Un modèle et un mécanisme d'exécution pour les bases de données actives

Maria Francisca Ghira Zinho Antunes

THESE

présentée par

Maria Francisca GHIRA ZINHO ANTUNES

pour obtenir le titre de

Docteur de l'Université Joseph Fourier - Grenoble I

(arrêté ministériel du 23 novembre 1988)

Spécialité : Informatique

UN MODELE ET UN MECANISME D'EXECUTION POUR LES BASES DE DONNEES ACTIVES

Thèse soutenue le : 7 novembre 1991

Composition du jury :

Yves Chiaramella	Président
Colette Rolland	Rapporteur
Michel Leonard	Rapporteur
Michel Adiba	Examinateur
Mauricio Lopez	Examinateur

Thèse préparée au sein du Laboratoire de Génie Informatique - IMAG

RESUME :

Cette thèse propose un modèle et un ensemble de mécanismes pour supporter le développement de bases de données actives.

D'une part, le modèle permet la description intégrée des aspects statiques et dynamiques des applications. Pour décrire la structure des traitements sur les informations trois concepts sont proposés : entité focale, activité et procédure.

D'autre part, ce modèle est supporté par un ensemble d'outils et de mécanismes permettant de lancer automatiquement, contrôler et suivre l'exécution des activités et procédures.

Un des mécanismes qui a mérité une attention particulière est celui du déclenchement automatique d'une activité ou d'une procédure. Ceci implique l'évaluation de préconditions liées aux activités et aux procédures. Leur évaluation est réalisée lors des mises à jour et nécessite potentiellement des accès complexes à la base de données. Cette évaluation se fait en deux phases :

- la première détermine le sous-ensemble de préconditions à évaluer pour une certaine mise à jour. Ce travail a abouti à l'établissement d'un ensemble de règles de correspondance entre les préconditions et les mises à jour.
- la deuxième réalise l'évaluation proprement dite des préconditions. Ce travail a abouti à la définition d'une méthode d'évaluation efficace qui utilise un ensemble de techniques d'optimisation.

D'autres mécanismes à souligner sont :

- l'interaction entre les mécanismes de déclenchement et d'exécution des activités et procédures avec le système transactionnel.
- le contrôle persistant de l'exécution des procédures.

Ce travail est complété par l'implémentation des différents services proposés.

MOTS-CLE :

Bases de données actives, modèle dynamique, activité, procédure, entité focale, évaluation de préconditions, déclenchement automatique, contrôle persistant.

Je tiens à remercier,

Monsieur Yves Chiaramella, directeur du Laboratoire de Génie Informatique au sein duquel s'est effectuée cette étude, et qui a bien voulu être le président du jury.

Madame Colette Rolland, Professeur à l'Université de Paris I Sorbonne qui a accepté d'être rapporteur de ma thèse.

Monsieur Michel Léonard, Professeur à l'Université de Genève, rapporteur de ma thèse, pour l'attention particulière qu'il a dédiée à mon travail et les suggestions qu'il a su faire.

Monsieur Michel Adiba, Professeur à l'Université "Joseph Fourier" de Grenoble, mon Directeur de thèse, pour m'avoir accueillie dans son équipe de recherche.

Monsieur Mauricio Lopez, Ingénieur au Centre de Recherche BULL qui est à l'origine de cette thèse et pour les conseils qu'il m'a prodigués. Il m'a toujours accompagnée, entendue et enthousiasmée. Je garderai toujours un très bon souvenir de nos nombreuses discussions.

Monsieur Christian Lenne, Ingénieur au Centre de Recherche BULL, le grand supporteur de cette thèse à tous les niveaux : conception, mise en œuvre, texte, matériel (et des problèmes il y en a eus) et même tirage. Qu'il en soit ici vivement remercié.

Madame Esperanza Pedraza, Docteur en Informatique qui a participé au projet DOEOIS. Elle a été la grande compagne de ce travail et de tout ce qu'il représente. Qu'elle trouve ici toute ma reconnaissance.

Monsieur Roland Balter, Directeur de l'antenne du Centre de Recherche BULL à Grenoble, pour sa présence stimulante et ses qualités humaines.

Monsieur Serge Rouveyrol, Ingénieur CNRS au LGI, pour son grand soutien technique. Lui et Mike Hollett, que je remercie également, m'ont toujours fait prendre du recul dans mon travail.

L'ensemble des participants du projet DOEOIS et plus particulièrement Monsieur Samer Haj Houssain. Mes collègues Olga Marino et Rodrigo Scioville toujours prêts à m'aider.

Jean-Pierre Person le grand sponsor de cette thèse qui a toujours su, par son calme, accompagner mon travail.

Que Annette, Francisco et tous les Ghira Zinho Antunes soient ici très chaleureusement remerciés.

... e obrigado Rita !

P.S. : Que le travail laborieux de Mesdames Riminy, Bert, d'Hollanda, Staukausser, Gulavar et Bessecani soit ici mis en valeur.

TABLE DES MATIERES

CHAPITRE 1

1. INTRODUCTION

Cette thèse a été réalisée dans le contexte de la conception et de l'implémentation d'un **Serveur d'Information pour la Bureautique** qui offre un support intégré pour la manipulation des aspects statiques et dynamiques des applications. L'aspect statique permet de décrire et manipuler les informations. L'aspect dynamique concerne les processus de manipulation des informations, en considérant leur modélisation, activation, exécution et suivi. C'est sur cet aspect - **la dynamique des applications** - que mon travail s'est concentré.

Plusieurs nouvelles lignes de recherche caractérisent ce travail. D'un côté nous avons la modélisation du comportement des applications qui est à l'origine des **SGBDs dynamiques**, et le déclenchement automatique des actions sur la base de données qui est le point de départ des **SGBDs Actifs**. D'un autre côté nous sommes concernés par un domaine d'application particulier - la bureautique - et en conséquence par les **SGBDs orientés bureautique**.

Ces trois concepts sont la base d'une nouvelle approche dans le domaine des bases de données ; dans cette introduction ces concepts seront présentés (sections 1.1 et 1.2). Après quoi, les problèmes propres à la conception d'un serveur d'information pour la bureautique seront énumérés ainsi que les solutions proposées (section 1.3). Ceci permet d'avoir une vue globale du contexte dans lequel ce travail s'insère. Finalement, l'introduction décrira la spécificité de mon apport et le plan de la thèse (section 1.4).

1.1. Des SGBDs Statiques Passifs aux SGBDs Dynamiques Actifs

Les concepts de SGBDs dynamiques sont apparus par opposition aux SGBDs statiques et ceux de SGBDs actifs par opposition aux SGBDs passifs [Lei79]. Les raisons qui mènent à la conception de ces nouveaux types de SGBDs viennent d'un souci de modélisation et d'automatisation du comportement des applications (voir figure 1.1).

Les premiers SGBDs ont été conçus pour des applications de gestion caractérisées par de grandes quantités de données non complexes (entier, réel, caractère, ...) à gérer et par un ensemble relativement petit de traitements simples sur ces données. Ceci implique que **l'approche traditionnelle** met en valeur la modélisation des **propriétés statiques** des applications, i.e., les objets[1] des applications, les propriétés de ces objets et les relations entre ces objets. La modélisation des propriétés dynamiques telles que les actions à réaliser sur les objets et les relations entre ces actions sont alors laissées de côté.

[1]Le mot objet fait ici référence à n'importe quoi - une chose, une donnée, une information,... - du monde réel qu'on veut modéliser.

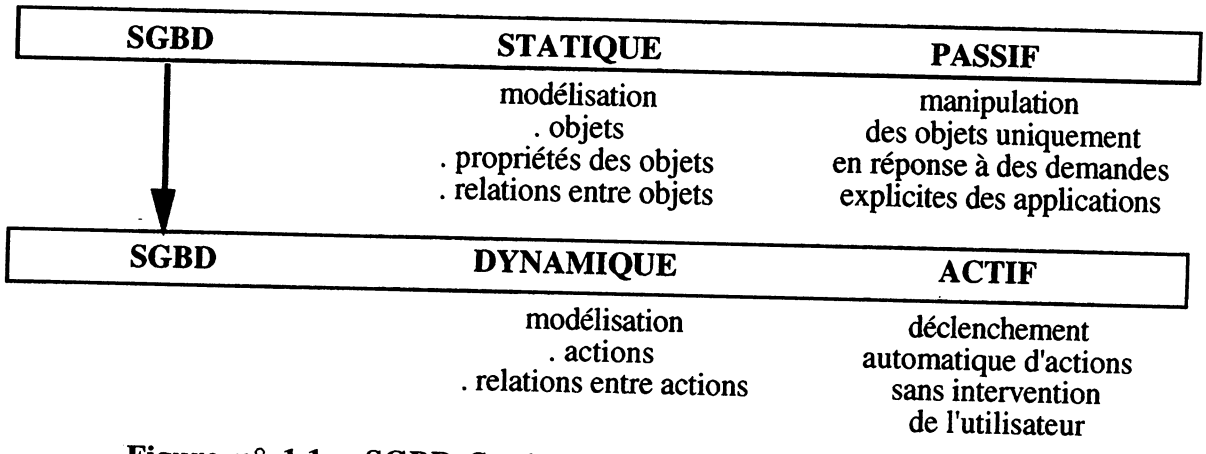


Figure n° 1.1 - SGBD Statique Passif - SGBD Dynamique Actif

Suivant cette approche, la conception d'une base de données est faite en trois temps : d'abord il est nécessaire d'extraire les propriétés structurelles des applications ; ensuite il faut concevoir un schéma pour représenter ces propriétés ; et finalement, les requêtes et programmes sur ce schéma sont réalisés. Les propriétés dynamiques sont alors définies en utilisant les primitives d'insertion, de suppression et de modification, et des abstractions procédurales qui ne sont pas incluses dans les modèles de données. Pour relier les aspects statiques et dynamiques, de nouvelles contraintes d'intégrité sont définies indépendamment des programmes et des schémas. Ces contraintes sont implémentées en utilisant des assertions, des déclencheurs ("triggers"), des procédures b.d., des conditions de vérification ("on-conditions", "check-clauses"), etc. Ces techniques implémentent les contraintes comme des effets secondaires des opérations. Ainsi, dans l'approche traditionnelle, les propriétés dynamiques sont spécifiées sous différentes formes et dans plusieurs lieux. Ceci complique la conception, la définition et l'analyse de l'intégrité sémantique [Bro81].

Cette approche diffère beaucoup de la vue des **langages de programmation** qui met en valeur **l'aspect dynamique**. Il est clair maintenant, que les deux aspects (statique et dynamique) doivent être traités au même niveau conceptuel et qu'ils sont nécessaires pour décrire les applications de façon complète. Ainsi, un des buts de la recherche actuelle est d'**intégrer** les modèles de données et des concepts, des outils et des techniques des langages de programmation pour spécifier de façon complète une base de données représentative d'une application. Des mécanismes d'abstraction sur les propriétés dynamiques sont alors proposés. Les principales adjonctions sont des formes de contrôle procédural avec lesquelles il est possible de composer des opérations abstraites (par comparaison aux abstractions sur les données). Dans ACM/PCM [Bro81] Brodie propose trois types d'abstractions de contrôle procédural qu'il compare à trois types d'abstractions de données : séquence / agrégation , choix / généralisation , répétition / association.

Dans ce cadre, des modèles ont été développés intégrant les propriétés statiques (structurelles) et dynamiques (de comportement). Dans le schéma conceptuel d'une application les propriétés des classes[2] des objets sont définies de façon complète d'une part, par la spécification de leur

[2]Nous utilisons le mot classe pour désigner une abstraction d'un ensemble d'objets du monde réel perçus comme

structure, et d'autre part, par la spécification de leur comportement pour chaque action valable sur ses objets. Le concept de hiérarchie de généralisation est aussi développé dans ces modèles : les classes peuvent se spécifier en héritant des propriétés structurelles mais aussi de celles de comportement. Ceci permet une conception aisée du schéma conceptuel des applications par des étapes de raffinement. Cette approche a été fortement influencée par les langages orientés objets et est illustrée aujourd'hui par plusieurs prototypes de SGBDs dits "Orientés Objets" : ORION [Bal88], O2 [Ban88], IRIS [Bee87], GEMSTONE [Mai86].

Par ailleurs, les SGBDs traditionnels sont considérés comme passifs car ils ne manipulent les données qu'en réponse à des **demandes explicites** des applications. Par opposition, un SGBD actif permet aux utilisateurs de spécifier des actions à **activer automatiquement**, sans l'intervention de l'utilisateur, quand certaines conditions sont remplies. Remarquons que la notion de SGBD dynamique est inclue dans celle de SGBD actif vu que les SGBDs actifs doivent également se charger de la gestion du comportement des applications.

Les concepts qui sont à l'origine des SGBDs actifs sont aussi les "triggers", les "on-conditions", ... qui, comme il a été dit, étaient utilisés au départ pour déclencher la vérification des contraintes d'intégrité. C'est avec les besoins des nouvelles applications telles que la bureautique (où le côté automatisation des tâches est très important) et la CAO (où les objets ont une structure très complexe et ont beaucoup de contraintes de dépendance entre eux) que nous voyons apparaître la notion de SGBD actif.

Dans ce contexte, des **généralisations de la notion de "trigger"** sont proposées [Day89] [Dit86] [Dit88] pour lesquelles les événements qui permettent de déclencher des actions sur la b.d. ainsi que le contenu de ces actions, ne sont pas limités aux simples primitives d'insertion, de suppression et de modification.

Dans le projet Hipac [Day86] la notion de **règle ECA** ("Event-Condition-Action") crée de nouvelles formes d'interaction entre les applications et le SGBD : les programmes d'application signalent des événements et le SGBD en exécutant les règles, lance d'autres programmes d'application. Ceci crée un nouveau paradigme pour construire des applications sur un SGBD : le contrôle logique est spécifié dans les règles ECA.

Un effort a été fait concernant la modélisation et la vérification automatique des contraintes de relation entre les différentes actions entreprises sur les objets b.d.. Plusieurs projets sont le résultat des recherches faites dans ce sens, par exemple, RUBIS [Lig87], TAXIS [Chu87]. TAXIS offre un modèle de données du type Entité-Association qui supporte les notions de classification, généralisation et agrégation. Quatre types d'entité sont considérés : donnée, transaction, exception et script. Les **scripts** sont considérés comme des entités actives permettant la modélisation de longs processus comme par exemple établir un contrat d'emploi. Leur définition est faite par un réseau de Petri augmenté : (i) des assertions liées aux différents états ; (ii) des conditions d'activation et des listes d'actions associées aux transitions ; (iii) des primitives de communication inter-processus. Un apport important de ce projet a été le développement d'un mécanisme d'évaluation automatique des assertions et des conditions d'activation.

---

ayant des propriétés communes.

In fine, pour faire la conception et l'implémentation d'un SGBD dynamique et actif il faut offrir :

1. Des moyens de définition de classes d'objets dynamiques au niveau du schéma conceptuel des applications. Ceci nécessite le développement d'abstractions de contrôle sur les propriétés dynamiques permettant de spécifier les relations entre les différentes actions valables sur la base de données.

2. Des mécanismes efficaces qui soient capables de contrôler automatiquement l'exécution correcte des actions, et de suivre l'évolution de la base de données de façon à détecter ses états révélateurs du lancement de groupes d'actions.

1.2. Informatisation du Milieu Bureautique

La bureautique est un domaine d'application qui a nécessité le développement de nouvelles technologies b.d. à cause des caractéristiques de l'information circulant dans un bureau et des tâches qui y sont réalisées. Toutefois les conceptss et les principes utilisés dans ce domaine s'appliquent à d'autres milieux ayant des besoins similaires. Ces besoins se caractérissent par :

- L'information circulant a différents types incluant des données multimédia (texte, voix, ...).

- Les tâches réalisées dans le bureau peuvent être bien définies, connues d'avance et très structurées, ou très aléatoires et peu structurées, ceci dépendant beaucoup de l'intervention de l'employé de bureau.

- Les tâches bureautiques peuvent être de courte durée ou très longues nécessitant l'intervention, en parallèle ou non, de plusieurs personnes (sécrétaires, administrateur, ...) dans des lieux différents et utilisant divers types d'outils.

- Le milieu bureautique dépend d'évènements intérieurs mais aussi extérieurs à son organisation. D'autre part, le travail pratiqué et les données multimédia manipulées ne correspondent pas à des standards précis. Par conséquent cet environnement est considéré très évolutif.

- Le suivi des tâches bureautiques est un aspect très important. Ceci doit aider l'employé de bureau à organiser son travail et à prendre des décisions.

L'informatisation des bureaux est aujourd'hui caractérisée par deux générations [Tsi85]. La **première génération** offre des outils d'automatisation ou d'aide à la réalisation d'une activité particulière, néanmoins il ne substituent ni éliminent l'activité en cause. Quelques exemples de ces outils sont : traitements de textes, tableurs, utilisation de bases de données, messagerie électronique, etc... Le problème le plus important de cette génération est l'intégration des différents outils proposés. La **deuxième génération** essaie de voir le bureau comme un tout et propose des

méthodes d'automatisation des tâches qui y sont réalisées. Nous mettons l'accent sur deux des outils développés pendant cette génération : les procédures de bureau et les modèles.

Procédures de Bureau

Pour automatiser le bureau, nous avons besoin d'automatiser les procédures qui y sont réalisées. Une procédure de bureau a été définie dans [Tsi85] comme une **combinaison d'étapes** qui atteignent un but général : une étape d'une procédure est une **activité** réalisée pendant une période de temps par une personne ou un groupe de personnes bien défini. Les outils utilisés pour décrire chaque étape d'une procédure et spécifier la façon de contrôler son exécution définissent plus exactement une procédure de bureau.

Dans le même article [Tsi85], un exemple illustre l'automatisation d'une tâche bureautique : la manipulation d'une lettre. Les outils de base (première génération) pour réaliser une telle tâche sont par exemple le traitement de texte et la messagerie électronique. Il faut ensuite des outils pour spécifier cette tâche en termes de procédures de bureau : ils doivent permettre la définition des différentes étapes, par exemple écriture de la lettre, correction, remplissage d'un message, signature, envoi, etc.

Les procédures de bureau ont beaucoup de parallélisme et d'interaction humaine. Le contrôle du parallélisme nécessite des outils permettant la définition d'**événements** et leur **coordination**. De plus, le contrôle de l'exécution d'une procédure doit être **persistant**, i.e. l'exécution d'une procédure peut être suspendue et reprise plus tard au point approprié. Un outil qui permet de **spécifier** et d'**exécuter** les procédures automatisables est essentiel en bureautique.

Modèles

Il y a deux grandes catégories de modèles : les modèles de conception et les modèles d'analyse [Tsi85].

Les **modèles de conception** ont un pouvoir de description très riche et offrent un éventail d'outils pour saisir les objets du bureau et leur comportement. Ces modèles mettent l'accent sur un aspect des activités bureautiques. Certains donnent de l'importance à la structure interne des objets, comme dans les modèles de documents [Bog83] [ECM85] [Rab85] ou dans les modèles de données pour la bureautique [Gib84]. D'autres se concentrent sur la description d'évènements et de leurs relations.

Chacun de ces modèles permet d'avoir une vue particulière du bureau. Ils proposent rarement des moyens d'automatisation en termes de procédures de bureau exécutables : ces modèles visent la **description** alors que les facilités des procédures de bureau visent une description exécutable [Tsi85].

Les **modèles d'analyse** voient le bureau de façon encore plus étroite. Ils essayent de faire correspondre un aspect du bureau particulier et très bien défini à un modèle connu (automate d'états finis, réseaux de Petri [Ric84], ...). Ces modèles proposent un grand éventail de résultats théoriques

qui peuvent être utilisés pour analyser le comportement du bureau.

Ces modèles mettent l'accent sur la **modélisation** et n'offrent pas de facilités pour définir des procédures de bureau et un système capable de les exécuter automatiquement.

L'objectif de la recherche actuelle en systèmes d'information pour la bureautique est l'**intégration**. Le contexte général de cette thèse rentre dans ce cadre en visant la conception et l'implémentation d'un serveur d'information pour la bureautique qui intègre la description des informations de bureau, les procédures de bureau et des tâches bureautiques non automatisables. Par la suite, je me place dans le contexte du projet Esprit DOEOIS - "Design and Operational Evaluation of Office Information Servers" - pour permettre de donner un aspect plus concret aux différents problèmes qui ont du être résolus afin de concevoir et réaliser un tel serveur. Le paragraphe suivant énumère ces problèmes et les solutions proposées.

1.3. Un Serveur d'Information pour la Bureautique

Pour informatiser le monde du bureau, il est clair maintenant que nous avons besoin : (i) des facilités offertes par les SGBDs pour modéliser les données, les stocker, les manipuler, ... ; (ii) d'un langage qui ait un grand pouvoir de description permettant la spécification des procédures de bureau et qui soit immédiatement exécutable ; (iii) d'offrir des moyens de contrôle persistant de l'exécution des procédures de bureau.

Remarquons que le premier point est très important vu que les informations d'un bureau doivent être stockées de façon persistante dans une base de données et que ce sont ces informations qui seront manipulées par les procédures de bureau. De plus, les conditions d'activation des procédures ainsi que le démarrage automatique des activités à l'intérieur d'une procédure, dépendent de l'état et des changements de ces informations.

Au départ plusieurs types de langages étaient susceptibles d'assurer l'écriture des tâches bureautiques. Ces langages n'ont aucune dépendance vis-à-vis du domaine bureautique mais paraissent adaptés aux exigences de ce milieu. Ces langages sont : langages de programmation conventionnels, langages de programmation base de données (p.ex. DBPL [Mat89], Galileo [Alb85], ADABTPL [Feg89], Pacal/R), langages de programmation persistants (p.ex. PS-Algol [PPR87], Napier [Atk89] [Bro89]), langages orientés modélisation des bureaux (p.ex. "Information Control Nets" [Ell79], "Predicate Transition Networks" [Ric84], "Office Analysis Methodology" [Sir82]). Le tableau de la figure 1.2 résume la façon dont chacun de ces langages répond aux exigences de base.

Ce tableau montre que les langages existants n'offrent pas les fonctionalités suffisantes pour gérer les activités et les procédures bureautiques. La solution retenue a donc été de proposer un langage permettant la description de longues séquences de travail qui soit intégré à un modèle de données particulier et d'implémenter un système capable d'exécuter, de suivre et de contrôler l'exécution des procédures de bureau.

LANGAGES / BESOINS	Langages de programmation conventionnels	Langages de programmation base de données	Langages de programmation persistants	Langages orientés modélisation des bureaux
Pouvoir de description de longues séquences de travail	o	-	o	+
Facilités des SGBDs	-	+	-	-
Contrôle persistant de l'exécution des procédures de bureau	-	-	+	-

\+ approprié o adaptable - n'assure pas

Figure n° 1.2 - Langages Candidats

Le modèle de données supporté est un modèle sémantique fonctionnel appelé FM (Fact Model) [DOE86c] : son langage sous-jacent est de haut niveau et s'appelle FML [Lop87].

Le langage développé pour décrire les relations entre les différentes étapes des procédures - OPL3 [DOE87a] - a été influencé par les langages orientés modélisation des bureaux qui sont très appropriés à la description de longues séquences de travail. Deux remarques méritent d'être faites sur ce choix :

- Les langages de programmation base de données conduiraient à un système dans lequel il ne serait pas possible de modéliser les procédures de bureau comme un tout, comme une unité cohérente ; elles seraient plutôt représentées par une collection d'activités associées à des déclencheurs ("triggers"). Pourtant, il est essentiel pour le contrôle et le suivi de l'exécution des procédures de bureau de les voir comme de longues séquences de travail, plutôt que comme des réactions individuelles à des événements. Entre autre, le contrôle persistant de l'exécution des procédures serait alors mis en défaut.

- Les langages de programmation persistants utilisent un système dans lequel les objets actifs (dans le cas du bureau, les activités et les procédures) sont représentés par des processus lourds. Or, potentiellement il y a beaucoup d'activités et de procédures actives dans un bureau. Il semblerait plus approprié de choisir un système qui créerait un processus (si nécessaire) pour chaque activité qui s'exécute vraiment et qui représenterait les autres activités (actives mais ne s'exécutant pas) et les procédures par des données persistantes.

Par la suite, nous énumérons les problèmes rencontrés dans la conception et l'implémentation du Serveur ; l'accent est mis sur l'aspect dynamique. Les solutions proposées, leurs avantages et les domaines - disciplines - qu'elles concernent sont également présentés. Ces problèmes sont groupés

en différentes classes : conception générale du modèle, exigences générales du modèle de procédures, concept d'activité de bureau, concept de procédure de bureau.

Conception Générale du Modèle

Problème de modélisation : Intégration des notions d'activité et de procédure (avec leurs fonctions de manipulation sous-jacentes) et du modèle de données (avec ses outils sous-jacents, p.ex. langage de manipulation, système transactionnel, ...).
Solution proposée : L'approche suivie dans l'intégration des activités et des procédures au modèle de données, a été de les considérer comme des entités spéciales des applications, i.e. de les modéliser en termes des concepts du modèle de données (entité et fait). L'approche alternative était de proposer un modèle conceptuel qui incluse les notions d'activité et de procédure au même niveau que les notions de modélisation de données. Cette dernière approche n'a pas été retenue : elle impliquerait une faible intégration des concepts, et les concepts à manipuler seraient nombreux ; elle conduirait à un modèle rigide empêchant une combinaison flexible des activités / procédures avec les autres entités des applications ; et finalement l'implémentation du système correspondant serait rigide et complexe .
Ainsi, il est préférable d'introduire les activités / procédures comme des classes d'entités dynamiques dont les instances correspondent aux tâches à exécuter. Ces classes sont prédéfinies en utilisant le LDD standard, néanmoins elles ont une sémantique particulière, i.e., elles sont traitées de façon spéciale dans l'implémentation.
Avantages : (i) Les activités et les procédures peuvent être manipulées en utilisant les LDD et LMD standards offerts par le système, ceci réduit le nombre de constructeurs et de concepts du langage. (ii) La complexité de l'implémentation est réduite car les données concernant les activités et les procédures sont manipulées en utilisant les mêmes mécanismes que les autres types de données. (iii) Le contrôle persistant des informations concernant les activités et les procédures est assuré de façon naturelle, vu que toute information contenue dans la base de données est persistante.
Désavantages : Dans la solution prise, les concepts d'activité et de procédure sont incorporés au niveau du modèle de façon indirecte : en effet, un seul concept est offert, celui d'entité. Par contre, la solution écartée offre un canevas uniforme pour la modélisation des données et des processus de manipulation de ces données.
L'introduction d'un nouveau concept à l'aide de constructeurs déjà existants peut être contraignante.
Domaines : modélisation des aspects dynamiques, intégration des aspects statiques et dynamiques

Exigences Générales du Modèle de Procédures

Première exigence : Le serveur doit supporter la définition des différents types de travail, des plus structurés aux plus indéfinis. De plus, il doit permettre une définition progressive et modulaire des applications.
Solution proposée : Le modèle proposé et le support d'exécution correspondant offrent trois niveaux différents de spécification du travail réalisé dans un bureau. Dans le premier niveau, il n'existe que des activités individuelles, les activités sont associées à une précondition d'activation : quand cette précondition devient vraie l'activité est habilitée à être exécutée. A ce niveau, les activités

ne sont pas associées à un corps d'exécution ; le serveur offre uniquement des informations sur l'état des activités. C'est donc à la charge de l'utilisateur d'exécuter les opérations correspondant au corps de l'activité. Dans le deuxième niveau, une caractéristique de plus est offerte : un programme est associé à chaque activité. Ce programme est considéré comme une boîte noire, il peut être écrit dans n'importe quel langage de programmation, et peut être déclenché automatiquement par le système ou manuellement par l'utilisateur. Dans le troisième niveau, la notion de procédure existe : les activités peuvent être groupées dans une procédure et les relations entre ces activités établies.
Avantages : Le modèle offre une progression vers les niveaux supérieurs : les activités définies dans le premier niveau sont compatibles avec celles de plus haut niveau. Ainsi, une approche modulaire est offerte pour l'aide à la conception des procédures.
Les applications non structurées, très prédominantes dans l'environnement bureautique, peuvent être modélisées par des ensembles d'activités (premier et deuxième niveau) plus au moins reliées entre elles à travers leurs préconditions. Par ailleurs, les applications bien structurées peuvent être modélisées à travers la notion de procédure (troisième niveau). L'approche à trois niveaux supporte ainsi la modélisation d'un éventail d'applications, allant des moins structurées au plus structurées.
Enfin, les différentes combinaisons des trois niveaux offrent la possibilité de construire une gamme de sytèmes selon les besoins des applications.
Domaines : modélisation de tâches bureautiques structurées et non structurées.

Deuxième exigence : Offrir un ensemble d'énoncés permettant la manipulation des activités / procédures, i.e., démarrer une activité, annuler une activité, suspendre une procédure, ...
Solution proposée : Toujours avec le souci d'intégrer le modèle des procédures au modèle de données, un ensemble de primitives de manipulation des procédures de bureau ayant une syntaxe très similaire au LMD a été défini (OPCL [DOE87a] pour Office Procedure Control Language). Une approche alternative aurait été de proposer une syntaxe du type langage de programmation (appel de procédures) mais ceci n'aurait pas offert le même pouvoir d'expression et la même flexibilité.
Avantages : (i) L'utilisateur est familiarisé avec le langage ; (ii) Un grand pouvoir d'expression et de flexibilité sont offerts ; (iii) Un des effets de ces primitives est de changer les informations - stockées dans la base de données - concernant l'état des activités / procédures : ceci est très facilement mis en œuvre vu que les primitives sont directement traduisibles en termes d'énoncés du LMD.
Domaines : manipulation de classes d'entités dynamiques.

Troisième exigence : Offrir des mécanismes et des outils permettant l'interaction de l'utilisateur avec l'état des procédures de bureau. Ceci doit permettre aux utilisateurs de faire le suivi des procédures de bureau.
Solution proposée: Vu que les activités et les procédures de bureau sont considérées comme des entités du schéma conceptuel, les informations sur leur état peuvent être obtenues en consultant la base de données avec le LMD standard. De plus, une interface graphique [Hus89] est proposée pour suivre les différents états des activités / procédures. Cette interface a deux modalités : la première permet l'interrogation de la base de données sur l'état des activités / procédures ; la deuxième permet de suivre en "temps réel" la vie d'une activité / procédure.

Avantages : L'interface guide l'employé de bureau dans son travail, l'aide à s'organiser et à prendre des décisions. Les facilités de consultation aident également l'utilisateur à construire ses procédures de façon progressive.
Domaines : Interface utilisateur / graphique.

Concept d'Activité de Bureau

Problème de modélisation : Spécification de l'information critique manipulée par l'activité et du contexte dans lequel une activité doit être exécutée.
Solution proposée : La solution retenue est basée sur le concept de "Focal Object" propre au modèle OAM [Sir82] [Ham83]. Il définit l'objet focal comme un objet base de données à travers lequel tous les objets manipulés par la procédure de bureau peuvent être accessibles en utilisant les associations définies dans le schéma conceptuel de l'application.
Dans le modèle proposé, une activité est associée à une classe d'entités particulière : l'entité focale. L'entité focale est le centre de l'information manipulée par l'activité. De plus, à chaque activité est associé un contexte d'activation ; il définit l'état des informations concernées par la précondition au moment où l'activité devient habilitée à être exécutée (i. e., la précondition est validée).
L'entité focale et le contexte d'activation font partie de la spécification de la précondition de l'activité. L'évaluation d'une précondition doit déterminer l'instance de l'entité focale et du contexte dans lequelle les activités habilitées en conséquence doivent s'exécuter.
Avantages : La notion d'entité focale permet une bonne intégration des aspects statiques et dynamiques des applications : le comportement d'une classe d'entités est en fait défini par l'ensemble des activités qui ont pour entité focale cette classe.
Pour certains types de précondition (préconditions qui reflètent un changement de la base de données) les entités référencées par la précondition peuvent changer de valeur pendant le temps écoulé entre le moment où la précondition devient vraie - et par conséquent l'activité est habilitée à être exécutée - et le moment où son exécution démarre. Le fait de garder le contexte d'activation permet d'accéder l'état de ces entités lors de l'activation et évite donc de verrouiller ces entités (bien sur l'état actuel de ces entités est également accessible). Ceci est très important car la période de temps pendant laquelle les entités devraient être verrouillées peut être assez longue ce qui nécessiterait des verrous de longue durée et empêcherait la base d'évoluer.

Problème de langage : Spécification des préconditions propres au déclenchement d'une activité. Ces préconditions doivent refléter des états et des changements de la base de données.
Solution proposée : Le langage de spécification de préconditions est un sur-ensemble du LMD.
Avantages : (i) L'utilisateur est familiarisé avec le langage. (ii) Les préconditions sont, en conséquence, spécifiées à un haut niveau sémantique, ainsi elles peuvent être exprimées de façon compacte et claire. (iii) La correspondance entre une précondition et une mise à jour (quelles préconditions doivent être évaluées pour une mise à jour) devient facile à établir au niveau sémantique et au niveau implémentation. (iv) Certains modules utilisés dans la compilation d'un énoncé du LMD peuvent être utilisés dans la compilation d'une précondition.
Désavantages : L'utilisation d'un même langage entraîne que certaines expressions ont une sémantique légèrement différente suivant le contexte dans lequel elles sont utilisées et ceci peut poser

des problèmes à l'utilisateur.
De plus, si un langage complètement nouveau était défini pour spécifier les préconditions, cela permettrait d'offrir plus de liberté pour les spécifier.
Domaines : Spécification de contraintes d'intégrité, de déclencheurs et de préconditions [Ham75].

Problème de sémantique : L'exécution d'une mise à jour peut provoquer l'évaluation d'un ensemble de préconditions. Bien que les préconditions soient exprimées en termes d'énoncés du LMD, leur correspondance avec les mises à jour n'est pas purement syntaxique. Ceci est dû à l'utilisation d'un modèle sémantique et un langage de haut niveau. Ainsi, établir cette correspondance nécessite de l'inférence au niveau sémantique, en particulier la hiérarchie de généralisation doit être prise en compte.
Solution proposée : La définition précise des énoncés de mise à jour et des préconditions s'est imposée. Un ensemble de règles de correspondance [Ant89b] a alors pu être défini. La méthode de base utilisée pour établir ces règles est la comparaison entre le type des entités spécifiées dans la mise à jour et le type des entités concernées par la précondition, ainsi que le type de l'opération de la mise à jour (insertion, suppression, modification) et le type de l'opération spécifiée dans la précondition.
Avantages : Les règles de correspondance établissent de façon formelle la sémantique des préconditions et sont la base de leur traitement. Comme il a été dit, les classes d'entités auxquelles une activité s'applique sont définies dans la précondition, d'une part par la spécification de l'entité focale et d'autre part par le contexte d'activation. Les règles de correspondance, en prenant en compte la hiérarchie de généralisation, induisent un héritage d'activités à travers la hiérarchie.
Domaines : Intelligence artificielle ("pattern-directed invocation") [Hew72] ; héritage à travers une hiérarchie de généralisation.

Problème d'implémentation : L'évaluation des préconditions est une partie très sensible du système. L'évaluateur peut devenir le goulot d'étranglement du système. Ceci est lié au fait qu'une base de données pour la bureautique peut être relativement importante. Par conséquent le schéma conceptuel des applications est assez complexe, spécifiant beaucoup de types de données avec leurs propriétés et associations. L'évaluation des préconditions nécessite, potentiellement, des accès complexes et fréquents à la base chaque fois qu'une mise à jour est réalisée (mise à jour qui peut modifier beaucoup de données !). Il faut donc minimiser le temps d'évaluation.
Solution proposée : Un mécanisme d'évaluation des préconditions [Ant89b] a été développé. Ses principales caractéristiques sont : (i) compilation des préconditions au moment de leur définition pour améliorer les performances lors de leur évaluation ; (ii) évaluation des préconditions intégrée à l'exécution des mises à jour pour minimiser le nombre de préconditions à évaluer et le nombre d'accès à la base de données ; (iii) évaluation des préconditions coordonnée avec la confirmation des transactions pour éviter l'habilitation d'activités en cas d'annulation.
Domaines : Evaluation de prédicats, vérification des contraintes d'intégrité [Ant87b], suivi des valeurs des objets b.d., plus généralement bases de données actives.

Problème d'exécution : Pour qu'une activité s'exécute correctement il faut : (i) offrir à l'activité

les moyens d'accéder à son contexte d'activation ; (ii) identifier le programme qui lui est associé et l'exécuter sur le site souhaité ; (iii) offrir des moyens de contrôle d'accès concurrents et un mécanisme de reprise.

Solution proposée : Un gestionnaire d'exécution [Don89] s'occupe du démarrage et de l'exécution des activités. Au moment du démarrage d'une activité, ce gestionnaire doit consulter le contexte d'activation et le passer comme paramètre au corps de l'exécution de l'activité. Le contexte d'activation a été stocké dans la base de données lorsque la précondition de l'activité est devenue vraie.

Ce gestionnaire lance l'exécution de l'activité basé sur un modèle de lancement des activités dans des sites distants. Il a aussi pour charge de stocker dans la base de données l'état des activités (lancée, finie, annulée, ...).

Comme toutes les applications, les activités peuvent utiliser le support transactionnel du serveur, néanmoins une restriction est faite : une transaction doit être entièremment incluse dans une activité. Les raisons qui mènent à imposer une telle restriction sont : (i) les activités limitent de façon naturelle des unités cohérentes de travail ; (ii) Il n'est pas souhaitable qu'une procédure de bureau soit reprise (défaite) entièrement lorsqu'une défaillance survient pendant son exécution, i. e. une défaillance ne doit en aucun cas produire l'annulation des activités déjà effectuées par la procédure [Ped88].

Les activités peuvent utiliser également le support de verrouillage [Ped88] du serveur. De plus, le gestionnaire d'exécution, au moment du démarrage d'une activité, verrouille les entités spécifiées dans la précondition et transmet ces verrous à l'activité.

Domaines : Gestion de processus, appel de procédures à distance, vue de la base de données, mécanismes transactionnels, mécanismes de verrouillage.

Concept de Procédure de Bureau

Problème de langage : Spécification des activités et des contraintes de précédence entre elles.

Solution proposée : Bien qu'il soit souhaitable de minimiser le nombre de constructeurs spéciaux du langage de définition, un sous langage pour décrire la structure des procédures comme un graphe orienté a été introduit. Ce sous langage est constitué d'un ensemble d'énoncés permettant le séquencement, le parallélisme, le choix et la répétition de l'exécution des activités. Il permet donc de spécifier les contraintes de précédence des activités à l'intérieur d'une procédure.

La description de la structure d'une procédure est appelée "schéma de la procédure".

Avantages : Ce sous-langage permet de suivre et de contrôler les procédures de bureau comme des unités cohérentes, plutôt que comme des collections d'activités liées par des préconditions.

Domaines : Description de (longues) séquences de travail.

Problème de modélisation : Passage de l'information d'une activité à une autre à l'intérieur d'une procédure.

Solution : Ce problème a été résolu à travers le concept d'entité focale. Une procédure de bureau est définie pour une certaine entité focale, les entités focales des activités qui la composent doivent être associées à l'entité focale de la procédure. La spécification des activités qui composent une procédure doit inclure les liens qui existent entre leur entité focale et celle de la procédure.

Avantages : Toutes les données manipulées par les différentes activités à l'intérieur de la

procédure sont accessibles à travers le schéma conceptuel.
Désavantages : Une certaine rigidité concernant les entités focales des activités à l'intérieur d'une procédure de bureau.
Domaines : Gestion du flux de données.

Problème d'exécution : Contrôle persistant de l'exécution d'une procédure.
Solution proposée : Toutes les informations concernant les activités / procédures sont stockées dans la base de données, en particulier, les informations sur l'état (habilitée, démarrée, finie, suspendue, ...) des activités / procédures. S'il y a au moins une des activités d'une procédure qui est en train de s'exécuter, le gestionnaire d'exécution tient à jour une table en mémoire centrale qui reflète l'état des différentes activités de la procédure. Cette table est initialisée en consultant les informations d'état stockées dans la base de données et est mise à jour, en même temps que la base de données, par le gestionnaire d'exécution chaque fois qu'une activité change d'état. A chaque primitive de manipulation des activités / procédures correspond un traitement qui manipule cette table dans le but de vérifier les contraintes de précédence.
Avantages : Le fait que les instances des activités et procédures soient stockées dans la base de données avec toutes les informations qui leurs sont associées, semble une façon naturelle d'assurer le contrôle persistant. De plus, ces instances sont accessibles par le LMD (partie interrogation) standard offert par le système.

Nous venons de survoler la façon de traiter les aspects dynamiques dans un serveur d'information pour la bureautique qui offre les fonctions de base pour la manipulation intégrée des données classiques, des documents et des tâches. Ce serveur peut être utilisé comme un service de base pour construire des systèmes bureautiques particuliers qui incluent des définitions de schéma, des interfaces utilisateur, etc. Bien que développées dans le cadre d'un serveur d'information pour la bureautique, les fonctions de gestion des tâches sont applicables à d'autres domaines.

1.4. La Thèse

Cette thèse traite du problème énoncé dans les paragraphes précédents en présentant un modèle qui permette la description des activités et procédures bureautiques et un système capable de faire leur gestion (exécution, suivi, etc...).

La figure 1.3 illustre les trois niveaux nécessaires à la modélisation et gestion des applications bases de données. Le niveau conceptuel offre des méthodologies de conception permettant la construction d'une représentation des applications. Le niveau opérationnel est donné par un modèle proposant une base formelle au développement des outils et techniques nécessaires à la mise en œuvre ; il permet ainsi de spécifier formellement les représentations faites des applications. Finalement, l'implémentation de ces outils et techniques est accomplie par le SGBD formant le niveau fonctionnel.

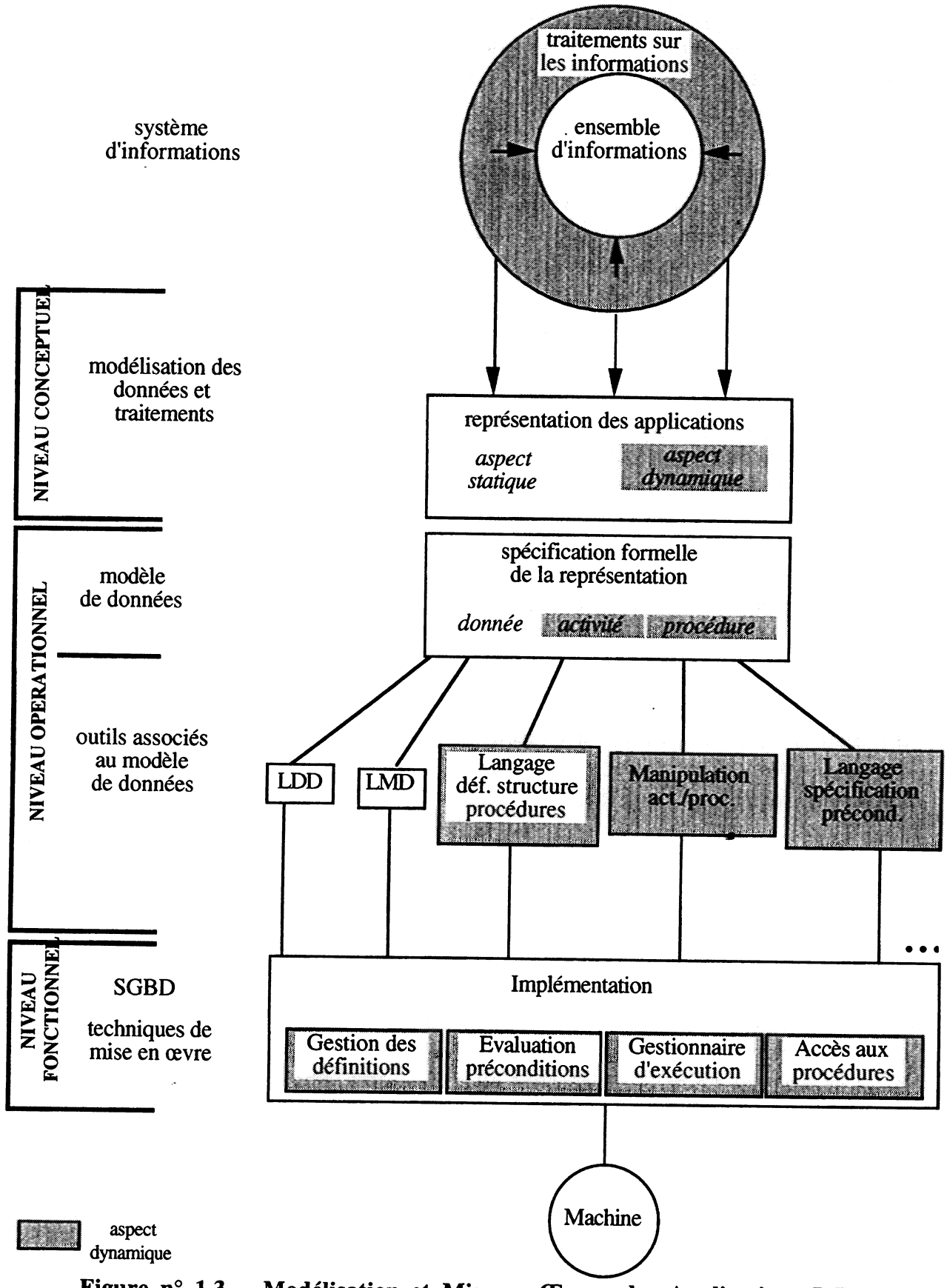


Figure n° 1.3. - Modélisation et Mise en Œuvre des Applications B.D.

Dans la figure, les aspects concernant la dynamique des applications ont été mis en évidence (en grisé) : ceci permet de situer le travail réalisé dans le cadre du projet DOEOIS. Le projet a traité le niveau opérationnel en introduisant les concepts d'activité et procédure dans le modèle de données. Comme conséquence, la définition de nouveaux outils comme par exemple le langage de spécification de préconditions ou le langage de définition de la structure des procédures a dû être faite. Les implications au niveau fonctionnel concernent l'offre d'un ensemble de services permettant de gérer les traitements sur les informations. Cela nécessite, entre autre, le développement de modules pour gérer les définitions et l'exécution des activités et procédures, ou pour évaluer les préconditions.

Ce travail offre ainsi un ensemble de services permettant le développement d'applications de gestion d'activités bureautiques pour aboutir à un système complet. Par exemple, plusieurs interfaces, comme le suivi de l'exécution des procédures de bureau, peuvent être construites.

Un point important de ce travail a été la colaboration avec l'équipe de Fraunhoffer Institute et de l'université de Stuttgart (partenaire du projet DOEOIS), qui a mené une étude sur les méthodologies de conception et spécifié un cahier des charges. La notion d'entité focale trouve son origine dans les travaux faits dans ce domaine. Au niveau opérationnel et fonctionnel il a été intéressant d'offrir explicitement cette notion : le lien entre les méthodologies de conception et les outils opérationnels a ainsi été renforcé.

Apport Personnel et Plan du Rapport

Pendant cette thèse, je me suis intéréssée en particulier à certains aspects qui se sont révélés critiques : ils seront développés plus en détail dans ce document. Ces aspects englobent la modélisation d'une activité et de ses différents composants (entité focale, contexte d'activation, précondition,...) ainsi que le déclenchement d'une procédure ou d'une activité (individuelle ou à l'intérieur d'une procédure). Ce travail a été mené selon les trois axes classiques : analyse, conception et implémentation.

Au niveau conceptuel, la modélisation d'une activité est faite en termes des concepts du modèle de données adopté. Le déclenchement automatique d'une activité ou procédure impliquant l'évaluation de leur précondition a amené à : (i) la définition précise de la syntaxe et de la sémantique des préconditions ; (ii) l'établissement des règles de correspondance entre les mises à jour et les préconditions ; (iii) la proposition d'un mécanisme efficace d'évaluation des préconditions ; (iv) l'évaluation des contraintes de précédance à l'intérieur d'une procédure.

Au niveau de l'étude analytique mon travail a consisté en l'étude des bases de données actives. En effet, si on considère l'automatisation des processus de manipulation de l'information, deux aspects des bases de données actives paraissent intéressants.

Le premier aspect est le concept de déclencheur et ses différentes généralisations : ce concept est extrêmement lié à la notion d'activité, entre autre plusieurs modèles l'utilisent pour la spécification de tâches simples. En conséquence je fournis un cadre de référence complet permettant d'analyser un système de déclencheurs et de juger de sa puissance. Ce cadre prend en compte la façon dont un déclencheur est modélisé mais aussi la façon dont il est exécuté.

Le deuxième aspect concerne la modélisation de longues séquences de travail ; ceci est lié à la notion de procédure de bureau. Une brève présentation de TAXIS a alors lieu : je considère ce système comme représentatif des travaux concernant la modélisation de longues tâches bureautiques. Non seulement un modèle a été proposé mais aussi des techniques d'implémentation permettant la vérification des conditions d'activation des traitements ont été développées.

Par ailleurs, un des points importants de cette thèse est l'évaluation des préconditions d'activités. Au commencement de ce travail il existait très peu de références décrivant des mécanismes d'évaluation de préconditions ; par contre, nous pouvions trouver plusieurs articles sur les méthodes d'évaluation des contraintes d'intégrité sur des SGBDs relationnels. Or, l'évaluation des contraintes d'intégrité semble étroitement liée à l'évaluation de préconditions. Pour cela les principes sur lesquels les méthodes d'évaluation de contraintes d'intégrité s'appuient ont été dégagés et leur application à l'évaluation de préconditions de déclenchement analysée.

Au niveau implémentation trois aspects ont été couverts : le traitement d'une précondition au moment de la définition de l'activité (compilation de la précondition) et au moment de l'exécution de mises à jour (évaluation de la précondition) ; implémentation du gestionnaire d'exécution des activités et procédures ; implémentation d'une interface graphique permettant la définition et le suivi de l'exécution des activités / procédures (ce dernier point n'est pas développé dans ce rapport car il ne présente essentiellement des problèmes d'interface homme-machine).

L'implémentation du mécanisme d'évaluation de préconditions a été un des points les plus importants. La contribution principale de la définition et de l'implémentation d'un tel mécanisme est une nouvelle étape dans l'évaluation efficace de prédicats dans un environnement base de données potentiellement complexe. Il est clair que l'absence d'une technique efficace qui puisse être facilement implémentée pour déclencher et évaluer des préconditions est un obstacle à l'utilisation de ce type de mécanismes qui sont extrêmement utiles dans la modélisation des aspects dynamiques des applications de façon déclarative. Cette situation est similaire à celle des mécanismes de vérification de contraintes d'intégrité, où un décalage très grand entre le développement théorique de méthodes et leur implémentation persiste.

Dans la page suivante le plan général de la thèse est présenté. Je voulais finir cette introduction en mettant en valeur le fait que les résultats présentés ici ont été possibles grâce à un fort travail d'équipe. Cette équipe était composée des différents partenaires du projet DOEOIS. Je tiens à nommer en particulier : Brian Caulfield et Sean Baker du Trinity College of Dublin avec qui le travail de modélisation des activités et procédures de bureau a été discuté ; Samer Haj Houssain, Christian Lenne et Esperanza Pedraza, du Centre de Recherche BULL à Grenoble, nous avons travaillé ensemble dans la définition d'une architecture optimisée du serveur d'information pour la bureautique ; Catherine Donnou, Jean-Claude Guyot, Ahmed Houssien, Jackson Martins Ramos et Gilles Vinçon, stagiaires qui ont permis de mener à bien l'implémentation du gestionnaire d'exécution et de l'interface graphique ; et Mauricio Lopez qui a collaboré à tous ces points et qui a travaillé particulièrement avec moi dans la définition précise de la syntaxe des préconditions et du LMD et dans l'établissement des règles de correspondance entre les mises à jour et les préconditions. Ce travail d'équipe a donc permis la concrétisation des différents aspects abordés!

INTRODUCTION	1	Contexte de la thèse, problématique d'un Serveur d'Information pour la bureautique
ETUDE ANALYTIQUE	2	Une étude sur les b.d. actives et en particulier sur les déclencheurs s'est avérée importante pour juger du concept d'activité. Une analyse sur l'application des principes d'évaluation de contraintes d'intégrité à l'évaluation des préconditions d'activité est réalisée. L'aspect modélisation de long processus d'information est abordé par la présentation de TAXIS.
MODELISATION DE LA DYNAMIQUE	3	Présentation du modèle conceptuel intégrant les aspects statiques et dynamiques d'une application.
	4	Présentation de la modélisation d'une activité, de ses composants et des outils sous-jacents (e.g., primitives de manipulation).
	5	Présentation de la modélisation d'une procédure, de ses composants (en particulier, de son schéma) et des outils sous-jacents.
TRAITEMENT DES PRECONDITIONS	6	Etablissement des règles de correspondance entre les mises à jour et les préconditions. Ces règles permettent de définir de façon formelle la sémantique des préconditions. Elles sont aussi la base du traitement des préconditions.
	7	Présentation du mécanisme d'évaluation des préconditions. Définition des règles de dérivation d'une forme de précondition évaluable efficacement. Application de plusieurs techniques d'optimisation.
MISE EN ŒVRE	8	Présentation de la mise en œuvre d'un Serveur d'Information pour la Bureautique, plus particulièrement, de l'implémentation du mécanisme d'évaluation de préconditions, de l'évaluation des contraintes de précédance, et du gestionnaire d'exécution .
CONCLUSION	9	Principaux résultats au niveau conceptuel, opérationnel et fonctionnel. Perspectives.

CHAPITRE 2

2. BASES DE DONNEES ACTIVES

Les bases de données actives sont définies par opposition aux bases de données conventionnelles dites passives qui ne manipulent les données qu'en réponse à des demandes explicites des utilisateurs ou des applications. Une base de données active est donc spécifiée de façon à réagir automatiquement à certaines situations en engageant des actions.

Le développement des bases de données actives permet d'une part d'uniformiser le traitement de certaines fonctions d'un SGBD comme la vérification des contraintes d'intégrité, le contrôle d'accès, la maintenance de vues et l'inférence et, d'autre part, d'élargir les fonctionnalités des SGBD, par exemple :

- Implémenter de façon plus fine et efficace les contraintes d'intégrité. Une contrainte d'intégrité peut être violée par différents types de mise à jour, ceci exige la réalisation d'actions (correctives ou d'avertissement par exemple) à caractère différent. Les bases de données actives permettent de contrôler automatiquement chaque type de mise à jour et d'appliquer l'action appropriée en cas de violation.

- Rendre plus modulaire et flexible la façon dont la maintenance des vues (plus généralement des données dérivées) est faite. En effet, les bases de données actives permettent de spécifier de façon déclarative les traitements à faire sur ces données (les données dérivées et les données dont elles dépendent) et les situations dans lesquelles les appliquer. En exécution, un mécanisme permet de détecter les situations critiques et de lancer les traitements correspondants.

- Assurer l'utilisation correcte des constructeurs d'abstraction de données propres aux modèles sémantiques (constructeur de généralisation et d'agrégation par exemple). La déclaration de classes de données construites sur d'autres classes déjà définies exige un traitement spécial des mises à jour sur leurs instances (p.ex., propagation des mises à jour dans une hiérarchie de généralisation). Les bases de données actives offrent des mécanismes capables de contrôler ces mises à jour et de déclencher les traitements nécessaires pour maintenir la base cohérente par rapport au schéma conceptuel.

- Contrôler l'utilisation de la base de données. Cela permet de réaliser des études de contrôle de la performance comme par exemple maintenance de statistiques nécessaires à l'optimisation de requêtes ou à l'organisation de la base de données. Une autre application est la détection des accès interdits à des données et la réaction conséquente.

- Lancer des tâches simples automatiquement. Contrôler les changements de la base de données permet de détecter des situations qui exigent un traitement particulier : suivre l'évolution du stock d'un certain article permet de détecter à quel moment il est devenu inférieur à un certain seuil et de produire une commande en conséquence.

- Permettre l'implémentation de long processus de manipulation de données en offrant un mécanisme qui contrôle les contraintes de précédance entre les différentes étapes composant la longue tâche. Ceci nécessite de suivre l'exécution de groupes d'actions (vues comme une unité) et de réveiller d'autres activités en conséquence.

- Fournir un formalisme et un mécanisme pour le support d'applications externes. Par exemple, les mécanismes offerts par les bases de données actives créent de nouvelles formes d'interaction entre les applications et le SGBD : les programmes d'application signalent des évènements et le SGBD en exécutant des actions lance d'autres programmes d'application. Les applications peuvent ainsi être construites selon un nouveau paradigme où le contrôle logique est spécifié de façon déclarative et indépendante (des programmes) par les outils offerts par les bases de données actives.

Au premier abord, on peut constater que la mise en œuvre d'une base de données active nécessite de fournir :

- Des moyens de définir, au niveau du schéma conceptuel des applications, des classes d'objets dynamiques et actifs.

- Un formalisme pour décrire des évènements, des situations et des réactions (à ces évènements et situations).

- Des mécanismes efficaces qui soient capables de suivre l'évolution de la base de données et de contrôler l'occurrence d'évènements révélateurs du lancement de groupes d'actions.

- Des techniques d'optimisation de l'évaluation de prédicats exprimant des situations critiques de la base de données.

La figure 2.1 présente de façon chronologique quelques uns des évènements qui ont contribué au développement des bases de données actives. Avant d'analyser ce tableau, il est intéressant de faire un rapide survol des prédécesseurs des bases de données actives.

Dans d'**autres domaines** de la science informatique, des concepts pour modéliser des réactions (à entreprendre automatiquement) à certaines situations ont été développés. En intelligence artificielle nous trouvons par exemple les concepts de règles de production [Hew72] [For77], de "daemons" ou d'objets actifs [Bob83]. Dans le domaine des langages de programmation l'accent a été mis sur le traitement des exceptions, p.ex. PL1, ADA, CLU. Dans ce dernier domaine, la notion d'évènement pour la modélisation d'applications en temps réel et pour la simulation de systèmes a aussi été

développée.

Il faut cependant noter que ces travaux imposent des restrictions importantes. La première restriction porte sur le stockage des objets en mémoire centrale. Ceci impose un nombre d'objets assez petit par opposition aux larges bases de données stockées en mémoire secondaire. La seconde concerne le modèle d'exécution. En effet, celui-ci est assez pauvre car il se limite le plus souvent à une exécution séquentielle. S'ils mettent en œuvre une exécution parallèle, il n'existe pas dans ce cas de mécanisme de reprise pour les objets partagés.

Dans le **domaine des bases de données**, les SGBDs traditionnels ne jouissaient pas d'un support explicite pour contrôler et suivre des situations de façon automatique et réagir en conséquence. Ainsi, ils faisaient usage des deux techniques suivantes. La première se résume dans l'interrogation périodique de la base de données pour déterminer si les situations (ou conditions) à détecter ont eu lieu. L'inconvénient de cette approche est la surcharge importante engendrée par le mécanisme lorsque la périodicité d'interrogation de la base de données est de courte durée. Le SGBD est alors en permanence interrogé même si dans la plupart des cas la réponse est vide. Par ailleurs, si la périodicité est de longue durée, plusieurs situations significatives peuvent avoir lieu sans pour autant être détectées. La deuxième "technique" est de noyer la détection des situations et l'appel à des actions de réaction dans le code des programmes d'application. L'inconvénient est évident à savoir le manque de modularité et la perte d'indépendance des données vis à vis des traitements.

La figure 2.1 montre la progression allant des premiers travaux qui supportent quelques unes des fonctionnalités des bases de données actives comme les "ON-Conditions" de CODASYL, jusqu'à de vrais SGBDs dits actifs comme HIPAC.

Au niveau de la **modélisation**, nous distinguons dans les travaux préalables aux bases de données actives la spécification de systèmes d'intégrité sémantique pour les SGBDs relationnels [Ham75] et [Cha76]. Un de ces systèmes définit un mécanisme de déclencheurs (Système R) : succinctement, un déclencheur est une action qui est exécutée automatiquement quand un prédicat est évalué comme vrai ; le prédicat peut refléter un état de la base de données ou un changement. Comme nous allons le voir, le concept de déclencheur joue un rôle très important dans les bases de données actives.

Au niveau des **techniques d'implémentation**, nous distinguons les travaux menés dans le but de vérifier des contraintes d'intégrité pour les SGBDs relationnels et les travaux de propagation des mises à jour pour la maintenance de vues, photographies, etc.

Avec l'apparition des **SGBDs sémantiques** et des nouveaux constructeurs de données qu'ils proposent, le maintien des contraintes d'intégrité inhérentes au schéma conceptuel a pris plus d'ampleur. [Ped88] et [Ant87] proposent un système de déclencheurs pour le maintien de ce type de cohérence. Ce mécanisme est complètement transparent à l'utilisateur : les déclencheurs sont générés automatiquement par le système au moment de la création de classes d'objets, leur but est la propagation des mises à jour faites sur les instances de ces classes.

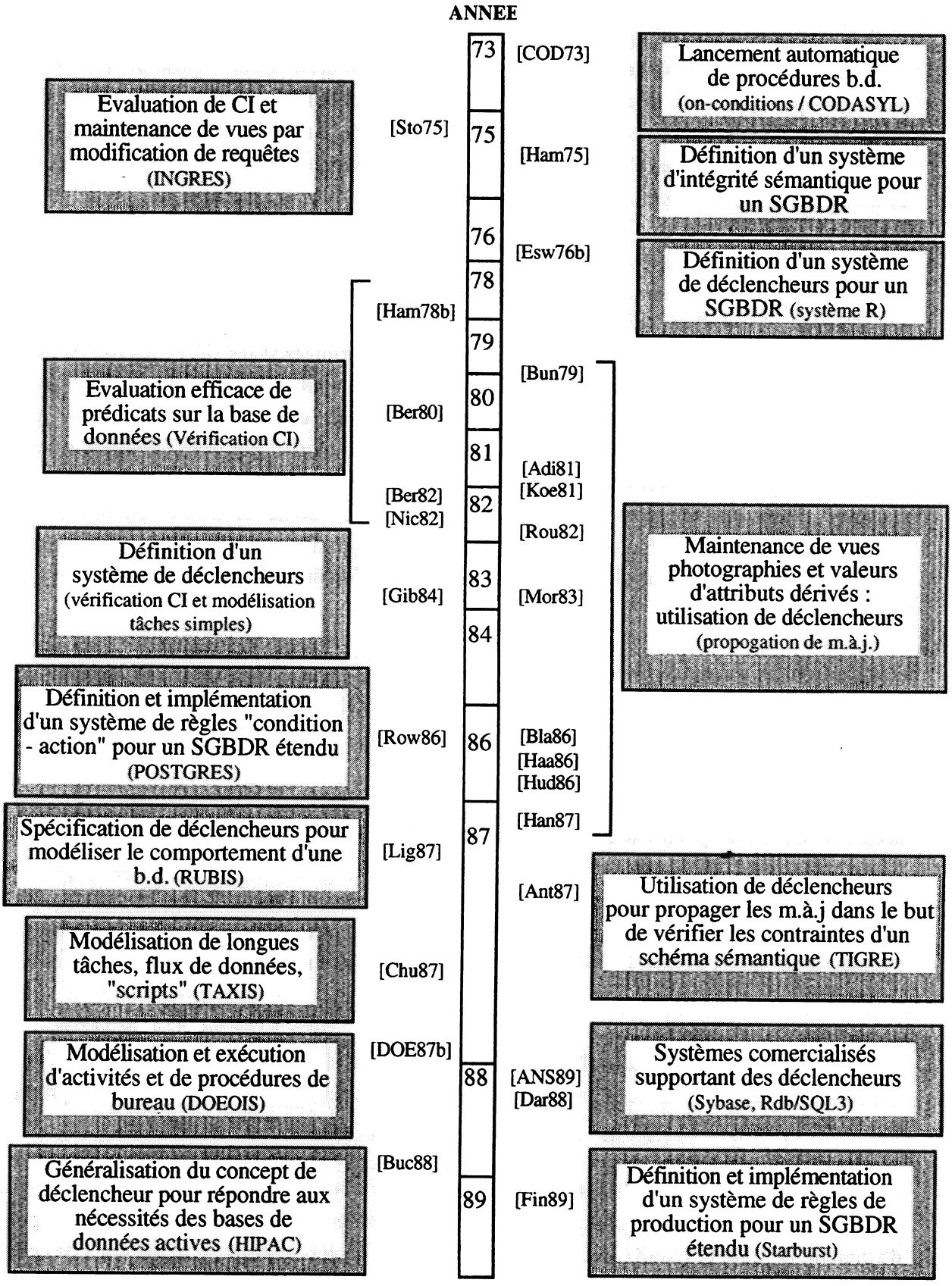


Figure n° 2.1 - Historique des Bases de Données Actives

Au niveau des SGBDs **orienté application** les travaux développant des fonctionnalités des bases de données actives se notent surtout dans les systèmes destinés à la CAO et à la bureautique.

En ce qui concerne la **CAO,** [Dit86] et [Dit87] généralisent la notion de déclencheur et développent un mécanisme, "évènement/déclencheur", pour la vérification de contraintes d'intégrité. Remarquons qu'en CAO, l'intégrité est assez complexe d'une part, car les objets ont des structures complexes et beaucoup d'inter-dépendances entre eux ; d'autre part, car la notion de transactions longues (développée pour la CAO) doit tolérer des inconsistances pendant une période de temps indéterminée qui peut être grande.

En ce qui concerne la **bureautique**, [Zlo82] développe un mécanisme de déclencheurs pour OBE ("Office-by-Example"). Ce mécanisme supporte la modélisation de procédures bureautiques routinières ; il permet la détection d'évènements de deux types : des changements de la base de données et des évènements temporels (ponctuels, p.ex. le 5.10.90 à 16h45, ou périodiques, p.ex. tous les lundis).

[Gib84] développe un modèle de données orienté objet pour la bureautique supportant un mécanisme de déclencheurs très simple permettant la vérification des contraintes d'intégrité ainsi que la modélisation de tâches simples comme la messagerie.

Et pour finir avec ce domaine d'application, nous ne pouvons pas manquer de référencer les travaux réalisés dans le cadre du projet TAXIS permettant la modélisation de **longues tâches bureautiques**. C'est à ce niveau que nous plaçons le travail développé dans le projet DOEOIS concernant la modélisation et l'exécution d'activités et de procédures de bureau.

Les travaux pour la modélisation des **bases de données dynamiques**, dont un des premiers a été la définition du modèle ACM/PCM [Bro81], aboutissent en 86/87 à la spécification de deux modèles qui prétendent traiter de façon complète les aspects temporels et dynamiques d'une base de données : RUBIS [Lig87] supportant la notion de déclencheur pour modéliser le comportement des applications et MTG [Guy86] permettant, entre autre, de spécifier des traitements associés à la modification des valeurs des objets, et des évènements décrivant les conditions de synchronisation des traitements..

Dans les travaux courants, nous trouvons la définition et le développement de prototypes de systèmes de **règles du type "condition-action"** : POSTGRES [Rowe86] et [Hea89] et Starburst [Fin89] ayant comme but le traitement uniforme du contrôle de l'intégrité des accès à la base de données, le traitement des données dérivées et un mécanisme capable de supporter de l'inférence.

Récemment, plusieurs SGBDs relationnels commerciaux comme Sybase [ANS88] et Rdb Release 3.1a supportent la notion de déclencheurs.

Finalement, dans le projet HIPAC un effort a été fait pour **généraliser la notion de déclencheur** de façon à supporter les différentes fonctionnalités d'une base de données active.

Par rapport à notre travail sur l'automatisation des processus de manipulation de l'information

bureautique, deux aspects des bases de données actives paraissent intéressants. Le premier est le concept de déclencheur et ses différentes généralisations : ce concept est extrêmement lié à la notion d'activité telle qu'elle a été décrite dans l'introduction (cf.1.3), nous venons de voir aussi que plusieurs modèles spécifient des tâches simples au moyen de ce concept. Le deuxième aspect concerne la modélisation de longues séquences de travail ; ceci est lié à la notion de procédure de bureau telle qu'elle a été introduite dans 1.2 et 1.3.

Ainsi, dans la suite de ce chapitre nous définissons dans le paragraphe 2.1 un cadre de référence permettant d'analyser les différents mécanismes de déclencheurs proposés. Le paragraphe 2.2 est dédié à une brève étude de TAXIS que nous considérons comme représentatif des travaux concernant la modélisation de longues tâches bureautiques : non seulement un modèle a été proposé mais aussi des techniques d'implémentation permettant la vérification des conditions d'activation des traitements ont été développées.

2.1. Déclencheurs

Ce paragraphe fournit un **cadre de référence** complet permettant d'analyser un système de déclencheurs et de juger de sa puissance. Ce cadre, ayant comme but de traiter les différentes facettes d'un déclencheur, doit prendre en compte, d'un côté, la façon dont un déclencheur est modélisé (p.ex., sa structure et la connaissance qui lui est attachée) et d'un autre côté, la façon dont il est exécuté (i.e., le modèle d'exécution qui lui est sous-jacent). C'est seulement ainsi qu'on peut aboutir à une analyse correcte et complète de la sémantique d'un mécanisme de déclencheurs.

La **modélisation d'un déclencheur** concerne :

- La structure du déclencheur, i.e., comment les évènements, situations et actions sont spécifiés. Il est intéressant d'étudier la puissance du langage permettant de spécifier cette structure, par exemple, quels évènements sont permis ? Uniquement des mises à jour basiques sur la base de données ou des évènements résumant tout un groupe d'actions ?

- Le contexte associé au déclencheur, i.e., la partie de la base de données que le déclencheur peut référencer (ou pour laquelle il s'exécute). Il est intéressant aussi de voir comment les différentes parties du déclencheur accèdent en exécution à ces informations.

- La façon dont le concept de déclencheur est inclus dans le modèle de données. Ceci est un indicateur du degré d'abstraction donné à ce concept et de l'intégration entre les aspects statiques et dynamiques au niveau conceptuel.

L'**exécution d'un déclencheur** concerne son activation (i.e., la détection de l'occurence d'un évènement), éventuellement l'évaluation de conditions sur la base de données et l'exécution d'actions de réaction. Ainsi, un modèle d'exécution doit indiquer précisément :

- Le moment où la condition est évaluée et l'action exécutée par rapport à l'occurrence de l'évènement.

- Le comportement transactionnel de l'exécution d'un déclencheur par rapport à l'évènement qui a déclenché cette exécution.

- Le comportement de l'exécution de plusieurs déclencheurs activés simultanément (i.e., par le même évènement).

- Le comportement de l'exécution de déclencheurs activés en cascade (i.e. l'exécution d'un déclencheur provoque l'activation d'un autre).

Finalement, il importe d'inclure dans ce cadre de référence les techniques d'implémentation d'un système de déclencheurs. En particulier, les techniques utilisées pour activer un déclencheur et pour évaluer les conditions sur la base de données. Ce dernier aspect est très important vu que l'évaluation de la partie condition est très délicate nécessitant des accès frequents à la base de données.

En fonction du cadre de référence proposé, plusieurs systèmes ont été étudiés et la façon dont ils répondent à cette grille est ici résumée. Les systèmes qui ont le plus mérité notre attention sont :

- Le système d'intégrité sémantique d'un SGBD relationnel proposé par Hammer et McLeod. Il est référencé dans le texte par **[Ham75]**. Ce système n'a été défini qu'au niveau des spécifications, aucune implémentation ayant eu lieu. Il représente un effort important au niveau de la recherche.

- Le sous-système de déclencheurs ("triggers") du **Système R** [Cha75] [Esw76b]. C'est dans ce système que pour la première fois le concept de déclencheur apparaît.

- Le mécanisme de déclencheurs ("triggers") proposé par le système **Sybase** [ANS88]. Nous présentons ce mécanisme car il est un des seuls pour lequel un produit est commercialisé.

- Le mécanisme de déclencheurs défini dans la proposition de norme ANSI et ISO de SQL3 [ANS89]. Il sera référencé dans le texte par "**SQL3**". Un prototype a été développé dans Rdb Release 3.1a.

- Le mécanisme de règles de production à orientation ensembliste de **Starburst** [Fin89a] [Fin89b]. Un prototype a été réalisé et est en extension.

- Le mécanisme de règles de **POSTGRES** [Hea89] [Sto86]. Un prototype a été développé ; son évaluation a abouti à la proposition d'un ensemble d'extensions en cours d'implémentation. L'analyse de ce mécanisme est important car il est basé sur une philosophie différente des autres propositions.

- Le mécanisme de déclencheurs ("triggers") du "Object-Oriented Office Data Model" développé

par Gibbs. Un prototype a été implémenté. Ce système est référencé dans le texte par **[Gib84]**.

- Le système de règles Evènement - Condition - Action de **HIPAC** [Bla88] [Bla89] [Buc88] [Cha89] [Cha90] [Day89]. L'analyse de ce mécanisme de déclencheurs a été très importante : comme il a déjà été dit, il offre une généralisation des déclencheurs pour répondre aux exigences d'une base de données active. Un prototype d'un sous-ensemble du mécanisme de règles a été implémenté pour évaluation [Cha90].

Enfin, le choix des différents systèmes comprend des systèmes relationnels classiques ; des systèmes relationnels étendus : Starburst et POSTGRES ; un système dit orienté objet pour la bureautique ; et HIPAC qui offre un niveau de généralisation intéressant.

L'ensemble de ces différents systèmes sont regroupés sous le terme de déclencheurs, ceci ne paraît pas un abus vu qu'ils proposent tous des mécanismes ayant comme but le déclenchement automatique d'une action quand un certain prédicat sur la base de données est vérifié.

L'organisation de ce paragraphe est guidée par deux axes : le premier, concernant la modélisation d'un déclencheur, est étudié dans les paragraphes 2.1.1 à 2.1.4, le deuxième, concernant la sémantique attachée à l'exécution des déclencheurs, est étudié dans 2.1.5. Un dernier paragraphe présente des techniques d'implémentation d'un système de déclencheurs.

Finalement, deux annexes accompagnent ce paragraphe : l'annexe I qui présente le tableau résultant de cette analyse et classe les différents mécanismes de façon systématique ; l'annexe II donne des exemples de déclencheurs utilisant la syntaxe proposée dans les différents systèmes en mettant l'accent sur leur évolution chronologique.

2.1.1. Structure Générale

Un déclencheur est un groupe d'actions qui doit être exécuté quand un certain prédicat est vérifié. Sa forme la plus simple est donnée par : "prédicat - action".

Les déclencheurs ont été utilisés dans le domaine des bases de données tout d'abord pour la vérification et l'évaluation des contraintes d'intégrité. Leur forme a l'avantage de spécifier d'une part la contrainte d'intégrité (CI) - le prédicat - et de détailler d'autre part une stratégie à prendre en cas de violation de la CI - le groupe d'actions.

Auparavant, la seule action possible en cas de violation d'une CI était le rejet de toute opération qui provoquait son viol. Avec les déclencheurs, d'autres types d'actions sont pris en compte. Par exemple : actions correctives (comme la répercussion des mises à jour), avertissements, exceptions. Les déclencheurs ont ainsi permis la spécification d'une classe plus grande de contraintes d'intégrité.

La structure générale d'un déclencheur a évoluée et en ce moment le type de déclencheur qui prévaut divise le prédicat en deux parties : une partie évènement et une partie condition (voir figure n° 2.2).

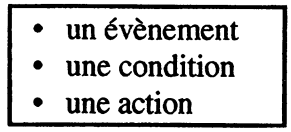


Figure n° 2.2 - Déclencheur Basique

Cette division a été proposée parce que la valeur d'un prédicat dans une base de données change si la base de données a été mise à jour, i. e., si un évènement a eu lieu.

Ainsi, l'évènement spécifie le type d'action qui doit avoir lieu pour que le déclencheur soit activé, i. e., pour que la condition soit évaluée et l'action exécutée en conséquence. La condition fait référence aux éléments concernés par l'évènement et à l'état général de la base de données. La condition doit être vérifiée pour que le groupe d'actions soit exécuté.

La division du prédicat en deux parties offre plusieurs avantages dont :

- Un **contrôle plus fin** sur l'évolution de la base de données. Par exemple, supposons la contrainte d'intégrité suivante : "tout article vendu doit être répertorié dans un inventaire". La vérification de cette contrainte doit se faire dans deux cas (si un nouvel article est vendu il faut vérifier s'il existe dans l'inventaire ; si un article est éliminé de l'inventaire il faut vérifier s'il n'est pas vendu). La mise en œuvre de cette CI par deux déclencheurs, chacun d'eux activé par un des évènements qui provoque la vérification de la contrainte, a comme conséquence l'application d'une stratégie différente selon les raisons qui ont mené à la violation de la CI (p.ex., demander des informations à l'utilisateur pour répertorier un nouvel article à vendre ; faire la répercussion de la suppression d'un article de l'inventaire dans les articles vendus). Si le prédicat n'était pas divisé en deux parties, l'action à engager serait la même dans les deux cas : probablement le rejet de la mise à jour qui active le déclencheur.

- Une **interprétation unique** du déclencheur. Cet avantage est très lié au précédent.

- L'augmentation du **pouvoir d'expression** vu que des évènements autres que les mises à jour basiques sur la base de données peuvent être spécifiés pour activer un déclencheur. Ceci sera plus détaillé dans 2.1.1.1, néanmoins pour donner un exemple tout simple, supposons la spécification d'un déclencheur qui serait activé lors d'une interrogation à la base de données : ceci est très utile pour le contrôle des droits d'accès ou la manipulation de vues. En conséquence, de nouvelles applications autres que les CI pourront être prises en compte.

- La spécification du **moment d'évaluation** de la condition. Par exemple, avant que l'action qui active le déclencheur soit exécutée ou après. Un autre moment important est la fin d'une transaction : ceci aide à mettre en œuvre la notion d'intégrité liée à une transaction.

- La spécification de différents **modes de couplage** évènement - condition et condition - action. Ceci est une généralisation de l'item précédent et concerne le moment d'application d'une des clauses du déclencheur par rapport aux autres : application immédiate ou différée.

- Une **division logique** entre changement et état de la base de données. Cette division au niveau logique reflète aussi une division au niveau implémentation : en effet, l'évaluation des deux parties se fait différemment. Cet avantage est d'ordre plus général.

En concluant nous pouvons dire que cette division permet un pouvoir d'expression plus grand, un contrôle plus fin, une spécification plus structurée, une implémentation simplifiée.

Bien que des formes de déclencheurs plus sophistiquées aient été proposées (p.ex. des déclencheurs composés de pré-conditions, post-conditions, etc.) nous pensons que la spécification d'un déclencheur dans ces trois parties est la forme la plus importante. Cette forme peut être aussi considérée comme une structure de base - "irréductible", en exagérant - parce que souvent les formes plus complexes peuvent être spécifiées par plusieurs de ces déclencheurs.

Ainsi, trois critères ont été retenus pour classifier un déclencheur selon sa structure générale (voir annexe I) : (1) l'**évènement** est-il spécifié explicitement ? (2) la **condition** est-elle spécifiée explicitement ? (3) l'**action** est-elle spécifiée explicitement ?

Dans ce qui suit les critères permettant de juger du pouvoir d'expression de chacune des clauses d'un déclencheur de base seront présentés.

2.1.1.1. Evènement

Un évènement est spécifié par une opération et les objets auxquels elle s'applique.

2.1.1.1.1. Opérations

Dans la plupart des implémentions de SGBDs relationnels (que ce soit des prototypes ou des produits) les seules opérations permises sont les **mises à jours de base** (insertion, suppression et modification d'un n-uplet).

L'opération de **sélection** est quelquefois, tolérée sur quelques systèmes : un déclencheur qui est activé par une telle opération est alors normalement nommé "Alerter" [Bun79a], [Sto86].

Au niveau des propositions - soit non implémentées, soit plus récentes - faites sur la spécification de la clause évènement, nous pouvons trouver d'autres types d'opérations :

- D'**autres opérations base de données**. Remarquons que cette possibilité est particulièrement intéressante dans le cas des SGBDs sémantiques offrant des langages de manipulation de plus haut niveau. Un exemple est la prise en compte d'opérations qui permettent de changer une instance de classe à l'intérieur d'une hiérarchie de spécialisation : dans DOEOIS les activités (cf. chp. 1) peuvent être activées par de telles opérations (cf. 4.2.1); de même pour [Gib84].
- Des **opérations structurées** définies dans [Ham75] comme une liste d'opérations de mise à

jour, d'énoncés d'interrogation et d'autres opérations qui ont été définies auparavant [Ham75].
Remarquons que le fait qu'une opération structurée puisse être spécifiée comme un évènement est très puissant : elle reflète tout un ensemble de changements de la base de données.
Un parallèle peut être dressé entre les opérations structurées et la partie exécutable des activités bureautiques telles qu'elles ont été définies dans le projet DOEOIS (cf. chp. 1). Comme nous allons le voir, dans les chapitres 5 et 8, la fin de l'exécution de toute activité est considérée comme un évènement qui est implicitement signalé par le système. Ceci permet d'engager plusieurs actions à la fin de l'exécution de chaque activité (par exemple, faire leur suivi). De plus, à l'intérieur d'une procédure, la combinaison de ces évènements permet de spécifier les contraintes de précédence entre les différentes activités la composant.

- Des **évènements explicites**. Il s'agit d'évènements explicitement définis par l'utilisateur. Ils sont aussi explicitement signalés par l'utilisateur ou par les programmes d'application.
  Ce type d'évènement est très utilisé en CAO ou en bureautique pour l'ordonnancement d'actions. D'ailleurs, l'utilisation de tels évènements pour la définition des contraintes de précédence des activités à l'intérieur d'une procédure a été envisagée dans le projet DOEOIS. Néanmoins, il a été préférable d'utiliser une signalisation implicite : voir item précédent.

- Des **évènements externes** à la base de données mais définis au niveau du SGBD en tant qu'évènements (i.e. qui sont associés à un module permettant leur détection ou signalisation). HIPAC propose l'utilisation de ce type d'évènements. Un évènement externe important est le temps ; remarquons néanmoins qu'il peut être aussi une donnée de la base de données.

Un dernier critère pour juger de la clause évènement est le fait qu'elle permette ou pas la **composition d'évènements**. Les différents constructeurs de composition qui ont été proposés sont les suivants :

- **Le constructeur de disjonction**. Ceci a été le constructeur le plus proposé, mais il est essentiellement appliqué (voire, autorisé) dans seulement deux cas :
  - la disjonction de plusieurs opérations qui expriment sémantiquement le même évènement. Par exemple, donner une valeur à une colonne d'une relation peut être fait par une opération d'insertion ou une opération de modification - Sybase.
  - la disjonction de différentes opérations qui ont la même influence dans la validation de la condition et qui conduisent à l'application de la même action. Par exemple, on veut contrôler la distance entre x et y, et appliquer la même action si cette distance est devenue inférieure à un certain seuil. La clause évènement d'un déclencheur qui exprimerait ceci serait la disjonction des deux évènements : déplacer x et déplacer y - HIPAC.

  Remarquons que la spécification de deux déclencheurs ayant les mêmes clauses condition et action et des clauses évènements différentes revient à exprimer un seul déclencheur avec la disjonction des evènements. Néanmoins, nous considérons que dans les deux cas cités ci-dessus le constructeur disjonctif offre une facilité syntaxique (voire de modélisation) qui peut être avantageuse.

Cette facilité est d'autant plus importante que le modèle de données est puissant. Par exemple, les modèles sémantiques ont des langages de manipulation d'un niveau plus grand que les langages relationnels et permettent la spécification d'opérations de mise à jour qui peuvent être vues comme une liste d'opérations de base. Il n'est donc pas rare de pouvoir exprimer le même évènement de base par différentes opérations. En particulier, dans le projet DOEOIS, ce type de disjonction a été prise en compte (cf. 4.2).

- **Le constructeur de conjonction**. Ce constructeur n'est quasiment jamais proposé. Ceci est dû au fait que la sémantique d'une conjonction n'est pas très claire, vu que la persistance de l'effet d'une opération est indéterminée et ne peut pas être assurée au moment où une autre opération a lieu.
  Il y a néanmoins un cas où ce constructeur est intéressant, il s'agit de la conjonction entre une opération base de données quelconque et l'opération de fin de transaction, p.ex., [Gib84] et HIPAC. Comme il a déjà été dit, ceci est surtout utile pour la spécification des contraintes d'intégrité.

