ETIC : un SGBD pour la CAO dans un environnement partagé

Marie-Christine Fauvet

THESE

présentée par

Marie Christine FAUVET

pour obtenir le titre de **DOCTEUR**

de L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(arrêté ministériel du 5 juillet 1984)

spécialité : **INFORMATIQUE**

ETIC :

Un SGBD pour la CAO dans un environnement partagé

thèse soutenue le : *12 septembre 1988*

COMPOSITION DU JURY :

Président :	*M. Yves CHIARAMELLA*
Rapporteurs :	*M Claude DELOBEL*
	M. Michel LEONARD
Examinateurs :	*M. François BANCILHON*
	M. Michel ADIBA

THESE PREPAREE AU SEIN DU LABORATOIRE DE GENIE INFORMATIQUE
A L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1 -

Je tiens à remercier,

Monsieur Y. Chiaramella, Professeur à l'Université Joseph Fourier et Directeur du Laboratoire de Génie Informatique pour avoir accepté de présider ce jury,

Monsieur C. Delobel, Professeur à l'Université d'Orsay, qui a bien voulu rapporter efficacement ce travail, et qui par ses remarques a contribué à la qualité du document,

Monsieur M. Léonard, Professeur à l'Université de Genève, qui grâce à sa patience et sa gentillesse, à ses questions et ses remarques m'a fait analyser les problèmes sous un autre jour, m'a aidée à faire progresser le document,

Monsieur M. Adiba, Professeur à l'Université Joseph Fourier, chef du Groupe Bases de Données, qui m'a accueillie dans son équipe et qui a dirigé mes recherches. Il m'a permis, par ses encouragements, de surmonter les angoisses du thésard moyen, et par ses critiques constructives a fait avancer mon travail,

Monsieur F. Bancilhon, Directeur du G.I.P. Altaïr, qui me fait l'honneur de s'intéresser à cette thèse et de participer au jury,

L'ENSIMAG, école de l'INP de Grenoble, qui en me recrutant dans son équipe d'enseignants, m'a permis d'exercer une activité intéressante et complémentaire à celle de la recherche,

Je ne pourrais pas terminer cette liste d'hommages sans saluer Catherine, Christine et Marie-France, Pierre-Claude, Philippe et Hervé qui ont su supporter avec philosophie mes égarements coléreux, m'écouter et me soutenir dans les moments difficiles,

Je n'ai plus beaucoup de place sur cette page pour dire à l'ensemble des copains du Laboratoire combien leur bonne humeur et leur ouverture d'esprit sont un soutien moral indispensable,

Je remercie enfin le service reprographie pour le soin qui a été apporté au tirage de cette thèse.

TABLE DES MATIERES

CHAPITRE 1

INTRODUCTION ET ETAT DE L'ART

table des matières

1. INTRODUCTION

L'idée de base de la Conception Assistée par Ordinateur (CAO), est l'association de la puissance de calcul et de stockage de l'ordinateur avec l'intuition, la réflexion et l'expertise de l'homme [DAV81]. Cette intégration a suivi plusieurs étapes au cours de l'histoire de la CAO. Longtemps dédiée à la conception des ordinateurs (fais-toi toi-même !!!), la CAO tend à être utilisée aujourd'hui dans des domaines de plus en plus divers et variés. Des besoins communs apparaissent dans les domaines de la conception de VLSI, du Génie Logiciel, de la production de documents, de l'architecture, etc .. Les premiers systèmes CAO consacrés à la conception de circuits, offrent un ensemble d'outils développés ponctuellement suivant les besoins et les progrès techniques des ordinateurs. Ces outils, (placement automatique, simulation, etc ..) utilisent des données hétérogènes et stockées dans différents fichiers. Malheureusement ce type d'organisation nécessite un gros travail de préparation des données, et d'interprétation des résultats.

Ces systèmes ont ensuite évolué en réalisant l'enchaînement des différentes étapes de la conception. L'avantage est que l'utilisateur de ce système est déchargé de la phase de préparation des données. L'inconvénient est la rigidité introduite par ce type de fonctionnement. On s'est en effet, rapidement aperçu qu'il est nécessaire de ne pas imposer de méthodologie aux concepteurs. Suivant le cas, il peut choisir un outil plutôt qu'un autre, une méthode plutôt qu'une autre. Les concepteurs doivent disposer de toute l'assistance nécessaire, avec la plus grande liberté possible.

Les systèmes CAO doivent offrir aux concepteurs un environnement dans lequel :

- des outils anciens et nouveaux, écrits dans différents langages de programmation (Pascal, Fortran, etc ..), peuvent être intégrés au moindre coût.

- la sécurité de fonctionnement (résistance aux pannes et reprise après panne) est assurée par un système de gestion de transactions prenant en compte le caractère particulier de l'activité de conception.

- aucune méthode de travail n'est imposée. Des outils souples et puissants, adaptables à la méthodologie de chacun sont disponibles. Ces

outils concernent la gestion des versions pour permettre l'archivage, le traçage de la conception, l'essai et l'erreur, etc ... Le concepteur peut à sa guise, essayer un outil ou une méthode, mémoriser les résultats tenter une autre méthode afin de comparer les performances. Puis, s'il s'aperçoit que le chemin choisi mène à une impasse, il peut alors reprendre sa conception dans un état antérieur. Le processus de conception est un processus itératif, basé sur le principe **essai-erreur**. B. David définit dans [DAV81] le processus de conception comme une succession de transformations de l'objet à un certain niveau de définition vers un autre plus précis et plus concret. Le schéma d'un objet n'est figé qu'en fin de la conception. Il est donc nécessaire d'assurer une grande flexibilité des structures et des valeurs des objets. Les modifications doivent se faire sans recompilation de schémas et sans rechargement de la base. Ce type de processus implique de plus de supporter et de gérer des **incohérences passagères** des objets en cours de conception.

- le partage des connaissances et des objets, ainsi que la coopération dans l'effort de conception est permis et géré. L'activité de conception est le plus souvent un travail d'équipe, au cours duquel des objets sont échangés, et d'autres conçus en commun.

- la répartition géographique des moyens matériels est prise en compte. Un système CAO est le plus souvent configuré en un ensemble de stations de travail évoluées, connectées entre elles par un réseau local. Un serveur de fichiers est souvent associé au réseau.

- l'interface homme-machine est conviviale et offre le plus souvent des éditeurs graphiques performants pour la description des objets. Une assistance dans le processus de conception est nécessaire. Elle peut prendre plusieurs formes : (1) assistance dans l'utilisation du système lui-même (concepts, fonctionnalités, etc ..), et (2) assistance pour la compréhension de ce qui est déjà conçu.

Un système CAO, en plus des fonctionnalités décrites ci-dessus, doit assurer la gestion d'un grande volume de données, complexes, fortement liées entre elles, et très diverses. Les Systèmes de Gestion de Bases de Données (SGBD) assurant entre autre, la modélisation et la manipulation des données, l'indépendance des données et des programmes, et la sécurité de fonctionnement, sont malgré tout insuffisants. R. Katz dans

[KAT85], présente les principaux points qui font qu'un SGBD commercialisé ne convient pas pour la gestion des données de CAO :

- les modèles de données, particulièrement le modèle relationnel, sont trop pauvres pour permettre l'expression de toute la sémantique des données.

- l'activité de conception consiste à faire évoluer les objets. Cette évolution s'applique aussi bien sur les valeurs des objets, que sur leur type. Cette dynamicité, caractéristique intrinsèque de la CAO, est mal gérée par les SGBD traditionnels. La modification des données est le plus souvent assurée par un utilisateur privilégié : l'administrateur de la base de données, et à des moments opportuns. Tandis qu'en CAO, les opérations de modification sont faites par n'importe quel concepteur, à n'importe quel moment.

- les SGBD disposent de mécanismes sophistiqués et efficaces d'accès aux données atomiques. En CAO, un objet est rarement atomique, il est le plus souvent composé de sous objets, par suite, l'accès à un objet ne délivre pas une, mais plusieurs données.

- la gestion des transactions : une transaction dans un SGBD relationnel, est une séquence d'opérations de base sur la base de données. Cette séquence d'actions, vise à transformer un état cohérent de la base de données en un autre état cohérent. Toutes les mises à jour sont faites ensemble, ou aucune n'est prise en compte. Ces caractéristiques ne sont pas valides en CAO. Nous développons en détail les problèmes soulevés dans la section 2.

- dans les SGBD les contraintes d'intégrité sont le plus souvent très pauvres, et une opération provoquant la violation d'une contrainte d'intégrité est rejetée. Or en CAO, cette opération doit être admise, car l'opération mise en jeu est complexe et coûteuse, et l'incohérence décelée doit être gérée.

- les SGBD savent gérer les accès concurrents aux données. Des méthodes efficaces de verrouillage sont utilisées, mais elles ne tirent pas parti du rôle de chacun des utilisateurs de la base de données. En effet, expliciter le partage de la conception, et grâce à cela, limiter le domaine d'action de chaque concepteur, permet de simplifier considérablement le problème de la gestion des accès concurrents aux données.

- des études ont été faites dans le but d'introduire dans les SGBD les notions de temps et d'historiques [ABC87]. Ces notions sont quelques

fois nécessaires en CAO, mais restent insuffisantes. En effet, de par sa nature essai-erreur, l'activité de conception nécessite de gérer de multiples versions d'un objet donné. Les mécanismes requis pour la gestion et la manipulation de ces versions nécessitent de les caractériser par leurs propriétés, leurs valeurs, etc ... Les problèmes inhérents à la gestion des versions sont développés dans la section 3 de ce chapitre.

Toutes les inadéquations présentées ci-dessus, expliquent que certaines études faites dans le domaine des Bases de Données pour la CAO, ont visé à décrire de nouveaux modèles de données et de nouvelles fonctionnalités, c'est à dire de nouveaux systèmes de gestion de bases de données, dédiés à la CAO [AFL86b], [DLO85], [KAT85], [RIE85]. D'autres études dans le même domaine, ont consisté à enrichir le modèle relationnel [LOR83], [SRG83].

2. LES TRANSACTIONS

2.1. La Problématique

Classiquement, un système de gestion de transactions vise à assurer la sécurité et la cohérence des données de la base. Afin de lever toute ambiguïté sur l'utilisation de certains termes, nous donnons au préalable quelques définitions.

Cohérence de la base de données : une base de données n'est pas une simple collection de données, rangées et structurées selon un certain modèle. Elle représente une image du monde réel, que l'on désire la plus fidèle et la plus juste possible. Un moyen de s'assurer d'une relative fidélité est d'obliger les données à vérifier un ensemble de propriétés : **les contraintes d'intégrité**. Toute donnée violant une contrainte n'est pas acceptée dans la base de données.

Transaction : dans [DEA82], une transaction est définie comme une séquence d'opérations transformant un état cohérent de la base de données, en un autre état cohérent.

Atomicité d'une transaction : une transaction est un tout. Soit toutes les opérations qui la constituent sont exécutées, soit elles sont toutes ignorées. On distingue classiquement trois cas [DEA82] : ***(1) validation :***

toutes les actions prévues sont exécutées, elles délivrent la base de données dans un état cohérent. ***(2) annulation*** *:* une opération de la transaction entraîne une incohérence des données : au moins une contrainte d'intégrité est violée. La transaction est annulée, toutes les actions faites depuis le début de la transaction sont défaites. ***(3) panne :*** un incident matériel ou logiciel provoque l'interruption de la transaction. Lors du redémarrage de la base de données, toutes les actions effectuées entre le début de la transaction et la panne devront être ignorées.

Accès concurrents : généralement plusieurs transactions s'exécutent en parallèle; si aucun contrôle de tels accès engendreraient un état incohérent. De nombreuses études ont été faites pour résoudre le problèmes des accès concurrents [DEA82], [LYN83] [DAT81], [BOK86], [GRA81], [ABV84], [DAT81]. Ces différentes approches sont basées sur le fait qu'une transaction pour une application dite classique (par rapport aux applications bureautiques, CAO, Génie Logiciel, ..), est atomique, de courte durée et concerne peu de données.

En se terminant, une transaction valide les mises à jour qu'elle a effectuées, à condition que l'état de la base de données qui en découle soit cohérent, les mises à jour sont alors physiquement enregistrées sur le disque. Il est possible d'interrompre volontairement une transaction et d'ignorer les opérations effectuées depuis le début de la transaction. Ainsi, dans le domaine des bases de données relationnelles, une transaction constitue **une unité de cohérence et de restauration.**

Nous allons montrer dans la suite de ce chapitre, que les mécanismes classiques offerts en base de données se révèlent insuffisants et le plus souvent inadaptés dans le cadre des applications CAO : la transaction vue comme unité de restauration et de cohérence est à remettre en cause.

Katz dans [KAW83], [KAT85], décrit les transactions CAO, par opposition aux transactions traditionnelles souvent assimilées à la notion de processus, comme dans le cas des réservations de vols, ou des opérations bancaires. Nous pouvons dégager plusieurs propriétés des transactions CAO :

Les Transactions CAO sont de longue durée :

Elles peuvent ainsi durer plusieurs jours et même plusieurs semaines. Dans l'absolu, une transaction démarre à l'initialisation de la conception, et se termine lorsque l'objet à concevoir est terminé. La solution pour résoudre les accès concurrents qui consiste à verrouiller les objets accédés n'est pas satisfaisante. En effet, forcer l'accès exclusif aux données partagées oblige les transactions (c'est à dire des utilisateurs) à attendre longtemps (trop !!) la libération des verrous qui est faite à la fin de la transaction propriétaire de ces verrous.

Les Transactions CAO manipulent un grand nombre de données :

L'accès à un objet se traduit par l'accès à un grand volume de données liées entre elles. Le verrouillage sur un grand nombre de données n'est pas non plus admissible, pour les mêmes raisons que celles données dans le point précédent.

Les mécanismes traditionnels sont insuffisants :

Un travail de conception est coûteux, et nécessite un haut niveau d'interactivité, de ce fait, les mécanismes classiques de prévention des interblocages qui forcent une transaction à attendre dans le cas d'un conflit, ou dans le meilleur des cas, temporisent son exécution ne sont pas satisfaisants.

Les Transactions CAO ne sont pas "tout ou rien" :

Une transaction classique est transparente pour l'utilisateur : soit elle termine et toutes les mises à jour sont effectuées, ou soit elle est annulée, et rien n'est fait. Cette notion d'atomicité est remise en cause car en CAO, chaque opération est coûteuse et le concepteur désire restaurer le plus possible de travail après une panne, par exemple. Le plus souvent l'incohérence des données est un fait toléré et géré. Il n'est pas pensable de rejeter un ensemble d'opérations longues et coûteuses qui produisent un objet incohérent. Cet objet doit être admis en tant que tel, et géré par le système. L'objectif visé est de concevoir un objet cohérent, mais il faut être conscient que celui-ci peut passer par des états incohérents au cours de sa conception.

Les principes énoncés dans le cadre des bases de données dites classiques se révèlent donc inapplicables. Les études effectuées pour offrir des mécanismes satisfaisants pour les applications CAO tirent profit, le plus souvent, de la nature particulière de l'activité de conception et de la structure des objets manipulés.

2.2. Les Modèles de Transactions

Deux tendances se dégagent dans les modèles proposés : les systèmes basés sur la notion de base publique, et de bases privées, et ceux basés sur la notion de transactions imbriquées. Les études faites dans le cadre des transactions longues visent à structurer les transactions en hiérarchie de transactions sur un ou plusieurs niveaux. Celles-ci se placent généralement dans le domaine de la bureautique. Dans un cadre comme celui des applications CAO ou Génie Logiciel, où l'activité est plus de concevoir et de faire évoluer des objets dans un contexte de coopération, les approches visent à coupler la notion de hiérarchie de bases de données et de hiérarchie de transactions.

2.2.1. Hiérarchie de Bases de Données

Les modèles proposés dans [KAW83] et [LOR83], sont basés sur les notions de système public (base publique) gérant un ensemble de bases privées. Le système public est vu comme un système partagé, multi-utilisateurs, et les systèmes privés sont des systèmes mono-utilisateur, dans lesquels ne se posent que les problèmes de restauration [LOR83].

Ce type de solution tire parti de la répartition de la connaissance au sein d'un projet de CAO. Certaines de ces connaissances sont communes et partagées, les droits des concepteurs sur celles-ci sont limités, d'autres sont privées, et seul le concepteur propriétaire y a accès.

Pour supprimer le problème d'accès concurrents, lorsqu'un concepteur désire modifier un objet, il l'extrait s'il en a le droit, de la base publique vers sa base privée. Il peut alors effectuer toutes les modifications qu'il désire pendant le temps qui lui est nécessaire. Lorsqu'il estime avoir terminé, il remet l'objet dans la base publique (sous contrôle du système public et sous certains critères). Dans [KAW83], le système de contrôle de la concurrence connaît à tout moment qui a extrait quoi, et quand.

Les systèmes proposés dans [KAW83], [LOR83] sont orientés versions, c'est à dire que lorsque le concepteur rend l'objet modifié, il ne remet qu'une nouvelle version de l'objet dans la base de données publique. Cette version devient la version en cours si elle est cohérente. La version en cours pourra ensuite être partagée entre les autres concepteurs, ils pourront l'extraire vers leur base privée pour la modifier.

Les problèmes dégagés par ce type de système, sont :

- une transaction est vue comme une séquence d'opérations de base. Cette vue des transactions CAO est limitative car il est souvent possible d'exécuter certaines actions en parallèle.

- les objets échangés sont forcément complets et cohérents. Deux concepteurs ne peuvent pas s'échanger d'objets sans passer par le système public, et par conséquent sans que ceux-ci soient cohérents.

Le mécanisme est donc lourd à utiliser et peu adapté au partage des tâches.

- si la conception d'un objet est partagée entre plusieurs concepteurs, chacun développe une version donnée de cet objet. A la fin de la conception, deux versions (au moins) ont été produites. Aucune solution n'apparaît pour assurer la fusion de ces deux versions, afin de produire l'objet final conçu.

2.2.2. Les Modèles de Transactions Imbriquées

L'idée de structurer une transaction comme un arbre de sous transactions, émerge particulièrement dans les domaines des bases de données avancées et de bases de données distribuées. Davies dans [DAV73], introduit l'idée de sphères de contrôle, très proche de celle de transaction imbriquée.

Le principe de transaction imbriquée étend celui de transaction sur un seul niveau, en introduisant une structure en hiérarchie de transactions [HAR87]. Chaque transaction est composée d'un ensemble de sous transactions qui doivent démarrer après, et terminer avant leur transaction parent [MOS82]. Les transactions ainsi imbriquées, offrent une structure de contrôle dynamique, capable de distribuer le travail entre les sous transactions (c'est la même notion que celle des programmes, appelant des sous programmes), d'exécuter les sous-transactions en parallèle dans la

mesure du possible, et de ne faire de retour arrière que sur un nombre minimum de sous transactions.

Les méthodes de verrouillage des données, diffèrent peu de celles appliquées dans le cas de transactions sur un seul niveau. Les différents principes de verrouillage et de restauration des données, appliqués aux transactions imbriquées sont décrits dans [MOS82] :

1) une transaction de peut accéder à une donnée, que si elle a préalablement acquis le verrou adapté (lecture ou écriture).

2) une transaction peut acquérir un verrou, si et seulement si **tous** les détenants du verrou, font partie de ses ancêtres.

3) quand une transaction valide (ou termine), ses ancêtres, s'il en existe, gardent les verrous qui avaient été acquis par la transaction qui vient de valider. Quand une transaction échoue, tous ses verrous sont détruits.

Cette méthode de verrouillage a le mérite d'être simple, et particulièrement bien adaptée à la manipulation d'objets structurés (objets complexes, ou objets CAO, documents, etc ...).

La restauration des données, est basée sur le principe des versions fantômes. C'est à dire qu'une version d'un objet est créée lors de l'acquisition par une transaction d'un verrou en écriture sur cet objet. Une optique optimiste est adoptée : les mises à jour sont faites directement dans la base, et la version fantôme n'est utilisée que lors d'une panne. Elle est alors restaurée. La restauration des données de la base obéit aux trois principes suivants [MOS82] :

1) quand une transaction demande un verrou sur un objet, en écriture, elle crée et possède une version de cet objet. Cette version est une copie de l'objet au moment de la prise du verrou.

2) quand une transaction valide, ses transactions ancêtres, s'il en existe, peuvent hériter de ses versions. Il y a en effet héritage d'une version vers un ancêtre, dans le cas où celui-ci ne la possède pas déjà.

3) quand une transaction échoue, les versions qu'elle possède sont restaurées dans la base. A la fin de la restauration, les verrous et les versions sont détruits.

Si ce protocole paraît simple, il pose néanmoins de nombreux problèmes de gestion des multiples versions générées par la pose des verrous.

Dans [BKK85], on introduit la notion de transactions coopérantes au travers du multi-fenêtrage des stations de travail. Pour un même

concepteur plusieurs transactions peuvent être exécutées en parallèle, parce qu'émises de fenêtres différentes. Ce modèle est une extension ce celui présenté dans [KLM84], qui prend en compte l'échange d'objets incomplets ou (et) incohérents. La notion de base semi-publique est introduite. Une transaction possède une base semi-publique dans laquelle elle stocke les objets modifiés que d'autres transactions peuvent extraire.

Celles-ci deviennent alors transactions filles de la transaction propriétaire de la base semi-publique. L'environnement de travail d'une transaction (les bases d'où elle peut extraire des objets) est donc composé de la base publique, et des bases semi-publiques des autres transactions (en plus de sa base semi-publique, et de sa base privée).

L'architecture du système en hiérarchie de bases de données, et calquée sur la structuration des transactions imbriquées. Si une transaction T, est fille de T', alors le système associé à T , est un sous-système de celui associé à T'.

Dans [KLM84], on n'autorise l'extraction d'objets qu'à partir des bases semi-publiques des transactions ancêtres, en effet cela permet d'éviter les problèmes de restructuration de l'arbre des transactions filles (voir figure 1).

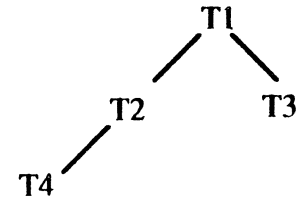


figure 1 : imbrication de transactions filles.

T4 ne peut pas extraire les objets de la base semi-publique de T3, mais seulement des bases semi-publiques de T1 ou de T2. Si T3 décide de remettre ses objets extraits dans la base semi-publique de T1, alors ceux-ci deviennent accessibles par T2 et T4.

Malgré tout, ce modèle présente quelques défauts, il considère une transaction CAO comme une séquence de transactions de courte durée. Il autorise une transaction à échanger des objets avec sa base semi-publique, à n'importe quel moment, et pas seulement en début ou en fin.

Dans [BKK85] on affine ce modèle pour résoudre ces problèmes. On structure pour cela les transactions de façon plus fine. Une transaction

est décomposée selon la nature même d'un projet de conception (voir figure 2, tirée de [BKK85]) :

transaction projet : qui regroupe les transactions des différents groupes de personnes travaillant sur la même conception.

transactions coopérantes : qui sont les transactions d'un même concepteur. En effet, compte tenu des techniques de multi-fenêtrage utilisées en conception, un concepteur peut travailler sur plusieurs tâches en parallèle. Il n'y a donc pas d'ordre à respecter dans l'exécution de transactions coopérantes.

transactions clientes sous-traitantes : cette décomposition découle du fait, qu'une tâche du concepteur peut être effectuée par un programme d'application. Celui-ci est structuré en sous-programmes qui eux-mêmes appellent d'autres sous-programmes, etc ... Certains des sous-programmes peuvent s'exécuter en parallèle. Les transactions coopérantes sont décomposées sur ce modèle. Une transaction coopérante est donc la racine d'une arborescence composée de transactions sous-traitantes (les sous-programmes), qui peuvent être en même temps, des transactions clientes (les programmes appelant). Les feuilles de cet arbre sont donc des transactions sous-traitantes, elles-mêmes composées d'un ensemble de transactions de courte durée, chacune étant une séquence d'opérations sur la base de données.

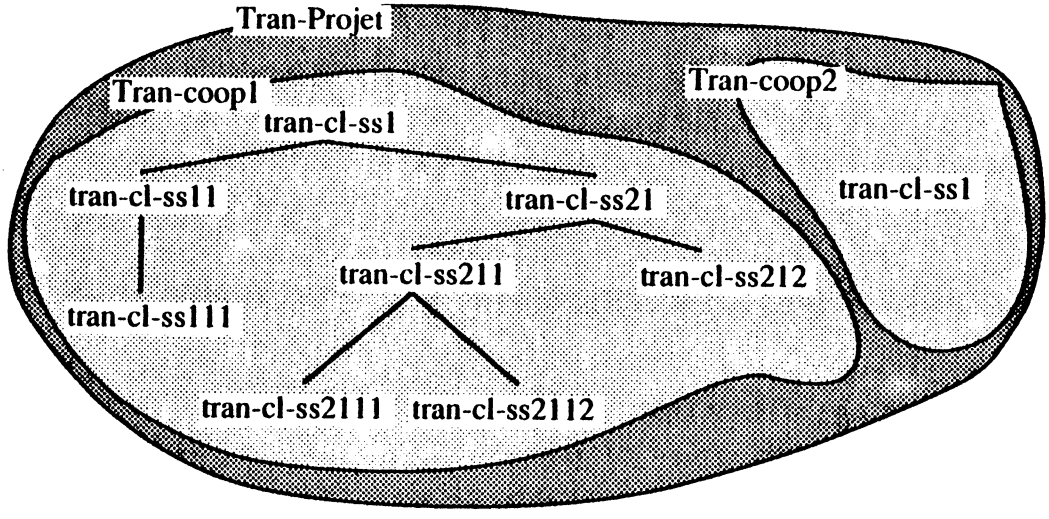


figure 2 : transactions clientes et sous-traitantes

Dans la figure 2, les *tran-cl-ss* ¡ sont les transactions clientes-sous-traitantes, les*Tran-coop* ¡ sont les transactions coopérantes et *Tran-projet* est la transaction projet.

Un tel modèle, tirant partie de la nature et de l'organisation de l'activité de la conception, permet donc :

- d'échanger des objets incomplets ou (et) incohérents.
- d'introduire un certain parallélisme dans l'exécution des opérations.
- d'intégrer les notions de versions : en effet la structuration en bases publique, privées et semi-publiques, permet d'entrevoir la gestion des versions d'un objet. On peut lier l'extraction et la remise des objets (par rapport à la base publique), à la notion de création de versions.

Dans [WAL84], les transactions sont décomposées selon un principe identique. Une application est définie par un ensemble de données et de fonctions manipulant, interrogeant, contrôlant ces données. Lorsqu'un utilisateur réfère une application il crée une transaction, et chaque fois que l'application réfère une fonction, elle crée une transaction fille pour celle-ci. Le processus se répète lorsqu'un sous-programme réfère à son tour un autre sous-programme.

On obtient une structure de transactions imbriquées similaires à la précédente. B. Walter dans [WAL84] introduit la notion de sphère de retrait et de sphère de sauvegarde. Les transactions de la même sphère de retrait se retireront en même temps (en cas d'échec par exemple), et celles de la même sphère de sauvegarde, valideront ensemble. Les transactions d'une même sphère forment un arbre. Lorsqu'une transaction T_1 crée une transaction T_2, elle indique si T_2 est une fille de T_1 (et fait donc partie de la même sphère), ou fait partie d'une autre sphère créée à cette occasion.

La notion de Saga est définie dans [GMS87], afin de prendre en compte les transactions longues et d'en résoudre certains problèmes. Une Saga est composée d'un ensemble de transactions T_1, T_2,...T_n, dont l'exécution peut être mêlée à celle d'autres transactions. Si une des transactions T_i échoue, il ne suffit pas de rendre la base dans l'état dans lequel elle était avant T_i. En effet, d'autres transactions ont été exécutées et ont modifié l'état de la base. L'idée est d'affecter à chaque T_i une

transaction de compensation C_i (i=1..n-1). Les séquences de transactions possibles sont alors :

(1) T_1, T_2.... T_n, tout c'est bien passé, la Saga entière est exécutée.

(2) T_1, T_2, ... T_i, C_i, C_{i-1},...C_1, T_i a été interrompue à cause d'une panne ou d'une détection d'incohérence.

Cette méthode, si elle admet l'exécution en parallèle de plusieurs transactions, limite néanmoins la structuration des transactions Saga à un seul niveau.

Ces différentes études montrent qu'il est nécessaire de :

- structurer les transactions en arborescence. L'atomicité des transactions est alors assurée pour les transactions feuilles de l'arbre.
- prendre en compte le fait qu'une équipe de concepteurs travaille dans un but commun, et qu'ils veulent pouvoir échanger des objets incomplets, incohérents, sans mettre en œuvre de mécanismes lourds à gérer et à utiliser. Ce point peut être résolu par l'utilisation de bases privées gérées par un système public (base publique), avec comme intermédiaires, des bases semi-publiques. Le problème de la coopération entre les concepteurs est le plus souvent résolu par un mécanisme de gestion de versions [KSW85], [KAW83] dans un environnement en hiérarchie de bases de données.
- ne pas provoquer d'attente lors d'accès concurrents, compte tenu de la durée d'une transaction. Cette contrainte peut être tenue, dans une certaine limite, en prenant en compte la structure même des objets CAO.
- ne pas considérer une transaction comme un tout, c'est à dire qu'en cas de faute, il est fortement souhaitable de restaurer le maximum du travail effectué par le concepteur depuis le dernier point de sauvegarde.

La structuration des transactions permet d'entrevoir des solutions quant à la reprise après panne. Mais les problèmes d'accès concurrents et de partage des objets restent entiers (ou presque). En effet, les auteurs proposent en majorité de partager un objet entre plusieurs concepteurs en donnant à chacun une version de cet objet. A la fin du travail de ces concepteurs plusieurs versions de l'objet ont été générées, et aucune solution n'est proposée quant à la fusion en un seul objet conçu. La notion

de version n'est pas à nos yeux une solution quant au partage de la conception. Il n'en demeure pas moins qu'il s'agit d'un besoin réel des concepteurs. La section suivante est consacrée aux différentes études portant sur le problème des versions.

Nous pensons de plus que la notion de transaction regroupant unité de cohérence et unité de reprise est à remettre en cause. Les structurations proposées (imbrication de transactions) sont fortement liées à une organisation du système en hiérarchie de bases de données. Cette architecture permet de régler les problèmes de reprise localement à chaque base de la hiérarchie. Mais les problèmes d'accès concurrents aux objets partagés ne sont pas pour autant réglés.

Nous montrons dans le chapitre 5 de ce document, qu'il est nécessaire de régler les problèmes de reprise et ceux de cohérence avec des mécanismes différents, car les informations manipulées dans les deux cas ne sont pas identiques.

3. LE TEMPS ET LES VERSIONS

3.1. La Problématique

Le temps et l'évolution des choses ont fasciné plus d'un auteur [BAR43], les concepteurs et les utilisateurs des systèmes de bases de données ne font pas exception à la règle. De nombreux domaines sont concernés par le problème de la gestion du temps et des versions. Les précurseurs en la matière sont certainement les concepteurs de logiciel. Le domaine du génie logiciel offre un terrain de recherche particulièrement riche en ce qui concerne la maintenance et l'évolution des programmes et de leurs multiples versions. Les applications de Conception Assistée par Ordinateur, posent le problème de la gestion des versions sous un angle un peu différent. En effet, le concept de version est souvent introduit pour résoudre le problème du partage d'un objet, ou pour supporter les différentes représentations d'un objet. Dans les applications orientées conception, l'activité des utilisateurs réside surtout dans la création de nouvelles informations alors que les utilisateurs des base de données classiques, visent à capturer les modifications de la réalité et à les répercuter sur les données de leurs bases. Dans ce cadre, des études ont été

faites afin de prendre en compte le temps, et de modéliser l'évolution des données dans le temps. Le concept d'historique émerge alors, plus que celui de version. On ne désire en effet, caractériser un état de la base (ou d'une partie des données de la base) que par rapport à une date, une durée ou un événement [ABC87].

Malgré tous les travaux effectués sur le sujet des versions, la terminologie utilisée est souvent différente suivant les domaines, et quelques fois même dans un domaine donné. Nous préciserons dans la mesure du possible le sens des mots utilisés dans tel ou tel domaine, et suivant tel ou tel auteur.

3.2. Les Propositions

3.2.1. En Conception Assistée par Ordinateur

Les concepteurs désirent générer et expérimenter plusieurs versions d'un objet (tentatives ou états de l'objet), avant d'en sélectionner une qui leurs soit satisfaisante. Les différentes alternatives ainsi développées utilisent divers techniques et outils, voire d'autres contraintes de conception. Plusieurs tentatives sont menées pour corriger des erreurs, améliorer les performances. Des versions antérieures (états d'un objet) sont modifiées pour les améliorer, ou essayer un axe de conception.

Tous ces besoins sont différemment pris en compte par les auteurs dans le domaine.

R.H. Katz, distingue dans [KAC86], [KCB86], trois types de liens structurels :

(1) lien de composition : les objets sont assemblés pour former un objet de plus haut niveau.

(2) lien d'équivalence : un objet est décrit par différentes représentations. Celle-ci dépendent de la technologie utilisée, ou du niveau de détail adopté. Par exemple, un circuit peut être décrit par sa représentation logique, électrique, etc .. Ces différentes représentations sont équivalentes.

(3) lien de dérivation : les différentes versions (états de l'objet) dérivées au cours de la vie de cet objet, en décrivent l'historique. Cet historique peut être structuré comme un arbre (voir figure 3).

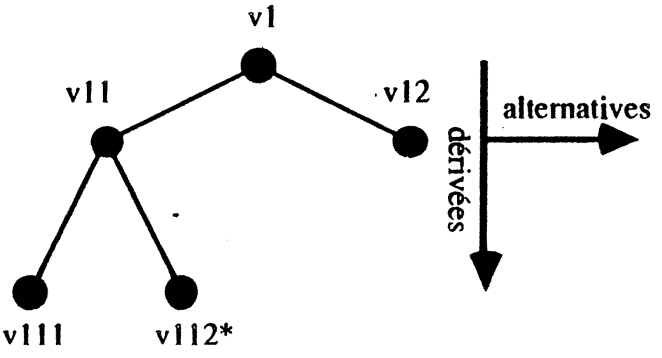


figure 3 : arbre de dérivation des versions

Plusieurs versions peuvent être dérivées de la même version. Dans la figure 3, v_{111} et v_{112} sont dérivées de v_{11}, pour cette raison, Katz considère que v_{111} et v_{112} sont alternatives. Dans un arbre de dérivation, une seule version est en cours d'évolution (ici v_{112}, marquée d'une *), il ne peut s'agir que d'une version feuille.

Katz définit le plan de composition orthogonal au plan de dérivation : la configuration. Une configuration d'un objet est une version particulière de cet objet, composée d'une version donnée de chaque objet qui le compose, et ainsi récursivement jusqu'aux objets de base. Il définit de même le plan d'équivalence des configurations d'un objet.

Pour la manipulation des versions, Katz distingue trois environnements : archivage, privé et semi-public. Une version v d'un objet est extraite de l'environnement d'archivage vers un environnement privé. Une nouvelle version v' est créée, prenant en compte les modifications effectuées sur v., puis v' est gelée et insérée dans l'environnement semi-public, où un processus de validation est appliqué. Si la version v' est valide elle est enfin insérée dans l'environnement d'archivage.

La partage d'un objet est résolu par l'affectation d'une version de cet objet à chacun des concepteurs.

Le modèle proposé dans [KBA85], [BAK85] est basé sur la même distinction entre versions et alternatives. Le modèle d'objet diffère car il ne prend pas seulement en compte la structure hiérarchique des objets. En plus de sa composition, un objet est décrit par son **interface**, et sa **réalisation**. Plusieurs réalisations d'un même objet peuvent être associées à la même interface (voir figures 4, 5, 6, tirées de [KBA85]), elles sont alors les différentes versions de cet objet. Le circuit décrit par les figures suivantes réalise l'addition de deux bits x et y, avec la retenue en entrée c_{in}, le résultat de l'addition est z, et la retenue en sortie est c_{out}.

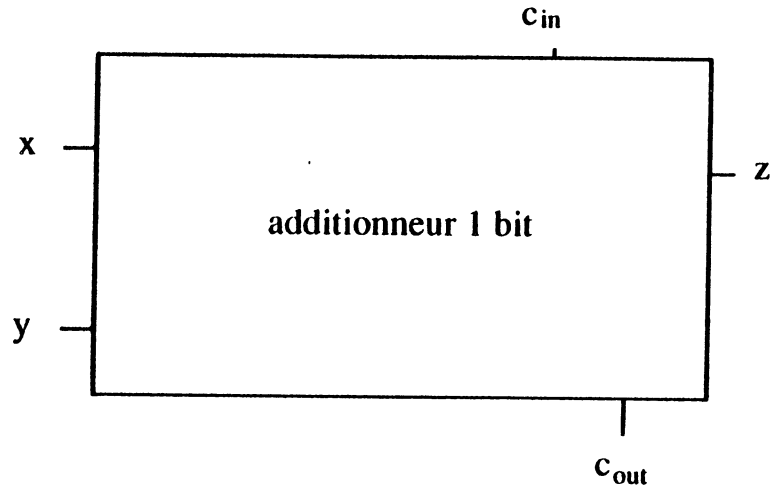


figure 4 : interface de l'additionneur 1 bit

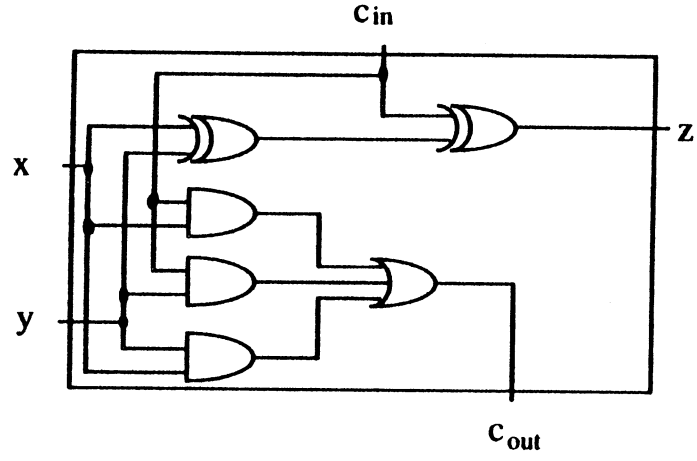


figure 5 : réalisation de l'additionneur en portes logiques

La figure précédente montre une réalisation de l'additionneur à l'aide des portes logiques XOR, OR et AND. Le choix de conception est ici d'utiliser des composants de base.

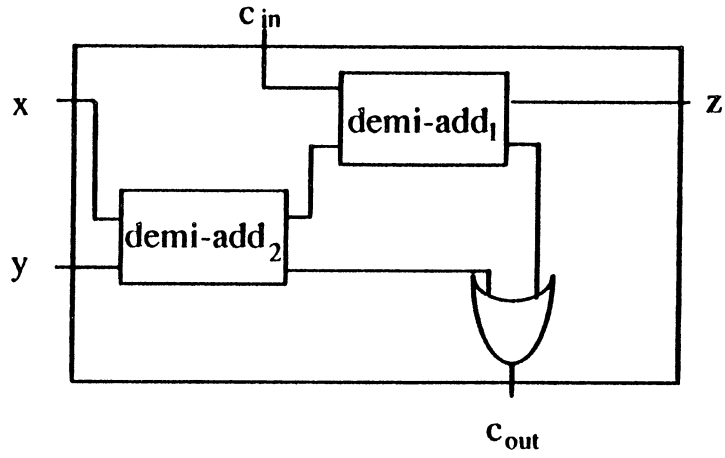


figure 6 : réalisation de l'additionneur en sous-circuits

La figure 6 montre une réalisation utilisant deux instances du circuit demi-add (demi-additionneur). Ici, le choix de conception est de décomposer le problème en sous problèmes, et combiner ensemble des circuits, afin de décrire des circuits de plus en plus complexes.

La figure 6, illustre l'utilisation d'un autre concept fondamental, qui est celui d'**instances** : l'additionneur 1 bit, utilise deux instances du demi-additionneur.

Seules des versions d'interfaces et de réalisations peuvent être définies.

Une extension de ce modèle d'objet et de versions [KBA85] est proposée dans [CKI86]. L'architecture du système proposé est basée sur une hiérarchie de bases de données (bases de données privées et semi-publiques, et base de données publiques). Une classification des versions en trois groupes est proposée dans cet environnement, celle-ci est faite selon la situation de la version dans la hiérarchie des bases de données :

(1) version **transitoire** : elle ne peut être modifiée ou détruite que par le concepteur qui l'a créée, elle réside dans la base privée du concepteur.

(2) version de **travail** : elle est stable, et ne peut être modifiée. Elle peut être supprimée par le concepteur qui l'a créée. Elle réside dans une base privée, ou semi-publique.

(3) version **stabibilisée** : elle ne peut être ni modifiée, ni détruite. Elle est gérée par le système public, dans la base de données publique.

Les mécanismes associés à ces différentes versions sont :

(1) la **promotion** : une version transitoire peut être explicitement déposée dans un système semi-public, elle devient alors version de travail

dans ce système, elle peut être ensuite promue dans le système public, elle devient alors version stabilisée.

(2) la **dérivation** : une version peut être dérivée d'une quelconque des versions (transitoire, de travail ou stabilisée), elle est alors une version transitoire, et réside dans le système privé du concepteur qui l'a dérivée. La dérivation d'une version à partir d'une version transitoire, provoque la promotion de cette dernière, et son insertion dans le système semi-public père.

Les études décrites ci-dessus visent à offrir des modèles prenant en compte la sémantique des versions d'objets CAO. L'accent est particulièrement mis sur la structuration (version alternatives et successives [KAC86], [KCB86], versions transitoires, de travail et validées [KBA85], [CKI86]) et l'évolution des versions (dérivation, promotion, gel et dégel [DIL85]).

L'étude présentée dans [KSW86] propose un partitionnement des versions d'un objet selon leurs propriétés. Le modèle de versions est similaire à celui décrit dans [CKI86] et [KBA85], le temps intervient dans la structuration des versions. Les versions sont en effet ordonnées de façon linéaire suivant leur date de création. Elles sont de plus liées entre elles par la relation (non linéaire) de dérivation.

C. Palisser se place dans les applications plus particulière du devis descriptif en CAO Architecture, et présente dans [PAL87] et [APA87] un modèle de versions basé sur l'approche de Katz [KAC86], [KCB86]. L'objectif est de prendre en compte l'évolution de l'objet à travers ses différentes représentations et les différentes alternatives de conception. Les mécanismes offerts sont basés sur les notions de validation, d'archivage de gel et de dégel [KAC86].

L'approche adoptée dans [RIE86] prend en compte l'ensemble des représentations d'un objet CAO à des niveaux morphologiques différents. Elle distingue les représentations d'un objet de leur implantation, et introduit pour cela la notion de versions de représentation et de versions d'implantation. Cette distinction est à rapprocher de celle faite dans [BAK85] .

3.2.2. En Génie Logiciel

Le problème du contrôle des versions est crucial dans le domaine du génie logiciel. Dans [BES86] un logiciel est vu comme un ensemble de

modules. Un module est en fait l'association d'une interface et d'un corps de programme. La notion de version est utilisée pour décrire les différentes alternatives possibles pour l'écriture d'un algorithme réalisant la même interface. Chacune de ses alternatives est une **version de réalisation**. Des modifications mineures peuvent être apportées à une telle version, pour corriger des erreurs par exemple, le nouvel objet prenant en compte ces modifications est une **révision**. La nuance entre révision et version de réalisation tient dans l'importance des modifications.

