Parrallélisme et transactions dans les bases de données à objets

Javam Castro Machado

▶ To cite this version:

Javam Castro Machado. Parrallélisme et transactions dans les bases de données à objets. Interface homme-machine [cs.HC]. Université Joseph-Fourier - Grenoble I, 1995. Français. NNT : . tel-00005039

HAL Id: tel-00005039
https://theses.hal.science/tel-00005039

Submitted on 24 Feb 2004

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

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

Thèse

présentée par

Javam de CASTRO MACHADO

pour obtenir le grade de Docteur
de l'Université Joseph Fourier - Grenoble 1
(arrêté ministériel du 30 Mars 1992)
Spécialité : **Informatique**

Parallélisme et Transactions dans les Bases de Données à Objets

---

Date de soutenance : 13 octobre 1995

Composition du jury :

Président :	Roland Balter
Rapporteurs :	Valéri Doliatovski
	Philippe Richard
Examinateurs :	Brigitte Plateau
	Christine Collet
	Michel Adiba

Thèse préparée au sein du
Laboratoire de Génie Informatique – IMAG

A Rosa,

Levi et Filipe

Remerciements

Je tiens à remercier :

Mr Roland Balter, Professeur à l'Université Joseph Fourier de Grenoble, pour m'avoir fait l'honneur de présider ce jury de thèse.

Mr Valéri Doliatovski, Professeur à l'Université de Rostov en Russie, et Mr Philippe Richard, Directeur de Recherches à l'Alcatel Alsthom Recherche, d'avoir bien voulu apporter leur jugement sur ce travail et qui par leurs remarques ont contribué à la qualité de ce document.

Mme Brigitte Plateau, Professeur à l'ENSIMAG, qui me fait l'honneur de participer à ce jury.

Mr Michel Adiba, Professeur à l'Université Joseph Fourier de Grenoble, sans qui cette thèse n'aurait pas pu voir le jour. Je voudrais lui exprimer ma profonde gratitude pour m'avoir accueilli dans son équipe et pour avoir diriger mes recherches.

Mme Christine Collet, Maître de Conférence à l'Université Joseph Fourier, pour sa disponibilité, ses critiques constructives et pour les discussions fructueuses tout au long de ces trois années de travail. Ses corrections et propositions lors de la rédaction ont été d'une grande importance pour la qualité de cette thèse.

Mr Yves Chiaramella, Professeur à l'Université Joseph Fourier et Directeur du Laboratoire de Génie Informatique, pour l'accueil au sein du LGI.

A la CAPES/Brésil et au Departamento de Computação de l'Universidade Federal do Ceará, pour m'avoir soutenu au long de la préparation de cette thèse.

Mes collègues du LGI, en particulier Boualem, Celso, Claudia, Hervé, Marie-Christine, Philippe et Thierry, pour les tête-à-têtes à la Kfet ou à la salle de réunions et pour les corrections "français" dans des versions préliminaires de ce document.

Keila et Walcélio, pour l'accueil chaleureux au début et encouragements lorsque cette thèse n'était qu'un projet "pour le futur". Mariza et Luis ainsi que Lara et Marcelo, pour l'amitié et la disponibilité d'un "coin de chez eux" lors de l'étape finale de ce travail. Kita, Monica et Marcio, Ofélia et Celso, pour la compagnie et le soutien tellement nécessaires dans des jours particulièrement difficiles.

Rosa, pour l'amour, la patience et la disponibilité dont elle a fait preuve tout au long de ces années de formation. Elle a su, par sa tendresse ou tout simplement par sa présence, me soutenir dans des moments où tout semblait s'écrouler. Nos discussions, jusqu'au tard le soir, sont, d'une certaine façon, à l'origine de plusieurs idées présentes dans ce travail.

Table des matières

1 Introduction **1**
1.1 Contexte et motivation 1
1.2 Problématique 2
1.3 Contribution de la thèse 3
1.3.1 Le parallélisme intra-transaction 4
1.3.2 Le parallélisme intra-application 4
1.3.3 Prototypage et exploitation du modèle 4
1.3.4 Etude de performances 5
1.4 Organisation de la thèse 5

2 L'État de l'Art **9**
2.1 Les Modèles de Programmation Parallèle 11
2.1.1 Parallélisme asynchrone 13
2.1.1.1 Multiprocesseurs à mémoire partagée 13
2.1.1.2 Multiprocesseurs sans mémoire commune 14
2.1.2 Partage des ressources 18
2.1.3 Parallélisation des programmes séquentiels 20
2.1.3.1 Dépendance des données 20
2.1.3.2 Transformation de code 21
2.2 Le parallélisme dans les langages à objets 23
2.2.1 Modèle d'exécution 23
2.2.1.1 Le modèle d'acteurs 23
2.2.1.2 Les objets 25
2.2.1.3 Classe et héritage 26
2.2.2 Concurrence Interne 27
2.2.2.1 Les objets séquentiels 27
2.2.2.2 Les objets quasi-concurrents 28
2.2.2.3 Les objets concurrents 29
2.2.3 Communication et Synchronisation 30

2.2.3.1 L'envoi de message 31
2.2.3.2 L'acceptation de message 32
2.3 Parallélisme dans les systèmes de gestion de bases de données 34
2.3.1 Traitement parallèle des requêtes 35
2.3.1.1 Parallélisme intra-opération 35
2.3.1.2 Parallélisme inter-opérations 41
2.3.2 Transactions et concurrence 43
2.3.2.1 Sérialisation 44
2.3.3 Relations de conflit 46
2.3.3.1 Commutativité 47
2.3.3.2 Dépendances séquentielles 49
2.3.3.3 Recouvrabilité 50
2.3.3.4 Localité associée à la commutativité et à la recouvrabilité 51
2.4 Conclusion 53

3 Parallélisation des Transactions 55
3.1 Le modèle à objets 57
3.1.1 Les objets et leurs types 58
3.1.2 Schéma et base de données 61
3.1.3 Transactions 62
3.2 Parallélisme intra-transaction 63
3.2.1 L'expression du parallélisme 64
3.2.1.1 Parallélisme explicite 64
3.2.1.2 Parallélisme par transformation 65
3.2.2 Mise en évidence du parallélisme 66
3.2.2.1 Environnement des opérations 66
3.2.2.2 Notation 67
3.2.2.3 Le mode d'accès 69
3.2.2.4 Fermeture transitive locale 71
3.2.2.5 La relation de conflit 75
3.2.2.6 La fermeture transitive au niveau du schéma 76
3.2.2.7 La compatibilité 79
3.3 Parallélisation des transactions 81
3.3.1 Le plan d'exécution source 81
3.3.2 Construction du plan d'exécution parallèle 82
3.3.2.1 Schéma transactionnel 82
3.3.2.2 L'algorithme de transformation du graphe 84

3.3.2.3 La cohérence sémantique des transactions 88
3.3.3 Les mécanismes de base pour la parallélisation 90
3.3.3.1 La création des activités 90
3.3.3.2 La synchronisation des activités 92
3.4 Conclusion 93

4 Expérimentation 95
4.1 O_2Parall 97
4.1.1 Architecture logique 98
4.1.2 Fonctions générales 99
4.1.3 Gestion du catalogue 101
4.1.3.1 Gestion des entités nommées 101
4.1.3.2 Gestion des types 102
4.1.3.3 Gestion des classes et des méthodes 103
4.1.4 Compatibilité des méthodes 105
4.1.4.1 Vecteurs d'accès 106
4.1.4.2 Matrice de compatibilité 108
4.1.5 Transformation des transactions 110
4.1.5.1 Applications et transactions 111
4.1.6 Modules du système 112
4.1.6.1 Le module Toolbox 114
4.1.6.2 Le module CatalogueMgr 114
4.1.6.3 Le module TrParallelyser 115
4.2 Parallélisation des règles 115
4.2.1 Règles du système NAOS 116
4.2.1.1 Définition des règles 116
4.2.1.2 Traitement de règles 117
4.2.2 Parallélisation des règles 119
4.2.2.1 Compatibilité de règles 120
4.2.2.2 Exécution parallèle des règles 122
4.2.3 Architecture du système 123
4.2.3.1 Générateur de tables de compatibilité 124
4.2.3.2 Parallélisateur de règles 124
4.3 Conclusion 125

5 Parallélisation des Applications 127
5.1 Transactions classiques et parallélisme 129
5.1.1 Applications et transactions 130
5.1.2 Compatibilité des transactions 130

5.1.3 Transactions parallèles 132
5.1.4 Discussion 134
5.2 Transactions emboîtées 134
5.2.1 Définition 135
5.2.1.1 Propriétés 137
5.2.1.2 Gestion des verrous 138
5.2.2 Transactions emboîtées multi-activités 142
5.2.2.1 Couplage fort 143
5.2.2.2 Couplage faible 145
5.3 Transactions emboîtées multi-activités par parallélisation 149
5.3.1 Motivation 150
5.3.2 Isolation des transactions et des activités 151
5.3.3 Parallélisme horizontal et vertical 152
5.3.4 Exemple 154
5.4 Conclusion 155

6 Etude des performances 157
6.1 Introduction 159
6.2 Le temps d'exécution d'une transaction 161
6.2.1 Les transactions séquentielles 161
6.2.2 Les transactions parallèles 163
6.3 Les gains apportés par la parallélisation 164
6.3.1 Nombre de blocs et degré de parallélisme 165
6.3.1.1 Graphe de parallélisation totale 165
6.3.1.2 Graphe sans parallélisation 166
6.3.1.3 Graphe de parallélisation partielle 166
6.3.2 La surcharge 167
6.3.2.1 Coût de la surcharge 168
6.3.2.2 Temps d'exécution des opérations 172
6.4 Evaluation des stratégies d'exécution 177
6.4.1 Complexité des opérations 178
6.4.2 Ordonnancement des opérations 180
6.5 Les performances et les transactions emboîtées avec parallélisme . . . 184
6.5.1 Temps d'exécution des transactions 184
6.5.2 Gains en performance 185
6.6 Conclusion 188

7 Conclusion 191

7.1 Les résultats 191

7.2 Les perspectives 193

A La Méthode Booch 209

A.1 Diagramme de classes 209

A.1.1 Les classes 209

A.1.2 Relations entre classes 210

A.2 Catégories de classes 212

A.3 Diagramme d'objets 212

A.3.1 Objets 212

A.3.2 Relations entre objets 213

A.3.2.1 Les messages 213

A.4 Diagramme de modules 213

A.4.1 Modules 214

A.4.2 Dépendances 214

Table des figures

2.1 Un exemple d'exécution en pipeline 12
2.2 MIMD à mémoire partagée 13
2.3 MIMD sans mémoire commune 14
2.4 Les commandes fork et join 15
2.5 L'appel de procédure à distance 17
2.6 La méthode de parallélisation des boucles 22
2.7 Comportement d'un acteur. 24
2.8 Définition d'une classe d'acteurs ACT++. 26
2.9 Objet actif séquentiel. 28
2.10 Objet actif quasi-concurrent. 29
2.11 Objet actif concurrent. 30
2.12 Les Types de Messages. 31
2.13 La synchronisation d'un acteur ACT++. 33
2.14 Exemple de distribution initiale des relations 38
2.15 Distribution des fragments d'un produit parallèle par boucles imbriquées 39
2.16 Exemple de produit parallèle par tri-composition 39
2.17 Exemple de produit parallèle par hachage 41
2.18 Arbres de traitement 42
2.19 Exécution d'une transaction 44
2.20 Exemples de transactions 45
2.21 Exemples de historiques 45
2.22 Structure d'un objet du type Compte 52

3.1 Un type de données abstrait 58
3.2 TADs et l'héritage 59
3.3 Les opérations du type Compte. 60
3.4 Exemple de base de données 61
3.5 Exemple de spécification de transaction 62
3.6 Arbre de flot de messages locaux 73
3.7 Arbre de flot de messages de l'opération virement 74

3.8 Exemple de conflit d'accès indirect 77
3.9 Arbre de flot de contrôle global 78
3.10 Exemple de matrice de compatibilité 80
3.11 Exemple de code d'une transaction séquentielle 82
3.12 Graphe de plan d'exécution parallèle 86
3.13 Transformation des itérations 87
3.14 Transformation des branches alternatives 88
3.15 Flot de contrôle avec parallélisme 90
3.16 Exemple de configuration d'une transaction 91
3.17 Exécution des instructions élémentaires 92
3.18 Exemple de transformation du code d'une transaction 92

4.1 Diagramme de catégories de classes 98
4.2 Diagramme des classes utilitaires 100
4.3 La modélisation des collections d'objets 101
4.4 Les entités nommées 102
4.5 Les types du schéma 103
4.6 Les classes du schéma 104
4.7 Les classes MethodCompat et AccessMode 105
4.8 Exemple de vecteur d'accès 106
4.9 Exemple de vecteurs d'accès 107
4.10 Construction des vecteurs d'accès 108
4.11 Vérification de compatibilité de vecteurs d'accès 109
4.12 Construction de la matrice de compatibilité 109
4.13 Un exemple de graphe de transaction source 110
4.14 Exemples de transformation de graphe source 111
4.15 Une application et ses transactions 112
4.16 Diagramme d'objets des classes Application et Transaction 113
4.17 Diagramme de modules du système 114
4.18 La définition d'une règle 116
4.19 Cycles d'exécution pour les règles immédiates 119
4.20 Cycles d'exécution pour des règles différées 120
4.21 Cycles d'exécution pour les règles immédiates et différées 123
4.22 Architécture du système 124

5.1 Exemple d'une application 131
5.2 Exemple de transformation d'application 134
5.3 Structure des transactions emboîtées 135
5.4 Les types de parallélisme 137

5.5 Exemple de contrôle de verrou et la délégation ascendante 141
5.6 Délégation ascendante et les verrous exclusifs 142
5.7 Exemple de contrôle de verrou et la délégation ascendante 142
5.8 Partage de données par les activités de Clouds 146
5.9 Exemple d'application Camelot 147
5.10 Les activités et les transactions dans Venari/ML 149
5.11 Les activités et le parallélisme horizontal des transactions 152
5.12 Les activités et le parallélisme vertical des transactions 153
5.13 Application avec des transactions emboîtées 154
5.14 Des transactions emboîtées multi-activités 155

6.1 Diagramme de Gantt d'une transaction séquentielle 162
6.2 Diagramme de Gantt d'une transaction avec parallélisme 163
6.3 Le temps d'exécution d'une transaction parallèle 165
6.4 Exemple d'un graphe de parallélisation partielle 167
6.5 Exemple détaillé de formation de la surcharge 169
6.6 Temps d'exécution des blocs d'opérations 170
6.7 Graphe d'une transaction avec temps d'exécution des opérations . . . 173
6.8 Temps d'exécution pour les opérations 174
6.9 Diagramme de Gantt pour un bloc ayant grande surcharge 175
6.10 La surcharge et l'ordonnancement décroissant des opérations 176
6.11 Exemple de regroupement des opérations 177
6.12 Exemple de chemin critique 178
6.13 Le regroupement des opérations 181
6.14 Ordonnancement des activités et le regroupement 182
6.15 Ordonnancement avec un "pool" d'activités 183
6.16 Le graphe d'une transaction emboîtée 185
6.17 Exemple d'un graphe d'application 186
6.18 Diagrammes de Gantt d'une application avec transactions emboîtées . 187

A.1 Les classes 210
A.2 Les relations entre les classes 211
A.3 Les relations entre les classes 211
A.4 Les catégories de classes 212
A.5 Les objets 213
A.6 Les envois des messages 214
A.7 Les modules des classes 214

Liste des tableaux

2.1 Commutativité des opérations du TAD Compte 49
2.2 Recouvrabilité des opérations du TAD Compte 51
2.3 Dépendances induites par les opérations du TAD Compte 53

3.1 Ensembles d'accès du type Compte. 70
3.2 Ensembles d'accès du type Compte après la fermeture transitive. . . 74
3.3 Mode d'accès pour les instructions d'action. 84

4.1 Les catégories, leurs classes et fonctions. 99
4.2 La relation de compatibilité pour un ensemble de règles. 121

5.1 Ensembles d'accès des instructions d'action. 132

Chapitre 1

Introduction

1.1 Contexte et motivation

Les systèmes de gestion de bases de données ont d'importants besoins en termes de performances. Des solutions matérielles aussi bien que logicielles ont été envisagées pour répondre à ces besoins. Du côté matériel, l'utilisation des machines parallèles multiprocesseurs est une solution qui peut apporter la puissance nécessaire aux systèmes de bases de données. Du côté logiciel, des techniques telles que l'optimisation des requêtes et la gestion des accès concurrents permettent à ces systèmes de servir plusieurs applications de manière efficace.

La concept de transaction est la première approche pour exploiter le parallélisme dans les systèmes de bases de données [EGLT76]. A travers la notion de *sérialisation*, on peut servir une application avant d'en finir une autre. Ainsi, le système est capable d'entrelacer l'exécution des *opérations* de plusieurs transactions, ce qui "simule" en quelle sorte le parallélisme. Dans cette approche, les systèmes de bases de données utilisent des machines monoprocesseurs incapables de fournir du vrai parallélisme. Pour pallier ce manque matériel, le modèle de transactions emboîtées a été introduit et permet l'exécution de sous-transactions en parallèle [Mos85].

L'utilisation des machines parallèles multiprocesseurs ouvre différentes possibilités d'exploitation du parallélisme dans les systèmes de bases de données. Non seulement le système peut offrir du (vrai) parallélisme entre les applications, mais on peut exploiter le parallélisme dans l'exécution d'une même application. Les premières approches sont apparues dans le cadre des systèmes de bases de données relationnelles [DG92, Omi95].

Ces systèmes exploitent le parallélisme inhérent aux requêtes exprimées dans le langage SQL [BBDW83, DG85, Val87]. Il ne s'agit plus de simuler le parallélisme entre plusieurs applications, mais d'exécuter une requête en parallèle. Une requête peut en effet être traitée par plusieurs processeurs. Le parallélisme est alors exploité

sous deux formes différentes :

- Le parallélisme *intra-operation* qui permet l'exécution d'une opération relationnelle sur différents fragments d'une relation. Par exemple, l'opération de projection d'une relation peut être effectuée en parallèle sur trois de ses fragments.

- Le parallélisme *intra-requête* qui permet l'exécution de plusieurs opérations relationnelles à l'intérieur de la même requête. Par exemple, pour une requête ayant une sélection sur une relation et une projection sur une autre, on peut exécuter en parallèle la sélection et la projection.

Dans les systèmes relationnels, le parallélisme est *implicite* et obtenu par traduction et *transformation* de la définition d'une requête. Ce processus est communément appelé *optimisation des requêtes*. La spécification d'une requête est effectuée exactement de la même façon pour les deux cas d'exécution : avec ou sans parallélisme.

Dans les nouvelles générations de bases de données, dites à objets, le système gère à la fois les données et les opérations sur celles-ci. Le fait d'avoir différentes formes d'expression de traitement des données, notamment impératives, ouvre d'autres possibilités d'exploitation du parallélisme.

1.2 Problématique

Cette thèse s'interesse à l'exploitation du parallélisme dans les applications utilisant des SGBD à objets. La programmation des applications se fait en utilisant dans la plupart des systèmes soit un langage de programmation à objet incluant des mécanismes de gestion des transactions et de la persistence, soit un langage de programmation étendu pour la gestion d'objets persistents.

Notre travail s'apparente à ceux réalisés pour l'exploitation du parallélisme dans les *langages à objets* qui considèrent la notion d'objet, en tant qu'unité d'encapsulation des données [WKH92]. Ainsi, la forme la plus naturelle pour exploiter le parallélisme dans ces langages consiste à exécuter en parallèle des opérations sur des objets distincts. On parle alors du parallélisme entre les objets. Toutefois, certaines opérations d'un objet sont *indépendantes* les unes des autres et peuvent donc être exécutées en parallèle. On parle alors du parallélisme entre les opérations d'un même objet.

Dans la plupart des langages à objets, le parallélisme est entièrement spécifié et contrôlé par l'utilisateur. On est donc dans un paradigme où le parallélisme est dit *explicite*. Le programmeur est responsable à la fois de la synchronisation des messages

entre les objets parallèles, mais aussi de l'exclusion mutuelle des opérations parallèles au sein d'un objet.

Nous cherchons à exploiter le parallélisme à d'autres niveaux d'abstraction que celui d'une requête SQL, à savoir le parallélisme intra-transaction et intra-application.

Le problème consiste à définir un modèle d'exécution avec parallélisme pour les applications base de données sans introduire les difficultés déjà connues dans le dévélopment des applications parallèles [Per87, Les93]. Pour cela, trois aspects sont à considérer :

- Nous devons déterminer l'unité de parallélisation. Autrement dit, l'abstraction logiciel qui sera exécutée en parallèle et son contexte d'exécution.
- Le non déterminisme dans l'exécution des applications doit être éliminé dans notre approche par des mécanismes de synchronisation des activités parallèles.
- Notre travail doit fournir des techniques automatiques de contrôle de partage de données de manière à isoler les activités les unes des autres.

1.3 Contribution de la thèse

Nous avons développé une première approche pour l'exploitation du parallélisme dans un langage de programmation pour bases de données [Dec93]. Dans cette approche, le parallélisme était exploité par l'exécution parallèle des objets dits *actifs* au sein d'une application. Nous avons défini différentes formes de communication et de synchronisation entre objets, de manière à ce que leur exécution parallèle soit contrôlée par le programmeur. Ce travail est décrit dans : [MCD+93a, MCD+93b].

Nous proposons ici une approche du parallélisme par transformation de code basée sur certaines propriétés [CM96]. Ainsi, quelque soit le code à paralléliser dans une application — transaction ou opération — une étape d'analyse du code est effectuée afin d'obtenir les informations nécessaires à la parallélisation. Les propriétés sont donc obtenues de manière statique et permettent la construction de *relations de compatibilité* entre les codes. Les codes compatibles peuvent à priori être parallélisées.

Nous avons défini un cadre formel pour spécifier la relation de compatibilité de code de même niveau d'abstraction. Ce formalisme permet de définir une *relation d'équivalence* entre une exécution séquentielle d'un niveau d'abstraction et une exécution parallèle de même niveau. Par exemple, lorsqu'on parallélise les opérations d'une transaction, nous pouvons montrer que le résultat de l'exécution séquentielle de ces opérations est équivalent à celui d'une exécution parallèle des mêmes opérations.

La parallélisation proprement dite du code consiste, entre autres, à créer des *activités* parallèles au sein d'une application et à insérer des *primitives de synchronisation* dans le code transformé. Ainsi nous introduisons du calcul asynchrone dans une application tout en contrôlant le non déterminisme inhérent à ce type de calcul. Nos transformations sont effectuées de telle sorte que le calcul asynchrone ne dépasse jamais le niveau abstraction dans lequel il est introduit.

1.3.1 Le parallélisme intra-transaction

Notre approche pour la parallélisation des transactions considère un modèle classique, où une transaction est une séquence d'opérations. Nous considérons les opérations des types de données abstraits ou les méthodes d'une classe. Le parallélisme intra-transaction consiste à transformer une transaction pour pouvoir exécuter ses opérations en parallèle. L'unité de parallélisation est donc l'opération.

Nous représentons le code d'une transaction par un graphe dont les nœuds décrivent les opérations et les arcs montrent l'ordre d'exécution. Nous avons mis en place des algorithmes pour déterminer la relation de compatibilité des opérations et pour transformer ce graphe dans un nouveau graphe qui correspond au code de la transaction avec parallélisme.

1.3.2 Le parallélisme intra-application

Notre approche pour l'exploitation du parallélisme intra-application étend tout d'abord le modèle de parallélisme intra-transaction de manière à considérer la transaction comme unité de parallélisation. Dans ce cas, les transactions qui forment l'application peuvent être parallélisées selon une relation de compatibilité. Nous avons donc étendu nos algorithmes pour calculer la compatibilité des transactions et pour transformer le code d'une application.

Deuxièmement nous avons considéré l'exploitation du parallélisme intra-application dans un contexte des transactions emboîtées. Nous proposons d'associer le parallélisme offert par le modèle des transactions emboîtées avec celui offert par transformation.

1.3.3 Prototypage et exploitation du modèle

Nous avons développé un premier prototype qui met en œuvre le modèle de parallélisation des transactions. Pour cela, nous avons utilisé le système de bases de données à objet O_2. Notre prototype calcule la relation de compatibilité des méthodes des classes d'un schéma O_2 et représente cette relation à travers une *matrice de com-*

patibilité. En outre nous avons développé une application O_2 ayant des transactions composées de méthodes. Notre prototype construit alors les graphes qui représentent les transactions de cette application et transforme ces graphes selon la matrice de compatibilité des méthodes.

Le prototype introduit le parallélisme par la création et la synchronisation des activités parallèles au sein du processus client O_2 qui exécute l'application. Le système étant développé sur une machine Unix monoprocesseur, les fonctions liées au parallélisme utilisent des processus légers.

Nous avons ensuite appliqué notre modèle au système de règles actives NAOS [CM95]. Dans NAOS, l'exécution de règles a lieu dans des *cycles d'exécution* [CCS94]. L'ordonnancement de l'exécution d'un ensemble de règles prend en compte la relation de compatibilité entre celles-ci. Nous construisons un plan d'exécution pour les règles candidates d'un cycle qui en détermine l'exécution séquentielle ou parallèle. Nous avons adapté nos algorithmes de parallélisation des transactions pour pouvoir construire le plan d'exécution parallèle pour les règles actives.

1.3.4 Etude de performances

Nous avons effectué une étude théorique sur les gains en performance dans l'exécution des transactions apportés par notre modèle de parallélisation. Cette étude est une première démarche pour la compréhension des aspects liés à la structuration des graphes des transactions et à l'évaluation du coût provenant de la gestion du parallélisme.

Nous avons montré, par l'intermédiaire des diagrammes présentant le comportement dynamique des transactions, que sous certaines conditions de surcharge et caractéristiques des opérations, notre approche apporte des gains significatifs en performances. Ces conditions sont fonction à la fois de l'architecture matérielle sur laquelle on peut exécuter les transactions parallèles et de l'application dont dépend le degré de compatibilité des opérations.

1.4 Organisation de la thèse

Le thèse est organisée de la manière suivante :

- Le chapitre 2 s'intérese aux différentes formes de parallélisme. Nous présentons tout d'abord des architectures matérielles et des concepts logiciels qui servent de base pour l'exécution parallèle de programmes. Dans ce contexte, nous donnons quelques éléments sur la parallélisation des programmes séquentiels, notamment en ce qui concerne la dépendances de données et la parallélisation d'itérations.

Deuxièmement, nous détaillons les approches pour le parallélisme dans les langages à objets. Nous nous intéressons plutôt aux aspects liés à la spécification et à la synchronisation du parallélisme dans ces langages. Enfin, le chapitre présente la gestion de la concurrence et du parallélisme dans les systèmes de bases de données. Dans ce contexte, nous détaillons les différentes formes de parallélisme pour le traitement des requêtes et nous nous intéressons à la gestion des transactions concurrentes.

- Le chapitre 3 est consacré à la présentation de notre modèle pour la parallélisation des transactions séquentielles. Il décrit notre approche formelle pour calculer la relation de compatibilité entre les opérations et montre son utilisation dans la parallélisation des transactions. Nous détaillons nos algorithmes pour construire et transformer un graphe représentant une transaction puis nous présentons les primitives utilisées pour paralléliser et pour synchroniser les opérations au sein de cette transaction.

- Le chapitre 4 est divisé en deux parties : la première décrit O_2Paral, un système qui met en œuvre le modèle de parallélisation. Nous détaillons la structure et le comportement des classes qui implantent ce système et nous présentons son intégration avec le système O_2. La deuxième partie montre notre approche pour la parallélisation des règles du système NAOS. Nous décrivons les éléments pour paralléliser les règles dans des cycles d'exécution et nous présentons les modules ajoutés au système NAOS. (*La notation utilisée dans ce chapitre pour décrire le prototype est présentée en annexe.*)

- Dans le chapitre 5 nous décrivons notre approche pour la parallélisation des applications. Nous analysons en premier lieu la parallélisation dans une contexte où nous ne disposons que des transactions classiques. Ensuite nous prenons comme hypothèse un modèle de transactions emboîtées. Nous montrons alors comment intégrer notre modèle de parallélisation des transactions avec celui des transactions emboîtées de manière à exploiter au maximum le parallélisme dans le cadre des applications.

- Le chapitre 6 présente une étude de performances correspondant aux gains apportés par la parallélisation des transactions. Nous détaillons tout d'abord les aspects qui déterminent les gains en performance apportés par la parallélisation des transactions. Ainsi nous pouvons analyser nos choix concernant la structure des graphes des transactions parallèles et étudier la surcharge résultant de la gestion du parallélisme pendant l'exécution de ces transactions. Puis nous mettons en évidence le problème de l'ordonnancement dans les blocs d'opéra-

tions parallèles. Finalement, nous présentons les gains en performance lorsqu'on prend en compte la parallélisation des applications dans le cadre des transactions emboîtées.

- Le chapitre 7 conclut cette thèse en mettent l'accent sur les apports du travail effectué et en donnant quelques perspectives de recherches futures, en particulier en ce qui concerne les évolutions du modèle de parallélisation sous un gestionnaire d'objets parallèles et la définition d'un modèle de coût capable de donner des éléments pour déterminer le gain en performance des applications apporté par le modèle.

Chapitre 2

L'État de l'Art

2.1 Les Modèles de Programmation Parallèle 11
2.1.1 Parallélisme asynchrone 13
2.1.1.1 Multiprocesseurs à mémoire partagée 13
2.1.1.2 Multiprocesseurs sans mémoire commune 14
2.1.2 Partage des ressources 18
2.1.3 Parallélisation des programmes séquentiels 20
2.1.3.1 Dépendance des données 20
2.1.3.2 Transformation de code 21
2.2 Le parallélisme dans les langages à objets 23
2.2.1 Modèle d'exécution 23
2.2.1.1 Le modèle d'acteurs 23
2.2.1.2 Les objets 25
2.2.1.3 Classe et héritage 26
2.2.2 Concurrence Interne 27
2.2.2.1 Les objets séquentiels 27
2.2.2.2 Les objets quasi-concurrents 28
2.2.2.3 Les objets concurrents 29
2.2.3 Communication et Synchronisation 30
2.2.3.1 L'envoi de message 31
2.2.3.2 L'acceptation de message 32
2.3 Parallélisme dans les systèmes de gestion de bases de données 34
2.3.1 Traitement parallèle des requêtes 35
2.3.1.1 Parallélisme intra-opération 35
2.3.1.2 Parallélisme inter-opérations 41
2.3.2 Transactions et concurrence 43
2.3.2.1 Sérialisation 44
2.3.3 Relations de conflit 46

2.3.3.1 Commutativité 47
2.3.3.2 Dépendances séquentielles 49
2.3.3.3 Recouvrabilité 50
2.3.3.4 Localité associée à la commutativité et à la recouvrabilité 51
2.4 Conclusion 53

L'État de l'Art

Dans ce chapitre, nous nous intéressons aux différentes formes de parallélisme. La section 2.1 présente des architectures matérielles et des concepts logiciels qui servent de base pour l'exécution parallèle de programmes. Dans ce contexte, nous donnons quelques éléments sur la parallélisation des programmes séquentiels, notamment en ce qui concerne la dépendances de données et la parallélisation d'itérations.

La section 2.2 décrit les approches pour le parallélisme dans les langages à objets. Nous nous intéressons plutôt aux aspects liés à la spécification et à la synchronisation du parallélisme dans ces langages.

La section 2.3 présente la gestion de la concurrence et du parallélisme dans les systèmes de bases de données. Dans ce contexte, nous détaillons les différentes formes de parallélisme pour le traitement des requêtes et nous nous intéressons à la gestion des transactions concurrentes.

2.1 Les Modèles de Programmation Parallèle

En général, les architectures des machines parallèles peuvent être divisées en machines pipeline, SIMD (en anglais, "Single Instruction, Multiple Data") et MIMD (en anglais, "Multiple Instruction, Multiple Data")[Fly72, LER92].

Dans les machines *pipelines*, une *instruction* est répartie en autant d'étapes que de processeurs. Les processeurs sont organisés en série comme dans une ligne de montage, par laquelle les données de l'instruction passent de processeurs en processeurs. Ces processeurs sont synchronisés par une seule unité de contrôle, ou *séquenceur*. Chaque processeur effectue son étape de l'instruction et délivre le résultat vers le prochain processeur. Le traitement de l'instruction est terminé en arrivant au bout de la chaîne des processeurs [Per87].

Considérons l'instruction i définie comme $a * b + c - d$ qui doit être exécutée par n tâches. Cette instruction peut être répartie en trois étapes, chacune faisant une opération de i. La figure 2.1 montre l'évolution de l'exécution de i dans les n tâches. R_a et R_b représentent respectivement le résultat de l'opération exécutée dans les processeurs P_a et P_b. Dans le temps T_1, il n'y a que la tâche 1 qui exécute la

première étape de l'instruction i dans le processeur P_a. Dans le temps T_2, deux tâches sont exécutées. P_a exécute la multiplication de i de la tâche 2, tandis que P_b exécute l'addition de i de la tâche 1. Dans le temps 3, trois tâches sont exécutées dans la chaîne de processeurs, où chaque processeur exécute une étape de i.

	T_1	T_2	T_3	...
P_a	$a_1 * b_1$	$a_2 * b_2$	$a_3 * b_3$	...
P_b		$R_a + c_1$	$R_a + c_2$	...
P_c			$R_b - d_1$	...

FIG. 2.1 - *Un exemple d'exécution en pipeline*

Le parallélisme fourni par les machines pipelines est bien adapté au traitement de gros volumes de données ayant la même structure où les opérations peuvent être structurées en sous-opérations. Par exemple, le traitement de matrices dans le calcul scientifique. Dans le contexte de bases des données, on peut exploiter cette forme de parallélisme dans l'exécution des requêtes SQL [DG92]. En général, une requête est transformée en un plan d'exécution composé d'opérations de l'algèbre relationnelle ; chaque opération exécute une partie de la requête et délivre le résultat vers la prochaine opération. Certaines opérations peuvent être enchaînées comme dans une exécution pipeline. Nous détaillons cet aspect dans la section 2.3.

Dans les machines SIMD, la même instruction est exécutée en parallèle sur un grand nombre de données, souvent placées dans un vecteur de registres [Bra93]. Ainsi que pour les machines pipelines, l'ensemble des processeurs est synchronisé par un seul séquenceur. Celui-ci détermine l'instruction qui doit être exécutée en parallèle par tous les processeurs. La parallélisation des programmes séquentiels est une des applications les plus importantes des machines SIMD [LER92]. Les programmes les mieux adaptés pour cette parallélisation sont ceux ayant une structure répétitive sur des données de structure régulière. L'exemple classique est la parallélisation des boucles portant sur des grosses matrices dans les applications de calcul scientifique.

Les machines pipelines et SIMD mettent en œuvre un paradigme de parallélisme *synchrone*. Dans ces machines, un programme parallèle possède une seule *activité de contrôle* (en anglais "thread of control") gérée par le séquenceur central. En général, le parallélisme fournit par ces machines est à grain fin. Ainsi, le parallélisme est exploité dans des tâches spécifiques agissant sur des données ayant des propriétés particulières. Des architectures moins "exotiques" sont fournies par des machines MIMD.

Dans les machines MIMD, plusieurs processeurs peuvent exécuter simultanément différentes instructions sur des données diverses. Ce type de machine ne prévoit pas de

séquenceur unique, chaque processeur opérant de manière indépendante. Par conséquent, il n'y a pas de synchronisation globale pour l'ensemble des processeurs. Le parallélisme est, donc, dit *asynchrone*. Chaque processeur possède sa propre activité de contrôle. Un programme peut être divisé en plusieurs activités parallèles indépendantes. Des techniques de synchronisation explicites sont nécessaires pour faire communiquer ces activités.

La plupart des SGBDs parallèles sont fondés sur des machines MIMD. Notre proposition est également ciblée vers un paradigme de parallélisme asynchrone. Dans la suite, nous détaillons les aspects liés à cette forme de parallélisme aussi bien que les mécanismes utilisés pour la synchronisation des activités parallèles.

2.1.1 Parallélisme asynchrone

Les machines MIMD diffèrent conformément au mode de partage de la mémoire entre les processeurs. On distingue schématiquement les multiprocesseurs à mémoire partagée et les multiprocesseurs sans mémoire commune.

2.1.1.1 Multiprocesseurs à mémoire partagée

Les multiprocesseurs à mémoire partagée représentent l'architecture parallèle MIMD la plus fortement couplée [Kra91]. L'échange des données et la synchronisation des activités parallèles sont faits via la mémoire. Celle-ci peut être accédée directement par les différents processeurs à travers un réseau d'interconnexion (cf. figure 2.2). Ce réseau devient un goulot d'étranglement puisque tout accès en mémoire passe par lui. On est ainsi confronté au problème de limitation du nombre de processeurs : si ce nombre est trop grand, l'utilisation du réseau devient plus critique [Les93]. De plus, le partage de la mémoire pose le problème de la synchronisation des accès aux données par les activités parallèles. Plusieurs techniques de synchronisation sont disponibles [Per87] et nous les rappelons dans la section 2.1.2.

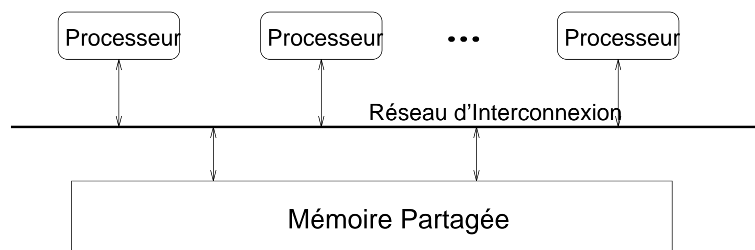


FIG. 2.2 - *MIMD à mémoire partagée*

2.1.1.2 Multiprocesseurs sans mémoire commune

Les multiprocesseurs sans mémoire commune sont constitués d'*unités de calcul*, réunissant processeur et mémoire locale, reliés par un réseau d'interconnexion (cf. figure 2.3). L'échange des données et la synchronisation des activités parallèles sont faits via des envois de message à travers le réseau. Ainsi, chaque processeur accède directement à sa mémoire locale sans passer par le réseau. En revanche, les données éloignées, autrement dit, les données placées dans une mémoire autre que la mémoire locale, ne sont accessibles que par envoi de message. Dans ce type d'architecture, le réseau ne devient pas un goulot d'étranglement car il est moins sollicité que dans les multiprocesseurs à mémoire partagée. Par conséquent, le nombre des processeurs peut être très grand (de l'ordre de 1k processeurs) sans mettre en cause la performance de la machine à cause de la surcharge d'utilisation du réseau. En revanche, l'absence d'espace partagé rend plus difficile la mise en œuvre des mécanismes permettant la synchronisation et la communication des activités parallèles.

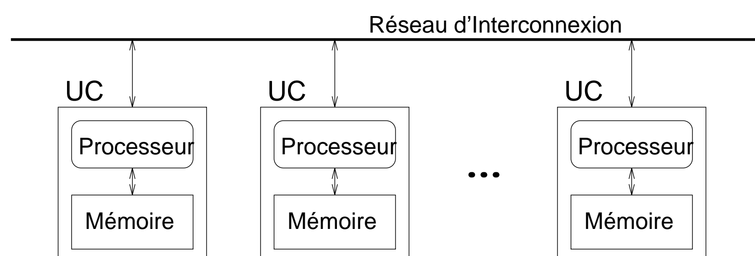


FIG. 2.3 - *MIMD sans mémoire commune*

Indépendemment du mode de partage de la mémoire, le parallélisme dans les machines parallèles asynchrones est souvent exprimé à l'aide des concepts systèmes. Chacun de ces concepts décrit une manière différente d'exploiter le parallélisme dans les divers langages de programmation. A la base de ces concepts, nous trouvons la notion de *processus*, utilisée normalement pour décrire l'exécution d'un programme. Une exécution séquentiel peut être représentée par un seul processus effectuant la suite des instructions définies dans celui-ci. Dans le cas d'un programme parallèle, on peut imaginer plusieurs de ces processus s'exécutant pour réaliser la *tâche* définie dans le programme. Dans la suite, nous décrivons les principaux concepts logiciels utilisés pour exprimer le parallélisme dans les langages de programmation [AS83, BST89, And91b].

- Le premier concept est mis en œuvre par les commandes fork et join. La commande fork permet la création d'un processus à partir d'un autre déjà existant.

Le processus exécutant le fork, appelé "père", crée un nouveau processus, appelé "fils", qui s'exécute en parallèle avec le père (cf. figure 2.4). L'exécution des deux processus est asynchrone. Au moment de la création, le fils est une copie du père. Dans la séquence après le fork, on a la possibilité de "spécialiser" l'exécution de chaque processus de sorte que ceux-ci exécutent deux codes différents. Dans le but de se synchroniser avec la fin du processus fils, le père exécute la commande join. Cette commande met en attente le processus père jusqu'à ce que son fils ait fini.

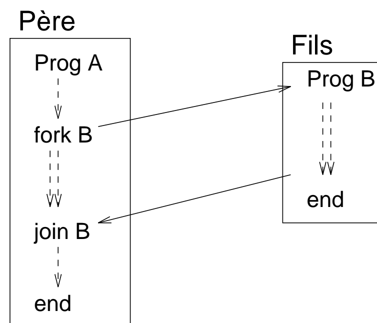


FIG. 2.4 - *Les commandes fork et join*

- Une forme de parallèlisme synchrone a été introduite par les commandes cobegin et coend. Ces commandes permettent l'exécution en parallèle d'un bloc de commandes. Par exemple, la commande

```
cobegin
    procedure1();
    procedure2();
    procedure3();
coend
```

  provoque la création de trois processus pour exécuter les procédures 1, 2, et 3. Le processus exécutant le cobegin (créateur) se synchronise au "coend"avec les trois autres processus. Autrement dit, lors de l'exécution de la commande coend, le créateur est bloqué en attendant que les processus créés au moment du cobegin soient terminés.

- La déclaration des processus dans un langage de programmation permet leur création dynamique à l'aide d'une commande du style fork. Le processus créateur

et le processus créé s'exécutent en parallèle après la création. Contrairement au fork classique, ce qui va être exécuté par le processus fils dépend uniquement du *type de processus* créé et non pas du processus créateur. L'exécution des processus est asynchrone. En général, le partage des données est contrôlé par des *moniteurs* associés à des *variables de conditions*. Dans des langages ayant la possibilité de déclarer des processus comme Argus [Lis88], la déclaration possède une partie dédiée à une étape d'initialisation qui est responsable du déclenchement de l'exécution du nouveau processus. Par exemple, dans le langage Ada, la partie task body est exécutée lors de création d'un processus (appelé *task*) [Ban91] :

```
task T1 is
    entry proc1;
    entry proc2;
    end T1;
task body T1 is
    begin
        loop
            select
                when <cond1> accept proc1;
                when <cond2> accept proc2;
            end select;
        end loop;
    end T1;
```

Un processus P créant un exemplaire de T1 peut ensuite appeler une des "entrées" donnée par la spécification de T1 (proc1 ou proc2). La clause select sert à synchroniser les processus. Si la condition est satisfaite, la commande accept est exécutée mettant P en attente.

- L'appel de procédure à distance (RPC) est un mécanisme qui permet à un processus d'appeler une procédure d'un autre processus. L'appel se fait par une commande call : call nom_de_procedure(paramètres_entrée; paramètres_retour);
Le processus exécutant la commande call est communément appelé *client*, tandis que celui qui exécute la procédure appelé est dit *serveur*. Un appel RPC peut être synchrone ou asynchrone. Dans le cas synchrone, le client est bloqué pendant que le service demandé est effectué. Dans le cas asynchrone, le client continue son exécution après le call, en parallèle avec l'exécution de la procédure demandée au serveur. Le serveur peut être mis en œuvre de différentes

manières. La plus simple utilise un processus qui boucle en exécutant les services demandés (cf. `serveur1` de la figure 2.5). Dans ce cas, les demandes sont traitées séquentiellement. Une autre possibilité, basée sur la création dynamique des processus, traite les demandes en parallèle. Le serveur crée des processus pour chaque procédure appelée (cf. `serveur2` de la figure 2.5). Une approche intermédiaire vise à baisser la surcharge de la création dynamique des processus. Le serveur est donc composé d'un "pool" de processus créés au moment de l'initialisation du serveur. Ces processus sont chargés de réaliser les services demandés au serveur qui sont donc effectués en parallèle par un nombre fixe de processus.

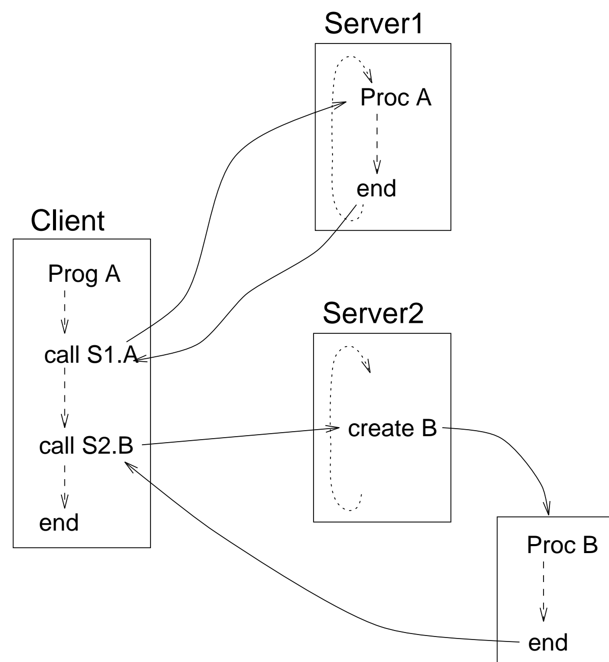


FIG. 2.5 - *L'appel de procédure à distance*

- Certains langages ont adopté une approche différente de celles décrites jusqu'à présent. Il s'agit des langages avec *parallélisme implicite*. Ces langages permettent le calcul parallèle sans avoir d'instruction spécifique pour le décrire. Le programmeur code son application comme si elle allait s'exécuter séquentiellement. Le compilateur effectue une étape de parallélisation du code selon certaines propriétés de l'application et suivant certaines règles de transformation. Le principal avantage de cette approche est le "confort" donné au programmeur puisqu'il n'a pas à gérer et à contrôler le parallélisme. En revanche, le parallélisme obtenu par l'étape de parallélisation est souvent non optimal. Autrement

dit, il peut y avoir du parallélisme à exploiter dans le code de l'application que le compilateur ne trouve pas.

Un modèle à mémoire partagée peut être mis en œuvre à la fois sur des machines à mémoire partagée, mais également sur des machines sans mémoire commune [BST89]. Dans tous les cas, le partage des ressources demande des techniques de synchronisation d'accès aux ressources par les activités parallèles.

2.1.2 Partage des ressources

Les processus parallèles ont besoin de synchroniser leurs actions dans le cas où ils accèdent à des ressources communes. Les régions d'un programme qui manipulent des ressources partagées sont communément appelées *sessions critiques*. Il est évident qu'une région d'un programme est critique par rapport à une région d'un autre uniquement si elles manipulent *la même ressource*. L'approche utilisée pour synchroniser les processus consiste à rendre séquentiel tout accès aux ressources partagées par les processus. Autrement dit, deux processus ne peuvent pas exécuter leurs sessions critiques en parallèle. Cette approche est appelée *exclusion mutuelle*. Les sémaphores et les moniteurs sont les mécanismes les plus utilisés pour la mettre en œuvre [And91a, Tan92] :

- Un sémaphore peut être vu comme un type abstrait de données dont l'état est composé d'un *entier* et d'une *liste de processus*. L'entier est utilisé pour gérer l'entrée en session critique, tandis que la liste contient tous les processus en attente de la libération de la ressource contrôlée par le sémaphore. Deux opérations atomiques, historiquement appelées P et V, sont définies par le sémaphore. P doit précéder l'entrée en session critique, et V doit la suivre :

  ```
  Proc1
  ...
  P(sémaphore);
      < session critique >
  V(sémaphore);
  ...
  ```

  P bloque le processus Proc1 si la valeur de l'entier de l'état du sémaphore, après une décrémentation, est négative et place Proc1 dans la file d'attente du sémaphore. V incrémente l'entier ; si celui n'est pas positif, V "réveille" le premier processus de la file d'attente du sémaphore.

