative. Une de nos perspectives futures est d'expérimenter cette approche en inversant les rôles. Dans cette vision, le mécanisme de déclencheurs servira comme noyau pour la réalisation des gestionnaires de types et d'objets.

3.2. OUVERTURE A D'AUTRES DOMAINES ET A D'AUTRES TRAVAUX

Nous pouvons identifier deux points essentiels :

• une partie du système Elen s'intéresse à l'interrogation de code source de logiciels. Cette étude [Che 91] est fondée sur l'association, à chaque partie du code source, de descripteurs exprimés dans un modèle issu des graphes conceptuels. Dans le cadre de notre travail il faudrait vérifier que ces descripteurs sont bien supportés par le concept *document* que nous avons défini.

Par ailleurs, un processus d'indexation de documents doit être élaboré pour pouvoir représenter le contenu sémantique de la documentation. Un processus d'indexation en génie logiciel doit prendre en considération l'évolution du corpus (documents existant en versions multiples), et l'aspect "multilanguage" des données (langue naturelle, langage de programmation, langage formel de description, etc). Dans ce cadre l'utilisation du mécanisme de déclencheurs est un support intéressant pour assurer la cohérence entre les documents et leur représentation sémantique.

• l'intégration au niveau du projet Aristote est un de nos objectifs futurs. En effet, ce travail s'insère dans le cadre de l'étude d'applications complexes pour le modèle Aristote. Il est intéressant à ce niveau d'étudier les fonctionnalités qu'il faut intégrer au niveau du modèle Aristote. Du fait qu'une approche générateur d'application a été prise au sein du projet Aristote, l'intégration dans Aristote est possible et réalisable.

BIBLIOGRAPHIE

[Abi 87] ABITBOUL S. & GRUMBACH S., *Bases de Données et Objets Structurés*, TSI, Vol. 6, No. 5, 1987.

[Adi 90] ADIBA M., *Management of Multimedia Complexe Objects in the 90's*, Rapport de Recherche Aristote, Rap008, LGI, Septembre 1990.

[Adi 89] ADIBA M. & COLLET C., *L'approche Objet pour les Bases de Données*, Rapport de Recherche Aristote, Survey-SUR001, LGI, Avril, Révision Septembre 1989.

[Aks 88] AKSCYN R.M., McKRACKEN D.L. & YODER E.A., *KMS: A Distributed Hypermedia System For Managing Knoledge in Organisations*, Com. of the ACM, Vol. 31, No. 7, Juillet 1988.

[Ald 86] ALDERSON A., BOTT M.F. & FALLA M.E., *An Ovarview of Eclipse, In Integrated Project Support Environments*, Ed. J. McDermid, Peter Peregrinus, 1986.

[Alf 80] ALFORD M.W., *Software Requirement Engineering Methodology (SERM), at the Age of FOUR*, Proc. COMPSAC 80, 1980.

[ANS 89] ANSI-X3H2-89-110/ISO/DBL CAN_3, *Database Language SQL2 and SQL3 Report*, February 1989.

[Ari 91] *Projet Aristote*, Rapport Intermédiaire, Rapport de Recherche Aristote, LGI, Juin 1991.

[Bal 85] BALZER R., *A 15 Years Perspectives on Automatic Programming*, IEEE Transactions on Software Engineering, Vol. SE-11, N° 11, Novembre, 1985.

[Ban 88b] BANCILHON F. et al., *The Design and Implementation of O2, an Object-Oriented Database System*, Altaïr Technical Report 20-88, Avril 13, 1988.

[Ban 87a] BANCILHON F., *Object-Oriented Database Systems*, Rapport Altaïr 14-87, Novembre 1987.

[Ban 87b] BANERJEE J. et al, *Data Model Issues for Object-Oriented Applications*, ACM Trans. on Office Information Systems, Vol. 5, January 1987.

[Ban 87c] BANERJEE J. & KIM W., *Semantic and implementation of Schema Evolution in Object-Oriented databases*, ACM-SIGMOD'87, 1987

[Bas 88] BASILI V.R. & ROMBACH H.D., *The TAME Project: Towords Improvement-Oriented Software Environments*, IEEE Transaction on Software Engineering, Vol. 14, No. 6, Juin 1988.

[Bau 88] BAUMANN P. & KOHLER D., *Archiving Versions and Configurations in database System for Software Engineering Environments*, Proc. of the Int. Workshop on software Version and configuration Control, Grassau, Janvier 27-29, 1988.

[Bax 86] BAXTER I. & al., *TMM: Software Maintenence by Transformation*, IEEE Software, Mai, 1986.

[Bel 88] BELKHATIR N., *Nomade : un Noyau d'environnement pour la programmantion Globale*, Thèse LGI-IMAG, Decembre,1988.

[Bel 92] BELKHATIR N. & ESTUBLIER J., *Supporting Software Maintenance Evolution Processes in the Adele System*, Proc. of the 30th annual ACM Southeast Conference , Raleigh, NC, Avril 8-10, 1992.

[Ben 87] BENDIFALLAH S. & SCACCHI W., *Understanding Software Maintenance Work*, IEEE Trans. on soft. Eng., Vol. SE-13, NO. 3, March 1987.

[Ber 90] BERRUT C., BRUANDET M.F., CHEVAL J.L., DECHAMBOUX P., JARWAH S. & MARTIN H., *La Gestion de la Dynamique : Systèmes de Gestion de Bases de Données vs Systèmes d'Intelligence Artificielle*, Rapport de Recherche Aristote, Survey-SUR003, LGI, Octobre 1990.

[Ber 89] BERNSTEIN D.B., SMOLENSKY & BELL B., *Design of a Constraint-Based Hypertext System to Augment human reasoning*, CU-CS-423-89, departement of Computer Science, University of Colorado, Janvier 1989.

[Ber 87] BERNSTEIN P.A., *Database System Support for Software Engineering -An Extended Abstract-*, ACM 1987.

[Ber 84] BERSOFF H.E., *Elements of Software Configuration Management*, IEEE Trans. on Soft. Eng., Vol. SE-10, NO. 1, January 1984.

[Big 88] BIGELOW J., Hypertext and CASE, IEEE Software, Mars 1988.

[Big 87] BIGELOW J., RILEY V., *Manipulating Source Code in DynamicDesign*, Hypertext'87 Papers, Chapel Hill, NC, Nov. 13-15, 1987.

[Bla 85] BLAIR D.C. & MARON M.E., *An Evaluation of Retrieval Effectiveness for Full-Text Document Retrieval System*, Communication of the ACM, Vol. 28, N0. 3, Mars 1985.

[Blu 88] BLUM B.I., *Documentation for Maintenance : A Hypetext design*, Proc. Conference on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[Boe 88] BOEHM B.W., *A Spiral Model for Software Development and Enhancement*, Computer, Mai, 1988.

[Boe 82] BOEHM B.W., *Les Facteurs du Coût du Logiciel*, TSI, Vol.1, N°1, 1982.

[Bra 84] BRANSTAD M. & POWELL BP.B., *Software Engineering Standards*, IEEE Transactions on software Engineering, Vol. SE-10, NO. 1, Janvier 1984.

[Bri 90] BRISSAUD F. & JIRAUDIN J.P., *Les Associations dans les Modèles de Données*, Rapport de Recherche Aristote, Survey-SUR002, LGI, Janvier 1990.

[Bru 90] BRUZA P.D. & WEIDE Th.P., *Two Level Hypermedia An Improved Architecture for Hypertext*, Proc. of the Database and Expert System Applications, DEXA'90, Springer Verlage, Vienne, Autriche, Septembre 1990.

[Buc 45] BUCH V., *As we May Think*, The Atlantic Journal, 176(1), 1945.

[Buc 85] BUCKLE J. K., *Software Configuration Management*, Ed. MacMillan Education LTD; England, 1985.