Des **versions d'interface** sont définies pour prendre en compte l'évolution des fonctionnalités du logiciel.

Enfin, une **configuration** est un ensemble de versions d'interface et de réalisations nécessaires à l'exécution du logiciel correspondant dans un environnement donné (par exemple, sous système d'exploitation UNIX).

Cette diversité des concepts est induite par la complexité et la difficulté d'expliciter la sémantique d'un programme. Il n'est pas possible en effet, comme en CAO de VLSI par exemple, de connaître la composition d'un objet en terme d'éléments aussi simple que des portes logiques. L'élément de base en génie logiciel est le texte, qui peut être programme, documentation, etc ..., et la seule connaissance que l'on peut avoir sur cet élément est donnée par des attributs qu'on lui associe généralement : date de création, type de programme ou de documentation, système d'exploitation, etc ...

Les différentes études [TIC85], [WINK], [BES86] montrent qu'un mécanisme de contrôle de versions est nécessaire. Un système de gestion de versions dans un environnement de génie logiciel, doit limiter le nombre de versions (on ne génère pas une version, ou une révision, parce qu'on a rajouté un caractère dans un programme).

Le modèle de versions généralement adopté ne prend en compte que l'évolution linéaire des versions. La notion d'alternative n'apparaît que dans [NOM88]. Une version n'est caractérisée que par son nom, et par des attributs (date, version livrée, etc..).

Bien que le domaine du génie logiciel soit précurseur en matière de gestion de versions, les modèles proposés restent néanmoins assez pauvres.

4. LE PROJET DE CAO, NOTRE APPROCHE

Le bilan de cet état de l'art est qu'aucune solution satisfaisante n'est proposée. Les accès concurrents aux objets sont résolus en partie par une architecture en hiérarchie de systèmes. Ainsi un concepteur travaille dans **son** système privé. S'il copie un objet partagé, il le fait à ses risques et périls. Le problème de la coopération entre les concepteurs est résolu en offrant des solutions qui à notre sens s'intéressent à une toute autre problématique : la gestion des versions d'objets CAO. En effet, si plusieurs concepteurs conçoivent chacun une partie d'un même objet, une copie de cet objet (une version) est donnée à chacun. A la fin de la conception, autant de versions qu'il y a de concepteurs, sont produites.

L'échec de ces différentes études réside dans le fait qu'aucun mécanisme n'est proposé pour mettre en évidence et gérer les interactions entre les concepteurs d'un même projet de CAO.

Notre approche est basée sur la question suivante : nous disposons d'un SGDB pour la CAO, dans un environnement mono-utilisateur comment un tel système pourrait-il être utilisé dans un environnement partagé ?

Nous tentons ici, d'y répondre :

Dans un projet de CAO, comme dans toute organisation d'entreprise, un ensemble d'équipes est chargé de la réalisation d'un projet. Il s'agit donc pour ces équipes de se partager le travail à réaliser, suivant les compétences des gens, ou leurs affinités (!?!). Ce projet peut être de nature très variée : architecture, réalisation d'un document, conception d'un circuit VLSI, d'un logiciel, etc ...

Chaque équipe doit réaliser la partie du travail qui lui est assignée dans le respect de certaines contraintes. En effet, l'ensemble des travaux réalisés par les équipes doit être **cohérent** par rapport au cahier des charges du projet. Le fait d'affecter une tâche à chaque équipe ne cloisonne pas leurs activités respectives. En effet, de nombreux objets stables ou non, sont échangés de façon informelle, et ne sont pas pris en charge par des mécanismes complexes de validation ou d'évaluation [KLM84], [KAW83], [KAL82], [BKK85], [LOR83].

Ce processus de partage (et de fonctionnement) peut se réitérer au sein de l'équipe, sur un nombre quelconque de niveaux. A chaque niveau, il est nécessaire en fin de conception, de vérifier que l'*assemblage* de chaque

travail est cohérent par rapport aux buts fixés. Enfin au plus haut niveau (celui du projet) l'objet conçu est le résultat de l'assemblage des différents travaux. Il doit répondre aux besoins formulés dans le cahier des charges.

Un système de gestion de base de données pour la CAO doit donc offrir un outil de définition et de gestion de projet, qui permette et gère le partage des objets sur un nombre quelconque (dans l'absolu) de niveaux. Il doit offrir aux concepteurs (qui agissent à un niveau quelconque) le moyen de travailler en équipe, c'est à dire de s'échanger des objets, des connaissances ou des idées, tout en respectant les objectifs fixés au départ (et en toute sécurité). La description du projet et de son organisation est faite par un utilisateur particulier : l'administrateur du projet. Lui seul possède le droit de décrire et de modifier l'organisation du projet et de l'activité de conception.

Nous proposons un système de gestion de projet avec une architecture en hiérarchie de sous-systèmes. L'idée est de fournir un environnement informatique dont l'architecture est calquée sur l'organisation de l'entreprise. Le système proposé est placé dans un environnement centralisé (figure 7). L'aspect réparti (serveur et stations de travail connectés sur un réseau local) ne sera pas abordé ici. La figure 7 montre comment s'organise un tel système de gestion de projet. Le cloisonnement entre les différents systèmes décrits dans la figure 7 est purement logique. Les systèmes cohabitent sur le même site (la même station de travail).

De manière très sommaire, le système est constitué d'un système public, d'un ensemble de sous-systèmes semi-publics, et privés. Le système public est lui-même partagé en deux zones : l'une contenant toutes les données cohérentes et complètes (ce sont des données figées), et l'autre, toutes les données qui font l'objet d'échanges entre les sous-systèmes fils. Le système public sert de plus à gérer les différentes équipes qui composent le projet. Il gère en fait, un ensemble de sous-systèmes (des systèmes semi-publics) propres à chacune des équipes définies par l'administrateur du projet.

Les systèmes semi-publics, gèrent les données partagées entre les différents sous-systèmes fils. Ces données sont incomplètes, et parfois incohérentes.

Chaque système gère ainsi un ensemble de sous-systèmes, jusqu'à atteindre les systèmes privés des concepteurs. Ces systèmes privés ne sont accessibles que par le concepteur propriétaire.

La base de programmes contient tous les programmes, qui sont les outils CAO, nécessaires à la réalisation de fonctions de calcul (ou de vérification) référencées dans un quelconque des systèmes (privés, semi-publics, publics). Ces programmes (en C, PASCAL, FORTRAN ..) pré-existent, il est donc nécessaire de les ré-utiliser sans avoir à tout ré-écrire. Chacun des systèmes peut accéder à cette base.

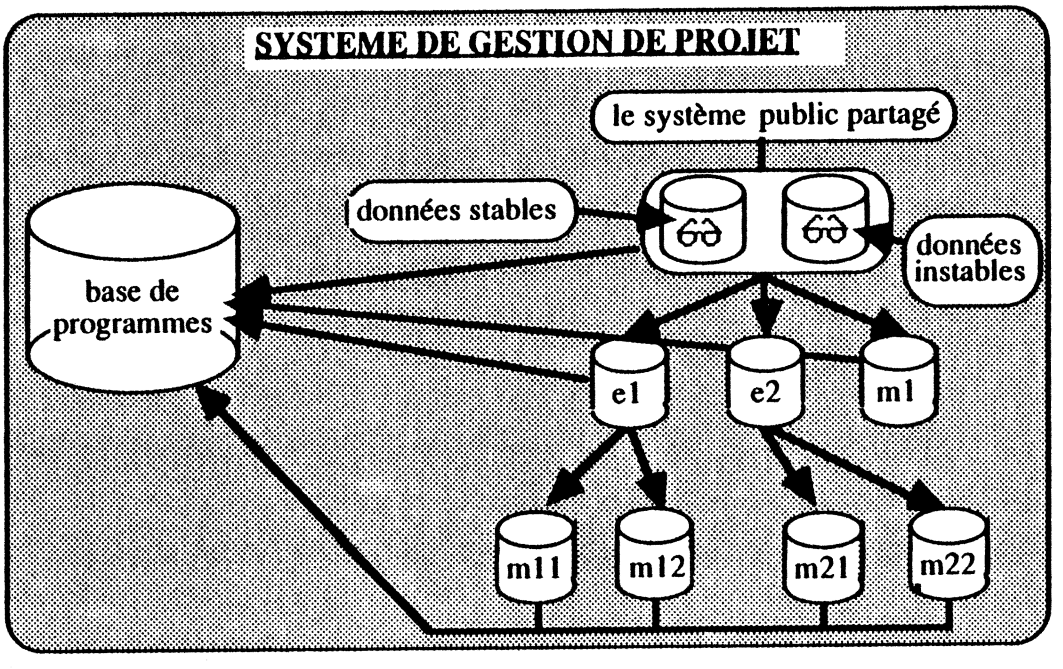


figure 7 : architecture d'un système de gestion de projet CAO.

Dans la figure 7, le système semi-public dédié à l'équipe e_1 constitue l'environnement de communication et d'échanges entre les systèmes privés des concepteurs m_{11} et m_{12}. De même, le système public constitue l'environnement d'échanges entre les systèmes semi-public des équipes e_1 et e_2 et le système privé du concepteur m_1.

Une telle architecture est souvent proposée pour résoudre les échanges entre les utilisateurs [KLM84], [KAW83], [KAL82], [BKK85], [LOR83]. Dans ces différentes études, deux systèmes privés peuvent s'échanger des objets instables, c'est à dire incomplets et (ou) incohérents. Ces échanges sont faits sans mise en oeuvre des mécanismes de validation,

car ils transitent par les systèmes semi-publics. Les objets ne sont évalués, ou validés que lorsqu'ils sont déposés dans le système public.

Nous proposons une organisation similaire permettant ainsi la coopération entre les concepteurs (ou entre les équipes) grâce à l'assouplissement des échanges. Nous ne figeons pas à priori le nombre de niveaux dans la hiérarchie des systèmes qui composent le projet. En effet, celle-ci reflète la structure d'un projet composé d'équipes, elle-mêmes composées d'équipes plus petites. Un système semi-public est associé à une équipe, et un système privé est associé à un concepteur (un membre d'une équipe).

La structuration d'un système CAO en hiérarchie de bases de données permet la définition d'environnements de travail privés et partagés. Il est nécessaire alors de décrire des protocoles d'échanges entre ces environnements. De tels protocoles doivent décrire et gérer le partage du travail entre les systèmes. Ils doivent bien sûr assurer la fusion des différents travaux.

5. CONCLUSION

Nous avons vu dans ce chapitre les solutions proposées dans la littérature pour résoudre les problèmes de gestion de transactions et de gestion de versions. Le partage des objets et de l'effort de conception est généralement abordé dans le cadre des études faites sur les transactions CAO. Nous pouvons faire plusieurs critiques sur les diverses solutions proposées.

La structuration des systèmes CAO en hiérarchies de bases de données, permet de contrôler l'accès aux objets, ainsi que leur état selon leur localisation. Ainsi une base de données publiques contient des objets stables (au sens terminés) et non modifiables, une base de données privées contient des objets en cours d'évolution et non copiables par les concepteurs autres que le propriétaire de la base. Il faut noter de plus qu'une telle structuration reflète l'organisation d'un projet de conception en équipes de concepteurs. Nous proposons des mécanismes spécifiques pour l'expression et la gestion de la coopération dans l'activité de conception et pour le partage des objets. Ces différents mécanismes définissent les protocoles de communication entre les systèmes public, semi-publics et privés.

Le problème posé par la gestion des versions en CAO, demeure un problème en tant que tel, et doit être résolu afin d'être intégré à un système partagé de gestion d'objets CAO.

Alors que les modèles de versions proposés en Génie Logiciel restent à nos jours assez pauvres, ceux développés dans le cadre des applications CAO (en architecture ou VLSI) offrent de nombreux concepts quelques fois difficiles à appréhender. En effet, offrir un système puissant c'est bien, mais si ce système est trop complexe pour être correctement utilisé c'est dommage. L'idée de base dans notre approche, est d'offrir un modèle avec peu de concepts, mais offrant des fonctionnalités qui permettent aux utilisateurs de personnaliser leur système et de le faire évoluer selon leurs besoins.

La première partie de notre étude a consisté en la réalisation du système CADB [FAU86], [FAR87], [FAR87b], qui deviendra le serveur d'objets de ETIC. Ce système réalise un sous ensemble minimum des fonctionnalités du modèle de données défini dans [RIE85]. Un formalisme de ce modèle de données est proposé dans le chapitre 2.

Une seconde partie de l'étude est consacrée à la définition des concepts nécessaires pour prendre en compte l'activité coopérative de concepteurs dans un cadre de projet [FAU87b]. Le chapitre 3 est consacré à cette étude.

Enfin une dernière partie de l'étude (chapitre 4) a visé à définir puis intégrer, les concepts de versions dans un environnement partagé et coopératif.

Grâce aux différents résultats obtenus, le système de transactions mis en œuvre dans ETIC se trouve simplifié. Le chapitre 5 met en évidence cette simplification et présente le gestionnaire des transactions.

Le résultat de notre étude se situe à plusieurs niveaux :

(1) la définition de concepts nécessaires dans un environnement CAO multi-concepteurs.

(2) la définition d'un modèle de version dans l'environnement défini ci-dessus. L'intérêt et l'originalité de cette approche, sont l'intégration d'un système évolutif de gestion de versions dans un environnement où le partage de l'activité de conception est géré par le système.

(3) la réalisation du prototype ETIC : la spécification des concepts décrits en (1) et (2) ont permis la réalisation du système ETIC que nous présentons au chapitre 6.

Enfin, dans le chapitre 7, nous concluons notre étude et nous proposons un ensemble de perspectives.

CHAPITRE 2
LE MODELE DE DONNEES

table des matières

1. INTRODUCTION

Nous présentons ici le modèle de données support de notre étude, défini par [RIE85], et nous en proposons une formalisation. Afin de faciliter la compréhension, nous avons replacé le vocabulaire utilisé par rapport au consensus qui paraît se mettre en place dans le domaine des modèles d'objets complexes et des modèles orientés objets.

Notre but ici n'est pas de redéfinir un modèle de données original, mais seulement de définir les concepts de base qui sont nécessaires à notre étude.

Nous nous intéressons donc, dans ce chapitre à la formalisation de l'objet à concevoir sans prendre en compte les différents états qu'il prend au cours du temps. Nous supposons que toutes les modifications sont faites sur l'objet, nous ne gérons pas les multiples versions d'un objet qui peuvent résulter de ces modifications ou de différents choix de conception. Le chapitre 4 est consacré à la prise en compte de ces besoins.

Un objet CAO est composé d'un ensemble de sous objets qui sont eux-mêmes des objets à concevoir, [KAT85], [DLO85], [BAK85]. Un objet à concevoir (OAC) est donc décrit par une arborescence (figure 1) dont les feuilles sont des objets de structure indécomposable (par exemple en CAO de VLSI : les ports d'entrée ou de sortie, les portes logiques, les entiers, etc...). Une telle structure de l'objet à concevoir est classiquement adoptée en CAO car elle résulte d'une analyse descendante de décomposition du problème en sous-problèmes.

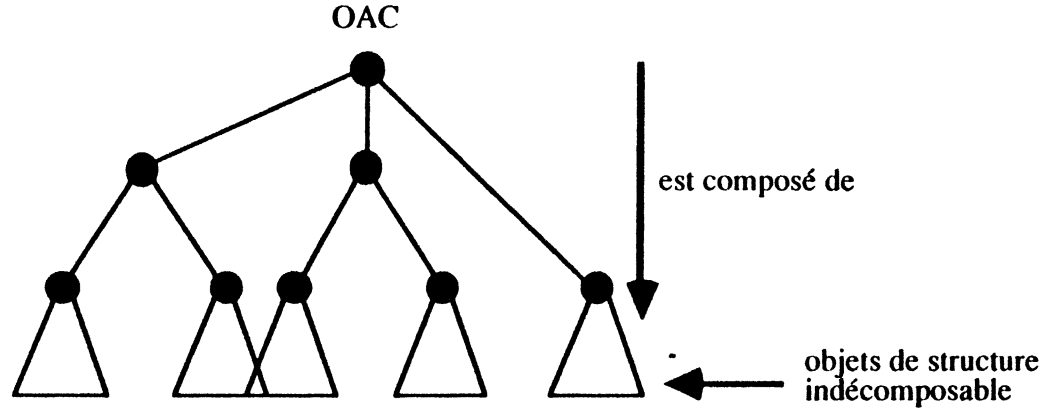


figure 1 : structure hiérarchique de l'objet à concevoir

En CAO de VLSI (Very Large Scale Integration) par exemple, exprimer la composition d'un circuit en sous circuits est une chose naturelle.

Un objet à concevoir est donc décrit par un arbre dont les feuilles sont des objets de base, et les noeuds l'expression de leur composition. Un objet peut entrer dans la composition de plusieurs objets de niveaux supérieurs. Un objet donné n'existe qu'en un seul exemplaire. On ne manipule que des **références** sur des objets. En effet dupliquer sur un même site, un objet autant de fois qu'il est utilisé, poserait des problèmes de place mémoire, de mises à jour et de vérification de la cohérence. Un objet est référé par un identificateur unique (voir section 3.3).

A chaque noeud on peut associer un ensemble de **contraintes d'intégrité** pour exprimer la cohérence de la composition. Celles-ci peuvent porter sur l'ensemble ou une partie des noeuds fils. De la même façon, il est possible de donner un ensemble de **liens** pour exprimer le calcul d'un ou de plusieurs noeuds. Il peut s'agir par exemple du calcul d'un composant, à partir d'un autre. Ce lien peut mettre en jeu un calcul, ou tout simplement une affectation (voir section 2.2.2).

De précédentes études ont porté sur la modélisation d'objets structurés ou complexes [ABI87], [COL88]. De nombreux modèles orientés objets émergent : Galileo ([AGO86]), O_2, ([LRV87]), Guide ([KRA87]).

Dans un environnement CAO, les concepteurs désirent connaître la structure des objets qu'ils manipulent. Pour cela, le principe d'encapsulation cher aux modèles orientés objets et qui restreint dans un certain sens la liberté du programmeur nous paraît trop rigide.

Nous permettons de plus l'expression de **contraintes** et de **liens** sur la structure des objets. Ces contraintes et ces liens sont activés automatiquement au moment approprié. Nous verrons au chapitre 6, les solutions offertes quant à l'exécution des contraintes (exécution au plus tôt, exécution différée).

Dans O_2, ou Guide, l'utilisation des méthodes ne permet pas la gestion de telles contraintes. La prise en compte des contraintes est exprimée par le programmeur dans l'écriture des méthodes. C'est lui qui gère la cohérence de son application. Cette approche procédurale pose problème, car il est nécessaire de prévoir la programmation des contraintes

à chaque fois qu'une méthode affecte les valeurs de l'objet, c'est à dire d'introduire la notion de déclencheur de contraintes.

Pour la définition du modèle de données, nous utilisons un formalisme inspiré de celui introduit dans [ABI87], [LER87].

2. LES TYPES

Les types permettent d'exprimer la structure des objets. Ils représentent des familles d'objets présentant des propriétés semblables et contraints de manière identique. Un type permet d'exprimer en une seule notion, le concept d'entité et de relation entre entités.

On modélisera par exemple, un type *Segment* par une *Extrémité* et une *Origine*. La relation qui lie deux points dans un espace est ici représentée par le schéma[1] de *Segment*.

2.1. les types de base

Ce sont les types qui servent de base à la conception. Il s'agit des schémas des objets de structure indécomposable qui sont les entiers, les réels, les caractères, les valeurs booléennes, et les chaînes de caractères, pour lesquels nom et valeur sont confondus (par exemple, l'entier de valeur 2, est identifié par 2). Dans certains domaines, les types de base peuvent être différents. En CAO de VLSI (Very Large Scale Integration) les types de base sont les entrées/sorties, les équipotentielles etc ...

2.2. les types construits

Un type construit est défini par son nom, ses attributs (A), les liens (L) et les contraintes (C) portant sur ses attributs, il est donc décrit par le triplet (A, L, C).

2.2.1. Les attributs

Les attributs d'un type sont donnés par :

---

[1] Pour alléger les tournures de phrases, nous emploierons indifféremment les termes schémas et types, qui sont à notre sens équivalents.

$A = [attr_1 : A_1,...,attr_n : A_n]$, avec $attr_i \neq attr_j$, pour tout i, j = 1, n. A est la partie structurelle du type; le plus souvent, A est utilisé pour nommer le type. L'expression $attr_i : A_i$ signifie que l'attribut $attr_i$ prend ses valeurs dans A_i. A_i est un type préalablement défini, il peut être construit, ou de base.

Le constructeur LIST permet d'exprimer qu'un attribut est de type nuplet :

$attr_q$: LIST A_q (min,max)

qui indique que l'attribut $attr_q$ est instancié par une liste d'objets de type A_q. Le couple (min, max) est facultatif. Le dimensionnement donné par *min* dans la description de $attr_q$, introduit une règle sur la complétude de l'objet, et celui donné par *max* dans cette même description introduit une règle sur sa cohérence. Si le nuplet $attr_q$ admet plus de *max* valeurs, l'objet est **incohérent**, s'il en admet moins de *min* , il est **incomplet**. Nous reviendrons sur les notions de complétude et de cohérence dans la section suivante (§ 3).

Nous avons vu qu'un attribut ne peut prendre ses valeurs que dans un type préalablement défini, c'est à dire connu, de ce fait il est impossible dans ce modèle, d'exprimer directement des énoncés cycliques. Nous verrons plus loin (section 5), une solution à ce problème, apportée par le constructeur ISA.

2.2.2. Les Liens et les Contraintes d'Intégrité

Les Liens et les Contraintes d'Intégrité expriment des connaissances génériques détenues sur un ensemble d'objets. Ils permettent d'introduire des mécanismes de gestion des structures, et de la cohérence des liaisons entre les objets.

La définition du type Segment (figure 2) illustre l'utilisation de lien et de contrainte. Pour Segment, il est nécessaire de définir auparavant le type Point :

Point	= ([x : Entier, y : Entier],[],{}	*pas de lien ni de contrainte*
Segment	= ([org : Point, ext : Point, lg : Entier],	
	[lg := distance (org,ext)],	*liste des liens*
	{ NOT confondues (org,ext) })	*liste des contraintes*

figure 2 : définition des types Point et Segment

Le concepteur considère ici qu'un Segment est décrit par son origine (*org*), son extrémité (*ext*) et sa longueur (*lg*).

Une Contrainte d'Intégrité exprime une relation entre les attributs d'un type. Une contrainte ainsi définie doit être respectée par tous les objets du type où elle est définie. Si un objet ne respecte pas une telle contrainte, il est **incohérent**.

Dans la figure 2, le concepteur exprime par la contrainte d'intégrité *NOT confondues (org, ext)* , que l'extrémité et l'origine des objets de Segment ne doivent pas être confondues. Lors de toute manipulation d'objet dans Segment, le système évalue (ou réévalue si nécessaire) cette contrainte afin de vérifier que celui-ci est cohérent. Il faut noter que l'échec dans l'évaluation d'une contrainte ne bloque pas le concepteur. Le système se contente de lui en signaler la présence et accepte une incohérence passagère. Dans le cas où les paramètres d'une contrainte ne sont pas tous connus, l'objet reste cohérent. Le fait de ne pas pouvoir évaluer une contrainte ne délivre pas d'incohérence. Par exemple, si l'extrémité ou l'origine d'un Segment n'a pas de valeur (est inconnue), la contrainte n'est pas évaluée et l'objet reste malgré tout cohérent.

Un **Lien** s'applique à un attribut du type auquel il impose une valeur, soit par un calcul, soit par une affectation. Il est calculé systématiquement et automatiquement par le système à la création de l'objet. Chaque fois que celui-ci est modifié il est recalculé si nécessaire .

De manière générale, les **liens** sont de la forme :

- $attr_q := fct\ (liste\text{-}paramètres)$

(le ":=" est utilisé ici, pour représenter l'affectation).

fct est une fonction décrite par l'usager où la liste des paramètres est conforme à sa signature (voir section 5);

• $attr_q$:= *désignation-attribut*
désignation-attribut est une désignation hiérarchique d'un attribut du type. Par exemple : l'expression *x.org* désigne l'abscisse de l'origine d'un objet de type Segment.

De même les **contraintes d'intégrité** sont de la forme :
• *fct (liste-paramètres)*
où fct est une fonction décrite par l'usager (voir section 5), et liste-paramètres la liste des paramètres conforme à la signature de la fonction.

3. LES OBJETS

3.1. Définitions

Nous pouvons au préalable donner un ensemble de définitions afin de clarifier la terminologie utilisée dans le cadre du modèle de données présenté ici :

1 : Un objet à concevoir est décrit par son type, (c'est à dire des attributs, des liens et des contraintes sur ses attributs) et par une instance de chacun de ses attributs.

2 : Concevoir un objet revient à décrire son type et à instancier ses attributs, et ceci récursivement sur chacun des sous objets. Instancier un attribut, c'est lui associer un identificateur d'objet. Celui-ci peut identifier un objet de base (de structure indécomposable) ou un objet à concevoir qui est alors nommé de manière non ambiguë.

3 : un attribut est **calculable** s'il existe un **lien** qui exprime son calcul. Les paramètres de lien peuvent être soit des attributs du type, soit des objets.

4 : un attribut calculable est **calculé** dès que tous les paramètres du lien qui le concerne sont connus, c'est à dire instanciés par des sous-

objets dont tous les attributs sont instanciés, et ceci récursivement jusqu'aux sous-objets de base.

5 : un objet est **complet** si et seulement si tous les attributs non calculables définis dans son type sont instanciés et si tous les attributs calculables sont calculés. Sinon, il est **incomplet**. Cette notion de complétude est en quelque sorte locale à l'objet. Par exemple, un objet peut être complet, même si un des sous-objets qui le composent est incomplet.

6 : une contrainte est **évaluable,** pour un objet donné, si elle est définie sur des attributs instanciés par des sous-objets dont tous les attributs sont instanciés, et ceci récursivement jusqu'aux sous-objets de base. Son évaluation est alors possible et délivre l'une des deux valeurs *vrai* ou *faux* .

7 : un objet est **cohérent** si et seulement si, toutes les contraintes évaluables définies sur les attributs, sont évaluées à vrai. Il est **incohérent** si au moins une contrainte évaluable, est évaluée à faux.

3.2. Généralisation

Un objet à concevoir est donc entièrement décrit par son type (**A, L, C**), et sa composition, c'est à dire l'instanciation de ses attributs (voir figure 3). La description des éléments du type est donnée par l'administrateur du projet, elle est nécessaire à l'expression de l'organisation du projet.

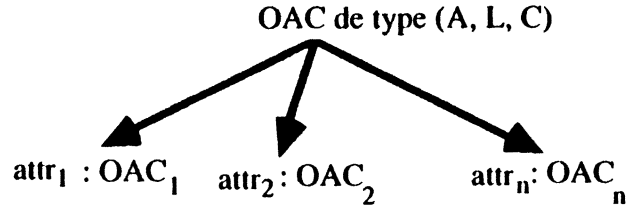


figure 3 : l'arbre de composition de OAC de type (A, L, C)

A un niveau quelconque de l'arbre décrivant la composition de l'objet à concevoir OAC, et pour le j^{ième} composant de ce niveau, on a donc :

$OAC_{i..j} = [attr_{i..j1} : OAC_{i..j1}, ...,attr_{i..jq} : OAC_{i..jq}]$,

d'un type $(A_{i..j}, L_{i..j}, C_{i..j},)$:

$A_{i..j} = [attr_{i..j1} : A_{i..j1}, ...,attr_{i..jn} : A_{i..jn}]$
$L_{i..j} = [L_{i..j1},...,L_{i..jp}]$
$C_{i..j} = \{C_{i..j1},...,C_{i..jr}\}$

$L_{i..j}$ est une séquence de liens $L_{i..j1},...,L_{i..jp}$, telle que pour tout $q \neq t$, si $q<t$ alors L_q est évalué avant L_t, avec $q \in [1..p]$ et $t \in [1..p]$. L'ordre des liens donnés dans Li..j, est laissé à la discrétion du concepteur. Il faut en effet s'assurer que si $L_{i..jp}$ utilise une valeur délivrée par $L_{i..jq}$, alors $Li..j_q$ est exécuté avant $L_{i..jp}$.

Un lien ou une contrainte est évalué immédiatement, c'est à dire dès que les attributs qui en constituent les paramètres sont instanciés. Le concepteur peut s'il le désire spécifier qu'une contrainte ne sera évaluée que lorsque tous les attributs seront instanciés. Il appartient au concepteur de choisir selon les cas, l'évaluation différée d'une contrainte, ou immédiate pour détecter au plus tôt d'éventuelles incohérences.

Cette description de $OAC_{i..j}$ est illustrée par la figure 4 :

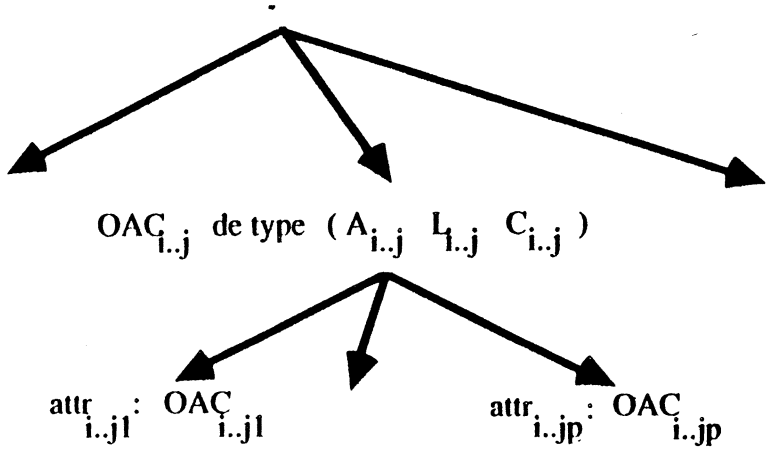


figure 4 : niveau quelconque de composition

Dans cette figure, les sous objets $OAC_{i..jq}$ pour q=1,p, sont de type ($A_{i..jq}$, $L_{i..jq}$, $C_{i..jq}$).

La récurrence s'arrête lorsqu'un type $T_{i..jq}$ est un type de base. Il peut exister des objets sans contrainte ($C_{i..jq} = \{\}$), ou (et) sans lien ($L_{i..jq} = []$). Un exemple d'objet à concevoir est donné plus loin dans la section 7.

3.3. Identification

Il y a deux types d'identificateurs :

(1) identificateur externe : il est composé du **nom** de l'objet, et de son type, il est unique dans la base des objets. Le nom est affecté par l'usager à la création de l'objet. Ce nom est unique dans le type de l'objet, et dans tous les types de la hiérarchie des types définie par la spécialisation.

C'est le moyen dont dispose l'utilisateur pour désigner sans ambiguïté tout objet. Le nom d'un objet peut être changé par l'usager.

(2) identificateur interne : c'est une valeur entière unique pour tous les objets, et inaccessible par l'usager. Ce surrogate est affecté une fois pour toutes, et ne peut pas être changé.

Lorsqu'un objet O est utilisé pour instancier un attribut de O', seul l'identificateur système (le surrogate) de O, est référencé dans O'. En d'autres termes, il n'y a pas copie de O dans O', un objet est partagé par un ensemble d'autres objets.

4. EXEMPLE

Nous donnons ici un exemple très simple de la conception d'un objet. Nous donnons la définition du type, et la description d'objets de ce type. Soit Segment, le nom du type de cet objet. Si nous choisissons de décrire un segment par son origine, son extrémité et sa longueur, Segment est décrit par (on identifie le nom du type et sa partie structurelle) :

- **Segment** ([origine : **Point**, extrémité : **Point**,
longueur : **Entier**], $L_{segment}$, $C_{segment}$)

où on a :

le type **Point** : ([abcisse : **Entier**, ordonnée : **Entier**] ,∅, ∅)
et dont les ensembles L_{Point} et C_{Point} sont vides,

Entier est un type de base, il n'a donc ni attribut, ni lien, ni contrainte.

- $L_{Segment}$ = [(longueur, dist(origine, extrémité))], qui exprime le calcul de la longueur par la fonction distance appliquée à l'origine et l'extrémité.
- $C_{Segment}$ = {NOT conf(origine, extrémité)}, qui indique qu'un Segment est cohérent si son origine et son extrémité ne sont pas confondues.

L'arbre de composition associé à un objet s1 de type Segment est donné dans la figure 5 suivante :

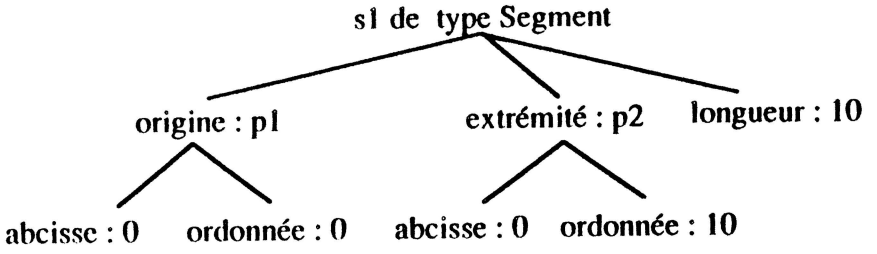


figure 5 : s1, complet et cohérent

En appliquant les définitions données dans la section 3.1, on peut dire que s1 est **complet** et **cohérent** dans Segment. Le lien est calculé, la contrainte évaluable est évaluée à vrai.

Nous pouvons donner d'autres exemples d'objets de type Segment :

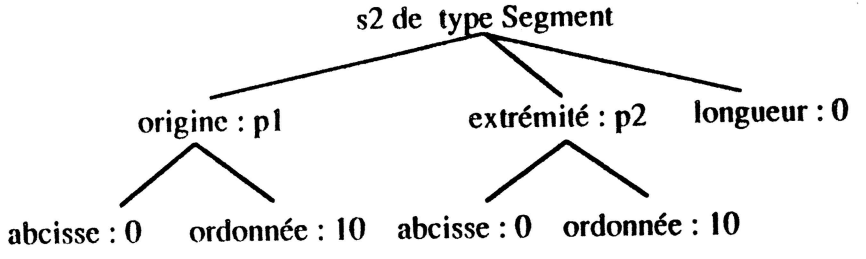


figure 6 : s2, complet et incohérent

s2 est **complet** et **incohérent** dans Segment, tous les attributs non-calculés sont instanciés, la longueur est calculée, et la contrainte est évaluée à faux (p1 et p2 sont confondus).

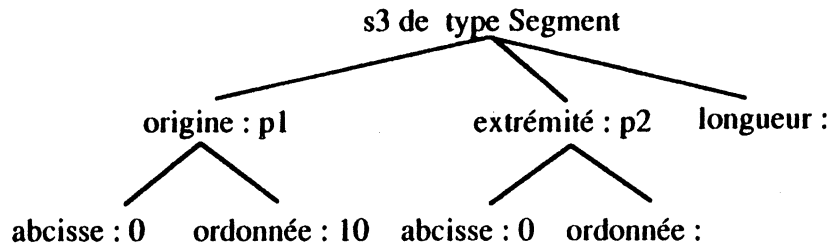


figure 7 : s3, incomplet et cohérent

s3 est **incomplet** et **cohérent** dans Segment. L'origine n'est pas instanciée, l'attribut calculable n'est pas calculé. La contrainte n'est pas évaluable.

5. LES FONCTIONS

Nous avons vu que la définition d'un type peut mettre en jeu un certain nombre de liens et de contraintes d'intégrité.

L'expression d'une contrainte ou de certains liens, contient une fonction qui sera exécutée lors de l'évaluation du lien ou de la contrainte à l'occasion de la création, ou de la modification d'un objet . De plus, une fonction peut être utilisée pour le calcul d'objet (voir la création d'objet dans le chapitre 6).

Une fonction peut être, soit prédéfinie, (par exemple, appartenance d'un objet à un type), soit définie par le concepteur lui-même.

Une fonction définie par le concepteur est réalisée par un programme utilisateur. Dans les domaine de conception, il existe de très nombreux programmes écrits en FORTRAN, PASCAL, C..... L'idée est de réutiliser ces programmes en n'écrivant que l'interface, car il n'est pas question de réécrire des simulateurs électriques de quelques milliers de lignes FORTRAN. Les fonctions permettent de faire la liaison entre les objets de la base et le monde extérieur.

Les fonctions sont décrites par l'usager, et sont utilisées dans l'expression des liens et des contraintes. Une fonction est décrite par :

- le programme qui la réalise,
- sa signature $A_1 * ... * A_n \rightarrow A_p$, où $A_1 * ... * A_n$ désigne le produit des types des objets en entrée de la fonction et A_p le type de l'objet en sortie.

Par exemple : soit *transptv* la fonction qui calcule la translation verticale d'un point, par un certain coefficient :

transptv (transptv) : Point * Entier --> Point

Cette fonction est réalisée par le programme *transptv*, qui admet un *Point* et un *Entier* en entrée, et qui calcule un point.

6. SPECIALISATION

Les méthodes de travail généralement utilisées dans les environnements CAO, consistent en la description de plus en plus précise et concrète de l'objet à concevoir. En d'autres termes, le type de l'objet à concevoir est défini selon ses grandes lignes, et spécialisé tout au long du processus de conception. Définir un sous type revient à définir une hiérarchisation et une spécialisation dans les types. Un tel mécanisme permet d'affiner successivement la définition d'un type dans le but d'obtenir la structure finale désirée. Cet affinage peut s'obtenir soit en rajoutant des **contraintes de spécialisation**, soit en rajoutant des **liens de spécialisation.** Le concepteur peut décider aussi d'enrichir la structure en rajoutant des **attributs supplémentaires** (figures 8 et 9).

```
Cell = ( [ type : Typerep, env : Enveloppe,
           comp : LIST Composant, équi : LIST Equipot],
           [],
           { NOT intersect (comp, équi) })
```

figure 8 : définition du type Cell

Un objet de type Cell, est décrit par son enveloppe (env), ses composants (comp), et ses équipotentielles (équi). On exprime qu'un tel objet est cohérent si ses composants et ses équipotentielles ne s'entrecoupent pas (NOT intersect (comp,équi)). Le concepteur peut affiner sa notion de Cell avec d'une part le sous type Cellbase, pour décrire les circuits de base (sans composant ni équipotentielle) et les circuits construits (qui sont les autres).

```
Cellbase ISA Cell = ([], [], {égal (card (comp),0), égal (card(équi),0)})
Cellconstr ISA Cell = ([], [],{ NOT égal (card (comp),0),
                                NOT égal (card(équi),0)})
```

figure 9 : définition des sous types Cellbase et Cellconstr

Un sous type hérite de la définition complète des types de plus haut niveau. Dans la figure 9, le type Cellbase hérite des attributs, des liens et des contraintes de Cell. Dans la description des sous types Cellbase et Cellconstr, on n'a donné aucun attribut supplémentaire, ni lien de spécialisation. L'affinage n'est fait que grâce aux contraintes de spécialisation. Si le concepteur définit un autre sous type de Cellbase, il hérite alors de la définition complète de Cell, puis de celle de Cellbase. Le mécanisme **dynamique** d'héritage des occurrences dans une arborescence d'ensembles, définie par la spécialisation est décrit dans la section 7 de ce chapitre.

De façon générale, un sous type est décrit par :

ST **ISA** *T*, où *ST* est décrit par (A', L', C'), et *T* par (A, L, C)

avec :

$A' = A + [attr'_1 : A_1,...,attr'_p : A_p]$, où tous les attributs de A' sont différents (tous les attributs supplémentaires sont différents entre eux, et différents de ceux dans A)

$C' = \{ C_1,...,C_k \} \cup C$;

$L' = L + [L_1,..L_q]$, où les L_q ne mettent pas en jeu les attributs liés par les liens de L.

Le type (A', L', C') hérite des attributs des liens et des contraintes de (A, L, C).

Ce mécanisme de sous typage est très proche de celui trouvé dans les modèles orientés objets. Mais il faut noter que le choix a été fait de se limiter à un héritage simple. En effet, contrairement à certains modèles [DDE88], [LRV88], un sous type, ne spécialise qu'un seul type donné.

Nous avons choisi de ne proposer qu'un héritage simple car le principe d'héritage tel qu'il est utilisé en CAO correspond à l'idée d'affinage, ou de progression dans le processus de conception. Cette idée est en quelque sorte trahie, si on accepte de construire un sous type de deux types dont les idées de base sont différentes (voir figure 10). Dans cette

logique, l'héritage multiple n'est acceptable que lorsqu'il est utilisé comme une simple économie de spécification des attributs d'un nouveau type.

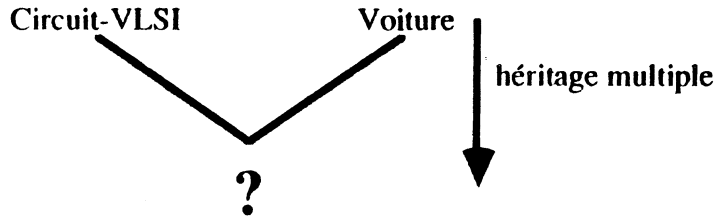


figure 10 : spécialisation de deux types

L'exemple choisi est quelque peu caricatural, mais pose le problème du sens à donner à une telle spécialisation.

La donnée de nouveaux attributs ou liens dans le sous-type ne doivent pas surcharger ceux décrits dans le type. La surcharge de contrainte est impossible, car une contrainte réfère toujours la même fonction, réalisée par un seul algorithme. Cette définition est externe au type, et commune à tous les types qui la réfèrent.

D'autre part, le mécanisme de spécialisation permet de résoudre le problème des énoncés récursifs communs en CAO (problèmes liés la gestion de nomenclature par exemple). Cette possibilité est intéressante dans la CAO de circuits VLSI, où la récursivité est souvent utilisée (par exemple, une cellule est composée d'autres cellules).

En effet, notre notion de type (section 2) impose une définition ascendante des types. Par exemple, Cell ne pourra être défini, que si les types Enveloppe, Composant, Typerep, et Equipot sont eux-mêmes définis. Il est donc impossible de définir un type T, comportant des attributs prenant leurs valeurs dans T.

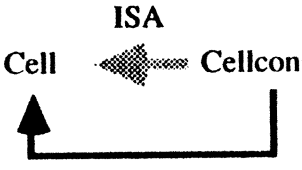
Un tel énoncé récursif peut être obtenu malgré tout par le biais du constructeur ISA :

- on définit un type *T1* de haut niveau.
- on définit un ensemble *T2* **ISA** *T1*, contenant des attributs supplémentaires prenant leurs valeurs dans *T1* (qui existe forcément).