- **Le constructeur de fermeture**. Ce constructeur n'a été proposé que dans le projet HIPAC et dans Starburst de façon implicite : sa forme est "opération1 : opération2" et nous le retenons uniquement si opération2 est l'opération de fin de transaction[1] (pour les raisons données dans l'item précédent). Sa sémantique est : le déclencheur est activé en fin de transaction si l'opération1 a eu lieu au moins une fois pendant la transaction (et il est activé une seule fois même si l'opération a été effectuée plusieurs fois pendant la transaction). Ceci est très utile pour les contraintes d'intégrité qui sont évaluées en fin de transaction pour tout l'ensemble d'objets modifiés par un même type de mise à jour.

2.1.1.1.2. Objets

En ce qui concerne les SGBDs relationnels, les opérations de mise à jour ne s'appliquent qu'à **une relation** : ainsi dans la clause évènement le plus souvent une seule relation est spécifiée. Et en conséquence, un déclencheur est associé à une seule relation.

Plusieurs SGBDs proposent un raffinement au niveau de la **colonne** d'une relation, p.ex., Sybase, permettant d'exprimer par exemple que le déclencheur est activé si une colonne particulière d'une certaine relation a été modifiée. L'exemple classique est la spécification d'un déclencheur qui est activé soit par l'insertion d'un n-uplet qui attribue une valeur à une colonne, soit par la modification d'un n-uplet qui assigne une nouvelle valeur à la même colonne.

Pour les "alerters" (déclencheurs activés par une opération d'interrogation) il est intéressant de pouvoir contrôler plusieurs relations en même temps. Ceci a été proposé par [Bun79] qui nomme un tel déclencheur "déclencheur complexe".

[1]Parfois opération 2 peut être considérée comme la fin d'une opération structurée, e.g, dans Starburst. Néanmoins la fin d'une telle opération coincide souvent avec la fin d'une transaction et traduit la fin d'une unité logique.

Les déclencheurs activés par une opération structurée peuvent être considérés comme s'appliquant à plusieurs relations (ceci dans le cas où les mises à jour à l'intérieur de l'opération structurée concernent plusieurs relations). Néanmoins, ces déclencheurs ne s'intéressent pas de façon particulière ou discriminante aux différentes mises à jour qui ont été réalisées mais plutôt au fait que l'ensemble de ces mises à jour réflète un certain état de la base de données.

Les déclencheurs qui s'appliquent à des relations particulières ou à des colonnes particulières sont classés "**locaux**". Un autre type de déclencheurs dits "**globaux**" reflètent un niveau d'abstraction supérieur [Ham75] [Gib84]. Ces déclencheurs sont appliqués non pas à une relation particulière ou à une colonne particulière mais à quelques colonnes ou à quelques relations caractérisées par une propriété commune. Par exemple, nous pouvons vouloir spécifier un déclencheur qui s'applique à toutes les colonnes d'un schéma, pourvu qu'elles soient de domaine "Nom" (le but de la condition serait par exemple de vérifier que les instances des colonnes sont un sous-ensemble d'un certain répertoire).

La proposition de déclencheurs globaux peut nécessiter l'introduction de nouveaux opérateurs de qualification au niveau de la clause condition [Ham75].

Il est intéressant de remarquer qu'un déclencheur global peut être implémenté par un ensemble de déclencheurs locaux. Néanmoins, le méta-niveau offert par les déclencheurs globaux est très utile au niveau conceptuel.

Par exemple, la portée des déclencheurs globaux varie suivant le fait que des relations soient créées ou détruites : si un déclencheur global - nommé "A" - est spécifié par un ensemble de déclencheurs locaux, il est possible alors qu'à la création d'une relation un nouveau déclencheur local doive être spécifié pour mettre en œuvre le déclencheur global A.

Ce type de déclencheurs a une application plus ample dans les SGBDs sémantiques qui ont des constructeurs d'abstraction puissants pour la conception d'un schéma conceptuel. Une application possible serait d'affiner la sémantique des opérations de mises à jour pour certains types d'objets agrégés.

Si un système de déclencheurs est défini pour un SGBD sémantique acceptant le constructeur de généralisation / spécialisation, l'application d'un déclencheur à une certaine classe d'objets peut être **héritée** par d'autres classes selon certaines règles. Dans le projet DOEOIS, ceci a été pris en compte et développé (cf. chapitre 6).

Les critères de classification de la clause évènement étant maintenant présentés, il est intéressant de se référer à l'annexe I et de voir comment ils apparaissent dans le tableau général de classement.

2.1.1.2. Condition

Le plus souvent la forme de la condition est :

- soit un **énoncé d'interrogation** du langage d'interrogation du SGBD sous-jacent. Dans ce cas la condition est considérée validée si l'évaluation de l'énoncé a comme résultat le retour d'un n-uplet au moins.

- soit la **partie qualification** d'un énoncé d'interrogation.

Le pouvoir d'expression de la partie condition dépend donc du langage d'interrogation du SGBD.

Plusieurs articles introduisent quelques opérateurs permettant une expression plus simple de certains types de CI comme par exemple : "subset-of ... in...", "one-to-one", "present-in".

Plusieurs SGBDs permettent de faire référence à l'ancienne et à la nouvelle valeur des objets modifiés par l'opération de déclenchement (ou d'activation). Ceci est souvent fait par l'introduction de mots-clés comme par exemple : "old" et "new", ou "inserted" et "deleted". Cette possibilité est particulièrement intéressante car elle permet l'expression de conditions de transition.

HIPAC permet l'expression d'une condition comme une collection d'énoncés d'interrogations. La condition est considérée vraie si tous les énoncés produisent un résultat non vide.

Au niveau recherche, et surtout pour le domaine des SGBDs orientés CAO (où les contraintes d'intégrité peuvent être très complexes vu qu'il y a beaucoup de dépendances entre les objets), un nouveau type de condition beaucoup plus puissant a été proposé. Il permet de définir une condition comme une **fonction booléenne**.

2.1.1.3. Action

Dans la plupart des implémentations qu'on peut trouver, la clause action est composée d'un groupe d'opérations admettant uniquement des **énoncés de mise à jour du LMD.**

Dans la plupart des systèmes classiques, p.ex., Sybase, SQL3, les énoncés d'**interrogation** qui renvoient des résultats à l'utilisateur ou à des programmes d'application ne sont pas permis dans la clause action. Cette restriction est faite car sans cela, les programmes d'application qui impliquent l'activation d'un tel déclencheur devraient prévoir la manipulation des résultats.

De même, dans ces systèmes classiques, des actions qui **modifient le schéma** de la base de données ne sont pas acceptées parce que le déclencheur est exécuté comme faisant partie de la transaction de l'opération de déclenchement (cf. 2.1.5.2).

Une opération importante qui a été intoduite est le **refus** de l'opération de déclenchement. Ceci est surtout utile pour l'implémentation des CI. Cette opération a été introduite à l'aide de mots clés comme "refuse".

Le droit d'**annuler la transaction** contenant l'opération de déclenchement est parfois proposé. Néanmoins, ceci dépend du mode d'exécution d'un déclencheur et sera plus détaillé dans 2.1.5.2.

Si on prend en compte des propositions faites au niveau des papiers [Ham75] ou des systèmes plus récents comme HIPAC, la clause action permet :

- l'introduction de constructeurs d'**abstraction procédurale** (p.ex. "if ... then ... else ...", "while ... do", ...) dans le groupe d'actions.
- de spécifier l'action comme un **appel à un programme**.

Ces propositions ont surtout été faites quand le but d'un système de déclencheurs allait au delà de la simple vérification de contraintes d'intégrité.

Si l'action admet des appels à des programmes d'application, l'exécution de la partie action ne s'effectuera pas forcément à l'intérieur du SGBD (ce qui est le cas de l'exécution d'actions n'admettant que des requêtes LMD) mais aussi à l'extérieur c'est à dire, par un nouveau processus (qui pourra éventuellement créer une nouvelle session du SGBD). Ceci crée de nouvelles formes d'interaction entre les applications et le SGBD et de plus permet : la récupération de fonctions ou outils existants, l'exécution de la partie action dans d'autres sites utilisant d'autres outils, etc.

Une activité, telle qu'elle a été définie au sein du projet DOEOIS, peut être vue comme un déclencheur dont la partie action est un appel à un programme externe. En conséquence, un gestionnaire s'occupant de l'exécution de ces actions a dû être développé : il définit, entre autre, un modèle de lancement d'activités dans des sites distants.

Hammer et McLeod [Ham75], probablement inspirés par la gestion des exceptions dans les langages de programmation et les différents types d'exception qui y sont définis (reprise, terminaison, "re-essai", propagation [Goo75]), ont classé des groupes d'actions : à chaque type d'action correspond une exécution particulière du déclencheur. Un exemple est l'exécution d'une opération structurée corrective et la "revérification" de la condition du déclencheur pour réagir en conséquence.

2.1.2. Contexte

Dans ce paragraphe, on s'intéresse au passage de données entre l'évènement, la condition et l'action, i. e. :

- Quelles données arrivent avec l'opération de déclenchement ? Et comment cette information est passée à la condition et à l'action ?
- Quelle portion de l'information utilisée dans l'évaluation d'une condition est passée à l'action? Et comment ?

La plupart des SGBDs relationnels n'ont pas beaucoup développé cet aspect, à cause de leurs limitations : un déclencheur est défini pour une relation et une opération de mise à jour et du fait que beaucoup ne font pas une distinction claire entre les trois parties du déclencheur.

Ainsi, en ce qui concerne le passage des informations de l'évènement à la condition nous avons surtout des systèmes où :

- Soit aucune description du contexte n'est faite.

- Soit le passage est fait implicitement et concerne l'état des n-uplets sur lesquels porte la mise à jour. Dans ce cas, si la condition est évaluée avant l'opération de déclenchement : pour une insertion aucun contexte n'est passé, pour une modification le contexte passé est l'état des n-uplets avant la modification et pour une suppression le dernier état des n-uplets. Si la condition est évaluée après l'exécution de la mise à jour : pour une insertion le contexte passé est le nouveau n-uplet, pour une modification le n-uplet modifié et pour une suppression aucun contexte n'est passé.

- Soit le contexte passé est l'ancien et le nouvel état des n-uplets concernés par la mise à jour. Ceci est fait explicitement par l'utilisation de mots-clés comme "old" et "new" ,p.ex., en SQL3 ou l'utilisation de relations logiques comme "inserted" et "deleted", p.ex., Sybase et Starburst.

Le contexte passé à l'action, est le même que celui passé de l'évènement à la condition moins les n-uplets pour lesquels la condition n'a pas été vérifiée : aucune information supplémentaire y est ajoutée[2].

Pour conclure avec les SGBD relationnels nous pouvons "réaffirmer" que cet aspect n'a pas vraiment été approfondi : le passage est fait soit implicitement, soit explicitement mais, dans ce dernier cas, uniquement par l'utilisation de mots clés.

Déjà chez [Gib84] le passage du contexte est traité de façon plus explicite : dans la clause évènement les objets peuvent être désignés par des variables qui seront liées aux objets de l'opération de déclenchement. Ces variables liées peuvent être utilisées dans les autres clauses.

Dans le projet HIPAC cet aspect a aussi été approfondi. Il est intéressant d'en parler pour prendre en compte les cas où un évènement est plus complexe qu'une opération de base appliquée à un seul objet.

Un peu à la manière de Gibbs la spécification de la clause évènement comprend la désignation d'une opération et une liste d'arguments (les objets auxquels l'opération fait référence) : ceci pour un évènement élémentaire (non composé). Quand une opération de déclenchement a lieu, la liaison des objets de l'opération aux arguments de la clause évènement est réalisée. Ceci est fait deux fois : avant l'opération (les arguments réflètent leur état avant l'opération) et après l'opération (les arguments réflètent leur état après l'opération). La liaison des arguments peut être faite par le système, par exemple pour les opérations bases de données, ou par l'utilisateur pour les opérations structurées définies par lui même, ou par des modules spécialisés pour les évènements externes.

Quand un évènement est une composition d'évènements élémentaires le contexte passé doit être aussi bien défini, par exemple :

---

[2]Remarquons que cet aspect (le passage du contexte) est aussi liée à comment l'exécution d'un déclencheur est faite : de façon ensembliste ou instance par instance. Pour ceci se référencer au paragraphe 2.1.3.

- le contexte de la disjonction de deux évènements - ev1 ou ev2 - est le contexte de ev1 ou le contexte de ev2.
- le contexte de la conjonction de deux évènements - ev1 et ev2 - est l'union du contexte de ev1 et ev2.
- le contexte de la fermeture - ev1 : ev2 - est l'union des contextes des différentes occurences de ev1 et du contexte de ev2.

Un autre point qui a été traité dans le projet HIPAC est le passage des informations utilisées par l'évaluation de la condition à l'action. Une condition étant une liste d'énoncés d'interrogation, le résultat (i. e. les objets retournés) de l'évaluation de ces énoncés est aussi passé à l'action.

Dans certains systèmes relationnels la condition est spécifiée de façon implicite et algorithmique à l'intérieur de l'action. Dans ces cas et selon la puissance du langage pour décrire l'action (par exemple autorisation de l'utilisation de variables), les résultats de l'évaluation de la condition peuvent être utilisés au niveau de l'action.

Une réflexion mérite d'être faite ici, concernant la structure des informations du contexte. Comme il a déjà été dit (et comme il sera développé dans le paragraphe 2.1.5.1), les moments où l'opération de déclenchement a lieu, l'évaluation de la condition est effectuée et l'action exécutée, peuvent être éloignés ; en conséquence le contexte doit être stocké pour pouvoir être récupéré au moment approprié.

Quelles sont les structures de stockage de ces informations ? [3]

Les différentes propositions faites jusqu'ici (même pour HIPAC) sont caractérisées par le fait que le contexte est stocké dans une seule relation. Ce qu'il importe de dire est que plus la structure de stockage est importante, plus complexe est leur gestion. Par extension, nous pouvons considérer le contexte comme une photographie d'une partie de l'état de la base de données au moment où l'évènement a lieu. Ceci rejoint le problème des "snapshots" [Adi81] ou même, de la gestion des historiques [Adi87].

Ce problème prend plus d'ampleur quand la partie action d'un déclencheur est une tâche qui peut être réalisée longtemps après l'opération de déclenchement. Bien sûr, ceci est un des points critiques développés dans le projet DOEOIS ; dans le chapitre 4 la notion de contexte d'activation propre à une activité est présentée.

2.1.3. Orientation Ensembliste Versus Instance par Instance

Le problème considéré dans ce paragraphe est résumé dans le fait qu'un évènement soit vu comme une opération ensembliste ou non. Par exemple, supposons un déclencheur ayant comme évènement la suppression d'employés ; supposons aussi qu'une suppression d'employés a lieu impliquant la destruction de *n* employés ; faut-il activer *n* déclencheurs, un pour chaque employé supprimé - **orientation instance par instance** - ou faut-il activer un seul déclencheur qui traitera

[3]Même si l'exécution du déclencheur est faite immédiatement après son activation, ce problème persiste car les données du contexte doivent être accessibles en utilisant le même formalisme que pour accéder à la base de données.

les *n* employés - **orientation ensembliste** ?

Les deux solutions ont été proposées : Sybase, Starburst et POSTGRES adoptent l'orientation ensembliste ; Système R et [Gib84] l'orientation instance par instance ; SQL3 et HIPAC permettent d'adopter la stratégie d'activation à la définition du déclencheur.

Les systèmes relationnels - Starburst en particulier - soutiennent que la spécification des déclencheurs selon l'orientation ensembliste est en accord avec l'orientation ensembliste de ces SGBDs. De plus, l'évaluation de requêtes dans de tels SGBDs applique des techniques d'optimisation qui pourraient être utiles à l'évaluation des déclencheurs. Un autre point qu'ils mettent en évidence est le fait qu'une telle orientation permet la spécification de conditions et actions qui ne sont pas exprimables par des déclencheurs qui suivent l'orientation instance par instance.

Mais ils soulignent aussi que des déclencheurs instance par instance peuvent subir une compilation dans le but de les exécuter de façon ensembliste [Mai88]. Néanmoins, le fait d'utiliser directement l'approche ensembliste peut éviter certaines complications, voir limitations.

En conclusion, je pense que l'important à retenir est que la façon d'implémenter, i. e., de concevoir, la condition et l'action est très dépendante de l'approche choisie.

Comme nous allons le voir dans 4.1 l'approche suivie dans DOEOIS est différente : elle est basée sur la notion d'entité focale. Très brièvement, une activité est habilitée à être exécutée (i.e. est activée) pour chaque instance de l'entité focale participant à l'opération de déclenchement et qui vérifie la condition.

2.1.4. Inclusion dans le Modèle

Un autre point important à étudier est la façon dont le concept de déclencheur est inclus dans le modèle et quelles sont les opérations permises sur les déclencheurs.

2.1.4.1. Définition

Au niveau de la définition, ce qu'on trouve plus couramment est que le concept de déclencheur est **indépendant** des concepts utilisés pour modéliser les données. Et de nouveaux constructeurs sont offerts pour créer et supprimer les déclencheurs.

Très récemment, des approches de **couplage** entre le modèle de données et celui de déclencheur ont été proposées p.ex., HIPAC, DOEOIS. Un déclencheur étant alors considéré comme une donnée spéciale - une donnée active - qui est manipulée de la même façon que n'importe quelle autre donnée.

Dans cette approche la différence entre une donnée active et les autres données est mise en oeuvre au niveau de l'implémentation. C'est-à-dire que le système reconnaît les données actives et les traite en conséquence. Par exemple, il les déclenchent automatiquement.

Au niveau conceptuel cette approche a l'avantage que les différents constructeurs offerts pour la conception des schémas sont aussi appliqués aux déclencheurs. Ainsi, suivant le modèle de données nous pouvons par exemple, lier les déclencheurs aux autres objets de l'application, définir des propriétés aux déclencheurs, créer des sous-classes de déclencheurs avec des propriétés spécifiques.

Ceci permet de regrouper les déclencheurs selon différents critères, ce qui est utile au niveau conceptuel - pour définir différents types d'exploitation de la base de données - mais aussi au niveau implémentation - pour réduire l'ampleur de la recherche des déclencheurs à activer pour une certaine opération.

2.1.4.2. Manipulation

Au niveau manipulation le principal avantage de l'approche couplage a déjà été cité : même manipulation que les autres objets. On peut aussi souligner qu'en conséquence les opérations faites sur les déclencheurs sont soumises aux mêmes lois transactionnelles que les autres objets.

Plusieurs SGBDs relationnels imposent une restriction au niveau de l'opération de création d'un déclencheur. Cette restriction ne permet pas de créer plus de trois déclencheurs qui s'appliquent à la même relation : un déclencheur pour chaque type de mise à jour de base.

Comme on le verra dans 2.1.5.3, ceci est fortement lié à la puissance du modèle d'exécution des déclencheurs. En effet, cette restriction est imposée pour empêcher que plusieurs déclencheurs soient activés par une même opération.

Deux nouvelles opérations ont été définies pour manipuler les déclencheurs : **charger** et **décharger**.

- Un déclencheur qui n'est pas chargé est un déclencheur inactif, i.e., il ne peut pas être activé par une opération et il n'est donc pas exécuté. Ces déclencheurs sont en repos et les opérations qui pourraient les activer ne sont pas suivies ou contrôlées.

- Un déclencheur qui est chargé est un déclencheur qui est en action, qui peut être activé et exécuté. En conséquence, les opérations qui peuvent les activer sont suivies.

Les opérations de chargement et déchargement sont très peu présentées dans les SGBDs relationnels. Et, dans le cas où ils offrent ces possibilités, ces opérations ne s'appliquent qu'à **un et un seul** déclencheur à la fois. Alors que ce qui est intéressant est de pouvoir charger et décharger des **groupes** de déclencheurs qui correspondent à des moments ou des circonstances particulières de l'évolution de la base de données.

Remarquons que charger ou décharger des groupes de déclencheurs sélectivement peut être facilement mis en œuvre dans les modèles où le concept de déclencheur est couplé au modèle de

données. Par exemple, on peut charger telle sous-classe de déclencheurs ou les déclencheurs liés à tel objet, etc. Cette facilité est offerte dans DOEOIS ; plusieurs exemples de son utilité sont présentés dans le chapitre 4.

2.1.5. Sémantique de l'Exécution

Le modèle d'exécution des déclencheurs introduit plus précisément la sémantique du concept de déclencheur. Des exemples flagrants sont les systèmes Sybase et Starburst : ils adoptent pratiquement la même structure pour définir un déclencheur, néanmoins les modèles d'exécution sous-jacents sont très divergents, ce qui conduit à deux mécanismes sémantiquement distincts.

Trois moments dans l'exécution d'un déclencheur peuvent être distingués : ils correspondent au traitement de chacune des parties de la structure d'un déclencheur. Nous avons ainsi :

- l'**activation** d'un déclencheur à partir de la clause évènement
- l'**évaluation** de la condition
- l'**exécution** de l'action

L'activation d'un déclencheur a lieu quand un évènement qui correspond à la clause évènement du déclencheur se produit. L'activation doit déterminer le contexte dans lequel la condition va être évaluée et l'action exécutée.

Le modèle d'exécution des déclencheurs doit spécifier précisément comment l'exécution des déclencheurs est faite : à quel moment la condition est évaluée et à quel moment l'action est exécutée ; quel est le comportement de l'exécution d'un déclencheur par rapport à la transaction de l'opération de déclenchement ; comment l'exécution de plusieurs déclencheurs activés par un même évènement se produit et comment l'exécution de déclencheurs activés en cascade est prise en compte. Les prochains sous-paragraphes ont comme but de présenter et analyser les différents modes d'exécution.

2.1.5.1. Moments d'Intervention

Il vient d'être dit que le traitement d'un déclencheur à l'exécution est fait en trois étapes, il importe d'analyser les moments où chacune de ces étapes est accomplie.

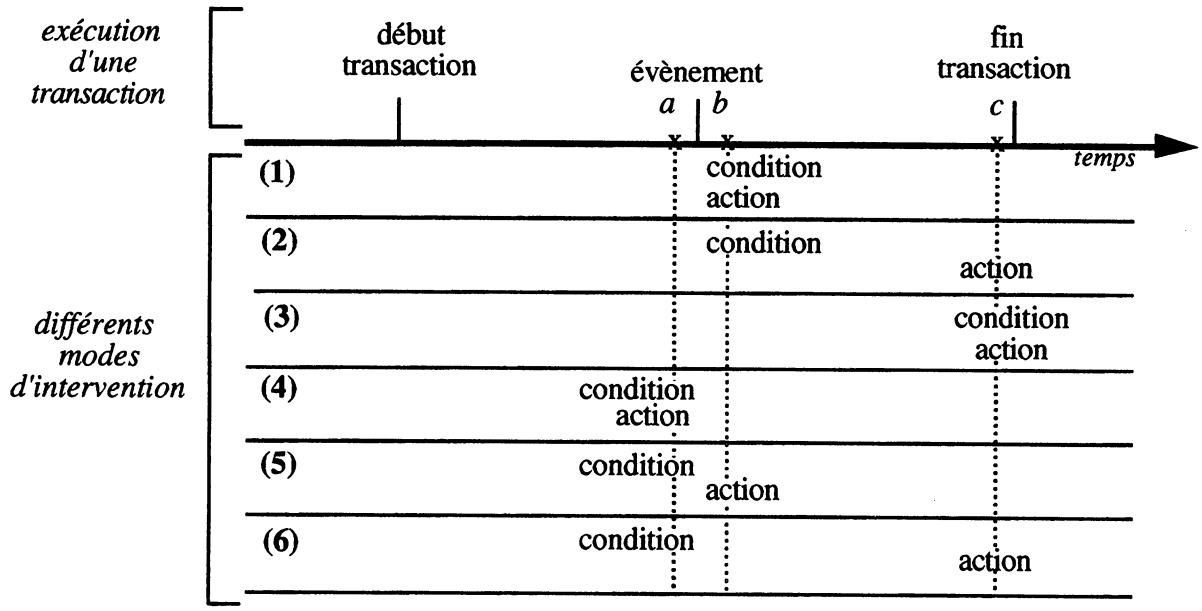
Nous suposons que toute mise à jour est réalisée à l'intérieur d'une transaction et nous prenons en compte deux moments : l'exécution d'une opération et la fin de la transaction qui englobe cette opération. Ceci nous mène à poser les questions suivantes :

- Est-ce que l'évaluation de la condition est faite immédiatement au moment où l'évènement a lieu ou postérieurement à la fin de la transaction ?

- Si l'évaluation est immédiate, est-ce qu'elle a lieu avant l'exécution de l'évènement de déclenchement ou après ?

- Est-ce que l'action est exécutée immédiatement après l'évaluation de la condition ou en différé à la fin de la transaction ?

Les réponses données par les différents systèmes de déclencheurs ont permis de définir six modes différents d'intervention : ils sont présentés dans la figure n° 2.3.



a juste avant l'évènement de déclenchement
b juste après l'évènement de déclenchement
c pendant la confirmation de la transaction

Figure n° 2.3 - Modes d'Intervention

En résumé :

- Les systèmes relationnels utilisent surtout les modes (1), p.ex., Sybase, SQL3, Système R, et (4), p.ex., SQL3, Système R. Certains prennent en compte le mode (2), p.ex., Starburst et [Ham75].
  Notons néanmons, que dans ces systèmes les différents modes d'intervention sont souvent implicites : une des raisons étant que, souvent, une distintion claire des trois parties du déclencheur n'est pas faite. Par exemple, dans plusieurs systèmes de déclencheurs la clause condition n'existe pas vraiment : elle est inclue implicitement dans la partie action, p.ex., Sybase, dont la condition et l'action sont forcément évaluées ensemble.

- Le système de déclencheur de HIPAC propose les trois modes (1), (2) et (3). Le mode à utiliser doit être explicitement spécifié au moment de la définition du déclencheur.

- Certains systèmes proposent les modes (5) et (6) comme dans [Gib84].

Nous n'avons pas trouvé de système où l'exécution de l'action est faite à un moment arbitraire : après la fin de la transaction à la demande d'un utilisateur ou d'une application. Ceci est néanmoins essentiel pour les systèmes d'information pour la bureautique comme nous allons le voir dans 4.6 et 5.2.

2.1.5.2. Transaction de Déclenchement et Exécution du Déclencheur

Un autre aspect très important relatif à l'exécution des déclencheurs et qui a de fortes implications dans la sémantique qui leur est attachée, est la manière dont l'exécution d'un déclencheur se comporte par rapport à la transaction de déclenchement (i.e., la transaction où l'opération de déclenchement a lieu).

Nous pouvons distinguer trois comportements (modes) différents : ils sont présentés dans la figure n° 2.4. Dans ce paragraphe, le terme "exécution du déclencheur" est utilisé pour désigner soit l'évaluation de la condition, soit l'exécution de l'action : chacun des trois modes peut être utilisé pour évaluer la condition ou pour exécuter l'action. Par exemple, dans HIPAC l'exécution d'un déclencheur peut utiliser le mode (B) pour évaluer la condition et le mode (C) pour exécuter l'action[4]. Remarquons aussi, que dans la figure nous ne prenons pas en compte les différents moments d'intervention : nous avons mis systématiquement l'exécution du déclencheur après l'évènement.

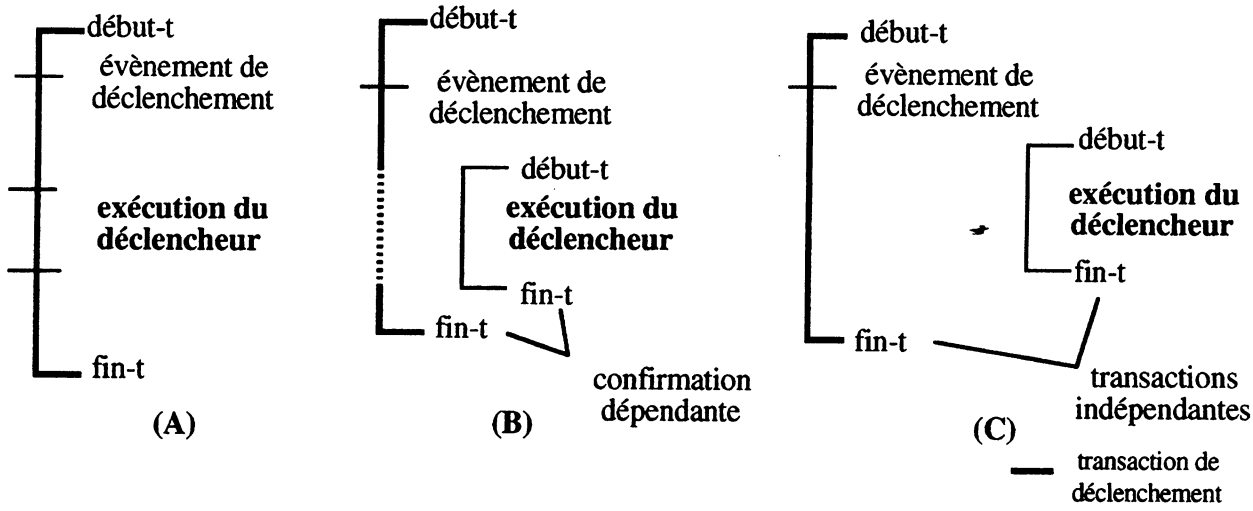


Figure n° 2.4 - Trois Modes d'Exécution d'un Déclencheur

Les trois modes d'exécution sont commentés ci-dessous :

(A) L'exécution du déclencheur **appartient à la transaction de déclenchement**. Ce mode est le plus répandu et quasiment le seul utilisé dans les SGBDs relationnels. Il est à noter que si pendant l'exécution du déclencheur il y a annulation de transaction alors toute la transaction (de déclenchement) est annulée.

[4]Un exemple est un déclencheur qui est activé quand le prix d'un article est mis à jour. Sa condition vérifie si le nouveau prix est différent de l'ancien, l'action affiche le nouveau prix.

(B) L'exécution du déclencheur est faite dans une **transaction imbriquée à confirmation dépendante** par rapport à la transaction de déclenchement. La transaction de déclenchement est suspendue pendant l'exécution de la transaction imbriquée.
Dans HIPAC ce mode est proposé en même temps que la possibilité d'annuler la transaction de déclenchement à partir de la transaction d'exécution du déclencheur. Bien que ce mode puisse permettre de modéliser d'autres situations que le mode (A), il est très intéressant dans les cas où un évènement active plusieurs déclencheurs : ceci sera étudié dans le prochain paragraphe.

(C) L'exécution du déclencheur est faite dans une **transaction complètement indépendante** de la transaction de déclenchement.
Ce mode, très peu proposé, est intéressant pour la modélisation de nouvelles applications comme la CAO ou la bureautique (cf. 4.6 et 5.2). Un exemple tout simple de son utilisation est la réalisation de statistiques sur l'utilisation de la base de données.

Pour finir, remarquons que toutes les combinaisons de modes pour l'évaluation de la condition et l'exécution de l'action ne sont pas valables. Par exemple, évaluer la condition dans une transaction séparée - mode (C) - et exécuter l'action avec les modes (A) ou (B) n'a pas de sens.

2.1.5.3. Exécution de Plusieurs Déclencheurs

Le sujet de ce paragraphe a comme point de départ la question suivante : Est-ce que plusieurs déclencheurs peuvent avoir la même clause évènement ? Si c'est possible, il est nécessaire d'analyser comment les différents déclencheurs activés par le même évènement sont exécutés.

Tout d'abord il est intéressant de savoir que dans la plupart des SGBDs relationnels - Sybase, SQL3 et Système R - un seul déclencheur peut être défini par évènement (i. e., un déclencheur par type de mise à jour sur une seule relation, cf. 2.1.4.2). Ainsi, la puissance du système de déclencheurs est très restreinte.

Dans les autres systèmes étudiés - et qui permettent la définition de plusieurs déclencheurs par évènement - deux types d'exécution sont possibles : les différents déclencheurs s'exécutent soit séquentiellement soit parallèlement. La figure 2.5 illustre les deux modes d'exécution de plusieurs déclencheurs activés par un même évènement. Le terme "exécution d'un déclencheur" a le même sens que dans le paragraphe précédent. De même, il n'est pas important ici, de savoir si l'exécution concurrente se fait par des transactions imbriquées dépendantes ou des transactions indépendantes ou si l'exécution séquentielle se fait dans la transaction de déclenchement ou par des transactions imbriquées exécutées séquentiellement.

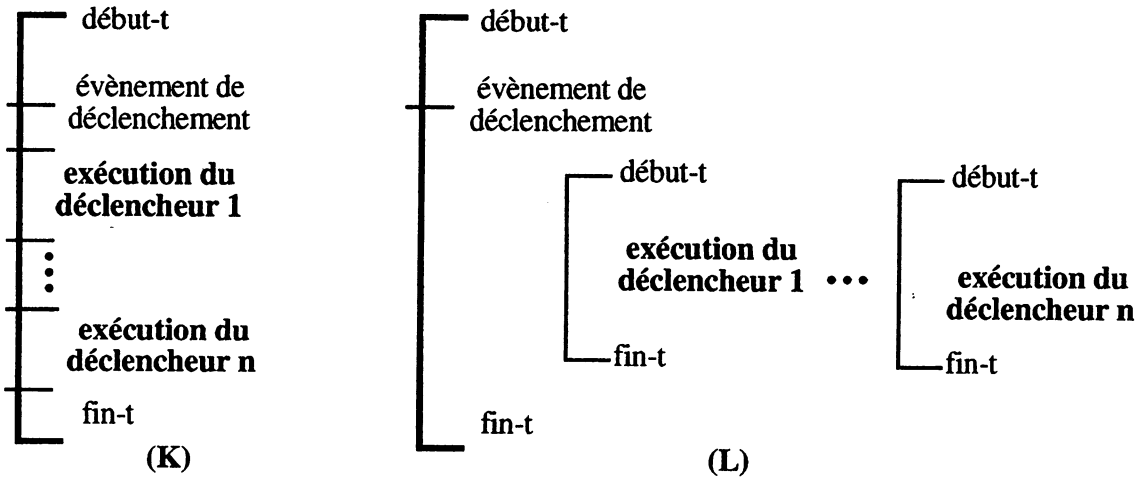


Figure n° 2.5 - Exécution Séquentielle (K) ou Parallèle (L)

Quelques commentaires sur les deux modes d'exécution suivent :

(A) **Exécution séquentielle**. Si le système de déclencheurs ne permet que l'exécution d'un déclencheur selon le mode (A) (cf. 2.1.5.2) alors l'exécution de plusieurs déclencheurs doit se faire forcément dans le mode (K).
Ce type d'exécution nécessite la résolution du conflit concernant l'ordre dans lequel les déclencheurs vont être exécutés. Deux SGBDs relationnels utilisent ce type d'exécution : POSTGRES résoud le conflit à partir de règles de priorité qui sont assignées statiquement à chaque déclencheur au moment de sa définition, Starburst propose plusieurs critères pour ordonner les déclencheurs de façon statique ou dynamique, i.e., à l'exécution. Dans ce dernier système tous les déclencheurs sont exécutés en fin de transaction ; ainsi, un des critères de choix dynamique est le plus récemment activé ou le moins récemment activé.

(B) **Exécution parallèle**. Si le système permet l'exécution d'un déclencheur selon le mode (B) ou (C) (cf. 2.1.5.2) alors l'exécution de plusieurs déclencheurs peut se faire selon le mode (L). Le modèle transactionnel sous-jacent au système doit gouverner la sémantique de l'exécution concurrente et assurer la sérialisabilité des différentes exécutions. Un exemple de système utilisant ce mode d'exécution est HIPAC.

2.1.5.4. Déclencheur Activé par Déclencheur

Le problème abordé dans ce paragraphe concerne le fait que l'exécution d'un déclencheur peut activer un autre déclencheur. Nous nommons ce phénomène déclenchement en cascade.

Tous les systèmes étudiés autorisent le déclenchement en cascade. Ceci concerne surtout la partie action où des mises à jour sont effectuées : rappelons nous que la plupart des systèmes n'accepte que des opérations de mise à jour dans la clause évènement (cf. 2.1.1.1).

D'après l'analyse des différents systèmes, trois critères pour juger de la sémantique attachée à

l'exécution d'une cascade de déclenchement ont été dégagés. Ils correspondent aux trois questions suivantes :

- Est-ce que l'exécution d'un déclencheur - qui provoque l'activation d'un autre déclencheur - est **interrompue** pour exécuter le déclencheur activé ?

- Est-ce que l'activation **récursive** du même déclencheur est permise ? i. e., peut-il y avoir l'exécution répétée d'un même déclencheur dans une cascade de déclenchement ?

- Est-ce que la **terminaison** d'une exécution en cascade est assurée ou non ?

En ce qui concerne le premier critère la figure 2.6 illustre les deux types d'exécution d'une cascade de déclenchement : mode (X) - l'exécution d'un déclencheur s'interrompt en faveur de l'exécution des déclencheurs qu'il active - et mode (Y) - l'exécution d'un déclencheur se fait jusqu'à la fin et ,seulement après, l'exécution des déclencheurs activés en conséquence a lieu.

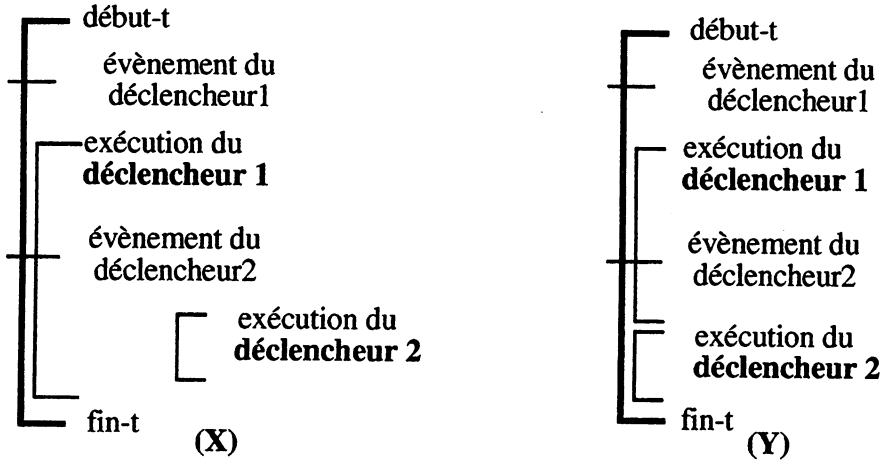


Figure n° 2.6 - Imbrication (X) ou Séquence (Y)

Le fait qu'un système utilise le mode (X) ou le mode (Y) n'est pas gratuit, et ceci dépend des modes d'intervention utilisés ainsi que des modes d'exécution. Il est donc intéressant de noter les corrélations suivantes :

- Si l'exécution d'un déclencheur est faite en fin de transaction - modes d'intervention (2), (3), et (6) - et le mode d'exécution est séquentiel (A), alors l'exécution d'une cascade de déclenchement est réalisée selon le mode (Y).

- Si l'exécution d'une méthode est faite immédiatement avec l'évènement - modes d'intervention (1), (4) et (5) - et le mode d'exécution est quelconque, alors l'exécution d'une cascade suit le mode (X).

- Si l'exécution d'une méthode est faite en fin de transaction - et le mode d'exécution est le (2) ou le (3), alors l'exécution d'une cascade applique le mode (X).

Analysons maintenant les différentes réponses au deuxième critère, i. e., l'activation récursive de déclencheurs dans une cascade. Si la terminaison de l'exécution d'une cascade est assurée ou non et comment ceci est fait, est aussi décrit ici. Nous distinguons deux cas :

- Le déclenchement récursif n'est pas permis. Regardons comment les trois systèmes - Sybase, SQL3 et POSTGRES - empêchent la récursivité.
  Dans Sybase l'exécution peut activer deux fois le même déclencheur, mais seul le premier déclencheur est exécuté. Un système où l'exécution d'une cascade de déclenchement n'admet pas la recursivité n'a pas besoin d'assurer la terminaison, néanmoins Sybase limite le nombre de niveaux de déclenchement en cacade à 16.
  SQL3 fait une vérification syntaxique, à la définition des déclencheurs, pour éviter la récursivité. Cette vérification est basée dans un graphe de définition de déclencheurs et évite plus d'une mise à jour à la même relation dans une cascade. Ceci implique que dans une cascade on ne peut exécuter que deux déclencheurs par relation.
  POSTGRES tient un graphe de dépendance de déclencheurs à l'exécution. La dépendance fait référence au fait qu'un déclencheur puisse modifier des données impliquant l'activation d'autres déclencheurs. Empêcher des cycles à l'intérieur d'un graphe implique interdire la récursivité.

- Le déclenchement récursif est permis. Dans ce cas nous avons : soit aucune facilité pour assurer la terminaison n'est offerte (HIPAC), soit des critères du type "time-out" (Système R) sont utilisés pour assurer la terminaison.

2.1.5.5. Résumé

En guise de conclusion de la partie sur la sémantique de l'exécution des déclencheurs regardons quelles sont les combinaisons les plus courantes et significatives.

La combinaison plus **classique**, utilisée dans Sybase, SQL3, Système R, Gibbs et POSTGRES[5], est la suivante :

- Mode d'intervention immédiat : (1) ou (4)
- Exécution du déclencheur appartenant à la transaction du déclencheur (A)
- Exécution séquentielle (K)
- Notion d'interruption (X)

Le système le plus **généralisé** et un des plus récents - HIPAC - adopte les modes suivants :

- Option entre les modes immédiat ou différé : (1) (2) (3)
- Option entre l'exécution dans une transaction imbriquée dépendante (B) ou une transaction indépendante (C)

[5]Postgres a quand-même une philosophie très différente des systèmes étudiés : ceci sera présenté dans le paragraphe 2.1.6

- Exécution parallèle de plusieurs déclencheurs (L)
- Notion d'interruption (X)

Nous pouvons bien sûr trouver des **variantes** comme Starburst où nous avons :
- Mode d'intervention différé (3)
- Exécution du déclencheur appartenant à la transaction du déclencheur (A)
- Exécution séquentielle (K)
- Pas de notion d'interruption (Y)

Le mécanisme de déclencheurs de Starburst[6] est surtout appliqué à la vérification de contraintes d'intégrité et en conséquence à la réalisation de mises à jour en cascade. Il peut être aussi appliqué au maintien de données dérivées.

En ce qui concerne les **activités bureautiques** telles qu'elles ont été définies dans le projet DOEOIS, les axes principaux de leur modèle d'exécution sont donnés par :
- Mode d'intervention : la condition est évaluée immédiatement avec l'évènement ; par contre, l'exécution de l'action peut être lancée soit automatiquement en fin de transaction (modes (2) ou (6)), soit manuellement à un moment arbritaire.
- L'évaluation des conditions de plusieurs activités déclenchées par le même évènement est réalisée de façon séquentielle (K). L'exécution des actions correspondantes peut être réalisée de façon parallèle en fin de transaction (si leur mode de lancement est automatique) ou de façon complètement arbritaire (si leur mode de lancement est manuel).
- L'exécution de l'action d'une activité peut impliquer la signalisation d'évènements ; à ce moment là, l'exécution est suspendue au bénéfice de l'évaluation des conditions et de la détermination des activités à habiliter (activer). Juste après, l'exécution suspendue est reprise vu que les activités habilitées en conséquence ne seront exécutées qu'en fin de transaction ou après.

2.1.6. Techniques d'Implémentation

Ce paragraphe présente des techniques d'implémentation de chacune des parties du traitement en exécution d'un déclencheur : activation du déclencheur, évaluation de la condition et exécution de l'action.

Une remarque d'ordre générale mérite d'être faite : le développement d'un mécanisme de déclencheurs a des implications dans un SGBD au niveau architectural, il importe alors de définir de façon précise les modules nécessaires à la mise en œuvre d'un tel mécanisme ainsi que leur interface avec les différentes parties du SGBD.

[6]Quelques précisions sur ce système méritent d'être présentés. En fait, la clause évènement d'un déclencheur utilise implicitement le constructeur de fermeture (cf. 2.1.1.1) i.e., toutes les mises à jour du même type réalisées sur la même relation n'activent qu'un seul déclencheur qui doit être exécuté en fin de transaction. Le contexte de ce déclencheur est l'état de la relation au moment de son exécution plus deux tables résumant les mises à jour faites pendant la transaction.

2.1.6.1. Activation

Un déclencheur est activé quand l'occurrence d'un évènement implique son exécution (i.e. l'évaluation de la condition et l'exécution de l'action en conséquence). Deux méthodes d'activation sont distinguées.

Première Méthode

Pour activer un déclencheur, le système doit tout d'abord détecter les évènements qui ont lieu. Suivant le type d'évènements acceptés, le système doit détecter des évènements base de données, recevoir des signaux externes ou temporels de la part des applications ou processus système, etc. [Day89]. La mise en correspondance entre un évènement qui vient d'être détecté et la clause évènement des différents déclencheurs doit alors déterminer quels sont les déclencheurs à activer. Si la clause évènement supporte des évènements composés, une étape de plus doit être réalisée pour déduire quels sont les déclencheurs activés. Selon le moment d'intervention (cf. 2.1.5.1) spécifié à la déclaration du déclencheur, celui-ci est exécuté tout de suite ou à un moment ultérieur.

La **détection** d'un évènement base de données est le plus souvent faite par le gestionnaire de requêtes du SGBD correspondant. Dans les systèmes orientés objets, la signalisation de l'évènement peut être incluse dans les méthodes qui implémentent les différentes requêtes (ou fonctions).

La **mise en correspondance** d'un évènement détecté avec la clause évènement des déclencheurs est faite en interrogeant des tables contenant une description de ces clauses. Plusieurs techniques, déjà connues, permettent d'accélérer ce type de recherche.

La mise en correspondance peut ne pas être le simple rapprochement syntaxique entre l'opération et les objets de l'évènement détecté et ceux de l'évènement de la clause du déclencheur. Ceci dépend en effet du modèle de données et du langage de manipulation. Dans le projet DOEOIS, ce problème a été étudié et a abouti à la définition d'un ensemble de règles guidant cette mise en correspondance (cf. chp. 6).

Deuxième Méthode

Cette méthode suit une philosophie complètement différente de la première, elle a été développée dans POSTGRES.

Deux types d'activation sont mis en œuvre. Pour les illustrer, prenons l'exemple du déclencheur suivant : "Le salaire de Jean doit être toujours mis à jour de façon à devenir égal au salaire de François". Le premier type d'activation implique la mise à jour du salaire de Jean au moment où le salaire de François change. Ce mécanisme est appelé "**early evaluation**" et l'exécution en cascade de déclencheurs selon ce type d'activation est connue sur le nom de "**forward chaining**". Le deuxième type d'activation implique que le salaire de Jean n'est changé (en fonction du salaire de François) qu'au moment où sa valeur est demandée. Ce dernier mécanisme, est appelé "**late**

evaluation" et le déclenchement en cascade reçoit le nom de "**backward chaining**".

L'utilisateur n'a pas besoin de connaître ces deux types d'activation : le système choisit lequel appliquer en utilisant des critères d'optimisation.

Un premier problème d'ordre sémantique se pose : dans certains cas l'application des deux modes d'activation peut produire des résultats différents. C'est le cas d'un déclencheur qui doit compter le nombre de fois où le salaire de François change [Day90]. Le système, n'ayant pas connaissance du comportement du déclencheur ne peut pas détecter des situations de ce type.

Ensuite trois restrictions doivent être imposées dans le choix du mode d'activation à appliquer. Elles correspondent aux problèmes suivants:

- Si la partie action du déclencheur est une procédure POSTQUEL le système doit avoir l'information sur le fait que le résultat de la procédure ne dépend pas du moment de l'exécution. Par exemple, toutes les procédures de contrôle d'accès qui nécessitent l'identité de l'utilisateur doivent être activées selon le mécanisme "early evaluation".

- Les objets liés à des déclencheurs activés en "late evaluation" ne peuvent pas être indexés car il n'y a pas moyen de connaître les valeurs actuelles des objets.

- Le mixage entre les déclencheurs activés en "late evaluation" et ceux activés en "early evaluation" peut mener à des incohérences. Supposons que le declencheur qui attribue à Jean le salaire de François est évalué en "late evaluation". Supposons aussi un autre déclencheur qui attribue à Paul le salaire de Jean activé en "early evaluation". Si le salaire de François change et qu'on questionne ensuite le salaire de Paul, la réponse sera erronée : le salaire de Jean n'a pas encore été mis à jour.

La mise en œuvre de ces mécanismes d'activation est faite en marquant les objets pour lesquels des déclencheurs ont été définis. Quand on accède aux objets (pour exécuter une requête), ces marques sont lues et les évènements signalés en conséquence. Le marquage est fait en utilisant des verrous physiques spéciaux. Bien sûr différents types de verrous sont prévus pour supporter les deux modes d'activation.

Des complications liées à ce type de signalisation d'évènements existent. D'abord, les verrous doivent être persistents et survivre aux pannes. Ensuite, il faut prévoir des situations du type suivant: à la définition du déclencheur de notre exemple, Jean n'existe pas encore dans la base de données. Et finalement, l'ampleur d'un déclencheur pouvant changer (p.ex., "donner le salaire X à tous les employés d'âge égal à François") le déclencheur doit être installé à nouveau pour verrouiler des objets différents.

Cette dernière complication vient du fait que dans POSTGRES l'activation du déclencheur et l'évaluation de la condition sont en effet confondues.

2.1.6.2. Evaluation

Jusqu'à récemment, les travaux menés pour le contrôle de l'intégrité sémantique constituaient quasiment le seul apport dans le domaine de l'évaluation efficace de prédicats sur la base de données.

Dans le cas général, vérifier qu'une contrainte d'intégrité est consistante avec l'état de la base de données, revient à évaluer une condition et à entreprendre une action en conséquence. Ainsi, les mécanismes d'évaluation des contraintes d'intégrité et les mécanismes de déclenchement d'actions sont proches.

Les méthodes d'évaluation de contraintes d'intégrité ont été un des points de départ du développement des mécanismes de déclencheurs. Néanmoins, l'analyse a montré que ces méthodes ne sont pas directement applicables à l'évaluation de conditions de déclenchement d'actions. La principale différence entre les deux mécanismes est que le concept d'intégrité sémantique est étroitement lié à la notion d'état valide de la base de données, alors que le fait qu'une condition soit vraie ou fausse n'a pas de rapport explicite avec la validité d'un état de la base de données. En conséquence, une contrainte d'intégrité ne peut pas être violée : une mise à jour qui provoque la non vérification d'une contrainte d'intégrité n'est pas exécutée. Par contre, le fait qu'une mise à jour implique la valeur vraie ou fausse d'une condition est sans conséquence sur l'exécution effective de la mise à jour.

Néanmoins, il est intéressant à ce niveau d'étudier les principes sur lesquels les méthodes d'évaluation de contraintes d'intégrité s'appuient et d'analyser la faculté d'adpatation de ces principes à l'évaluation des conditions de déclencheurs.

Afin de pouvoir présenter les différents principes et de les illustrer par des exemples, on présente ici le schéma d'une base de données relationnelle. En effet, les différentes méthodes d'évaluation de contraintes d'intégrité ont été développées sur des SGBDs Relationnels : ils ont été les premiers à proposer une base formelle pour décrire l'intégrité sémantique. Pour plus d'informations sur ce sujet le lecteur pourra se référer à [Ant87b].

SCHEMA DE LA BASE DE DONNEES

D = { DEPARTEMENT, EMPLOYE, FOURNISSEUR, ACHAT, VENTE, CATALOGUE }

SCHEMAS DES RELATIONS

DEPARTEMENT (DEPT,BUDGET, DIRECTEUR)
EMPLOYE (EMP, SALAIRE, DEPT)
FOURNISSEUR (ENTREPRISE, DEPT, ARTICLE)
ACHAT (DEPT, ARTICLE, COUT)
VENTE (DEPT, ARTICLE, PRIX)
CATALOGUE (ARTICLE, TYPE)

DOMAINE DES ATTRIBUTS

BUDGET, SALAIRE, COUT, PRIX - Réels avec deux décimales

DEPT, DIRECTEUR, EMP, ENTREPRISE, ARTICLE, TYPE - Chaîne de caractères

INTERPRETATION INTUITIVE

<DP, B, D> ∈ DEPARTEMENT ⇒ Le dépatement de nom DP a comme budget B et directeur D.

<E, S, D> ∈ EMPLOYE ⇒ L'employé de nom E a comme salaire S et département D.

<E, D, A> ∈ FOURNISSEUR ⇒ L'entreprise E fournit l'article A au département D.

<D, A, C> ∈ ACHAT ⇒ Le département D achète l'article A pour un coût C.

<D, A, P> ∈ VENTE ⇒ Le département D vend l'article A au prix de P.

<A, T> ∈ CATALOGUE ⇒ L'article A est du type T.

Nous présentons maintenant les différents principes. Il faut remarquer qu'il ne s'agit pas d'un ensemble homogène de principes pour construire une méthode. Ainsi, on peut trouver par exemple des principes qui sont opposés ou des principes qui se recouvrent.

1. **Tirer profit du fait que la base de données est dans un état valide avant chaque requête de mise à jour.** Par conséquent, quand la base de données est mise à jour, seule la partie de la base de données dont l'intégrité peut être affectée par la mise à jour doit être évaluée. Ceci permet de déterminer le sous-ensemble de l'ensemble des contraintes d'intégrité, qui doit être évalué pour une mise à jour et dériver une forme simplifiée pour chacune des contraintes d'intégrité de ce sous-ensemble qui soit suffisante pour évaluer la contrainte d'intégrité originale.

Exemple 2.1 :

C.I.	: "Le salaire de **chaque** employé doit être supérieur à 5 000FF"
m.à.j.	: Insérer un nouvel employé
C.I. simplifiée	: "**Le nouvel** employé doit gagner plus de 5 000FF par mois"

Exemple 2.2 :

C.I.	: "**Une entreprise** ne peut fournir l'article A qu'à un seul département"
m.à.j.	: Insérer le n-uplet <E, D, A> dans la relation FOURNISSEUR
C.I. simplifiée	: "**L'entreprise E** ne doit pas fournir l'article A à un département différent de D.

Il est clair que ce principe échoue par rapport à l'évaluation des conditions des déclencheurs. Remarquons néanmoins, que si on sait que le déclencheur implémente une contrainte d'intégrité, ce principe peut être utilisé.

On peut aussi utiliser la connaissance sur la clause action du déclencheur pour déterminer si la condition est toujours fausse avant chaque mise à jour, auquel cas ce principe devient

applicable. Ceci n'est quand même pas facile à implémenter à l'exception des systèmes qui n'acceptent dans la clause action qu'une classe très restreinte et simple d'énoncés.

2. **Utiliser de l'information redondante.** La base de données D est étendue avec des informations redondantes qui résument le contenu d'un ensemble de données de D. L'évaluation des contraintes d'intégrité est faite en utilisant ces informations plutôt que toutes les valeurs de l'ensemble qu'elles caractérisent. Le coût de maintien de ces informations doit être faible par rapport au gain qu'elles permettent. Ces informations peuvent être :

   - **Des fonctions d'agrégation**

     Exemple 2.3 :

     C.I. : "Le coût d'achat doit être plus petit que le prix de vente"

     m.à.j. : Insérer le n-uplet <D, A, 50> dans la relation ACHATS

     C.I. modifiée : "**Le minimum** des prix de vente de l'article A, pour le département D, doit être supérieur à 50FF

   - **Des ensembles d'informations**

     Exemple 2.4 :

     C.I. : "Il y a au moins un article de type T3 qui est fourni par toutes les entreprises"

     m.à.j. : Insérer <E, D, A> dans la relation FOURNISSEUR

     On suppose que le type de l'article est T3, et qu'il s'agit d'une nouvelle entreprise. Dans les autres cas la contrainte ne serait pas violée.

     Ensemble d'informations connu : L'ensemble de tous les articles qui vérifient la contrainte, nommé VT3

     C.I. modifiée : " L'article A appartient à l'ensemble VT3".

     Ce principe est tout a fait applicable à l'évaluation des conditions. Mais remarquons que dans certains cas, il est surtout intéressant quand la condition est évaluée à faux. Ceci parce qu'il se peut que, si la condition est évaluée à vrai, il faut quand même accéder aux valeurs (qu'on a évité de consulter pendant l'évaluation) pour construire le contexte qui doit être passé à l'action (cf. 2.1.2).

3. **Modifier la contrainte d'intégrité en fonction de la requête de mise à jour** de façon à ce que la contrainte puisse être évaluée dans la base de données avant l'exécution de la mise à jour. Ceci évite que la mise à jour soit défaite en cas de violation de la contrainte.
   Tous les exemples ci-dessus ont suivi ce principe : reprenons le dernier.

Exemple 2.5 :

C.I. : "Il y a au moins un article de type T3 qui est fourni par toutes les entreprises"

m.à.j. : Insérer <E, D, A> dans la relation FOURNISSEUR

C.I. simplifiée : "L'article A est fourni par toutes les entreprises"

Le but essentiel de ce principe pour l'évaluation des contraintes d'intégrité est d'éviter que la mise à jour soit défaite. Or ceci n'a pas de sens en ce qui concerne l'évaluation des conditions. Ce principe a permis aussi le développement de méthodes de simplification des contraintes d'intégrité [Ber80] [Ber81] [Bla81] [Hsu85]. Par rapport à l'évaluation des conditions, remarquons que la forme du prédicat d'un déclencheur, évènement - condition, applique intrinsèquement ce principe (cf. 2.1.1).

4. **Chaque mise à jour est modifiée de façon à ne pas violer les contraintes d'intégrité.** La partie qualification de chaque requête de mise à jour doit être étendue de façon à déterminer les n-uplets à modifier et les n-uplets qui ne violent pas la contrainte.

Exemple 2.6 :

C.I. : "Le salaire d'un employé doit être supérieur à 5000FF"

m.à.j. : Insérer le n-uplet <Michelin, 7000, D> dans la relation EMPLOYE

m.à.j. modifiée : Insérer l'employé <Michelin, 7000, D> **où 7000 > 5000**

Exemple 2.7 :

C.I. : "Les employés doivent gagner moins de dix fois le budget de leur département"

m.à.j. : Augmenter de 500 FF le salaire de l'employé 'Falido'

m.à.j. modifiée : Augmenter de 500 FF le salaire de l'employé 'Falido' qui a un salaire, **qui augmenté de 500FF, est moins que dix fois le budget de son département.**

Ce principe a abouti au développement de la méthode "Query Modification" [Sto75].
Bien sûr ce principe n'est pas applicable car, comme il était dit, la valeur vraie ou fausse de la condition n'a aucune conséquence pour l'exécution d'une mise à jour.

5. **Utiliser la connaissance de la contrainte d'intégrité plus la connaissance de la requête de mise à jour**, pour identifier les conditions qui peuvent causer la violation de la contrainte. Sur ce principe deux types de méthodes peuvent être distinguées :
   - Celles du premier type s'appliquent à un sous ensemble de contraintes particulières

     Exemple 2.8 :

     C.I. : "Les employés doivent gagner moins de dix fois le budget de leur département"

     m.à.j. : Augmenter de 10 000 le budget du département D

     Preuve : Puisque le budget augmente la C.I. ne sera pas violée

- Celles du deuxième type utilisent un démonstrateur de théorèmes.

Ce prinipe est la base de certaines méthodes de vérification des contraites d'intégrité : [Ham78b] et [Sar77].
Le problème avec ce principe par rapport à l'évaluation des conditions de déclencheurs est que les méthodes développées sur lui appliquent très souvent le principe 1. Il y a néanmoins certains cas où on peut utiliser la connaissance de la mise à jour - en particulier les constantes contenues dans la mise à jour - pour évaluer la condition en évitant des accès à la base de données. Nous avons développé ceci dans le projet DOEOIS, nous utilisons les informations contenues dans la mise à jour ainsi que toutes les informations acquises pour l'exécuter afin d'éviter des accès répétés à la base de données. HIPAC [Day90] propose aussi l'application de ceci : il nomme la technique de combinaison de la détection de l'évènement et de l'évaluation de la condition.

6. **Générer des requêtes les plus sélectives possible pour tester la contrainte d'intégrité.** Une fois qu'une contrainte a été simplifiée (selon les principes 1 et 3 par exemple), l'étape suivante est de la tester dans l'état courant de la base de données. C'est-à-dire de la traduire en interrogations et calculs qui soient suffisants pour déterminer si la contrainte simplifiée est vraie dans l'état courant de la base de données.

Exemple 2.9 :

C.I.	: **Chaque article** dans la relation CATALOGUE doit être acheté et vendu"
m.à.j.	: Insérer le n-uplet <A, T1> dans la relation CATALOGUE
C.I. simplifiée	: "**L'article A** doit être acheté et vendu"
Requêtes	: Y-a-t-il un n-uplet dans la relation ACHATS dont l'attribut ARTICLE a comme valeur A? Y-a-t-il un n-uplet dans la relation VENTES dont l'attribut ARTICLE a comme valeur A?
Calculs	: La réponse à chacune des requêtes ne doit pas être vide

Cette solution est applicable à l'évaluation de conditions. Remarquons que souvent la forme d'une condition est déjà un énoncé d'interrogation et son évaluation peut donc jouir des techniques déjà développées pour l'optimisation de requêtes.

7. **Isoler les n-uplets à mettre à jour** dans des relations temporaires. Ce principe est opposé au principe 4 où le contrôle d'intégrité n'est pas séparé de l'exécution des requêtes.
Le gain des méthodes appliquant ce principe est important si la taille des relations temporaires est petite par rapport à la relation permanente.
Ce principe est aussi applicable à l'évaluation de conditions. Par exemple Sybase et Starburst utilisent explicitement les relations "inserted" et "deleted" qui résument les n-uplets modifiés par une mise à jour. HIPAC utilise, pour évaluer une condition, des relations "delta" qui

combinent les relations "inserted" et "deleted". Dans ce projet il a été développé une algèbre basée sur ces relations [Bla89]. La relation "delta" permet de traiter tous les types de changement de la base de données de façon uniforme et permet de traiter aisément les contraintes de transition. Ce dernier avantage est important vu que l'algèbre relationnelle utilisée pour exprimer des interrogations et manipuler des objets ne supporte pas de façon élégante le concept de changement, essentiel pour exprimer des conditions qui incluent des états différents de la base de données.

8. **Evaluer par ordre croissant de complexité** les contraintes d'intégrité qui peuvent être violées lors d'une mise à jour. Ce principe ne s'applique pas à l'évaluation d'une seule contrainte comme les principes 2 à 7, mais à l'ensemble des contraintes d'intégrité à évaluer pour une mise à jour donnée. C'est par exemple : l'évaluation d'abord des contraintes intra-relationnelles et après des contraintes inter-relationnelles.
   Remarquons que ce principe appliqué à l'évaluation d'un ensemble de contraintes d'intégrité est important car si on commence par évaluer une contrainte simple et si elle est violée alors la mise à jour ne se fait pas et les autres contraintes - plus complexes - ne sont pas évaluées. Ceci ne s'applique pas à l'évaluation d'un ensemble de conditions vu que toutes les conditions doivent être évaluées.
   Une application du principe à l'évaluation de conditions de déclencheurs concerne l'évaluation d'une seule condition : évaluer les sous-expressions de la condition par ordre de complexité croissante.
   D'autre part, quand un ensemble de conditions doit être évalué, le principe suivant peut être appliqué : **exploiter les sous-expressions communes des différentes conditions** à évaluer. Ces sous-expressions sont évaluées une seule fois au moment de la mise à jour qui provoque l'évaluation de l'ensemble des conditions. Ce principe a été développé par nous (cf. chapitre 7) et proposé aussi par Dayal [Day90].

Pour finir ce paragraphe, il est intéressant de dire que les différentes implémentations n'appliquent quasiment pas de techniques d'optimisation. Sybase et Starburst proposent l'utilisation des relations "inserted" et "deleted" mais ceci doit être fait explicitement dans l'expression du déclencheur ce qui n'est pas très élégant. Le système R, [Ham75] et SQL3 n'ont pas produit de prototype et n'ont pas spécifié non plus une méthode efficace d'évaluation. [Gib84] implémente dans un prototype les déclencheurs : pas de technique particulière pour l'optimisation de l'évaluation; néanmoins les conditions qu'il accepte sont très simples. HIPAC a donc proposé plusieurs optimisations ; sont-elles implémentées au niveau du prototype ? De plus, chez HIPAC beaucoup de propositions sont plutôt des idées n'ayant pas été développées jusqu'à aujourd'hui.

2.1.6.3. Exécution

Ce paragraphe est dédié aux systèmes de déclencheurs plus avancés qui permettent des appels à des programmes d'application dans leur clause action (p.ex. HIPAC, DOEOIS). L'exécution de ces applications ne s'effectue pas à l'intérieur du SGBD mais par nouveaux processus. L'important à

dire est qu'une **architecture de communication** doit alors être définie : néanmoins, ceci ne pose pas de problème majeur.

Ces systèmes peuvent être utilisés comme une nouvelle méthode "d'interfaçage" entre les applications externes et le SGBD. Pour finir, il est intéressant de faire connaître les constatations citées dans [Day89] après la modélisation d'une application avec HIPAC :

- Les interactions entre les programmes d'application sont faites via les déclencheurs.
- Les programmes d'applications tendent à être plus simples car le contrôle logique est codé dans les déclencheurs.
- La plupart des déclencheurs contiennent des appels à des programmes d'application plutôt que des opérations base de données.
- Pour modifier le comportement d'une application les déclencheurs sont modifiés plutôt que les programmes eux-mêmes.

2.2. Modélisation de Long Processus :TAXIS

TAXIS a développé un modèle sémantique pour la conception de systèmes d'information interactifs. Deux aspects importants de ce projet sont la gestion de longs processus ("scripts") et la vérification de contraintes d'intégrité sémantiques (assertions).

2.2.1. Modélisation d'un "Script"

Un "script" est construit à partir d'un réseau de Petri augmenté : (i) d'assertions liées aux différents états ; (ii) de conditions d'activation et de listes d'actions associées aux transitions ; (iii) de primitives de communication inter-processus. La figure n° 2.7 illustre le "script gestion d'un employé".

Script GestionEmployé (Employé)

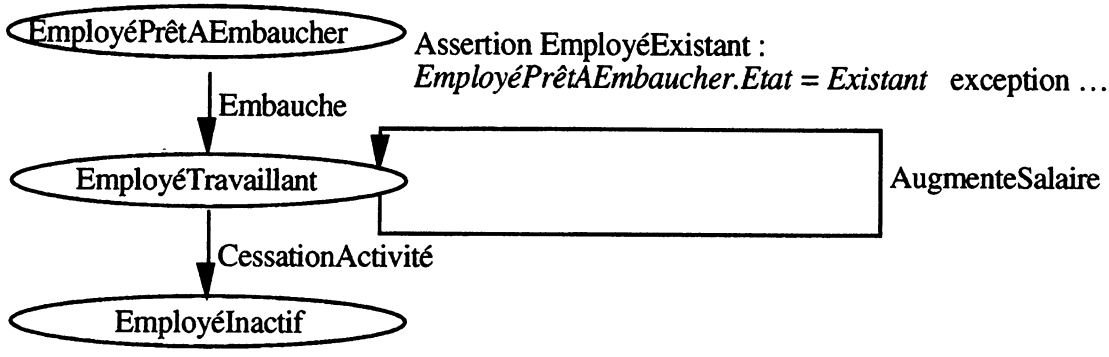


Figure n° 2.7 - Le "Script Gestion d'un Employé"

Ce "script" gère le salaire d'un employé depuis qu'il est embauché jusqu'à ce qu'il soit parti. Il est supposé que le salaire d'un employé est augmenté chaque année. Le réseau de Petri qui spécifie ce "script" est constitué de trois états : EmployéPrêtAEmbaucher, EmployéTravaillant et EmployéInactif ; et de trois transitions : Embauche, AugmenteSalaire et CessationActivité. La première transition attribue un salaire à un employé et affecte une date à une variable globale : DateAugmenteSalaire. La deuxième transition augmente le salaire de l'employé et met à jour la variable DateAugmenteSalaire.

Un exemple de condition est c1 qui est associée à la deuxième transition : elle exige que la date actuelle soit supérieure à la date d'augmention du salaire pour que la transition puisse être lancée.

Les trois états sont associés à la classe de données Employé. Un exemple d'assertion liée à un état est l'assertion EmployéExistant associée à EmployéPrêtAEmbaucher. Elle exige que l'employé existe toujours pour que la transition Embauche puisse avoir lieu.

Les "scripts" ont été complètement intégrés au contexte de TAXIS qui a développé un modèle orienté entité supportant les concepts de classification, généralisation et agrégation. Les instances de "scripts" comme toutes les autres entités sont assemblées en classes et ces classes sont organisées dans une hiérarchie de généralisation avec leur états et transitions définis en termes d'agrégation.

En effet, TAXIS offre plusieurs types de classes. Nous avons par exemple les "**DataClasses**" (modélisant des données), les "**TransactionClasses**" (modélisant les transitions de la base de données et les processus de courte durée), les "**ExceptionClasses**" (modélisant des exceptions qui peuvent surgir lors de l'exécution d'une instance de "TransactionClasses"), les "**StateClasses**" (modélisant les états d'un "script"), les "**ScriptClasses**" (modélisant les processus de longue durée, comparables à des procédures dans le Fact Model), etc...

Pour chaque type de classes, un ensemble de **catégories d'attributs** leur est associé. Ainsi par exemple, les "**DataClasses**" ont des attributs des catégories "unchangeable attributs", "changeable atttributs", "computed attributs", ... ; les "**TransactionClasses**" ont des attributs des catégories "parameters", "locals", "prerequisites" (préconditions), "actions", "postrequisites" (post-conditions) ; les "**StateClasses**" ont entre autre des attributs de la catégorie "assertions" ; les "**ScriptClasses**" ont des attributs des catégories "states", "transitions", etc...

Un des soucis de TAXIS a été que la définition et la manipulation des attributs des différents types de classes soient traitées - dans la mesure du possible - **uniformément** à travers les différentes catégories ; ceci en ce qui concerne la définition du langage et l'implémentation.

2.2.2. Etats d'un "Script"

Pour lancer une transition deux conditions sont requises. D'abord tous les états d'entrée de la transition (p.ex., EmployéPrêtAEmbaucher est un état d'entrée de la transition Embauche) doivent être actifs. Ensuite, toutes les conditions de la transition doivent être satisfaites. Une fois que ces deux conditions sont accomplies, la transition est lancée, ses actions sont exécutées et si les post-conditions sont toutes vraies alors ses états d'entrée sont "désactivés" et ses états de sortie activés (p.ex., EmployéTravailant est un état de sortie de la transition Embauche).

De plus, l'exécution des actions peut être suspendue par la communication "inter-script" ou par l'interaction entre l'utilisateur et le système. Elle est reprise à la réception d'un message de la part d'un autre "script". Ainsi, les transitions ne sont pas traitées comme des évènements ponctuels : elles peuvent être actives pendant de longues périodes de temps (de l'ordre de quelques jours ou mois) dans le but de la synchronisation entre processus ou entre l'utilisateur et le système.

Les différents états d'une transition sont résumés dans la figure 2.8.

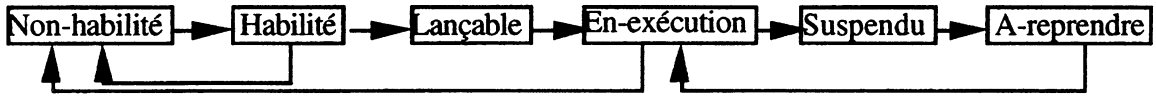


Figure n° 2.8 - Les Etats d'une Instance d'une Transition

Le sens de chacun de ces états est :

Non-habilité	:	les états d'entrée ne sont pas tous actifs.
Habilité	:	les états d'entrée sont tous actifs mais les conditions de transition ne sont pas vérifiées.
Lançable	:	les états d'entrée sont tous actifs et les conditions de transition sont vérifiées.
En-exécution	:	les actions sont en train d'être exécutées, ou les post-conditions en train d'être testées, ou les états d'entrée et de sortie en train d'être activés ou désactivés. Une fois l'exécution terminée, la transition revient à l'état Non-habilité.
Suspendu	:	en attente à cause d'une commande de synchronisation.
A-reprendre	:	un message de synchronisation de la part d'un autre "script" a été reçu et la transition peut reprendre son exécution.

2.2.3. Techniques d'Implémentation

De façon générale il y a un grand nombre d'instances de "scripts" actives, leur gestion doit être faite de façon efficace. Il est nécessaire de faire appel à une statégie pour déterminer et faire la gestion des transitions habilitées à être activées (exécutées) ou qui sont actuellement activées. Ainsi, il semble important d'étudier comment la gestion des transitions dans les états non-habilité, habilité et lançable est faite. TAXIS analyse deux techniques : l'une a été développée par Zisman [Zis78] et l'autre basée sur un système de déclencheurs a été développée par eux-mêmes.

Stratégie "Vérification Cyclique"

Cette stratégie est celle développée par Zisman. Elle maintient un ensemble contenant toutes les transitions habilitées : cet ensemble est balayé de façon séquentielle et cyclique pour évaluer les conditions des différentes transitions et en trouver une qui soit lançable. Une fois que cette transition est lancée, le balayage continue.

Pendant ce balayage, les états d'entrée des transitions sont aussi analysés : si une transition est

devenue non-habilitée elle est supprimée de l'ensemble.

Cette stratégie est performante si le nombre de transitions lançables est proche du nombre de transitions habilitées. Néanmoins, il a été constaté que dans la plupart des cas le nombre de transitions habilitées est nettement supérieur au nombre de transitions lançables. L'application de cette stratégie dans cette situation n'est pas du tout performante.

Stratégie "Déclenchement Automatique"

L'idée de base de cette stratégie est de traiter les conditions des transitions comme des conditions de déclenchement. Un mécanisme de déclencheur doit alors évaluer ces conditions à l'occurence d'un évènement qui peut modifier leur valeur.

Chaque transition lançable doit être lancée : pour ceci, si le système utilise des machines parallèles, elles peuvent être lancées de façon concurrente ; dans le cas contraire, elles sont mises dans une file d'attente. La gestion de cette file n'est pas importante ici, on souligne uniquement que toute transition de la file sera lancée : elles ne peuvent pas passer à l'état habilité ou non-habilité.

Les problèmes de performance de cette méthode sont liés à l'efficacité du mécanisme de déclenchement sous-jacent. Pour développer un tel mécanisme, TAXIS utilise plusieurs techniques d'optimisation. Ces techniques ont été tout d'abord développées pour l'évaluation efficace des assertions liées aux classes de données (entités). En effet, l'expression d'une condition est la même que celle d'une assertion, exception faite qu'une condition peut contenir des prédicats pour la communication inter-script ou pour l'interaction utilisateur / système.

Les principales techniques utilisées sont :

- Utilisation de la méthode de Hammer et Sarin [Sar77] [Ham78] : à partir de la condition déterminer statiquement un test simple tel que si ce dernier est vrai au moment de l'exécution de la requête l'évaluation de la contrainte initiale soit évitée. Les tests sont basés sur la comparaison des valeurs avant et après une mise à jour (cf. exemple 2.8).
- Utilisation de l'information redondante [Ber80] [Bla81] : maximum et minimum d'un ensemble de valeurs (cf. exemple 2.3).
- Evaluation de sous-expressions communes (cf. 2.1.6.2).

L'expression des conditions est basée sur la logique du premier ordre. Plusieurs restrictions à ces expressions ont été imposées pour éviter l'évaluation de conditions trop complexes.

2.2.4. Conclusion

Au niveau conceptuel deux points paraissent importants :

- La modélisation d'une longue séquence de travail pourrait être réalisée par un ensemble de déclencheurs. Néanmoins, cette approche est moins riche que celle qui vient d'être décrite, cette dernière offrant un outil qui permet la modélisation de plusieurs séquences de travail comme une unité cohérente. Même si on peut comparer une transition à un déclencheur, la notion de "script" offre un niveau de conception supérieur.

- L'intégration du concept de "script" au modèle de données de TAXIS est très importante, vu que dans la mesure du possible les "scripts" peuvent être manipulés en utilisant les mêmes facilités qui sont offertes pour manipuler les autres entités.

Au niveau implémentation :

- Par rapport à un mécanisme de déclencheurs, l'implémentation des "scripts" nécessite le stockage des différents états de leurs instances ainsi que leur gestion (p.ex. le passage d'un état à un autre).

- Il est très important de voir l'applicabilité des mécanismes de déclencheurs dans l'implémentation des instances de transition et la gestion du passage d'une transition habilitée à une transition lançable. De même, on souligne l'application des méthodes d'optimisation des contraintes d'intégrité à la mise en œuvre du mécanisme de déclencheurs.

Les "scripts" n'ont pas été implémentés, tout au moins jusqu'au dernier document publié.

CHAPITRE 3

3. LE MODELE CONCEPTUEL

Ce chapitre présente la modélisation des aspects statiques et dynamiques des applications ainsi que leur intégration.

Pour cela, le modèle de données est présenté en 3.1 ; ses langages de définition (3.2) et de manipulation (3.3) sont décrits puisqu'ils sont utilisés pour modéliser et manipuler les aspects dynamiques. Les aspects dynamiques d'une application sont spécifiés à l'aide d'un modèle de procédures de bureau permettant la description de longues séquences de travail. Le couplage du modèle de procédures au modèle de données est étudié dans 3.4.

3.1. Le Modèle de Données

Ce paragraphe présente les caractéristiques principales du "Fact Model" (FM)[1] pour une description plus complète le lecteur pourra se reporter aux documents [DOE86a] [DOE86b].

Les serveurs d'information pour la bureautique (SIB) doivent permettre aux clients le stockage et la manipulation d'informations multimédia en plus des types de données classiques. De plus, ils doivent supporter des procédures de bureau qui peuvent impliquer plusieurs clients et serveurs.

Dans ce contexte, la définition d'un modèle de données pour un SIB est très importante et complexe. Plusieurs problèmes viennent du fait que d'une part, l'information d'un bureau est très hétérogène et ne peut pas être facilement structurée en termes d'enregistrements, et d'autre part, les relations entre les différentes données sont plus complexes que dans les applications traditionnelles. De plus, l'utilisateur final d'un SIB est considéré comme ayant très peu d'expérience dans l'utilisation d'outils informatiques.

Vu ces exigences, le FM est composé de deux niveaux :

- Le **niveau haut** - celui qui nous concerne dans ce chapitre - offre un mécanisme sémantique pour décrire et manipuler le problème de la connaissance.

- Le **niveau bas** offre les moyens de masquer l'hétérogénéité des informations (entités) d'un bureau, permettant ainsi au niveau haut d'avoir une représentation homogène des données. Une entité dans le niveau bas (considérée comme un type de base) est un **type de données abstrait** : elle a une représentation interne et une structure (non perçue dans le niveau haut), un ensemble d'opérations (qui peuvent être appelées par le niveau haut) et un protocole qui contrôle le flux des données entre les deux niveaux.

[1] DOEOIS est un projet ESPRIT : ainsi, les différents travaux ont été développés en anglais. Ceci justifie le fait que tous les noms propres, ainsi que les mots-clés sont en anglais.

Pour la définition du niveau haut nous nous sommes placés, tout d'abord, au niveau des modèles sémantiques vu les exigences du milieu bureautique ; la politique de base étant de proposer un modèle ayant un minimum de concepts à manipuler. Pour cela, nous nous sommes inspirés de DAPLEX, un modèle (sémantique) fonctionnel irréductible. Ainsi, le FM est un dérivé de DAPLEX développé dans le but de combler les déficiences de ce dernier pour modéliser l'environnement bureautique. En particulier il doit permettre la gestion des informations bureautiques ainsi que des processus de gestion de ces informations.

Nous donnons rapidement les principales caratéristiques du Fact Model (FM) :

- Les deux concepts de base du FM sont l'**entité** et le **fait**.

- Une entité (classe d'entités) est l'abstraction d'un ensemble d'instances possibles du monde réel, perçu comme ayant des propriétés communes (p.exp. Personne est une entité). Une instance d'une entité est la représentation symbolique des objets eux mêmes (p.exp. Viriato est une Personne).

- Un fait concerne deux aspects : l'**aspect descriptif** qui le définit comme une association entre deux entités et l'**aspect de manipulation** qui précise comment accéder à ses composants. Les faits sont manipulés par des fonctions : la déclaration d'un fait définit implicitement un ensemble de fonctions permettant d'accéder directement à tous ses composants sans qu'il soit nécessaire d'ajouter des définitions explicites.
  Par rapport à DAPLEX, la notion de fait est nouvelle ; en effet, les deux concepts de base de ce modèle sont : l'entité et la fonction. Une fonction déclare une seule façon logique d'accéder à une donnée : les fonctions inverses doivent être déclarées explicitement ; remarquons que si la fonction contient plus d'un argument, une déclaration algorithmique des fonctions inverses doit être faite.

- Comme dans DAPLEX et dans la plupart des modèles sémantiques, le FM ne fait pas de distinction entre ensembles et types. Un type est considéré comme un ensemble possible d'instances. Ainsi, un seul symbole dénote en même temps : un **type**, une **classe d'entités** et un **ensemble d'instances d'entités**.

- Une distinction est faite entre **entité virtuelle** et **entité littérale**. Une entité littérale est une valeur qui appartient à un type de base, p.ex. la chaîne de caractères "Rablegucherentos" ou l'entier 55. Une entité virtuelle est décrite en termes d'associations avec d'autres entités (virtuelles ou littérales). Par exemple, l'entité Personne est décrite par son nom, par le fait qu'elle est l'enfant d'une autre personne, etc...

- Le FM offre une **hiérarchie de généralisation complète** : elle est construite à partir de l'application des opérateurs ensemblistes.

- Le FM permet la définition de **multi-fonctions**, i.e., fonctions qui retournent des valeurs de plus d'un type. Ceci est une conséquence du fait de pouvoir définir de nouveaux types à partir de l'application des opérateurs ensemblistes (p.ex. l'union).

- Le FM utilise des **règles d'inférence** qui décrivent quels sont les faits dérivés de l'ensemble des faits de base déclarés dans le schéma. Ces règles sont appliquées entre autre à la spécification incomplète d'énoncés d'interrogation.

- Parmi les contraintes inhérentes à un schéma conceptuel FM, nous donnons de l'importance aux suivantes : celles définies comme conséquence du fait que le FM offre une hiérarchie de généralisation complète, et la spécification de contraintes de participation et de cardinalité au niveau des faits.

- Le **langage d'interrogation du FM est ensembliste**. Ceci est analogue au modèle relationnel ou à GEMSTONE, mais différent de la plupart des modèles sémantiques qui sont du type "instance par instance". Cette caractéristique permet d'écrire des énoncés d'interrogation plus concis. De plus, les utilisateurs étant pour la plupart familiarisés avec le modèle relationnel, qui offre lui aussi un langage d'interrogation ensembliste, on peut prévoir que ce type d'orientation leur est préférable.

- Le langage de manipulation du FM utilise la **notation par points** - comme le projet [Gib84] et GEMSTONE [Mai86] - au lieu de la notation par paranthèses utilisée dans DAPLEX. La notation par points nous semble plus naturelle et plus pratique.

3.2. Le Langage de Définition

La présentation faite dans ce paragraphe est basé sur des exemples. Par ailleurs, une définition plus complète du langage de définition du FM est donnée dans les documents [DOE86a] [DOE86b] [Lop87] [Ped88].

Spécification des entités

Les types prédéfinis du modèle sont "**Entity**" et les **types de base**. "Entity" est la classe de toutes les entités virtuelles qu'on peut définir. Les types de base correspondent aux entités littérales ; les instances des types de base sont considérées comme atomiques.