  Malgré leur simplicité, les sémaphores peuvent facilement conduire à des erreurs de programmation quelquefois difficile à trouver. Il suffit de ne pas libérer la

ressource (opération V) après que le processus soit passé en session critique. Cela peut se produire car le contrôle du passage en session critique est réparti dans tous les processus parallèles. Alors, d'autres mécanismes de plus haut niveau d'abstraction comme les moniteurs ont été proposés.

- Un moniteur est également un type abstrait de données qui représente à la fois les ressources qu'il est censé protéger, mais aussi les procédures qui manipulent ces ressources. Contrairement aux sémaphores, la manipulation de ressources est effectuée dans le moniteur et non pas dans les processus qui veulent utiliser les ressources. Un moniteur encapsule un ensemble de valeurs qui représentent l'état d'une ressource accessible uniquement par les procédures qu'il exporte :

```
monitor Monit1
    < déclaration des ressources >
    condition c;
    ...
    procedure proc1();
    ...
    procedure proc2();
    ...
end monitor;
```

Un processus qui veut utiliser la ressource définie dans Monit1 appelle une de ses procédures :

```
Proc1
...
Monit1.proc1();
...
```

Il faut noter qu'il n'y a aucun contrôle d'utilisation de la ressource partagée dans le processus. Cela est fait entièrement dans le moniteur.

Les procédures sont toujours exécutées en exclusion mutuelle. Par conséquent, un seul processus à la fois est actif dans un moniteur. Un moniteur gère également une file de processus en attente de libération de la ressource. Dans notre exemple, si la ressource contrôlée par Monit1 est prise au moment de l'exécution de Monit1.proc1();, le moniteur met Proc1 dans une file d'attente. Lors de la libération de la ressource, le moniteur “réveille” le premier processus de cette file. L'exclusion mutuelle des procédures est souvent mise en œuvre par des

sémaphores. Cela est fait automatiquement par le compilateur en considérant chaque procédure comme une session critique.

Un aspect important est la synchronisation des processus dans le moniteur à travers les opérations wait et signal associées aux *variables de condition*. Un processus qui exécute wait sur une variable de condition est bloqué et libère le moniteur. Il sera réveillé par un autre processus qui exécute signal sur la même variable de condition. Les variables de condition ne sont pas utilisées pour contrôler l'accès à la ressource. Cela est fait par l'exclusion mutuelle des procédures du moniteur, comme nous venons de l'expliquer. Cependant, ces variables permettent d'exprimer des conditions d'attente dépendantes de la ressource.

2.1.3 Parallélisation des programmes séquentiels

En général, la parallélisation des programmes séquentiels est effectuée par le compilateur du langage en se fondant sur certaines caractéristiques des programmes. Le compilateur réalise des *transformations* dans le code du programme séquentiel pour pouvoir générer du code parallèle à exécuter [BGS93].

Dans le contexte des langages impératifs, la plupart des travaux de parallélisation de code portent sur les *boucles* [LKK85, AK87, PW86, LER92]. La raison en est simple : les boucles sont des structures répétitives dont l'exécution coûte chère du point du vue de la performance d'un programme. Si l'on peut exécuter en parallèle les n itérations d'une boucle, on gagne, en principe, en performance.

2.1.3.1 Dépendance des données

Il est évident que la transformation du code d'un programme doit assurer la cohérence sémantique de celui-ci. Autrement dit, pour toute exécution correcte, les résultats obtenus par l'exécution séquentielle et par celle parallèle doivent être équivalents. Dans le cas des boucles, les techniques de parallélisation utilisent la *relation de dépendance des données* entre les commandes pour assurer la cohérence du programme [AKPW83, FOW87, WB87]. À chaque instruction c, on associe deux ensembles :

- IN(c) contient toutes les variables dont les valeurs sont *lues* par l'instruction c.
- OUT(c) contient toutes les variables dont les valeurs sont *modifiées* par l'instruction c.

On définit également un *ordre d'exécution* communément désigné par Θ. $c_i \Theta c_j$ détermine que c_i doit être exécutée *avant* c_j.

Par exemple, dans la boucle :

```
for i = 1 to n do
(c1) A[i] = C[i];
(c2) B[i] = A[i];
end
```

il y a un ordre d'exécution entre c_1 et c_2 ($c_1 \, \Theta \, c_2$). Les ensembles associés à ces instructions sont :

$\mathsf{IN}(c_1) = \{ \mathsf{C[i]} \}$
$\mathsf{OUT}(c_1) = \{ \mathsf{A[i]} \}$
$\mathsf{IN}(c_2) = \{ \mathsf{A[i]} \}$
$\mathsf{OUT}(c_2) = \{ \mathsf{B[i]} \}$

Une instruction c_j dépend d'une instruction c_i, si (1) c_i et c_j accèdent à une variable v et, au moins, une des instructions modifie la valeur de v ; (2) dans une exécution séquentielle du programme, c_i est exécutée avant c_j. La condition (1) peut être vérifiée à l'aide des ensembles IN et OUT associés aux instructions. Pour la condition (2), on utilise l'ordre d'exécution Θ. On peut exprimer la relation de dépendance des données δ par la formule :

$$c_i \, \delta \, c_j \Rightarrow (c_i \, \Theta \, c_j \wedge \quad \begin{array}{l} \mathsf{OUT}(c_i) \cap \mathsf{IN}(c_j) \neq \emptyset \vee \\ \mathsf{IN}(c_i) \cap \mathsf{OUT}(c_j) \neq \emptyset \vee \\ \mathsf{OUT}(c_i) \cap \mathsf{OUT}(c_j) \neq \emptyset) \end{array}$$

Dans notre exemple, c_1 dépend de c_2 ($c_1 \, \delta \, c_2$ est vrai) car $c_1 \, \Theta \, c_2$ et $\mathsf{OUT}(c_1) \cap \mathsf{IN}(c_2) \neq \emptyset$.

2.1.3.2 Transformation de code

En général, on distingue deux méthodes de transformation des boucles selon l'architecture de la machine parallèle [Bra93] (cf. section 2.1.1). Dans le cas des machines synchrones, on utilise une méthode de *vectorisation* du code de la boucle. Lorsque la machine parallèle est asynchrone, on utilise plutôt des méthodes de *parallélisation* du code de la boucle.

La méthode consiste à diriger chaque itération d'une boucle vers un processeur différent. Pour une boucle avec n itérations, on aurait n processus exécutant chacun une des itérations sur un processeur (cf. figure 2.6). Lorsqu'il n'y a pas autant de processeurs que d'itérations à réaliser, on crée un nombre de processus correspondant au nombre des processeurs et on effectue des groupements des itérations dans les processus. Ainsi, chaque processus exécute un nombre m d'itérations avec $m < n$.

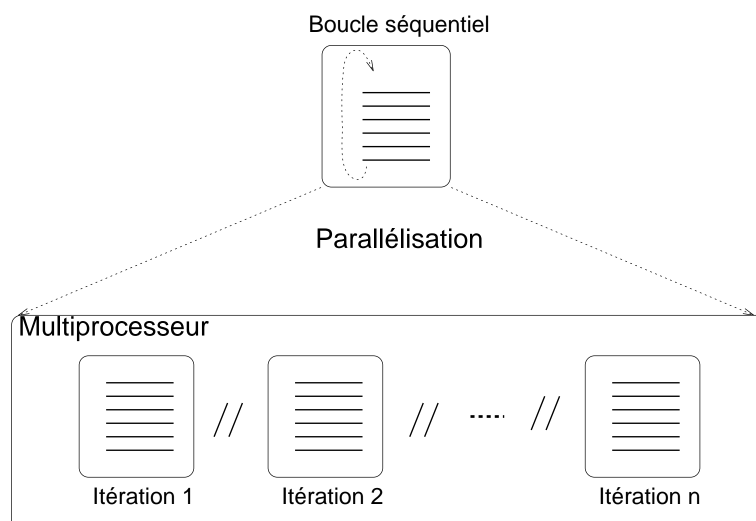


FIG. 2.6 - *La méthode de parallélisation des boucles*

La parallélisation exécute donc les instructions de chaque itération de la boucle de manière séquentielle. On peut noter que l'ordre d'exécution des commandes est maintenu. Par conséquent, la dépendance des données dans une itération n'est pas considérée. En revanche, la dépendance des données entre les commandes de deux itérations différentes est cruciale pour la parallélisation. Revenons à notre exemple. Une première itération de la boucle pourrait être :

```
A[1] = C[1];
B[1] = A[1];
```

Cette itération peut être exécutée en parallèle avec la prochaine itération :

```
A[2] = C[2];
B[2] = A[2];
```

et ainsi de suite. On voit qu'il n'y pas de dépendance des données entre les commandes de deux itérations. Par contre, lorsqu'on considère la boucle suivante :

```
for i = 1 to n do
(c1) A[i] = B[i] + C[i];
(c2) D[i] = A[i] + E[i-1];
(c3) E[i] = C[i] * 2;
end
```

On voit qu'il existe une dépendance entre c_2 et c_3 sur la variable E[i] qui dépasse les limites d'une itération. Comme résultat, on a une dépendance entre deux commandes

de deux itérations différentes. Cela empêche la parallélisation ou, à la limite, demande l'utilisation des techniques de contrôle de partage de données telles que le sémaphore [1].

2.2 Le parallélisme dans les langages à objets

Cette section présente les approches pour le parallélisme dans les langages de programmation à objets (LPOO) concurrents [Pap92, TNW91, WKH92, YT87]. Dans ces langages, nous nous intéressons au *modèle d'exécution*, surtout aux aspects liés à l'état passif ou actif des objets pendant l'exécution. Nous comparons également les langages selon la *concurrence* fournie entre objets. Nous décrivons plus particulièrement les mécanismes de *communication* et de *synchronisation* entre les objets actifs parallèles.

2.2.1 Modèle d'exécution

La plupart des modèles d'exécution des langages à objets concurrents sont basés sur le modèle d'acteurs. Nous commençons par une étude de ce modèle. Ensuite nous analysons le "comportement" des objets par rapport au modèle d'exécution. La possibilité de création dynamique des objets et le partage des tâches à exécuter sont finalement étudiés à l'aide des systèmes de types des langages.

2.2.1.1 Le modèle d'acteurs

L'abstraction d'acteur utilise le mécanisme d'échange de messages comme base pour l'exécution parallèle [Agh86, MNC+89]. Ce modèle respecte le principe d'encapsulation défini dans les approches à objets en y ajoutant celui d'activité. Les acteurs sont des entités actives capables de réagir à une demande de service par leurs partenaires. L'encapsulation est garantie par une interface fonctionnelle. Les actions réalisées par chaque acteur sont indépendantes et le seul moyen d'échange d'information entre les acteurs est l'envoi de message. Les acteurs communiquent entre eux pour se confier ou se déléguer des tâches et se transmettre des résultats. Ils sont composés de deux parties :

- Les attributs qui sont des données locales et correspondent à l'état des acteurs. Ils servent aussi à stocker l'adresse des boîtes aux lettres des autres acteurs

1. En fait, certaines méthodes de parallélisation transforment le code de la boucle même en cas de dépendance entre itérations. Outre la création des processus parallèles pour exécuter les itérations, ces méthodes introduisent des sémaphores dans le code afin de synchroniser l'accès aux variables cause de la dépendance.

directement connus. Les attributs sont encapsulées par les fonctions du comportement.

- Un comportement qui définit l'ensemble des actions pouvant être exécutées par l'acteur. Nous trouvons ici la notion classique d'abstraction de données lorsque un acteur est défini par son comportement et non par sa représentation physique.

Un acteur est une entité autonome. Il possède toute l'information nécessaire pour effectuer les services demandés. En traitant un message, un acteur peut communiquer avec d'autres acteurs dont il connaît l'adresse ; il peut créer de nouveaux acteurs ; il peut aussi spécifier un *comportement de remplacement* pour traiter des nouveaux messages qui arrivent dans sa boîte aux lettres (cf. figure 2.7). Après le remplacement, un acteur est capable d'effectuer immédiatement la tâche demandée par le prochain message. Par conséquent, l'acteur remplacé et l'acteur remplaçant exécutent des tâches en parallèle. Le processus de remplacement est intrinsèquement concurrent [Agh86].

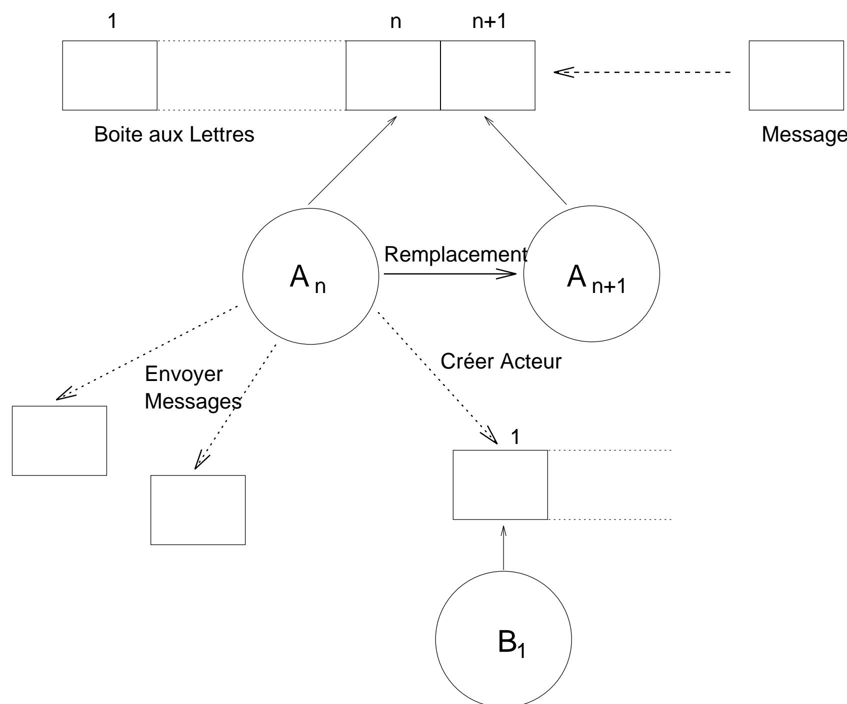


FIG. 2.7 - *Comportement d'un acteur.*

Les acteurs communiquent les uns avec les autres à travers un mécanisme asynchrone d'envoi de messages. L'acteur qui envoie des messages est appelé *émetteur* alors que celui qui en reçoit est appelé *cible*. Lorsque l'émetteur envoie un message, il n'a pas besoin d'attendre la réponse pour continuer son calcul. Le résultat du calcul

démarré par le message peut-être envoyé vers un troisième acteur (autre que l'émetteur) en utilisant le mécanisme de *continuation*. Ce mécanisme permet à l'émetteur de désigner l'acteur auquel doit être transmis le résultat du message.

Au cours de son existence, un acteur peut recevoir plusieurs messages provenant de divers acteurs et demandant des services différents. Par rapport au traitement des messages qui arrivent, le modèle définit deux types d'acteurs :

- Les *acteurs sérialisés* sont capables de changer leur état interne, c'est à dire la valeur des attributs, et doivent donc traiter les messages un par un. Les messages qui arrivent dans l'intervalle sont mis en file d'attente dans la boîte aux lettres de l'acteur.

- Les *acteurs non sérialisés* ne peuvent pas changer leur état interne. Les messages qui arrivent vers ce type d'acteur peuvent être traités en parallèle.

Le parallélisme obtenu par des acteurs non sérialisés est dit *intra-acteur*. Dans un autre niveau, le parallélisme *inter-acteurs* est caractérisé par l'exécution concurrente de plusieurs acteurs.

2.2.1.2 Les objets

Dans la plupart des systèmes à objets un calcul est provoqué par un envoi de message vers un objet pour exécuter une méthode. Cet objet peut ensuite envoyer des messages vers d'autres objets afin d'exécuter d'autres méthodes. Les messages sont envoyés de façon synchrone et le calcul se termine lorsque la méthode initiale a fini son exécution. Dans ces systèmes, les objets sont dits *passifs* dans la mesure où ils doivent recevoir un message pour réaliser une action.

D'autres approches étendent cette vision des objets en introduisant la notion d'objet *actif*. Un objet actif peut déclencher des méthodes de manière concurrente et réaliser des actions de façon asynchrone, sans avoir besoin de recevoir de message le demandant. Ces approches utilisent toujours du *parallélisme* afin d'augmenter les performances du système.

Les langages à objets concurrents peuvent être classés selon les types d'objets qu'ils offrent :

- *L'approche hétérogène*: Les objets passifs et actifs sont supportés. Citons, par exemple, ACT++ [KL90] et Eiffel Parallèle [Car89]. Le langage ACT++ étend le langage C++ avec un ensemble de classes où les instances sont des acteurs. Les objets passifs sont des objets C++ ordinaires. Un acteur contient plusieurs objets passifs, mais un objet passif est connu d'un seul acteur.

- *L'approche homogène*: Les objets sont tous actifs, tout comme dans le modèle d'acteurs. Les langages ABCL/1 [YBS86], Hybrid [Nie87], POOL [Ame87] et Sina [AT88] en sont des exemples. Dans ces langages, un objet est une entité ayant des données et des procédures qui agissent sur ces données. Le seul moyen de communication entre les objets est l'envoi de message.

2.2.1.3 Classe et héritage

Certains langages à objets concurrents possèdent les concepts de classes et de hiérarchies des classes avec héritage. D'autres offrent plutôt des abstractions des données à travers les types abstraits de données (TAD). Dans ce cas, le langage ne possède pas le concept de sous-typage ni d'héritage entre les types. En revanche, certains langages offrent le mécanisme de *délégation* de tâches qui peut être vu comme une sorte d'héritage [Ste87]. Ce mécanisme est utilisé par un objet pour retransmettre un message qu'il ne sait pas traiter — le message ne coïncide pas avec une de ses méthodes. Dans ce cas, l'objet délègue le service à un autre objet, appelé *proxy*, qui est capable de traiter le message.

Les langages ACT++, Hybrid et Eiffel Parallèle proposent les concepts de classe et d'héritage. Dans ACT++, les classes sont réparties en classes d'objets passifs et classes d'acteurs. Une classe d'acteur hérite, directement ou indirectement, d'une classe particulière qui contient la définition des données et des opérations nécessaires au contrôle de l'exécution d'un objet actif. Les objets actifs du langage Eiffel Parallèle sont forcément instances d'une classe qui hérite d'une autre classe pré-définie ayant la possibilité de créer des processus.

```
class boundedBuffer : Actor
{     inArray  buff[MAX];
      in       in, out;
      ...
      public:
      buffer()
      {        /* initialisation */
               ...
               become emptyBuffer;
      }
      void put ( in item )
      ...
}
```

FIG. 2.8 - *Définition d'une classe d'acteurs ACT++.*

Le langage ABCL/1 ne possède pas le concept de classe. Par contre un objet peut créer d'autres objets ayant les mêmes caractéristiques que lui : un tel objet peut être vu comme un *prototype* d'objets. ABCL/1 offre également la délégation de tâches. Dans les langages POOL et Sina, un TAD est formé de deux parties : interface et local. La partie interface définit les méthodes visibles à l'extérieur du TAD. La partie local contient les variables internes, la réalisation des méthodes et une procédure spéciale chargée de faire l'initialisation (des objets) du TAD.

La figure 2.8 présente la définition ACT++ d'une classe d'objet actif. En particulier, on peut remarquer l'héritage de la classe Actor déterminée dans la première ligne de la définition. A chaque fois qu'on crée une instance de boundedBuffer, la procédure d'initialisation, appelée buffer, est exécutée, rendant l'objet actif.

Le tableau suivant présente un résumé des différents aspects des modèles des objets associés aux langages étudiés.

Les Langages	*Les Objets*		*Le Typage*	
	Homogène	*Hétérogene*	*Classe*	*Type Abstrait*
ABCL/1 [a]	√			
ACT++		√	√	
Eiffel Parallèle		√	√	
Hybrid	√		√	
POOL	√			√
Sina	√			√

[a] ABCL/1 est un langage non-typé

2.2.2 Concurrence Interne

Du point de vue de l'exécution, les objets actifs possèdent des points d'entrée déterminés par l'interface fonctionnelle et une ou plusieurs *activités*(de l'anglais "threads of control") qui peuvent être actives ou bloquées. Chaque activité est représentée par une structure de données qui contient des informations sur le contrôle et sur l'état de l'exécution. Selon la classification donnée par [Weg87], les objets actifs peuvent être séquentiels, quasi-concurrents et concurrents.

2.2.2.1 Les objets séquentiels

Les objets actifs séquentiels possèdent une seule activité de contrôle (figure 2.9). C'est sont les objets sérialisés du modèle d'acteurs. En général, ces objets ont des files d'attente où sont mis les messages à traiter. Ils les traitent les uns après les autres selon certaines instructions de synchronisation. Les objets du langage POOL,

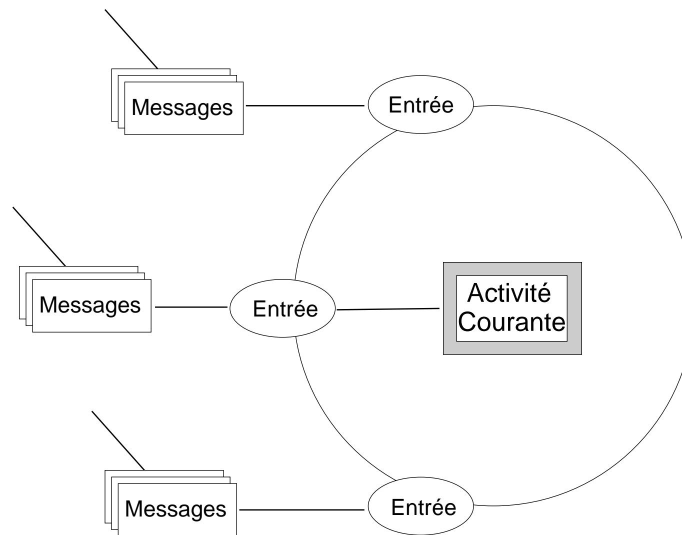


FIG. 2.9 - *Objet actif séquentiel.*

les objets actifs du langage Eiffel Parallèle, et les objets sérialisés de ABCL/1 sont séquentiels.

Dans POOL aussi bien que dans Eiffel Parallèle, un objet actif possède plusieurs objets passifs mais un seul peut exécuter une méthode à la fois. Ainsi dans un objet actif, les méthodes des objets passifs s'exécutent toujours de manière séquentielle. En plus, un objet passif est entièrement encapsulé dans un objet actif. Dans les deux langages, la source de parallélisme est l'exécution concurrente de plusieurs objets actifs. Comme les objets actifs peuvent être créés dynamiquement, à un instant donné, une application aura plusieurs objets actifs chacun réalisant une partie de la tâche globale.

2.2.2.2 Les objets quasi-concurrents

Les objets actifs quasi-concurrents possèdent plusieurs activités mais une seule est active à la fois (cf. figure 2.10). En plus des files de messages à traiter, ces objets ont également des files d'activités bloquées au sein de l'objet. Lorsqu'une activité se termine ou devient bloquée, l'objet a le choix entre réveiller une activité bloquée ou traiter un message. Ce choix est déterminé par des instructions de synchronisation du langage. Les *domaines* du langage Hybrid en sont des exemples.

Dans Hybrid, les objets actifs sont séquentiels. Cependant ces objets sont rassemblés en domaines, qui constituent en quelque sorte les unités du parallélisme. En effet, un domaine est équivalent à un processus, ayant un ensemble d'objets actifs. Bien qu'un domaine possède plusieurs objets actifs, un seul peut être actif à la fois

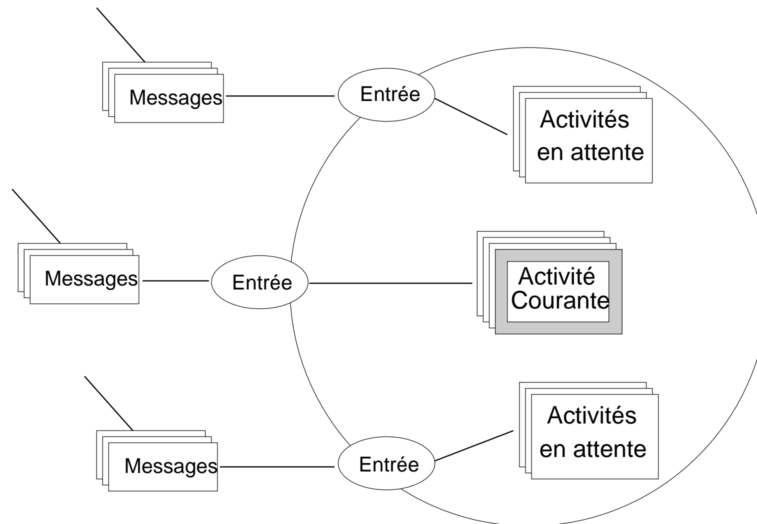


FIG. 2.10 - *Objet actif quasi-concurrent.*

pour traiter un message. La différence entre un domaine et un objet actif de l'approche séquentielle est due au fait qu'un domaine peut traiter un nouveau message lorsque son activité est bloquée. Dans le cas des objets actifs séquentiels, l'activité courante de l'objet actif doit se terminer, pour qu'il puisse traiter un nouveau message.

2.2.2.3 Les objets concurrents

Les objets actifs concurrents peuvent avoir plusieurs activités à la fois (cf. figure 2.11). Ces objets peuvent exécuter en parallèle des méthodes demandées par les messages qu'ils reçoivent. Ainsi le parallélisme est possible à deux niveaux, par l'exécution parallèle de plusieurs objets actifs et par l'exécution parallèle des méthodes au sein des objets. Le modèle d'exécution est donc très proche de celui du modèle d'acteurs. Les langages ACT++ et Sina fournissent des objets concurrents. Les objets ABCL/1 non-sérialisés et les *guardians* du langage Argus [Lis88] sont aussi des exemples.

Dans Sina, l'instruction `detach` offre la possibilité de traiter les messages de manière concurrente. Lorsqu'elle est utilisée dans le corps une méthode, l'objet peut traiter un nouveau message pendant que l'exécution de la méthode originale continue. Dans le langage ACT++, c'est l'instruction `become` qui offre la même fonctionnalité. Ces instructions modélisent le *comportement de remplacement* vu dans le modèle d'acteurs.

Dans les approches precédentes, objets séquentiels et quasi-concurrents, il n'y avait pas de problème de partage des données car une seule activité était active à la fois. Dans les objets actifs concurrents, ce problème doit être étudié puisque les activités

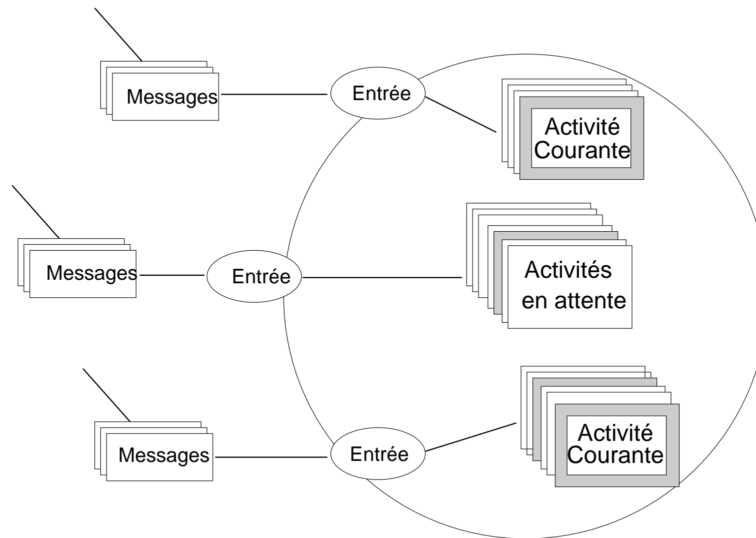


FIG. 2.11 - *Objet actif concurrent.*

parallèles partagent l'état des objets. En général, des mécanismes tels que ceux que nous avons montrés dans la section 2.1.2 sont utilisés pour contrôler le partage des données. Ce problème ne se pose pas pour les objets ABCL/1 non-sérialisés qui, par définition, ne possèdent pas d'état.

La concurrence interne aux objets des langages est résumée dans le tableau ci-dessous.

Les Langages	*La Concurrence Interne*		
	Séquentiel	*Quasi-Concurrent*	*Concurrent*
ABCL/1	√		√
ACT++			√
Eiffel Parallèle	√		
Hybrid		√	
POOL	√		
Sina			√

2.2.3 Communication et Synchronisation

La communication asynchrone entre plusieurs objets actifs est à la base de l'introduction du parallélisme dans la plupart des langages étudiés ici. Un échange de messages entre deux objets actifs concerne un objet envoyant le message (l'objet émetteur) et un autre recevant ce message (l'objet cible). Les langages offrent des formes diverses de communication.

2.2.3.1 L'envoi de message

Du point de vue de l'émetteur, les langages offrent trois types des messages (cf. figure 2.12). Ces types sont différents selon le point de vue de la synchronisation entre l'émetteur et la cible.

- *Message synchrone*: l'émetteur d'un message synchrone attend que l'objet lui retourne le résultat avant de reprendre son activité.

- *Message asynchrone*: à la suite d'un envoi de message asynchrone, l'émetteur n'attend pas que le message soit reçu et traité par la cible pour continuer son activité. En général, une des deux situations suivantes se produit :

  - Le message ne produit aucune réponse, donc l'activité de l'émetteur devient indépendante de l'exécution de la méthode appelée.
  - L'émetteur se sert d'un mécanisme de continuation, (cf. section 2.2.1.1) pour déterminer l'objet auquel doit être envoyé le résultat de l'exécution de la méthode appelée.

- *Message anticipé*: ce type de message permet à l'émetteur d'envoyer un message dont il n'utilisera le résultat qu'ultérieurement. L'émetteur peut continuer son activité jusqu'au moment où il a effectivement besoin du résultat. Si le résultat n'est pas encore disponible, l'émetteur se met en attente.

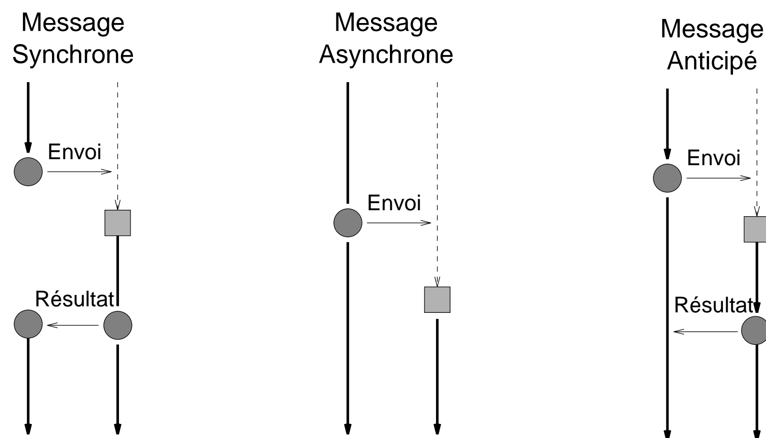


FIG. 2.12 - *Les Types de Messages.*

La plupart des langages vus jusqu'à présent possèdent des messages synchrones. En particulier, les objets passifs dans ACT++ et dans Eiffel Parallèle n'échangent que des messages de ce type. De même pour les objets (actifs) de Sina et de POOL.

Les messages synchrones du langage Hybrid sont non-bloquants par rapport au domaine de l'objet qui les envoie. Ces messages permettent à un domaine de réveiller un autre objet pendant que l'émetteur attend la réponse. Lorsque le résultat lui est retourné, il peut reprendre l'activité interrompue. Du point de vue des objets émetteurs, les messages sont tous synchrones.

Dans ACT++, les messages asynchrones sont divisés en *messages de demande* et *messages de réponse*. Ces messages sont supportés par des objets spéciaux appelés *boîte aux lettres* (en anglais *mail boxes*). Le langage distingue deux types de boîte aux lettres : les *Mbox* dédiés aux messages de demande et les *Cbox* supportant ceux de réponse. Dans une communication, l'objet émetteur envoie le message vers le *Mbox* attaché à l'objet cible. La cible de la communication envoie le message de réponse, s'il y en a un, vers le *Cbox* attaché à l'objet émetteur. L'émetteur peut donc obtenir le résultat de la communication du *Cbox* qui a reçu la réponse. Les messages de demande et de réponse sont envoyés en utilisant des instructions non-blocantes. En revanche, lorsqu'un émetteur veut obtenir le résultat d'un *Cbox*, il bloque sont exécution. Ce mécanisme de communication permet la mise en œuvre des trois types des messages décrits ci-dessus.

Le langage ABCL/1 fournit également les messages synchrones, asynchrones et anticipés. L'objet qui reçoit la réponse dans un message synchrone ou dans un message anticipé est toujours celui qui a émis le message original. Par contre dans un message asynchrone l'objet n'est probablement pas intéressé par une réponse, mais il peut demander à l'objet cible, d'envoyer le résultat vers un troisième objet, appelé "reply destination", à travers le mécanisme de *continuation*.

2.2.3.2 L'acceptation de message

Du point de vue de l'objet cible de la communication, l'élément d'analyse est sa capacité à accepter des demandes conditionnelles de services. Lorsqu'un message arrive, un objet cible peut se trouver dans un des trois états suivants : *actif* traitant un message préalable, *bloqué* en attentant qu'une pré-condition soit remplie ou *dormant* prêt à traiter un nouveau message. Les approches pour l'acceptation conditionnelle des messages sont les suivantes :

- *Acceptation explicitée* : le traitement des messages est synchronisé avec une opération du type "accept" exécutée explicitement par l'objet cible. C'est l'approche des langages POOL et ABCL/1.

- *Conditions d'activation* : des conditions implicites ou explicites dans l'état de l'objet cible déterminent quand un message peut être traité. Les exemples sont les langages ACT++, Hybrid et Sina.

Dans ABCL/1, lorsque l'instruction select est exécutée, l'objet entre dans l'état *en attente* de recevoir le message déclaré dans cette instruction. Cette instruction peut être utilisée dans les corps des méthodes. L'instruction answer du langage POOL possède la même sémantique. En revanche, cette instruction ne peut être trouvée que dans la partie dédiée à l'initialisation des objets POOL.

La figure 2.13 complète la définition de la classe ACT++ boundedBuffer initialement montrée dans la section 2.2.1.3. En particulier, notez les clauses behavior et become qui servent à introduire de la synchronisation dans la définition de la classe. La clause behavior indique les états dans lesquels un acteur peut se trouver. Pour chaque état, on détermine les méthodes qui peuvent être exécutées. Lors de l'arrivée d'un message, un objet déclenche la méthode correspondante si son état actuel le permet. Dans le cas contraire, le message est placé dans une file d'attente.

```
class boundedBuffer : Actor
{       inArray  buff[MAX];
        in       in, out;
        behavior:
                 emptyBuffer = { put() }
                 fullBuffer = { get() }
                 partialBuffer = { put(), get() }
        public:
        buffer()
        {        initialisation
                 ...
                 become emptyBuffer;
        }
        void put( in item )
        {        insert(item);
                 if (full(buff))    become fullBuffer;
                 else               become partialBuffer;
        }
        ...
}
```

FIG. 2.13 - *La synchronisation d'un acteur ACT++.*

La clause become, qui apparaît dans le code des méthodes, modifie l'état de l'objet dans le but de synchroniser l'exécution des méthodes selon la clause behavior. Par exemple, lorsque l'état de l'objet devient partialBuffer à travers l'exécution de la méthode put(), il peut par la suite exécuter à la fois d'autres méthodes put(), mais aussi des méthodes get().

Les tableaux suivants résument les aspects liés à la communication et à la synchronisation des messages.

Les Langages	*Communication et Synchronisation - l'émetteur* [a]		
	Mes. Synchrone	*Mes. Asynchrone*	*Mes. Anticipé*
ABCL/1	√	√	√
ACT++	√	√	√
Eiffel Parallèle		√	
Hybrid	√		
POOL	√		
Sina	√		

[a] Nous considérons uniquement la communication entre les objets actifs

Les Langages	*Communication et Synchronisation - la cible*	
	Acceptation Explicitée	*Conditions d'Activation*
ABCL/1	√	
ACT++		√
Eiffel Parallèle	√	
Hybrid		√
POOL	√	
Sina		√

2.3 Parallélisme dans les systèmes de gestion de bases de données

L'exploitation du parallélisme dans les SGBDs apparaît à la fois dans l'exécution concurrente de plusieurs transactions, mais aussi dans l'exécution parallèle d'une requête SQL. Les transactions concurrentes sont offertes par tous les systèmes multi-utilisateurs en se basant sur des mécanismes de contrôle d'accès concurrents aux données [BHG87]. Nous détaillons ces aspects dans la section 2.3.2.

Des nombreux prototypes de recherches proposent l'exploitation du parallélisme dans le traitement d'une requête SQL [DG90, Omi95]. La plupart de ces prototypes est construite sur des machines MIMD (cf. section 2.1). Les systèmes Bubba [BAC+90], Gamma [DGS+90] et SDC [KO90] possèdent une architecture de base sans mémoire commune, tandis que les systèmes DBS3 [BCL93], Volcano [Gra94] et XPRS [SKPO88] sont bâtis sur des machines à mémoire partagée.

Le type de la machine sur laquelle est installé le SGBD influe dans le placement des données et dans la possibilité d'extension du système (en anglais *scalability*). Dans les systèmes à mémoire partagée les données sont accessibles directement par les processeurs. Cela simplifie certaines techniques de parallélisation en éliminant la redistribution des données avant l'exécution des opérateurs de l'algèbre relationnelle. En revanche, les systèmes à mémoire partagée ne peuvent par atteindre des milliers, voire des centaines de processeurs, à cause du goulot d'étranglement représenté par le réseau d'interconnexion. Ainsi la plupart de ces systèmes sont limités à des machines ayant des dizaines des processeurs.

Dans les systèmes sans mémoire commune, les données sont réparties et doivent être échangées avant d'exécuter, par exemple, un produit sur deux relations placées sur des unités de calcul différentes. Une fois effectuée la redistribution des données, chaque unité de calcul exécute en parallèle indépendemment des autres. On obtient donc un haut degré de parallélisme. De plus, ces systèmes peuvent, en théorie, augmenter le nombre de processeurs de façon infinie.

Dans les deux prochaines sections, nous allons détailler les principales techniques utilisées dans ces systèmes pour exploiter le parallélisme dans le traitement des requêtes.

2.3.1 Traitement parallèle des requêtes

L'exécution parallèle permet, à un SGBD, de réduire le temps de réponse pour une requête donnée. Dans une requête, on introduit le parallélisme à deux niveaux : intra-opération et inter-opérations.

2.3.1.1 Parallélisme intra-opération

Le principe de base pour la parallélisation d'une opération de l'algèbre relationnelle consiste, dans un premier temps, à partager les relations dans des fragments et, puis à exécuter, en parallèle, l'opération sur chaque fragment. Les relations sont horizontalement divisées dans des fragments qui sont ensuite placés dans plusieurs unités de calcul (UC). Chaque fragment peut être manipulé de manière indépendante des autres fragments. Lors de l'exécution, l'opération est diffusée vers les UCs ayant un fragment de la relation. Ensuite, l'opération est menée en parallèle sur chaque fragment. Finalement, les résultats de l'opération dans chaque UC sont composés pour donner le résultat global de l'opération.

On distingue généralement deux aspects importants dans la distribution des n-uplets des relations :

1. Le schéma de distribution. On propose essentiellement trois schémas : distribu-

tion circulaire, intervalle des valeurs et hachage [DG90, Omi95].

2. Le degré de distribution détermine le nombre de UCs sur lesquels les fragments sont placés. Deux approches sont proposées dans la littérature : la distribution totale et la distribution partielle [LKB87, OV91, CABK88, GD90a].

Ces aspects sont plus ou moins combinés dans différentes approches pour la fragmentation des relations. Le but de chaque approche est d'équilibrer à la fois l'occupation des données dans les disques, mais aussi la charge de chaque UC dans l'exécution parallèle d'une opération.

Les schémas de distribution des données

La distribution circulaire partage les n-uplets d'une relation dans n disques de façon à ce que le i-ème n-uplet soit placé dans le (i mod n)ième disque. Dans le cas où le nombre de n-uplets est multiple du nombre de disques, on obtient une division uniforme de la relation. Par conséquent, cette approche fournit une bonne distribution des données sur les disques. Elle fournit également un excellent équilibre des charges d'entrée/sortie entre les UCs lors d'un parcours séquentiel de toute la relation. En revanche, si l'opération possède une condition de recherche, telle que <attribut> = <valeur> dans une sélection, on peut être amené à faire des entrées/sorties non nécessaires à des disques ayant des fragments dont les n-uplets ne satisfont pas la condition.

L'approche intervalle des valeurs a été donc proposée justement dans le but d'augmenter la sélectivité des fragments dans l'exécution des opérations ayant des conditions de recherche. Dans cette approche, les fragments sont déterminés selon un attribut de distribution. Le domaine de cet attribut est divisé en autant d'intervalles que de fragments désirés. Par exemple, si l'attribut de distribution est du type Texte et si la relation doit être placée dans trois disques, alors on peut diviser les valeurs dans les intervalles [A-I], [J-R] et [S-Z], un pour chaque fragment. Les n-uplets sont ensuite placés dans les fragments tel que la valeur de l'attribut se trouve dans l'intervalle associé au fragment.

Dans l'approche intervalle de valeurs, on peut affecter à un sous-ensemble des UCs, une opération ayant une condition sur l'attribut de distribution. On obtient ainsi une concentration de l'ensemble des n-uplets satisfaisant la condition. En revanche, selon les intervalles choisis, cette concentration peut être non-uniforme dans le sous-ensemble des UCs. Dans ce cas, la charge due à l'exécution de l'opération est déséquilibrée, car la plus grande part de la tâche est réaliser par peu d'UCs.

L'approche de la distribution par hachage consiste à diviser les n-uplets d'une relation selon le résultat d'une fonction appliquée à l'attribut de distribution. Ce résultat

donne le numéro du fragment dans lequel le n-uplet doit être placé, et par conséquent, l'UC. La même fonction est ensuite utilisée dans la diffusion d'une opération pour la diriger vers une UC particulière. Cette approche est bien adaptée aux opérations dont la condition de recherche porte sur l'attribut de distribution. Cependant elle ne présente pas de bonnes performances pour les opérations ayant des conditions de recherche du type intervalle de valeurs puisque l'opération doit être diffusée vers tous les UCs qui possèdent un fragment de la relation [GD90b].

Les trois schémas de distribution que nous venons de décrire sont proposés dans les systèmes Bubba [CABK88], Gamma [GD90a] et Arbre [LDH+89]. Dans Gamma, la distribution des relations est dite *hybride* car d'autres informations sont prises en compte pour placer une relation. Parmi ces informations, nous pouvons citer notamment l'utilisation des *échantillons* des relations dans le but de mieux distribuer ces relations [DNSS92]. Dans Bubba, la distribution par intervalle des valeurs prend en compte la fréquence d'accès aux données des relations. L'approche place les n-uplets fréquemment accédés dans plus de disques dans le but de mieux équilibrer la charge entre les UCs au moment de l'exécution parallèle d'une opération [CABK88].

Le degré de distribution des données

La distribution totale consiste à répartir chaque relation en autant de fragments que d'UCs et ensuite placer chaque fragment dans le disque de chaque UC. Dans un système avec n UCs, les relations sont donc distribuées en n fragments. Cette approche, adoptée dans Gamma [DGS+90], a l'avantage de fournir un grand nombre de fragments pour une relation, ce qui peut augmenter le parallélisme intra-opération. En revanche, dans certains cas, la surcharge d'initialisation d'une tâche dans chaque UC devient trop lourde par rapport au gain de l'exécution parallèle de l'opération [DGS88].

La distribution partielle partage chaque relation dans un nombre de fragments plus petit que le nombre de UCs. Ainsi chaque relation est placée dans un sous-ensemble d'UCs du système. Intuitivement, on obtient moins de parallélisme intra-opération car il y aura moins de fragments par relation. Par contre, le coût d'initialisation d'une tâche avec des activités parallèles est moins important car il y aura moins d'activités à initialiser. La difficulté de cette approche réside dans la détermination du compromis entre le nombre de fragments et le coût d'initialisation des tâches parallèles. On trouve cette approche dans les systèmes Bubba et DBS3 [BSCD91].

Parallélisation du produit

La plupart des travaux de parallélisation des opérations de l'algèbre relationnelle portent sur le produit de relation[2]. Il existe plusieurs raisons à cela [BBDW83, OV91,

2. “Produit” est équivalent à *natural join*, en anglais.

Omi95]. Le principe de base consiste à redistribuer les relations entre les UCs, décomposer le produit en *instances du produit* et diffuser ces instances, chacune portant donc sur les fragments de la relation dans une UC. L'ensemble des résultats des instances des produits doit être équivalent à l'exécution du produit sur les relations entières.

La redistribution des relations peut être évitée si l'algorithme du produit se sert de la distribution initiale des relations sur les UCs. Pour cela, il faut que les fragments des relations dans chaque UC contiennent tous les n-uplets susceptibles d'être combinés. Autrement dit, le schéma de distribution pour les deux relations doit être le même et doit porter, dans chaque relation, sur l'attribut utilisé pour faire le produit. Finalement, les relations doivent être distribuées dans le même nombre d'UCs.

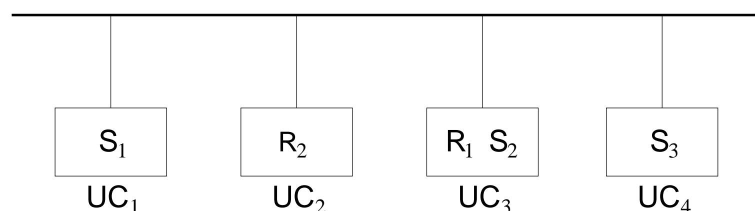


FIG. 2.14 - *Exemple de distribution initiale des relations*

Dans la suite, nous expliquons les trois principales méthodes de parallélisation du produit de relations : parallélisation de produits par boucles imbriquées, par tri-composition, et par hachage [DG92, SD89, WDYT91]. Nous ne faisons aucune hypothèse sur la distribution initiale des relations, par conséquent, la phase de redistribution initiale doit être étudiée. Cette phase est souvent différente, selon la méthode de parallélisation utilisée. Dans nos exemples, nous allons utiliser deux relations R et S, dont R est plus petite que S. Nous supposons que R est partagée en 2 fragments, R_1 et R_2, chacun placé dans une UC, et que S est partagée en 3 fragments, S_1, S_2 et S_3, également placés dans des UCs différentes. R_1 et S_2 sont placés sur la même UC (cf. figure 2.14). Nous cherchons à faire le produit de R par S, noté $R \bowtie S$. L'attribut de chaque relation utilisé pour faire le produit est appelé *attribut du produit*.

La méthode des *boucles imbriquées* consiste à diffuser tous les fragments de la relation plus petite, disons R, vers tous les UCs ayant un fragment de S [BBDW83, OV91, DNB93]. On aurait donc, dans chaque UC ayant un fragment de S, une copie de la relation R (cf. figure 2.15)[3]. Les n-uplets de la relation R sont ensuite comparés avec chaque n-uplets des fragments de la relation S. Cette phase de comparaison des

3. Dans la figure, nous montrons uniquement les UCs concernant l'exécution du produit, aussi bien que les fragments participants dans le produit dans chaque UC.