Exemple :

qui exprime qu'une cellule est constituée de cellules

Ce type d'expression est interdit, mais il est possible de faire :



toute Cellule construite (Cellconstr) est une cellule (Cell)

7. HERITAGE

Le mécanisme dont nous décrivons les principes dans cette section, est mis en œuvre afin de typer les objets le plus finement possible. L'idée est de permettre au concepteur de connaître à tous moments les objets d'un sous type donné, l'exemple suivant illustre ce mécanisme. Nous reprenons le type Cell défini dans la section 6, figure 8.

```
Cell = ( [ type : Typerep, env : Enveloppe,
           comp : LIST Composant, équi : LIST Equipot], [],
           { NOT intersect (comp, équi) } )
```

figure 11 : définition de Cell

Les sous types Cellbase et Cellconstr spécialisant Cell sont rappelés dans la figure suivante (figure 11) :

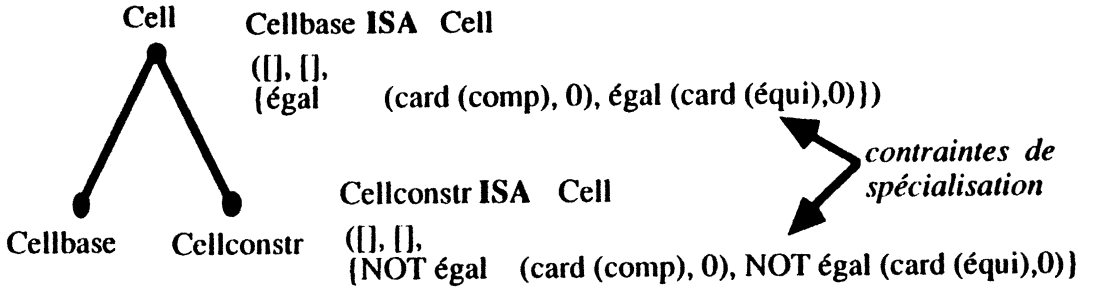


figure 12 : la spécialisation

Nous pouvons spécialiser le type des cellules construites, en définissant le sous type Cell-logique (ISA de Cellconstr). Nous obtenons la hiérarchie de sous-types suivante :

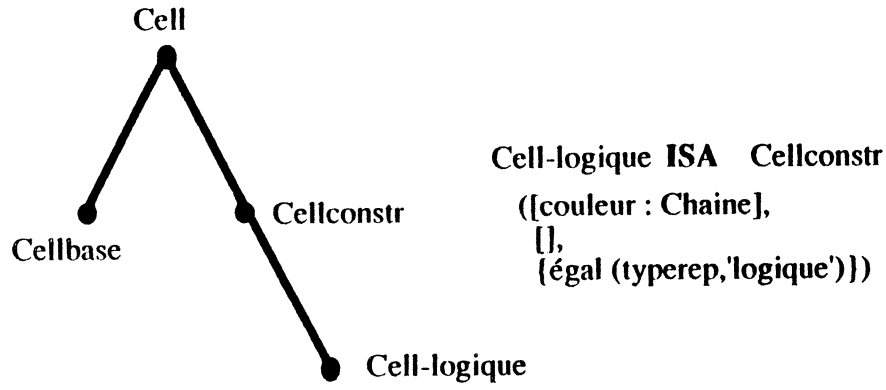


figure 13 : spécialisation de Cellconstr

La spécialisation est faite ici en donnant une contrainte de spécialisation *égal (type,'logique')* et un attribut supplémentaire *couleur : Chaîne* . Le type Cell-logique, regroupe ainsi tous les objets qui sont des cellules construites (de Cellconstr), dont le type est logique.

Le mécanisme d'héritage s'applique sur tous les objets, créés dans un sous type, ou dans un type construit spécialisé par un ou plusieurs sous types.

Si un objet créé dans Cell, ou dans Cellconstr, est cohérent (ne viole aucune contrainte définie dans son type), alors l'héritage est tenté dans le sous type. L'héritage est autorisé dans un sous type, si toutes les contraintes de spécialisation définies sur ses attributs sont vérifiées. C'est à dire, si l'objet est cohérent dans le sous type. Si l'héritage est autorisé dans un type, il est tenté récursivement dans ses sous types. La figure 14 illustre ce cas appliqué à la hiérarchie de types décrites dans la figure 13.

Les liens et les contraintes définis dans un sous type se comportent donc en règles de **cohérence forte** lors de l'héritage des objets. Par opposition, les contraintes définies dans un type se comportent, lors de la création d'une occurrence dans celui-ci, en règles de **cohérence faible.**

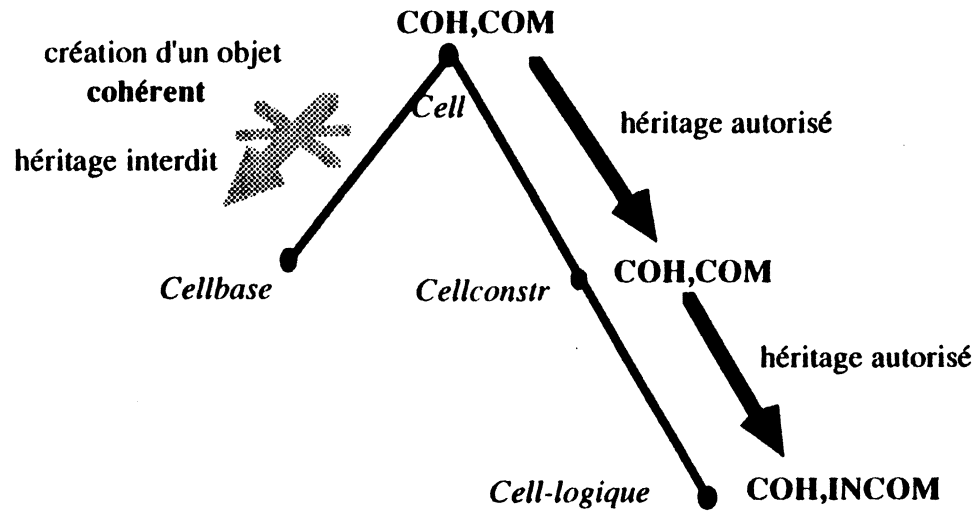


figure 14 : mécanisme d'héritage

Prenons l'exemple d'un objet créé dans Cell, et admettant au moins un composant (attribut comp de Cell), et au moins deux équipotentielles (attribut équi de Cell). L'objet est complet et cohérent, il est hérité dans Cellconstr, car il y est cohérent (les deux contraintes de spécialisation ne sont pas évaluées à faux). Il est de plus complet dans Cellconstr (Cellconstr n'admet pas d'attribut supplémentaire). Il est de même hérité dans Cell-logique où il est cohérent et incomplet : la contrainte de spécialisation est vérifiée, et l'attribut supplémentaire *couleur* n'est pas instancié.

L'objet hérité dans Cell-logique possède un attribut *couleur* qui lui a été rajouté au moment de l'héritage. Cet attribut va resté inconnu (l'objet incomplet) jusqu'à ce que le concepteur modifie l'objet, et instancie l'attribut *couleur*.

La figure 15 suivante, illustre le mécanisme d'héritage mis en jeu lors de la création d'un objet dans un sous type ISA. Le système tente d'une part l'héritage dans les sous types fils (ici Cell-logique), puis il effectue d'autre part l'insertion de l'objet dans le type père.

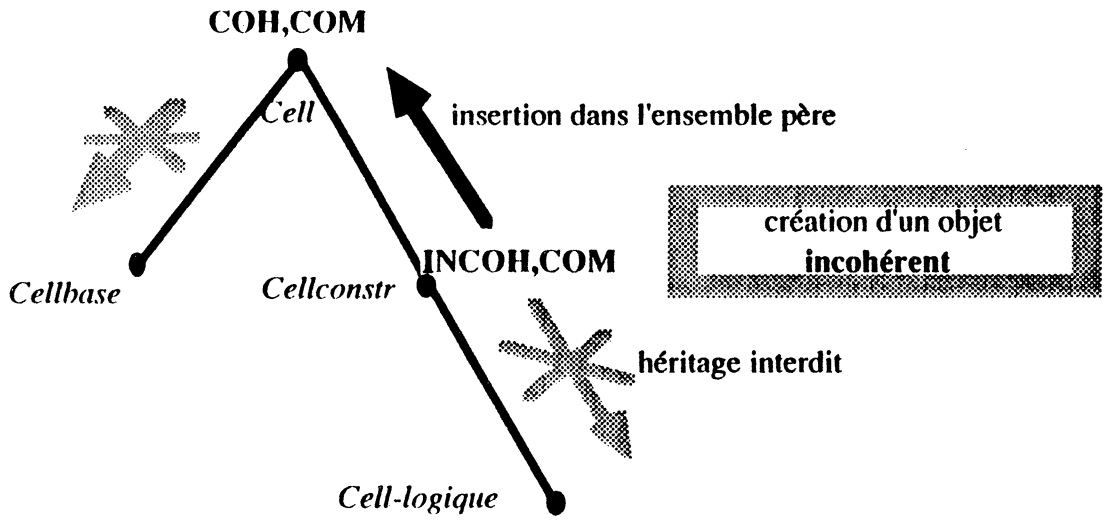


figure 15 : héritage et insertion d'un objet

Supposons par exemple que l'objet soit créé dans Cellconstr et qu'il viole la contrainte *NOT égal (card (compo),0)*. L'objet n'est donc pas hérité dans Cell-logique. Il est inséré dans Cell, où il complet et cohérent.

L'héritage est tenté à partir de Cell; si Cell était un sous-type, l'insertion serait exécutée récursivement sur son sur-type.

8. CONCLUSION

Les concepts que nous avons présentés ici, constituent un sous ensemble de ceux du modèle défini dans [RIE85], point de départ de notre étude. Dans ce chapitre nous avons proposé une formalisation de ce modèle dont les caractéristiques sont très proches de celles des modèles orientés objets.

Partant de ce modèle, nous avons tout d'abord réalisé un serveur d'objets [FAR87], [FAR87b], qui permet la définition des types et des sous types, la création et la manipulation des objets, ainsi que l'héritage présenté dans la section 6. Ce serveur est décrit au chapitre 6.

Les trois sections suivantes montrent l'apport de ce système quant aux problèmes posés par la CAO.

8.1. Dépendances Inter-Objets

En plus des règles de cohérence internes à un objet (les contraintes d'intégrité), le serveur permet de définir des **liens** inter-objets. Le concepteur peut exprimer qu'un objet est construit à partir d'un autre, et qu'il évolue avec lui. L'expression de ces liens, et leur gestion dynamique est assurée. On peut ainsi exprimer des connaissances générales sur le domaine, mais aussi sur un objet particulier.

8.2. Dynamicité

Les objets CAO sont dynamiques en valeurs et en structure. Rien n'est jamais figé. Le serveur offre des mécanismes d'extension de la structure des objets. Il prend à sa charge, la répercussion de ces modifications, et assure la cohérence des schémas en signalant les objets devenus incohérents par ces modifications. Dans certains cas, la répercussion de modification peut ne pas être souhaitée. Le concepteur peut le signaler au système : c'est le principe de **rupture de lien**. Au moment de leur conception, on lie deux objets pour des raisons de commodité par exemple, puis on décide de leur donner une évolution propre. La dynamicité doit être permise, mais ses conséquences (les répercussions de mise à jour) doivent être contrôlées.

La modification de type n'est pas implantée. Elle pose en effet des problèmes non triviaux de maintien de la cohérence [BAN87]. Une étude portant sur l'évolution des schémas sera effectuée plus tard.

8.3. Incomplétude et Incohérence

Un objet CAO, le plus souvent, n'est jamais complet avant d'être terminé. Cela signifie tout simplement qu'un objet CAO est, la plupart du temps **incomplet**. Il faut noter de plus, que les contraintes d'intégrité définies par le concepteur peuvent à chaque instant être violées, sans pour autant bloquer la conception. Pour permettre et gérer de tels états de l'objet CAO, le serveur distingue les différents cas suivants :

- complet et incohérent :

L'objet est terminé, mais une contrainte d'intégrité au moins est violée. Il faut revenir à la version précédente de l'objet, et recommencer. Faire et défaire, c'est toujours travailler ...

- incomplet et incohérent :

On est en cours de conception, et on s'aperçoit qu'il y a au moins une règle que l'objet ne respecte pas, il faut revenir à la version précédente de l'objet.

- incomplet et cohérent :

L'objet n'est pas terminé, et il ne viole aucune règle de cohérence. Il est possible que ce soit parce qu'on ne peut pas évaluer une contrainte, du fait de l'incomplétude. En effet, si une contrainte ne peut pas être évaluée parce qu'il manque de l'information, ETIC considère que l'objet est cohérent (on laisse le bénéfice du doute ..).

- complet et cohérent :

L'objet est terminé, et il respecte les règles de cohérence qui ont été définies par le concepteur. Tout est parfait ...

Cet état est celui vers lequel le concepteur essaie de tendre pendant tout le processus de conception.

Dans les chapitres suivants nous proposons d'étendre un modèle d'objets CAO, afin de traiter :

1) l'activité coopérative des concepteurs au sein d'un projet de CAO (chapitre 3).

2) la gestion des versions, nécessaires dans les environnements de conception (chapitre 4).

3) la gestion des transactions (chapitre 5), qui se trouve simplifiée grâce aux concepts et aux mécanismes introduits dans les chapitres précédents.

Enfin, le chapitre 6 décrit le prototype **ETIC**, qui intègre le serveur d'objets présenté ici, et qui réalise les extensions prenant en compte les trois points précédents.

CHAPITRE 3

UN SGBD DANS UN ENVIRONNEMENT PARTAGE

table des matières

1. INTRODUCTION

Nous avons vu dans le chapitre introductif, que les solutions proposées généralement pour la gestion des objets partagés ne sont pas satisfaisantes car le plus souvent basées sur le concept de versions [BKK85], [KAL82], [KLM84], [LOR83]. Les solutions ne proposent en fait rien pour la coopération, mais résolvent le problème des accès concurrents à un objet. Le principe généralement admis dans les systèmes CAO, est que l'on donne une version (une copie) de l'objet à chaque concepteur. Puis les concepteurs effectuent dessus des modifications. Lorsque les versions sont terminées (complètes) le système les stabilise (on dit aussi gèle) et les archive. La difficulté réside dans la fusion des différentes versions en un seul objet : l'objet conçu. Aucune solution ne ressort explicitement de ces différentes études.

Nous définissons ici les concepts nécessaires à la gestion d'un projet de CAO conformément à l'approche adoptée dans la section 4 du chapitre 1.

La section 2, introduit les concepts de **vue** et de **projet**, et montre comment la coopération et le partage des objets et pris en compte dans un projet de conception. La section 3 décrit en détail les différentes fonctionnalités nécessaires dans un tel environnement, c'est à dire les protocoles d'échanges entre les différents systèmes. Enfin dans la section 4, nous proposons en conclusion une analyse critique de nos propositions.

2. LE PROJET

Nous définissons dans cette section, lès concepts nécessaires à la définition et au fonctionnement d'un projet de conception. La section 2.1 présente ces nouveaux concepts. Dans les sections 2.3 et 2.4, nous montrons leurs apports dans l'activité des équipes du projet et lors de la synthèse des travaux des équipes.

La description du projet en termes d'équipes, puis celle des équipes en termes d'équipes plus petites, et finalement en termes de concepteurs, tire parti de la structure hiérarchique de l'objet à concevoir, tel que nous l'avons défini dans le chapitre 2. Cette organisation est de plus

classique dans le cadre d'une entreprise : elle est formée de départements, chacun composé d'équipes, etc ..

Une équipe doit instancier une partie de l'arbre décrivant le type de l'objet à concevoir (figure 1). Il peut s'agir d'un ou de plusieurs sous-arbres.

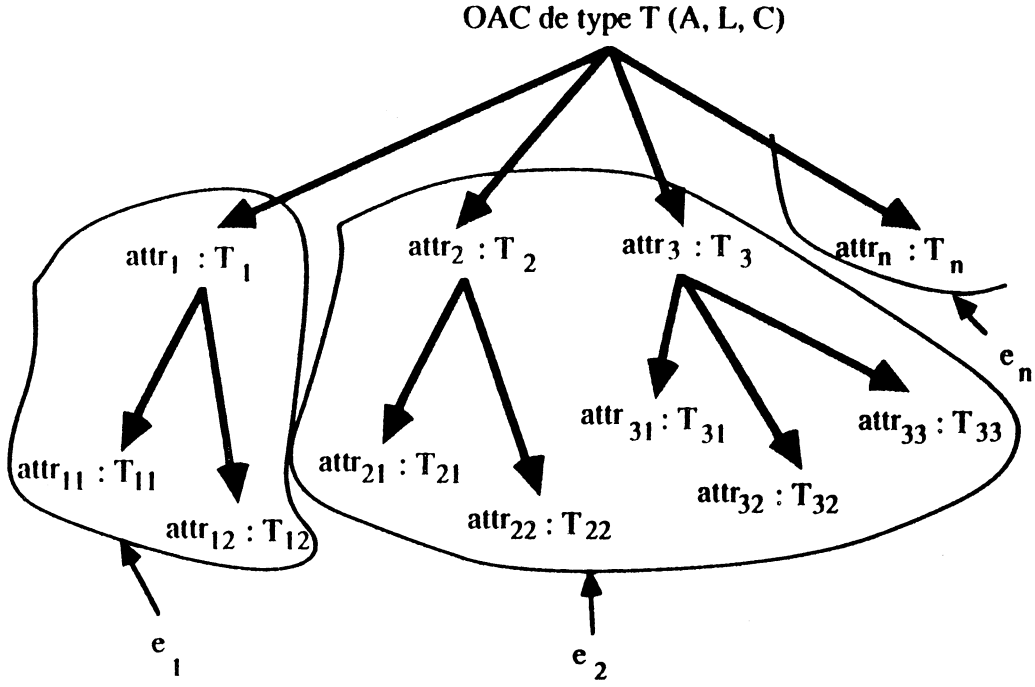


figure 1 : le partage de l'objet à concevoir

Les mécanismes qui permettent d'exprimer la répartition du travail de conception à l'intérieur du projet s'appliquent à l'objet à concevoir tel qu'il est décrit dans le chapitre 2. Ils permettent de sélectionner dans l'arbre (figure 1) un chemin de la racine à un sous arbre.

La description du partage de la tâche pour la conception de OAC de type T (voir figure 1) est la suivante :

1 : le projet est constitué de p équipes dont e_1 et e_2. Ce choix peut se justifier par la structure de l'objet à concevoir, ou par les affinités des gens, etc ..

2 : le travail des équipes e_1 et e_2 est exprimé par :

- e_1 : $attr_1.T$
- e_2 : $attr_2.T$, $attr_3.T$

Où les $attr_i$ sont de type T_i, décrits par les triplets (A_i, L_i, C_i). En d'autres termes cela signifie que e_1 doit concevoir un objet OAC_1 dans T_1, et e_2 deux objets, OAC_2 dans T_2 et OAC_3 dans T_3.

Remarque : les T_i ne sont pas tous distincts.

3 : pour chaque équipe e_1, e_2,.., e_p on décrit selon le même principe la partage de la conception des sous objets (on ne donne que l'exemple pour l'équipe e_2) :

e_2 est structurée en deux sous équipes e_{21} et e_{22}, qui ont pour tâches :

- e_{21} : *.T_2, $attr_{31}.T_3$ (* signifie tous les attributs)
- e_{22} : $attr_{32}.T_3$, $attr_{33}.T_3$

Cette répartition est illustrée par la figure 2 :

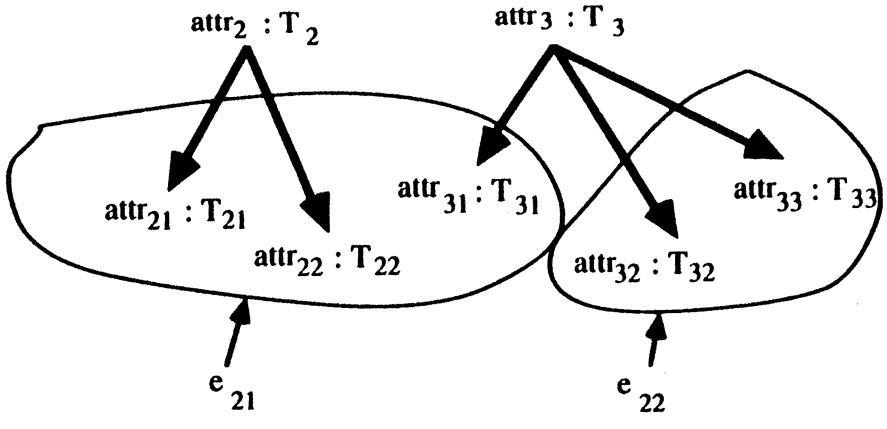


figure 2 : le partage de la tâche dans e_2

Ces expressions sont des désignations hiérarchiques à travers les attributs. Aucune expression de sélection n'est donnée, car à l'initialisation de la conception, aucun attribut de l'objet à concevoir n'est instancié. Dans le cas où un objet est donné comme point de départ de la conception, les expressions précédentes sont suffixées par le nom de cet objet. Le résultat est alors un objet, c'est à dire un type avec certains attributs instanciés (ou tous). Dans ce cas le travail de conception consiste en la modification de cet objet, pour l'améliorer, changer certaines de ses performances ou caractéristiques, etc ...

Globalement, le travail est terminé lorsque tous les attributs A de T sont connus : OAC_1, OAC_2 et OAC_3 sont terminés, on les utilise pour instancier les attributs de T, tous les L sont évalués, et toutes les contraintes C sont évaluées à vrai. C'est à dire lorsque OAC (l'Objet à Concevoir) est **complet** et **cohérent.**

En d'autres termes, la conception est terminée lorsque tous les sous objets composant l'objet à concevoir sont eux-mêmes complets et cohérents c'est à dire, lorsque chaque équipe a terminé.

Les **liens** et les **contraintes** sont évalués lorsque les attributs sur lesquels ils portent sont connus. Par exemple dès que les sous objets OAC_2 (de type T_2) et OAC_3 (de type T_3) sont terminés, il est possible d'évaluer les liens et les contraintes définis dans OAC et ne concernant que OAC_2 et OAC_3. Ces liens et ces contraintes s'ils existent sont dits **locaux** à l'équipe e_2 et sont évalués dans l'environnement de e_2. Ceci permet d'évaluer au plus tôt les contraintes et les liens, et de modifier sans attendre, les sous objets en cas d'incohérence.

2.1. Définitions

1 : On appelle **VUE** l'ensemble des opérations qui décrivent la part de travail affectée à une équipe ou un concepteur. Une vue est liée à une équipe, dont elle décrit l'activité. Une vue regroupe donc un ensemble de projections sur un ensemble d'attributs d'un ou de plusieurs types, le résultat est alors un ensemble de types. Dans le cas où une sélection est exprimée, le résultat délivré est un ensemble d'objets. A l'initialisation de la conception, aucun attribut n'est instancié, une sélection n'est donc pas nécessaire, mais lorsque des objets pré-existent et que l'objet à concevoir va être élaboré à partir d'eux, alors une sélection devient indispensable.

2 : Le **PROJET** est un ensemble d'informations qui donnent pour chaque équipe sa composition et la **VUE** qui lui est affectée, et ceci récursivement jusqu'à obtenir la description des équipes en termes de concepteurs. Ces informations décrivent un ensemble de droits associés à chaque composant du projet. Ces droits concernent les commandes autorisées à chacun des composants.

Le concept de vue, adopté ici est proche de la notion de vue matérialisée [BLT86]. Résoudre le problème de la confidentialité et du contrôle d'accès aux données, est un point commun entre de nombreuses études [ADI81], [ADL80], [DEA82], [DEN87], [BEH88] portant sur les vues matérialisées ou non. Une vue décrit en effet, un environnement de travail pour un concepteur ou une équipe et limite leurs droits à cet environnement.

L'expression d'une vue est pour nous plus limitée que dans celles décrites dans ces différentes études. En effet, nous n'offrons pas la possibilité d'exprimer une opération équivalente à celle du produit de l'algèbre relationnelle. Cela ne correspond à aucun besoin des applications que nous visons. De plus, permettre une telle opération pose le problème de la mise à jour de la vue [SLW88], [BLT86].

La modification de l'expression d'une vue implique dans notre environnement la modification des types des objets manipulés dans les sous systèmes. La modification de type ne sera pas traitée dans cette thèse, mais fera l'objet d'une étude ultérieure.

Le problème des vues en CAO, est posé dans [DAV81], [FAU87], [APA87], [AUT88]. L'objectif visé est la réduction du volume de données manipulées par un concepteur, et la définition d'une base de données pour ce concepteur c'est à dire son environnement de travail, afin de résoudre dans une certaine mesure les problèmes d'accès concurrents aux données. Le problème de la coopération dans une activité de conception est abordé dans [FAU87a], [FAU88], études qui proposent une solution que nous développons dans ce chapitre.

2.2. Exemple

Plaçons nous dans le domaine de la CAO de VLSI. Le projet à réaliser est le circuit dont le dessin logique est donné par la figure 3.

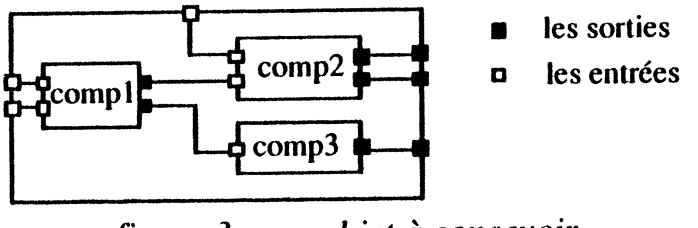


figure 3 : un objet à concevoir

Le but du projet consiste à concevoir la représentation en dessins de masque du circuit pour en permettre la réalisation. Classiquement, en CAO de VLSI la conception d'un circuit passe par les différentes étapes suivantes :

- la représentation **fonctionnelle** du circuit, on peut dire aussi le cahier des charges du circuit (cette représentation fonctionnelle a permis de déduire la représentation logique donnée par la figure 3).

- la représentation **logique** : on décrit le circuit en terme de composants (qui sont eux-mêmes des circuits) et de portes logiques.

- la représentation **électrique** : on transforme la représentation précédente en une représentation électrique (c'est à dire en termes de transistors).

- la représentation **implantée** qui donne le dessin des masques (les zones de silicium, l'aluminium etc ..) et qui permettra la réalisation physique du circuit.

Le passage d'une de ces étapes à la suivante est linéaire (dans certains domaines de la CAO, cette linéarité n'est pas systématique). Il existe des outils pour déduire automatiquement la représentation électrique de la représentation logique.

Le chef de projet doit définir de façon générale, la structure de l'objet à concevoir, car c'est sur cette structure que seront appliquées les vues. Dans notre exemple, le chef de projet décrit le circuit à concevoir en terme de trois composants, $comp_1$, $comp_2$ et $comp_3$ pour chacune des représentations. Il définit les spécifications de ce circuit :

- la fonction réalisée,
- le nombre de ses entrées (ici 3) et le nombre de ses sorties (3),
- ses propriétés techniques sont connues et fixées (sous forme de contraintes par exemple). Ce sont par exemple, le non-recouvrement de ses composants, la connectique des composants, des entrées et des sorties.

Les trois composants du circuit sont de même type. Ceci est une propriété de ce type d'applications. Un circuit qu'il soit logique, électrique ou implanté est toujours décrit en termes de composants, d'équipotentielles, d'une enveloppe, d'entrées et de sorties. La seule différence est donnée par les objets de base utilisés. Pour un circuit logique, on utilisera des portes logiques, pour un circuit électrique des transistors.

La description de ce circuit dans le formalisme donné dans la section 2 de ce chapitre est (figure 4) :

```
(Circuit , Lcircuit, ∅)
    avec Circuit = [rep-f : Rep-Circuit,    * représentation fonctionnelle*
    rep-l : Rep-Circuit,                    * représentation logique       *
    rep-e : Rep-Circuit,                    * représentation électrique    *
    rep-i : Rep-Circuit ]                   * représentation implantée     *
Lcircuit      = [(rep-e,déduction(rep-l)]

(Rep-Circuit, ∅, CRep-Circuit)
    avec Rep-Circuit =
            [entrées : List Ent,            * interface du circuit         *
            sorties : List Sort,            *                              *
            enveloppe : Env                 *                              *
            équipot : List Equi,            * composition du circuit       *
            comp1 : Composant,              *                              *
            comp2 : Composant,              *                              *
            comp3 : Composant ]             *                              *

CRep-Circuit = {NOT intersecte (équipot),
            NOT recouvre (comp1,comp2,comp3)}
```

figure 4 : définition du type Circuit

Les types Sort, Ent, Equi, et Composant sont décrits de la même façon. Le constructeur List permet de construire une liste d'objets d'un type donné pour instancier un attribut.

Le chef de projet décide (avec ou sans l'accord des concepteurs !) de la répartition de la conception de l'objet (voir figure 5).

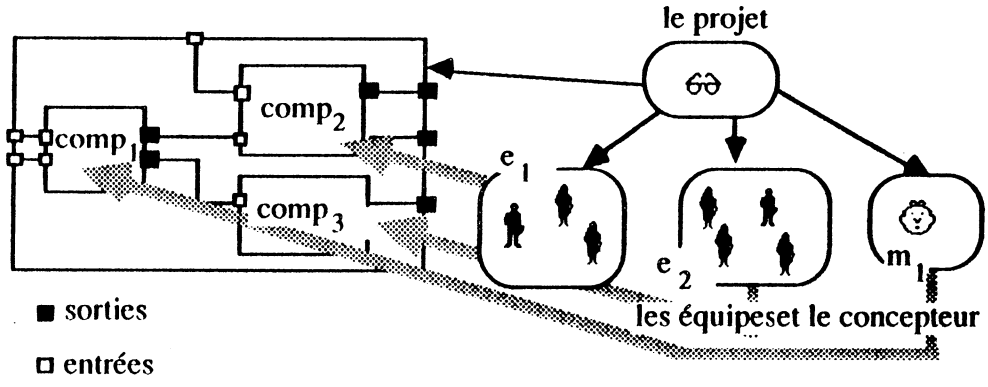


figure 5 : la répartition du travail dans le projet

L'activité du projet est décrite par la figure 6 :

- E = [e1, e2, m1], l'ensemble des équipes du projet
- v(m1) = [comp1.rep-f, comp1.rep-l, comp1.rep-i]
- v(e1) = [comp2.rep-f, comp2.rep-l, comp2.rep-i]
- v(e2) = [comp3.rep-f, comp3.rep-l, comp3.rep-i]

figure 6 : répartition de la tâche dans le projet

Le lien de $L_{circuit}$, nécessite la connaissance de la représentation logique complète du circuit, il est pour cette raison global au projet. Quant à la contrainte *NOT recouvre* dans $C_{rep\text{-}circuit}$, elle nécessite pour chaque représentation la connaissance des trois composants, elle est donc globale au projet.

Les attributs *entrées*, *sorties*, *équipot*, *enveloppe* n'ont pas été affectés. Ils devront être instanciés par ailleurs, par exemple par un autre équipe non encore définie. Par suite, la contrainte *NOT intersecte* qui porte sur *équipot*, est globale au projet.

Le même processus est réitéré pour les équipes e_1 et e_2, puis récursivement jusqu'à obtenir une description en termes de concepteurs.

2.3. Le travail dans les équipes

La tâche de chaque équipe ou concepteur est maintenant définie. Elle s'exprime par l'affectation d'une vue, qui se matérialise par un type d'objet, et un objet à concevoir dans ce type. Le but recherché est la conception d'un objet **complet** et **cohérent** dans ce type. Pour cela, les

membres du projet dispose de tous les outils nécessaires : les programmes, et les diverses commandes de création de types et de sous types, et de manipulation d'objets.

L'activité de conception étant un travail coopératif, les échanges entre les concepteurs d'une équipe doivent être souples (ou entre plusieurs équipes), car ils sont le plus souvent informels et concernent des **objets incomplets et (ou) incohérents.**

Dans l'exemple précédent (§ 2.1) les équipes 1 et 2 peuvent tenter l'**assemblage** de leurs vues respectives, pour évaluer la qualité du travail réalisé jusqu'à maintenant, même si ceux-ci ne sont pas terminés. On veut s'assurer par exemple, qu'à un moment donné on ne viole aucune contrainte définie globalement. Certaines contraintes globales peuvent être évaluables, il est intéressant alors de savoir si elles sont évaluées à vrai. Si oui les équipes ne changerons sans doute rien à ce qui est fait, sinon elles devront reprendre le travail effectué et le modifier. Rappelons, qu'une contrainte n'est violée, que lorsqu'elle a été évaluée à faux. Le fait qu'il existe des contraintes non-évaluables, n'induit pas que l'objet correspondant est incohérent (contrairement à un principe assez général qui consiste à considérer un objet non terminé comme incohérent). Cet **assemblage** est en fait une suite d'opérations réalisées par le système au niveau supérieur (dans notre exemple : le projet) et qui sont : l'instanciation des attributs du type de l'objet à concevoir (Circuit) et l'évaluation des liens et des contraintes globales définis au niveau supérieur. Ces différentes opérations sont **automatiquement** exécutées par le système au moment de l'assemblage, et délivrent un objet dont la durée de vie est temporaire; elles sont déduites par le système pour chaque vue décrite au niveau supérieur.

2.4. Lorsque chaque équipe a terminé

A la fin de la conception, lorsque les équipes ont terminé leur portion de travail, c'est à dire lorsque les composants dont elles étaient chargées sont **cohérents** et **complets**, et s'ils vérifient les contraintes **locales**, qui sont alors toutes évaluables (les liens **locaux** sont préalablement calculés), il reste à assembler les différentes vues, et à effectuer leur connexion dans le respect des contraintes définies **globalement** au niveau du projet. Cette opération est assurée par la fonction de **remise** dont les fonctionnalités sont celles de l'**assemblage.** Une différence importante est à noter : si le résultat de cette remise délivre une

incohérence au niveau supérieur, elle est **annulée**. La vue correspondante est alors renvoyée au concepteur ou équipe propriétaire, pour être modifiée.

Dans le cas où une contrainte est violée lors de la remise d'une portion de travail, le système décide celle(s) qu'il faut retirer. Si, dans notre exemple, $comp_1$ est terminé lorsque $comp_2$ est fini à son tour, et si $comp_2$ met en évidence une incohérence, de part sa connexion avec $comp_1$, il faut déterminer quel composant $comp_1$ ou $comp_2$ doit être modifié. La partie à modifier est rendue à l'équipe chargée de sa conception. Ce retrait obéit à une **stratégie**, choisie par le chef de projet parmi celles proposées par le système (par exemple, retrait de la dernière partie remise, affectation d'une pondération à chaque partie, et retrait de celle de poids le plus faible parmi celles mises en cause par l'incohérence, etc ...).

Dans certains domaines et selon la répartition choisie, le chef de projet veut pouvoir exprimer un **ordonnancement** dans la résolution des tâches. Par exemple, en CAO-VLSI, si la répartition s'est faite sur la base des représentations (fonctionnelle, logique, électrique, ..), la représentation logique doit être remise avant la représentation électrique. Le système permet l'expression de cet ordonnancement décidé par le chef de projet.

La stratégie de retrait et l'ordonnancement des vues sont développés dans la section suivante.

3. LES FONCTIONNALITES

Nous présentons dans cette section l'ensemble des fonctionnalités nécessaires au système de gestion de projet proposé dans les sections précédentes. Ces fonctionnalités permettent la définition du projet (§ 3.1.2) et des équipes qui le composent ainsi que la description des vues (§ 3.1.1) qui leur sont associées. L'architecture proposée dans la section 4 du chapitre introductif est adoptée ici. Nous décrivons dans cette section les protocoles de communication entre les différents sous-systèmes composant le projet (§ 3.2).

3.1. Le Système Public

Le système public gère les bases de données partagées par l'ensemble des équipes du projet : la base de connaissances et la base du projet. Les objets stockés dans la base de connaissances sont des objets

stables, complets et cohérents. Seul le chef de projet est habilité à détruire ou créer des objets dans cette base. Les autres concepteurs n'ont qu'un droit de lecture. Les objets gérés par cette base sont les connaissances acquises au cours de la vie de l'entreprise et nécessaires au déroulement du projet, ainsi que les objets conçus au cours du projet, mais qui ont atteint un état stable.

La base de projet, contient tous les objets instables. Il s'agit des objets échangés entre les équipes des sous systèmes fils. Ceux-ci sont cohérents dans cette base bien que le plus souvent incomplets. Ces objets peuvent être incomplets, voire même incohérents. Cette base contient de plus les objets en cours de conception. Ce sont les objets dont la conception a été distribuée entre les équipes propriétaires des systèmes fils par le mécanisme de vues. Un tel objet est incomplet, car toutes les vues définies sur celui-ci n'ont pas été remises, il est cohérent car la remise d'une vue garantit la cohérence (voir § 3.2.1.2).

C'est au niveau du système public, que le chef de projet, décrit la structure de l'objet à concevoir, celle du projet, et l'organisation, le fonctionnement et les droits des équipes.

3.1.1. La définition des vues

La définition des vues est effectuée dans le système public du projet. Grâce à cela, le système stocke et déduit de l'information qui lui sera nécessaire lors de la génération des sous-systèmes à l'initialisation de la conception, et lors des processus d'assemblage et de remise de vue évoqués dans les sections 3.2.

Il est possible de considérer une vue comme un objet d'un type prédéfini sans lien ni contrainte :

Vue ([type : Chaîne, opérations : LIST Projection])

Une vue est identifiée par son nom, elle s'applique sur un certain type (donné ici par le nom du type, c'est à dire une chaîne de caractères). Elle est réalisée par une liste d'opérations de projection.

Un objet de ce type vue est donné dans l'exemple suivant :

v1 ([type : Circuit, opérations : [comp1.rep-f, comp1.rep-l, comp1.rep-e]])

Le nom de la vue est un identificateur unique pour le projet. Les opérations de projection sur un type (par exemple Circuit, décrit dans la

section 2.2 de ce chapitre) sont contrôlées par le système. A partir de l'information ainsi donnée, le système déduit les contraintes et les liens globaux : qui sont dans notre exemple les contraintes *NOT intersecte(équipot)*, et *NOT recouvre (comp1, comp2, comp3)* et le lien *rep-e:=déduction(rep-l)*. En effet, ce lien et ces contraintes admettent comme paramètres des attributs, qui ne sont pas entièrement référencés par la vue. Par suite ils ne peuvent pas être évalués avec les seuls attributs de la vue. Ces contraintes et ce lien ne seront évaluables que lorsqu'on en connaîtra tous les paramètres, qui sont ici pris en compte dans plusieurs vues, et ils sont pour cette raison globaux à cet ensemble de vues.

3.1.2. La définition du projet

La première phase de la conception consiste en la définition du projet et de son organisation. Le système offre pour cela un environnement de définition de projet.

Le système oblige le chef de projet à définir l'ensemble des équipes niveau par niveau. Puis pour chacune des équipes ainsi définies, le système en demandera la composition et l'organisation (le processus de définition se fait en largeur d'abord dans la hiérarchie).

Ce mode de définition permet à l'utilisateur de garder une idée synthétique et claire de l'équipe sur laquelle il travaille. La figure 7 montre l'enchaînement des opérations dans cet environnement, pour un niveau donné.

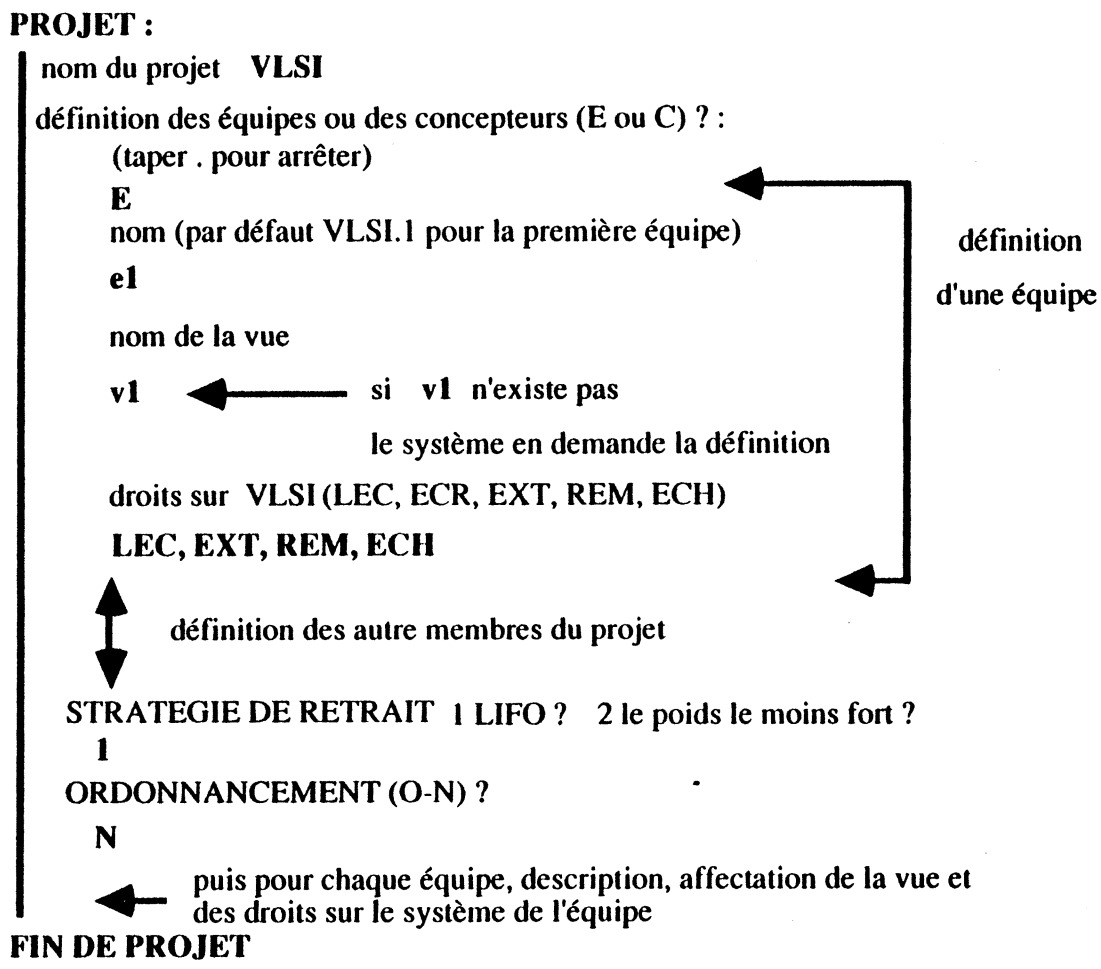


figure 7 : la définition d'un projet

La figure 7 montre la définition d'un projet VLSI (destiné à concevoir un objet de type Circuit. Elle décrit pour ce projet, la définition d'une équipe *e1*, à laquelle on affecte la vue *v1*. Si *v1* n'existe pas, le système en demande la définition (voir figure 13). Le chef de projet accorde pour cette équipe, les droits en lecture et échanges d'objets (*LEC, ECH*) et en exécution et remise de vue (*EXT, REM*). Si aucun droit en exécution n'est donné à une équipe (ou à un concepteur), elle ne peut exécuter que la vue dont elle est propriétaire. Toutes les équipes du projet sont décrites suivant le même processus. Le lecteur trouvera en annexe B, la définition complète du projet.

Pour l'ensemble des équipes décrites (qui définissent ensemble un niveau dans la hiérarchie des systèmes), le chef de projet doit choisir une stratégie de retrait, et un ordonnancement (celui-ci est facultatif). Dans la figure 7, la stratégie choisie consiste à retirer la dernière vue remise, qui a révélé une incohérence. Dans le cas où la deuxième option a été choisie, le

système aurait demandé l'affectation d'une pondération à chaque équipe (ou concepteur) déclarée. Ici, aucun ordonnancement n'est imposé par le chef de projet. Dans le cas contraire le système demande d'affecter un numéro d'ordre aux équipes (ou concepteurs) concernées par cet ordonnancement. Celui-ci peut être partiel.

La définition d'un projet est cohérente si elle satisfait les conditions suivantes :

1 : il n'y a pas recouvrement des vues (les intersections des vues deux à deux sont vides) sinon c'est donc le cas où il existe au moins deux vues dont l'intersection est non vide (elles ont des attributs en commun), alors une règle est donnée pour la remise de cette intersection. Cette règle peut être donnée par la stratégie du premier arrivé, celle du plus fort ou enfin une propriété : comparaison des objets, contraintes vérifiées ou non , (voir **stratégie de conflit**).