Dans les types de base on trouve les types classiques (entier, chaîne,...) mais également des types necessaires aux applications bureautiques. Ces types permettent la modélisation de données multimédia. Pour cela on trouve : texte, image, audio, graphique, temps, durée, intervalle et oda (type particulier qui correspond à la représentation des documents selon la norme ODA [ECM85]). D'autres types de base, necessaires à la définition des activités et procédures de bureau seront introduits dans les chapitres 4 et 5. Comme il a déjà été dit (cf. section 3.1) un type de base enferme le concept de **type abstrait** au sens des langages de programmation.

Les nouvelles classes / ensembles sont définies à l'aide d'un énoncé de la forme :

NouvelEnsemble : EnsemblePère

où NouvelEnsemble est un sous-ensemble de EnsemblePère et EnsemblePère est une expression ensembliste qui peut être de quatre formes :

- **EnsembleDéfini**.Un ensemble déjà défini. Exemples :

  Personne : Entity

  définit Personne comme un sous-ensemble de toutes les entités virtuelles ;

  Employé : Personne

  définit Employé comme un sous-ensemble des personnes ;

  Age : Integer

  définit Age comme un sous-ensemble du type de base Integer.

- **EnsembleDéfini1 OpérateurEnsemble EnsembleDéfini2**, où "OpérateurEnsemble" est un des opérateurs +, -, *, correspondant respectivement à l'union, différence et intersection d'ensembles. Exemples :

  Chômeur : Personne - Employé

  définit la différence ensembliste entre Personne et Employé : les personnes qui sont des employés ne sont pas des chômeurs ;

  Emprunteur : Employé + Département

  définit l'union ensembliste entre Employé et Département : tout emprunteur est soit un Employé soit un Département.

- **EnsembleNumérique (Intervalle)**, où "EnsembleNumérique" est l'ensemble des entiers, des réels ou un sous-ensemble de ceux-là précédemment défini, et Intervalle est de la forme : BorneInférieure .. BorneSupérieure. Exemple :

  Age : Integer (0 .. 120)

  contraint un âge à être compris entre zéro et cent vingt.

- **(ListeLittérale)**. Une liste de chaînes de caractères exprimée par 'chaîne', ..., 'chaîne'. Exemple :

  Couleur : ('indigo', 'azuré', 'carmin', 'pourpre')

  contraint les couleurs possibles à : indigo, azuré, carmin, pourpre.

Spécification des faits

Un fait est défini en spécifiant les entités participantes ainsi que le rôle qu'elles jouent dans le fait. La forme générale du fait est :

[Rôle1: Entité1, Rôle2 : Entité2, ..., RôleN : EntityN]

Par exemple, le fait entre un Employé et un Département où Employé joue le rôle de Directeur et le Département celui de Dépt-Dirigé est défini par

[Directeur : Employé, Dépt-Dirigé : Département]

Comme il a été dit, la définition d'un fait entraîne la définition des fonctions qui permettent d'accéder à ses composants. Le nom de ces fonctions est le rôle que les entités jouent dans le fait. Ainsi, la déclaration du fait de l'exemple implique la définition des deux fonctions : Directeur (Département) -> Employé[2] et Dépt-Dirigé (Employé) -> Département.

Lorsqu'il n'y a pas d'ambiguité, le rôle et l'entité peuvent coincider. Par exemple, pour modéliser le fait que les employés appartiennent à un certain département nous avons

[Employé : Employé, Département : Département]

Dans ce cas, un énoncé simplifié est autorisé où uniquement l'entité qui participe au fait est spécifiée. Ainsi l'énoncé précédent devient

[Employé, Département]

Les fonctions ici définies sont Employé (Département) -> Employé et Département (Employé) -> Département. On peut considérer aussi que Département est une propriété de l'entité Employé ou vice-versa. Bien sûr, les entités littérales n'ont pas de propriétés, par exemple le fait :

[Personne, Age]

définit la fonction Age (Personne) -> Age mais ne définit pas la fonction Personne (Age) -> Personne.

Dans la définition des faits on peut spécifier des **contraintes de participation aux associations** avec le symbole "*". Exemple :

[*Employé, Département]

signifie que tous les employés doivent être liés à un département, mais que les départements peuvent exister indépendamment des employés.

Dans la définition de faits on peut introduire des **contraintes de cardinalité**. Ces contraintes s'expriment sous forme (CardMin, CardMax). "CardMin" sert à exprimer si la fonction est partielle (auquel cas elle prend la valeur 0) ou totale (valeur 1). "CardMax" sert à exprimer si la fonction est monovaluée (valeur 1) ou multivaluée (la valeur est * représentant un nombre quelconque supérieur à zéro). Exemple :

[Employé(0,*), Département(1,1)]

signifie qu'un employé doit être lié au moins et au plus à un département, alors qu'un département peut-être lié à aucun ou à un nombre quelconque d'employés.

Un type particulier de faits est autorisé dans le FM : les faits constants. Ce fait permet de définir une fonction constante. Exemple

[Personne, Description : 'Une personne est définie par ...']

La seule fonction ici définie est Description (Personne) -> 'Une personne est définie par ...'. Ce type de fait a été introduit parce que dans le FM - comme dans tous les modèles où il y a une distinction rigide entre types et instances - il n'est pas possible d'associer une classe à une entité littérale.

---

[2] "->" indique l'ensemble d'arrivée de la fonction.

Les fonctions définies sur une entité sont héritées à travers la hiérarchie de spécialisation d'un schéma conceptuel. Il convient de définir plus précisément les différents types de généralisation / spécialisation supportés par le FM (ces concepts seront utilisés dans la suite du rapport). Ainsi, la figure 3.1 donne la sémantique de chaque type de spécialisation. Cette sémantique est présentée selon deux points de vue : le point de vue ensembliste qui définit l'ensemble d'instances de la classe spécialisée - et le point de vue du type qui définit les fonctions héritées par la classe spécialisée. La notation utilisée est la suivante :

{A}	Ensembles d'instances de la classe A
F(A)	Fonctions applicables à la classe A
B ← F(A)	B hérite de F(A)

où A et B sont deux classes d'entités virtuelles quelconques.

TYPE		FORME	ASPECT ENSEMBLISTE	ASPECT TYPE
SPECIALISATION SIMPLE		A : Entity B : A	$\{B\} \subset \{A\}$	B ← F(A)
SPECIALISATION COMPOSEE	DIFFERENCE	A : Entity B : A C : A - B	$\{C\} \subset (\{A\} - \{B\})$	C ← F(A)
	INTERSECTION	A : Entity B : A C : A D : B * C	$\{D\} \equiv (\{B\} \cap \{C\})$	D ← F(B) et D ← F(C)
	UNION (Polymorphisme)	A : Entity B : Entity C : A + B	$\{C\} \subset (\{A\} \cup \{B\})$	Si C est un A C ← F(A) Si C est un B C ← F(B)

Figure n°3.1. - Spécialisation / Généralisation dans le FM

3.3. Le Langage de Manipulation

"Fact Model Language" (FML) [Lop87] [Ped88] est le langage de manipulation pour le FM. Sa syntaxe est du type du langage relationnel SQL. Les énoncés de manipulation appartiennent à plusieurs catégories : les énoncés d'**interrogation** ("SELECT") et les énoncés de **mise à jour** qui englobent les insertions ("INSERT") les suppressions ("DELETE") et les modifications ("ASSERT").

L'interrogation, de même que les énoncés de mise à jour, portent exclusivement sur les **faits**. Pour qu'une entité puisse exister dans la base de données et pour qu'elle ait un sens il faut qu'au moins un fait lui soit associé. Donc l'insertion d'une entité sans caractérisation d'au moins un de ses

faits n'a pas de sens. Une sélection porte sur les faits associés à l'entité. Une suppression d'une entité a comme conséquence la suppression des faits qui lui sont reliés. De même une modification vise les faits propres à l'entité.

Quant à la manipulation des faits, comme il a été dit précédemment, une notation par points a été adoptée. Par exemple Employé.Département est la notation pour la fonction Département (Employé).

Je présente ici la syntaxe de FML uniquement à l'aide d'exemples des différents types d'énoncés de manipulation, le langage étant défini plus précisement dans les chapitres 6 et 7. Ces exemples sont basés sur le schéma conceptuel suivant, par souci de simplicité seules les associations binaires sont prises en compte.

Employé, Département, Contrat : Entity ;
Vendeur : Employé ;
Nom, Téléphone : String ;
Num, Salaire, Budget : Integer ;
[Employé (0,1) , Nom (1,1)] ;
Employé (0,1) , Salaire (1,1)]
[Employé (1,1) , Département (1,*)] ;
[Contrat (0,1) , Num (1,1)] ;
[Département (0,1) , Budget (1,1)] ;

Interrogation

La forme générale des énoncés est :

```
SELECT   ...
FROM     ...
WHERE    ...
```

La clause "SELECT" précise l'information qu'on veut obtenir. Vu qu'il faut spécifier les faits par rapport aux entités qu'on veut consulter, le contexte de la requête est exprimé en partie ici. La clause "FROM" est facultative, elle précise aussi le contexte de la requête et sert à déclarer des variables de désignation qui peuvent être utilisées dans les autres clauses ; toute variable doit être déclarée dans cette clause. La clause "WHERE" est aussi facultative et exprime les condition que doivent satisfaire les données sélectionnées.

L'opérateur ensembliste d'inclusion (noté ici "IN") est accepté dans une requête, ainsi que les opérateurs classiques d'agrégation : "count", "sum, "average", "max", "min".

Exemple 3.1 : Lister le nom des employés qui travaillent dans le département "jouets".

```
SELECT   Employé.Nom
WHERE    Employé IN {Département.Employé
                     WHERE Département.Nom = "jouets"}
```

Exemple 3.2 : Lister le nom des vendeurs qui sont liés à des contrats d'un montant supérieur à 5000 . Lister aussi le numéro et le montant de ces contrats.

```
SELECT   v.Nom, c.Num, c.Montant
FROM     Vendeur v.Contrat c
WHERE    c.Montant > 5000
```

La clause "FROM" indique ici que le contexte de la question s'applique aux vendeurs ainsi qu'aux contrats qui leurs sont associés.

Insertion d'une entité virtuelle

Exemple 3.3 : Insérer un nouvel employé en donnant son nom, et en lui attribuant ses numéros de téléphone et un département.

```
INSERT   Employé
WITH     Nom = 'Viriato'
AND      Téléphone = {876543, 765432}
AND      Département =Département WHERE Département.Nom = 'Outils'
```

Cet énoncé spécifie la classe d'une nouvelle entité à insérer - clause "INSERT" - et affecte des valeurs à quelques fonctions de la nouvelle entité : il est obligatoire d'affecter une valeur à au moins une fonction. Dans l'exemple, ces fonctions sont Employé.Nom et Employé.Département. L'entité littérale 'Viriato' (de même pour les numéros de téléphone) est une nouvelle entité qui est insérée dans la base de données comme conséquence de l'exécution de cet énoncé. Par contre, le département qui est attribué à l'employé existe déjà dans la base de données et est qualifié par une clause "WHERE". Toutes les fonctions sur la nouvelle entité ayant comme résultat une entité virtuelle doivent être ainsi spécifiées.

Cet énoncé n'ayant pas de clause "FROM", la déclaration de variables est interdite.

Insertion d'entités existantes dans d'autres classes d'entités

Cette opération n'est possible seulement qu'entre classes d'entités virtuelles qui ont un ancêtre commun dans leurs hiérarchies de généralisation.

Exemple 3.4 : L'employé Viriato devient un vendeur.

```
INSERT   Vendeur e
FROM     Employé e
WHERE    e.Nom = 'Viriato'
```

Insertion de faits

Exemple 3.5 : Nommer l'employé Viriato directeur du département 301.

```
INSERT   [Directeur e, Dépt-Dirigé d]
FROM     Employé e, Département d
WHERE    e.Nom = 'Viriato'
AND      d.Num = 301
```

La sémantique de cet énoncé est : *créer un fait entre un employé et un département où l'employé joue le rôle de Directeur et le département celui de Dépt-Dirigé.* Ou en termes fonctionnels : *donner la valeur e à la fonction Directeur appliquée au Département d* et inversement *donner la valeur d à la fonction Dépt-Dirigé appliquée à l'Employé e.* Ainsi en termes fonctionnels nous aurions écrit

```
INSERT   d.Directeur = e, e.Dépt-Dirigé = d
```

Remarquons que cette mise à jour modifie les deux entités - Employé et Département - par le fait d'établir une relation entre elles.

La clause "FROM" sert comme pour tout énoncé à préciser le contexte de l'énoncé et la clause "WHERE" à qualifier les entités à lier. L'énoncé d'insertion de faits appliqué à des entités virtuelles ne sert qu'à lier des **entités déjà existantes** : une nouvelle entité virtuelle ne peut être insérée que par l'énoncé "INSERT ... WITH".

Bien sûr, selon le type du fait à insérer - définissant des fonctions monovaluées ou multivaluées - et suivant la qualification des entités à mettre à jour donnée dans la clause "WHERE", ce type d'énoncés peut impliquer l'insertion d'un ou plusieurs faits. Pour définir plus précisément la sémantique de cet énoncé nous retiendrons :

- Affecter une valeur à une fonction monovaluée ne peut être fait que si la fonction n'avait pas de valeur auparavant.
- Affecter une valeur à une fonction multivaluée revient à ajouter une valeur à l'ensemble des valeurs de la fonction.

Exemple 3.6 : Insérer un fait entre les entités déjà existantes : le vendeur Viriato et le contrat 301.

```
INSERT   [Vendeur v, Contrat c]
FROM     Vendeur v, Contrat c
WHERE    v.Nom = 'Viriato'
AND      c.Num = 301
```

Cet exemple illustre le même type d'insertion que ci-dessus, néanmoins dans ce cas les rôles coïncident avec les entités participant au fait.

Exemple 3.7 : Relier un numéro de téléphone à l'employé Viriato.

```
INSERT   [Employé e, Téléphone 987654]
FROM     Employé e
WHERE    e.Nom = 'Viriato'
```

Cet exemple illustre une insertion de faits entre une entité virtuelle et une entité littérale.

Remarquons que dans ce cas le fait est établit entre une entité déjà existante et une nouvelle entité littérale - le téléphone 987654 - qui est inséré dans la base de données. En termes fonctionnels la clause "INSERT" devient

```
INSERT    e.Téléphone = 987654
```

Pour affecter plusieurs nouveaux numéros de téléphone, on aurait l'expression

```
INSERT    [Employé e, Téléphone {876543 765432}]
```

Exemple 3.8 : Relier un numéro de téléphone à l'employé Viriato.

```
INSERT    [Employé e, Téléphone t]
FROM      Employé e, Téléphone t          FAUX !
WHERE     e.Nom = 'Viriato'
AND       t = 987654
```

L'expression de cette mise à jour est fausse vu que la clause "WHERE" a comme sémantique - dans n'importe quel énoncé - qualifier les entités à lier : elle ne peut pas être utilisée pour insérer une nouvelle entité, comme c'est le cas du numéro de téléphone 987654. De plus, la déclaration "Téléphone t" n'est pas autorisée ; FM ne permet pas d'accéder à une entité littérale isolée. Bien que la déclaration "Employé : Entity" spécifie implicitement la fonction Employé (), la déclaration "Téléphone : Integer" ne spécifie aucune fonction. De plus, le fait "[Employé, Téléphone]" ne spécifie que la fonction Téléphone (Employé).

Exemple 3.9 : Affecter à l'employé Viriato le même numéro de téléphone qu'à l'employé Dahomey.

```
INSERT    [Employé e, Téléphone t]
FROM      Employé e, Employé e1.Téléphone t
WHERE     e.Nom = 'Viriato'
AND       e1.Nom = 'Dahomey'
```

Cet exemple illustre la création d'un fait entre une entité virtuelle et une entité littérale déjà existante dans la base de données.

Suppression d'entités virtuelles

Exemple 3.10 : Supprimer l'employé Viriato.

```
DELETE    Employé
WHERE     Employé.Nom = 'Viriato'
```

L'exécution de cette mise à jour implique que Viriato n'existe plus ni en tant qu'Employé, ni en tant que Vendeur. Tous les faits qui sont liés à Viriato seront en conséquence supprimés. Ainsi, cet énoncé spécifie dans la clause "DELETE" la classe des entités à supprimer. Ces entités seront qualifiées dans la clause "WHERE" et bien sûr, une clause "FROM" est autorisée.

Suppression de faits

Exemple 3.11 : Détacher du département de jouets les employés qui ont plus de 65 ans.

```
DELETE   [Département d, Employé e]
FROM     Département d, Employé e
WHERE    d.Nom = 'jouets'
AND      e.Age > 65
```

Cet énoncé est le symétrique de celui d'insertion d'entités.

Exemple 3.12 : Supprimer le numéro de téléphone 987654 à l'employé Viriato.

```
DELETE   [Employé e, Téléphone t]
FROM     Employé e, Téléphone t
WHERE    e.Nom = 'Viriato'
AND      t = 987654
```

Remarquons qu'ici l'expression "t = 987654" est correcte : nous sommes en train de qualifier t.

Modification de faits

Exemple 3.13 : Remplacer le numéro de téléphone de l'employé Viriato par 456789.

```
ASSERT   e.Téléphone = 456789
FROM     Employé e
WHERE    e.Nom = 'Viriato'
```

Que la fonction Téléphone (Employé) soit monovaluée ou multivaluée, l'exécution de cet énoncé provoque la suppression du (des) ancien (s) numéro (s) de téléphone de Mr. Viriato. Si la fonction Téléphone (Employé) n'avait pas de valeur avant l'exécution de cet énoncé, alors l'effet de cette opération serait le même que celui d'une insertion de faits.

Exemple 3.14 : Remplacer le numéro de téléphone 987654 de l'employé Viriato par 456789.

```
ASSERT   e.Téléphone = 456789
FROM     Employé e
WHERE    e.Nom = 'Viriato'
AND      e.Téléphone = 987654
```

Cet énoncé est appliqué dans le cas où la fonction Téléphone (Employé) est multivaluée. Il permet de modifier une des valeurs d'une propriété multivaluée.

Les remarques faites pour l'énoncé d'insertion de faits sur la façon dont une nouvelle valeur d'une fonction ayant comme résultat une entité littérale est affectée, sont également valables pour un "ASSERT".

Cet énoncé permet aussi d'affecter des valeurs à différentes fonctions d'une entité (ou groupe

d'entités de la même classe) virtuelle :

```
ASSERT    Employé.Téléphone = ...
          Employé.Adresse = ...
FROM      ...
```

L'"ASSERT" permet aussi de définir la valeur à affecter à la fonction (à modifier) sous forme d'une expression. Les exemples suivants illustrent ceci.

Exemple 3.15 : Augmenter de dix pour cent le salaire de tous les employés

```
ASSERT    Employé.Salaire = Employé.Salaire * 1.1
```

Exemple 3.16 : L'employé Viriato change de département vers le département de jouets.

```
ASSERT    Employé.Département = (Département WHERE
                    Département.Name ='jouets')
WHERE     Employé.Nom = 'Viriato'
```

3.4. Couplage du Modèle de Procédures au Modèle de Données

Deux propositions sur la façon dont les activités et les procédures devraient être représentées au niveau du modèle ont été étudiés : la première a comme but la conception d'un seul modèle permettant l'**intégration** des notions de données et de celles d'activités et de procédures ; la deuxième vise le **couplage** du modèle d'activités et de procédures au modèle de données. Les raisons qui nous ont conduit au choix de la dernière solution sont présentées ci-dessous. Après quoi une description de comment le couplage est fait a lieu.

L'approche "intégration" offre un canevas uniforme pour la modélisation et la manipulation de données et des processus de manipulation de ces données. Ceci se traduit ici par offrir les trois concepts - **Entité, Activité, Procédure** - au même niveau de modélisation. La figure 3.2 présente une possibilité de schéma conceptuel qui illustre bien cette situation : remarquons le parallélisme entre la définition de l'Employé *x* comme une instance d'Employé qui est une sous-classe de ClasseEntités, et la définition de Commission *y* comme une instance de Commission qui est une sous-classe de ClasseActivités (de même pour GénContrat et ClasseProcédures).

Dans cette approche, à chaque concept correspond un **ensemble différent de constructeurs** permettant leur définition et manipulation.

Le principal avantage de cette solution est le fait que nous avons un modèle **uniforme** englobant trois concepts différents. Les principaux inconvénients sont :

- Un grand nombre de concepts à manipuler. Chaque concept du modèle (entité, activité et procédure) a un ensemble de constructeurs qui lui est propre.

- Une faible intégration des trois concepts et par conséquent une rigidité empêchant une combinaison flexible des activités et procédures avec les autres entités des applications. Ce désavantage a des conséquences assez importantes ; en effet, les associations des activités et procédures avec des entités statiques sont essentielles dans la spécification de l'information critique manipulée par les processus. De plus, des opérations particulières aux entités dynamiques (p.ex., charger et décharger une classe d'activités - cf. 2.1.4.2) sont plus intéressantes quand elles sont réalisées de façon sélective ; or cette sélection porte très souvent sur la qualification de l'information manipulée par les processus (p.ex., charger les activités ayant comme entité focale Vendeur).

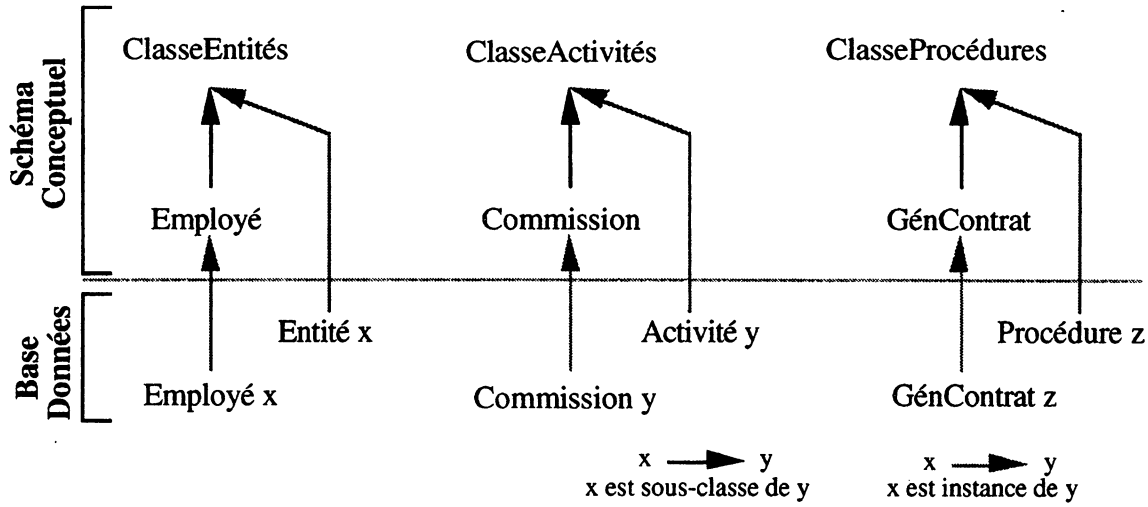


Figure n° 3.2. - Un Schéma Conceptuel pour la Solution "Intégration"

L'approche couplage est caractérisée par le fait que les seuls concepts qui existent au niveau du modèle sont les concepts propres à la modélisation des données : dans le cas du Fact Model les concepts d'**entité** et de **fait**. Le modèle, ainsi que son langage de définition, n'offrent pas de constructeurs spécifiques à la modélisation des activités et des procédures. Ainsi, il faut modéliser au niveau du schéma conceptuel les activités et les procédures en termes d'entités et de faits. Bien que ces entités dynamiques, actives, aient un comportement identique à une entité statique, elles ont de plus une sémantique particulière (p.ex. elles sont exécutables). Cette sémantique est essentiellement donnée par la façon dont l'implémentation traite les entités dynamiques. Dans le cas du Fact Model nous retenons aussi que l'utilisation de types abstraits (niveau bas du FM ; cf. 3.1) permet de définir correctement des propriétés spécifiques aux activités et procédures non modélisables avec les types de base simple. Il reste quand même quelques opérations particulières sur des entités dynamiques qui nécessitent l'introduction de nouveaux constructeurs.

Les principaux avantages de cette solution sont :

- Les activités et les procédures peuvent être manipulées en utilisant les langages de définition et de manipulation standards offerts par le Fact Model. Ceci réduit le nombre de concepts et de constructeurs du langage.

- Les associations des activités et procédures avec les entités statiques sont modélisées comme n'importe quelle autre association. Ceci est opposé à l'approche intégration et facilite la modélisation et la manipulation de l'information critique utilisée par les processus.

- La complexité de l'implémentation est réduite car les données concernant les activités et les procédures sont manipulées en utilisant les mêmes mécanismes que les autres types de données.

Le désavantage essentiel de cette approche est le fait que les concepts d'activité et de procédure sont incorporés au niveau du modèle de façon artificielle : ils restent des notions des applications.

Ayant pris comme politique de prendre un modèle qui ait un minimum de concepts et de symboles à manipuler, il semble que la deuxième solution soit plus appropriée. Cette solution favorise aussi la mise en œuvre d'un prototype (but du projet) vu que l'architecture du système correspondant est simplifiée par le fait que les activités et les procédures sont des entités standards.

Dans le paragraphe 2.1.4 nous avons étudié comment le concept de déclencheur est inclus dans les modèles de données. L'approche couplage a été présentée comme une alternative aux solutions traditionnelles qui modélisent un déclencheur en utilisant des concepts indépendants de ceux permettant la modélisation des données. Cette démarche a été suivie dans le projet HIPAC qui lance le slogan *"Rules are objects too"* [Buc88].

Description du Couplage

Les activités et les procédures sont alors considérées comme des entités : la figure 3.3 présente cette solution. Quatre classes d'entités dynamiques sont définies au même niveau que les classes d'entités statiques, p.ex. Employé, pour modéliser les activités et les procédures :

- **ClasseActivités** : Classe de toutes les classes d'activités. Par exemple classe à laquelle appartient Commission.

- **Activités** : Classe de toutes les instances des classes d'activités. Par exemple, classe à laquelle appartient l'instance Commission *y*.

- **ClasseProcédures** : Classe de toutes les classes de procédures. Par exemple, la classe de procédures GénContrat est une de ses instances.

- **Procédures** : Classe de toutes les instances de classes de procédures. Par exemple, GénContrat n° 301 appartient à cette classe.

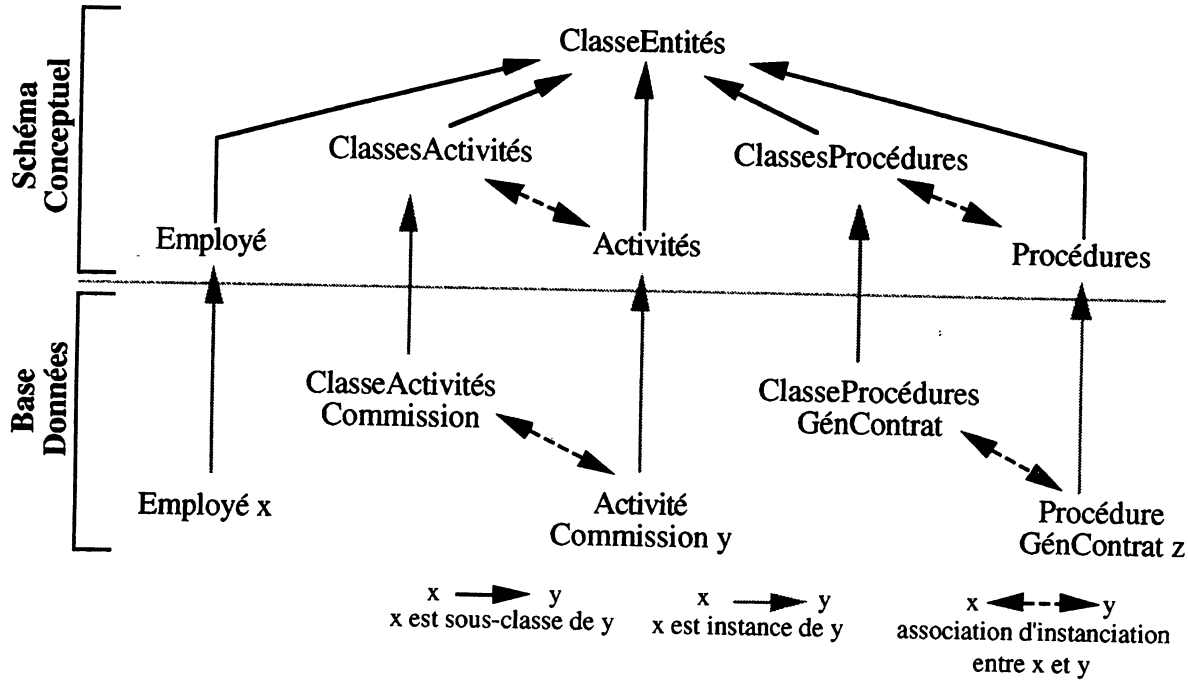


Figure n° 3.3 - Solution Adoptée pour un Schéma Conceptuel FM

On peut remarquer que les concepts de classe d'activité et d'instance d'activité (idem pour les procédures) sont représentés au niveau du schéma conceptuel. Ceci est intéressant vu qu'il permet de faire une distinction entre l'aspect **descriptif** d'une activité (classe ClassesActivités) et son aspect **exécutable** (classe Activités). En effet, les deux concepts (classe et instance d'activité) sont basiquement différents et il faut permettre d'associer des propriétés différentes à ClassesActivités et à Activités. Par exemple, nous disons que ClassesActivités est associée à une propriété nommée ClasseEtat qui indique si l'activité est chargée ou non (cf. 2.1.4.2) ; par ailleurs, il peut être défini pour Activités une propriété nommée Etat ayant comme valeurs 'habilité' (l'instance de l'activité est habilitée à être exécutée), 'lancé' (le corps d'exécution de l'activité s'exécute), etc...

L'**association d'instanciation** ("instance-de/classe-de") permet de faire un lien entre les instances de ClassesActivités et les instances d'Activités.

De façon informelle et avec le seul souci de montrer que cette solution offre un canevas convenable pour la modélisation des activités et des procédures de bureau, la figure 3.4 illustre de façon graphique les classes d'entités dynamiques prédéfinies pour tous les schémas conceptuels des applications. Cette figure décrit aussi la classe d'activités Commission - insérée dans la base à partir de l'énoncé standard d'insertion d'entités (cf. 3.2) - et son instance Commission y - créée automatiquement par le système lors d'une mise à jour qui a validé la précondition. Pour cela, la description de l'activité Commission est présentée dans la page suivante. Ceci permet aussi d'avoir une idée des différents composants d'une activité : ils seront étudiés en détail dans le chapitre suivant.

ACTIVITE COMMISSION

Description : *'Cette activité augmente la commission d'un vendeur s'il établit un contrat d'un montant supérieur à 5000 FF'*
Description de l'activité.

Nom : *'Commission'* Nom de l'activité.

EntitéFocale : *Vendeur* Centre de l'information manipulée par l'activité.

Précondition : *Le vendeur établit un contrat d'un montant supérieur à 5000*
Condition sous laquelle l'activité est habilitée à être exécutée.

CorpsExéc : *'Commission.exec'* Nom du fichier contenant les opérations nécessaires pour augmenter la commission. Partie exécutable de l'activité.

EtatClasse : *'chargé'* Indique si la précondition est évaluable (des activités peuvent être habilitées à être exécutées) ou non évaluable (pas d'habilitation d'activités).

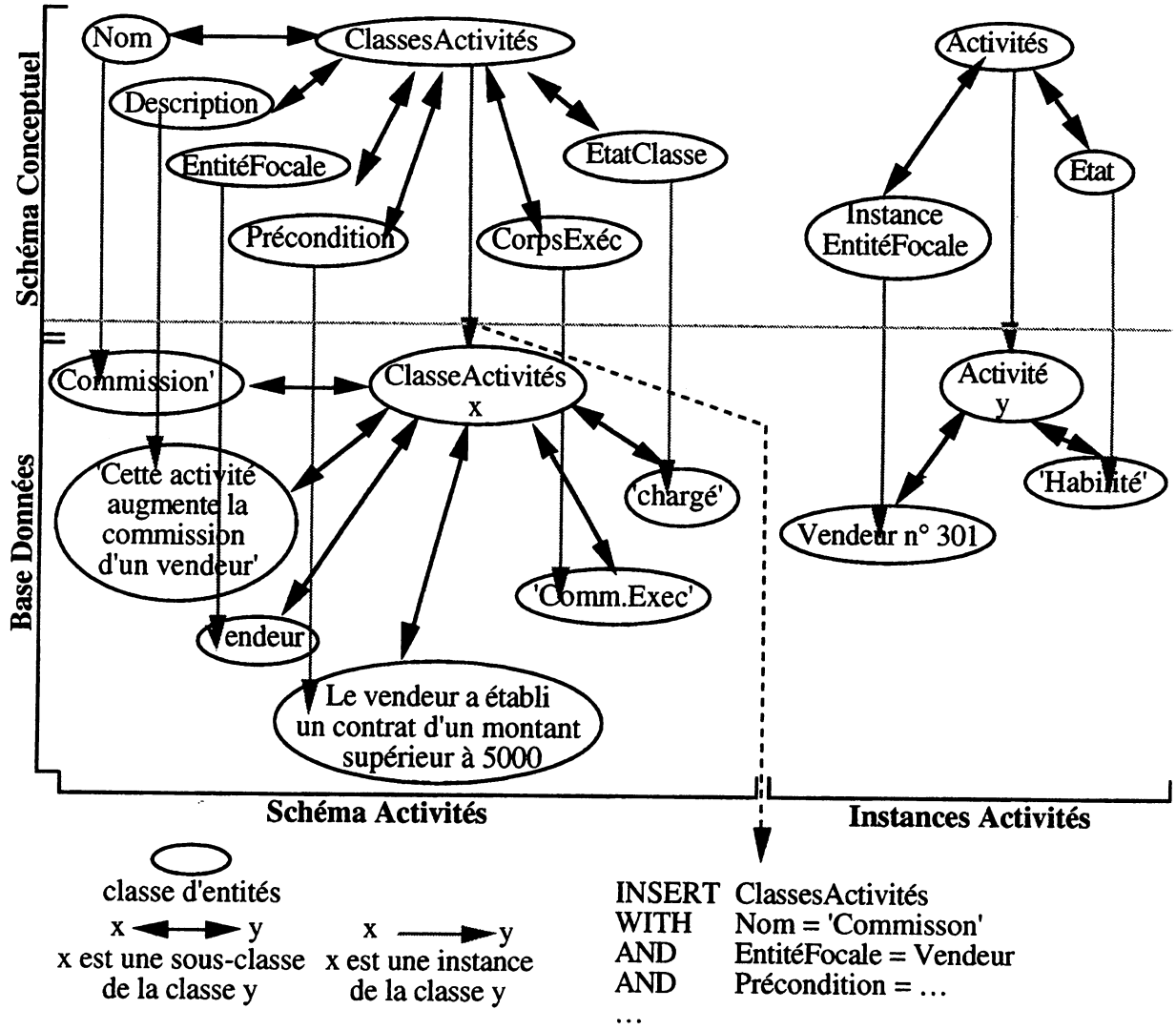


Figure n° 3.4 - Modélisation de l'Activité Commission

CHAPITRE 4

4. MODELISATION DES ACTIVITES

Le concept d'activité offre le niveau de base d'un support permettant l'automatisation de tâches bureautiques. Ce concept et sa modélisation sont introduits dans ce chapitre. D'une part, les composants d'une activité sont analysés en détail. D'autre part, les outils nécessaires à la mise en œuvre d'un système capable d'exécuter, de suivre et de contrôler de façon persistante l'exécution d'une activité sont présentés.

Comme il a été vu dans le chapitre précédent, une activité et ses composants sont présentés et définis en termes du Fact Model. Ces définitions sont standards à toutes les applications : elles sont générées automatiquement par le système à la création d'une base de données.

Ainsi, nous commençons ce chapitre par la définition en FM de ce qui est une activité. Tout d'abord les classes ClassesActivités - nommée "**Activity-Class**" dans FM - et Activités - "**Activity**" dans FM - et leur association d'instanciation ont été définies par les énoncés :

Activity-Class : Entity
Activity : Entity
[Activity-Class (1,1), Activity (0,*)]

Deux des composants d'une activité ne posent aucun problème ni au niveau sémantique ni au niveau modélisation : il s'agit du nom de la classe d'activités et de sa description (i.e. un texte qui décrit son objectif). Ils sont définis comme suit :

[Activity-Class (1,1), Name (1,1): String]
[Activity-Class (1,*), Description (0,1) : Text]

Les autres composants seront expliqués dans les paragraphes suivants. Comme on le verra, des types de base (cf. 3.1) ont du être introduits pour permettre une modélisation correcte et simple.

4.1. Entité Focale

Toute classe d'activités est associée à une **classe d'entités** - la classe de l'entité focale - qui est considérée comme le centre de l'information critique de la classe d'activités, i.e. la classe d'entité focale doit être associée dans le schéma conceptuel aux classes d'entités manipulées par l'activité.

Une instance d'activité est associée à une **instance de l'entité focale**, p.ex., vu que la classe d'activités "Commission" a comme classe d'entité focale Vendeur, alors une instance de Commission doit être associée à une instance de Vendeur (par exemple le vendeur n° 99). Nous

disons aussi q'une instance d'activité s'exécute pour une instance de l'entité focale.

L'entité focale doit être une des **entités virtuelles** concernées par la **précondition**, p.ex., pour l'activité "Commission" Vendeur et Contrat sont les deux entités focales possibles. La validation de la précondition doit déterminer les instances d'activités à créer ainsi que leur entités focales. Dans l'exemple de Commission la précondition est "Le vendeur a établi un contrat d'un montant supérieur à 5000". Ceci implique qu'une instance de l'activité "Commission" est créée pour **chaque** vendeur qui a été lié a un contrat et qui satisfait la condition (le contrat est supérieur à 5000).

La notion d'entité focale est très importante du point de vue conception : elle **induit une façon de concevoir** les applications. Pour illustrer ceci, un exemple d'activité et de la façon de l'instancier selon plusieurs situations va être présenté.

Supposons le **schéma conceptuel** suivant :

Budget : 50000 .. 200000
Salaire : 5000 .. 20000
Type : ('A', 'B', 'C')
Département : Entity
Employé : Entity
[Département (0,*), Employé (1,*)]
[Département (0,*), Budget (1,1)]
[Département (0,*), Type (1,1)]
[Employé (0,*), Salaire (1,*)]

Soit l'activité **A** ayant comme **précondition** : "le budget d'un département est devenu plus petit que dix fois le salaire d'un des employés qui lui est affecté". Son **corps d'exécution** exprime le fait d'augmenter le budget du département de façon qu'aucun employé ne gagne plus que 10% du budget.

Supposons aussi qu'une **mise à jour** qui diminue de 15% le budget des départements de type 'B', est exécutée impliquant ainsi que le budget du département **d1** devient plus petit que dix fois le salaire de ses employés **e1** et **e2**, de même pour le département **d2** et l'employé **e1**.

Cette situation entraîne la **validation de la précondition** : regardons alors comment l'activité A va être instanciée. Trois situations sont prises en compte : l'entité focale est le département, l'entité focale est l'employé, l'activité n'a pas d'entité focale. Cette dernière situation n'est pas admise dans le modèle, néanmoins elle est présentée pour mettre en évidence l'importance de l'entité focale.

Situation 1 - L'entité focale est **Département**

Pour chaque instance de Département qui vérifie la précondition, une instance de A est créée ; nous aurons donc deux instances d'activités :

Activité **A1** ayant comme entité focale **d1**

Activité **A2** ayant comme entité focale **d2**

On peut imaginer que la façon appropriée d'implémenter le corps d'exécution de l'activité A soit d'augmenter le budget de chaque département en fonction des salaires de ses employés qui dépassent de dix pour cent ce budget. Ainsi,

l'activité **A1** augmente le budget de **d1** en fonction des salaires de **e1** et **e2**, et l'activité **A2** augmente le budget de **d2** en fonction du salaire de **e1**.

Situation 2 - L'entité focale est **Employé**

Pour chaque instance d'Employé qui vérifie la précondition, une instance de A est créée ; nous aurons donc deux instances d'activités :

Activité **A3** ayant comme entité focale **e1**

Activité **A4** ayant comme entité focale **e2**

Dans ce cas le corps d'exécution d'une activité est implémenté différemment : à partir de l'analyse du salaire d'un employé, augmenter les budgets des départements où il est affecté et qui ne sont pas dix fois plus grands que son salaire. Ainsi,

l'activité **A3** augmente en fonction du salaire de **e1** le budget de **d1** et de **d2**, et l'activité **A4** augmente en fonction du salaire de **e2** le budget de **d1**.

Remarquons que cette dernière activité peut ne plus avoir de sens vu que le budget de d1 a déjà été augmenté par l'activité A3.

Situation 3 - **Pas** d'entité focale

En supposant qu'une activité n'a pas d'entité focale, plusieurs possibilités d'instanciation existent, comme par exemple :

(i) Créer une instance par mise à jour (orientation ensembliste ; cf. 2.1.3) ; dans notre exemple une seule instanciation de l'activité A aurait lieu.

(ii) Créer une instance par fait (instance de fait) modifié (orientation instance par instance) ; dans l'exemple les deux faits modifiés sont d1 et son budget, et d2 et son budget ; deux activités seraient instanciées en conséquence, et on serait dans une situation similaire à la situation 1.

(iii) La troisième possibilité prend en compte la partie du schéma conceptuel désignée dans la précondition, p.ex., pour l'activité A elle est [Département, Budget], [Département, Employé], [Employé, Salaire] ; l'instanciation serait faite pour chaque instance de cette partie du schéma qui vérifie la précondition.

Cette dernière possibilité est détaillée ici : trois instances d'activités seraient créées

Activité **A5** prenant en compte **d1** et son **budget** et **e1** et son **salaire**

Activité **A6** prenant en compte **d1** et son **budget** et **e2** et son **salaire**

Activité **A7** prenant en compte **d2** et son **budget** et **e1** et son **salaire**

Ici, l'implémentation du corps d'exécution impliquerait que chaque instance Ai augmente le budget d'un département en fonction du salaire d'un de ses employés qui dépasse dix pour cent du budget. Ainsi,

l'activité **A5** augmenterait le budget de **d1** en fonction du salaire de **e1**,
l'activité **A6** augmenterait le budget de **d1** en fonction du salaire de **e2**, et
l'activité **A7** augmenterait le budget de **d2** en fonction du salaire de **e1**.

L'exécution de A5 peut provoquer la création d'une nouvelle instance de l'activité A - A8 - et ainsi de suite jusqu'à la stabilité.

En analysant cet exemple quelques remarques méritent d'être faites :

- La façon dont une activité est instanciée implique l'application d'un raisonnement particulier dans l'implémentation du corps d'exécution de l'activité, et par conséquent, dans la conception des applications.

- Associer une entité focale à une activité est une solution qui aide l'utilisateur à concevoir ses applications, car la création (instanciation) d'une activité est faite en fonction de l'état ou des changements d'une entité (l'entité focale) : on s'intéresse à **l'évolution d'une entité**. Ceci est sémantiquement plus riche que de s'intéresser, de façon absolue, aux mises à jour à réaliser.

- Pour cet exemple, prendre comme entité focale Département pour l'activité A semble la solution la plus naturelle. De plus, cette solution n'implique pas le déclenchement d'activités en cascade, ce qui peut être moins évident à concevoir.

- Dans (iii) de la situation 3, la base de données est vue de façon plate : le fait qu'un département ait plusieurs employés est perdu. De même la richesse sémantique du modèle est mise en question. Cette remarque peut aussi être vraie pour (ii) si le fait modifié est multivalué; dans ce cas la solution (ii) ne coïnciderait pas avec la solution 1.

De façon plus générale trois arguments permettent de valoriser l'existence d'une entité focale par activité. Le premier est le fait que le **comportement d'une classe d'entités** peut être défini par les activités (et plus tard par les procédures) qui lui sont appliquées. Le deuxième est le fait que l'entité focale permet une bonne **liaison entre les aspects statiques et dynamiques** d'une application, ceci au niveau conceptuel (une classe d'activités s'applique à une classe d'entités) et au niveau base de données (une activité s'applique à une entité). Le troisième concerne les études réalisées dans le domaine des méthodologies de conception : ils sont à l'origine de la notion de l'entité focale comme un moyen pour modéliser le flux de données. Le fait d'offrir explicitement cette notion au niveau opérationnel **renforce les liens entre les méthodologies de conception et les outils opérationnels**. De plus, les avantages de la notion d'entité focale du point de vue opérationnel sont aussi importants : à ce niveau, elle s'assimile à la philosophie très fonctionnelle des approches orienté objet qui facilitent la mise en œuvre des outils.

Définitions en Termes du Modèle de Données

Nous venons de voir que toute classe d'activités est associée à une classe d'entités focales - e. g., la classe d'activités Commission est associée à la classe d'entités focales Vendeur - et que toute instance d'activité est associée à une instance de l'entité focale - p.ex., une instance de l'activité Commission doit être associée à une instance de Vendeur.

Pour représenter ceci au niveau du schéma conceptuel il faut que (voir figure 4.1) Activity-Class soit associée à une classe représentant toutes les classes d'entités focales - nous appelons cette classe "**FE-Class**" - et que Activity soit associée à une classe représentant toutes les instances d'entités focales - cette classe est nommée "**FE**". Nous avons ainsi la définition des faits suivants :

[Activity-Class (1,*), FE-Class (1,1)]
[Activiy (1,*), FE (1,1)]

Le problème qui se pose est comment définir FE-Class et FE. Au premier abord ces concepts appartiennent à un méta-niveau : or, le Fact Model n'admet pas ce niveau.

Dans l'association entre*Vendeur* et la classe d'activités Commission, *Vendeur* ne peut pas être la classe d'entités virtuelles représentant tous les vendeurs (celle-ci est définie au niveau du schéma conceptuel).*Vendeur* est uniquement la **désignation** de la classe d'entités Vendeur. Ainsi FE-Class est définie comme la classe de toutes les désignations de classes d'entités virtuelles (à l'exception de Activity-Class et de Activity).

En termes du modèle de données, FE-Class est définie comme un type de base nommé **VClass-designation** ("V" pour "Virtual"), en conséquence nous avons :

FE-Class : VClass-Designation

VClassDesignation est en fait un type abstrait (il appartient au niveau bas du FM ; cf. 3.1) simple. Ainsi, il a une structure interne : la structure d'une instance de FE-Class est une référence à la classe d'entités qu'elle désigne, plus une chaîne de caractères représentant son nom. La seule opération permise sur ce type est la nomination de la classe de données qu'elle désigne (voir exemple 4.1 plus loin).

Voyons maintenant ce qui se passe avec FE. Une instance de FE est de plus une instance d'une classe quelconque d'entités virtuelles (à l'exception de Activity-Class et de Activity), e. g., la classe Vendeur. Ainsi, du point de vue ensembliste, FE est un sous-ensemble de l'union de toutes les instances des classes d'entités virtuelles. En ce qui concerne le type, FE hérite des fonctions d'une classe d'entités virtuelles particulière. Dans l'exemple de l'activité Commission, FE hérite des fonctions applicables à un Vendeur.

Ceci est modélisé dans le Fact Model en définissant FE comme l'union (cf. figure 3.1) de toutes les classes d'entités virtuelles (à l'exception de Activity-Class et de Activity) définies dans le schéma conceptuel d'une application.

La figure 4.1 illustre le schéma conceptuel d'une application comme il vient d'être défini. Ici, FE est du type

FE : Vendeur + Contrat + Département

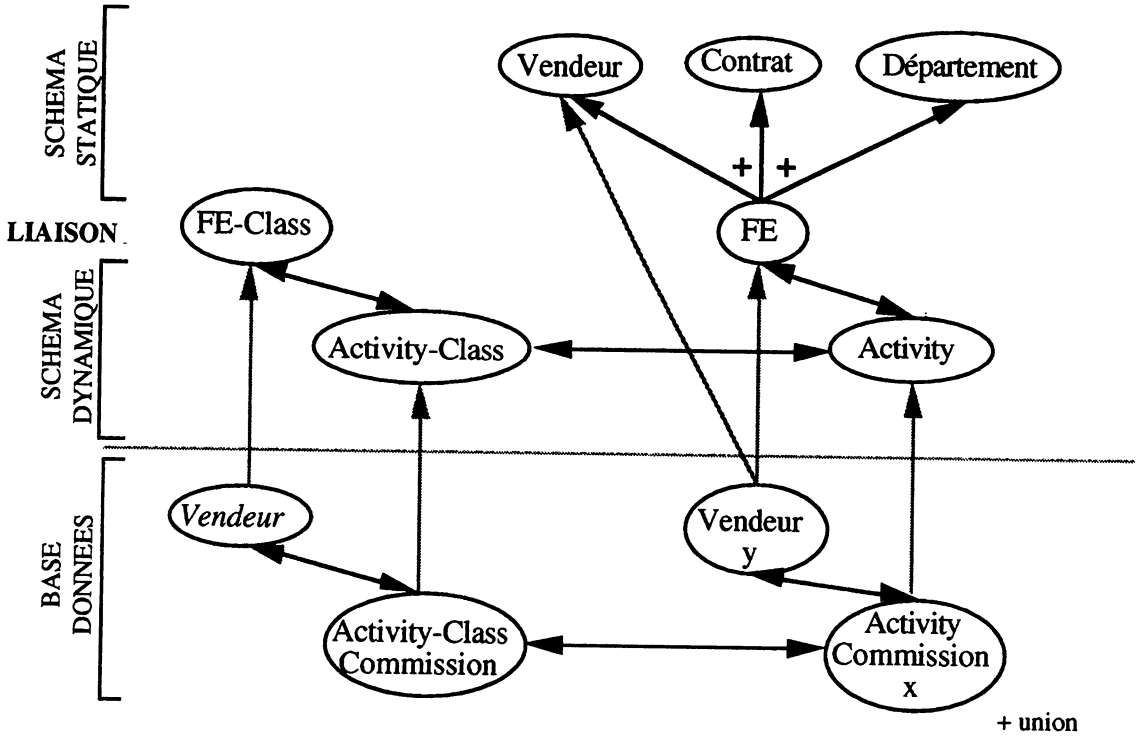


Figure n° 4.1 - Modélisation de FE-Class et de FE

Cette figure illustre aussi le fait que l'entité focale permet une liaison entre les aspects statiques et dynamiques d'une application. Bien que ceci n'ait été mis en évidence qu'au niveau du schéma, la même chose aurait pu être faite au niveau de la base de données : le dessin résultant serait néanmoins lourd.

Remarquons que chaque fois qu'une classe d'entités virtuelles est définie dans le schéma conceptuel d'une application, à la définition de FE on doit ajouter l'union de cette nouvelle classe. Ceci ne pose pas de problèmes car la définition du schéma dynamique des applications est, comme il a déjà été dit, faite automatiquement par le système.

Manipulations sur l'Entité Focale

FE-Class et FE sont des entités qui ont - entre autre - un comportement identique à celui de n'importe quelle entité : quelques exemples seront donnés ici.

Exemple 4.1 : Insérer la classe d'activité Commission avec sa classe d'entités focales.

```
INSERT    Activity-Class
WITH      Name = 'Commission'
AND       FE-Class = Vendeur[1]
```

[1]Dans l'expression "FE-Class = Vendeur", une référence implicite à l'opération nomination du type abstrait VClass-designation est faite. En effet, cette opération est la seule permise sur ce type, ainsi elle n'a pas besoin d'être spécifiée expicitement.

Une instance d'activité est toujours créée automatiquement par le système au moment où la précondition de la classe d'activités correspondante est validée. Ainsi, des énoncés du type INSERT Activity WITH ... ne sont pas permis.

Exemple 4.2 : Donner la classe d'entité focale de l'activité Commission.

```
SELECT   FE-Class
FROM     Activity-Class
WHERE    Name = 'Commission'
```

Exemple 4.3 : Lister le nom et le salaire des vendeurs qui sont des entités focales des instances de l'activité Commission.

```
SELECT   a.FE.Nom, a.FE.Salaire
FROM     Activity a. Activity-Class c
WHERE    c.Name = 'Commission'
```

Ces énoncés offrent une première idée sur l'importance de modéliser les associations des activités / procédures avec les entités statiques de la même façon que les associations des entités statiques entre elles. Ceci facilite la manipulation de l'information critique utilisée par les processus et permet de regrouper les activités / procédures selon des critères qui qualifient cette information.

4.2. Précondition

La précondition d'une classe d'activités spécifie les conditions selon lesquelles il est approprié de créer une instance de cette activité et de l'habiliter à être exécutée. Elle est exprimée en termes des entités représentées dans la base de données du SIB : elle doit refléter les états et changements de ces entités. Sémantiquement, les préconditions sont divisées en deux catégories :

- Préconditions de **changement**. Ces préconditions permettent la modélisation du type de situations suivant : si un évènement a lieu dans un certain contexte, alors il faut agir en conséquence ; peu importe ce qui se passe entre le moment où l'évènement arrive et le moment où on agit. Un exemple de ce type de précondition, appliqué au système de réservation d'une compagnie d'avions, serait : *si une personne s'inscrit pour un vol, alors il faut faire certaines démarches, e. g., son billet,* ***même si*** *ce vol est annulé suite à une grève*. Plus précisément, la sémantique de ce type de préconditions est : si tel évènement a lieu dans un certain contexte alors (i) la précondition est validée, (ii) une instance de l'activité est créée et habilitée à être exécutée et (iii) cette instance d'activité reste habilitée jusqu'à ce que démarre son exécution. Ainsi, une précondition de changement, qui a été validée, le reste jusqu'au début de l'exécution de l'activité.

- Préconditions d'**état**. Les situations modélisées avec ces préconditions sont : si la base de données est dans un certain état alors il faut agir, mais il ne faut plus le faire si entretemps la base de données n'est plus dans l'état souhaité. Reprenant l'application de réservations, un

exemple de ce type de préconditions serait : *si un vol est complet et si quelqu'un désire une réservation, alors il faut l'inscrire dans une file d'attente, mais si entretemps - jusqu'au moment où on va réellement l'inscrire - quelqu'un a annulé sa réservation pour le même vol alors* ***il ne faut plus*** *le mettre en file d'attente : une réservation peut lui être attribuée de suite.* Plus précisément, la sémantique de ces préconditions est comme suit : si la base de données est dans un tel état alors (i) la précondition est validée, (ii) une instance de l'activité est créée et habilitée à être exécutée, (iii) cette instance d'activité est supprimée si jamais l'entité n'est plus dans l'état souhaité. Ainsi, une précondition d'état validée peut devenir fausse avant que l'activité s'exécute.

... Mais les choses ne sont jamais aussi claires dans la vie réelle et le monde n'a jamais pû être divisé en deux. En effet, décider de modéliser une situation avec des activités dont les préconditions sont du type "état" ou "changement" et le mode de démarrage est automatique ou manuel dépend beaucoup de la façon de concevoir de l'utilisateur. Néanmoins, nous pensons qu'il est utile et pertinent de proposer ces différentes possibilités, dans la suite de ce chapitre plusieurs exemples illustrent l'utilité des différentes combinaisons entre le type de la précondition et le mode de lancement d'une activité.

4.2.1. Préconditions de Changement

D'après la sémantique donnée ci-dessus, pour spécifier une précondition de ce type, il faut :

- D'une part, identifier **l'évènement qui doit avoir lieu** pour que la précondition soit validée. L'évènement doit être une action base de données, i.e., les différentes mises à jour offertes par le FML. L'évènemet est alors défini par :
  - (i) Un type de mise à jour, ceci définit sept sous-types (cf. 3.3) de préconditions : insertion d'une entité virtuelle, insertion d'entités virtuelles existantes dans d'autres classes, insertion de faits, suppression d'entités virtuelles, suppression d'entités d'une classe spécialisée, suppression de faits et modification de faits.
  - (ii) Les classes d'entités concernées par la mise à jour, i.e., les classes d'entités dont les instances seront mises à jour.

- D'autre part, identifier **le contexte dans lequel l'action doit avoir lieu** pour que la précondition soit validée. Ce contexte est défini par des conditions sur les entités à mettre à jour et éventuellement sur l'état général de la base de données.

On peut constater que la forme générale des énoncés FML - i. e., "<Type Opération> ... FROM ... WHERE ...", exception faite pour l'insertion d'une entité dont la forme est "INSERT ... WITH ..." - s'adapte très bien à ce qu'il est necessaire de spécifier pour définir une précondition. L'exemple suivant illustre ceci.

Exemple 4.4 : Cet exemple est une des écritures possibles pour la précondition de l'activité Commission.

```
TEST   INSERT   [Vendeur v, Contrat c]
       FROM     Vendeur v, Contrat c
       WHERE    c.montant > 5000
```

Ici, l'action (évènement) identifiée est : *des vendeurs sont associés à des contrats*. Dans l'exemple, ceci est identifié essentiellement par la clause "TEST INSERT". Le contexte dans lequel cette action est faite indique que : *les contrats à mettre à jour doivent être d'un montant supérieur à 5000*. Ceci est spécifié essentiellement dans la clause "WHERE". La clause "FROM" sert, comme pour une requête, à mieux préciser la partie de la base de données concernée par la précondition ainsi qu'à déclarer des variables.

Remarquons la similarité entre l'expression de cette précondition et l'expression de l'énoncé de manipulation donné dans l'exemple 3.6 (insérer un fait entre le vendeur Viriato et le contrat numéro 301). Au niveau syntaxique la différence essentielle est que l'expression d'une précondition commence par le mot-clé "TEST". En conséquence, et vu que l'utilisateur est familarisé avec le FML, il a été décidé que le langage de définition des préconditions est un sur-ensemble du FML. Ceci implique, de plus, une spécification des préconditions à un haut niveau sémantique.

La précondition de l'activité Commission n'est pas encore spécifiée de façon complète : dans l'expression d'une précondition il faut spécifier quelle entité joue le rôle d'entité focale de l'activité correspondante. En effet, la validation d'une précondition doit déterminer les activités à créer et leurs entités focales. Nous disons aussi que la précondition est validée pour une entité focale. L'entité focale est spécifiée par :

(i) la variable de nom réservé "FE" ou (ii) la présence du caractère "&"

Exemple 4.5 :

```
TEST   INSERT   [Vendeur FE, Contrat c]
       FROM     Vendeur FE, Contrat c
       WHERE    c.montant > 5000
```

Exemple 4.6 :

```
TEST   INSERT   [Vendeur v&, Contrat c]
       FROM     Vendeur v&, Contrat c
       WHERE    c.montant > 5000
```

La précondition est maintenant définie complètement. Sa sémantique est : *créer une activité pour **chaque vendeur** à qui on associe des contrats (au moins un) d'un montant supérieur à 5000.*

L'entité focale doit être une des entités concernées par la précondition. Or, dans une précondition, sont spécifiées les entités suivantes :

(i) les entités à mettre à jour ;
(ii) des entités associées aux entités à mettre à jour et pour lesquelles sont posées des conditions ;
(iii) éventuellement des entités qui n'ont aucun rapport avec les entités à mettre à jour et pour lesquelles sont posées des conditions. Un exemple serait d'ajouter à la précondition de l'activité Commission le fait qu'il faut qu'on soit dans une certaine période de l'année pour que la précondition soit validée.

Il semble naturel que l'entité focale soit une entité virtuelle appartenant à (i) ou (ii). Nous pensons même que dans la plupart des cas l'entité focale est une des entités à mettre à jour.

Syntaxiquement, le contexte d'une précondition étant - comme pour une mise à jour - spécifié dans les clauses "TEST <Opération>" et "FROM", l'entité focale peut être :

(i) une des entités de la clause "TEST <Opération>" ;
(ii) une des entités virtuelles de la clause "FROM" qui est associée (dans cette clause) à une des entités de (i).

Pour l'énoncé d'insertion d'une nouvelle entité, l'entité focale peut être soit l'entité à insérer, soit une des entités virtuelles qui est une propriété directe de la nouvelle entité et qui est spécifiée dans la précondition : ceci parce que cet énoncé n'a pas de clause "FROM".

Cette règle syntaxique est restrictive vu que, même si ce sont les clauses "TEST <Opération>" et "FROM" qui spécifient le contexte général d'une précondition, il se peut très bien que certaines associations d'entités virtuelles avec les entités à mettre à jour soient spécifiées dans la clause "WHERE". De plus la clause "FROM" n'est pas obligatoire dans le FML.

Il faut néanmoins constater que cette restriction est uniquement syntaxique, i. e., on oblige l'utilisateur à spécifier le contexte qui pour lui est significatif dans les deux premières clauses de la précondition. La raison principale qui nous a mené à poser cette restriction est fonction de la façon de spécifier le contexte d'activation et est donnée dans le paragraphe suivant (cf. 4.3) : ici nous ajoutons uniquement qu'il s'agit d'un souci d'éviter l'écriture de préconditions absurdement complexes.

Ci-dessous, une série d'exemples de différents types de préconditions est présentée dans le but de préciser leur sémantique et leur syntaxe quand celle-ci est différente du FML.

Préconditions d'Insertion de Faits

Les deux exemples suivants illustrent les situations 1 et 2 du paragraphe précédent (cf. 4.1).

Exemple 4.7 : Créer une instance d'activité pour chaque département qui se voit attribuer un budget qui est plus petit que dix fois le salaire d'un de ses employés.

TEST	INSERT	[Département FE, Budget b]
	FROM	Département FE.Employé e, Département.Budget b
	WHERE	b < 10 * e.Salaire[2]

Exemple 4.8 : Créer une instance d'activité pour chaque employé appartenant à un département qui se voit attribuer un nouveau budget inférieur à dix fois son salaire.

[2]La variable b a été introduite dans le but de comparer le nouveau budget d'un département au salaire de ses employés. Une entité littérale ne pouvant pas être accessible de façon isolée (i.e. la fonction Budget () n'existe pas ; cf. exemple 3.8), la déclaration de b a dû être faite par rapport à la fonction qui la definit, i.e., Budget (Département).

TEST	INSERT	[Département d, Budget b]
	FROM	Département d.Employé FE, Département.Budget b
	WHERE	b < 10 * FE.Salaire

Le corps d'exécution de l'activité A du paragraphe 4.1 a comme but d'augmenter le budget des départements. Dans ce cas, il a été constaté que prendre Département comme entité focale (exemple 4.7) est plus correct. Mais, si on suppose une autre activité - Activité B - dont l'objectif est d'appliquer un traitement spécial aux employés qui gagnent plus que dix pour cent du budget de leur département, alors il est plus correct de prendre Employé comme entité focale (exemple 4.8). Ainsi, ces deux préconditions peuvent avoir un sens ; remarquons qu'elles impliquent des instanciations différentes des activités correspondantes :

- Pour A ayant comme précondition celle de l'exemple 4.7, une instance peut être créée par fait modifié (cf. figure 4.2 (a)).
- Pour B ayant comme précondition celle de l'exemple 4.8, plusieurs instances peuvent être créées par fait modifié (cf. figure 4.2 (b)).

D'autres exemples peuvent illustrer un troisième cas : une instance peut être créée par plusieurs faits modifiés (cf. figure 4.2 (c)).

Nous venons donc de voir de façon plus précise et en utilisant la syntaxe proposée comment les activités sont instanciées ; la figure 4.2 présente les trois cas d'instanciation possibles.

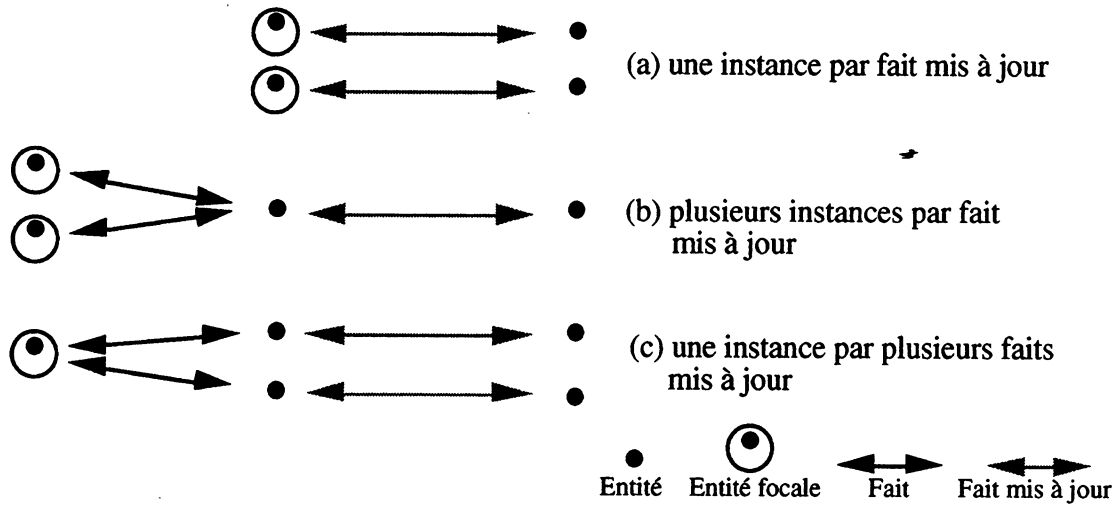


Figure n° 4.2 - Instanciation des Activités

Exemple 4.9 : Créer une activité pour chaque employé qui se voit attribuer un salaire qui est 1,2 fois plus grand que la moyenne des salaires de son département.

TEST	INSERT	[Employé FE, Salaire s]
	FROM	Employé FE.Département d, Employé.Salaire s
	WHERE	Avg (d.Employé.Salaire) *1.2 < s

Comme pour un énoncé FML la clause "WHERE" sert à qualifier les entités qui vont être liées. Ainsi, la moyenne calculée est celle d'avant l'insertion des faits. En effet, l'expression "d.Employé.Salaire" accède aux salaires des employés du département de l'employé qui va recevoir un nouveau salaire.

Remarquons que le seul cas où il y aurait pû avoir une ambiguïté est celui pour lequel la condition porte sur l'ensemble des valeurs de la fonction à modifier.

Précondition de Modification de Faits

Exemple 4.10 : Créer une activité pour chaque employé dont le salaire diminue.

```
TEST   ASSERT   FE.Salaire = s
       FROM     Employé FE, Employé.Salaire s
       WHERE    FE.Salaire > s
```

Comme pour un énoncé FML l'expression "FE.Salaire" de la clause "WHERE" fait référence à l'ancien salaire de l'employé. Par contre, s fait référence à la nouvelle valeur. Ainsi, cette condition permet de comparer l'ancienne et la nouvelle valeur du fait modifié.

Préconditions d'Insertion d'une Entité

Exemple 4.11 : Si un nouvel employé est inséré, alors créer une instance de l'activté correspondante dont l'entité focale est cet employé.

```
TEST   INSERT   Employé &
```

Cette précondition ne spécifie aucune condition sur l'état de la base de données. Remarquons que dans FML l'énoncé de mise à jour "INSERT Employé" est interdit : en effet, on ne peut pas insérer une entité virtuelle sans spécifier au moins une valeur d'une de ses associations.

Exemple 4.12 : Si un nouvel employé est inséré et qu'on lui attribue les départements de type "A" alors créer une instance de l'activité correspondante pour chaque département de type "A".

```
TEST   INSERT   Employé
       WITH     Département & = {Département WHERE
                Département.Type = 'A'}
```

Exemple 4.13 : Si un nouvel employé est inséré avec un salaire supérieur à 10000 et un département quelconque, alors créer une instance de l'activité correspondante ayant comme entité focale l'employé.

```
TEST   INSERT   Employé &
       WITH     Salaire > 10000
       AND      Département
```

Remarquons que dans un énoncé FML l'expression "WITH Salaire > 10000" n'est pas permise :

une affectation doit être spécifiée. Ici, l'expression "AND Département" indique sémantiquement *"et un département lui a été attribué"* : ceci n'est pas non plus accepté pour les énoncés de mise à jour du FML.

A ce niveau, les autres types de précondition ne posent pas de problèmes particuliers, ainsi nous ne donnerons plus que deux exemples.

Exemple 4.14 : Créer une activité pour chaque Employé qui devient un Vendeur.

```
TEST   INSERT   Vendeur FE
       FROM     Employé FE
```

Exemple 4.15 : Créer une activité pour chaque département de type "A" dont on supprime son directeur.

```
TEST   DELETE   [Directeur e, Dépt-Dirigé FE]
       FROM     Employé e, Département FE
       WHERE    FE.Type = 'A'.
```

A propos de ce type de précondition, une dernière remarque mérite d'être faite : si une classe d'activités a une précondition du type changement, alors une même instance de son entité focale peut être liée à plusieurs de ses instances (instances de l'activité).

4.2.2. Préconditions d'Etat

Comme il a été dit, ces préconditions doivent exprimer un état de la base de données. Maintenant, si nous prenons en compte le fait que la validation d'une précondition est réalisée en ayant comme objectif la détermination d'une entité focale pour une nouvelle activité, alors nous pouvons dire qu'elle teste essentiellement l'état d'une entité dans la base de données.

Une syntaxe qui convient à cette sémantique est :

```
TEST   STATE    <Entity>
       FROM     ...
       WHERE    ...
```

La clause "TEST STATE" spécifie l'entité à tester, c'est-à-dire l'entité focale. La clause "FROM" spécifie, comme toujours, le contexte de la précondition. Et la clause "WHERE" spécifie des conditions indiquant l'état que l'entité focale doit vérifier. Bien sûr, des conditions n'ayant pas de rapport direct avec l'entité peuvent être spécifiées dans cette clause ; néanmoins ceci est assez rare.

Dans ces préconditions, contrairement à celles de changement, il n'y a pas besoin d'identifier l'entité focale par la variable FE ou le caractère "*", ceci peut néanmoins se faire. Quelques exemples sont ensuite présentés.

Exemple 4.16 : Créer une activité pour chaque département dont la moyenne des salaires de ses employés est supérieure à 15000.

```
TEST   STATE    FE
       FROM     Département FE.Employé e
       WHERE    Avg (e.Salaire) > 15000
```

Exemple 4.17 : Créer une activité pour chaque département si la moyenne des salaires de tous les employés (de tous les départements) est supérieure à 15000.

```
TEST   STATE    FE
       FROM     Département FE , Employé e
       WHERE    Avg (e.Salaire) > 15000
```

Ces deux exemples permettent d'illustrer l'importance de l'entité focale dans l'instanciation d'activités associées à des préconditions d'état. Dans le dernier exemple, la condition de la clause "WHERE" est indépendante de l'entité focale : si cette condition est validée, alors la précondition est validée pour chaque département.

4.2.3. Conclusion

Dans le chapitre 2, la **forme** de base d'un déclencheur a été établie comme composée par un évènement, une condition et une action (cf. 2.1.1). Le prédicat selon lequel l'action est déclenchée est alors divisé en deux parties : la partie évènement et la partie condition.

Dans la syntaxe des préconditions de changement nous distinguons aussi un évènement, donné par la clause "TEST <Opération>" et une condition, donnée par la clause "WHERE" (la clause "FROM sert à préciser la partie de la base de données concernée par la précondition). Néanmoins, dans une précondition, les spécifications de l'évènement et de la condition sont confondues dans un seul énoncé dans le but de pouvoir utiliser le langage de manipulation pour définir les préconditions. Or, ceci peut impliquer quelques restrictions car on essaye de donner à une précondition une sémantique très proche de celle d'un énoncé de manipulation. Par exemple, quand on se réfère dans la clause "WHERE" d'une précondition à l'ensemble des valeurs de la fonction à mettre à jour, l'état testé est celui antérieur à l'exécution de la mise à jour (cf. exemple 4.9).

D'un autre côté, les préconditions d'état ne spécifient pas explicitement un évènement. Bien que ce type de préconditions ne permette pas un contrôle aussi fin sur la base de données que celles de changement, elles sont intéressantes vu leur forme très concise. Par exemple, la précondition de l'exemple 4.16 correspond à tout un ensemble de préconditions de changement : il faut plusieurs préconditions pour contrôler les changements du fait entre Département et Employé (préconditions d'insertion de faits, modification et suppression) et plusieurs pour contrôler l'association entre Employé et Salaire.

Nous pensons que les deux types de précondition se complètent. Ceci est plus remarquable quand on verra l'éventail des activités qu'on peut spécifier à partir des différentes combinaisons entre le type de précondition et le mode de lancement d'une activité (cf. 4.5).

Dans le chapitre 2 la liste des différents **types d'évènements de déclenchement** possibles a été dressée (cf. 2.1.1.1.1). Pour les préconditions d'activités, seules des opérations sur la base de données sont permises. En effet, nous sommes en train de décrire le niveau bas du modèle de procédures ; par ailleurs, les préconditions de précédence entre les activités internes à une procédure admettent d'autres types d'évènements beaucoup plus puissants, ainsi que la composition d'évènements atomiques.

Remarquons aussi que le FML, langage utilisé pour spécifier les préconditions, est un langage sémantique et est donc de plus haut niveau que le langage relationnel. Ainsi, les opérations base de données admises comme évènements ne comprennent pas uniquement les trois opérations de mise à jour basiques et s'appliquent à plus d'un objet (par opposition aux mises à jour relationnelles qui ne s'appliquent qu'à une seule relation).

La **façon dont une activité est instanciée** englobe les deux orientations étudiées dans 2.1.3: ensembliste et instance par instance. L'approche adoptée, basée sur l'entité focale, est originale et induit une façon intéressante de concevoir les applications.

Par rapport au **contexte passé de l'évènement à la condition**, la même démarche que [Gib84] ou que celle du projet HIPAC a été prise : il est passé en utilisant explicitement des variables. Le contexte passé de la condition à l'action (corps d'exécution) sera étudié dans le paragraphe suivant.

Pour définir précisément la sémantique des préconditions, il est encore necessaire d'étudier les **règles qui gouvernent l'activation** d'une précondition (i.e., quelles mises à jour provoquent l'évaluation d'une précondition). Bien que les préconditions soient exprimées en termes d'énoncés du FML, leur correspondance avec les mises à jour n'est pas purement syntaxique. Ceci est dû à l'utilisation d'un modèle sémantique et d'un langage de manipulation de haut niveau. Ainsi, établir cette correspondance nécessite de l'inférence au niveau sémantique : en particulier, la hiérarchie de généralisation et le fait que certaines opérations de mise à jour incluent sémantiquement d'autres opérations, doivent être pris en compte. Ces règles de correspondance ne sont établies qu'au chapitre 6 ; ici, je présente uniquement un exemple qui illustre ce qui vient d'être dit. Cet exemple concerne le fait que la précondition ci-dessous (exemple 4.18) est activée par la mise à jour (l'exemple 4.19) :

Exemple 4.18 : Créer une activité pour chaque employé qui reçoit un nouveau salaire.

```
TEST        INSERT    [Employé FE, Salaire s]
            FROM      Employé FE , Employé.Salaire s
```

Exemple 4.19 : Augmenter de deux pour cent le salaire de tous les vendeurs

```
            ASSERT    Vendeur.Salaire = Vendeur.Salaire * 1.02
```

Bien que l'opération de la précondition soit une insertion de faits et celle de la mise à jour une modification de faits, et que l'opération de la précondition s'applique à un employé alors que celle de

la mise à jour s'applique à un vendeur, il y a quand même activation. En effet, une modification inclue sémantiquement une insertion de faits et les activités sont héritées à travers la hiérarchie de généralisation par les entités qui concernent leur précondition.

Définitions en Termes du Modèle de Données

Toute classe d'activités est associée à une précondition, ainsi la déclaration suivante s'impose :

[Activity-Class (1,*), Precondition (1,1)]

"Precondition" est définie comme un type de base : un type abstrait appartenant au niveau bas du FM. Sa structure interne résulte de sa compilation et n'est présentée qu'au chapitre 8, mais elle comprend aussi la chaîne de caractères représentant l'énoncé de la précondition. La seule opération accessible aux utilisateurs est l'édition de la précondition. Par ailleurs, au niveau de l'implémentation, plusieurs opérations sont définies comme l'activation et l'évaluation de la précondition. Le nom de ce type de base est "OP-Pcd" ("OP" pour "Office Procedure" et "Pcd" pour "Precondition"). Nous avons alors :

Precondition : OP-Pcd

4.3. Contexte d'Activation

Ce paragraphe commence par un exemple dont l'objectif est d'illustrer la notion de contexte d'activation.

L'exemple concerne l'activité "Comm-Projet" : cette activité met en œuvre le fait que chaque fois qu'un employé est affecté à un projet, une commission lui est attribuée. La valeur de cette commission est calculée à partir de la catégorie du projet.

L'énoncé de la précondition est :

```
TEST    INSERT   [Employé FE, Projet p]
        FROM     Employé FE , Projet p
```

L'exécution de l'activité ne se fait pas dès que la précondition est devenue vraie : son mode de lancement est manuel (cf. 4.5). Ainsi, l'activité ne sera exécutée que plus tard à la demande d'un utilisateur : par exemple celui qui est chargé de l'attribution des commissions.

Supposons maintenant que l'employé Viriato est affecté au projet DOEOIS (i.e., un fait entre l'employé Viriato et le projet DOEOIS a été inséré). Ceci implique la création d'une instance de l'activité "Comm-Projet" - appelons cette instance Ai - ayant comme entité focale l'employé Viriato. Ai est donc habilitée à être exécutée ; néanmoins, le système attend une demande explicite d'un utilisateur pour lancer son exécution.

Que se passe-t-il si, avant l'exécution de l'activité Ai, Viriato quite ce projet, i.e., si le fait entre Viriato et le projet DOEOIS est supprimé de la base de données ?

Quand l'activité sera lancée elle ne va pas pouvoir accéder aux informations qui lui sont essentielles. En effet, l'activité doit s'exécuter pour l'employé Viriato (l'entité focale) et elle a besoin

de connaître le projet qui lui a été affecté (et pour lequel la précondition est devenue validée) pour consulter sa catégorie et en déduire la commission à affecter. Ainsi, pour que Ai puisse s'exécuter correctement il lui faut connaître les informations suivantes :

- son entité focale, i.e, l'entité virtuelle représentant l'employé Viriato
- le projet auquel Viriato a été affecté au moment où la précondition a été validée, i.e., l'entité virtuelle représentant le projet DOEOIS.

Il va de soi que l'accès à l'entité focale pendant l'exécution d'une activité doit être assuré : ceci est la base de la notion d'instance d'activité créée pour une instance d'entité focale. D'ailleurs, l'entité focale peut être vue comme un paramètre du corps d'exécution de l'activité.

Le problème qui se pose est que des entités liées à l'entité focale et qui ont contribué à la validation de la précondition peuvent changer d'état pendant la période de temps écoulée depuis la validation de la précondition jusqu'à l'exécution de l'activité.

Verrouiller tous les faits référencés dans la précondition au moment de sa validation et jusqu'à l'exécution de l'activité créée en conséquence, serait une solution inacceptable : le temps pendant lequel les faits seraient indisponibles pourrait être très grand et empêcherait la base de données d'évoluer.

Brièvement, la solution adoptée a été de garder la valeur des faits référencés dans la précondition au moment où la précondition est validée et une instance d'activité est créée. L'ensemble des valeurs de ces faits est connu comme le **contexte d'activation** d'une activité. Ce contexte est stocké dans la base de données au moment de la création d'une instance d'activité et gardé jusqu'à la fin de son exécution. Des moyens d'accès au contexte d'activation de l'intérieur d'une activité sont offerts par le système, comme on le verra plus tard.

Bien qu'il ne soit pas envisageable de verrouiller les faits référencés dans une précondition au moment de sa validation, il est néanmoins souhaitable de verrouiller l'information critique d'une activité (i.e., l'entité focale et la valeur actuelle des faits de la précondition) pendant son exécution.

Remarquons que la notion de contexte d'activation n'a pas beaucoup de sens pour les activités associées à des préconditions d'état. En effet, une telle activité est lancée dans un état de la base de données qui valide la précondition : rappelons nous que ces activités sont créées et supprimées selon que leur précondition est vraie ou devient fausse. Ainsi, au moment du lancement de l'exécution d'une telle activité les faits référencés par la précondition sont dans l'état qui a impliqué la création de cette activité.

Garder le contexte d'activation est alors important pour les activités ayant une précondition de changement. Dans l'exemple d'illustration, une activité manuelle a été prise car ce type de lancement met plus en évidence la nécessité de garder le contexte d'activation : un grand laps de temps peut se passer entre l'habilitation et le lancement d'une activité. Mais le contexte d'activation est aussi important pour les cas des activités automatiques ; ceci pour les raisons suivantes :

- Supposons que la fonction qui lie un employé à un projet est multivaluée et que Viriato était dejà lié à d'autres projets ; à l'exécution de l'activité on ne s'intéresse pas à tous les projets auxquels il a été affecté mais à celui qui a impliqué la validation de la précondition. Ainsi, le

contexte d'activation est important pour permettre l'identification du changement qui a validé la précondition et pour lequel une activité doit être exécutée.

- Dans le cas des activités dont les préconditions expriment une suppression de faits ou d'entités, au moment de leur exécution, ces faits n'existent plus dans la base de données et ne sont donc plus accessibles. Le contexte d'activation doit permettre de garder leur valeur. Remarquons que pour ce type de problème (ainsi que pour le précédent) la solution de verrouillage au moment de la validation de la précondition n'est pas satisfaisante : elle n'a même pas de sens.

- Une précondition devient vraie au moment d'une mise à jour, mais l'habilitation de l'activité créée en conséquence n'a lieu qu'à la fin de la transaction où la mise à jour à été réalisée[3]. De plus, l'exécution d'une activité ne se fait pas dans la transaction où elle a été habilitée à être exécutée et peut-être pas dans la même session SIB (cf. 4.6) ; ceci ayant comme conséquence que les verrous sur les faits de la précondition peuvent ne pas être hérités.
  Ainsi, les faits référencés par la précondition peuvent changer de valeur pendant le moment de validation de la précondition et l'habilitation de l'activité créée en conséquence.

La notion de contexte d'activation ainsi que les cas d'activité pour lesquels ce concept est important viennent d'être présentés. Maintenant, il est intéressant de définir plus précisément ce qu'est le contexte d'activation.

Dans un premier temps nous avons défini le contexte d'activation comme l'ensemble des valeurs des faits référencés dans la précondition. L'idéal serait qu'il soit constitué de tous les faits référencés dans la précondition mais ceci impliquerait une surcharge pour le système. Cette surcharge n'est pas toujours justifiable vu que pour certains cas l'activité n'a pas besoin de toutes ces informations. Ainsi, plusieurs solutions ont été examinées :

(i) Le contexte d'activation est constitué uniquement des entités (constituant des fois des faits) de la clause "TEST <opération>". Si ces entités sont des entités virtuelles, on offre à l'utilisateur la possibilité de les référencer et d'accéder ainsi à leur faits : au niveau implémentation ceci est fait en gardant l'identificateur interne de ces entités. S'il s'agit d'entités littérales, alors on stocke leur valeur.

(ii) Le contexte d'activation est constitué de (i) et des entités de la clause "FROM" qui sont liées aux entités de (i);

(iii) Les entités ou faits constituant le contexte d'activation doivent être explicitement définis par l'utilisateur au moment de la définition d'une classe d'activités.

La troisième solution bien qu'intéressante peut être une source d'erreurs de la part de l'utilisateur quand il sélectionne les entités qui constituent le contexte d'activation : pour cette raison la solution

[3]Ceci est fait dans le but d'éviter d'habiliter des activités à l'intérieur de transactions qui peuvent être postérieurement annulées ; dans 4.6 et 7 ce sujet sera repris.

(iii) a été écartée pour l'instant.

La première solution semble assez restrictive. De plus, l'entité focale peut ne pas être présente dans la clause "TEST <opération>" (cf. 4.2), alors qu'il est naturel qu'elle appartienne au contexte d'activation.

Finalement c'est la deuxième solution qui a été retenue. Cette solution n'est pas très restrictive : l'utilisateur peut déclarer dans la clause "FROM" toutes les variables de la précondition s'il souhaite qu'elles appartiennent au contexte d'activation. De plus, il peut aussi y spécifier des faits sur lesquels ne sont pas posés des conditions dans la clause "WHERE", i.e., des faits spécifiés seulement dans le but d'appartenir au contexte d'activation. Par exemple, si pour l'activité "Comm-Projet" on veut garder la catégorie du projet, il suffit d'écrire la clause "FROM" de la façon suivante: "FROM Employé FE, Projet p.Catégorie c". Nous ne prétendons pas que la solution choisie soit la meilleure, seulement une évaluation du prototype réalisé permettra de conclure là-dessus.