[CAI 85] *CAIS. Rational for the DoD Requirements and Design Criteria for the Comon APSE Interface set (CAIS)*, Institute of Defence Analyses, US DoD 1801 North Beauregard Street, Alexandria, Virginia 22311, 1985.

[Cal 88] CALLISS F.W., KHALIL M., MUNRO M. & WARD M., *A Knoledge-Based System for Software Maintenance*, Proc. Conference on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[Cam 89] CAMPAGNONI F.R. & EHRLICH K., *Information Retrieval Using a Hypetext-based Help System*, Proc. of the Twelfth Annual Int. ACM-SIGIR Conference on Research and Development in information Retrieval, Cambridge, Massachusetts USA, Juin 25-28, 1989.

[Cam 91] CAMPBELL J.C. & al, *ALF Project*, ALF/UDO-VG/WP-4-4-4-D1, Decembre 1991.

[Cam 87] CAMPBELL B. & GOODMANN J.M., *HAM : A General Purpose Hypertext Abstract Machine*, Hypertext'87 Parers, Chapel Hill, NC, Nov. 13-15, 1987.

[Car 89] CARANDO P., SHADOW Fusing Hypertext with AI, IEEE Expert, hiver 1989.

[Car 88] CAREY M.J., DEWITT D.J., VANDENBERG S.L., *A Data Model and Query Language for EXODUS*, Proc. ACM SIGMOD, Chicago, June 1-3, 1988.

[Cas 89] CASEAU Y., A Formal System for Producing Demons From Rules in an Object-Oriented Databases, Proc. of the DOOD'89, 1989.

[Cha 88] CHAPIN N., *Software Maintenance Life Cycle*, Proc. Conference on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[Che 91] CHEVALLET J.P., ELEN: Un Système d'Interrogation d'une Base de Logiciels, INFORSID'91, Paris, 4-7 juin, 1991.

[Che 76] CHEN P., *The Entity Relationship Model-Toward a Unified View of Data*, ACM Transaction on databases Systems, Vol. 1, No. 1, March 1976, PP. 9-36.

[Chi 86] CHIARAMELLA Y., BRUANDET M. F., DEFUDE B., KERKOUBA D., *IOTA: a Full-Text Information Retrieval System*, in Proc. of SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy, 1986.

[Coe 85] COEFFREY James, *Document Databases*, Van Nostrand Reinhold Company (ed.), NewYork, 1985.

[Col 87a] COLLET C., *Les Formulaire Complexe dans les les Bases de Données Multimedia*, Thèse de Doctorat de l'Université de Grenoble, Novembre 1987.

[Col 87b] COLLIER G.H., *Thoth-II: Hypertext with Explicit Semantics*, Proceedings of tthe Hypertext'87 Workshop, Chapil Hill , North California, Novembre13-15, 1987.

[Con 87] CONKLIN J., *Hypertext : Introduction and Survey*, IEEE Computer, Vol. 20, n°9, Septembre 1987.

[Con 90a] CONRADI R. & HOLAGER P., *Change-Oriented Versionning: Rational and Evaluation*, Thierd International Workshop, Software Engineering and its Applications, Toulouse, 1990.

[Con 90b] CONRADI R., OSJORD E., WESTBY P.H. & LIU C., *Software Process Management in Epos: Design and Initial Implementation*, Thierd International Workshop, Software Engineering and its Applications, Toulouse, 1990.

[Cro 89a] CROFT W.B. & TURTLE H., *A Retrieval Model for Incorporating Hypertext Links*, Proc Hypertext '89, November 1989.

[Cro 89b] CROUCH D.B., CROUCH C.J. & ANDREAS G., *The Use of Cluster Hierarchies in Hypertext Information Retrieval*, Proc. Hypertext'89, Novembre 1989.

[Dam 88] DAMIER C. & DEFUDE B., *Un Modèle de Données pour les Applications Géographiques*, 4ième Journée des Bases de Données Avancées, Bénodet, Mai 1988.

[Dan 90] DANIEL-VATTON M.C., Hypertextes : des Principes Communs et des Variations, TSI, Vol. 9, No. 6, 1990.

[Day 88] DAYAL U., BUCHMANN A.P. & McCARTHY D.R., *Rules are Objects Too: A Knoledge Model For An Active, Object-Oriented Database System*, Lecture Notes in Computer Science, Septembre 1988.

[Del 86] DELISLE N.M. & SCHWARTZ M.D., *Neptune: a Hypertext System for CAD Applications*, Proc. of Int. Conf. on Management of Data, Washington, D. C., Mai 22-30, ACM, New York, 1986, pp 132-143.

[Del 87] DELISLE N. & SCHWARTZ M., *Contexts- A Partitioning Concept for Hypertext*, ACM Trzn. on Office Information Systems, Vol. 5, N° 2, April 1987.

[Del 82] DELOBEL C. & ADIBA M., *Bases de Données et Systèmes Relationnels*, Dunod Informatiques, 1982.

[Der 91] DERNIAME J.C. & ZUCKER J.D., *Le Système ALF ou La Construction D'Ateliers de Logiciel en Partant de modèles de Procédes*, UNIVERDUSTRIE 91, FIRTECH, Nancy 1991.

[Der 76] DE REMER F. & KRON H.H., *Programming in the Larg Versus Programming in the Small*, IEEE Trans. on Soft. Eng., SE-2(2), Juin 1976.

[Dev 90] DEVANBU P., BRACHMAN R.J., SELFRIDGE P.G. & BALLARD B.W., *LaSSIE: a Knoledge-Bases Software Information System*, IEEE 1990.

[Dit 87] DITTRICH K.R., GOTTARD W. & LOCKEMANN P.C., *DAMOKLES _ A Database System for Software Engineering Environments*, Proc. of int. workshop on advanced Prog. Environment, Trondheim Norway, 16-18 Juin, 1986. Springler Verlag (ed.)LNCS 244, Fevrier 1987.

[Dit 86] DITTRICH K.R., KOTZ K.R. & MULLE A.M., *An Event/Trigger Mechanisme to Enforce Complexe Consistency Contraints in Design Databases*, SIGMOD Record, Vol 15, No 3, 1986.

[Dit 85] DITTRICH K.R., LORIE R.A., *Version Support for Engineering Database Systems*, Research Report, IBM Research Laboratory, San Jose, California 95193.

[Eas 82] EASTMAN C. & LAFUE G., *Semantic Integrity transactions in Design Databases*, File Structures and Data Bases for CAD, novembre 1989.

[Eng 63] ENGELBART D.C., *A Conceptual Framework for the Augmentation of Man's Intellect. In Vistas in Information Handling*, Vol.1, P. D. Howerton & D. C. Weeks, eds. Spartan Books, Washington, D.C., 1963, pp. 1-29.

[EUR 87] *EURAC. Requirements and Design Criteria for Tool Support Interface (EURAC)*, GIE Emeraude and Selenia and Software Science Limited 1987 Version 3, 1987.

[Fau 91] FAUVET M.C., *Modeling and Managing Histories in an Object Oriented Environment*, Rapport de Recherche Aristote, RAP015, LGI, Septembre 1991.

[Fau 88] FAUVET M.C., *ETIC : Un SGBD pour la CAO dans un Environnement Partagé*, Thèse de Doctorat de l'Université Joseph Fourier de Greneble(1), Sept, 1988.

[Fle 88] FLETTON N.T. & MUNRO M., *Redocumenting Software Systems Using Hypertext Technology*, Proc. Conf. on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[Fur 90] FURUTA R. & STOTTS P.D., *Generalising Hypertext : Domains of the Trellis Model*, T.S.I, Vol. 9, No. 6, 1990.

[Gal 86] GALLO F., MINOT R. & THOMAS I., *The Object Management System of PCTE as a Software Engineering Database Management System*, Proc. of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software development, Palo Alto, California, Decembre 9-11, 1986.

[Gam 88] GAMALEL-DIN S.A. & OSTERWEIL L.J., *New Perspectives on Software Maintenance Processes*, Proc. Conf. on Software maintenance, Phoenix, Arizona, October 24-27, 1988.