2 : l'ensemble des vues définies sur un type permet de reconstituer ce type, en d'autres termes, tous les noeuds de l'arbre de composition (voir figure 1, section 2 de ce chapitre) sont désignés par au moins une vue.

Le processus défini dans ce paragraphe est réitéré sur chaque équipe décrite, et s'arrête lorsque toutes les équipes sont définies en terme de concepteurs.

3.1.3 .Stratégies de retrait et de conflit

Ces stratégies sont mises en œuvre lorsqu'une remise de vue provoque une incohérence (retrait), et lorsqu'il y a recouvrement des vues dans la définition d'un projet (conflit). Dans le premier cas, il s'agit de déterminer quelle vue doit être retirée pour rétablir la cohérence. Dans le second cas, un ou plusieurs objets ont été conçus par des équipes différentes, il s'agit de choisir parmi les solutions proposées.

Dans les deux cas, il est nécessaire d'associer un ordre (si possible total) aux vues définies au même niveau.

Plusieurs choix sont alors possibles :

- LIFO, ou la stratégie du premier arrivé. La vue remise la première est celle qui prime.

- poids le plus fort : on associe un numéro d'ordre à chacune des vues, le retrait peut se faire sur la vue de moindre poids, et la résolution du

conflit en retenant les résultats de la vue de poids le plus fort. Ce numéro d'ordre peut être affecté par l'administrateur du projet, ou calculé par une fonction judicieusement choisie.

Il faut remarquer qu'il est quelques fois difficile d'obtenir un ordre total sur les vues, dans le cas où le numéro d'ordre est calculé par une fonction. S'il y a ambiguïté, le système demande l'intervention de l'administrateur du projet afin de lever cette ambiguïté. Les fonctions utilisées ici sont celles que nous avons définies au chapitre 2.

3.2. Les Systèmes Semi-Publics

Ce sont les systèmes intermédiaires (il peut y avoir un nombre quelconque de systèmes semi-publics suivant l'organisation du projet). Ils contiennent des objets instables, partagés par tous les sous-systèmes fils.

Les systèmes semi-publics gèrent deux types de données :

- les objets instables (incomplets) mais **cohérents**. Ils sont le résultat de remises de vues par un ou plusieurs sous-systèmes. Ces objets sont figés et ne peuvent en aucun cas être renvoyés pour modification vers un sous-système.

- les objets partagés, instables : **incomplets**, ou (et) **incohérents**. Ils peuvent être soit le résultat d'une remise temporaire d'une vue, soit la copie d'un objet faite par un sous-système. La durée de vie d'un tel objet est temporaire.

Le modèle de ces objets est identique à celui décrit dans le système public. Adopter le même modèle de données dans tous les systèmes constituant le projet permet d'homogénéiser et de simplifier les échanges entre eux.

Toutes les fonctionnalités de base du serveur d'objets de ETIC (définition d'un type, d'une fonction, création et modification d'un objet) sont disponibles dans chacun des systèmes (publics, semi-publics, privés) pour son propriétaire.

Les échanges possibles entre un système semi-public et son système père sont de deux types : la **déposition** ou la **lecture** d'un objet instable (dans une optique d'échanges et de coopérations entre les équipes), l'**exécution** ou la **remise** d'une vue (dans une optique de partage de la tâche, puis réalisation de cette tâche par l'équipe assignée). Nous supposons dans ce paragraphe, que le système père est le système public. Le fait que le

système père soit un système semi-public ou public ne change rien aux fonctionnalités offertes et à leur réalisation.

3.2.1. La manipulation des vues

Ces opérations répondent aux besoins liés à la répartition de l'effort de conception. Elles permettent aux différents concepteurs de se partager le travail à faire, dans le respect du cahier des charges de l'objet à concevoir, et en toute sécurité.

3.2.1.1. L'exécution d'une vue

L'exécution est définie sur une vue décrite dans la base mère. A l'initialisation de la conception, l'exécution de la vue se matérialise par la dérivation d'une nouvelle base, semi-publique pour une équipe, ou privée pour un concepteur.

Lorsque cette opération est faite en cours de conception, elle se matérialise par l'enrichissement de la base à l'origine de la demande. Cet enrichissement met en cause la versionnalisation[1] des objets (vues, types, objets, etc..). La versionnalisation est assurée par un système de gestion de versions que nous décrivons dans le chapitre 4. Dans le cadre de ce chapitre, nous nous limitons à l'extraction d'une seule vue. Cette limite sera levée lors de l'intégration du système de gestion de versions dans le système de gestion de projet.

A l'initialisation de la conception, l'extraction est effectuée **automatiquement** lorsqu'un concepteur se connecte au système de gestion de projet. Le système **génère** tous les sous-systèmes permettant de lier dans la hiérarchie de systèmes, le système privé du concepteur et le système public. Grâce aux informations acquises lors de la définition des vues des équipes et des concepteurs, le système connaît la séquence de vues à exécuter pour générer le système privé du concepteur qui se connecte.

3.2.1.2. La remise d'une vue

La remise de la vue est effectuée lorsque l'équipe estime avoir terminé son travail. L'objet conçu dans sa base semi-publique est, dans celle-ci, **complet et cohérent**.

---

1 versionnalisation : action de copier une entité quelconque (objet,type,vue) pour en obtenir une nouvelle version.

Aucune remise portant sur un objet incohérent ou incomplet n'est acceptée.

Le système public calcule si possible les liens, et évalue (si possible) les contraintes. Une contrainte (ou un lien) ne peut être évaluée que si tous les paramètres de la fonction mise en jeu sont connus (instanciés). Dans le cas où cette évaluation met en évidence une incohérence, le système applique la stratégie de retrait pour déterminer le ou les travaux à renvoyer dans les bases semi-publiques des équipes, afin d'y être revus et corrigés.

Plusieurs remises successives de la part de la même équipe sont pour le moment interdites, suite à la limitation faite précédemment. En effet, il est nécessaire de pouvoir répondre à la question : que fait-on d'une vue qui a déjà été remise, et pour laquelle un autre remise est proposée ? Le processus de versionnalisation intervient de même ici.

Les équipes peuvent, si elles le désirent, tenter l'**assemblage** de leurs travaux, pour tester leurs qualités. Il s'agit dans ce cas d'une remise **temporaire** qui sera suivie immédiatement après d'une suppression de l'objet remis dans la base publique. Cette remise ne met pas en jeu le mécanisme de retrait du système public.

Enfin, l'ordre des remises des vues doit être cohérent avec l'ordonnancement défini par le chef de projet.

3.2.2. Echanges d'objets instables

De tels échanges sont motivés par le désir et la nécessité de coopérer dans l'effort de conception. Ce besoin se dégage dans de très nombreux domaines de la CAO (génie logiciel architecture, conception de circuits, ..)

L'équipe propriétaire d'un objet met celui-ci à la disposition des autres équipes. Cet objet est un objet instable, sujet à des modifications de la part de son propriétaire. Il peut être incomplet ou (et) incohérent.

Lorsque l'équipe met cet objet dans la base de projet elle effectue en fait une copie. Les autres équipes peuvent lire cet objet, le copier dans leur système privé, en sachant que celui-ci peut être modifié à tout moment sans notification.

Le système gère à la demande des concepteurs des avis de mises à jour. Une équipe utilisatrice d'un objet ainsi partagé peut demander à être informée de toute modification sur cet objet. Cette demande doit être faite lorsque celle-ci lit l'objet, ou le copie dans son système privé. Le système public gère pour cela un contexte de modification pour les objets soumis à une telle demande.

3.3. Les Systèmes Privés

Les systèmes privés se distinguent des autres systèmes seulement parce qu'ils ne sont pas partagés. Les fonctionnalités offertes dans les systèmes privés sont identiques à celles des systèmes semi-publics. Elles concernent l'exécution et la remise de vues (de et vers le système père), ainsi que les fonctionnalités de base offertes par le serveur de versions (voir chapitre 6). Le concepteur propriétaire est le seul à pouvoir accéder à sa base.

Les échanges entre un système privé et son système père (public ou semi-public) mettent en jeu les mêmes fonctionnalités et les mêmes mécanismes que ceux définis à la section 3.2. entre un système semi-public et le système public.

4. CONCLUSION

Nous avons présenté un système de gestion de projet dans un environnement CAO. Celui-ci prend en compte les besoins en coopération et en partage de l'effort de conception, mis en évidence dans des environnements de conception.

Le système proposé offre des outils pour l'expression du partage de la conception, et assure dynamiquement et automatiquement l'assemblage des travaux résultants de ce partage.

Notre étude place son originalité dans le fait qu'elle propose un outil de partage et une solution pour la mise en commun des travaux réalisés dans les équipes. Les solutions présentées dans la littérature sont orientées versions. Le coopération est permise grâce à la versionnalisation des objets partagés, mais ces études ne proposent pas de solutions quant à la distribution et à l'assemblage des tâches pour la conception des objets [BKK85], [KAL82], [KLM84], [LOR83].

Cette étude consiste de plus en une première étape dans la résolution des transactions en Conception Assistée par Ordinateur. En effet, du fait de leur durée et du volume de données manipulées, les solutions pour régler les accès concurrents dans les SGBD classiques ne sont pas applicables. Offrir un outil pour décrire et contrôler le partage de l'objet à concevoir, dans un environnement de hiérarchie de bases de données simplifie considérablement le contrôle des accès concurrents aux objets [KAW83], [FAU87]. En effet, les fonctionnalités proposées dans ce chapitre, offertes dans un environnement en hiérarchie de systèmes permettent d'une part de diminuer le nombre d'accès concurrents aux objets, et d'autre part, de mieux gérer ceux qui subsistent.

Nous développons en détail la gestion de ces accès concurrents, dans le chapitre 5.

CHAPITRE 4
LE MODELE DE VERSIONS

table des matières

1. INTRODUCTION

Nous avons vu dans les chapitres précédents les problèmes liés à la gestion des objets CAO dans un contexte de projet. Nous avons proposé des concepts (VUE et PROJET) afin d'offrir aux concepteurs les outils qui leurs sont nécessaires pour mener à bien un projet de Conception Assistée par Ordinateur. Grâce au système proposé, les concepteurs peuvent créer, modifier, échanger, valider des objets CAO, qui sont, nous l'avons vu des objets complexes. Ces fonctionnalités ne suffisent plus lorsque l'on introduit la notion de temps. De plus, dans un environnement CAO l'activité est typiquement de tenter d'améliorer un objet, de le corriger lorsqu'il y a des erreurs, d'expérimenter de nouveaux outils, le concepteur ne se contente donc pas d'un seul état de l'objet à concevoir.

Le problème de la gestion de versions n'est pas nouveau. Il apparaît lorsque plusieurs intervenants utilisent et modifient des ressources qui doivent rester cohérentes, qu'il s'agisse de fichiers, de programmes ou de données d'une base. En génie Logiciel par exemple, le concept de version est utilisé pour désigner un programme amélioré, qui remplace l'ancien ou un programme qui offre les mêmes fonctionnalités, mais qui est disponible sur un autre type de matériel. Une version peut aussi être un état clef d'un objet en cours de conception, qui correspond à une étape contractuelle par exemple. En CAO, une version d'un objet peut aussi désigner une tentative d'amélioration de cet objet, ou le choix d'une autre solution technologique. On peut citer ainsi de nombreuses utilisations du concept de version, dans de multiples domaines. Pour résumer nous pouvons donc dire que le but principal d'un système de gestion de versions est d'offrir au concepteur un outil de suivi de son activité.

Un **serveur de versions** permet, dans un environnement de conception (CAO en architecture, en VLSI, génie logiciel ..) de cataloguer, de retrouver et de caractériser les différents états de l'objet à concevoir : les **versions** de cet objet, qu'il s'agisse d'un circuit, d'un programme ou d'un bâtiment. Ces fonctionnalités sont disponibles aussi bien dans un environnement multi-concepteurs, que mono-concepteur. Dans un environnement multi-concepteurs, proposant le concept de **vue** introduit au chapitre précédent, le système de gestion de versions offre des solutions aux problèmes de remise multiple d'une vue, et d'accès concurrents aux objets

partagés. Dans de nombreux systèmes [KAC86], [KLM84], [LOR83], la notion de version est souvent utilisée pour apporter des solutions au problème de reprise après panne et de restauration.

Le but de ce chapitre est de définir les concepts nécessaires à la spécification d'un tel serveur. Nous proposons dans un premier temps (section 1), un modèle de versions adapté à la nature des objets manipulés en CAO. L'idée est de fournir un modèle le plus simple possible, mais évolutif. Nous n'introduisons que deux concepts : les versions **dérivées** et les versions **alternatives**.

Dans un deuxième temps (section 3) nous décrivons l'**évolution** des versions dans les différents systèmes d'un projet de conception. Puis, enfin dans les sections 4 et 5, nous montrons comment le modèle de versions peut être enrichi suivant les besoins des concepteur grâce aux outils de **caractérisation** et de **contrôle** des versions qui sont offerts par le serveur.

Une première approche [RIE86] avait été faite dans le cadre de l'étude CADB pour introduire les concepts de versions d'objets. Celle-ci décrivait un modèle plus complexe, et non évolutif. Quelques outils de contrôle des versions y sont proposés.

L'originalité de notre étude, se place dans le fait que le modèle proposé est simple et évolutif, et que les fonctionnalités offertes au niveau évolution, caractérisation et contrôle des versions sont intégrées dans le système de gestion de projet proposé au chapitre 3.

D'autres études ont été faites afin de proposer des systèmes de gestion de versions [APA87], [DIL85], [KAC86]. Les modèles proposés sont souvent complexes contenant de nombreux concepts, et ne sont jamais intégrés à un système de gestion de projet permettant de gérer la coopération entre les concepteurs et le partage des objets. Mise à part la désignation des versions par leur nom, aucun outil de recherche, ou de caractérisation satisfaisant ne sont offerts. Une analyse de ces différentes propositions est faite dans le chapitre 1 consacré à l'état de l'art.

Nous limitons notre étude aux versions d'objets d'un type donné et fixé. Nous n'abordons pas dans ce chapitre les versions de types qui posent des problèmes non triviaux de maintenance de la cohérence. Des études sont faites à ce sujet et proposent quelques solutions [BAN87]. L'évolution des schémas devrait être prise en compte dans un travail ultérieur.

2. LE MODELE DE VERSIONS

Compte tenu des multiples utilisations de la notion de version, nous nous contentons de donner une définition générale du concept de version qui nous paraît commune à tous les domaines : **une version est un état de l'objet que le système ou l'utilisateur veut conserver.**

2.1. Le Processus de Conception

Le processus de conception peut être décrit comme une séquence d'actions qui tend à transformer un objet dans un certain état vers un autre plus concret, plus précis [DAV81]. Si nous nous plaçons dans un processus linéaire, chaque nouvel état est dérivé du précédent. Dans le cas des applications CAO, ce processus est rarement linéaire. En effet, les possibilités techniques sont diverses, aussi pour résoudre un problème plusieurs solutions sont possibles et chacune doit être développée. Il arrive qu'une de ces solutions mène à une impasse, elle est alors abandonnée.

Dans le domaine particulier de la CAO de VLSI, le processus de conception suit un chemin clairement défini. Les concepteurs doivent concevoir dans l'ordre et pour chaque circuit :

- sa représentation fonctionnelle ;
- sa représentation logique : en terme de portes logiques ;
- sa représentation électrique : en terme de transistors ;
- sa représentation implantée : le dessin de masques qui permettra sa réalisation.

Ces quatre étapes de la conception d'un circuit VLSI mettent en évidence des états contractuels du circuit pour les concepteurs (voir figure 1). Chaque état fait l'objet de vérification afin de déceler au plus tôt d'éventuelles erreurs.

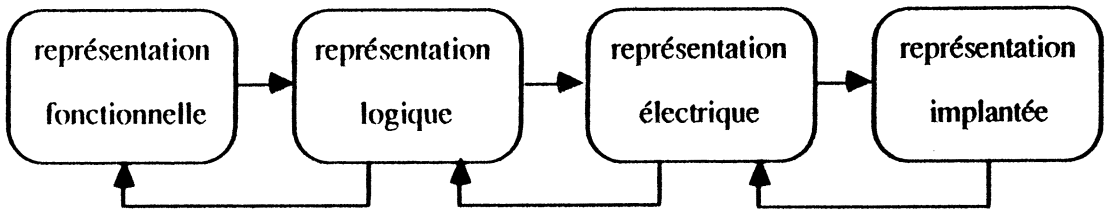


figure 1 : le processus de conception en CAO VLSI

On retrouve ce type de processus dans d'autres domaines que la CAO de VLSI. En architecture par exemple [AFL86b], la conception d'un bâtiment doit passer par différentes étapes telles que l'avant projet sommaire, l'avant projet détaillé, etc ..

Ainsi, le processus de conception est *décrit* par un certain nombre d'étapes clefs, voire obligatoires. L'activité de conception qui permet de passer d'une de ces étapes à une autre est basée sur le principe essai-erreur. Plusieurs tentatives peuvent être faites, avec plus ou moins de réussite. Plusieurs solutions peuvent être valables, même si une seule sera retenue. Le concepteur peut revenir sur des états antérieurs, abandonner des solutions, valider des états qu'il estime terminés. Il s'avère que les remises en question se limitent généralement aux états de l'objet dans la même étape clef. Par exemple, il sera rarement admis, qu'un concepteur travaillant sur la représentation implantée d'un circuit, remette en question des choix effectués au niveau de la représentation fonctionnelle. Si cette remise en question est indispensable elle sera mise en oeuvre avec beaucoup de précautions, car elle implique des répercussions dans tout le projet.

2.2. La Version d'un Objet

Nous l'avons vu, la nature d'un processus de conception, dans n'importe quel domaine, nécessite de prendre en compte et de gérer plusieurs états de l'objet à concevoir : ses différentes **versions**. Nous proposons ici une définition du concept version d'un objet structuré.

Nous avons vu au chapitre 3, qu'un objet est décrit par sa structure, c'est à dire par un ensemble d'attributs, et un ensemble de liens et de contraintes définis sur ses attributs. Nous avons vu de plus que concevoir un objet c'est décrire sa structure et instancier ses attributs. Si nous fixons cette structure, une version de l'objet à concevoir est alors une instanciation donnée des attributs. **En d'autres termes la version d'un objet est**

décrite par l'ensemble des objets qui la composent, chacun dans une version donnée.

La figure 2 ci-dessous décrit deux versions d'un objet à concevoir tel que nous l'avons défini dans la section 2 du chapitre 3.

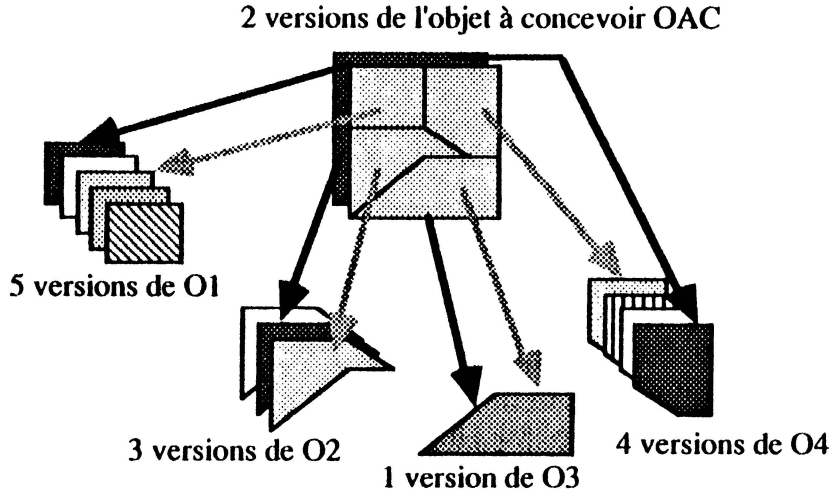


figure 2 : deux versions d'un objet à concevoir

Une version d'un objet (ici OAC) est décrite par une version de chacun de ses sous-objets (O1, O2, O3, O4). Celles-ci peuvent être toutes différentes (cas de O1, O2, O4), ou partagées par une ou plusieurs versions de l'objet (c'est le cas de O3). La relation de composition définie au chapitre 3 s'applique de la même façon aux versions d'objets.

2.2.1. Versions Dérivées

Si nous associons une version à chacun des états de l'objet évoluant dans le temps, nous définissons ainsi le concept de versions **dérivées** (une pour chacun de ces états).

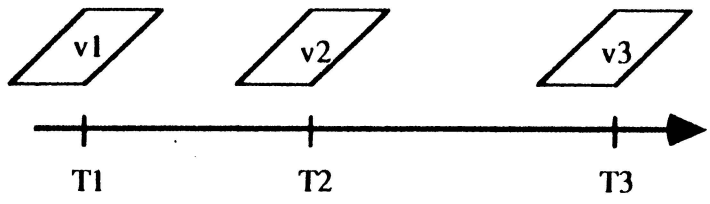


figure 3 : versions dérivées d'un objet

Une version dérivée est construite à partir de sa version antérieure (v2 est dérivée de v1). Cette notion de version dérivée permet de

constituer un historique de l'objet à concevoir. On est capable de situer une version par rapport à sa version antérieure, par rapport à sa version dérivée, et par rapport au temps (date, période de temps écoulé, etc ..).

Dans les applications CAO, le processus de conception nous l'avons vu est rarement linéaire. Le concepteur ne dérive pas seulement une version d'une version donnée. La notion de version dérivée devient alors insuffisante. Nous présentons dans la section suivante le concept de version **alternative**, qui permet de prendre en compte un tel processus.

2.2.2. Versions Alternatives

Dans un processus de conception le concepteur peut, pour un problème donné, développer plusieurs solutions : les **alternatives**. Cela revient à dériver plusieurs versions à partir de la même version (figure 4).

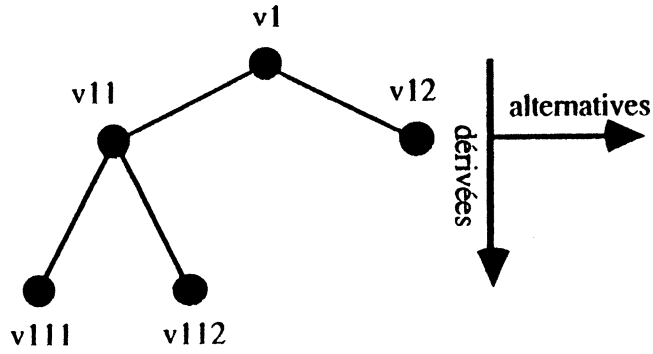


figure 4 : arbre de dérivation des versions d'un objet

Les versions v11 et v12 sont dérivées de la même version v1, elles sont pour cette raison **alternatives**. Elle représentent deux branches possibles pour la conception. Une alternative peut être développée, pour expérimenter un autre outil de conception, pour essayer une autre technologie, etc .. Une branche peut être abandonnée, si elle conduit à une impasse : l'outil ne convient pas, ou la technologie est inadaptée.

La notion d'alternative a été introduite dans [RIE86], [KAL82] pour prendre en compte explicitement des choix différents pour la conception d'un objet. L'alternative est alors temporaire, c'est un essai, ou une tentative.

En résumé, nous avons mis en évidence, deux notions de base pour la caractérisation des versions : les versions **alternatives** et les versions **dérivées**. Ces deux notions permettent de situer, ou de

caractériser une version par rapport au processus qui l'a générée. Elles sont indépendantes de l'application et applicables dans de nombreux domaines de la CAO.

Ces notions sont insuffisantes, car elle ne prennent pas en compte la sémantique de la version (pourquoi est-elle créée, par qui, quelles sont ses propriétés, etc ..), ni l'état de la version vis à vis du système (s'agit-il d'une version en cours de modification, ou a-t-elle été figée, etc ..). Nous explicitons dans la section suivante, les concepts nécessaires à la prise en compte de ces différentes notions.

2.3. Version, composition, spécialisation

Nous avons introduit la notion de **composition** d'un objet. Nous avons vu que ce concept résulte d'une part, de l'analyse d'un problème en sous problèmes, qui est une démarche classique, et d'autre part de la nature même des objets considérés.

Dans le cadre d'un processus de conception, mis en œuvre dans de nombreux domaines (génie logiciel, CAO architecture, CAO VLSI, etc ..), nous avons vu que le principe de **spécialisation** introduit par les langages centrés objets, et pris en compte dans tous les modèles orientés objets, permet de structurer les types en types et sous-types. Nous enrichissons ce principe de spécialisation, en permettant de définir non seulement des attributs supplémentaires dans un sous types, mais de plus de définir des contraintes et des liens de spécialisation.

Enfin, nous avons défini un modèle de **versions**, visant à satisfaire les besoins des concepteurs d'une part, et les obligations d'un processus de conception d'autre part.

Il est intéressant d'étudier, le comportement d'une version d'un objet dans un tel environnement.

Si on considère une version V à un instant donné, dans le type T dans lequel elle a été créée, elle est décrite par sa composition. C'est à dire par les valeurs qui instancient ses attributs, et par son contexte de création (cohérente, complète, ..). Si T est sous type de T', alors V est aussi de type T'. Cette notion est classiquement adoptée dans les modèles orientés objets. Dans le cas où il existe T", sous type de T, et si les contraintes et les liens de spécialisation définis dans T", qui se comportent nous l'avons vu en règles de cohérence forte, sont évalués à vrai, alors V est aussi de type T". V est

décrite dans T'' par les valeurs qui instancient les attributs hérités de T, et ceux définis dans T''. Ces derniers sont rajoutés par le système au moment de l'héritage de V dans T'', et restent inconnus, jusqu'au moment où le concepteur les instancie.

Cela signifie, qu'une version peut être héritées dans différents types, d'un même arbre défini par la spécialisation. Au cours du temps, la version est soumise à des modifications, certaines contraintes qui étaient évaluées à vrai, peuvent être violées et vice versa, et ainsi, satisfaire ne plus être héritées dans ces types, mais dans d'autres types.

On peut donc dire, que non seulement une version peut satisfaire plusieurs types, mais de plus qu'elle peut changer de type au cours du temps.

L'avantage majeur que l'on peut tirer de ce mécanisme, est qu'à un instant donné, on a une topologie fine et précise des versions des objets dans le système. Ceci est un avantage particulièrement dans les domaines de conception, où le but est de définir un type d'objet de plus en plus fin, en prenant en compte des connaissances nouvellement acquises, ou en le contraignant plus.

3. CARACTERISATION DES VERSIONS

Nous avons défini plus haut les concepts de versions dérivées, alternatives. Il est nécessaire, suivant le problème abordé de fournir des critères plus fins pour la caractérisation d'un état de l'objet à concevoir. Par exemple, en CAO VLSI, les représentations logiques, électriques , .. correspondent à des étapes clefs de la conception. Les représentations logiques et électriques d'un même objet par exemple sont équivalentes. Un système de gestion de versions doit permettre l'expression de cette équivalence. La notion de représentation permet en fait d'exprimer une certaine vision que le concepteur a de l'objet à concevoir. Le concepteur doit donc pouvoir exprimer cette vision. En génie logiciel, il est nécessaire de caractériser les différentes révisions d'un module.

Nous proposons de définir le **contexte de création** d'une version (voir section § 3.1) : dans quel but, pourquoi a-t-elle été créée, que représente t-elle ? Par exemple, le concepteur veut rassembler les versions générées au cours de la conception, et qui sont des représentations différentes de son objet.

D'autre part une version peut être caractérisée par les propriétés qu'elle vérifie. Par exemple, dans le cadre d'une application VLSI, il est intéressant de caractériser parmi un ensemble de circuits, les plus rapides, ou ceux qui consomment plus qu'une certaine valeur, etc .. En génie logiciel, il sera intéressant de reconnaître toutes les versions de programmes dont le type de système est UNIX. Nous proposons pour cela d'organiser l'ensemble des versions d'un objet en **classes d'équivalence** (voir § 3.3). Chaque classe contient les versions vérifiant le même ensemble (vide ou non) de règles. Le concepteur peut ainsi classer toutes les versions générées au cours de la conception d'un objet. Il est important de noter qu'il s'agit d'une classification faite à posteriori, qui n'est pas prise en compte par la modélisation.

Une version peut être caractérisée par son état d'avancement. Nous avons vu plus haut, que le concepteur peut décrire un ensemble de contraintes sur les attributs décrivant l'objet. Une version est cohérente si elle vérifie ces contraintes, elle est complète si tous les attributs sont instanciés. Le but du concepteur, est d'obtenir une version complète et cohérente de l'objet à concevoir. Nous avons introduit dans le chapitre 2, les notions de complétude et de cohérence d'un objet, qui, appliquées aux versions sont insuffisantes. Dans la section 3.2, nous définissons deux notions plus fines : la **complétude globale** et **la cohérence globale**.

En résumé nous proposons de caractériser les versions d'un objet par :

- sa situation dans l'arbre de dérivation : dérivée de qui, alternative de qui ?
- son contexte de création : s'agit-il d'une version qui correspond à un état particulier de l'objet à concevoir, par exemple une représentation ?
- ses propriétés : à quelle classe d'équivalence appartient-elle ?
- son état : est-elle globalement cohérente et (ou) globalement complète ?

Ces caractéristiques d'une version sont indépendantes de son évolution dans le système. En effet, une version peut être modifiée à un instant donné, puis archivée plus tard puis encore reprise pour modification, etc ... Il est donc nécessaire de gérer cette évolution que nous abordons dans la section 4.

3.1. Le Contexte de Création

Le contexte de création d'une version, est un ensemble d'informations **automatiquement** déduites par le système à la création de la version, et maintenues de la même façon au cours de la vie de cette version. Il représente la connaissance liée au statut de la version : pour quoi, pour qui, quand. Ces informations sont :

- la version ancêtre (celle à partir de laquelle on l'a dérivée), s'il s'agit d'une version racine d'un arbre de dérivation des versions, cette information est inconnue (l'attribut n'est pas instancié) ;
- la date de création de la version ;
- l'auteur de la création (l'identificateur de l'utilisateur) ;
- la date de la dernière modification ;
- l'auteur de cette dernière modification. Cette information est nécessaire lorsque la version est partagée entre plusieurs utilisateurs ;
- le type de la version. Celui-ci peut, suivant le domaine d'application de la CAO, prendre des valeurs différentes. Seule cette information, est donnée par le concepteur, elle est optionnelle. Par exemple, en CAO de VLSI il peut s'agir de représentation, en génie logiciel de révision.

3.2. L'Etat

L'état d'une version la caractérise en ce qui concerne sa cohérence et sa complétude. Nous avons déjà défini les notions de cohérence et de complétude, pour tout objet de conception. Mais, ces notions sont insuffisantes dans un environnement de versions. En effet, la notion de version cohérente et complète, signifie une version terminée, c'est à dire composée de versions elles mêmes cohérentes et complètes au sens ***terminées***. Or au chapitre 2, nous avons vu que ces notions sont locales à l'objet. Celui-ci est complet, si tous ses attributs sont instanciés, même par des objets incomplets. Il est cohérent si toutes les contraintes évaluables définies sur son schéma (ses attributs) sont évaluées à vrai, même si un des objets qui le compose est incohérent.

Nous introduisons les notions de cohérence, et de complétude d'une version, qui sont **globales** à l'ensemble des versions la composant.

Intuitivement, une version terminée est une version à 100% cohérente et à 100% complète.

Définitions :

- le **degré de complétude** d'une version est le rapport du nombre total de des valeurs de base qui la composent (nombre de feuilles de l'arbre de composition de la version).

- le **degré de cohérence** d'une version est le rapport du nombre total des contraintes évaluées à vrai, définies sur l'ensemble de ses composants par le nombre total des contraintes définies sur l'ensemble de ses composants. Si une contrainte définie sur ses attributs est violée, ce nombre est 0.

- une version est **globalement complète**, si son degré de complétude est égal à 1.

- de même, une version est **globalement cohérente** si son degré de cohérence est égal à 1.

Exemple :

Nous donnons ci-dessous, la définition du type décrivant un voilier, nous utilisons pour cela le formalisme introduit au chapitre 3. Soit Voilier le nom du type :

Voilier : ([coque : Jauge, propulsion : Voilure, port : Chaîne], Ø, Ø)
Jauge ([longueur : Décimal, largeur : Décimal,
tirant_eau : Décimal, déplacement[1] : Poids], Ø, Ø)
Propulsion : ([foc : Surface, gd-voile : Surface, spi : Surface], Ø, Ø).

Le lecteur néophyte en la matière trouvera en annexe A du document, un schéma explicatif quant à cette modélisation.

Pour ne pas surcharger l'exemple, nous n'avons défini aucun lien ni contrainte.

La figure (figure 5) suivante donne l'arbre de composition associé à une version de l'objet *mon-rêve* dans Voilier :

---

1 le déplacement d'un navire, est le poids du volume d'eau qu'il occupe lorsqu'il flotte

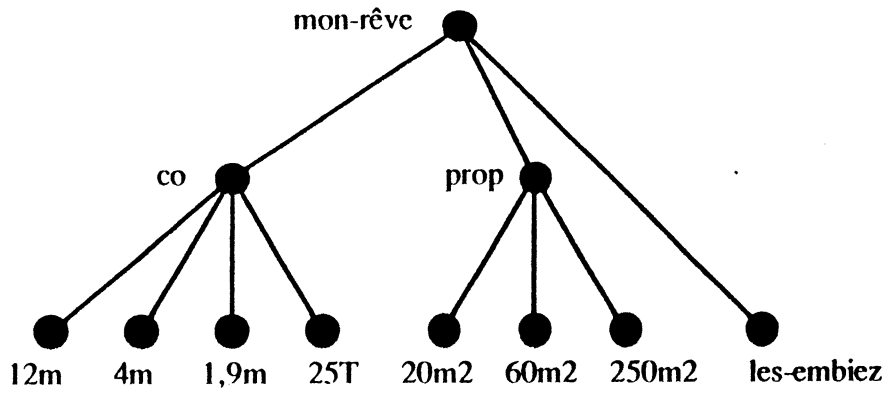


figure 5 : une version de Voilier

L'attribut coque est instancié par la version *co* dans Jauge, l'attribut propulsion est instancié par la version *prop* dans Voilure.

Les versions *co* et *prop* ont des attributs instanciés par des objets de base n'étant pas soumis à la versionnalisation (12m pour la longueur, 4m pour la largeur, 1,9m pour le tirant d'eau, 20m2 pour le foc, 60m2 pour la grand-voile, et enfin 250m2 pour le spi). *les-embiez* est un objet de base (non versionnalisable) instanciant *port* dans Voilier.

En résumé, cette version *mon-rêve*, est composée des versions *co* et *prop* .

Pour calculer le nombre le total de versions de mon-rêve il suffit de faire la somme de :

- nombre total de versions de co : 4
- nombre total de versions de prop : 3
- nombre d'objets de base : 1

soit 8

L'algorithme associé est :

si v est feuille **alors** { v est un objet de base}

nbrtot :=1

sinon nbrtot := $\sum_{i=1,nbatt(v)}$ { nbrtot (attr(i,v))} **fin si**

nbrtot est le résultat (c'est la fonction récursive de paramètre v)
nbfils (v) est une fonction calculant le nombre d'attributs de v.
attr(i,v) délivre la version de l'objet instanciant le $i^{ième}$ attribut de v.

Le nombre de versions complètes est calculé de façon similaire, en ne prenant en compte que les attributs instanciés (voir figure 6).

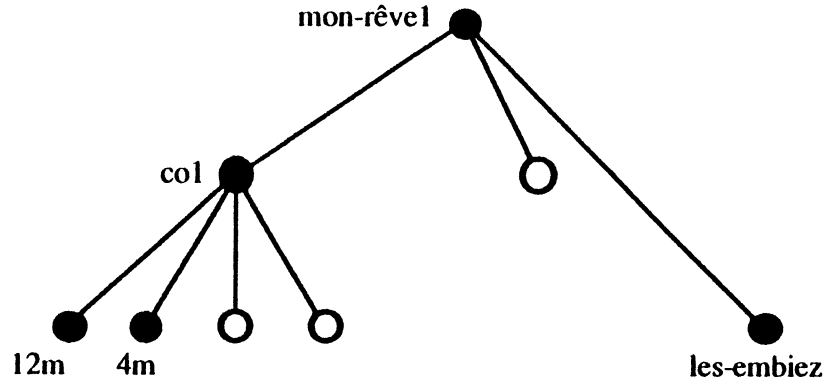


figure 6 : calcul du nombre de versions complètes

La version mon-rêve1, est globalement incomplète. Son nombre de versions complètes est la somme de :

- nombre de versions complètes pour la coque : 2
- nombre de versions complètes pour la propulsion : 0
- le port d'attache est connu : 1

soit 3.

Le degré de complétude de la version mon-rêve1 est donc 3/8= 0,375

Pour illustrer le calcul du degré de cohérence nous pouvons contraindre les schémas utilisés plus haut :

- dans voilier : compatibilité (déplacement.coque, foc.propulsion, gd-voile.propulsion, spi.propulsion). Cette contrainte permet de vérifier que la surface de voilure, est suffisante par rapport au déplacement du voilier.

- dans propulsion : équilibre-gf (foc,gd-voile), qui vérifie les bonnes proportions entre la surface de la grand-voile et celle du foc;

équilibre-gfs (foc,gd-voile,spi) qui effectue la même vérification, mais de façon globale sur les trois voiles.

Le nombre total de contraintes dans Voilier est donc 3. Il est calculé par l'algorithme suivant :

si v est feuille **alors**
nbrcont := 0
sinon
nbrcont := $\sum_{i=1,nbatt(v)}$ { nbrcont (attr(i,v)) + cont(v)}
fin si

nbrcont délivre le résultat.
cont donne le nombre de contraintes définies sur le schéma de v.

Le nombre total de contraintes évaluées à vrai, est calculé de manière identique. Dans notre exemple, si la contrainte de Voilier est évaluée à faux, alors le degré de cohérence de la version mon-rêve est 0. Sinon, supposons qu'une seule contrainte de propulsion soit vérifiée, alors la degré de cohérence de la versions mon-rêve est 2/3.

3.3. Les Classes d'Equivalence

Définir une relation d'équivalence sur les versions d'un objet permet de les caractériser suivant un ensemble de critères [KSW86]. L'idée de base est de structurer les versions d'un objet, pour permettre au concepteur d'effectuer ses recherches parmi un nombre limité de versions satisfaisant certaines conditions qu'il aura définies par avance.

La relation d'équivalence est définie comme suit :

$V_i \mathcal{R} V_j$, si V_i et V_j vérifient le même ensemble de règles. Ces règles sont définies par le concepteur avec le même formalisme que celui utilisé pour les contraintes définies dans les types.

Un ensemble E de règles applicables à un objet est donné par le concepteur. Chaque classe d'équivalence est associée à un sous-ensemble de E. Une version appartient à une classe d'équivalence, si elle vérifie la conjonction des règles associées à cette classe.

Remarque : une version $\mathcal{V}$ vérifie une règle si celle-ci est évaluable pour $\mathcal{V}$, et si elle est évaluée à vrai pour $\mathcal{V}$.

L'ensemble des parties de E peut être calculé automatiquement [CHE87] et de façon exhaustive, par le système. Ce mode d'utilisation permet d'obtenir tous les regroupements possibles des règles définies dans E.

Par exemple :

soit $\{a, b, c, d\}$ un ensemble de 4 règles.

Il y aura dans notre exemple, 16 classes d'équivalence correspondant aux 16 parties de l'ensemble $\{a, b, c, d\}$.

Ce calcul automatique, s'il soulage le concepteur d'un certain travail, génère en outre un ensemble de classes d'équivalence qui n'ont aucun sens, soit parce qu'il existe des règles contradictoires associées à la même classe, soit par ce que le regroupement de règles dans une même partie n'a aucun sens dans la réalité. Il est donc nécessaire d'offrir un autre mode de production, en plus du calcul automatique.

Le concepteur peut s'il le désire donner lui même l'ensemble des règles associées à chaque classe, par exemple : $\{a, b\}$, $\{a, c, d\}$ deux ensembles de règles qui permettent de construire trois classes :

C1 = ensemble des versions vérifiant **a** ET **b**

C2 = ensemble des versions vérifiant **a** ET **c** ET **d**

C3 = ensemble des versions qui ne sont ni dans **C1** ni dans **C2**,

Ainsi, une classe contient toutes les versions équivalentes selon les règles qui l'ont définie.

Exemple :

Sur le type Voilier, défini plus haut on peut écrire les règles suivantes :

- rapide (déplacement.coque, propulsion), un bateau rapide est un bateau dont la surface de voiles est *suffisante* par rapport à son déplacement.

- confortable (largeur.coque, longueur.coque), un bateau confortable est un bateau plutôt large.

- bateau de prés (foc.propulsion, gd-voile.propulsion, tirant-eau.coque, largeur.coque, longueur.coque), enfin un bateau de près, c'est à dire capable de remonter au vent avec une bonne vitesse, possède une quille profonde (tirant d'eau), un foc de grande surface par rapport à la celle de la grand'voile, et une coque longue et de petite largeur.

Grâce à ces trois règles, on peut maintenant classer les différentes versions d'un objet de type voilier. La figure 7 illustre cette classification.

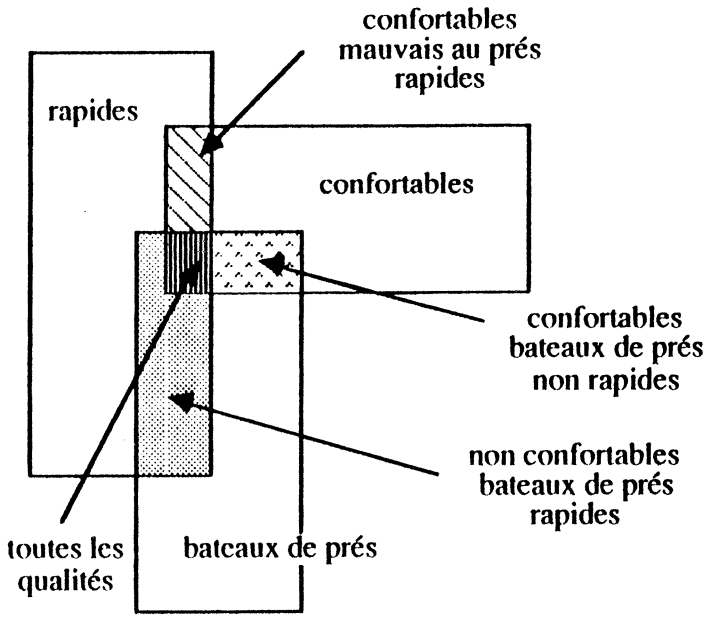


figure 7 : une classification des versions d'un objet de type Voilier

Dans la figure précédente, toutes les versions qui ne vérifient aucune règle n'apparaissent pas explicitement. Elles appartiennent au complémentaire de la réunion des trois ensembles donnés.

4. LA MANIPULATION DES VERSIONS

Nous avons défini la notion de version et nous avons décrit un ensemble de solutions pour permettre la caractérisation et la classification des versions d'un objet, d'un type donné.

Nous présentons maintenant les différentes étapes de la vie d'une version. C'est de la responsabilité de l'usager de décider si tel état de l'objet doit être ou non considéré comme une version. A partir de là, nous lui proposons des mécanismes pour gérer ses versions. Le passage d'une étape de la vie d'une version vers une autre est laissé au choix du concepteur. Nous introduirons dans la section 5, des outils d'aide et de contrôle pour l'évolution des versions.