Nous considérons le contexte d'activation comme une photographie d'une partie de l'état de la base de données (la partie qui constitue le contexte d'activation). Ainsi, le contexte d'activation peut être vu comme une instance d'une partie du schéma conceptuel. Cette partie du schéma est définie par les clauses "TEST <Operation>" et "FROM". Par exemple, pour l'activité "Comm-Projet" ce schéma est :

Employé : Entity
Projet : Entity
[Employé (1,*) , Projet (0,*)]

Pour une précondition d'insertions la photographie est prise juste après l'exécution de la mise à jour qui a provoquée sa validation. Pour une précondition de suppressions la photographie est prise juste avant la mise à jour. Pour une précondition de modification de faits, la photographie est prise avant et après la mise à jour.

Ainsi, la partie du schéma conceptuel des faits appartenant au contexte d'activation a deux instanciations : celle qui correspond à l'état actuel de la base de données et celle qui correspond au contexte d'activation, i.e., à l'état - au moment de la validation de la précondition - des entités et des faits qui ont contribué à la satisfaction de la précondition. Le contexte d'activation n'est accessible que de l'intérieur de l'activité correspondante.

La figure 4.3 illustre ce qui vient d'être dit. Elle reprend l'exemple de l'activité "Comm-Projet" et considère la clause "FROM" de sa précondition comme : "FROM Employé FE, Projet p.Catégorie c". On suppose qu'une mise à jour qui a lié l'employé e1 au projet p1 de catégorie "A" a validé la précondition. Ainsi, une activité a été habilitée à être exécutée, ayant un contexte d'activation constitué de l'employé e1 lié au projet p1 de catégorie "A".

Dans la figure, trois cas différents sont considérés :

(a) L'état des faits et entités du contexte d'activation n'a pas changé. L'ensemble de valeurs du contexte d'activation est un sous-ensemble de la base de données actuelle.

(b) Dans l'état actuel de la base de données l'employé e1 n'est plus lié au projet p1, de plus, le projet p1 a changé de catégorie. L'ensemble de valeurs du contexte d'activation n'a qu'une intersection avec l'état actuel de la base de données : {e1}.

(c) Dans l'état actuel de la base de données le projet p1 n'existe plus. L'ensemble de valeurs du contexte d'activation n'a qu'une intersection avec l'état actuel de la base de données : {e1}.

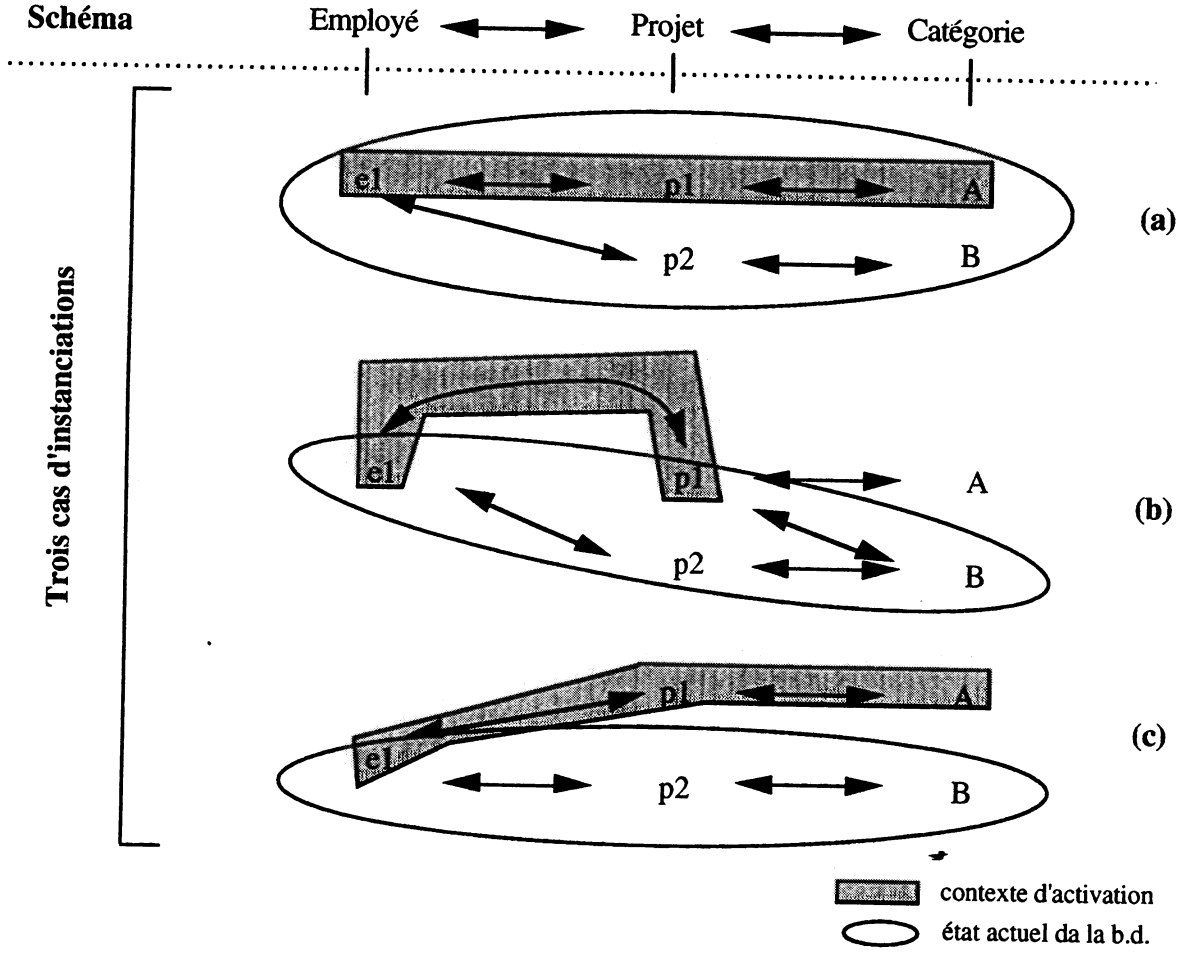


Figure n° 4.3 - Contexte d'Activation et Etat Actuel

L'intérêt d'avoir cette vue sur le contexte d'activation est de pouvoir accéder aux valeurs des faits du contexte au moment de la validation de la précondition en utilisant les mêmes mécanismes que pour accéder aux valeurs actuelles de ces faits. Comme il a été dit ci-dessus, la valeur du contexte d'activation n'est accessible que de l'intérieur de l'exécution de l'activité correspondante. Ainsi, l'utilisateur ou le programme d'application qui réalise l'activité a la possibilité d'accéder soit à la base de données dans son état actuel, soit au contexte d'activation. Il faut donc offrir les moyens de faire la distinction entre les deux instanciations du schéma conceptuel des faits constituant le contexte d'activation. En conséquence, il a été établi que :

- La valeur actuelle de la base de données est accessible comme toujours. Par exemple, la valeur de la fonction **Projet** () est pour les figures 4.3 (a) et (b) l'ensemble {p1, p2}, et pour (c)

l'ensemble {p2}

- Pour spécifier qu'on se réfère à la valeur d'un fait (d'une fonction) dans le contexte d'activation (ie. au moment de la validation de la précondition) on utilise le caractère guillemet. Ainsi, la fonction **Projet"** () a comme valeur pour 4.3 (a), (b) et (c) l'ensemble {p1}.

Quelques exemples suivent :

- La fonction **Projet".Catégorie** accède à la catégorie actuelle du projet (des projets, si insertion multivaluée) qui a été affecté à l'employé au moment de la validation de la précondition. Sa valeur est pour 4.3 (a) l'ensemble {'A'} et pour (b) l'ensemble {'B'} et pour (c) elle est indéfinie (NULL).

- La fonction **Projet".Catégorie"** accède à la catégorie à laquelle appartenait le projet qui a été affecté à l'employé au moment de la validation de la précondition. Sa valeur est pour 4.3 (a), (b) et (c) l'ensemble {'A'}.

- La fonction **Employé".Projet** accède aux projets actuels de l'employé qui a contribué à la validation de la précondition. Sa valeur est pour 4.3 (a) l'ensemble {p1, p2} et pour (b) et (c) l'ensemble {p2}.

- La fonction **Employé".Projet"** accède aux projets qui ont été liées à employé lors de la validation de la précondition. Remarquons que pour cette précondition Employé".Projet = Projet".

Quelques remarques suivent :

- L'**entité focale**. *Employé"* désigne en fait l'entité focale de l'activité ; ainsi que ce soit de l'intérieur ou de l'extérieur de l'activité cette entité peut être accessible par FE : rappelons nous que FE est associé à l'activité dans le schéma conceptuel de l'application (cf. figure 4.1 dans § 4.1). La fonction Employé".Projet s'écrit alors FE.Projet.

- La notion de **contribution à la satisfaction de la précondition**. Dans le contexte d'activation sont gardés seulement les faits dont la valeur a contribué à la satisfaction de la précondition. Reprenons la précondition de l'exemple 4.7 :

```
TEST    INSERT    [Département FE, Budget b]
                  FROM    Département FE.Employé e, Département.Budget b
                  WHERE   b < 10 * e.Salaire
```

Si la précondition est validée, une activité est créée dont le contexte d'activation est constitué du département mis à jour, de son budget et des employés de ce département dont le salaire est supérieur à 10% le budget : les autres employés ne sont pas stockés dans le contexte d'activation.

- Les **chemins cycliques**. Une autre version de la précondition de l'exemple 4.9 est :

  TEST INSERT [Employé FE, Salaire s]
  FROM Employé FE.Département.Employé e, Employé .Salaire s
  WHERE Avg (e.Salaire) *1.2 < s

  Dans la clause "FROM" il y a un chemin cyclique, i.e., une expression de combinaison de fonctions où la même fonction apparaît plus d'une fois : "Employé FE.Département.Employé e. Le contexte d'activation induit est constitué de l'employé FE et des départements dont l'ensemble des employés satisfait la précondition : nous n'acceptons pas que ces employés fassent partie du contexte. En effet, la partie du schéma conceptuel référencée dans cette expression est :

  Employé : Entity
  Département : Entity
  [Employé, Département]

  Or, l'entité Employé ne peut pas avoir deux sens : être l'entité focale d'une part et être l'ensemble des employés du département de l'employé représentant l'entité focale d'autre part. Ainsi, si dans la clause "FROM" d'une précondition nous avons la combinaison de fonctions : f1.f2. ... .fn (où f1 appartient à la clause "TEST <opération>") ; alors les valeurs gardées dans le contexte d'activation sont celles de f1, f1.f2, f1.f2.f3, ... tant qu'il n'y a pas répétition de fonctions.

Malheureusement, les travaux sur le contexte d'activation n'ont pas été menés à terme : une formalisation de sa définition, de la déduction de son schéma conceptuel à partir d'une précondition ainsi que des expressions combinant des références à lui et à la valeur actuelle de ces faits, s'impose. Le sujet central de cette thèse n'est pas celui-là. Néanmoins, l'idée est là et dans le chapitre 8 est développe la façon dont l'implémentation du contexte d'activation peut être réalisée.

Il est intéressant de souligner qu'un SGBD offrant une gestion des historiques [Bui86] aurait été d'une grande utilité pour faire la gestion du contexte d'activation.

Pour finir, j'aimerais mettre l'accent sur l'intérêt de cette notion :

- Par rapport à la façon dont les mécanismes de déclencheurs étudiés dans le chapitre 2 passent le contexte de l'évènement et de la condition à l'action (cf.2.1.2), nous pensons que la notion de contexte d'activation présentée ici est plus **riche**. En effet, les mécanismes de déclencheurs passent le contexte soit de façon implicite et très limitative, soit par l'utilisation de variables. Or, le contexte d'activation a un schéma conceptuel qui lui est associé et il peut être vu comme une photographie de la base de données.

- La notion du contexte d'activation permet d'accéder de façon **homogène** aux valeurs des faits de la précondition au moment de sa validation et au moment de l'exécution de l'activité.

- En conséquence de la richesse de cette notion, la syntaxe utilisée pour accéder aux faits du contexte d'activation est **concise** par rapport à celle proposée dans les mécanismes de déclencheurs qui utilisent des variables. Par exemple, supposons la clause "FROM" de la

précondition de l'activité "Comm-Projet" : "FROM Projet p.Catégorie c, ..." ; au moment de l'exécution de l'activité nous accédons aux catégories actuelles des projets qui ont validé la précondition par l'expression Projet".Catégorie. Par contre avec la solution utilisant des variables il aurait fallu faire référence à la valeur actuelle de la fonction Projet.Catégorie et la qualifier par les projets dont la valeur appartient à l'ensemble de valeurs de la variable p.

- La solution du contexte d'activation est mieux adaptée que celle des mécanismes de déclencheurs étudiés au **milieu bureautique** où les actions à entreprendre peuvent ne pas être prédéfinies à l'avance et être exécutées longtemps après que l'évènement de déclenchement ait eu lieu. Les mécanismes de déclencheurs se rapportent surtout à des environnements où les actions sont réalisées au plus tard à la fin de la transaction de déclenchement.

4.4. Corps d'Exécution

Deux types d'activités sont proposés :

- Les activités de plus bas niveau qui **ne sont pas associées à un corps d'exécution**, i.e., à une action prédéfinie. Dans ce cas, l'utilisateur est prévenu que la précondition d'une activité est devenue vraie et que l'activité est habilitée à être exécutée. L'utilisateur a alors la flexibilité de choisir une action à entreprendre et de la réaliser. Ainsi, l'effet de chaque instance d'une classe d'activités peut être différent.

- Les activités de plus haut niveau qui **sont associées à un corps d'exécution**. Ces activités modélisent des tâches plus structurées et mieux connues de l'utilisateur : il peut, à l'insertion d'une classe d'activités, prédéfinir une action à entreprendre en cas de validation de la précondition et l'associer à l'activité. Ainsi, toutes les instances d'une classe d'activités ont le même effet.

Le corps d'exécution est défini entre autre par un programme. De façon générale, ce programme manipule l'entité focale et le contexte d'activation. Le programme est considéré comme une boîte noire par le système, et son contenu est de la responsabilité du programmeur de l'application. Le programme peut être écrit dans n'importe quel langage ; par exemple, il peut être une séquence d'opérations base de données, du code écrit dans un langage de programmation admettant des accès à la base de données, du code écrit dans un langage autonome, etc. Souvent ce programme active des outils, tels que les éditeurs de formulaires ou de documents, le courrier électronique, etc., ou des procédures spécifiques comme des programmes de comptabilité ou de facturation.

Pour l'instant le seul langage qui admet des références à la base de données du SIB est le FML ; ceci est très restrictif vu qu'il y a une vraie nécessité à lier des applications déjà existantes au support des procédures de bureau. Ainsi, une autre étape dans ce projet s'impose, néanmoins elle sort du cadre du travail présenté dans ce document.

Définition en Termes du Modèle de Données

Vu qu'une activité peut avoir ou non un corps d'exécution qui lui est associé, nous avons la définition standard suivante :

[Activity-Class (1,*) , Execution-Body (0,1)]

Le corps d'exécution d'une activité est défini par un programme mais aussi par d'autres éléments, comme par exemple, le site où le programme doit être exécuté. Les autres composants du corps d'exécution ne seront présentés que dans le paragraphe 4.6 où est analysée la façon dont l'exécution d'une activité est lancée. Ainsi, le corps d'exécution est une entité virtuelle définie par plusieurs associations (entre autres l'association avec un programme). Ceci implique :

Execution-Body : Entity
[Execution-Body (0,*), Program-Name (1,1)]
Program-Name : String

"Program-Name" est le nom du programme qui réalise l'objectif de l'activité.

4.5. Mode de Lancement

Comme il a déjà été dit plusieurs fois, deux modes de lancement de l'exécution du corps d'une activité sont offerts :

- **Mode automatique** : l'exécution de l'activité est déclenchée automatiquement par le système dès que l'activité est habilitée à être exécutée.

- **Mode manuel** : une fois qu'une activité est habilitée à être exécutée, le système doit recevoir une commande explicite (cf. 4.6) de l'utilisateur ou d'une application pour lancer l'exécution de l'activité.

En termes du modèle de données, définir le mode de lancement ne pose pas de problèmes :

[Activity-Class (1,*), Start-Mode (1,1)]
Start-Mode : ('Automatic', 'Manual')

La combinaison de ces deux modes de lancement avec les deux types de précondition proposés (de changement et d'état) et avec le fait qu'une activité puisse avoir ou non un programme associé permet de modéliser toute une gamme d'applications. Dans ce paragraphe, un exemple typique d'une activité manuelle avec une précondition d'état et d'une activité automatique avec une précondition de changement est donné ; je pense que ceci éclaircira aussi la différence entre les deux types de précondition.

Exemple 4.20 :

Cet exemple est basé sur une nouvelle application : la gestion des projets ESPRIT ; elle est présentée dans l'annexe III. Il modélise le fait que quand le stock d'un certain article (identifié par sa référence) plus la quantité du même article qui a éventuellement déjà été commandée est inférieur à un certain seuil, alors une nouvelle commande doit être envoyée.

Ceci est modélisé par deux activités. La première fait la préparation et l'envoi de la commande ; elle est manuelle car la présence d'un employé de bureau est necessaire. La deuxième sert à avertir cet employé qu'une commande doit être faite dès que le seuil n'est plus respecté.

```
(1)   INSERT   Activity-Class
      WITH     Name = 'Commande-Papeterie'
      AND      FE-Class = Article-Papet
      AND      Precondition =
               TEST   STATE   FE
                      FROM    Article-Papet FE (Seuil s, Stock s1, Stock-
                              Commandé s2)
                      WHERE   s1 + s2 < s
      AND      Mode = 'Manual'
      AND      Execution-Body[4] = { • Préparation de la commande
                                     • Envoie de la commande
                                     }

(2)   INSERT   Activity-Class
      WITH     Name = 'Avertissemnt-Commande'
      AND      FE-Class = Article-Papet
      AND      Precondition =
               TEST   INSERT   [Article-Papet FE, Stock sf]
                      FROM     Article-Papet FE (Seuil s, Stock-
                               Commandé s2) , Article-Papet.Stock s1[5]
                      WHERE    s1 + s2 < s
      AND      Mode = 'Automatic'
      AND      Execution-Body = { • Envoyer un message au responsable en
                                  avertissant qu'une commande doit être établie et
                                  envoyée}
```

La première activité illustre les préconditions d'état pour les activités manuelles : elles sont essentiellement caractérisées par le fait qu'une telle précondition peut être validée et habiliter l'activité correspondant à être exécutée, et ensuite devenir fausse. Dans ce cas, l'activité est supprimée. Dans

[4]Ici le corps d'exécution est donné sous forme de commentaire.

[5]L'expression "Article-Papet FE (Seuil s, Stock-Commandé s2)" désigne un trajet arborescent dans le schéma conceptuel. On fait référence au seuil et au stock commandé d'un certain article de papeterie.

l'exemple, quand le stock existant plus le stock déjà commandé est inférieur au seuil, une activité est créée. Si entretemps, il devient (pour n'importe quelle raison) supérieur au seuil avant que l'activité soit exécutée, alors l'activité est éliminée de la base de données.

Cette première activité illustre aussi une autre caractéristique des préconditions d'état : si à la fin de l'exécution d'une activité (qui a été créée pour une certaine entité focale), la précondition est toujours vraie pour la même entité focale, alors une autre activité est créée de suite. Ainsi, si l'activité "Commande-Papeterie" ne commande pas une quantité d'articles suffisante pour dépasser le seuil, alors à la fin de son exécution une autre activité est habilitée à être exécutée ayant comme entité focale le même article.

La deuxième activité illustre les préconditions de changement pour les activités automatiques. Remarquons que si la précondition était d'état, alors une activité serait créée et exécutée sans arrêt jusqu'à ce que la condition ne soit plus vraie : elle bouclerait !

4.6. Etat d'une Instance

Les états d'une instance d'activité ont déjà été introduits de façon intuitive. Dans ce paragraphe ils sont établis plus précisément : les primitives permettant le passage d'un état à un autre sont aussi présentées.

La figure 4.4 présente les différents états d'une activité.

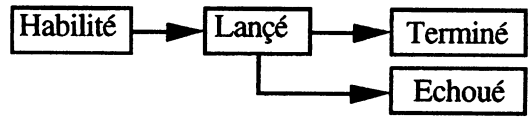


Figure n° 4.4 - Etats d'une Instance d'Activité

Quand la précondition d'une classe d'activités devient vraie pour une certaine entité focale, alors une instance d'activité est créée et habilitée à être exécutée. Ainsi, à la création - nous disons aussi à l'instanciation - d'une activité son état est habilité ("enabled").

L'instance d'activité peut alors passer à l'état lancé ("started") ; ceci peut se faire de deux façons : soit automatiquement par le système (mode de lancement automatique ; cf. 4.5), dans ce cas, l'activité passe de l'état habilité à l'état lancé immédiatement ; soit par la reception de la primitive "START" (lancement en mode manuel) réalisée par l'utilisateur ou par une application, dans ce cas, l'activité dans l'état habilité attend cet ordre pour passer à l'état lancé.

La terminaison de l'exécution d'une activité peut être normale et elle passe de l'état lancé à l'état terminé ("finished"). La terminaison doit être toujours signalée explicitement au système par l'utilisation de la primitive "END", ceci dans les deux cas : l'activité a ou n'a pas un programme associé.

Par ailleurs, il peut être necessaire de terminer une activité anormalement : elle passe alors de

l'état lancé à l'état échoué ("failed"). Deux primitives permettent d'effectuer ce passage : la primitive "ABORT", utilisée uniquement de l'intérieur de l'activité, et la primitive "FAIL" qui peut être réalisée par le système ou par une autre session SIB.

En termes du modèle de données, la définition de l'état d'une activité ne pose pas de problèmes. Nous avons les déclarations standards suivantes :

[Activity (0,*), Activity-Status (1,1)]
Activity-Status : ('Enabled', 'Started', 'Finished', 'Failed')

L'ensemble des primitives de manipulation des activités et des procédures constitue le "Office Procedure Control Language" (OPCL). La syntaxe proposée pour spécifier les primitives est un sur-ensemble du FML. Ainsi, l'utilisateur peut, sans avoir besoin d'un nouveau langage, sélectionner un groupe d'instances d'activités - qualifié par exemple en fonction de critères concernant l'information qu'elles manipulent - et lancer leur exécution, ou faire une autre manipulation.

La syntaxe est alors de la forme : "<nom de la primitive> ... FROM ... WHERE ...". Les primitives effectuées de l'intérieur d'une activité et concernant cette même activité n'ont pas besoin de la clause "FROM" et "WHERE". Par exemple, à la fin de l'exécution d'une activité, pour signaler sa terminaison normale, il suffit d'écrire "END Activity" : le système est bien sûr capable d'identifier l'activité.

Une présentation plus détaillée de ces primitives va permettre de mieux comprendre quelques caractéristiques du modèle d'exécution sous-jacent aux activités. Le lancement et la terminaison d'une activité seront étudiés ainsi que les liens entre l'exécution d'une activité et, d'une part, le système transactionnel et, d'autre part, le système de verrouillage des données.

Tout d'abord, une description du principe essentiel du modèle d'exécution en ce qui concerne le système transactionnel s'impose. Ce principe dit qu'**une transaction ne peut pas dépasser les bornes d'une activité**, i.e., une activité ne peut pas démarrer à l'intérieur d'une transaction, et à la fin d'une activité toute transaction doit être terminée. Par contre, à l'intérieur d'une activité tout appel au système transactionnel est permis ; ainsi, une ou plusieurs transactions (également pour les transactions imbriquées) peuvent être incluses dans une activité. Deux raisons principales mènent à imposer une telle restriction. La première est relative au fait que les activités limitent de façon naturelle des unités cohésives de travail. La deuxième concernent le niveau des procédures de bureau: il n'est pas souhaitable qu'une procédure soit reprise (défaite) entièrement lorsqu'une défaillance survient pendant son exécution, i.e., une défaillance ne doit en aucun cas produire l'annulation des activités déjà effectuées par la procédure [Ped88]. Ainsi, les activités sont considérées comme l'**unité de reprise maximale** en ce qui concerne le support des procédures de bureau.

Pour des questions de flexibilité, au début d'une activité une transaction n'est pas lancée implicitement ; l'utilisateur le fera explicitement s'il le souhaite.

Voyons maintenant la fonction des différentes primitives et la façon dont elles assurent ce principe.

Lancement d'une Activité

Nous distinguons quatre cas selon le mode de lancement et le fait qu'une activité ait ou non un corps d'exécution qui lui est associé.

CAS 1 : Activité automatique ou manuelle avec programme associé

Quand une activité automatique est lancée une nouvelle session SIB est implicitement ouverte. Les faits du contexte d'activation[6] sont verrouillés et les verrous sont passés à cette session. L'exécution de l'activité se déroule alors dans cette session. La fin de l'exécution coïncide avec la fermeture de la session.

Au moment de la définition de la classe d'activités l'utilisateur doit spécifier le site où il veut que le programme associé soit exécuté ; un caractère spécial - "!" - est réservé pour indiquer que le site d'exécution est celui où l'activité a été habilitée à être lancée. De même une adresse du terminal d'exécution doit être spécifiée à la définition de la classe ; un caractère spécial - "!" - est réservé pour indiquer que l'adresse du terminal est celle où la mise à jour provoquant la validation de la précondition a été réalisée.

CAS 2 : Activité manuelle sans programme associé

Dans ce cas, l'utilisateur a la responsabilité de l'exécution d'une action. La réalisation de cette activité se fera dans la session SIB, le site et le terminal d'où provient l'ordre de lancement. Le système transmet des verrous sur le contexte d'activation à cette session et permet l'accès à la valeur du contexte d'activation lors de la validation de la précondition. Pendant l'exécution de l'action, les verrous qui ont été déjà acquis par la session sont hérités.

CAS 3 : Activité automatique sans programme associé

Cette combinaison ne semble pas très cohérente : si une application provoque l'habilitation d'une activité sans programme et si l'utilisateur ayant la responsabilité de réaliser l'activité n'est pas présent, alors quelques problèmes se posent, p.ex., ouverture d'une session et verrouillage du contexte d'activation pendant une durée indéterminée qui peut être très longue.

D'après cette description, le corps d'exécution d'une activité est aussi associé à un site, à un terminal et à une indication sur la session où les instances de l'activité seront réalisées. Ainsi, au niveau du modèle, il faut inclure les définitions suivantes :

[Execution-Body (0,*), Site-Name (0,1)]
par défaut le site d'exécution est celui d'où provient l'ordre de lancement (cet ordre est implicite pour les activités ayant un mode de lancement automatique)

[6]La valeur des faits du contexte d'activation au moment de la validation de la précondition est stockée dans la base de données et n'est pas modifiable. Dans ce paragraphe, quand on parle de contexte d'activation on se réfère à la valeur actuelle de ses faits

[Execution-Body (0,*), Device (0,1)]
par défaut l'adresse du terminal est celle d'où provient l'ordre de lancement
[Execution-Body (0,*), Session (0,1)]
par défaut une nouvelle session est ouverte
Site-Name, Device : String
un caractère spécial "!" est réservé pour indiquer que le site ou l'adresse du terminal sont ceux d'où provient l'ordre de lancement
Session : ('same-session', 'another-session')

Si l'activité n'a pas de corps d'exécution associé, alors aucune spécification ne doit être faite à la définition de la classe d'activités : ses instances seront lancées dans le site, le terminal et la session d'où provient l'ordre de lancement.

Remarquons que d'autres propositions auraient pu être faites. Par exemple, dans le cas des activités automatiques, laisser le libre choix au système du site où lancer l'exécution ; ceci aurait plusieurs avantages surtout dans un environnement distribué. Pour les activités manuelles, il pourrait être intéressant d'intégrer à l'ordre de lancement le site et l'adresse du terminal ; ceci offrirait plus de flexibilité que la spécification lors de la définition de la classe d'activités. Cette discussion a été menée dans [DOE88a] et elle n'est pas essentielle ici.

Terminaison d'une Activité

La **terminaison normale** d'une activité est effectuée par la primitive "END". Par rapport aux systèmes transactionnels et de verrouillage, les effets de cette primitive sont :

- La confirmation implicite de toutes les transactions.

- Si une session SIB a été ouverte implicitement au lancement de l'exécution de l'activité, alors sa fermeture a lieu à la terminaison de l'activité.

- La libération des verrous acquis par l'activité ou leur propagation à la session où l'activité se déroule. En effet, la primitive "END" permet de spécifier, si on le souhaite le fait de libérer ou propager les verrous.

La **terminaison anormale** d'une activité est effectuée par les primitives "ABORT" ou "FAIL". La sémantique attachée à ces ordres est :

- Si une activité est associée à une précondition de changement, alors, si elle échoue, elle est relancée tout de suite automatiquement par le système.

- Si une activité est associée à une précondition d'état, alors, si elle échoue, elle est relancée tout

de suite par le système si la précondition est toujours validée pour son entité focale, sinon elle est supprimée : son exécution n'a plus de sens à être réalisée.

Par rapport aux systèmes transactionnels et de verrouillage, les effets de ces primitives sont :

- L'annulation implicite de toutes les transactions.
- Si une session SIB a été ouverte implicitement pour l'exécution de l'activité, alors sa fermeture a lieu.
- La libération systématique des verrous acquis par l'activité.

4.7. Etat d'une Classe

L'état d'une classe d'activités indique si elle est chargée ou déchargée. Comme il a été dit dans le chapitre 2 (cf. 2.1.4.2), si une activité est chargée alors sa précondition peut être activée et évaluée, sinon la précondition est au repos, elle n'est pas contrôlée et aucune activité n'est habilitée en conséquence.

Au niveau du modèle de données l'état d'une classe d'activités est défini par les énoncés :

[Activity-Class (0,*), Class-Status (1,1)]
Class-Status : ('Load', 'Unload')

A l'insertion d'une nouvelle classe d'activités, elle est déchargée ("unload") ; pour la charger une primitive spéciale doit être utilisée : la primitive "LOAD". Un exemple suit.

Exemple 4.21 : Charger toutes les classes d'activité dont l'entité focale est Contrat.

```
LOAD    a
FROM    Activity-Class a
WHERE   a.FE-Class = Contrat
```

La primitive "UNLOAD" a le rôle opposé : elle décharge des classes d'activités.

Remarquons que le chargement et le déchargement d'activités est une **opération ensembliste**. Comme nous avons vu dans le chapitre 2, ceci est très important et permet de définir différentes périodes d'exploitation de la base de données.

Maintenant que tous les composants d'une activité sont définis, je présente un autre exemple d'application, basé sur le schéma conceptuel de l'annexe III, illustrant l'utilisation de différents types d'activités ainsi que des primitives de manipulation.

Exemple 4.22 :

Pour faire le suivi de l'exécution d'un contrat, il est necessaire de calculer au début de chaque mois, le montant des dépenses du mois précédent. Dans cet exemple, nous nous intéressons au calcul des dépenses relatives au personnel. Pour simplifier nous supposons qu'il n'existe qu'un seul contrat ESPRIT par entreprise : ainsi, tout le personnel ESPRIT d'une entreprise travaille pour le même contrat.

Le calcul des dépenses du personnel est basé sur le nombre d'heures réalisées par tout le personnel pendant le mois. En fait, chaque fois qu'une fiche personnelle de dépense *(i.e., Fiche-Pers)* est insérée dans la base de données avec le nombre d'heures que l'employé a travaillé pour le contrat pendant le mois, le montant de dépenses du mois relatives au personnel *(i.e. Dép-Mensuelles.Montant-Pers)* doit être incrémenté. Néanmoins l'entreprise exige que ceci ne soit fait qu'au début du mois suivant et uniquement quand tous les employés ont inséré leur fiche personnel avec le nombre d'heures de travail *(i.e. le nombre de Fich-Pers pour le mois doit être égal au nombre de Pers-ESPRIT).*

L'application est modélisée avec trois activités :

- La première activité - "Incrémenter-Montant-Pers" - contrôle le fait qu'une fiche personnelle avec son nombre d'heures soit insérée dans la base de données. Son objetif est d'incrémenter le total des dépenses mensuelles concernant le personnel avec le coût du nombre de ces heures.
  Ainsi, une instance d'activité est créée et habilitée à être exécutée à l'insertion d'une fiche de dépenses avec le nombre d'heures respectif ; néanmoins vu les exigences de l'entreprise, elle est manuelle et ne sera exécutée qu'au début du mois suivant.

- La deuxième activité - "Lancer-Calcul-Montant-Pers" - a comme but de lancer l'exécution des instances de l'activité "Incrémenter-Montant-Pers". Le lancement doit se faire au début du mois quand toutes les fiches personnelles de dépenses sont dans la base de données.
  Cette activité n'a besoin d'être chargée qu'au début de chaque mois et jusqu'à ce que toutes les fiches personnelles soient insérées : à ce moment-là, elle lance alors les activités d'incrémentation et se décharge.

- La troisième activité - "Initier-Calcul-Dép-Mensuelle" - a comme fonction principale de charger la classe d'activités "Lancer-Calcul-Montant-Pers". Il s'agit d'une activité automatique habilitée à être exécutée au changement de mois.

```
(1)   INSERT    Activity-Class
      WITH      Name = "Incrémenter-Montant-Pers"
      AND       FE-Class = Fiche-Pers
      AND       Precondition =
                TEST   INSERT    [Fiche-Pers FE , Nbr-Heures h]
                       FROM      Fiche-Pers FE, Fiche-Pers.Nbr-Heures h
      AND       Mode = 'Manual'
      AND       Execution-Body = { • Ajouter au montant des dépenses du
                                     personnel le coût de FE.Nbr-Heures }

(2)   INSERT    Activity-Class
      WITH      Name = 'Lancer-Calcul-Montant-Pers'
      AND       FE = Dép-Mensuel
      AND       Precondition =
                TEST   STATE     FE
                       FROM      Dép-Mensuel FE
                       WHERE     COUNT (Fiche-Pers WHERE Mois =
                                 Date.Mois-1) = COUNT (Pers-ESPRIT)
      AND       Mode = 'Automatic'
      AND       Execution-Body={START a
                               FROM      Activity a.Activity-Class c
                               WHERE     c.Name = 'Incrémenter-Montant-Pers'
                               AND       a.FE.Mois = Date.Mois - 1

                                • UNLOAD Activity-Class
                                  WHERE  Name = 'Lancer-Calcul-Montant-Pers'
                               }

(3)   INSERT    Activity-Class
      WITH      Name = "Initier-Calcul-Dép-Mensuel"
      AND       FE = Date
      AND       Precondition =
                TEST   INSERT    [Date FE, Mois m]
                       FROM      Date FE, Date.Mois m
      AND       Mode = 'Automatic'
      AND       Execution-Body={ •  INSERT Dép-Mensuel
                                   WITH   Mois = Date.Mois -1
                                   AND    Montant-Pers = 0 ...

                                • LOAD   Activity-Class
                                  WITH   Name = 'Lancer-Calcul-Montant-Pers'
                               }
```

La première activité est un exemple de la combinaison du mode manuel avec une précondition de changement. Souvent, le mode manuel est utilisé pour modéliser des activités dont le programme associé nécessite de l'interaction avec l'utilisateur. Le cas de l'activité (1) est différent : son programme n'a pas besoin de la présence d'un employé de bureau mais le moment approprié pour lancer son exécution ne coïncide pas avec la validation de la précondition et doit être détecté automatiquement.

La deuxième activité est un des exemples de la combinaison du mode automatique avec une précondition d'état. Remarquons que la précondition est assez complexe : plusieurs changements de la base de données doivent être contrôlés pour son activation (p.ex., insertion de Fich-Pers, suppression d'un Pers-ESPRIT, etc.) et de plus deux fonctions d'agrégation sont utilisées. Cette activité illustre aussi très bien l'utilité de la primitive "START".

La troisième activité est une combinaison du mode automatique et d'une précondition de changement. Elle illustre l'importance de la primitive "LOAD" : l'activité "Lancer-Calcul-Montant-Pers" n'est chargée que pendant une courte période : du début de chaque mois jusqu'à ce que il ne manque plus de fiches personnelles à insérer (normalement les employés doivent remplir et insérer leurs fiches vers la fin de chaque mois). Ainsi, la précondition de l'activité (2) qui est complexe et d'évaluation coûteuse ne sera contrôlée que pendant une certaine période. Nous avons donc un avantage au niveau performance mais aussi au niveau conceptuel : spécification de périodes d'exploitation différentes.

Dans le paragraphe 4.6 il a été dit que les faits du contexte d'activation sont verrouillés au lancement d'une activité. Bien sûr ceci ne s'applique pas à des données globales qui subissent un traitement spécial de la part du système, p.ex., le temps. Ainsi, l'entité focale de cette activité et ses associations ne sont pas verrouillées pendant l'exécution de l'activité.

4.8. Conclusion

Le concept d'activité et la façon dont il est modélisé vient d'être présenté de façon complète. L'**aspect couplage** de la notion d'activité au modèle de données est très important. Déjà dans le chapitre 2 (cf. 2.1.4), il a été mis en évidence les avantages de la modélisation des déclencheurs en utilisant les concepts du modèle de données : le projet HIPAC dévéloppé en même temps que DOEOIS suit cette approche.

Dans DOEOIS ce couplage a été mené à terme non seulement par le fait qu'une activité est vue comme une entité (cf. 3.4) mais aussi par la **notion d'entité focale**. L'entité focale joue un rôle primordial dans l'intégration des aspects statiques et dynamiques des applications. Entre autre, elle permet de définir le comportement d'une classe d'entités par les activités qui lui sont appliquées.

Du point de vue conception des applications la notion d'entité focale est donc très importante : ele induit une façon de concevoir particulière. Le fait que cette notion soit dérivée des travaux réalisés dans le domaine des méthodologies de conception valide le rôle qu'elle joue à ce niveau.

La notion d'entité focale est très liée à celle de **précondition** : les préconditions sont validées pour une instance de l'entité focale de la classe d'activités qui leur est associée.

Dans le modèle, deux types de préconditions ont été définis : celles qui testent l'état de l'entité focale et celles qui contrôlent les changements de l'entité focale et ses faits. La distinction entre ces deux types de précondition et leur combinaison avec les modes de lancement d'une activité offrent un support souple pour modéliser les différents types d'activités.

Le fait d'avoir travaillé dans un domaine d'application particulier, la bureautique, a eu plusieurs implications dans la définition des composants des activités. Quelques unes des caractéristiques de ce milieu qui ont influencé la modélisation d'une activité sont : les actions à entreprendre peuvent ne pas être prédéfinies à l'avance, ces actions peuvent être exécutées longtemps après que l'évènement de déclenchement ait eu lieu, et elles nécessitent très souvent de l'interaction avec l'employé de bureau. Ainsi, d'autres problèmes que ceux traités par les systèmes de déclencheurs ont dû être résolus. Ceci a abouti à :

- La définition d'un **corps d'exécution** pas seulement comme une action à réaliser mais aussi par d'autres informations concernant l'exécution même de cette action (site, session SIB, terminal).

- La notion de **contexte d'activation** comme une photographie de la base de données au moment de la validation de la précondition.

- La gestion des différents **états d'une instance d'activité** et la définition d'un ensemble de **primitives de manipulation**. Les liens entre l'exécution de ces primitives et les **systèmes transactionnels et de verrouillage** ont également dû être définis de façon précise.

Un aspect qui n'a pas été traité est l'application des activités à la définition et l'implémentation d'un système de **contrôle d'intégrité sémantique**. Ceci est tout à fait envisageable, néanmoins le modèle d'exécution sous-jacent devrait prendre en compte certains cas qui ont été laissé de côté ; comme par exemple, l'exécution d'activités automatiques au moment où l'évènement de déclenchement a lieu dans la même transaction que la transaction de déclenchement.

CHAPITRE 5

5. MODELISATION DES PROCEDURES

Le concept de procédure modélise le niveau haut d'un support permettant l'automatisation de tâches bureautiques. En effet, ce concept est édifié sur celui d'activité : une procédure est construite à partir d'une "bibliothèque" de classes d'activités en les ordonnant dans un mode approprié pour produire l'effet souhaité.

La structure de ce chapitre est similaire à celle du 4 : le concept de procédure, ses composants et les outils permettant leur gestion sont analysés et discutés.

Nous introduisons tout de suite les définitions standards permettant la définition d'une procédure. Comme pour les activités, les classes ClassesProcédures - "**Procedure-Class**" dans FM - et Procedures - "**Procedure**" dans FM - et leur association d'instanciation (cf. 3.4) ont été définies par les énoncés :

Procedure-Class : Entity
Procedure : Entity
[Procedure-Class (1,1), Procedure (0,*)]

Une classe de procédures a beaucoup de composants communs avec ceux d'une classe d'activités: elle a un **nom**, une **description**, une classe d'**entité focale** et une **précondition**. Ces instances sont créées de la même façon que sont créées les instances d'une classe d'activité. Elle a un **mode de lancement** qui peut être automatique ou manuel et son **état** peut être chargé ou déchargé. Les définitions suivantes sont alors évidentes:

[Procedure-Class (1,1), Name (1,1)]
[Procedure-Class (1,*), Description (0,1) : Text]
[Procedure-Class (1,*), FE-Class (1,1)]
[Procedure (1,*), FE (1,1)]
[Procedure-Class (1,*), Precondition (1,1)]
[Procedure-Class (1,*), Start-Mode (1,1)]
[Procedure-Class (0,*), Class-Status (1,1)]

Ce qui est particulier à une procédure est le fait qu'elle soit aussi définie par un ensemble d'activités. A une procédure est associé un **schéma d'exécution** spécifiant les contraintes de précédence entre les activités ainsi que les liens qui existent entre leur entité focale et celle de la procédure. Ce sujet est développé dans le paragraphe suivant.

Les **états d'une instance** de procédure ne sont pas les mêmes que ceux d'une instance

d'activité. Par exemple, l'exécution d'une procédure peut être suspendue. Ceci est traité dans le paragraphe 5.2.

5.1. Schéma d'Exécution

Le schéma d'exécution d'une procédure doit spécifier :

- Le **nom des classes d'activités** qui accomplissent l'objectif de la procédure.

- La **correspondance entre les entités focales** des activités et celles des procédures. En effet, le passage d'information à l'intérieur d'une procédure se fait par la notion d'entité focale de la procédure : elle est liée dans le schéma conceptuel aux différentes entités focales des activités qui composent la procédure. Par exemple, supposons que l'entité focale d'une procédure soit *Vendeur* et qu'une des activités qui la compose ait comme entité focale *Contrat* : il faut trouver un moyen pour permettre d'exprimer que les instances de cette activité s'intéressent *aux contrats du Vendeur qui est associé à l'instance de la procédure.*

- L'**ordre dans lequel les activités doivent être exécutées**, i.e. spécifier les contraintes de précédence entre les activités. En général quatre structures de contrôle doivent être offertes :
  - Séquence - il doit être possible d'exécuter une série d'activités de manière séquentielle.
  - Parallélisme - il doit être possible qu'un nombre d'activités ou de séries d'activités, soit exécuté en parallèle.
  - Synchronisation - il doit être possible que le lancement d'une activité soit synchronisé avec la terminaison de plusieurs autres activités.
  - Choix - il doit être possible de choisir entre un nombre d'activités ou de séries d'activités.

Un langage - **OPL3** - a été développé [DOE87a] pour pouvoir définir le schéma d'une procédure. Ce langage est basé sur les structures de contrôle des langages de programmation de haut niveau. Il est présenté ici de façon très sommaire à travers quelques exemples. Il offre plusieurs énoncés pour implémenter les quatre structures de contrôle énumérées ci-dessus ; de plus un énoncé permet la spécification explicite de la répétition d'une activité ou série d'activités.

Brièvement, un schéma OPL3 est défini par une liste d'énoncés ; un énoncé est :

- Soit l'implémentation d'une des structures de contrôle offertes. Par exemple, l'énoncé "COBEGIN <liste parallèle d'énoncés> COEND" est une des possibilités proposées pour implémenter le parallélisme.

- Soit la désignation d'une classe d'activités.

La désignation d'une classe d'activités est donnée par son nom, son entité focale et la façon dont cette entité est liée à l'entité focale de la procédure. Par exemple, supposons que l'entité focale d'une

procédure soit *Vendeur* et l'entité focale de l'activité Commission - appartenant à la cette procédure - soit *Contrat de ce Vendeur*. La désignation de cette activité s'écrit :

Commission (Contrat : Vendeur.Contrat)*

De façon générale :

<nom de la classe d'activités> (<classe de l'entité focale de la classe d'activités> : <classe de l'entité focale de la classe de procédures> . <composition de fonctions>)*

<composition de fonctions> spécifie le chemin dans le schéma conceptuel de l'application qui lie les deux classes d'entités focales. Le caractère "*" indique qu'il s'agit de l'entité focale de la procédure.

Pour illustrer ce qui vient d'être dit, un exemple d'une procédure de bureau suit.

Exemple 5.1 :

Dans l'exemple 4.22 nous nous sommes intéréssés au montant des dépenses mensuelles d'un contrat relatives au personnel. Pour faire le suivi mensuel complet d'un contrat il faut aussi prendre en compte les dépenses réalisées dans le cadre de l'équipement et des missions effectuées par les employés.

Cette tâche va être accomplie par la procédure "Récapitulation-Mensuelle". Au début de chaque mois une dépense mensuelle vide est liée à un contrat : à ce moment-là une instance de la procédure "Récapitulation-Mensuelle" est créée pour chaque contrat. Cette procédure va réaliser trois traitements en parallèle : le calcul du total mensuel des dépenses personnelles, des missions et de l'équipement. A la fin de ces trois traitements l'état budgétaire du contrat va pouvoir être réalisé ; suite à quoi un justificatif est produit.

Le calcul des différents types de dépenses se fait selon le même raisonnement : seul celui correspondant aux dépenses personnelles sera plus détaillé ici. Ce traitement est accompli par deux activités qui doivent s'exécuter séquentiellement. La première - "Init-Personnel" - est instanciée quand toutes les fiches de dépenses personnelles ont été insérées dans la base de données (ceci est la condition pour que le calcul puisse se faire : cf. exemple 4.22) ; son corps d'exécution initie à zéro le montant des dépenses personnelles (i.e., *Dép-Mensuelles.Montant-Pers*). La deuxième est l'activité "Incrémenter-Montant-Pers" spécifiée dans l'exemple 4.22. En fait, pour chaque instance de la procédure "Récapitulation-Mensuelle" plusieurs instances d'"Incrémenter-Montant-Pers" vont être exécutées : une pour chaque employé assigné au contrat (i.e., une pour chaque *Fich-Pers*, correspondante au mois en question, insérée dans la base de données).

De la même façon le calcul des dépenses relatives à l'équipement est réalisé par les activités "Init-Equipement" et "Incrémenter-Montant-Equip", et celui concernant les missions, par les activités "Init-Missions" et "Incrémenter-Montant-Miss".

Les activités "Calcul-Etat-Budget" et "Produire-Justificatif" accomplissent les deux dernières tâches de la procédure.

L'énoncé d'insertion de la classe de procédure "Récapitulation-Mensuelle" est alors :

```
INSERT    Procedure-Class
WITH      Name = "Récapitulation-Mensuelle"
AND       FE-Class = Contrat
AND       Precondition =
          TEST   INSERT    Dép-Mensuel
                 WITH      Contrat &
AND       Mode = "Automatic"
AND       Schema =
          BEGIN-SCHEMA
            COBEGIN
                Init-Personnel (Dép-Mensuel : Contrat *.Dép-Mensuel)
                Incrémenter-Montant-Pers (Dép-Mensuelle :
                                          Contrat *.Dép-Mensuel)
              || Init-Equipement (Dép-Mensuel :
                                          Contrat *.Dép-Mensuel)
                Incrémenter-Montant-Equip (Dép-Mensuelle :
                                          Contrat *.Dép-Mensuel)
              || Init-Mission (Dép-Mensuel : Contrat *.Dép-Mensuel)
                Incrémenter-Montant-Miss (Dép-Mensuelle :
                                          Contrat *.Dép-Mensuel)
            COEND
            Calcul-Etat-Budget (Etat : Contrat * . Etat)
            Produire-Justificatif (Justificatif : Contrat * . Justificatif)
          END-SCHEMA
```

La définition de schéma en termes du modèle de données n'a pas encore été donnée. A partir de l'exemple ci-dessus, nous pouvons déduire l'existence de l'association suivante :

[Procedure-Class (1,*), Schema (1,1)]

De plus, les associations suivantes sont définies :

[Procedure-Class (0,*), Activity-Class (1,*)]

[Procedure (0,1), Activity (0,*)]

A l'insertion d'une classe de procédures, l'utilisateur n'a pas besoin de donner les valeurs à l'association qui lie la procédure aux classes d'activités la composant. En effet, à partir de la spécification du schéma, le système peut déduire la valeur de cette association et la rentrer automatiquement dans la base de données. L'intérêt d'introduire ces deux faits est de permettre à l'utilisateur de réaliser, en utilisant le FML, des interrogations sur les activités qui composent une activité.

"Schema" est défini comme un type de base - "OP-Schema" : plus précisément il est un type abstrait appartenant au niveau bas du FM. Comme pour une précondition, sa structure interne résulte de sa compilation et n'est présentée qu'au chapitre 8. Ici, il est uniquement intéressant de dire que la

chaîne de caractères qui représente le schéma fait partie de sa structure interne. La seule opération accessible aux utilisateurs est l'édition du schéma. Par ailleurs, au niveau de l'implémentation, plusieurs opérations sont définies comme par exemple l'évaluation des contraintes de précédence. La définition standard du schéma est :

Schema : OP-Schema

Les activités qui apparaissent dans le schéma d'un énoncé d'insertion d'une procédure doivent déjà exister auparavant. Ces activités ont pû être définies comme autonomes (i.e., ne faisant pas partie d'une procédure) ou non[1]. Le système impose que :

- Une classe d'activités ne peut pas être chargée en tant qu'activité autonome et en tant qu'activité appartenant à une procédure. Ceci parce qu'il ne serait pas possible de décider du contexte dans lequel ces instances doivent être exécutées : dans celui de la procédure ou dans un contexte indépendant. Cette règle est assurée par l'implémentation de la primitive de chargement (LOAD Procedure-Class ; voir ci-dessous).

- Une classe d'activités peut appartenir à plusieurs classes de procédures. Souvent dans ce cas, les contextes dans lequel cette classe d'activités est définie à l'intérieur des différentes classes de procédures sont différents.

Ainsi, il a été établi que **charger une classe de procédures** implique que toutes les activités qui la composent et qui ne sont pas chargées soient chargées. Les activités qui la composent et qui étaient chargées en tant qu'activités autonomes n'existeront plus qu'en fonction de cette procédure.

Décharger une classe de procédures implique que si une activité n'existe qu'en fonction de cette procédure alors elle est déchargée.

Comme pour une activité, l'insertion d'une classe de procédures lui attribue implicitement l'état déchargé.

Un aspect important à considérer est le fait qu'à l'intérieur d'une procédure une classe d'activités puisse avoir plusieurs instances : ceci est le cas de l'activité "Incrémenter-Montant-Pers" appartenant à la procédure "Récapitulation-Mensuelle". En effet, lorsque, pendant l'exécution d'une procédure, on arrive à un point où il faut lancer l'exécution des instances d'une activité, alors toutes ses instances dans l'état habilité sont exécutées.

La description du schéma d'une procédure n'est pas développée plus en détail dans ce document : pour plus d'informations, se reporter à [DOE87a]. La mise en œuvre du gestionnaire de procédures est présentée au chapitre 8 ; ce gestionnaire offre les services de base pour pouvoir assurer les structures de contrôle élémentaires (i.e. séquence, parallélisme, synchronisation et choix). Ainsi, le niveau de ce gestionnaire n'exige pas une présentation plus détaillée des différents énoncés de spécification du schéma.

---

[1]Pour faire cette distinction différents niveaux d'activités sont définis dans le modèle [DOE87a] ; ce détail n'est pas très important ici et il ne sera pas présenté.

5.2. Etats d'une Procédure

Dans le paragraphe 4.6 les états d'une instance d'activité ont été définis comme habilité, lancé, terminé et échoué. Les états d'une instance de procédure diffèrent de ceux-là pour deux raisons :

- La notion d'état lancé est plus subtile. En effet, une procédure qui a été lancée (soit automatiquement soit à la réception de la primitive "START") peut avoir deux états : l'état **actif**, i.e., au moins une des activités qui la composent est en train de s'exécuter (est dans l'état lancé), et l'état **inactif**, i.e., aucune des activités qui la composent n'est lancée.

- Une procédure peut être suspendue, i.e, le lancement de l'exécution des activités qui la composent est interdit temporairement. On dit alors que la procédure est dans l'état **suspendu**. La primitive "SUSPEND" permet le passage des états actif et inactif à l'état suspendu et la primitive "RESUME" a l'effet inverse.

La figure 5.1 présente les différents états d'une procédure. Les primitives "START", "END", "ABORT" et "FAIL" sont utilisées à ce niveau pour lancer, terminer ou faire échouer une activité à l'intérieur d'une procédure ou une procédure elle-même. Leur sémantique est très proche de celle donnée pour les activités autonomes, néanmoins quelques points méritent d'être éclaircis (la figure précédente sera ainsi commentée). Les rapports entre le support des procédures de bureau et les systèmes transactionnels et de verrouillage sont aussi décrits ci-dessous.

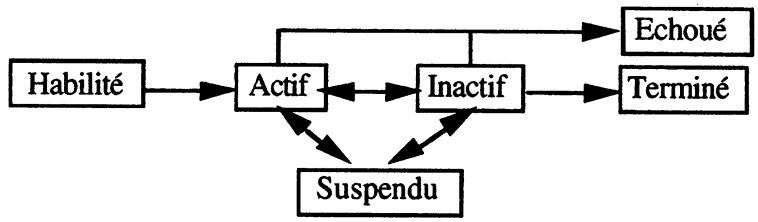


Figure n° 5.1 - Etats d'une Instance de Procédure

Lancement d'une Procédure

Au lancement d'une activité autonome, des verrous sur le contexte d'activation sont acquis et passés à l'activité. Au lancement d'une procédure, il serait aussi souhaitable que des verrous posés sur les faits de la précondition soient acquis. Néanmoins, ceci pose quelques problèmes : une procédure n'existe qu'en tant que collection d'activités, si sa première activité n'est pas habilitée à être exécutée ou si elle est habilitée mais pas lancée (activité manuelle) au moment où la procédure est démarrée, alors les verrous acquis ne peuvent pas être passés.

Pour résoudre ce problème tout schéma de procédure doit commencer par une **activité spéciale** qui est automatique et qui a une précondition évaluée toujours comme vraie. Ainsi, cette activité est toujours habilitée à être exécutée et elle est lancée dès que la procédure est lancée (c'est pour ça qu'une procédure passe de l'état habilité à l'état actif : le passage de l'état habilité à l'état inactif n'a jamais lieu). Les verrous acquis par la procédure à son lancement sont alors propagés directement à cette activité. Le but principal de cette activité est de libérer les verrous à la fin de son exécution.

De façon plus générale, le support des procédures de bureau doit offrir un contrôle persistant sur l'exécution d'une procédure. Entre autre, ceci veut dire que les verrous doivent survivre aux pannes ou plus particulièrement, qu'on doit être capable de passer les verrous d'une activité à une autre. Ces exigences impliquent la nécessité d'un mécanisme de protection qui ne soit pas volatile : basé sur des verrous persistants. Or, ce service n'est pas offert par le SIB.

Pour remédier à cette situation, les verrous acquis par une activité devraient être libérés à la fin de son exécution : en cas de panne, le système peut alors reprendre l'activité sans se soucier de la gestion des verrous perdus. De plus, cette solution aurait l'avantage que les verrous ne soient jamais acquis pendant de longues périodes de temps en attente du lancement d'une activité.

L'utilisateur étant averti de cette situation n'est quand même pas obligé de libérer les verrous à la fin de l'exécution d'une activité (voir les conséquences de la primitive "END" dans le paragraphe 5.6 : elles s'appliquent aussi à une activité interne à une procédure) mais ces verrous ne peuvent être propagés à l'activité suivante que si elle s'exécute dans la même session. Dans ce cas (de propagation), l'utilisateur doit être conscient que dans l'eventualité d'une panne ces verrous sont définitivement perdus.

Sans vouloir justifier le fait que le problème de passage de verrous à travers les activités d'une procédure (plus généralement le manque d'un mécanisme de protection non volatile) n'a pas vraiment été résolu et mérite plus d'attention, il est intéressant de remarquer que dans le milieu bureautique le mécanisme d'autorisation d'accès à la base de données est très important et peut être utilisé artificiellement pour résoudre quelques problèmes soulevés. En effet, dans ce milieu l'information est soit verrouillée pendant de courtes périodes de temps, soit contrôlée pour une longue période par un utilisateur qui accomplit un rôle particulier dans le bureau.

Par exemple, l'activité spéciale qui débute l'exécution d'une procédure peut être utilisée pour donner des droits d'accès (aux informations critiques de la procédure) à l'utilisateur qui est responsable d'accomplir les différentes tâches de l'application et les enlever aux autres utilisateurs.

Par rapport au système transactionnel, il a été établi que, comme pour une activité, une procédure ne peut pas être lancée à l'intérieur d'une transaction. Offrir la possibilté de définir de longues transactions qui engloberaient plusieurs activités n'a pas de sens vu que l'activité est, comme il a déjà été dit (cf. 4.6), l'unité maximale de reprise.

Terminaison d'une Procédure

Terminer une procédure (ainsi que terminer une activité à l'intérieur d'une procédure) normalement a une sémantique equivalente à la terminaison d'une activité autonome (cf. 4.6).

Faire échouer une procédure implique la terminaison anormale de toutes ses activités. De même que pour une activité, si la procédure est associée à une précondition de changement alors elle est immédiatement relancée ; par contre, si elle est associée à une précondition d'état, alors elle n'est relancée que si la précondition est toujours vraie. Les implications du fait de terminer anormalement une procédure sont assez sérieuses et les primitives "ABORT" et "FAIL" doivent être très

soigneusemnt appliquées dans ce contexte. Entre autre, rappelons nous qu'une procédure n'est pas transactionnelle, en conséquence il n'est pas possible de défaire ses effets : en particulier, les effets des activités déjà accomplies ou des transactions confirmées à l'intérieur d'activités dont l'état (au moment de l'annulation) était lancé persisteront.

Par rapport au système de verrouillage, échouer une procédure implique que les verrous acquis par les différentes activités qui étaient dans l'état lancé sont libérés.

Suspension d'une Procédure

La syntaxe des primitives "SUSPEND" et "RESUME" est la même que pour les autres primitives, i.e., "<nom de la primitive> ... FROM ...WHERE ...". A la réception de la primitive de suspension toutes les activités qui étaient lancées doivent se terminer normalement, après quoi la procédure est effectivement suspendue. L'exécution de la primitive "RESUME" a comme effet de continuer l'exécution de la procédure à partir du point où elle a été suspendue.

L'exécution de ces primitives n'a pas de conséquence vis-à-vis des systèmes transactionnels ou de verrouillage vu que leur effet a lieu après la terminaison de toutes les activités qui étaient lancées.

Définitions en Termes du Modèle de Données

Le fait qu'une instance de procédure ait un état, est défini par les énoncés suivants :

```
[Procedure (0,*), Procedure-Status (1,1)]
Procedure-Status : ('Enabled', 'Active', 'Inactive', 'Finished',
                    'Failed', 'Suspended')
```

Les informations sur l'état des procédures sont ainsi maintenues dans la base de données. L'utilisateur peut utiliser le FML pour examiner le déroulement de la procédure. A ce niveau, l'utilité d'une interface graphique est très important. Une première version, assez élémentaire, d'une telle interface a été développée [Hus89] ; elle permet d'une part, la consultation aisée des états des procédures et activités et, d'autre part, de suivre en "temps réel" la vie d'une procédure ou activité.

5.3. Conclusion

Les concepts d'activité et de procédure permettent la modélisation d'un large rang de types de tâches bureautiques. D'un côté, les tâches très peu structurées sont supportées par le système de façon très rudimentaire mais très flexible : utilisation de la notion d'activité automatique ou manuelle et avec ou sans un corps d'exécution associé (une tâche pouvant alors être modélisée par des activités plus ou moins reliées entre elles). D'un autre côté, les tâches plus structurées sont supportées de façon plus complète vu qu'elles sont plus facilement formalisées dans le cadre d'un modèle : utilisation de la notion de procédure de bureau.

Ainsi, un modèle à deux niveaux a été proposé : le niveau activité et le niveau procédure. Le fait que le niveau supérieur (celui des procédures) soit construit à partir des concepts du niveau inférieur est très important : une progression vers le niveau supérieur est offerte vu que les activités définies dans le niveau bas sont compatibles avec celles du niveau haut. Une approche modulaire est alors offerte pour l'aide à la conception des applications.

Cette façon de voir les choses n'est pas la même dans TAXIS où une transition (i.e. une étape d'un "script") ne peut exister en tant que telle : elle est toujours définie pour un "script". De plus, la notion de transition chez TAXIS est moins souple que celle d'activité chez DOEOIS : une transition n'admet que le mode de lancement automatique. Ainsi, par exemple les tâches qui nécessitent la présence d'un employé de bureau sont difficiles à modéliser.

Un autre atout du modéle de procédures a été la notion d'**entité focale**. Cette notion existe aussi dans les deux niveaux : celui des activités et celui des procédures. La modélisation des aspects actifs et dynamiques d'une application revient bien à modéliser l'évolution des entités du schéma conceptuel de l'application.

Par ailleurs, le modèle proposé est assez simplifié et présente quelques **limitations** par rapport aux exigences du milieu bureautique (ces exigences sont réunies dans [DOE88]) :

- Il n'admet pas de **sous-procédures**, i.e., une procédure est un réseau d'activités sans procédures imbriquées. Cette facilité est offerte dans TAXIS. Nous pensons néanmoins que cette possibilité n'est pas des plus essentielles car beaucoup des situations pour lesquelles les sous-procédures sont utiles peuvent être aussi modélisées de façon adéquate par une procédure qui en lance une autre (à travers la primitive "START").

- Il n'offre pas un support capable de **manipuler les exceptions**. Par exemple, il peut être nécessaire qu'un utilisateur priviligié puisse permettre à une instance de procédure de suivre un trajet différent de celui spécifié dans son schéma (p.ex., sauter une étape ou revenir à un état précédent) dans des situations imprévues ou à l'occurrence d'une exception. TAXIS a intégré la manipulation d'exceptions à son modèle au niveau des transitions : de ce point de vue il est plus riche.

- Il n'offre pas de support capable de modéliser et manipuler les concepts d'**acteurs, agents et rôles**. Ces concepts sont pourtant très importants dans le milieu bureautique.

Au niveau du modèle d'exécution, l'utilité d'un **mécanisme de protection non volatile** a été mis en évidence. Ce service n'est pas supporté par le système, néanmoins une solution simple et fiable est proposée (cf. 5.2).

Enfin quelques remarques sur les préconditions des activités et les contraintes de précédence :

- Il a été constaté que les préconditions des activités internes à une procédure sont souvent plus **simples** que celles des activités autonomes. Ceci vient du fait que les contraintes de

précédence sont très puissantes.

- Une contrainte de précédence peut être vue comme une **combinaison d'évènements élémentaires** (voir chapitre 2), ces évènements étant par exemple la fin d'une activité. Remarquons que l'évènement fin d'activité est très puissant vu qu'il résume tout un ensemble de changements de la base de données.

Ainsi, des situations plus complexes sont modélisées par la combinaison des préconditions d'activités et des contraintes de précédence. Ceci est sémantiquement plus clair et riche que de permettre l'écriture de préconditions comme des conjonctions / disjonctions de "sous-préconditions". D'ailleurs quelques inconvénients et limitations de cette dernière approche ont déjà été présentés dans le chapitre 2 (cf. 2.1.1.1).

Conclusion de la partie modélisation de la dynamique (ch. 3, 4 et 5)

Les chapitres 3, 4 et 5 présentent la façon dont les aspects dynamiques sont modélisés, nous profitons pour faire ici une conclusion de cette partie. Les points importants, au niveau conceptuel, du sujet de cette thèse ont été décrits : la modélisation des aspects dynamiques et leur intégration à la modélisation des aspects statiques.

Les deux aspects (statique et dynamique) sont traités de façon uniforme. La figure 6.2 spécifie le traitement d'une des facettes de la vente d'une voiture ; elle synthétise ainsi le modèle proposé et les différents langages et concepts sous-jacents. De façon plus détaillée, elle illustre :

- L'utilisation des mêmes concepts pour modéliser le schéma statique (tableau 1) et dynamique (tableau 2) de l'application. Dans le tableau 2, la plupart des définitions standards permettant la modélisation des concepts d'activité et de procédures sont présentées de façon simplifiée.

- L'utilisation du même langage pour manipuler les entités statiques (tableau 3) et les deux types d'entités dynamiques : activité (tableau 4) et procédure (tableau 5).

- La similarité entre le langage de spécification d'une précondition (précondition du tableau 4) et le FML (p.ex., deuxième énoncé du tableau 3).

- L'utilisation des primitives de manipulation des activités (tableau 6) et des procédures (tableau 7). De même, il est à remarquer la similarité de l'expression d'une primitive (p.ex., deuxième énoncé du tableau 6) et de celle d'un énoncé d'interrogation (p.ex.,<troisième énoncé du tableau 3).

- Les primitives de manipulation des activités/ rocédures illustrent bien comment la combinaison entre les entités statiques et dynamiques est facile grâce au concept d'entité focale (p.ex., troisième énoncé du tableau 6).

SCHEMA CONCEPTUEL

SCHEMA STATIQUE

tab 1

Nom : String
Numéro : Integer
Montant : Integer
Salaire : Integer
Personne : Entity
Client : Personne
Employé : Personne
Vendeur : Employé
Contrat : Entity
Commande : Entity
Vente : Entity
[Personne, Nom]
[Employé, Salaire]
[Contrat, Numéro]
[Vente, Numéro]
[Vente, Montant]
[Vente, Commande]
[Vente, Contrat]
[Vente, Client]
[Vente, Vendeur]

SCHEMA DYNAMIQUE

tab 2

Concept d'activité

Activity-Class : Entity
Activity : Entity
[Activity-Class, Activity]
[Activity-Class, Name]
[Activity-Class, Description]
[Activity-Class, FE-Class]
[Activity-Class, Execution-Body]
[Activity-Class, Start-Mode]
[Activity-Class, Class-Status]
[Activity, Activity-Satus]
[Activity, FE]

Concept de procédure

Procedure-Class : Entity
Procedure: Entity
[Procedure-Class, Activity]
[Procedure-Class, Name]
[Procedure-Class, Description]
[Procedure-Class, FE-Class]
[Procedure-Class, Activity-Class]
[Procedure-Class, Schema]
[Procedure-Class, Start-Mode]
[Procedure-Class, Class-Status]
[Procedure, Activity]
[Procedure, Procedure-Satus]
[Procedure, FE]

FE-Class : VClassDesignation
FE : Personne+Client+Employé+Vendeur+Contrat+Commande+Vente
Precondition : OP-Pcd ; Schema : OP-Schema
Name : String ; Description : Text ; Execution-Body : Entity
Start-Mode : ('Automatic', 'Manual') ; Class-Status : ('Load', 'Unload')
Activity-Status : ('Enabled', 'Started', 'Finished', 'Failed')
Procedure-Status : ('Enabled', 'Active', 'Inactive', 'Finished', 'Failed')

LANGAGE DE MANIPULATION DE DONNEES

tab 3

```
INSERT Vendeur
WITH Nom = 'Viriato'
```

```
INSERT [Vente v1, Vendeur v2]
FROM Vente v1, Vendeur v2
WHERE v1.Numéro = 123
AND v2.Nom = 'Viriato
```

```
SELECT v.Nom
FROM Vendeur v
WHERE v.Salaire > 15000
```

tab 4

```
INSERT  Activity-Class
WITH Name = 'Commission'
AND  Description = 'Cette activi...'
AND  FE-Class = Vendeur
AND  Precondition =  TEST
   INSERT [Vente v1, Vendeur v2]
   FROM Vente v1, Vendeur v2
   WHERE  v1.Montant > 5000
AND ...
```

```
SELECT a.FE-Class
FROM  Activity-Class a
WHERE a.Name = 'Commission'
```

tab 5

```
INSERT Procedure-Class
WITH  Name = 'Vente-Voiture'
AND Description = '...'
AND FE-Class = Vente
AND  Precondition =
      TEST INSERT Vente
AND  Schema =
    BEGIN-SCHEMA
     Détail-Client (Client :
        Vente*.Client)
        Générer.Contrat (Contrat :
        Vente*.Contrat) ...
     END-SCHEMA
AND ...
```

OPCL

tab 6

```
LOAD a
FROM Activity-Class a
WHERE  a.Nom = 'Commission'
```

```
START a
FROM Activity a, Activity-Class b
WHERE b.Name = 'Comiossion'
AND a.FE.Nom = 'Viriato'
```

tab 7

```
UNLOAD p
FROM Procedure-Calss p
WHERE p.FE-Class = 'Vente'
```

```
SUSPEND p
FROM Procedure p.
            Procedure-Class b
WHERE b.Name = 'Vente-Voiture'
AND p.FE.Numéro = 123
```

Figure n° 5.2 - Facette de la Vente d'une Voiture

CHAPITRE 6

6. REGLES DE CORRESPONDANCE ENTRE MISES A JOUR ET PRECONDITIONS

L'évaluation d'une précondition (qu'elle soit de changement ou d'état) n'a de sens que lors de l'exécution d'une mise à jour. Evidemment, une mise à jour ne provoque pas l'évaluation de toutes les préconditions définies dans la base mais uniquement d'un sous-ensemble constitué de celles qui ont un rapport direct avec la mise à jour. Ce rapport est défini par les règles de correspondance entre les préconditions et les mises à jour : en effet, ces règles établissent de façon précise quelles préconditions doivent être évaluées pour une certaine mise à jour. En d'autres termes, nous disons que les règles de correspondance gouvernent l'**activation d'une précondition**.

La méthode de base pour déterminer ces règles est la comparaison d'une part, du type des entités spécifiées dans une mise à jour et le type des entités concernées par la précondition, et d'autre part, du type d'opération de la mise à jour et du type de la précondition (précondition de changement du type insertion d'entités ou insertion de faits, etc.).

Bien que les préconditions soient exprimées en termes d'énoncés de manipulation, leur correspondance avec les mises à jour n'est pas purement syntaxique. Ceci est dû à l'utilisation d'un modèle sémantique et d'un langage de manipulation de haut niveau. Ainsi, établir cette correspondance nécessite de l'inférence au niveau sémantique. Trois aspects sont à prendre en compte :

- La **hiérarchie de généralisation** définie dans le schéma conceptuel d'une application. Une précondition concernant certaines classes d'entités s'applique aussi à d'autres classes construites à partir des premières par des opérateurs ensemblistes. Nous parlons ainsi d'un **"héritage" de préconditions.**
  Il importe alors de déterminer comment cet héritage s'effectue à travers une hiérarchie de généralisation : les différents types de spécialisation (cf. figure 3.1) doivent être pris en compte.

- L'**inclusion de faits**. Jusqu'à ce stade du document les différents exemples du schéma conceptuel ne concernaient (par souci de clarté et de simplicité) que des faits binaires. Néanmoins, la définition des faits donnée dans le paragraphe 3.1 spécifie un fait comme étant une association n-aire entre classes d'entités. Dans ce contexte, l'inclusion de faits définit une sorte de hiérarchie de généralisation pour les faits. L'idée de base est : étant donné un certain fait [A, B, C] alors le fait [A, B] est une généralisation de [A, B, C] et le fait [A, B, C, D] est une spécialisation. Un exemple est l'association existante parmi les entités virtuelles Commande et Article et l'entité littérale (entière) Quantité. Cette association est exprimée par le

fait [Commande, Article, Quantité] ; le fait [Commande, Article] est une généralisation qui définit le lien entre une commande et les articles commandés sans égard aux quantités souhaitées. L'inclusion de faits permet ainsi de faire abstraction de certains détails d'un fait. Bien sûr, ce type d'abstraction doit être pris en compte par les règles de correspondance : une précondition concernant un certain fait doit s'appliquer aussi à d'autres faits construits ou déduits à partir du premier.
Il importe alors de déterminer les règles qui gouvernent l'application d'une précondition concernant un fait à ses généralisations.

- L'**inclusion sémantique d'énoncés de manipulation**. Le fait d'utiliser un langage de manipulation de haut niveau a impliqué que certaines opérations de mise à jour incluent sémantiquement d'autres opérations. Par exemple, une insertion d'entités (qui crée une nouvelle entité et attribue des valeurs à plusieurs de ses faits) inclue sémantiquement plusieurs insertions de faits.
  De même, il faut déterminer précisément le rapport d'inclusion entre les différents types de mise à jour pour pouvoir le prendre en compte dans l'établissement des règles de correspondance.

Une première idée sur ce que sont les règles de correspondance et les différents facteurs dont elles dépendent est maintenant acquise. Pour pouvoir établir ces règles de façon formelle, un travail préalable s'impose, comme par exemple : la définition de manière stricte d'une partie du FML et de son sur-ensemble permettant de spécifier les préconditions ; un ensemble de définitions concernant une hiérarchie de généralisation ; l'inclusion sémantique d'opérations de mise à jour ; etc.

Par souci de compréhension il a été décidé de présenter les règles de correspondance par type de précondition et d'y introduire au fur et à mesure les définitions nécessaires. L'autre solution de présentation était d'introduire de façon systématique d'abord la définition des différents types d'énoncés, celle d'une hiérarchie de généralisation, celle des règles d'inclusion de faits, etc., et seulement après d'établir l'ensemble des règles de correspondance. La première solution semblant plus méthodologique est celle qui a été retenue.

Le paragraphe 6.1 présente les règles de correspondance entre une **mise à jour et une précondition de changement du même type** (p.ex., mise à jour et précondition d'insertion de faits). Pour chaque règle (i.e., pour chaque type de précondition de changement) la présentation se base sur un ensemble d'exemples de correspondance. Ceci permettra de bien comprendre la sémantique des mises à jour et des préconditions : leur définition précise peut alors avoir lieu. De même, les exemples sont utilisés pour illustrer comment les règles de correspondance ont été établies: elles seront alors introduites. Un ensemble de définitions concernant la hiérarchie de généralisation est donné chaque fois que c'est nécessaire.

A la fin du paragraphe 6.1 les différents types d'énoncés sont définis et leur sémantique est bien comprise ; on pourra plus facilement établir les règles données au paragraphe 6.2 concernant des **préconditions de changement et des mises à jour de types différents**.

Finalement le paragraphe 6.3 est dédié aux **préconditions d'état**.

6.1. Précondition et Mise à Jour du Même Type

Ce paragraphe est divisé en six sous-paragraphes : un pour chaque type de précondition de changement.

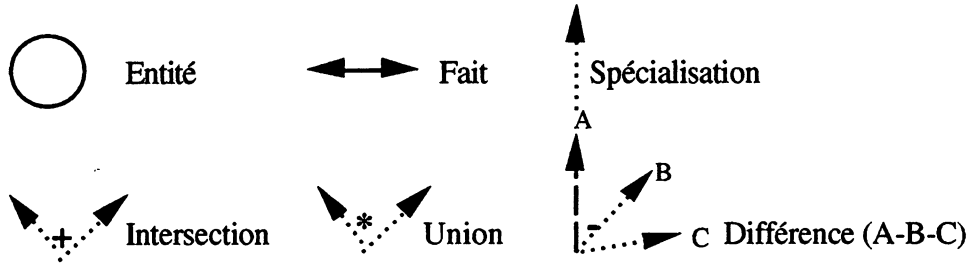
6.1.1. Insertion de Faits

Comme il a été dit, pour présenter chaque règle, un ensemble d'exemples de mises à jour, de préconditions et de leur correspondance est développé. Ces exemples se basent sur le schéma conceptuel suivant qui prend en compte les différents types de spécialisation : spécialisation simple ou composée (i.e., différence, intersection et union).

Département : Entity ;
Véhicule : Entity ;
Personne : Entity ;
Mâle : Personne ;
Employé : Personne ;
Vendeur : Employé ;
Chômeur : Personne - Employé ;
EmpMâle : Employé * Mâle ;
Cadre : Employé + Chômeur ;
Emprunteur : Employé + Département ;
Camion : Véhicule ;
[Personne, Véhicule] ;

Ce schéma conceptuel est illustré dans la figure 6.1. La légende utilisée dans cette figure est la suivante.

Légende des diagrammes des schémas conceptuels



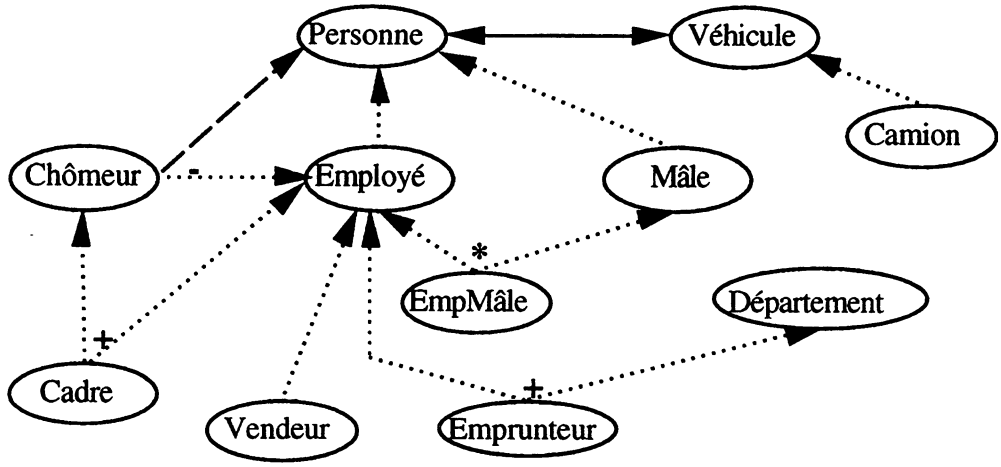


Figure n° 6.1 - Exemple de Schéma Conceptuel (Hiérarchie de Généralisation)

La figure 6.2 présente des préconditions et des mises à jour d'insertion de faits et leur correspondance : elle est commentée ci-dessous.

Ce sont le type d'opération d'une mise à jour et le type de la précondition, ainsi que les classes d'entités concernées par celles-ci qui permettent d'établir la correspondance entre une mise à jour et une précondition. Ainsi, les règles de correspondance sont établies à partir de la clause "TEST <Opération>" d'une précondition et de la clause "<Opération>" d'une mise à jour, ainsi que de leur clause "FROM" (où les types des variables de la première clause sont définis). C'est pourquoi dans les exemples de la figure 6.2 les énoncés ne comprennent pas de clause "WHERE". En effet, cette dernière clause contient les conditions à vérifier au cas où une précondition a été sélectionnée pour être évaluée pour une certaine mise à jour.

Analysons la **précondition p1**. Ses clauses "TEST <Opération>" et "FROM" indiquent que chaque fois qu'un fait est créé entre une personne et un véhicule (i.e., une instance du fait [Personne: Personne, Véhicule : Véhicule] est créée) et que de plus cette personne est un employé alors la précondition doit être évaluée.

En termes fonctionnels p1 doit être évaluée quand la fonction Personne (Véhicule) reçoit comme valeur un employé et la fonction Véhicule (Employé) reçoit comme valeur un véhicule. Remarquons que la fonction Véhicule (Personne) est héritée par les classes spécialisées de Personne (cf. figure 3.1) : il est alors possible de parler de la fonction **Véhicule (Employé)** et l'énoncé p1 est correct.

Remarquons que l'entité focale n'est pas spécifiée dans la précondition p1. En effet, ce sont les différentes entités concernées par la précondition (i.e., les entités de la clause "TEST <Opération>" ainsi que celles de la clause "FROM" qui précisent le type des premières) qui sont prises en compte dans les règles de correspondance : l'entité focale n'y joue pas de rôle particulier.

Les mises à jour (identifiées par *mi* dans la figure) créent des instances du fait [Personne, Véhicule] ; leur clause "FROM" impose des restrictions sur le type des personnes qui seront associées à des véhicules. Ces restrictions portent sur la hiérarchie de généralisation (p.ex., la personne est un employé ou un chômeur). Nous allons étudier la correspondance entre p1 et m1,

..., m8. Ceci permet de comprendre comment une précondition d'insertion de faits s'applique à travers une hiérarchie de généralisation.

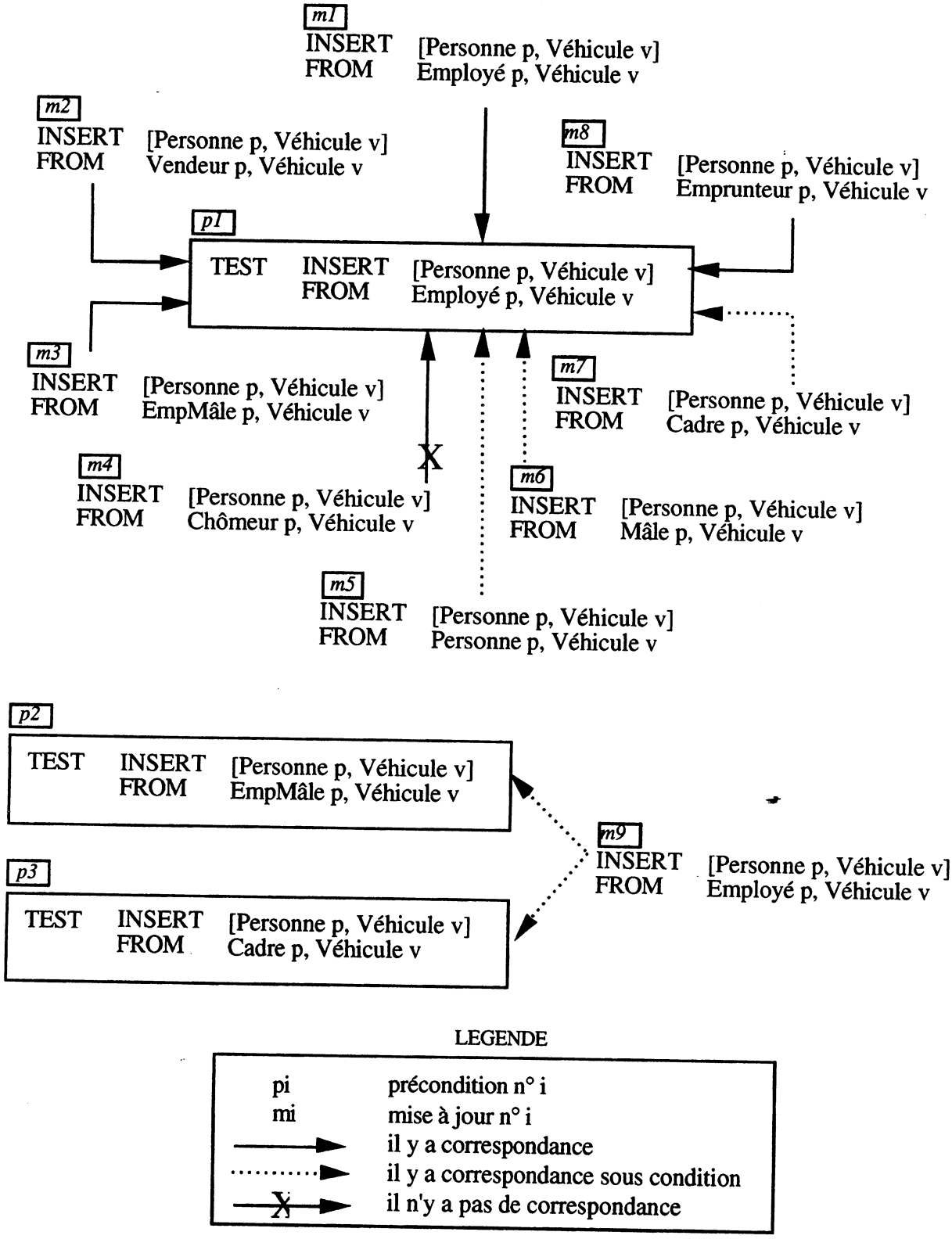


Figure n° 6.2 - Exemples de Correspondance : Précondition et Mise à Jour du Type Insertion de Faits

La **mise à jour m1** a précisément la même syntaxe que la précondition p1. Sa sémantique indique qu'un (ou plusieurs) fait est créé entre une personne qui est aussi un employé et un véhicule. Dans ce cas, il est évident que p1 doit être évaluée, nous disons alors qu'*il y a correspondance* entre p1 et m1.