[Gan 90] GANTI M., GOYAL P. & PODAR S., *An Object-Oriented Software Application Architecture*, IEEE, 1990.

[Gar 90] GARG P.G. & SCACCHI W., *A Hypertext System to Manage Software Life-Cycle Documents*, IEEE Software, Vol. 7, N. 3, Mai 1990.

[Gar 88a] GARG P. K., *Abstraction Mechanisms in Hypertext*, Comm. of the ACM, Vol 31, No. 7, Juillet 1988.

[Gar 88b] GARG P.K. & SCACCHI W., *A Software Hypertext Environment*, Int. Workshop on Software Version and Configuration Control, ACM, Janvier 1988, Grassau FRG.

[Gar 87] GARG P.K. & SCACCHI W., *On Designing Intelligent Hypertext System for Information Management in Software Engineering*, Proc. of the Hypertext'87 Workshop, Chapil Hill , North California, Nov.13-15, 1987.

[GEC 86] *Software Engineering Handbook*, Staff of General Electric Company, McGraw-Hill, 1986.

[God 92] GODART C. & CHAROY F., *Bases de Données pour le Génie Logiciel*, Masson Paris Milan Barcelone Bonn, 1992.

[Gor 86] GORDA U., FACCHETTI G., *Concept Browser: A System for Interactive Creation of Dynamic Documentation*, Text Processing and Document Manipulation: Proc. of the Int. Conf., Ed. Van Vliet J.C., Campridge University Press, 1986.

[Gre 83] GREGORY R., *XANADU Hypertext from the Future*, Dr. Dobb's Journal, No. 75, Janvier 1983.

[Gut 77] GUTTAG J., *Abstract Data Types and the development of Data Structures*, Comm. of ACM, vol 20, N° 6, Juin 1977.

[Hab 86] HABERMANN A.N., NOTKIN D., *Gandalf: Software Development Environment*, IEEE Trans. on Soft. Eng., Vol. SE-12, N° 12, December 1986.

[Hal 90] HALASZ F. & SCHWARZ, *The Dexter Hypertext Reference Model*; in Hypertexte Standardisation Workshop, Janvier 1990.

[Hal 88] HALASZ F., *Reflections on NoteCards: Seven Isues for the Next Generation of Hypermedia System*; Communications of the ACM, Vol. 31, No. 7, Juillet 1988.

[Han 89] HANSON E.N., *An Initial Report on the Design of Ariel: a DBMS with an Integrated Production Rule System*, SIGMOD RECORD, Vol. 18, No. 3, Septembre 1989.

[Har 85] HARPER D. J., DUNNION J., SHERWOOD-SMITH M.& RIJSBURGEN V., *Mistral-ODM: A Basic Database Model,* in Information Processing and Management, Vol. 22, N° 2, 1986, pp. 83-107.

[Hec 82] HECHT H., *Requirement Documentation- A Management Oriented Approach*, In Proc. of the NBS FIPS Software Documentation Workshop, Mar. 3, 1982, at NBS, Gaithersburg, MD, Ed. Institut for Computer Sciences and Technology, National Bureau of Standards, Washington, DC20234.

[Hoh 88] HOHENSTEIN U. & GOGOLLA M., *A Calculus for Extended Entity-Relationship Model Incorporaating Arbitrary Data Operations and Aggregate Functions*, Proc. of the seventh International Conference on Entity relationship Approach, Ed. Carlo Batini, Roma, Italy, Novembre 16-18, 1988.

[Hor 86] HOROWITZ E. & WILLIAMSON R.C. , *SODOS: A software Documentation Support Environment- Its Definition*, IEEE Transaction on Soft. Eng. Vol. SE-12, n°8, Aout 1986.

[Hor 84] HOROWITZ E. & WILLIAMSON R.C. , *SODOS: A software Documentation Support Environment- Its Use*, Dep. Comput. Sci, Univ. Southern California, Los Angeles, Tech. Paper TR-84-313, 1984.

[Hud 88] HUDSON S.E. & KING R., *The Cactis Project: Database Support for Software Environments*, IEEE Trans. on Soft. Eng., Vol. 14, No. 6, Juin 1989.

[Hud 87] HUDSON S.E. & KING R., *Object-Oriented database Support for Software Environments*, ACM-SIGMOD'87.

[Huf 88] HUFFMAN J.E. & BURGESS C.G., *Partially Automated In-Line Documentation (PAID) : Design and Implementation of a Software Maintenance Tool*, Proc. Conference on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[INR 83] Actes des journées sur la manipulation de documents, Rennes 4-6,Mai 1983, préparés par Jaques André, INRIA.

[Jar 92] JARWAH S. & BRUANDET M.F., *ELEN Prototype: an Active System for Document Management in Software Engineering*, Proc. DEXA'92, Valence, Espagne, 2-4 Septembre 1992.

[Jar 90a] JARWAH S. & BRUANDET M.F., *An Object-Oriented Model for Hypertext Databases: Application to Document Management in Software Engineering*, Rapport de Recherche Aristote, RAP004, LGI, Janvier 1990.

[Jar 90b] JARWAH S. & BRUANDET M.F., *Hypertexte Database Model for Information Management in Software Engineering*, Proc. DEXA'90, Vienne, Autriche, 29-31 Août 1990.

[Jar 89] JARWAH S. & CHEVALLET J.P., *Spécification d'ELEN, un système pour la gestion et l'interrogation de document et de logiciel*, Second International workshop, Software Engineering and its Applications, Toulouse, 4-8 decembre, 1989.

[Kai 88] KAISER G.E. & BARGHOUTI N.S., *Database Support for Knoledge-Based Engineering Environments*, IEEE Expert, été 1988.

[Kai 87] KAISER G.E., FEILER P.H., *An Architecture for Intelligent Assistance in Software Development*, 9th International Conference on Software Engineering , San Francisco, CA, Mars, 1987.

[Kat 90] KATZ R.H., *Toward a Unified Framework for Vesion Modeling in Engineering Databases*, ACM Computing Surveys, Vol. 22, No. 4, Decembre 1990.

[Kat 85] KATZ R.H., *Information Management for Engineering Design*, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

[Kat 84] KATZ R. H. & LEHMAN T. J., *Database Support for Version and Alternatives of Large Design Files*, IEEE Trans. on Software Engineering, SE-10(2), Mars 1984, pp 191-200.

[Kat 82] KATZ R.H., *A Database Approach for Managing VLSI Design Data*, In Proc. 19th ACM/IEEE Design Automat. Conf., Las Vegas, NV, Juin 1982.

[Kem 87] KEMPER A., LOCKMANN P.C. & WALLRATH M., *An Object-Oriented Database System for Engineering Applications*, ACM-SIGMOD'87.

[Ket 88] KETABCHI M.A., BERZINS V., *Mathematical Model of Composite Objects and Its Application for Organizing Engineering Databases*, IEEE Trans. on Soft. Eng., Vol 14, NO.1, Janvier 1988.

[Kim 89] KIM W., BERTINO E. & Garza J.F., *Composite Object Revisited*, Proc. ACM-SIGMOD'89.

[Kim 88a] KIM W. & CHOU H., *Version of Schema for Object-Oriented Databases*, Conference VLDB, Los Angeles, Août, 1988.

[Kim 88b] KIM W. & al, *Integrating an Object-Oriented Programming with a database System*, Proc. OOPSLA'88 Septembre 25-30, 1988.

[Kim 88c] KIM W., CHOU H. & BANERJEE J., *Operation and Implementation of Complex Objects*, IEEE Trans. on soft. Eng., SE-14, NO. 7, Juillet 1988.

[Kim 87] KIM W. et al., *Composite Object Support in an Object-Oriented Database System*, Proc. 2nd OOPSLA Conf., Orlando, FL, 1987.