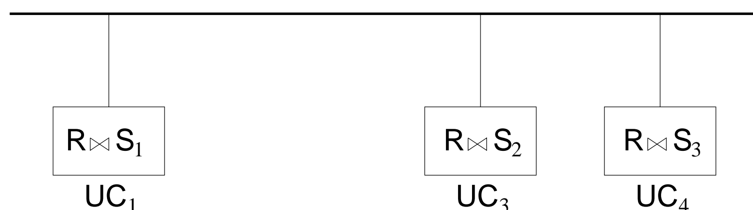


FIG. 2.15 - *Distribution des fragments d'un produit parallèle par boucles imbriquées*

n-uplets est effectuée en parallèle dans chaque UC. La diffusion des fragments de R peut être faite également en parallèle par chaque UC ayant un fragment de R.

La méthode est efficace si la relation la plus petite peut être entièrement chargée en mémoire dans chaque UC. Ainsi la méthode poursuit un seul balayage de chaque S_i, car un n-uplet de S_i est comparé avec tous les n-uplets de R dans une itération de la boucle.

La première phase de la méthode par *tri-composition* consiste à redistribuer les deux relations selon le même critère de distribution sur l'attribut du produit. Dans Gamma, on utilise une fonction de hachage sur cet attribut, associée à une table de fragmentation (en anglais *split table*) qui sert à placer les fragments [SD89]. La fonction de hachage sur l'attribut ($h(R.A)$) donne toujours une valeur entre 1 et n, où n est le nombre d'UCs. La table de fragmentation indique l'UC vers lequel le n-uplet doit être placé, selon le résultat de la fonction de hachage (cf. figure 2.16).

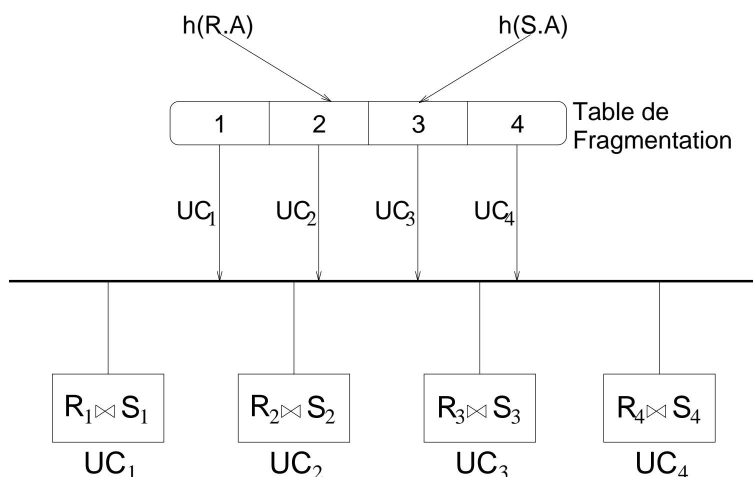


FIG. 2.16 - *Exemple de produit parallèle par tri-composition*

La deuxième phase de la méthode exécute le produit par tri-composition classique

dans chaque UC [EN89]. L'avantage de cette méthode est due au fait que chaque n-uplet de R n'a pas besoin d'être comparé avec chaque n-uplet de S, puisque les fragments sont triés. De plus, il n'y a pas de duplication de R dans chaque UC participant au produit.

La méthode de produit parallèle par *hachage* peut être résumée dans les phases suivantes :

1. Fragmenter la relation la plus petite, disons R, selon un critère de distribution sur l'attribut du produit. Normalement on utilise une fonction de hachage sur l'attribut [DG92, Omi95]. Distribuer les R_i, ($0 < i \leq n$ où n est le nombre d'UCs), résultantes de la fragmentation, vers les UCs.

2. Au fur et à mesure que les n-uplets arrivent dans chaque UC, construire une *table de hachage* en mémoire pour chaque R_i. On utilise une fonction de hachage différente de celle utilisée pour fragmenter R. Si R_i ne peut pas être entièrement chargée en mémoire dans chaque UC, créer des paquets (en anglais *buckets*) de n-uplets de telle sorte que chaque paquet puisse être chargé en mémoire (cf. figure 2.17). Un fragment R_i peut être encore divisé en m paquets ($R_{ij}, 0 < j \leq m$). Les paquets sont stockés dans des fichiers temporaires en attendant qu'il y ait de la place en mémoire pour les traiter.

3. Fragmenter la deuxième relation, disons S, selon le même critère et distribuer les S_i.

4. Au fur et à mesure qu'une UC reçoit un n-uplet de S_i, on effectue le sondage (en anglais *probing*) de la table d'hachage en mémoire pour comparer les n-uplets. De manière équivalente aux fragments de R, créer les paquets S_{ij} en disque pour traitement ultérieur.

5. Une fois terminé le traitement R_{i1} avec S_{i1}, créer la table d'hachage et faire le sondage pour les paquets restants dans chaque UC.

Il existe plusieurs variantes de la méthode de produit par hachage décrite ci-dessus [SD89, KO90, SC90, ZG90, OL92]. Les méthodes proposées par [LTS90] et par [Omi91] se distinguent des autres citées puisqu'elles sont proposées pour un système à mémoire partagée. La plupart des autres méthodes essaient de réduire les effets de la distribution non-uniforme des données (en anglais *data skew*) dans la performance des algorithmes de produit par hachage [WDJ91]. En général, ce produit est plus performant par rapport au produit par boucle imbriquée et par tri-composition.

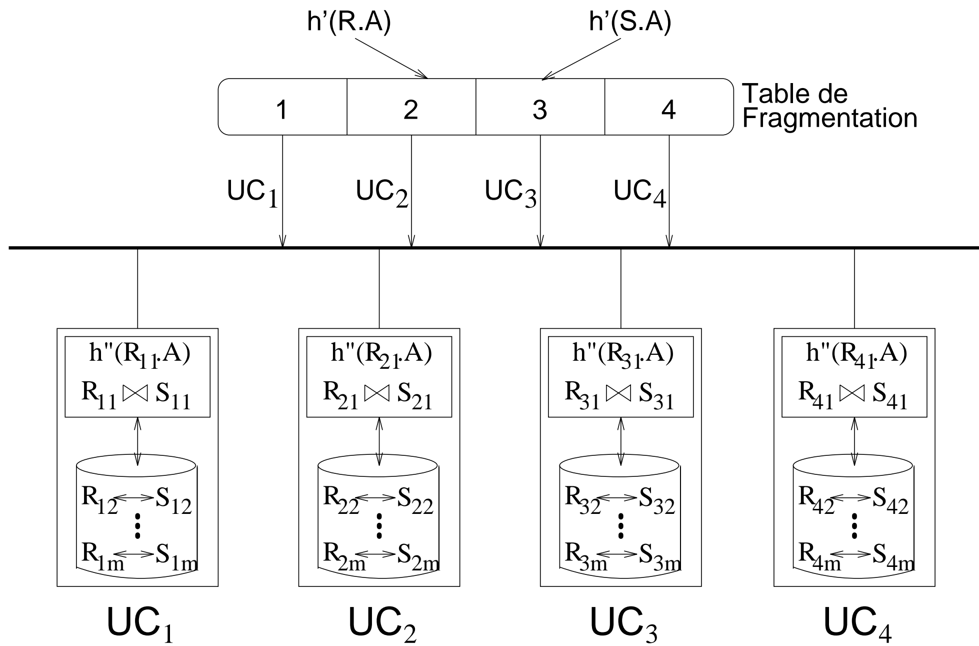


FIG. 2.17 - *Exemple de produit parallèle par hachage*

2.3.1.2 Parallélisme inter-opérations

Le parallélisme inter-opérations consiste à exécuter, en parallèle, les opérations d'un plan d'exécution d'une requête. En général, ces plans sont représentés par des arbres de traitement [SD90, ZZBS93]. Un arbre de traitement est un arbre binaire où les feuilles sont des relations et les nœuds internes des opérations (cf. figure 2.18). Les plans donnés par ces types d'arbres sont exécutés à partir des feuilles jusqu'à l'opération racine de l'arbre. Pendant l'exécution, le résultat d'une opération forme une relation intermédiaire qui est ensuite utilisée comme opérande par la prochaine opération et ainsi de suite. Les arbres de la figure 2.18 montrent des plans d'exécution pour une requête du type :

Select R.*
from R, S, T, U
where R.a = S.a and S.b = T.b and T.c = U.c

L'ordre d'exécution des opérations du plan d'exécution donné par chaque type d'arbre est le suivant :

- Arbre gauche (en anglais *left-deep*) et arbre zig-zag : $(((R \bowtie S) \bowtie T) \bowtie U)$.
- Arbre droit (en anglais *right-deep*) : $(R \bowtie (S \bowtie (T \bowtie U)))$.
- Arbre "bushy" : $((R \bowtie S) \bowtie (T \bowtie U))$.

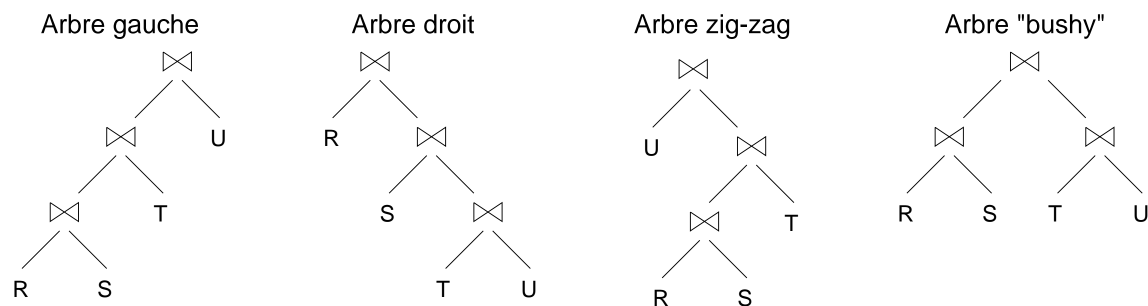


FIG. 2.18 - *Arbres de traitement*

Dans les arbres gauche, droit et zig-zag, un opérande au moins doit être une relation de base, tandis que pour les arbres “bushy” les deux opérandes peuvent être des relations de base ou des relations intermédiaires. La relation intermédiaire est toujours à gauche d'une opération dans les arbres gauches et à droite dans les arbres droits. Ces relations peuvent être soit à gauche soit à droite d'une opération dans les arbres zig-zag.

Le type d'arbre de traitement utilisé pour représenter une requête met en évidence deux formes de parallélisme inter-opérations : le parallélisme vertical et le parallélisme horizontal.

- Le *parallélisme vertical* se produit lors de l'exécution en *pipeline* de deux ou plusieurs opérations consécutives, comme nous l'avons expliqué dans la section 2.1. Dans ce type de parallélisme, une séquence d'opérations de l'algèbre forme un *pipeline*, où une opération produit des données qui sont consommées par la prochaine opération comme dans une ligne de montage. Le parallélisme est dû au fait que chaque opération du *pipeline* fait une partie de la tâche globale en même temps, mais sur des données différentes [Les93].

  Les arbres gauche et droit rendent facile l'exécution en *pipeline* des opérations d'un plan d'exécution [SD90]. Selon la méthode de produit utilisée, on peut balayer, en parallèle, les relations de base et effectuer les produits au fur et à mesure que les n-uplets des relations intermédiaires sont traités. Les arbres zig-zag permettent la division du plan d'exécution en plusieurs *phases* où certaines opérations sont exécutées en *pipeline* [ZZBS93]. Une approche semblable aux arbres zig-zag est présentée dans [CLYY92] où les arbres sont dites *segmentés*.

- Le *parallélisme horizontal* se produit lors de l'exécution parallèle et indépendante de deux opérations. Il s'agit notamment de l'exécution des sous-arbres indépendants dans un arbre “bushy”. Dans la figure 2.18, on pourrait exécuter

$(R \bowtie S)$ en parallèle avec $(T \bowtie U)$ de l'arbre "bushy". Ce type de parallélisme est trouvé, en particulier, dans [LST91] et dans le système XPRS [Hon92].

2.3.2 Transactions et concurrence

Dans les systèmes de bases de données, les opérations d'un programme d'application qui accèdent à la base sont groupées dans des *transactions* [EGLT76]. Du point de vue du système, l'exécution d'une opération est considérée comme une action atomique. Cela veut dire que le système exécute une transaction comme une séquence d'actions atomiques portant sur les objets de la base. Pour l'instant, on considère les actions suivantes [DA82] :

- *Commencer*: initialiser une transaction;
- *Valider*: terminer une transaction;
- *Annuler*: arrêter une transaction et défaire ses effets sur la base;
- *Lire(objet)* : charger une image de l'objet depuis la base dans l'espace de travail de la transaction;
- *Écrire(objet)* : écrire la valeur de l'objet dans la base à partir d'une image de cet objet dans l'espace de travail de la transaction.

En général, les transactions possèdent les propriétés suivantes, dites ACID [Gra81, GR93] :

- *Atomicité*: la séquence d'actions d'une transaction est indivisible. Soit toutes les actions de la transaction sont exécutées, soit aucune ne l'est. Il s'agit du principe du tout ou rien.
- *Cohérence*: quand on considère l'exécution d'une transaction toute seule, cette exécution amène la base de données d'un état cohérent vers un nouvel état également cohérent (cf. figure 2.19). Il est évident que, pendant l'exécution d'une transaction, la base peut se trouver dans un état intermédiaire non cohérent.
- *Isolation*: les actions d'une transaction sont isolées. Cela veut dire que les résultats intermédiaires d'une transaction ne sont pas visibles des autres transactions.
- *Durabilité*: il est impossible d'annuler les résultats d'une transaction qui s'est bien terminée.

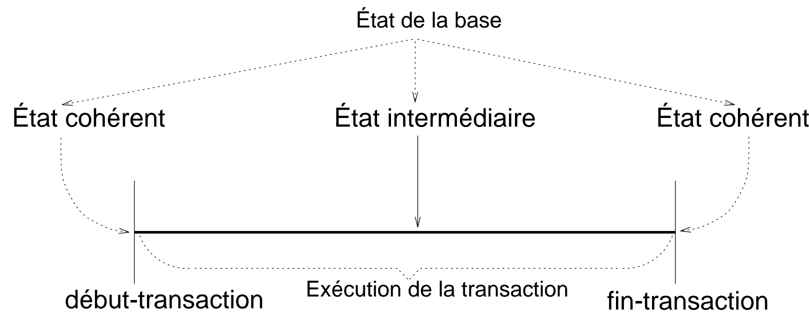


FIG. 2.19 - *Exécution d'une transaction*

Les transactions qui possèdent les propriétés ACID sont habituellement appelées *transactions atomiques*. Ces propriétés sont assurées par un ensemble de protocoles. Une partie de ces protocoles assure l'atomicité d'exécution c'est à dire la cohérence et l'isolation, tandis que l'autre partie assure l'atomicité en cas de panne : les propriétés d'atomicité et de durabilité[4].

Dans cette section, on s'intéresse plutôt à l'atomicité d'exécution. Ses protocoles, appelés *protocoles de contrôle de concurrence*, permettent au système de base de données d'exécuter plusieurs transactions concurrentes, tout en assurant les deux propriétés de l'atomicité d'exécution. Pour cela, ces protocoles utilisent un *critère de cohérence* : l'exécution concurrente d'un ensemble de transactions est dite *cohérente* si elle respecte ce critère. La plupart des protocoles de contrôle de concurrence utilise la *sérialisation* (en anglais *serializability*) comme critère de cohérence [BHG87, AE92b].

2.3.2.1 Sérialisation

Avant de discuter de la sérialisation proprement dite, nous avons besoin de définir quelques concepts. Considérons un ensemble de transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Un *historique* H, défini sur $\mathcal{T}$, donne un ordre partiel des actions appartenant aux transactions de $\mathcal{T}$ [BHG87]. Dans H, les actions des T_i de $\mathcal{T}$ peuvent être entrelacées avec les actions de T_j. Un type particulier d'historique, appelé *historique séquentiel*, possède des transactions entièrement exécutées les unes après les autres en séquence. Par conséquent, les actions de ces transactions ne sont pas entrelacées.

La figure 2.21 donne deux historiques possibles pour trois transactions données dans la figure 2.20. L'historique 1 est un historique séquentiel. On voit clairement l'ordre d'exécution des transactions, dans l'exemple T_2 puis T_1 et ensuite T_3. En

4. Comme le montre [Wei89a], les deux familles de protocoles ne sont pas tout à fait indépendantes. On les sépare ici pour des raisons didactiques.

T_1	T_2	T_3
commencer	commencer	commencer
lire(x)	écrire(x)	lire(x)
écrite(x)	écrire(y)	lire(y)
valider	lire(z)	lire(z)
	valider	valider

FIG. 2.20 - *Exemples de transactions*

revanche, l'historique 2 présente une exécution pour les mêmes trois transactions, mais cette fois avec l'entrelacement des actions. Dans les deux cas, les historiques respectent l'ordre des actions de chaque transaction participante. Par exemple, l'opération écrire(y) de T_2 est exécutée avant l'opération lire(z) de la même transaction dans les deux historiques. Lorsqu'on considère uniquement une transaction, l'ordre d'exécution de ses actions est donc totalement défini.

Historique 1			Historique 2		
T_2	:	commencer	T_2	:	commencer
T_2	:	écrire(x)	T_2	:	écrire(x)
T_2	:	écrire(y)	T_1	:	Commencer
T_2	:	lire(z)	T_1	:	lire(x)
T_2	:	Valider	T_3	:	Commencer
T_1	:	Commencer	T_3	:	lire(x)
T_1	:	lire(x)	T_1	:	écrire(x)
T_1	:	Écrite(x)	T_1	:	Valider
T_1	:	Valider	T_2	:	écrire(y)
T_3	:	Commencer	T_3	:	lire(y)
T_3	:	lire(x)	T_2	:	lire(z)
T_3	:	lire(y)	T_2	:	Valider
T_3	:	lire(z)	T_3	:	lire(z)
T_3	:	valider	T_3	:	valider

FIG. 2.21 - *Exemples de historiques*

Par induction de la propriété de cohérence, l'exécution d'une suite des transactions atomiques sur un état cohérent de la base amène cette base vers un état également cohérent. Par conséquent, l'historique 1 de la figure 2.21 présente une exécution cohérente des transactions T_1, T_2 et T_3. On considère qu'un historique *équivalent* à un historique cohérent est également cohérent. Alors si l'on peut prouver l'équivalence entre l'historique 1 et l'historique 2 de la figure 2.21, on doit garantir que l'historique

2 est lui aussi cohérent.

La notion d'équivalence entre les historiques est basée sur les résultats observables de l'exécution des transactions composant les historiques [BHG87]. Deux historiques, H_1 et H_2, composés par les mêmes transactions, sont équivalents, noté $H_1 \equiv H_2$, s'ils produisent les mêmes résultats et les mêmes effets sur une base de données. Le principe de la sérialisation considère un historique cohérent, dit *historique sérialisable* (en anglais *serializable*), s'il est équivalent à un historique séquentiel [DA82]. Par exemple, l'historique 2 de la figure 2.21 est un historique sérialisable.

Les protocoles de contrôle de concurrence qu'utilisent la sérialisation comme critère de cohérence ordonnent les actions des transactions d'un historique de telle sorte que l'historique soit sérialisable. Ainsi ces protocoles sont capables d'assurer un comportement cohérent d'un ensemble de transactions, malgré l'entrelacement de leurs actions. On considère une action comme l'occurrence d'une opération sur un objet de la base. Ainsi "lire(x)" est une action qui représente l'exécution de l'opération "lire" sur l'objet "x". "lire(z)" est une autre occurrence de la même opération sur un autre objet.

L'ordonnancement des actions dans un historique sérialisable est déterminé par certaines relations entre les opérations, dites *relations de conflit* [Kor83]. Les problèmes proviennent de l'ordre dans lequel sont exécutées *les actions de différentes transactions* lorsque ces actions concernent les mêmes objets. Si deux opérations sont en conflit, leurs actions sur les mêmes objets dans un historique doivent être correctement ordonnancées pour que l'historique soit sérialisable.

Quand on considère uniquement les opérations "lire" et "écrire", on a une relation de conflit déjà classique, appelée *compatibilité* [DA82, Kor83, Pap86, BHG87]. Toutefois, avec les systèmes de bases de données à objet, l'ensemble des opérations qu'on peut appliquer sur les objets s'est largement enrichi [Kim90, DLR91, Cat91, CAD+94]. De plus, ces opérations sont typées et contiennent beaucoup plus de sémantique que les opérations "lire" et "écrire" classiques. Alors, dans le but d'augmenter la concurrence entre les transactions tout en assurant la sérialisation des historiques, des nouvelles relations de conflit ont été proposées [SZ89, CFR89, Elm92].

2.3.3 Relations de conflit

En général, deux opérations sont en conflit quand leurs effets sont dépendants de l'ordre de leurs exécutions. Dans un historique sérialisable, les opérations qui sont en conflit apparaissent dans le même ordre que dans un historique séquentiel équivalent. Les relations de conflit que nous analysons dans la suite sont proposées dans la littérature pour la sérialisation des transactions atomiques.

2.3.3.1 Commutativité

La commutativité est la relation entre opérations la plus utilisée pour déterminer si ces opérations peuvent s'exécuter de manière concurrente. Deux opérations sont en conflit si elles ne commutent pas, c'est à dire, le résultat d'une séquence qui contient les opérations dépend de l'ordre dans laquelle les opérations sont exécutées.

Lorsqu'on considère des Types de Données Abstraits (TAD), on trouve plusieurs façons de déterminer la commutativité des opérations [RB87, BR88, Wei88, CRR91, MM93]. L'objectif est toujours d'obtenir moins de conflit entre les opérations et ainsi d'avoir plus de concurrence. En général, on utilise la sémantique des opérations du TAD pour augmenter leur commutativité. Dans ce cadre, il y a deux approches :

1. Commutativité basée sur l'effet localisé des opérations sur un objet [BR88, MM93].

2. Commutativité fondée sur l'effet des opérations sur un objet tenant en compte des paramètres de sortie.

La commutativité et l'effet localisé

Dans [BR88], chaque objet possède une représentation sur la forme de graphe où les sommets sont des objets et les arêtes sont des relations “composé-de”. A chaque opération est associé un *ensemble d'effets*, contenant les sommets et les arêtes qui ont été modifiés par l'opération. Deux opérations commutent si l'intersection entre leurs ensemble d'effets est vide.

L'approche proposée par [MM93] définit la commutativité des méthodes d'une classe par l'analyse du code source au moment de la compilation. A chaque méthode est associé un *vecteur d'accès* qui spécifie le mode d'accès de la méthode aux variables d'instances de la classe. Deux méthodes commutent si leurs vecteurs d'accès commutent. Deux vecteurs d'accès commutent si leurs modes d'accès sur chaque variable d'instance commune sont compatibles.

Considérons un type abstrait Personne dont la structure est donnée par [nom, adresse, date_naissance]. Soient deux opérations pour ce type, mod_nom et mod_add qui modifient respectivement les variables nom et adresse. Quand on considère l'approche de [BR88], on voit que ces opérations commutent puisqu'elles ne concernent pas les mêmes arêtes de la structure de l'objet. Elles commutent également dans l'approche proposée par [MM93] car leurs vecteurs d'accès commutent.

La commutativité et les paramètres de sortie

D'autres approches définissent la relation de commutativité par rapport à l'état des objets et les valeurs des paramètres de sortie de l'exécution d'une opération. Dans ces

approches, deux opérations d'un type abstrait commutent si elles produisent le même état final et retournent des paramètres de sortie identiques lorsqu'on les applique sur un même objet dans n'importe quel ordre.

L'approche proposée dans [RB87] définit une *interaction* comme étant une paire $\langle p, r \rangle$ signifiant que l'exécution de l'opération p retourne le résultat r. La spécification d'un type abstrait comporte l'ensemble des interactions possibles et aussi une fonction de compatibilité entre toutes les interactions déclarées. D'après la spécification, il est possible de dériver une matrice de compatibilité entre les interactions. Deux interactions sont compatibles lorsque leurs opérations commutent dans tous les états d'un objet du type.

L'approche de [Wei88] propose deux formes de commutativité : en avant et en arrière. Dans les deux cas, la commutativité est définie en terme de l'état de l'objet produit par l'exécution des opérations aussi bien que des paramètres de sortie. Ce qui les différencie est l'hypothèse faite sur la valeur à partir de laquelle les opérations sont appliquées.

La commutativité en avant nécessite que les opérations soient appliquées sur un état cohérent de l'objet (un état produit par une transaction validée) et que toutes les séquences possibles entre les opérations sur cet état soient équivalentes. L'algorithme de [Wei88] propose l'utilisation des listes d'intentions pour ordonner les opérations dans ce type de commutativité.

Pour la commutativité en arrière, les opérations sont appliquées sur les états intermédiaires des objets. Pour que deux opérations commutent, l'état final de l'objet produit par les deux séquences possibles des opérations doit être le même. Dans [Wei88], l'algorithme proposé utilise des journaux-défaire car on doit pouvoir défaire certaines opérations lorsqu'un état intermédiaire de l'objet n'est plus valide.

Prenons un type abstrait Compte muni des trois opérations :

- Créditer(x) : ajoute x au solde du Compte.
- Débiter(x) : retranche x au solde du Compte si celui-ci est suffisant. Dans le cas contraire, l'opération ne change pas la valeur de solde et délivre une exception.
- Solde() : délivre la valeur du solde.

La commutativité des opérations du type Compte est donnée par la table 2.1. Dans le cas d'une exécution correcte, les opérations délivrent 0. L'opération Solde, par exemple, commute avec Débiter, dans le cas d'échec de Débiter. En fait, le résultat délivré montre que l'exécution de l'opération a modifié l'état de l'objet. Dans le cas d'échec, il n'y a pas eu de modification. C'est justement dans ce cas qu'on peut obtenir plus de commutativité.

	⟨Créditer(x),0⟩	⟨Débiter(x),0⟩	⟨Débiter(x),-1⟩	⟨Solde(x),0⟩
⟨Créditer(x),0⟩	non	non	non	non
⟨Débiter(x),0⟩	non	non	non	non
⟨Débiter(x),-1⟩	non	non	oui	oui
⟨Solde(x),0⟩	non	non	oui	oui

TAB. 2.1 - *Commutativité des opérations du TAD Compte*

2.3.3.2 Dépendances séquentielles

Une relation de conflit entre deux opérations sur un objet peut être définie en termes de dépendance séquentielle [HW88]. Une opération p est en conflit avec une autre opération q si p peut *invalider* q lorsque p apparaît plus tôt que q dans un historique séquentiel. De manière informelle, s'il existe deux historiques H_1 et H_2 tels que les séquences $H_1 \circ p \circ H_2$ et $H_1 \circ H_2 \circ q$ soient légales, et $H_1 \circ p \circ H_2 \circ q$ soit illégale, alors p invalide q ou q dépend de p[5].

La relation d'invalidation entre deux opérations induit une dépendance entre leurs transactions correspondantes. Selon l'approche proposée par [HW88], deux transactions n'ont pas de dépendance et peuvent être exécutées dans n'importe quel ordre si aucune des opérations de l'une n'invalide aucune opération de l'autre. L'approche utilise des mises à jour différées avec des listes d'intention. L'algorithme proposé dans [HW88] utilise le verrouillage et l'estampillage. A chaque objet sont associées les informations suivantes :

- Une liste d'intention pour chaque transaction comprenant les opérations qui doivent être appliquées à l'objet en cas de validation de la transaction ;
- L'état valide de l'objet. Cet état traduit les effets de toutes les transactions ayant validé sur l'objet, avec l'ordre de validation déterminé par estampillage ;
- Un ensemble de verrous qui associe chaque opération à une transaction active ayant exécuté l'opération.

Un verrou sur un objet est obtenu lorsque l'opération appelée n'établit pas de relation d'invalidation avec les autres opérations non validées dans l'objet. Le protocole prend en compte les paramètres et le résultat de l'exécution de l'opération dans la vérification de dépendance entre les opérations. On calcule l'effet de l'opération, dit

5. La notion de légalité est directement liée à la sémantique de l'objet sur lequel s'applique p et q. Par exemple, quant il s'agit d'un compte bancaire, on peut établir que le solde ne doit jamais être négatif. Ainsi une séquence qui amène à un solde négatif est considérée comme illégale

vision dans [HW88], par une pseudo-exécution sur l'état de l'objet. Dans le cas où le verrou sur l'objet est obtenu, l'opération est ajoutée à la liste d'intention de l'objet associée à la transaction correspondante. L'état de l'objet n'est pas modifié car il doit être toujours valide. Au moment de la validation, la liste d'intention est utilisée pour mettre à jour l'état validé de l'objet en tenant compte de l'ordre des estampilles.

L'utilisation des listes d'intention évite l'occurrence de flux d'information entre les transactions actives car les modifications sur les objets ne sont faites qu'au moment de la validation. Ceci évite la formation de dépendances de validation et d'annulation entre les transactions qui peuvent donc être validées dans n'importe quel ordre. L'utilisation de l'estampillage garantit la validation de la transaction dans le même ordre pour tous les objets ayant participé à une transaction.

2.3.3.3 Recouvrabilité

La propriété de recouvrabilité augmente la concurrence entre les opérations qui ne commutent pas, tout en évitant la formation de cascades d'annulations [Bad89, BR92]. Deux opérations qui ne commutent pas, mais qui sont recouvrables, peuvent s'exécuter en parallèle. Cependant, l'ordre de validation de leurs transactions doit être leur ordre d'exécution. Ainsi, la transaction qui introduit le conflit provoque délibérément une relation de dépendance de validation entre elle-même et une autre transaction concurrente. Les cascades d'annulations sont évitées car aucune dépendance d'annulation n'est formée entre les transactions, donc une des deux transactions peut être validée même si l'autre ne l'est pas.

La spécification d'un type abstrait donne l'ensemble des opérations du type et une table de recouvrabilité pour ses opérations. Dans [BR92], la relation de recouvrabilité prend en compte la valeur des paramètres et le résultat produit par l'exécution des opérations. En revanche, la recouvrabilité est indépendante de l'état de l'objet sur lequel les opérations sont exécutées. Considérons l'exécution d'une opération p sur un état initial quelconque e d'un objet. Une opération q est recouvrable par rapport à p, si l'exécution de q sur e est indépendante de p. Autrement dit, l'exécution de q sur e produit *le même résultat* dans les deux cas : (1) p est exécutée avant q sur e où (2) p n'est pas exécutée sur e.

Dans un historique des transactions concurrentes, une transaction T_i qui exécute une opération recouvrable par rapport à une opération d'une transaction T_j établit une dépendance de validation entre elle-même et T_j. En d'autres termes, T_i ne peut valider qu'après la terminaison de T_j, avec succès ou pas.

Dans l'approche basée sur la relation de commutativité entre les opérations, la sérialisation des historiques est garantie en empêchant deux opérations qui ne commutent pas de s'exécuter en parallèle. Cela évite la formation de dépendance d'an-

nulation entre les transactions et, par conséquent, les cascades d'annulation. Quand on utilise la recouvrabilité, certaines opérations, qui ne commutent pas, peuvent être recouvrables. Dans ce cas, les opérations sont autorisées à s'exécuter en parallèle, tout en gardant la propriété de sérialisation des historiques. Pour cela, il suffit de ne pas permettre la formation de cycle dans la relation de dépendance de validation entre les transactions [BR92]. Étant donné un seul objet, à chaque fois qu'une transaction appelle une opération qui est recouvrable par rapport à d'autres opérations des transactions encore actives, l'algorithme de contrôle de concurrence interdit la formation de cycle dans le graphe de dépendance de validation. S'il n'y a pas de cycle produit par l'introduction de la nouvelle opération, alors elle peut être exécutée, sinon elle est mise en attente jusqu'au moment où une des transactions qui participe au cycle est terminée.

La recouvrabilité des opérations du type Compte est montrée dans la table 2.2. On considère les opérations des colonnes comme exécutées, tandis que celles de lignes sont appelées. Lorsqu'on compare avec la relation de commutativité des mêmes opérations du type Compte (cf. table 2.1), on aperçoit qu'il y plus d'opérations recouvrables que d'opérations qui commutent. Ainsi, on peut obtenir plus de concurrence entre transactions tout en assurant la sérialisation des historiques.

Notez que la recouvrabilité est une relation *asymétrique*. Le résultat de l'exécution de Créditer ne dépend pas de l'exécution précédente d'aucune des opérations. Ainsi, Créditer est recouvrable par rapport à toutes les autres. En revanche, le résultat de l'exécution de Solde est différent selon que l'on fait Créditer ou Débiter avant. Le résultat de l'exécution de Débiter change si l'on fait Créditer avant. Par exemple, Débiter(15) d'une compte de solde initialement 10 délivre -1, mais peut délivrer 0 dans le cas on fait Créditer(5) avant.

	⟨Créditer(x)⟩	⟨Débiter(x)⟩	⟨Débiter(x),-1⟩	⟨Solde(x)⟩
⟨Créditer(x),0⟩	oui	oui	oui	oui
⟨Débiter(x)⟩	non	oui	oui	oui
⟨Débiter(x),-1⟩	non	non	oui	oui
⟨Solde(x)⟩	non	non	oui	oui

TAB. 2.2 - *Recouvrabilité des opérations du TAD* Compte

2.3.3.4 Localité associée à la commutativité et à la recouvrabilité

L'approche proposée dans [CRR91] prend en compte l'organisation structurelle des objets pour augmenter la concurrence des opérations et encore produire des his-

toriques cohérents. Lorsque deux opérations concernent deux parties différentes d'un même objet, aucun conflit n'est établi entre les opérations qui ainsi peuvent s'exécuter de manière concurrente. L'approche est fondée sur la notion de localité d'une opération portant sur un objet. L'utilisation de la localité des opérations a été déjà exploitée dans [BR88, MM93] (cf. section 2.3.3.1). La différence entre l'approche de [CRR91] et les approches de [BR88, MM93] est due au fait que, dans [CRR91], il y a un raffinement des dépendances induites par la relation de conflit à cause de l'utilisation de la recouvrabilité, tandis que dans [BR88, MM93] seule la commutativité des opérations est considérée.

Comme dans [BR88], un objet est représenté par un graphe. La localité d'une opération est définie par l'ensemble des sommets modifiés par l'opération. Deux opérations sur un objet sont en conflit si l'intersection de leurs localités est non vide. La relation de conflit est transformée dans une matrice de compatibilité entre les opérations d'un type abstrait. Chaque entrée de la matrice de compatibilité spécifie la dépendance induite par l'exécution concurrente de deux opérations. Selon la sémantique des opérations, soit elles commutent et aucune dépendance n'est établie, soit elles ne commutent pas mais une est recouvrable par rapport à l'autre. Dans ce cas, la table contient une entrée indiquant la dépendance de validation. Dans le cas où les opérations ni commutent, ni sont recouvrables, une dépendance d'annulation est marquée dans la table de compatibilité.

Dans un historique, si deux transactions exécutent des opérations recouvrables, alors les transactions forment une dépendance de validation. Si deux transactions exécutent des opérations qui commutent, alors ces transactions forment une dépendance d'annulation. Dans le cas où les opérations ne possèdent pas de conflit, aucune dépendance n'est établie entre leurs transactions correspondantes.

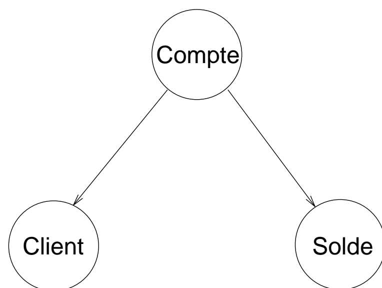


FIG. 2.22 - *Structure d'un objet du type* Compte

Reprenons le TAD Compte. Si l'on considère un compte concernant un client et ayant un solde, on pourrait représenter un objet comme le montre la figure 2.22. On

ajoute une opération appelée Cnom(x) qui délivre le nom du titulaire du compte. La table 2.3 donne les dépendances induites par l'exécution concurrente des opérations pour ce type. DV dénote dépendance de validation, DA dénote dépendance d'annulation et PD détermine qu'il n'y a pas de dépendance. Notez que la nouvelle opération ne cause aucune dépendance par rapport aux autres opérations car elle modifie une partie différente du graphe de l'objet. On voit également que un "non" de la table 2.2 est remplacé par DA, car les opérations qui ne sont pas recouvrables induisent des dépendances d'annulation.

	⟨Créditer(x)⟩	⟨Débiter(x)⟩	⟨Débiter(x),-1⟩	⟨Solde(x)⟩	⟨Cnom(x)⟩
⟨Créditer(x),0⟩	DV	DV	PD	PD	PD
⟨Débiter(x)⟩	DA	DV	PD	PD	PD
⟨Débiter(x),-1⟩	DA	DA	PD	PD	PD
⟨Solde(x)⟩	DA	DA	PD	PD	PD
⟨Cnom(x)⟩	PD	PD	PD	PD	PD

TAB. 2.3 - *Dépendances induites par les opérations du TAD Compte*

2.4 Conclusion

Dans ce chapitre nous avons examiné différents formes de parallélisme. Premièrement nous avons présenté des architectures matériels et de concepts logiciels qui servent de base pour l'exécution parallèle de programmes. Du côté matériel, nous avons détaillé les architectures multiprocesseurs sur lesquelles est fondé notre travail. Nous avons étudié plusieurs techniques logiciel d'abord pour exprimer le parallélisme et ensuite pour faire communiquer et synchroniser les activités parallèles.

Dans un système où les données sont partagées par ces activités, les techniques de synchronisation des accès communs aux données sont fondamentales pour assurer l'isolation des activités. Dans l'approche dite exclusion mutuelle le programmeur identifie et déclare explicitement les sessions critiques de son programme, de façon qu'elles soient dynamiquement synchronisées et ne manipulent jamais les mêmes données en parallèle. Cette approche a l'avantage de simplifier l'implantation du compilateur mais rend difficile la tâche de programmer la synchronisation. Surtout si l'on considère la gestion des interblocages entre les activités parallèles.

Nous avons étudié l'utilisation de mécanismes d'exclusion mutuelle pour notre approche. Une solution possible consiste à considérer chaque classe d'un schéma comme un moniteur : les attributs du type de la classe definissent la ressource encapsulée et les méthodes les procédures. Les méthodes sont donc exécutées en exclusion mutuelle.

Cela peut être fait en associant un sémaphore sur les attributs et en introduisant des opérations P et V au début et à la fin du code des méthodes. Alors, dans le code d'une transaction, on peut transformer l'appel d'une méthode dans un appel asynchrone (cf. RPC dans la section 2.1.1) afin de pouvoir appeler plusieurs méthodes à la fois. Les sémaphores assurent alors l'exclusion mutuelle des méthodes parallèles. Nous n'avons pas retenu cette solution pour les raisons suivantes :

- La gestion de passage en session critique à chaque fois qu'on exécute une méthode pourrait pénaliser les performances du système. Il y aurait des opérations additionnelles à réaliser et des files d'attente de processus à contrôler. Tout le contrôle des ressources partagées serait fait au moment de l'exécution.

- Cette solution pourrait conduire à des situations d'interblocage entre méthodes. Les solutions classiques pour résoudre ce problème [DA82] pourraient pénaliser encore plus les performances du système.

Dans l'approche dite dépendance de données le compilateur du langage identifie les sessions des programmes qui ne sont pas critiques afin de générer du code pour les exécuter en parallèle. Cette approche augmente la complexité du compilateur et permet difficilement l'exploitation maximal du parallélisme, car il s'agit d'une approche fondée sur des hypothèses pessimistes. Néanmoins la programmation est beaucoup plus aisée par le fait que le programmeur n'est pas le responsable de la synchronisation des activités parallèles.

Nous avons également étudié l'expression et le contrôle du parallélisme dans les langages à objets concurrents. Dans ces langages, la notion d'objet en tant qu'unité d'encapsulation ne permet pas la manipulation directe de l'état des objets si ce n'est que par des méthodes. Ainsi, si l'on utilise l'objet comme unité de parallélisme, la synchronisation des accès en parallèle à l'état des objets est automatiquement réalisée. Par contre, lorsqu'on veut raffiner le parallélisme afin d'exécuter des méthodes en parallèle, on doit les synchroniser.

Nous avons examiné les différents approches pour l'exploitation du parallélisme dans les systèmes de bases de données. Nous avons pu constater que la plupart des travaux portent sur le traitement des requêtes. En général, l'approche consiste à paralléliser certaines opérations de l'algèbre relationnelle. Dans ce contexte, nous avons étudié le parallélisme intra-opération et inter-opération. Dans les deux cas, le parallélisme est obtenu par traduction et transformation de la définition de la requête.

Finalement, nous avons présenté la gestion des transactions concurrentes. Nous nous sommes intéressés surtout à l'aspect isolation des transactions, notamment en ce qui concerne la sérialisation des historiques et les relations de conflit entre les méthodes des transactions.

Chapitre 3

Parallélisation des Transactions

3.1 Le modèle à objets 57
  3.1.1 Les objets et leurs types 58
  3.1.2 Schéma et base de données 61
  3.1.3 Transactions 62
3.2 Parallélisme intra-transaction 63
  3.2.1 L'expression du parallélisme 64
    3.2.1.1 Parallélisme explicite 64
    3.2.1.2 Parallélisme par transformation 65
  3.2.2 Mise en évidence du parallélisme 66
    3.2.2.1 Environnement des opérations 66
    3.2.2.2 Notation 67
    3.2.2.3 Le mode d'accès 69
    3.2.2.4 Fermeture transitive locale 71
    3.2.2.5 La relation de conflit 75
    3.2.2.6 La fermeture transitive au niveau du schéma 76
    3.2.2.7 La compatibilité 79
3.3 Parallélisation des transactions 81
  3.3.1 Le plan d'exécution source 81
  3.3.2 Construction du plan d'exécution parallèle 82
    3.3.2.1 Schéma transactionnel 82
    3.3.2.2 L'algorithme de transformation du graphe 84
    3.3.2.3 La cohérence sémantique des transactions 88
  3.3.3 Les mécanismes de base pour la parallélisation 90
    3.3.3.1 La création des activités 90
    3.3.3.2 La synchronisation des activités 92
3.4 Conclusion 93

Parallélisation des Transactions

Ce chapitre est consacré à la présentation de notre modèle pour la parallélisation des transactions séquentielles. En général, les transactions manipulent des objets par l'intermédiaire des opérations définies dans leur type. Dans notre modèle, la parallélisation consiste à transformer la définition d'une transaction afin de pouvoir exécuter en parallèle des opérations sur les objets. Ces transformations sont fondées sur certaines propriétés des opérations déterminées statiquement. Ces propriétés assurent la cohérence sémantique des transactions transformées, sachant que certaines opérations sont exécutées en parallèle.

La parallélisation des transactions est effectuée en deux étapes distinctes :

1. A partir de la spécification des opérations des types abstraits, détermination de la compatibilité des opérations.

2. A la compilation de la transaction, transformation du code en utilisant la compatibilité des opérations.

Ce chapitre est organisé de la manière suivante : dans la section 3.1 nous détaillons le modèle à objet utilisé dans notre approche. Puis la section 3.2 met en évidence la façon dont nous exploitons le parallélisme intra-transaction et montre comment nous déterminons la relation de compatibilité entre les opérations. Finalement, la section 3.3 décrit les transformations que nous effectuons sur le code d'une transaction pour exécuter les opérations et explique les mécanismes de base utilisés pour paralléliser et synchroniser l'exécution d'une transaction.

3.1 Le modèle à objets

Les caractéristiques de notre modèle sont celles des modèles proposés par les principaux SGBDs à objets [Deu91, KGBW90, LLOW91, SK91]. Pour plus de détails sur ces caractéristiques voir [ABD+92, Cat91, CW85, DLR91, KL89]. Ici nous détaillons les aspects les plus importants du modèle vis-à-vis de la parallélisation des transactions.

3.1.1 Les objets et leurs types

Chaque objet est instance d'un *type abstrait* (TAD). Un TAD définit d'une part les attributs qui déterminent la structure de l'état de ses objets et, d'autre part, les opérations applicables aux objets du type. Chaque attribut est défini par un identificateur et un type qui peut être atomique ou abstrait. Les types atomiques sont entier, réel, chaîne de caractères, date, booléen. La valeur d'un attribut, dont le type est atomique, est stockée dans l'état de l'objet. La valeur d'un attribut ayant comme type un TAD est un référence vers une instance du TAD. Les attributs de type abstrait permettent de représenter des liens de composition entre les objets. Un objet peut être composant de plusieurs autres. Tous les objets d'un TAD possèdent la même structure de données mais aussi le même comportement.

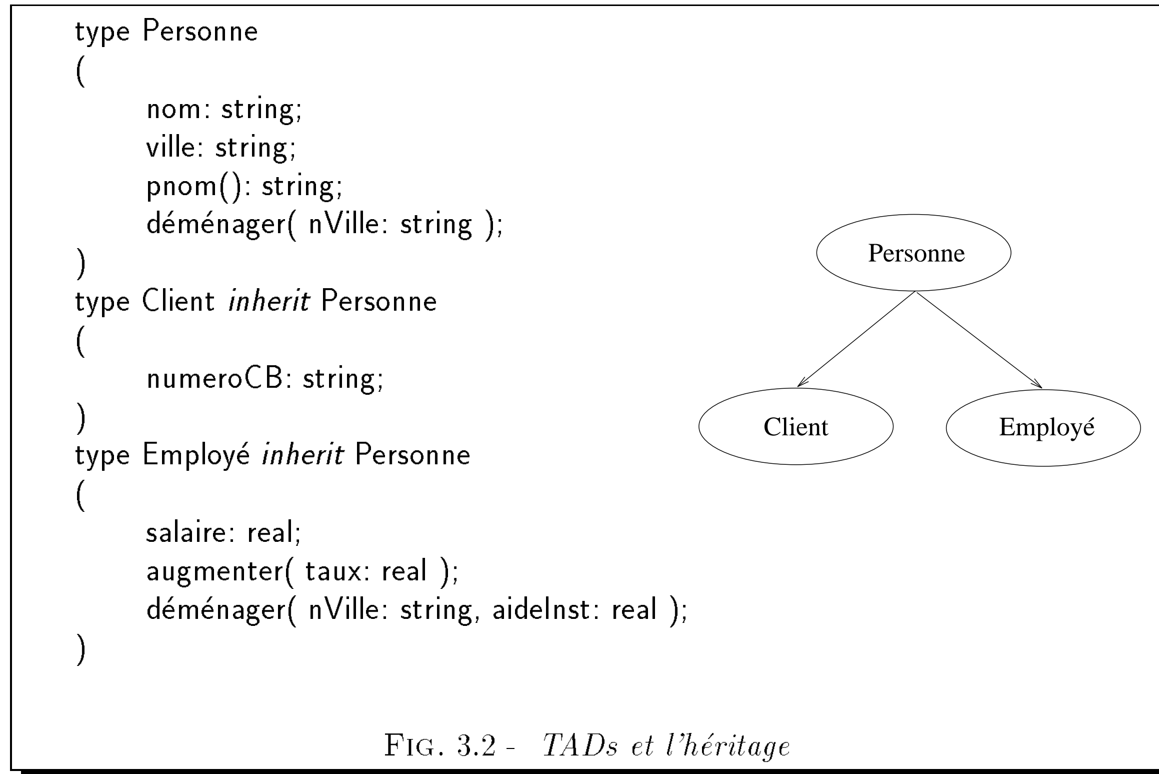
```
type Compte
(
    ncompte: string;
    client: Client;
    soldej: real;
    )
    clientnom(): string;
    créditer( valeur: integer );
    débiter( valeur: integer ): integer;
    virement( ccred: Compte; valeur: integer ): integer;
    solde(): real;
)
```

FIG. 3.1 - *Un type de données abstrait*

La figure 3.1 donne un exemple de type abstrait de données : le type Compte. La structure d'un objet du type Compte est composée des attributs ncompte, client et soldej, tandis que son comportement est défini par les opérations clientnom, créditer, débiter, virement et solde. L'attribut client est du type Client, tous les autres attributs ont des types atomiques. Parmi les opérations de Compte, certaines ont des paramètres d'entrée tandis que d'autres ont des paramètres de sortie. Par exemple, l'opération virement a deux paramètres, le premier du type Compte et le second du type integer, et retourne une valeur de type integer.