La **mise à jour m2** impose que les personnes qui seront associées à des véhicules soient forcément des vendeurs (de par la clause "FROM"). Or, tout Vendeur est aussi Employé : en conséquence, *il y a correspondance* entre p1 et m2.

En effet, nous pouvons conclure que la précondition p1 doit être évaluée chaque fois qu'une personne qui est un employé ou une **spécialisation simple** d'employé est associée à un véhicule. En termes fonctionnels p1 doit être évaluée chaque fois que la fonction Véhicule appliquée à un employé ou à une spécialisation simple d'un employé reçoit une valeur et vice-versa que la fonction Personne (Véhicule) reçoit comme valeur un employé ou une spécialisation simple d'employé.

La **mise à jour m3** implique que les personnes qui seront associées à des véhicules soient des EmpMâle. Un EmpMâle est une personne qui est en même temps un employé et un mâle. Ainsi, comme conséquence de cette mise à jour, une valeur est attribuée à la fonction Véhicule appliquée à EmpMâle qui est un employé et la fonction Personne (Véhicule) reçoit comme valeur un EmpMâle : *il y a correspondance* entre p1 et m3.

Nous pouvons donc ajouter que non seulement p1 doit être évaluée quand une personne, qui est un employé ou une spécialisation d'employé est associée à un véhicule mais aussi quand il s'agit d'une **spécialisation par intersection** (EmpMâle est dit une spécialisation par intersection d'Employé ou de Mâle).

La **mise à jour m4** associe des personnes qui sont des chômeurs à des véhicules. Or, un Chômeur est défini comme la **différence** entre Personne et Employé (i.e., Chômeur est une personne qui n'est pas un employé). L'exécution de cette mise à jour n'attribue aucune valeur à la fonction Véhicule (Employé) : ceci suffit pour conclure qu'*il n'y a pas de correspondance* entre p1 et m4.

La **mise à jour m5** ne donne pas d'informations supplémentaires sur les personnes qui sont associées à des véhicules par l'insertion d'instances du fait [Personne, Véhicule]. Or, une personne est une **généralisation** d'Employé : ainsi elle peut être un employé comme ne pas l'être. En conséquence, d'après cet énoncé, nous ne pouvons pas savoir si la fonction Véhicule (Employé) va recevoir une valeur lors de l'exécution de la mise à jour. Il faut consulter la base de données pour savoir si la personne est un employé - et en conséquence évaluer la précondition p1 - ou non. Ainsi, statiquement nous ne pouvons pas conclure sur la correspondance entre p1 et m5, la précondition p1 est néanmoins sélectionnée pour être traitée au moment de l'exécution proprement dite de la mise à jour où tous les accès à la base de données sont réunis. Dans ce cas, nous disons qu'*il y a correspondance sous condition* entre p1 et m5.

La **mise à jour m6** va associer des mâles à des véhicules. Les mâles sont des personnes et par

conséquent peuvent être des employés mais ils peuvent ne pas l'être aussi. Nous nous trouvons donc dans la même situation que pour m5 : *il y a correspondance sous condition* entre p1 et m6.

Par rapport à la mise à jour m5 nous avons conclu que la correspondance sous condition était établie pour les généralisations d'Employé. Or, d'après m6 ceci est vrai pour les **spécialisations simples des généralisations d'Employé**.

La **mise à jour m7** associe des cadres à des véhicules. Un cadre est défini par **union** et peut être un employé ou un chômeur (i.e., une personne qui n'est pas un employé). Ainsi, statiquement nous ne pouvons pas conclure sur la correspondance entre p1 et m7 : il est nécessaire de consulter la base de données. Nous sommes donc dans une situation comparable à celle de m5 : *il y a correspondance sous condition* entre p1 et m7.

La **mise à jour m8** associe des emprunteurs à des véhicules. Un Emprunteur, tout comme un Cadre, est construit à partir d'Employé par union avec une autre entité virtuelle. On serait donc tentés de conclure qu'il y a correspondance sous condition. Néanmoins par la sémantique de l'énoncé m8 nous pouvons être sûrs que l'emprunteur est un employé et non un département[1]. En effet, l'objectif de la mise à jour m8 est d'associer un **emprunteur dans le rôle de personne** à un véhicule. Or, pour qu'un emprunteur puisse jouer un tel rôle il doit être un employé et non un département (rappel sur l'héritage des fonctions par les entités construites par union : figure 3.1). Nous pouvons donc affirmer qu'*il y a correspondance* entre p1 et m8.

Ce cas est assez particulier : le fait de savoir à partir de quelles entités Emprunteur est construit par union et de prendre en compte le rôle de la classe référencée dans le fait à mettre à jour a permis d'affiner notre conclusion sur la correspondance. En efet, cette classe (Emprunteur) est construite par union d'entités qui sont soit des employés, soit des entités qui **n'appartiennent pas à la hiérarchie de généralisation** qui englobe Employé ; et donc qui ne peuvent pas jouer le rôle de personne dans le fait.

Ainsi, on peut conclure que quand l'entité spécifiée dans la mise à jour est construite par **union** à partir d'employé, **plusieurs cas sont à considérer** dépendant des classes qui constituent les différents termes de l'union.

Remarquons aussi que cette mise à jour (m8) est peu naturelle. En effet, l'ampleur de l'association entre Personne et Véhicule n'englobe pas Département. Ainsi, appliquer les fonctions qui découlent de cette association à emprunteur peut paraître peu naturel.

La **précondition p2** porte sur la mise à jour du même fait que la précondition p1 mais elle ne doit être évaluée que quand la fonction Personne (Véhicule) reçoit comme valeur un EmpMâle et la fonction Véhicule (EmpMâle) reçoit comme valeur Véhicule. Cette précondition va illustrer le cas où l'entité qui joue le rôle de personne (i.e. EmpMâle) est construite à partir de Personne par intersection.

---

[1]Cette remarque est faite par rapport à la sémantique de la mise à jour. Il n'est pas garanti que la partie qualification de m8 ne concerne pas les départements au lieu des employés. Dans ce cas, aucune erreur ne sera détecté à la compilation de m8. Par contre, son exécution provoquera une erreur : en conséquence, m8 n'est pas exécutée et les préconditions correspondantes ne sont pas évaluées.

La **mise à jour m9** lie des personnes à des véhicules, ces personnes devant être des employés. Un Employé peut ne pas être un Mâle, ainsi il n'est pas sûr que la précondition p2 doive être évaluée pour la mise à jour m9 : il faut consulter la base de données pour en savoir plus. Statiquement nous ne pouvons pas conclure qu'il y a correspondance entre p2 et m9.

En effet, ce cas est semblable à p1 et m5 : Employé est considéré comme une généralisation de EmpMâle et le comportement de m9 par rapport à p2 est le même que celui de m5 (où Personne est une généralisation d'Employé) par rapport à p1 : *il y a correspondance sous condition* entre p2 et m9.

La **précondition p3** illustre le cas où l'entité qui joue le rôle de Personne (i.e. cadre) est construite à partir de Personne par union. Un employé peut être un cadre comme ne pas l'être, ainsi *il y a correspondance sous condition* entre p3 et m9.

Il n'est pas correct de considérer Employé comme une généralisation de Cadre. Mais Employé peut être considéré comme une généralisation des cadres qui peuvent jouer le rôle de personne dans le fait [Personne, Véhicule]. C'est-à-dire qu'Employé peut être considéré comme une généralisation des employés cadres. Ceci justifie que la correspondance entre p3 et m9 aît le même comportement que celle entre p1 et m5.

Remarquons que si la clause "FROM" de p3 imposait que les personnes à associer aux véhicules soient des emprunteurs ("FROM Emprunteur p ...") alors il y aurait aussi correspondance sous condition : Employé serait alors considéré comme une généralisation des employés emprunteurs.

Nous venons d'étudier la correspondance entre des préconditions et des mises à jour qui concernent l'insertion d'un même fait ([Personne, Véhicule]). Les énoncés des préconditions et mises à jour imposent des restrictions (spécifiées dans leur clause "FROM") sur une des classes d'entités (Personne) qui participent au fait. Ces restrictions portent sur la hiérarchie de généralisation, elles indiquent que seul un sous-ensemble de cette classe (Personne) est concerné par l'énoncé : le sous-ensemble doit être une classe spécialisée (spécialisation simple ou composée) de la classe participant au fait. Précisément, les exemples ont étudié le comportement des règles (i.e., il y a correspondance, il y a correspondance sous condition ou il n'y a pas de correspondance) vis-à-vis du rapport hiérarchique existant entre la (sous) classe spécifiée dans l'énoncé de la mise à jour (p.ex., Vendeur pour m2) et la (sous) classe correspondante spécifiée dans la précondition (p.ex., Employé pour p1). D'après les résultats, nous pouvons conclure que :

1. Si **toutes** les entités de la classe de mise à jour appartiennent aussi à la classe de la précondition[2] alors il y a correspondance.
   Dans ce cas, la terminologie utilisée est la suivante : les entités de la classe de la mise à jour **sont sûrement** des entités de la classe de la précondition. Par exemple, pour m2 et p1 nous disons *"Vendeur est sûrement un Employé"*.
   Ici sont englobés les cas suivants : la classe de la mise à jour est la même (cf. m1), est une spécialisation (cf. m2) ou est une spécialisation par intersection (cf. m3) de la classe de la précondition (cf. p1).

[2]Cette affirmation, ainsi que celles qui suivent dans 2., 3., et 4, est faite de façon statique, i.e. d'après le schéma conceptuel, sans consulter la base de données.

Un dernier cas qui n'a pas été étudié dans les exemples est à ajouter : la classe de la mise à jour est construite par union de plusieurs classes dont les entités sont sûrement des entités de la classe de la précondition.

2. Si seules **quelques** entités de la classe de la mise à jour sont susceptibles d'appartenir à la classe de la précondition alors il y a correspondance sous condition.
   Nous disons alors que les entités de la classe de la mise à jour **sont probablement** des entités de la classe de la précondition (vue positiviste). Par exemple, pour m5 et p1 nous disons *"Personne est probablement un Employé"*.
   Ici sont englobés les cas suivants : la classe de la mise à jour est une généralisation (cf. m5) ou une spécialisation d'une généralisation (cf. m6) de la classe de la précondition (cf. p1); quelques cas (ils seront définis plus tard) où la classe de la mise à jour est construite par union (cf. m7) ; la classe de la précondition est construite par union (cf. m9 et p2) ou par intersection (cf. m9 et p3) à partir de la classe de la mise à jour.
   Un dernier cas qui n'a pas été considéré est à ajouter : la classe de la mise à jour est construite par union de classes dont une au moins a des entités qui sont probablement des entités de la classe de la précondition.

3. Si **aucune** des entités de la classe de la mise à jour n'appartient à la sous-classe de la précondition alors il n'y a pas de correspondance.
   La terminologie utilisée est : les entités de la classe de mise à jour **sont différentes** des entités de la classe de la précondition. Par exemple, pour m4 et p1 nous disons *"Chômeur est différent d'Employé"*.
   Ici est englobé le cas où la classe de la mise à jour est construite par différence (cf. m4) à partir de la classe de la précondition (cf. p1).

4. Si seules **quelques** entités de la classe de la mise à jour sont susceptibles d'appartenir à la classe de la précondition **mais**, si à partir du rôle que ces entités jouent dans la mise à jour, on peut **déduire** qu'elles appartiennent à la classe de la précondition, alors il y a correspondance.
   La terminologie utilisée est : les entités de la mise à jour **sont par déduction** des entités de la classe de la précondition. Par exemple, pour m8 et p1 nous disons *"Emprunteur est par déduction un Employé"*.
   Ici est pris en compte le cas où la classe de la mise à jour est construite par union d'au moins une classe dont toutes les entités sont sûrement des entités de la classe de la précondition et d'autres classes dont les entités n'appartiennent pas à la hiérarchie de généralisation qui englobe la classe de la précondition (cf. m8 et p1).

Les concepts qui viennent d'être introduits (les entités d'une classe sont sûrement, sont probablement, sont différents ou sont par déduction des entités d'une autre classe) de façon intuitive sont définis par rapport aux classes d'entités qui participent à un fait et aux spécialisations de ces classes. Dans ce qui suit, la définition du contexte dans lequel ces concepts sont établis et des concepts eux mêmes est présentée.

Définition des Concepts Introduits

L'objectif de ce paragraphe est l'étude des règles de correspondance qui concernent les préconditions et les mises à jour d'insertion de faits. Supposons le fait suivant :

[F1: E1, ..., Fi : Ei, ..., Fn : En]

où Ei, $1 \leq i \leq n$ est une classe d'entités et Fi, $1 \leq i \leq n$ est le nom d'une fonction définie par le fait (cf. 3.2). Ces fonctions vont être héritées à travers la hiérarchie de généralisation. Par exemple, le fait [Personne, Véhicule] définit la fonction Véhicule sur Personne : Véhicule (Personne). Cette fonction est héritée, entre autre, par Employé ; on parle alors de la fonction Véhicule (Employé).

Une mise à jour de ce fait modifie la valeur des fonctions sur les entités liées directement par le fait (p.ex., Véhicule (Personne)) ou la valeur de ces fonctions sur des sous-classes des classes liées par le fait (p.ex., Véhicule (Employé)).

Ainsi, la forme d'une insertion d'instances d'un fait est la suivante :

INSERT [F1 v1, ..., Fi vi, ..., Fn vn]
FROM A1 v1, ..., Ai vi,..., An vn

vi, $1 \leq i \leq n$, est une variable qui est définie dans la clause FROM. Sa définition consiste à associer la variable à une classe d'entités Ai (dans laquelle la variable prendra ses valeurs). Ai, $1 \leq i \leq n$, est : soit la classe Ei, soit une classe d'entités spécialisée (spécialisation simple ou composée) de Ei qui par conséquence a hérité des fonctions définies sur Ei par le fait [F1 : E1, ..., Fn : En].

Voyons de façon plus formelle quelles sont les classes d'entités qui peuvent être des Ai. Comme il vient d'être affirmé, un Ai doit appartenir à l'ensemble de classes qui héritent des fonctions définies sur Ei par le fait [F1 : E1, ..., Fn : En]. Cet ensemble est noté :

Hiér-Ei

Nous disons d'une classe C appartenant à Hiér-Ei que :

C est un Hiér-Ei

Une classe d'entités C quelconque est un Hiér-Ei si une des conditions suivantes est remplie:

(i) $C \equiv Ei$
(ii) C est une spécialisation simple d'un Hiér-Ei
(iii) C est construit par intersection d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante :

$\exists i>1 \mid B_i$ est un Hiér-Ei

(iv) C est construit par union d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante:

$\exists i>1 \mid B_i$ est un Hiér-Ei

(v) C est construit par différence à partir de la classe D et d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante :

$C = D - B_1 - B_2 - \ldots - B_n$
où D est un Hiér-Ei
et $\forall i \geq 1 \mid B_i$ est un Hiér-Ei (évidemment)

Commentaire sur la condition (iv):

Supposons le schéma conceptuel : E1, E2, E11 : Entity ; E3 : E1+ E2, [E1, E11]. E3 est composée soit par des entités de E1 et dans ce cas il hérite la fonction E11(E1), soit par des

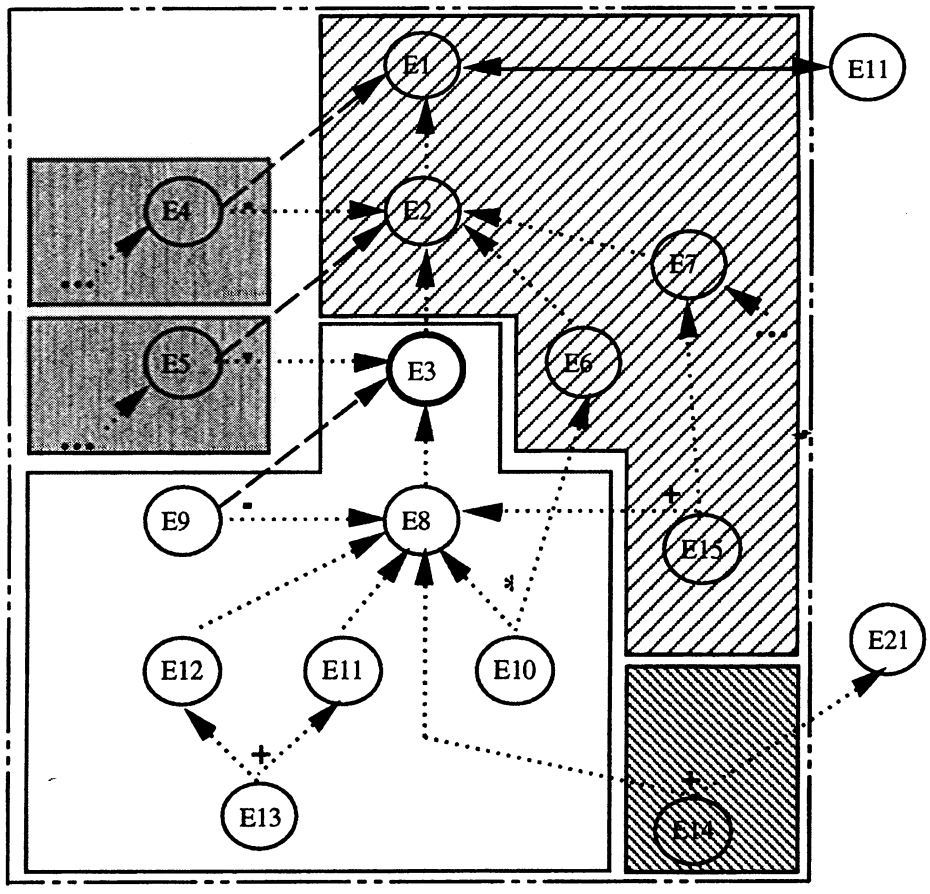
entités de E2 et dans ce cas il n'hérite pas des fonctions définies par le fait [E1, E11]. Nous considérons malgré tout E3 comme étant un Hiér E1.

L'ensemble Hiér-Ei jouit d'une PROPRIETE très importante :

En prenant une classe A quelconque qui appartient à Hiér-Ei, cet ensemble peut être partitionné en quatre sous-ensembles disjoints :

- Les classes dont les entités sont sûrement des entités de A
- Les classes dont les entités sont probablement des entités de A
- Les classes dont les entités sont différentes des entités de A
- Les classes dont les entités sont par déduction des entités de A

Dans ce contexte les quatre concepts introduits de façon intuitive jusqu'ici peuvent être définis.La figure 6.3 donne un exemple de schéma conceptuel où un fait entre les classes E1 et E11 est défini. L'ensemble Hiér-E1 y est mis en évidence. Cet ensemble est partitionné en quatre groupes en fonction de la classe E3 qui est une spécialisation simple de E1. Les figures 6.4, 6.5 et 6.6 illustrent le même phénomène mais en prenant en compte d'autres cas plus pointus.



Légende pour cette figure et les suivantes :

Figure n°6.3 - Qui est un E3

Sûr-A

Le sous-ensemble des classes qui appartiennent à Hiér-Ei et dont les entités qui les instancient sont **sûrement** des entités instanciant une autre classe A appartenant aussi à Hiér-Ei est appelé :

Sûr-A

Nous disons d'une classe C appartenant à Sûr-A que :

C est un Sûr-A

C est un Sûr-A si une des conditions suivantes est remplie:

(i) $C \equiv A$

(ii) C est une spécialisation simple d'un Sûr-A

(iii) C est construit par intersection d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante :

$\exists i \geq 1 \mid B_i$ est un Sûr-A

(iv) C est construit par union d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante:

$\forall i \geq 1 \mid B_i$ est un Sûr-A

(v) C est construit par différence à partir de la classe D et d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante :

$C : D - B_1 - B_2 - \ldots - B_n$

où D est un Sûr-A

Commentaire sur la condition (iii):

Remarquons que l'intersection d'un ensemble de classes dont une au moins est un Sûr-A et une autre est un Diff-A ("différent de A" voir définition ci-dessous), est l'ensemble vide.

Remarquons aussi que l'intersection d'une classe qui est un Hiér-Ei avec une classe qui n'est pas un Hiér-Ei n'a pas de sens dans le contexte qui nous intéresse.

Diff-A

Le sous-ensemble des classes qui appartiennent à Hiér-Ei et dont les entités qui les instancient sont **différentes** des entités instanciant une autre classe A appartenant aussi à Hiér-Ei est appelé :

Diff-A

Nous disons d'une classe C appartenant à Diff-A que :

C est un Diff-A

Les classes qui sont des Diff-A sont :

- soit construites par différence par rapport à A ou à des classes à partir desquelles A est construite. Ce sous-ensemble de classes qui sont des Diff-A est nommé Diff*-A.
- soit construites à partir de classes qui sont des Diff*-A en respectant un ensemble de règles.

Tout d'abord l'ensemble Diff*-A va être défini de façon récursive. Cette définition est illustrée par la figure 6.4 qui prend en compte les différents cas traités et donne un exemple de la puissance de l'application de la récursivité (cf. fig 6.4(d)). Ensuite, les règles pour qu'une classe C quelconque soit un Diff-A sont présentées.

Définition de Diff*-A :

Diff*-A = Différent-A $\cup$ Diff*-Ascendant-A

Diff*-($\cap_{1\leq i\leq n} A_i$) = ($\cup_{1\leq i\leq n}$ Diff*-A_i) *cf. figure 6.4(c)*

Diff*-($\cup_{1\leq i\leq n} A_i$) = ($\cap_{1\leq i\leq n}$ Diff*-A_i) *cf. figure 6.4(b)*

Diff*-$\varnothing = \varnothing$

Une classe C quelconque est un Différent-A ssi : *cf. figure 6.4(a)*

$\exists X$ est une classe $\mid$ (C : X - A - B_1 - ... $B_n \vee$ A : X - C - B_1 - ... - B_n)

où $\forall 1\leq i\leq n \mid B_i$ est une classe d'entités

Définition de Ascendant-A :

Ascendant de A est formé par les classes qui ***appartiennent à Hiér-Ei*** *et à partir desquelles A est construit.*

- si A est du type A : Entity

  Ascendant-A = $\varnothing$
- si A est du type A : C

  alors si C est un Hiér-Ei, Ascendant-A = C

  si C n'est pas un Hiér-Ei, Ascendant-A = $\varnothing$
- si A est du type A : C - B_1 - ... - Bn où B_i, $1\leq i\leq n$, est une classe d'entités

  alors si C est un Hiér-Ei, Ascendant-A = C

  si C n'est pas un Hiér-Ei, Ascendant-A = $\varnothing$
- si A : C_1 * C_2 * ... * Cn où C_i, $1\leq i\leq n$, est une classe

  Ascendant-A = «$_{1\leq i\leq m} D_i$ où $\{C_i\}_{1\leq i\leq n} \supseteq \{D_i\}_{1\leq i\leq m}$ et D_i est un Hiér-Ei
- si A est du type A : $C_1 + C_2 + \ldots + C_n$ où C_i, $1\leq i\leq n$, est une classe

  Ascendant-A = »$_{1\leq i\leq m} D_i$ où $\{C_i\}_{1\leq i\leq n} \supseteq \{D_i\}_{1\leq i\leq m}$ et D_i est un Hiér-Ei

C est un Diff-A si une des conditions suivantes est remplie:

(i) C est Diff*-A

(ii) C est une spécialisation simple d'un Diff-A

(iii) C est construit par intersection d'un ensemble de classes B_i, $1\leq i\leq n$, en respectant la règle suivante :

$\exists i\geq 1 \mid B_i$ est un Diff-A

(iv) C est construit par union d'un ensemble de classes B_i, $1\leq i\leq n$, en respectant la règle suivante:

$\forall i\geq 1 \mid (B_i$ est un Diff-A $\vee$ NON(B_i est un Hiér-Ei))

(v) C est construit par différence à partir de la classe D et d'un ensemble de classes B_i, $1\leq i\leq n$, en

respectant la règle suivante :

$C : D - B_1 - B_2 - \ldots - B_n$

où D est un Diff-A

Commentaire sur la condition (iii):

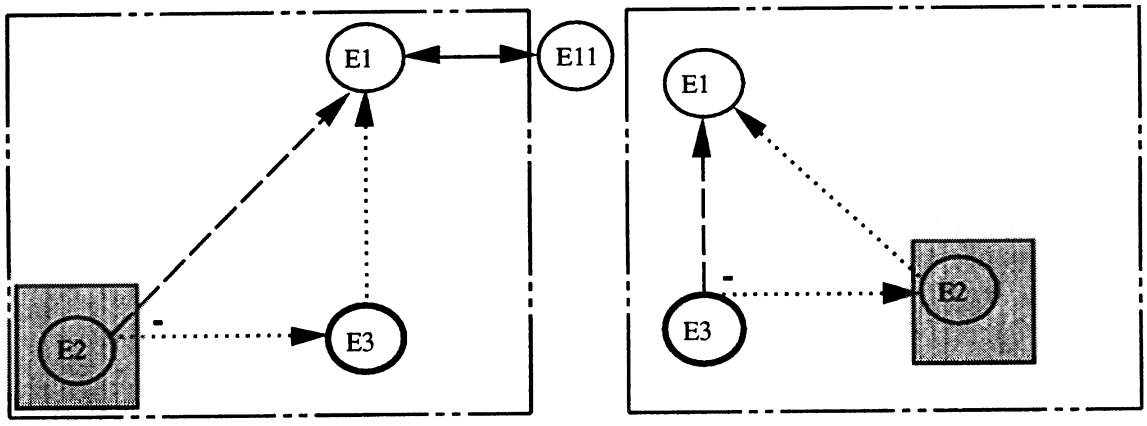
En conséquence de cette condition, nous avons que

$\forall\, j \neq i \mid$ (Bj est un Diff-A $\vee$ Bj est un Prob-A)

Il a déjà été remarqué que l'intersection d'un Diff-A avec un Sûr-A est vide, de même l'intersection d'un Diff-A avec un Déd-A est vide. Par conséquent, l'intersection d'un Diff-A avec une classe qui n'est pas un Hiér-Ei n'a pas de sens dans le contexte qui nous intéresse.

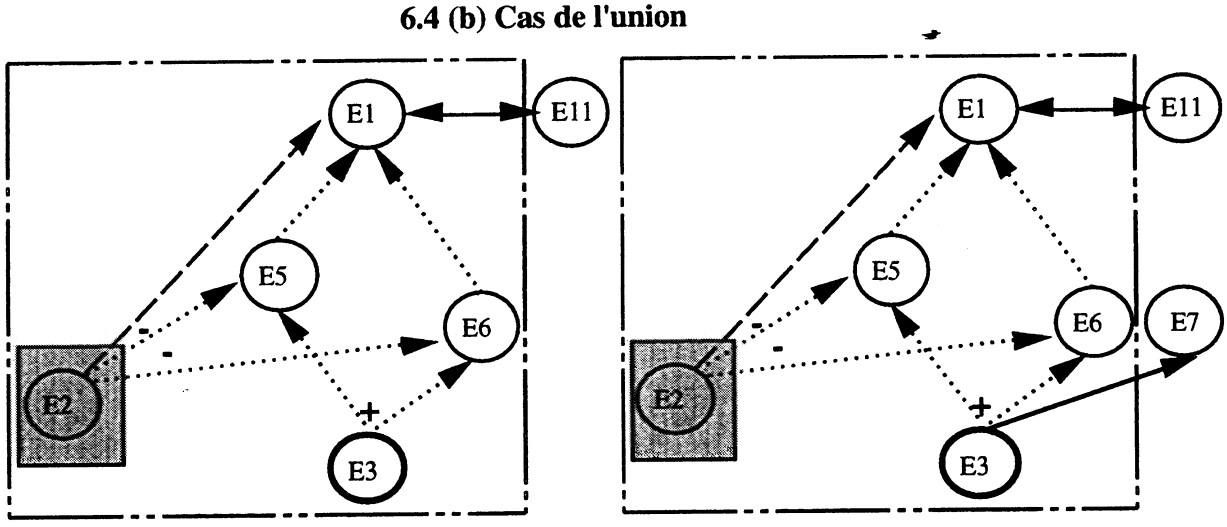
Figure n° 6.4 - Diff*-E3

6.4 (a) Différent-E3



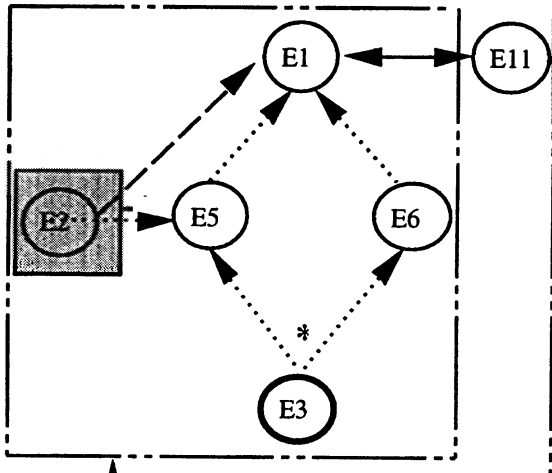
E2 est un Différent-E3

6.4 (b) Cas de l'union



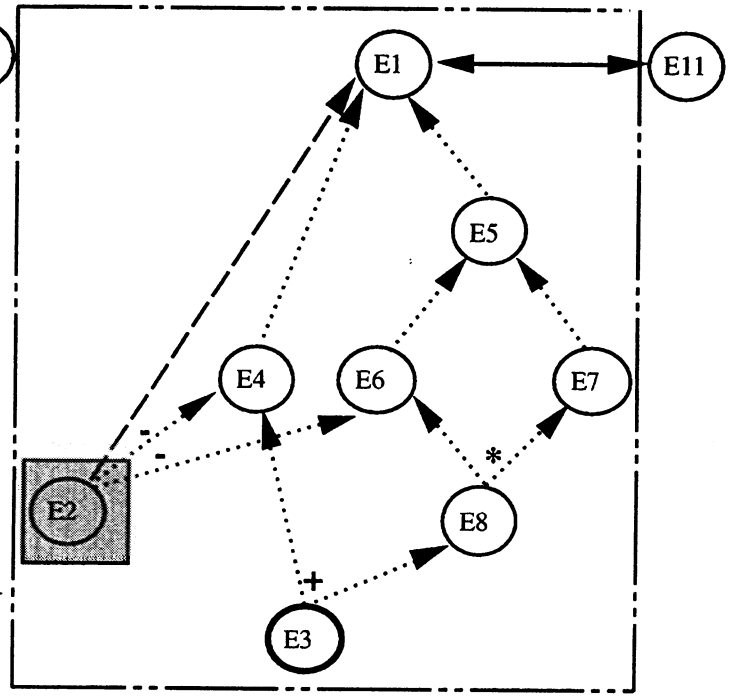
E2 est un Diff*-E3
car il est Diff*-E5 et Diff*-E6,
et E7 n'est pas un Hiér-E1

6.4 (c) Cas de l'intersection



E2 est un Diff*-E3
car il est un Diff*-E5

6.4 (d) Application de la récursivité



E2 est un Diff*-E3
car il est un Diff*-E4
et un Diff*-E8 (comme
conséquence d'être un
Diff*-E6)

Prob-A

Le sous-ensemble des classes qui appartiennent à Hiér-Ei et dont les entités qui les instancient sont **probablement** des entités instanciant une autre classe A appartenant aussi à Hiér-Ei est appelé:

Prob-A

Nous disons d'une classe C appartenant à Prob-A que :

C est un Prob-A

Les classes qui sont des Prob-A sont :

a. soit des classes à partir desquelles A est construit. Ces classes appartiennent à un sous-ensemble nommé Ancêtre-A.

b. soit des classes construites à partir des classes définies en (a) ou à partir de A.

Tout d'abord l'ensemble Ancêtre-A va être défini. Ensuite les règles pourqu'une classe C quelconque soit un Prob-A sont présentées.

La définition de Ancêtre-A est faite hors du contexte de Hiér-Ei. Nous disons d'une classe C appartenant à Ancêtre-A que :

C est un Ancêtre-A

C est un Ancêtre-A si une des conditions suivantes est remplie:

(i) A est une spécialisation simple de C

(ii) Il existe un Ancêtre-A qui est une spécialisation simple de C

(iii) Il existe une classe D qui est un Ancêtre-A et qui est construite par intersection à partir de C (i.e., C est une des classes termes de l'intersection, cf. figure 6.5(a)).

(iv) Il existe une classe D qui est un Ancêtre-A et qui est construite par union à partir de C (i.e., C est une des classes termes de l'union, cf. figure 6.5(b)).

(v) Il existe une classe D qui est un Ancêtre-A et qui est construite par différence à partir de C et d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante (cf. figure 6.5(c)):

$$D : C - B_1 - B_2 - \ldots - B_n$$

C est un Prob-A si une des conditions suivantes est remplie :

(i) C est un Ancêtre-A et C est un Hiér-Ei

(ii) C est une spécialisation simple d'un Prob-A et C n'est pas A et n'est pas un Diff-A.

(iii) C est construit par intersection d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante :

$$\forall i \geq 1 \mid B_i \text{ est un Prob-A}$$

(iv) C est construit par union d'un ensemble de classes B_i, $1 \leq i \leq n$, en respectant la règle suivante:

(a) $\exists i \geq 1 \mid B_i$ est un Prob-A

ou (b) $(\exists i \geq 1 \mid B_i$ est un Sûr-A$) \wedge (\exists j \geq 1 \mid B_j$ est un Diff-A)

ou (c) $(\exists i \geq 1 \mid B_i$ est un Diff-A$) \wedge (\exists j \geq 1 \mid B_j$ est un Déd-A)

(v) C n'est pas un Diff-A et C est construit par différence à partir de la classe D et d'un ensemble de classes B_i, $i \geq 1$, en respectant la règle suivante :

$$C : D - B_1 - B_2 - \ldots - B_n$$

où D est un Prob-A

et $\forall i \geq 1 \mid B_i \neq A$

Commentaire sur la condition (i) :

Même si A est un Hiér-Ei, Ancêtre-A définit un ensemble de classes qui ne sont pas forcément des Hiér-Ei. Comme tout Prob-A doit être défini dans le contexte Hiér-Ei, nous ajoutons à la condition (i) : C est un Hiér-Ei.

Commentaire sur la condition (ii) :

Dans la figure 6.5(c) est illustré le cas où E4 est une spécialisation d'un Prob-E3 (E1) et E4 n'est pas un Prob-E3 car il est avant tout un Diff-E3. En effet, dans ce cas les Prob-A sont définis tout d'abord par exclusion des Diff-A.

Commentaire sur la condition (v) :

Comme pour la condition (ii) la définition des classes Prob-A donnée par la condition (v) est faite par exclusion des Diff-A. Par exemple, C : X_1 - A ou C : X_1 - X_2 où X_1 et X_2 sont des Prob-A peut impliquer que C est un Diff-A.

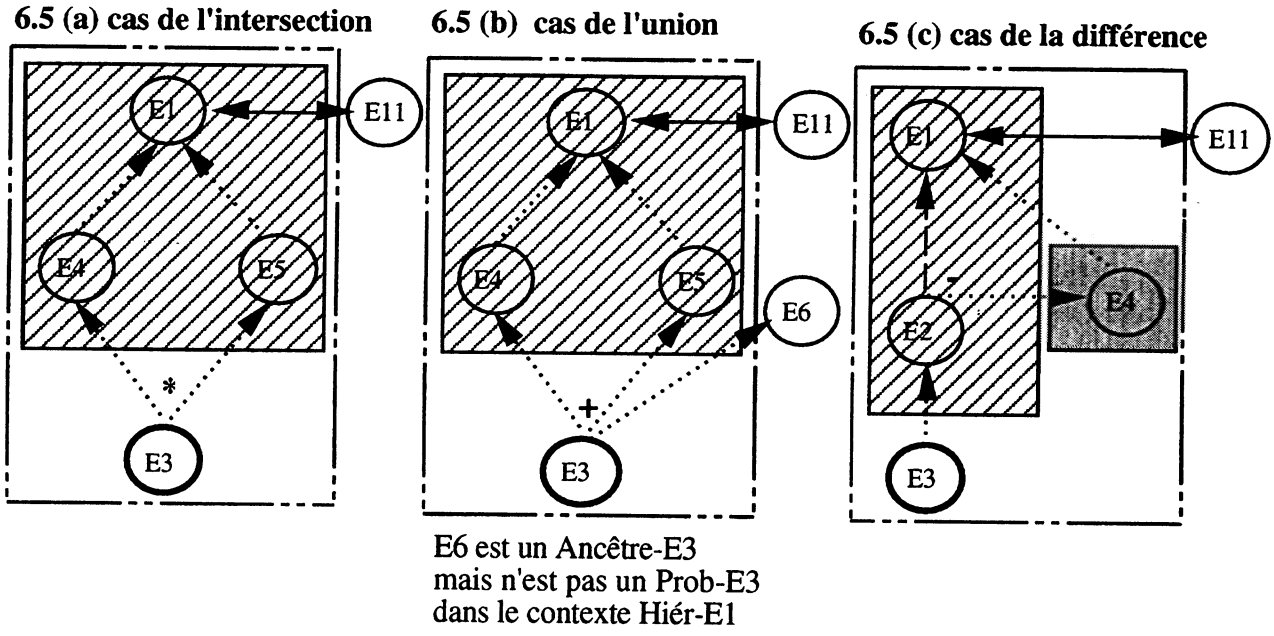


Figure 6.5 - Prob-A : Cas des Classes à partir Desquelles A est Construit

Déd-A

Le sous-ensemble des classes qui appartiennent à Hiér-Ei et dont les entités qui les instancient sont **par déduction** des entités instanciant une autre classe A appartenant aussi à Hiér-Ei est appelé:

Déd-A

Nous disons d'une classe C appartenant à Déd-A que :

C est un Déd-A

C est un Déd-A si une des conditions suivantes est remplie :

(i) C est construit par union d'un ensemble de classes B_i, $1 \le i \le n$, en respectant la règle suivante:

$(\forall i \ge 1 \mid (B_i \text{ est un Sûr-A} \vee \text{NON}(B_i \text{ est un Hiér-Ei}))) \wedge$

$(\exists j \ge 1 \mid B_j \text{ est un Sûr-A}) \wedge$

$(\exists j \ge 1 \mid \text{NON}(B_j \text{ est un Hiér-A}))$

(ii) C est une spécialisation simple d'un Déd-A

(iii) C est construit par intersection d'un ensemble de classes B_i, $1 \le i \le n$, en respectant la règle suivante :

$(\exists i \ge 1 \mid B_i \text{ est un Déd-A}) \wedge$

$(\forall j \ne i \mid B_j \text{ est un Déd-A} \vee B_j \text{ est un Prob-A})$

(iv) C est construit par union d'un ensemble de classes B_i, $1 \le i \le n$, en respectant la règle suivante:

$(\forall i \ge 1 \mid (B_i \text{ est un Déd-A} \vee B_i \text{ est un Sûr-A} \vee$

$\text{NON}(B_i \text{ est un Hiér-Ei}))) \wedge$

$(\exists j \ge 1 \mid \text{NON}(B_j \text{ est un Sûr-A}))$

(v) C est construit par différence à partir de la classe D et d'un ensemble de classes B_i, $i \geq 1$, en respectant la règle suivante :

$$C : D - B_1 - B_2 - \ldots - B_n$$

où D est un Déd-A

Commentaire sur la condition (i) et (iv) :
Dans la figure 6.3, E14 est un Déd-E3 : il est défini par l'union de E8 qui est un Sûr-E3 et de E21 qui n'est pas un Hiér-E3. L'union de E14 avec un autre Déd-E3 ou un autre Sûr-E3, ou une autre classe qui n'est pas un Hiér-E3 (comme par exemple une généralisation de E1) a pour résultat un Déd-E3.

Commentaire sur la condition (iii):
Pour l'intersection, un Déd-A l'emporte sur un Prob-A de même un Sûr-A l'emporte sur un Prob-A (cf. définition de Sûr-A, condition (ii)).

Une autre notion va être introduite pour être utilisée dans d'autres paragraphes : il s'agit de **Descendant-A**. Descendant-A est l'ensemble de classes qui sont construites à partir de A. Sa définition peut se faire par rapport à Sûr-A et Déd-A :

$$\text{Descendant-A} = \text{Sûr-A} \cup \text{Déd-A}$$

Nous achevons ici la définition précise des concepts introduits, relatifs à une hiérarchie de généralisation. Ces concepts vont être utilisés dans l'expression des règles de correspondance pour comparer les classes désignées dans une mise à jour et celles référencées dans une précondition.

Pour établir ces règles nous avons besoin d'étudier aussi la prise en compte de l'inclusion de faits.

Prise en Compte de l'Inclusion de Faits

Jusqu'à maintenant nous n'avons considéré que des mises à jour sur des faits de base, i.e., qui sont définis dans le schéma conceptuel d'une application. L'objectif de ce paragraphe est la prise en compte, dans les règles de correspondance, des mises à jour d'insertion de faits qui sont dérivés des faits de base. Supposons alors la définition, au niveau du schéma conceptuel, du fait de base suivant:

F1 : [F1: E1, ..., Fi : Ei, ..., Fn : En]

La définition de ce fait implique celle des faits qui sont ses généralisations. Le fait

[G1 : K1, ..., Gi : Ki, ..., Gm : Km]

est une généralisation du fait de base ci-dessus ssi :

$$\forall 1 \leq i \leq m \; \exists 1 \leq j \leq n \mid (Gi \equiv Fj \wedge Ki \equiv Ej)$$

Et par conséquent $\{Fi\}_{1 \leq i \leq n} \supseteq \{Gi\}_{1 \leq i \leq m}$ et $m \leq n$

Les mises à jour des généralisations de [F1 : E1, ..., Fn : En] sont alors possibles. Il est alors

établi que la forme d'une mise à jour concernant le fait $\mathcal{F}1$ est la suivante[3] :

MAJ1 : INSERT [G1 v1, ..., Gi vi, ..., Gm vm]
FROM A1 v1, ..., Ai vi,..., Am vm

où $\forall 1 \leq i \leq m \; \exists 1 \leq j \leq n \mid (Gi \equiv Fj \wedge Ai$ est un Hiér-Ej)

De même la forme d'une précondition est la suivante :

PCD1 : TEST INSERT [H1 v1, ..., Hi vi, ..., Hl vl]
FROM B1 v1, ..., Bi vi,..., Bl vl

Cette précondition est héritée par les spécialisations de l'association qu'elle met à jour. De façon concrète ceci veut dire que la précondition doit être évaluée pour une mise à jour qui insère soit le même fait soit une spécialisation de ce fait. Cette précondition doit être évaluée lors de la mise à jour ci-dessus ssi (bien sûr pour qu'il y ait vraiment correspondance il faut de plus comparer les liens hiérarchiques entre les Ai, $1 \leq i \leq m$, et les Bi, $1 \leq i \leq l$) :

$$\{Gi\}_{1 \leq i \leq m} \supseteq \{Hi\}_{1 \leq i \leq l}$$

Dans ce cas, la précondition porte aussi sur le fait $\mathcal{F}1$ et nous avons

$$\forall 1 \leq i \leq l \; \exists 1 \leq j \leq n \mid (Hi \equiv Fj \wedge Bi \text{ est un Hiér-Ej})$$

Finalement, nous avons toutes les données pour pouvoir établir la règle de correspondance. Nous précisons juste le sens donné aux trois types d'existence de correspondance entre une mise à jour et une précondition.

Il y a correspondance : La précondition doit être évaluée à l'exécution de la mise à jour.

Il y a correspondance sous condition : Il faut consulter la base de données pour savoir si la précondition doit être évaluée à l'exécution de la mise à jour.

Il n'y a pas de correspondance : La précondition ne doit pas être évaluée à l'exécution de la mise à jour.

Règle de Correspondance

Les règles de correspondance sont présentées sous-forme algorithmique ; l'ensemble d'informations utilisées (p.ex., $\mathcal{F}1$, MAJ1 et PCD1) est schématisé en début de règle ; les commentaires sont faits entre deux symboles "%". Voyons alors quand est-ce qu'il y a correspondance entre MAJ1 sur $\mathcal{F}1$ et PCD1.

[3]Remarquons que dans le cas général les variables *vi* peuvent être définies par une composition de fonctions. Mais ce qui nous intéresse c'est la classe qui représente l'ensemble d'arrivée de cette composition de fonctions : cette classe est ici désignée par *Ai*.

REGLE1

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fi Ei $1 \le i \le n$	Gi Ai $1 \le i \le m$	Hi Bi $1 \le i \le l$	*Fonctions* *Classes*

SI $\{Gi\}_{1 \le i \le m} \supseteq \{Hi\}_{1 \le i \le l}$

ALORS **SI** $\forall 1 \le i \le l \ \exists 1 \le j \le m \ \exists 1 \le k \le n \mid (Hi \equiv Gj \equiv Fk \wedge$ (a)
(Ai est un Sûr-Bj dans Hiér-Ek $\vee$
Ai est un Déd-Bj dans Hiér-Ek))
ALORS Il y a correspondance
SINON **SI** ($\exists 1 \le i \le l \ \exists 1 \le j \le m \ \exists 1 \le k \le n \mid (Hi \equiv Gj \equiv Fk \wedge$
Ai est un Prob-Bj dans Hiér-Ek)) $\wedge$
($\forall 1 \le i \le l \ \exists 1 \le j \le m \ \exists 1 \le k \le n \mid (Hi \equiv Gj \equiv Fk \wedge$
NON(Ai est un Diff-Bj dans Hiér-Ek)))
ALORS Il y a correspondance sous condition
SINON *% Ai est un Diff-Bj dans Hiér-Ek %*
Il n'y a pas de correspondance

SINON Il n'y a pas de correspondance

Remarquons que cette règle a été établie pour des faits qui lient uniquement des classes d'entités virtuelles. Le cas des faits qui lient des classes d'entités virtuelles et littérales a été laissé de côté. En effet, ces cas ne posent aucun problème (une entité littérale est définie sur un type de base, en conséquence toute la problématique sur la hiérarchie de généralisation n'a pas de sens) par contre les intégrer aux règles aurait alourdi leur écriture (en particulier, la forme d'un fait, d'une précondition et d'une mise à jour serait plus complexe). Ainsi, il a été préférable de les ignorer dans ce paragraphe et les suivants.

Cette règle clot le paragraphe 6.1.1 ; ensuite nous présenterons les règles de correspondance pour d'autres types de mises à jour. Les autres paragraphes seront moins lourds vu que la plupart des notions ont été introduites dans celui-là.

6.1.2. Insertion d'Une Entité

La sémantique d'un énoncé de ce type (cf. 3.3) est d'insérer une nouvelle entité dans la base de données et d'affecter des valeurs à un sous-ensemble de ses fonctions. Par exemple l'énoncé suivant:

INSERT Employé
WITH Véhicule = Camion WHERE ...

crée une instance de la classe Employé et affecte un Camion à la fonction Véhicule(Employé). Le camion à affecter est défini dans la partie WHERE de l'énoncé : cette partie de l'énoncé n'est pas importante ici, elle sera étudiée dans le chapitre 7 qui décrit l'évaluation proprement dite des préconditions. A partir du nom de cette fonction ("Véhicule") il est possible de déterminer sans ambiguïté le fait correspondant[4]. Dans cet exemple le fait est

[Personne : Personne, Véhicule : Véhicule]

Une façon d'exprimer cet énoncé de mise à jour est de dire qu'il crée un nouvel employé et ensuite fait l'insertion du fait suivant :

INSERT [Personne p, Véhicule v]
FROM Employé p, Camion c
WHERE ...

A partir de cette constatation l'établissement de la correspondance entre une mise à jour et une précondition d'insertion d'entités peut se faire en trois phases. Avant de les énumerer un exemple de précondition est présenté.

TEST INSERT Personne
WITH Véhicule = Véhicule WHERE ...
AND Loisir = Loisir WHERE ...

1. Comparaison de la classe dont une entité va être insérée par la mise à jour (Employé) et de la classe désignée dans la clause "INSERT" de la précondition (Personne). Brièvement, il y a correspondance si l'entité insérée appartient aussi à la classe désignée dans la précondition. A ce niveau il y a correspondance entre les exemples de mise à jour et de précondition donnés ci-dessus.

2. Comparaison du nom des fonctions qui vont recevoir une valeur par la mise à jour (Véhicule) et des fonctions désignées dans la précondition (Véhicule, Loisir). Intuitivement il y a correspondance si l'ensemble des fonctions désignées dans la précondition est contenu dans l'ensemble des fonctions de la mise à jour. Dans l'exemple ci-dessus il n'y a pas de correspondance à ce niveau car la mise à jour n'affecte pas une valeur à la fonction Loisir appliquée à une Personne).

3. Application de la règle de correspondance d'insertion de faits (cf. 6.1.1) à chaque insertion de faits incluse (de façon implicite) dans la mise à jour.

Les phases 2 et 3 ne posent pas de problème majeur. Regardons plus en détail la phase 1.

[4]Deux faits liant les mêmes entités ont des noms de rôles ou fonctions différents.

Phase 1 :

Pour qu'il y ait correspondance à ce niveau il faut que l'entité insérée par la mise à jour appartienne à la classe d'entités désignée dans la précondition. La figure 6.6 illustre différents cas de clauses "INSERT" de mises à jour et de préconditions et leur comportement par rapport à la règle de correspondance. A partir de la lecture de cette figure on peut déduire que :

- Si la classe de l'entité insérée par la mise à jour appartient à l'ensemble des classes qui sont des **Sûr** de la classe de la précondition (cf. m1, m2 et m3) alors **il y a correspondance.**

- Si la classe de l'entité insérée par la mise à jour appartient à l'ensemble des **Diff** de la classe de la précondition (cf. m4) alors **il n'y a pas de correspondance.**

- Si la classe de l'entité insérée par la mise à jour appartient à l'ensemble des **Prob** de la classe de la précondition (cf. m5 et m6) alors **il n'y a pas de correspondance.**
  Par exemple, la mise à jour m5 insère une Personne, or elle n'est pas, pour l'instant, un Employé. Pour l'être il faut utiliser un énoncé de changement de classe (cf. 6.1.6). Ainsi, il n'y a pas de raison d'évaluer une précondition qui s'intéresse à l'insertion d'un employé.

La mise à jour m7 est une insertion directe dans une classe union : ce type d'insertion est acceptée dans le FML. Pour m7 il est à remarquer qu'uniquement à partir de la clause "INSERT" il n'est pas possible de décider si Emprunteur est un Employé ou un Département. Néanmoins, à partir des fonctions désignées dans la clause "WITH" il peut être possible de déterminer l'appartenance de cet emprunteur à une des deux classes. Par exemple, supposons l'énoncé suivant : "INSERT Emprunteur WITH Véhicule = ...", alors on peut déduire que cet Emprunteur est un Employé car il est associé à un Véhicule. Emprunteur est alors **par déduction** un Employé et **il y a correspondance** entre cette mise à jour et p1.

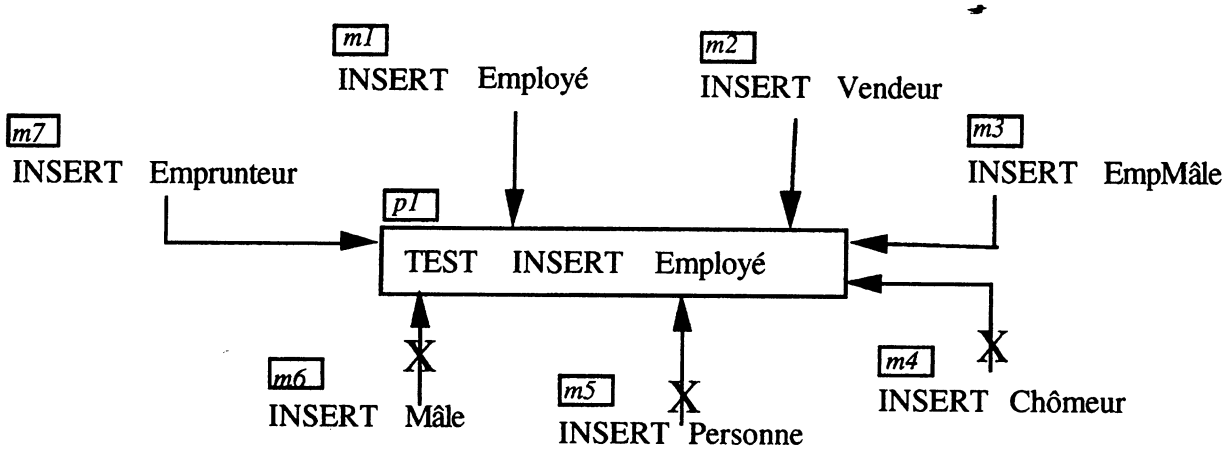


Figure n° 6.6 - Exemples de Correspondance : Précondition et Mise à Jour du Type Insertion d'une Entité

Les parties de l'énoncé qui sont prises en compte par les règles de correspondance sont le nom de

la classe de l'entité à insérer, la désignation des fonctions qui reçoivent une valeur, et la classe dans laquelle ces fonctions prennent une valeur. Ainsi, dans le but d'établir les règles de correspondance la forme d'une mise à jour est définie comme :

MAJ2 : INSERT A
WITH $G1_1 = A1_1$
AND $G1_2 = A1_2$
...
AND $G1_p = A1_p$

où A est une classe et $\forall 1 \leq i \leq p$ | ($A1_i$ est une classe $\wedge$ $G1_i$ est une fonction applicable à A).

Cette mise à jour implique l'existence des faits ci-dessous. Ils peuvent être des faits de base ou des généralisations de faits de base (i.e., ils ne sont pas définis dans le schéma conceptuel).

F21 : [$F1_1 : E1_1, F2_1 : E2_1$]
F2₂ : [$F1_2 : E1_2, F2_2 : E2_2$]
...
F2p : [$F1_p : E1_p, F2_p : E2_p$]

où $\forall 1 \leq i \leq p$ | (($A1_i$ est un Hiér-$E1_i$ $\wedge$ A est un Hiér-$E2_i$) $\wedge$ ($G1_i \equiv F1_i$)).

Les insertions de faits implicites dans cette mise à jour sont les suivantes :

MAJ21 : INSERT [$G1_1$ v1, $G2_1$ v2]
FROM $A1_1$ v1, $A2_1$ v2
...
MAJ2p : INSERT [$G1_p$ vp, $G2_p$ v2]
FROM $A1_p$ vp, $A2_p$ v2

où $\forall 1 \leq i \leq p$ | ($G2_i \in \{F1_i, F2_i\}$). Remarquons que $\forall 1 \leq i \leq p$ ($A2_i \equiv A$). Nous utilisons cette notation pour être homogène avec le reste du paragraphe (en particulier, avec 6.1.5).

De même la forme d'une précondition est définie par :

PCD2 : TEST INSERT B
WITH $H1_1 = B1_1$
AND $H1_2 = B1_2$
...
AND $H1_q = B1_q$

où B est une classe et $\forall 1 \leq i \leq p$ | ($B1_i$ est une classe $\wedge$ $H1_i$ est une fonction applicable à B).

Un ensemble de préconditions d'insertions de faits est aussi inclus de façon implicite dans PCD2. Pour $1 \leq i \leq q$, nous avons les préconditions suivantes :

PCD2i : TEST INSERT [$H1_i$ v1, $H2_i$ v2]
FROM $B1_i$ v1, $B2_i$ v2

De même, remarquons que $\forall 1 \leq i \leq p$ ($B2_i \equiv B$).

La règle de correspondance peut alors être établie.

REGLE2

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fsi Esi 1≤s≤2 1≤i≤p	Gsi A, Asi 1≤s≤2 1≤i≤p	Hsi B, Bsi 1≤s≤2 1≤i≤q	*Fonctions* *Classes*

SI (A est un Sûr-B dans le contexte Hiér-B ∨
A est un Déd-B dans le contexte Hiér-B)

ALORS **SI** $\{G1_i\}_{1\le i\le p} \supseteq \{H1_i\}_{1\le i\le q}$ | (a)
ALORS Rechercher l'existence de correspondance |
pour chaque $1\le i\le q$ en appliquant la partie (a) de REGLE1 |
à PCD2i et à MAJ2i où $H1_i \equiv G1_j$ |
avec m=n=l=2 et $\forall 1\le s\le 2$ (Fs≡Fsi ∧ Gs≡Gsi ∧ Hs≡Hsi |
∧ Es≡Esi ∧ As≡Asi ∧ Bs≡Bs_i) |
%correspondance avec l'ensemble d'informations de REGLE1% |
SI $\forall 1\le i\le q$ il y a correspondance *%entre $MAJ2_i$ et $PCD2_i$%* |
ALORS il y a correspondance *%entre MAJ2 et PCD2%* |
SINON SI $\forall 1\le i<\le q$ (il y a correspondance sous condition |
∨ il y a correspondance) |
ALORS Il y a correspondance sous condition |
SINON il n'y a pas de correspondance |
SINON Il n'y a pas de correspondance |
SINON Il n'y a pas de correspondance

6.1.3. Suppression de Faits

La sémantique d'un énoncé de cette catégorie est la suppression de faits qui lient un ensemble d'entités. Comme il a été dit (cf. 3.3) cette mise à jour est la symétrique d'une insertion de faits. La syntaxe des clauses "DELETE" et "FROM" est la même que celle des clauses "INSERT" et "FROM" d'un énoncé d'insertion de faits (à part bien sûr les mots clés "DELETE" et "INSERT"). Un exemple d'une mise à jour et précondition de ce type est :

DELETE [Personne p Véhicule v]
FROM Vendeur p, Véhicule v

TEST DELETE [Personne p Véhicule v]
FROM Employé p, Véhicule v

A noter la similarité de cette mise à jour et de cette précondition avec m2 et p1 de la figure 6.2. La précondition doit être évaluée quand un ou plusieurs véhicules sont supprimés de l'ensemble

de valeurs de la fonction Véhicule(Employé) et inversement quand un ou plusieurs employés sont supprimés de l'ensemble de valeurs de la fonction Personne(Véhicule). La mise à jour supprime des véhicules dans l'ensemble de valeurs de la fonction Véhicule(Vendeur) et des vendeurs dans l'ensemble de valeurs de la fonction Personne(Véhicule). Or, tout Vendeur est un Employé et en conséquence il y a correspondance.

En effet, les principes qui guident l'établissement de la règle de correspondance de suppression de faits est le même que celui de la règle d'insertion de faits. Une suppression de faits sur F1 (cf. 6.1.1) a la forme suivante :

MAJ3 : DELETE [G1 v1, ..., Gi vi, ..., Gm vm]
FROM A1 v12, ..., Ai vi,..., Am vm

où $\forall 1 \leq i \leq m \; \exists 1 \leq j \leq n \mid (Gi \equiv Fj \wedge Ai$ est un Hiér-Ej)

De même la forme d'une précondition est la suivante :

PCD3 : TEST DELETE [H1 v1, ..., Hi vi, ..., Hl vl]
FROM B1 v1, ..., Bi vi,..., Bl vl

où $\forall 1 \leq i \leq l \; \exists 1 \leq j \leq n \mid (Hi \equiv Fj \wedge Bi$ est un Hiér-Ej)

La règle entre MAJ3 et PCD3 peut alors être établie de façon très simple.

REGLE3

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fi Ei $1 \leq i \leq n$	Gi Ai $1 \leq i \leq m$	Hi Bi $1 \leq i \leq l$	*Fonctions* *Classes*

Appliquer la REGLE1 à cet ensemble d'informations.

6.1.4. Supression d'Entités

La sémantique de cet énoncé est la suppression d'entités ainsi que des faits qui leur sont associés. L'énoncé est syntaxiquement constitué d'une clause "DELETE" (entre autre) qui spécifie la classe des entités à supprimer (cf. 3.3). C'est cette classe qui nous intéresse pour établir la règle de correspondance.

La figure 6.7 présente quelques exemples de mises à jour et d'une précondition de ce type et illustre leur comportement par rapport à la règle.

Un commentaire d'ordre général sur la sémantique de cet énoncé mérite d'être fait. Si un Vendeur est détruit, il continue à exister quand même en tant qu'Employé et que Personne. Par contre, si Personne est supprimée alors elle n'existe plus dans la base de données ; même si elle était un Employé et un Vendeur. Ceci est un peu l'inverse de ce qui arrive avec l'énoncé d'insertion d'entités: si un Vendeur est inséré alors il est aussi inséré en tant qu'Employé et que Personne ; par contre si une Personne est insérée, cette insertion n'est propagée ni à Employé ni à Vendeur.

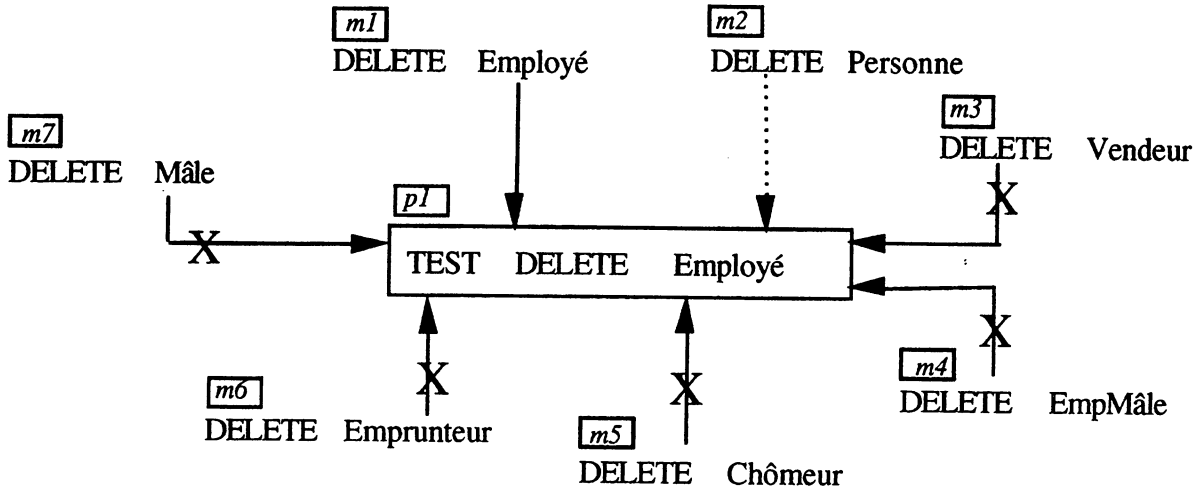


Figure n° 6.7 - Exemples de Correspondance : Précondition et Mise à Jour du Type Suppression d'Entités

En conséquence la règle de correspondance d'une suppression d'entités est en quelque sorte l'inverse de la phase 1 de la règle d'insertion d'une entité. Pour illustrer, quelques commentaires à la figure 6.7 suivent :

- Si la classe de la mise à jour **est la même** que la classe de la précondition (cf. m1) alors **il y a évidemment correspondance**.

- Si la classe de la mise à jour est un **Ancêtre** de la classe de la précondition (m2) alors **il y a correspondance sous condition**. Par exemple pour m2 et p1 nous ne sommes pas sûrs que la Personne supprimée soit un Employé : il faut consulter la base de données. La définition de Ancêtre-A est donnée dans le paragraphe 6.1.1 et rejoint celle d'un Prob-A à partir duquel A est construit.

- Si la classe de la mise à jour est un **Prob** de la classe de la précondition et n'est **pas un Ancêtre** de cette classe (cf.m7) alors **il n'y a pas de correspondance**. En effet, la suppression d'un Mâle n'entraîne pas la suppression de ses ancêtres, i.e., Personne.

- Dans les autres cas, i.e., la classe de la mise à jour est un **Sûr** (cf. m3 et m4), un **Diff** (cf. m5) ou un **Déd** (éventuellement cf. m6) alors **il n'y a pas de correspondance**.

L'établissement de la règle de correspondance ne pose aucun problème. Supposons la mise à jour MAJ4 et la précondition PCD4.

MAJ4 : DELETE A

PCD4 : TEST DELETE B

où A et B sont des classes d'entités.
La règle de correspondance est la suivante :

REGLE4

ENSEMBLE D'INFORMATIONS

Mise à Jour	Précondition	
A	B	*Classes*

SI A ≡ B

ALORS Il y a correspondance
SINON **SI** A est un Ancêtre-B
ALORS Il y a correspondance sous condition
SINON Il n'y a pas de correspondance

6.1.5. Modification de Faits

La sémantique d'un énoncé de cette catégorie (cf. 3.3) est la suppression de faits suivie d'une insertion de faits du même type. Au niveau syntaxique la forme de la clause "ASSERT" est ressemblante à celle de la clause "WITH" d'un énoncé d'insertion d'entités. Par exemple la mise à jour suivante affecte une nouvelle valeur (un camion) à la fonction Véhicule appliquée à un Vendeur.

ASSERT Vendeur.Véhicule = Camion WHERE *<Qualification du camion>*
WHERE *<Qualification du Vendeur>*

Une clause "FROM" peut être utilisée pour la déclaration de variables. Par exemple, cette mise à jour peut s'écrire "ASSERT v.Véhicule = Camion WHERE ... FROM Vendeur v, ... WHERE ...". Néanmoins, à ce niveau, ce qui nous intéresse est la classe où la fonction désignée dans la clause "ASSERT" prend ses valeurs. Dans ce cas la forme de la mise à jour peut se ramener à :

MAJ5 : ASSERT A.G1$_1$ = A1$_1$
A.G1$_2$ = A1$_2$
...
A.G1$_p$ = A1$_p$

où A est une classe et $\forall 1 \leq i \leq p$ | (A1$_i$ est une classe $\wedge$ G1$_i$ est une fonction applicable à A). On peut remarquer que plusieurs fonctions sur A peuvent être modifiées par le même énoncé.

Comme pour une insertion d'une entité, cet énoncé implique l'existence d'un ensemble de faits. Pour $1 \leq i \leq p$ nous avons le fait suivant :

F5$_p$: [F1$_i$: E1$_i$, F2$_i$: E2$_i$]

où $\forall 1 \leq i \leq p$ | ((A1$_i$ est un Hiér-E1$_i$ $\wedge$ A est un Hiér-E2$_i$) $\wedge$ (G1$_i$ $\in$ {F1$_i$, F2$_i$}))

Les suppressions et insertions de faits implicites dans cette mise à jour sont pour $1 \leq i \leq q$,

MAJ5i1 : DELETE [G1$_i$ v1, G2$_i$ v2]
FROM A1$_i$ v1, A2$_i$ v2

MAJ5i2 : INSERT [G1$_i$ v1, G2$_i$ v2]
FROM A1$_i$ v1, A2$_i$ v2

De même la forme d'une précondition est définie par :

PCD5 : TEST ASSERT B.H11= B11
B.H12 = B12
...
B.H1p = B1p

B et B$_i$, $1 \leq i \leq l$, sont des classes d'entités ; les H1$_i$, $1 \leq i \leq l$ sont des fonctions applicables à B.

Un ensemble de préconditions de suppression et d'insertions de faits est aussi inclus de façon implite dans PCD5. Pour $1 \leq i \leq q$, nous avons les préconditions suivantes :

PCD5i1 : TEST DELETE [H1$_i$ v1, H2$_i$ v2]
FROM B1$_i$ v1, B2$_i$ v2

PCD5i2 : TEST INSERT [H1$_i$ v1, H2$_i$ v2]
FROM B1$_i$ v1, B2$_i$ v2

Remarquons que le type de la correspondance existant entre PCD5i1 et MAJ5i1 où G1$_i$ = H1$_j$ est le même qui existe entre PCD5i2 et MAJ5i2. Ainsi, nous n'avons besoin que de comparer les mises à jour et les préconditions d'insertions de faits (ou celles de suppressions, mais pas des deux) implicites dans MAJ5 et PCD5. Remarquons de plus que MAJ5i1 et MAJ2$_i$, PCD5i1 et PCD2i, et $\mathcal{F}$5$_i$ et $\mathcal{F}$2$_i$ sont égaux ; en conséquence, la partie (a) de la REGLE2 peut être appliquée directement.

REGLE5

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fsi	Gsi	Hsi	*Fonctions*
Esi	A, Asi	B, Bsi	*Classes*
$1 \leq s \leq 2$ $1 \leq i \leq p$	$1 \leq s \leq 2$ $1 \leq i \leq p$	$1 \leq s \leq 2$ $1 \leq i \leq q$	

Appliquer la partie (a) de REGLE2 à cet ensemble d'informations.

6.1.6. Changement de Classe

La sémantique de cet énoncé est d'insérer une entité déjà existante et (évidemment) appartenant à une certaine classe, dans une autre classe d'entités qui est bien sûr une spécialisation (simple ou composée) de la première. La clause "INSERT" d'un tel énoncé sert à spécifier la classe dans

laquelle on va insérer l'entité. Dans la clause "FROM" est spécifiée la (éventuellement, une des) classe à laquelle l'entité appartient (avant la mise à jour). Ainsi, "INSERT Vendeur FROM Employé WHERE ..." implique que les Employés qualifiés dans la clause "WHERE" sont aussi, après l'exécution de la mise à jour, des Vendeurs. A partir des clauses "INSERT" et "FROM", les règles de correspondance peuvent être établies. La forme d'une mise à jour et d'une précondition de ce type est donnée ci-dessous.

MAJ6 : INSERT A1
FROM A2

PCD6 : TEST INSERT B1
FROM B2

Quelques exemples illustrant le comportement de la règle de correspondance sont présentés dans la figure 6.8.

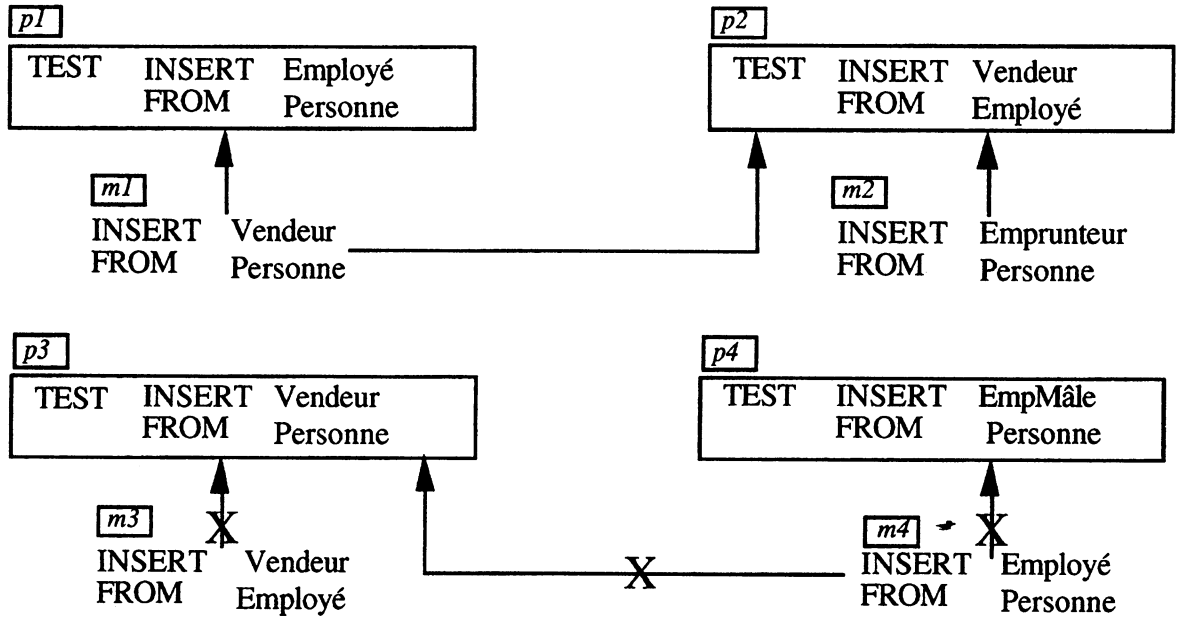


Figure n° 6.8 - Exemples de Correspondance : Précondition et Mise à Jour du Type Changement de Classe

Intuitivement, il y a correspondance entre MAJ6 et PCD6 si les classes de la précondition sont englobées par les classes de la mise à jour, i.e. si A1 est un Ancêtre de B1 et si A2 est un Descendant de B2.

La lecture de la figure 6.8 se fait facilement : une seule remarque sur p4 et m4 a lieu d'être faite. Dans le SIB tous les changements de classe sont faits de façon manuelle. Néanmoins, plusieurs systèmes ont un mode d'insertion automatique par rapport aux changements de classe. Par exemple, si j'insère une Personne dans Employé et que cette Personne est aussi un Mâle, alors la mise à jour devrait être propagée à la classe EmpMâle (qui est en fait l'ensemble de **tous** les employés qui sont des mâles). Dans ce cas il y aurait correspondance sous condition entre m4 et p4.

REGLE6

ENSEMBLE D'INFORMATIONS

Mise à Jour	Précondition	
A1, A2	B1, B2	*Classes*

SI ((A1 est un Ancêtre-B1 $\vee$ A1 $\equiv$ B1) $\wedge$ (A2 est un Descendant-B2 $\vee$ A2 $\equiv$ B2))
ALORS Il y a correspondance sous condition
SINON Il n'y a pas de correspondance

6.2. Précondition et Mise à Jour de Types Différents

Dans l'introductionde ce chapitre les trois points importants pour l'établissement des règles de correspondance ont été mis en évidence : la hiérarchie de généralisation, l'inclusion de faits et **l'inclusion sémantique d'énoncés de manipulation**. Ce paragraphe-ci étudie la prise en compte du dernier aspect par les règles de correspondance.

Par la lecture des paragraphes précédents et en particulier, à partir de la forme des différents types de mise à jour, il a pu être constaté que certains types d'énoncés de manipulation en incluent sémantiquement d'autres. Ceci est une caractéristique des langages sémantiques de haut niveau offrant un éventail large de possibilités pour que l'utilisateur puisse exprimer ses requêtes.

La conséquence de cet aspect au niveau des règles de correspondance est qu'il est possible qu'une précondition doive être évaluée pas seulement par une mise à jour du même type mais aussi par des énoncés de type différent. Il faut alors déterminer le rapport d'inclusion entre les différents types d'énoncés de manipulation.

Nous avons déjà vu qu'une insertion d'entités ou une modification de faits incluent sémantiquement une insertion de faits. Ceci a comme conséquence qu'une précondition du type insertion de faits doit être évaluée par trois types d'énoncés : insertion de faits, insertion d'une entité et modification de faits. Le même raisonnement s'applique pour les suppressions, i.e., une précondition de suppression de faits est évaluée pour : une suppression de faits, une suppression d'entités et une modification de faits.

Vu que dans les paragraphes précédents les énoncés de mise à jour et de précondition du type insertion d'entités et modification de faits ont déjà été présentés en fonction des insertions et suppressions de faits implicites, l'établissement des règles de correspondance est très simple.

6.2.1. Précondition d'Insertion de Faits et Mise à Jour d'Insertion d'une Entité

Une mise à jour d'insertion d'une entité vis-à-vis d'une précondition d'insertion de faits implique

d'analyser quelles insertions de faits incluses de façon implicite dans la mise à jour peuvent provoquer l'évaluation de la précondition.

Dans le paragraphe 6.1.2 la liste d'insertion de faits implicite dans une insertion d'entités a été déduite. Néanmoins, dans 6.1.2 nous ne nous interessions qu'aux faits binaires car on établissait la correspondance avec une précondition du même type. Ici le problème est légèrement différent vu que nous allons comparer la mise à jour avec une précondition d'insertion de faits où les faits peuvent être n-aires. Pour illustrer, supposons le schéma conceptuel, la mise à jour et les préconditions suivants :

Vente, Personne : Entity ;
Vendeur, Client : Personne ;
[La-Vente : Vente, Le-Vendeur : Vendeur, Le-Client : Client]

```
m1 :            INSERT  Vente
                WITH    Le-Vendeur = Vendeur WHERE ...
                AND     Le-Client = Client WHERE ...

p1 :    TEST    INSERT  [La-Vente l, Le-Vendeur v]
                FROM    Vente l, Vendeur v
                WHERE   ...

p2 :    TEST    INSERT [La-Vente l, Le-Vendeur v, Le-Client c]
                FROM    Vente l, Vendeur v, Client c
                WHERE   ...
```

D'après 6.1.2 deux insertions de faits, m1.1 et m1.2, sont implicites dans m1, elles concernent respectivement les faits suivants :

[La-Vente : Vente, Le-Vendeur : Vendeur]
[La-Vente : Vente, Le-Client : Client]

D'après la règle de correspondance d'insertion de faits, m1.1 implique bien l'évaluation de p1, mais ni m1.1, ni m1.2 n'impliquent l'évaluation de p2. Ceci parce que l'ensemble de fonctions mis à jour par un de ces énoncés n'inclue pas l'ensemble de fonctions mis à jour par la précondition. Or, intuitivement ceci semble incohérent. En effet, la mise à jour m1 implique bien l'insertion du fait [La-Vente : Vente, Le-Vendeur : Vendeur, Le-Client : Client] (et m1.1 et m1.2 sont des conséquences de cette insertion) et donc l'évaluation de p2.

En conséquence, pour décider de la correspondance entre ce type de mise à jour et de précondition, il faut :

1. Analyser si le fait de la précondition est mis à jour par l'insertion d'entités. Ceci peut être réalisé facilement en comparant l'ensemble de fonctions mis à jour par la précondition et celui de la mise à jour. Dans l'exemple, nous voyons que l'ensemble des fonctions de la précondition, {La-Vente, Le-Vendeur, Le-Client}, contient de plus, par rapport aux fonctions de la clause "WITH" de la mise à jour, la fonction La-Vente. Or, cette fonction est incluse de

façon implicite dans la mise à jour : elle est l'inverse de Le-Vendeur(Vente) ou de Le-Client(Vente).
Ainsi, on peut établir que si la précondition ne contient qu'une fonction de plus que la clause "WITH" de la mise à jour et que si cette fonction a comme ensemble d'arrivée l'entité à insérer ou un Ancêtre de cette entité, alors le résultat de la comparaison est positif.

2. Exprimer alors la partie de la mise à jour qui concerne le même fait que la précondition en terme d'une insertion de faits. On peut alors appliquer la partie (a) de la règle d'insertion de faits à la mise à jour déduite et à la précondition.

La règle de correspondance va être établie pour MAJ2 et PCD1 sur $\mathcal{F}1$.

REGLE7

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fi	Gsi	Hi	*Fonctions*
Ei	A, Asi	Bi	*Classes*
$1 \le i \le n$	$1 \le s \le 2$ $1 \le i \le p$	$1 \le i \le l$	

SI $\exists K \mid (K = \{Ki\}_{1 \le i \le r} \wedge \{G1_i\}1 \le i \le p \supseteq K \wedge \{Hi\}1 \le i \le l \supseteq K \wedge r = l - 1) \wedge$
$\exists 1 \le j \le l \mid (Hj \notin K \wedge Hj$ a comme ensemble d'arrivée A ou un Ancêtre-A)

ALORS Appliquer la partie (a) de REGLE2 à PCD1et à la mise à jour :
INSERT [G1 v1, ..., Gr vr, Gr+1 vr+1]
FROM A1 v1, ..., Ar vr, A vr+1
où $(\{Gi\}1 \le i \le r = \{Ki\}1 \le i \le r) \wedge$
$(\exists 1 \le k \le l \mid Gr+1 \equiv Hk) \wedge$
$(\forall 1 \le j \le r \; \exists 1 \le i \le p \mid (Gj \equiv G1_i \wedge Aj \equiv A1_i))$
avec m = r+1

SINON Il n'y a pas de correspondance

6.2.2. Précondition d'Insertion ou de Suppression de Faits et Mise à Jour de Modification de Faits

Le problème posé dans ce paragraphe est très similaire à celui du paragraphe précédent. En effet, dans un énoncé de modification de faits est inclus de façon implicite un ensemble de suppressions et d'insertions de faits : ceci a été étudié dans 6.1.5. La forme syntaxique de l'énoncé est très proche de celui d'une insertion d'entités et le raisonnement qui a été appliqué dans 6.2.1 peut être adopté ici. En conséquence, la règle de correspondance entre MAJ5 et PCD1 ou PCD3 est la même que celle entre MAJ2 et PCD1. Comme pour l'établissement de REGLE5 nous n'avons besoin de prendre en compte que les insertions (ou les suppressions) implicites dans l'énoncé de modification.

REGLE8

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fi	Gsi	Hi	*Fonctions*
Ei	A, Asi	Bi	*Classes*
$1 \le i \le n$	$1 \le s \le 2$ $1 \le i \le p$	$1 \le i \le l$	

Appliquer la règle 7 à cet ensemble d'informations.

6.2.3. Précondition de Suppression de Faits et Mise à Jour de Suppression d'Entités

La supression d'une entité implique la suppression de tous les faits auxquels elle participe. Nous avons aussi vu (cf. 6.1.4) qu'un énoncé de suppression d'entités implique la suppression des entités de la classe désignée dans la clause "DELETE" de l'énoncé, mais pas leur suppression des classes ancêtres.

La figure 6.9 donne des exemples de mise à jour de suppression d'entités, de préconditions de suppression de faits et de leur comportement par rapport à la règle de correspondance.

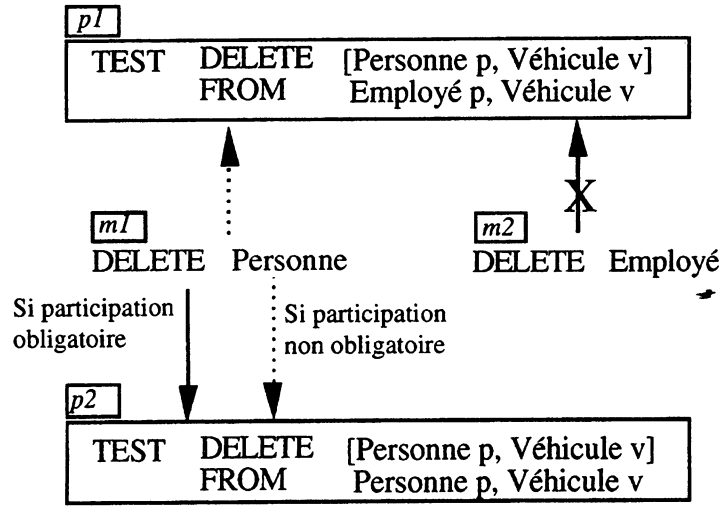


Figure n° 6.9 - Exemples de Correspondance : Précondition de Suppression de Faits et Mise à Jour de Suppression d'Entités

Quelques commentaires sur cette figure suivent :

1. En ce qui concerne **p1 et m2**, la précondition doit être évaluée si un fait entre une Personne et un Véhicule est supprimé et si de plus cette Personne est un Employé. La mise à jour supprime des employés de la base de données et en conséquence tous les faits propres à ces employés. Or, le fait [Personne, Véhicule] n'est pas propre à Employé mais à Personne et il

n'est donc pas supprimé par m2. Ceci justifie qu'il n'y ait pas de correspondance entre m2 et p1.
Nous pouvons alors conclure qu'une condition nécessaire pour qu'il y ait correspondance, est que la classe désignée dans la clause "DELETE" de la mise à jour soit une des classes qui est liée par le fait (de la précondition) dans sa définition.

2. En ce qui concerne **m1 et p1**, la mise à jour implique bien la suppression d'instances du fait spécifié dans la précondition. Il faut alors, pour qu'il y ait correspondance, que les personnes à supprimer soient aussi des employés. Ceci implique des accès à la base de données, en conséquence il y a correspondance sous condition entre m1 et p1.
Ainsi, si la condition 1 est vérifiée et si la classe d'entités supprimée est un Ancêtre d'une classe où une des fonctions du fait de la précondition prend ses valeurs, alors il y a correspondance sous condition.