[Kim 85] KIM W. & BATORY D.S., *A Model and Strorage Technique for Versions of VLSI CAD Objects*, Pro. of the Inter. Conf. on Foundations of Data Organisation, KYOTO, Japan, May 21-24, 1985.

[Kot 88] KOTZ A., DITTRICH K.R. & MULLE J.A., *Supporting Semantic Rules by a Generalized Event/Trigget Mechanism*, Proc. EDBT'88, Venice, Italy, Mars 14-18, 1988.

[Kra 82] KRAKOWIAK S., *Systèmes intégrés de production de logiciel : concepts et réalisations*, TSI, Vol. 1, N°. 3, 1982.

[Lam 88] LAMSWEERDE A.V., DELCOURT B., DELOR E., SCHAYES M.C. & CHAMPAGNE R., *Generic Lifecycle Support in the ALMA Environment, IEEE Transaction on Software Engineering*, Vol. 14, No. 6, Juin 1988.

[Lan 90] LANGE D.B., *A Formal Model for Hypertext*, Hypertext Standardisation Workshop, Janvier 1990.

[Lan 88] LANDIS L.D., HYLAND P.M., GILBERT A.L. & FINE A.J., *Documentation in a Software Maintenance Environment*, Proc. Conference on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[Leb 84] LEBLANG D.B. & CHASE R.P., *Computer-Aided Software Engineering in a Destributed Workstation Environment*, Proc. of the ACM SIGPLAN SIGSOFT, Soft.Eng. Symposium on Practical Soft. Development Environment, April 1984.

[Lec 87] LECLUSE C., RICHARD P. & VELEZ F., *O2, an Object Oriented Data Model*, Rapport Altaïr 10-87, 15 Septembre, 1987.

[Leg 89] LEGAIT A., OQUENDO F & OLDFIELD D., *MASP: A Model for Assisted Software Processes*, Lecture Notes in Computer Science 467, edited by G. Goos & J. Hartmanis, FredLong(Ed.). Proc. Soft. Eng. Envi, International Worckshop on Environments, Chinon, France, Septembre, 1989.

[Loo 88] *ENVOS LOOPS, Lyric/Medley Release*, User Manual, Xerox PARC, Juillet 1988.

[Lu 90] LU X., *Document Retrieval: A Structural Approach*, Information Processing and Management, Vol 26, NO. 2, PP. 209-218, 1990.

[Maa 89] MAAREK Y.S. & SMADJA F.A., *Full Text Indexing Based on Lexical Relations an Application: Software Libraries*, Proc. of the Twelfth Annual Int. ACM-SIGIR

Conference on Research and Development in information Retrieval, Cambridge, Massachusetts USA, Juin 25-28, 1989.

[Mac 90] MACLEOD I.A., *Strorage and Retrieval of Structured Documents*, Information Processing and Management, Vol. 26, NO. 2, PP. 209-218, 1990.

[Mar 91] MARTIN H., *Contrôle de la Cohérence dans les Bases Objets: Une approche par le Comportement*, Thèse de Doctorat, Université Joseph Fourier, Grenoble, Juillet 1991.

[Mar 88] MARCHIONINI G. & SHNEIDERMAN B., *Finding Facts vs. Browsing Knoledge in Hypertext Systems*, IEEE Computer, Janvier 1988.

[Mar 86] MARZULLO K., *Jasmine: A Software System Modelling Facility*, Proc. of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software development, Palo Alto, califomia, Decembre 9-11, 1986.

[Mas 89] MASINI et al., *Les Langages à Objets*, Interedition (ed.), Paris, 1989.

[McC 89] McCARTHY D.R. & DAYAL U., *The Architecture of a Database management System*, ACM SIGMOD'89.

[Mit 89] MITSCHANG B., *Extending the Renational Algebra to Capture Complex Objects*, Proc. of the Fiftennth Inter. Conf. on Very Large Data Bases, Amsterdam, 1989.

[Moo 76] MOORE J.S., *The Interlisp Virtual Machine Specification*, Xerox PARC, CSL-76-5, 1976.

[Mos 82] MOSS J.E., *Nested Transactions and reliable distributed computing*, Proc 2nd Symposium on Reliability in Distributed Software and Databases Systems, 1982.

[Mul 87] MULLIN D., *FORTUNE- A Documentation Support System for Software Engineers*, Proc. of the 1st ESEC , Strasbourg, 1987.

[Nav 88] NAVATHE S.B. & PILLALAMARRI M.K., *OOER: Toward Making the E-R Approach Object-Oriented*, Proc. of the Seventh International Conference on Entity relationship Approach, Ed. Carlo Batini, Roma, Italy, Nov 16-18, 1988.

[Nel 81] NELSON T.H., *Literary Machin*, T. N. Nelson, Swarthmore, PA., 1981.

[Nes 86] NESTOR J.R., *Toward a Persistent Object Base*, Proc. of Int. Workshop on Advanced Prog. Environment, Trondheim Norway, 16-18 Juin, 1986. Springler Verlag (ed.)LNCS 244, Fevrier 1987.

[Nie 90] NIELSEN J., *The Art of Navigation through Hypertext*, Communication of the ACM, Vol. 33, NO. 3, Mars 1990.

[Obe 88] OBERNDORF P.A., *The Common ADA Programming Support Environment (APSE) Interface Set (CAIS)*, IEEE Transaction on Software Engineering, Vol. 14, No. 6, Juin 1988.

[Oko 82] O'KORN L.J., *System Development Methodology and Documentation Practices*, In Proc. of the NBS FIPS Software Documentation Workshop, Mars 3, 1982, at NBS, Gaithersburg, MD, Ed. Institut for Computer Sciences and Technology, National Bureau of Standards, Washington, DC20234.

[Oma 90] OMAN P., *Maintenance Tools*, IEEE Software, Vol. 7, N. 3, Mai 1990.

[Onu 85] ONUGBE E., *Functional Requierement for Software Engineering Database Management System*, Computer Science Forum, Honeywell, Vol. 9, No. 1, Janvier 1985.

[Oqu 90] OQUENDO F.S., *Contribution à l'Etude des Bases de Données pour le Génie Logiciel, Modèle et Fonctionnalités d'un Système de Gestion d'Objets pour Environnement de Génie Logiciel Assisté par Ordinateur*, Thèse de Doctorat en Informatique de l'Université de Grenoble II, Octobre 1990.

[Oqu 89] OQUENDO F. & al, *Modeling Composite Objects in a Software Engineering Object Management System*, Proc. Thierd International Workshop on Computer-Aided Software Engineering London, England, Juillet 1989.

[Ore 87] OREN T., *The Architecture of Hypertexts*, In Hypertext'87 Papers, Chapill Hill, NC, 1987, Novembre 13-15.

[Ozs 87] OZSOYOGLU, YUAN L.Y., *A Normal Form for Nested Relations*, ACM TODS, Vol 12, No 1, Mars 1987.

[Par 90] PARISI-PRESICCE F., *A Rule-Based Approach to Modular System design*, IEEE 1990.

[Pau 88] PAUL. & NEGRET J.M., *SOFTM : A Software Maintenance Expert System in Prolog*, Proc. Conference on Software maintenance, Phoenix, Arizona, Octobre 24-27, 1988.

[Pen 89] PENEDO M.H. & STUCLE E.D., *TRW's SEE Saga*, Lecture Notes in Computer Science 467, Edited By G.Goos & J. Hartmanis, FredLong(Ed.). Proc. Soft. Eng. Environment Inter. Workshop on Environments, Chinon, France, Septembre 1989.

[Pen 86] PENEDO M.H., *Prototyping a Project Master Data Base for Software Engineering Environment*, Proc. of int. workshop on advanced Prog. Environment, Trondheim Norway, 16-18 Juin, 1986. Springler Verlag (ed.)LNCS 244, Fevrier 1987.