La section 4.1 décrit l'évolution des versions dans l'environnement privé des concepteurs, puis la section 4.2 intègre cette évolution aux concepts de vues et de projet dans les systèmes partagés.

4.1. Dans les Systèmes privés

Les concepteurs, après extraction de la vue qui leur est assignée (voir chapitre 3), disposent d'un environnement privé dans lequel ils ont à leur disposition tous les outils nécessaires à leur travail de conception (objets de base, outils de conception, commandes de manipulations d'objets, etc ..).

L'opération d'extraction a délivré un type d'objet : leur objet à concevoir. Ils peuvent instancier ce type, l'affiner (voir création d'occurrence, constructeur ISA, chapitre 1).

Les multiples tentatives qui sont faites pour instancier le schéma donné, pour améliorer l'objet obtenu, produisent un certain nombre de versions de cet objet. Les opérations de création, de dérivation, de gel, et de dégel vont permettre aux concepteurs de faire évoluer et de multiplier les versions de leur objet, jusqu'à obtenir la version désirée : celle qui est globalement complète et globalement cohérente, et qui offre certaines performances ou caractéristiques définies par des règles.

4.1.1. Version courante, version gelée

La **version courante** d'un objet est la version en cours de modifications. A l'initialisation de la conception, une version courante de l'objet à concevoir est créée. Elle représente le point de départ du processus de conception.

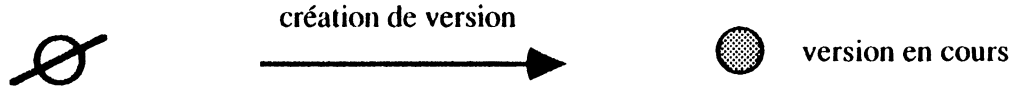


figure 8 : la création initiale de la version en cours

Aucune version n'existait avant la création. Une seule version existe après : la version en cours.

Une version d'un objet est soit dans un état courant, soit dans un état gelé. Une **version gelée** est une version stable, sur laquelle toute modification est interdite. Les seules opérations possibles sur une version gelée sont le dégel et la dérivation.

Ces notions de versions courantes et versions gelées ne sont pas nouvelles [PAL87], [KAC86].

4.1.2. Gel et dégel

Le gel de la version courante la stabilise. Cette fonctionnalité permet de figer un état de l'objet.

Le résultat de cette opération est que la version visée n'est plus la version en cours (voir figure 9[1]).

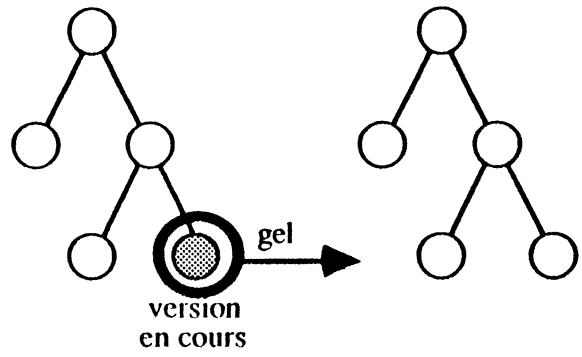


figure 9 : gel d'une version

Dans la figure précédente, la version courante qui est feuille dans l'arbre de dérivation des versions de l'objet à concevoir, est gelée.

Il est interdit de modifier une version gelée. Les seules opérations possibles sont celles de dégel et de dérivation.

Le **dégel** d'une version est l'opération duale du **gel** : la version visée devient la version en cours. Si une version en cours existait auparavant, elle est automatiquement gelée par le système.

---

1 l'arbre représenté dans cette figure est un arbre de dérivation de versions d'un objet. Les nœuds laissés en blanc sont des versions gelées, les nœuds grisés sont des versions en cours.

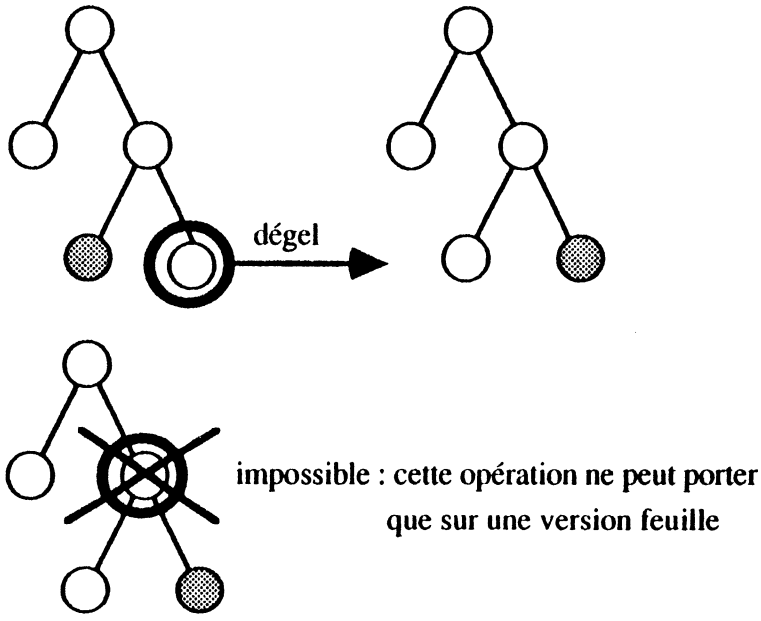


figure 10 : dégel d'une version

Le dégel sur une version qui possède des versions dérivées n'a pas de sens. En effet, la relation de dérivation entre deux versions n'existe plus si celle qui a servi de base évolue. On dérive une version à partir d'une autre car on veut l'améliorer, ou la modifier en gardant la même logique de conception. Il existe une relation entre ces deux versions, qui s'exprime grâce à la dérivation.

Seul le dégel sur une feuille de l'arbre de dérivation des versions d'un objet est autorisé.

Les opérations de **gel** et **dégel** ne produisent pas de nouvelles versions. Elles permettent uniquement d'archiver ou de rendre en cours, les versions existantes.

4.1.3. Création et dérivation

Nous avons vu (section 4.1.1), qu'à l'initialisation de la conception une version de l'objet à concevoir est créée. Le concepteur peut, en cours de conception, créer une nouvelle version. Cette fonctionnalité lui permet de démarrer une nouvelle branche dans l'arbre de dérivation des versions de l'objet à concevoir : le concepteur tente des choix radicalement différents pour sa conception.

La version créée par cette action devient la **version en cours.** Dans le cas où une version était en cours, le système la **gèle**

automatiquement. En effet, seule une version peut être en cours à un instant donné dans un système donné.

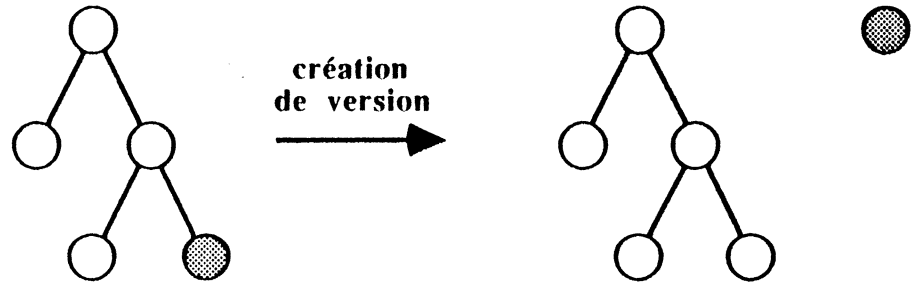


figure 11 : création de version

Dans ce cas, la création d'une version a induit le gel de la version en cours, et la création d'une nouvelle version en cours déconnectée de l'arbre de dérivation.

Dériver une version permet de produire une nouvelle version de l'objet en cours de conception. L'opération s'applique à une version en cours ou non, et en délivre une copie qui est alors la version en cours. Si la version à partir de laquelle on a fait la dérivation était en cours, elle devient gelée, sinon il n'y a pas de changement dans sa situation (voir figure 12).

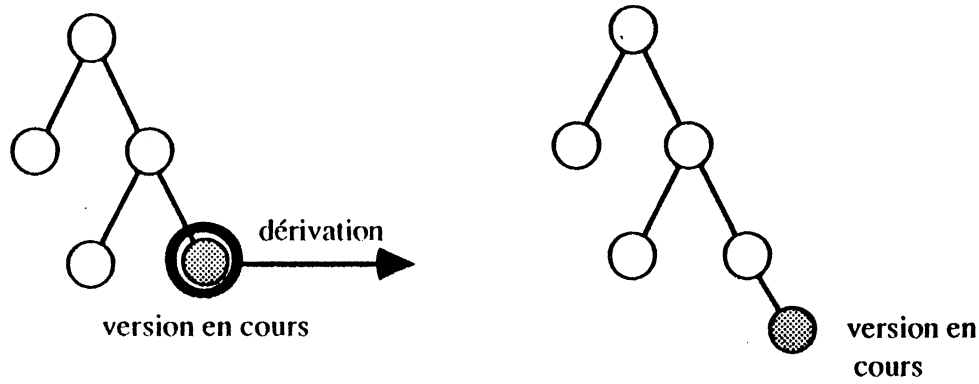


figure 12a: la dérivation de version

La figure 12a, illustre l'affinage de l'objet. On fige un état de celui-ci, et on progresse dans le processus de conception. Les choix de conception reste les mêmes. On veut simplement figé un état de l'objet.

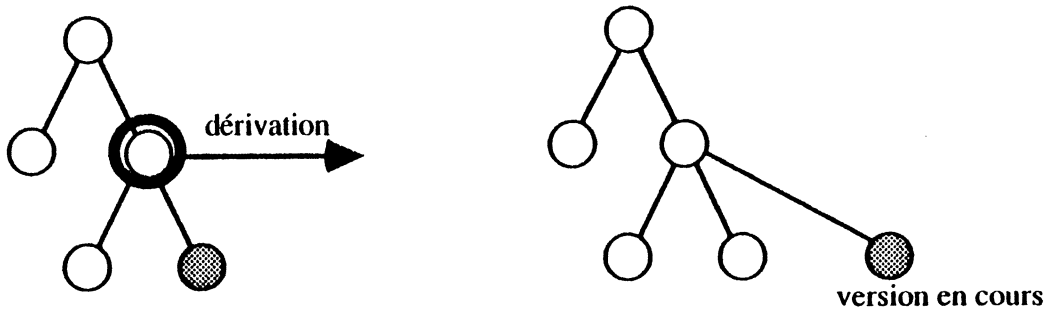


figure 12b : la dérivation de version

Le deuxième cas (figure 12b), illustre la dérivation d'une **alternative** : le démarrage d'une nouvelle branche de la conception, pour essayer un autre outil, tenter une autre solution. Ce principe est à utiliser lorsque une certaine logique de conception est respectée. Il est évident que ce type de contrainte d'utilisation est la charge du concepteur. Certain contrôle peuvent être effectués (voir § 5).

4.1.4. Abandon

L'opération d'abandon sur une version permet d'élaguer l'arbre de dérivation des versions. Elle signifie que l'axe de conception choisi s'est révélé être une erreur, ou d'aucune utilité. Cette opération porte sur une version, elle entraîne la destruction de la version visée, ainsi que de toutes ses versions dérivées (voir figure 13).

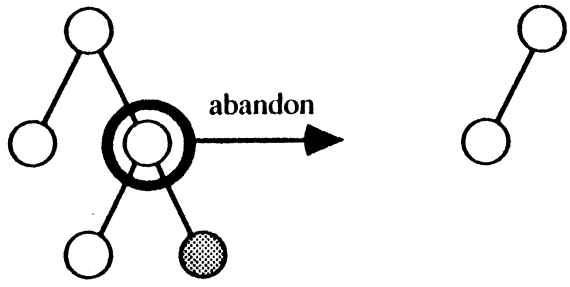


figure 13 : abandon de version

L'axe ne répond plus aux besoins de la conception, on choisit de détruire toutes les versions qui en découlent.

Un abandon de version peut porter sur une version quelconque, en cours ou non. Dans le cas où la version en cours est abandonnée il n'y a plus de version en cours. Le concepteur devra choisir une version feuille, et la dégeler pour continuer le processus de conception, ou bien appliquer la dérivation sur une quelconque des versions gelées.

4.2. Dans les Systèmes Partagés

Dans un système partagé toute opération est une requête émise par un des systèmes fils, ou par l'administrateur du projet. Dans le cas d'un système semi-public, les fils peuvent être des systèmes privés ou semi-publics, dans le cas du système public, les fils sont généralement, des systèmes semi-publics. Lorsque l'administrateur du projet a décrit le projet et affecté à chacun son travail, les requêtes concernent alors les opérations de :

(1) extraction de vue, à l'initialisation de la conception
(2) remise de vue, à la fin de la conception
(3) lecture d'un objet instable
(4) écriture d'un objet instable

Ces opérations sont décrites dans le chapitre 3. Les deux premières opérations sont contrôlées par le système, et soumises aux règles décrites par l'administrateur du projet (voir définition d'un projet). Elles portent sur des objets, et plus précisément sur des versions d'objets.

Les opérations 3 et 4 sont de simples copies de versions d'objets.

L'évolution d'une version dans un système partagé, est entièrement déterminée par les opérations de remise de vues. Une remise de vue peut modifier une version (la version en cours), ou en dériver une nouvelle. Les opérations de gel et d'abandon sont soumises à la volonté de l'administrateur du projet. Nous verrons dans la section 5, quels sont les outils de contrôle de ces opérations, grâce à des règles décrites par l'administrateur.

4.2.1. Création

Une version est créée lors de la première remise dans le système public ou semi-public correspondant, par un sous-système. Elle est la version en cours du système. Toutes les remises suivantes, concernant chacune une vue différente viendront modifier la version en cours.

Prenons par exemple, la conception d'un objet OAC, qui a été partagée entre trois vues *vue1*, *vue2* et *vue3* distribuées entre E1 et E2. E1 doit concevoir les parties définies de OAC définies par *vue1* et *vue2*, et E2 doit concevoir celle définie par *vue3*.

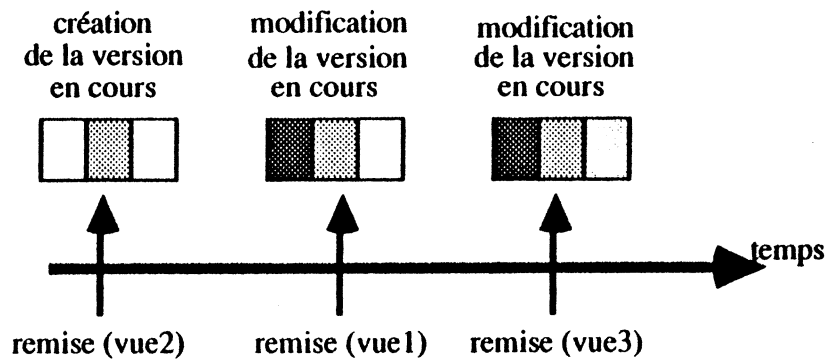


figure 14 : remises successives des vues

La figure 14 illustre le cas (idéal et rarissime) où chaque vue n'est remise qu'une seule fois. L'objet à concevoir est représenté dans cette figure par un rectangle, divisé en trois parties, chacune associée à une vue. Si la vue est remise (terminée), la partie qui lui correspond dans la figure est grisée. On a associé à chacune des vues un niveau de gris différent.

Les équipes E1 et E2 n'effectuent qu'une seule remise pour chacune des vues qui leur ont été affectées. Lors de la remise de *vue3*, seule une version de l'objet à concevoir a été développée.

Mais il peut arriver qu'une vue soit remise plusieurs fois, on dit alors qu'il y a remise multiple.

Une remise multiple induira la dérivation d'une nouvelle version.

4.2.2. Dérivation

Dans un système partagé, si on admet les remises multiples, il est nécessaire de laisser cohabiter plusieurs versions en cours. L'exemple suivant illustre le problème posé.

Soit un objet O, dont la conception est partagée en trois vues de O : A, B, C. La conception de O est terminée lorsque les trois vues sont remises, en respectant les règles de la conception.

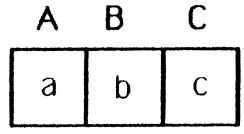


figure 15 : cas de remise simple

Dans la figure 15, chaque vue n'a été remise qu'une fois, ainsi, sans autre intervention, une seule version de *O* a été générée (la remise de *A* a porté sur *a* , *B* sur *b* et *C* sur *c*).

Dans la figure suivante, l'opération *R* (a_i) (respectivement *R* (b_i), *R* (c_i)) correspond à la $i_{ième}$ remise, qui s'applique à la vue *A* (respectivement *B*, *C*).

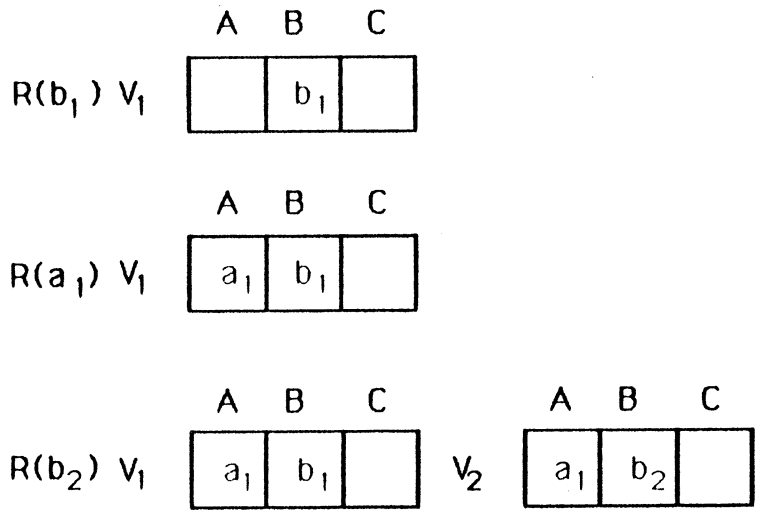


figure 16 : remise multiple de B

B fait l'objet d'une nouvelle remise. Une version V_2 est alors **alternative** de V_1. V_2 est une copie de V_1 dans laquelle la remise de B sur b_1 est remplacée par celle sur b_2, une nouvelle tentative est faite pour la conception de B. Toute remise viendra maintenant s'effectuer sur V_1 et V_2 (voir figure 17).

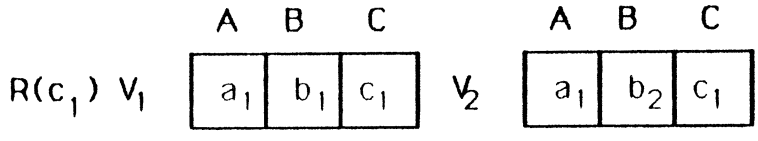


figure 17 : remise simple de C

V_1 et V_2 sont considérées comme les deux versions en cours du système partagé. Toute remise simple, s'effectue sur toutes les versions en

cours. Toute remise multiple, vient dupliquer chaque version en cours, afin de produire autant d'alternatives (figure suivante) :

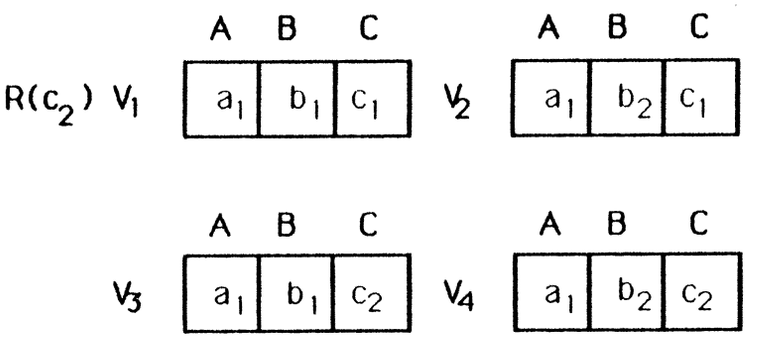


figure 18 : remise multiple de C

Dans le cas où aucune opération de gel et de dérivation n'est faite par ailleurs, quatre versions alternatives sont produites par le processus décrit par les figures précédentes.

Dans le cas général, c'est à dire lorsqu'un certain nombre d'opérations de gel ont été effectuées, les versions en cours sont les versions feuilles de l'arbre des versions. Une remise multiple vient alors dupliquer chacune des versions feuilles (figure 19) : dans chaque duplication, la remise multiple de la vue, remplace la remise précédente de la vue.

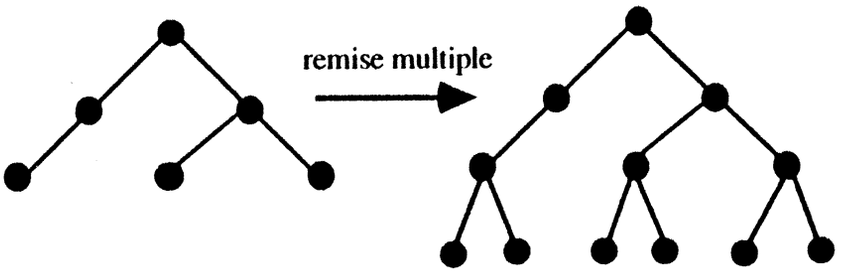


figure 19 : remise multiple

Dans l'absolu, le système doit accepter et gérer toutes les remises multiples demandées. En effet, lorsqu'un concepteur (ou une équipe) demande une remise multiple, c'est pour proposer une solution qui est aussi bonne, sinon meilleure que celle qu'il a proposée avant. Une version, parmi toutes les versions feuilles de l'arbre de dérivation est particulière, c'est

celle dont toutes les remises sont simples, ou correspondent à la première remise. C'est le cas de V_1 dans la figure 18.

Il est évident qu'un tel processus conduit vers l'explosion combinatoire du nombre de versions d'un objet. Nous proposons dans la section 5, des outils permettant d'éviter cette explosion.

4.2.3. Gel, Dégel et Abandon

Les opérations de gel, de dégel et d'abandon sont explicitement demandées par l'administrateur du projet. Elles produisent le même résultat que dans les systèmes privés.

5. LE CONTROLE DES VERSIONS

Nous avons vu dans la section précédente, que le concepteur peut faire évoluer les versions d'un objet par l'intermédiaire de cinq commandes (création, dérivation, gel, dégel et abandon). Il est nécessaire de générer suffisamment de versions pour qu'elles décrivent fidèlement la conception de cet objet. Il faut de plus que ces versions soit significatives, c'est à dire qu'elles correspondent à une étape clef de la conception. Mais, il faut en contrôler le nombre. En effet, on ne dispose pas de mémoire illimitée (bien que la limite soit poussée de plus en plus loin) et la prolifération des versions peut conduire le concepteur à passer la majeure partie de son temps à reconnaître les versions qui l'intéressent, et à détruire les autres. En Génie Logiciel par exemple, l'étude faite dans le cadre du système RCS, [TIC85] met l'accent sur ce point et propose de séparer l'édition du programme de sa versionnalisation, en laissant au concepteur l'initiative de la création d'une version. Opter pour une telle solution permet en effet de limiter le nombre de versions mais ne garantit pas une trace fidèle de la conception.

Un des rôles principaux d'un système de gestion de versions est donc de limiter l'accroissement trop important du nombre de versions d'un objet, et d'obliger un nombre minimum de ces versions pour avoir une trace significative de la conception. Il doit donc permettre de définir et de gérer des règles pour la génération des versions. Ces règles sont définies pour un projet donné, par l'administrateur du projet, suivant le même formalisme que celui utilisé pour la définition des contraintes.

La modification d'une version, peut entraîner un grand nombre de modifications sur les versions qui l'utilisent (elle peut servir de base de

calcul, elle peut instancier une attribut d'un versions donnée, ..). Ces modifications sont de différents types :

- la cohérence de la version peut être modifiée, cela implique la répercussion de cette modification sur toutes les versions appartenant au même arbre de composition.

- la complétude : ici se pose le même problème de répercussion.

- un nouveau calcul des liens sur les attributs : ceux-ci peuvent prendre une nouvelle valeur de base, ou être instanciés par une occurrence différente générée par le système à cette occasion. Dans ce cas, on peut choisir de répercuter en cascade dans l'arbre de composition, ou de créer une nouvelle version prenant en compte ces répercussions, et appliquer ce principe récursivement jusqu'à la racine de l'arbre de composition.

La propagation des mises à jour peut être contrôlée, afin de limiter le nombre de versions produites par une mise à jour, en ne propageant les mises à jour que sur certaines versions de l'objet [KCH87]. Nous proposons un contrôle de la mise à jour à la carte. C'est à dire que le concepteur choisit la portée d'une modification : sur toute la base, ou sur certaines versions répondant à un certain critère (voir chapitre 6, sur la réalisation).

Nous différencions dans notre étude modification et création de version. Nous avons vu en effet, qu'il n'est pas raisonnable de dériver une nouvelle version chaque fois qu'une modification est à prendre en compte.

Pratiquement, le système de contrôle des versions agit chaque fois qu'il y production d'une nouvelle version (par la création ou la dérivation). Il agit donc sur toute action qui conduit à l'archivage d'un état donné de l'objet en cours de conception.

Ce système peut **autoriser** ou **interdire**, mais aussi **obliger** :

- obligation : pour assurer un nombre minimum de versions, afin de produire une trace significative de la conception, le système prend à sa charge la dérivation de versions sans intervention de l'utilisateur. Il obéit pour cela aux **règles d'obligation** définies par l'administrateur, et éventuellement par les concepteurs sur leur système privé.

- autorisation, interdiction : pour ne pas générer un nombre prohibitif de versions, le système interdit dans certains cas la dérivation, ou la création d'une version. Les **règles d'autorisation** ou **d'interdiction** sont définies comme précédemment.

Nous définissons dans la section 5.1 les règles d'obligation en illustrant leur utilisation. La même étude est faite dans la section 5.2, pour les règles d'interdiction-autorisation.

5.1. Obligation

La décision d'obliger la dérivation d'une version peut être faite suivant une certaine fréquence, ou suivant son évolution.

5.1.1. Fréquence

Ce critère de dérivation est indépendant de l'état, ou du contenu de la version. Il répond seulement à la demande : il est nécessaire de dériver une version de l'objet tous les jours, ou toutes les semaines, etc ..

L'opération de dérivation est faite par le système, la seule action de l'utilisateur peut porter sur le nom de la version ainsi dérivée. Dans le cas où aucun nom n'est fourni, le système le calcule en concaténant au nom de la version mère un entier correspondant au numéro d'ordre de la version dérivée dans la liste de ses versions soeurs (voir figure 20) :

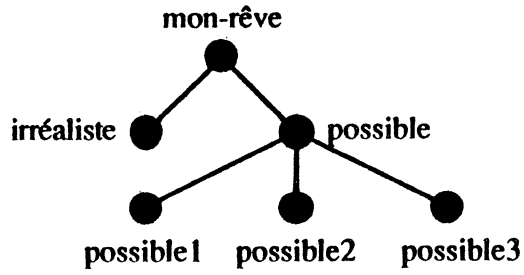


figure 20 : affectation automatique du nom

La règle permettant d'exprimer l'obligation sur la fréquence est de la forme : **fréquence-max (type,unité,nb-unités)**. (voir définition des règles d'obligation, chapitre 6 sur réalisation).

Où on a :

- **type** : qui désigne un type qui peut être le résultat de l'exécution d'une vue.

- **unité** : qui indique l'unité de temps choisie (l'année !!, le mois, le jour, l'heure ou la minute).

- **nb-unités** : qui est l'intervalle de temps maximum, séparant deux générations de versions.

Cette règle s'applique donc à tous les objets du type donné. Pour la référer, l'utilisateur (administrateur ou concepteur) n'a pas à définir la fonction correspondante, il s'agit en effet d'une fonction prédéfinie. Le concepteur ne peut utiliser cette règle que si :

- le type visé n'est soumis à aucune autre règle de fréquence
- **ou** si la fréquence indiquée par le concepteur est inférieure à toutes les fréquences données par des règles préalablement définies sur le même schéma.

5.1.2. Evolution

Contrairement au critère précédent, celui-ci s'intéresse au contenu de l'objet et à son comportement.

On peut obliger la dérivation d'une version :

(1) lorsqu'elle change de **classe d'équivalence**

(2) lorsqu'elle change d'**état**, c'est à dire lorsqu'elle passe de globalement incohérente à globalement cohérente, et vice et versa, ou bien, lorsqu'elle devient globalement complète ou incomplète

(3) lorsqu'elle change de **contexte** (d'auteur,...)

Cela revient en fait à estimer que ces changements sont en quelque sorte *importants* , et qu'ils méritent d'être notés dans le traçage de la conception.

Une telle règle d'obligation s'exprime pour un schéma d'objet donné. Le concepteur ou l'administrateur spécifie alors si celle-ci porte sur la classe ou (et) l'état ou (et) le contexte. (Voir définition des règles d'obligation, chapitre 6).

5.2. Interdiction, Autorisation

Ce type de contrôle permet de ne retenir que les versions significatives de l'objet et d'en limiter le nombre. Il agit de même sur la fréquence et aussi sur les propriétés que l'objet vérifie.

5.2.1. Fréquence

La règle exprimée dans ce cas est sensiblement identique à celle décrite dans la section 5.1.1. En effet, il s'agit ici de donner un nombre maximum de versions sur une période donnée, par exemple le jour : on n'autorise qu'une nouvelle version (par création ou dérivation), pour les objets d'un type donné, pour chaque jour. Cette règle est de la forme :

fréquence-min (type,unité,nb-unités)

fréquence-min est de même, une fonction prédéfinie. Le concepteur peut décrire ses propres règles d'interdiction sur la fréquence, si la fréquence donnée est supérieure à celle préalablement définie, ou si aucune règle d'interdiction sur la fréquence n'était donnée.

5.2.2. Acceptabilité

La notion d'acceptabilité a préalablement été introduite dans [RIE86]. Elle a pour but d'exprimer un critère de qualité sur les versions d'un objet. On n'autorise le gel (à travers les instructions de dérivation, création ou gel) d'une nouvelle version, que si la version gelée à cette occasion vérifie un ensemble donné de critères de qualité.

Les règles d'acceptabilité sont décrites suivant le même formalisme que celui utilisé pour les contraintes. Elles sont définies sur un type donné, et s'applique à tous les objets de ce type.

6. CONCLUSION

Nous avons proposé dans ce chapitre un système de gestion de versions. Nous avons décrit un modèle basé sur deux notions seulement : les versions dérivées et les version alternatives. Nous offrons aux concepteurs et à l'administrateur du projet des outils pour enrichir ce modèle, suivant leurs besoins. Ces outils sont basés sur (1) les classes d'équivalence, (2) l'état de la version, (3) le contexte de création.

Les fonctionnalités disponibles sont : (1) la création de version, (2) la dérivation (3) le gel (4) le dégel et (5) l'abandon. Le concepteur dispose de ces 5 fonctions pour concevoir son objet (le créer, le modifier dans un schéma donné).

Nous proposons un ensemble de fonctionnalités destinées à contrôler le nombre et la pertinence des versions générées pour un certain schéma.

Les commandes de manipulation de versions sont adaptées aux besoins des différents systèmes (privés ou partagés). Le modèle de versions, les outils de contrôle sont par contre indépendants des notions de vues et de projet décrites dans le chapitre 3. Elles s'appliquent et s'intègrent en effet dans n'importe lequel des systèmes privés, semi-publics ou public.

CHAPITRE 5

LE GESTIONNAIRE DES TRANSACTIONS

table des matières

1. INTRODUCTION

Nous avons montré dans le chapitre 1 (Introduction et état de l'art) les problèmes posés par les transactions et par les objets CAO en général. Nous avons proposé dans le chapitre 3, les concepts de vue et de projet, qui permettent dans un environnement partagé, d'expliciter et de contrôler certains accès aux objets.

Nous décrivons dans la section 2 de ce chapitre l'interface entre l'usager et le système. Nous anticipons un peu sur le chapitre 6 consacré au prototype, mais il est nécessaire d'introduire ce dialogue car notre notion de transaction lui est fortement liée.

Dans la section 3, nous redéfinissons les notions de base liées à la gestion des transactions. Ceci nous permet d'introduire un ensemble de mécanismes qui répondent aux besoins de contrôles de cohérence, de reprise après panne, et d'abandon du travail en cours.

Nous abordons enfin les problèmes de concurrence d'accès aux données. Nous ne présentons ici qu'une ébauche de solution qui demanderait des développements futurs.

2. LE DIALOGUE

Le dialogue entre le concepteur et le système est fortement interactif. Il est dirigé par le système. Il est à la fois le langage de définition et le langage de manipulation des objets. Nous allons montrer dans cette section l'importance du dialogue dans la gestion des transactions.

2.1. Commande

Au cours d'une session du système plusieurs **commandes** peuvent être soumises par un usager. Elles peuvent concernées :

(1) la définition d'un projet

(2) toutes les opérations sur les vues (définition, exécution, remise)

(3) les opérations sur les types (définition d'un type et d'un sous type)

(4) toutes les opérations sur les versions et les objets (création, dérivation, modification, gel, dégel, abandon, copie et dépôt).

La figure suivante retrace le déroulement d'une commande de définition de type. Le point de vue programmation des commandes est décrit au chapitre 6, sur le prototype ETIC.

```
DEF-TYPE Point                          <retour chariot>
        x : Entier                      <retour chariot>
        y : Entier                      <retour chariot>
  FDEF                    <retour chariot>
```

figure 1 : commande de définition de type

La commande de définition de type est activée par l'instruction *DEF-TYPE Point* , et elle se termine par *FDEF*.

A la fin de la commande, le type (ici Point) est effectivement enregistré dans la base.

2.2. Ligne de commande

Une ligne de commande est une instruction donnée par le concepteur. Dans la figure précédente (figure 1), une ligne de commande est de la forme :

y : Entier <retour chariot>.

Celle-ci indique que Point admet un attribut *y*, de type *Entier*. Cette ligne n'est acceptée par le système, que si elle vérifie un ensemble de règles dites **règles d'admissibilité** que nous décrivons en détail dans la section 3.1 de ce chapitre.

3. LES TRANSACTIONS

L'étude effectuée au chapitre 1 a permis de mettre en évidence qu'une transaction représente généralement, à la fois l'unité de cohérence et l'unité de restauration. Nous pensons qu'il est nécessaire dans un environnement comme celui de la CAO de remettre en cause ce principe. La figure suivante montre le concept d'atomicité adopté dans les systèmes de gestion de transactions classiques.

début transaction => la base de données est dans un état **cohérent**
(toutes les contraintes d'intégrité sont évaluées à vrai)

action$_1$

.

action$_i$

.

Fin Transaction ☛ si une action met en évidence une incohérence de la base de données, si une panne ou un accès concurrent survient, toutes les actions de la transaction sont défaites, et ignorées.

=> la base de données est dans un état cohérent

figure 2 : une transaction = un ensemble d'actions

Nous avons montré qu'ignorer un ensemble d'actions, dans le cas de l'arrêt imprévu d'une transaction est inacceptable dans un environnement CAO (chapitre 1).

Nous proposons de définir trois notions différentes :

1 : Unité d'admissibilité : un ensemble d'actions qui sont ignorées lors de la violation des règles d'admissibilité. Nous rappelons qu'un objet peut être temporairement incohérent, au sens des contraintes d'intégrité définies dans son type.

2 : Unité de restauration : un ensemble d'actions qui sont ignorées lors d'une panne.

3 : Unité d'abandon : un ensemble d'actions qui sont ignorées sur décision de l'usager.

Nous différencions ces unités car elles correspondent à des besoins différents et sont associées à des mécanismes différents. Nous nous trouvons ainsi devant la difficulté de définir le mot transaction. Nous ne désirons pas redéfinir un terme bien connu du domaine, car ce serait introduire la plus grande confusion dans l'esprit du lecteur. Nous nous abstenons donc de donner une définition et nous admettons que la notion de transaction, telle qu'elle est définie de façon classique, ne trouve pas d'équivalent dans notre étude.

3.1. Unité d'admissibilité

Nous avons dans une étude précédente [FAU87] défini les différents types de règles que le système doit évaluer :

(1) Les règles d'admissibilité qui regroupe les règles **syntaxiques** sur les noms d'objet, de type, de version, etc .., les règles **morphologiques** des objets, des types, etc ..., et enfin les règles **structurelles** définies sur les objets et les versions d'objets.

Par exemple :

Le nom d'un type est une chaîne alphabétique dont la première lettre au moins est une majuscule est une **règle syntaxique.**

Un type construit est défini par un nom unique, un ensemble non vide d'attributs, un ensemble de liens et de contraintes, qui peut être vide est une règle **morphologique.** La plupart des règles morphologiques sont imposées par le dialogue. Par exemple, lors de la définition d'un type, le système demande au moins un attribut, et ensuite les ensembles de liens et de contraintes (qui peuvent être vides).

Les règles **structurelles** expriment par exemple, que si un attribut *a* est défini d'un type *E*, alors la version (ou l'objet) qui instancie *a* est de type *E*.

Les règles d'admissibilité sont **inviolables.** Elles sont vérifiées le plus tôt possible afin d'offrir le maximum d'interactivité avec le concepteur.

(2) les contraintes d'intégrité sont définies dans un type donné et indiquent si un objet (ou une version) est **cohérent(e) ou non.** Nous rappelons que les objets (ou les versions) incohérents(es) sont acceptés(es) et gérés(es) par le système, mais l'usager garde la responsabilité de cette incohérence. En effet, abandonner l'objet incohérent ou le modifier pour le rendre cohérent reste à sa charge.

Nous rappelons qu'une remise de vue (chapitre 3) n'est acceptée que si elle ne produit pas d'incohérence (les contraintes d'intégrité évaluables, doivent être vérifiées). Nous avons vu plusieurs stratégies de retrait pour remédier à cette situation. Il est nécessaire ici de ne pas identifier une remise de vue à une unité d'admissibilité. En effet, selon les cas, les mécanismes mis en jeu par le système sont différents, et plus complexes que l'annulation brutale de l'action qui a provoqué l'incohérence, ils ne rejettent pas forcément la vue qui provoque cette incohérence.

En résumé, nous dégageons un ensemble de règles inviolables (syntaxiques, morphologiques, et structurelles) qui sont vérifiées à chaque ligne de commande, et l'ensemble des contraintes d'intégrité (ou règles de cohérence), qui peuvent être évaluées à faux, mettant ainsi en évidence une incohérence pour une version d'objet (ou un objet).

Nous pouvons maintenant définir l'**unité d'admissibilité** : c'est un ensemble d'actions cohérentes selon les règles inviolables, par conséquent une unité d'admissibilité est une ligne de commande. Le système offre quand même la possibilité au concepteur d'ignorer une commande entière (voir figure 3).

```
DEF-TYPE Segment                <retour chariot>
 org : Point                    <rc>
 ext : Point                    <rc>
 lg : Entier                    <rc>
 .                              <rc>
 NOT confondues (org,ext)       <rc>

 ERREUR : le paramètre ext est inconnu     {message du système, indiquant }
 On (c)ontinue ou on (a)bandonne ?         {la violation d'une règle d'admissibilité }
 c              <rc>            {on peut choisir l'abandon de la commande}
 NOT confondues (org,ext)       <rc>
 lg := distance (org,ext)       <rc>
 FDEF                           <rc>
```

figure 3 : violation d'une règle d'admissibilité

La figure 3 retrace la définition du type construit Segment. Ce type admet trois attributs. Leur description est correcte (pas d'erreur syntaxique, Point et Entier sont des types connus dans la base). L'usager indique par un ".", qu'il a terminé la donnée des attributs.

La donnée de la contrainte d'intégrité provoque l'envoi d'un message par le système (le paramètre *ext* n'identifie aucun objet, ni aucun attribut). La système propose alors de continuer : l'usager reprend la définition du type en ignorant la contrainte erronée. Il est aussi possible d'abandonner la définition du type.

3.2. Unité de restauration

Dans le cas d'une panne, le système doit restaurer le maximum d'informations. Nous avons vu en effet dans le chapitre introductif, que le coût d'une action est élevé en termes d'efforts du concepteur et de temps machine (exécution de programme complexe par exemple).

Nous offrons un mécanisme de restauration, qui assure à l'usager de ne perdre dans le pire des cas, qu'une seule ligne de commande. Reprenons l'exemple précédent, décrivant la définition d'un type :

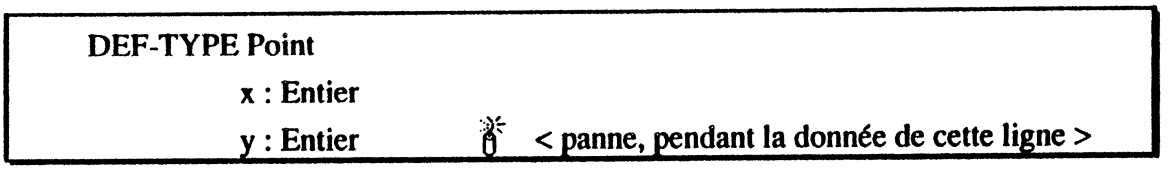


figure 4 : une panne pendant une définition de type

La seule action perdue dans ce cas est la donnée de la ligne décrivant *y* dans *Point.*

Au redémarrage du système, le type Point est proposé au concepteur (figure 5). Le concepteur doit décider s'il termine la définition du type, ou s'il l'abandonne.

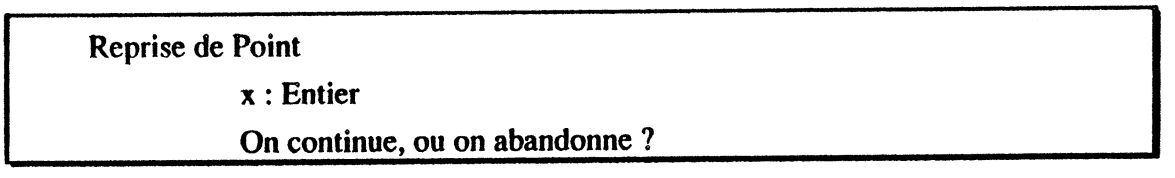


figure 5 : le reprise de la définition de Point

Le concepteur peut choisir de continuer et terminer la définition, ou d'abandonner, et la définition est définitivement ignorée.

Ce mécanisme propose une reprise au niveau de la ligne de commande. Dans certains cas, et sur **décision** de l'usager (choisir abandonner dans la figure 5), la commande entière peut être ignorée.

Pour la reprise d'une commande de création d'objet ou de version, le système peut donner au concepteur des informations sur la pertinence de l'objet (ou la version) restauré(e). La figure 6, illustre ce principe. Nous

montrons ci-dessous, le dialogue pour la création d'un objet. Nous utilisons pour cela, la définition du type Point donnée dans la figure 1.

```
CO Point                    {création d'un objet[1] dans Point}
    nom :                   { le système demande le nom de l'objet }
    p1
    x :                     {le système demande la donnée de l'abscisse du Point}
    10                      { le concepteur donne la valeur 10}
    y :
    0                       {le concepteur donne la valeur 0}
    autre commande ?        {la commande est terminée, on passe à autre chose}
```

figure 6 : la création d'un objet

Lorsque le concepteur voit apparaître le message *autre commande ?*, il est alors certain que le Point p1 est effectivement stocké dans sa base.

La figure suivante (figure 7) illustre l'occurrence d'une panne au cours de la donnée d'une ligne de commande :

```
CO Point
        nom :
        p2
        x :
        0
        y :    <panne du système>
```

figure 7 : une panne pendant la création d'un objet

L'objet p2 dans Point, en partie créé survit à la panne. A la reprise, le système calcule le degré de connaissance de p2 (en fonction de la définition du type Point). Le degré de connaissance est sensiblement différent du degré de complétude introduit au chapitre 4. En effet, il mesure

---

1 Pour la création d'une version, le principe de reprise s'applique d'une manière strictement identique.

le nombre d'attributs instanciés, par rapport au nombre d'attributs définis dans le type, et localement au type. Dans notre exemple :

- le nombre d'attributs du type est 2
- le nombre d'attributs connus dans p2 est 1

Le degré de connaissance de p2 dans Point, est par suite 1/2.