3. En ce qui concerne **m1 et p2** la condition 1 (ci-dessus) est vérifiée ; de plus la classe des entités à supprimer est la même qu'une classe où une des fonctions du fait de la précondition prend ses valeurs. Il devrait alors y avoir correspondance. Néanmoins, si la contrainte de cardinalité du fait n'oblige pas toute Personne à être associée à un Véhicule, alors statiquement nous ne pouvons pas savoir si un fait entre Personne et Véhicule est réellement supprimé et dans ce cas il y a correspondance sous condition.

La règle de correspondance entre MAJ4 et PCD3 peut être établie.

REGLE9

ENSEMBLE D'INFORMATIONS

Fait	Mise à Jour	Précondition	
Fi Ei $1 \le i \le n$	A	Hi Bi $1 \le i \le l$	*Classes*

SI $\exists 1 \le i \le n \mid (A = Ei \wedge A = Bi$

ALORS **SI** les contraintes de cardinalité de [F1, ..., Fn]
obligent tout A à participer au fait[5]
ALORS Il y a correspondance
SINON Il y a correspondance sous condition

SINON **SI** $\exists 1 \le i \le n \mid (A =$ Ei $\wedge$ Bi est un Descendant-A)
ALORS Il y a correspondance sous condition
SINON Il n'y a pas de correspondance

[5]Cette expression n'est pas détaillée ici.

6.3. Préconditions d'Etat

Pour étudier quelles sont les mises à jour qui peuvent provoquer l'évaluation d'une précondition d'état, l'exemple 4.16 est repris :

TEST STATE FE
FROM Département FE.Employé e
WHERE Avg (e.Salaire) ≥ 15000

Cette précondition doit être évaluée lors des mises à jour suivantes :

m1 : Insertion d'un fait entre Département et Employé.
m2 : Insertion d'un fait entre Employé et Salaire
m3 : Suppression d'un fait entre Département et Employé
m4 : Suppression d'un fait entre Employé et Salaire
m5 : Modification de la fonction Département(Employé) ou de la fonction Employé(Département).
m6 : Modification de la fonction Employé(Salaire).
m7 : Insertion d'un nouveau Département si un ensemble d'employés lui est affecté.
m8 : Insertion d'un nouvel Employé si un salaire lui est affecté.
m9 : Suppression d'un Employé.

Ce qu'il y a de commun dans les mises à jour m1, ..., m9, et qui peut faire changer la valeur logique de la précondition d'état ci-dessus, est qu'elles modifient l'ensemble de valeurs des faits [Département, Employé] et [Employé, Salaire]. Ainsi, il est possible de trouver un ensemble de préconditions d'insertions et de suppressions de faits tel que si une mise à jour implique l'évaluation de cette précondition d'état, elle implique aussi l'évaluation d'au moins une des préconditions de changement de cet ensemble.

Pour cet exemple, au moins une des préconditions p1, ..., p4 ci-dessous, doit être évaluée si la précondition d'état doit être évaluée, i.e., si une des mises à jour m1, ..., m9 s'exécute.

p1 : TEST INSERT [Département d, Employé e]
FROM Département d, Employé e
Evaluée lors des mises à jour m1, m5 et m7

p2 : TEST INSERT [Employé e, Salaire k]
FROM Employé e où k est une constante
Evaluée lors des mises à jour m2, m6 et m8

p3 : TEST DELETE [Département d, Employé e]
FROM Département d, Employé e
Evaluée lors des mises à jour m3, m5 et m9

p4 : TEST DELETE [Employé e, Salaire k]
FROM Employé e où k est une constante
Evaluée lors des mises à jour m4, m6 et m9

Cet ensemble de préconditions est constitué d'une précondition d'insertion de faits et d'une de suppression de faits pour chaque fonction désignée dans la précondition d'état. Remarquons aussi que les préconditions de suppression de faits n'ont été nécessaires que parce qu'une fonction d'agrégation est utilisée dans la précondition d'état.

En effet, par rapport aux règles de correspondance toute précondition d'état peut s'écrire comme une combinaison disjonctive de préconditions de changement. Par conséquent la recherche d'existence de correspondance se fera avec les règles déjà établies.

L'expression d'une précondition d'état en une combinaison disjonctive de préconditions de changement n'est pas détaillée dans ce rapport : seule une classe restreinte de ce type de préconditions a mérité notre attention et une formalisation générale n'a donc pas eu lieu. Cependant, les principes qui doivent être appliqués pour générer la combinaison disjonctive de préconditions de changement sont énumérés ci-dessous. Remarquons qu'il est intéressant que cette combinaison ait un nombre minimal de préconditions.

Principes pour générer les préconditions de changement :

- Pour chaque fonction désignée dans la précondition d'état générer une précondition d'insertion de faits. Des indications pour réaliser la génération suivent.
  Désignation de la fonction dans la précondition d'état : A2.F1
  Fait correspondant : [F1 : E1, F2 : E2] où A2 est un Descendant-E2
  Précondition de changement à générer : TEST INSERT [F1 v1, F2 v2]
  FROM E1 v1, A2 v2

- Pour chaque fonction désignée dans la précondition d'état qui participe à la formation d'un ensemble sur lequel une fonction d'agrégation ou une expression ensembliste sont appliquées, générer une précondition de suppression de faits. Les indications de génération présentées ci-dessus sont applicables à ce cas.

- Pour chaque classe d'entités E désignée de façon isolée dans la précondition d'état et sur laquelle une fonction d'agrégation ou une expression ensembliste sont appliquées (p.ex., ("COUNT (Employé)" ou "e in Vendeur"), générer une précondition d'insertion d'entités du type "INSERT E" et une précondition de suppressions d'entités du type "DELETE E".

Par ailleurs, ce type de préconditions pose un problème d'ordre différent. Dans les paragraphes 4.2 et 4.5 leur sémantique a été présentée :

Si une activité manuelle est liée à une précondition d'état, alors une instance d'activité est créée

et habilitée à être exécutée pour une certaine entité focale lorsque la précondition devient vraie (pour cette entité focale). Néanmoins, l'entité focale peut changer d'état et la précondition être évaluée comme faux avant que l'instance de l'activité ne soit exécutée. Il faut alors supprimer l'instance de la liste d'activités habilitées à être exécutées.

Ainsi, le problème est d'offrir un mécanisme qui permette de "désactiver" les activités habilitées à être exécutées. Pour cela, il faut contrôler qu'une précondition qui est vraie pour une entité particulière est devenue fausse.

Pour détecter qu'une précondition peut devenir vraie, certaines actions bases de données sur des **classes d'entités** doivent être contrôlées : ceci peut se faire sans consulter la base de données. Par contre, pour détecter qu'une précondition peut devenir fausse pour une entité focale particulière, il faut contrôler des actions base de données sur des **instances d'entités** : des accès à la base sont alors nécessaires. Or, ceci peut être extrêmement lourd pour le système !

Pour résoudre ce problème, plusieurs solutions ont été envisagées. Par exemple, au lieu de faire la détection en analysant des requêtes, la faire en surveillant certaines associations dans la base. En particulier, on peut associer chaque instance d'activité et la liste des faits qui peuvent rendre la précondition fausse pour leur entité focale.

Finalement, une autre solution, pas très élégante, a été adoptée. Elle consiste à ne pas surveiller les changements de valeur d'une précondition de vraie à fausse. Ainsi, une instance d'activité habilitée à être exécutée, le reste jusqu'à ce qu'un utilisateur ou une application demande son exécution ou tout simplement des informations sur elle. Au moment de l'exécution d'une requête qui porte sur une telle activité, le système doit évaluer la précondition pour l'entité focale correspondante et réagir en fonction de sa valeur vraie ou fausse.

Ce qui choque le plus dans cette façon de résoudre le problème est que certaines informations dans la base de données peuvent être temporairement incohérentes : une activité est habilitée à être exécutée alors que sa précondition est fausse pour son entité focale. Néanmoins, tout accès à ces informations implique leur mise à jour si jamais elles ne réflètent pas l'état de la base de données : ainsi, l'utilisateur n'est pas mis en défaut par cette politique de résolution.

Bien que cette solution ne résolve pas vraiment le problème - elle l'évite plutôt - elle s'est avérée la plus efficace : considérablement moins de charge pour le système et plus facile à implémenter. Le développement d'un prototype était, je le rappelle, une des préocupations du projet.

Enfin, remarquons aussi que ce problème ne se pose que pour les activités manuelles, les activités automatiques étant lancées lors de la confirmation des actions qui impliquent leur création et habilitation à être exécutées.

6.4. Règle Générale

Ce paragraphe présente la règle générale qui permet, au moment où une mise à jour doit être exécutée, de sélectionner les préconditions à évaluer.

Si mise à jour du type	Alors pour toute précondition du type	Appliquer
MAJ1	PCD1	REGLE1
MAJ2	PCD2	REGLE2
	PCD1	REGLE7
MAJ3	PCD3	REGLE3
MAJ4	PCD4	REGLE4
	PCD1	REGLE9
MAJ5	PCD5	REGLE5
	PCD1	REGLE8
	PCD3	REGLE8
MAJ6	PCD6	REGLE6

Figure n° 6.10 - Règle générale

6.5. Conclusion

Nous achevons ici l'étude des règles de correspondance permettant d'activer une précondition lors d'une mise à jour. Ce travail peut être éclairé de deux points de vue différents. D'un point de vue conceptuel, les règles de correspondance établissent de façon formelle la sémantique des préconditions. D'un point de vue technique elles sont la base du traitement des préconditions : sélection et évaluation (cf. 8.3). Comme nous l'avons vu, trois aspects influencent l'établissement de ces règles : la prise en compte de la hiérarchie de généralisation, l'inclusion de faits et l'inclusion sémantique des opérations. Ces points sont repris dans cette conclusion.

La Hiérarchie de Généralisation

D'un point de vue conceptuel, l'application d'une précondition à travers la hiérarchie de généralisation est un des points importants de ce travail. Nous nous basons sur un modèle sémantique et un langage de haut niveau : les préconditions doivent réfléter ce niveau de conception. A travers ce paragraphe la puissance d'une précondition relativement à son champ d'application a été mise en évidence.

Une bonne partie de ce chapitre a été dédiée à la formalisation de la sémantique qui gouverne l'héritage des préconditions. Soulignons aussi que ceci a été mené en considérant la spécialisation simple mais aussi les trois types de spécialisation composée : union, intersection et différence. Nous considérons ce travail original et général. D'une part, les définitions relatives à la hiérarchie de généralisation (Hiér-A, Sûr-A, Prob-A, Diff-A, Déd-A, Ancêtre-A, Descendant-A) ne dépendent quasiment pas de la syntaxe utilisée. D'autre part, bien que l'écriture des règles de correspondance s'appuie sur le FML, les principes utilisés pour prendre en compte l'aspect hiérarchie de généralisation dans les règles de correspondance sont généraux. L'essentiel de ces principes est :

- L'applicabilité d'une précondition portant sur les manipulations de faits.
- L'applicabilité d'une précondition de création d'une entité.
- L'applicabilité d'une précondition de suppression d'entités
- L'applicabilité d'une précondition de changement de classe.

D'un point de vue technique, ce travail est aussi important car en sélectionnant de façon fine les préconditions à évaluer lors d'une mise à jour, l'évaluation d'un ensemble de préconditions est évitée ce qui entraîne moins d'accès à la base de données.

Dans beaucoup de cas, même s'il n'y a pas de correspondance entre une mise à jour et une précondition, la précondition peut être évaluée (sans provoquer des incohérences) : bien sûr son évaluation aboutira à la valeur fausse. En effet, souvent on aurait pû évaluer, pour une mise à jour donnée, l'ensemble de toutes les préconditions qui sont compatibles avec cette mise à jour vis-à-vis du type de l'opération et des fonctions référencées. Par exemple, si nous avons une précondition d'insertion d'instances du fait [F1, F2, ..., Fn], elle peut être évaluée pour toute mise à jour d'insertion sur le même fait. Ceci sans égard aux rapports hiérarchiques entre les classes où les fonctions désignées dans la précondition et celles désignées dans les mise à jour prennent leurs valeurs.

Mais ce que nous prétendons, entre autre, est de rendre minimal le nombre de préconditions à évaluer ; et pour cela les règles de correspondance sont essentielles. Ainsi, l'application de ces règles dans le traitement d'une précondition mène à une implémentation efficace : elle constitue une étape dans l'évaluation optimisée des préconditions. Dans ce sens, nous disons que les règles de correspondance sont la base du traitement d'une précondition.

Enfin, insistons ici sur le fait que, de ce point de vue, l'utilisateur n'a pas besoin de connaître les règles de correspondance (même si intuitivement elles ne sont pas difficiles à saisir) pour exprimer des préconditions.

L'Inclusion de Faits

Par rapport aux préconditions sur les manipulations des faits, il est intéressant de retenir l'applicabilité d'une précondition sur un fait, aux mises à jour des faits spécialisés. Ceci est un autre aspect qui a dû être pris en compte pour que les préconditions reflètent le niveau conceptuel du modèle FM et de son langage.

L'Inclusion Sémantique des Opérations

La prise en compte de ce facteur dans l'établissement des règles de correspondance est assez particulière au FML (à la sémantique propre de ses énoncés de manipulation) et n'est donc pas applicable à n'importe quel langage. Il est néanmoins intéressant de retenir que cet aspect est à prendre en compte dans n'importe quel système de préconditions pour un modèle sémantique : il facilite l'écriture de préconditions de façon compacte (p.ex., une précondition d'insertion de faits peut être activée par trois types de mises à jour différents).

CHAPITRE 7

7. EVALUATION DES PRECONDITIONS

Le chapitre 6 définit le principe de sélection du sous-ensemble des préconditions qui doivent être évaluées pour une mise à jour particulière. Ceci constitue la première étape de l'évaluation des préconditions. La deuxième partie consiste en l'évaluation proprement dite de la précondition, plus précisément de sa partie qualification. Le but de cette partie est alors d'obtenir l'ensemble des informations (constitué d'entités et/ou faits) qui valident la précondition. La troisième partie consiste en la création et l'habilitation des activités avec leur entité focale et contexte d'activation. Pour accomplir cette partie sont utilisées les informations obtenues dans la partie 2.

Ce chapitre décrit la deuxième et troisième parties de l'évaluation des préconditions.

L'évaluation brutale - en dehors de tout souci d'optimisation - d'un ensemble de préconditions consiste à prendre chaque précondition de façon isolée et à l'évaluer[1] indépendamment de l'exécution[2] de la mise à jour. Ceci mène à évaluer la partie qualification d'une précondition dans l'état de la base de données avant l'exécution de la mise à jour, la réévaluer après et comparer les résultats (pour déduire les informations qui vérifient la précondition et qui sont mises à jour). Une autre solution serait d'ajouter la partie qualification de la précondition avec la partie qualification de la mise à jour et réaliser alors l'évaluation dans l'état de la base de données avant (cf. 4.2.1) la mise à jour.

Il n'est pas difficile de se rendre compte de la charge que représente pour le système l'application de tels mécanismes. Il est alors clair, qu'il faut appliquer des techniques d'optimisation. Ce chapitre décrit comment l'évaluation de la partie qualification des préconditions est réalisée en appliquant des techniques d'optimisation. Pour l'introduire nous donnons une idée générale des principes utilisés dans l'évaluation du sous-ensemble de préconditions lors d'une mise à jour.

Le principe qui est à la base du développement des techniques d'optimisation est que **l'évaluation des préconditions se fasse de façon intégrée à l'exécution de la mise à jour** en vue de diminuer des accès à la base de données. Ainsi, nous divisons les techniques d'optimisation en deux groupes :

- Le premier concerne l'intégration de la précondition et de la mise à jour, pour déduire une expression optimisée de la partie qualification de la précondition. Cette expression permet l'évaluation de la précondition : nous la nommons squelette d'évaluation. Elle est constituée de tests et de requêtes à la base de données.

[1] Dorénavant, dans ce chapitre, le terme évaluation de préconditions fait référence à l'évaluation proprement dite, i.e. après l'application des règles de correspondance.

[2] De même, le terme exécution de la mise à jour est utilisé pour désigner l'exécution proprement dite : après la compilation de la mise à jour.

- Le deuxième concerne l'évaluation de squelettes d'évaluation correspondant au sous-ensemble de préconditions.

Pour **dériver le squelette d'évaluation** d'une précondition, deux techniques d'optimisation sont utilisées :

1. Limiter l'évaluation de la précondition à l'ensemble d'informations qui a été mis à jour. Ceci peut être réalisé sans accès à la base de données, en profitant des informations suivantes :

   - Les valeurs des entités littérales à insérer qui sont contenues dans l'énoncé de mise à jour.

   - Les résultats (encore dans le cache en mémoire centrale) des accès[3] faits à la base de données dans le but d'identifier les entités à mettre à jour. Ces informations sont composées des identificateurs des entités virtuelles et de constantes dans le cas d'entités littérales.
     Ces accès sont réalisés par l'interpréteur de la mise à jour. Néanmoins, dans quelques cas, le gestionnaire de transactions accomplit aussi un ensemble de accès - dans le but de construire le journal de reprise - qui obtient de façon plus précise l'ensemble des entités qui ont réellement été mises à jour. Ceci est le cas, par exemple, d'une mise à jour de suppression d'entités : pour son exécution, l'interpréteur sélectionne les identificateurs des entités à supprimer, le gestionnaire de transactions, sélectionne les identificateurs de toutes les entités qui participent aux faits sur l'entité à supprimer. Ceci peut être intéressant pour évaluer une précondition de suppression de faits sur la même entité.

2. Simplifier des expressions à évaluer en utilisant des informations utilisées dans l'exécution de la mise à jour. Ces informations peuvent être de deux types :

   - Des constantes contenues dans la partie qualification de la mise à jour. Pour donner un exemple de ce type d'optimisation, supposons une précondition devant être évaluée lorsqu'un employé est affecté à un département et qui impose dans sa partie qualification que l'employé appartienne à une certaine catégorie ('ingénieur', 'sécretaire', ...). Supposons aussi une mise à jour qui implique l'évaluation de cette précondition et qui qualifie les employés par leur appartenance à une catégorie particulière qui est désignée dans la mise à jour. Or, cette information peut éviter la sélection de la catégorie des employés lors de l'évaluation de la précondition.

   - Des informations (en mémoire centrale) sélectionnées par le gestionnaire de transactions sur des entités qui ne sont pas les entités qui doivent être mises à jour pour que la précondition soit évaluée. Illustrons ceci par un exemple : supposons une précondition qui doit être évaluée quand un fait entre un département et un employé est supprimé et qui impose dans sa partie qualification que les employés gagnent plus d'une certaine somme. Supposons aussi une mise à jour de suppression d'un employé : au moment de son

[3]Un accès est une requête relationnelle puisque notre système est construit sur un SGBD Relationnel.

exécution, la construction du journal de reprise implique la sélection de tous les faits auxquels l'employé participe. Un de ces faits contient le salaire ; cette information peut être utilisée lors de l'évaluation de la précondition.

En ce qui concerne **l'exécution des squelettes d'évaluation** d'un ensemble de préconditions, deux techniques d'optimisation peuvent être utilisées :

3. Exploitation de sous-conditions équivalentes dans plusieurs préconditions à évaluer pour la même mise à jour. Le but est d'évaluer un seule fois les sous-conditions équivalentes.
   L'intérêt de cette optimisation est qu'au moment de la compilation des préconditions on peut détecter - en utilisant les règles de correspondance - des groupes de préconditions à évaluer pour la même mise à jour. L'application de cette technique se fait à ce moment-là et évite une charge pendant l'exécution des mises à jour.

4. Evaluation des sous-conditions, contenues dans le squelette d'évaluation de la précondition, par ordre croissant de complexité. Ceci évite d'évaluer des conditions plus complexes lorsque des conditions plus simples ne sont pas satisfaites.

Les trois paragraphes suivants s'occupent de l'évaluation des préconditions. En particulier ils abordent :

7.1. La génération du squelette d'évaluation en tenant compte de la technique d'optimisation 1 : i.e., limitation de l'évaluation de la précondition à l'ensemble d'informations mises à jour. La technique 4 est aussi appliquée à ce niveau : elle consiste en l'organisation des tests et des requêtes dans le squelette d'évaluation par ordre croissant de complexité. Ce paragraphe constitue la partie la plus importante de l'évaluation : il présente la méthode de base.

7.2. L'optimisation du squelette d'évaluation en tenant compte de la technique d'optimisation 2 : i.e., tirer profit des informations contenues dans l'énoncé de manipulation et dans le journal de reprise.

7.3. L'exécution d'un ensemble de squelettes d'évaluation en tenant compte de la technique d'évaluation 3 : i.e., exploiter des sous-conditions communes à plusieurs préconditions.

Le paragraphe 7.4 s'occupe de la création et habilitation des activités lors de la validation d'une précondition.

Finalement une conclusion est présentée.

7.1. Génération du Squelette d'Evaluation

La première phase du traitement d'une précondition, pour préparer son évaluation lors des mises

à jour, consiste à dériver à partir de la précondition son squelette d'évaluation constitué de tests et de requêtes à la base de données. Très brièvement, le contexte dans lequel ces requêtes sont posées, inclue trois prémisses :

1. La requête doit sélectionner les entités/faits qui ont été mis à jour et pour lesquels la précondition est vraie.
2. D'après 4.2.1 la requête doit être évaluée avant la mise à jour. En fait, pour les manipulations sur les associations, le seul cas où l'évaluation avant et après peut mener à des résultats différents est celui pour lesquels la qualification de la précondition porte sur l'ensemble des valeurs de la fonction à modifier (cf. 4.2.1).
3. Le but de cette requête est de sélectionner toutes les entités et faits qui ont contribué à la satisfaction de la précondition (cf. 4.2.2) ; la création et l'habilitation des activités avec leur entité focale et contexte d'activation se fera plus tard à partir de ces informations (cf. 7.4).

Ce paragraphe présente donc la génération du squelette d'évaluation, en sachant que nous prétendons limiter l'évaluation de la précondition aux informations mises à jour. Il est divisé en cinq sous-paragraphes correspondant aux différents types de préconditions : les préconditions de modification de faits et les préconditions d'état ne sont pas traitées dans ce rapport.

7.1.1. Précondition d'Insertion de Faits

Dans ce paragraphe nous supposons tout d'abord que les préconditions d'insertion de faits sont activées par des mises à jour du même type. La stratégie pour générer les tests et les requêtes qui constituent le squelette d'évaluation dépend du type des préconditions : plusieurs cas sont donc à considérer. Dans la suite ils seront énumérés et illustrés par des exemples.

CAS 1 - Précondition sur des entités virtuelles qualifiées de façon isolée

Le but de l'évaluation de la précondition p1 ci-dessous, est de détecter les employés ingénieurs et les départements administratifs qui sont associés lors d'une mise à jour.

```
p1 :    TEST    INSERT  [Employé e, Département d]
                FROM    Employé e, Département d
                WHERE   e. Catégorie = 'Ingénieur'
                AND     d. Groupe = ' Administratif'
```

Les deux requêtes suivantes permettent d'évaluer p1.

```
r1 :            SELECT  <Identificateurs des employés>                  a1
                WHERE   <Qualification de la précondition
                        concernant les employés>                        b1
                AND     <Qualification des employés à mettre à jour>    c1
```

r2 :	SELECT	<Identificateurs des départements>	*a2*
	WHERE	<Qualification de la précondition concernant les départements>	*b2*
	AND	<Qualification des départements à mettre à jour>	*c2*

La requête r1 (r2) sélectionne les employés (départements) qui sont associés lors d'une mise à jour et qui contribuent à la satisfaction de p1. Remarquons que tous les employés obtenus par l'évaluation de r1 sont associés à tous les départements obtenus par l'évaluation de r2 (comme conséquence de la mise à jour). Nous connaissons alors toutes les instances du fait [Employé,Département] qui valident la précondition.

Détaillons alors les différentes clauses de ces requêtes. Dans notre système (SIB) toute entité virtuelle est identifiée de façon interne par un identificateur unique que nous appelons surrogate. Au niveau interne ce surrogate peut être accessible, e.g., l'expression "Employé.Surr" fait référence aux surrogates de tous les employés. Ainsi, *a1* et *a2* peuvent être exprimés sans problème.

Le but de *c1* et *c2* est justement de limiter l'évaluation de la précondition à l'ensemble d'informations à mettre à jour. Pour étudier ces alinéas, il faut savoir que dans le cas général, une mise à jour FML est toujours compilée en deux types d'opérations primitives sur la base de données:

(1) Des **opérations de sélection** pour obtenir les entités à mettre à jour.

(2) Une **opération de modification** pour réaliser la mise à jour.

Par exemple l'énoncé "INSERT [Employé e, Département d] ..." est traduit en : (1) une requête permettant l'identification des employés à mettre à jour et une autre pour les départements ; (2) Une modification, pour ajouter les associations à la base de données. Lors de l'interprétation, les opérations de sélection sont exécutées et les entités à mettre à jour sont ainsi obtenues.

Le premier profit qu'on peut tirer du fait d'intégrer l'évaluation de la précondition à l'exécution de la mise à jour est de pouvoir consulter les informations obtenues lors de l'exécution de la partie (1) de la mise à jour.

Le résultat des requêtes (1) faites à la base de données est, dans le cas d'entités virtuelles, un ensemble de surrogates d'employés et un ensemble de surrogates de départements, que nous notons: $\{Surr_j\text{-Maj-e}\}_{j\geq 1}$, $\{Surr_j\text{-Maj-d}\}_{j\geq 1}$ où e et d font référence respectivement aux employés et aux départements .

Ainsi, *c1* peut être exprimé par le fait que les surrogates des employés à sélectionner soient contenus dans $\{Surr_j\text{-Maj-e}\}_{j\geq 1}$. De même pour *c2* et les départements. Les requêtes s'écrivent alors :

r1.1 :	SELECT	e.Surr
	FROM	Employé e
	WHERE	e.Catégorie = 'Ingénieur'
	AND	e.Surr IN $\{Surr_j\text{-Maj-e}\}$

```
r1.2 :      SELECT  d.Surr
            FROM    Département d
            WHERE   d.Groupe = 'Administratif'
            AND     d.Surr IN {Surr_j-Maj-d}
```

Ces requêtes doivent s'exécuter après la partie (1) de la mise à jour et, dans le cas de cette précondition, indifféremment avant ou après la partie (2). Néanmoins, comme pour certaines préconditions l'évaluation doit se faire impérativement avant la partie (2) il a été décidé d'évaluer toute précondition avant (2).

Le squelette d'évaluation de p1 est donc constitué de ces deux requêtes. Les ensembles de surrogates référencés dans r1.1 et r1.2 sont en fait, des paramètres qui seront instanciés lors de l'évaluation de la précondition pour une mise à jour particulière.

Dans cet exemple, pour chaque entité virtuelle participant au fait, une requête a été générée avec sa partie qualification constituée da la partie qualification de la précondition relative à cette entité. Ceci a pû être fait car chaque entité a été qualifiée de façon isolée, i.e., chaque condition dans la clause "WHERE" ne concernait qu'une des entités du fait (soit employé, soit département). Néanmoins, il y a des cas où cette méthode n'est pas efficace. Il s'agit des cas où une expression de la qualification de la précondition concerne plusieurs entités participant au fait désigné dans la clause "INSERT".

CAS 2 - Précondition portant sur des entités virtuelles qualifiées ensemble

La précondition suivante s'intéresse aux employés et aux départements associés lors d'une mise à jour, quand ces employés ont un salaire qui dépasse les dix pourcent du budget du (nouveau) département.

```
p2 :    TEST    INSERT  [Employé e, Département d]
                FROM    Employé e, Département d
                WHERE   e.Salaire * 10 > d. Budget
```

Dans p2 l'expression "e.Salaire * 10 > d. Budget" qualifie simultanément deux entités du fait à insérer : les employés et les départements. La façon correcte de l'évaluer est d'exécuter la requête suivante :

```
r2 :        SELECT  e. Surr, d.Surr
            FROM    Employé e, Département d
            WHERE   e.Salaire * 10 > d. Budget
            AND     e.Surr IN {Surr_j-Maj-e}
            AND     d.Surr IN {Surrj-Maj-d}
```

De façon générale, pour toute précondition ne portant que sur des entités virtuelles, il est possible de générer une seule requête permettant de l'évaluer : cette requête serait du type r2. Par exemple, pour évaluer p1 nous aurions pû très bien générer une seule requête à la base de données qui qualifie, dans sa clause "WHERE", les employés et les départements. Néanmoins, ceci serait moins efficace, en particulier, le système SIB est construit sur un SGBD relationnel et les requêtes FMQL sont directement traduites en SQL. Ainsi, faire évaluer p1 par une seule requête impliquerait l'évaluation d'un produit cartesien sur les relations représentant respectivement les employés et les départements. La solution consistant à évaluer p1 par r1.1 et r1.2 est donc optimale.

En conséquence, même s'il existe une stratégie générale pour évaluer ce type de préconditions, il est préférable d'évaluer les conditions sur une des entités participant au fait de la clause "INSERT" de façon isolée chaque fois que c'est possible.

Remarquons que dans le cas général d'une précondition on peut avoir un groupe d'entités qui sont qualifiées ensemble et d'autres qui sont qualifiées de façon isolée. Dans ce cas, le même principe est appliqué : nous générons une requête pour chaque entité qualifiée de façon isolée et une requête par groupe d'entités qualifiées simultanément.

CAS 3 - Précondition portant sur une entité littérale qualifiée de façon isolée

3(a) - L'entité littérale est comparée à une valeur déjà stockée dans la base de données

La précondition p3 doit être évaluée comme vraie, chaque fois qu'on affecte à un Employé un Salaire égal à celui de Jean.

```
p3 :    TEST    INSERT  [Employé e, Salaire s]
                FROM    Employé e, Employé e1.Salaire s
                WHERE   e1.Nom ='Jean'
```

D'abord remarquons qu'aucune expression dans la partie qualification ne porte sur les employés à mettre à jour (i.e., sur la variable e). Ainsi, il n'est pas nécessaire de générer une requête à la base de données concernant ces employés : tous les employés qui ont été mis à jour contribuent à la validation de la précondition.

En ce qui concerne les salaires à affecter il faut voir lesquels participent à la validation de la précondition. Pour cela il faut vérifier si le salaire affecté aux employés lors d'une telle mise à jour est égal à celui de Jean.

Les salaires affectés aux employés lors d'une mise à jour sont, soit contenus dans l'énoncé de la mise à jour, soit obtenus par l'interpréteur comme résultat de l'exécution de la partie (1) de la mise à jour. Dans le cas des entités littérales, le résultat d'une requête (1) est un ensemble de constantes que nous notons $\{K_j\text{-Maj-v}\}_{j\geq 1}$ où v est le nom d'une variable qui désigne l'entité littérale ; nous avons j=1 si la fonction est monovaluée.

Pour une mise à jour qui provoque l'évaluation de p3, l'interpréteur a alors comme information le salaire à mettre à jour : noté K_1-Maj-s.

La requête générée à la base de données pour évaluer cette précondition est la suivante .

```
r3 :        SELECT  e1.Salaire
            FROM    Employé e1
            WHERE   e1.Nom = 'Jean'
            AND     e1.Salaire = K1-Maj-s
```

3(b) - L'entité littérale est comparée à une valeur constante

Dans p3 la valeur des salaires à affecter aux employés lors d'une mise à jour est qualifiée par comparaison à des valeurs déjà stockées dans la base de données, qui sont désignées dans l'énoncé par une expression qui permet de les retrouver. Par ailleurs, une précondition peut qualifier le salaire affecté aux employés par des constantes désignées directement dans l'énoncé. La précondition p4 en est un exemple : elle est évaluée comme vraie si le salaire affecté dans une mise à jour est supérieur à 10 000 F.

```
p4 :    TEST    INSERT  [Employé e, Salaire s]
                FROM    Employé e, Employé e1.Salaire s
                WHERE   s > 10000
```

L'évaluation de p4 se fait sans accès à la base de données : il suffit de comparer le salaire affecté dans la mise à jour à 10000. Dans ce cas, nous générons un test permettant de filtrer les salaires mis à jour.

```
Test produit :      K1-Maj-s > 10000
```

CAS 4 - Précondition concernant une entité virtuelle et une littérale qualifiées simultanément

4(a) - La fonction sur l'entité littérale est monovaluée

La précondition p5 est évaluée comme vraie lors d'une mise à jour qui affecte un budget à un département si ce budget est inférieur à 10 fois le salaire d'un des employés du département. Nous supposons que la fonction Budget(Département) est monovaluée.

```
p5 :    TEST    INSERT  [Département d, Budget b]
                FROM    Département d , Département d1. Budget b
                WHERE   d.Employé.Salaire * 10 > b
```

L'expression "e.Salaire * 10 > b" qualifie deux entités du fait à mettre à jour (tout comme pour p2). Néanmoins, lors d'une mise à jour nous connaissons la valeur constante qui instancie la variable b : K_1-Maj-b.

Ainsi, l'expression ci-dessus va comparer le salaire des employés des départements à mettre à jour et une constante : en conséquence nous pouvons considérer que cette expression ne qualifie

qu'une entité du fait, et générer la requête r5.1 pour son évaluation.

r5.1 :
```
SELECT  d1.Surr
FROM    Département d1
WHERE   d1.Employé.Salaire * 10 > K1-Maj-b
```

4(b) - La fonction sur l'entité littérale est multivaluée

Dans le cas où la fonction Budget (Département) est multivaluée l'évaluation de p5 est plus complexe. Le but de l'évaluation serait de détecter les **paires** (département, budget) qui seront mis à jour et qui vérifient la partie qualification de la précondition. Si l'évaluation se faisait après la mise à jour les budgets seraient stockés dans la base de données et nous aurions produit la requête r5.2.

r5.2 :
```
SELECT  d.Surr, d.Budget
FROM    Département d
WHERE   d.Employé.Salaire * 10 > d.Budget
AND     d.Surr IN {Surrj-Maj-d}
AND     d.Budget IN {Kj-Maj-b}
```

Néanmoins, il y a certains cas où il est faux d'évaluer la précondition après la mise à jour. Et pour être homogène avec l'évaluation d'autres types de précondition, l'évaluation se fera toujours avant l'exécution de la partie (2) d'une mise à jour.

La solution qui a été retenue, est de créer pour ces préconditions une classe d'entités de travail où on puisse stocker les entités littérales à insérer (dans ce cas les budgets). La création d'une telle classe d'entités peut se faire une fois pour toutes, au moment de la préparation de la précondition à l'évaluation (compilation de la précondition, cf. 9). L'instantiation de cette classe se fait au moment de l'évaluation de la précondition. Supposons alors que nous créons la classe T1 ayant comme association [T1, Budget : Integer] et que nous l'instancions avec {K_j-Maj-b}. Dans ce contexte, nous pouvons évaluer la précondition en exécutant la requête r5.3 .

r5.3 :
```
SELECT  d.Surr, t.Budget
FROM    Département d , T1 t
WHERE   d.Employé.Salaire * 10 > t.Budget
AND     d.Surr IN {Surrj-Maj-e}
```

Evaluer r5.3 est même plus efficace qu'évaluer r5.2 car nous travaillons avec une relation qui risque d'être de taille très petite. En effet, dans la plupart des cas on n'associe jamais un ensemble très grand d'entités littérales à une entité virtuelle.

La solution qui vient d'être illustrée est, en fait, la solution générale pour l'évaluation de toute précondition concernant des entités littérales. En effet, pour évaluer p4 et p5 (dans le cas où

Budget(Département) est monovaluée) nous aurions pû appliquer la même solution de création d'une entité de travail. Néanmoins, dans ces cas particuliers ceci s'est avéré inutile : une solution optimale (qui évite tout accès à la base de données) peut être utilisée.

Supposons alors la partie qualification d'une précondition concernant une entité littérale. La stratégie pour générer le squelette d'évaluation serait de créer pour chaque condition qui compare l'entité littérale à une constante, un test filtre. Toutes les autres conditions seraient incluses dans une seule requête à la base de données. En effet, il est préférable de générer une seule requête, vu que les différentes conditions portent toutes sur la même entité. Lors d'une mise à jour, les tests seraient d'abord évalués et ensuite les requêtes exécutées : bien sûr, les requêtes ne porteraient que sur les entités littérales filtrées par les tests.

Or, ceci n'est pas toujours possible. Par exemple, supposons que la partie "WHERE" d'une précondition soit composée de trois conditions - c1, c2 et c3 - portant toutes sur la même entité littérale. L'expression de la clause "WHERE" est :

(c1 ou c2) et c3

Nous supposons aussi que pour évaluer c2 et c3 des accès à la base de données soient nécéssaires. Dans ce cas, il n'y a pas de sens à générer un test pour c1 et une seule requête pour c2 et c3 : l'évaluation serait faussée. En conséquence, pour un tel cas, il a été décidé que c1, c2 et c3 seraient évalués ensemble par une seule requête. Cette requête fera référence à une entité de travail pour évaluer c1.

CAS 5 - Précondition concernant deux entités littérales qualifiées ensemble

Après avoir étudié les cas précédents, ce cas-ci ne présente plus de mystère. En effet :

(a) Si la condition qualifiant simultanément les deux entités littérales est du type "v1 opérateur v2" (où v1 et v2 désignent les entités littérales), alors un test filtre est généré (dans la mesure du possible) pour l'évaluer.

(b) Si la condition qualifiant simultanément les deux entités littérales est du type "v1 opérateur expression" (où expression qualifie la valeur d'une des entités littérales), alors le problème est résolu comme pour 4(a) et 4(b).

(c) Si la condition qualifiant simultanément les deux entités littérales est du type "expression1 opérateur expression2" (où expression1 et expression2 qualifient respectivement les deux entités littérales), alors une requête à la base de données doit être générée en sélectionnant les valeurs des deux entités littérales qui sont qualifiées par cette condition.

Les différents cas que nous venons d'étudier ont permis d'illustrer les stratégies à appliquer pour générer le squelette d'évaluation d'une précondition, ainsi que les critères qui permettent de décider de la bonne stratégie. Dans la suite, est décrite de façon précise la dérivation du squelette d'évaluation à partir d'une précondition. Dans ce rapport nous supposons que la clause "WHERE" est composée d'un ensemble de conditions liées uniquement par des opérateurs de conjonction. Cette restriction permet de mieux mettre en évidence les différentes stratégies utilisées, et de simplifier l'écriture de la règle permettant de générer le squelette d'évaluation. Par exemple, comme il a déjà été remarqué (cf. fin cas 4(b)) avec cette restriction, il est possible de créer un test filtre pour

toutes les conditions qui comparent une entité littérale à insérer et une constante, sans se soucier si cette condition est à l'intérieur d'une expression entre parenthèses et doit être évaluée avec un autre type de condition, etc.

A la fin des différentes définitions nous ajoutons des commentaires en indiquant les modifications à apporter à la règle de génération, dans le cas où la clause "WHERE" est une expression booléenne (en utilisant les opérateurs de conjonction, disjonction et des expressions entre parenthèses) constituée de conditions.

Définitions

Reprenons le fait F1 de 6.1.1.

F1 : [F1: E1, F2 : E2, ..., Fi : Ei, ..., Fn : En]

Et la précondition PCD1, plus détaillée[4] ici:

PCD1 : TEST INSERT [H1 v_1, H2 v_2, ..., Hi v_i, ..., Hl v_l]
FROM Déf-v_1, Déf-v_2, ..., Déf-v_i,..., Déf-v_l

où $\{Fi\}_{1\leq i\leq n} \supseteq \{Hi\}_{1\leq i\leq l}$

et Déf-v_i, $1\leq i\leq l$, définit la classe d'entités (ou un sous-ensemble de la classe d'entités) où la variable vi prend ses valeurs. Déf-v_i peut avoir une des deux formes suivantes :

- La désignation d'une classe d'entités qui est un Hiér de la classe sur laquelle Hi est définie.
- Une composition de fonctions sur une classe d'entités où la fonction qualifiant v_i a comme ensemble d'arrivée une classe qui est un Hiér de la classe sur laquelle Hi est défini.

Ces deux formes peuvent avoir la même expression quand on pense en termes purement fonctionnels. En effet, la déclaration d'une classe d'entités définit implicitement une fonction à zéro argument. Par exemple, "Employé : Entity" déclare la classe d'entités Employé et la fonction Employé() ayant comme ensemble d'arrivée Entity (la classe de toutes les entités de la base de données)[5].

Un exemple de Déf-v_i est :

$$Ki_1\ v_{i1} \,.\, Ki_2\ v_{i2} \,.\, \mathbf{Ki_3\ v_i} \,.\, Ki_4\ v_{i4} \,.\, Ki_5\ v_{i5}$$

où Kij est une fonction et v_{ij} est une variable.

Ici, la variable v_i prend ses valeurs dans l'ensemble d'arrivée de la fonction Ki_3.

Ki_3 est définie sur une composition de fonctions (les fonctions à sa gauche dans l'expression ci-dessus). Le nombre des fonctions à gauche de Ki_3 est désigné dans la suite par "r-1" (pour cet exemple r = 3).

Une composition de fonctions est définie sur Ki_3 (les fonctions à sa droite). Le nombre total de

[4] Remarquons que dans le cas général, il est possible de déclarer dans la clause "FROM" des variables qui ne sont pas liées aux variables de la clause "INSERT". Ceci est utilisé pour des énoncés assez complexes. Nous ne prenons pas en compte ce type de déclarations. D'une part, rien ne serait changé dans principe d'évaluation ; d'autre part, les définitions seraient alourdies.

[5] Seules les entités retournées par la fonction Employé() appartiennent à la classe Employé. Ceci est le même point de vue que DAPLEX [Shi81].

fonctions de Déf-v_i est désigné dans la suite par "s" (dans l'exemple s = 5).

Remarquons que dans cet exemple nous avons lié une variable à chaque fonction de la composition. Dans la suite, nous supposons aussi que chaque fonction désignée dans la clause "FROM" est liée à une variable.

Ainsi, $\forall 1 \leq i \leq l$ | Déf-v_i est de la forme :

si r =1 et s =1, Kir v_i

si r >1 et s = r, Comp-Ki1→Kir-1 . Kir v_i

si r =1 et s ≥1, Kir v_i . Comp-Kir+1→Kis

si r >1 et s > r, Comp-Ki1→Kir-1 . Kir v_i . Comp-Kir+1→Kis

où si x > y, Comp-Kix→Kiy = Comp-Kix→Kiy-1 . Kiy v_{iy}

si x = y, Comp-Kix→Kiy = Kix v_{ix}

et $\forall 1 \leq j \leq s$ | (Kij est une fonction $\wedge$ v_{ij} est une variable)

et $\exists 1 \leq t \leq n$ | (Hi ≡ Ft $\wedge$ l'ensemble d'arrivée de Kir est un Hiér-Et)

Ceci impose que la variable vi prenne des valeurs dans une classe qui est un Hiér de la classe sur laquelle Hi est définie

Nous allons maintenant analyser la clause "WHERE" d'une précondition pour permettre de définir les critères qui impliquent le choix d'une stratégie pour la génération du squelette d'évaluation.

La clause "WHERE" est une conjonction de conditions qui portent sur les variables déclarées dans la clause "FROM" de la précondition. Ces conditions sont notées "c_i". Chaque c_i est de la forme suivante :

(a) y.G1. ... .Gt op1 valeur

(b) y.G1. ... .Gt op1 x.L1. ... .Ls

(c) y.g1. ... .gt op2 expr-globale

où $\exists 1 \leq i \leq n$ | ($y \equiv v_i \vee y \equiv v_{ij}$)

et $\exists 1 \leq i \leq n$ | ($x \equiv v_i \vee x \equiv v_{ij}$)

et $\forall 1 \leq i \leq t$ | Gi est une fonction

et $\forall 1 \leq i \leq s$ | Li est une fonction

et op1 est un opérateur scalaire ou ensembliste

et op2 est l'opérateur d'égalité ou d'appartenance à un ensemble

et valeur est une constante ou un ensemble de constantes

et expr-globale est une expression qui n'est pas liée aux variables déclarées dans la clause FROM, et qui est du type "H1. ... . Hs WHERE <conditions sur Hi>"

Un ensemble de définitions suit.

Définition de $c_j(v_i)$

$c_j(v_i)$ est une condition (appartenant à la clause "WHERE" d'une précondition) qui qualifie une entité participant au fait (de la clause "INSERT") et qui est désignée par la variable v_i. En, considérant les trois formes d'une condition, nous disons qu'une condition qualifie vi ssi:

$$y \equiv v_i \vee y \equiv v_{ik} \vee x \equiv v_i \vee x \equiv v_{ik}$$

Définition de $c_j(v_i, v_t)$

$c_j(v_i,v_t)$ est une condition qui qualifie simultanément deux entités participant au fait et qui sont respectivement désignées par les variables v_i et v_t. Nous avons $c_j(v_i,v_t)$ ssi

$$c_j(v_i) \wedge c_j(v_t)$$

Définition de $c_j[v_i]$

$c_j[v_i]$ est une condition qui qualifie de façon isolée une des entités participant au fait et qui est désignée par la variable v_i. Une condition est un $c_j[v_i]$ ssi

$$c_j \text{ est de la forme (b)} \wedge c_j(v_i) \wedge \forall s \neq i \mid \text{NON}(y \equiv v_s \vee y \equiv v_{sk} \vee x \equiv v_s \vee x \equiv v_{sk})$$

Définition de $\text{Groupe}_j(v)$

Groupej(v) est défini par un ensemble d'entités participant au fait et désignées respectivement par un ensemble de variables vi, tel que la propriété suivante est vérifiée :

$$\forall v_i \in \text{Groupe}_j(v)\ \exists v_t \in \text{Groupe}_j(v)\ \exists c_k \mid c_k(v_i,v_t)$$

Définition de Isolé(v)

Isolé(v) est l'ensemble des entités participant au fait qui sont qualifiées de façon isolée. Une entité désignée par une variable v_i appartient à Isolé(v) ssi toutes les conditions qui la qualifient le font d'une façon isolée, i.e.

$$\forall j \geq 1 \mid c_j(v_i) \Rightarrow c_j[v_i]$$

Définition de $c(v_i)$

$c(v_i)$ est l'ensemble de toutes les conditions c_j qui qualifient la variable vi. Nous avons alors la propriété suivante :

$$\forall j \geq 1 \mid c_j(v_i) \in c(v_i)$$

Définition de $c(\text{Groupe}_j)$

$c(\text{Groupe}_j)$ est l'ensemble de toutes les conditions c_j qui qualifient au moins une des entités (des variables v_i) qui appartiennent à $\text{Groupe}_j(v)$.

La règle qui permet de générer le squelette d'évaluation d'une précondition d'insertion de faits peut alors être établie.

REGLE DE GENERATION DU SQUELETTE D'EVALUATION

POUR TOUT $v_i \in$ Isolé(v) **FAIRE**
  SI v_i désigne une entité virtuelle
  ALORS Générer REQUETE-A *cas1*
  SINON *% vi désigne une entité littérale %*
    POUR TOUT $c_j(v_i) \in c(v_i)$ de la forme "v_i op1 valeur" **FAIRE**
      Générer FILTRE-B *cas3(b)*
      Enlever $c_j(v_i)$ de $c(v_i)$
    Générer REQUETE-C sur $c(v_i)$ *cas3(a)*

POUR TOUT $Groupe_j$(v) **FAIRE**
  SI Tout $v_i \in Groupe_j$(v) désigne une entité virtuelle
  ALORS GénérerREQUETE-D *cas2*
  SINON *%il existe des entités littérales%*
    POUR TOUT $c_j[v_i]$ de la forme "v_i op valeur" **FAIRE**
      Générer FILTRE-B *cas3(a)*
      Enlever la condition de $c(Groupe_j)$
    POUR TOUT $c_j(v_i,v_u)$ de la forme "v_i op v_u" **FAIRE**
      Générer FILTRE-E *cas5(a)*
      Enlever la condition de $c(Groupe_j)$
    Générer REQUETE-F sur $c(Groupe_j)$ *cas4(a) 4(b) 5(b) 5(c)*

ORGANISER-FILTRE-REQUETE
TRAITER-RESULTAT

Détaillons les expressions de cette règle qui sont soulignées.

REQUETE-A

Cette requête est illustrée par r1.1 et r1.2 générées pour accomplir l'évaluation de p1.

SELECT	v_i.Surr,
	v_{ir-1}.Surr, ..., v_{i1}.Surr,
	V_{ir+1}.Surr, ..., v_{is}.Surr
FROM	Déf-v_i
WHERE	$c(v_i)$
AND	v_i.Surr IN {$Surr_j$-Maj-v_i}

Tous les surrogates des entités désignées dans Déf-v_i par des variables sont sélectionnés dans la clause "SELECT". Ceci est fait dans le but de construire le contexte d'activation des activités créées comme conséquence de cette évaluation.

FILTRE-B

Le but du test à générer est de filtrer l'ensemble de constantes qui vont être instanciées par la variable v_i. Cet ensemble est $\{K_j\text{-Maj-}v_i\}$. Ainsi, l'entrée du test est cet ensemble, la sortie est le même ensemble diminué des constantes qui ne vérifient pas la condition.

REQUETE-C

Cette requête est illustrée par r3 créée à partir de p3.

Déf-v_i et les conditions $c(v_i)$ doivent subir quelques modifications pour que la requête puisse être générée.

Tout d'abord, dans un énoncé d'interrogation FML, nous ne pouvons pas lier une variable à une entité littérale. Ainsi, Déf-v_i devient :

Comp-$K_{i1} \rightarrow K_{ir-1}$

De plus, toute condition de la forme "vi op expression" devient de la forme :

v_{ir-1}.Hi op expression

Hi est la fonction de la clause "INSERT" qui est liée à la variable v_i.

La forme de la REQUETE-C peut alors être établie.

SELECT	v_{ir-1}.Hi,
	v_{ir-1}.Surr, ..., v_{i1}.Surr,
FROM	Déf-v_i
WHERE	$c(v_i)$
AND	v_{ir-1}.Hi IN $\{K_j\text{-Maj-}v_i\}$

REQUETE-D

Cette requête est illustrée par r2 créée à partir de p2.

Pour éviter de renommer les variables supposons que l'ensemble des v_i appartenant à $Groupe_j(v)$ est : v_1, v_2, .,v_t. Supposons aussi que la cardinalité de cet ensemble est désignée par "t".

SELECT	v_1.Surr,
	v_{1r-1}.Surr, ..., v_{11}.Surr,
	V_{1r+1}.Surr, ..., v_{1s}.Surr,
	...
	v_t.Surr,
	v_{tr-1}.Surr, ..., v_{t1}.Surr,
	V_{tr+1}.Surr, ..., v_{ts}.Surr
FROM	Déf-v_1, ..., Déf-v_t
WHERE	$c(v_1)$ ... $c(v_l)$
AND	v_1.Surr IN {$Surr_j$-Maj-v_1}
AND	...
AND	v_t.Surr IN {$Surr_j$-Maj-v_t}

Dans la clause "SELECT" sont référencés les surrogates des entités participant au fait, et des entités qui leur sont liées dans la clause "FROM" : elles constituent le contexte d'activation.

Remarquons que, à chaque v_i, correspondent des valeurs de r et de s différentes mais toujours dans le souci de ne pas alourdir l'écriture nous avons préféré ne pas faire des sous-sous- ... -sous-indices !

FILTRE-E

Le but de ce test est de filtrer les ensembles de constantes qui vont instancier les variables v_i et v_u. Ces ensembles sont {K_j-Maj-v_i} et {K_j-Maj-v_u}. Les entrées du test sont alors ces ensembles, les sorties sont les mêmes ensembles diminués des constantes qui n'ont pas vérifié la condition. Brièvement, le test réalise le produit cartésien des deux ensembles pour évaluer la condition.

REQUETE-F

Cette requête est illustrée par r5.1 et r5.3 créées pour évaluer p5.

Comme pour la génération de REQUETE-C, les Déf-v_i et les $c(v_i)$ correspondant à des entités littérales doivent subir des modifications.

Supposons toute variable v_i désignant des entités littérales : deux cas sont à considérer selon que la fonction Hi (liée à v_i) est monovaluée ou multivaluée.

Si Hi est monovaluée (cf. r5.1) les modifications sont les mêmes que pour la REQUETE-C. C'est-à-dire, Déf-vi devient Comp-$Ki_1 \rightarrow Ki_{r-1}$; et toute $c_j(v_i)$ de la forme "v_i op expression" devient de la forme "v_{ir-1}.Hi op expression".

Si Hi est multivaluée (cf. r5.3) alors il faut créer une classe d'entités de travail : nommons-la "Ti". Les déclarations suivantes sont alors générées :

Ti : Entity ;

[Ti , Hi] ;

Cette association doit être instanciée chaque fois que la précondition est évaluée. Pour cela nous générons :

INSERT Ti
WITH Hi = {K_j-Maj-v_i}

A la fin de l'évaluation de la précondition, les instances de la précondition sont supprimées de la base de données. Pour cela l'énoncé suivant est généré :

DELETE Ti

Déf-vi devient : Comp-$Ki_1 \rightarrow Ki_{r-1}$, Ti t_i

où t_i est une variable. Dans r5.3, Déf-v_i a été réduit à Ti t_i : en effet, dans cet exemple les conditions $c(v_i)$ n'étaient posées que sur la variable v_i (budget b), elles ne concernaient pas les variables v_{ij} (e.g. département d1). Dans le cas général, nous pouvons trouver des conditions sur v_i (qui seront dirigées vers l'entité Ti) et des conditions sur les v_{ij} ; en conséquence nous devons garder la partie Comp-$Ki_1 \rightarrow Ki_{r-1}$ de Déf-v_i.

Toute $c_j(v_i)$ de la forme "v_i op expression" devient de la forme :

t_i.Hi op expression

De plus, nous devons faire le lien entre les variables v_{ij} et la variable t_i. Pour cela, une condition est ajoutée à la clause "WHERE" :

t_i.Hi = v_{ir-1}.Hi

Après les modifications sur les conditions portant sur des entités littérales correspondant à des fonctions monovaluées, il se peut que, pour certaines de ces entités, plus aucune condition ne portent sur elles. Par exemple, dans r.5.1 aucune condition ne porte sur budget. Ces entités ne seront alors pas référencées dans la requête que nous allons générer.

Ainsi, pour éviter de renommer des variables nous supposons que les entités littérales sur lesquelles il y a encore des conditions les qualifiant, sont désignées par v_i où $1 \leq i \leq u$ si Hi est multivaluée sinon $y \leq i \leq x$ avec $y > u$, et pour les entités virtuelles $z \leq i \leq t$, avec $z > x$.

SELECT	t_1.H1,	----------	
	v_{1r-1}.Surr, ..., v_{11}.Surr,	|	*informationssur les*
	...	|	*entités littérales*
	t_u.Hu,	|	*avec Hi multivaluée*
	v_{ur-1}.Surr, ..., v_{u1}.Surr,	----------	
	v_{yr-1}.H_y,	----------	
	v_{yr-1}.Surr, ..., v_{y1}.Surr,	|	*informations sur les*
	...	|	*entités littérales*
	v_{xr-1}.H_x	|	*avec Hi monovaluée*
	v_{xr-1}.Surr, ..., v_{x1}.Surr	----------	
	v_z.Surr,	----------	
	v_{zr-1}.Surr, ..., v_{z1}.Surr,	|	
	z_{r+1}.Surr, ..., v_{zs}.Surr,	|	*surrogates*
	...	|	*des*
	v_t.Surr,	|	*entités virtuelles*
	v_{tr-1}.Surr, ..., v_{t1}.Surr,	|	
	V_{tr+1}.Surr, ..., v_{ts}.Surr	----------	
FROM	Déf-v_1, ..., Déf-v_u,		
	Déf-v_y, ..., Déf-v_x,		
	Déf-v_z, ..., Déf-v_t		
WHERE	$c(v_1)$ ... $c(v_u)$		
AND	$c(v_y)$ ... $c(v_x)$		
AND	$c(v_z)$ ... $c(v_t)$		
AND	t_1.H1 = v_{1r-1}.H1	----------	*lien entre les entités de*
	...	|	*travail et Déf-v_i*
AND	t_u.Hu = v_{ur-1}.Hu	----------	
AND	v_{yr-1}.H_y IN {K_j-Maj-v_y}	----------	*conditions sur les entités*
	...	|	*avec Hi monovaluée*
AND	v_{xr-1}.H_x IN {K_j-Maj-v_x}	----------	
AND	v_z.Surr IN {$Surr_j$-Maj-v_z}	----------	*conditions sur les*
	...	|	*entités virtuelles*
AND	v_t.Surr IN {$Surr_j$-Maj-v_t}	----------	

ORGANISER-FILTRE-REQUETE

La règle de génération du squelette d'évaluation génère un ensemble de tests filtres et de requêtes: il faut alors les ordonner. Comme la clause "WHERE" d'une précondition n'est constituée que des conjonctions de conditions, l'ordre dans lequel les différents filtres et requêtes sont évalués n'est pas important vis-à-vis du résultat final. Par contre, il est important relativement au fait d'optimiser le

temps d'évaluation des préconditions. En effet, il vaut mieux évaluer les tests et les requêtes par ordre croissant de complexité. Ceci a deux avantages. Le premier est que, au cas où un des tests ou des requêtes échoue, nous évitons l'évaluation des restants qui est plus coûteuse. Le deuxième est que l'évaluation d'un test ou d'une requête élimine des ensembles d'entités à mettre à jour, celles qui ne vérifient pas la précondition ; ainsi, les requêtes plus complexes sont évaluées sur des ensembles d'informations plus petits.

La complexité des conditions contenues dans une précondition est mesurée après l'application des techniques d'optimisation. La complexité est classifiée en tenant compte du temps d'évaluation des conditions. Tout d'abord, les tests filtres sont moins complexes que les requêtes car ils ne nécessitent pas d'accès à la base de données. Ainsi, ils seront évalués avant les requêtes.

Les tests filtres sont de la forme "v op valeur" où v est une variable désignant une entité littérale. Leur complexité dépend de :

- v est liée à une fonction monovaluée ou multi-valuée.
- "op" est un opérateur scalaire ou ensembliste (et en conséquence "valeur " est une constante ou un ensemble de constantes).

Par ailleurs, une requête est, d'après notre implémentation, traduite en SQL. Ainsi, les critères pour définir la complexité d'une requête SQL(e.g., nombre de jointures) peuvent s'appliquer ici.

TRAITER-RESULTAT

Le résultat de l'exécution du squelette d'évaluation est une expression entre parenthèses qui reflète une liste des instances de chemins du schéma conceptuel qui vérifient la précondition. Ceci est aussi la forme du résultat d'une requête FML. Quand le squelette est constitué de plusieurs tests filtres et requêtes, les différents résultats doivent être traités de façon à former une seule liste. En fait, ce sujet est traité plus en détail dans le paragraphe sur la création des activités (cf. 7.4).

Clause "WHERE" Comme une Expression Booléenne

Quelques commentaires suivent indiquant brièvement les modifications à apporter à la règle de génération du squelette d'évaluation pour prendre en compte n'importe quelle expression booléenne dans la clause "WHERE".

- Le FILTRE-B ne peut pas être généré pour toutes les conditions de la forme "vi op valeur" (cf. fin cas 4(b)). Ainsi, REQUETE-C qui prend en compte des conditions du type "vi op expression" doit tenir compte également des conditions du type "vi op valeur". Pour cela, il faut créer une entité de travail, comme il a été fait pour la REQUETE-F.
  Cette remarque est aussi applicable au FILTRE-E et à la REQUETE-E qui doit aussi prendre en compte des conditions du type "vi op valeur" et "vi op vi".

- La division entre Isolé(v) et $Groupe_j(v)$ est moins nette. Supposons par exemple une précondition dont sa clause "WHERE" est de la forme : $(c_1(Groupe_1(v))$ ou $c_2[v_3])$ et $c_3(Groupe_1(v))$ où $Groupe_1(v) = \{v_1, v_2\}$ et Isolé(v) = $\{v_3\}$. Dans ce cas, il n'y a pas de sens à générer une requête pour $c(Groupe_1(v))$ (i.e. une requête pour c_1 et c_3) et une autre requête pour c_2 : l'évaluation serait faussée.
  Deux solutions sont possibles. La première consiste à générer trois requêtes : une pour chaque condition. La deuxième consiste à générer une seule requête qui évalue les trois conditions. L'application de la première solution implique des accès répétés aux mêmes informations dans la base de données vu que c_1 et c_2 portent sur les mêmes entités. Ainsi, dans la plupart des cas, il est préférable d'inclure $c(v_3)$ dans $c(Groupe_1(v))$ et de générer une seule requête.

- ORGANISER-FILTRE-REQUETE doit ordonner les différents tests filtres et requêtes générés de façon à reproduire l'expression booléenne de la clause "WHERE".

Evaluation Lorsqu'il y a Correspondance Sous Condition

La règle de génération du squelette d'évaluation crée des tests et des requêtes lorsqu'une entité participant au fait à insérer est qualifiée par la précondition. Ces tests et requêtes s'appliquent aussi bien dans les cas où il y a correspondance entre la précondition et la mise à jour que dans les cas où la correspondance est sous condition. En effet, les requêtes sont générées dans le contexte spécifié dans la clause "FROM" d'une précondition.

Par contre, si une entité participant au fait n'est pas qualifiée, alors la règle de génération considère que toutes ces instances qui vont être mises à jour vérifient la précondition : ceci est vrai dans le cas où il y a correspondance. Mais ce n'est pas correct dans le cas où il y a correspondance sous condition.

Supposons par exemple la précondition suivante :

```
TEST    INSERT  [Personne p, Véhicule v]
        FROM    Employé p, Véhicule v
```

Et la mise à jour :

```
        INSERT  [Personne p, Véhicule v]
        FROM    Personne p, Véhicule v
        WHERE   ...
```

Dans ce cas il y a correspondance sous condition. Pour évaluer la précondition, il faut néanmoins générer la requête suivante :

```
        SELECT  p.Surr
        FROM    Employé p
        WHERE   p.Surr IN {Surrj-Maj-p}
```

La règle doit alors être modifiée dans le but de prendre en compte ces cas.

Evaluation Lors d'une Mise à Jour d'un Autre Type

Comme il a été vu dans 6 une précondition d'insertion de faits peut être évaluée lors d'une mise à jour de modification de faits ou d'insertion d'une entité. En ce qui concerne l'opération de modification de faits, la règle de génération du squelette d'évaluation est applicable sans subir aucune modification. En ce qui concerne l'opération d'insertion d'une entité, l'application de la règle est plus délicate.

Supposons la mise à jour suivante :

m1 :	INSERT	Employé
	WITH	Nom = 'Gregorio'
	AND	Département = Département WHERE Numéro =12

Les préconditions concernant l'insertion d'instances du fait [Employé , Département], qui ne qualifient que la fonction Département, sont évaluables en appliquant directement la règle de génération du squelette d'évaluation.

Le préconditions d'insertion concernant le même fait et qui qualifient la fonction Employé (en faisant référence à d'autres propriétés que son nom), sont aussi évaluables par l'application directe de la règle de génération. Néanmoins, nous savons d'avance que l'évaluation échouera vu que l'employé à mettre à jour n'est qualifié que par son département. Il faut donc détecter ces cas et éviter l'évaluation de la précondition.

Considérons maintenant la précondition p1 et la mise à jour m1 augmentée de la clause suivante: "AND Catégorie = 'Ingénieur' ". Sémantiquement il faut décider si la précondition p1 doit être évaluée pour cette mise à jour. C'est à dire, doit-on prendre en compte les autres insertions de faits contenues dans l'insertion d'entités pour évaluer p1 ? Ceci semble souhaitable. Dans ce cas l'application directe de la règle de génération n'est pas possible. En effet, la règle considère que nous lions deux entités déjà existantes et qualifiées dans la base de données : or, avant l'exécution de la mise à jour l'employé Gregorio n'existe pas encore dans la base. Deux solutions ont été étudiées, elles sont décrites ci-dessous.

La première consiste à faire exécuter l'insertion de l'employé avec sa catégorie avant l'évaluation de la précondition. On pourrait alors appliquer la règle sans problème. Néanmoins, remarquons que si nous avions une préconditon p2 sur une insertion du fait [Employé , Catégorie] qui fasse référence dans sa partie qualification à Département, alors il y aurait un conflit. Cette solution ne resout donc pas tous les cas.

La deuxième solution consiste à générer des tests et des requêtes permettant l'évaluation de la précondition avant l'exécution de toutes les insertions contenues dans la mise à jour. Il faut alors étudier comment on peut tirer profit de la règle de génération. Le but des tests et des requêtes à générer est de sélectionner, à partir des valeurs affectées à une propriété (de l'entité de la clause

"INSERT" de la mise à jour) celles qui vérifient la précondition. Ainsi, supposons la mise à jour et la précondition suivantes :

maj : INSERT A
WITH G1 = ...
AND G2 = ...

pcd : TEST INSERT [G0 v_0 , G1 v_1]
FROM G2 v_2.G0 v_0, G1 v_1
WHERE $c(v_2,v_1)$

La requête à générer pour évaluer la précondition serait :

SELECT v_2.Surr, v_1.Surr
FROM G2 v_2 , G1 v_1
WHERE $c(v_2,v_1)$
AND v_2.Surr IN {$Surr_j$-Maj-v_2}
AND v_1.Surr IN {$Surr_j$-Maj-v_1}

On peur remarquer qu'en appliquant la règle de génération à la précondition suivante nous obtenons comme résultat la requête d'interrogation ci-dessus.

pcd : TEST INSERT [G2 v_2 , G1 v_1]
FROM G2 v_2, G1 v_1
WHERE $c(v_2,v_1)$

L'idée est là : A partir d'une précondition - p - et de la forme de la mise à jour, nous dérivons une précondition - p' - à laquelle on peut appliquer directement la règle de génération du squelette d'évaluation.

Le principe permettant de dériver p' a été illustré. Brièvement, p' considère l'insertion d'un fait constitué de toutes les fonctions qui :

(1) sont affectées par la mise à jour à la nouvelle entité à insérer
(2) sont référencées dans p.

Il est à remarquer que la précondition p' est artificielle et n'est dérivée que dans le but de pouvoir appliquer la règle de génération. En effet, pour l'exemple ci-dessus, le fait [G2, G1] peut même ne pas exister.

7.1.2. Précondition d'Insertion d'une Entité

A ce stade du rapport, l'évaluation d'une précondition d'insertion d'une entité n'est pas un

problème. Ces préconditions posent des conditions à chaque propriété (fonction) qui reçoit une valeur lors d'une mise à jour du même type. Chaque condition caractérise une seule fonction qui va être mise à jour, i.e., les fonctions mises à jour sont **qualifiées de façon isolée**. Nous pouvons alors, pour chaque condition, dériver une précondition p' d'insertion de faits à laquelle on peut appliquer la règle de génération du squelette d'évaluation. Un exemple suit :

```
p :      TEST    INSERT  Employé
                 WITH    Catégorie = 'Ingénieur'
                 AND     Salaire = Employé.Salaire WHERE
                                   Employé.Nom ='Jean'
                 AND     Département = Département WHERE
                                   Département.Type = 'Administratif'

p'1 :    TEST    INSERT  [Catégorie c, Employé e]
                 FROM    Employé.Catégorie c, Employé e
                 WHERE   c = 'Ingénieur'

p'2 :    TEST    INSERT  [Salaire s, Employé e]
                 FROM    Employé e1.Salaire s, Employé e
                 WHERE   e1.Nom = 'Jean'

p'3 :    TEST    INSERT  [Département d, Employé e]
                 FROM    Département d, Employé e
                 WHERE   d.Type = 'Administratif'
```

Etablisons alors la méthode de dérivation des p'_i. La forme d'une précondition d'insertion d'une entité est :

```
p :      TEST    INSERT  B
                 WITH    c1[H11]
                 AND     ...
                 AND     cq[H1q]
```

où $H1_j$, $1 \leq i \leq q$, est une fonction sur B
et $c_i[H1_i]$ est une condition qui qualifie de façon isolée $H1_i$ et qui est de la forme :

(a) $H1_i$ op1 valeur

ou (b) $H1_i$ op1 Déf-$H1_i$ WHERE c_i(Déf-$H1_i$)
avec Déf-$H1_i$ = Ki_1. ... . Ki_r
où Ki_j, $1 \leq j \leq r$, est une fonction
et r est le nombre de fonctions de la composition,

r est normalement très petit vu qu'il n'existe pas de clause "FROM" où on puisse déclarer des variables

et K_{ir} prend ses valeurs dans une classe qui est un Hiér de l'ensemble d'arrivée de $H1_i$

et c_i(Déf-$H1_i$) sont des conditions portant sur les K_{ij}

Pour chaque **i** une précondition de la forme suivante est générée.

p'_i : TEST INSERT [$H1_i$ v_1, $H2_i$ v_2]
FROM Déf'-$H1_i$ v_1, B v_2
WHERE c'_i[$H1_i$]

Dans le paragraphe 6.1.2 on peut trouver comment la clause INSERT de cette précondition à été dérivée. Par ailleurs nous avons :

- si c_i[$H1_i$] est de la forme (a) alors *(cf. p'_1)*

  Déf'-$H1_i$ = $H2_i$.$H1_i$

  et c'_i[$H1_i$] = v_1 op valeur

- si c_i[$H1_i$] est de la forme (b), alors *(cf. p'_2 et p'_3)*

  Déf'-$H1_i$ = K_{i1} v_{i1}. … . K_{ir-1} v_{ir-1}.K_{ir}

  nous introduisons des variables pour qu'il n'y ait pas d'ambiguïté

  c'_i($H1_i$) = c'_i(Déf-$H1_i$)

  c'_i(Déf-$H1_i$) est dérivée à partir de c_i(Déf-$H1_i$) de façon à faire référence aux variables introduites

7.1.3. Précondition de Suppression de Faits

Nous allons voir dans quelle mesure l'application de la règle de génération du squelette d'évaluation d'une précondition d'insertion de faits, aux préconditions de suppression de faits est possible et efficace. Pour cela les différents cas étudiés dans 7.1.1 sont repris dans ce paragraphe. Brièvement, une des différences entre l'évaluation d'une précondition d'insertion et celle de suppression est que, pour cette dernière, nous pouvons faire référence dans une requête aux faits à supprimer : ils sont dans la base de données car l'évaluation est faite avant l'exécution de la mise à jour. Pour les insertions ceci n'était pas possible.

Avant l'énumération des différents cas, regardons quelles sont les informations obtenues dans le but d'exécuter une suppression de faits par l'interpréteur ou le gestionnaire de transactions. En fait, le journal de reprise est constitué, pour une telle mise à jour, de toutes les instances de faits qui sont supprimées. Ainsi, supposons le fait :

[H1, …, Hr, Hr+1, …, Hl]

où Hi, $1 \leq i \leq r$, est une fonction ayant comme ensemble d'arrivée une entité virtuelle
et Hi, $r+1 \leq i \leq l$, est une fonction ayant comme ensemble d'arrivée une entité littérale
Pour cette association, le journal de reprise contient l'ensemble d'instances à supprimer, i.e.

$$[Surr_j\text{-Maj-H1}, \ldots, Surr_j\text{-Maj-Hr}, K_j\text{-Maj-Hr+1}, \ldots, K_j\text{-Maj-Hl}]\ 1 \leq j \leq x$$

où x est le nombre d'instances à supprimer.

CAS 1 - Précondition sur des entités virtuelles qualifiées de façon isolée

La règle[6] génère pour chaque entité qualifiée de façon isolée, une requête permettant de sélectionner, à partir des entités à mettre à jour, celles qui vérifient la précondition. Ceci peut se faire aussi dans le cas d'une suppression de faits vu qu'à partir du journal de reprise nous savons comment les entités sélectionnées par les différentes requêtes sont liées.

Mais il est aussi possible d'appliquer une autre solution qui consiste en la génération d'une seule requête portant sur les valeurs du fait à supprimer. Par exemple si p1 était une suppression de faits, nous aurions pû générer la requête suivante pour l'évaluer :

```
r1.3 :   SELECT  e.Surr, d.Surr
         FROM    Employé e.Département d
         WHERE   e.Catégorie = 'Ingénieur'
         AND     d.Type = 'Administratif'
         AND     e.Surr IN {Surrj-Maj-e}
         AND     d.Surr IN {Surrj-Maj-d}
```

Remarquons que la référence dans la clause "FROM" porte sur Employé.Département n'était pas possible pour les insertions.

On peut se poser la question pour savoir laquelle des deux solutions est la plus efficace : cette discussion doit prendre en compte l'implémentation. Dans notre cas, la discussion se ramène à comparer le temps d'évaluation de l'exécution d'une jointure réalisée par le SGBD relationnel sous-jacent (pour la solution qui consiste à générer la requête r1.3) et le temps de réalisation d'une sorte de jointure (comparaison entre les surrogates obtenus et les instances de fait à supprimer contenues dans le journal de reprise) faite dans le langage de programmation C (pour la solution qui consiste à générer r1.1 et r1.2). C'est la dernière solution qui a été retenue.

CAS 2 - Précondition portant sur des entités virtuelles qualifiées ensemble

Pour ce type de préconditions la règle génère une seule requête sur le produit cartésien des différentes entités participant au fait.

Ceci est tout à fait applicable dans le cas des suppressions. Mais dans ce cas, les entités à mettre à jour sont liées dans la base de données. En conséquence, la requête doit porter plutôt sur la jointure des entités à mettre à jour. Par exemple, si p2 était une précondition de suppression, la clause "FROM" de la requête r2 serait "Employé e. Département d" au lieu de "Employé e, Département d".

[6] "Règle" désigne dans ce paragraphe la règle de génération du squelette d'évaluation pour les préconditions d'insertion de faits.

CAS 3 - Précondition portant sur une entité littérale qualifiée de façon isolée

3(a) - L'entité littérale est comparée à une valeur déjà stockée dans la base de données

La règle génère une requête selon les mêmes principes que pour le cas 1. Les remarques faites pour le cas 1 sont donc applicables ici.

3(b) - L'entité littérale est comparée à une valeur constante

Dans ce cas la règle génère un test filtre. Bien sûr, ceci est tout à fait applicable au cas des suppressions et est également efficace.

CAS 4 - Précondition concernant une entité virtuelle et une littérale qualifiées ensemble

4(a) - La fonction sur l'entité littérale est monovaluée

Le principe utilisé pour les insertions de faits est d'instancier les désignations de l'entité littérale dans la partie qualification de l'énoncé de la précondition avec sa valeur, et procéder ensuite comme pour le cas 1. Pour les suppressions le même raisonnement peut être appliqué.

4(b) - La fonction sur l'entité littérale est multivalué

La règle pour les insertions génère une entité de travail et une association pour stocker les valeurs de l'entité littérale. Ceci est fait car ces valeurs n'existent pas dans la base de données. Pour le cas des suppressions cet artifice n'est pas nécessaire : les valeurs existent dans la base de données avant l'exécution de la mise à jour. Par exemple si p5 était une précondition de suppression de faits nous aurions pû l'évaluer avec la requête r5.2 (cf. 7.1.1).

Néanmoins, comme nous l'avons déjà vu (cas 4(b) 7.1.1), la solution qui mène à créer une entité de travail peut être intéressante du point de vue du temps d'évaluation. Pour appliquer cette solution aux suppressions, l'entité de travail doit être créée de façon à stocker les valeurs des paires (entité-virtuelle, entité-littérale) et pas seulement les valeurs des entités littérales comme c'était le cas pour les insertions.

En conclusion, on peut considérer que la règle s'applique aux cas des suppressions de faits, néanmoins elle peut être modifiée pour tenir compte des quelques optimisations suggérées ci-dessus.

7.1.4. Précondition de Suppression d'Entités

En général, le squelette d'évaluation doit être constitué de tests et de requêtes à la base de données qui permettent d'évaluer la précondition sur l'ensemble d'entités à supprimer. Dans le cas de la suppression d'entités, toutes les préconditions sont traduites par une seule requête à la base de

données, qualifiée par les entités à supprimer. A ce niveau nous ne réalisons plus aucune optimisation car toutes les entités sont virtuelles et appartiennent à la même classe (remarquer la similarité avec le cas 1 de 7.1.1). Par contre, dans 7.2 plusieurs optimisations de la clause "WHERE" sont proposées en tenant compte d'autres informations que celles des entités à supprimer.

Par l'interpréteur nous obtenons, lors d'une mise à jour, tous les identificateurs des entités à supprimer : $\{Surr_j\text{-}Maj\}_{j\geq 1}$.

Ainsi, pour une précondition de la forme :

p : TEST DELETE v_1
FROM Déf-v_1
WHERE $c(v_1)$

La requête suivante est alors générée pour évaluer p.

r : SELECT v_1.Surr
FROM Déf-v_1
WHERE $c(v_1)$
AND v_1.Surr IN $\{Surr_j\text{-}Maj\text{-}v_1\}$

Déf-v1 est défini dans 7.1.1.

7.1.5. Précondition de Changement de Classe

Le même principe que pour les suppressions d'entités est appliqué : une seule requête est générée, qualifiée par les entités qui font changer de classe.

Ainsi, supposons une précondition de la forme :

p : TEST INSERT B1 v_1
FROM B2 v_1
WHERE $c(v_1)$

où B2 est la classe à laquelle les entités désignées par v1 appartiennent
et B1 est la classe où ces entités vont être insérées : B1 est un Descendant-B2

La requête suivante est alors générée pour évaluer p.

r : SELECT v_1.Surr
FROM B2 v_1
WHERE c(v_1)
AND v_1.Surr IN {Surr$_j$-Maj-v_1}

Nous terminons ici le paragraphe 7.1 ; dans la conclusion du chapitre les principaux apports de la méthode de dérivation du squelette d'évaluation sont résumés (cf. 7.5).

7.2. Optimisations sur le Squelette d'Evaluation

Dans ce paragraphe sont proposées quelques optimisations à faire sur le squelette d'évaluation en tenant compte d'informations - auxquelles on peut accéder lors de l'exécution d'une mise à jour - autres que la liste d'entités à mettre à jour.

Comme il a déjà été dit dans l'introduction de ce chapitre, ces informations peuvent être de deux types : des constantes contenues dans la partie qualification de l'énoncé de la mise à jour, ou des informations contenues dans le journal de reprise.

7.2.1. Tirer Profit des Constantes Contenues dans la Mise à Jour

Pour illustrer ce type d'optimisation considérons une précondition - p6 - d'insertion sur le fait [Département, Employé] dont une des requêtes générées pour l'évaluer est r6.

r6 : SELECT e.Surr
FROM Employé e
WHERE e.Catégorie IN ('Ingénieur', 'Sécretaire')
AND e.Surr IN {Surr$_j$-Maj-e}

Supposons aussi la mise à jour suivante qui provoque l'évaluation de p6.

m6 : INSERT [Département d , Employé e]
FROM Département d , Employé e
WHERE d.Nom = 'Etudes'
AND e.Catégorie = 'Ingénieur'

Dans m6 nous avons, comme information, la catégorie des employés qui vont être affectés aux départements. Cette information peut éviter l'exécution de r6 : elle permet de générer un test filtre (comme pour le cas 3 (b) de 7.1.1) qui est toujours plus efficace.

C'est ce type d'optimisations que nous voulons prendre en compte. Pour détecter quand est-ce

qu'il est possible de l'appliquer, il faut réaliser une mise en correspondance entre les conditions des clauses "WHERE" des requêtes accomplissant l'évaluation de la précondition et des mises à jour.

Nous nous intéressons aux conditions de la forme (a) i.e., "y.G1. ... .Gt op valeur" (cf. 7.1.1). La variable "y" doit être liée dans la clause "FROM" de l'énoncé à une des entités à mettre à jour. Nous supposons ici que cette entité est désignée par la variable "v".

Remarquons que si nous avons une expression dans la qualification de la requête de la forme (a) et qu'elle est égale à une expression dans la qualification de la mise à jour, ceci ne suffit pas pour affirmer qu'il y a correspondance entre les deux conditions. Il faut, de plus, que les variables "y" de la mise à jour et de la requête soient liées de la même façon à la variable v.

Ainsi, les expressions entre lesquelles il faut établir la correspondance sont :

Liaison-v→y.y.G1. ... .Gt op valeur

où Liaison-v→y est la partie de Déf-v qui lie la variable v à la variable y.

L'établissement de la correspondance consiste en l'égalité des fonctions désignées dans chaque condition.

Il est à remarquer qu'il y a des cas où aucune optimisation n'est possible. Par exemple, si la condition de la précondition est "v.G1 > 1000" et celle de la mise à jour est "v.G2 > 5000".

La génération du squelette d'évaluation est faite lors de la création d'une précondition une fois pour toutes, et le squelette est enregistré dans un catalogue de préconditions (qui, comme on verra plus tard, est chargé en mémoire centrale). Par contre, les optimisations proposées ici ne sont possibles que lors de l'exécution des mises à jour. Pour cela, il est avantageux de garder trace dans le catalogue des conditions qui sont des candidates à ce type d'optimisation.

7.2.2. Tirer Profit des Informations Contenues dans le Journal de Reprise

Ce type d'optimisation ne s'applique qu'aux préconditions sur des suppressions lorsque leur évaluation est provoquée par une suppression d'**entités**. Il pourrait s'appliquer aussi au cas des préconditions de modifications de faits mais ce cas n'est pas traité dans ce rapport. L'optimisation à faire est différente selon qu'on accéde à la valeur, dans le journal de reprise, d'une entité littérale ou d'une entité virtuelle.

Cas des entités littérales

Supposons la précondition p7.

```
p7 :    TEST    DELETE  [Département d , Employé e]
                FROM    Département d , Employé e
                WHERE   e.Salaire > 18000
```

Supposons aussi une mise à jour qui supprime des employés de la base de données. Pour une

telle mise à jour, le journal de reprise est constitué de toutes les instances de faits auxquels les employés à supprimer participent. Ainsi, nous pouvons accéder aux valeurs de toutes les fonctions qui s'appliquent directement à Employé. Entre autre, nous connaissons la valeur du salaire des employés à supprimer. Dans ce cas la requête générée pour évaluer p7 peut être évitée : un test filtre est généré à sa place.

Ce type d'optimisation s'applique à toutes les conditions d'une précondition de suppression, de la forme "v.G1 op valeur", où v désigne les entités à supprimer et G1 une fonction.

Cas des entités virtuelles

Supposons la précondition p8.

```
p8 :    TEST    DELETE  [Département d , Employé e]
                FROM    Département d , Employé e.Projet p
                WHERE   c(p)
```

où c(p) désigne une condition quelconque sur la variable p.
Pour accomplir son évaluation la requête m8 est générée.

```
r8.1 :          SELECT  e.Surr
                FROM    Employé e.Projet p
                WHERE   c(p)
                AND     e IN {Surr_j-Maj-e}
```

Lors d'une suppression d'employés, p8 doit être évaluée. A ce moment nous connaissons les identificateurs des projets qui sont liés aux employés à supprimer. Pour tirer profit de cette information nous avons deux possibilités. La première consiste à réduire l'ensemble de la base de données sur lequel porte la requête r8.1. Pour cela r8.1 est augmenté de la qualification des projets : "p.Surr IN {Surrj-Maj-p}". La deuxième consiste à faire exécuter r8.2 plutôt que r8.1.

```
r8.1 :          SELECT  p.Surr
                FROM    Projet p
                WHERE   c(p)
                AND     p IN {Surr_j-Maj-p}
```

Comme dans le journal de reprise sont enregistrées les paires (surrogate-employé , surrogate-projet) pour tous les employés à supprimer, nous pouvons évaluer sans problème la précondition.
Vu notre implémentation sur un SGBD relationnel, la deuxième solution est plus efficace : la consultation du journal de reprise pour former les instances du fait [Employé, Projet] qui vérifient la précondition est plus optimal qu'une jointure de deux relations.

Ce type d'optimisation s'applique aux conditions qui portent sur une propriété directe des entités à supprimer.