[Pen 85] PENEDO M.H. & STUKLE E.D., *PMDB: A project Master Database Software Engineering Environment*, Proc. of the Int. Conf. Software Engineering, CSPress, Los Mamitoc, Calif., Aug., 1985, pp. 150-157.

[Per 88] PERRY D.E. & KAISER G.E., *Models of Software Development Environments*, IEEE Sftware, Mars 1988.

[Pil 87] PILLAMARRI M.K., *Toward a Semantic Data Model Based on Object-Oriented and Entity-Relationship Concepts*, M. S., Departement of Electrical Engineering, University of Florida, Decembre 1987.

[Pre 87] PRESSMANN R.S., *Software Engineering A Partitionner's Approach*, 2eme Edition, Roger S. Pressmann, McGraw-Hill, 1987.

[Pri 87] PRIETO-DIAZ R. & FREEMAN P., *Classifying Software for Reusability*, IEEE Software, Janvier, 1987.

[Pun 88] PUNCELLO P. & al, *ASPIS: A Knoledge-Bases CASE Environment*, IEEE Software, Mars, 1988.

[Ram 88] RAMANATHAN J., SARKAR S., *Providing Customarized Assistance for Software Life cycle Approaches*, IEEE Transaction on Software Engineering, Vol. 14, No. 6, Juin 1988.

[Rav 90] RAVALET D. & BRIAND H., *Transformation d'un Schema Entité-Association En Base de Données Orientées-Objets*, Thierd International Workshop, Software Engineering and its Applications, Toulouse, 1990.

[Rei 90] REISMAN S., *Management and Integrated Tools*, IEEE Software, Vol. 7, N. 3, Mai 1990.

[Ric 90] RICHARD G. & RIZK A., *Quelques idées pour une modélisation des systèmes hypertextes*, TSI, Vol. 9, No. 6, 1990.

[Ris 89] RISCH T., *Monitoring Database Objects*, Proc. of the Fifteenth Inter. Conf. on Very Large Data Bases, Amsterdam, 1989.

[Rob 81] ROBERSTON G., McCRACKEN D.L. & NEWELL, *The ZOG Approach to Man-Machine Communication.*, Inter-national Journal of Man-Machine Studies, 14, 1981.

[Roh 88] ROHRBACH R. & SEIWALD C., *Galilio : A Software Maintenance Environment*, Proc. of the Int. Workshop on Soft. Version and Configuration Control, Grassau, Janvier 27-29, 1988.

[Ros 89] ROSENTHAL A., CHAKRAVARTY S. & BLAUSTEIN B., *Situation Monitoring for Active Databases*, Proc. of the Fifteenth Inter. Conf. on Very Large Data Bases, Amsterdam, 1989.

[Rud 86] RUDMIK A., *Choosing an Environment Data model*, Proc. of Int. Workshop on Advanced Prog. Environment, Trondheim Norway, 16-18 Juin, 1986. Springler Verlag (ed.)LNCS 244, February 1987.

[Sal 89] SALTON G., Buckley C., *On the Automatic Generation of Content Links in Hypertext*, TR 89-993, Technical Report, Departement of Computer Science cornell University Ithaca, NY, Avril, 1989.

[Sal 83] SALTON G., McGILL M. J., *Introduction to Modern Information Retrieval*, Mcgraw Hill Book Company, New York, 1983.

[San 89] SANDVAD E., *Hypertext in an Object-Oriented Environment*, Proc. WOODMAN'89, BIGRE63-64, Mai 1989.

[Sar 88] SARNAK N., BERNSTEIN R. & KRUSKAL V., *Creation and Maintenance of Multiple Versions*, Proc. of the Int. Workshop on Soft. Version and Configuration Control, Grassau, January 27-29, 1988.

[Sav 91] SAVOY J., *Multiple Indexing Schemes and Multiple Search Strategies in Hypertext Systems*, Rapport n° 765, Universioté de Montréal, Avril 1991.

[Sav 90] SAVOY J., *Les Sources des Hypertextes*, Une bibliographie Commentée, TSI, Vol. 9, No. 6, 1990.

[Sav 89] SAVOY J., *Hypertexte : Concepts et Problèmes*, Rapport de Recherche, Université de Montréal, 1989.

[Sca 88] SCACCHI W., *Modeling Software Evolution : A Knoledge-Based Approach*, Proc. of the 4th Int. Software Process Workshop, Moretonhampstead, Devon, UK, 11-13 Mai 1988, Colin Tully (ed.).

[Sch 90] SCHWARTZ C., *Content Based Text Handling*, Information Processing and Management, Vol 26, NO. 2, PP. 209-218, 1990.

[Sch 88] SCHWANKE R.W. & KAISER G.E., *Living with Inconsistency in Large Systems*, Proc. of the Int. Workshop on Software Version and Configuration Control, Grassau, Janvier 27-29, 1988.

[Sco 88] SCOTT E.H. & KING R., *The Cactis Project: Database Support for Software Environmrnts*, IEEE Transaction on Software Engineering, Vol. 14, No. 6, Juin 1988.

[SEE 89] *The Software Engineering Environments*, Research and Practice, editor Keith H. Benett, 1989.

[Smi 88] SMITH J.B. & WEISS S.F., *Hypertext*, Comm. of the ACM, Vol. 31, N°. 7, Juillet 1988.

[Smi 87] SMITH K.E. & ZDONIK S.B., *Intemedia: A Case Study of the Differences Between Relational and Object-Oriented Database System*, In Proc. of the Conf. on Object-Oriented Database Systems, Langages and Applications (OOPSLA'87) Orlando, FL, Octobre 4-8, 1987.

[Smi 77] SMITH J.M. & SMITH D.C.P., *Database Abstractions: Aggregation and Generalisation*, ACM Transactions on Databases Systems,Vol. 2, N° 2.

[Sny 86] SNYDER A., *Encapsulation and Inheritance in Object-Oriented Programming Languages*, OOPSLA'86 Proceedings, Sepembre 1986.

[Sto 89] STOTTS P.D. & FURUTA R., *Petri-Net-Based Hypertext: Document Structure with Browsing Semantics*; ACM Trans. on Information Systems, Vol. 7, No. 1, Janvier 1989, pp 3-29.

[Tei 77] TEICHROW D. & HERSHEY E., *PSL/PSA: A Computer Aided Technic for Structured Documentation and Analysis of Information Processing Systems.* IEEE Trans. on Soft. Eng. SE-3, 1, Janvier 1977, pp. 41-48.

[Tic 88] TICHY W.F., *Tools for Software Configuration Management*, Proc. of the Int. Workshop on software Version and configuration Control, Grassau, Janvier 27-29, 1988.

[Tic 85] TICHY W., *RCS- A System for Version Control*, Software Practice and Experience, Vol. 15(7), Juillet 1985.

[Tho 89] THOMAS I., *PCTE Interfaces: Supporting Tools in Software Engineering Environments*, IEEE Software, Novembre 1989.

[Tom 89] TOMPA F.W.N. , *A Data Model for Flexible Hypetrtext Database Systems*, ACM Trans. on Information Systems, Vol. 7, No. 1, Janvier 1989, pp 85-100.

[Tri 86] TRIGG R. & WEISER M., *TEXTNET: A Network-Based Approach to Text Handling*, ACM, Trans. Off. Inf. Sys., Vol. 4, N°. 1, Jan. 1986, pp 1-23.

[Tri 82] TRING T.C., *ADD: An Automated Tool for Program Design and Documentation*, In Proceedings of the NBS FIPS Software Documentation Workshop, Gaithersburg, MD, Mars 3, 1982, Ed. Institut for Computer Sciences and Technology, National Bureau of Standards, Washington, DC20234.

[TSI 86] Technique et Science Informatiques, un numéro spécial: Edition Electrronique et Manipulation de documents, Vol. 5, N°4, 1986.

[Val 87] VALDURIEZ P., *Objets Complexes dans les Systèmes de Bases de Données Relationnels*, TSI, Vol. 6, No. 5, 1987.