Les TADs forment un graphe d'héritage sans cycle. Les sommets représentent les types et les arêtes montrent la relation d'héritage. Les sommets de départ des arêtes sont les super-types tandis que les sommets d'arrivée des arêtes sont les sous-types. Les sous-types héritent des propriétés de ses super-types. L'héritage peut être simple ou multiple. Dans l'héritage simple, un sous-type hérite des propriétés d'un seul super-

type. Si l'héritage est multiple, un sous-type hérite des propriétés de ses super-types. Dans le cas d'héritage multiple, les conflits de noms des propriétés doivent être résolus.



```
type Personne
(
    nom: string;
    ville: string;
    pnom(): string;
    déménager( nVille: string );
)
type Client inherit Personne
(
    numeroCB: string;
)
type Employé inherit Personne
(
    salaire: real;
    augmenter( taux: real );
    déménager( nVille: string, aideInst: real );
)
```


FIG. 3.2 - *TADs et l'héritage*

La figure 3.2 donne la définition de trois types, Personne, Client et Employé. Le graphe de la figure montre la relation d'héritage donnée par la définition des types. Il s'agit d'un cas d'héritage simple. Dans l'exemple, tous les attributs et les opérations du type Personne sont hérités par le Client et par Employé.

Le mécanisme d'*envoi de message* permet à un objet émetteur de demander à un objet cible d'exécuter une de opérations de la cible. Un message décrit l'opération qui doit être exécutée par l'objet et, si nécessaire, les valeurs passées comme paramètres pour l'exécution de l'opération. Par exemple, un objet *cl* du type Client peut envoyer un message vers un objet *cm* du type Compte pour créditer 1000F dans le compte. Cela se fait par le message suivant : cm.créditer(1000);. Un envoi de message a la forme générale <var.op(par)> où var référence l'objet vers lequel on envoie le message pour exécuter l'opération op() avec les paramêtres par. L'opération op() doit être parmi celles définies dans le type abstrait de l'objet référencé par var.

La figure 3.3 présente le code des opérations du type Compte défini dans la figure 3.1. L'objet courant est désigné dans le code des opérations par self. Ainsi, lorsqu'on veut envoyer un message à l'objet courant, on le fait en envoyant un message vers self. Par exemple, dans le code de virement il y a un envoi de message à self pour exécuter l'opération débiter. Par conséquent, un compte qui exécute l'opé-

ration virement envoi un message à lui-même pour exécuter débiter. Dans l'opération clientnom, on peut noter l'envoi de message vers un objet du type Client référencé par l'attribut client.

```
string clientnom() {
    return client.pnom();
}

int débiter( int val ) {
    if ( soldej < val )
        return 0;
    else
        soldej = soldej - val;
    return 1;
}

real solde() {
    return soldej;
}

créditer( int val ) {
    soldej = soldej + val;
}

int virement( Compte ccred, int val ) {
    int res;
    res = self.débiter( val );
    if ( res == 1 ) {
        ccred.créditer( val );
        return 1;
    }
    return 0;
}
```

FIG. 3.3 - *Les opérations du type* Compte.

Au moment de la compilation, on ne peut pas toujours déterminer précisément le type de l'objet cible d'un envoi de message. C'est pourquoi, la liaison entre l'envoi de message et le code de l'opération à exécuter est faite de manière dynamique (*liaison dynamique*. L'opération à exécuter dépend du type abstrait de l'objet cible. En effet, il peut y avoir redéfinition d'opérations dans une hiérarchie d'héritage. Dans ce cas on dit que l'opération est *surchargée*.

Revenons à l'exemple de la figure 3.2. L'opération déménager définie dans le type Personne est héritée par le type Employé. Cependant elle y est redéfinie. Lorsqu'on envoie un message à un objet p pour exécuter déménager, deux cas sont possibles :

- l'objet p est du type Personne. Dans ce cas, on exécute le code de l'opération déménager définie dans Personne;

- l'objet p est du type Employé. Le code de déménager dans Employé sera exécuté.

Les objets peuvent être persistants ou temporaires. Les objets temporaires disparaissent à la fin de l'exécution de l'application qui les a créés. En revanche, les objets persistants créés par une application restent indépendamment de l'exécution de celle-ci. Un objet persistant peut devenir temporaire et vice-versa. Par conséquent, un type abstrait peut avoir des objets persistants ou temporaires. La façon dont les

objets sont rendus persistants ne met pas en cause notre approche pour la parallélisation des transactions. Aussi, nous ne détaillons pas cet aspect. L'important à retenir est que les objets persistants et temporaires sont manipulés de la même manière.

3.1.2 Schéma et base de données

Un *schéma* est un *ensemble fini* de définitions de types abstraits de données. Ces types sont liés par des liens d'héritage et/ou des liens de composition. Du point de vue des liens entre les types, un schéma doit être *cohérent*. La définition d'un type doit utiliser uniquement des types atomiques et/ou des types abstrait définis dans le schéma. Autrement dit, tous les types utilisés dans la définition d'un attribut, d'un paramètre d'opération et dans la clause inherit doivent être décrits dans le schéma. Les identificateurs de types et de leurs propriétés, attributs et opérations, doivent être uniques dans le schéma.

Les types de la figure 3.1 et de la figure 3.2 définissent un schéma ayant quatre types abstraits : Compte, Personne, Client et Employé. Le type Compte fait référence au type Client par l'intermédiaire de l'attribut client. Un objet du type Compte peut envoyer des messages vers un objet du type Client, en particulier, celui référencé par son attribut client. De plus, les types Client et Employé sont liés au type Personne par la relation d'héritage.

Dans notre modèle, une *base de données* est toujours associée à un schéma. Une base est donc un ensemble d'instances de types abstraits définis dans son schéma de référence.

La figure 3.4 présente un exemple de base de données associée au schéma défini par les types de la figure 3.1 et de la figure 3.2. Nous représentons les identificateurs d'objet (oid) avec un # suivi d'un numéro. La base est composée de quatre objets : les objets avec les oids #53 et #22 du type Client, l'objet #102 du type Employé et l'objet #30 du type Compte. Noter la valeur de l'attribut client du compte de numéro 125 qui est l'oid de l'instance de Client (#53) de nom Dupond.

```
(#102, nom: "Dupont",          (#53, nom: "Dupond",
       ville: "Grenoble",            ville: "Paris",
       salaire: 10000 )              numeroCB: "5322762909883754" )

(#30,  ncompte: 125,           (#22, nom: "Martin",
       client: #53,                  ville: "Montpellier",
       soldej: 3000 )                numeroCB: "8454142480389535" )
```

FIG. 3.4 - *Exemple de base de données*

3.1.3 Transactions

Les transactions sont des entités actives qui agissent sur les objets à travers des opérations. Les transactions peuvent être créées, annulées ou validées. Nous considérons, pour l'instant, le modèle classique des transactions [BHG87, GR93]. Dans ce modèle, une transaction est exécutée de manière séquentielle. En plus, ce modèle ne permet pas la création d'une transaction dans une transaction existante.

La définition d'une transaction donne les opérations qui vont être exécutées au sein de la transaction mais aussi une relation d'ordre total entre ces opérations. La définition d'une transaction ne fait aucune distinction entre les opérations sur les objets persistants et celles sur les objets temporaires. Une telle définition contient du code séquentiel qui, en général, est composé des instructions d'un langage de programmation étendu avec des instructions pour la gestion des transactions. Les instructions contenues dans une transaction sont de trois types :

- *Les instructions d'action* telles que l'affectation et l'envoi de message. Elles consultent et modifient l'état du processus dans lequel la transaction est en train de s'exécuter.
- *Les instructions de branchement* servent à contrôler le flot d'exécution de la transaction. Par exemple, nous pouvons citer les instructions conditionnelles telles que le if et le case et les instructions itératives comme le while et le for.
- *Les instructions transactionnelles* gèrent les transactions : création, annulation et validation. Nous les noterons respectivement commencer, annuler et valider.

```
    Compte c1, c2;
    string nom;
    real a, b;
1.  commencer;
2.  nom = c1.clientnom();
3.  a = c1.solde();
4.  b = c2.solde();
5.  if ( a > b*2 ) {
6.      c1.virement( c2, 1000 );
7.      b = c2.solde(); }
8.  valider;
```

FIG. 3.5 - *Exemple de spécification de transaction*

La figure 3.5 montre un exemple de définition d'une transaction. La ligne 1 contient l'instruction pour démarrer tandis que la ligne 8 contient l'instruction pour terminer.

Les instructions valider et annuler peuvent apparaître au milieu du code de la transaction, toutefois il faut retenir que la transaction est terminée dès qu'une des instructions valider ou annuler est atteinte au cours de son exécution.

Dans la même figure, les lignes 2, 3, 4, 6 et 7 donnent des exemples d'instructions d'action. Les opérations utilisées ont été déclarées dans le type Compte (voir page 60). La ligne 5 montre une instruction de branchement.

Du point de vue de l'exécution, nous modélisons les transactions comme des suites finies d'instructions, dont nous nous intéressons plus particulièrement aux envois de message. Au cours de son existence, une transaction peut manipuler plusieurs objets de types différents par l'intermédiaire des messages. La fin d'une transaction est déterminée soit par une validation rendant persistantes et durables les modifications réalisées sur les objets, soit par une annulation ce qui ramène ces objets dans l'état dans lequel ils se trouvaient au début de la transaction.

Dans notre modèle, nous considérons uniquement les transactions *bien formées*, qui respectent les règles suivantes :

1. Une transaction est créée lorsqu'une action *Commencer* se produit, c'est la seule action pour démarrer une transaction. Aucune autre action d'une transaction ne peut se produire *avant* l'action *Commencer*.

2. Une transaction se termine lors de l'exécution de l'instruction *Valider* ou *Annuler* (action de terminaison). Une action transactionnelle de terminaison ne peut se produire qu'*après* l'exécution d'une action *Commencer*.

3. Après l'exécution de l'action *Commencer*, d'autres actions peuvent se produire jusqu'au moment de l'exécution d'une instruction transactionnelle de terminaison. Après cela, aucune action ne peut avoir lieu.

Dans la suite, l'exécution d'une instruction d'action qui envoie un message vers un objet pour exécuter une opération est désignée tout simplement par l'exécution d'une opération.

3.2 Parallélisme intra-transaction

L'exécution d'une transaction T est cohérente si l'exécution de ses opérations est conforme à l'ordre total donné par la définition de T. Cependant certaines opérations sont indépendantes les unes des autres. Cette indépendance nous permet d'avoir un ordre partiel des opérations et d'obtenir tout de même un résultat final cohérent. Si la relation d'ordre des opérations d'une transaction est partielle, nous avons la possibilité d'exécuter certaines de ses opérations de manière parallèle tout en assurant un résultat

cohérent. Nous revenons sur ce point dans la section 3.3. Il faut tout d'abord décrire comment nous introduisons le parallélisme dans la définition des transactions.

3.2.1 L'expression du parallélisme

Lorsqu'on souhaite changer la relation d'ordre entre les opérations d'une transaction de telle sorte qu'on puisse exécuter des opérations en parallèle, on doit modifier la définition de la transaction pour prendre en compte le parallélisme. Deux possibilités sont envisageables : soit le programmeur détermine explicitement le parallélisme, soit l'analyseur du langage, par l'intermédiaire d'une étape de transformation du code, génère automatiquement le parallélisme. Dans la suite nous détaillons ces deux approches.

3.2.1.1 Parallélisme explicite

L'expression du parallélisme explicite peut être faite de deux manières différentes :

- (1) Par l'utilisation des primitives du langage. Dans cette approche, le langage de programmation fournit des primitives pour créer et synchroniser des activités parallèles. Pour la création, on utilise souvent des primitives du type *fork/join* ou *cobegin/coend* [AS83]. La synchronisation de ces activités est faite par des mécanismes tels que la commande *select*, les conditions d'activation, et les contraintes d'activation [Per87, KMV$^+$90, CL91]. Le programmeur utilise les primitives pour créer les activités parallèles et pour contrôler l'accès aux données partagées par ces activités.

- (2) Au travers d'objets actifs. Dans cette approche le parallélisme se traduit par l'exécution concurrente de plusieurs objets actifs. On a donc deux types d'objets :

  1. Des objets tels que ceux décrits dans la section 3.1 et que nous appelons ici objets passifs ;
  2. Des objets qui fournissent des abstractions fonctionnelles comme les processus et que nous appelons ici objets actifs. Comme nous l'avons montré dans la section 2.2, ces objets sont des instances de classes d'objet actifs. La définition d'une telle classe décrit à la fois les fonctions fournies par les objets mais aussi la synchronisation de ses fonctions.

  Dans une transaction, une activité parallèle est déclenchée au moment de la création d'un objet actif. Le programmeur doit décrire la synchronisation des objets de la transaction en tenant compte de la synchronisation spécifiée dans

leurs classes. Les objets actifs et passifs, communiquent par l'envoi de message. L'interaction entre les deux niveaux de messages doit aussi être gérée par le programmeur. Le modèle d'exécution devient donc complexe, ce qui rend plus difficile la programmation des transactions.

Les deux approches ci-dessus ont un certain nombre d'avantages (⊕) et d'inconvenients (⊖) :

⊕ Le programmeur décrit explicitement l'exécution parallèle du code de l'application. C'est lui/elle qui connaît le mieux la sémantique de l'application, et peut l'utiliser pour exploiter le parallélisme.

⊖ Le fait que le programmeur soit responsable de la synchronisation des activités parallèles rend la programmation plus difficile.

⊖ Le programmeur doit également assurer la cohérence des données vis-à-vis de l'exécution parallèle des activités. Le partage de données doit être explicitement contrôlé par l'intermédiaire de mécanismes du type verrouillage, moniteurs ou sémaphores [And91a].

⊖ Les applications bases de données existantes doivent être réécrites afin d'exploiter le parallélisme.

3.2.1.2 Parallélisme par transformation

Dans le parallélisme par transformation de code, tel que nous l'avons vu dans la section 2.1.3, le compilateur du langage de programmation effectue des transformations dans un programme séquentiel pour pouvoir générer du code parallèle. En général, la parallélisation du code est réalisée en deux étapes :

1. Analyse du code du programme séquentiel pour pouvoir déterminer les parties du programme qui sont parallélisables.

2. Transformation du code en ajoutant des commandes pour la création des activités. Lorsque cela est nécessaire, le compilateur introduit également des commandes pour synchroniser les activités parallèles.

Le parallélisme par transformation de code possède des avantages (⊕) et des inconvénients (⊖) :

⊕ Le parallélisme par transformation cache aux programmeurs les détails de création des activités et, par conséquent, le contrôle d'accès concurrents aux données. Cela simplifie la programmation car le programmeur est libéré de la gestion des activités parallèles.

⊕ La complexité de la programmation parallèle dans l'approche explicite amène souvent à des erreurs de programmation. Lorsque le parallélisme est introduit par le compilateur, le code généré est plus sûr.

⊖ Le parallélisme généré par un compilateur est souvent non-optimal, car il ne connaît pas assez la sémantique de l'application pour pouvoir exploiter au maximum le parallélisme.

Notre approche est inspirée de celle sur laquelle est basé le parallélisme par transformation. La transformation du code d'une transaction consiste, dans un premier temps, à mettre en évidence le parallélisme possible entre les opérations. Ensuite, nous utilisons nos algorithmes de transformation automatique du code de la transaction pour produire un *plan d'exécution* qui prenne en compte le parallélisme intra-transaction.

3.2.2 Mise en évidence du parallélisme

La mise en évidence du parallélisme est basée sur le mode d'accès des opérations vis-à-vis des objets. Par exemple, une opération peut modifier un attribut d'un objet, tandis qu'une autre lit la valeur d'un autre attribut du même objet. Selon leur mode d'accès, certaines opérations d'une transaction sont indépendantes et ainsi n'interfèrent pas les unes avec les autres. Nous disons que ces opérations sont *compatibles*.

La compatibilité entre les opérations est donc utilisée comme critère de correction pour l'exécution d'une transaction avec parallélisme. Nous montrons que l'exécution parallèle des opérations compatibles dans une transaction ne modifie pas le résultat final vis-à-vis de l'exécution séquentielle de la même transaction.

Dans la suite, nous allons définir la relation de compatibilité et montrer comment la compatibilité entre les opérations est déterminée.

3.2.2.1 Environnement des opérations

L'environnement d'une opération d'un type abstrait est composé essentiellement par les attributs du type, les variables locales de l'opération et les paramètres d'entrée.

- Toute opération d'un type peut accéder à tous les attributs du type. Ceux-ci sont donc partagés par toutes les opérations du type. Par exemple, l'opération augmenter du type Employé modifie la valeur de l'attribut salaire, tandis que l'opération déménager lit la valeur du même attribut. L'héritage est pris en compte dans la détermination de l'environnement d'une opération. Par conséquent, les attributs hérités par un type sont inclus dans l'environnement de

toute opération du type. Par exemple, déménager modifie la valeur de l'attribut ville hérité du type Personne.

- Les variables locales sont déclarées dans le code des opérations. Ces variables ne sont visibles qu'à l'intérieur de l'opération dans laquelle elles sont déclarées. Par exemple, la variable res de l'opération virement du type Compte est visible uniquement par le code de virement.

- L'environnement d'une opération est composé également des paramètres formels déclarés dans la signature de l'opération. Notre modèle prévoit uniquement des passages de paramètre par valeur. Par conséquent, les paramètres sont traités comme des variables locales de l'opération. Par exemple, seule l'opération virement peut accéder la valeur de son paramètre ccred.

3.2.2.2 Notation

Fonction partielle :

Considérons $Type = \{$ *type_abstrait, type_atomique, opération* $\}$ et Idf l'ensemble des Identificateurs.

Soient $i \in Idf$ et $t \in Type$, t/i est une fonction partielle qui appartient à l'ensemble des fonctions $ENV = Idf \longrightarrow Type$:

$$t/i = \lambda y.\ si\ y = i \text{ alors } t \text{ sinon la fonction n'est pas définie.}$$

Domaine d'une fonction t/i :

$$Def(t/i) = \{x | t/i(x) \text{ est défini}\}$$

Attributs et opérations d'un type abstrait

La définition d'un attribut d'un type abstrait T est de la forme $a : t$ où a est un identificateur et t est un type abstrait ou atomique. Nous représentons une telle définition à travers une fonction $t/T.a$. Par exemple, la fonction $type_atomique/Compte.soldej$ décrit le fait que soldej est un attribut défini dans le type abstrait Compte et que son type est atomique. Alors que $type_abstrait/Compte.client$ est une fonction qui décrit le fait que client est également un attribut défini dans Compte et que son type est abstrait.

Étant donnée la définition d'un type T, les attributs de T sont représentés par :

$$\alpha^T_{Att} = \bigcup_{i=1}^{n} t_i/T.a_i, t_i \in \{type_atomique, type_abstrait\} \wedge a_i \in Idf$$

$Def(\alpha^T_{Att})$ représente l'ensemble des identificateurs d'attributs du type abstrait T. Par exemple :

$$Def(\alpha_{Att}^{Employé}) = \{Employé.nom, Employé.ville, Employé.salaire\}$$

La définition d'une opération p d'un type abstrait T peut être donnée par une fonction *opération/T.p.* Ainsi la fonction *opération/Compte.virement* décrit le fait que virement est une opération définie dans le type abstrait Compte.

Étant donnée la définition d'un type abstrait T, ses opérations sont représentées par :

$$\alpha^T_{Op} = \bigcup_{i=1}^{n} opération/T.p_i, p_i \in Idf$$

$Def(\alpha^T_{Op})$ est l'ensemble des identificateurs des opérations du type abstrait T. Par exemple :

$$Def(\alpha_{Op}^{Employé}) = \{Employé.pnom, Employé.augmenter, Employé.déménager\}$$

Schéma

Comme nous l'avons expliqué dans la section 3.1.2, un schéma est un ensemble de définitions de types abstraits. Etant donné un schéma S, nous pouvons construire α^S, un ensemble de fonctions partielles où chacune décrit un type abstrait du schéma S. Par exemple, pour le schéma donné dans la section 3.1.1, désormais appelé schéma SE, la fonction *type_abstrait/Personne* appartient à α^{SE} et décrit le fait que Personne est un type abstrait défini dans SE.

Étant donné un schéma S avec n types abstraits, ceux-ci sont représentés par :

$$\alpha^S = \bigcup_{i=1}^{n} type_abstrait/T_i, T_i \in Idf$$

$Def(\alpha^S)$ est l'ensemble des noms de types abstraits du schéma S. Par exemple :

$$Def(\alpha^{SE}_T) = \{Personne, Client, Employé, Compte\}$$

L'ensemble des identificateurs des opérations de tous les types d'un schéma S peut également être donné par :

$$OP^S = \bigcup_{i=1}^{n} Def(\alpha^{T_i}_{Op}), T_i \in \alpha^S)$$

Dans notre schéma exemple SE, l'ensemble de toutes les opérations du schéma est donné par :

$$OP^{SE} = \left\{ \begin{array}{l} Personne.pnom, Personne.déménager, \\ Client.pnom, Client.déménager, \\ Employé.pnom, Employé.augmenter, \\ Employé.déménager, Compte.clientnom, \\ Compte.créditer, Compte.débiter, \\ Compte.virement, Compte.solde \end{array} \right\}$$

3.2.2.3 Le mode d'accès

Lorsqu'on considère les environnements des opérations, les seuls éléments qui peuvent être partagés par plusieurs opérations sont les attributs. De plus, si le principe d'encapsulation est assuré, ce partage ne se fait que dans le contexte d'un type abstrait. Par conséquent, nous déterminons les modes d'accès des opérations d'un type abstrait sur ses attributs. Dans la suite, nous traitons les modes d'accès des opérations pour un seul type abstrait donnée.

Une opération d'un type T peut avoir un mode d'accès *lecture* ou un mode d'accès *écriture* sur un attribut défini dans T :

- *Accès en lecture.* L'opération accède l'attribut sans le modifier. Par exemple, l'opération solde du type Compte lit la valeur de l'attribut soldej et la renvoie comme résultat (cf. figure 3.3). L'opération solde possède donc un mode d'accès lecture sur l'attribut soldej.

- *Accès en écriture.* L'opération modifie la valeur de l'attribut. L'opération créditer est un exemple d'une telle opération puisqu'elle ajoute à la valeur de soldej un crédit passé en paramètre (cf. figure 3.3). Ainsi créditer a un mode d'accès écriture sur l'attribut soldej.

Dans le corps d'une opération, le mode d'accès lecture sur un attribut est marqué par la présence de l'identificateur de l'attribut uniquement dans les instructions du langage qui ne mettent pas à jour sa valeur. L'instruction return soldej de l'opération solde, et l'instruction if (soldej < val) de l'opération débiter sont des exemples de telles instructions. Bien évidement l'ensemble des instructions ayant une telle sémantique est fonction du langage utilisé.

Le mode d'accès écriture sur un attribut est caractérisé par la modification de la valeur de l'attribut dans le code de l'opération. Cela se fait, en général, par l'intermédiaire d'une affectation. L'instruction soldej = soldej + val; de l'opération créditer est un exemple. Cet exemple montre que l'attribut soldej est à la fois lu et écrit par l'opération. Dans ce cas, le mode d'accès est écriture. En général, quand une opération

possède à la fois le mode lecture et le mode écriture sur un attribut, nous considérons l'accès en écriture comme étant le mode d'accès de l'opération sur l'attribut.

Pour chaque opération p d'un type abstrait T, nous associons deux ensembles qui déterminent le mode d'accès de p sur les attributs de T. Le premier comporte tous les attributs de T qui sont accédés par p mais qui ne sont pas modifiés. Le deuxième contient tous les attributs de T qui sont modifiés dans le code de p (mode d'accès écriture).

Soit *mode* une fonction qui délivre le mode d'accès sur un identificateur :

$$mode : Idf \longrightarrow \{lecture, écriture\}$$

Définition 3.1 (Ensembles d'accès) *Soit p une opération d'un type abstrait T, i.e., $p \in Def(\alpha^T_{Op})$, les ensembles d'accès de p sont définis par:*

$écriture_p = \{a \in Def(\alpha^T_{Att}) \mid mode(a) = écriture\}$
$lecture_p = \{a \in Def(\alpha^T_{Att}) \mid mode(a) = lecture \wedge mode(a) \neq écriture\}$

Nous utilisons la fonction β_p pour associer l'opération p à ses ensembles d'accès :

$$\beta_p = p \longrightarrow lecture_p \times écriture_p$$

La fonction β_p appartient à l'ensemble de fonctions $Idf \longrightarrow 2^{Idf} \times 2^{Idf}$ de telle sorte que $\beta_p[1] = lecture_p$ et $\beta_p[2] = écriture_p$

Les ensembles d'accès des k opérations d'un type abstrait T sont donnés par :

$$\beta_T = \bigcup_{i=1}^{n} \beta_{p_i}, p_i \in \alpha^T_{Op}$$

La fonction β_T peut être représentée par un tableau. Le tableau 3.1 montre les ensembles *lecture* et *écriture* pour les opérations du type Compte. Il est important de signaler le cas où une opération envoie un message par l'intermédiaire d'un attribut qui fait référence à un objet. L'attribut appartient à l'ensemble lecture de l'opération. C'est justement le cas de l'attribut client dans l'opération clientnom.

Opération	**lecture**	**écriture**	**ext**
clientnom	{ client }	∅	
créditer	∅	{ soldej }	
débiter	∅	{ soldej }	
virement	∅	∅	
solde	{ soldej }	∅	

TAB. 3.1 - *Ensembles d'accès du type Compte.*

Comme nous l'avons montré dans la section 3.1.1, une opération définie dans un type abstrait peut être redéfinie dans un sous-type. Alors, il peut se produire que,

dans une hiérarchie des types, une même opération ait deux modes d'accès différents sur un attribut.

Par exemple, l'opération déménager dans le type Personne modifie la valeur de l'attribut ville. Par conséquent, ville appartient à l'ensemble écriture de déménager. On considère également que l'opération déménager dans le type Employé modifie l'attribut salaire de l'employé mais ne fait que lire l'attribut ville (l'entreprise ne possède pas de filiale dans d'autres villes). Alors, l'attribut ville appartient à l'ensemble lecture de déménager du type Employé.

Le problème qui se pose par la ré-définition d'une opération est dû à la liaison dynamique. Au moment de la compilation, on ne connaît pas le code de l'opération à exécuter. Celui-ci dépend du type de l'objet vers lequel on envoie un message pour exécuter l'opération redéfinie. Dans notre exemple, si l'on envoie un message pour exécuter déménager vers un objet du type Personne, c'est le code de déménager défini dans le type Personne qui va s'exécuter. De la même manière, si le message est envoyé vers un objet du type Employé, c'est le code défini dans Employé qui va s'exécuter.

Dans notre approche, on unifie les modes d'accès sur les attributs pour les opérations redéfinies en prenant le mode d'accès le plus restrictif. Cela se fait par la promotion des modes d'accès des opérations redéfinies en utilisant l'opérateur de jointure ($\vee$) de la théorie de treillis [Kor83]. Selon cet opérateur, la jointure de deux modes d'accès égaux donne le même mode d'accès. En revanche, la jointure de deux modes d'accès différents est faite de la manière suivante[1] :

1. $\perp \vee \textit{lecture} \Rightarrow \textit{lecture}$
2. $\perp \vee \textit{écriture} \Rightarrow \textit{écriture}$
3. $\textit{lecture} \vee \textit{écriture} \Rightarrow \textit{écriture}$

Dans l'exemple de l'opération déménager, cet opération aura un mode d'accès écriture sur ville tant pour le type Personne que pour le type Employé. Tout envoi de message vers un objet pour exécuter déménager sera donc traité comme si le code à exécuter modifiait la valeur de l'attribut ville.

Il est évident qu'il s'agit d'une approche pessimiste. Cependant, cette approche assure une définition correcte de la relation de compatibilité entre les opérations, sachant que celle-ci est déterminée de manière statique.

3.2.2.4 Fermeture transitive locale

Jusqu'à présent nous avons considéré uniquement les accès sur les attributs présents dans le code d'une opération p. Ces attributs sont directement accédés par p. Toutefois, dans p, il peut y avoir des envois de message à l'objet courant, dénoté par

1. $\perp$ dénote le non accès de l'opération à l'attribut .

self, pour exécuter une des opérations du type. Par exemple, dans le type Compte, l'opération virement envoie un message vers le compte courant pour exécuter l'opération débiter (cf. figure 3.3). Alors, à chaque fois que virement est exécutée, débiter le sera aussi.

En effet, un envoi de message vers self doit être traité de la même façon qu'un envoi de message vers un objet désigné par une variable du même type que self. Par exemple, dans l'opération virement, les envois de message à self et à ccred sont traités comme des envois de message vers le même objet. La raison est simple : on ne connaît pas, au moment de la compilation, le contenu de la variable ccred. Rien n'empêche que ccred, au moment de l'exécution, puisse désigner l'objet courant. Dans ce cas, tous les envois de message à ccred sont également des envois de message à self. Dans la suite, quand on se rapporte à l'envoi de message vers l'objet courant, on considère les envois de message à self, mais aussi ceux utilisant des variables du même type que self.

Dans le cas où une opération p possède dans son code un envoi de message pour exécuter une autre opération q définie dans le même type T de p, nous disons que p "hérite" des ensembles d'accès de q. Autrement dit, si q possède un mode d'accès lecture (écriture) sur un attribut, alors ce mode d'accès est également pris en compte dans p.

L'opération q peut, elle aussi, avoir dans son code des envois de message à l'objet courant pour exécuter d'autres opérations de T et ainsi de suite. Pour déterminer les ensembles d'accès d'une opération p, il faut donc calculer sa fermeture transitive. Il s'agit d'une fermeture transitive locale puisque nous considérons uniquement les envois de messages à des objets du même type que p.

Dans le corps d'une opération p, un envoi de message à l'objet courant pour exécuter une opération q peut modifier le mode d'accès de p sur un attribut de deux façons :

- L'opération q peut engendrer des nouveaux attributs aux ensembles lecture et écriture de p. C'est notamment le cas où l'ensemble lecture (respectivement écriture) de q possède des attributs autres que ceux qui sont déjà dans l'ensemble lecture (respectivement écriture) de p. Nous ajoutons alors les attributs de l'ensemble lecture (respectivement écriture) de q à l'ensemble lecture (respectivement écriture) de p.

- L'opération q peut déplacer des attributs de l'ensemble lecture vers l'ensemble écriture, par une promotion du mode d'accès. Cela se produit lorsqu'un attribut apparaît à la fois dans l'ensemble lecture de p, mais aussi dans l'ensemble écriture de q. La promotion d'un mode d'accès se fait par l'utilisation de l'opérateur

de jointure ($\vee$) de la théorie de treillis (voir section 3.2.2.3).

Nous calculons la fermeture transitive locale d'une opération en considérant son arbre de flot de messages locaux. Un sommet dénote une opération, tandis qu'une arête désigne la présence, dans le corps de l'opération, d'un envoi de message à l'objet courant pour exécuter une des opérations du type. Dans la figure 3.6, on considère que p, q, r, s, t et u sont des opérations d'un même type abstrait. Ainsi, p envoie un message à l'objet courant pour exécuter q et r, r envoie également des messages pour exécuter s et t, et finalement s envoie un message à self pour exécuter u.

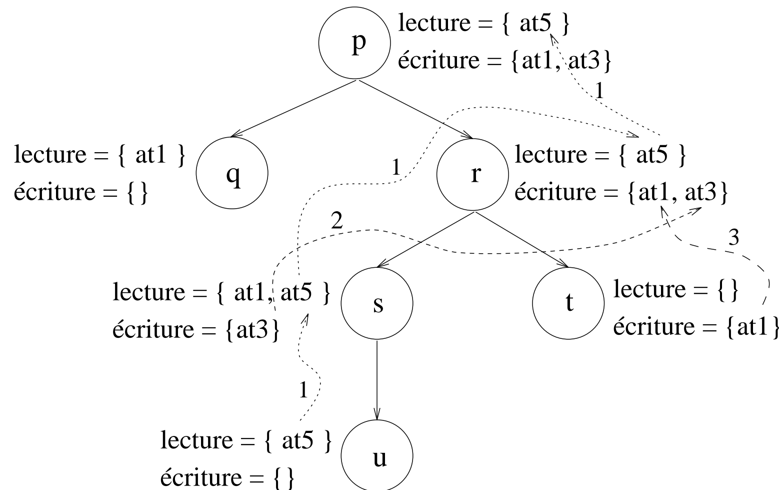


FIG. 3.6 - *Arbre de flot de messages locaux*

Les ensembles d'accès d'une opération sont donc modifiés selon la fermeture transitive locale de l'opération. Prenons l'exemple de la figure 3.6. L'ensemble lecture de l'opération s est augmenté de l'attribut at5 lu dans l'opération u. Cet attribut est ultérieurement ajouté aux ensembles lectures de toutes les opérations au fur et à mesure qu'on monte dans l'arbre jusqu'à l'opération racine (traces numérotés 1). L'attribut at3 est ajouté à l'ensemble écriture de l'opération r car at3 est un élément de l'ensemble écriture de s et r envoie un message à l'objet courant pour exécuter s (trace 2). Par contre, at1 est aussi dans l'ensemble lecture de s. Malgré cela, at1 n'appartient pas à lecture de r car il appartient déjà à l'ensemble écriture de r. Nous pouvons constater que la promotion des modes d'accès fait que les opérations des sommets racines ont toujours les modes d'accès les plus restrictifs sur un attribut de leurs sous-arbres.

Revenons au type Compte. La figure 3.7 montre l'arbre de flot de messages pour l'opération virement. C'est la seule opération qui envoie des messages à l'objet courant (self). L'attribut soldej est alors ajouté à l'écriture de virement puisque soldej appartient à l'ensemble écriture de débiter et virement appelle débiter. Cela peut également

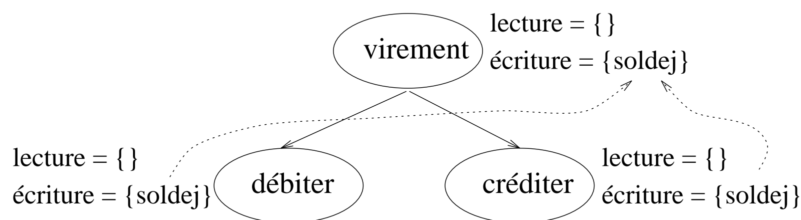


FIG. 3.7 - *Arbre de flot de messages de l'opération virement*

se faire par l'intermédiaire de l'envoi de message à ccred pour exécuter créditer. Le tableau 3.2 montre les nouvelles valeurs des ensembles d'accès des opérations du type Compte. Le symbole $\surd$ dans la colonne **ext** indique que la fermeture transitive de l'opération de la ligne correspondante a été calculée.

Opération	lecture	écriture	ext
clientnom	{ client }	$\emptyset$	$\surd$
créditer	$\emptyset$	{ soldej }	$\surd$
débiter	$\emptyset$	{ soldej }	$\surd$
virement	$\emptyset$	{ soldej }	$\surd$
solde	{ soldej }	$\emptyset$	$\surd$

TAB. 3.2 - *Ensembles d'accès du type Compte après la fermeture transitive.*

Les modifications des ensembles d'accès sont effectuées par l'algorithme ci-dessous. On considère que toutes les opérations de l'arbre de flot de messages sont définies et que leurs ensembles d'accès sont calculés, c'est à dire, β_T est défini.

```
      Algorithm Ensemble d'accès
      Input: L'arbre de flot de messages locaux de p et βT
      Ouput: Les ensembles d'accès de p modifiés
(1)   extend(node, root, βT) {
(2)       if ( node = nil ) then
(3)           root.extended = True;
(4)       else
(5)           while ( node ≠ nil ) {
(6)               extend(succ(node), node, βT );
(7)               βroot[1] = βroot[1] ∪ βnode[1];
(8)               βroot[2] = βroot[2] ∪ βnode[2];
(9)               node = alt(node);
(10)          };
(11)          βroot[1] = βroot[1] - βroot[2];
(12)  }
```

L'algorithme fait un parcours en profondeur de l'arbre de flot de messages. A chaque fois qu'un sommet feuille est atteint, on modifie les ensembles d'accès de son sommet père en tenant compte des ensembles d'accès du sommet feuille. On poursuit le même calcul pour les autres fils du sommet père, jusqu'à ce que tout l'arbre soit traité. Les lignes 7 et 8 de l'algorithme effectuent les modifications des ensembles d'accès du sommet père. La ligne 7 modifie l'ensemble lecture de l'opération désignée par `root` tandis que la ligne 8 modifie l'ensemble écriture de la même opération. Après avoir traité tous les fils d'un sommet, on effectue la promotion des modes d'accès des ensembles du père (ligne 11 de l'algorithme).

3.2.2.5 La relation de conflit

La relation de conflit est une relation binaire entre les opérations d'un type abstrait T. Elle est déterminée statiquement à partir des modes d'accès des opérations donnés par leurs ensembles lecture et écriture. La relation de conflit est vraie entre deux opérations lorsque des conflits d'accès sont détectés dans les opérations sur des attributs du type abstrait.

Définition 3.2 (Relation de conflit) *Deux opérations d'un type abstrait T sont en conflit quand un attribut de l'ensemble écriture de l'une est aussi un élément de l'ensemble lecture ou de l'ensemble écriture de l'autre. Plus formellement :*

$$\begin{aligned}\forall p, q \in \alpha_{Op}^{T}\,[\,conflit(p,q) - \quad & lecture_p \cap écriture_q \neq \emptyset \vee \\ & écriture_p \cap lecture_q \neq \emptyset \vee \\ & écriture_p \cap écriture_q \neq \emptyset\,]\end{aligned}$$

Selon les ensembles lecture et écriture des opérations du type Compte montrés dans le tableau 3.2, les relations de conflit suivantes sont vraies pour les opérations de ce type :

$$\begin{array}{ll} conflit(créditer, créditer) & conflit(créditer, débiter) \\ conflit(créditer, virement) & conflit(créditer, solde) \\ conflit(débiter, débiter) & conflit(débiter, virement) \\ conflit(débiter, solde) & conflit(virement, virement) \\ conflit(virement, solde) & \end{array}$$

La liste ci-dessus montre clairement deux aspects importants de la relation de conflit. Premièrement, une opération dont l'ensemble écriture n'est pas vide est toujours en conflit avec elle même. Cela est facilement prouvé par le fait que si $écriture_p \neq$

$\emptyset$ alors $écriture_p \cap écriture_p$ est toujours non-vide et équivalent à $écriture_p$. Deuxièmement, la relation de conflit est *symétrique*. Par conséquent, si $conflit(créditer, solde)$ est vraie alors $conflit(solde, créditer)$ est également vraie. On peut noter que cette relation n'est pas transitive.

La relation de conflit dans notre approche est comparable aux relation de conflit entre les opérations des TADs que nous avons étudié dans la section 2.3.3. Comme nous l'avons montré dans le chapitre 2, la compatibilité est donnée dans la spécification des TADs. La vérification de la compatibilité est effectuée en temps d'exécution puisque l'objectif consiste en augmenter la concurrence entre les transactions. En revanche dans notre approche la compatibilité des opérations est dérivée de leur code et calculer en temps de compilation.

3.2.2.6 La fermeture transitive au niveau du schéma

Pour les opérations d'un type abstrait, la relation de conflit décrit les conflits d'accès sur les attributs définis dans le type. Cependant, ces opérations peuvent envoyer des messages à des instances d'autres types abstraits. Dans ce cas, l'exécution d'une opération d'un type entraîne l'exécution des opérations des autres types. Il s'agit donc de la fermeture transitive de l'opération, en considérant uniquement des envois de message à des instances des autres types du schéma. Nous l'appelons fermeture transitive au niveau du schéma puisque nous considérons toutes les opérations de tous les types du schéma.

La figure 3.8 présente quatre opérations : p du type T_1, q du type T_2, r et s du type T_3. Les opérations p et q ne peuvent pas être en conflit car elles appartiennent à deux types différents. L'opération p possède, dans son code, un envoi de message vers un objet du type T_3 pour exécuter l'opération r. L'opération q possède également un envoi de message vers un objet du type T_3, mais pour exécuter l'opération s. A chaque fois que p est exécutée, r l'est aussi. De même pour q et s. Considèrons que r et s de T_3 sont en conflit. Lorsqu'on analyse globalement l'exécution de p suivie de r et l'exécution de q suivie de s, on détermine le conflit d'accès entre p et q qui est dû au conflit entre les opérations r et s.

En général, une opération p envoie un message vers un objet d'un autre type désigné par un des composants de l'environnement de p. Ainsi, l'objet cible du message doit être référencé soit par un attribut du type de p, soit par un paramètre ou par une variable locale de p. Le message envoyé engendre l'exécution d'une des opérations du type de l'objet cible. Puisque cette opération possède ces propres ensembles d'accès, nous devons considérer ces ensembles afin de déterminer les effets indirects de l'exécution de l'opération p sur l'état de l'objet cible.

Nous avons deux manières différentes pour déterminer les effets de p sur les objets

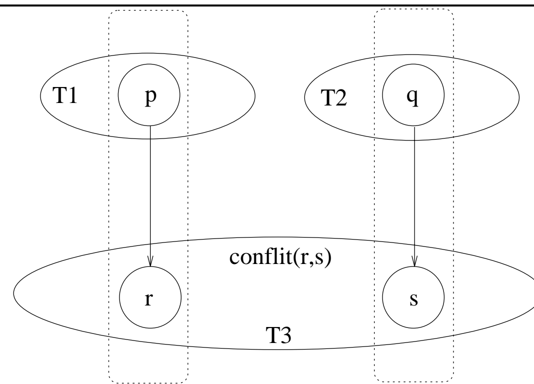


FIG. 3.8 - *Exemple de conflit d'accès indirect*

référencés dans son environnement :

- Modifier les ensembles d'accès des opérations de telle sorte que ceux-ci contiennent des attributs de plusieurs types abstraits. Dans ce cas, une opération possède également des modes d'accès sur des attributs des autres types, sachant que l'accès est toujours indirect, par l'intermédiaire d'un message. Par conséquent, le calcul de la fermeture transitive (voir section 3.2.2.4) pour déterminer les messages à l'objet courant est aussi valable pour les messages vers des objets d'autres types. Cette solution présente les avantages et les inconvénients suivants :

  ⊕ La compatibilité est déterminée uniquement sur la base du mode d'accès des opérations sur les attributs des types abstrait. En principe, il s'agit d'une solution simple.

  ⊖ Cette solution ne respecte pas le principe d'encapsulation des données entre les types abstraits car les attributs définis dans un type abstrait peuvent alors apparaître dans un ensemble lecture (écriture) d'une opération d'un autre type abstrait.

- Associer, à chaque opération p, un ensemble qui contient, outre l'opération p, toutes les opérations des autres types abstraits pouvant être appelées par p par un envoi de message. La compatibilité entre deux opérations est calculée ensuite en termes de la relation de conflit entre les opérations de ces ensembles. Cette solution présente les avantages et les inconvénients suivants :

  ⊕ Cette solution respecte l'encapsulation entre les types puisqu'aucune opération n'a besoin de connaître la structure des types autres que celui dans lequel elle a été définie.

- ⊕ On récupère le travail réalisé pour déterminer la relation de conflit entre les opérations d'un même type pour déterminer la compatibilité entre les opérations.
- ⊖ Une méthode capable de mettre en évidence les interdépendances entre les opérations de types différents doit être définie, puis un nouvel algorithme pour déterminer la compatibilité ayant comme base la relation de conflit et l'interdépendance entre ces opérations doit être réalisé.

Nous choisissons la dernière solution. Avant de définir la relation de compatibilité nous expliquons la façon dont nous construisons les nouveaux ensembles associés aux opérations.

Dans le corps d'une opération, nous nous intéressons donc aux envois de message à des instances de types autres que le type de l'objet courant. Ces messages sont utilisés pour la construction d'un arbre de flot de messages globaux semblable à l'arbre de flot de messages locaux de la section 3.2.2.4. Nous utilisons cet arbre pour déterminer l'interdépendance entre les opérations de plusieurs types, autrement dit, l'arbre représente la *fermeture transitive reflexive* de tous les envois de messages à des objets pour exécuter des opérations des autres types à partir de l'opération courante.

La figure 3.9 donne un exemple d'un arbre de flot de messages globaux. Un sommet dénote une opération et une arête représente un envoi de message. L'opération p du type T_1 contient des envois de messages à des objets des types T_2 et T_3 pour exécuter, respectivement, les opérations q et r. De même, l'opération q de T_2 envoie deux messages vers un objet du type T_3 pour exécuter r et s, et un message vers un objet du type T_4 pour exécuter l'opération t.

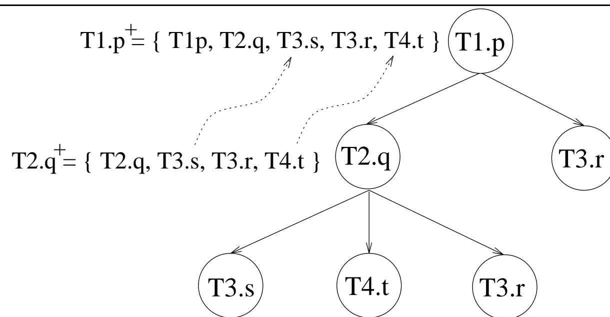


FIG. 3.9 - *Arbre de flot de contrôle global*

La fermeture transitive réflexive d'une opération p est représentée par un ensemble d'opérations, appelé p^+. Dans la figure 3.9, $T_2.q^+$ est composé de $T_2.q$ et de toutes les opérations des autres types qui peuvent être appelées par un des envois de messages depuis $T_2.q$, c'est à dire, $T_3.s$, $T_4.t$ et $T_3.r$. En ce qui concerne l'opération p du type

T_1, elle a deux messages : un vers une instance de T_2 pour exécuter l'opération q et un autre vers une instance de T_3 pour exécuter r. Par conséquent, $T_1.p^+$ inclut les opérations $T_1.p$, $T_2.q$ et $T_3.r$. En plus, par transitivité à travers $T_2.q$, $T_1.p^+$ contient également $T_3.s$ et $T_4.t$ (les pointillés de la figure 3.9). Notez que l'opération $T_3.r$ est directement incluse dans $T_1.p^+$ par un envoi de message dans $T_1.p$

Notation. Nous noterons $p \longrightarrow q$ la présence, dans le code de l'opération p, d'un envoi de message vers un objet d'un type abstrait différent de celui de l'objet courant pour exécuter l'opération q.

Définition 3.3 (Ensemble p^+) *Soit S un schéma,*

$$\forall p, q, r \in OP^S \Rightarrow p^+ = \{q | (p = q) \vee (p \longrightarrow q) \vee (\exists r \mid p \longrightarrow r \wedge q \in r^+)\}$$

Lorsqu'on considère le type Compte, on constate qu'il n'y a qu'une opération qui envoie un message à une instance d'un autre type. Il s'agit de l'opération clientnom qui envoie un message vers une instance du type Client pour exécuter l'opération pnom. L'ensemble $Compte.clientnom^+$ est donc composé par l'opération Compte.clientnom et par l'opération Client.pnom. Pour toutes les autres opérations du type Compte, on a $p^+ = \{p\}$.