Remarquons que les optimisations présentées dans ce paragraphe peuvent être faites au moment de la création d'une précondition. En fait, pour une précondition de suppression de faits le squelette d'évaluation doit proposer, pour l'évaluation des conditions caractérisées ci-dessus, deux stratégies : l'une qui s'applique lors de l'évaluation provoquée par une suppression de faits et l'autre lors d'une suppression d'entités.

7.3. Exploitation de Conditions Communes à Plusieurs Préconditions

Ce paragraphe présente juste une idée de la façon dont ce type d'optimisation peut être accompli. Pour rester brefs, nous nous plaçons dans le cas des manipulations sur les faits.

Pour pouvoir détecter quand est-ce que deux préconditions sont évaluables pour la même mise à jour, nous reprenons la forme des énoncés de précondition et mises à jour présentées dans le chapitre 7. Ainsi, supposons les deux préconditions suivantes.

```
p9 :      TEST     INSERT  [H1 v1, H2 v2]
                   FROM    B1 v1, B2 v2

p10 :     TEST     INSERT  [H1 v1, H2 v2]
                   FROM    B3 v1, B4 v2
```

Si B1 ≡ B3 et B2 ≡ B4, alors il est sûr que p9 et p10 sont toujours évaluables pour les mêmes mises à jour. Il importe alors de déterminer s'il y a des conditions communes qui peuvent être évaluées ensemble : ceci sera un peu plus détaillé à la fin du paragraphe.

Nous introduisons la mise à jour m11 pour illustrer quelques cas où cette optimisation peut intervenir si nous n'avons pas B1 ≡ B3 et B2 ≡ B4.

```
m11 :              INSERT  [H1 v1, H2 v2]
                   FROM    B5 v1, B6 v2
```

Supposons B1 ≡ B3 ≡ B5 :

- Si B6 est un Ancêtre-B4 et B4 est un Ancêtre-B2, alors p9 et p10 sont évaluables lors de l'exécution de m11.
  Si dans la qualification de p9 et de p10 il y a une condition commune portant sur v2 (à part que v2 désigne pour p9, B2 et pour p10, B4), il est intéressant de l'évaluer une seule fois. Dans ce cas, l'évaluation de la condition devrait porter sur B4 : on obtiendrait ainsi les entités (ou faits) qui vérifient la condition de p10. Pour obtenir celles qui vérifient la condition de p9, il faut sélectionner, à partir des entités qui vérifient la condition de p10, celles qui appartiennent à la classe B2.

- B6 est un Descendant-B4 et B4 est un Descendant-B2, alors il y a correspondance sous condition entre m11 et p9 et entre m11 et p10. Nous supposons comme ci-dessus, qu'il y a une condition sur v2 commune aux deux préconditions. Dans ce cas il faut évaluer la condition sur B2 : on obtiendrait ainsi les entités qui vérifient la condition de p9. De même pour obtenir celles qui vérifient la condition de p10, il faut sélectionner, à partir des entités qui vérifient la condition de p9, celles qui appartiennent à la classe B4.

Ces exemples donnent une idée des cas où ce type d'optimisation peut intervenir. Nous ne faisons pas dans ce rapport une présentation exhaustive de tous les cas. Mais il est à retenir qu'il est important, lors de la création des préconditions portant sur le même fait, de déterminer et de garder trace de leurs conditions communes et d'une stratégie pour pouvoir les évaluer ensemble quand elles sont activées lors de la même mise à jour.

La détermination de conditions communes peut se faire simplement en analysant le squelette d'évaluation des différentes préconditions, et en détectant les tests ou les requêtes équivalentes. Mais on peut aller plus loin, et comparer des sous-conditions dans la qualification des différentes requêtes qui sont équivalentes et intéressantes à évaluer une seule fois. Cette dernière proposition est plus complexe car elle implique la division d'une requête en sous-requêtes et la matérialisation en mémoire centrale des différents résultats.

7.4. Création d'Activités

Nous abordons ce paragraphe par un exemple donnant une idée générale de la façon dont les instances d'activités sont créées avec leur entité focale et contexte d'activation, lors de la validation d'une précondition.

Nous supposons la précondition p1 (cf. 7.1.1) sur le fait [Département, Employé] et l'exécution des requêtes : r1.1 qui sélectionne les employés qui vérifient p1, et r1.2 qui sélectionne les départements qui vérifient p1. Supposons aussi que le résultat de r1.1 est les employés e1 et e2 et le résultat de r1.2 est les départements d1, d2 et d3. La précondition p1 est liée à une activité A ayant comme entité focale Employé et comme contexte d'activation le fait [Employé, Département].

La validation de la précondition implique alors la création des instances suivantes :

A1 Entité focale : e1
Contexte d'activation : e1 (d1, d2, d3)

A2 Entité focale : e2
Contexte d'activation : e2 (d1, d2, d3)

L'expression "e1 (d1, d2, d3)" fait référence à l'instance du fait créé entre l'employé e1 et les départements d1, d2 et d3. Ceci est la notation utilisée pour exprimer le résultat d'une requête FML : une expression entre parenthèses qui reflète une liste d'instances de chemins[7] du schéma

conceptuel. Si la requête sélectionne les identificateurs d'instances des classes E1, ..., En et que le chemin E1. ... .En est linéaire, alors le résultat est de la forme :

$$(e_{11}\,(e_{12}\,(\,\ldots\,(e_{n-11}\,(e_{n1},\,\ldots,\,e_{nf})\,\ldots\,)$$

où e_{ij} est une instance de Ei, et f est le nombre d'instances e_n qui sont associées à e_{n-11}. Nous disons que cette liste est factorisée par la classe E1. Bien sûr, toute liste peut être modifiée de façon à être factorisée par une classe Ei, $1 \leq i \leq n$, quelconque.

Le résultat du squelette d'évaluation est aussi de ce type :

- Si le squelette est constitué d'une seule requête, nous obtenons tout de suite le résultat sous cette forme.

- Si le squelette est constitué de plusieurs requêtes et/ou tests, comme dans l'exemple ci-dessus, alors il faut traiter les différents résultats pour le présenter sous cette forme[8].

Ainsi, le résultat d'un squelette d'évaluation est une liste d'instances de chemins dans le schéma conceptuel qui vérifient la précondition, représentée par une expression entre parenthèses.

Pour la création des activités lors de la validation d'une précondition, il est intéressant que l'expression entre parenthèses représentant le résultat du squelette d'évaluation soit factorisée par la classe de l'entité focale. Dans ce cas, pour chaque instance de l'entité focale de la liste résultat, une instance de l'activité correspondante est créée. Elle a comme contexte d'activation la liste que l'instance de l'entité focale factorise.

Ainsi, pour accomplir la création des activités, nous traitons le résultat du squelette d'évaluation de façon à ce qu'il soit factorisé par la classe de l'entité focale.

Ce raisonnement est appliqué à tous les types d'énoncés de précondition. Nous ne détaillons pas plus ce paragraphe : terminons seulement par un exemple du résultat de l'exécution du squelette d'évaluation d'une précondition d'insertion d'une entité et de la création des activités conséquentes. L'exemple illustre le cas où le squelette d'évaluation est composé de plusieurs requêtes à la base de données et où le chemin des instances qui vérifient la précondition est arborescent. Il permet aussi de donner une idée générale sur ce chapitre.

---

[7] Les chemins sont tracés à travers les différents faits : du point de vue fonctionnel, ils représentent des compositions de fonctions.

[8] Nous ne détaillons pas ici comment le résultat est traité : ceci ne représente pas de difficulté et a été fait par ailleurs.

Supposons alors la précondition :

TEST	INSERT	Employé
	WITH	Département <op ...>
	AND	Catégorie <op ...>
	AND	Salaire <op ...>

L'évaluation de cette précondition se fait par trois requêtes ou tests :

- Un portant sur Département. Supposons que son résultat est d1, d2 et d3.
- Un portant sur Catégorie. Supposons que son résultat est c1, c2.
- Un portant sur Salaire. Supposons que son résultat est s1.

L'identificateur de l'employé à insérer est e1.

L'entité focale de cette précondition peut être Employé ou Département. :

- Si elle est Employé, la liste suivante est créée :
  (e1((d1, d2, d3), (c1, c2), (s1)))
  En conséquence, une seule instance d'activité est créée ayant comme entité focale e1 et comme contexte d'activation la liste résultat.

- Si elle est Département, la liste suivante est créée :
  (d1 (e1 ((c1, c2), (s1))), d2 (e1 ((c1, c2), (s1))), d3 (e1 ((c1, c2), (s1))))
  En conséquence, trois instances d'activités sont créées :

  A1 Entité focale : d1
  Contexte d'activation : d1 (e1 ((c1, c2), (s1)))

  A2 Entité focale : d2
  Contexte d'activation : d2 (e1 ((c1, c2), (s1)))

  A3 Entité focale : d3
  Contexte d'activation : d3 (e1 ((c1, c2), (s1)))

  Remarquons que dans le cas général le contexte d'activation n'est pas toujours le même pour toutes les intances d'activités créées. Par exemple, considérons la précondition d'insertion des faits "TEST INSERT [Personne p, Véhicule v] FROM Département d.Employé p, Véhicule v", les départements *d* appartiennent au contexte d'activation. Dans ce cas, nous pouvons avoir des instances différentes du contexte chaque fois que la précondition est validée : en effet, les employés associés aux véhicules peuvent ne pas appartenir au même département.

7.5. Evaluation et Intégrité Sémantique

Toutes les mise à jour exécutées à l'intérieur d'une transaction restent invisibles à l'environnement externe jusqu'à la confirmation de la transaction (principe d'isolation) et ses effets disparaîssent si la transaction est annulée (principe d'atomicité). C'est donc, au moment de la confirmation d'une transaction que peuvent être effectivement enregistrées les activités qui ont été hablilitées par des opérations exécutées dans la transaction. Ceci aurait pu suggérer une stratégie d'évaluation de préconditions où les préconditions sélectionnées soient collectionnées pendant la transaction pour être évaluées uniquement au moment de la confirmation de la transaction. Cette stratégie a l'avantage d'éviter la surcharge due à l'évaluation des préconditions pendant la transaction et concentre cette surcharge à la fin de la transaction. De plus, si la transaction est annulée, le travail d'évaluation des préconditions est évité, i.e., aucune précondition n'a été évaluée. D'un autre côté cette stratégie a le désavantage qu'il n'est pas possible d'intégrer l'évaluation des préconditions avec le traitement des mises à jour : ceci entraîne une surcharge beaucoup plus grande au moment de la confirmation de la transaction. Du reste, des conditions qui étaient vraies au moment de l'exécution de la mise à jour peuvent être fausses à la fin de la transaction, la conséquence étant que les activités qui devraient être habilitées ne le seront pas.

Comme il a été dit dans les paragraphes précédents, l'approche retenue ici consiste à évaluer des préconditions lors du traitement des mises à jour pendant la transaction. Ainsi, l'évaluateur de préconditions accède à la base de données en lecture seulement pour obtenir l'information nécessaire à la vérification des conditions. Au fur et à mesure que les préconditions deviennent vraies, les activités habilitées en conséquence et leur contexte d'activation sont stockés dans la mémoire centrale jusqu'à la fin de la transaction. Si la transaction est annulée, vu que l'évaluateur de préconditions n'a jamais mis à jour la base de données, seul le travail habituel est à réaliser, i.e., celui concernant les mises à jour exécutées par l'application. Si la transaction est confirmée, une sous-transaction est exécutée pour stocker dans la base de données l'information concernant les nouvelles activités habilitées. Cette approche optimiste assure que toutes les activités pertinentes sont habilitées et produit une évaluation des préconditions plus optimale.

7.6. Conclusion

Le gain important de la méthode générale d'évaluation des préconditions vient du fait que cette évaluation est étroitement liée à l'exécution des mises à jour.

Tout d'abord, cette prémisse a permis de limiter l'évaluation de la précondition aux informations mises à jour. L'approche adoptée pour tenir compte de cette optimisation peut être vue comme si on instanciait la précondition avec les valeurs des entités/faits à mettre à jour. Ainsi, nous obtenons une expression de la précondition où :

- L'évaluation des conditions sur une entité virtuelle de la clause "TEST" de la précondition est limitée aux entités mises à jour, ce qui évite :
  - Soit de refaire l'évaluation de la partie qualification de la mise à jour. Dans ce cas, le gain

de temps de calcul et d'accès à la base de données est proportionnel à la complexité des conditions contenues dans la mise à jour et à la taille de la base de données.

- Soit de comparer les états avant et après la mise à jour pour détecter les faits mis à jour. Dans ce cas, le temps gagné est proportionnel au nombre de classes d'entités concernées par la précondition (i.e., les classes de la clause "TEST") et également à la taille de la base de données.

- Les conditions du type "<entité littérale à mettre à jour> op valeur" ne nécessitent aucun accès à la base de données pour être évaluées. Nous évitons ainsi un ou plusieurs accès à la base de données, dépendant du fait que la fonction sur l'entité littérale est monovaluée ou non, et de la façon dont les fonctions multivaluées sont implémentées.

- Les conditions du type "<expression sur entité virtuelle> op <entité littérale à mettre à jour>" sont modifiées :
  a. Soit dans une condition du type "<expression sur entité virtuelle> op constantes". Dans ce cas, l'expression ne porte plus que sur l'entité virtuelle : les accès évités dépendent de la façon dont le fait composé au moins de ces deux entités est implémenté.
  b. Soit dans une condition qui porte sur une entité de travail instanciée par les valeurs de l'entité littérale à mettre à jour. Les instances de l'entité de travail sont normalement peu nombreuses. Le gain par rapport au temps d'évaluation dépend alors de la comparaison entre le nombre d'instances de l'entité littérale dans la base de données et le nombre d'instances de l'entité de travail.

Nous venons de donner un aperçu sur les optimisations générales appliquées au moment de la dérivation du squelette d'évaluation. De plus, pour certaines combinaisons de préconditions et mises à jour, d'autres types d'optimisation peuvent être appliqués.

Pour les préconditions de suppression d'entités ou de faits et les mises à jour de suppression d'entités, nous pouvons instancier la précondition avec des valeurs contenues dans le journal de reprise (cf. 8.2.2). De même, les constantes contenues dans une mise à jour peuvent permettre d'instancier certaines variables dans la partie qualification de la précondition (cf. 7.2.1).

Comparaison avec l'Evaluation de Contraintes d'Intégrité

Dans 2.1.6.2 nous avons étudié l'applicabilité des principes d'évaluation des contraintes d'intégrité sur des SGBD Relationnels à l'évaluation de préconditions. Beaucoup de principes n'étaient pas applicables car les préconditions ne sont pas liées à la notion d'état valide de la base de données. Ainsi, nous ne pouvons pas partir du principe (applicable aux contraintes) que la précondition est fausse avant son évaluation. Néanmoins plusieurs de ces principes sont intervenus dans l'établissement de la méthode générale :

- Le **principe 7** : Isoler les n-uplets à mettre à jour dans des relations temporaires pour diminuer le champ d'évaluation de la précondition.

Pour certains cas nous appliquons directement ce principe (mis à part que nous utilisons un modèle de plus haut niveau que le modèle relationnel).
Mais dans le cas général il est appliqué un peu différemment : les requêtes permettant d'évaluer la précondition sont qualifiées avec les identificateurs des entités à mettre à jour.

- Le **principe 3** : Modifier la contrainte d'intégrité en fonction de la requête. D'un certain point de vue, nous pouvons dire que ce principe 3 est utilisé : l'instanciation d'une précondition avec les entités/faits à mettre à jour peut être considérée comme l'application de ce principe (cf. exemple 2.5).

- Le **principe 5** : Utiliser la connaissance de la mise à jour pour évaluer les contraintes. Ce principe est appliqué quand on tire profit des constantes contenues dans la mise à jour.

- Le **principe 6** : Générer des requêtes les plus sélectives possibles pour tester la contrainte d'intégrité. Ce principe est appliqué quand on évalue de façon séparée les conditions qui qualifient de façon isolée une entité particulière.

- Le **principe 8** : Evaluer par ordre croissant de complexité. Ce principe est appliqué quand on organise les différentes requêtes et tests dans un squelette d'évaluation.

Par ailleurs, un autre principe a été appliqué : exploitation de conditions communes à plusieurs préconditions.

Il est à remarquer que le principe 2 consistant à garder dans la base de données des valeurs redondantes (matérialisation de fonctions d'agrégation ou de certains ensembles d'informations) aurait pu être utilisé avec précaution pour certaines classes de préconditions. Pour faire cela il faut caractériser le groupe de préconditions qui pourrait profiter de ce type d'optimisation. Entre autre, l'évaluation des préconditions concernant les fonctions d'agrégation[9] pourrait prendre en compte ce principe.

Préconditions d'Etat

Nous voulons présenter dans cette conclusion quelques commentaires sur l'évaluation des préconditions d'état. Comme nous avons vu dans le parargraphe 6.3 l'évaluation de ces préconditions est faite dans deux circonstances différentes. La première (deuxième) doit détecter qu'une précondition qui était fausse (vraie) devient vraie (fausse). Ceci implique deux stratégies d'évaluation complètement différentes :

1. **Passage de faux à vraie**. Pour appliquer les règles de correspondance dans ce cas, nous avons transformé les clauses "TEST" et "FROM" de la précondition dans une disjonction de clauses "TEST" et "FROM" de préconditions de changement. Pour cela nous ne nous sommes

[9]Ce type de précondition n'a pas été traité dans ce chapitre. Brièvement leur évaluation nécessite la génération d'une requête à la base de données ayant uniquement comme but le calcul de la valeur de la fonction d'agrégation.

pas occupés de la partie qualification car elle n'était pas importante à ce moment-là. Mais notre but est de dériver à partir de la précondition d'état une disjonction de préconditions de changement. Dans ce cas, nous pouvons appliquer la règle de génération du squelette d'évaluation à chaque précondition de changement.

2. **Passage de vraie à faux**. L'évaluation d'une précondition dans ce cas se fait quand un utilisateur ou une application veulent accéder à une instance de l'activité liée à cette précondition: la précondition n'est pas activée par une mise à jour. Ainsi, nous générons une requête à partir de la précondition en l'instanciant avec l'identificateur de l'entité focale correspondant à l'activité.

Dans le chapitre suivant nous allons voir les aspects techniques du traitement des préconditions. En particulier, la mise en œuvre de l'activation et de l'évaluation d'une précondition ainsi que de la création des activités sont étudiés.

CHAPITRE 8

8. IMPLEMENTATION

Un des points forts du projet DOEOIS est la réalisation d'un prototype du Serveur d'Information pour la Bureautique. Le prototype SIB offre les services de base pour le développement d'applications de gestion des tâches bureautiques. Un point essentiel pour celà est la mise en œuvre du mécanisme d'évaluation des préconditions.

Ce chapitre présente dans un premier temps l'architecture générale du prorotype et les éléments nécessaires à la mise en œuvre des aspects dynamiques des applications. La suite du chapitre est consacrée à l'implémentation des mécanismes liés à la notion d'activité et de procédure.

Toutefois, le prototype contient certaines limitations par rapport aux différentes facilités présentées dans ce document. Nous énumérons en conclusion les différentes restrictions faites.

8.1. Architecture Générale

L'objectif de ce paragraphe est de présenter l'architecture générale du prototype de SIB développé dans le projet DOEOIS. Cette présentation est faite en deux parties. La première situe les applications par rapport au serveur SIB et la seconde donne des détails sur le SIB. Pour finir, il sera mis en évidence les différents éléments nécessaires au traitement des aspects dynamiques des applications.

Lien Applications - SIB

La figure 8.1 schématise les liens entre les applications et le serveur SIB. L'architecture générale du SIB est du type Client-Serveur. La partie Client est constituée de différentes applications qui vont intéragir avec le Serveur ; ces applications utilisent une interface fonctionnelle (HLI et LLI). La partie Serveur, i.e., SIB, réalise les services offerts par l'interface.

Ce type d'architecture a été adopté dans le but de permettre la distribution des applications sur des sites autres que le site du noyau SIB.

Plusieurs applications, s'appuyant sur l'interface externe, sont fournies. Ce sont des outils d'aide au développement des applications. Nous trouvons :

- Une interface graphique pour la définition d'un schéma conceptuel statique.
- Une interface graphique pour l'interrogation de la base de données.
- Une interface graphique pour la définition d'activités et procédures de bureau.

- Une interface graphique pour le suivi d'activités et procédures de bureau.
- Un éditeur de documents ODA.

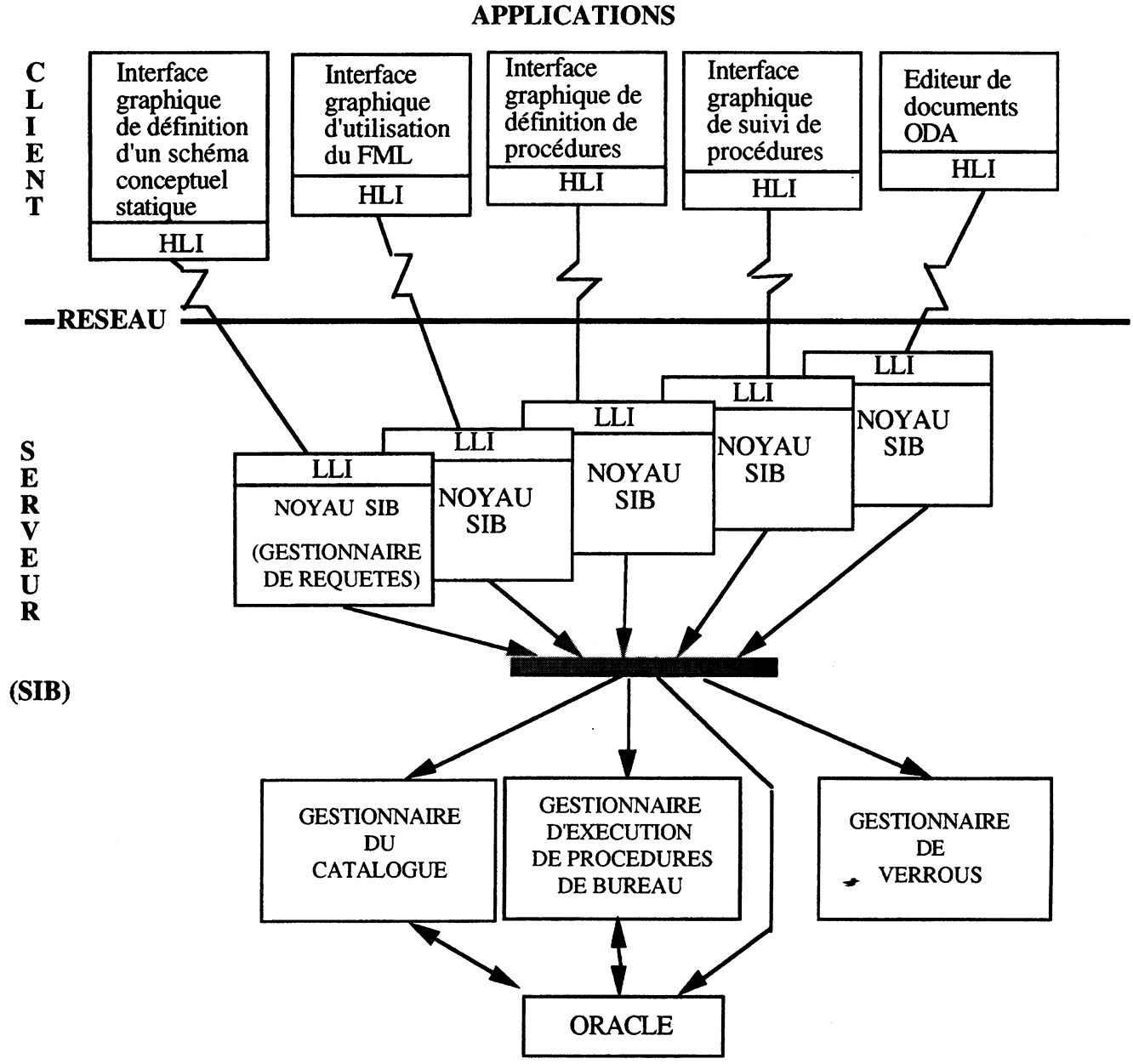


Figure n° 8.1 - Lien entre les Applications et SIB

Le SIB est composé d'un noyau appelé le Gestionnaire de Requêtes (un processus par application) et de quatre processus résidants. Les quatre processus et leur rôle respectif sont :

- **SGBD Relationnel** (ORACLE dans notre cas) : assure la persistance des données et l'accès basique à celles-ci.
- **Gestionnaire du Catalogue** : assure la correspondance entre le Fact Model et le modèle relationnel supporté par le SGBD choisi.
- **Gestionnaire de Verrous** : assure le contrôle de l'accès concurrent.
- **Gestionnaire d'Exécution de Procédures de Bureau** : assure l'exécution, le contrôle

et le suivi des procédures de bureau.

La communication entre les applications et le noyau SIB est réalisée par un mécanisme d'appel de procédures à distance (RPC), via l'outil "Courrier".

Les fonctions utilisées dans l'appel par procédure sont regroupées en deux interfaces : HLI ("High Level Interface") et LLI ("Low Level Interface"). HLI est le représentant du Serveur dans la partie Client et offre un niveau de fonctionnalité supérieur à celui de LLI : une requête HLI peut être transformée en un ensemble de requêtes LLI. Les fonctions de ces interfaces permettent entre autre l'ouverture ou la fermeture d'une session SIB, la spécification de la base de données sur laquelle on désire travailler, la demande d'exécution d'un énoncé (FML ou OPCL, etc.) et la récupération de résultats. Pour plus d'informations sur cette interface, le lecteur pourra se référer à [DOE87c].

L'activation du SIB implique le lancement des quatre processus résidants. Commencer une session SIB entraîne le lancement d'un Gestionnaire de Requêtes sur le site du serveur SIB. La communication entre le processus Gestionnaire de Requêtes et les quatre processus résidants se fait par le mécanisme de "sockets" du système UNIX (pour des raisons d'efficacité il a été préférable d'utiliser directement les sockets plutôt que des outils plus sophistiqués comme PRC). La durée de vie du Gestionnaire de Requêtes est celle de l'application qui l'a lancé.

Noyau SIB

Détaillons maintenant l'architecture du noyau SIB (i.e., du Gestionnaire de Requêtes). Elle est présentée dans la figure 8.2. Le gestionnaire de Requêtes doit traiter l'ensemble des énoncés du FML (qui inclue les langages de définition, d'interrogation et de mise à jour) et du OPCL (langage de contrôle des procédures de bureau cf. 4.6 et 5.2). Il peut être considéré comme divisé en deux parties. L'une responsable de la compilation des différents énoncés et l'autre de leur interprétation. La partie **compilation** correspond aux modules Analyseur Lexico-Syntaxique, Analyseur Sémantique et Traducteur. La partie **interprétation** est accomplie par l'interpréteur en collaboration avec les modules du Gestionnaire de Transactions, de l'Evaluateur de Préconditions et du Gestionnaire de Documents.

Les différents modules de ce Gestionnaire et leurs principales fonctions sont énumérés ci-dessous.

- **Analyseur Lexico-Syntaxique** : contrôle la syntaxe de la requête reçue et la transforme en une représentation intermédiaire (RI) arborescente. Il est construit à l'aide des outils UNIX, Lex et Yacc et EMIR [Len89], un outil de gestion de structures arborescentes attribuées.

- **Analyseur Sémantique** : recherche les éventuelles erreurs sémantiques de la requête. Décore l'arbre généré pendant l'analyse syntaxique avec des informations liées aux structures de stockage (informations fournies par le gestionnaire du catalogue).

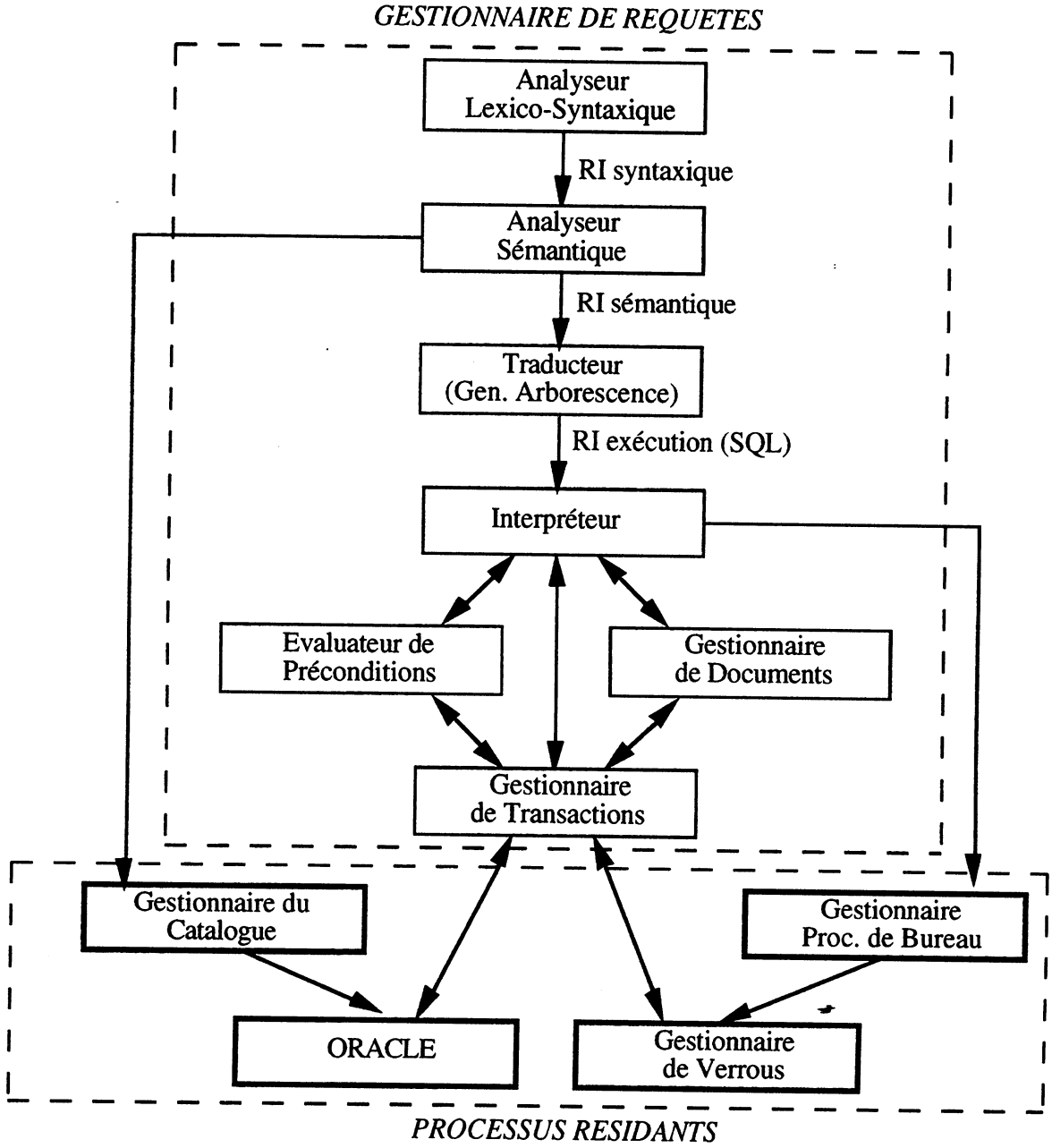


Figure n° 8.2 - Architecture du Noyau SIB

- **Traducteur** : transforme l'arbre sémantique en un arbre d'opérateurs. Fait appel à des sous-modules particuliers qui rajoutent des opérateurs propres à l'Evaluateur de Préconditions, au Gestionnaire de Requêtes, etc.

- **Interpréteur** : applique la fonction liée à chaque opérateur de l'arbre en la soumettant au module concerné (Gestionnaire de Transactions, SGBD, Gestionnaire de Documents, etc.). Assure la séquence et contrôle l'exécution des opérations. Retourne le résultat ou le code d'erreur de l'énoncé.

- **Evaluateur de Préconditions** : interprète la partie de l'arbre concernant l'évaluation des préconditions. Signale éventuellement au Gestionnaire de Procédures de Bureau les activités/procédures à créer lors d'un énoncé de mise à jour.

- **Gestionnaire de Documents** : assure l'insertion et l'extraction de documents dans la base de données. Fournit des fonctions de recherche sur le contenu des documents par filtrage.

- **Gestionnaire de Transactions** : assure le contrôle de l'accès concurrent aux données à l'intérieur et à l'extérieur d'une transaction, tout au long de la durée d'une session, par l'intermédiaire du gestionnaire de verrous. Assure la reprise après panne pour les transactions. Contribue à l'exécution des activités et des procédures de bureau.

Mise en Œuvre des Aspects Dynamiques

Nous allons maintenant considérer la mise en œuvre des aspects dynamiques. Les modules et processus nécessaires à leur implémentation seront énumérés et situés dans l'architecture générale du prototype.

Pour traiter les aspects dynamiques d'une application nous avons besoin de mettre en œuvre trois fonctions :

1. La définition des classes d'activités et de procédures
2. L'évaluation des préconditions lors de l'exécution de mises à jour qui peuvent provoquer la création d'instances d'activités.
3. L'exécution d'énoncés du OPCL qui impliquent le lancement, la terminaison ou la suspension des activités/procédures de bureau.

Quelques détails sur la mise en œuvre de ces fonctions suivent.

Définition d'Activités et de Procédures

Comme nous avons vu dans la partie 2 de ce rapport tout schéma conceptuel contient un ensemble de définitions standards qui définissent ce que sont une activité et une procédure. Les classes "Activity-Class", "Activity", "Procedure-Class", "Procedure" et leur associations sont ainsi déclarées. En conséquence, la définition d'une classe d'activités/procédures particulière se fait à partir d'énoncés de mise à jour FML : une activité est créée de la même façon qu'une instance d'une classe d'entités est insérée dans la base de données.

Néanmoins, certaines propriétés (associations) de ces classes nécessitent un traitement particulier, comme par exemple la précondition ou le schéma d'une procédure. En effet, la définition de la classe de préconditions ou du schéma se fait par l'utilisation de types abstraits (cf. 4.2 et 5.1) qui impliquent l'implémentation d'un ensemble de fonctions permettant leur traitement. Par exemple, comme nous allons le voir, la compilation d'une précondition consiste, entre autre, en la dérivation de son squelette d'évaluation.

En conséquence, un ensemble de fonctions est implémenté au niveau de l'Analyseur Lexico-

Syntaxique, de l'Analyseur Sémantique et du Traducteur, pour permettre l'accomplissement de la définition des activités et des procédures.

Evaluation des Préconditions Lors des Mises à Jour

Comme il a été vu dans les chapitres 6 et 7 de ce rapport l'évaluation des préconditions se fait lors des mises à jour. L'évaluation d'une précondition est réalisée tout d'abord en deux étapes. La première permet de **sélectionner les préconditions** qui doivent être évaluées pour la mise à jour en vigueur : pour cela les règles de correspondance établies au chapitre 6 sont appliquées. La deuxième accomplit l'**évaluation** proprement dite en exécutant le squelette d'évaluation (cf. 7).

Ces deux étapes sont réalisées à des moments différents. Très brièvement la sélection des préconditions se fait pendant la **compilation** d'une mise à jour. En effet, l'arbre sémantique de la mise à jour contient les informations nécessaires à l'établissement de la correspondance. Ainsi, un sous-module est implémenté au niveau du traducteur : il sélectionne les préconditions à évaluer et rajoute leur squelette d'évaluation à l'arbre de la mise à jour. La phase d'évaluation se fait lors de l'**interprétation** d'une mise à jour. Cette évaluation consiste en l'interprétation du sous-arbre correspondant aux préconditions, et est accomplie par l'Evaluateur de Préconditions.

Par ailleurs, nous pouvons considérer une troisième phase dans l'évaluation des préconditions, consistant à **créer des activités**. Ainsi, lors de **la fin de toute transaction** l'Evaluateur de Préconditions doit avertir le Gestionnaire d'Exécution des activités/procédures à créer.

Traitement des Enoncés OPCL

Le langage OPCL constitue un sur-ensemble du FML. La compilation de ces énoncés peut être réalisée, pour une bonne partie par les modules de compilation déjà présentés. Néanmoins, le développement de certaines fonctions est nécessaire pour prendre en compte les particularités syntaxiques et sémantiques des primitives OPCL.

Brièvement un énoncé OPCL est traduit en deux opérations : (1) la sélection des activités/procédures à manipuler et (2) l'exécution de l'objectif de l'énoncé (i.e. lancer, terminer ou suspendre une activité/procédure).

En ce qui concerne l'interprétation de ces primitives, (1) est réalisée par l'interpréteur. Par contre, l'accomplissement de (2) nécessite le développement d'un gestionnaire d'exécution. Pour cela, comme il a déjà été dit nous avons implémenté un processus résidant ayant comme rôle l'exécution, le contrôle et le suivi des activités et des procédures.

Ce paragraphe se termine ici : le traitement des aspects dynamiques sera présenté dans les prochains paragraphes ; nous traitons en particulier les points suivants :

- Compilation d'une Précondition (8.2)
- Compilation d'un Schéma de Procédure (8.3)
- Evaluation des Préconditions (8.4)
- Exécution et Contrôle des Activités et Procédures (8.5)

8.2. Compilation d'une Précondition

La compilation d'une précondition est réalisée lors de la définition d'une classe d'activités. Son but est de générer et de stocker l'information nécessaire à l'évaluation de la précondition avec un minimum de travail. Cette information est gardée dans un catalogue - Catalogue des Préconditions - résidant en mémoire centrale pour lequel une copie sur disque est maintenue : en fait, ce catalogue appartient au catalogue général d'une base de travail. Il est essentiel que le catalogue soit en mémoire centrale car toute mise à jour provoque sa consultation pour la sélection et l'évaluation des préconditions.

Ce catalogue contient quatre types d'informations différents :

1. L'information nécessaire pour faire la sélection des préconditions lors des mises à jour.
2. L'information nécessaire pour évaluer les préconditions lors des mises à jour.
3. L'information nécessaire pour créer les activités lors de la validation de la précondition.
4. L'information nécessaire pour le verrouillage des informations concernées par les préconditions.

Ces types d'informations sont un peu plus détaillés ci-dessous.

1. Informations permettant la **sélection des préconditions**

Ces informations sont essentiellement composées du type de la précondition (p.ex., insertion de faits, insertion d'entités etc.) et des faits, fonctions et classes impliqués (plus précisément ces données sont celles contenues dans l'ensemble d'informations, concernant les préconditions, établi par les règles de correspondance présentées au chapitre 6). Cette information est stockée dans des tables triées pour accélérer l' accès.

2. Informations permettant l'**évaluation de la précondition**

C'est le squelette d'évaluation de la précondition qui permet l'évaluation d'une précondition : il est généré à la compilation de la précondition par l'application de la règle de génération établie au chapitre 7.

Pour que les différentes requêtes contenues dans le squelette puissent être interprétées, elles doivent être traduites en un arbre d'opérateurs contenant des requêtes SQL : elles sont donc soumises au traducteur du Gestionnaire de Requêtes.

Rappelons-nous que le squelette d'évaluation contient plusieurs paramètres p.ex., les identificateurs des entités à mettre à jour. En effet, comme on le verra dans 8.3 le squelette est instancié lors des mises à jour.

Le squelette d'évaluation est généré sous forme d'arbre : c'est la représentation textuelle de cet arbre qui est stockée dans le catalogue. Deux fonctions permettant respectivement le passage d'un arbre à une chaîne de caractères et vice-versa ont été développées et sont utilisées pour stocker et récupérer le squelette d'évaluation dans le catalogue. En fait, tout arbre est stocké sous cette forme.

Par ailleurs, une des optimisations proposées sur le squelette d'évaluation n'est applicable qu'au moment de l'exécution des mises à jour : il s'agit du fait de tirer profit des constantes contenues dans la partie qualification des énoncés de mise à jour (cf.7.2.1). Le catalogue doit contenir les informations nécessaires pour que l'application de cette stratégie se fasse de façon efficace lors des mises à jour. Voyons alors quelles sont ces informations. Le principe de cette optimisation est de réaliser la mise en correspondance entre les conditions du type "y.G1. ... .Gt op valeur" contenues dans la mise à jour et dans la précondition. En conséquence, nous devons garder trace de ce type de conditions à la compilation de la précondition : précisément ces conditions sont récupérées de l'arbre sémantique de la précondition (voir ci-dessous) et enregistrées dans le catalogue.

3. Informations permettant la **création des activités**

Pour créer les activités lors de la validation d'une précondition nous avons besoin de connaître quelle est la classe de l'entité focale ainsi que les chemins du schéma conceptuel qui constituent le contexte d'activation. Ces informations sont récupérées de l'arbre sémantique de la précondition et stockées dans le catalogue.

4. Informations permettant le **verrouillage des informations**

Comme il a été dit dans 7.5, quand les préconditions sont évaluées, les informations qu'elles consultent doivent être verrouillées en lecture. Il faut alors stocker dans le catalogue un ensemble de requêtes à envoyer au Gestionnaire de Verrous permettant de verrouiller ces informations et de les libérer à la fin de l'évaluation.

Traitement d'une Précondition à la Compilation

Regardons maintenant comment une précondition est traitée par le Gestionnaire de Requêtes lors de sa définition :

1. La précondition suit, tout d'abord, le chemin de n'importe quelle requête FML, i.e., elle passe par l'Analyseur Lexico-Syntaxique et par l'Analyseur Sémantique. Pour cela, il faut que ces analyseurs prennent en compte les particularités des préconditions. Par rapport à l'analyse à réaliser dans le cas des mises à jour, les modifications sont peu nombreuses vu que les énoncés des mises à jour et des préconditions sont très proches (cf. 4.2).

2 A la sortie de l'Analyseur Sémantique, l'arbre sémantique est envoyé à une fonction qui s'occupe de dériver sa forme interne, i.e., la forme sous laquelle la précondition est stockée dans le catalogue.
Cette fonction utilise le Traducteur du Gestionnaire de Requêtes pour traduire les requêtes contenues dans le squelette d'évaluation.
La sortie de cette fonction est un arbre d'opérateurs permettant à l'interpréteur de stocker la définition de la précondition dans la base de données et l'association à sa classe d'activités.

La précondition n'est stockée dans le catalogue qu'au moment du chargement de sa classe d'activités (cf. 4.7).

Stockage du Contexte d'Activation

C'est pendant la compilation d'une précondition que le contexte d'activation d'une précondition est déterminé. Dans ce sous-paragraphe nous discutons sur les structures de données à adopter pour stocker les instances du contexte d'activation.

Le contexte d'activation a été défini (cf. 4.3) comme la photographie d'une partie de l'état de la base de données au moment de la validation de la précondition.

Comme il a été dit, le contexte d'activation a un schéma conceptuel : il est constitué de la partie du schéma général correspondant à la partie de la base de données qui a été photographiée.

La solution qui semble la plus naturelle pour réaliser son stockage est de créer les structures de données correspondant à son schéma. Ainsi, pour chaque activité les structures de stockage correspondant à la partie du schéma conceptuel du contexte d'activation sont dupliquées dans la base de données. L'une des structures contient l'état actuel de la base de données, l'autre, l'état aux moments de la création des différentes instances de l'activité.

A chaque classe d'activités correspondent alors des structures de stockage du contexte d'activation différentes.

Cette solution a l'avantage de permettre l'accès de façon homogène aux valeurs des faits au moment de la création des instances d'activités (plus précisément, de la validation de la précondition) et au moment de l'exécution des activités. A l'exécution d'une instance d'activité un masque doit être posé sur la structure de stockage des différents contextes d'activation appartenant à la même classe d'activités. L'intérêt de ce masque est de ne laisser l'instance d'activité accèder qu'à son contexte d'activation.

Ceci n'a pas été implémenté : cet aspect n'a pas été traité en profondeur dans le projet. En fait, nous avons adopté une solution de facilité qui permet de stocker le contexte d'activation de toutes les activités dans une seule relation. Pour cela, nous avions deux façons de procéder :

1. Enregistrer la liste d'instances de chemins du contexte d'activation (cf.7.4) sous une représentation textuelle. Ceci exige un travail d'interprétation au moment où on veut accéder au contexte d'activation de l'intérieur d'une activité.

2. Faire des restrictions sur le contexte d'activation de façon à ce qu'il puisse être stocké dans une structure "plate", comme une table, par exemple. Le contexte d'activation est un chemin dans le schéma conceptuel, il peut être stocké dans une structure "plate" s'il est du type : F1. ... .Fn-1.Fn, où Fi, $1 \leq i \leq n-1$ est une fonction monovaluée et Fn est une fonction monovaluée ou multivaluée.
   Cette solution exige aussi un traitement d'interprétation quand on accède au contexte d'activation de l'intérieur d'une activité. Mais vu les restrictions imposées le travail à fournir est énormément réduit.

Pour l'implémentation d'un premier prototype c'est la deuxième solution qui a été retenue. Nous ne détaillons pas ici les différents constituants de la table qui enregistre les instances du contexte d'activation.

8.3. Evaluation des Préconditions

Le principe le plus important d'optimisation de l'évaluation des préconditions est que cette évaluation doit être intégrée à l'exécution de la mise à jour. Le squelette d'évaluation généré lors de la compilation d'une précondition reflète l'application de ce principe. Au niveau implémentation il implique une évaluation en deux étapes :

1. La **sélection des préconditions** à évaluer est intégrée à la **compilation de la mise à jour**.
2. L'**évaluation des préconditions** sélectionnées est intégrée à l'**interprétation de la mise à jour**.

8.3.1. Sélection des Préconditions

Compiler une mise à jour implique consulter le catalogue de la base de données pour obtenir des informations sémantiques sur les entités concernées et ses associations, et des informations sur le stockage de ces entités indiquant quelles structures de données doivent être interrogées ou modifiées. Ces informations sont du même type que celles nécessaires à la sélection des préconditions concernées par la mise à jour. En réalisant la compilation de la mise à jour et la sélection des préconditions de façon intégrée le nombre d'accès au catalogue, de structures de données et d'étapes intermédiaires est réduit. De plus, les accès au catalogue sont ainsi restreints à l'étape de compilation de l'énoncé, ce qui évite des accès concurrents au catalogue lors de l'exécution.

A la fin de la traduction d'une mise à jour une fonction, appartenant au Gestionnaire d'Exécution, est appelée pour accomplir la sélection des préconditions à évaluer pour cette mise à jour particulière. Cette fonction a accès aux arbres (représentant la mise à jour) issus de l'Analyseur Sémantique et du Traducteur.

La sélection des préconditions implique la réalisation des tâches suivantes :

1. Faire la mise en correspondance entre la mise à jour et l'ensemble des préconditions du catalogue. Ceci est fait en appliquant les règles de correspondance établies au chapitre 6. Ces règles sont implémentées par une fonction qui accède d'une part à l'arbre sémantique de la mise à jour et d'autre part au premier type d'information généré lors de la compilation de la précondition (cf. 8.2).

2. Pour chaque précondition sélectionnée, traiter le squelette d'évaluation. Le squelette d'évaluation considère, pour certains types de préconditions, plusieurs stratégies. Par exemple, pour une insertion de faits il propose une stratégie pour évaluer des mises à jour d'insertion de faits et une autre pour les mises à jour d'insertion d'une entité. Il faut décider, à ce moment-là, de la stratégie à appliquer.
   L'application de l'optimisation consistant à tirer profit des constantes contenues dans la partie qualification de la mise à jour est aussi réalisée à ce moment-là, par la mise en correspondance de certaines conditions contenues dans la mise à jour et dans la précondition (cf. 7.2.1 et 8.2). Ceci est implémenté par une fonction qui consulte l'arbre sémantique de la mise à jour et certaines informations dans le catalogue des préconditions (cf. 8.2). L'objectif final de cette fonction est de modifier le squelette d'évaluation de façon à substituer certaines requêtes par des tests filtres qui font référence aux constantes dans la mise à jour.

3. Rajouter à l'arbre de la mise à jour : la partie du squelette d'évaluation contenant la bonne stratégie d'évaluation, la description de l'entité focale et du contexte d'activation et les requêtes de verrouillage. Etablir les références entre les informations de la mise à jour et le squelette d'évaluation de la précondition. La figure 8.3 illustre, de façon figurative, comment ces références sont établies. Elle est commentée ci-dessous.

Nous supposons la précondition suivante :

```
TEST     INSERT  [Employé e, Département d]
         FROM    Employé e, Département d
         WHERE   e. Catégorie IN ('Ingénieur', 'Secrétaire')
```

La figure 8.3 (a) illustre le cas où il faut la condition "e. Catégorie IN ('Ingénieur', Secrétaire')" peut être vérifiée directement sans consulter la base de données vu que la catégorie de l'employé est mentionnée dans la mise à jour. La figure 8.3 (b) illustre le cas où il faut un accès à la base de données pour obtenir la catégorie des employés concernés par la mise à jour.

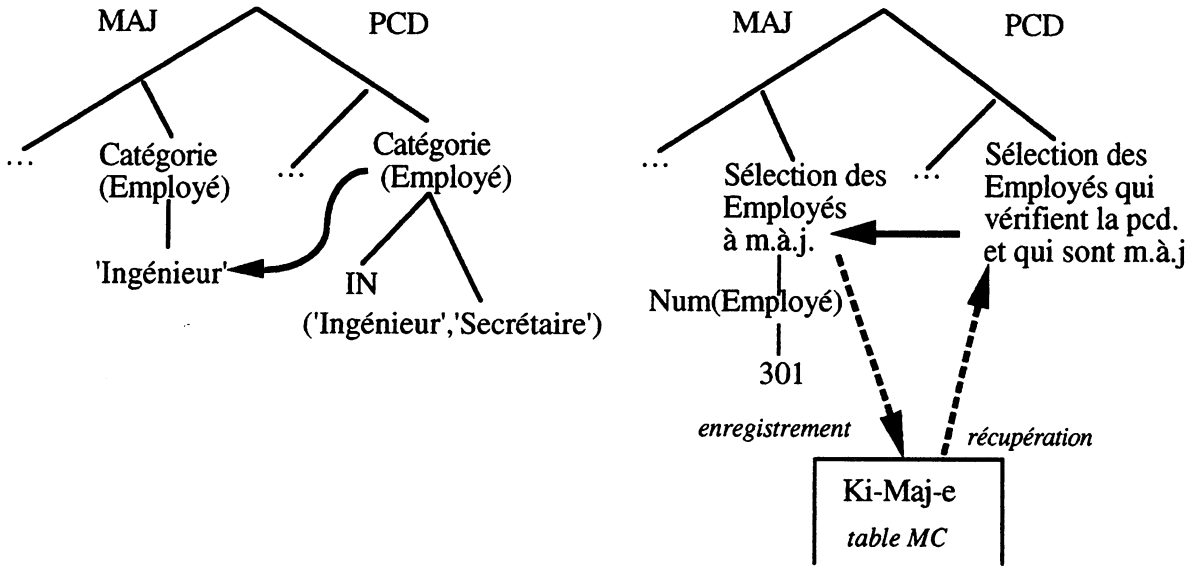


Figure 8.3 - Sortie de la Traduction d'une Mise à Jour

8.3.2. Evaluation Proprement Dite

Une fois que la compilation d'une mise à jour est accomplie, l'arbre issu du Traducteur est transmis à l'Interpréteur qui effectue son exécution.

Comme il a déjà été vu, dans le cas général cet arbre contient la traduction de la mise à jour en deux types d'opérations : des opérations de sélection pour obtenir les entités concernées par la mise à jour et des opérations de modification pour réaliser la mise à jour.

Lors de l'interprétation les opérations de sélection sont exécutées, les entités concernées par la mise à jour sont ainsi obtenues. A ce moment, le Gestionnaire de Transactions accède directement à cette information pour construire le journal de reprise. Ensuite, la même information est passée à l'Evaluateur de Préconditions qui l'utilise pour évaluer les préconditions, et si elles sont validées, pour construire les activités à créer avec leur entité focale et leur contexte d'activation.

L'Evaluateur de Préconditions réalise l'évaluation en trois étapes :

1. Instanciation du squelette d'évaluation avec des informations sélectionnées par l' Interpréteur et le Gestionnaire de Transactions ou avec des informations contenues dans la mise à jour.

2. Exécution du squelette d'évaluation instancié. Si la précondition est validée, toutes les instances de chemin du contexte d'activation qui la satisfont sont enregistrées.

3. Si la précondition est validée les activités correspondantes sont construites avec leurs entité focale et contexte d'activation.

Nous ne détaillons plus ici le rôle de l'Evaluateur de Préconditions. La figure 8.4 donne une rapide idée de la façon dont les opérations pour interpréter une mise à jour et évaluer les préconditions correspondantes sont réalisées. Elle considère la précondition et la mise à jour suivantes :

```
pcd :    TEST     DELETE  [Vendeur v, Localité l]
                  FROM    Vendeur v, Vendeur.Localité l
                  WHERE   l. Catégorie IN ('l1', 'l2')

maj :             DELETE  Vendeur
                  WHERE   Vendeur.Département = 'Outils'
```

Nous avons supposé une association entre Vendeur et l'entité littérale Localité.

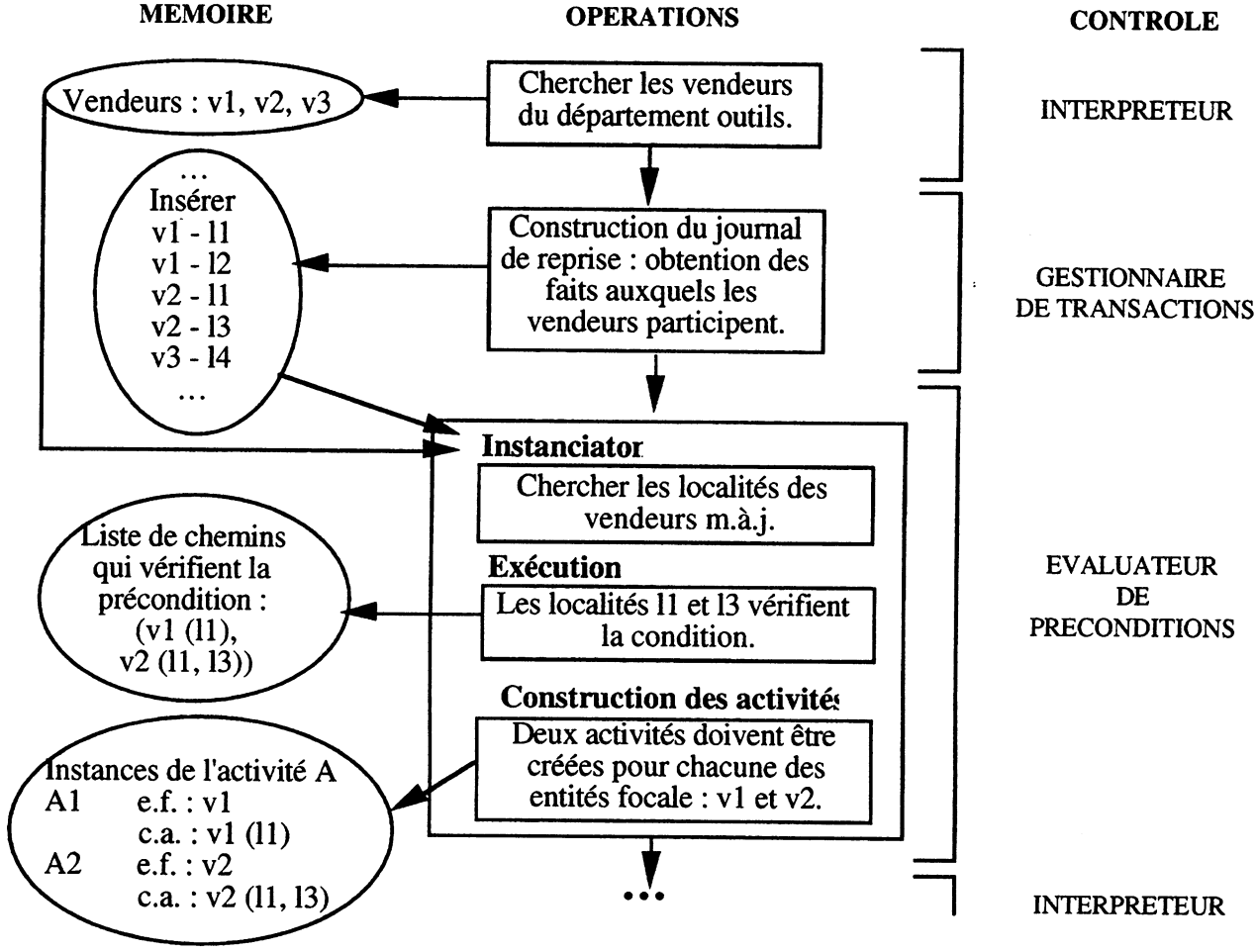


Figure n° 8.4 - Evaluation d'une Précondition

8.4. Compilation d'un Schéma de Procédure

La définition d'une procédure est réalisée par un énoncé d'insertion d'une entité. Cet énoncé est traité par le Gestionnaire de Requêtes comme toute autre requête FML. Néanmoins, le schéma de la procédure étant défini comme un type abstrait, nécessite un traitement spécial. Pour décrire le schéma, le langage OPL3 est utilisé (cf. 5.1): il permet la séquence, le parallélisme, la synchronisation et le choix de l'ensemble d'activités qui constitue une procédure.

La compilation d'un schéma résulte en un arbre qui décrit la façon dont les activités sont ordonnées à l'intérieur d'une procédure. C'est cet arbre qui va permettre d'évaluer les contraintes de précédance pour qu'une activité à l'intérieur d'une procédure puisse être lancée. Ainsi, il doit être stocké de façon à ce que cette évaluation puisse aboutir efficacement.

Nous avons défini une relation dans la base de données ("Cl-Schéma) qui permet de stocker le schéma des procédures. De plus, une autre relation (Inst-Schéma) est nécessaire pour stocker les instances des schémas. Plus précisément, pour une instance de procédure cette relation contient les

instances de ces activités ainsi que des références à leur place dans l'arbre du schéma de la procédure.

Le Gestionnaire d'Exécution utilise ces deux relations pour évaluer les contraintes de précédance (ceci sera détaillé dans 8.5.2). Brièvement, s'il y a au moins une des instances des activités de l'instance d'une procédure qui est en train de s'exécuter le Gestionnaire d'Exécution tient à jour une table en mémoire centrale qui contient :

- La description du schéma contenue dans la relation "Cl-Schéma".
- La partie de la relation "Inst-Schéma" qui concerne l'instance de la procédure qu'on veut traiter.
- Des informations sur l'état des instances d'activités qui constituent la procédure. Ces informations sont stockées dans les structures de base correspondant aux définitions standards de Procédure et Activité.

A partir de là, le Gestionnaire d'Exécution peut très facilement contrôler que l'exécution des activités respecte le schéma de la procédure comme on le verra plus tard.

La description des relations Cl-Schéma et Inst-Schéma suit.

La relation **Cl-Schéma** décrit les schémas des classes de procédures. Pour chaque classe de procédures il y a autant de n-uplets que de classes d'activités qui constituent le schéma. Les constituants de cette relation sont tous de type entier, ils sont énumérés ci-dessous :

Cl-Procédure : Surrogate[1] de la classe de procédures à laquelle appartient le schéma.
Cl-Activité : Surrogate d'une des classes d'activités qui constitue Cl-Procédure.
Id-Dép : Identificateur de l'activité Cl-Activité à l'intérieur de la procédure. Cet identificateur est généré de façon interne et permet entre autre : de faire un lien rapide avec les instances d'activités stockées dans Inst-Schéma, et d'éviter des ambigüités lorsqu'un schéma contient plusieurs occurences de la même classe d'activités.
Pt-Syncro : Contient le nombre d'activités desquelles Cl-Activité dépend directement.
Suivant : Surrogate d'une des classes d'activités qui dépend directement de la calsse Cl-Activité.
Parallél : Si l'activité Cl-Activité doit être lancée en parallèle avec d'autres, alors Id-Dép vaut le Id-Dép correspondant à une des classes d'activités à lancer en parallèle, sinon il vaut zéro.
Choix : Si Cl-Activité doit être exécutée en alternative avec d'autres, alors Choix vaut le Id-Dép d'un des classes d'activités alternatives, sinon il vaut zéro.

La relation **Inst-Schéma** contient les instances d'activités appartenant à une instance de procédure. Pour chaque instance de procédures il y a autant de n-uplets que d'instances d'activités qui lui sont associées. Les constituants de cette relation sont tous de type entier, ils sont énumérés ci-dessous :

Inst-Activité : Surrogate d'une instance d'activité associée à une procédure.
Inst-Procédure : Surrogate de l'instance de procédures à laquelle Inst-Activité est associée.

[1]Nous utilisons le terme surrogate pour désigner identificateur unique.

Id-Dép : Identificateur de la classe d'activités, à l'intérieur de la procédure, à laquelle Inst-Activité est associée. Ceci permet de faire le lien avec la relation Cl-Schéma.
Autre-Inst : Comme nous avons vu dans 5.1, à une classe d'activités à l'intérieur d'une procédure peuvent correspondre plusieurs instances d'activités à exécuter. Si plusieurs instances d'activités dans cette relation ont la même valeur de Id-Dép, alors Autre-Inst vaut le surrogate d'une de ces instances, sinon il vaut zéro.

Nous ne présentons ici que les structures de données, le traitement fait sur ces relations sera détaillé en 8.5.1.

8.5. Exécution et Contrôle des Activités et Procédures de Bureau

La fonction principale du Gestionnaire d'Exécution est de contrôler l'habilitation, le lancement et l'exécution des activités et procédures. Nous nous plaçons tout d'abord au niveau activité (8.5.1). Après quoi, les extensions nécessaires pour aborder le niveau procédure seront présentées (8.5.2). A ce niveau, une tâche très importante doit être accomplie par ce gestionnaire, il s'agit de l'évaluation des contraintes de précédance des activités à l'intérieur d'une procédure dans le but de pouvoir décider des activités à lancer.

8.5.1. Gestion des Activités

Pour accomplir sa fonction principale, le Gestionnaire d'Exécution traite des ordres de deux types venus du noyau SIB. Ce sont :

a. **Des alertes de la part de l'Evaluateur de Préconditions.** Ces alertes informent le Gestionnaire d'Exécution que de nouvelles instances d'activités doivent être créées et lui fournissent les informations nécessaires à la création des instances d'activités (i.e., leur identificateur interne, leur instance d'entité focale et leur contexte d'activation).
La réception de cet ordre implique donc la création effective des instances d'activités dans la base de données avec leur état habilité. De plus, si jamais l'activité est automatique, le Gestionnaire d'Exécution produit un ordre interne de lancement de l'activité. La réaction à cet ordre est décrite ci-dessous (b).

b. **Des requêtes d'exécution de la part de l'interpréteur.** Ces requêtes sont générées comme conséquence de l'exécution d'une des primitives OPCL. Ces primitives demandent soit le lancement, soit la terminaison d'un ensemble d'instances d'activités. Il est à noter que l'interpréteur sélectionne tout d'abord les activités auxquelles les primitives s'appliquent et envoie leurs identificateurs au Gestionnaire d'Exécution. Très brièvement, nous avons :
   - **Lors d'un ordre de lancement**, le Gestionnaire d'Exécution doit tout d'abord consulter la base de données pour obtenir certaines informations comme le corps des

activités à exécuter, le contexte d'activation, le site et la session où l'activité doit être exécutée, ainsi que l'adresse du terminal (cf.4.6). Ensuite, le Gestionnaire d'Exécution doit lancer l'activité et mettre à jour son état dans la base de données.

- **Lors d'un ordre de terminaison**, le Gestionnaire d'Exécution doit mettre à jour l'état des instances des activités concernées par la commande.

Détaillons alors comment la mise en œuvre du lancement proprement dit d'une activité est accomplie. Tout d'abord nous considérons plusieurs cas :

- Si **l'activité est associée à un programme**, un processus est lancé pour permettre l'exécution de l'activité. Ce processus lance une nouvelle session SIB pour que l'activité puisse mener à bout les accès au SIB. A la fin de l'activité un ordre de terminaison est issu de cette session SIB et le Gestionnaire d'Exécution est ainsi averti. Dans la suite nous classifions ces activités de ***type 1***.
  Maintenant, au niveau implémentation nous traitons d'un mode particulier **les activités qui n'ont pas besoin d'accéder au SIB**. Ces activités ne sont pas très représentatives : souvent elles sont définies dans le but de récupérer des outils déjà existants (avant SIB) comme par exemple des traitements de textes. Pour exécuter ces activités ce n'est pas la peine d'ouvrir une session SIB : son seul rôle serait de traiter les primitives de terminaison. Ainsi, à la compilation de l'activité cette primitive est modifiée par le retour d'un code de terminaison. Le rôle du Gestionnaire d'Exécution est alors de lancer un processus permettant l'exécution de l'activité et de mettre à jour la base de données lors de la réception du code de terminaison. Dans la suite nous classifions ces activités de ***type2***.

- Si **l'activité n'est pas associée à un programme**, l'exécution de l'activité se passe dans la session SIB qui a demandé son lancement. Dans ce cas, aucun processus n'est lancé : le Gestionnaire d'Exécution a comme fonction de mettre à jour l'état de l'activité. A la fin de l'activité l'utilisateur ou l'application doit demander l'exécution d'une primitive de terminaison. L'interpréteur appelle alors le Gestionnaire d'Exécution pour qu'il accomplisse la terminaison de l'activité, i.e., il effectue la mise à jour de l'état de l'activité. Dans la suite nous classifions ces activités de ***type3***.

Pour implémenter le lancement d'une activitré nous avons réalisé un modèle client-serveur entre le Gestionnaire d'Exécution et le site où on souhaite exécuter l'activité. L'architecture de ce modèle est présentée dans la figure 8.5. Cette figure prend en compte les trois types d'activités définis ci-dessus. Elle est commentée par la suite.

Dans chaque site d'exécution il existe un serveur (un processus résidant sur l'ensemble des clients SIB) ayant comme rôle la réception d'un ordre de lancement d'une activité par le Gestionnaire d'Exécution.

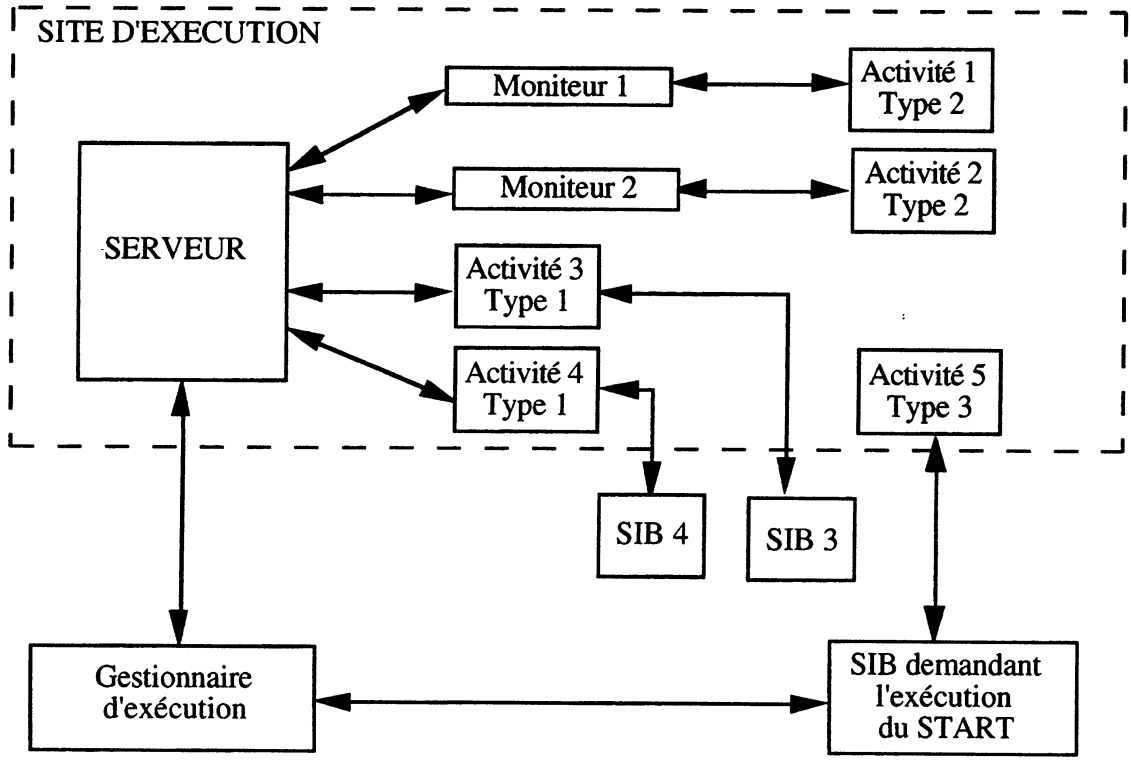


Figure n° 8.5 - Exécution d'Activités

Le lancement des activités des trois types est accompli de façon suivante :

- **Activité *type1*** : le Gestionnaire d'Exécution lance un ordre de lancement au serveur du site où on souhaite exécuter l'activité. Ce serveur lance le processus ayant comme but l'exécution de l'activité, avertit le Gestionnaire d'Exécution et se libère. Le processus réalisant l'activité se connecte à une nouvelle session SIB. Cette session se chargera d'avertir le Gestionnaire d'Exécution de la terminaison de l'activité.

- **Activité *type2*** : le Gestionnaire d'Exécution lance un ordre de lancement au serveur du site où on souhaite exécuter l'activité. Ce serveur crée un processus moniteur (pour chaque activité à lancer). Ce moniteur est chargé de lancer un processus pour l'exécution de l'activité, d'avertir le Gestionnaire d'exécution que l'activité a été lancée, d'attendre la fin de l'exécution de l'activité et d'avertir la terminaison de l'activité au Gestionnaire d'Exécution. La durée de vie d'un moniteur est celle de l'exécution d'une activité.
  Le fait de créer un moniteur par activité à lancer permet au serveur de se libérer et pouvoir recevoir de nouveaux ordres de lancement.

- **Activité *type3*** : Le Gestionnaire d'Exécution a comme rôle la mise à jour des états des activités lors de la réception d'un ordre de lancement ou de terminaison.

La communication entre les différents processus se fait toujours par appel de procédures à distance (RPC). La figure 8.6 présente la trace temporelle de l'exécution d'une activité et décrit le lancement d'une activité de chaque type.

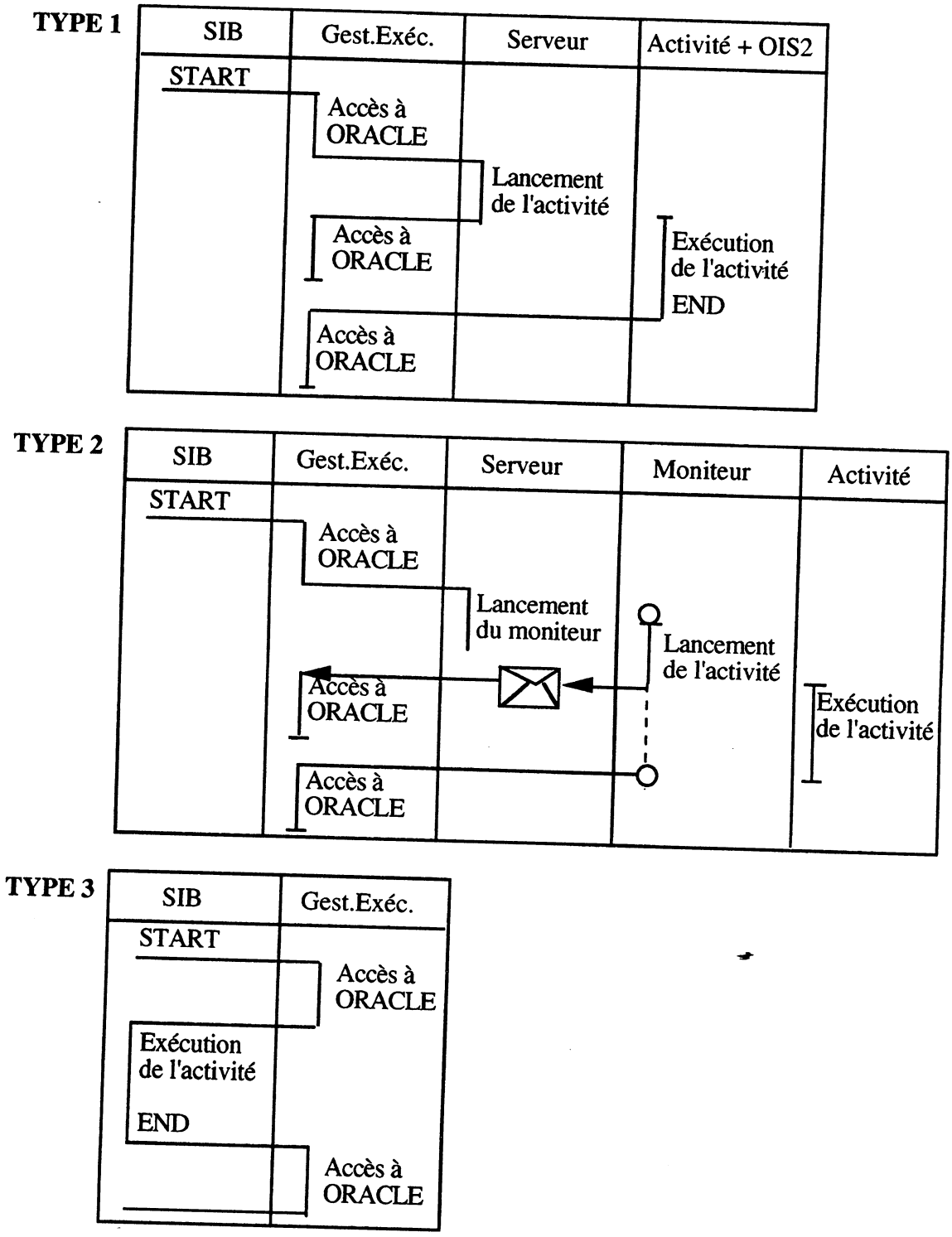


Figure n° 8.6 - Trace Temporelle de l'Exécution d'une Activité

Nous finissons ici la description du Gestionnaire d'Exécution en ce qui concerne une activité : pour plus de détails le lecteur pourra se référer à [Hou89].

8.5.2. Gestion des Procédures

Ce paragraphe fait la description de la façon dont le Gestionnaire d'Exécution évalue les contraintes de précédance dans le but de pouvoir déterminer les activités à lancer. Une première idée a été présentée dans 8.4.

Le Gestionnaire d'Exécution tient une structure de données en mémoire centrale qui est instanciée à partir d'informations contenues dans la base de données. Cette structure permet de contrôler l'exécution d'une instance de procédure. Elle concerne seulement les procédures dont au moins une activité est en train de s'exécuter. Les autres procédures sont considérées en repos (et elles peuvent rester en repos pour une longue période de temps) : en conséquence, le Gestionnaire d'Exécution ne tient aucune informations sur elles en mémoire centrale.

Chaque fois qu'une procédure sort du repos, cette structure en mémoire centrale est chargée à partir de la base de données.

La structure du Gestionnaire d'Exécution est illustrée dans la figure 8.7 ; elle est définie par trois tables :

a. La première table, **Inst-Proc**, contient la liste des instances de procédures dont au moins une activité est en train de s'exécuter.
   Une entrée dans cette table contient alors le surrogate de l'instance d'une procédure et une référence à une table Etat-Schéma décrite dans b.

b. La deuxième table, **Etat-Schéma**, décrit l'état général dans lequel l'exécution de la procédure se trouve. Cette table a une entrée pour chaque classe d'activités associée à la classe de procédures : la classe d'activités est identifiée ici par son identificateur à l'intérieur de la procédure (Id-Dép). Cette table contient :
   - La description du schéma de la classe de procédures correspondant à l'instance de procédure qui pointe sur la table.
     Cette description est obtenue directement à partir de la relation Cl-Schéma (cf. 8.4), en particulier, de ses constituants Id-Dép, Pt-Synchro, Suivant, Parallèle et Choix.
   - L'état général de l'exécution de ce schéma. Cet état est décrit par deux colonnes. La colonne "Accomplie" indique si toutes les instances de la classe identifiée par Id-Dép sont dans l'état terminé. La colonne "Compteur" indique le nombre de classes d'activités - dont la Classe Id-Dép dépend directement - qui ont déjà été accomplies.
     Au chargement de la table, ces informations sont obtenues en consultant la relation Inst-Schéma (cf. 8.4), et les structures de stockage (dans la base de données) qui représentent les instances de la classe Activity et ses associations.

c. La troisième table, **Inst-Activité**, donne l'état de chaque instance d'activité appartenant au schéma. Elle a une entrée par instance d'activité et contient son surrogate, son état (i.e. habilitée, lancée, ...) et le Id-Dép de la classes d'activités correspondante.

Nous insistons sur le fait que cette table existe car à chaque classe d'activités dans le schéma peut correspondre un ensemble d'instances.
Au chargement de cette table les informations sont obtenues par consultation de la relation Inst-Schéma et des structures de stockage (dans la base de données) correspondant à la classe "Activity" et ces associations.

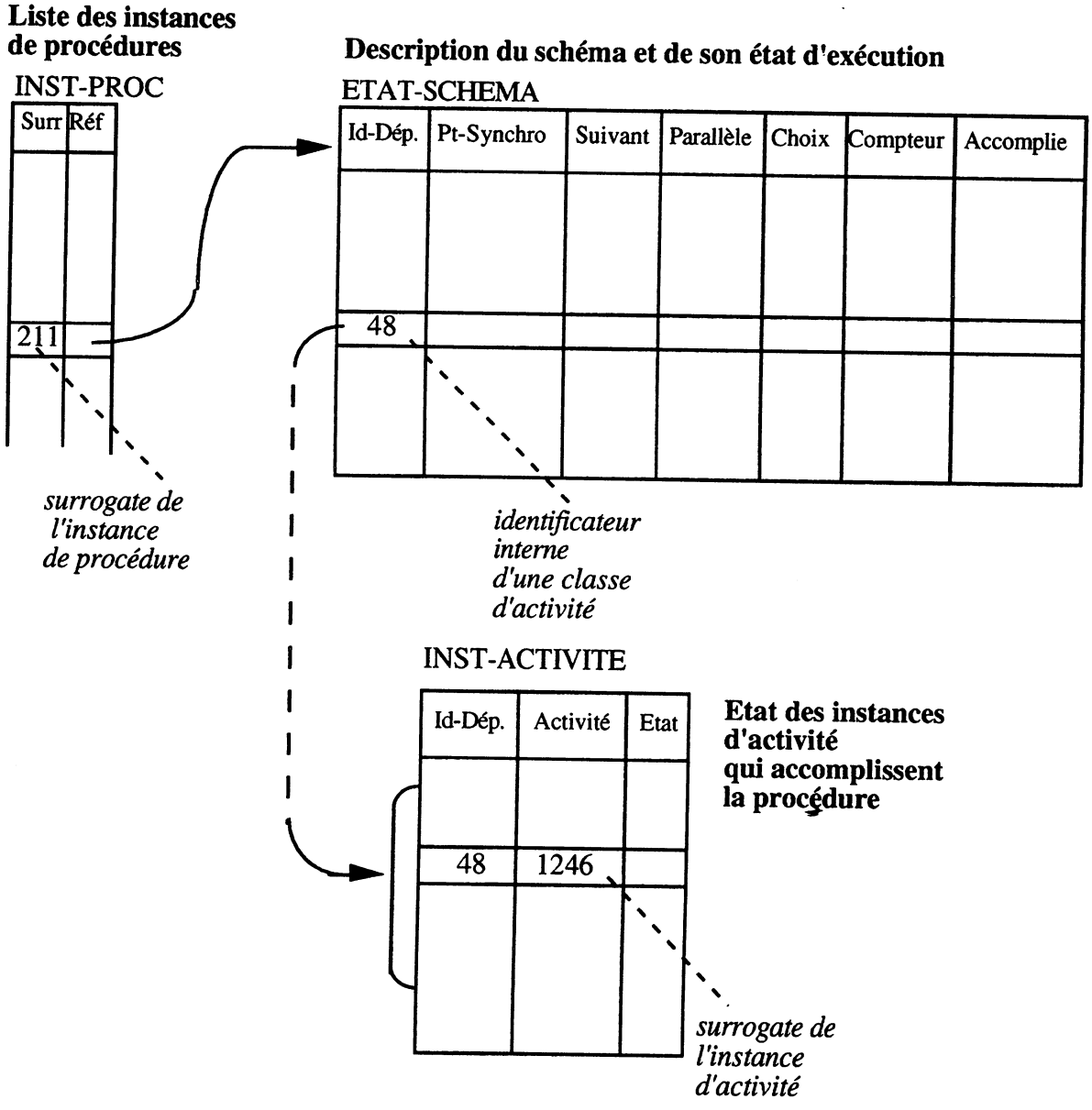


Figure n° 8.7 - Structures permettant le Contrôle de l'Exécution des Procédures

A chaque primitive OPCL (en particulier celles d'habilitation, de lancement ou de terminaison d'une activité à l'intérieur d'une procédure) correspond une fonction dans le Gestionnaire d'Exécution qui manipule les tables qui viennent d'être décrites. Nous ne détaillons pas ici les algorithmes de ces fonctions, néanmoins quelques commentaires suivent :

- La primitive d'**habilitation** d'une activité implique l'ajout d'une entrée dans la table Inst-Activité et la répercussion de cet ajout sur la base de données.

- La primitive de **lancement** d'une activité implique la vérification des contraintes de précédance. Si Etat-Schéma[Pt-Synchro] = Etat-Schéma[Compteur] alors ces contraintes sont satisfaites ; dans ce cas, si l'activité à lancer est dans l'état habilité, le lancement peut aboutir.
  Le lancement proprement dit d'une activité à l'intérieur d'une procédure suit le même chemin que celui d'une activité individuelle (cf. 8.5.1).
  Evidemment la mise à jour de l'état de l'activité doit être effectuée en mémoire centrale et dans la base de données.

- La primitive de **terminaison** d'une activité provoque :
  - La mise à jour de l'état de l'activité en mémoire centrale et dans la base de données.
  - Si toutes les instances d'activités, qui ont le même Id-Dép que l'activité qui vient de se terminer, sont dans l'état "Terminé", alors :
    - mettre à jour Etat-Schéma[Accomplie] de la classe d'activité Id-Dép.
    - mettre à jour Etat-Schéma[Compteur] des activités dépendant de celle qui vient de se terminer.
    - vérifier s'il y a d'autres activités à lancer (en séquence , en paralléle ou alternativement).
    - vérifier si la procédure est finie, i.e., toutes les activités ont une valeur positive de Etat-Schéma[Accomplie].

Nous venons de survoler les principales fonctions dont le Gestionnaire d'Exécution est responsable pour accomplir la gestion des activités et des procédures. Nous n'avons pas abordé tous les aspects, mais seulement ceux qui nous semblent les plus importants.Toutefois, il est important de signaler:

Les primitives d'habilitation, lancement ou terminaison sont systématiquement traduites en un ensemble de requêtes au Gestionnaire de Transactions et à celui de Verrous (et évidemment au Gestionnaire d'Exécution). Ceci est fait dan le but de mettre en œuvre les liens entre l'exécution d'une activité et le système transactionnel et le système de verrouillage de données.

Par exemple, une primitive de terminaison implique la génération de requêtes qui permettent de confirmer toutes les transactions ouvertes pendant l'exécution de l'activité et la libération ou la propagation des verrous à la session SIB où l'activité se déroule.

8.6. Conclusion

Nous venons de discuter les aspects les plus importants de la mise en œuvre d'un prototype d'un Serveur d'Information pour la Bureautique. L'implémentation des aspects dynamiques d'un tel Serveur nous a permis de définir les différents éléments nécessaires à leur réalisation, et leurs interactions. En effet, dans ce chapitre, ces éléments et leurs interfaces avec les autres modules/processus du Serveur ont été traités.

Nous pouvons aussi conclure, qu'au niveau implémentation, la prise en compte des aspects dynamiques ne représente pas un travail très important une fois que :

- Les modèles conceptuels statiques et dynamiques sont bien intégrés (cf. 3, 4 et 5).