[Vel 84] VELEZ F., *Un Modele et un Langage pour les Bases de Données Généralisées*, Thèse de Doctotrat de l'INPG, Septembre 1984.

[Was 81] WASSERMAN A.I., *Tutorial: Software Development Environments*, In Proc. COMPSAC, 1981, IEEE Comput. Soc., EHO 187-5.

[Was 80] WASSERMAN A.I., *Towards Integrated Software Development Environment*, International Seminar on Software Engineering Applications, Vol. 1980, pp. 1-21.

[Wei 84] WIENER R. & SINCOVEC R., *Software Engineering with Modula-2 and ADA*, John Wiley & Sons, 1984.

[Wil 88] WILLIAMS L.G., *Software Process Modelling: A Behavioral Approach*, Proc. of the Int. Conf. on Soft. Eng., Rafles City, Singapore, Avril 11-15, 1988.

[Wil 86] WILE D.S. & ALLARD D.G., *Worlds : an Organisation Structure for Object-Bases*, Proc. of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development, Palo Alto, California, Decembre 9-11, 1986.

[Woe 86] WOELK D., KIM W. & LUTHER W., *An Objact Oriented Approach to Multimedia Databases*, ACM SIGMOD Intl. Conf. on Management of Data, 1986, pp. 311-325.

[Xer 85] *Interlisp-D Reference Manual*, Volumes I-II-III, Xerox PARC 1985.

[You 88] YOUNG M., TAYLOR R. & TROUP D., *Software Environment Architectures and User Interface Facilities*, IEEE Transaction on Software Engineering, Vol 14, N°. 6, Juin 1988.

[Zav 82] ZAVELER S. A., *A Proposed Documentation Standard Based on System Decomposition and Information Based Approach*, In Proc. of the NBS FIPS Software Documentation Workshop, Mars 3, 1982, at NBS, Gaithersburg, MD, Ed. Institut for Computer Sciences and Technology, National Bureau of Standards, Washington, DC20234.

[Zdo 86] ZDONIK S.B., *Version Management in an Object-Oriented Database*, Proc. of int. Workshop on Advanced Prog. Environment, Trondheim Norway, 16-18 Juin, 1986. Springler Verlag (ed.)LNCS 244, Fevrier 1987.