3.2.2.7 La compatibilité

La compatibilité est une relation binaire entre les opérations de tous les types abstraits d'un schéma. Nous utilisons la fermeture transitive au niveau du schéma et la relation de conflit pour définir la compatibilité entre les opérations. Deux opérations sont *compatibles* si toutes les opérations du même type appartenant à leur fermeture transitive au niveau du schéma ne possèdent pas de conflit d'accès. Plus formellement :

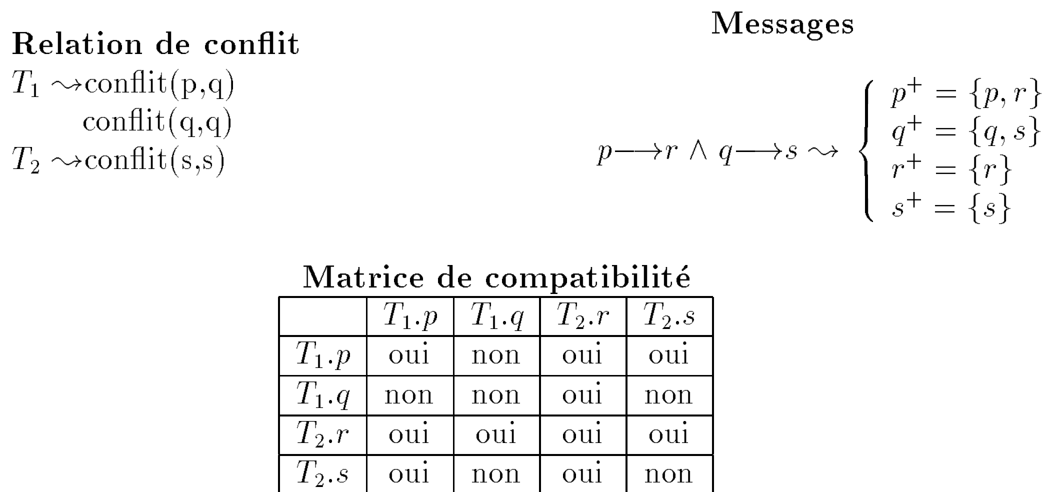
Définition 3.4 (compatible) *Soient S un schéma et T un type abstrait défini dans S,*

$$\forall p, q \in OP^S, compatible(p, q) - \\ \forall r \in p^+, \forall s \in q^+ \Rightarrow [(r \in \alpha^T_{Op} \wedge s \in \alpha^T_{Op}) \Rightarrow \neg(conflit(r, s))]$$

Selon la définition 3.4, la relation de compatibilité dépend directement de la relation de conflit entre les opérations de même type. Deux opérations sont compatibles uniquement si toutes les opérations de leurs ensembles p^+ de même type prises deux à deux ne possèdent pas de conflit d'accès. Deux opérations de types différents ne sont jamais en conflit selon la Définition 3.2). On considère uniquement les opérations de même type.

La relation *compatible*, appliquée à deux opérations du même type abstrait et qui n'ont pas d'envois de messages à des objets d'autres types, est strictement l'inverse de la relation de conflit. C'est notamment le cas où l'ensemble p^+ de chacune des opérations d'un type est égal à l'opération elle-même.

La relation *compatible* est utilisée pour la création d'une matrice de compatibilité entre toutes les opérations d'un schéma. Cette matrice est associée au schéma dans lequel sont définis les types abstraits qui contiennent les opérations. Les lignes et les colonnes de la matrice de compatibilité désignent les opérations. Une entrée de la matrice contient le résultat de la relation compatible appliquée aux opérations représentées par la ligne et la colonne correspondantes.



	$T_1.p$	$T_1.q$	$T_2.r$	$T_2.s$
$T_1.p$	oui	non	oui	oui
$T_1.q$	non	non	oui	non
$T_2.r$	oui	oui	oui	oui
$T_2.s$	oui	non	oui	non

FIG. 3.10 - *Exemple de matrice de compatibilité*

Prenons l'exemple de la figure 3.10. Le type T_1 possède deux opérations, p et q, tandis que le type T_2 possède l'opération r et l'opération s. L'exemple montre la relation de conflit entre les opérations, la fermeture transitive de chaque opération des deux types, et donne la matrice de compatibilité pour ces opérations. Dans le type T_1, p est en conflit d'accès avec q et q est en conflit avec elle-même. En revanche, pour le type T_2, seule l'opération s est en conflit avec elle-même. Il y a deux envois de messages : p envoie un message à un objet du type T_2 pour exécuter r et q envoie un message à un objet de T_2 pour exécuter s. Aucune opération du type T_2 possède des envois de message.

3.3 Parallélisation des transactions

Notre démarche pour la parallélisation du code d'une transaction suit une approche transformationnelle dans laquelle nous partons d'un code séquentiel de la transaction et nous générons un code parallèle équivalent. La parallélisation est réalisée au niveau des opérations de la transaction. Nous utilisons la compatibilité entre ces opérations pour déterminer celles qui peuvent être parallélisées. Le code généré est structuré dans des blocs, en tenant compte de l'ordre total des opérations dans le code source. Chaque bloc regroupe les opérations qui peuvent être parallélisées et qui se suivent dans le code source. Nous synchronisons ces blocs afin qu'ils soient exécutés les uns après les autres. Tous ces aspects sont détaillés dans la suite de cette section.

La section 3.3.1 décrit la représentation du plan d'exécution source d'une transaction, sous la forme d'un graphe. La construction du plan d'exécution parallèle est expliquée dans la section 3.3.2 : nous détaillons l'algorithme de transformation du graphe du plan d'exécution source et nous prouvons que les transformations effectuées assurent la cohérence sémantique du code la transaction. Finalement, la section 3.3.3 présente les mécanismes de base que nous utilisons pour paralléliser et synchroniser les opérations au sein des transactions.

3.3.1 Le plan d'exécution source

La figure 3.11 montre le code de la transaction définie dans la section 3.1.3. Nous le rappelons ici car nous allons utiliser souvent ce code comme exemple de transaction séquentielle qui sera parallélisée.

La définition d'une transaction détermine un plan d'exécution qui établit une relation d'ordre entre les instructions de la transaction. Cette relation est totalement définie puisque le code de la transaction spécifie, dans le premier niveau, toutes les instructions qui doivent être exécutées séquentiellement. Nous représentons ce plan d'exécution par un graphe orienté où les sommets désignent les instructions et les arêtes indiquent la relation d'ordre entre elles. Une arête orientée joint une instruction i à une autre j si j suit immédiatement i dans le code source de la transaction. Dans ce cas, on dit que i est *prédécesseur* de j et que j est *successeur* de i. La figure 3.11 montre le graphe, appelé *Graphe de Plan d'Exécution Séquentielle* (GPES), qui correspond à la transaction de la figure 3.11. Le GPES montre le flot d'exécution d'une transaction.

Chaque type d'instruction est représenté par un sommet de type différent dans le GPES. Les instructions transactionnelles sont représentées par des rectangles arrondies (▯), les instructions de branchement par des losanges ($\diamond$) et les actions par des cercles (◯). L'algorithme de transformation des transactions, que nous proposons ultérieurement, traite chaque type de sommet de manière différente.

```
        Compte c1, c2;
        string nom;
        real a, b;
(1)     commencer;
(2)     nom = c1.clientnom();
(3)     a = c1.solde();
(4)     b = c2.solde();
(5)     if ( a > b*2 ) {
(6)           c1.virement( c2, 1000 );
(7)           b = c2.solde(); }
(8)     valider;
```

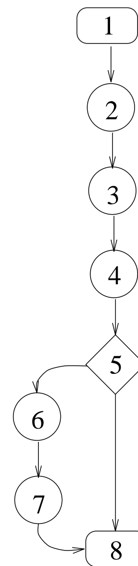


FIG. 3.11 - *Exemple de code d'une transaction séquentielle*

3.3.2 Construction du plan d'exécution parallèle

Notre approche pour la parallélisation du plan d'exécution séquentielle consiste à paralléliser les instructions d'action qui participent à ce plan. Ceci se justifie pour les raisons suivantes :

- Les instructions de branchement présentent, en général, des dépendances de données vis-à-vis des instructions qui les précèdent et/ou celles qui les succèdent [AKPW83, WB87]. Cette dépendance interdit l'exécution parallèle des instructions [LER92, Bra93].
- Les instructions transactionnelles délimitent les transactions. Elles définissent les principaux points de synchronisation dans l'exécution du code des transactions.

Les instructions de branchement et les instructions transactionnelles sont exécutées de manière séquentielle en respectant l'ordre d'exécution donné par le plan source de la transaction.

3.3.2.1 Schéma transactionnel

Une transaction est exécutée dans un environnement d'exécution défini par les règles de portée des noms du langage de programmation. Dans notre modèle cet environnement est appelé *schéma transactionnel* qui est composé de variables déclarées dans le programme d'application et qui sont visibles à l'intérieur de la transaction.

Dans la figure 3.11, les variables c1 et c2 du type Compte, la variable nom du type string et les variables a et b du type real forment le schéma transactionnel de l'exemple.

Les instructions d'action d'une transaction agissent sur les variables du schéma. Comme les opérations d'un type abstrait accèdent et modifient les composants de leur environnement, les instructions d'action d'une transaction peuvent accéder et/ou mettre à jour ces variables.

Les instructions d'action, qui n'envoient pas de message, sont dites *instructions élémentaires*. Celles qui possèdent un envoi de message sont appelées *instructions envoi de message*. Le mode d'accès des instructions élémentaires sur le schéma transactionnel a la même sémantique que le mode d'accès utilisé dans la relation de conflit (voir section 3.2.2.3). En revanche, le mode d'accès des instructions envoi de message est determiné par les modifications que le message effectue sur le schéma transactionnel. Cela peut se faire de trois manières différentes :

- Une variable du schéma transactionnel est la cible de l'envoi de message. Dans ce cas, la variable est forcément d'un type abstrait. Nous considérons que l'instruction a un mode d'accès lecture sur la variable. Par exemple, la transaction de la figure 3.11 possède un mode d'accès lecture sur la variable c1 des lignes 2 et 3.

- Une variable est passée comme paramètre dans un envoi de message. De nouveau, le mode d'accès de l'instruction sur la variable est lecture puisque la valeur de la variable ne peut pas être modifiée par l'opération appelée.[2] Dans la ligne 6 de la figure 3.11, on trouve un exemple d'une telle variable.

- Une variable est modifiée par la valeur retournée par l'exécution d'une opération. On retrouve la sémantique de l'instruction d'affectation. Dans ce cas, l'instruction possède un mode d'accès écriture sur la variable. C'est le cas des variables a et b des lignes 3 et 4 de la figure 3.11.

A partir des modes d'accès des instructions d'action sur les variables du schéma transactionnel, nous pouvons construire les ensembles lecture et écriture pour chacune de ces instructions. La table 3.3 montre les ensembles lecture et écriture des instructions de la transaction de la figure 3.11.

Nous pouvons ensuite appliquer la relation de conflit (Définition 3.2) sur les instructions d'action. La relation de conflit entre deux instructions détermine l'exclusion

2. Rappelons que notre modèle ne prévoit pas de passage par référence. Si la variable est une référence vers un objet, la valeur associée à la variable elle-même est passée par valeur et l'accès au contenu de l'objet référencé est traité par la compatibilité globale.

Instruction	lecture	écriture
(2)	{ c1 }	{ nom }
(3)	{ c1 }	{ a }
(4)	{ c2 }	{ b }
(6)	{ c1,c2 }	∅
(7)	{ c2 }	{ b }

TAB. 3.3 - *Mode d'accès pour les instructions d'action.*

mutuelle dans leur exécution. Lorsque deux instructions sont en conflit, nous interdisons leurs exécutions en parallèle. L'exécution parallèle de deux instructions d'action composant le plan d'exécution source est possible si et seulement si :

- Elles n'ont pas de conflit d'accès sur le schéma transactionnel.
- Les opérations demandées par l'intermédiaire de ces instructions sont compatibles.

Notation L'opération p démandée par l'envoi de message dans une instruction d'action i est dénotée par p_i.

Définition 3.5 (parallélisable) *Soient deux instructions d'action i et j d'une transaction, p et q des opérations, alors :*

$$parall\acute{e}lisable(i,j) - \neg conflit(i,j) \wedge compatible(p_i, q_j)$$

3.3.2.2 L'algorithme de transformation du graphe

La transformation du plan d'exécution séquentielle vers un plan d'exécution avec parallélisme prend en compte la relation d'ordre entre les sommets du GPES et la relation *parallélisable* entre les instructions de la transaction. Cette relation assure l'équivalence entre exécution séquentielle et parallèle de deux instructions.

Nous poursuivons la restructuration du GPES par des transformations successives dans la relation prédécesseur/successeur entre les sommets du graphe. Les transformations structurent donc le GPES dans des blocs des sommets ayant des instructions parallélisables. Ces blocs sont séparés à la fois par des points de synchronisation, mais aussi par des séries des sommets consécutives. L'algorithme de restructuration du GPES crée un graphe d'exécution qui est utilisé dans la génération d'un plan d'exécution pour la transaction.

Nous démarrons depuis le sommet du GPES qui représente le début de la transaction et nous construisons le *graphe de plan d'exécution parallèle* (GPEP) en appliquant l'algorithme suivant :

```
Algorithme : Plan d'exécution parallèle
Données : Le GPES, les ensemble lecture et écriture
          de chaque instruction d'action.
Résultat : Le GPEP

(1)  Créer un ensemble de sommets ESP, initialement vide;
(2)  Tant qu'il existe des sommets à traiter faire;
(3)     prédécesseur = sommet courant;
(4)     Si prédécesseur contient une instruction d'action alors ;
(5)        Ajouter prédécesseur l'ensemble ESP;
(6)        sommet courant = sommet successeur;
(7)        Tant que l'instruction de successeur est parallélisable
              avec toutes les instructions des sommets du ESP faire ;
(8)           Ajouter successeur à l'ensemble ESP;
(9)           pred(successeur) = prédécesseur;
(10           Prendre un nouveau successeur;
(11)       fin;
(12)       Si Cardinalité(ESP) > 1;
(13)          Ajouter point de synchronisation;
(14)          Pour tous les sommets du ESP;
(15)             suc(sommets) = point de synchronisation;
(16)             sommet courant = point de synchronisation;
(17)       sinon
(18)          sommet courant = successeur;
(19)       ESP = ∅;
(20)    fin;
```

Le graphe généré par l'algorithme ci-dessus est utilisé pour construire un plan d'exécution parallèle. L'algorithme structure le code de la transaction de telle sorte que les instructions adjacentes parallélisables forment des blocs suivis de points de synchronisation. Les instructions dans ces blocs vont être exécutées en parallèle. Ces blocs sont représentés par l'ensemble des sommets ESP (abrégé de Ensemble de Sommets Parallélisables) de l'algorithme (lignes 1, 5, 7, 11, 18). L'objectif de l'ensemble ESP est de rassembler tous les sommets d'un bloc dont les instructions sont parallélisables. Lorsqu'un sommet ayant une instruction d'action est atteint, l'algorithme

commence à remplir l'ensemble ESP, jusqu'à ce que l'instruction du prochain sommet possède une des propriétés suivantes (ligne 7) :

- L'instruction n'est pas une instruction d'action. Cela veut dire qu'il s'agit d'une instruction de branchement ou transactionnelle.
- C'est une instruction d'action, mais elle n'est pas parallélisable avec *une* des instructions déjà dans l'ensemble ESP courant.

Selon la ligne 7 de l'algorithme, si l'instruction du prochain sommet n'est pas une action, la condition pour passer à la ligne 8 n'est pas satisfaite. De plus, cette condition vérifie également la relation parallélisable entre l'instruction du prochain sommet et toutes celles des sommets de l'ensemble ESP.

Les lignes 11 à 15 de l'algorithme servent à créer un point de synchronisation *après les blocs d'instructions parallèles*. Il est possible que l'ensemble ESP possède une seule instruction d'action. C'est notamment le cas où l'algorithme ne rentre pas dans la boucle des lignes 7 à 10, car l'instruction du prochain sommet du GPES possède une des propriétés citées ci-dessus. Néanmoins, les points de synchronisation ne sont nécessaires que lorsqu'ils sont précédés par un bloc d'instructions parallélisables ayant au moins deux instructions. Nous testons alors la cardinalité de l'ensemble ESP afin d'être sûrs que le bloc avant le point de synchronisation contient plusieurs instructions parallélisables.

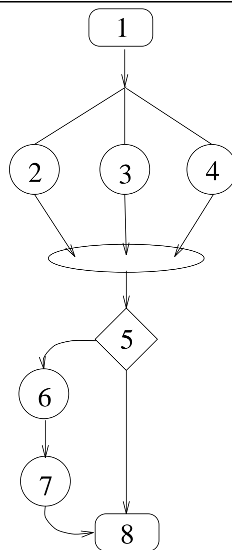


FIG. 3.12 - *Graphe de plan d'exécution parallèle*

Appliquons l'algorithme au GPES de la figure 3.11. Le graphe de plan d'exécution parallèle généré par l'algorithme est montré dans la figure 3.12. L'ovale représente un

point de synchronisation. Les instructions des sommets 2, 3 et 4 sont parallélisables, car elles sont toutes des instructions d'action, il n'y a pas de conflit entre elles et elles sont compatibles. Comme elles sont adjacentes dans le GPES, l'algorithme les parallélise dans le nouveau graphe. Ensuite, il insère un point de synchronisation avant de générer l'instruction du sommet 5, qui est une instruction de branchement.

Les instructions des sommets 6 et 7 sont aussi des instructions d'action et ne présentent pas de conflit d'accès sur le schéma transactionnel (voir le tableau 3.3). Par contre, elles ne sont pas compatibles, puisqu'elles sont en conflit (voir section 3.2.2.5, pour les opérations du type Compte à la page 75). L'algorithme maintient cette partie du graphe telle qu'elle était dans le GPES.

Dans la suite, nous montrons l'application de l'algorithme sur des exemples de GPES afin d'expliquer le traitement des différents types d'instructions de branchement.

Instructions itératives

La figure 3.13 montre le GPES d'une transaction où il y a une itération dont les deux instructions sont parallélisables. Le GPEP généré place le point de synchronisation juste après ces instructions mais avant le nouveau passage dans l'itération. Le code représenté par le GPEP assure donc la synchronisation de l'exécution du code de la transaction à chaque fois que le test de l'itération doit être faite.

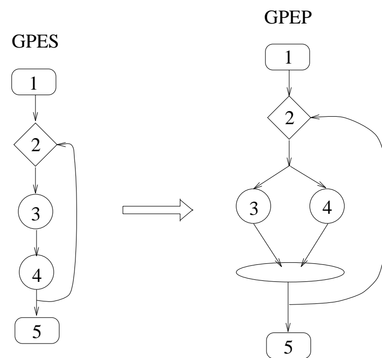


FIG. 3.13 - *Transformation des itérations*

Instructions conditionnelles

Le GPES de la figure 3.14 montre une structure conditionnelle avec deux branches alternatives (si-alors-sinon). Selon la structure, les instructions des sommets 5 et 6 ne seront jamais exécutées si celles des sommets 3 et 4 le sont, et vice-versa. Par conséquent, la relation *parallélisable* doit être vraie pour les instructions d'action de chaque branche, de manière indépendante. Pour que les instructions 3 et 4 soient parallélisées, il suffit que leurs instructions soient parallélisables entre elles, indépendamment de celles des sommets 5 et 6. Il en est de même pour les sommets 5 et 6 vis-à-vis

des sommets 3 et 4. Après une branche avec des sommets parallélisés, l'algorithme introduit un point de synchronisation avant de passer à l'instruction suivante.

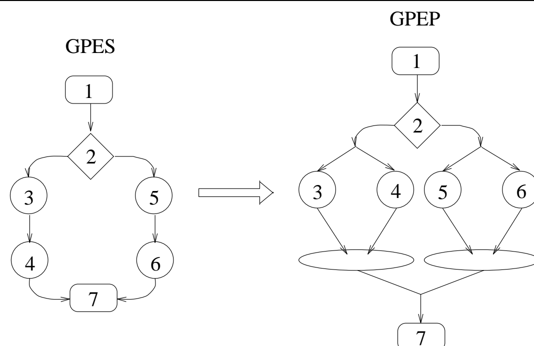


Fig. 3.14 - *Transformation des branches alternatives*

3.3.2.3 La cohérence sémantique des transactions

La construction du plan d'exécution parallèle préserve la cohérence du plan d'exécution séquentielle. Cela veut dire que les résultats de l'exécution d'une transaction séquentielle sont *équivalents* à ceux de l'exécution du code de la même transaction, après qu'elle ait été transformée par l'algorithme que nous venons de décrire. Nous assurons que les transformations effectuées sur le code source d'une transaction préserve la logique du code de la transaction, ce que nous prouvons dans la suite.

La notion d'équivalence entre le code source séquentiel et le code parallèle généré s'appuie sur la relation *parallélisable* entre les instructions qui forment la transaction. Dans notre algorithme, cette relation garantit l'exclusion mutuelle des instructions qui ne peuvent pas s'exécuter en parallèle. Ainsi, l'exécution séquentielle de deux instructions parallélisables rend les mêmes résultats que l'exécution parallèle des mêmes instructions.

Le GPEP ne montre que le premier niveau de parallélisme entre les instructions d'une transaction. Il est évident qu'un envoi de message vers un objet dans une instruction d'action peut engendrer l'exécution de plusieurs opérations des différents types abstraits. Le parallélisme entre deux instructions dans un premier niveau peut traduire, en effet, l'exécution parallèle de plusieurs opérations. C'est pour cela que nous utilisons la relation *compatible* dans la définition de la relation *parallélisable*. Ainsi, toutes les opérations qui sont appelées depuis les instructions parallélisées sont compatibles, donc n'interfèrent pas les unes avec les autres.

Notation Nous utilisons $\rightarrow$ pour dénoter l'exécution séquentielle et $\|$ pour dénoter l'exécution en paralléle de deux instructions.

Théorème 3.1 *Si deux instructions i et j sont parallélisables, alors $i \rightarrow j \equiv i \| j$.*

Preuve. La preuve est immédiate par construction de la relation *parallélisable* entre les instructions i et j car cette relation assure l'indépendance du code de i et de j. Si parallélisable(i,j) est vraie alors i et j ne modifient pas les mêmes données. Par conséquent, l'exécution de i puis j rend le même résultat que l'exécution parallèle de i et de j. □

L'algorithme peut paralléliser plus de deux instructions dans un bloc d'instructions parallèles. Cependant, il exige que toutes les instructions du bloc soient parallélisables, prises deux à deux. Cela est nécessaire car la relation *parallélisable* n'est pas transitive. Il est possible que $parall\acute{e}lisable(a, b)$ et $parall\acute{e}lisable(b, c)$ soient vraies, mais $parall\acute{e}lisable(a, c)$ soit faux. Il suffit qu'il ait un conflit d'accès au schéma transactionnel entre a et c pour que cela se produise.

Théorème 3.2 *Si les n instructions d'un ensemble I sont toutes parallélisables prises deux à deux, alors*

$$\forall i_j \in I, 0 < j \leq n, i_1 \rightarrow, \ldots, \rightarrow i_j \rightarrow, \ldots, \rightarrow i_n \equiv i_1 \|, \ldots, \| i_j \|, \ldots, \| i_n.$$

Preuve. Aucune des instructions de l'ensemble I n'intervient dans l'exécution des autres instructions du même ensemble, car elles sont toutes parallélisables entre elles. Par conséquent, l'exécution séquentielle de l'ensemble des instructions donne le même résultat que l'exécution parallèle des mêmes instructions. □

Si les instructions des blocs parallèles sont parallélisables, l'ordre total d'exécution entre elles n'est pas nécessaire pour assurer un comportement cohérent de toute la transaction. Alors la relation d'ordre total entre les instructions d'une transaction séquentielle peut devenir une relation d'ordre partiel dans une transaction parallèle sans mettre en cause le résultat final de la transaction[3].

Le non-déterminisme introduit par l'exécution parallèle des instructions est contrôlé par les points de synchronisation. Ainsi l'ordre d'exécution défini initialement dans le GPES est globalement maintenu dans le GPEP grâce aux points de synchronisation. Le fait que les instructions de branchement ne soient pas parallélisables assure le synchronisme dans le flot d'exécution de la transaction à chaque fois qu'une décision doit être prise. Ainsi, on ne exécute pas une instruction d'une boucle en même temps que la condition de la boucle est évaluée.

Théorème 3.3 *L'exécution du code séquentiel d'une transaction est équivalente à l'exécution du code parallèle de la même transaction, transformée par l'algorithme introduit dans la Section 3.3.2.2.*

3. On peut trouver une définition mathématique de la relation d'ordre partiel dans [BHG87]

Preuve. Immédiat du Théorème 3.1, du Théorème 3.2 et de la cohérence de l'ordre d'exécution de la transaction maintenue par les points de synchronisation. □

3.3.3 Les mécanismes de base pour la parallélisation

Les transformations réalisées sur le code source d'une transaction séquentielle introduisent du *parallélisme asynchrone* [Per87, Bra93] et des mécanismes de synchronisation tels que les *barrières* [Les93, Cos93]. Essentiellement, ces transformations remplacent les instructions parallélisables par des codes qui vont exécuter ces instructions de manière asynchrone. L'exécution asynchrone est ensuite synchronisée par les barrières placées à chaque point de synchronisation du code transformé.

Dans une transaction séquentielle, l'envoi de message transfère le flot de contrôle de la transaction vers l'opération appelée afin de l'exécuter. Comme l'envoi de message est synchrone, l'exécution du code de la transaction est bloquée, en attendant que l'exécution de l'opération soit terminée (cf. figure 3.15). Dans notre modèle, l'exécution des instructions parallélisées ne bloque pas le flot de contrôle de la transaction, au contraire, le code de la transaction continue son exécution en parallèle avec les instructions parallélisées (cf. figure 3.15).

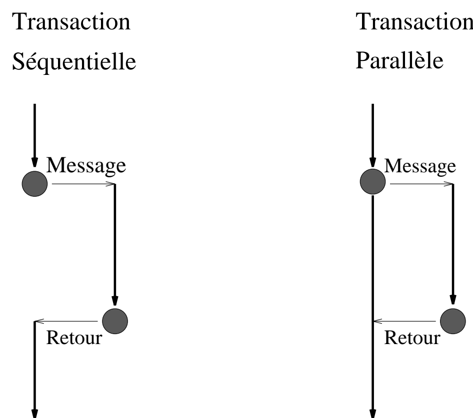


FIG. 3.15 - *Flot de contrôle avec parallélisme*

3.3.3.1 La création des activités

Le parallélisme asynchrone est accompli par la création de *processus légers*, dans lesquels sont exécutées les opérations demandées par les instructions parallélisables. Nous les appelons *activités*. Une activité s'exécute de manière séquentielle et partage le schéma transactionnel avec les autres activités qui sont en train d'être exécutées. Une fois l'opération terminée, son activité correspondante est détruite.

L'exécution d'une transaction transformée peut avoir une configuration comme celle qui est montrée dans la figure 3.16. Au début, toute transaction possède une seule activité. Au fur et à mesure que la transaction avance dans son exécution, plusieurs activités sont créées puis détruites. Le code généré structure donc l'exécution de la transaction comme une succession d'activités séquentielles et parallèles. Cette structuration apporte deux aspects importants :

- Le contrôle du non déterminisme introduit par le parallélisme asynchrone.
- A la fin d'une transaction, une seule activité sera active, puisqu'on synchronise toutes les activités avant d'exécuter le code séquentiel correspondant à une instruction transactionnelle.

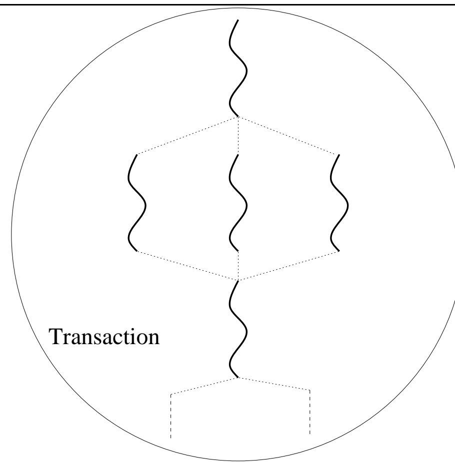


FIG. 3.16 - *Exemple de configuration d'une transaction*

Nous ne créons pas une activité pour chacune des instructions parallélisables. En particulier, les instructions d'action élémentaires d'un bloc d'instructions parallélisables ne sont pas exécutées chacune dans une activité. La raison en est simple et s'explique par le fait que la création d'une activité pourrait engendrer trop de surcharge par rapport à l'exécution d'une instruction élémentaire. Par conséquent, seule les instructions d'envoi de message sont exécutées dans des nouvelles activités parallèles. Les instructions élémentaires parallélisables s'exécutent les unes après les autres au sein de l'activité de la transaction qui continue toujours son exécution après avoir créé les nouvelles activités.

La figure 3.17 présente le GPEP d'une transaction avec un bloc de 5 instructions parallélisables. Parmi celles-ci, les instructions 3 et 4 sont élémentaires. Les autres sont des instructions d'envoi de message. Ainsi, le code généré engendre trois nouvelles activités pour exécuter, chacune, une instruction envoi de message. L'activité

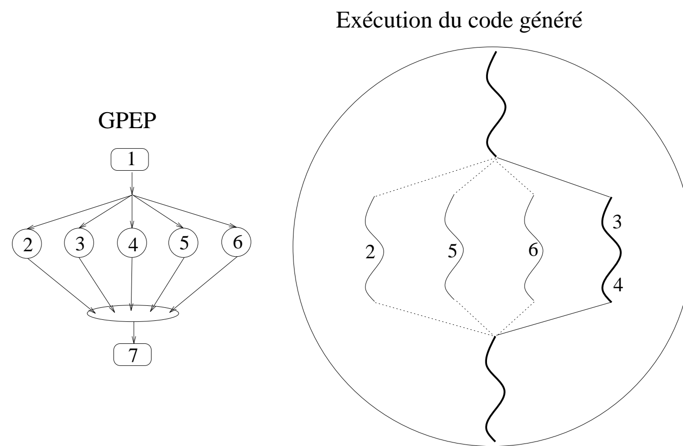


FIG. 3.17 - *Exécution des instructions élémentaires*

de la transaction continue son exécution en parallèle avec les nouvelles activités. C'est l'activité de la transaction qui va donc exécuter les deux instructions élémentaires.

3.3.3.2 La synchronisation des activités

Nous synchronisons les activités parallèles à travers une variation des barrières. Une barrière est définie comme un point dans un programme où tous les processus parallèles s'attendent les uns les autres [Les93]. Dans notre modèle, c'est l'activité principale qui attend l'exécution de toutes les activités qu'elle a créé, pour continuer son exécution. Par conséquent, l'activité principale joue le rôle de coordonnateur de l'exécution de la transaction. Elle est responsable de la création de toutes les autres activités et de leur synchronisation à travers les barrières.

```
(1)  commencer;
(2)  lwp_create(p2, c1.clientnom, nom, 0);
(3)  lwp_create(p3, c1.solde, a, 0);
(4)  lwp_create(p4, c2.solde, b, 0);
(5)  barrier(p2, p3, p4);
(6)  if ( a > b*2 ) {
(7)       c1.virement( c2, 1000 );
(8)       b = c2.solde(); }
(9)  valider;
```

FIG. 3.18 - *Exemple de transformation du code d'une transaction*

La figure 3.18 montre le code de la transaction de la figure 3.11 après les transformations que nous venons d'expliquer. Ce code correspond au GPEP donné dans

la figure 3.12. Les lignes 2, 3 et 4 créent chacune une activité parallèle pour exécuter une instruction envoi de message. L'identificateur de l'opération à exécuter est passé comme paramètre au moment de la création de l'activité. Les autres paramètres servent à :

- Identifier l'activité.
- Passer l'identificateur qui doit recevoir le résultat de l'exécution de l'opération.
- Donner le nombre de paramètres passés à l'opération.
- Les paramètres de l'opération.

La ligne 5 montre l'utilisation d'une barrière. L'activité principale engendre les trois activités des lignes 2 à 4 et continue son exécution. Lorsque l'activité principale atteint la ligne 5, elle se bloque en attendant que les activités créées soient finies. Les activités sur lesquelles elle est bloquée sont identifiées par les identificateurs d'activité passés comme paramètres pour la barrière. Lors de la fin des activités p2, p3 et p4, l'activité principale continue normalement son exécution jusqu'à la fin de la transaction.

3.4 Conclusion

Dans ce chapitre nous avons présenté notre modèle pour la parallélisation des transactions séquentielles. Tout d'abord nous avons décrit un formalisme permettant de définir les modes d'accès des opérations sur les attributs et calculer la relation de compatibilité entre les opérations. Deux ensembles d'accès contiennent respectivement les attributs lus et ceux modifiés par une opération. Ces ensembles ne concernent que les attributs qui sont directement ou indirectement manipulés par l'opération. Les attributs manipulés de manière indirecte sont données par un algorithme que calcule la fermeture transitive locale d'une opération.

Nous avons défini une relation de conflit entre les opérations d'un type fondée sur leurs ensembles d'accès. Toutefois, une opération d'un type peut appeler des opérations d'autres types. Nous avons donc calculer la fermeture transitive d'une opération par rapport aux opérations définies dans les autres types du schéma. Cette fermeture transitive et la relation de conflit sont la base pour la définition de la compatibilité entre les opérations.

Nous avons ensuite détaillé notre démarche pour la transformation du code des transactions. Nous construisons d'abord un graphe que représente le code d'une transaction. Ce graphe est ensuite transformé par un algorithme qui utilise la relation

de compatibilité. Le graphe transformé est la base pour la génération du code de la transaction avec parallélisme. Finalement nous présentons les primitives utilisées pour créer des activités et pour les synchroniser dans cette transaction.

Notre modèle peut être utilisé dans des systèmes de base de données à condition qu'on respecte l'encapsulation des objets. L'utilisation des mécanismes qui permettent l'accès directe à l'état des objets met en cause la définition de la compatibilité des opérations, surtout lorsqu'on considère les opérations de différents types abstraits.

Le calcule de la relation de compatibilité par analyse de code rend aisé la spécification et la mise en œuvre des types abstraits, par opposition à d'autres approches où la matrice de compatibilité doit être spécifiée. Le fait que cette relation soit statiquement déterminée nous amène à se baser sur certaines hypothèses pessimistes que nous avons expliqués au cours de ce chapitre. En revanche, une approche dynamique exigerait une gestion de verrouillage des objets différentes et complémentaires à celle déjà mise en œuvre par le système de base de données. La relation entre les deux gestionnaires n'étant pas évidente, nous avons choisi pour la définition statique de la compatibilité. De plus, l'approche dynamique résulterait dans des surcharges importantes en temps d'exécution, que nous avons éliminé avec l'approche statique.

Notre approche produit de l'exécution asynchrone des opérations dans une transaction sans introduit des problèmes liés aux non déterminisme. Nos techniques de création et de synchronisation des activités assurent d'une part l'isolation des activités et d'autre par l'existence d'une seule activité au début et à la fin d'une transaction.

Chapitre 4

Expérimentation

4.1 O_2Parall 97
4.1.1 Architecture logique 98
4.1.2 Fonctions générales 99
4.1.3 Gestion du catalogue 101
4.1.3.1 Gestion des entités nommées 101
4.1.3.2 Gestion des types 102
4.1.3.3 Gestion des classes et des méthodes 103
4.1.4 Compatibilité des méthodes 105
4.1.4.1 Vecteurs d'accès 106
4.1.4.2 Matrice de compatibilité 108
4.1.5 Transformation des transactions 110
4.1.5.1 Applications et transactions 111
4.1.6 Modules du système 112
4.1.6.1 Le module Toolbox 114
4.1.6.2 Le module CatalogueMgr 114
4.1.6.3 Le module TrParallelyser 115
4.2 Parallélisation des règles 115
4.2.1 Règles du système NAOS 116
4.2.1.1 Définition des règles 116
4.2.1.2 Traitement de règles 117
4.2.2 Parallélisation des règles 119
4.2.2.1 Compatibilité de règles 120
4.2.2.2 Exécution parallèle des règles 122
4.2.3 Architecture du système 123
4.2.3.1 Générateur de tables de compatibilité 124
4.2.3.2 Parallélisateur de règles 124
4.3 Conclusion 125

Expérimentation

Ce chapitre présente les implantations du modèle de parallélisation des transactions décrit dans le chapitre 3. La section 4.1 présente l'implantation faite dans le contexte du SGBD O_2. La section 4.2 présente l'application de notre modèle au système de règles actives NAOS.

4.1 O_2Parall

Nous avons utilisé le système de gestion de bases de données à objets O_2 [AC93, Tec94] comme système de base. Le prototype est une application C++ [GOP92, Str92] qui communique avec le système O_2 par son interface O_2API. Cette interface permet de manipuler un schéma ou des entités O_2 par l'intermédiaire d'un ensemble de primitives C.

L'utilisation du système O_2 présente à notre avis les avantages et les inconvenients suivants :

- ⊕ Nous avons pu développer une *application test* pour le prototype. Cette application codée en C++ utilise également l'interface O_2API. Elle possède un schéma, une base O_2 et est composée des transactions de récupération et de mise à jour des objets de la base. Les transactions et le schéma de l'application test sont transformés en données d'entrée du prototype. L'exécution du prototype montre les transformations à faire sur les transactions de l'application test afin de pouvoir exécuter leurs méthodes en parallèle.

- ⊕ Le choix du système O_2 en particulier s'appuie d'abord sur la disponibilité du système et sur l'expérience de recherche avec ce système déjà acquise par d'autres membres de l'équipe. De plus, l'interface O_2API offre des fonctions d'accès à toutes les données d'un schéma dont nous avons besoin.

- ⊖ nous ne disposions pas du source de l'analyseur syntaxique du langage O_2. Par conséquent, certains fonctions expliquées dans le chapitre 3 n'ont pas pu être réalisées ou ont eu leurs données d'entrée fournies directement dans le prototype.

C'est le cas notamment des fonctions qui dépendent directement de l'analyse du code des méthodes.

La section 4.1.1 décrit l'architecture logique de notre prototype. La section 4.1.3 détaille la gestion des définitions d'un schéma O_2. Dans la section 4.1.4 nous décrivons la création et la gestion des tables de compatibilités des méthodes. La section 4.1.5 décrit la transformation des transactions des applications. Finalement la section 4.1.6 montre les modules d'implantation de notre prototype.

4.1.1 Architecture logique

Nous modélisons les concepts liés à notre système en utilisant la notation donnée par la méthode orientée-objet de spécification de systèmes proposée par Booch [Boo94]. Nous avons choisi cette méthode pour deux raisons. Tout d'abord, il s'agit d'une méthode riche en concepts, permettant ainsi de modéliser tous les aspects liés au paradigme d'objet. Malgré sa richesse en concepts, la méthode fournit un ensemble de concepts dits *essentiels* qui permet une modélisation simple et propre du système. Deuxièmement, la méthode est plus adaptée à l'étape de conception du cycle de vie d'un logiciel, justement l'étape que nous intéressait le plus. Nous décrivons plus en détails la méthode et sa notation dans l'annexe A.

La modélisation de notre système suit une approche descendante où nous montrons d'abord les catégories des classes et nous décrivons ensuite les classes et leurs associations. Cette approche permet de donner d'abort une vision globale, fonctionnelle, du système et ensuite de la détailler par la description des structures des classes.

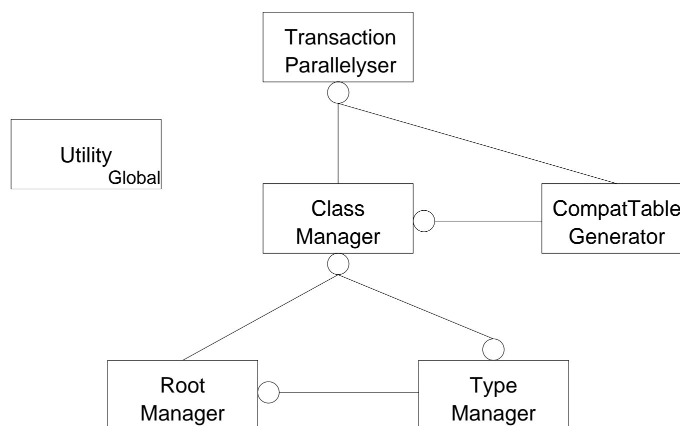


FIG. 4.1 - *Diagramme de catégories de classes*

Les classes de notre système sont groupées dans 6 catégories (voir figure 4.1). Les

relations entre les catégories montrent l'utilisation des services. Par exemple, la catégorie Class Manager utilise les services de CompatTable Generator. En fait, les relations dans les diagrammes de catégories sont une abstraction des relations d'utilisation des services dans les diagrammes de classes. Cela explique la même notation pour les relations dans les deux types de diagrammes.

Chaque catégorie rassemble des classes dans l'objectif d'accomplir un service du système. Par exemple, les classes de la catégorie Transaction Parallyser effectuent la transformation des transactions, tandis que les classes de Type Manager gèrent les définitions de types du schéma d'une application. La table 4.1 résume les catégories des classes.

Catégories	*Classes*	*Fonctions*
Utility	O2, PO2, Schema, PO2Error	Fonctions générales, connection avec le Système O_2
Root Manager	SchemaRoot, Root	Gestion des entités nommées
Type Manager	SchemaType, Type, AtoType Tuple, Attribute, Collection	Gestion des types du schéma
Class Manager	SchemaClass, PClass Method, Param	Gestion des classes, construction de la matrice de compatibilité, transformation des transactions
CompatTable Generator	MethodCompat, AccessMode	Construction de la matrice de compatibilité
Transaction Parallelyser	Application, Transaction Vertex	Transformation des transactions

TAB. 4.1 - *Les catégories, leurs classes et fonctions.*

Si l'on considère que les classes des catégories Class Manager, Type Manager et Root Manager effectuent la gestion des définitions d'un schéma O_2, nous pouvons partager le système en quatres services principaux que nous détaillons par la suite :

1. Les fonctions générales.

2. La gestion du catalogue.

3. La construction de la matrice de compatibilité.

4. La transformation des transactions.

4.1.2 Fonctions générales

Les fonctions générales sont accomplies par les classes de la catégorie Utility. Cette catégorie est composée des classes utilitaires. Le rôle de ces classes consiste à fournir

des services communs utilisés par la plupart des classes des autres catégories. Ainsi, cette catégorie est identifiée comme *globale* (voir figure 4.1). Les services fournis sont essentiellement la gestion des collections d'objets C++, la gestion des exceptions et l'interface avec le Système O_2. La catégorie Utility est composée des classes OrderedCltn, O2[1], PO2, Schema et PO2Error (cf. figure 4.2).

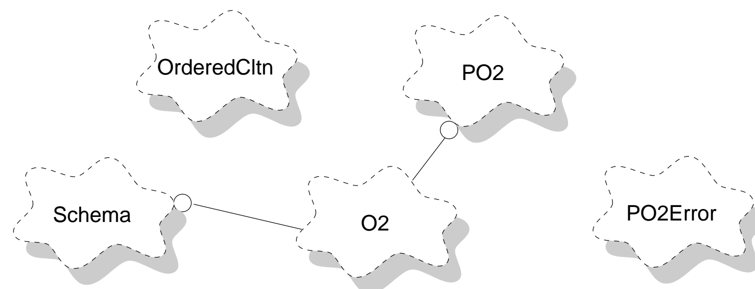


FIG. 4.2 - *Diagramme des classes utilitaires*

La classe OrderedCltn est utilisée pour créer des collections ordonnées d'objets d'une même classe C++. Une collection peut avoir des instances d'Object ou des instances d'une des sous-classes d'Object. Par conséquent, toutes les classes, dont les instances sont groupées dans des collections, doivent être forcement sous-classes d'Object. Lorsqu'on veut modéliser un attribut d'une classe comme une collection d'objets, on va le définir comme étant un pointeur vers un objet de OrderedCltn. Cette classe fournit un ensemble de méthodes pour insérer des objets dans une collection et pour les récupérer.

Par exemple, la figure 4.3 montre une partie de la définition des classes Attribute et Tuple. Une instance de la classe Tuple fait référence à un ensemble d'instances de la classe Attribute pour représenter le fait qu'un n-uplet est composé de plusieurs attributs. Alors, un attribut de la classe Tuple est une réference vers un objet de la classe OrderedCltn qui contient un ensemble d'instances d'Attribute. Cela est possible car Attribute est sous-classe d'Object.

La classe Schema permet d'établir une connexion avec un serveur O_2 et d'ouvrir un schéma et une base sur lesquels s'exécute notre système. Pour cela, nous utilisons les services fournis par la classe O2. Cette classe modélise une interface fonctionnelle fournie par le Système O_2. Par conséquent, elle ne possède pas de structure et présente uniquement un ensemble de déclarations de fonctions C mises en œuvre par O_2.

La classe PO2 permet la création de plusieurs types enumération utilisés dans notre système et l'affichage de certaines propriétés des objets. Finalement la classe PO2Error

1. Nous utilisons la notation O2 pour référencer la classe qui modélise le Système O_2.

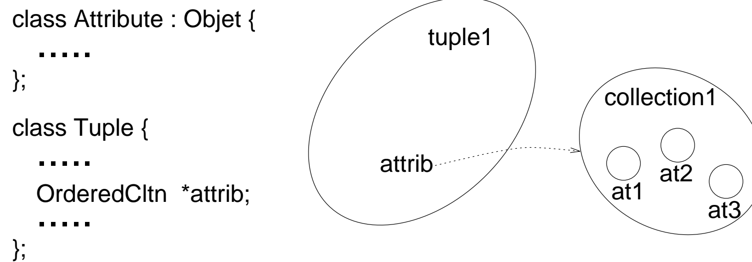


FIG. 4.3 - *La modélisation des collections d'objets*

modélise les exceptions du système. A chaque fois qu'une exception se produit, une instance de cette classe est créée et un message pour traiter la cause de l'exception lui est envoyé.

Dans nos diagrammes de classes, nous ne représentons pas les relations entre les classes utilitaires et celles des autres catégories. Cela simplifie les diagrammes et, par conséquent, augmente la compréhension des concepts du domaine. Une partie de ces relations sera montrée à travers les diagrammes d'objet où l'on vera apparaître les échanges de messages entre les instances des classes utilitaires et celles des autres classes du système.

4.1.3 Gestion du catalogue

Les applications transformées par notre système possèdent un *schéma* de base de données défini à l'aide du système O_2. Nous chargeons ce schéma dans des structures de données C++ afin de faciliter la manipulation des informations nécessaires à la parallélisation. Nous avons, donc, une *image*, en mémoire sous la forme d'un ensemble d'objets C++. Nous l'appelons *catalogue de l'application*.

La gestion du catalogue est accomplie par les classes des catégories Root Manager, Type Manager et Class Manager.

4.1.3.1 Gestion des entités nommées

La catégorie Root Manager gère les entités nommées ou racines de persistance définies dans le schéma. Cette catégorie est composée des classes SchemaRoot et Root. La classe SchemaRoot met en œuvre des méthodes permettant la gestion d'un ensemble d'entités nommées d'un schéma (figure 4.4). Un objet de cette classe possède un attribut dont la valeur est une référence à une collection d'objets. Cette collection contient des instances de la classe Root qui modélise les entités nommées du schéma de l'application. Certaines méthodes de la classe SchemaRoot ont pour but de récupé-

rer toutes les informations d'un schéma O_2 concernant les entités nommées. D'autres méthodes fournissent ces informations aux autres classes de notre système.

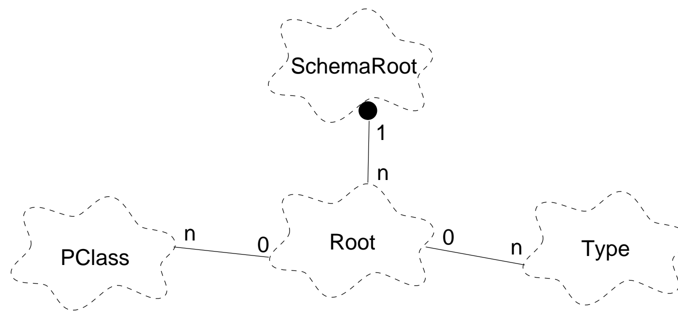


FIG. 4.4 - *Les entités nommées*

4.1.3.2 Gestion des types

La catégorie Type Manager gère les types du catalogue : les types pre-définis, utilisateurs et les types des classes. Type Manager offre des services à la catégorie Class Manager, lorsqu'un service réalisé par une des classes de cette catégorie nécessite la définition d'un type. Type Manager offre également des services à la catégorie Root Manager, utilisés pour déterminer le type des valeurs nommées. En revanche, Type Manager utilise les services de Class Manager lors de la participation d'une classe dans la définition d'un type. Les relations de Type Manager avec les autres catégories de classes sont présentées dans la figure 4.1.