Si la panne intervient pendant l'évaluation d'un lien, ou d'une contrainte, le système reprend cette évaluation et termine les évaluations de tous les liens et toutes les contraintes. Néanmoins, l'objet est proposé au concepteur à la reprise, cohérent ou non, avec un degré de connaissance de 1 (tous les attributs sont connus). Ce mécanisme est nécessaire dans ce cas, car le concepteur peut ne pas savoir exactement, quelles sont les dommages provoqués par la panne sur l'objet en cours de création.

Le principe est appliqué pour la commande modification d'objet ou de version.

3.3. Unité d'abandon

Le concepteur doit pouvoir décider d'abandonner son travail. Il faut à cette occasion défaire les actions qui ont été exécutées.

Nous proposons pour cela la notion d'**unité d'abandon**, c'est à dire une séquence d'actions, qui commence par l'instruction **DEBUT**, et qui se termine par **FIN**, ces deux instructions peuvent être émises par le concepteur à n'importe quel moment.

Lorsque DEBUT n'est pas spécifié, l'unité d'abandon (par défaut) est la ligne de commande. La figure 8, illustre la définition d'une unité d'abandon.

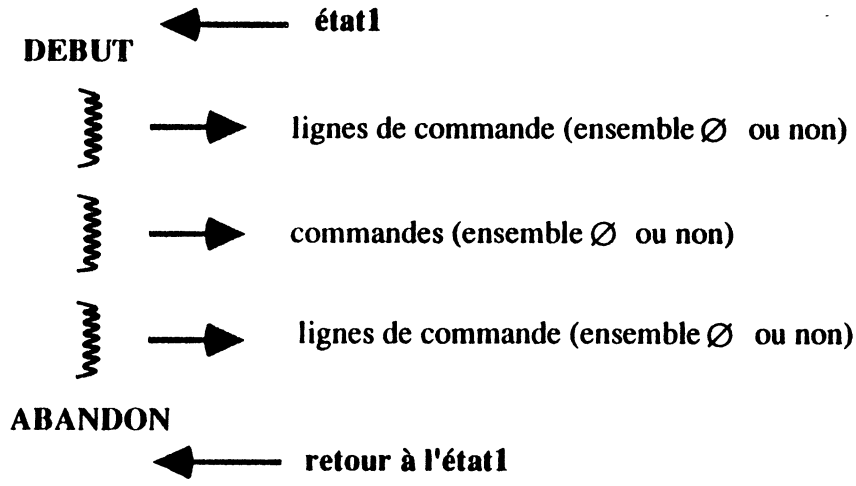


figure 8 : unité d'abandon

Lorsqu'aucun abandon n'est demandé et que le concepteur donne l'instruction FIN, toutes les actions sont prises en compte, le système se trouve alors dans un nouvel état.

La figure 9 montre l'imbrication des unités d'abandon. Cette fonctionnalité correspond aux notions de transactions imbriquées développées dans [MOS82], [HAR87], [BKK85], [WAL84], [GMS87]. Le principe présenté ici, ne met pas en jeu d'exécutions parallèles de transactions. En effet, la notion d'imbrication ne concerne que l'unité d'abandon.

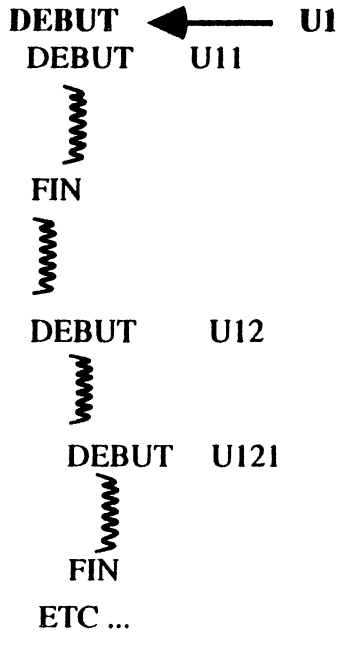


figure 9 : imbrication d'unités d'abandon

Une unité d'abandon peut être composée d'un ensemble quelconque d'unités d'abandon imbriquées. On réitère la même définition sur chaque unité imbriquée et ceci récursivement. Nous pouvons représenter cette imbrication par une arborescence (figure 10).

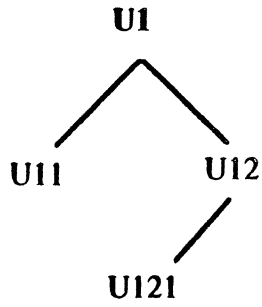


figure 10 : représentation arborescente de l'imbrication

Si une unité imbriquée est abandonnée, toutes ses unités filles (imbriquées) sont abandonnées.

Une unité termine, si toutes ses unités filles sont terminées, et si elle même n'est pas abandonnée.

En appliquant ce principe aux unités d'abandon de la figure 10 on a par exemple : si U_{12} abandonne, alors U_{121} est abandonnée. U_1 ne terminera que si U_{11} et U_{12} terminent sans abandonner. U_{12} terminera si toutes les unités qui lui sont imbriquées terminent.

4. LA CONCURRENCE

Nous ne développons dans cette section, qu'un certain nombre d'éléments pour la prise en compte de la concurrence d'accès aux entités[1] du système. Nous étudions ici, l'ensemble des informations (entités) qu'il est nécessaire de protéger lors d'accès concurrents. Puis, nous introduisons (section 4.2), les études futures à réaliser afin de proposer une solution acceptable au problème de la concurrence d'accès dans un environnement pour la CAO.

---

1 une entité est un objet, une version d'objet, un type, un sous type ou une fonction.

4.1. La Portée des Modifications

Il est nécessaire avant toutes choses, de décrire la portée des mises à jour effectuées lors de la manipulation d'une version d'objet, lors de la définition d'un type, d'un sous type ou d'une fonction, qui sont les seules actions qui peuvent entraîner des mises à jour en cascade (c'est à dire des modifications d'objets, de versions, de type, etc ...). Ces mises à jour en cascade sont décrites en détail au chapitre 6 sur la réalisation du serveur d'objets.

Contexte d'utilisation : toute entité du système possède un contexte d'utilisation, qui permet d'identifier toutes les entités qui l'utilisent. Cette notion a été introduite dans [RIE85], pour les objets seulement.

Le contexte d'utilisation d'une fonction contient les types et sous-types qui la référencent dans un lien ou une contrainte. Le contexte d'utilisation d'un type (ou d'un sous-type) contient les fonctions, les types et les sous-types qui l'utilisent, de même que les versions d'objets qui sont référencés par un de ses liens ou une de ses contraintes. Enfin, le contexte d'utilisation d'une version contient l'ensemble des versions qui l'utilisent, et dans quel but.

Contexte de création : le contexte de création n'est défini que pour les objets et les versions d'objet. Il donne :

- l'objet (ou la version) est complet ou incomplet.
- l'objet (ou la version) est cohérent ou incohérent.
- l'objet (ou la version) est calculé, par lien donné avec un certain ensemble de paramètres.
- la date de création (pour les versions seulement)
- l'auteur de la création
- etc ....

Nous pouvons maintenant définir la portée de chaque action possible :

(1) la création ou la définition d'une entité, modifie les contextes d'utilisation des entités qui la référencent, et lit les entités qu'elle utilise.

(2) la modification d'une entité entraîne :

- la modification du contexte d'utilisation des entités qu'elle utilise (il peut s'agir par exemple des versions d'objet qui la composent).

- la lecture des entités qu'elle utilise.
- la modification de son contexte de création
- la modification des entités de son contexte d'utilisation, et pour celles-ci récursivement jusqu'à obtenir des entités au contexte d'utilisation vide (voir figure 11).

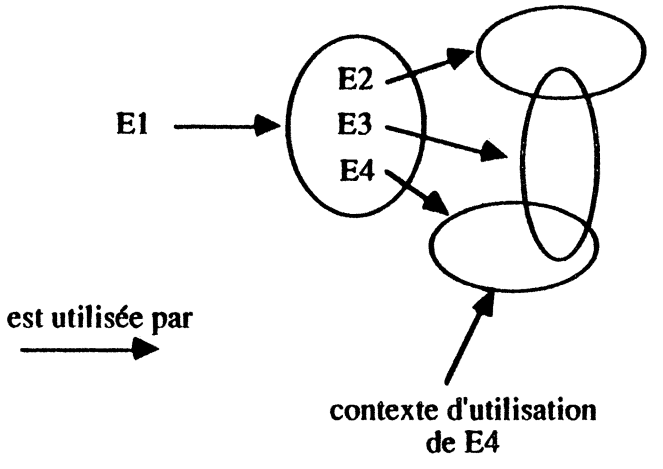


figure 11 : contextes d'utilisation

(3) la lecture ou la copie d'une version ou d'un objet, entraîne la lecture ou la copie des versions ou objets qui la composent, et ainsi de suite jusqu'aux objets de base.

Nous voyons ainsi que si l'on voulait appliquer des mécanismes de verrouillage, il faudrait poser des verrous en fonction du contexte d'utilisation des entités, ce qui est différent des SGBD classiques. En particulier, il nous semble impossible d'avoir en CAO, des mécanismes de verrouillage implicite des objets manipulés par une transaction.

4.2. Le Contrôle de la Concurrence

4.2.1. Dans les Systèmes Privés

Les nouvelles interfaces utilisateurs autorisées par les stations de travail telles que APOLLO, SUN, etc..., permettent aux utilisateurs de gérer plusieurs tâches en parallèle particulièrement grâce au multi-fenêtrage et à la gestion de plusieurs processus. Ce type de fonctionnalités implique donc

de proposer un système CAO qui autorise et gère les activités *parallèles* du **concepteur,** dans son environnement privé.

Deux approches sont possibles :

(1) redéfinir la notion de systèmes privés : un système privé est un système alloué à une fenêtre et une seule. Les mécanismes de définition, de remise et d'exécution de vues, de lecture et de déposition d'objets instables décrits dans les chapitres précédents s'appliquent de la même façon. Le système du concepteur devient alors semi-public, et chaque fenêtre attachée à un processus accède à un système privé. L'inconvénient de cette approche, appliquée au niveau d'un concepteur, est le manque de souplesse. La rigidité, bien que faible, introduite pour les besoins de gestion d'un projet, risque de gêner le concepteur et d'appauvrir les fonctionnalités offertes par le multi-fenêtrage.

(2) contrôler les accès aux données de façon plus classique, en utilisant les principes de verrouillage proposés dans les base de données dites traditionnelles avec intervention du concepteur.

La dernière approche paraît la plus satisfaisante. Nous désirons en effet offrir aux concepteurs un environnement le plus souple possible. Les concepteurs de façon générale disposent d'une certaine expertise dans leur domaine, il est alors intéressant de leur fournir des outils qui utilisent cette expertise. Dans cette optique, nous proposons un système de contrôle de la concurrence basé sur le verrouillage de données avec dialogue avec l'utilisateur. L'idée est de signaler l'accès concurrent au concepteur, et de lui proposer les solutions (possibles à cet instant) pour la résolution du conflit. Nous pensons qu'il n'est pas nécessaire de bloquer systématiquement l'accès à une donnée dans le cas de conflit. Il peut s'agir par exemple du cas où un des processus en concurrence émane d'une fenêtre illisible, et oubliée par le concepteur. Ce dernier est capable alors de prendre les mesures qu'il convient pour éliminer le conflit.

Nous n'aborderons pas cette étude dans le cadre cette thèse. En effet la résolution des conflits dans un environnement privé, avec coopération du concepteur, peut faire l'objet d'études futures prolongeant notre travail.

Nous supposons donc, que les systèmes privés sont mono-utilisateur, et surtout mono-tâche.

4.2.2. Dans les Systèmes Partagés

Le problème de la concurrence d'accès dans les environnements partagés se posent de manière presque identique à celui des systèmes privés. Par contre, ces problèmes sont soulevés ponctuellement, et plus rarement. De plus, le système peut, dans la majeure partie des cas, connaître exactement la nature du conflit : en fonctionnement normal, tous les accès sont pilotés par les commandes de manipulation des vues.

Là aussi, un travail plus approfondi est donc à faire.

Dans le cas du prototype ETIC, les commandes seront soumises en séquence aux systèmes partagés et ne poseront pas de problème de concurrence.

6. CONCLUSION

Nous avons proposé dans ce chapitre un certain nombre de mécanismes assurant les fonctions de contrôle de l'admissibilité, de reprise et d'abandon, basés sur trois concepts que nous avons définis :

- l'unité d'admissibilité
- l'unité d'abandon
- l'unité de reprise

Nous avons vu que ces mécanismes sont bien adaptés au système proposé. Leur mise en œuvre est décrite dans le chapitre suivant consacré au prototype ETIC.

Les problèmes de concurrence ont été abordés de façon succincte, mais nous avons donné un certain nombre d'éléments de réponse. Nous avons vu qu'il est possible de coupler les mécanismes de contrôle de la concurrence à un dialogue avec le concepteur.

Des travaux futurs doivent être effectués afin de proposer une solution plus générale à ce problème.

CHAPITRE 6
LA REALISATION, ETIC

table des matières

1. INTRODUCTION

Nous avons présenté dans ce document, un ensemble de concepts et de fonctionnalités nécessaires à la gestion d'un projet pour la CAO. Nous décrivons maintenant le prototype qui réalise et valide les idées émises au cours de notre étude.

Un prototype du système ETIC est actuellement opérationnel sur station APOLLO DN330, sous Unix BSD4.2. Une première version de ce prototype, réalisant le modèle décrit au chapitre 2, a été écrit au cours de l'étude [FAR87], [FAR87b]. Le système de gestion de projet ETIC, va en utiliser les fonctionnalités d'un serveur d'objets. L'idée est de développer le système de gestion de projet de versions le plus indépendant possible du serveur d'objets, afin de le ré-utiliser plus tard sur d'autres serveurs d'objets.

Dans la section 2, nous présentons l'architecture générale du système de gestion de projet dans un environnement centralisé.

Dans la section 3 nous décrivons le serveur d'objets, et dans la section 4, le serveur de versions.

La section 5 présente l'ensemble des mécanismes mis en œuvre, liés à la dynamicité des données des applications visées.

La section 6 est consacrée à la réalisation du système de Reprise Abandon et Concurrence.

Un outil d'aide a été développé ainsi qu'une interface utilisateur, dont nous présentons les fonctionnalités dans la section 7.

Enfin avant de conclure sur l'expérimentation réalisée, nous justifions dans la section 8, les choix effectués quant à la réalisation.

2. ARCHITECTURE

Le système ETIC est implanté sur un seul site. Nous avons vu que l'aspect réparti n'est pas pris en compte dans notre étude. Le code binaire du système existe donc en un seul exemplaire, et gère les accès aux bases organisées en arborescence (voir figure 1).

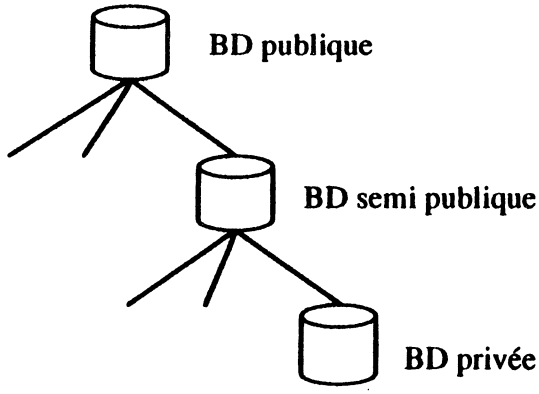


figure 1 : organisation des bases de données

Dans la figure précédente, la base publique, possède un ensemble de bases semi-publiques, qui chacune, possède un ensemble de bases privées. Nous avons adopté cette limitation pour des raisons de lisibilité, mais le nombre de bases d'une base donnée n'est pas limité. Chaque base contient les informations nécessaires à la gestion des bases qu'elle possède.

La figure suivante (figure 2) montre le système ETIC dans un environnement de projet. Nous retrouvons la base de données de connaissances (BDC) qui contient l'acquis de l'entreprise et qui n'est pas modifiable pendant le déroulement normal du projet. Les résultats obtenus pourront être intégrés dans le but d'utilisations futures. La base de programmes (BDP) contient les programmes qui réalisent les fonctions utilisées (ou potentiellement utilisables) pendant le projet. Ces deux bases sont partagées par tous les usagers du projet.

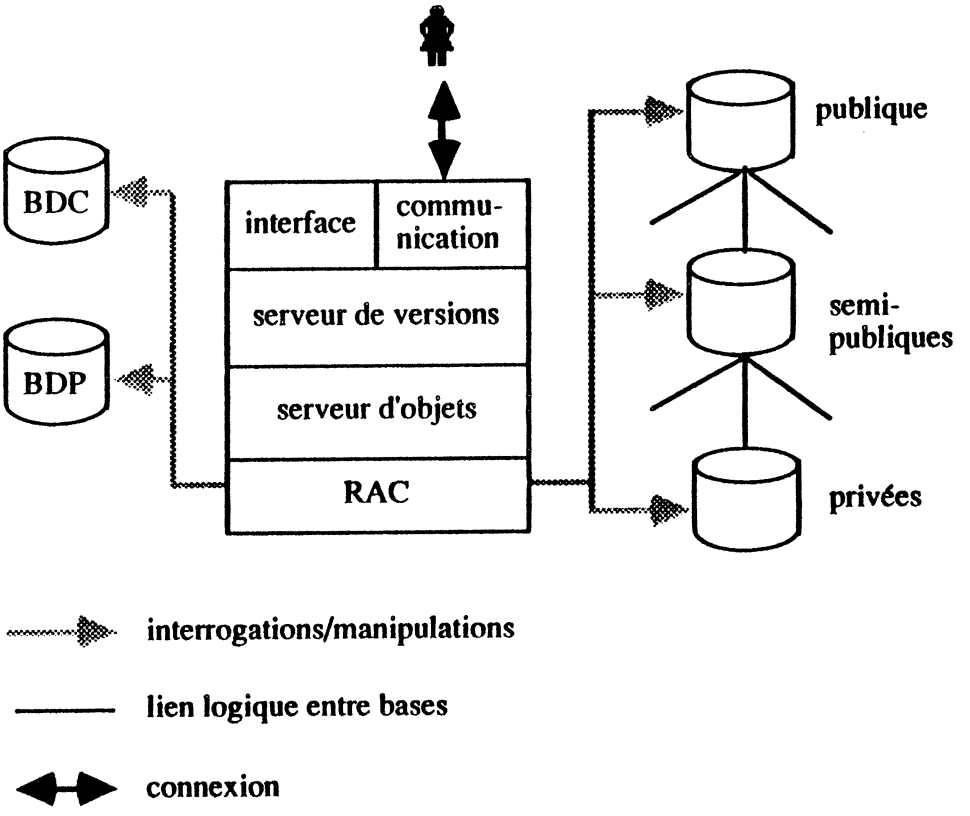


figure 2 : architecture de ETIC

Le système de **communication** gère les connexions des usagers et les échanges entre les bases (exécution et remise de vues, dépôt et lecture d'objets ou de versions).

L'**interface** gère les interactions entre l'usager et le serveur de versions (section 7).

Le **serveur de versions** (section 4) pilote le **serveur d'objets** (section 3) conformément aux besoins exprimés par l'usager.

Enfin, le système **RAC** (Reprise Abandon Concurrence, section 6) assure les mécanismes de reprise, d'abandon, de règlement des conflits d'accès, ainsi que les accès à la base de données correspondante.

3. LE SERVEUR D'OBJETS

Le serveur d'objets est le noyau du système ETIC. C'est lui qui assure la définition des types, des sous types et des fonctions, ainsi que la manipulation et l'interrogation des informations de la base (objets et versions d'objets). Nous présentons ici les fonctionnalités offertes aux concepteurs (définition des types §3.1., sous types §3.2. et fonctions §3.3).

L'usager accède à ces trois commandes à travers le serveur de versions, bien que celui-ci ne joue aucun rôle. Cette solution a été choisie pour des raisons de souplesse d'utilisation.

Remarque : chaque fonctionnalité est illustrée par le dialogue correspondant entre l'usager et le système. Les mots en caractères gras sont donnés par l'usager, ceux en italiques ne sont que des commentaires, et les autres sont ceux affichés par le système. Les exemples utilisés donnent une vision simplifiée à extrême de la construction navale (!).

Nous ne rappelons pas les caractéristiques du modèle de données mis en œuvre dans le système (voir chapitre 2 sur le modèle de données).

3.1. La définition des types

La définition d'un type est effectuée par la donnée d'attributs, de liens et de contraintes sur les attributs.

Environnement de définition de types :

nom : *le type est identifié par son nom*

Coque

attributs (. pour arrêter)

longueur : Décimal *ses attributs doivent prendre leurs valeurs*

largeur : Décimal *dans un type connu ou de base*

tirant-d'eau : Décimal

déplacement : Décimal

nb-volume : Entier

.

liens et contraintes sur les attributs (taper . pour arrêter)

déplacement := déplace (longueur,largeur) *la fonction déplace est définie par ailleurs*

.

Une coque est décrite, par sa longueur, sa largeur son tirant d'eau (voir le schéma donné dans l'annexe A), son déplacement et le nombre de volumes qui la composent (un volume pour un monocoque, deux volumes pour un catamaran, etc ..). Le déplacement de la coque peut être calculé par une fonction dont les paramètres sont la longueur, la largeur et le tirant d'eau.

Le type Coque ainsi défini peut être utilisé dans un type composé de plus haut niveau :

Environnement de définition de type	
nom :	
Bâtiment	
attributs (. pour arrêter)	
armateur : Chaîne	
mise-à-flot : Date	*Date est un type prédéfini*
port-attache : Chaîne	
flotteur : Coque	*Coque est défini par ailleurs (exemple précédent)*
propulseur : Chaîne	
usage : Chaîne	
.	

Un bâtiment est défini par son armateur, la date de sa mise à l'eau, son port d'attache, son flotteur qui est de type Coque préalablement défini, son type de propulseur (par exemple voiles, moteur, rames) et son usage (commercial, plaisance, répressif).

Le type Bâtiment n'admet ni lien ni contrainte. Date est un type structuré, prédéfini et décrit par un jour, un mois, une année qui prennent leurs valeurs dans le type Entier, et qui sont contraints de manière classique : le mois compris entre 1 et 12, le jour entre 1 et 31.

3.2. La définition des sous-types

La définition des types suivants illustrent l'utilisation du lien **ISA**, le sous-type *Yacht* spécialise le type *Bâtiment* :

Environnement de définition de type
nom :
Yacht ISA Bâtiment
attributs de spécialisation (. pour arrêter)
skipper : Chaîne
.
contraintes et (ou liens) de spécialisation
usage:='plaisance'
.

Le type ISA ainsi défini hérite de la définition de Bâtiment donnée plus haut. La spécialisation est donnée par un attribut supplémentaire (skipper) et par un lien de spécialisation qui associe à l'attribut usage défini dans Bâtiment la valeur 'plaisance'. Lors de la création dans Yacht ce lien agira comme un calcul, et lors de l'héritage il agira comme une contrainte forte. Ne seront héritées que les versions d'objets de Bâtiment dont l'usage est égal à plaisance. On aurait pu exprimer ce fait par une contrainte de spécialisation :

égal (usage,'plaisance')

Le type Bâtiment peut de même, être spécialisé comme suit :

Environnement de définition de type
nom :
Galère ISA Bâtiment
attributs supplémentaires (taper . pour arrêter)
garde-chiourme[1] : Chaîne
galériens : LIST Prisonnier *le type Prisonnier est défini par ailleurs*
nb-rameur : Entier
puissance : Entier
.
contraintes et (ou) liens de spécialisation
usage:='répressif'
puissance := rendement-rameur (nb-rameur,déplacement.flotteur)
. *la puissance de la Galère est calculée par la fonction rendement-rameur, avec le paramètre nb-rameur défini dans le type ISA, et le paramètre déplacement de flotteur défini dans le type Bâtiment spécialisé par Galère.*

Dans la définition précédente on a défini l'attribut galériens, comme une liste de valeurs dans Prisonnier.

3.3. La définition des fonctions

Nous avons vu que la définition d'un type peut mettre en jeu un certain nombre d'attributs calculés et de contraintes.

L'expression d'un attribut calculé ou d'une contrainte contient une fonction qui sera exécutée lors de l'évaluation de l'attribut ou de la contrainte à l'occasion de la création, ou de la modification d'une version. De plus, une fonction peut être utilisée pour le calcul de version (voir le calcul de flotteur-rapide dans Coque, donné section 4.1.1.5.).

Une fonction peut être, soit prédéfinie, (elle est alors définie dans la base de connaissances partagées du projet), soit définie par le concepteur lui-même (elle est alors définie dans la base privée du concepteur). Dans les deux cas, elle est réalisée par un programme de la base de programmes du projet. Ceux-ci sont écrits en FORTRAN, PASCAL, C.... L'idée est de les réutiliser en ne réécrivant que l'interface.

---

1 chiourme : ensemble de rameurs d'une galère, de forçats [ROB88].

Environnement de définition de fonction
nom de la fonction :
dilate
algorithme (retour si identique nom de fonction) :
<retour>
types de départ (produit cartésien *, ou LIST) :
Coque * Entier * Entier
type d'arrivée :
Coque *fin de définition de la fonction*

La fonction dilate admet en entrée une version de coque et deux entiers. Le premier entier est le coefficient de dilatation en longueur, et le suivant celui en largeur. Le type de la valeur délivrée est une coque.

Le concepteur peut définir une fonction, à condition que celle-ci référence un algorithme connu du système, c'est à dire existant dans la base de programmes. Dans le cas où les ensembles de départ ou d'arrivée, n'existent pas dans la base, la fonction est acceptée, mais est inutilisable (elle est marquée NON-VALIDE).

Dans le cas contraire, il doit y avoir cohérence entre le point d'entrée de l'algorithme, et les ensembles référencés (si la cohérence est vérifiée, la fonction est acceptée et elle est marquée VALIDE). Une fonction n'est utilisable que lorsqu'elle est VALIDE.

Par exemple, si la fonction référence le type Coque, le système doit reconnaître dans l'activant de l'algorithme la structure d'une Coque (c'est à dire 5 entiers).

3.4. La manipulation des objets

Nous n'autorisons le concepteur à manipuler un objet qu'à travers une de ses versions. Ainsi un objet est créé, lorsqu'une version est créée (voir création de version § 4.1.1 de ce chapitre). De même un objet est supprimé lorsque toutes ses versions sont abandonnées (voir abandon de version § 4.1.5). La manipulation d'objets est donc pilotée par le serveur de versions dont nous décrivons les fonctionnalités dans la section suivante. Nous insisterons à chaque fois, sur le rôle tenu par le serveur de versions, et celui tenu par le serveur d'objets.

4. LE SERVEUR DE VERSIONS

Le concepteur n'interagit sur les objets que via le serveur de versions. En effet, le serveur de versions pilote le serveur d'objets en fonction des besoins du concepteur. Nous décrivons dans cette section l'ensembles des fonctionnalités mise à sa disposition.

4.1. La manipulation des versions

4.1.1. Création

La section 3 précédente a présenté la définition des types, nous allons voir maintenant les différentes alternatives pour créer et manipuler des versions d'objets dans les types, c'est à dire les différentes méthodes de conception offertes par ETIC.

La création d'une version correspond au démarrage d'un processus de conception (l'objet correspondant n'existe pas) ou d'un nouvel axe de conception pour un objet existant.

Le serveur de versions assure la saisie du nom d'une version pour un objet donné, dans un type donné, puis passe ces trois paramètres au serveur d'objets qui réalise véritablement la création.

4.1.1.1. La conception ascendante

La méthode de conception ascendante consiste à construire des versions, puis à les utiliser pour composer des versions d'objets de plus haut niveau. Cette composition s'exprime ici en affectant à un attribut le nom d'une version.

Environnement de création de version		
dans quel type ? :		
Bâtiment		
pour un objet	(n)ouveau	--> *pour démarrer un processus de conception*
	(e)xistant ? :	--> *pour démarrer un nouvel arbre de dérivation*
n		*d'un objet donné*
nom de l'objet:		
vagabond		
nom de la version :		*le nom d'une version est unique pour un objet*
v1	*donné*	
armateur :		
'Marie-Christine'		*les objets de base n'admettent pas de versions*
mise-à-flot :		
v1(mise-à-lo-vagabond)		*la date de mise à l'eau est instanciée par une*
flotteur :		*version v1 de l'objet mise-à-lo vagabond*
		préalablement créée
v1(flotteur-vagabond)		*même remarque pour le flotteur*
propulseur :		
'voiles'		
usage :		
'plaisance'		*fin de création de la version*

Toute affectation d'une version à un attribut crée un lien d'**utilisation** entre la version désignée, et celle que l'on construit. L'ensemble des liens d'utilisation d'une version définit ainsi son **contexte d'utilisation.** Cette information est maintenue par le système tout au long de la vie de chaque version. Elle est exploitée lorsqu'une version est modifiée : on connaît ainsi toutes les versions d'objets sur lesquelles il faut répercuter ces modifications.

4.1.1.2. La création imbriquée

L'idée est de permettre au concepteur de créer des versions pour l'instanciation d'attributs *(flotteur)* lors de la création d'une version *(v1(vagabond2))*.

```
Environnement de création de version
  dans quel type ? :
  Bâtiment
  pour un objet   (n)ouveau
                  (e)xistant ? :
  n
  nom de l'objet:
  vagabond2
  nom de la version :              le nom d'une version est unique pour un objet
  v1                               donné
  armateur :
  'Marie-Christine'
  date :
  v1(mise-à-lo-vagabond)           la date de mise à l'eau est instanciée par un objet
  flotteur :                       préalablement créé
  v1(flotteur-vagabond2)
  v1(flotteur-vagabond2) inconnue, création de v1(flotteur-vagabond2) dans Coque :
     longueur :
     12
     largeur :
     4,5
     tirant-d'eau :
     1,7
     nbre-de-volume :              l'attribut déplacement est automatiquement
                                   calculé
     1                             le lien défini dans le type Coque
                                   fin de création de version
  propulseur :
  'voiles'
  usage :
  'plaisance'                      fin de création de version
```

Cet exemple illustre le cas où l'objet *flotteur-vagabond2* n'existe pas dans la base. On crée alors, à la fois la version et l'objet. L'exemple suivant montre la création d'une version pour un objet existant :

Environnement de création de version
dans quel type ? :
Bâtiment
pour un objet (n)ouveau
(e)xistant ? :
e
nom de l'objet:
vagabond2 *celui-ci doit exister dans la base dans Bâtiment*
nom de la version :
v1
1 --> duplication d'une version existante *on peut initialiser la nouvelle*
2 --> création à partir de rien *version à l'aide d'une autre, puis*
2 *modifier certaines valeurs des*
armateur : *attributs*
'Marie-Christine'
date :
v1(mise-à-lo-vagabond)
flotteur :
v2(flotteur-vagabond2)
la version v2 de flotteur-vagabond2 est inconnue,
1 --> création de v2(flotteur-vagabond2) dans Coque
2 --> choix d'une autre version de flotteur-vagabond2 dans Coque
1 *dans le cas où 2 est l'option choisie, le système*
longueur : *propose l'ensemble des versions de flotteur-*
12 *vagabond2, parmi lesquelles l'usager doit choisir*
largeur :
4,5
tirant-d'eau :
1,7
nbre-de-volume :
1
fin de création de version
propulseur :
'voiles'
usage :
'plaisance' *fin de création de version*

Cette méthode de conception reste cependant ascendante, en effet, elle est strictement équivalente à la méthode de conception qui consiste à créer *v2(flotteur-vagabond2)*, puis à la référencer dans *v1(vagabond2)*.

4.1.1.3. La conception descendante

```
Environnement de création de version
  dans quel type ? :
  Bâtiment
  nom de l'objet :
  vagabond3
  nom de la version :
  v1
  armateur :
  'Marie-Christine'
  date :
  /
      création imbriquée système dans Date
      jour :
      21
      mois :
      10
      année :
      1960                          fin de création d'objet
  flotteur :
  v2(flotteur-vagabond2)
  propulseur :
  'voiles'
  usage :
  'plaisance'     fin de création de version
```

Cet exemple illustre l'utilisation de la conception descendante par l'instanciation de date. Un objet système est créé pour instancier date de *v1(vagabond3)*; si *v1(vagabond3)* est supprimé cet objet l'est aussi : il n'existe qu'en temps que date de *v1(vagabond3)*. Un tel objet est appelé objet système, et n'admet aucune version. Il est inaccessible par l'usager autrement que comme valeur d'un attribut d'une version d'un objet donné.

4.1.1.4. Attribut calculé

Cette méthode d'instanciation offre au concepteur un mécanisme automatique d'instanciation d'un attribut pour une version donnée.

Environnement de création de version
dans quel type ? :
Bâtiment
nom de l'objet :
vagabond4
nom de la version :
v1
armateur :
'Marie-Christine'
date :
v1(mise-à-lo-vagabond)
flotteur :
/
création imbriquée système dans Coque :
dilate (v1(flotteur-vagabond).Coque,0.5,2) *la longueur (respectivement la largeur) de la coque de vagabond3, est la longueur (respectivement la largeur) de flotteur-vagabond augmentée de 2 unités (respectivement de 0.5 unités)*
propulseur :
'voiles'
usage :
'plaisance' *fin de création de version*

Le flotteur de *v1(vagabond4)* est calculé par la fonction dilate. Le paramètre *v1(flotteur-vagabond).Coque* est une version dans Coque. Si celle-ci est modifiée, alors le calcul est ré-effectué et la version *v1(vagabond4)* est modifiée. La fonction dilate utilisée pour le calcul du flotteur de *v1(vagabond4)* doit être préalablement définie.

4.1.1.5. Version calculée

Le système offre au concepteur la possibilité de calculer une version à partie d'une, ou de plusieurs autres, à l'aide d'une fonction qu'il

aura préalablement définie. Les versions sont alors liées, et si un des paramètres de la fonction est modifié, alors celle-ci est ré-exécutée.

```
Environnement de création de version
  dans quel type :
  Coque
  nom de l'objet :
  flotteur-rapide
  nom de la version :
  v1
  F (o-n) ?                          par évaluation d'une fonction, oui ou non ?
  o
  dilate (v1(flotteur-vagabond).Coque,0.5,3)      fin de création de version
```

4.1.1.6. Objets Inachevés

Le concepteur peut, s'il le désire, ne pas instancier certains attributs (voire tous) d'une version :

```
Environnement de création de version
  dans quel type ? :
  Bâtiment
  nom de l'objet :
  vagabond5
  nom de la version :
  v1
  armateur :
  'Marie-Christine'
  date :
  mise-à-lo-vagabond
  flotteur :
      <rc>
  propulseur :
  'voiles'
  usage :
  'plaisance'     fin de création de version
```

Ce mécanisme permet au concepteur d'exprimer une connaissance imparfaite des versions d'objets. Le système marque *v1(vagabond5)* **incomplète.**

Remarque : Nous n'avons défini aucune algèbre sur les valeurs nulles. Le système reconnaît et traite d'une façon particulière les attributs inconnus des versions .

4.1.1.7. La création dans un sous-type

Dans l'exemple suivant, nous montre la création d'une version dans le type *Galère* qui spécialise le type *Bâtiment.*

Environnement de création de version
dans quel type ? :
Galère
nom de l'objet :
la-thèse
nom de la version :
v1
armateur : *attributs hérités*
'lgi'
date :
/
jour :
12
mois :
9
année :
1988 *fin de création d'objet*

suite de la page précédente

flotteur :
flotteur.v1(vagabond).Bâtiment
propulseur :
'rameurs'
garde-chiourme : *attributs de spécialisation*
'Michel'
nbr-rameurs
1
galériens attribut LIST, taper . pour arrêter
galériens1 :
'Marie-Christine'
galériens2 :
.

fin de création de version

Nous pouvons faire plusieurs remarques quant à cet exemple :

- l'attribut *usage* est maintenant automatiquement instancié par le lien défini dans *Galère*.

- le système demande dans l'ordre, d'instancier les attributs des types de plus haut niveau, puis du sous-type dans lequel est effectuée la création.

- l'attribut flotteur est instancié par la désignation hiérarchique de la valeur de l'attribut flotteur de la version *v1* de *vagabond* dans *Bâtiment (flotteur.v1(vagabond).Bâtiment)*. Si cette valeur est modifiée dans *v1(vagabond)* alors *v1(la-thèse)* est modifiée.

4.1.2. Dérivation

La dérivation d'une version, est effectuée pour figer une version (celle qui va être sélectionnée), et en produire une nouvelle. Cette dernière devient la version en cours. S'il existait au préalable une version en cours, elle est alors figée.

Environnement de dérivation de version :
dans quel type :
Bâtiment
pour quel objet :
vagabond
expression de sélection d'une version :
on sélectionne une seule version
nom de la version dérivée :
v6 *fin de dérivation*

L'opération de sélection d'une version, est détaillée dans la section 4.2. de ce chapitre. La version *v6* de *vagabond* que l'on vient de générer par dérivation, est donc copiée à partir de la version sélectionnée, et devient la version en cours que le concepteur peut modifier.

4.1.3. Gel et Dégel

Le dégel porte sur une version feuille de l'arbre de dérivation des versions d'un objet. La version sélectionnée devient la version en cours peut alors être modifiée. Nous rappelons que le dégel d'une version non feuille, c'est à dire qui admet des versions dérivées, est interdit.

Environnement de dégel d'une version :
dans quel type :
Bâtiment
pour quel objet :
vagabond
expression de sélection d'une version :
on sélectionne une seule version : v1
la version v1 est la version en cours *fin de dégel*

L'opération de gel ne peut porter que sur la version en cours qui est unique. Celle-ci est donc gelée et ne peut pas être modifiée.

4.1.4. Modification de la version en cours

Une modification ne peut porter que sur des versions d'objets de types structurés ou ISA. Le concepteur ne peut modifier que les versions

d'objets utilisateurs c'est à dire, ceux qu'il a explicitement créés. Il lui est impossible de modifier un objet sans version, créé à l'occasion d'une création imbriquée système par exemple. Le système prend en charge la répercussion automatique des modifications.

```
Environnement de manipulation de versions :
  modification de la version en cours :
  la version v1 de vagabond a été créée par valeur dans Bâtiment :
  F (O, N, <retour>) :
  N
  armateur : 'Marie-Christine'          l'ancienne valeur est 'Marie-Christine'
  <retour>                              pas de modification de valeur
  mise-à-flot : v1(mise-à-lo-vagabond)
  xxx                                   la date est maintenant inconnue
  flotteur : v1(flotteur-vagabond)
  v2(flotteur-vagabond)                 la valeur pour flotteur est modifiée
  La version v2 de flotteur-vagabond n'existe pas dans Coque :
                les mêmes fonctionnalités que pour la création de version sont
                applicables ici (attribut calculé, création imbriquée, création de
version
                etc..)
  propulseur : 'voiles'
     <retour>
  usage : 'plaisance'
     <retour>
```

La version est modifiée : on recalcule si besoin ses attributs calculés, et on ré-évalue si besoin les contraintes définies dans le type.

Les répercussions provoquées par une modification sont décrites en détail dans la section 5 de ce chapitre.

4.1.5. Abandon

L'abandon d'une version exprime la volonté d'abandonner un axe de conception dans l'arbre de dérivation d'un objet donné. Une version est sélectionnée, puis elle est supprimée ainsi que toutes ses versions dérivées.

Cette fonctionnalité pose problème car une version peut être utilisée par de nombreuses autres versions (instanciation d'attributs, calculs, désignation etc ...). Plusieurs solutions sont envisageables pour les versions utilisatrices d'une version supprimée :

(1) l'instanciation ou le calcul, etc .. est annulé, la version utilisatrice devient incomplète. Si celle-ci était calculée, son statut devient difficilement définissable.

(2) la version supprimée est remplacée par une version qui lui est la plus proche possible : par exemple, sa version mère dans son arbre de dérivation.

(3) les liens créés par l'instanciation, le calcul ou la désignation sont rompus, et il n'y a pas de modifications apportées aux versions utilisatrices.

Ces trois solutions sont acceptables et peuvent être mises en œuvre : la rupture du lien (3) est choisie, au moment de l'instanciation, du calcul ou de la désignation, le remplacement (2) est choisi au moment de l'abandon, le contexte d'utilisation de la version abandonnée n'est pas vide. Si ni (2) ni (3) ne sont applicables alors le principe (1) est mis en œuvre.

Environnement de manipulation de versions
abandon de versions :
dans quel type ? :
Bâtiment
nom de l'objet :
vagabond2
expressions de sélection de une version
on sélectionne une seule version : v3
fin d'abandon

La version *v3* de *vagabond*, et toutes ses versions dérivées sont abandonnées. Les répercussions mises en œuvre pas ces abandons sont décrites en détail dans la section 5.

4.2. L'interrogation des versions

4.2.1. La langage de sélection

Nous décrivons dans cette section les différents moyens disponibles pour la sélection des versions. Ceux-ci sont utilisés pour la consultation d'une part, et d'autre part lors du choix de la version à dégeler, à dériver, ou à abandonner.

Le langage de sélection est un langage fonctionnel sans variable, utilisant des fonctions booléennes et les opérateurs booléens OU, ET et NOT, mais sans parenthèse. Par exemple :

```
dans quel type :
  Bâtiment
  sélection de version :
  état(coh) ou not inf(long.flotteur,larg.flotteur)
  il y a trois versions sélectionnées, voulez-vous :
      (a)ffiner la sélection
      (i)gnorer la sélection
      (c)onsulter les versions sélectionnées
  ...
```

L'expression précédente sélectionne, toutes les versions d'objets dans Bâtiment qui sont cohérentes *(état(coh))* et dont la coqué est plus longue que large *(not inf(long.flotteur,larg.flotteur))*. Il est possible ensuite de diminuer le nombre de versions sélectionnées en affinant la sélection, de revenir à la sélection précédente en ignorant la sélection que l'on vient d'exécuter, ou de consulter les versions sélectionnées.

Il existe plusieurs types de fonctions utilisables dans l'expression d'une fonction :

- les fonctions booléennes définies par l'usager dans sa base privée. Ceci permet au concepteur d'étendre le langage de sélection selon ses besoins, et selon le type d'application visé. En effet, on n'utilisera pas les mêmes fonctions de sélection dans une application CAO de VLSI, et dans une application CAO d'architecture.

- les fonctions booléennes prédéfinies : elles concernent particulièrement la manipulation des dates :

jour (date,10) :	versions dont l'attribut jour de date est égal à 10
mois (date,12) :	idem, mais sur le mois.
année (date,1960) :	idem mais sur l'année
depuis (date,1960-10-21) :	version dont l'attribut date représente une date postérieure au 21 octobre 1960.
avant (date,1960-10-21) :	idem, mais la date est antérieure
pendant (date,1960-10-21,1988-10-21) :	version dont la date est dans l'intervalle donné.

Dans les exemples précédents, l'attribut date doit être défini de type Date, qui est un type prédéfini admettant les attributs jour, mois et année de type Entier. Ces derniers sont soumis aux contraintes d'intégrité classiques sur les dates (mois inférieur à 12, etc ..).

Les fonctions classiques de comparaison sont également prédéfinies (sup, inf, égal, ...).