ANNEXE

EXTRAIT DE L'IMPLANTATION MONTRANT LES DEFINITIONS DES CLASSES

```
(DEFINE-FILE-INFO  READTABLE "XCL"  PACKAGE "INTERLISP")
  (FILECREATED "31-Mar-92 16:02:10"   |{DSK}<home>rechdoc>jarwa>LISPFILES>PROJETELEN.;19|   231489

      |changes| |to:|  (METHODS |Doc.PutCompToCard| |Doc.GetContent| |Doc.GetAtt1| |Doc.Display|
                           |Doc.Delete| |Doc.AppendRef1| |Doc.AppendProp1| |Doc.AppendComp1|
                           |DocType.NewO| |DocType.NewGO|)

      |previous| |date:| "31-Mar-92 15:03:07" |{DSK}<home>rechdoc>jarwa>LISPFILES>PROJETELEN.;18|)


; Copyright (c) 1990, 1991, 1992 by SAHAR JARWA LGI-IMAG GRENOBLE FRANCE.  All rights reserved.

(PRETTYCOMPRINT PROJETELENCOMS)

(RPAQQ   PROJETELENCOMS
       ((CLASSES |AtomicType| |BasicType| |CompRelation| |CompositeType| |Composition| |Content|
              |Descriptor| |Doc| |DocType| |ElenBasicType| |GComposantAV| |GDoc| |GObject| |GType|
              |Integer| |LispType| |MethodMeta| |MyMeta| |MyMethod| |MyObject| |Project|
              |ProjectMeta| |RefRelation| |Reference| |Relation| |RelationType| |Sequence|
              |SimpleType| |SourceCode| |String| |Super| |Text| |Transaction| |Type| |VDoc|
              |VObject| |VType|)
        (METHODS |AtomicType.HasValue| |BasicType.HasValue| |CompRelation.NewOne|
              |Composition.Delete| |Composition.Extract| |Doc.AppendComp1| |Doc.AppendProp1|
              |Doc.AppendRef1| |Doc.Delete| |Doc.Display| |Doc.EditContent| |Doc.GetAtt|
              |Doc.GetAtt1| |Doc.GetContent| |Doc.PutComp1| |Doc.PutCompToCard| |Doc.PutContent|
              |Doc.PutContent1| |Doc.PutDesc1| |Doc.PutProp1| |Doc.PutRef1| |Doc.PutRefToCard|
              |DocType.AddComp1| |DocType.AddLoopsMethToCards| |DocType.AddMethod|
              |DocType.AddMethodToCards| |DocType.AddRef1| |DocType.EdMeth| |DocType.EliminateDoc|
              |DocType.NewDoc| |DocType.NewGO| |DocType.NewO| |ElenBasicType.Create|
              |ElenBasicType.NewO| |GComposantAV.GetWrappedValueOnly| |GDoc.NewV|
              |GObject.AppendCompAll| |GObject.AppendRefAll| |GObject.AttachCompAll|
              |GObject.AttachRefAll| |GObject.Delete| |GObject.GetAtt1| |GObject.GetDefaultVersion|
              |GObject.GetProp| |GObject.GetThisComp| |GObject.GetThisProp| |GObject.GetThisRel|
              |GObject.LastVersion| |GObject.NewV| |GObject.PutCompAll| |GObject.PutPropAll|
              |GObject.PutRefAll| |GObject.PutSeqCompAll| |GObject.PutSeqRefAll| |GObject.Send|
              |GObject.SetDefaultVersion| |GType.New| |MethodMeta.New| |MyMeta.DefDoc|
              |MyMeta.DefProject| |MyMeta.DefType| |MyMeta.Define| |MyMeta.EdDef|
              |MyMethod.Activate| |MyObject.AppendAtt| |MyObject.AppendComp| |MyObject.AppendComp1|
              |MyObject.AppendProp| |MyObject.AppendProp1| |MyObject.AppendRef|
              |MyObject.AppendRef1| |MyObject.AttachAtt| |MyObject.AttachComp| |MyObject.AttachProp|
              |MyObject.AttachRef| |MyObject.CompositeAtt| |MyObject.CreationDate| |MyObject.Delete|
              |MyObject.FindModAtt| |MyObject.FindObjects| |MyObject.GetAtt| |MyObject.GetAtt1|
              |MyObject.GoodPropValue| |MyObject.PutAtt| |MyObject.PutComp| |MyObject.PutComp1|
              |MyObject.PutProp| |MyObject.PutProp1| |MyObject.PutRef| |MyObject.PutRef1|
              |MyObject.PutSeqComp| |MyObject.PutSeqComp1| |MyObject.PutSeqProp|
              |MyObject.PutSeqProp1| |MyObject.PutSeqRef| |MyObject.PutSeqRef1|
              |MyObject.ReferenceAtt| |MyObject.Send| |Project.AllDocTypes| |Project.AllTypes|
              |Project.Close| |Project.Init| |Project.LoadType| |Project.ModifyNoteCardTypes|
              |Project.Open| |Project.Open1| |ProjectMeta.Define| |Reference.Extract|
              |RelationType.New| |Sequence.BonObjectsWithType| |Sequence.BonValuesWithType|
              |Sequence.BonWithType| |SourceCode.Edit| |Text.Edit| |Transaction.Abort|
              |Transaction.Handle| |Transaction.Notify| |Transaction.Validate| |Type.AddAtt|
              |Type.AddAtt1| |Type.AddAtts| |Type.AddComp| |Type.AddComp1| |Type.AddMethod|
              |Type.AddProp| |Type.AddProp1| |Type.AddRef| |Type.AddRef1| |Type.BadObjSpec|
              |Type.CompositeAtt| |Type.CreateClass| |Type.EdDef| |Type.EdMeth| |Type.Eliminate|
              |Type.EliminateMeth| |Type.HasAnObject| |Type.HasGObject| |Type.HasVObject|
              |Type.NewGO| |Type.NewO| |Type.NewObject| |VDoc.DeleteOnly| |VDoc.Derive|
              |VDoc.PutRefToCard| |VObject.AppendComp1| |VObject.AppendProp1| |VObject.AppendRef1|
              |VObject.CopyAtts| |VObject.CreateValue| |VObject.Delete| |VObject.DeleteOnly|
              |VObject.Derive| |VObject.Derive1| |VObject.GetAtt1| |VObject.LastDerived|
```

```
            |VObject.PutCompl| |VObject.PutProp1| |VObject.PutRef1| |VType.New|)
        (INSTANCES |ElenPromptW| |Type|)
        (FNS |ActivateMeths| |BadListAtts| |BeginTrans| |ChangDoc| |ChangMeth| |ChangMethDoc|
             |Change| |ChangeProject| |DocBadListAtts| |GoodDocTypeDef| |GoodMethDef|
             |GoodProjectDef| |GoodTypeDef| |SendMessage| |SortFonc|)
        (VARIABLES ELENINSTFILE ELENOPENEDPROJECT ELENPROMPTW ELENSTREAM)
        (I.S.OPRS |Comps| |Content| |DefineDocType| |DefineProject| |DefineType| |Descriptor| D\o
             |ExportedTypes| |GeneralUsers| I\f |ImportedTypes| I\sA |Method| |Notify| |OfMode| O\n
             |PropagatedValues| |Props| |Refs| |SuperUsers| |Versioned| |WithArgs| |WithPriority|)
        (DECLARE\: DONTEVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVARS (ADDVARS (NLAMA)
                                                                           (NLAML)
                                                                           (LAMA)))))

(DEFCLASSES |AtomicType| |BasicType| |CompRelation| |CompositeType| |Composition| |Content|
       |Descriptor| |Doc| |DocType| |ElenBasicType| |GComposantAV| |GDoc| |GObject| |GType| |Integer|
       |LispType| |MethodMeta| |MyMeta| |MyMethod| |MyObject| |Project| |ProjectMeta| |RefRelation|
       |Reference| |Relation| |RelationType| |Sequence| |SimpleType| |SourceCode| |String| |Super|
       |Text| |Transaction| |Type| |VDoc| |VObject| |VType|)
(DEFCLASS   |AtomicType|
   (|MetaClass| |MyMeta| |Edited:|                          (* \; "Edited 14-Sep-90 11:51 by Sahar")
         )
   (|Supers| |Type|))

(DEFCLASS   |BasicType|
   (|MetaClass| |MyMeta| |Edited:|                          (* \; "Edited 17-Jun-91 14:13 by Sahar")
         )
   (|Supers| |AtomicType|))

(DEFCLASS   |CompRelation|
   (|MetaClass| |MetaClass| |Edited:|                       (* \; "Edited  9-Aug-90 17:46 by Sahar")
         )
   (|Supers| |RelationType|))

(DEFCLASS   |CompositeType|
   (|MetaClass| |MyMeta| |Edited:|                          (* \; "Edited 14-Sep-90 11:46 by Sahar")
         )
   (|Supers| |Type|))

(DEFCLASS   |Composition|
   (|MetaClass| |CompRelation| |Edited:|                    (* \; "Edited  9-Aug-90 18:12 by Sahar")
         )
   (|Supers| |Relation|))

(DEFCLASS   |Content|
   (|MetaClass| |CompRelation| |Edited:|                    (* \; "Edited 21-Jun-91 15:25 by Sahar")
         )
   (|Supers| |Composition|))

(DEFCLASS   |Descriptor|
   (|MetaClass| |CompRelation| |Edited:|                    (* \; "Edited 21-Jun-91 15:26 by Sahar")
         )
   (|Supers| |Composition|))

(DEFCLASS   |Doc|
   (|MetaClass| |Type| |Edited:|                            (* \; "Edited  1-Jul-91 15:40 by Sahar")
         )
   (|Supers| |MyObject|)
   (|InstanceVariables| (|Card| |ThisCard| |doc|            (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |DocType|
```

```
   (|MetaClass| |MyMeta| |Edited:|                          (* \; "Edited 30-Mar-92 16:08 by jarwa")
         )
   (|Supers| |CompositeType|))

(DEFCLASS   |ElenBasicType|
   (|MetaClass| |MyMeta| |Edited:|                          (* \; "Edited 14-Sep-90 11:56 by Sahar")
         )
   (|Supers| |BasicType|))

(DEFCLASS   |GComposantAV|
   (|MetaClass| |Class| |Edited:|                           (* \; "Edited 16-Oct-90 17:19 by Sahar")
         )
   (|Supers| |LocalStateActiveValue|)
   (|InstanceVariables| (|localStat| NIL |doc|              (* IV |added| |by| JARWA)  )))

(DEFCLASS   |GDoc|
   (|MetaClass| |Type| |Edited:|                            (* \; "Edited  1-Jul-91 15:43 by Sahar")
         )
   (|Supers| |Doc| |GObject|)
   (|InstanceVariables| (|Card| |ThisCard| |doc|           (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |GObject|
   (|MetaClass| |GType| |Edited:|                           (* \; "Edited 13-Aug-90 10:14 by Sahar")
         )
   (|Supers| |MyObject|)
   (|InstanceVariables| (|DefaultVersion| NIL |doc|        (* IV |added| |by| |Sahar|)  )
         (|LastVersion| NIL |doc|                          (* IV |added| |by| |Sahar|)  )
         (|Alternatives| NIL |doc|                         (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |GType|