La catégorie Type Manager est en fait le serveur de types du catalogue : ses classes,SchemaType, Type, AtoType, Tuple, Collection et Attribute, offrent les définitions de types aux classes des autres catégories.

Une instance de la classe SchemaType représente l'ensemble des types définis dans un schéma O_2 (figure 4.5). Les définitions des types sont groupés dans un attribut qui fait référence à une collection d'objets de la classe Type.

Un objet de la classe Type représente un type du schéma. Dans O_2, un type peut être *atomique*, peut décrire un *n-uplet* ou une *collection*. Les classes AtoType, Tuple et Collection de la figure 4.5 modélisent les différents types d'un schéma. Ces classes sont toutes sous-classes de la classes Type et par conséquent elles héritent des propriétés de Type.

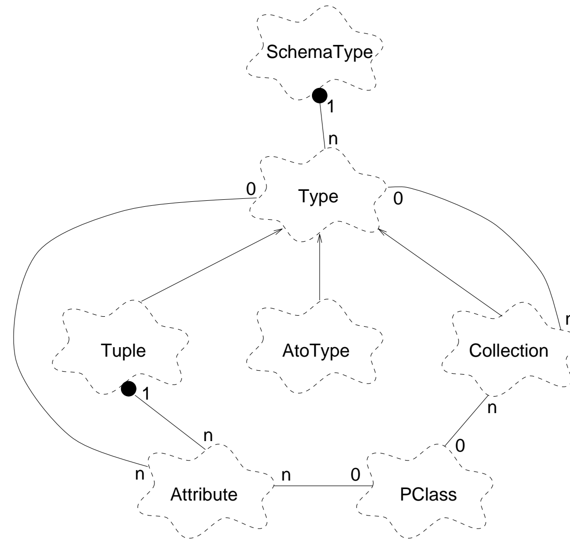


FIG. 4.5 - *Les types du schéma*

4.1.3.3 Gestion des classes et des méthodes

La catégorie Class Manager est sans doute la catégorie dont les classes sont les plus solicitées. Cela se traduit par l'ensemble de services fournis, mais aussi par l'ensemble de demande de services depuis cette catégorie (figure 4.1). La catégorie Class Manager gère les classes et leurs méthodes au travers des classes SchemaClass, PClass, Method et Param.

Contrairement à la catégorie Type Manager, les instances gérées par les classes de Class Manager sont, d'une part, obtenues du schéma et, d'autre part, produites par notre système. Ces dernières sont présentes dans le catalogue mais n'existent pas dans un schéma O_2. C'est le cas, notamment, des matrices de compatibilité associées aux classes et des vecteurs d'accès associés aux méthodes (voir la section 4.1.4).

Une instance de la classe SchemaClass possède un attribut de la classe OrderedCltn qui fait référence à l'ensemble des classes d'un schéma O_2. La représentation des classes sous la forme d'un ensemble, et non sous la forme d'une hiérarchie, rendent plus simple le traitement et la récupération des classes dans notre système. Nous modélisons l'héritage par des ensembles de super-classes et de sous-classes associés à chaque classe (cf. figure 4.6).

Les classes

Une classe du schéma est modélisée par une instance de la classe PClass. Cette classe possède plusieurs relations avec d'autres classes de notre système :

- La relation avec la classe Type modélise le fait qu'une classe possède un type.

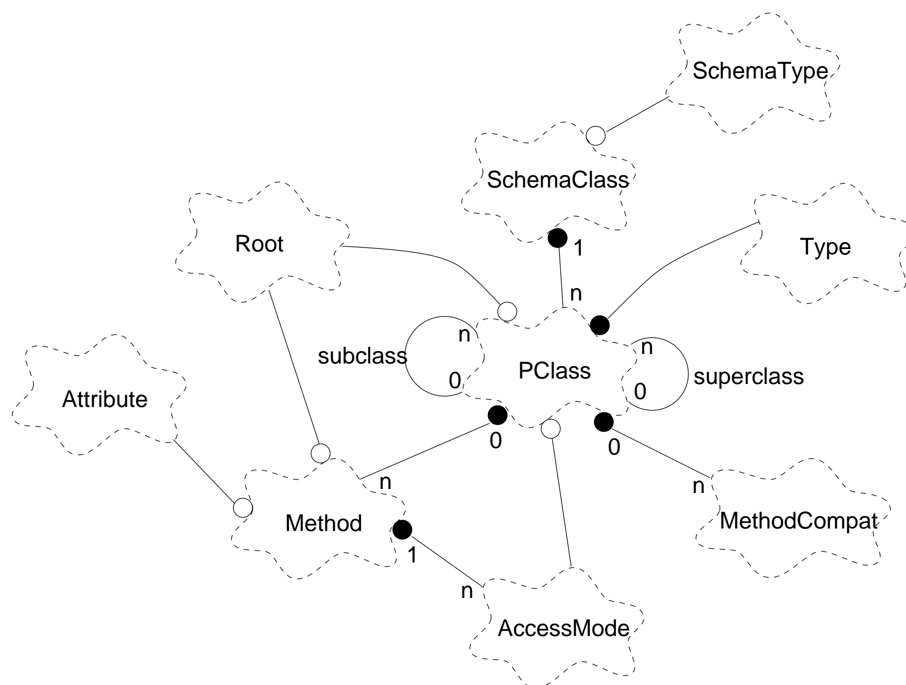


FIG. 4.6 - *Les classes du schéma*

- La relation avec la classe Method sert à modéliser le fait qu'une classe possède plusieurs méthodes. Une instance de PClass a une référence vers une collection qui contient un ensemble de définitions de méthodes.
- La relation dénotées superclass et subclass sont utilisées pour modéliser la hiérarchie des classes. Ces relations sont mises en œuvre par des références vers des collections d'objets.
- Une instance de PClass possède également une matrice qui donne la compatibilité des méthodes de la classe. Nous la modélisons par une relation entre PClass et la classe MethodCompat. Nous détaillions les aspects liés à la classe MethodCompat ainsi que la relation entre PClass et AccessMode dans la section 4.1.4.

Les méthodes des classes possèdent une signature qui décrit les paramètres et un ensemble de modes d'accès. Les modes d'accès sont représentés par une relation entre Method et la classe AccessMode. La classe Method utilise les services des classes Root et Attribut pour la définition des modes d'accès. Le mode d'accès d'une méthode sur un attribut ou sur une entité nommée est donné par un élément de l'ensemble de modes d'accès. Pour chaque attribut et pour chaque entité nommée, le mode d'accès est défini, même si la méthode ne manipule pas l'attribut ou l'entité.

4.1.4 Compatibilité des méthodes

La gestion des matrices de compatibilité est effectuée par les instances des classes de deux catégories différentes. La quasi-totalité des services pour construire une matrice de compatibilité est effectuée par des méthodes des classes PClass et Method, qui elles mêmes utilisent les services des classes MethodCompat et AccessMode. Ces deux dernières forment la catégorie CompatTable Generator. La construction de la matrice de compatibilité est donc accomplie par des classes des catégories Class Manager et CompatTable Generator.

La classe MethodCompat possède deux relations : une avec la classe PClass et une avec la classe Method (cf. figure 4.7). La relation avec PClass modélise le fait qu'une classe possède une matrice de compatibilité. En effet, une instance de MethodCompat représente la compatibilité entre deux méthodes d'une classe. Ainsi, une matrice de compatibilité est mise en œuvre par une référence dans l'instance de PClass vers une collection d'instances de MethodCompat.

La relation entre MethodCompat et la classe Method modélise le fait qu'une entrée de la matrice de compatibilité concerne deux méthodes. D'ailleurs cela explique la cardinalité de cette relation. Chaque instance de MethodCompat possède la valeur de deux *identificateurs de méthodes* et un attribut du type logique, qui stocke le résultat du calcul de la compatibilité entre ces méthodes.

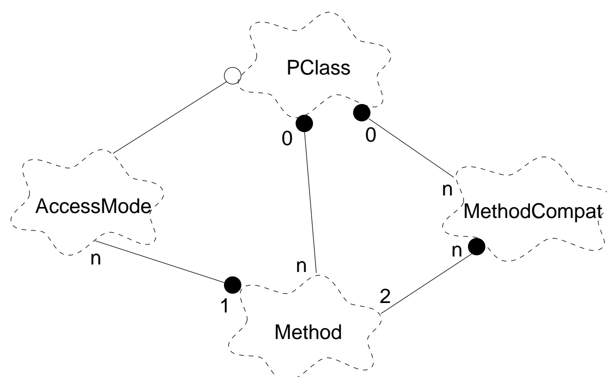


FIG. 4.7 - *Les classes MethodCompat et AccessMode*

La construction de la matrice de compatibilité est effectuée en deux étapes :

1. Tout d'abord on calcule le *vecteur d'accès* pour chaque méthode des classes.

2. Finalement on détermine la compatibilité des méthodes selon les modes d'accès fournis par les vecteurs d'accès des méthodes.

4.1.4.1 Vecteurs d'accès

Les modes d'accès des méthodes sur les attributs de leur classe et sur les entités nommées sont représentés sous la forme d'un vecteur d'accès. Chaque entrée d'un vecteur d'accès est un pair <attribut ou entité, mode d'accès> qui donne le mode d'accès de la méthode sur l'attribut ou sur l'entité nommée (voir figure 4.8 où R = "Read", W = "Write" et N = "No access"). Chaque méthode d'une classe possède son vecteur d'accès. De plus, pour une classe donnée, les vecteurs des méthodes de la classe ont tous la même taille. Autrement dit, ils ont tous les mêmes attributs et les mêmes entités nommées.

Meth1

At1	At2	.....	Atn	RT1	RT2	.....	RTn
N	W	.....	N	R	N	.....	N

FIG. 4.8 - *Exemple de vecteur d'accès*

Il y a certainement plusieurs façons de représenter les modes d'accès. Déjà on peut séparer les accès aux attributs des accès aux entités nommées puisque les entités nommées sont globales à toutes les méthodes de toutes les classes et les attributs sont globaux uniquement aux méthodes de leur classe.

Nous avons également considéré la possibilité de représenter les modes d'accès à travers une matrice attachée aux classes. Dans une telle matrice, les colonnes seraient les attributs et les entités nommées, tandis que les lignes seraient les méthodes de la classe. Cette façon de représenter les modes d'accès serait plus performante, au moins en ce qui concerne l'espace mémoire, puisque l'on ne duplique pas la représentation des attributs. Cependant, nous l'avons trouvé, du point de vue conceptuelle, moins intéressante que la représentation par vecteur d'accès, car les modes d'accès sont des propriétés d'une méthode et non pas d'une classe.

Les modes d'accès des méthodes sur les objets d'une classe sont déterminés selon le type de la classe. Lorsque le type de la classe est atomique ou s'il s'agit d'une collection, le mode d'accès est défini pour l'objet entier. Par conséquent, dans la version courante du prototype, nous considérons, du point de vue des accès des méthodes, les listes et les ensembles comme un seul objet. Autrement dit, l'accès à un élément d'une liste est traité dans notre système comme un accès à la liste. Ce choix permet la simplification du calcul du mode d'accès sur les collections puisque nous utilisons le même traitement pour les trois types différents de collections du système O_2.

Le contrôle du mode d'accès sour les n-uplets est efféctué au niveau des attributs. Toutefois, nous considérons uniquement le premier niveau de définition. Cela veut dire qu'un attribut ayant un type n-uplet aura un seul mode d'accès qui concerne toute son structure. Ainsi, nous avons fait le choix d'utiliser uniquement le premier niveau de définition des n-uplets dans le calcul des vecteurs d'accès. On aurait pu choisir une approche de contrôle d'accès plus fin, en considérant les attributs à n'importe quel niveau de la définition des n-uplets. Cependant, cette approche est plus complexe à mettre en œuvre et apporte peu de résultat dans la validation de notre modèle.

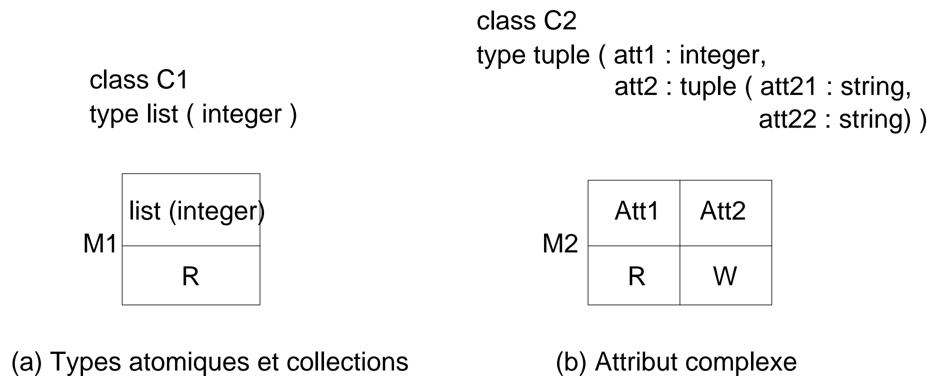


FIG. 4.9 - *Exemple de vecteurs d'accès*

La figure 4.9 montre deux exemples de vecteurs d'accès. Le vecteur d'accès (a) concerne une méthode de la classe C1, dont le type est une collection. Notez un seule mode d'accès sur la collection entière. La partie (b) de la figure donne la définition de la classe C2 dont le type décrit un n-uplet ayant deux attributs att1 et att2. L'attribut att2 décrit également un n-uplet. Malgré cela, les vecteurs d'accès des méthodes de la classe ne prennent en compte que les attributs définis dans le premier niveau de définition, c'est à dire, les attributs att1 et att2.

La figure 4.10 montre les échanges entre les objets de plusieurs classes afin de construire les vecteurs d'accès des méthodes. La méthode methAccMode parcourt l'ensemble des méthodes de la classe en envoyant à chaque instance de Method un message pour calculer ses modes d'accès. La méthode setDav récupère le type de la classe et calcule le mode d'accès selon ce type et le mode d'accès sur les entités nommées. La méthode directAccVectorExp ajoute à l'ensemble des méthodes d'une classe les méthodes héritées des super-classes et dirige le calcul de mode d'accès pour ces méthodes. La méthode setDavExp détermine les modes d'accès des méthodes héritées. Dans le cas de re-définition de la méthode héritée, setDavExp démarre la jointure des modes d'accès pour les super-classes.

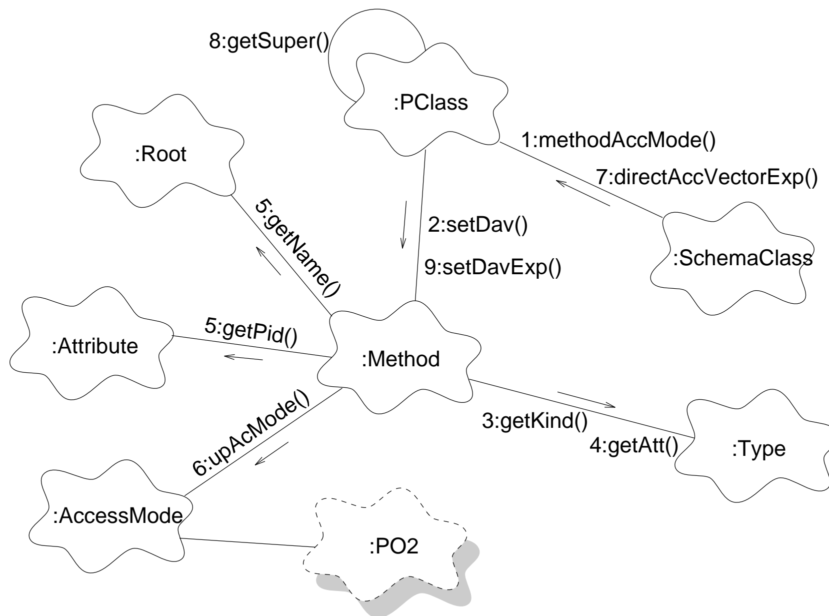


FIG. 4.10 - *Construction des vecteurs d'accès*

Pour calculer les modes d'accès, la structure d'une la classe est d'abord demandée au type de la classe à travers les messages 3:getKind() et 4:getAtt() (cf. figure 4.10). Ensuite le message 6:upAcMode() met à jour le mode d'accès de la méthode sur un objet, un attribut ou une entité nommée. Le message 9:setDavExp() est placé en dernier dans notre diagramme d'objet mais il peut redémarrer la procédure de mis à jour des modes d'accès. Par conséquent, les messages envoyés depuis :Method vers :Type, :Root, :Attribute et :AccessMode peuvent être envoyés plusieurs fois pendant l'exécution de setDavExp.

4.1.4.2 Matrice de compatibilité

Chaque classe du catalogue possède une matrice de compatibilité de ses méthodes. Les lignes et les colonnes de la matrice représentent les méthodes et chacune des entrées est un boolean qui donne la compatibilité des méthodes. Lorsqu'on a besoin de connaître la compatibilité de deux méthodes d'une classe, il suffit de récupérer l'information qui correspond aux méthodes dans la matrice.

La matrice de compatibilité est le résultat de la vérification de la compatibilité entre les vecteurs d'accès des méthodes. Deux vecteurs d'accès sont compatibles s'il n'y a pas de conflit d'accès sur les mêmes attributs ou entités nommées des deux vecteurs (cf. figure 4.11). La construction de la matrice de compatibilité d'une classe est simple : pour chaque paire de méthodes de la classe, on vérifie la compatibilité de leurs

vecteurs d'accès. Le résultat de cette vérification est stocké par l'objet représentant la compatibilité des deux méthodes dans la matrice.

	At1	At2	....	Atn	RT1	....
M1	N	W	....	R	R	....
M2	R	N	....	N	R	....
Compatible ?	OUI	OUI	....	OUI	OUI	....

FIG. 4.11 - *Vérification de compatibilité de vecteurs d'accès*

La figure 4.12 montre les itérations entre les objets pour construire la matrice. L'objet :SchemaClass envoie le message 1:setCompatTable() vers chaque instance de PClass pour calculer la matrice de compatibilité de ses méthodes. Les vecteurs d'accès des méthodes de :PClass sont obtenus par les messages 2:getDav() et 3:getAccMode(). La méthode compatCheck détermine enfin si les vecteurs d'accès des deux méthodes sont compatibles.

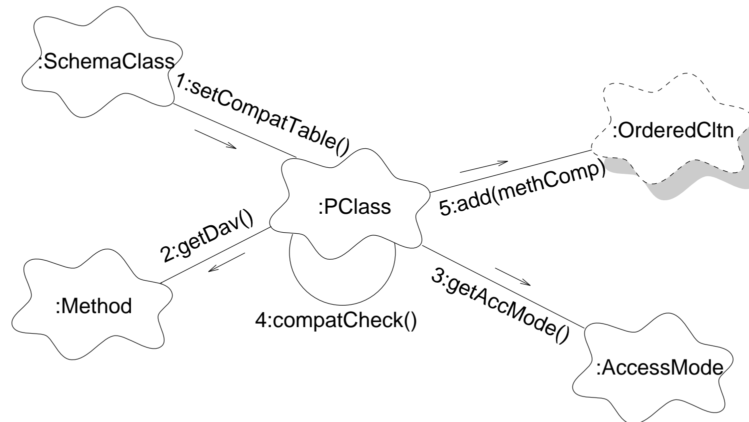


FIG. 4.12 - *Construction de la matrice de compatibilité*

En fait, la matrice n'est pas vraiment nécessaire car la compatibilité des vecteurs d'accès peut être déterminée chaque fois que le calcul de la compatibilité de deux méthodes est nécessaire. Cependant, la consultation de la matrice est beaucoup plus

performante que le calcul de la compatibilité de deux vecteurs d'accès. Ainsi, nous avons fait le choix de matérialiser les matrices de compatibilité et de les associer à chaque classe du catalogue.

Dans la version actuelle du prototype nous ne construisons pas la matrice de compatibilité pour toutes les méthodes du schéma. Cela est du fait que la construction du graphe de flot de contrôle global dépend de l'analyse du code des méthodes, tâche qui n'est pas effectuée dans la version actuelle du système.

4.1.5 Transformation des transactions

La catégorie Transaction Parallelyser est composée de trois classes : Application, Transaction et Vertex. L'objectif de cette catégorie de classes consiste à transformer le code séquentiel d'une transaction en un code avec parallélisme. En fait, nous ne modifions pas le code de la transaction. Nous créons plutôt une représentation de ce code sous la forme d'un graphe sur lequel nous effectuons nos transformations.

La parallélisation est réalisée en deux étapes. La première consiste à construire le graphe qui représente le code source de la transaction et la seconde construit un nouveau graphe pour cette transaction par des transformations sur le graphe source en tenant compte de la compatibilité des méthodes des sommets du graphe.

Le graphe source est une séquence de sommets ayant une seule arête d'entrée et une seule arête de sortie. Les sommets au début et à la fin de la séquence sont des sommets spéciaux qui représentent le début et la fin de la transaction, respectivement. Entre eux, les sommets représentent les méthodes de la transaction étant appelées dans l'ordre donné par le graphe.

La figure 4.13 montre un exemple d'un graphe source d'une transaction. Les sommets D et F représentent respectivement le début et la fin de la transaction. Les autres sommets représentent les méthodes appelées par la transaction. L'ordre des appels des méthodes est représenté dans le graphe par les arêtes. Le système construit un graphe pour chaque transaction de l'application et associe, à chaque sommet, la méthode correspondante du catalogue.

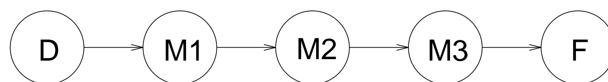


FIG. 4.13 - *Un exemple de graphe de transaction source*

Les transformations réalisées pour construire un nouveau graphe consistent à changer les relations prédécesseur/successeur du graphe source. Dans le graphe transformé,

un sommet peut avoir plusieurs arêtes en entrée et plusieurs en sortie. Les sommets ayant le même prédécesseur modélisent le parallélisme entre les méthodes qu'ils représentent. La condition pour que deux sommets aient le même prédécesseur est récupérée de la matrice de compatibilité et doit déterminer que les méthodes des sommets soient compatibles.

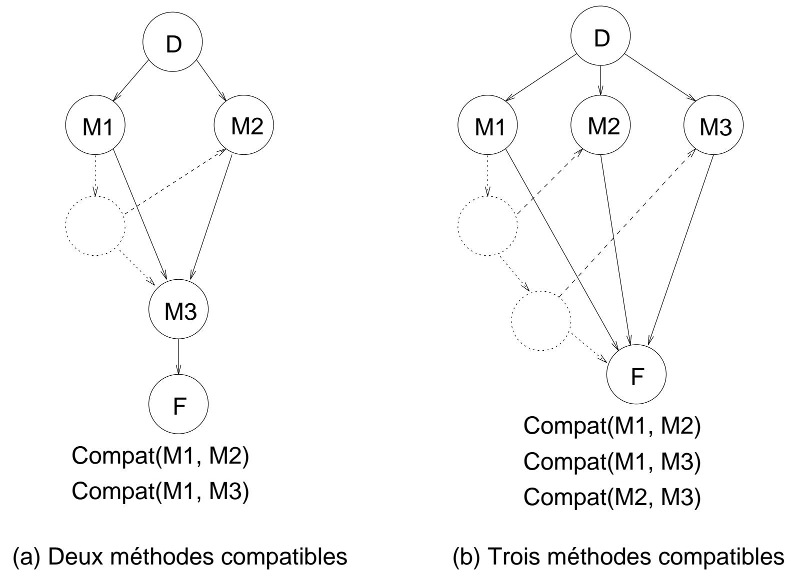


FIG. 4.14 - *Exemples de transformation de graphe source*

La figure 4.14 montre deux exemples de transformation du graphe source de la figure 4.13. Les informations de compatibilité des méthodes corresponds à chaque exemple de transformation. Dans la partie (a), la méthode M1 est compatible avec la méthode M2. Alors la méthode M2 est déplacée pour représenter la parallélisation de M1 avec M2. La méthode M3 est compatible avec M1, mais ne l'est pas avec la méthode M2. Ainsi, le sommet M3 ne peut pas avoir le même prédécesseur que le sommet M2. Il devient donc successeur du sommet de M2 et, par conséquent, du sommet de M1. Dans la partie (b) de la figure, les trois méthodes sont compatibles. Leurs sommets possèdent le même prédécesseur et, comme la transaction est finie, le même successeur.

4.1.5.1 Applications et transactions

Dans notre système une application est composée des transactions. Une instance de la classe Application possède un seul attribut qui fait référence à un objet de la classe OrderedCltn. Celui-ci contiendra l'ensemble des transactions de l'application (cf. figure 4.15).

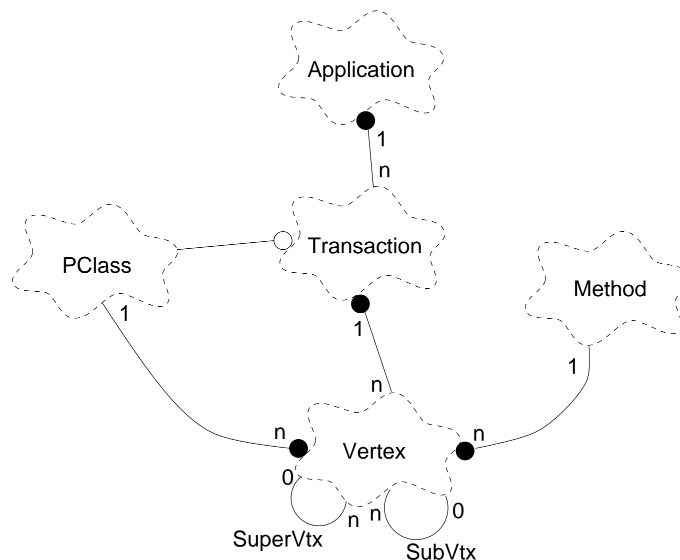


FIG. 4.15 - *Une application et ses transactions*

Une transaction est un graphe de méthodes. Une arête du graphe est une instance de la classe Vertex. Par exemple, pour le graphe de la figure 4.14, le graphe est représenté par une instance de la classe Transaction tandis que chaque sommet est une instance de la classe Vertex. Le graphe d'une transaction est mis en œuvre de la même façon que le graphe d'héritage entre les classes. Ainsi, chaque arête garde l'ensemble des arêtes prédécesseurs et l'ensemble des arêtes successeurs. Cela est modélisé par les relations superVtx et subVtx de la classe Vertex (figure 4.15).

Outre que les ensembles des arêtes prédécesseurs et successeurs, une arête possède également un identificateur d'arête, un identificateur de méthode et sa classe. Noter la cardinalité des relations entre Vertex et les classes Method et PClass dans la figure 4.15.

Le diagramme d'objets des classes Application et Transaction est montré dans la figure 4.16. Le message setGraph démarre la construction du graphe source d'une transaction. Les messages 1 à 4 sont concernés par la construction du graphe source. Les autres messages créent un nouveau graphe, appelé GPEP dans la section 3.3.2.2, à travers des transformations dans le graphe source. Noter les séquences de messages 7 à 9 où l'on récupère une arête successeur du graphe source et vérifie la compatibilité de la méthode de l'arête avec la méthode des autres arêtes successeurs.

4.1.6 Modules du système

La description des classes du système et de leur structuration en catégories ne montre pas la façon dont le système est effectivement réalisé. Cette description permet

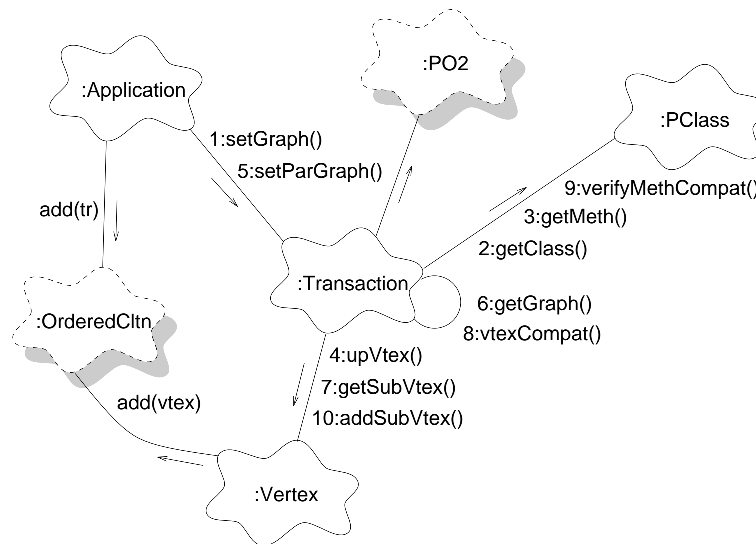


FIG. 4.16 - *Diagramme d'objets des classes* Application *et* Transaction

toutefois de définir une architecture du système specifiée par un modèle physique qui décrit l'organisation des logiciels composants du système [Boo94]. Nous utilisons les *diagrammes de modules* pour décrire cette organisation. Un module est formé de classes décrites dans le modèle logique.

Notre système est structuré en modules selon deux critères principaux. Premièrement, nous prenons en compte l'aspect fonctionnel fourni par les catégories de classes. La structuration modulaire est plus ou moins dérivée de l'organisation des classes en catégories. Il est naturel que les classes fortement liées par des relations dans le but d'accomplir une tâche soient mises en œuvre par un seule module. Deuxièmement, le couplage entre les classes en termes de demandes de services et de flots de données montre un certain dégré de relation physique entre les classes qui détermine également la structuration de ces classes dans les modules du système.

Nous proposons la structuration modulaire de la figure 4.17. Le système est donc composé de trois modules : Toolbox, CatalogueMgr et TrParallelyser. Les relations entre les modules donne leurs dépendances de compilation. Autrement dit, les classes d'un module ayant une flèche vers un autre module utilisent les services des classes de celui-ci.

En générale, la définition de l'architecture d'un système implique la description, d'une part, des principales *fonctions* effectuées par chaque module et, d'autre part, des *interfaces* entre les modules.

Dans une conception à objet, l'interface entre les modules du système est composée par l'ensemble des méthodes des classes de chaque module. L'ensemble de méthodes

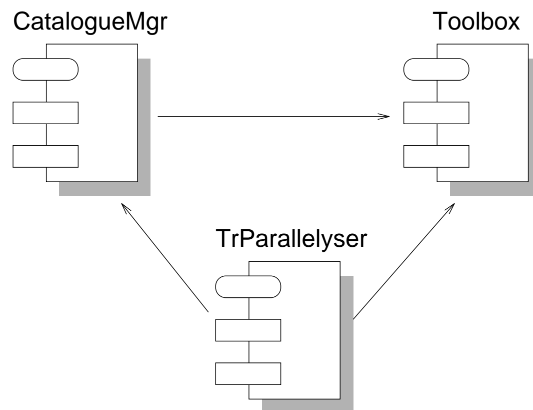


Fig. 4.17 - *Diagramme de modules du système*

des classes ont fait partie du sujet de la section 4.1.1. Par conséquent, nous n'allons pas à nouveau les détailler ici. Lorsque cela est nécessaire, nous rappellerons certains aspects de la définition des méthodes importants à la compréhension de la mise en œuvre du système dans sa totalité.

4.1.6.1 Le module Toolbox

Le module Toolbox implante les classes de la catégorie Utility, sauf la classe O2. Ce que nous avons modélisé comme étant la classe O2 n'est pas vraiment une classe. Il s'agit plutôt d'un fichier de spécification en langage “C” des structures de données et de l'interface fonctionnelle API vers le SGBD O_2.

La principal fonction du module Toolbox est fournir des services divers aux autres modules. Ces services se traduisent par des structures de données (définitions de types) et par des procédures ou des méthodes qui servent d'aide à la réalisation des fonctions principales du système.

4.1.6.2 Le module CatalogueMgr

Le module CatalogueMgr groupe les quatres catégories : Root Manager, Type Manager, Class Manager et CompatTable Generator. Par conséquent, la gestion du catalogue et la construction de la matrice de compatibilité sont les principales fonctions du module. Nous avons choisi de mettre en œuvre ces quatres catégories de classes dans un seule module pour les raisons suivantes :

- Les classes qui implantent la gestion du catalogue sont fortement liées par diverses relations : association, utilisation, et héritage (voir les diagrammes de

classes et d'objets de la section 4.1.1).

- Le procédé de création et de gestion de la matrice de compatibilité est mis en œuvre par les classes MethodCompat et AccessMode de la catégorie CompatTable Generator, mais également par les classes PClass et Module de la catégorie Class Manager. Ainsi, il y une interaction intense entre ces classes, surtout en ce qui concerne le flot d'objets des classes de CompatTable Generator vers les classes de Class Manager et le flot de messages dans le sens inverse.

Le module CatalogueMgr fournit au module TrParallelyser l'ensemble de classes du catalogue avec, bien évidement, leurs méthodes et leur table de compatibilité.

4.1.6.3 Le module TrParallelyser

Le module TrParallelyser correspond exactement à la catégorie Transaction Parallelyser. Sa fonction principale a été détaillée dans la section 4.1.5 et consiste à construire un graphe qui représente le code source d'une transaction et puis le transformer afin de représenter le code cible pour la transaction avec parallélisme.

Ce module ne fournit aucune données aux autres modules du système. Il est surtout un client du module CatalogueMgr duquel il obtient toutes les informations du catalogue nécessaires aux transformations qu'il effectue sur le graphe. L'interface entre les deux modules est essentiellement composée des méthodes de la classe PClass et de la classe Method.

4.2 Parallélisation des règles

Dans cette section, nous expliquons comment l'approche de parallélisation basée sur la transformation de code peut être utilisée pour paralléliser l'exécution des règles du système NAOS (*Native Active Object System*) [CCS94, CHCA94, CC96, CM95] dévéloppé dans le cadre du Projet GOODSTEP [GOO94]. Le Système NAOS est intégré dans le SGBD O_2, et gère des règles actives du type *Evénement*(E)-*Condition*(C)-*Action*(A). La sémantique générale d'une règle est la suivante : lorsque survient un événement E, si la condition C est vérifiée, alors exécuter l'action A.

La section 4.2.1 décrit les règles du système NAOS. Nous faisons une bref description des caractérisques principales du système en regardant plus en détail son modèle d'exécution. La section 4.2.2 montre la parallélisation des règles. Nous expliquons donc la relation de compatibilité des règles et la façon dont le système calcule l'ordonnancement des règles dans des cycles d'exécution. Cette section se termine avec quelques exemples qui montrent la parallélisation de deux types de règles fournies par NAOS.

4.2.1 Règles du système NAOS

Les règles NAOS appartiennent à un schéma et sont définies au même niveau que les classes et les applications O_2. Deux types de règles sont considérées : les règles *immédiates* et les règles *différées*. Une règle immédiate est exécutée immédiatement après la détection de l'événement déclenchant. L'exécution d'une règle differée est repousée à la fin de la transaction dans laquelle la règle est déclenchée.

Lors de l'exécution d'une application, les règles du schéma de l'application deviennent "actives" : les types d'événements spécifiés par ces règles sont abonnés au noyau du système O_2. Ces événements sont produits pendant l'exécution de l'application, et leurs règles respectives sont déclenchées et sont exécutées dans des *cycles d'exécution*.

4.2.1.1 Définition des règles

La figure 4.18 donne la définition d'une règle. Cette règle appartient à un schéma dans lequel la classe Person est définie. Chaque instance de Person possède un nom et un âge. La règle "age_control" vérifie la cohérence de la valeur de l'attribut âge, i.e., le nouvel âge doit être plus grand que l'ancien.

Au moment de l'exécution, une règle est associée à une *delta-structure* que contiendra les données associées à un ou à plusieurs événements qui l'ont déclenchée. Les delta-structures sont désignées par un identificateur donné dans la définition de la règle par la clause "with". Le type d'une delta-structure dépend du type de l'événement et du mode de couplage de la règle (immédiate ou différée). Les données de la delta-structure sont accessibles par la règle à travers une interface fonctionnelle.

Dans la règle "age_control", le delta-élément associé au type d'événement est dénoté par "p". Son type est défini comme tuple(**entity**: Person, **component**: integer). L'identificateur "p" dénote donc l'instance de Person dont on souhaite modifier l'âge *avant* l'exécution la modification, tandis que "new(p)" dénote la même instance, mais dans un état qui serait produit *après* la modification de l'âge.

```
create rule age_control
coupling immediate
on before update Person->age with p
if p->age < new(p)->age
do { display( "Decreasing age is not allowed:
                the actual transaction is aborted");
     abort; }
```

FIG. 4.18 - *La définition d'une règle*

L'Evénement d'une règle spécifie quels sont les types d'événements qui peuvent déclencher la règle. NAOS détecte des événements primitifs et des événements composites. Les événements composites sont des séquences, des unions et des négations d'événements. Nous ne les détaillons pas ici. Les événements primitifs sont divisés en deux catégories :

- Les événements de manipulation d'entités (objets ou valeurs O_2) sont générés par la manipulation de toute ou d'une partie d'une entité. Les opérations qui peuvent générer ces types d'événements sont la création, l'accès et la mise à jour d'une entité, le fait qu'une entité devienne persistante ou temporaire, l'insertion ou la suppression d'une entité dans/d'une collection, et l'appel de méthodes.
- Les événements applicatifs sont associés au début ou à la fin d'une transaction, d'un programme ou d'une application O_2.

Dans l'exemple de la figure 4.18, la clause on définit le type d'événement before update(Person, age). Un événement de ce type aura lieu immédiatement *avant* la modification de l'attribut "age" d'une instance de "Person". Une telle modification peut être faite directement si l'attribut est public ou par l'intermédiaire d'un appel de méthode de la classe "Person".

La Condition d'une règle spécifie des prédicats sur des entités (persistantss ou temporaires). Ces prédicats sont exprimés sous la forme d'une requête $\mathrm{O_2}$SQL/OQL. Une telle requête peut porter sur toute la base ou sur des données associées à l'événement déclencheur. Les requêtes peuvent également appeler des méthodes.

La clause if de la règle "age_control" est évaluée à "vrai" si la valeur courante de l'attribut "age" de l'objet dénotée par "p" est plus petite que la nouvelle valeur pour cet attribut. Dans ce cas, l'action est exécutée, sinon elle ne l'est pas et l'exécution de la règle se termine.

L'Action est formée d'un bloc $\mathrm{O_2}$C pouvant manipuler des entités persistantes ou temporaires. La création et la validation d'une transaction sont interdites dans le code d'une action, cependant l'annulation d'une transaction ne l'est pas. L'utilisation du mot-clé "instead" dans la clause on indique que l'action doit être exécutée à la place de l'opération qui a déclenchée la règle. Cela est possible uniquement dans des règles immédiates dont le type d'événement présente la clause "before". L'action de la règle "age_control" affiche un message sur l'écran et annule la transaction courante.

4.2.1.2 Traitement de règles

Quand un événement est produit, plusieurs règles sont déclenchées. Il s'agit de déterminer dans quel ordre ces règles vont être exécutées. Dans NAOS, cet ordre est

déterminé par des *priorités* entre règles. Le système attribue des priorités par défaut, basées sur l'ordre de définition des règles. Le programmeur peut ensuite surcharger ces priorités en spécifiant des relations d'ordre entre des couples de règles : si une règle r_1 précède une règle r_2 et si les deux règles sont déclenchées par le même événement, alors r_1 est exécutée avant r_2.

Le modèle de transactions du système O_2 sert de base pour déterminer quand et comment les règles sont exécutées. Ce modèle offre uniquement des transactions classiques, on n'a pas la possibilité de créer des transactions découplées, ni de créer des transactions dans une transaction déjà existante, comme dans un modèle de transactions imbriquées [BDK92]. Par conséquent, les conditions et actions sont toujours exécutées dans la transaction déclenchante. Pour les règles immédiates, la condition est évaluée tout de suite après l'occurence de l'événement et, si celle-ci est satisfaite, l'action est exécutée. Pour les règles différées, l'évaluation de la condition et l'exécution de l'action auront lieu après la dernière opération de la transaction déclenchante, mais juste avant sa terminaison.

L'exécution de la partie action d'une règle peut générer de nouveaux événements qui peuvent à leur tour déclencher d'autres règles et ainsi de suite. Une fois encore, il s'agit de déterminer dans quel ordre ces règles vont être exécutées. Pour établir cet ordre, NAOS a introduit la notion de *cycle d'exécution*. Un cycle d'exécution est une suite d'opérations exécutées dans une transaction (la transaction courante est dite cycle 0), ou pendant l'exécution de la partie action d'une règle. Quel que soit le mode de couplage considéré, une règle est toujours exécutée dans un cycle différent de celui dans lequel le ou les événements qui l'ont déclenchée ont été générés. Si plusieurs règles doivent être exécutées dans le même cycle, elles le sont suivant leur priorité comme nous l'avons vu précedemment. Les règles immédiates sont exécutées en *profondeur d'abord*, tandis que les règles différées en *largeur d'abord*.

La figure 4.19 donne un exemple des cycles d'exécution pour des règles immédiates. Toutes les règles déclenchées dans un cycle (r_1 et r_2) sont exécutées immédiatement dans un nouveau cycle d'exécution imbriqué (cycle 1(a) et cycle 1(b), respectivement). Les événements produits dans un cycle d'exécution son traités sans tenir compte des autres règles déjà déclenchées. C'est pour cela que les règles r_{1a} et r_{1b} sont exécutées avant la règle r_2. L'ordre d'exécution final pour les règles immédiates de la figure 4.19 est : r_1, r_{1a}, r_{1b}, r_2 et r_{2a}, en considérant que r_1 précède r_2 et que r_{1a} précède r_{2a}.

La figure 4.20 montre un exemple de cycles d'exécution pour des règles différées. Les règles déclenchées dans un cycle n sont exécutées dans un nouveau cycle n+1 après l'exécution de toutes les règles du cycle n. Dans la figure 4.20, les règles r_1 et r_2 sont exécutées dans le cycle 1. Les règles r_{1a} et r_{1b}, déclenchées par r_1, sont exécutées dans le cycle 2, après l'exécution de toutes les règles du cycle 1. L'ordre d'exécution

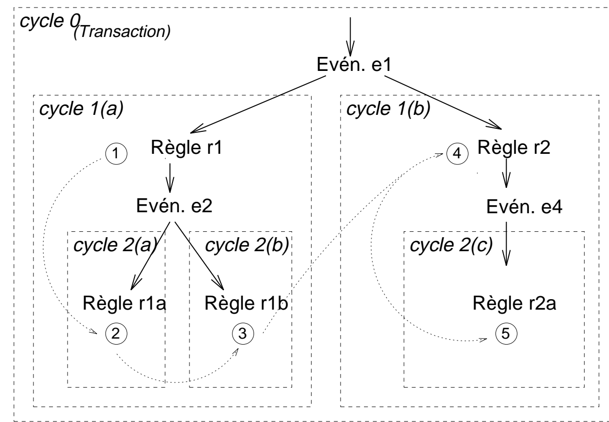


FIG. 4.19 - *Cycles d'exécution pour les règles immédiates*

final est : r_1, r_2, r_{1a}, r_{2a} et r_{1b}, en considérant que r_1 précède r_2 et que r_{1a} précède r_{2a} qui à son tour précède r_{1b}. La transaction se termine après le dernier cycle, i.e., le cycle qui ne déclenche plus de règles différée. Dans notre exemple, après le cycle 2.

4.2.2 Parallélisation des règles

Comme nous l'avons montré dans la section précédente, les règles d'un cycle d'exécution sont toutes exécutées de manière synchrone. Dans cette section, nous considérons l'exécution des règles à la fois de manière synchrone et de manière *asynchrone*. Notre approche consiste à exploiter le *parallélisme implicite* entre les règles d'un cycle. La construction du *plan d'exécution* pour l'ordonnancement de l'exécution des règles d'un cycle est basée sur la *compatibilité* de ces règles et sur leur relation de priorité.

La définition de la relation de compatibilité des règles utilise la relation compatibilité des méthodes telle que nous l'avons défini dans le Chapitre 3 car les règles peuvent appeler des méthodes. L'exécution d'une règle peut également déclencher des nouvelles règles, dont la définition de la compatibilité entre les règles doit tenir compte. Cependant, nous ne considérons que les déclenchements de règles immédiates, puisque les règles différées déclenchées dans un cycle n sont toujours exécutées dans le cycle n+1, et par conséquent ne peuvent pas être exécuter en parallèle avec les règles du cycle n. En conclusion, la compatibilité des règles doit considérer les appels de méthodes et les déclenchement de règles immédiates.

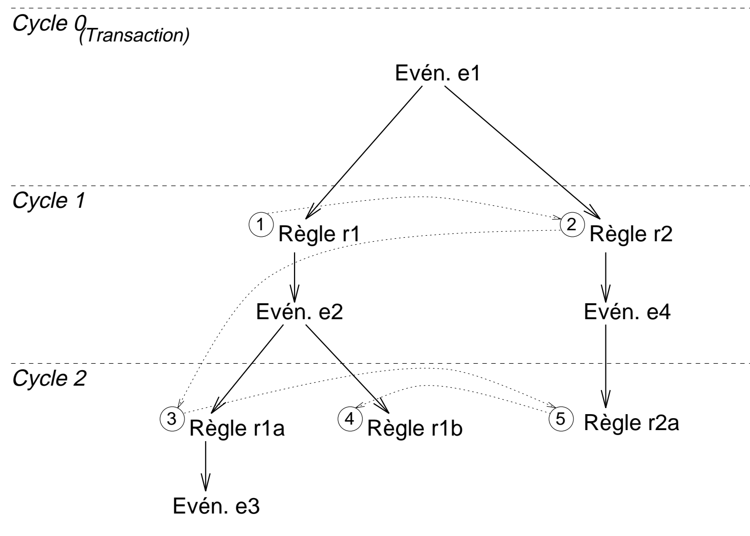


FIG. 4.20 - *Cycles d'exécution pour des règles différées*

4.2.2.1 Compatibilité de règles

Nous utilisons les concepts introduits dans la section 3.2.2 pour définir la compatibilité des règles (pour plus de détails voir [CM95]). Notre approche pour définir cette compatibilité est basée sur trois aspects :

- Les *ensembles d'accès* des règles comparables aux ensembles d'accès des méthodes (voir section 3.2.2.3).
- La relation de *compatibilité entre les méthodes* du schéma (voir section 3.2.2.7).
- La relation de *compatibilité entre les règles immédiates.*

Les ensembles d'accès des règles

Les ensembles d'accès des règles sont construits en tenant compte des codes des règles Nous considérons que la condition et l'action forment le code d'une règle et ainsi ces parties de la règle sont traitées comme un seule code source. La delta-structure associée à la règle est considérée comme un paramètre pour le code de la règle. Dans ce code, on peut manipuler les instances des classes du schéma par des envois de messages ou directement si l'instance est publique. Par conséquent, tous comme pour les méthodes, les règles peuvent accéder et/ou modifier les attributs des objets persistants ou temporaires et des racines de persistance.

Les méthodes appelées et les règles immédiates déclenchées

Une règle possède un ensemble éventuellement vide de méthodes et de règles immédiates. Les méthodes sont directement appelées par la règles, tandis que les règles sont déclenchées par un événement produit lors de l'exécution de la règle. Une règle, ses méthodes appelées et ses règles immédiates déclenchées sont toutes exécutées dans un même cycle d'exécution.