- les fonctions booléennes portant sur l'état des versions :

dérivée (v1(vagabond2)) :	versions dérivées de v1 de vagabond2
alternative (v1(vagabond2)) :	versions alternatives
précède (v1(vagabond2)) :	versions ancêtres (enfin !)
état(coh) :	versions cohérentes
état(incoh) :	versions incohérentes
état(com) :	versions complètes
état(incom) :	versions incomplètes
degré (com,1) :	versions dont le degré de complétude est 1
degré (coh,1) :	versions dont le degré de cohérence est 1
nom(v1(vagabond2)) :	versions dont le nom est v1 de vagabond2

Remarque : les degrés de complétude et de cohérence (définis dans la section 3.2. du chapitre 4) sont calculés au moment de l'évaluation de la fonction degré, alors que le fait que la version soit cohérente ou complète (localement) est stocké.

4.2.2. La définition des classes d'équivalence

Le prototype accepte la définition des classes d'équivalence sur un type donné, telle que nous l'avons décrite dans la section 3.3 du chapitre 4. Le système offre pour cela deux fonctionnalités :

- la **définition** proprement dite d'une relation d'équivalence : c'est à dire la définition de classes par l'affectation d'un ensemble de règles à chacune des classes. Nous rappelons qu'une classe regroupe les versions d'un type donné, qui vérifient toutes les règles associées à la classe. Une version vérifie une règle, si celle-ci est évaluable et évaluée à vrai pour la version. Cette définition est différente de celle que nous avons donné pour l'évaluation des contraintes d'intégrité. En effet, le but ici est de caractériser des versions en fin de conception, on peut donc supposer que les versions incomplètes n'intéressent à priori pas l'usager dans ce type de manipulation.

- la **mise à niveau** d'une relation d'équivalence : c'est à dire, la tentative d'affectation de toutes les versions d'un type donné, à toutes les classes d'une relation donnée. Cette opération est effectuée à la demande, car elle est coûteuse en temps du fait du nombre de règles à vérifier et de leur complexité.

L'exemple suivant en montre l'utilisation :

```
Définition d'une relation d'équivalence
  dans quel type ?
  Bâtiment
  nom de la relation
  taille                          on peut définir plusieurs relations dans un type donné
  génération des classes en mode
      (a)utomatique
      (m)anuel
  a                               on choisit la génération automatique des classes
  donner chaque règle terminée par un <retour> Taper . pour arrêter
  inf (long.flotteur,10)          il doit y avoir compatibilité entre les fonctions
  inf (larg.flotteur,2)           et les paramètres utilisés
  .
```

Chaque règle utilise une ou plusieurs fonctions prises parmi celles décrites dans la section précédente, et connectées par OU et ET, et

préfixées ou non par NOT. Les classes générées par le mode automatique sont pour cet exemple :

{} qui contiendra les versions ne vérifiant ni 1 ni 2 ;
{1} qui contiendra les versions vérifiant 1 ;
{2} qui contiendra les versions vérifiant 2 ;
{1,2} qui contiendra les versions vérifiant 1 et 2.

L'usager peut utiliser le mode manuel de génération des classes : il affecte à chaque classe proposée par le système une ou plusieurs règles.

```
Définition d'une relation d'équivalence
  dans quel type ?
  Bâtiment
  nom de la relation
  taille1
  génération des classes en mode
      (a)utomatique
      (m)anuel
  m
  classe1 : donner chaque règle terminée par un <retour> Taper . pour arrêter
  inf (long.flotteur,10)
  inf (larg.flotteur,2)
  .
  classe2 : donner chaque règle terminée par un <retour> Taper . pour arrêter
  inf (long.flotteur,10)
  .
  classe3 : donner chaque règle terminée par un <retour> Taper . pour arrêter
  inf (larg.flotteur,2)
  .
  classe4 : donner chaque règle terminée par un <retour> Taper . pour arrêter
  .               la classe 4, contiendra les versions qui ne seront dans aucune des
                  autres
```

La relation *taille1* définie ici de façon manuelle, est identique à la relation *taille* définie précédemment.

4.2.3. L'interrogation dans les classes d'équivalence

L'exemple suivant montre le processus d'interrogation dans les classes d'équivalence telles qu'elles ont été définies dans la section précédente :

```
Interrogation des classes d'équivalence
  dans quel type
  Bâtiment
  pour quelle relation
  taille
  la date de la dernière mise à niveau est 1988-8-19-20:18  la dernière remise à niveau
  remise à niveau (o-n) :                                   été faite le 19 août 1988 à 20h18
  o
  Les règles suivantes ont été utilisées pour la définition des classes :
  1 inf (long.flotteur,10)
  2 inf (larg.flotteur,2)
  La classe {} contient les versions :
  v1 de vagabond2                                           v1 de vagabond2 ne vérifie aucune règle
  La classe {1} contient les versions :
  v2 de vagabond2
  La classe {2} contient les versions :
  v2 de vagabond2, v1 de vagabond4                          v1 de vagabond4 ne vérifie que la règle 2
  La classe {1,2} contient les versions :
  v2 de vagabond2                                           v2 de vagabond2 vérifie les deux règles
```

L'exemple précédent montre la consultation des classes d'une relation d'équivalence, avec la demande d'une remise à niveau qui est facultative.

4.3. Le contrôle des versions

Nous avons vu dans le chapitre 4, qu'il est nécessaire de contrôler le nombre de versions générées dans un type donné. Il faut assurer un nombre minimal de versions, afin d'avoir une trace significative, et un nombre maximal afin que le concepteur ne soit pas noyé dans une masse de versions.

Nous permettons donc, de définir des contrôles qui obligent la génération d'une version, et d'autres qui l'interdisent. Ces contrôles sont réalisés par des règles définies par l'administrateur du projet. L'usager, peut dans son environnement privé définir ses propres contrôles.

La génération d'une version signifie qu'on dérive une nouvelle version, à partir de celle qui est en cours (voir dérivation de version). S'il n'existe aucune version en cours, tous les contrôles sont différés jusqu'au dégel, la dérivation ou la création d'une version (actions qui produisent une version en cours).

4.3.1. L'obligation

Les contrôles d'obligation permettent de générer suffisamment de versions pour avoir une trace significative. Les règles qui traduisent ce type de contrôle sont de deux types :

- **fréquence** : par exemple, pour exprimer qu'il est nécessaire de générer au moins une version par jour, on écrira :

fréquence-max (Bâtiment,jour,1)

Le serveur de version oblige ainsi la génération d'au moins une version par jour, pour le type *Bâtiment.*

- **évolution** : ce type de règles permet de traduire la nécessité de générer une nouvelle version, dès qu'elle change de **classe** d'équivalence, d'**état** (elle devient complète par exemple), ou de **contexte** (elle était créée par valeur, maintenant elle est calculée). Par exemple, pour exprimer qu'une version d'un objet de *Bâtiment* doit être générée lorsqu'elle change de classe pour la relation *taille* , on écrira :

évolution(Bâtiment,relation(taille))

Le système génère automatiquement une version lorsqu'une des règles définies pour l'obligation est vérifiée (c'est à dire évaluable, et évaluée à vrai).

4.3.2. Autorisation

Les contrôles traduisant l'autorisation (ou l'interdiction) de générer une version sont de deux types :

- la **fréquence** : les règles qui permettent d'exprimer par exemple l'interdiction de créer plus de 2 versions dans une semaine :

fréquence-min (Bâtiment,semaine,2)

- l'**acceptabilité** : on désire interdire dans ce cas le gel de version ne vérifiant pas un ensemble de critère de qualité, chaque critère étant traduit par une règle. Par exemple :

rapide (Bâtiment)

qui vérifie qu'un navire est *rapide*, en fonction de sa coque, et la puissance de son propulseur.

Remarque : dans la version actuelle du prototype, le contrôle des versions n'est pas implanté. Cette réalisation sera effective à très court terme car les mécanismes de saisie et d'exécution des règles existent déjà.

5. LA DYNAMICITE

Nous présentons dans cette section toutes les opérations qui traduisent la dynamicité des objets manipulées dans les applications CAO. Celles-ci peuvent être explicitement demandées par le concepteur, ou exécutées par le système, dans le but de maintenir la cohérence de la base de données.

5.1. Répercussions de la définition d'un type

Les actions effectuées dans ce cadre sont entièrement prises en charge par le système.

5.1.1. sur les fonctions

La définition d'un type, peut entraîner la validation d'une fonction. Celle-ci devient alors utilisable. Dans le cas où la cohérence entre l'activant de l'algorithme et la définition de la fonction n'est pas vérifiée, il faut, soit détruire ou renommer le type, soit détruire ou renommer la fonction.

5.1.2. sur les versions d'objets

Dans le cas de la définition d'un type ISA, les versions d'objets existant déjà dans la base, et cohérentes dans le sur-type du type que l'on vient de définir, doivent être héritées, si possible, dans ce dernier. Le

mécanisme d'héritage décrit dans la section 7 du chapitre 2 est mis en œuvre de même que celui d'insertion dans les sur-types du nouveau type ISA.

5.2. Répercussions de la création d'un objet

L'objet est enregistré dans la base, une première version lui est affecté, dans le cas d'un objet *sans version* (c'est à dire créé par le système) celle-ci restera unique. Bien qu'un tel objet système soit sensé être un objet sans version, nous avons choisi cette solution, pour gérer des informations de structure homogène. Ainsi un objet système admet une version unique, et est reconnaissable car son nom n'est pas instancié.

- son **identification** : nom et surrogate (numéro affecté et géré par le système). Le nom des objets est unique dans le type donné, il est affecté par le concepteur à la création de l'objet, il peut être modifié. L'utilisateur ne peut accéder à un objet que par son nom. Un objet système ne porte pas de nom, il est identifié par le système, par son surrogate.

- la **liste de ses versions** : à chaque objet est associée la liste de ses versions. Cette liste est ordonnée, et donne le nom des versions dans l'ordre chronologique dans lequel elles ont été créées.

5.3. Répercussions de la création d'une version

Une version est associée à un objet, son nom appartient donc à la liste des versions de celui-ci.

Les informations générées à la création d'une version concernent :

- sa **filiation** : sa version mère

- ses **propriétés** : cohérence, complétude, mode de création et éventuellement identificateur du lien qui en a permis le calcul, la date de création, le nom du créateur, etc ... Ceci décrit le **contexte de création** de la version.

- ses **attributs** : pour chaque attribut, son nom, le nom de la version, de l'objet qui l'instancie, le mode de l'instanciation (par lien fonctionnel, de désignation, par nom etc ..), et éventuellement le numéro du lien qui en a permis l'instanciation. Le couple (nom de version, nom d'objet) permet d'identifier une version de façon unique, pour cet objet dans un type donné.

Si la version est cohérente dans le type où elle a été créée, le système tente son héritage dans les types ISA, s'ils existent. Si la version est créée dans un sous-type, elle est insérée dans tous les sur-types de celui-ci. Ces deux mécanismes sont décrits dans le chapitre 2.

Le système met à jour le **contexte d'utilisation** des versions qui ont été référencées pendant le processus de création.

On trouvera en annexe C, un exemple d'informations générées par une session dans ETIC, ainsi que leur signification.

5.4. Répercussions de la modification d'une version

5.4.1. Sur la version elle-même

Le contexte de création de la version peut avoir été modifié. Par exemple, la version peut avoir été créée par valeur, puis modifiée et calculée.

Les attributs calculés sont ré-évalués si nécessaire, et de même pour les contraintes. Le schéma d'héritage de la version dans les types spécialisés est recalculé. Par exemple, une version qui devient incohérente suite à une modification, n'est plus héritée dans aucun des types spécialisés du type où elle a été créée.

5.4.2. sur les autres versions

Grâce aux contextes d'utilisation de la version, le système retrouve chaque version affectée par les modifications. Pour chacune d'entre-elles, il répercute la modification, et recalcule s'il y a lieu les contraintes et les liens. Il est alors nécessaire d'appeler récursivement la procédure de modification de versions sur le contexte d'utilisation de chaque version ainsi modifiée.

5.5. Répercussions du gel ou dégel d'une version

Le gel ne peut porter que sur la version en cours. La seule conséquence du gel, est qu'il n'y a plus de version en cours.

Dans le cas du dégel, la version sélectionnée à dégeler devient la version en cours. On rappelle que celle-ci est unique. L'arbre de dérivation de l'objet concerné n'est pas modifié.

5.6. Répercussions de la dérivation d'une version

La dérivation d'une version d'un objet donné, entraîne la modification de l'arbre de dérivation de l'objet, auquel appartient la version. Exemple :

dérivation de version :
dans quel type :
Bâtiment
pour quel objet :
vagabond2
sélection de une version : *le système oblige la sélection d'une version unique*
état(coh)
Il y a trois versions sélectionnées? Voulez-vous :
(a)ffiner la sélection
(i)gnorer la sélection
(c)onsulter la sélection
a
sélection :
nom(v1(vagabond2))
Il y a une version sélectionnée. Voulez-vous :
(a)ffiner la sélection
(i)gnorer la sélection
(c)onsulter la sélection et continuer
c
nom de la nouvelle version :
v4

La version *v4* de *vagabond2* est la nouvelle version en cours, elle est insérée dans l'arbre de dérivation :

- sa version mère est *v1*
- elle n'a pas de versions filles
- elle est copiée à partir de *v1*

5.7. Répercussions de l'abandon d'une version

La version à abandonner est sélectionnée selon le même principe que pour la dérivation. Toutes les versions dont la version sélectionnée est ancêtre sont abandonnées. Pour chacune d'entre elles, deux cas sont à envisager :

(1) la version a un contexte d'utilisation vide : c'est à dire elle n'est utilisée par aucune autre version. Alors, elle est supprimée et toutes les informations liées à la version sont supprimées de la base. Son nom est supprimé de la liste des versions de l'objet concerné.

(2) la version a un contexte d'utilisation non vide : dans chaque version l'utilisant, on la remplace par la version mère de celle qui a été sélectionnée pour l'abandon. On effectue alors les répercussions prévues pour la modification d'une version, récursivement sur le contexte d'utilisation de le version modifiée (voir section 5.3 de ce chapitre). Puis la version est physiquement supprimée, comme dans le cas (1).

L'arbre de dérivation est modifié, il ne contient plus les versions abandonnées.

6. LE SYSTEME RAC

Le système Reprise Abandon Concurrence, n'assure dans le prototype actuel que les fonctionnalités liées aux unités d'admissibilité, de reprise et d'abandon.

6.1. Unité d'admissibilité

Les règles d'admissibilité telles que nous les avons définies dans le chapitre 5, sont vérifiées par le système au plus tôt. Elles sont évaluées à chaque ligne frappée au clavier.

Exemple :

```
Environnement de définition de type
  nom du type
  bâtiment
  ** erreur de syntaxe sur le nom du type
  Bâtiment
  attributs :
  armateur : Personne
  ** Personne type inconnu dans la base
  armateur : Chaîne
  etc ....
```

6.2. Unité d'abandon

Au cours de la définition d'un type, d'un sous-type ou d'une fonction, et de la manipulation d'une version, le système génère un ensemble d'informations, décrivant et représentant le type, le sous-type, la fonction ou la version et l'objet. Chaque ligne donnée par l'usager provoque la modification, la destruction d'une information existante, ou la création d'une nouvelle information. Ces opérations ne sont pas faites sur la base de données, mais dans un espace de travail, qui est une copie de la base de données. Tout ajout ou destruction d'informations se fait dans cet espace et est enregistré dans un journal (une modification se traduit par une destruction suivie d'un ajout). Ces ajouts et destructions sont enregistrés dans le journal dans l'ordre inverse de l'ordre chronologique dans lequel ils ont été effectués dans l'espace de travail. Une information détruite dans l'espace est enregistrée dans le journal avec la marque *moins*, une information ajoutée, est enregistrée avec la marque *plus* .

L'usager dispose des instructions DEBUT, FIN et ABANDON, afin de structurer ses unités d'abandon selon ses besoins (voir chapitre 5). L'unité d'abandon par défaut est la commande (définition d'un type ou d'une fonction, création ou modification de version, etc ..).

Le traitement associé à ces instructions est :

- DEBUT : imbrication d'une nouvelle unité d'abandon. L'instruction est simplement enregistrée dans le journal.

- FIN : s'il s'agit de l'unité d'abandon racine de l'arbre d'imbrication, elle doit être validée ainsi que toutes ses unités filles. Le système identifie l'unité d'abandon racine en consultant le journal. Chaque ligne du journal, jusqu'à trouver l'instruction DEBUT marquant le début de l'unité d'abandon validée est traitée comme suit :

* si l'information est marquée moins, elle est retirée de la base de données, puis supprimée du journal.

* si l'information est marquée plus, elle est ajoutée dans la base de données, puis supprimée du journal.

S'il l'instruction FIN, termine une autre unité d'abandon que l'unité racine, elle est simplement enregistrée dans le journal.

- ABANDON : on abandonne l'unité courante, et toutes les unités filles de l'unité courante. Il suffit pour cela d'annuler chaque ligne du journal, à partir du début, jusqu'à trouver la marque DEBUT, marquant le début de l'unité abandonnée. L'annulation d'une ligne est traitée comme suit :

* si l'information est marquée plus, elle est supprimée de l'espace de travail, et du journal.

* si l'information est marquée moins, elle est ajoutée dans l'espace de travail, et supprimée du journal.

Le début d'une commande n'est considéré comme la marque de début d'une unité d'abandon, que dans le cas où aucune instruction DEBUT n'a été donnée. La fin de la commande est alors traitée comme la fin d'une unité d'abandon.

Dans la cas d'un abandon, plusieurs commandes peuvent avoir été annulées. Le système doit non seulement défaire le travail abandonné, mais aussi retrouver le dialogue correspondant. Il utilise pour cela un autre journal, qui contient l'enchaînement des commandes soumises par l'usager, et qui est maintenu en parallèle avec le journal de la base de données. Le début d'une commande provoque l'enregistrement de son nom dans le journal de commandes. Lors de la validation de la commande, son nom est supprimé du journal de commandes.

6.3. Unité de reprise

Lors de la connexion d'un usager, et du choix de la base, le système consulte le journal associé à cette base. Si celui-ci est vide, cela signifie que la dernière session s'est déroulée correctement. Sinon, il faut mettre en œuvre le mécanisme de reprise. Celui-ci, enregistre dans l'espace de travail, initialisé avec la base de données correspondante, les instructions lues dans le journal. Celles-ci sont traitées dans l'ordre inverse de celui dans lequel elles sont rangées dans le journal. Enregistrer une information signifie :

- si elle est marquée moins, la supprimer de l'espace de travail ;
- si elle est marquée plus, l'ajouter dans l'espace de travail ;

Le système, grâce au journal de la base et au journal de commandes associé, retrouve l'état du dialogue.

7. L'INTERFACE CONCEPTEUR-ETIC

Nous avons essayer d'offrir au concepteur un outil le plus convivial possible. Nous n'avons pas explorer le domaine de recherche sur les interfaces homme-machine, ni le domaine du traitement des langues naturelles, en effet cela n'entrait pas le cadre de cette thèse. Le mode questions-réponses adopté le plus souvent dans les SGBD n'est jamais très apprécié, car il demande souvent beaucoup de texte que le concepteur doit taper au clavier.

De plus, dans le processus de conception le concepteur doit connaître le contenu de la base (le nom des ensembles qui sont déjà définis, les occurrences, les fonctions etc ...). Cette connaissance globale étant rarissime, le nombre d'erreurs croît avec la nervosité du concepteur et la taille de ses bases.

Enfin, la dernière remarque est celle qui concerne la compréhension par le concepteur, d'une part du modèle de données, et d'autre part du schéma qui a été décrit. Ce schéma a pu être décrit par un concepteur (par exemple le chef de projet), mais il pourra être utilisé par d'autres concepteurs (ceux-ci manipuleront des occurrences dans des ensembles). De plus, au fil de la conception, les schémas et les objets deviennent de plus en plus complexes, et il est difficile d'en avoir une idée synthétique.

L'interface utilisateur que nous présentons dans cette section utilise les diverses possibilités graphique (écran bitmap) et de multi-fenêtrage disponibles sur les stations de travail Apollo. L'interface proposée, n'est pas seulement un outil de dialogue entre le concepteur et le système ETIC, mais aussi un outil d'aide à la conception.

7.1. La Communication Concepteur-ETIC

La figure 3 montre l'écran de la station à un moment quelconque d'une session, alors que la fonction d'aide est en attente.

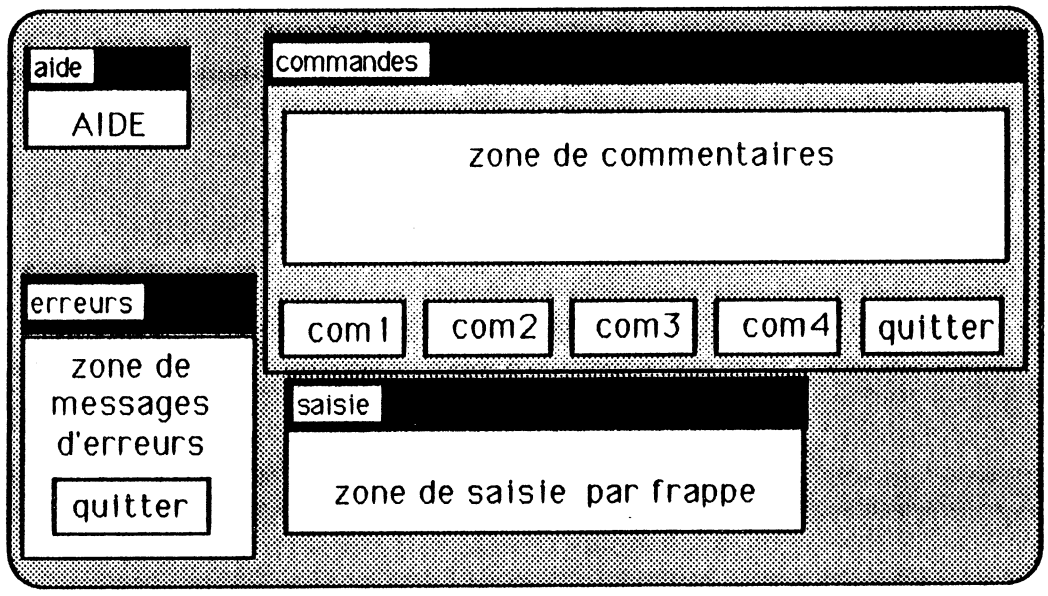


figure 3 : un écran quelconque

La figure 3 montre un écran en fonctionnement normal de ETIC, c'est à dire avec la fonction AIDE en attente. Le dialogue entre ETIC et le concepteur utilise trois fenêtres :

1 La fenêtre **commandes**, contient à un instant donné, la liste des commandes activables. Dans le cas où aucune commande n'est activable, c'est par exemple le cas où un type est en cours de définition, cette fenêtre est fermée.

2 La fenêtre **saisie** est celle dans laquelle le concepteur doit frapper les mots non sélectionnables par le clic de la souris (les noms des types à définir, les noms de fonctions, etc ..).

3 Tous les messages d'erreur sont envoyés dans la fenêtre **erreur**, qui se ferme lorsque l'utilisateur clique sur *quitter* dans cette fenêtre.

Toute acquisition d'informations se fait par frappe au clavier, et tout choix dans un menu se fait par un clic de la souris.

L'utilisateur peut demander de l'aide à n'importe quel moment en *cliquant* avec la souris le mot AIDE dans la fenêtre **aide**. La section suivante décrit les fonctionnalités offertes par la fonction d'assistance.

7.2. Aide au Concepteur

Nous décrivons ici l'interface d'aide, à la disposition de l'utilisateur.

Pour obtenir une assistance, il suffit à l'utilisateur de *cliquer* avec la souris dans la fenêtre **aide** (voir figure 4). La fenêtre montrée dans la figure 4 propose deux types d'orientations :

- liste des types
- listes des fonctions

Toutes les informations concernant les types et les versions de leurs objets sont obtenues en cliquant *liste des types* , et toutes les informations concernant les fonctions, et les programmes les réalisant sont obtenues en cliquant l'autre option, *liste des fonctions.*

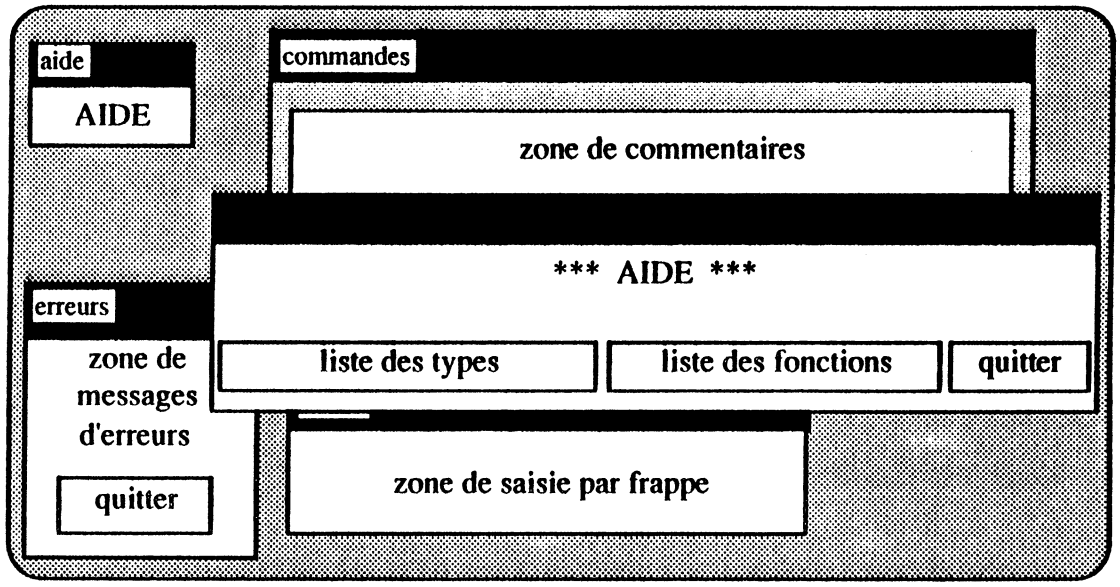


figure 4 : une demande d'assistance

Le concepteur peut vouloir connaître la liste des types ou la définition d'un type donné, ou aussi, la liste des versions d'un objet d'un type.

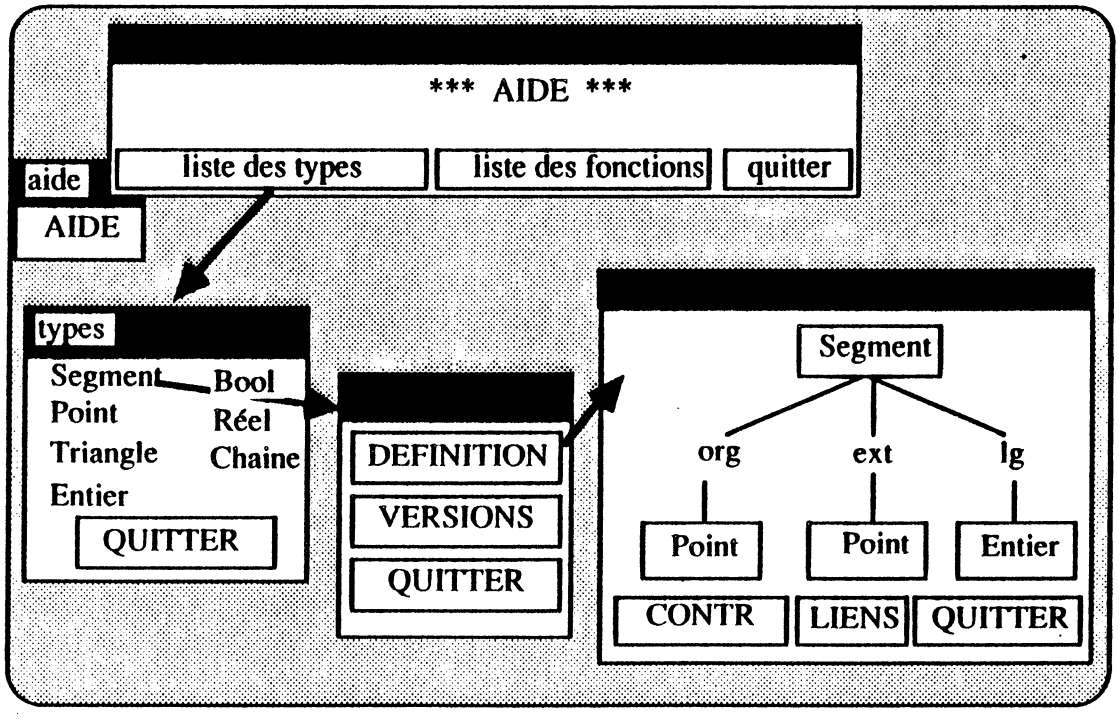


figure 5: le concepteur désire connaître la structure d'un ensemble

Dans la figure 5, le concepteur désire dans un premier temps, la liste des types préalablement définis (option *liste des ensembles*). Puis parmi la liste donnée dans la fenêtre **types**, il choisit un type (en cliquant dessus avec la souris) duquel il veut connaître la définition (**DEFINITION** de la fenêtre **choix**). La définition donnée ne contient ni liens ni contraintes, l'utilisateur peut en demander l'édition en cliquant sur *CONTRAINTES* ou sur *LIENS* dans la fenêtre correspondante.

Le concepteur peut cliquer à tout moment un mot quelconque d'une fenêtre quelconque. Suivant le type de ce mot, le système ouvre une fenêtre et propose dans celle-ci, l'ensemble des actions possibles sur ce mot. Par exemple, s'il s'agit d'un nom de type, les actions possibles sont : *DEFINITION* , *LISTE DES VERSIONS* ; s'il s'agit d'un nom de fonctions les actions possibles sont : *DEFINITION, ALGORITHME,* etc... Le concepteur peut ainsi demander la définition d'un type, niveau par niveau, jusqu'à sa définition en termes de types de base, tout aussi bien que la structure d'un type dont le nom apparaît dans une fenêtre décrivant une fonction.

L'idée pour guider le concepteur dans la recherche des informations qu'il désire, est de procéder sur plusieurs niveaux, chacun attaché à une fenêtre. Le concepteur pourra ainsi consulter plusieurs informations à la fois, et les conserver toutes sous les yeux (en augmentant ou diminuant les fenêtres, en les arrangeant dans l'écran, etc ..). La solution choisie pour réaliser cette interface est d'exploiter toutes les possibilités de la machine (station de travail Apollo) en matière de multi-fenêtrage, de désignation avec la souris, et de gestion des bitmaps.

7.3. Architecture

L'interface décrite dans le paragraphe précédent est opérationnelle sur station Apollo[1].

Le système actuel est entièrement écrit en DProlog. Les informations gérées par ETIC sont-elles mêmes des clauses DProlog (des

---

1 Cette interface a été réalisée en partie par un étudiant de l'IUT d'Informatique de Grenoble [DOU87]

faits ou des prédicats). La figure 6 montre comment s'articulent le programme d'aide et le système ETIC autour des modules qui réalisent les fonctionnalités d'une interface (dessin d'une fenêtre, gestion d'évènements, etc ..).

L'objectif prioritaire dans la réalisation du programme d'aide, est d'avoir un ensemble de modules les *plus portables* possible. Dans cet objectif, nous avons décidé d'écrire ces modules en langage C. De plus, nous envisageons dans un délai à court terme, d'utiliser le SGBD Relationnel ORACLE pour gérer certaines informations de ETIC (en particulier les faits). La consultation et la mise à jour des relations d'ORACLE se feront via le langage C. Le langage Pascal est destiné à interfacer Dprolog.

Un ensemble de modules C ont été écrits réalisant toutes les fonctionnalités de base nécessaire à une interface ETIC-concepteur. Par exemple, dessin d'un menu dans une fenêtre, affichage d'une fenêtre, rafraîchissement d'une fenêtre, récupération d'un mot sélectionné, etc ..

Ces modules (écrits en C) sont pilotés depuis ETIC ou AIDE, deux applications écrites en Dprolog. La communication entre Dprolog et C se faisant par Pascal.

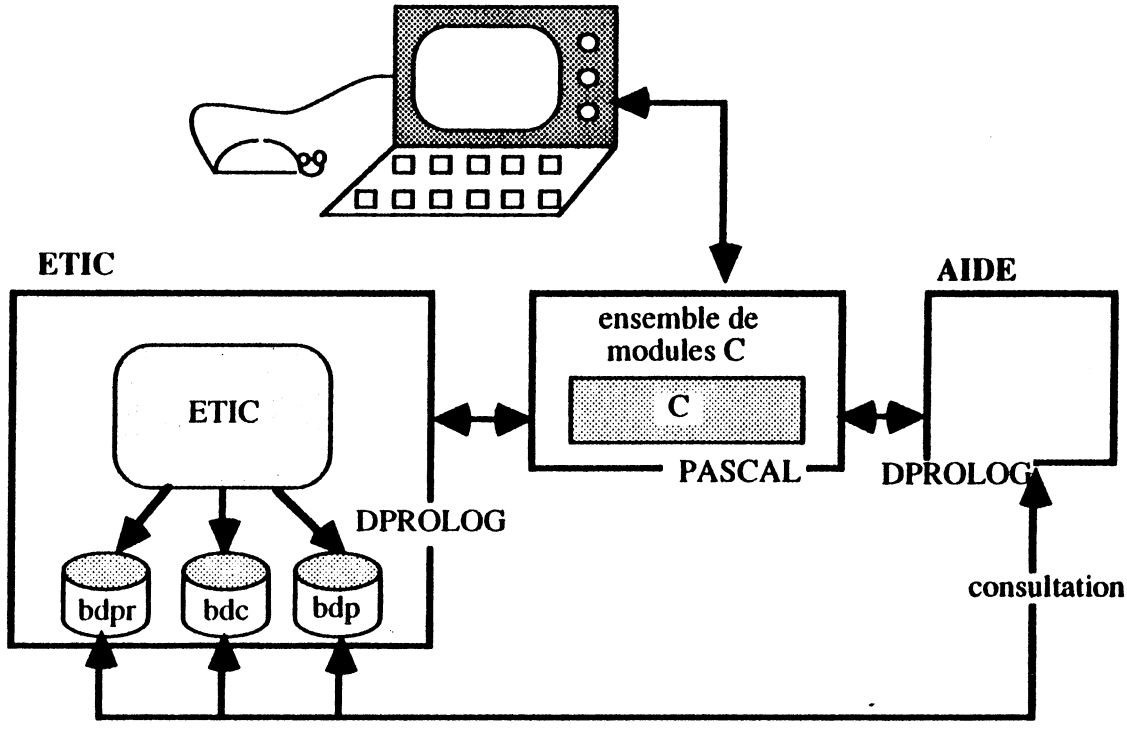


figure 6 : architecture de ETIC et de son programme d'AIDE

L'intégration des trois langages (Dprolog, C et Pascal), ainsi que la manipulation de fenêtres contenant des zones de textes, de longueur quelconque, et contenant des informations sélectionnables, ont constitué un obstacle non négligeable à l'utilisation d'un éditeur de dialogue (par exemple DIALOG disponible sur station Apollo). La solution choisie a consisté à utiliser des primitives graphiques et de gestion de fenêtres disponibles sur station Apollo (librairie GPR, Graphics Primitives Routines).

8. LA REALISATION

Le système ETIC est entièrement écrit en prolog (mise à part l'interface décrite dans la section précédente. Il est opérationnel sur station Apollo (DN330), et offre toutes les fonctionnalités décrites dans ce chapitre. Le choix de Prolog a été fait pour plusieurs raisons :

- rapidité de programmation : on développe vite de grosses applications.

- gestion des faits de taille variables dans des banques de clauses (fonctionnalité propre à la version du langage prolog utilisé) qui permet un

accès plus efficace qu'à un fichier texte classique et moins lourd qu'à un SGBD tel Oracle par exemple. Compte tenu de nos moyens en effectif, nous n'avons pas réalisé dans le cadre de cet étude, l'interface avec Oracle. L'accès aux clauses d'une banque est assuré par Prolog, par filtrage sur la tête de la clause, et selon le principe d'unification [CLM85], [DOH84]

Nous avons choisi un langage prolog permettant l'activation automatique d'un prédicat (le prédicat *GO*, seule possibilité de réaliser des pseudo-modules), et ouvert vers son environnement extérieur (interface avec Pascal).

Le tableau donné en annexe D, donne la taille des modules composant le prototype ETIC.

9. CONCLUSION

Nous avons décrit dans ce chapitre les fonctionnalités opérationnelles dans le prototype. Celles-ci concernent le serveur d'objets, le serveur de versions, le système RAC et l'interface ETIC-concepteur. Le système de communication (voir figure 1 au début du chapitre) reste à analyser et programmer.

Le prototype ETIC obtenu, atteint les limites imposées par le langage utilisé. En effet, malgré la structuration du prototype en modules, la gestion d'une telle application écrite en Prolog devient difficile (passages des paramètres entre modules via des fichiers textes, les modules sont long à charger en mémoire centrale, quelque soit leur longueur et ne doivent pas dépasser 4500 lignes, etc ...).

Le but cherché, lors de la réalisation de ETIC est atteint, en effet, nous pouvons ainsi montrer la faisabilité et l'apport des fonctionnalités que nous avons proposées. Nous devons réalisé le système de communication, puis nous pourrons proposer un système complet de gestion de projet CAO.

CHAPITRE 7

CONCLUSIONS ET PERSPECTIVES

table des matières

1. CONCLUSIONS

Le sujet que nous avons choisi d'aborder dans cette thèse se place dans le domaine de la gestion des données pour les applications CAO (Conception Assistée par Ordinateur), dans un environnement coopératif multi-usagers. Il recouvre les problèmes de :

- la dynamique des données : il faut gérer l'évolution des données dans le temps, et savoir caractériser les différentes étapes de leur évolution. Le travail de conception consiste à développer plusieurs solutions pour la conception d'un objet, à les évaluer, puis à choisir la *meilleure*, c'est à dire celle qui convient le mieux pour le problème visé. Il faut donc stocker ces différentes solutions, offrir des outils pour leur évaluation et leur caractérisation ;
- la coopération dans l'organisation d'un projet de conception : on ne peut se contenter d'interdire ou d'accepter un accès à une donnée. L'activité de conception présente en effet des caractères particuliers : coût et durée d'une tâche, répartition des tâches, travail d'équipes, coopération, etc...;
- le partage des données : les solutions apportées jusqu'à présent, aux problèmes de la gestion des transactions sont remises en cause.

La définition du système de gestion de projet ETIC est une réponse aux problèmes engendrés par les trois points ci-dessus. Il offre :

- un modèle de données qui permet d'exprimer la complexité des structures des données de CAO et des liens entre ces données (chapitre 2);
- un modèle de versions simple, puissant et évolutif avec lequel on peut tracer le processus de conception de façon pertinente et significative (suffisament de versions, mais pas trop), et qui offre des outils qui permettent en fin de conception de construire une classification des versions selon leurs propriétés, ou les performances (chapitre 4) ;
- un ensemble de fonctionnalités pour décrire et gérer le projet. Celles-ci permettent d'exprimer le partage des objets, et ainsi les accès à ces objets (chapitre 3);
- un système de gestion de la concurrence, de reprise et d'abandon mettant en œuvre des mécanismes adaptés aux besoins de l'usager et aux caractères particuliers des applications visées (chapitre 6).

En constatant l'insuffisance des concepts proposés dans de précédentes solutions (que nous décrivons dans le chapitre 1) nous avons voulu adopter une approche originale des problèmes abordés :

- le partage des données n'est pas résolu par des copies multiples comme c'est le cas dans la plupart des solutions proposées, mais par l'analyse et l'expression de l'organisation et de la répartition du travail au sein d'un projet, pour cela nous tirons parti de la structure des données conçues et manipulées.

- les concepts définis pour la gestion des versions, se veulent simples mais évolutifs. Ils permettent la caractérisation et le contrôle des versions des données.

Le travail théorique réalisé a permis la spécification et la programmation d'un prototype opérationnel (chapitre 7) permettant la gestion des données et des versions d'un projet de CAO.

2. LIMITES DE L'APPROCHE

Les solutions proposées quant à l'expression de l'organisation du projet de conception sont adaptées au modèle orienté objet que nous avons choisi. En effet, si nous voulions utiliser les concepts que nous avons défini à un autre modèle de données, nous devrions remettre en cause certains principes, comme la définition et l'exécution des vues. La répartition du travail dans un cadre différent devrait être ré-étudiée.

La gestion des contraintes d'intégrité est quelque fois inadaptée. Rappelons qu'un objet incohérent est admis dans la base. Dans certains cas, par exemple pour les objets de type date, cette définition ne convient pas. Il faut faire en effet la distinction entre les objets à concevoir, qui sont soumis à de multiples modifications afin de tenter différentes solutions, et les objets d'utilisation courante, sur lesquels les modifications ne signifient pas nouvelles tentatives (c'est la cas d'une date).

Nous avons considéré qu'un objet n'existe qu'à travers ses versions. Ce principe est rigide et quelques fois inadapté. Il serait nécessaire de ne l'appliquer qu'à certains types d'objets, désignés par l'usager. De même, nous ne permettons d'instancier un attribut que par une version d'objet, une plus grande puissance d'expression devient possible en offrant la possibilité d'instancier les attributs avec un objet. Ceci entraîne des problèmes non triviaux de gestion des versions. En effet, supposons qu'une version V de O_1, utilise pour un de ses attributs un objet O_2

admettant plusieurs versions. Si l'attribut correspondant est référencé dans un lien, la valeur du lien est certainement différente selon la version de O_2 considérée. On peut faire la même remarque pour une contrainte. On peut envisager la solution suivante : gérer autant de versions supplémentaires de O_1 qu'il y a de valeurs pour un lien ou une contrainte référençant O_2. Cette solution génère dans la plupart des cas, un grand nombre de nouvelles versions, elle n'est pour cette raison pas satisfaisante. Une autre étude serait nécessaire pour prendre un tel propblème et y apporter une solution intéressante.

Il faut signaler une limite liée au prototype et au langage de programmation choisi. Le système atteint une taille (voir annexe D) que Prolog n'est plus capable de gérer de façon fiable.

3. PERSPECTIVES

Les travaux futurs que nous pouvons envisager sont de deux types :

- les extensions possibles aussi bien de l'étude théorique que pratique;
- les ouvertures vers d'autres domaines que la CAO.

3.1. Extensions

Du fait de l'importance des problèmes abordés, notre étude théorique laisse un ensemble de questions en suspend :

- le contrôle de la cohérence de l'assemblage des tâches réalisées par différents concepteurs, oblige dans le cas de la détection d'une incohérence, de refaire une ou plusieurs des tâches. La solution que nous proposons, est basée sur seulement deux stratégies : la stratégie du plus fort, et la stratégie LIFO (Last In First Out). Il serait nécessaire ici, de définir un système de diagnostic de cette incohérence, ainsi qu'un ensemble de stratégies de retrait des tâches.
- les mécanismes de gestion de la concurrence sont insuffisants. Dans l'environnement que nous avons défini, un système de verrouillage est suffisant dans les systèmes public et semi-publics, mais inadapté dans les systèmes privés. Il faut pour obtenir des résultats satisfaisants, étudier de façon précise le comportement du concepteur face à sa station de travail (utilisation du multi-fenêtrage particulièrement). En effet, bien que privés,

les systèmes des concepteurs posent un problème non négligeable d'accès concurrents aux données. Nous avons poser le problème dans le chapitre 5, et proposer quelques idées de solutions.

- nous n'offrons pas de véritable langage d'interrogation des données. Il serait intéressant d'étudier les résultats des études faites dans le domaine des objets complexes et des systèmes orientés objets, pour tenter de définir une algèbre des données définies dans un système comme ETIC.