   (|MetaClass| |MyMeta| |Edited:|                         (* \; "Edited 26-Dec-91 14:51 by Sahar")
         )
   (|Supers| |Class|))

(DEFCLASS   |Integer|
   (|MetaClass| |LispType| |Edited:|                        (* \; "Edited  2-Oct-90 17:27 by Sahar")
         )
   (|Supers| |Object|))

(DEFCLASS   |LispType|
   (|MetaClass| |MyMeta| |Edited:|                         (* \; "Edited 14-Sep-90 11:55 by Sahar")
         )
   (|Supers| |BasicType|))

(DEFCLASS   |MethodMeta|
   (|MetaClass| |MetaClass| |Edited:|                      (* \; "Edited 13-Dec-91 16:53 by Sahar")
         )
   (|Supers| |Class|))

(DEFCLASS   |MyMeta|
   (|MetaClass| |MetaClass| |Edited:|                      (* \; "Edited  9-Aug-90 11:22 by Sahar")
         )
   (|Supers| |MetaClass|))

(DEFCLASS   |MyMethod|
   (|MetaClass| |MethodMeta| |Edited:|                     (* \; "Edited 13-Dec-91 16:54 by Sahar")
         )
   (|Supers| |Object|)
   (|InstanceVariables| (|DefExp| NIL |doc|                (* IV |Added| |by| |Sahar|)  )
         (|Sel| NIL |doc|                                  (* IV |added| |by| |Sahar|)  )
```

```
          (|Mode| '|Immediat| |doc|                              (* IV |Added| B\y |Sahr|)  )
          (|Priority| 1 |doc|                                    (* IV |Added| |by| |Sahar|)  )
          (  |Notify|   NIL |doc| (*IV |Adde| |by| |Sahar|))
          (|Event| NIL |doc|                                     (* IV |Added| \b \y |Sahar|)  )
          (|Cond| NIL |doc|                                      (* IV |Added| |by| |Sahar|)  )))

(DEFCLASS   |MyObject|
   (|MetaClass| |VType| |Edited:|                                (* \; "Edited 23-Mar-92 15:37 by jarwa")
          )
   (|Supers| |Object|)
   (|ClassVariables| (|ObjectType| NIL |doc|                     (* CV |added| |by| |Sahar|)  ))
   (|InstanceVariables| (|CreationDate| NIL |:initForm| (DATE)
                              |doc|                             (* IV |added| |by| |Sahar|)
                              )
          (|ReferedBy| NIL |doc|                                 (* IV |added| |by| |Sahar|)  )
          (|ChildOf| NIL |doc|                                   (* IV |added| |by| |sahar|)  )
          (|Children| NIL |doc|                                  (* IV |added| |by| |Sahar|)  )
          (|References| NIL |doc|                                (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |Project|
   (|MetaClass| |ProjectMeta| |Edited:|                         (* \; "Edited 29-Mar-92 10:48 by jarwa")
          )
   (|Supers| |MyObject|)
   (|InstanceVariables| (  |SuperUsers|   NIL |doc|             (* IV |added| |by| |Sahar|)  )
          (  |GeneralUsers|   NIL |doc|                         (* IV |added| |by| |Sahar|)  )
          (  |ImportedTypes|   NIL |doc|                        (* IV |added| |by| |Sahar|)  )
          (|DefinedTypes| NIL |doc|                              (* IV |added| |by| |Sahar|)  )
          (|DefinitionExpr| NIL |doc|                            (* IV |added| |by| |Sahar|)  )
          (|OpenedPreviously| NIL |doc|                          (* IV |added| |by| |Sahar|)  )
          (|ModifiedSchem| NIL |doc|                             (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |ProjectMeta|
   (|MetaClass| |MetaClass| |Edited:|                           (* \; "Edited 29-Nov-91 19:08 by Sahar")
          )
   (|Supers| |Class|))

(DEFCLASS   |RefRelation|
   (|MetaClass| |MetaClass| |Edited:|                           (* \; "Edited  9-Aug-90 17:48 by Sahar")
          )
   (|Supers| |RelationType|))

(DEFCLASS   |Reference|
   (|MetaClass| |RefRelation| |Edited:|                         (* \; "Edited  9-Aug-90 18:13 by Sahar")
          )
   (|Supers| |Relation|))

(DEFCLASS   |Relation|
   (|MetaClass| |RelationType| |Edited:|                        (* \; "Edited 11-Jul-91 14:54 by Sahar")
          )
   (|Supers| |Object|)
   (|ClassVariables| (|SType| NIL |doc|                         (* CV |added| |by| |Sahar|)  )
          (|DType| NIL |doc|                                     (* CV |added| |by| |sahar|)  )
          (|Cons| NIL |doc|                                      (* CV |added| |by| |Sahar|)  )
          (|Cardinality| NIL |doc|                               (* CV |added| |by| |Sahar|)  )
          (|Invariant| NIL |doc|                                 (* CV |added| |by| |Sahar|)  ))
   (|InstanceVariables| (|SObject| NIL |doc|                     (* IV |added| |by| |Sahar|)  )
          (|DObject| NIL |doc|                                   (* IV |added| |by| |Sahar|)  )
          (|DCard| NIL |doc|                                     (* IV |added| |by| |Sahar|)  )
          (|Link| NIL |doc|                                      (* IV |added| |by| |Sahar|)  )))
```

```
(DEFCLASS   |RelationType|
   (|MetaClass| |MyMeta| |Edited:|                              (* \; "Edited 26-Dec-91 14:54 by Sahar")
          )
   (|Supers| |Class|))

(DEFCLASS   |Sequence|
   (|MetaClass| |Class| |Edited:|                               (* \; "Edited 23-Oct-90 18:38 by Sahar")
          )
   (|Supers| |Object|))

(DEFCLASS   |SimpleType|
   (|MetaClass| |MyMeta| |Edited:|                              (* \; "Edited 14-Sep-90 11:45 by Sahar")
          )
   (|Supers| |Type|))

(DEFCLASS   |SourceCode|
   (|MetaClass| |ElenBasicType| |Edited:|                       (* \; "Edited 22-Mar-92 17:49 by jarwa")
          )
   (|Supers| |Doc|))

(DEFCLASS   |String|
   (|MetaClass| |LispType| |Edited:|                            (* \; "Edited  2-Oct-90 17:27 by Sahar")
          )
   (|Supers| |Object|))

(DEFCLASS   |Super|
   (|MetaClass| |Type| |Edited:|                                (* \; "Edited 19-Jul-91 17:00 by Sahar")
          )
   (|Supers| |Object|)
   (|ClassVariables| (|Methods| NIL |doc|                       (* CV |added| |by| |sahar|)  )
          (  |PropagatedValues|   NIL |doc|                    (* CV |added| |by| |sahar|)  )
          (  |Versioned|   NIL |doc|                           (* CV |added| |by| |Sahar|)  )
          (|DefinitionExpr| (  |DefineType|   \t2   |Versioned|   NIL   |PropagatedValues|   NIL
                               I\sA   NIL   |Props|   NIL   |Comps|   ((\c NIL 0 \t1))
                               |Refs|   ((\r NIL 0 \t1)))
                |doc|                                          (* CV |addedby| |Sahar|)
                )))

(DEFCLASS   |Text|
   (|MetaClass| |ElenBasicType| |Edited:|                       (* \; "Edited  8-Jul-91 11:15 by Sahar")
          )
   (|Supers| |Doc|))

(DEFCLASS   |Transaction|
   (|MetaClass| |Class| |Edited:|                               (* \; "Edited 31-Jul-91 15:58 by Sahar")
          )
   (|Supers| |Object|)
   (|InstanceVariables| (|Process| NIL |doc|                    (* IV |added| |by| |Sahar|)  )
          (|Event| NIL |doc|                                    (* IV |added| |by| |Sahar|)  )
          (|Sel| NIL |doc|                                      (* IV |added| |by| |Sahar|)  )
          (|Object| NIL |doc|                                   (* IV |added| |by| |Sahar|)  )
          (|Args| NIL |doc|                                     (* IV |added| |by| |Sahar|)  )
          (|ImmediateMeths| NIL |doc|                           (* IV |added| |by| |Sahar|)  )
          (|DifferedMeths| NIL |doc|                            (* IV |added| |by| |Sahar|)  )
          (|RefuseMeths| NIL |doc|                              (* IV |added| |by| |Sahar|)  )
          (|Children| NIL |doc|                                 (* IV |added| |by| |Sahar|)  )
          (|Parent| NIL |doc|                                   (* IV |added| |by| |Sahar|)  )
          (|State| NIL |doc|                                    (* IV |added| |by| |Sahar|)  )
          (|Result| NIL |doc|                                   (* IV |added| |by| |Sahar|)  )))
```

```
(DEFCLASS   |Type|
   (|MetaClass| |MyMeta| |Edited:|                          (* \; "Edited 30-Mar-92 16:07 by jarwa")
         )
   (|Supers| |Class|)
   (|ClassVariables| (|DefinitionExpr| NIL |doc|             (* CV |added| |by| |sahar|)  ))
   (|InstanceVariables| (|Methods| NIL |doc|                 (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |VDoc|
   (|MetaClass| |Type| |Edited:|                             (* \; "Edited 21-Jun-91 09:54 by Sahar")
         )
   (|Supers| |VObject| |Doc|))

(DEFCLASS   |VObject|
   (|MetaClass| |VType| |Edited:|                            (* \; "Edited 13-Aug-90 10:15 by Sahar")
         )
   (|Supers| |MyObject|)
   (|InstanceVariables| (|DerivedVersions| NIL |doc|         (* IV |added| |by| |Sahar|)  )
         (|OriginVersion| NIL |doc|                          (* IV |added| |by| |Sahar|)  )
         (|GenericObject| NIL |doc|                          (* IV |added| |by| |Sahar|)  )
         (|State| NIL |doc|                                  (* IV |added| |by| |Sahar|)  )))

(DEFCLASS   |VType|
   (|MetaClass| |MyMeta| |Edited:|                           (* \; "Edited 26-Dec-91 14:53 by Sahar")
         )
```