Dans l'exemple de règles immédiates de la figure 4.19, on peut considérer que l'événement e_1 est issue d'une règle r qui peut être une règle immédiate ou différée. L'ensemble de règles immédiates déclenchées par r est composé de r_1 et r_2, tandis que celui de la règle r_1 est composé des règles r_{1a} et r_{1b}. Il faut noter que les règles r_1 et r_2 sont exécutées respectivement dans les cycles 1(a) et 1(b) qui sont emboîtés dans le cycle la règle r. Les cycles d'exécution des règles r_{1a} et r_{1b} sont emboîtés dans le cycle de r_1 et ainsi de suite. Ainsi si l'on souhaite paralléliser les règles r_1 et r_2, par exemple, on doit tenir compte des règles r_1, r_2 et r_{2a} car elles aussi vont être exécutées en parallèle.

Relation de compatibilité entre les règles

Deux règles de même type sont en conflit si un élément d'un ensemble d'accès en écriture d'une des règles appartient à un ensemble d'accès de l'autre règle, et si les méthodes appelées par les règles ne sont pas compatibles. Pour déterminer la relation de conflit entre deux règles (immédiates ou différées), nous utilisons les ensembles d'accès des règles et la relation de compatibilité entre les méthodes appelées par les règles.

Toutefois, deux règles d'un même type sont compatibles si elles ne sont pas en conflit et si les règles immédiates qui elles déclenchent sont compatibles. Pour déterminer la compatibilité de deux règles immédiates ou des deux règles différées, on s'est basée sur la relation de conflit entre les règles et sur la relation de compatibilité entre les règles immédiates déclanchées.

Comme nous l'avons fait pour les méthodes, la relation de compatibilité pour un ensemble de règles peut être représentée par une matrice. Par exemple, la table 4.2 représente la compatibilité entre les règles de la figure 4.19.

	r_1	r_2	r_{1a}	r_{1b}	r_{2a}
r_1	yes	yes	yes	no	yes
r_2	yes	no	yes	yes	no
r_{1a}	yes	yes	no	no	yes
r_{1b}	no	yes	no	no	yes
r_{2a}	yes	no	yes	yes	yes

TAB. 4.2 - *La relation de compatibilité pour un ensemble de règles.*

4.2.2.2 Exécution parallèle des règles

A chaque fois qu'un ensemble de règles, dites règles candidates, doit être exécuter — à la fin d'un cycle pour les règles différées et après l'occurence d'un événement pour les règles immédiates — on construit un plan d'exécution qui donne l'ordonnancement de l'exécution de ces règles. On peut alors avoir des règles d'un cycle différé qui s'exécutent en parallèle et des règles immédiates qui sont traitées en parallèle (dans des cycles parallèles).

L'utilisation des cycles d'exécution comme limite de synchronisation ne permet pas l'exécution des programmes ou des transactions en parallèle avec des exécutions des règles. Ainsi, toutes les règles d'un cycle doivent finir leur exécution pour qu'un nouveau plan soit construit et un nouveau cycle démarre. En revanche, les règles d'un cycle peuvent être exécutées de manière séquentielle ou en parallèle. Par conséquent, dans le cas de règles différées, on a exécution synchrone de cycles, même si dans chaque cycle on permet une exécution asynchrone de règles. Dans le cas de règles immédiates, on permet une exécution synchrone ou asynchrones des règles d'un cycle. Si deux règles immediates sont compatibles, le plan d'exécution détermine leurs exécutions en parallèle, sinon les règles sont exécutées de manière séquentielle en tenant compte de leurs priorités.

Exemples

Considerons les règles différées de la figure 4.20. La relation de compatibilité entre ces règles est donnée par la matrice de la table 4.2. Rapellons la relation de précédence pour les règles de l'exemple : $r_{1a} < r_2 < r_{1b}$. Dans nos exemples $r_1 \parallel r_2$ dénote l'exécution parallèle de deux règles, tandis que $r_1 \rightarrow r_2$ dénote l'exécution séquentielle des règles.

L'événement e_1, qui a lieu dans le cycle 0, déclenche les règles r_1 et r_2. Ces règles sont considérées pour l'exécution dans le cycle 1. Selon la table 4.2, r_1 et r_2 sont compatibles, donc elles peuvent être exécutées en parallèle. L'exécution de la règle r_1 déclenche deux règles, r_{1a} et r_{1b}, qui ne sont exécutées qu'après l'exécution des règles du cycle 1. Autrement dit, avant l'exécution de ces règles, r_2 est exécutée et produit l'événement e_4 qui déclenche la règle r_{2a}.

La table 4.2 montre que r_{1a} est compatible avec r_{2a}, mais elle n'est pas compatible avec r_{1b}; r_{1b} est compatible avec r_{2a}. Selon la relation de précédence, r_{1a} doit passer avant r_{1b}. Par conséquent, r_{1a} et r_{2a} peuvent être exécutées en parallèle, cependant r_{1a} et r_{1b} doivent être exécutées de manière séquentielle. Comme résultat, le plan d'exécution pour les règles du cycle 2 détermine l'exécution en parallèle des règles r_{1a} et r_{2a} suivie de l'exécution de r_{1b}. L'ordre final d'exécution sera : $(r_1 \parallel r_2) \rightarrow ((r_{1a} \parallel r_{2a}) \rightarrow r_{1b})$.

Examinons maintenant les règles immédiates de la figure 4.19 et considérons que la table 4.2 donne la matrice de compatibilité pour ces règles. Comme r_1 et r_2 sont compatibles, elles peuvent être ordonnancées pour une exécution en parallèle. Ce n'est pas le cas pour les règles r_{1a} et r_{1b}, qui doivent être exécutées en parallèle selon la relation de précédence. Alors, l'ordonnancement final sera : $(r_1 \rightarrow r_{1a} \rightarrow r_{1b}) \parallel (r_2 \rightarrow r_{2a})$. Notez qu'il n'y a aucun synchronisme entre le sous-arbre de règles de racines r_1 et cellui de racines r_2. Cela est dû au fait que toutes le règles immédiates déclenchées par r_1 sont compatibles avec toutes celles déclenchées par r_2.

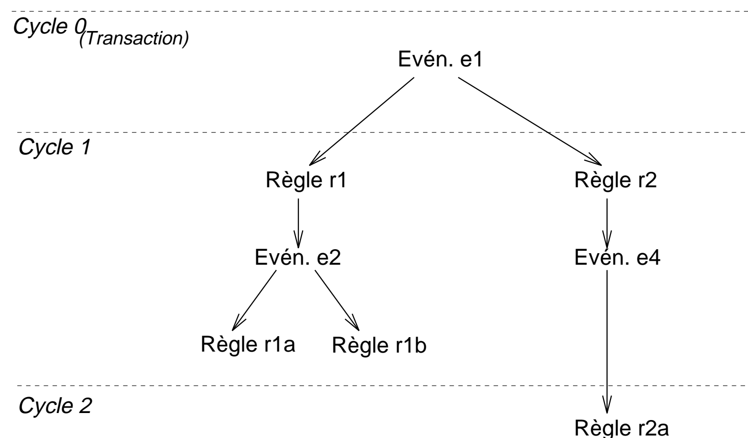


FIG. 4.21 - *Cycles d'exécution pour les règles immédiates et différées*

Considérons le cas où il faut tenir compte de deux types de règles lors de la construction du plan. La figure 4.21 montre une exécution en cascade des règles différées r_1 et r_2. Dans cet exemple, r_1 déclenche deux règles immédiates, r_{1a} et r_{1b}, tandis que r_2 déclenche une règle différée (r_{2a}). Considérons encore une fois la table 4.2 comme représentant la matrice de compatibilité pour ces règles. Les règles candidates du cycle 1 peuvent être exécutées en parallèle car r_1 (et ses règles déclenchées) est compatible avec r_2. La règle r_{2a} sera exécutée dans le cycle 2, après la terminaison de l'exécution de toutes les règles du cycle 1. L'ordre final est donné par : $((r_1 \rightarrow r_{1a} \rightarrow r_{1b}) \parallel r_2) \rightarrow r_{2a}$.

4.2.3 Architecture du système

La figure 4.22 montre l'architecture du système NAOS étendu. Les modules de définition et d'exécution de règles et le détecteur d'événements sont les composants principaux de NAOS (pour plus de détail voir [CC96]). Deux nouveau modules effectuent la *génération des tables de compatibilité* des règles et la *construction des*

plans d'exécution avec parallélisme entre les règles candidates. Ces modules doivent être intégrés respectivement dans le module de définition de règles et dans le module d'exécution de règles.

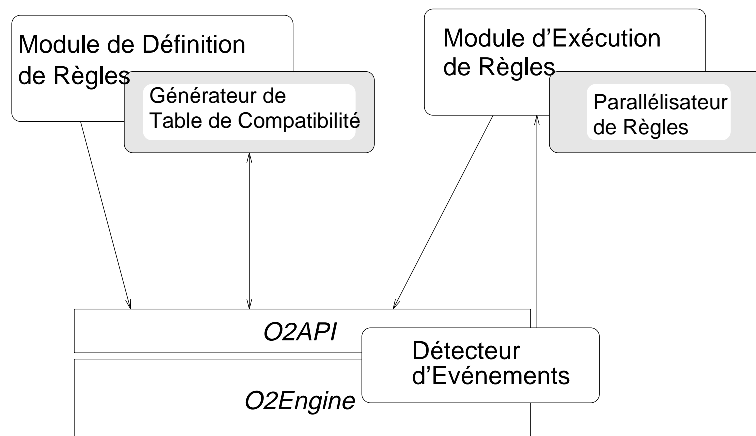


FIG. 4.22 - *Architécture du système*

4.2.3.1 Générateur de tables de compatibilité

Le module de définition de règles traduit des spécifications de règles telles que nous avons donné dans la section 4.2.1.1 vers des objets et des méthodes O_2. La partie condition et la partie action d'une règle sont transformées en méthodes des classes O_2 qui modélisent les règles. Alors, la construction de la table de compatibilité des règles consiste à déterminer la compatibilité des méthodes de ces classes par une procédure équivalente à celle donnée dans la section 4.1.4.

Le module générateur de table de compatibilité calcule les vecteurs d'accès des méthodes qui correspondent à la condition et à l'action d'une règle. Ces vecteur sont ensuite composés pour former le vecteur d'accès de la règle. La composition des deux vecteurs est effectuée de tel sorte que le vecteur de la règle aura les modes d'accès les plus restrictifs des deux vecteurs sur des entités nommées communes. La table de compatibilité peut ensuite être construite et stockée sous la forme d'une liste O_2 où chaque élément donne la compatibilité entre deux règles du schéma.

4.2.3.2 Parallélisateur de règles

Le parallélisateur de règles intervient dans l'étape d'ordonnancement des règles candidates d'un cycle d'exécution. Certaines règles sont ordonnancés pour une exécution séquentielle tandis que d'autres vont être exécutées en parallèle.

La construction du plan d'exécution des règles tient compte des informations de mode de couplage, le calcul de l'effet net, les priorités et la compatibilité entre les règles. En particulier, le parallélisateur des règles utilise les relations de priorité et de compatibilité des règles pour déterminer le plan d'exécution de règles d'un cycle. La procédure pour vérifier la compatibilité des règles et ensuite l'utiliser dans la construction du plan d'exécution est équivalente à celle de la parallélisation des méthodes d'une transaction présentée dans la section 4.1.5.

Le parallélisateur des règles est également responsable de la création des activités pour l'exécution parallèle des règles et de leur synchronisation. Ainsi, dans le plan d'exécution, la condition et l'action des règles parallèles sont exécutées par des activités créées dynamiquement par le parallélisateur des règles. Le plan d'exécution est construit de telle sorte qu'à la fin d'un cycle toutes ses règles, séquentielles ou parallèles, ont fini leurs exécutions.

4.3 Conclusion

Dans ce chapitre nous avons d'abord décrit la conception et la mise en œuvre de notre prototype de parallélisation des transactions développé à l'aide du système O_2. Pour cela nous avons adopté une approche de conception par objet qui permet de détailler la structure et le comportement des classes qui implantent le prototype. Nous avons présenté l'intégration du prototype avec le système O_2.

L'architecture logique du prototype montre une structure modulaire qui permet de l'isoler du système O_2. Cette interface est mise en œuvre par les classes de la gestion du catalogue. Cette approche augmente la portabilité du prototype dans la mesure où l'intégration avec un autre système de base de données nécessite uniquement la révision de ces classes.

Le prototype a été entièrement écrit en C++ et a une taille d'environ 7000 lignes de code pour l'ensemble de spécification et d'implantation des classes.

Nous avons ensuite présenter notre approche pour la parallélisation des règles du système NAOS. Pour cela, nous avons utilisé la relation de compatibilité entre les méthodes. Nous avons défini une relation de compatibilité pour les règles fondée sur celle pour les méthodes. Nous avons décrit la construction des plans d'exécution des règles d'un cycle basé sur la relation de compatibilité des règles et la relation de priorité.

Finalement nous avons présenté les modules ajoutés au système NAOS pour permettre la construction d'une table de compatibilité des règles et pour construire les plans d'exécution des règles dans les cycles.

Chapitre 5

Parallélisation des Applications

5.1 Transactions classiques et parallélisme 129
5.1.1 Applications et transactions 130
5.1.2 Compatibilité des transactions 130
5.1.3 Transactions parallèles 132
5.1.4 Discussion 134
5.2 Transactions emboîtées 134
5.2.1 Définition 135
5.2.1.1 Propriétés 137
5.2.1.2 Gestion des verrous 138
5.2.2 Transactions emboîtées multi-activités 142
5.2.2.1 Couplage fort 143
5.2.2.2 Couplage faible 145
5.3 Transactions emboîtées multi-activités par parallélisation 149
5.3.1 Motivation 150
5.3.2 Isolation des transactions et des activités 151
5.3.3 Parallélisme horizontal et vertical 152
5.3.4 Exemple 154
5.4 Conclusion 155

Parallélisation des Applications

Dans le chapitre 3 nous avons expliqué notre approche pour l'exploitation du parallélisme dans le contexte des transactions. Ce chapitre analyse l'exploitation du parallélisme dans le contexte des applications. Une étude sur les différents façons d'exploiter le parallélisme dans un programme séquentiel s'avérerait trop complexe en dehors du contexte des applications bases de données. Par conséquent, nous analysons la possibilité de paralléliser une application en tenant compte des divers types de transactions. Nous considérons plus particulièrement les transactions classiques et emboîtées.

Le fait de considérer des transactions classiques permet la généralisation du modèle de parallélisation proposé au chapitre 3. Nous développons ces idées dans la section 5.1.

Nous étudions en section 5.2 le modèle de transactions emboîtées. Nous décrivons les techniques utilisées pour gérer l'exécution parallèle des (sous)transactions dans une application.

La section 5.3 présente nos extensions au modèle de transactions emboîtées pour exploiter le parallélisme intra-transaction.

5.1 Transactions classiques et parallélisme

Dans un modèle classique, les transactions d'une application sont exécutées de manière séquentielle. Le SGBD offre des techniques pour contrôler la concurrence entre les transactions de plusieurs applications, mais il ne fournit pas de mécanisme pour gérer le parallélisme possible entre les transactions d'une même application. Les transactions d'un ensemble d'applications peuvent être exécutées de manière concurrente, cependant chaque application s'exécute de manière séquentielle. Cela n'est pas surprenant puisque le modèle a été conçu pour des SGBD, fondés sur des systèmes d'exploitation multi-tâches.

Dans cette section, nous examinons la parallélisation des applications par la transformation de code. L'objetif consiste à exécuter en parallèle les transactions d'une même application. L'approche proposée est basée sur le modèle de parallélisation dé-

fini au chapitre 3. Nous cherchons à établir de façon statique l'indépendance entre deux transactions d'une même application permettant de les exécuter en parallèle. L'indépendance des transactions est directement dérivée de la relation de compatibilité entre leurs opérations. Nous utilisons cette relation pour déterminer les *transactions parallélisables*. Nous présentons également dans cette section les modifications apportées à l'algorithme de parallélisation des opérations d'une transaction.

5.1.1 Applications et transactions

Nous utilisons les notions d'instructions, de transactions et de variables d'une application définies dans le chapitre 3. Une application est définie comme un graphe de flot d'exécution semblable au GPES. Les sommets du graphe représentent les instructions de l'application et les arêtes indiquent leur ordre d'exécution. Nous adoptons la notation graphique introduite dans la section 3.3.1. Une transaction est représentée par un seul sommet qui encapsule son plan d'exécution parallèle.

La figure 5.1 montre un exemple d'application et sa représentation graphique. Les sommets T1 et T2 dénotent les transactions qui démarrent respectivement aux lignes 2 et 12. Les sommets représentant les transactions encapsulent leurs instructions. Les autres instructions de l'application sont directement représentées dans le graphe. Chaque sommet transactionnel peut être "étendu" de façon à ce que le graphe de la transaction soit visible. Par exemple, la transaction du sommet T1 est celle que nous avons utilisée tout au long du chapitre 3. Son graphe, après transformation, est donné par le plan d'exécution parallèle de la figure 3.12 à la page 86.

Les transactions qui forment une application sont exécutées de manière séquentielle en suivant l'ordre donné par le graphe de l'application. Cependant, certaines transactions sont *indépendantes* et peuvent être exécutées en parallèle. Nous pouvons donc appliquer notre modèle de parallélisation des transactions aux applications.

5.1.2 Compatibilité des transactions

Dans notre modèle du chapitre 3, l'indépendance des opérations est donnée par une relation de compatibilité représentée par une matrice. Lorsqu'on considère les applications, nous avons deux manières de déterminer l'indépendance entre les transactions :

- L'approche statique qui est celle que nous avons proposée pour la parallélisation des opérations des transactions. On doit alors définir une relation de compatibilité entre les transactions afin de pouvoir les exécuter en parallèle tout en gardant la cohérence sémantique de l'application. La relation de compatibilité

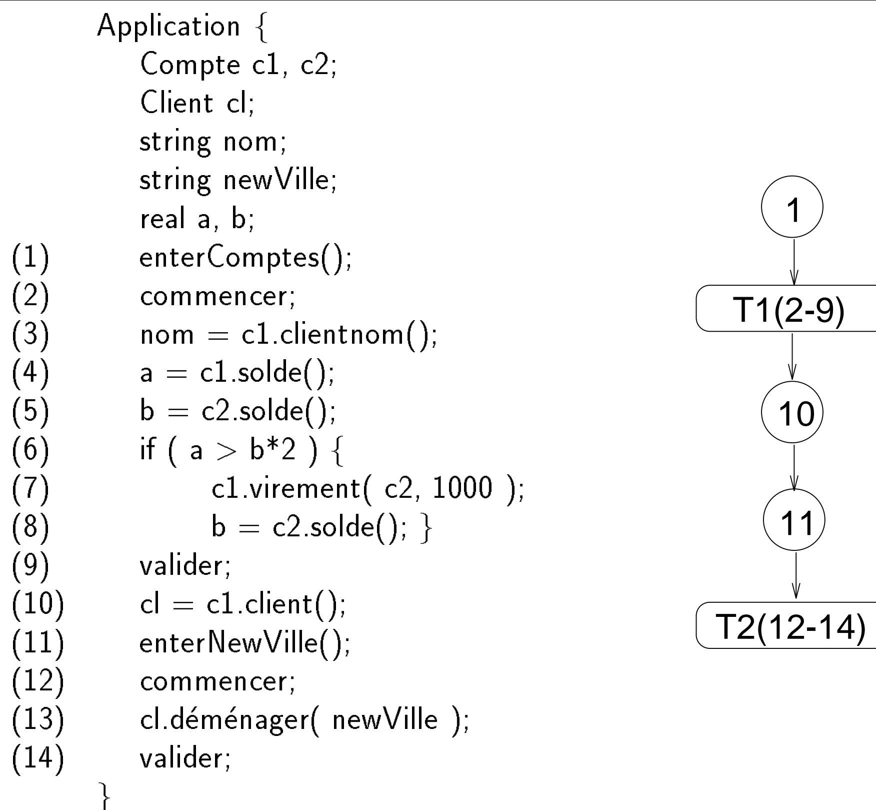


FIG. 5.1 - *Exemple d'une application*

est fondée sur le mode d'accès des instructions des transactions sur les variables de l'application. Les transactions compatibles sont celles qui ne possèdent pas de conflit d'accès sur les mêmes données et qui sont, par conséquent, parallélisables.

- L'approche dynamique est fondée sur les techniques classiques de contrôle de concurrence pour les transactions, parmi celles la plus utilisée on trouve le verrouillage des données [Pap86, GR93]. Cependant, dans un modèle de transactions atomiques, cette approche n'est pas envisageable puisque le gestionnaire des transactions du SGBD n'est pas capable de gérer des transactions parallèles dans une même application. Nous allons plutôt développer l'approche statique et nous réservons l'approche dynamique pour la section 5.3.

Comme nous l'avons montré dans la figure 5.1, le graphe d'une application est composé des sommets qui représentent des instructions et des transactions. Pour les raisons déjà évoquées dans la section 3.3.2, les instructions de branchement sont exécutées de manière séquentielle. Ainsi, dans la parallélisation d'une application, nous ne considérons que les instructions d'action.

A chaque sommet représentant une instruction d'action, nous pouvons associer deux ensembles qui donnent les modes d'accès de l'instruction sur les variables de l'application. Ces ensembles sont semblables aux ensembles d'accès des instructions d'une transaction de la section 3.3.2. Nous les utilisons pour déterminer la relation de conflit entre les instructions. La table 5.1 montre les ensembles d'accès des instructions d'action de l'application de la figure 5.1.

Instruction	lecture	écriture
(1)	$\emptyset$	{ c1, c2 }
(10)	{ c1 }	{ cl }
(11)	$\emptyset$	{ newVille }

TAB. 5.1 - *Ensembles d'accès des instructions d'action.*

Compte tenu des ensembles d'accès des instructions d'action d'une application, il est possible de calculer la relation *parallélisable* (Définition 3.5). Toutefois, pour pouvoir paralléliser le code d'une application, nous avons besoin de connaître aussi bien l'indépendance entre ses instructions et ses transactions que l'indépendance entre ses transactions. Si l'on considère une instruction de l'application comme une transaction ayant une seule instruction, déterminer l'indépendance entre instructions et transactions revient à déterminer l'indépendance entre deux transactions. Ainsi, nous définissons une nouvelle relation *ttParallélisable* qui donne l'indépendance entre deux transactions.

Définition 5.1 (ttParallélisable) *Soient i et j des instructions d'action, t et u des transactions, alors :*

$$ttParallélisable(t, u) - \forall i \in t,\ j \in u,\ parallélisable(i, j)$$

Cette relation permet la transformation du graphe d'une application vers un graphe avec parallélisme entre les transactions. Comme elle est fondée sur la relation *parallélisable*, nous pouvons garantir l'indépendance des instructions de deux transactions (voir Théorèmes 3.1 et 3.2 et leurs preuves à la section 3.3.2.3).

5.1.3 Transactions parallèles

Notre approche pour la transformation d'un graphe d'application suit celle proposée pour la transformation d'un graphe de transaction. Le graphe transformé est organisé en blocs des sommets ayant des transactions, suivis par des sommets ayant une instruction de synchronisation. Les sommets de ces blocs possèdent le même sommet prédécesseur et successeur, qui représente d'ailleurs le point de synchronisation.

Contrairement à l'algorithme de transformation des transactions, qui utilisait la relation *parallélisable*, celui de la transformation des applications utilise la relation *ttparallélisable* (Définition 5.1) pour construire les blocs de sommets. Par conséquent, pour que deux sommets du graphe d'une application forment un bloc, il faut que la transaction représentée par un sommet soit *ttparallélisable* avec celle représentée par l'autre.

En ce qui concerne la génération de code, nous utilisons également l'approche proposée pour la transformation des transactions. Ainsi, nous introduisons des commandes pour créer des activités parallèles dans lesquelles nous exécutons les transactions. Celles-ci sont synchronisées par une commande qui correspond au sommet successeur d'un bloc de sommets. Bien évidemment, on ne crée pas une activité à chaque fois qu'on a une instruction de l'application à exécuter. Ces instructions sont exécutées dans l'activité principale de l'application qui continue son exécution après avoir déclenché les activités correspondantes aux transactions parallèles.

Exemple

Reprenons l'exemple d'application de la figure 5.1. Pour simplifier le discours, nous appelons transactions les instructions représentées par les sommets 1, 10 et 11. La transaction du sommet 1 modifie la valeur des variables c1 et c2, selon le tableau 5.1. Ces variables sont utilisées dans la transaction représentée par le sommet T1. Par conséquent, la transaction du sommet 1 n'est pas *ttparallélisable* avec la transaction T1. En revanche, celle-ci ne met pas à jour les variables cl et c1, utilisées dans la transaction du sommet 10. Ainsi, ces deux transactions sont *ttparallélisables*. Une situation semblable est vérifiée lorsqu'on considère les transactions des sommets 10 et 11 puisque celles-ci ne manipulent pas de variables communes. De plus, la transaction du sommet 11 est également *ttparallélisable* avec la transaction du sommet T1. Cela n'est pas le cas quand on prend en compte les sommets 11 et T2. Leurs transactions ne sont pas *ttparallélisables* car la variable newVille est modifiée dans 11 et passée comme paramètre, donc accédée, dans une instruction de T2.

Le graphe transformé de l'application de la figure 5.1 est donné dans la figure 5.2(a). Les sommets dont les transactions sont *ttparallélisables* sont groupés dans un bloc pour dénoter le parallélisme entre leurs transactions. On introduit un point de synchronisation après le bloc de sommets avant de représenter le sommet qui correspond à la transaction T2. La figure 5.2(b) montre l'exécution de l'application après la génération du code correspondant au graphe transformé.

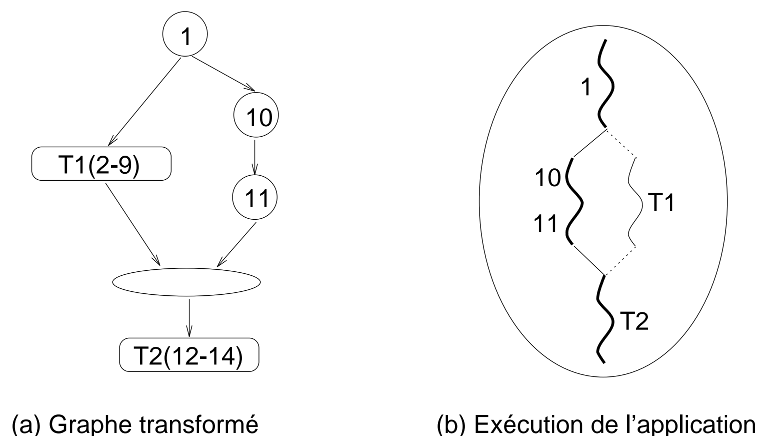


FIG. 5.2 - *Exemple de transformation d'application*

5.1.4 Discussion

Nous avons présenté une approche pour paralléliser le code d'une application où l'unité de parallélisation est la transaction. L'approche proposée est une *généralisation* du modèle que nous proposons pour les transactions. L'indépendance entre les transactions est la condition requise pour paralléliser une application.

Cependant la complexité du code des applications peut rendre très difficile la détermination de cette indépendance. Ce problème est signalé par la plupart des approches de parallélisation des programmes séquentiels [Bar78, LKK85, LER92, BGS93]. Il est dû d'une part, au partage, par toutes les transactions d'une application, d'un environnement d'exécution commun et d'autre part à la taille du code des transactions. En général, plus les transactions sont complexes, plus il sera compliqué de déterminer leurs indépendances.

Une solution intéressante consiste à laisser à l'application la responsabilité d'exprimer le parallélisme entre les transactions alors que le parallélisme intra-transaction est déterminé par une approche implicite telle comme nous l'avons définie dans le modèle du chapitre 3. Nous allons développer cette approche dans la section 5.3, où nous utilisons un modèle de transactions emboîtées.

5.2 Transactions emboîtées

Le modèle de transactions emboîtées permet l'exploitation du parallélisme à travers l'exécution parallèle de sous-transactions. Le parallélisme est donc décrit directement par l'application à travers la création des sous-transactions. Dans ce cas, la gestion du parallélisme au sein de l'application reste à la charge du gestionnaire des

transactions. Celui-ci gère automatiquement la création des activités pour exécuter les sous-transactions, le contrôle des accès concurrents aux données partagées par ces sous-transactions et la détection d'interblocage.

L'expression du parallélisme uniquement par l'intermédiaire des sous-transactions apporte une sémantique transactionnelle aux parties de l'application qui sont exécutées en parallèle. Cela maintient l'isolation des transactions, même en présence du parallélisme, à cause des propriétés transactionnelles. Cependant cette approche est trop restrictive vis-à-vis du parallélisme. D'une part, parce que la surcharge découlant de la gestion des transactions peut réduire les gains en performance apportés par le parallélisme. D'autre part, parce que l'on est obligé de créer des transactions à chaque fois que l'on souhaite exploiter du parallélisme. Dans cette section nous examinons la façon dont le modèle permet l'exécution parallèle des transactions, tout en assurant leurs propriétés.

5.2.1 Définition

Les transactions emboîtées sont structurées dans des hiérarchies des transactions et sous-transactions sous la forme d'un *arbre de décomposition* [CFR89]. Une transaction est, donc, découpée en plusieurs sous-transactions dans des niveaux d'emboîtement arbitraires. Il n'y a pas de contrainte en ce que concerne la profondeur de l'arbre de décomposition. La transaction à l'origine d'un arbre est appelée *transaction racine globale* (TL-Transaction en anglais). Nous la distinguons des transactions racines des sous-arbres. La figure 5.3 montre un arbre de décomposition où la transaction T1 est la racine globale et les transactions T5, T6, T3 et T7 sont les feuilles. T2 est la racine du sous-arbre formé de T2, T5 et T6.

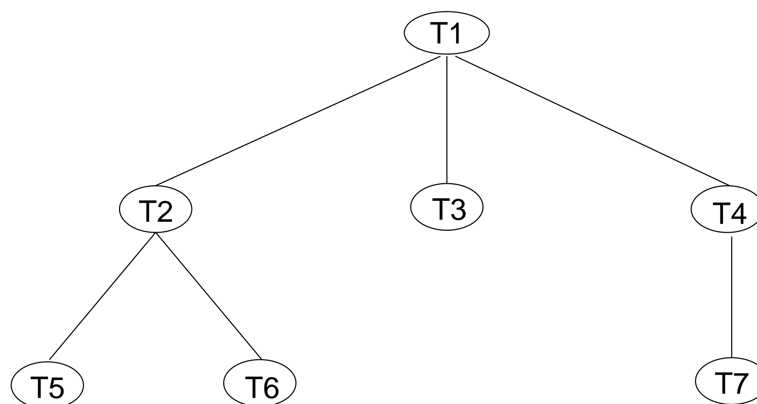


FIG. 5.3 - *Structure des transactions emboîtées*

Nous distinguons la *transaction mère*, celle qui déclenche une sous-transaction, et la *transaction fille*, celle qui a été déclenchée. Par exemple, T1 est la mère de la transaction T3, et celle-ci est sa fille. Nous appelons *ancêtres* d'une transaction t toutes les transactions de la fermeture transitive reflexive de t jusqu'à la racine. Les *descendantes* d'une transaction t sont les transactions appartenant à l'arbre de décomposition dont t est la racine. Dans la figure 5.3, T7, T4 et T1 forment l'ensemble des ancêtres de T7, tandis que T4 et T7 sont les descendantes de T4.

Les transactions emboîtées ont deux caractéristiques fondamentales qui les différencient des transactions classiques :

- Une transaction fille peut être annulée sans que cela force l'annulation de sa transaction mère. Étant un événement explicite du modèle, l'annulation d'une transaction fille est traitée au sein de sa transaction mère selon l'importance et la sémantique spécifique de la fille. Cette caractéristique permet un contrôle plus fin des reprises après panne que celui des transactions classiques. Bien évidemment on ne considère pas les transactions atomiques avec des points de reprise. Cette approche fournit un contrôle de reprise semblable à celui des transactions emboîtées. Cependant on ne peut pas traiter une "annulation" jusqu'au point de d'entrelacement car *le modèle* ne reconnaît pas cette "annulation".

- Une transaction mère est capable d'exécuter plusieurs transactions filles de manière concurrente. Ainsi on peut exploiter le parallélisme dans un arbre de décomposition. Par exemple, la figure 5.4(b) montre l'exécution de l'arbre de décomposition de la figure 5.4(a). Deux formes de parallélisme sont possibles :

  – Le *parallélisme vertical* qui permet l'exécution parallèle d'une transaction par rapport à ses filles. Dans la figure 5.4 cette forme de parallélisme est représentée par les exécutions des transactions T1 et T2 ou celles de T1 et T3. La transaction mère T1 continue son exécution après avoir créée la transaction fille T2, et ensuite crée l'autre transaction fille T3. Il faut noter que la transaction mère T1 s'exécute en parallèle avec ses transactions filles T2 et T3.

    Le modèle originel de transactions emboîtées proposé par Moss [Mos85] n'offre pas cette forme de parallélisme. Dans ce modèle, la création d'une transaction fille bloque l'exécution de la transaction mère. Des systèmes tels que Argus [Lis88] et Camelot [EMS91] adoptent le modèle de Moss. Le système Camelot utilise un constructeur tel que cobegin/coend (voir Chapitre 2) pour déclencher un ensemble de transactions filles parallèles. L'exécution de la transaction mère est arrêtée jusqu'à que toutes les filles

soient terminées (validées ou annulées). Ainsi, dans Camelot, il n'y a pas de parallélisme entre la mère et ses filles.

– Le *parallélisme horizontal* qui est caractérisé par l'exécution parallèle des filles. L'exemple de la figure 5.4 montre cette forme de parallélisme par l'exécution parallèle des transactions T2 et T3. Ce type de parallélisme est fourni par la quasi-totalité des systèmes ayant des transactions emboîtées (Clouds [DLA88], Eden [ABLN85], Camelot et Argus). Certaines approches de transactions emboîtées, notamment celle proposée récemment par l'OMG [CAD+94], ne supportent pas le parallélisme entre transactions filles.

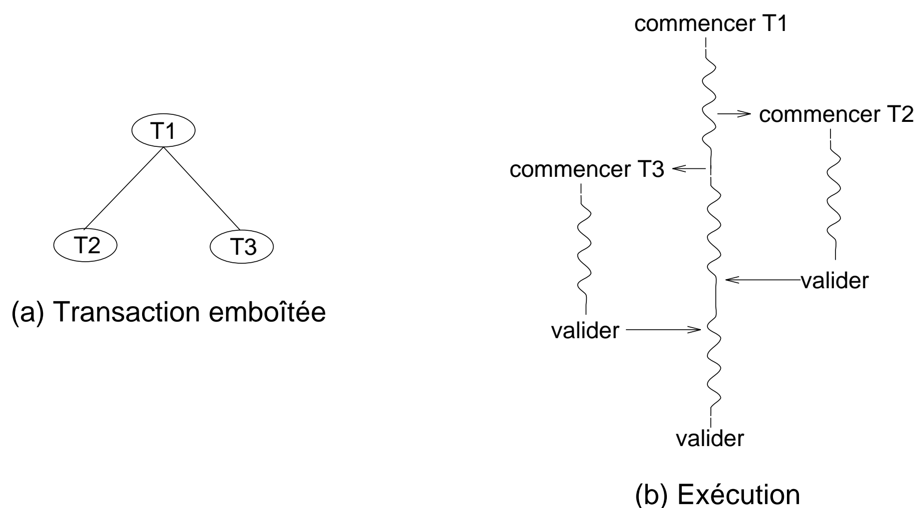


FIG. 5.4 - *Les types de parallélisme*

5.2.1.1 Propriétés

Les différentes transactions d'un arbre de décomposition ne possèdent pas les mêmes propriétés. La transaction racine globale est ACID (voir Chapitre 2 pour la définition des ces propriétés) et présente donc la même sémantique qu'une transaction atomique vis-à-vis des transactions qui n'appartiennent pas à son arbre de décomposition. En revanche, les sous-transactions ne possèdent pas la propriétés de Durabilité. Leurs résultats peuvent être perdus après "validation", notamment dans le cas d'annulation de la transaction racine globale. Ainsi, les modifications apportées aux données par une sous-transaction ne sont répercutées sur la base qu'au moment de la validation de la transaction racine globale. Autrement dit, l'annulation d'une

transaction engendre l'annulation de toutes ses descendantes, même si celles-ci sont déjà "validées". Bien évidemment, une transaction mère ne peut valider qu'après la terminaison de toutes ses filles.

Utilisant la nomenclature ACTA [Chr91], la relation ancêtre/descendante est caractérisée par la *dépendance d'annulation* d'une transaction par rapport à ses ancêtres et par la *dépendance de terminaison* d'une transaction vis-à-vis de ses descendantes.

Chaque transaction fille s'exécute de manière atomique et sérialisable par rapport à ses sœurs. Plus généralement, lorsqu'on considère un seul niveau de l'arbre de décomposition, toutes les transactions sont atomiques entre elles. De même que la racine globale est ACID par rapport aux autres transactions racines globales, les sous-transactions sont ACID vis-à-vis de leurs sœurs. L'ordre de validation des transactions sœurs donne l'ordre de sérialisation des transactions de même niveau.

Dans le modèle originel de Moss [Mos85], les opérations sur les données ne peuvent être exécutées que dans les *transactions feuilles*. Ainsi, les transactions mères servent à structurer le flot d'exécution et à déterminer le moment de la création des filles. Cette approche est également suivie par [FLMW89, FLW92] où on utilise la sémantique des opérations des TADs dans le contrôle de concurrence des transactions emboîtées. D'autres variations du modèle originel permettent la manipulation de données dans n'importe quelle transaction d'un arbre de décomposition [Wei89b, CC91, Elm92]. On n'impose pas la restriction d'accès aux données dans les feuilles. Dans nos propositions (section 5.3), nous adoptons la même approche et nous considérons donc un modèle où toutes les transactions peuvent manipuler la base de données.

5.2.1.2 Gestion des verrous

En général, la propriété d'isolation des transactions emboîtées est assurée par des protocoles de contrôle de concurrence pessimistes utilisant des verrous sur les données. La relation mère/fille entre les transactions apporte de nouveaux aspects dans le contrôle de concurrence entre les transactions d'un même arbre, puisqu'une transaction fille peut manipuler des données verrouillées par sa mère. Il s'agit, donc, d'exploiter le parallélisme dans un arbre de décomposition tout en assurant la sérialisation entre ses transactions. Nous allons utiliser la nomenclature introduite par [HR93] pour expliquer la gestion des verrous dans un arbre de décomposition.

Notation

Avant de manipuler un objet, une transaction doit acquérir un verrou sur cet objet. Nous utilisons les modes de verrouillage classiques [DA82] :

- S dénote un verrou en mode partagé.

- X dénote un verrou en mode exclusif.

Lorsqu'une transaction T acquiert un verrou sur un objet O dans le mode M, nous disons que T *tient* un verrou dans le mode M sur O. Par exemple, pour modifier un objet O, une transaction T doit tenir un verrou X sur O.

Une transaction peut également *retenir* un verrou sur un objet dans un des modes de verrous. La rétention d'un verrou sur un objet ne donne pas le droit à une transaction de manipuler l'objet. En revanche, un verrou retenu par une transaction T est interprété par toute transaction en dehors du sous-arbre de T comme un verrou classique. Par conséquent, lorsqu'une transaction retient un verrou sur un objet en mode exclusif, toute autre transaction qui n'appartient pas à la sous-arbre de T ne peut acquérir un verrou sur le même objet. Lorsqu'une transaction retient un verrou sur un objet, elle ne peut le relâcher qu'au moment de sa terminaison. Nous utilisons la notation suivante[1] :

- h:S(O) : une transaction *tient* un verrou en mode partagé sur l'objet O.
- h:X(O) : une transaction *tient* un verrou en mode exclusif sur l'objet O.
- r:S(O) : une transaction *retient* un verrou en mode partagé sur l'objet O.
- r:X(O) : une transaction *retient* un verrou en mode exclusif sur l'objet O.

Délégation de verrous

Dans le modèle de transactions emboîtées, il y a deux formes de mouvement des verrous entre les transactions d'un arbre de décomposition :

- La *délégation descendante* permet aux transactions descendantes d'une transaction T d'acquérir un verrou sur un objet déjà retenu par T. Par exemple, lorsqu'une transaction T *retient* un verrou en mode exclusif sur un objet O, toutes ses transactions filles peuvent acquérir un verrou en mode exclusif sur le même objet. Bien évidemment, une seule sous-transaction tient le verrou dans ce mode. Cette forme de délégation est prévue dans le modèle de base proposé par Moss et est offerte par tous les systèmes ayant des transactions emboîtées. Cependant d'autres approches ont fait évoluer le modèle en permettant une fille de T d'acquérir un verrou sur un objet déjà *tenu* par T [DLA88, HR93]. Dans ce cas, T qui tient par exemple h:S(O) met en disponibilité, de manière altruiste, son verrou vers ses descendantes par la transformation de h:S(O) en r:S(O). Alors n'importe quelle descendante de T peut après avoir h:S(O). Il est évident

1. "h" vient de *hold* et "r" de *retain* en anglais.

qu'une telle approche ne respecte pas la propriété d'isolation des transactions puisqu'il peut y avoir de partage de données entre une transaction et une de ses filles. Cependant cela ne pose pas de problème car (1) la transaction mère ne peut pas acquérir le verrou sur le même objet avant la fin de sa fille et (2) l'annulation de la mère provoque automatiquement l'annulation de ses filles.

- La *délégation ascendante* a toujours lieu au moment de la validation d'une sous-transaction. Elle consiste pour une transaction fille à transmettre à sa mère tous ses verrous (tenus et retenus). Les verrous tenus par la fille deviennent des verrous retenus par la mère. Ainsi d'autres filles de la même mère peuvent acquérir de verrous sur le même objet.

Règles d'acquisition

En conclusion, les verrous sont acquis par les transactions selon les règles suivantes :

1. Une transaction T peut acquérir un verrou en mode exclusif (1) si aucune transaction ne tient le verrou en mode exclusif ou en mode partagé et (2) si toutes les transactions qui retiennent le verrou en mode exclusif ou partagé sont ancêtres de T.

2. Une transaction T peut acquérir un verrou en mode partagé (1) si aucune transaction ne tient le verrou en mode exclusif et (2) si toutes les transactions qui retiennent le verrou en mode exclusif sont ancêtres de T.

3. Après la délégation ascendante, la transaction mère retient les verrous dans le même mode dans lequel ils était tenus ou retenus par la transaction fille.

4. Dans le cas d'annulation d'une sous-transaction, ses verrous sont relâchés et la délégation ascendante n'a pas lieu. Cette annulation ne met pas en cause les verrous des ancêtres de la sous-transaction.

5. Une transaction T tenant un verrou peut l'offrir à ses transactions filles. Après cela, T retient le verrou dans le même mode qu'elle le tenait avant.

Nous allons expliquer l'utilisation des règles des verrous à travers des exemples des arbres de décomposition. Dans les figures 5.5, 5.6 et 5.7, une ligne formée de traits délimite l'ensemble des transactions capables d'acquérir un verrou en mode exclusif, tandis qu'une ligne pointillée montre l'ensemble des transactions pouvant acquérir en verrou en mode partagé.

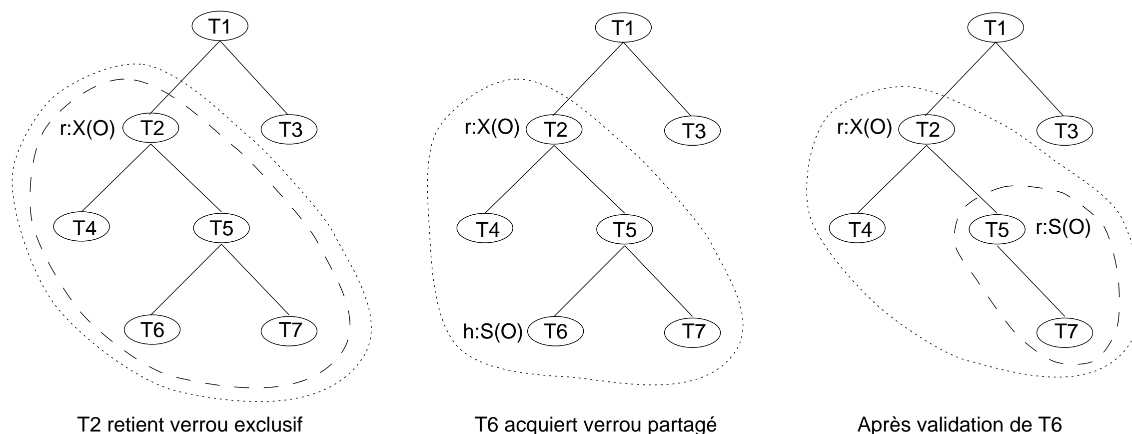


Fig. 5.5 - *Exemple de contrôle de verrou et la délégation ascendante*

La figure 5.5 montre un arbre de décomposition où la transaction T2 retient un verrou en mode exclusif sur un objet O. Alors, toutes les descendantes de T2 peuvent acquérir un verrou sur O dans n'importe quelle mode (règles 1 et 2). Cependant, les autres transactions de l'arbre n'ont pas le droit d'acquérir un verrou sur le même objet, notamment T1, mère de T2. Lorsque T6 obtient le verrou en mode partagé, les descendante de T2 peuvent encore verrouiller O dans le mode partagé, mais aucune autre transaction de l'arbre peut acquérir un verrou en mode exclusif. Après la validation de T6, la délégation ascendante a lieu et T5 retient donc le verrou sur O dans le mode partagé (règle 3). A cause de la règle 1, l'ensemble de transactions pouvant acquérir un verrou un mode exclusif sur O est formé uniquement par le sous-arbre dont T5 est la racine.

Dans la figure 5.6, la transaction T6 acquiert un verrou sur l'objet O en mode exclusif au lieu d'acquérir en mode partagé. Comme résultat, aucune autre transaction de l'arbre ne peut avoir un verrou sur O tant que la transaction T6 soit active (règle 1). Après la validation de T6, T5 retient le verrou en mode exclusif, ce qui limite l'ensemble de transactions pouvant acquérir un verrou sur O à T5 et T7. A ce point T2, mais aussi T5, retiennent un verrou un mode exclusif sur l'objet. Cependant, T2 et T4 n'ont pas le droit d'acquérir de verrou sur O car elles ne sont pas filles de T5 (règle 2).

La figure 5.7 donne un exemple de délégation descendante (règle 5). La transaction T2 tient un verrou en mode exclusif sur l'objet O, alors aucune autre transaction ne peut acquérir de verrou sur O. T2 peut donc offrir le verrou sur cet objet à ses transactions filles. Il suffit de retenir le verrou sur l'objet au lieu de le tenir pour que ces descendantes puissent verrouiller l'objet dans n'importe quelle mode. T2 garde

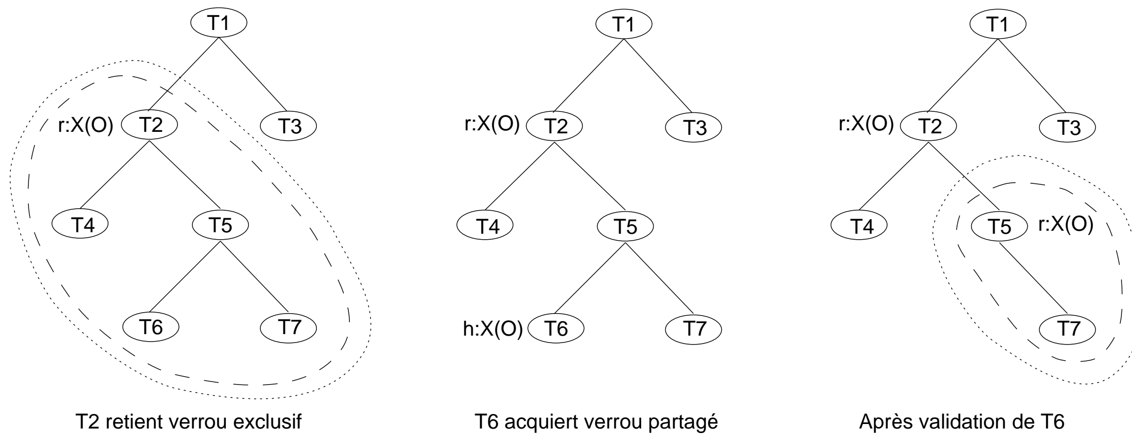


FIG. 5.6 - *Délégation ascendante et les verrous exclusifs*

toujours le contrôle sur le verrouillage de l'objet O car les transactions en dehors de son sous-arbre ne peuvent toujours pas verrouiller l'objet. Après la délégation descendante, on se retrouve au début de la configuration utilisée dans les exemples des figure 5.5 et 5.6. Notez que si une des descendantes de T2 verrouille l'objet en mode exclusif, T2 ne pourra plus accéder à l'objet jusqu'à la validation des transactions filles tenant le verrou.