- La définition formelle de l'évaluation des préconditions et des différentes techniques d'optimisation est réalisée (cf. 6 et 7).

Nous finissons cette conclusion en énumérant les restrictions effectuées au niveau du prototype :

- En ce qui concerne le FML, seules des associations binaires ont été implémentées. Remarquez que ceci ne diminue pas le pouvoir d'expression du modèle.
  La mise en œuvre de l'énoncé de modification de faits n'a pas été réalisée.

- Par rapport au mécanisme d'évaluation des préconditions, nous avons implémenté seulement le cas des insertions. Ceci pour une question de temps et non pour des raisons de difficulté. En effet, comme nous l'avons vu dans les chapitres 6 et 7, une fois que le mécanisme d'évaluation a été défini pour les préconditions d'insertion de faits, le traitement des autres types de préconditions va de soi.

- Relativement au Gestionnaire d'Exécution, il a été implémenté mais il n'a pas été intégré au Serveur.

De plus, une interface graphique pour la définition des activités et des procédures ainsi que pour le suivi de leur exécution a été développée bien que son intégration à ce Serveur n'ait pas été réalisée. Elle n'a pas été présentée ici : le lecteur pourra se référer à [Hou89]

CHAPITRE 9

9. CONCLUSION

Cette thèse propose un modèle et un mécanisme d'exécution pour les bases de données actives. Nous nous plaçons dans le cadre d'un **support de base** offrant les services nécessaires au développement d'applications de gestion d'activités et de procédures pour aboutir à un système complet. Le fait de se placer à ce niveau a impliqué une certaine démarche :

- D'un côté, nous avons été amenés à prendre en compte les **besoins des applications**, plus précisément, de l'expression des aspects dynamiques vis-à-vis de l'application. Il a donc été nécessaire de proposer un modèle ainsi que des langages de spécification permettant aux applications de décrire leur aspect dynamique. Ceci concerne les niveaux conceptuel et opérationnel.

- D'un autre côté, nous avons été amenés à prendre en compte les **mécanismes sous-jacents aux SGBD**. Nous avons alors traité l'aspect fonctionnel permettant la mise en œuvre du modèle et ses langages en termes de mécanismes implémentés dans le SGBD.

En conclusion, cette approche nécessite le traitement des aspects dynamiques de façon complète. Ainsi, nous abordons d'une part, les différents niveaux nécessaires à la modélisation et gestion des applications bases de données : niveau **conceptuel**, niveau **opérationnel** et niveau **fonctionnel** (cf.1.4). D'autre part, pour les deux derniers niveaux, discussion, illustration et analyse en profondeur ont abouti à leur mise en œuvre.

Bien que développés dans le cadre d'un serveur pour la bureautique, les fonctions de gestion des tâches sont applicables à d'autres domaines.

Les principaux résultats sont repris ici. Après quoi, nous ébaucherons les lignes de perspectives.

1. Niveau Conceptuel

Deux niveaux de représentation des traitements sur les données ont été définis. Le niveau de base introduit la notion d'**activité** comme une tâche réalisée pendant une certaine période par une personne ou un groupe de personnes bien défini. Le niveau haut introduit la notion de **procédure** édifiée sur celle d'activité : une procédure est construite à partir d'une "bibliothèque" d'activités en les ordonnant de façon à produire l'éffet souhaité.

Le lien entre la partie statique et dynamique a été traité par la notion d'**entité focale**. Cette notion permet de définir le comportement des données statiques par les activités et procédures qui lui sont appliquées et est un bon moyen pour modéliser le flux de données à l'intérieur d'une procédure.

2. Niveau Opérationnel

Nous distingons deux aspects dans le niveau opérationnel : la spécification formelle de la représentation des activités et procédures au niveau d'un **modèle** et les **outils et techniques** nécéssaires à la mise en œuvre de ce modèle.

2.1. Modèle

Les activités et les procédures sont considérées comme des entités spéciales - les entités dynamiques - des applications : elles sont modélisées en termes du modèle de données. Nous avons abouti à un modèle permettant de définir d'un côté l'aspect **descriptif** des activités (classe *ClasseActivités*) et des procédures (classe *ClasseProcédures*) et d'un autre côté leur aspect **exécutable** (classes *Activités* et *Procédures*).

La notion d'entité focale est le véhicule utilisé pour modéliser les associations entre les entités statiques et les entités dynamiques : ceci au niveau du schéma conceptuel et de la base de données. Toute entité statique est un candidat à être l'entité focale d'une activité ou d'une procédure.

2.1.1. Modélisation de la Description d'une Activité

Une classe d'activités est décrite par la classe d'entité focale à laquelle elle s'applique mais aussi par :

- **Un contexte d'activation** : il décrit l'information critique manipulée par l'activité. Cette information est décrite dans une partie du schéma conceptuel par un chemin défini dans la précondition.
- **Une précondition** : elle spécifie les conditions nécessaires pour qu'une instance d'activité puisse être créée. Deux types de précondition ont été définis : les préconditions de changement qui contrôlent les changements portant sur l'entité focale, et les préconditions d'état qui testent l'état dans lequel l'entité focale se trouve.
- **Un corps d'exécution** : il définit le traitement à entreprendre pendant l'exécution d'une activité. Ce corps est considéré comme une "boîte noire".
- **Un mode de lancement** : il définit le moment de lancement de l'exécution d'une activité : lors de son habilitation (i.e., à la validation de la précondition) ou arbitrairement à la demande d'une application.
- **Un état** : il spécifie si l'activité est chargée - dans ce cas, sa précondition est évaluée lors des mises à jour et des instances d'activité peuvent être créées - ou déchargée - dans ce cas, sa précondition n'est pas évaluée.

2.1.2. Modélisation de l'Exécution d'une Activité

Une activité est décrite par l'instance de l'entité focale à laquelle elle s'applique mais aussi par :

- **Une instance du contexte d'activation** : elle est considérée comme une photographie de la base de données au moment où la précondition a été validée.

- **Un état** : indiquant si l'activité est habilitée à être lancée, si elle est lancée ou comment son exécution a abouti (terminaison normale ou anormale).

2.1.3. Modélisation de la Description d'une Procédure

Une classe de procédures est liée aux mêmes informations qu'une classe d'activités, de plus elle est associée à :

- **Un ensemble de classes d'activités** : les classes activités dont elle est composée.
- **Un schéma** : il indique les contraintes de précédance des activités à l'intérieur de la procédure ainsi que les liens entre les entités focales des activités et celle de la procédure.

2.1.4. Modélisation de l'Exécution d'une Procédure

Une procédure est définie tout d'abord comme une activité (à part qu'elle peut avoir d'autres types d'état : suspendu, actif et inactif). De plus, elle est définie par :

- **Un ensemble d'activités** : les activités qui permettent à la procédure d'atteindre son objectif.
- **Une instance d'un schéma** : elle indique l'état d'exécution dans lequel la procédure se trouve.

2.2. Outils et Techniques

Vu que les activités et les procédures sont considérées comme des entités du schéma conceptuel des applications, elles sont, pour une bonne partie de leurs fonctions, manipulées en utilisant les LDD et LMD standards offerts par le système. Néanmoins, elles ont une sémantique particulière qui doit être prise en compte ; pour cela, plusieurs langages ont dû être définis. Ces langages restent dans la mesure du possible proches du DML.

2.2.1. Langage de Spécification des Préconditions

Ce langage est un sur-ensemble du DML. L'introduction de la notion de préconditions et de son langage à nécessité une étude permettant leur formalisation. D'un côté, nous avons étudié les **règles de correspondance** entre les mises à jour et les préconditions et d'un autre côté, une **méthode d'évaluation** a dû être développée.

2.2.1.1. Règles de Correspondance entre les Mises à Jour et les Préconditions

Ces règles gouvernent l'activation d'une précondition lors des mises à jour. Ce travail peut être éclairé de deux points de vue : d'un point de vue conceptuel, les règles de correspondance établissent de façon formelle la sémantique des préconditions. D'un point de vue technique elles sont la base du traitement des préconditions.

Trois aspects influencent l'établissement de ces règles : la prise en compte de la **hiérarchie de généralisation**, l'**inclusion de faits** et l'**inclusion sémantique des opérations**.

Nous voulons mettre l'accent sur l'importance de l'aspect qui concerne l'application d'une précondition à travers la hiérarchie de généralisation. Nous avons dédié une bonne partie de ce rapport à la formalisation de la sémantique qui gouverne l'héritage des préconditions. Soulignons aussi que ceci a été mené en considérant la spécialisation simple mais aussi les trois types de spécialisation composée : union, intersection et différence.

2.2.1.2. Définition d'une Méthode d'Evaluation

Le principe qui est à la base de cette méthode est que l'évaluation des préconditions se fait de façon intégrée à l'exécution de la mise à jour en vue de diminuer les accès à la base de données. Ce principe est mis en œuvre par la dérivation à partir de la précondition d'une expression optimisée - le **squelette d'évaluation** - de la partie qualification de la précondition. Grâce à ce principe, la dérivation du squelette d'évaluation peut jouir de l'application des deux techniques d'optimisations suivantes :

- Limitation de l'évaluation de la précondition à l'ensemble d'informations qui est mis à jour. Ceci est fait en récupérant des résultats intermédiaires obtenus dans l'exécution de la mise à jour.
- Simplification des sous-conditions à évaluer en utilisant des informations contenues dans la partie qualification de la mise à jour ou des informations sélectionnées dans le but de construire le journal de reprise (propre au système transactionnel).

En ce qui concerne l'exécution du squelette d'évaluation deux autres techniques d'optimisation ont été prises en compte :

- Exploitation de sous-conditions communes à plusieurs préconditions. L'intérêt est de ne pas répéter les mêmes accès à la base de données.
- Evaluation des sous-conditions contenues dans le squelette d'évaluation d'une précondition par ordre croissant de complexité. Ceci évite d'évaluer des conditions complexes lorsque des conditions plus simples ne sont pas satisfaites.

2.2.2. Langage de Définition d'un Schéma de Procédure

Ce langage offre un ensemble de structures de contrôle permettant de spécifier la séquence, le parallélisme, la synchronisation et le choix d'un ensemble d'activités. De plus, il permet de spécifier les liens entre l'entité focale de la procédure et les entités focales des activités qui lui sont associées. Ceci est fait par la spécification du chemin dans le schéma conceptuel qui lie les deux entités.

La définition d'un schéma est traitée par le système de façon à produire sa représentation sous forme d'un arbre qui sera utilisée comme référence pour contrôler l'exécution des procédures.

2.2.3. Primitives de Gestion des Activités et Procédures

Ces primitives permettent de faire la gestion des états d'une classe d'activité ou de procédures (charger et décharger une classe) et de leurs instances (lancer, activer, désactiver, terminer normalement ou anormalement l'exécution d'une instance). La mise en œuvre des primitives

relatives aux instances définit le **modèle d'exécution** et son interaction avec les modèles transactionnel et de verrouillage de données. Deux aspects sont à considérer dans la définition d'un modèle d'exécution : l'habilitation et l'exécution des activités et des procédures.

L'**habilitation des activités et des procédures** est synchronisée avec la confirmation de la transaction de déclenchement (c.à.d., la transaction qui provoque la validation de la précondition) pour éviter le lancement d'activités ou procédures dans des circonstances incohérentes (par exemple en cas d'annulation des transactions).

Considérons maintenant l'**exécution des activités**. Nous partons du principe que les activités délimitent de façon naturelle des unités cohésives de travail. Ainsi, une des premisses qui est à la base de l'interaction entre le modèle d'exécution et le système transactionnel est que les transactions ne doivent pas dépasser les bornes d'une activité.

En ce qui concerne les interactions avec le système de verrouillage, tout d'abord les informations du contexte d'exécution sont verrouillées au lancement d'une activité. Cette activité peut hériter des verrous de la session où elle s'exécute et elle peut les propager à la session à la fin de son exécution.

Au niveau de l'**exécution des procédures** nous aurions pu imaginer un modèle d'exécution supporté par des transactions de longue durée et des verrous persistants ; ceci dans le but d'assurer le contrôle persistant de l'exécution. Néanmoins les caractéristiques particulières des traitements pris en compte nous ont mené à des solutions plus simples.

D'une part au niveau du système transactionnel nous sommes guidés par deux principes :

- Il n'est pas souhaitable qu'une procédure soit reprise (défaite) entièrement lorsqu'une défaillance survient pendant son exécution.
- L'annulation d'une activité ne peut en aucun cas entraîner l'annulation d'autres activités.

Ces deux exigences impliquent que les activités soient considérées comme l'unité maximale de reprise.

D'autre part, au niveau du système de verrouillage le problème principal concerne le verrouillage d'informations pendant de longues périodes de temps (l'exécution d'une procédure peut prendre des mois !). En analysant les caractéristiques des traitements pris en compte nous avons conclu que soit l'information a nécessité d'être verrouillée pendant de courtes périodes de temps, soit l'information est contrôlée pour une longue période de temps par un utilisateur particulier. Pour modéliser ceci nous n'avons pas besoin de verrous persistants . Ainsi, nous proposons un système utilisant des verrous volatiles : la première activité d'une procédure est démarrée automatiquement et son objectif est de donner des droits d'accès à un utilisateur particulier et de libérer les verrous acquis lors du lancement de la procédure.

En conclusion nous avons abouti à un modèle d'exécution simple (supporté par un système transactionnel simple) et puissant.

3. Niveau Fonctionnel

Vu que les activités et les procédures sont entre autre manipulées en utilisant les LDD et LMD standards offerts par le système, l'implémentation des outils et techniques permettant leur gestion est accomplie en partie par les éléments du SGBD qui sont chargés de la mise en œuvre de l'aspect statique. Néanmoins, les nouveaux outils décrits dans 2.2 nécessitent le développement de nouveaux modules au niveau fonctionnel.

3.1 Compilation d'une Précondition

La compilation d'une précondition est réalisée lors de la définition d'une classe d'activités. Les analyseurs syntaxique et sémantique standards doivent prendre en compte les particularités des énoncés de préconditions. Par ailleurs, le but de sa traduction et interprétation est de générer et de stocker l'information nécessaire à évaluer la précondition avec un minimum de travail. Ainsi, ces informations doivent permettre l'application aisée des règles de correspondance pour qu'on puisser réaliser de façon efficace la sélection des préconditions à évaluer lors des mises à jour. De même le squelette d'évaluation doit être contenu dans ces informations : il permettra d'accomplir l'évaluation optimisée des préconditions sélectionnées.

3.2. Evaluation d'une Précondition

L'évaluation des préconditions se fait lors des mises à jour. Comme nous l'avons dit, le principe le plus important d'optimisation de l'évaluation des préconditions est qu'elle est étroitement liée à l'exécution de la mise à jour. Le squelette d'évaluation généré lors de la compilation d'une précondition reflète l'application de ce principe. Au niveau implémentation il implique une évaluation en deux étapes :

- Sélection des préconditions à évaluer intégrée à la compilation de la mise à jour.
- Evaluation des préconditions sélectionnées intégrée à l'exécution de la mise à jour.

Par ailleurs, nous pouvons considérer une troisième phase dans l'évaluation des préconditions, consistant à créer des activités. Ainsi, lors de la confirmation de toute transaction le Gestionnaire d'Exécution (voir ci-dessous) est averti des activités et procédures à créer.

3.3. Compilation des Primitives de Gestion des Activités et Procédures

Le langage de spécification de ces primitives constitue un sur-ensemble du FML. La compilation de ces énoncés peut être réalisée, pour une bonne partie, par les modules de compilation standards. Néanmoins, leur traduction et interprétation a un but très particulier. Brièvement, une primitive est traduite en plusieurs types d'opération :

- Sélection des activités ou procédures à manipuler.
- Requêtes à envoyer aux gestionnaires de transaction et de verrouillage de données dans le but d'implémenter quelques unes des exigences du modèle d'exécution sous-jacent.
- Requêtes à envoyer au gestionnaire d'exécution (voir ci-dessous) ayant comme objectif l'exécution proprement dite de la primitive (p.ex., son lancement ou sa terminaison).

3.4. Gestionnaire d'Exécution

La fonction principale du Gestionnaire d'Exécution est de contrôler l'habilitation, le lancement et la terminaison des activités et procédures. Au niveau procédure, une tâche très importante doit être accomplie dans le but d'implémenter ces primitives : il s'agit de l'évaluation des contraintes de précédence des activités à l'intérieur d'une procédure pour pouvoir déterminer les activités à lancer.

Pour implémenter le lancement d'une activité nous avons réalisé un modèle client-serveur entre le Gestionnaire d'Exécution et le site où l'on souhaite exécuter l'activité.

Pour l'évaluation des contraintes de précédence le Gestionnaire d'Exécution maintient une structure de données en mémoire centrale qui est instanciée à partir d'informations contenues dans la base de données. Cette structure permet de contrôler l'exécution d'une instance de procédure. Elle concerne seulement les procédures dont au moins une activité est en train de s'exécuter. Les autres procédures sont considérées en repos (et elles peuvent rester en repos pour une longue période de temps) : en conséquence, le Gestionnaire d'Exécution ne maintient sur elles aucune information en mémoire centrale.

Nous terminons ici la description de la manière dont les niveaux nécessaires à la modélisation et gestion des aspects dynamiques d'un serveur d'information pour la bureautique ont été traités. Pour conclure, nous sommes arrivés à la construction d'un prototype qui offre les services de base permettant le développement d'applications de gestion d'activités bureautiques pour aboutir à un système complet.

Perspectives

Nous présentons ici quelques propositions d'améliorations et extensions à faire à notre travail. Quatre points sont abordés : la notion d'entité focale, la notion de contexte d'activation, la méthode d'évaluation et finalement la généralité du modèle et du mécanisme d'exécution proposé.

La notion d'entité focale

Plusieurs arguments permettent de valoriser l'existence d'une entité focale par activité : modélisation de l'évolution du comportement d'une classe d'entités, liaison entre les aspects statiques et dynamiques des applications. Nous avons vu aussi l'intêret de l'entité focale dans la création (instanciation) des activités.

Au niveau d'une procédure l'entité focale est utilisée pour modéliser le flux des données. En effet, l'entité focale d'une procédure est liée dans le schéma conceptuel aux entités focales des activités qui la composent. Cette façon de modéliser le flux de données résulte d'études réalisées dans le domaine des méthodologies de conception.

L'utilisation du concept d'entité focale nécessite une évaluation dans le cadre d'applications réelles pour mettre en évidence son apport au niveau modélisation. Nous pensons que cette évaluation est d'un grand intérêt car elle peut valider les travaux menés dans les domaines des méthodologies de conception.

La notion de contexte d'activation

La notion de contexte d'activation comme photographie (cf. 4.3 et 8.2) d'une partie de la base de données (partie concernée par la précondition) au moment de la validation de la précondition semble très intéressant. Entre autre, elle permet d'accéder à partir de l'intérieur d'une activité, à l'état de la base au moment de la validation de la précondition et à son état actuel, de façon homogène. De plus,

cette façon de voir le contexte d'activation s'adapte très bien au millieu bureautique où les actions à entreprendre peuvent être exécutées longtemps après que l'évènement de déclenchement ait eu lieu.

Un travail de fond dans ce domaine (p. ex. formalisation de la définition de contexte d'activation, déduction de son schéma conceptuel à partir d'une précondition, traitement des expressions contenant des références au contexte d'activation) mérite d'être réalisé.

La méthode d'évaluation des préconditions

Dans le domaine de l'évaluation efficace de prédicats sur les informations d'une base de données, plusieurs travaux théoriques concernant l'évaluation de contraintes d'intégrité sur des S.G.B.D. relationnels ont été réalisés. Nous avons déjà vu que les principes de base des méthodes d'évaluation proposées dans ce cadre ne s'appliquent pas à l'évaluation des préconditions. La raison principale est que la notion de contrainte d'intégrité est extrêmement liée à celle d'état valide d'une base de données, alors que l'évaluation à vrai ou à faux d'une précondition n'a pas de rapport explicite avec la validité d'une base de données.

Néanmoins, nous avons adapté certains principes utilisés dans l'évaluation des contraintes d'intégrité (cf. 7.6). D'autres principes, comme celui consistant à garder dans la base de données des valeurs redondantes (p.ex., la matérialisation de fonctions d'agrégation ou de certains ensembles d'informations) aurait pu être utilisés par certaines classes de préconditions. Un travail dans ce domaine (p.ex. caractérisation des classes de préconditions qui pourrait profiter de ce type d'optimisation, adaptation de la méthode) permettra d'enrichir la méthode d'évaluation.

Généralité du modèle et du mécanisme d'exécution

Nous avons travaillé sur un modèle particulier (modèle fonctionnel) et un système particulier (entre autres nous utilisons le SGBD ORACLE pour assurer la persistance des données et l'accès basique à celles-ci). Plusieurs travaux développés dans cette thèse sont généraux et peuvent s'appliquer à d'autres modèles. Un exemple est le travail mené pour établir les règles de correspondance : bien que l'écriture de ces règles prennent en compte notre langage de manipulation, les principes appliqués sont généraux à tous les modèles sémantiques.

Par ailleurs, d'autres travaux, bien que facilement généralisables, s'appuient sur des caractéristiques propres à notre système. Il serait donc intéressant de spécifier de façon précise et générale, d'une part les fonctionnalités d'un module de gestion de préconditions et d'autre part ses besoins au niveau de la mise en œuvre (par exemple, les besoins en termes d'accès aux données comme les accès au journal de reprise). Ainsi, la gestion de préconditions et en particulier la méthode d'évaluation développée ici pourrait à moindre coût être intégrée par les constructeurs des S.G.B.D.

De façon plus générale, vu l'importance que les bases de données actives prennent en ce moment, il est intéressant de mener une étude sur l'application de notre travail à des systèmes commerciaux futurs et à des systèmes représentatifs des nouvelles approches base de données comme le modèle orienté objet.

CHAPITRE 10

10. BIBLIOGRAPHIE

Ce chapitre fournit la compilation des références bibliographiques contenues dans ce document.
La référence à un article est faite en utilisant les trois premières lettres du co-auteur ayant le nom le plus petit dans l'ordre alphabétique et deux chiffres indiquant l'année de publication. L'ensemble des références est organisé par ordre alphabétique.

Avant l'énumération des articles, nous présentons les notations utilisées pour quelques uns des ouvrages les plus cités :

Conferences

- Journées Bases de Données Avancées : **BD3**
- International Conference on Very Large Data Bases : **VLDB**
- International Conference on Extending Database Technologie : **EDBT**
- International Conference on the Management of Data : **ACM SIGMOD**
- International Conference on the Entity-Relationship Approach : **ERA**
- International Computer Software and Applications Conference : **IEEE COMPSAC**

Workshops

- International Workshop on Object-Oriented Database Systems : **IWOODBS**
- International Workshop on Database Programming Languages : **IWDPL**
- International Workshop on Persistent Objects Systems : **IWPOS**

Revues

- ACM Transactions on Database Systems : **ACM TODS**
- ACM Transactions on Information Systems : **ACM TOIS**
- ACM SIGMOG on the Management of Data : **SIGMOD RECORD**
- ACM Communications : **CACM**

Références

[Adi81] Michel Adiba
Derived Relations : A Unified Mechanism for Views, Snapshots and Distributed Data
7VLDB81

[Adi86] Michel Adiba, Ngoc Bui Quang
Historical MultiMédia Databases
VLDB86, Kyoto, Japon

[Adi87] Michel Adiba, Ngoc Bui Quang
Aspects Temporels, Historiques et Dynamiques des Bases de Données
TSI, Vol. 6, N° 5 - Dunod Afcet 87

[Alb85] A. Albano, L. Cardelli, R. Orsini
Galileo : a Strongly Typed Conceptual Language
ACM TODS Pg. 230-260, Vol.10, N° 2, Juin 85

[ANS89] ANSI-ISO Working Group
Data Base Languages SQL2 and SQL3
Working draft, ANSI X3H2-89-110, ISO DBL CAN-3, Fév. 89

[Ant86] Francisca Antunes
Gestion des Types Abstraits dans le SGBD TIGRE
Rapport interne LGI-IMAG, Grenoble, 86

[Ant87a] Francisca Antunes
Mécanismes de Spécification de Déclencheurs
Rapport de Travail - Première version, LGI - IMAG, Avril 87

[Ant87b] Francisca Antunes
Méthodes d'Evaluation des Contraintes d'Intégrité dans les SGBD Relationnels
DOEOIS : ESPRIT Project 231, BUL-DM-04.01, Dec. 87
Aussi rapport BULL DRPA/OSI/90009

[Ant89a] Francisca Antunes, Mauricio Lopez, Esperanza Pedraza
Avaliação das Pré-condições das Atividades de Escritório
4° Simpósio Brasileiro de Banco de Dados, Pg. 273-284, Campinas - SP, Brasil, 5-7 Avril 89

[Ant89b] Francisca Antunes, Mauricio Lopez
A Mechanism for Efficient Evaluation of Activity Preconditions in a Database Environment
5BD3, Genève, Suisse, 26-28 Sept. 89
Aussi rapport BULL DE/DRPA/OSI/90003

[Ant90] Francisca Antunes, Sean Baker, Brian Caulfield, Mauricio Lopez, Marc Sheppard
A Pragmatic Approach for Integrating Data Management and Tasks Management : Modelling and Implementation Issues
2EDBT90, Pg. 421-436, Venice, Italie, 26-30 Mars 90
Aussi rapport BULL DDRPA/OSI/90014

[Atk89] M. P. Atkinson, A. Brown, R. Carry, R. Connor, A. Dearle, R. Morrison
The Napier Type System
Proc. 2nd Int. Workshop on Persistent Objects Systems : Their Design, Implementation and Use, Pg. 253-269, 89

[Bal88] N. Ballou, J. Banerjee, H. T. Chou, J. Garza, W. Kim, D. Woelk
Integrating an Object-Oriented Programming System with a Database System
OOPSLA 1988, San Diego, CA

[Ban88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, F. Velez
The Design and Implementation of O2 an Object Oriented Database System
Rapport Technique ALTAIR, 20-28, Avril 88

[Bar89] F. Barbic, M.G. Fugini, R. Miocchi, B. Pernici, J. R. Rames, C. Rolland
C-TODS : an Automatic Tool for Office System Conceptual Design
ACM TOIS, Vol. 7, N° 4, pg. 378-419, Oct. 89

[Bee87] D. Beech, H. Cate, R. Chow, D. Fishman, al.
IRIS : an Object-Oriented Database Management System
ACM TOIS, Vol. 6, N°. 5, 87

[Ber80] Philip A. Bernstein, Barbara T. Blaustein, Edmund M. Clarke
Fast Maintenance of Semantic Integrity Assertions using Redundant Aggregate Data
6VLDB80, Pg. 126-136, Montreal, Canada, Pg. 126-136, Oct. 80

[Ber81] Philip A. Bernstein, Barbara T. Blaustein, Edmund M. Clarke
A Simplification Algorithm for Integrity Assertions and Concrete Views
IEEE COMPSAC 81, 5th International Computer Software and Applications Conference, Proceedings, Chicago, Nov. 81

[Ber82] Philip A. Bernstein, Barbara T. Baunstein
Fast Methods for Testing Quantified Relational Calculus Assertions
ACM SIGMOD, Pg. 39-50, Orlando, Florida, 2-4 Juin 82

[Bla81] Barbara T. Blaunstein
Enforcing Database Assertions: Techniques and Applications
Ph. D. Thesis, Harvard University, 81
Aussi TR-21-81, Center for Research in Computing Tecnhology, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts, 81

[Bla88] Barbara Blaunstein, A. Buchmann, M.J. Sarin, U. Chakravarthy, M. Hsu, R. Ledin, M. Livny, D. McCarthy, A. Rosenthal, S. Sarin
The HIPAC Project : Combining Active Databases and Timing Constraints
SIGMOD RECORD, Vol. 17, N° 1, Pg. 51-69, Mars 88

[Bla86] J. Blakley, P. Larson, F. Tompa
Efficiently Updating Materialized Views
ACM SIGMOD, Pg. 61-71, 86

[Bla89] Jose Blakeley, Barbara Blaunstein, Sharma Chakravarthy, Arnon Rosenthal
Situation Monitoring for Active Databases
15VLDB89, Pg. 455-464, Amsterdam, The Netherlands, 22-25 Aout 89

[Bog83] J. Bogo, H. Richy, I. Vatton
Proposition de Modèle pour la Manipulation des Documents
Rapport de Recherche TIGRE No. 3, CII-HB, IMAG, Grenoble, Nov 82

[Bro78] Michael Lawrence Brodie
Specifications and Verifications of Data Base Semantic Integrity
Ph. D. Thesis, University of Toronto, April 78
Aussi Tecnical Report CSRG-91 (Thesis), Apr 78

[Bro81] Michael Lawrence Brodie
On Modelling Behavioural Semantics of Database
7VLDB81, Pg. 32-41, Cannes, France, 9-11 Sept. 81

[Bro89] A.L. Brown, R. C. H. Connor, A. Dearle, R. Morrison
Napier88 Reference Manual
Persistent Programming Research Report PPRR-77-89, University of St. Andrews,89

[Buc88] Alejandro P. Buchmann, Umeshwar Dayal, dennis R. McCarthy
Rules are Objects too : A Knowledge Model for an Active, Object-Oriented Database System
Proc. 2nd IWOODS, pg. 129-143, West Germany, Sept. 88

[Bui86] Ngoc Bui Quang
Aspects Dynamiques et Gestion du Temps dans les Systèmes de Bases de Données Généralisées
Thèse de docteur de l'INPG, Grenoble, 20 Nov. 86

[Bun79] O. Peter Buneman, Eric K. Clemons
Efficiently Monitoring Relational Databases
ACM TODS, Vol. 4, No.3, Pg 368-382, Sep 79

[Bur87] Reginald Bursens, Jacques Guyot
Temps + Dynamique + Référentiel = Historique
3BD3, Pg. 187-203, Port-Camargue, France, 20 - 22 Mai 87

[Cha89] Sharma Chakravathy
Rule Management and Evaluation : An Active DBMS Perspective
SIGMOD RECORD, Vol. 18, N° 3, Pg. 20-28, Sept 89

[Cha90] Sharma Chakravathy, Susan Nesson
Making an Object-Oriented DBMS Active : Design, Implementation, and Evaluation of a Prototype
2EDBT90, Pg. 393-406, Venice, Italie, 26-30 Mars 90

[Chu84] Kyungwha Lawrence Chung
An Extended Taxis Compiler
M. Sc. Thesis, Dept. of Computer Science, University of Toronto, Jan 84
Aussi CSRG Technical Note 37, 1984

[Chu87] K. Lawrence Chung, David Lauzon, Alex Borgida Jonh Mylopoulos, Brian A. Nixon, Martin Stanley
Design of a compiler for a Semantic Data Model
Technical Note CSRI-44, Mai 87

[Chu88] Kyungwha Lawrence Chung, Jonh Mylopoulos, Brian A. Nixon, Daniel Rios-Zertuche
Process Management and Assertion Enforcement for a Semantic Data Model
1EDBT88, Venice, Italy, 14-18 Mars 88
Aussi Technical Report CSRI-204, Computer Systems Research Institute, University of Toronto, Canada, Jan. 88

[COD73] CODASYL Data Description Language Committee
CODASYL Data Description Language
Journal of Development, NBS Handbook 113, Juin 73

[Col87] Christine Collet
Formulaires Complexes pour Bases de Données Multimédia
Thèse de doctorat de l'Université Joseph Fourrier, Grenoble, Nov. 87

[Dar88] M. Darnovsky, J. Bowman
TRANSACT-SQL User's Guide
Sybase, Inc. 87

[Day89] Umeshwar Dayal, Dennis R. McCarthy
The Architecture of an Active Data Base Management System
ACM SIGMOD, Portland, Oregon, Mai 31-2 Juin 89, SIGMOD-RECORD Vol. 18, N° 2, Pg 215-224, Jun 89

[Day90] Umeshwar Dayal
Active Databases
Ceci n'est pas une référence à un papier mais au "tutorial" qui a eu lieu dans le 2EDBT90, Venice, Italie, 26 Mars

[Dit86] Klaus R. Dittrich, Angelica M. Kotz, Jutta A. Mulle
An Event/Tigger Mechanism to Enforce Complex Consistency Constraints in Design Databases
ACM SIGMOD RECORD, Vol. 15, No. 3, Pg 22-36, Sept 86

[Dit88] Klaus R. Dittrich, Angelika M.Kotz, Jutta a. Mulle
Supporting Semantic Rules by a Generalised Event/Trigger Mechanism
1EDBT88, Pg. 76-91, Venice, Italie, 14-18 Mars, 88

[DOE86a] Groupe DOEOIS
Review of Data Models and Data Management Systems
DOEOIS : ESPRIT Project 231, 1B2, Jan 86

[DOE86b] Groupe DOEOIS
Office Information Server Data Model Description
DOEOIS : ESPRIT Project 231, 2B1, Sept. 86

[DOE86c] Groupe DOEOIS
The Fact Model : A Semantic Data Model for Complex Databases
DOEOIS : ESPRIT Project 231, OIS-DM-01.0, Sept. 86
Aussi ETW Conference, Sept. 86

[DOE87a] Groupe DOEOIS
Office Procedure Support Manual
DOEOIS : ESPRIT Project 231, OIS-OP-01.0, Sept 87

[DOE87b] Groupe DOEOIS
Office Procedures : A Formalism and Support Environment
DOEOIS : ESPRIT Project 231, OIS-OP-02.01
Aussi ETW Conference Proceedings, Sept. 87

[DOE87c] Groupe DOEOIS
Client OIS Distributed Interaction Model
DOEOIS : ESPRIT Project 231, OIS3CL, Sept 87

[Don89] Catherine Donnou, Jean-Claude GUYOT
Réalisation d'un Gestionnaire d'Exécution d'Activités défini dans le cadre d'un Serveur d'Information pour la Bureautique (SIB)
Rapport de stage de la MST Expert en Systèmes Techniques, Université Joseph Fourrier, Aout 89

[ECM85] European Computer Manufacture Association
Standard ECMA-101 Office Document Architecture
Sept 85

[[Ell80] Clarence A. Ellis, Gary J. Nutt
Office Information Systems and Computer Science
ACM Computing Surveys, Vol. 12, No. 1, Mars 80

[Esw76] Kapali P. Eswaran
Specifications, Implementations and Interactions of a Trigger Subsystem in a Relational Database System
IBM Research Report RJ1820, San Jose CA, Aout 76

[Feg89] L. Fegaras, T. Sheard, D. Stemple
The ADABTPL Type System
2IW DPL, Pg. 243-254, Salishan, Oregon, 89

[Fin89a] Sheldon J. Finkelstein, Jennifer Widom
A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems
SIGMOG RECORD, Vol.18, N° 3, Pg. 36 - 45, Sept. 89
Aussi IBM Research Report, IBM Almaden Research Center, Juin 89

[Gal84] H. Gallaire ,J. Minker, J. Nicolas et al.
Advances in Data Bases
H. Gallaire ,J. Minker, J. Nicolas (editors), Plenum Press, 1984

[Gib84] Simon J. Gibbs
An Object-Oriented Office Data Model
Technical Report CSGR-154, Computer Systems Research Group, University of Toronto, Janv. 84

[Goo75] J. B. Goodenough
Exception Handling : Issues and a Proposed Notation
CACM, Déc. 75

[Haa86] B. Lindsay, L. Haas, C. Mohan
A Snapshot Differential Refresh Algorithm
ACM SIGMOD, Pg. 53-60, 86

[Ham75] Michael M. Hammer, Dennis J. McLeod
Semantic Integrity in a Relational Data Base System
VLDB75, Pg. 25-47, Framigham, U.S.A., 22-24 Sept 75

[Ham76] Michael M. Hammer, Dennis J. McLeod
A Framework for Data Base Semantic Integrity
Proceedings of the International Conference on Software Engineering, Pg 498-504, San Francisco, CA, 13-15 Oct 76

[Ham78] Michael M. Hammer, S. K. Sarin
Efficient Monitoring of Database Assertions
ACM SIGMOD, Pg. 159, Austin, Texas, Mai 31-2 juin 78

[Han87] E. Hanson
A Performance Analysis of View Materialization Strategies
ACM SIGMOD, Pg. 440-453, Août 87

[Hea89] Marti Hearst, Spyros Potamianos, Michael Stonebraker
A Commentary on the POSTGRES Rules System
SICMOD RECORD, Vol. 18, N° 3, Pg. 5-11, Sept 89

[Hen84] Lawrencea J. Henschen, William W. McCune, Shamin A. Naqvi
Compiling Constraint-Checking Programs from First-Order Formulas
In [Gal84], Pg.145-169

[Hew72] C. Hewit
Description and Theorical Analysis (using schemata) of PLANNER : A Language for Proving Theorems and Mnipulating Models in a Robot
AI-TR-258, MIT AI Lab., 72

[Hsu85] Arding Hsu and Tomasz Imielinski
Integrity Checking for Multiple Updates (Preliminary Version)
ACM SIGMOD, Pg 152-168, Austin, TX, Mai 85

[Hsu90] Mechun Hsu, Yuli Zhou
A Theory for Rule Triggering Systems
2EDBT90, Pg. 407-421,Venice, Italie, 26-30 Mars 90

[Hud86] S. Hudson, R. King
CACTIS : A Database System for Specifying Functinally-Defined Data
1IW OODBS, Pg. 26-37, Sept. 86

[Hus89] Ahmed Hussien, Jackson Martins Ramos, Gilles Vinçon
Interface Graphique pour l'Insertion d'Activités Bureautiques
Rapport de projet de l'année Spéciale de l'ENSIMAG, INPG, Juin 89

[Koe81] Shaye Koening, Robert Paige
A Transformal Framework for the Automatic Control of Virtual Data
7VLDB81, Pg. 306-318, Cannes, France,Sept. 81

[Lei79] S. Leifert, C. Richard, Colette Roland
Tools for Information System Dynamics Management
5VLDB79, Pg. 251-261, Rio de JAneiro, Brasil 3-5 Oct. 79

[Len89] Christian Lenne
EMIR : Un Outil de Gestion de Structures Arborescentes Attribuées
Rapport interne BULL, Déc. 89

[Lig87] Jean-Yves Lignat, Philipe Nobecourt, Colette Rolland
Gestion de la Dynamique des Données dans un SGBD Relationnel
3BD3, Pg. 205-227, Port-Camargue, France, 20-22, Mai 87

[Lop87] Mauricio Lopez
Data Definition and Data Manipulation Services
DOEOIS : ESPRIT project 231, OIS-DM-01.01, Avr.87

[Mai86] D. Maier, A. Ottis, A. Purdy, J. Stein
Developpement of an Object-Oriented DBMS
OOPSLA 86, Sept. 86

[Mai88] C. de Mandreville, Eric Simon
Deciding Whether a Production Rule is Relational Computable
Proc. Int. Conf. on Database Theory, Bruges, Belgique, Sept. 88

[Mat89] F. Mathes, J. Schmidt
The Type System DBPL
2IW DPL, Pg. 252-260, Salishan, Oregon, 89

[Mic88] Alain Michard
An Office Representation Framework for Planning and Monitoring Office Activities
Rapport de Recherche INRIA, n0. 891, Aout 88

[Mor83] Matthew Morgenstern
Active Databases as a Paradigm for Enhanced Computing Environments
9VLDB83, Pg. 34-42, Florence, Italie, Oct31-2Nov 83

[Nic82] Jean Marie Nicolas
Logic for Improving Integrity Checking in Relational Data Bases
Acta Informatica Vol. 18, Fasc. 3, Pg 227-253, Dec 82

[Nie89] O. M. Nierstrasz, D. C. Tsichritzis
Integrated Office Systems
Object-Oriented Concepts, Databases, and Applications, Edité par Won Kim and Frederic Lochovsky, ACM Press - Frontier series, 89

[Par88] J. Park, A. Seger
Using Common Subexpressions to Optimize Multiple Queries
Proc. 4th Int. Conf. on Data Engineering, Pg. 311-318, Los Angeles 88

[Ped86] Esperanza Pedraza
Intégrité et Cohérence d'une Base de Données dans un Contexte Bureautique
Publication interne - Equipe base de données, L.G.I., IMAG, Grenoble, 86

[Ped87] Esperanza Pedraza
Spécification du Mécanisme de Reprise et de Synchronisation pour un OIS
DOEOIS : ESPRIT Project 231, Rapport Interne
LGI-IMAG et Centre de Recherche BULL, Grenoble, Juil. 87

[Ped88] Esperanza Pedraza
SGBD Sémantiques pour un Environement Bureautique : Intégrité et Gestion de Transactions
Thèse de Docteur Ingénieur, Université Joseph Fourrier - Grenoble I, Nov. 88

[PPR87] Persistent Programming Research Group
The PS-Algol Reference Manual
4ème Ed., Technical Report PPRR-12-87, Universities of Glasgow and St. Andrews, 87

[Rab85] Fausto Rabitti
A Data Model for Multimedia Documents
Article 10 in D. Tschritzis (ed.), Office Automation, Spring Verlag 85

[Ric84] G. Richter
Netzmodelle für die Bürokommunnikation
Teil 2, Informatik Spektrum 7, Pg 28-40, 84

[Ris89] Tore Rish
Monitoring Database Objects
15VLDB89, Pg. 445-453, Amsterdam, The Netherlands, 22-25 Août 89

[Rou82] N. Roussopoulos
View Indexing in Relational Databases
ACM TODS, Vol. 7 N° 2, Pg. 258-290, Juin 82

[Row86] Lawrence Rowe, Michael Stonebraker
The Design of the POSTGRES Rules System
The POSTGRES papers, Pg 48-66, Memoradum No. UCB/ERL M85/86, Berkeley, 5 Nov. 86

[Sar77] S. K. Sarin
Automatic Synthesis of Efficient Procedures for Database Integrity Checking
M. Sc. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sep 7

[Shi81] David W. Shipman
The Functional Data Model and the Data Language DAPLEX
ACM TODS, Vol. 6, N° 1, Pg. 140-173, Mars 81

[Sir82] M. A. Sirbu et al.
OAM : An Office Analysis Methodology
Office Automation Conference Digest, AFIPS, 82

[Sto75] Michael Stonebraker
Implementation of Integrity Constraints and Views by Query Modification
ACM SIGMOD, Pg. 65-78, San Jose CA, 14-16 Mai 75

[Sto86] Michael Stonebraker
Triggers and Inference in Database Systems
In [Bro86a], Pg. 297-314
Aussi Rapport interne, LGI-IMAG, Grenoble, 85

[Tsi85] D. Tsichritzis
Office Automation Tools
1BD3, INRIA, St.Pierre de la Chartreuse, Mars 85

[Vel84] Fernando Velez
Un Modèle et un Langage pour les Bases de Données Généralisées
Thèse de docteur ingénieur, Projet TIGRE, INPG, grenoble, Sept 84

[Zlo77] Moshé M. Zloof
The System for Business Automation (SBA) : Programming Language
CACM, Vol. 20, N° 6, Pg. 385-396, Juin 77

[Zlo80] Moshé M. Zloof
A Language for Office and Business Automation
Research Report RC 8091, IBM, Jan. 80

[Zlo82] Moshé M. Zloof
Office-By-Example : A Business Language that Unifies Data, Word Processing and Electronic Mail.
IBM Systems Journal, Vol. 21, No. 3, Pg. 272-302, 82

ANNEXE I

I. CLASSIFICATION DES MECANISMES DE DECLENCHEURS

Le paragraphe 2.1 du chapitre sur les bases de données actives fait une analyse des systèmes de déclencheurs. Il présente les critères de jugement et un résumé de quelques uns de ces systèmes considérés comme représentatifs. Le point de départ de ce travail a été l'étude systématique des différents mécanismes de déclencheurs. Cette étude est présentée dans cette annexe sous forme de tableaux. Les tableaux sont divisés en trois groupes :

1. **Données générales** sur les systèmes qui vont être présentés.
2. **Modélisation des déclencheurs** par les différents systèmes.
3. **Sémantique de l'exécution** sous-jacente à chaque système de déclencheurs.

A chacun de ces groupes correspondent deux tableaux : l'un décrit succintement les critères utilisés ; l'autre présente les réponses données par chaque système aux critères de jugement. Dans ce dernier type de tableau, l'occurence d'un tiret indique que le critère n'a pas de sens. Certaines réponses nécessitent un commentaire : dans ce cas, une lettre minuscule est placée dans la case. Cette lettre fait référence à un des commentaires ci-dessous.

Bien sûr, il est intéressant de se référer au paragraphe 2.1.

Commentaires référencés dans les tableaux

a. Le système POSTGRES a été proposé en 86. Dans cette première version un mécanisme de déclencheurs a été spécifié - POSTGRES Rule System (PRS). Un prototype de ce mécanisme a été élaboré et son évaluation a abouti à un ensemble d'extensions, puis à la définition de POSTGRES Rule System II (PRS II).
   Les deux systèmes sont pris en compte ici : s'il y a une différence dans la façon dont ils répondent aux critères, les deux points de vue sont présentés, précédés par "1" pour PRS et par "2" pour PRS II.

b. Le système proposé par Hammer et McLeod a comme but la spécification d'un sous-système d'intégrité sémantique. Il a été un des premiers efforts dans la spécification complète d'une contrainte. La forme d'une contrainte est très complète et proche de la forme de base d'un déclencheur. Néanmoins, la sémantique attachée à la structure de ces contraintes n'est pas toujours la même que pour les déclencheurs. En particulier, elle dépend du type d'action entreprise. Par exemple, si l'action à engager est une opération structurée ou un programme d'application, alors la condition est vérifiée une deuxième fois après l'exécution de l'action.

c. Dans le système R et Sybase la clause condition n'existe qu'implicitement. En fait, elle est donnée de façon algorithmique à l'intérieur de l'action : l'action est de la forme "si condition alors action1 sinon action2".

d. Une des critiques faites au premier système de déclencheurs de POSTGRES (PRS) est que les évènements sont implicites. De ce fait, un contrôle fin sur les mises à jour n'est pas possible et tous les cas d'intégrité référentielle[1] ne peuvent pas être implémentés.
En fait, la forme des déclencheurs (PRS) spécifie uniquement une action et la sémantique sous-jacente est que l'action doit logiquement être vue comme étant en exécution permanente.
Dans PRS II, l'utilisation explicite des trois clauses a été adoptée.

e. Gibbs a développé un système de déclencheurs pour la bureautique avec l'objectif d'assurer l'intégrité sémantique et de modéliser certaines tâches simples. Il a été un des premiers à proposer des déclencheurs avec une forme très simple spécifiée de façon déclarative (pas algorithmique). La structure de ces déclencheurs a une clause de plus que les déclencheurs de base : la clause erreur contenant un message à envoyer si la condition est fausse ; dans ce cas l'opération d'activation n'est pas exécutée (i. e. le refus de l'opération de déclenchement est implicite).

f. Dans [Ham75] les opérations de mise à jour et d'interrogation ne sont permises qu'à l'intérieur d'une opération structurée.

g. Extensions proposées.

h. Appel à des procédures POSTQUEL.

i. Les déclencheurs chez Gibbs ne peuvent spécifier qu'une condition simple de la forme "terme opération terme" où terme est soit une constante soit une composition de fonctions sur un objet p.ex., la propriété d'un objet.

j. La notion de passage d'information dans PRS est très implicite et dépend beaucoup de la façon dont l'activation du déclencheur est faite. En effet, un déclencheur est activé si on accède à un n-uplet qui est marqué par un verrou spécial : c'est cet n-uplet qui forme le contexte dans le quel le déclencheur va être exécuté.
Dans PRS II, comme les évènements sont explicites, il y a plus de sens à référencer les valeurs du n-uplet mis à jour.

k. Chez HIPAC un évènement est défini comme une fonction appliquée à plusieurs objets. Cette fonction peut être une mise à jour de base ou une opération beaucoup plus complexe. Le contexte passé à la condition est la valeur des arguments avant l'exécution de la fonction et après, ou la valeur des arguments d'entrée et ceux de sortie. HIPAC définit aussi le contexte

[1]L'intégrité référentielle est toute contrainte de la forme : "tout enfant doit avoir un parent".

d'exécution dans le cas des évènements composés : voir 2.1.2.

l. Trois déclencheurs par relation sont permis, i.e., un déclencheur par type d'opération de mise à jour. Cette restriction est faite dans le but d'empêcher que plusieurs déclencheurs soient activés par le même évènement.

m. POSTGRES fait aussi l'évaluation en différé d'un déclencheur mais ceci n'est défini qu'au niveau de la méthode qui implémente l'évaluation du déclencheur. Ainsi, si la méthode utilisée est "late evaluation" alors l'évènement du déclencheur ne se fait qu'au moment où quelqu'un interroge les n-uplets concernés par le déclencehur.
Dans ce cas l'exécution du déclencheur a lieu dans la transaction où l'énoncé d'interrogation est exécutée et non pas dans la transaction de déclenchement.

TABLEAUX DE CLASSEMENT DES MECANISMES DE DECLENCHEURS

1. DONNEES GENERALES

Description

S.G.B.D.R. / CRITERIA	*Systèmes étudiés*
Modèle de Données	*Type du modèle : rel / sém. / o.o. ...*
Année	*Date*
Etat Développement	*papier / prototype / produit*
Bibliographie	*Références bibliographiques*

1. DONNEES GENERALES

Classement

CRITERIA \ S.G.B.D.R.	[Ham75]	R	Sybase	SQL3	Starburst	POSTGRES	[Gib84]	HIPAC
Modèle de Données	rel.	rel.	rel.	rel.	rel. étendu	rel. étendu	o.o. office	o.o.
Année	75	76	88	89	89	1 ; 86 2 ; 89 a	84	88 - 90
Etat Développement	papier	papier	produit	produit	prototype extension	1 prototype 2 papier	prototype	prototype évaluation
Biblographie	[Ham75]	[Cha75] [Esw76b]	[Dar88]	[ANS89]	[Fin89a] [Fin89b]	[Row86] [Hea89]	[Gib84]	[Bla88] [Bla89] [Buc88] [Buc89] [Cha89] [Cha90] [Day88] [Day89] [Hsu88]

2. MODELISATION DES DECLENCHEURS

Description

2.1 Structure Générale

CRITERIA \ S.G.B.D.R.			*Systèmes étudiés*
Forme		Evènement	*Sont-ils explicites ?* *oui / non*
		Condition	
		Action	
		Autres	*Autres clauses ou comentaires*
Evènement	Opération	M.à.j. Basiques	*Sont-ils permis ?* *oui / non*
		Interrogation	
		Autres Op. BD	
		Op. Structurées	
		Ev. Explicites	
		Ev. Externes	
		Composition d'Ev.	*Constructeurs permis : disj. / conj. / ferm. (voir restrictions 3.1.1.1.1)*
	Objets	Classe	*Pour les modèles relationnels classe = relation* *Peut-on spécifier une classe ou plusieurs ?*
		Propriété	*Affinement au niveau d'une propriété permis ?*
		Global	*Déclencheurs globaux autorisés ?* *oui / non*

2.1. Structure Générale

CRITERIA \ S.G.B.D.R.			[Ham75]	R	Sybase	SQL3	Starburst	POSTGRES	[Gib84]	HIPAC
Forme		Evénement	oui	oui	oui	oui	oui	1 implicite [d] 2 oui	oui	oui
		Condition	oui	dans action [c]	dans action [c]	oui	oui	1 implicite [d] 2oui	oui	oui
		Action	oui	oui	oui	oui	oui	oui	oui	oui
		Autres	particulier [b]	—	—	—	—	1 pas de division [d]	message erreur [e]	—
Evènement	Opération	M.à.j. Basiques	oui	oui	oui	oui	oui	oui	oui	oui
		Interrogation	oui	non	non	non	non	oui	oui	oui
		Autres Op. BD	non	non	non	non	non	non	oui	oui
		Op. Structurées	oui	non	non	non	non	non	non	oui
		Ev. Explicites	non	non	non	non	non	non	non	non
		Ev. Externes	non	non	non	non	non	non	non	oui
		Composition d'Ev.	disj. implicite	non	disj.	non	non	disj. implicite	non	disj./conj. ferm.
	Objets	Classe	1 rel	1 rel	1 rel	1 rel	1 rel	1 rel	pls. classes	pls. classes
		Propriété	oui	non	oui	oui	non	oui	oui	oui
		Global	oui	non	non	non	non	non	oui	non

2.1. Structure Générale (suite)

Description

CRITERIA \ S.G.B.D.R.		*Systèmes étudiés*
Condition	Forme	*Sous quelle forme la condition est exprimée ?*
	Transition	*Conditions sur les transitions d'état permises ?*
Action	Op. M.à.j.	*Sont-elles permises ?* *oui / non*
	Interrogation	
	Autres Op. BD	
	Refus	
	Annuler Transaction	
	Abstr. Procédural	*Abstractions procédurales permises ?* *oui / non*
	Appel Pgrm. Application	*Appels à des programmes d'application permis ?* *oui / non*

2.2. Contexte

Evt. -> Cond.	*Quelles données sont passées à la condition ?*
Cond. -> Action	*Données passées à l'action en plus de celles qui viennent de l'évènement*
Passage	*Comment sont elles passées ?* *implicitement / explicitement : mots-clés , variables*

2.1. Structure Générale (suite)

Classement

CRITERIA \ S.G.B.D.R.		[Ham75]	R	Sybase	SQL3	Starburst	POSTGRES	[Gib84]	HIPAC
Condition	Forme	qualif.	qualif.	select	qualif.	qualif.	qualif.	qualifi. simple [i]	pls. sélect.
	Transition	old new	old new	deleted inserted	old new	deleted inserted	2 old new	variables	variables
Action	Op. M.à.j.	oui [f]	oui	oui	oui	oui	oui	oui	oui
	Interrogation	oui [f]	non	non	non	oui [g]	oui	oui	oui
	Autres Op. BD	implicite	non	non	non	non	non	oui	oui
	Refus	?	non	non	non	non	oui	implicite	non
	Annuler Transaction	non	?	oui	non	non [g]	non	oui	oui
	Abstr. Procédural	oui	non	?	non	non	non	oui	oui
	Appel Pgm. Application	oui	non	non	non	non	oui [h]	non	oui

2.2. Contexte

Evt. -> Cond.	n-uplet m.à.j.	n-uplet m.à.j.	n-uplet m.à.j.	n-uplet m.à.j.	n-uplet m.à.j.	2 n-uplet m.à.j. [j]	objets évènt.	objets évènt. [k]
Cond. -> Action	non	non	non	non	non	non	non	résultat condition
Passage	mots clés	mots clés	mots clés	mots clés	mots clés	2 mots clès	variables	variables

Description

2.3. Orientation

CRITERIA \ S.G.B.D.R.	*Systèmes étudiés*
Orientation	*Ensembliste ou instance par instance ?* *oui / non*

2.4. Modèle

Définition		*Concept de déclencheur indépendant du modèle ou approche couplage ?* *indépendant / couplage*
Manip.	Restrictions Création	*Restrictions au niveau du nombre de déclencheurs par classe ?*
	Charg. / Décharg.	*Chargement et déchargement sont des opérations proposées ?* *de façon sélective ou déclencheur par déclencheur ?*

2.3. Orientation

Classement

CRITERIA \ S.G.B.D.R.	[Ham75]	R	Sybase	SQL3	Starburst	POSTGRES	[Gib84]	HIPAC
Orientation	ens. / inst.	inst.	ens.	ens. / inst.	ens.	ens.	inst.	ens. / inst.

2.4. Modèle

Définition		indép.	indép.	indép.	indép.	indép.	indép.	indép.	indép.
Manip.	Restrictions Création	non	3 l maximum	3 l maximum	3 l maximum	non	non	non	non
	Charg. / Décharg.	non	non	non	non	non	1. décl./décl 2 sélectif	non	sélectif

Description

CRITERIA / S.G.B.D.R.			*Systèmes étudiés*
Mode Intervention	1	cond. après action après	*"avant" et "après" expriment une exécution immédiate juste avant l'évènement ou juste après (respectivement)* *"fin-T" indique une execution en différé à la fin de la transaction* *(modes de 1 à 6, cf. 3.1.5.1)*
	2	cond. après action fin-T	
	3	cond. fin-T action fin-T	
	4	cond. avant action avant	
	5	cond. avant action après	
	6	cond. avant action fin-T	
Mode Exécution	A	trans déclen.	*Rapport de l'exécution du déclencheur avec la transaction de déclenchement (modes A, B et C cf. 3.1.5.2)*
	B	trans imb. dépend.	
	C	trans. indep.	
Plusieurs Déclencheurs		Autorisation	*Est-il permis que plusieurs déclencheurs soient activés par le même évènement ?*
	K	Séquentielle	*L'exécution de plusieurs déclencheurs se fait de façon séquentielle ? (cf. 3.1.5.3)*
		Résol. conflit	*Si mode K résolution de conflit ? Statiquement ou dynamiquement ?*
	L	Parallèle	*L'exécution de plusieurs déclencheurs se fait de façon parallèle ? (cf. 3.1.5.3)*
Exécution Cascade	X	Imbrication	*Mode d'exécution de déclencheurs activés en cascade (modes X et Y, cf. 3.1.5.4)*
	Y	Séquence	
		Récursion	*Dans une cascade un même déclencheur peut être activé plusieurs fois ?*
		Terminaison	*La terminaison d'une cascade est-elle assurée ? Comment ?*

3. SEMANTIQUE DE L'EXECUTION

Classement

CRITERIA \ S.G.B.D.R.			[Ham75]	R	Sybase	SQL3	Starburst	POSTGRES	[Gib84]	HIPAC
Mode Intervention	1	cond. après action après	oui	oui	oui	oui	non	oui	non	oui
	2	cond. après action fin-T	non	non	non	non	oui	non	non	oui
	3	cond. fin-T action fin-T	oui	non	non	non	non	non m	non	oui
	4	cond. avant action avant	non	oui	non	oui	non	non	non	non
	5	cond. avant action après	non	non	non	non	non	non	oui	non
	6	cond. avant action fin-T	non	non	non	non	non	non	oui	non
Mode Exécution	A	trans déclen.	non défini	oui	oui	oui	oui	oui m	oui	non
	B	trans imb. dépend.		non	non	non	non	non	non	oui
	C	trans. indep.		non	non	non	non	non	non	oui
Plusieurs Déclencheurs		Autorisation		non	non	non	oui	oui	oui	oui
	K	Séquentielle		—	—	—	oui	oui	oui	non
		Résol. conflit		—	—	—	statique dynamique	statique (priorité)	arbitaire	—
	L	Parallèle		—	—	—	non	non	non	oui
Exécution Cascade	X	Imbrication		oui	oui	oui	non	oui	oui	oui
	Y	Séquence		non	non	non	oui	non	non	non
		Récursion		oui	non	non	oui	non	oui	oui
		Terminaison		time-out	16 niveaux	—	non	—	non	non

ANNEXE II

II. EXEMPLES DE DECLENCHEURS

Cette annexe donne une idée des différents types de langages pour définir les déclencheurs. Par système étudié, un exemple est présenté ; ils sont organisés par ordre chronologique. Chaque exemple est constitué de :

- la définition du déclencheur dans la syntaxe du langage du système ;
- des commentaires (en italique) indiquant à quoi chaque clause correspond dans la terminologie utilisée dans le paragraphe 2.1 ;
- des commentaires (à droite des exemples) mettant l'accent sur les caractéristiques les plus importantes de la définition des déclencheurs.

Les différents exemples et langages de spécification sont brièvement commentés ci-dessous. Ils sont tirés de la documentation étudiée. Bien sûr, il est intéressant de se référer au paragraphe 2.1.

Hammer et McLeod - Système d'Intégrité Sémantique

Ce système représente un apport dans la spécification complète et structurée d'une contrainte d'intégrité.

Deux exemples sont donnés. Le premier définit la contrainte suivante : un employé doit avoir un salaire supérieur à 20 000 ou alors le budget de son département diminué de son salaire doit être inférieur à 100 000. Dans le cas de violation, un programme fixant un nouveau budget pour le département est exécuté dans le but de corriger la situation. La deuxième contrainte illustre la notion de contrainte d'intégrité globale (cf. 2.1.1.1.2) : elle énonce qu'une colonne quelconque de domaine Nom, appartenant à une relation quelconque, doit être un sous-ensemble des noms des employés.

Entre autre, il est important de noter que cette proposition est assez avancée, permettant la spécification d'un évènement comme une opération structurée et d'une action comme un appel à un programme d'application.

Système R

Ce système a été le premier à proposer des déclencheurs. L'évènement est spécifié de façon explicite mais la condition est donnée de façon implicite et algorithmique à l'intérieur de l'action.

Le déclencheur de l'exemple exige que le nouveau salaire d'un employé soit compris entre 0 et 12000. L'opération de refus n'existe pas : dans la clause action, une opération de compensation est spécifiée dans le but de défaire l'évènement de déclenchement.

Nous pouvons remarquer l'introduction des mots-clés "OLD" et "NEW" permettant la spécification de conditions de transition ; et des mots-clés "BEFORE", "AFTER", et "COMMIT" permettant la spécification du moment d'exécution du déclencheur.

Gibbs - Modèle Objet pour la Bureautique

La structure de ces déclencheurs est spécifiée de façon tout à fait déclarative, elle est très structurée et simple.

Le déclencheur de l'exemple évite, d'une part, qu'une catégorie inexistante soit attribuée à un employé (le refus de l'opération est fait automatiquement si la condition n'est pas vérifiée) et d'autre part, si un employé accède à une catégorie existante, son salaire est automatiquement mis à jour.

Remarquons la déclaration des variables dans la clause évènement et son utilisation dans les clauses condition et action.

Il est possible d'écrire des déclencheurs globaux.

Event / Trigger Mechanism

Ce système [Dit86] [Dit88] a été développé pour la vérification de contraintes d'intégrité en CAO. Dans ce domaine, la notion d'intégrité est assez complexe (cf. introduction chapitre 2) et a amené à une première généralisation de la notion de déclencheur. Deux extensions importantes ont été d'abord l'activation du déclencheur à des moments arbritaires, ensuite la spécification de la condition par des algorithmes qui produisent une valeur boléenne et qui comprennent des accès à la base de données mais aussi des constructeurs de programmation.

Les contraintes, les évènements et les actions peuvent être définis explicitement indépendemment d'un déclencheur. A chacun, correspond un ensemble d'opérateurs, par exemple, "CHECK CONSTRAINT" force une contrainte à être vérifiée et retourne un des deux évènements : "CONST.OK" ou "CONST.FAIL". Un autre exemple est "RAISE EVENT" qui signale explicitement un évènement.

La définition d'un déclencheur peut faire référence aux évènements, contraintes ou actions déjà définis. Les deux déclencheurs de l'exemple mettent en œuvre la contrainte suivante : avant de commencer l'édition d'une version (d'un circuit intégré par exemple) il faut vérifier si son schéma logique est complètement défini et testé pour la simulation.

Remarquons aussi l'introduction des opérations de chargement et déchargement ("ACTIVATE" et "DEACTIVATE") d'un déclencheur : ici, ces opérations ne s'appliquent qu'à un seul déclencheur.

POSTGRES Rule System

La syntaxe d'un déclencheur est la même que celle d'une requête du langage de manipulation précédée par le mot-clé "ALWAYS". Sa sémantique est que la requête doit être logiquement vue comme étant en exécution permanente. Ainsi, dans l'exemple, POSTGRES doit assurer que quand on accède au salaire de Jean, il est égal au salaire de François.

Bien sûr les spécifications de l'évènement, de la condition et de l'action sont confondues.

POSTGRES Rule System II

Une nouvelle syntaxe a été proposée dans POSTGRES pour combler les lacunes de PRS ; nous en énumérons deux :

- L'impossibilité de spécifier des contraintes de transition. En effet, la vision "en exécution permanente" d'un déclencheur est en contradiction avec la sémantique d'une condition de transition (qui n'est évaluable qu'au moment d'un changement). De plus, PRS n'offre pas de moyens pour faire référence à la valeur d'un n-uplet avant et après une mise à jour.

- Le contrôle sur l'évolution de la base de données n'est pas assez fin. Par exemple, il n'est pas possible d'implémenter tous les cas d'intégrité référentielle. Ce problème vient du fait que les opérations qui doivent être contrôlées ne sont pas explicitement spécifiées.

Ainsi, deux des changements proposés dans PRS II sont l'introduction d'une clause évènement et la possibilité de faire référence aux anciennes et nouvelles valeurs des n-uplets modifiés par un évènement. L'exemple de déclencheur est le même que pour PRS.

Starburst

La forme des déclencheurs dans ce système est assez similaire à celle de PRS II ou à celle de SQL3. En fait, la particularité de ce système est fonction de son modèle d'exécution.

Remarquons l'utilisation des relations "INSERTED" et "DELETED" (cf. 2.1.2). Ces relations ont été proposées pour des raisons d'optimisation, néanmoins leur manipulation au niveau de la définition d'un déclencheur ne semble pas très élégante (voir aussi Sybase).

Le déclencheur de l'exemple établit que, quand il y a des employés qui sont insérés, si la nouvelle moyenne des salaires (de tous les employés) est supérieure à 15 000, alors diminuer de 10 % le salaire des nouveaux employés.

Sybase

La clause évènement du déclencheur est divisée en deux parties : la spécification de la relation associée au déclencheur et la spécification du type d'opération de déclenchement. La clause action est spécifiée de façon similaire au système R : la condition est incluse de façon algorithmique dans l'action.

Le déclencheur de l'exemple implémente une contrainte d'intégrité référentielle : le département d'un employé doit exister dans la relation représentant le département. De plus, si la contrainte est vérifiée au moment de la mise à jour du département d'un employé, le déclencheur met à jour le nombre d'employés travaillant pour chaque département.

Ce déclencheur est activé par un "INSERT" ou un "UPDATE". La mise à jour du nombre d'employés de chaque département est assez complexe : il doit être augmenté du nombre d'employés ("COUNT") de la relation "INSERTED" travaillant pour ce département ("GROUP BY") et diminué du nombre d'employés de la relation "DELETED" travaillant pour ce département.

SQL3

La syntaxe de définition d'un déclencheur ressemble à celle de PRS II et de Starburst. Dans la clause évènement, un raffinement au niveau de l'attribut pour l'opération de modification est permis comme pour SYBASE. La clause "REFERENCING" n'est utilisée que dans le cas d'une opération de modification ("UPDATE") et permet de spécifier la façon dont on fera référence, dans les clauses condition et action, à l'ancienne et à la nouvelle valeur d'un n-uplet mis à jour. La présence des mots clés "FOR EACH ROW" permet de spécifier que le déclencheur doit être exécuté instance par instance et non pas de façon ensembliste.

Ce déclencheur énonce que si un employé est augmenté de 20 %, alors il faut mettre à jour la relation GrandeProgression.

HIPAC

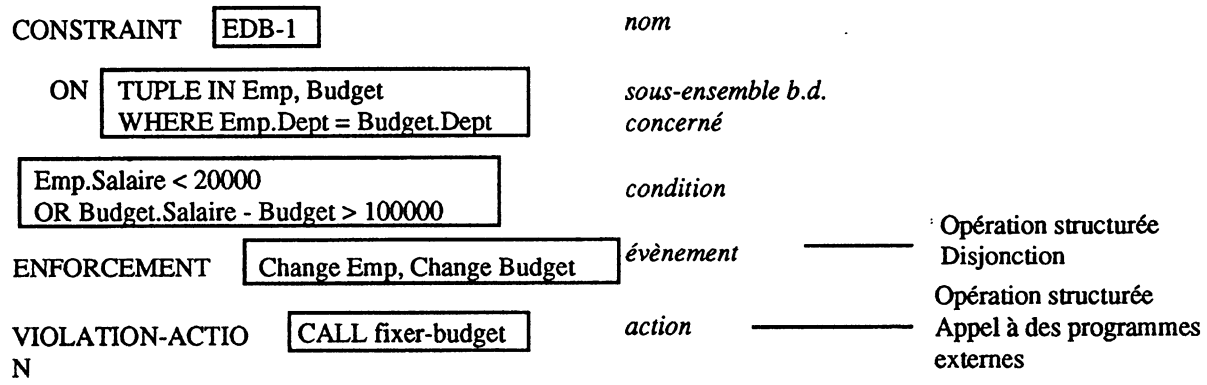
La spécification d'un déclencheur inclut les trois clauses d'un déclencheur de base plus deux clauses permettant la spécification du moment (immédiat ou differré) et du mode d'exécution (p.ex. dans une transaction imbriquée à confirmation dépendante ou dans une transaction séparée à confirmation indépendante) de la condition et de l'action. Inclure des clauses permettant la spécification du modèle d'exécution du déclencheur est nouveau et représente un apport important. Dans HIPAC, le modèle d'exécution est en effet plus puissant et flexible.

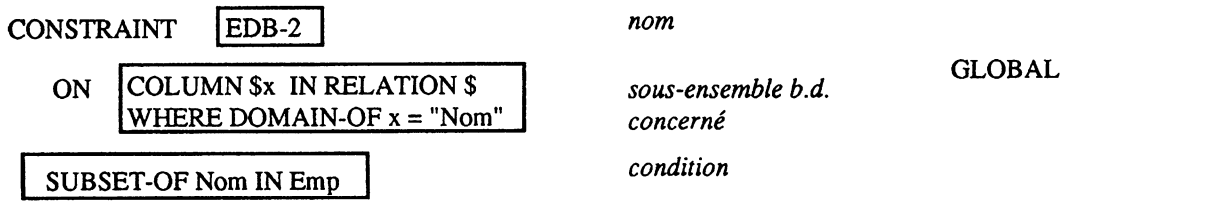
Le déclencheur de l'exemple est activé quand une cible change de position, la condition doit déterminer quels sont les navires menacés (i.e. les navires qui étaient éloignés de plus de 10 unités de distance de la cible et qui sont maintenant proches de moins de 10 unités). L'action à entreprendre est tout d'abord un avertissement à l'utilisateur, suivi du chargement d'un ensemble de règles et de la signalisation explicite d'un évènement.

Il est très important de voir l'utilisation des variables dans les trois clauses, des opérations de chargement et déchargement d'un ensemble de règles et la signalisation explicite d'un évènement.

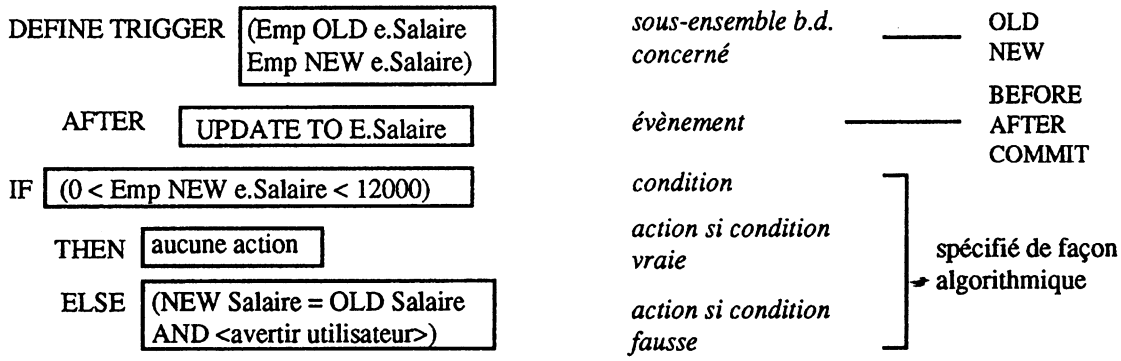
EXEMPLES DE DECLENCHEURS ➡

75 HAMMER ET McLEOD - SYSTEME D'INTEGRITE SEMANTIQUE

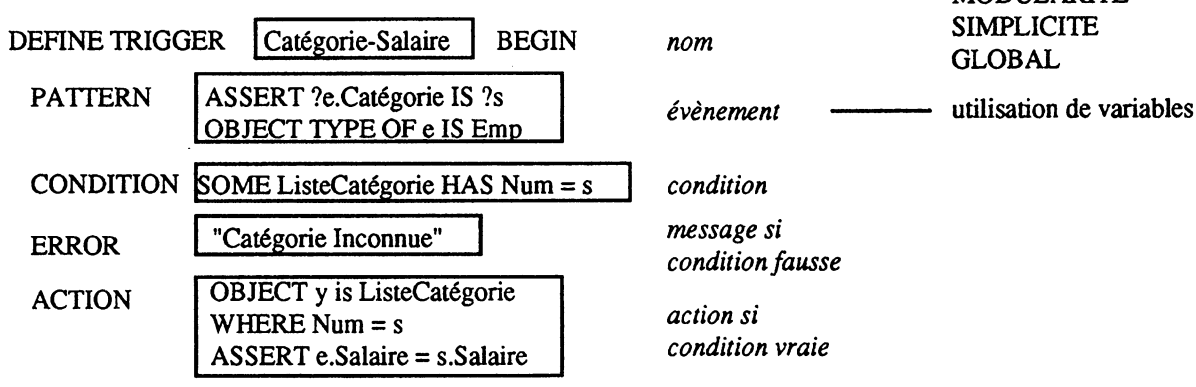




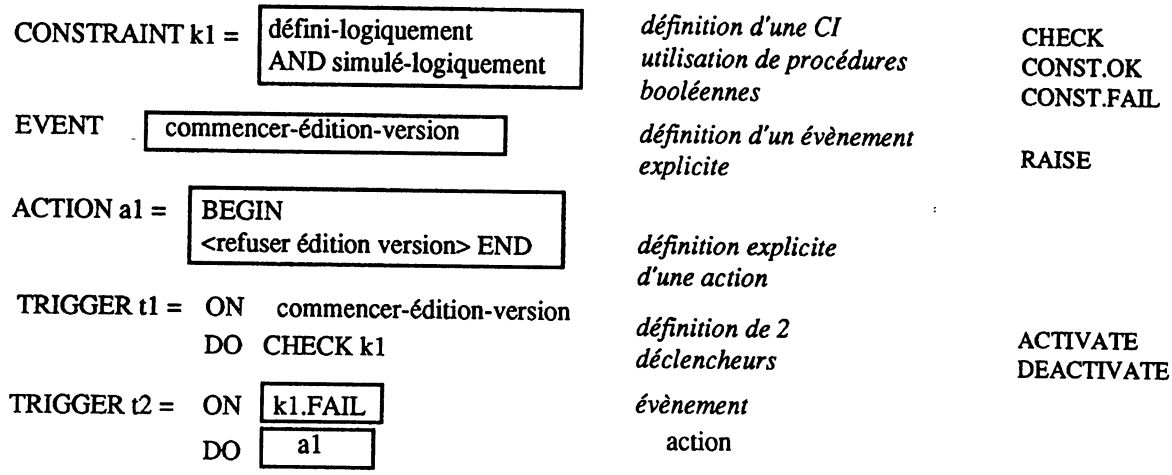
75 / 76 SYSTEME R



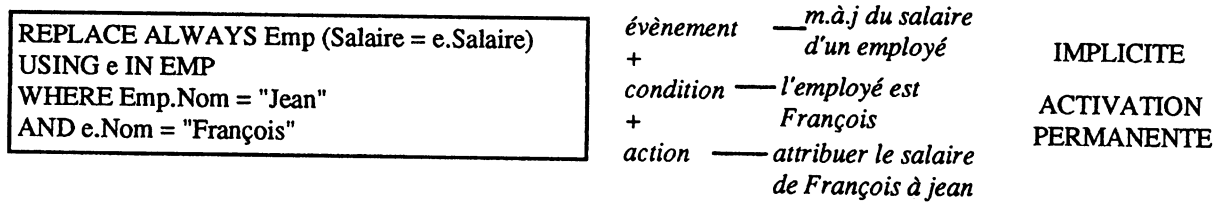
84 GIBBS - MODELE OBJET POUR LA BUREAUTIQUE



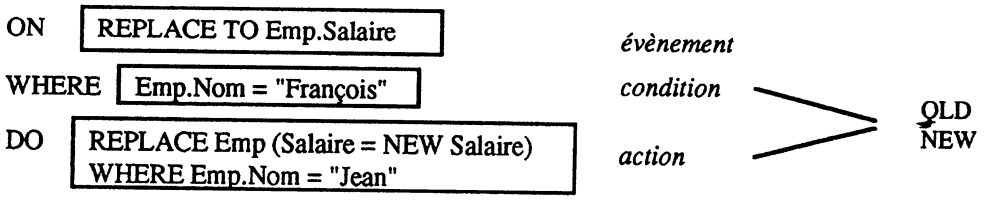
86 EVENT / TRIGGER MECHANISM - C.A.O.



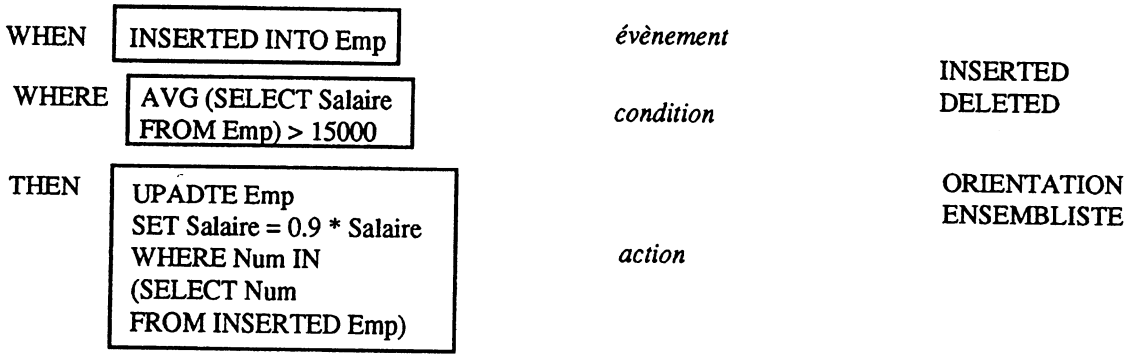
86 POSTGRES RULE SYSTEM



89 POSTGERS RULE SYSTEM II



89 STARBURST



88 SYBASE

```
CREATE TRIGGER [EmpMaj]                    nom
ON             [Emp]                       objet de l'évènement
FOR            [UPDATE AS                  opération de l'évènement ——— Raffinement au niveau de l'attribut
                IF UPDATE Ndept]
BEGIN
  IF   [SELECT COUNT (*)                   condition                    ┐
        FROM INSERTED, Dept                                           │
        WHERE INSERTED.Ndept = Dept.Num]                              │ donnée de façon
  THEN [ROLLBACK TRANSACTION]              action si condition vraie    │ algorithmique
  ELSE [mettre à jour le nombre            action si condition fausse   │
        d'employés dans la relation Dept,                             │   INSERTED
        basé sur les relations INSERTED                               │   DELETED
        et DELETED]                                                   ┘
END
```

89 SQL3

```
CREATE TRIGGER [SalEmp]                    nom
BEFORE         [UPDATE OF Salaire ON Emp]  évènement ——— raffinement au niveau de l'attribut
REFERENCING    [OLD Oldemp NEW Newemp]     contexte  ——— OLD
                                                         NEW
WHEN           [(Newemp.Salaire - Oldemp.Salaire >
                0.2*Oldemp.Salaire)]       condition
[INSERT INTO GrandeProgression (Nom, Augment)
 VALUES (Oldemp.Nom, Newemp.Salaire -
 Oldemp.Salaire / Oldemp.Salaire)]          action
FOR            [EACH ROW]                  spécif. instance/instance ——→ INSTANCE / INSTANCE ENSEMBLISTE
```

89 HIPAC

RULE 1 (TRIGGER)

EVENT :

```
Rapport-Position [C:Cible, Old_Pos [Lat,Long],
                 New-Pos[Lat,Long]]
```

évènement ——— utilisation de variables

CONDITTION :

COUPLING-MODE : DECOUPLED, CAUSALLY-DEPENDENT

mode d'évaluation de la condition — IMMEDIATE, DEFERRED, COUPLED, DECOUPLED, CAUSALLY-DEPENDENT, CAUSALLY-INDEPENDENT

QUERY :

```
Menace [Navire, Pos-Nav] =
 FOR N in Navire WHERE
     Distance (Position(N),Old-Pos)>10
     AND Distance (Position(N), New-Pos)<10
   RETRIEVE [N, Position (N)]
 ENDFOR
```

condition ——— collection d'interrogations, utilisation de variables

ACTION :

COUPLING_MODE : IMMEDIATE

mode d'exécution de l'action — même chose que pour la condition

OPERATION :

```
FOR X in Menace
   Afficher ("code rouge", Navire(X),
      Pos-Nav(X), C, New-Pos)
   ENABLE {R IN Règles-Code-Rouge}
   SIGNAL (Initiation-Mesures (Pos-Nav, C))
ENDFOR
```

action — utilisation variables, utilisation résultats condition

ENABLE ensemble de règles
DISABLE idem
SIGNAL évènement

ANNEXE III

III. SCHEMA CONCEPTUEL : CONCESSIONNAIRE DE VOITURES

Certte annexe présente le schéma conceptuel utilisé pour illustrer les exemples d'activités et de procédures donnés dans les chapitres 4 et 5.

Contrat : Entity ;
[Contrat (1,1) , Name (1,1)]
Employé : Entity ;
Pers-ESPRIT : Entity ;
[Contrat (1,*) , Pesr-ESPRIT (1,*)] ;
[Employé (1,1) , Num (1,1)] ;
[Employé (1,*), Salaire (1,1)] ;
Fiche-Pers : Entity ;
[Pers-ESPRIT (1,*) , Fiche-Pers (0,*)] ;
[Fiche-Pers (1,*) , Mois (1,1)] ;
[Fiche-Pers (1,*) , Nbr-Heures (1,1)] ;
Equipement : Entity ;
[Contrat (1,*) , Equipement (0,*)] ;
Dép-Equipement : Entity ;
[Dép-Equipement (1,*), Equipement (0,*)] ;
Dép-Mensuel : Entity ;
[Dép-Mensuel (1,*) , Contrat (1, *)] ;
[Dép-Mensuelle (1,1), Mois (1,1)] ;
[Dép-Mensuelle (1,1), Montant-Pers (1,1)] ;
[Dép-Mensuelle (1,1), Montant-Equipement (1,1)] ;
Status : Entity ;
[Contrat : (1,*) , Etat (1,1)] ;
Justificatif :Document ;
[Contrat (1,*), Justificatif (1,1)] ;
Nom : String ;
Num : Integer ;
Département : Entity ;
Contrat : Entity ;
[Employé (0,1) , Nom (1,1)] ;
Employé (0,1) , Salaire (1,1)]
[Employé (1,1) , Département]
[Contrat (0,1) , Num (1,1)]
[Département (0,1) , Budget (1,1)]

RESUME :

Cette thèse propose un modèle et un ensemble de mécanismes pour supporter le développement de bases de données actives.

D'une part, le modèle permet la description intégrée des aspects statiques et dynamiques des applications. Pour décrire la structure des traitements sur les informations trois concepts sont proposés : entité focale, activité et procédure.

D'autre part, ce modèle est supporté par un ensemble d'outils et de mécanismes permettant de lancer automatiquement, contrôler et suivre l'exécution des activités et procédures.

Un des mécanismes qui a mérité une attention particulière est celui du déclenchement automatique d'une activité ou d'une procédure. Ceci implique l'évaluation de préconditions liées aux activités et aux procédures. Leur évaluation est réalisée lors des mises à jour et nécessite potentiellement des accès complexes à la base de données. Cette évaluation se fait en deux phases :

- la première détermine le sous-ensemble de préconditions à évaluer pour une certaine mise à jour. Ce travail a abouti à l'établissement d'un ensemble de règles de correspondance entre les préconditions et les mises à jour.
- la deuxième réalise l'évaluation proprement dite des préconditions. Ce travail a abouti à la définition d'une méthode d'évaluation efficace qui utilise un ensemble de techniques d'optimisation.

D'autres mécanismes à souligner sont :

- l'interaction entre les mécanismes de déclenchement et d'exécution des activités et procédures avec le système transactionnel.
- le contrôle persistant de l'exécution des procédures.

Ce travail est complété par l'implémentation des différents services proposés.

MOTS-CLE :

Bases de données actives, modèle dynamique, activité, procédure, entité focale, évaluation de préconditions, déclenchement automatique, contrôle persistant.