- enfin, nous n'avons pas pris en compte la modification de schémas.

Le prototype réalise un sous ensemble des fonctionnalités définies au cours de l'étude théorique. Celles-ci regroupent en effet :

- les fonctions du serveur d'objets;

- les fonctions du serveur de versions, sauf celles concernant le contrôle des versions (acceptabilité, contrôle de la fréquence de dérivation, obligation de dérivation, etc..). Les mécanismes nécessaires à ce contrôle sont programmés, et doivent être intégrés. Le contrôle des versions sera pour cette raison disponible dans un délai à court terme.

- les fonctions du système de communication n'ont pu être réalisées, non pas pour des raisons techniques, mais simplement par manque de resssources humaines;

- l'interface Concepteur-ETIC, mérite d'être améliorée et complétée, en prenant en compte l'évolution des fonctionnalités des primitives graphiques et de gestion d'écran du système d'exploitation utilisé.

3.2. Ouvertures vers d'autres domaines

Nous avons étudié les problèmes posés par le génie logiciel [NOM88], [DES86], la gestion de documents structurés [QUI87], et la gestion de données multimédia [ABI87], [COL87], [COL88]. Il existe entre tous ces domaines et celui de la CAO une intersection non négligeable de besoins. Nous pensons que certains des concepts que nous avons définis sont applicables à ces autres domaines. Le serveur de versions offre des fonctionnalités adaptées aux applications des domaines cités. Les solutions que nous avons proposées pour la gestion de la coopération dans l'activité de conception paraissent utilisables dans le cadre de la conception de documents.

La prise en compte des contraintes d'intégrité est inexistante dans le courant actuel des SGBD orientés objets [LER87], [MAS86]. Ainsi nous pensons que certains des mécanismes que nous avons mis en œuvre (exécution automatique de contraintes d'intégrité, rupture de liens, héritage, etc..), méritent d'être étudiés dans le cadre de tels systèmes.

REFERENCES BIBLIOGRAPHIQUES

[ABC87]
Michel ADIBA, Ngoc BUI QUANG, Christine COLLET
Aspects temporels, historiques et dynamiques des bases de données
TSI, vol. 6, N°5, 1987

[ABI87]
Serge ABITEBOUL
Bases de données et objets structurés
TSI Vol.6 N°5, 1987

[ABV84]
Serge ABITEBOUL, Victor VIANU
Transactions in Relational Databases
Proc. of VLBD Conference, Singapore, August 1984

[ADI81]
Michel ADIBA
Derived relations, a unified mechanims for views, snapshots and distributed data
Proc of the 7th VLDB, Cannes (France) 1981

[ADL80]
Michel ADIBA, Bruce G. LINDSAY
Database snapshots
Proc of the 6th VLDB, ??? 1980

[AFL86b]
Jacques AUTRAN, Michel FLORENZANO, Jacques LEMAITRE, Carole PALISSER,
SGBD Avancés et CAO Architecture, une application à la description d'un projet de bâtiment.
Actes Conf. CAO et Robotique en Architecture et BTP, Marseille 1986

[AGO86]
A. ALBANO, G. GHELLI, M.E. OCCHIUTO, R. ORSINI
A strongly typed, interactive object-oriented database programming language
Proc. of International Workshop an object-oriented database systems, Asilomar (Californie) 1986

[APA87]
Jacques AUTRAN, Carol PALISSER
Vues et versions d'objets Complexes
3ièmes Journées Bases de Données Avancées, Port Camargue (France), Mai 1987

[AUT88]
Jacques AUTRAN
Vues d'objets en CAO, Illustration dans le domaine de l'Architecture.
Proc. of the 7th European Conference on CAD/CAM and Computer Graphics, Paris, Feb. 1988.

[BAK85]
D.S. BATORY, Won KIM
Modelling Concepts for VLSI CAD Objects
ACM Transactions on Databases Systems, Vol. 10, N° 3, Sept. 1985

[BAN87]
Jay BANERJEE and al.
Semantics and implementation of schema evolution in object oriented databases
SIGMOD Record, Vol.16, N°3, Dec 1987

[BAR43]
René BARJAVEL
Le voyageur imprudent
Denœl 1943, ré-édition 1958

[BEH88]
E. BERTINO, L.M. HAAS
Views and security in distributed database management systems
Proc of EDBT conference, Venice (Italy) March 1988

[BES86]
Nouredime BELKATIR, Jacky ESTUBLIER
Protection and Cooperation in a software engineering environnement
Proc. of International Workshop, Trondheim, Norway, June 1986

[BKK85]
François BANCILHON, Won KIM, Henry F. KORTH
A Model of CAD Transactions
Proc. of VLDB Conference, Stockholm 1985

[BLT86]
José A. BLAKELEY, Per-Åke LARSON, Franck Wm. TOMPA
Efficiently updating materialized views
Proc. of ACM SIGMOD, June 1986

[BOK86]
Claude BOCKSENBAUM, Michèle CART, Jean FERRIE, Jean François PONS,
Le Contrôle de cohérence dans les bases de données réparties,
Revue MBD, N°2, Janvier 1986

[CHE87]
Jean Pierre CHEVALLET
Cohérence et schémas dynamiques dans les bases de données avancées
Rapport de DEA, USTMG-INPG, Grenoble Juillet 1987

[CKI86]
Hong Tai CHOU, Won KIM
A unifying framework for version control in a CAD environment
Proc. of VLDB Conference, Kyoto, Aug. 1986

[COL87]
Christine COLLET
Les formulaires complexes dans les bases de données multimédia
Thèse de Nouveau Doctorat de l'Université Grenoble 1, Nov. 1987

[COL88]
Christine COLLET
Gestion d'objet complexes au travers de formulaires dynamiques
4ième Journées de bases de données avancées, Benodet (France) Mai 1988

[CLM85]
William F. CLOCKSIN, Christopher S. MELLISH
Programmer en PROLOG
Editions Eyrolles, 1985

[DAT81]
C.J. DATE
Introduction to Database Systems (3rd Edition),
Addison Wesley 1981

[DAV73]
Ch. T. DAVIES
Recovery Semantics for a DB-DC Systems
Proc. ACM National Conference 28, 1973

[DAV81]
Bertrand DAVID,
Méthodologie pour la Construction de Systèmes CAO, SIGMA CAO,
Thèse d'état USMG-INPG, Grenoble 1981

[DDE88]
Christophe DAMIER, Bruno DEFUDE
Un modèle de données pour les informations géographiques
4ièmes Journées Bases de Données Avancées, Bénodet (France), Mai 1988

[DEA82]
Claude DELOBEL, Michel ADIBA,
Bases de Données et Systèmes Relationnels,
Dunod Informatique, 1982

[DEN87]
D.E. DENNING, and al.
Views for multilevel database security
IEEE transactions on software engineering, Vol. se-13, N° 2, Feb. 1987

[DIL85]
Klaus R. DITTRICH, Raymand A. LORIE
Version Support for Engineering Database Systems
IBM Research Laboratory, Computer Science RJ 4769 (50628) July 1985

[DLO85]
Klaus DITTRICH, Raymond LORIE,
Object Oriented Database Concepts for Engineering Applications
IBM Research Laboratory, Computer Science RJ 4691 (50029), Aug. 1985

[DOH84]
Philippe DONZ, Rosalie HURTADO
Le langage D-Prolog
Editions Editests, 1984

[ESC87]
Christian ESCULIER
La tolérance sémantique dans les bases de données
3ièmes Journées de Base de Données Avancées, Port Camargue (France), Mai 1987

[FAR87]
Marie Christine FAUVET, Dominique RIEU
CADB, un système de gestion de bases de données et de connaissances pour la CAO
Proc. of the 6th European Conference on CAD/CAM and Computer Graphics, Paris, Feb 1987

[FAR87b]
Marie Christine FAUVET, Dominique RIEU
Une interface d'aide à la décision pour CADB
Journées d'étude sur le thème des postes de travail dans les systèmes d'informations, Seyssel, Juin 1987, à paraître

[FAU86]
Marie Christine FAUVET,
CADB,un SGBD pour la CAO, Présentation du Prototype,
Rapport de Recherche TIGRE 33, Avril 1986

[FAU87]
Marie Christine FAUVET
Gestion de transactions pour les bases de données CAO
3ièmes Journées de bases de données avancées, Port-Camargue (France) Mai 1987

[FAU87b]
Marie Christine FAUVET
Le partage des objets dans un SGBD pour la CAO
Revue CFAO et Infographie, Vol. 2, N°4, Ed HERMES, 1987.

[FAU88]
Marie Christine FAUVET
ETIC, a multi-users CAD database management system
RR N°720-I, LGI-IMAG, May 1988

[GMS87]
Hector GARCIA-MOLINA, Kenneth SALEM
SAGAS
SIGMOD Record, VOL. 16, N°3, Dec 1987

[GRA81]
Jim GRAY
The Transactions Concept, Virtues and Limitations
Proc. of VLDB Conference, Cannes 1981

[HAR87]
Theo HAERDER, Kurt ROTHERMEL
Concepts for Transaction Recovery in Nested Transactions
SIGMOD Record, Vol. 16, N° 3, Dec 1987

[KAC86]
Randy H. KATZ, M. ANWARRUDIN, Ellis CHANG
A version server for computer aided design data
Proc. of the 23rd Design Automation Conference, Las Vegas June 1986

[KAL82]
Randy H. KATZ, Tobin J. LEHMAN
Storage structures for versions and alternatives
Computer sciences report N°479, University of Wisconsin Madison, July 1982

[KAT85]
Randy H. KATZ,
Information Management for Engineering Design,
Springer Verlag Berlin, 1985

[KAW83]
Randy H. KATZ, Schlomo WEISS
Transaction Management for Design Databases
Computer Sciences Department, Univ. of Wisconsin Madison, Technical Report N°496, Feb. 1983

[KBA85]
Won KIM, D.S. BATORY
A Model and storage Technique for versions of VLSI CAD objects
Proc. of the International Conference on Foundations of Data Organization, Kyoto Japan, May 1985.

[KCB86]
Randy H. KATZ, Ellis CHANG, Rajiv BHATEJA,
Version modelling concepts for computer aided design databases
Proc. of ACM Sigmod Conference, Washington Mai 1986

[KCH87]
Randy H. KATZ, Ellis CHANG
Managing change in a computer aided desing database
Proc. of the 13th VLDB Conference, Brighton

[KLM84]
Won KIM, Raymond LORIE, Dan McNABB, Wil PLOUFFE
A Transaction Mechanism for Engineering Design Databases
Proc. of VLDB Conference Singapore August 1984

[KRA87]
Sacha KRAKOWIAK
Guide, a distributed system object
ACM operating system review, 1987

[KSW85]
Peter KLAHOLD, Gunter SCHLAGETER, Wolfgang WILKES
A Transaction Model supporting Complex Applications in Integrated Information Systems
SIGMOD Record Vol. 14 N° 4, Dec. 1985

[KSW86]
Peter KLAHOLD, Gunter SCHLAGETER, Wolfgang WILKES
A general Model for version management in databases
Proc. of 12th Conference on VLDB, Kyoto (Japan), August 1986

[LER87]
Christophe LECLUSE, Philippe RICHARD, Fernando VELEZ
O2, an object oriented data model
Rapport Technique Altaïr 10-87, sept 1987

[LOP83]
Raymond LORIE, Wilfred PLOUFFE
Complex objects and their use in design transactions
Engineering Design Applications, Database Week, San Jose, May 1983

[LOR83]
Raymond LORIE, Wilfred PLOUFFE,
Complex Objects and their use in Design Transactions.
Proc. of Databases Week, San Jose, May 1983

[LRV87]
Christophe LECLUSE, Philippe RICHARD, Fernando VELEZ
O2, an oriented data model
Rapport Altaïr 10-87, Sept 87

[LRV88]
C. LECLUSE, P. RICHARD, F. VELEZ
Un modèle de données orienté-objets
4ièmes Journées Bases de Données Avancées, Bénodet (France), Mai 1988

[LYN83]
Nancy A. LYNCH
Multilevel Atomicity, A New Correctness Criterion for Database Concurrency Control
ACM Transactions on Database Systems, Vol. 8, N°4, Dec. 1983

[MAS86]
David MAIER, Jacob STEIN
Development of an Object-Oriented DBMS
OOPSLA'86 Proceedings

[MOS82]
J. Eliot B. MOSS
Nested transactions and reliable distributed computing
Proc 2nd Symposium on Reliability in Distributed Software and Database Systems, 1982

[NGO86]
Bui Quang NGOC
Aspects dynamiques et gestion du temps dans les systèmes de bases de données généralisées
Thèse Docteur de l'INP Grenoble, Nov. 1986

[NOM88]
N. BELKATIR, J. ESTUBLIER
NOMADE, A kernel for programming in the large
IFIP, work group 2.4, Deauville, March 88

[PAL87]
Carol PALISSER
Un système de gestion de versions pour la CAO en architecture
Rapport Interne GRTC N°253, Décembre 1987

[QUI87]
Vincent QUINT
Une approche de l'édition structurelle des documents
Thèse d'état de l'Université de Grenoble 1, Mai 1987

[RIE85]
Dominique RIEU,
Modèle et Fonctionnalités d'un SGBD pour les Applications CAO,
Thèse de Nouveau Doctorat, INPG, Grenoble 1985

[RIE86]
Dominique RIEU
Nature état et dynamicité de l'objet CAO
2ièmes Journées Bases de Données Avancées, Giens (France), Avril 1986

[SLW88]
Amit P. SHETH, James A. LARSON, Evan WATKINS
TAILOR, a tool for updating views
Proc of EDBT Conference, Venice (Italy) March 1988

[SRG83]
M. STONEBRAKER, RUBENSTEIN, GUTTMAN,
Application of abstract data types and abstract indices to CAD databases,
ACM SIGMOD, San Jose, May 1983

[TIC85]
Walter F. TICHY
RCS - A system for version control
Software practice and experience, Vol 15(7), July 1985

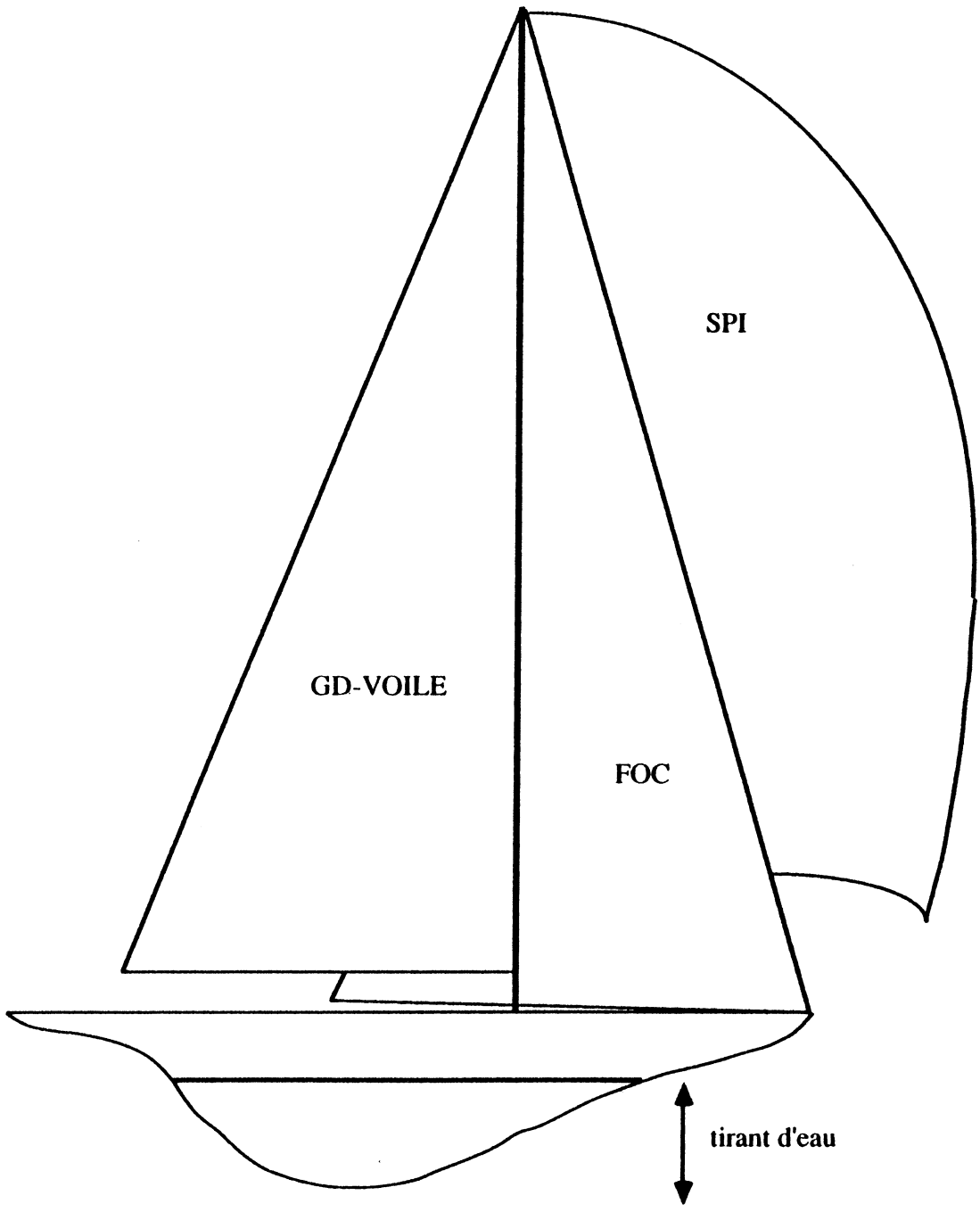
[WAL84]
Bernd WALTER
Nested Transactions with Multiple Commit Points, An Approach to the Structuring of Advanced Databases Applications
Proc. of VLDB, Singapore, AUG 1984

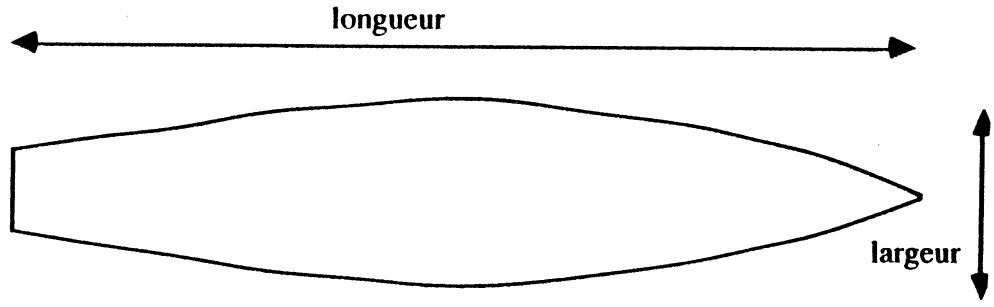
[WINK]
J.F.H. WINKLER
The integration of version control into programming languages
Proc. of International Workshop, Trondheim, Norway, June 1986

ANNEXES

table des matières

ANNEXE A.
PLAN DESCRIPTIF D'UN VOILIER




longueur
largeur

ANNEXE B.
LA DEFINITION DE PROJET

L'objet à concevoir est décrit par le type suivant :

Circuit : ($A_{circuit}$, $L_{circuit}$, ∅)
avec **$A_{circuit}$**= [rep-f : Rep-Circuit, * représentation fonctionnelle *
rep-l : Rep-Circuit, * représentation logique *
rep-e : Rep-Circuit, * représentation électrique *
rep-i : Rep-Circuit] * représentation implantée *
$L_{circuit}$= [(rep-e,déduction(rep-l)]

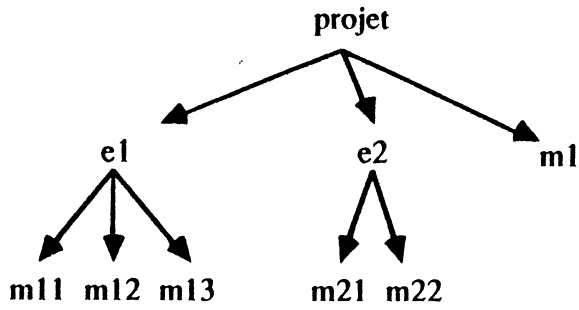
Rep-Circuit : ($A_{Rep\text{-}Circuit}$, ∅, $C_{Rep\text{-}Circuit}$)
avec **$A_{Rep\text{-}Circuit}$** = [entrées : List Ent, * interface du circuit *
sorties : List Sort, * *
enveloppe : Env * *
équipot : List Equi, * composition du circuit *
$comp_1$: Composant, * *
$comp_2$: Composant, * *
$comp_3$: Composant] * *

$C_{Rep\text{-}Circuit}$ = {NOT intersecte (équipot),
NOT recouvre ($comp_1$,$comp_2$,$comp_3$)}

La répartition du projet en équipes est la suivante :

niveau 1 : deux équipes e_1 et e_2 et un concepteur m1
niveau 2 : dans e_1 : 3 concepteurs m_{11}, m_{12}, m_{13}
dans e_2 : 2 concepteurs m_{21}, m_{22}

La hiérarchie de systèmes est donnée par la figure suivante :



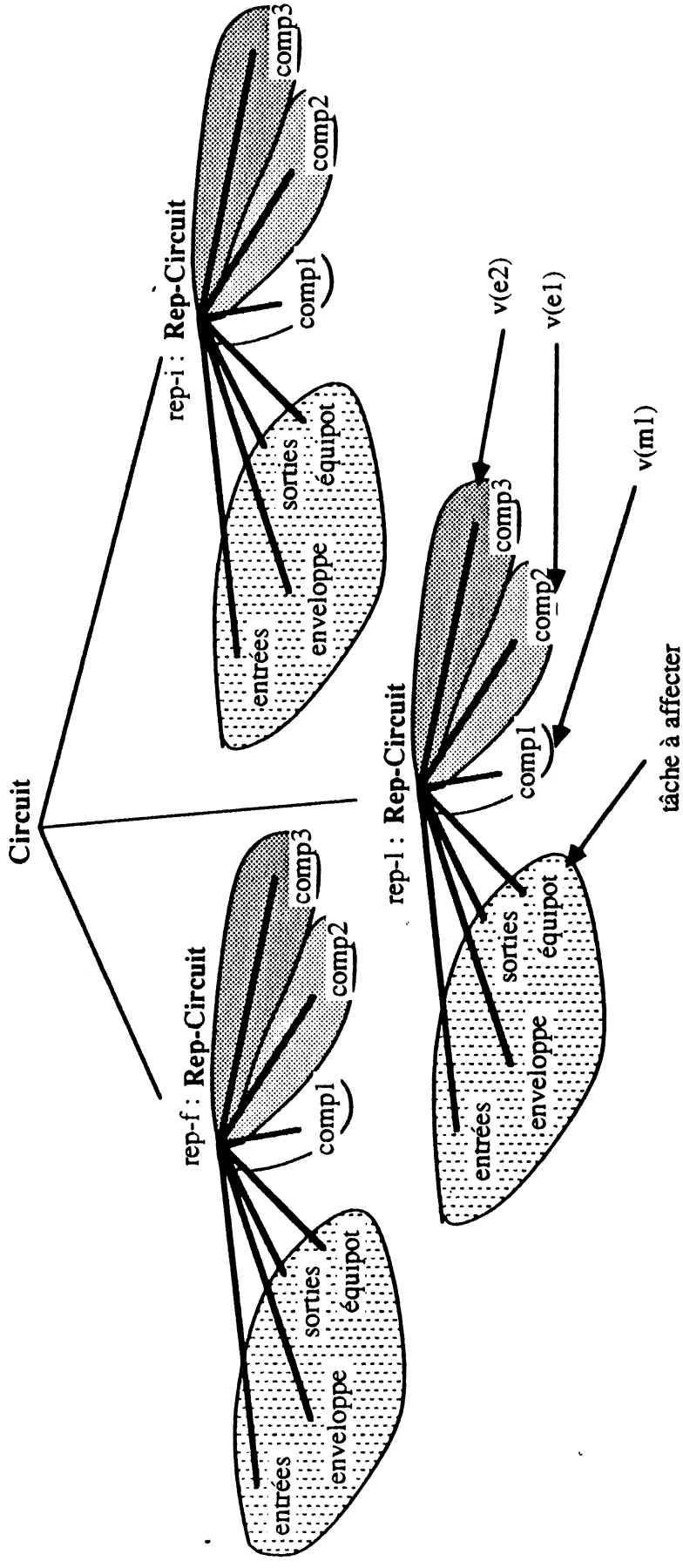
Il est y a donc trois vues à définir au niveau projet, trois au niveau de e_1 et deux au niveau de e_2:

- E = [e_1, e_2, m_1], l'ensemble des équipes du projet
- v(m_1) = [$comp_1$.rep-f, $comp_1$.rep-l, $comp_1$.rep-i]
- v(e1) = [$comp_2$.rep-f, $comp_2$.rep-l, $comp_2$.rep-i]
- v(e2) = [$comp_3$.rep-f, $comp_3$.rep-l, $comp_3$.rep-i]
- v(m11) = [$comp_2$.rep-f]
- v(m12) = [$comp_2$.rep-l]
- v(m13) = [$comp_2$.rep-i]
- v(m21) = [$comp_3$.rep-f, $comp_3$.rep-l]
- v(m22) = [$comp_3$.rep-i]

La décomposition pour chaque équipe s'est faite sur la base des différentes représentations des composants (fonctionnelle, logique, implantée).

La figure de la page suivante, montre sur l'arbre de composition de l'objet, la répartition des différentes tâches. Chaque niveau de gris correspond aux tâches allouées aux équipes (du niveau 2).

Le lien défini dans le type Circuit, est évalué au niveau du projet, dès que la représentation logique du Circuit est connue (rep-l). La contrainte NOT recouvre ($comp_1$, $comp_2$, $comp_3$) est évaluée localement à chaque équipe (elle nécessite la connaissance des trois composants d'une représentation, indépendamment des autres représentations. Quant à la contrainte NOT intersecte (équipot), elle évaluée au niveau dans lequel sont instanciées les équipotentielles.


Circuit
rep-i : Rep-Circuit
comp3
comp2
comp1
entrées
enveloppe
sorties
équipot
rep-f : Rep-Circuit
comp3
comp2
comp1
entrées
enveloppe
sorties
équipot
rep-l : Rep-Circuit
comp3
comp2
comp1
entrées
enveloppe
sorties
équipot
v(e2)
v(e1)
v(m1)
tâche à affecter

La définition complète du projet est donc :

DEF-P

nom du projet :	**Circuit**
définition des équipes (E ou C) :	**E**
nom :	**e1**
vue associée :	**v(e1)**
droits sur VLSI :	**LECT, EXT, REM, ECH**
équipes (E ou C) :	**E**
nom :	**e2**
vue associée :	**v(e2)**
droits sur VLSI :	**LECT, EXT, REM, ECH**
équipes : (E ou C) :	**C**
nom :	**m1**
vue associée :	**v(m1)**
droits sur VLSI :	**tous**
équipes (E ou C) : .	

Les vues affectées ne définissent pas le type Circuit, affectation de :

entrées.rep-f :	**ADM**
sorties.rep-f :	**ADM**
etc ..	{ Les tâches non affectuées seront }
	{ allouées à l'administrateur du projet }

STRATEGIE DE RETRAIT 1 LIFO 2 poids le plus fort : **2**

v(e1) : **1**
v(e2) : **1**
v(m1) : **2**

ORDONNANCEMENT (O-N) : **O**

V(e1) : **1**
v(e2) : **1**
v(m1) : **2**

Le même processus est répété pour les équipes e_1 et e_2

ANNEXE C.
LA GENERATION DES INFORMATIONS

Nous décrivons dans cet annexe, une banque des faits engendrés par un ensemble de commandes soumises au système. Cette banque n'est pas complète. En effet, nous n'avons mis que des faits significatifs afin de ne pas surcharger l'exemple. La section 1, montre les faits de la base privée d'un concepteur, et la section 2, ceux de la base de connaissances d'un projet.

C.1. Une base Privée

C.1.1.les faits décrivant les versions et les objets

!bâtiment([v1,vagabond2],	*la version v1 de vagabond2*
[coh,com,v,_x1],	*est cohérente et complète, créée par valeur*
[[armateur,mariech,an,_x2],	*son attribut armateur est instancié par nom,*
[misealo,[v1,misealovag],an,_x3],	*par l'objet de base mariech*
[port,Brest,an,_x4],	
[flotteur,[v1,flotvag2],an,_5],	*son attribut flotteur est instancié par nom*
[propulseur,voile,an,_x6],	*par la version v1 de l'objet flotvag2*
[usage,plaisance,an,_x7]]).	

```
!bâtiment([v1,vagabond5],                    la version v1 de vagabond5
[coh,incom,v,_x1],                           est incomplète
   [[armateur,mariech,an,_x2],
   [misealo,[v1,misealovag2],an,_x3],
   [port,nil,ai,_x4],                        son attribut port n'est pas instancié
   [flotteur,nil,ai,_x5],
   [propulseur,voile,an,_x6],
   [usage,plaisance,an,_x7]]).

!bâtiment([v2,vagabond2],
[coh,com,v,_x1],
   [[armateur,mariech,an,_x2],
   [misealo,[v1,misealovag],an,_x3],
   [port,Toulon,an,_x4],
   [flotteur,[v1,flotvag2],an,_x5],
   [propulseur,voile,an,_x6],
   [usage,plaisance,an,_x7]]).

!coque([ni,10],                              la version unique, de l'objet système
                                             de surrogate 10
   [coh,com,lfp,1],                          dans !coque est calculée par le lien
                                                ponctuel N°1
   [[long,13,an,_x1],
   [larg,5,an,_x2],
   [tirant,2,an,_x3],
   [depl,65,an,1],                           son attribut depl est calcul par le lien
   [nbvol,1,an,_x4]]).                       fonctionnel N°1 défini dans le type
```

```
!coque([v1,flotvag2],
[coh,com,v,_x1],
   [[long,12,an,_x2],
   [larg,5,an,_x3],
   [tirant,2,an,_x4],
   [nbvol,1,an,_x5],
   [depl,60,an,1],
   [long,20,an,_x6],
   [tirant,3,an,_x7],
   [depl,60,an,1]]).

!prisonnier([v1,mcf],
[coh,com,v,_x1],
   [[nompat,fauvet,an,_x2],
   [prénom,mariech,an,_x3],
   [dateemp,[ni,14],an,2]]).
```

l'attribut dateemp, est calculée par lien fonctionnel, qui engendre la création d'un objet système (de surrogate 14 dans !date)

```
!date([ni,14],
[coh,com,lf,2],
   [[année,88,an,_x1],
   [mois,8,an,_x2],
   [jour,24,an,_x3]]).
```

C.1.2.les faits décrivant la filiation des versions

```
mère(!bâtiment,vagabond,v1,[]).
```

la version v1 de vagabond dans !bâtiment n'admet aucune version dérivée

```
mère(!bâtiment,vagabond2,v1,[]).
mère(!bâtiment,vagabond2,v2,[]).
mère(!bâtiment,vagabond3,v1,[]).
mère(!bâtiment,vagabond4,v1,[]).
mère(!bâtiment,vagabond5,v1,[v2,v4,v5]).
mère(!coque,flotrapide,v1,[]).
mère(!coque,flotvag,v1,[]).
mère(!coque,flotvag2,v1,[]).
```

les versions v2 v4 et v5, sont filles de v1 dans l'arbre de dérivation de l'objet vagabond5 de !bâtiment

mère(!coque,flotvag2,v1,[]).
mère(!date,misealovag,v1,[]).
mère(!date,misealovag2,v1,[]).
mère(!prisonnier,mcf,v1,[]).

objet(!bâtiment,1,vagabond,[v1]).
objet(!bâtiment,12,vagabond5,[v1,v2,v4,v5]).
objet(!bâtiment,4,vagabond2,[v1,v2]).
objet(!bâtiment,6,vagabond3,[v1]).

l'objet de surrogate 4, vagabond2 dans !bâtiment, admet v1 et v2 comme versions

objet(!bâtiment,8,vagabond4,[v1]).
objet(!coque,11,flotrapide,[v1]).
objet(!coque,3,flotvag,[v1]).
objet(!coque,5,flotvag2,[v1,v1]).
objet(!date,2,misealovag,[v1]).
objet(!date,9,misealovag2,[v1]).
objet(!prisonnier,13,mcf,[v1]).

racine(!bâtiment,vagabond2,v2).
racine(!bâtiment,vagabond2,v1).

v1 et v2 de vagabond2 sont racines d'un arbre de dérivation de cet objet

racine(!bâtiment,vagabond,v1).
racine(!bâtiment,vagabond3,v1).
racine(!bâtiment,vagabond4,v1).
racine(!bâtiment,vagabond5,v1).
racine(!coque,flotrapide,v1).
racine(!coque,flotvag,v1).
racine(!coque,flotvag2,v1).
racine(!coque,flotvag2,v1).
racine(!date,misealovag,v1).
racine(!date,misealovag2,v1).
racine(!prisonnier,mcf,v1).

C.1.3.les faits décrivants les types

```
ens(!bâtiment,struct).                    le type !bâtiment est structuré
ens(!coque,struct).
ens(!galère,isa).                         le type !galère est ISA
ens(!prisonnier,struct).

isa(!galère,!bâtiment).                   le type !galère spécialise le type
                                          !bâtiment

ensstruct(!bâtiment,                      le type !bâtiment est décrit par les
                                          attributs
[[[armateur],!chaîne],                    armateur dans !chaîne
[[misealo],!date],                        misealo dans !date
[[port],!chaîne],                         etc ....
[[flotteur],!coque],
[[propulseur],!chaîne],
[[usage],!chaîne]]).

ensstruct(!coque,
[[[long],!entier],
[[larg],!entier],
[[tirant],!entier],
[[depl],!entier],
[[nbvol],!entier]]).

ensstruct(!prisonnier,
[[[nompat],!chaîne],
[[prénom],!chaîne],
[[dateemp],!date]]).
```

```
ensstructfct(!prisonnier,                    le type !prisonnier admet un attribut
                                             calculé
[[[dateemp],!date,2]]).                      l'attribut dateemp de type !date est
                                             calculé par le lien N° 2

ensstructspe(!galère,
[[[garde],!chaîne],                          les attributs supplémentaires du type
[[galériens],list(!prisonnier,1,_x1)],       !galère sont : garde dans !chaîne,
[[nbrameur],!entier],                        etc ...
[[puissance],!entier]]).

ensstructfctspe(!galère,                     le type ISA !galère admet 2 attributs
        [[[usage],!chaîne,3],                calculés : usage par le lien de
                                                spécialisation
   [[puissance],!entier,4]]).                N°3, et puissance par le lien N°4

ensstructnfct(!bâtiment,                     les attributs non calculés du type
                                             !bâtiment
[[[armateur],!chaîne],                       sont : armateur dans !chaîne,
[[misealo],!date],                           etc ...
[[port],!chaîne],
[[flotteur],!coque],
[[propulseur],!chaîne],
[[usage],!chaîne]]).

ensstructnfct(!prisonnier,
[[[nompat],!chaîne],
[[prénom],!chaîne]]).
```

ensstructnfctspe(!galère,
[[[garde],!chaîne],
[[galériens],list(!prisonnier,1,_x1)],
[[nbrameur],!entier]]).

C.1.4.les faits décrivants les fonctions

fonction(déplace,déplace,!entier,[!entier,!entier,!entier],valide,1). *la fonction valide déplace, réalisée par le programme déplace, admet en entrée 3 !entiers et délivre un !entier.*

C.1.5.les faits décrivant les liens (attributs et versions calculés)

lien(2,[date_système,[]],lf). *le lien 2 (qui calcule un attribut), utilise la fonction date_système sans paramètre*

lien(3,[identité, *le lien 3, calcule un attribut par une simple*
[[[chaîne(répressif)]]]],ld). *affectation. L'attribut prend la valeur répressif dans !chaîne*

lienponct(2,[dilate, *le lien ponctuel N°2 (calcul d'une version),*
[[[[v1,flotvag]]], *utilise la fonction dilate avec les paramètres*
[[ent(1)]],[[ent(3)]]]],lfp). *v1 de flotvag, et les entiers 1 et 3.*

lienponct(3,[identité, *le lien ponctuel 3, exprime*
[[[flotteur],[[v1,vagabond],!bâtiment]]]],ldp). *l'instanciation d'un attribut d'une version par la valeur de flotteur, de v1 de vagabond dans !bâtiment*

surlien(4). *on a défini 4 liens dans des types*
surlp(3).*3 liens ponctuels*
surocc(16). *16 objets*

C.1.6.les faits décrivant l'utilisation des types, fonctions et versions

utilens(!coque,[dilate]). *la fonction dilate utilise le type !coque*

```
utilfct(date_système,[[lf,2]]).
utilfct(déplace,[[lf,1]]).
utilfct(dilate,[[lfp,1],[lfp,2]]).
utilfct(identité,[[ld,3]]).
utilfct(rendu-rameur,[[lf,4]]).
```

les liens fonctionnels ponctuels (lfp) utilisent le type !date

```
utilocc([[v1,flotvag],!coque],
[[ldp,[[v1,vagabond],!bâtiment]],
[lfp,[[ni,10],!coque]],
[lfp,[[v1,flotrapide],!coque]]]).
```

la version v1 de flotvag dans !coque, est utilisée par :
v1 de vagabond pour instancier un attribut
l'objet système N°10, pour sont calcul
la version v1 de flotrapide pour son calcul

```
utilocc([[v1,flotvag2],!coque],
[[ldp,[[v1,vagabond2],!bâtiment]],
[ldp,[[v2,vagabond2],!bâtiment]],
[ldp,[[v1,vagabond3],!bâtiment]]]).

utilocc([[v1,misealovag2],!date],
[[ldp,[[v1,vagabond4],!bâtiment]],
[ldp,[[v1,vagabond5],!bâtiment]]]).

utilocc([[v1,misealovag],!date],
[[ldp,[[v1,vagabond],!bâtiment]],
[ldp,[[v1,vagabond2],!bâtiment]],
[ldp,[[v2,vagabond2],!bâtiment]]]).

version-courante(!prisonnier,[v1,mcf]).
```

la version courante est v1 de mcf dans !prisonnier

C.2. La base de Connaissances

Les faits suivants sont stockés dans la base de connaissances, et ne sont pas modifiables.

C.2.1.les types prédéfinis et les types de base

ens(!réel,base). *le type !réel est un type de base*
ens(!entier,base).
ens(!bool,base).
ens (!chaîne,base).
ens(!date,struct). *le type !date est un type construit,*
ens(!montre,struct).

C.2.2.les fonctions prédéfinies

fonction (inf,inf,!bool,[!entier,!entier],valide,1) .
fonction (sup,sup,!bool,[!entier,!entier],valide,1) .
fonction (moyenne,moyenne,!entier,[list(!entier,1,_)],valide,1).
fonction (plus,plus,!entier,[!entier,!entier],valide,1).
fonction (moins,moins,[!entier,!entier],valide,1).

fonction (egalgen,egalgen,!bool,[!chaîne,!chaîne],valide,1).
fonction (égal,egalgen,!bool,[!entier,!entier],valide,1).
fonction (appartient,appartient,!bool,[!chaîne,!chaîne],valide,2). *les fonctions de type 2*
fonction (existefonct,existefonct,!bool,[!chaîne],valide,2). *applicables à tous les*
fonction (ensbutde,ensbutde,!bool,[!chaîne,!chaîne],valide,2). *types*
fonction (occbutde,occbutde,!bool,[!chaîne,!chaîne],valide,2).

fonction (date_système,date_système,!date,[],valide,1). *fonctions prédéfinies*
fonction (heure_système,heure_système,!montre,[],valide,1). *sur les dates*
fonction (depuish,depuis,!bool,[!montre,!montre],valide,1).
fonction (depuisd,depuis,!bool,[!date,!date],valide,1).
fonction (avantd,avant,!bool,[!date,!date],valide,1).
fonction (pendantd,pendant,!bool,[!date,!date],valide,1).
fonction (avanth,avant,!bool,[!montre,!montre],valide,1).
fonction (pendant,pendant,!bool,[!montre,!montre],valide,1).

C.2.3.la description des types prédéfinis

ensstruct(!date,[[[année],!entier],[[mois],!entier],[[jour],!entier]]).
ensstructnfct(!date,[[[année],!entier],[[mois],!entier],[[jour],!entier]]).

```
enscontrain(!date,[1,2,3,4]).
```
le type !date admet 4 contraintes, décrites par les faits suivants :

```
contrain(1,[inf,[[[mois],[occ]],[[ent(12)]]]],o).
```
l'attribut mois doit être inférieur (fonction inf) à l'entier 12

```
contrain(2,[sup,[[[jour],[occ]],[[ent(31)]]]],n).
```
l'attribut jour ne doit pas être supérieur à 31, etc ..

```
contrain(3,[sup,[[[année],[occ]],[[ent(1)]]]],o).
contrain(4,[sup,[[[jour],[occ]],[[ent(1)]]]],o).
contrain(11,[sup,[[[mois],[occ]],[[ent(1)]]]],o).

ensstruct(!montre,[[[heure],!entier],[[min],!entier],[[sec],!entier]]).
ensstructnfct(!montre,[[[heure],!entier],[[min],!entier],[[sec],!entier]]).
enscontrain(!montre,[5,6,7,8,9,10]).

contrain(5,[sup,[[[heure],[occ]],[[ent(1)]]]],o).
contrain(6,[sup,[[[min],[occ]],[[ent(1)]]]],o).
contrain(7,[sup,[[[sec],[occ]],[[ent(1)]]]],o).
contrain(8,[inf,[[[heure],[occ]],[[ent(23)]]]],o).
contrain(9,[inf,[[[min],[occ]],[[ent(59)]]]],o).
contrain(10,[inf,[[[sec],[occ]],[[ent(59)]]]],o).

surctr(11).
```
il y a 11 contraintes définies.

```
utilfct(inf,[[ctr,1],[ctr,8],[ctr,9],[ctr,10]]).
```
la fonction inf est utilisée par les contraintes 1, 8, 9, 10

```
utilfct(sup,[[ctr,2],[ctr,3],[ctr,4],[ctr,5],[ctr,6],[ctr,7]]).
```

ANNEXE D. QUELQUES MESURES SUR ETIC

Le tableau suivant donne la dimension des modules du système en termes de lignes et de clauses :

ETIC	LIGNES	CLAUSES
analyseur syntaxique	672	155
création d'objets	3080	579
modification d'objets	4259	714
répercussion de modifications	3670	655
manipulation de versions	2990	543
définition de types	2584	549
définition de fonctions	849	170
gestion des modules	1111	91
TOTAL	**19215**	**3456**

Le tableau suivant donne les mesures en termes de lignes des modules en C, Pascal et Prolog nécessaires à la gestion de l'interface concepteur-ETIC décrite dans le section 7 du chapitre 6 :

	LIGNES
Pascal	391
C	2409
Prolog	1144
TOTAL	**3944**

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT-INGENIEUR, DOCTORAT USTMG

Vu les dispositions de l'Arrêté du **16 avril 1974**,

Vu les dispositions de l'Arrêté du **5 juillet 1984**,

Vu les rapports de M. Claude DELOBEL

M. Michel LEONARD

Mlle FAUVET Marie Christine est autorisé

à présenter une thèse en vue de l'obtention du Doctorat de

l'Université Joseph Fourier Grenoble I, Spécialité Informatique

Grenoble, le 19 JUIL. 1988

Le Président de l'Université Scientifique Technologique et Médicale

UNIVERSITÉ SCIENTIFIQUE ET MÉDICALE DE GRENOBLE

J.J PAYAN