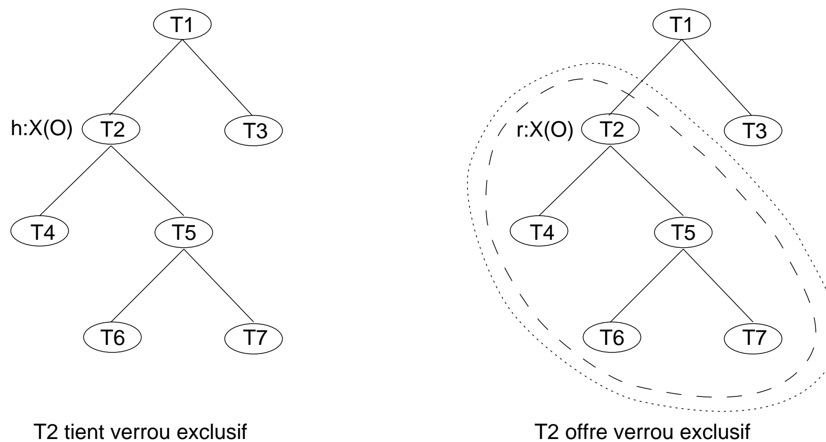


FIG. 5.7 - *Exemple de contrôle de verrou et la délégation ascendante*

5.2.2 Transactions emboîtées multi-activités

Compte tenu du modèle de transactions emboîtées, une application peut être modélisée comme une suite de transactions emboîtées dans lesquelles il y a, à la

fois du parallélisme vertical mais aussi du parallélisme horizontal. Dans une telle approche, le parallélisme est toujours lié au concept de transaction de tel sorte qu'il dépend directement de ce concept. Autrement dit, on ne peut avoir du parallélisme dans une application qu'au travers des transactions. En effet :

- La *propriété d'isolation* des transactions permet l'exécution parallèle sans interférence entre les transactions. Malgré le parallélisme entre les transactions, on obtient des résultats finaux équivalent à celui d'une exécution séquentielle des transactions. Par conséquent, en partant du principe qu'une exécution séquentielle est correcte, toute exécution parallèle équivalente est également correcte (voir section 2.3.2 pour plus de détails sur les propriétés des transactions et, en particulier, la définition de sérialisation).

- Il existe une *relation binaire* entre le concept de transaction et la *notion d'activité*, telle que nous l'avons vu dans les chapitres 2 et 3. Ainsi, une activité est générée lors de la création d'une transaction et celle-ci ne peut créer d'autres activités qu'au travers des transactions. C'est pourquoi on parle de parallélisme entre les sous-transactions et non pas de parallélisme entre les activités.

La relation entre les transactions et les activités prend plusieurs formes selon le modèle de transactions et la possibilité de gérer explicitement les activités dans les divers systèmes. Dans certains systèmes l'activité n'existe pas en tant que concept du modèle d'exécution du système de transactions et, par conséquent, elle ne peut pas être explicitement manipulée. Nous disons que le *couplage entre transactions et activités est fort*, puisqu'à chaque transaction correspond toujours une seule activité.

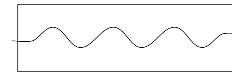
Dans d'autres systèmes les deux concepts existent. Nous disons alors que le couplage est faible, car les deux concepts sont reconnus par le modèle d'exécution et peuvent être utilisés de manière indépendante.

5.2.2.1 Couplage fort

En général, une transaction se limite à une seule activité qui correspond à l'activité de l'application ou bien à une activité générée de manière automatique par la création de la transaction. Dans ce cas, l'activité créée se termine à la fin de sa transaction. Dans les figures qui suivent, nous représentons une transaction par un carré et une activité par une ligne ondulée.

- Les *transactions classiques* possèdent une seule activité qui est, en fait, l'activité correspondante à l'application dans laquelle la transaction est créée.

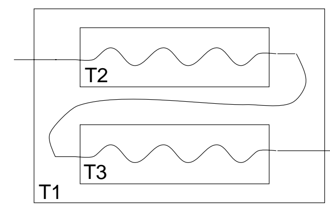
Dans la figure ci-contre, la transaction est exécutée par l'activité de l'application qui continue après la terminaison de la transaction.



Ainsi, dans les transactions classiques, la création d'une transaction n'entraîne aucune création d'activité.

- Les *transactions emboîtées mono-activité* offrent la possibilité de créer des sous-transactions qui sont toutes exécutées par une seule activité. Il s'agit également de l'activité correspondante à l'application. Les transactions d'un arbre de décomposition sont exécutées les unes après les autres par la même activité qui passe de transaction en transaction.

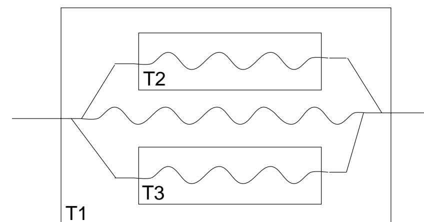
  Dans la figure ci-contre on voit la transaction T1 qui crée deux sous-transactions T2 et T3. A la fin de sf T2, l'activité continue et exécute ensuite T3. Après avoir exécuter T2 et T3, l'activité reprend le contexte de la transaction T1 et l'application.



  Ce modèle de transaction ne fournit aucune des formes de parallélisme que nous avons décrite dans la section 5.2.1. Le modèle d'exécution rassemble à un modèle classique d'appel de procédures avec l'avantage de pouvoir défaire l'exécution de certaines procédures dans le cas de défaillance. Certains modèles de transactions emboîtées pour des SGBD à objet suivent cette approche [HH91, AE92a, CF92, MRW+93].

- Dans les *transactions emboîtées mono-activité avec parallélisme* chaque transaction possède une seule activité mais la création d'une transaction entraîne également la création d'une activité. Par conséquent, une transaction ayant des sous-transactions peut avoir plusieurs activités correspondant chacune à une sous-transaction.

  La figure ci-contre montre le cas où il y a du parallélisme vertical entre les transactions. La transaction T1 est exécutée par l'activité de l'application. En revanche T2 et T3, sous-transactions de T1, sont exécutées chacune par une activité différente.



  Dans ce modèle, la création des activités est implicite et effectuée au sein des transactions par la création de sous-transactions. A la fin d'une sous-transaction son activité correspondante est arrêtée.

5.2.2.2 Couplage faible

Dans cette approche on peut à la fois gérer des transactions emboîtées mais aussi des activités au sein d'une application. Nous trouvons dans la littérature trois systèmes représentatifs de cette approche : Clouds [DLA88, CD89], Camelot [EMS91] et plus récemment Venari/ML [WFH+93, HKM+94]. L'étude de ces systèmes permet de mettre en évidance les techniques utilisées pour exploiter les parallélisme et les mécanismes de synchronisation des transactions et des activités mis en œuvre dans des tels systèmes.

L'approche Clouds

Le Système Clouds offre plusieurs types d'activités que nous divisons ici en activités standards et activités transactionnelles (transactions). Les activités standards sont celles que nous avons présentées dans le chapitre 2. Les activités transactionnelles forment un cas particulier d'activité. Elles sont atomiques et isolées les unes des autres. L'isolation des activités transactionnelles est gérée automatiquement par le système par un mécanisme de verrouillage implicite de données.

Les activités transactionnelles sont structurées dans des arbres de décomposition semblables à ceux des transactions emboîtées. La création d'une activité au sein d'une activité transactionnelle (activité mère) entraîne une sorte d'"héritage" des caractéristiques de la mère vers sa fille de sorte que la fille est également une activité transactionnelle. Par conséquent, il suffit de créer une activité transactionnelle au sein d'une application pour que toutes les autres activités (sous-transactions) créées à partir de celle-ci soient également transactionnelles.

Le système Clouds n'utilise pas le protocole de verrouillage lors de l'exécution d'une activité standard. Ainsi, il ne peut pas garantir l'isolation des activités. Le contrôle d'accès concurrent aux données doit être fait par l'application à travers des mécanismes tels que les sémaphores ou les moniteurs. De plus, dans le cas de mises à jour de données partagées par une activité standard et une activité transactionnelle, le système est incapable d'assurer l'isolation de cette dernière. Dans ce cas, la cohabitation des deux concepts dans une application met en cause les propriétés des transactions, car celles-ci ne sont plus isolées.

La figure 5.8 montre un exemple d'exécution d'une application Clouds (le cercle représente l'application). A un moment donné, l'application possède une transaction (T1) et deux activités standards A1 et A2. Le partage de données des transactions est automatiquement contrôlé par le système. En revanche ce contrôle ne concerne pas les activités A1 et A2. Ainsi, des données modifiées par la transaction (par exemple x=10 dans la sous-transaction T3 de T1) peuvent également être modifiées par une des activités standards *avant* la validation de T3 (par exemple x=20 dans l'activité

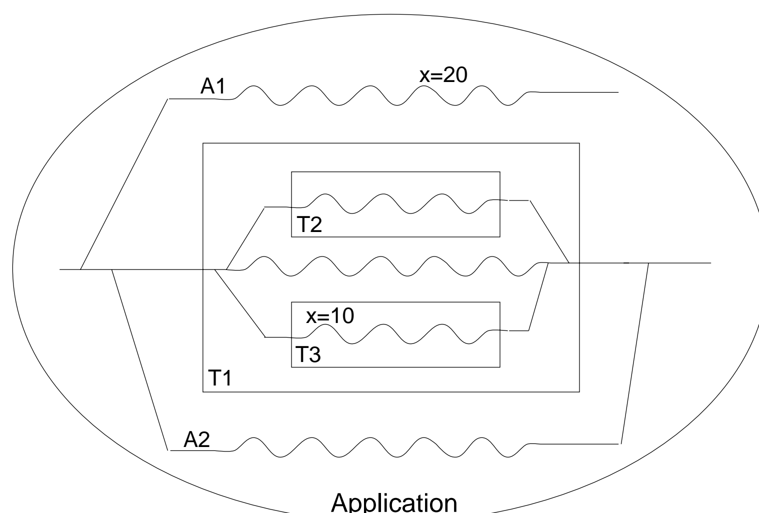


FIG. 5.8 - *Partage de données par les activités de Clouds*

A1). Cela a comme conséquence une perte de mise à jour dans la transaction T3 et, par conséquent, des valeurs non cohérentes dans la transaction T1 après la validation de T3. En outre, il n'y a pas de synchronisation entre les activités standards et les activités transactionnelles. Comme cela est le cas pour l'activité A1 et la transaction T1. L'activité A1 continue même après la validation de T1.

Le Système Camelot

La relation entre les activités et les transactions emboîtées dans le Système Camelot diffère de celle de Clouds sous plusieurs aspects. D'abord, une transaction peut avoir plusieurs activités dans Camelot, ce qui n'est pas possible dans Clouds puisque la création d'une activité dans une transaction entraîne automatiquement la création d'une sous-transaction à cause de l'"héritage" des caractéristiques entre les activités. Ensuite, dans Camelot, l'activité d'une transaction est toujours bloquée lors de la création d'une nouvelle transaction. Par conséquent, l'activité d'une transaction mère et d'une transaction fille n'exécutent jamais en parallèle. De plus, il n'est pas possible de créer des activités asynchrones dans Camelot. Le parallélisme entre les sous-transactions, et d'ailleurs entre les activités, est toujours fourni par l'utilisation d'un constructeur du type cobegin/coend.

L'isolation des sous-transactions dans Camelot est assurée par un mécanisme de verrouillage explicite de données. Avant la manipulation d'un objet, une transaction doit démander explicitement un verrou sur l'objet. Les verrous sont associés aux transactions. Si une transaction possède plusieurs activités parallèles et que celles-ci demandent, par exemple, un verrou sur un même objet en mode exclusif, elles

auront le droit de modifier l'objet puisque le verrou est acquis par la transaction. Du point de vue de extérieur à la transaction, celle-ci est atomique et isolée des autres transactions. Cependant, le système ne peut pas assurer une exécution correcte d'une transaction lorsqu'elle possède plusieurs activités.

Une application Camelot est donnée dans la figure 5.9. La transaction T1 crée une sous-transaction, T2 et deux activités A1 et A2. T1 est tout de suite bloquée et se poursuit après la terminaison des transactions et des activités qu'elle a créées. Le partage de données entre T2, T3 et T4 est contrôlé par le mécanisme de verrouillage et ne pose aucun problème (pourvu que les verrous soient bien demandés avant la manipulation des données).

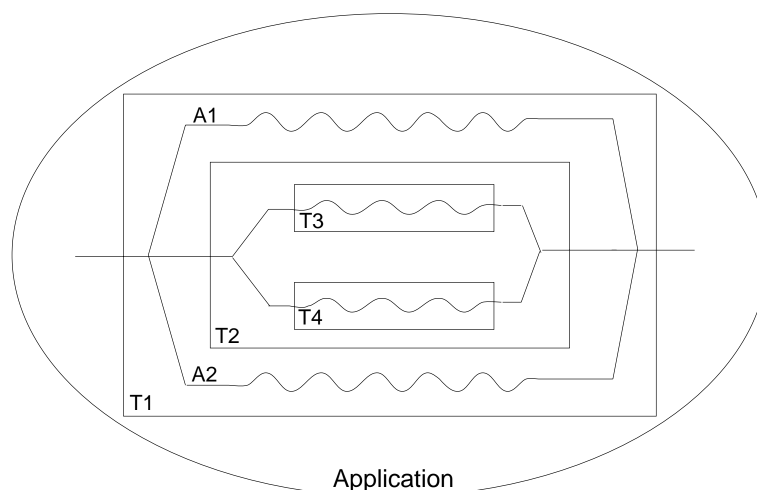


Fig. 5.9 - *Exemple d'application Camelot*

En revanche, le partage des données par les activités A1 et A2 n'est pas contrôlé par le mécanisme de verrouillage car ces deux activités s'exécutent dans une même transaction. Le résultat de la transaction T1 est donc imprévisible. Encore plus subtil : l'isolation des transactions T1 et de ses transactions filles ne peut pas être assurée car T1 partage les verrous avec ses transactions filles [2]. Par conséquent, une des activités de T1 peut modifier des données manipulées par une de ses sous-transactions. Lorsqu'on utilise uniquement des transactions, sans créer des activités, on n'est pas confronté à ce problème car la création d'une transaction arrête l'exécution de la transaction mère.

Le Système Venari/ML

2. Camelot utilise le protocole de contrôle de concurrence défini dans le modèle de transactions emboîtées de Moss où il n'y pas du parallélisme vertical.

L'objectif du Système Venari/ML consiste à étendre le langage fonctionnel *Standard ML* avec les concepts d'activité et de transaction [HKM+94]. Le concept de transaction est fourni dans Venari/ML par la composition, dans une activité transactionnelle, des propriétés des trois types d'activités suivantes :

- *Activités avec des verrous partagés/exclusifs (S/X)* : ces activités utilisent les verrous S/X pour contrôler l'accès à des données partagées "modifiables" (dans une application *Standard ML* il n'y a que certains types de données que peuvent être modifiés).

- *Activités offrant la persistance des données* : les données de ce type d'activité persistent à la fin de l'activité.

- *Activités atomiques* : l'effet de ces activités sur des données peuvent être défaits dans le cas de défaillance.

Le fait d'utiliser le concept d'activité pour construire des transactions n'élimine pas la cohabitation des deux concepts. Les activités de Venari/ML peuvent être entièrement indépendantes des transactions au sein d'une application. Ainsi on n'est confronté au problème d'accès concurrent à des données par les activités et par les transactions. Les transactions s'appuient sur un mécanisme de verrous S/X pour fournir l'isolation entre eux. En revanche, ce mécanisme ne peut pas être utilisé par les activités qui n'ont pas une sémantique transactionnelle. Ces activités utilisent des verrous d'exclusion mutuelle (*mutex*) pour contrôler le partage de données entre les activités. Dans les deux cas, les verrous sont explicitement demandés par l'application.

Les transactions du système Venari/ML peuvent être emboîtées. Le protocole de verrouillage est dérivé de celui du modèle de transactions emboîtées de Moss. Cependant, dans Venari/ML, la création des sous-transactions ne bloque pas l'exécution de la transaction mère. Par conséquent, le système offre le parallélisme vertical. Comme nous l'avons vu, le protocole de Moss ne garantit pas l'isolation des transactions en présence du parallélisme vertical. Alors, le protocole originel a été légèrement modifié pour pouvoir isoler la transaction mère de ses transactions filles : les verrous sur les objets doivent être re-demandés avant manipulation. Autrement dit, une transaction racine est obligée d'acquérir un verrou sur un objet avant de le modifier, même si elle avait déjà verrouillé l'objet en mode exclusif. Cette modification dans le protocole permet le blocage de la transaction mère uniquement au moment où elle a des problèmes de conflit d'accès à des données avec une de ses transactions filles.

Une transaction emboîtée peut, à la fois avoir des transactions filles mais également des activités qui ne sont pas des transactions, à l'exemple du système Camelot (voir la

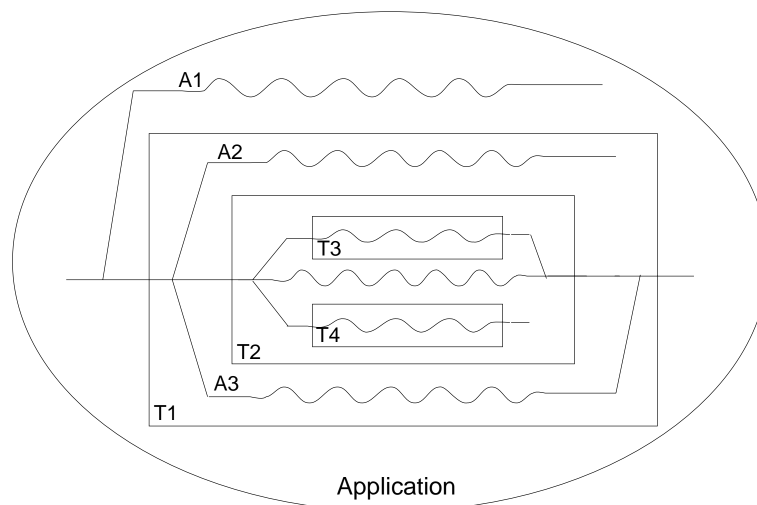


Fig. 5.10 - *Les activités et les transactions dans Venari/ML*

transaction T1 de la figure 5.9). On revient donc aux problèmes posés par le partage du contexte d'une transaction par des activités et par des sous-transactions.

La figure 5.10 donne un exemple d'application Venari/ML, où l'on note la présence de plusieurs activités et transactions emboîtées dans la transaction T1. Parmi toutes ces activités, une seule doit traverser la frontière délimitée par T1. Toutes les autres activités sont finies ou automatiquement "tuées" par l'activité principale de la transaction au moment de sa validation. Les activités sont asynchrones, ce qui oblige l'utilisation de barrières pour les synchroniser avant la validation. L'activité A1 se déroule dans le contexte de l'application indépendamment de toutes transactions. Ainsi cette activité peut mettre à jour n'importe quelle donnée de l'application, y compris celles verrouillées par une des transactions de T1.

5.3 Transactions emboîtées multi-activités par parallélisation

Notre approche fournit du parallélisme au sein des transactions sous deux formes différentes : à travers des sous-transactions, comme dans un modèle de transactions emboîtées classique, et à travers des activités dans une approche par transformation du code. Du point de vue de l'application, le modèle de transactions reste aussi simple à utiliser que le modèle de transactions emboîtées. Cependant nous introduisons une nouvelle technique qui exploite le parallélisme au sein des transactions qui reste, d'une certaine manière, cachée à l'application car il s'agit de la transformation de code.

5.3.1 Motivation

Dans l'approche à couplage fort entre transactions et activités, la notion d'activité n'est pas visible dans le modèle des transactions. Par conséquent on ne peut pas manipuler explicitement les activités au sein des transactions. Les transactions sont soit exécutées par l'activité de l'application, soit exécutées par une activité spécialement créée au profit de la transaction. Dans le premier cas, la gestion de partage des ressources de l'application ne se pose pas car il n'y a pas de parallélisme dans l'application. Les transactions classiques ou emboîtées sont toutes exécutées les unes après les autres.

Dans le cas où il y a création d'une activité pour exécuter une sous-transaction (*transactions emboîtées mono-activité avec parallélisme*), le problème de partage des données entre les activités parallèles est géré automatiquement par le système des transactions. Ainsi, on utilise le même mécanisme pour contrôler le partage des données entre les transactions et les activités parallèles d'une application. Cela est possible à cause de la relation binaire entre les transactions et les activités. Le système garantit qu'il aura toujours une seule activité par transaction (activité transactionnelle) et qu'il n'y a pas d'activité parallèle exécutée en dehors des transactions.

L'utilisation du mécanisme de verrouillage pour synchroniser les accès des données par les activités transactionnelles a l'avantage d'assurer l'isolation des transactions, malgré le parallélisme entre les activités au sein de l'application. De plus, le problème de *verrous mortels* est également traité par le système des transactions et, par conséquent, devient invisible en ce qui concerne l'application. Celle-ci n'a donc pas à gérer le problème d'interblocage entre les transactions parallèles.

En outre, dans l'approche de *transactions emboîtées mono-activité avec parallélisme*, ce parallélisme ne peut être exploité que par la création des transactions. Cependant, dans certains cas, il est souhaitable de pouvoir exécuter des activités en parallèle sans être obliger de créer des transactions. Surtout quand on considère le surchage de la gestion des transactions dans une application, par exemple : la gestion des verrous, la gestion des journaux, le contrôle de l'interblocage. Ainsi, il serait souhaitable de faire cohabiter les deux concepts dans un système de transactions, tout en assurant l'isolation des transactions entre elles et par rapport aux activités parallèles.

En ce qui concerne l'approche à couplage faible, les concepts de transactions et d'activités sont fournis, mais l'isolation des transactions ne peut pas être assurée par le système. L'isolation des transactions est assurée lorsqu'on n'utilise pas en même temps les activités parallèles et les transactions. Cela limite le parallélisme dans l'application uniquement par des transactions, et nous amène à une l'approche de *transactions*

emboîtées mono-activité avec parallélisme.

Le problème de fiabilité des transactions dans l'approche à couplage faible est dû notamment à deux caractéristiques des systèmes représentatifs de l'approche :

- Les systèmes utilisent des mécanisme différents pour garantir l'isolation des transactions et l'exclusion mutuelle des activités. Ces mécanismes sont *dynamiques* et ne communiquent pas les uns avec les autres. Par conséquent les données verrouillées par une transaction peuvent être modifiées par une activité et vice-versa.

- Le verrouillage explicite des données avant manipulation laisse à l'application la responsabilité du contrôle de concurrence entre les transactions, sauf dans le système Clouds qui utilise le verrouillage implicite. Ainsi, l'isolation des transactions devient dépendant de l'application. Comme résultat, le gestionnaire de transactions n'est plus capable d'assurer les propriétés transactionnelles.

Outre le contrôle d'accès concurrent entre les transactions, la gestion d'interblocage est également effectuée par l'application. Le gestionnaire de transactions ne prend pas en compte les verrous mortels au moment de concéder un verrou sur un objet à une transaction. Ainsi le système peut se trouver dans un état bloqué, où aucune des transactions d'une application ne peut continuer son exécution, car deux transactions de même niveau d'un arbre de décomposition sont en attente l'une par l'autre. Par conséquent, leur transaction mère ne peut pas valider à cause de la dépendance de terminaison et cela jusqu'à la transaction racine globale. Dans une telle situation, le gestionaire de transactions ne peut plus garantir la propriété d'atomicité, car il n'est plus capable d'assurer la terminaison des transactions.

Il serait donc intéressant de pouvoir exécuter des activités parallèles dans un modèle de transactions emboîtées sans mettre en cause les propriétés des transactions et sans avoir à créer des sous-transactions pour exploiter le parallélisme. Une telle approche doit (1) assurer l'isolation des transactions de façon à ce que les activités parallèles n'interviennent pas sur les données manipulées par les transactions et (2) laisser le contrôle de l'interblocage entre les transactions sous la responsabilité du gestionnaire de transactions.

5.3.2 Isolation des transactions et des activités

Dans l'approche que nous proposons, le contrôle de partage de données entre les transactions parallèles est toujours dynamique et effectué par un mécanisme de verrouillage implicite. Le protocole de verrouillage suit l'ensemble de règles données dans

la section 5.2.1.2. Ainsi nous pouvons garantir l'isolation des transactions lorsqu'on considère uniquement les transactions.

En revanche, le contrôle de partage de données entre les activités parallèles d'une transaction est effectué de manière statique, en se fondant sur la compatibilité des opérations de la transaction. Quand on considère uniquement les activités parallèles d'une transaction, nous pouvons assurer l'indépendance de ces activités (voir section 3.3.2.3). Par conséquent, on élimine le problème de non-déterminisme rencontré dans le système Camelot du à l'exécution de plusieurs activités dans une transaction sous le contrôle de l'application.

La cohabitation de deux concepts ne met pas en cause les propriétés des transactions. Cela est vrai en particulier dans notre modèle car il ne permet pas de parallélisme en dehors des transactions. Par conséquent, une application ne peut pas avoir une activité standard, i.e. non-transactionnelle, et une activité transactionnelle qui s'exécutent au même temps.

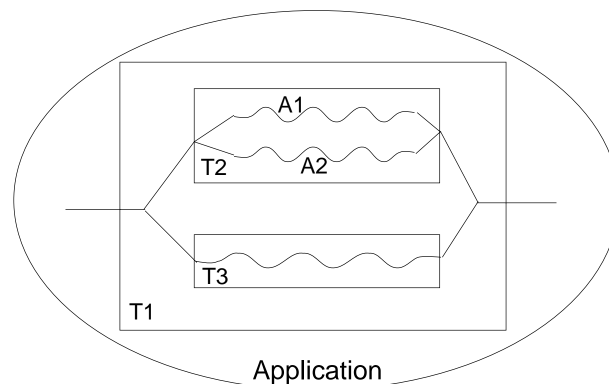


FIG. 5.11 - *Les activités et le parallélisme horizontal des transactions*

5.3.3 Parallélisme horizontal et vertical

La cohabitation des transactions et des activités au sein d'une transaction peut être analysée selon les types de parallélisme entre les transactions :

- Pour le parallélisme horizontal, le partage des objets entre les activités des transactions de même niveau est contrôlé par le mécanisme de verrouillage. Par exemple, la figure 5.11 montre une transaction emboîtée ayant des transactions filles, T2 et T3. Le fait que T2 ait deux activités parallèles ne provoque pas d'interférence entre T2 et T3 car les activités de T2 doivent toujours obtenir un verrou sur un objet avant de le manipuler. Ainsi, à un instant donné, l'activité

A1 peut se trouver bloquée en attendant la libération d'un objet verrouillé par T3 tandis que A2 continue normalement son exécution.

- Lorsqu'on considère le parallélisme vertical, le mécanisme de verrouillage est également capable d'isoler les activités parallèles d'une transaction mère par rapport aux activités parallèles des ses transactions filles. Prenons l'exemple de la figure 5.12 : les activités de la transaction T1 sont toutes les deux isolées des activités de la transaction T2, selon les règles de verrouillage de la section 5.2.1.2. Les activités A1 et A2 de la transaction T1 peuvent manipuler un objet uniquement dans le cas où T1 possède le verrou correspondant sur l'objet.

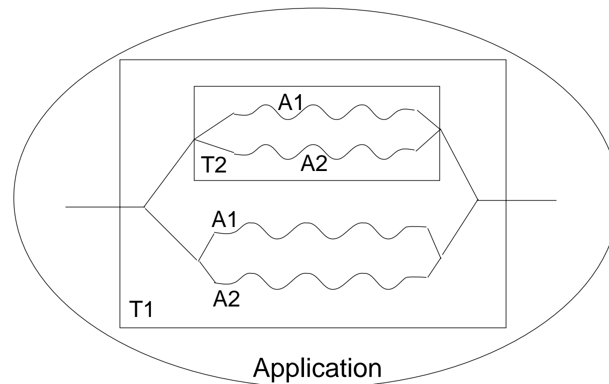
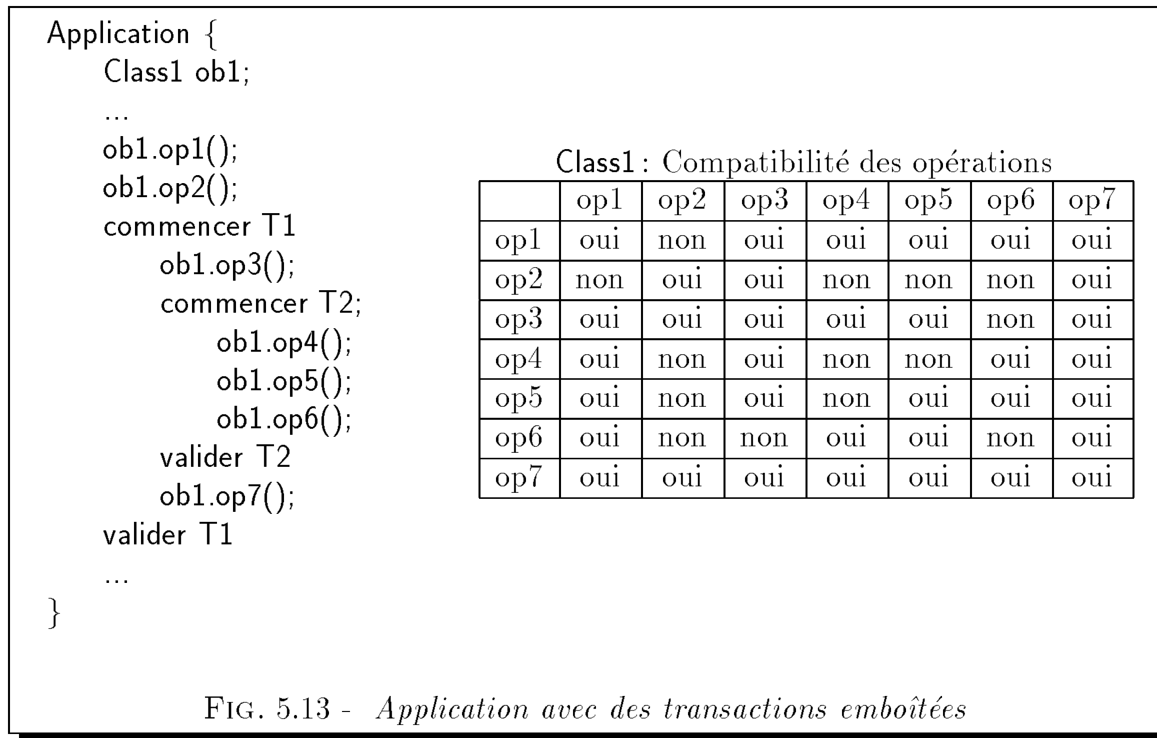


FIG. 5.12 - *Les activités et le parallélisme vertical des transactions*

Les verrous sont demandés par les activités qui s'exécutent aux sein des transactions. Toutefois, ils ne sont pas associés aux activités, mais aux transactions. Ainsi, c'est la transaction T2 de la figure 5.12 qui acquiert un verrou suite une demande faite dans le contexte de l'activité A2. Cette approche ne met pas en cause le contrôle d'interblocage effectué par le gestionnaire de transactions, car une activité d'une transaction ne s'est jamais bloquée par une autre activité de la même transaction. Par conséquent, le gestionnaire de transactions peut assurer la terminaison des transactions.

Notez que l'indépendance des activités d'une transaction élimine le problème de double demande de verrou en conflit par une même transaction à cause du parallélisme, car les activités parallèles ne modifient pas les mêmes objets. Si un objet est modifié plusieurs fois dans une transaction, nous pouvons garantir, par la propriété de compatibilité des opérations, que ces modifications ne sont jamais effectuées en parallèle. Ainsi une transaction ne pourra pas demander en parallèle des verrous en mode de conflit.

5.3.4 Exemple



```
Application {
    Class1 ob1;
    ...
    ob1.op1();
    ob1.op2();
    commencer T1
        ob1.op3();
        commencer T2;
            ob1.op4();
            ob1.op5();
            ob1.op6();
        valider T2
        ob1.op7();
    valider T1
    ...
}
```

Class1 : Compatibilité des opérations

	op1	op2	op3	op4	op5	op6	op7
op1	oui	non	oui	oui	oui	oui	oui
op2	non	oui	oui	non	non	non	oui
op3	oui	oui	oui	oui	oui	non	oui
op4	oui	non	oui	non	non	oui	oui
op5	oui	non	oui	non	oui	oui	oui
op6	oui	non	non	oui	oui	non	oui
op7	oui	oui	oui	oui	oui	oui	oui

FIG. 5.13 - *Application avec des transactions emboîtées*

La figure 5.13 présente le code d'une application où l'on déclare et crée un objet ob1, instance de la classe Class1. Plusieurs messages sont envoyés à ob1. La table à droite du code de l'application donne la compatibilité entre les opérations de la classe Class1. L'application déclenche la transaction T1 qui elle-même demarre une transaction T2. Certains messages sont envoyés dans le contexte de l'application, donc en dehors des transactions, tandis que d'autres sont envoyés à partir des transactions T1 et T2. Toutes les opérations appelées par T1 et par T2 sont compatibles, sauf op4 par rapport à op5 et op3 par rapport à op6.

La figure 5.14 montre le déroulement de l'exécution de l'application de la figure 5.13 après avoir été transformée. Les opérations op1 et op2 sont exécutées de manière séquentielle avant la création de la transaction T1. Le parallélisme vertical permet l'exécution de l'opération op3 et, en parallèle, créer une nouvelle transaction, appelée T2. Notez que l'opération op7 est également exécutée en parallèle avec l'opération op3 et la transaction T2. Cela est possible car op3 est compatible avec op7.

En ce qui concerne l'exécution de T2, l'opération op4 doit être exécutée avant l'opération op5 car celles-ci ne sont pas compatibles. Une fois op4 terminée, T2 exécute en parallèle les opérations op5 et op6.

Comme dans le modèle du chapitre 3, la fin d'une transaction marque un point de

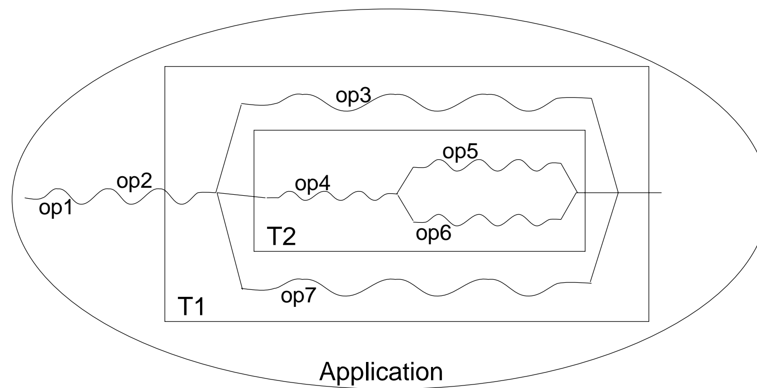


FIG. 5.14 - *Des transactions emboîtées multi-activités*

synchronisation entre les activités de la transaction. Par conséquent, il y aura toujours une seule activité à la fin des transactions. Ainsi, toutes les activités de la transaction T1, à savoir, celle qui exécute op3, celle qui corresponde à la transaction T2 et celle qui exécute op7, sont synchronisées avant la validation de T1.

5.4 Conclusion

Nous avons analysé en premier lieu la parallélisation des applications dans une contexte où nous ne disposons que des transactions classiques. Nous avons donc étendu le modèle présenté au chapitre 3 pour pouvoir l'employer au niveau des applications.

Nous avons ensuite considéré un modèle de transactions emboîtées et mis en évidence la relation entre transactions et activités dans plusieurs systèmes. Dans les systèmes où les transactions et les activités sont toutes les deux explicitement manipulées, certaines propriétés transactionnelles sont difficilement rassurées. Cela nous a motivé pour proposer une approche de parallélisation des transactions emboîtées. Cette approche offre des transactions multi-activités par transformation de code. Ainsi, nous proposons un modèle qui fournit à la fois les activités et les transactions tout en assurant l'isolation de celles-ci.

Chapitre 6

Etude des performances

6.1 Introduction 159

6.2 Le temps d'exécution d'une transaction 161

 6.2.1 Les transactions séquentielles 161

 6.2.2 Les transactions parallèles 163

6.3 Les gains apportés par la parallélisation 164

 6.3.1 Nombre de blocs et degré de parallélisme 165

  6.3.1.1 Graphe de parallélisation totale 165

  6.3.1.2 Graphe sans parallélisation 166

  6.3.1.3 Graphe de parallélisation partielle 166

 6.3.2 La surcharge 167

  6.3.2.1 Coût de la surcharge 168

  6.3.2.2 Temps d'exécution des opérations 172

6.4 Evaluation des stratégies d'exécution 177

 6.4.1 Complexité des opérations 178

 6.4.2 Ordonnancement des opérations 180

6.5 Les performances et les transactions emboîtées avec parallélisme . . . 184

 6.5.1 Temps d'exécution des transactions 184

 6.5.2 Gains en performance 185

6.6 Conclusion 188

Etude des performances

6.1 Introduction

Ce chapitre présente une étude de performances correspondant aux gains apportés par la parallélisation des transactions. L'objectif de cette étude consiste à :

- Identifier le coût de la surcharge de la gestion du parallélisme résultant de l'exécution parallèle des opérations ;
- Montrer que les transformations effectuées par notre algorithme de parallélisation apportent des gains de performances significatifs selon une relation entre la surcharge et la complexité des opérations ;
- Donner des critères de choix des stratégies différentes dans la génération du code parallèle. Ces critères sont utiles à la définition d'un modèle de coût qui pourrait piloter un module de parallélisation de transactions.

Il s'agit d'une étude théorique sur les gains en performance dans laquelle nous n'avons pas pu faire de mesures. L'expérimentation décrite dans le Chapitre 4 a été réalisée sur une architecture de machine monoprocesseur, dont le parallélisme exploité ne peut être que le *virtuel*. De plus, nous avons fait le choix d'expérimenter sur le système de gestion de bases de données à objets O_2 qui était disponible au sein de l'équipe dans laquelle nous travaillons. A l'heure actuelle, ce système ne fourni aucun support à l'exécution parallèle des applications. Ce système ayant une architecture client-serveur, nous avons pu générer du parallélisme virtuel au niveau des clients, mais les messages sont toujours traités de manière séquentielle au niveau du serveur.

Etant données ces conditions d'expérimentation, la prise de mesures sur des simulations ne montrerait probablement pas de gains très significatifs. Au contraire, il serait possible d'avoir des exécutions parallèles moins efficaces que des exécutions séquentielles à cause du coût de la gestion du parallélisme associé au parallélisme virtuel. Nous proposons donc cette étude des performances dans le but de mieux comprendre

les aspects liés à la surcharge produite par la gestion du parallélisme dans notre approche. Une fois ces aspects identifiés, nous montrons les gains de performances étant données certaines conditions de surcharge définies par une architecture matérielle et certaines caractéristiques des opérations définies par une application.

Le gain en performance est également fonction de la structuration des activités parallèles au sein des transactions. Nous avons ainsi considéré le problème de l'*ordonnancement des opérations parallèles* dans un graphe transactionnel. Nous montrons que l'utilisation d'*heuristiques* simples dans l'ordonnancement peut optimiser l'exécution parallèle des opérations d'un bloc et, par conséquent, améliorer davantage le temps d'exécution des transactions.

Notre étude considère une architecture de machine parallèle générique avec mémoire partagée et suffisamment de processeurs pour exécuter toutes les opérations d'un bloc en parallèle [Val90, FS92]. Cette hypothèse est raisonnable car notre approche a tendance à générer des blocs d'opérations parallèles qui ne possèdent pas énormément d'opérations. Cela est dû d'une part à une approche statique pour le calcul de la relation de compatibilité entre les opérations et d'autre part à la configuration des graphes générés par l'étape de transformation de code. Le calcul de la relation de compatibilité ainsi que la structuration des graphes transformés ont été décrits dans le Chapitre 3. Dans ce chapitre, nous évoquerons ces aspects à chaque fois qu'ils affectent les problèmes liés aux performances.

Cette étude ne comporte pas de comparaison avec d'autres approches pour la parallélisation de transactions car, à notre connaissance, aucun système de gestion de bases de données ne suit une approche semblable. Nous cherchons surtout à comparer les exécutions séquentielles des transactions avec des exécutions parallèles équivalentes produites par nos transformations. Différentes manières de structurer le graphe d'une transaction avec parallélisme sont également analysées. Cela permet l'évaluation de notre approche vis à vis des choix que nous avons fait pour l'ordonnancement des opérations d'un bloc.

Ce chapitre est organisé de la manière suivante : la section 6.2 présente les composants du calcul du temps d'exécution des transactions séquentielles et parallèles. La section 6.3 décrit les aspects qui déterminent les gains en performance apportés par la parallélisation des transactions. Ainsi dans cette section nous analysons nos choix concernant la structure des graphes des transactions parallèles et nous étudions la surcharge résultant de la gestion du parallélisme pendant l'exécution de ces transactions. La section 6.4 met en évidence le problème de l'ordonnancement des opérations d'un bloc. Enfin, la section 6.5 présente les gains en performance lorsqu'on prend en compte la parallélisation des applications dans le cadre des transactions emboîtées.

6.2 Le temps d'exécution d'une transaction

L'exécution d'une transaction consiste tout d'abord à effectuer une étape d'initialisation de la transaction, puis à exécuter ses opérations et à terminer avec des procédures pour valider ou annuler la transaction. Nous pouvons considérer le temps d'exécution d'une transaction comme étant la somme du temps d'initialisation, du temps pris pour exécuter les opérations, et du temps de terminaison.

Nous utilisons le graphe qui donne le plan d'exécution d'une transaction pour pouvoir mesurer son temps d'exécution. Pour cela, nous ajoutons à chaque sommet qui représente une action, le temps nécessaire pour exécuter l'opération correspondante. Dans un premier temps, nous allons utiliser un temps constant pour toutes les opérations d'une transaction.

Le graphe d'une transaction est construit au moment de la compilation. Il est donc impossible de connaître à cet instant les résultats de l'évaluation des primitives de branchement conditionnel présentes dans le code. Nous avons fait le choix de prendre le chemin le plus long pour pouvoir évaluer le temps global d'exécution d'une transaction. Il faut noter qu'un tel choix ne met pas en cause nos analyses de performances pour les raisons suivantes :

- Lorsqu'on compare une exécution séquentielle et une exécution parallèle de la même transaction, nous considérons toujours le même flot d'exécution.

- Nous assurons l'exécution séquentielle d'une transaction à l'entrée et à la sortie d'un bloc de commandes qui dépend d'une primitive de branchement conditionnel. Par conséquent, deux chemins de code possibles ne seront jamais exécutés en parallèle. De plus, pour un ensemble de données en entrée, une transaction exécute toujours le même code, indépendamment du modèle d'exécution (voir section 3.3.2.3 pour plus de détails).

L'ordonnancement d'une transaction est représenté par un *Diagramme de Gantt* qui montre de manière graphique le parallélisme entre les opérations indépendamment de l'architecture de la machine parallèle sous-jacente [LER92]. Un diagramme donne le flot d'exécution des opérations par unité de temps.

6.2.1 Les transactions séquentielles

Comme nous l'avons montré dans le Chapitre 3 par l'intermédiaire du graphe du plan d'exécution séquentielle d'une transaction, les opérations représentées par les sommets sont exécutées de manière séquentielle, par une seule activité. Le Diagramme

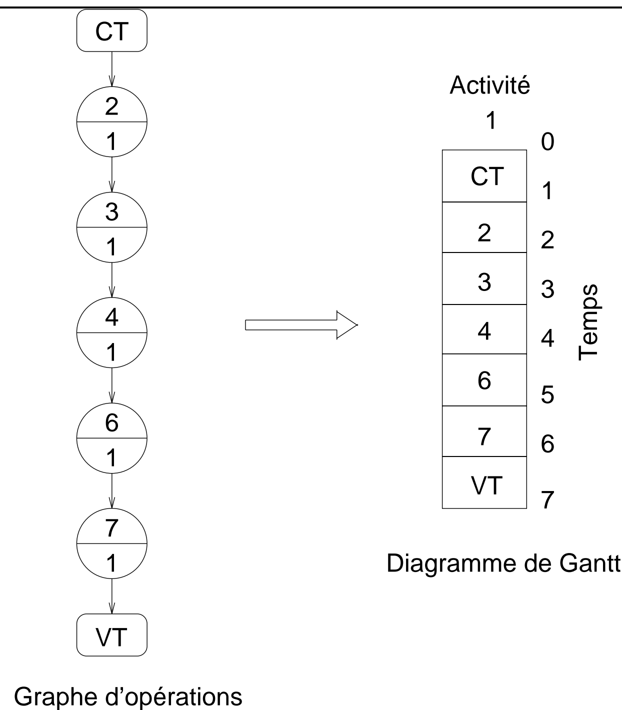


FIG. 6.1 - *Diagramme de Gantt d'une transaction séquentielle*

de Gantt d'un tel graphe représente donc l'ensemble des sommets et le temps d'exécution de la transaction correspond au temps de l'exécution séquentielle des opérations.

Comme exemple, nous pouvons reprendre la transaction de la figure 3.11. Son graphe d'opérations et le Diagramme de Gantt correspondant sont données dans la figure 6.1. Les sommets CT et VT représentent respectivement les instructions transactionnelles "Commencer" et "Valider" une transaction. Dans chaque sommet du graphe de la transaction nous représentons le numéro de l'opération dans la moitiée supérieure tandis que le temps d'exécution est montré dans la moitiée inférieure. Tous les sommets ont le même temps d'exécution qui correspond à 1 unité de temps (graphes UET [Kit94]). Ainsi pour une transaction séquentielle ayant 5 opérations, son temps d'exécution donné par le diagramme est de 7 unités de temps (ut) qui correspondent à 1ut pour initialiser la transaction, 5ut pour exécuter leur opérations et 1ut pour la valider. Plus généralement, le temps d'exécution d'une transaction composée de n opérations est donné par :

$$TS = Ti + \sum_{i=1}^{n} Top_i + Tv$$

où Ti est le temps d'initialisation ;

Top_i est le temps d'exécution d'une opération ; et

Tv est le temps de validation (annulation)

6.2.2 Les transactions parallèles

Contrairement aux transactions séquentielles, les transactions parallèles possèdent plusieurs activités, comme nous avons expliqué dans le Chapitre 3 par l'intermédiaire du graphe du plan d'exécution parallèle (GPEP). Certaines opérations du GPEP doivent être exécutées de manière séquentielle, tandis que d'autres sont ordonnancées pour une exécution parallèle. Les opérations séquentielles sont en effet exécutées par l'activité principale. Toutefois cette activité crée d'autres activités pour exécuter les opérations parallèles, après quoi les activités créées se terminent.

La figure 6.2 présente le graphe d'opérations et le Diagramme de Gantt qui correspondent au GPEP de la figure 3.12. Le diagramme montre l'exécution des opérations 2, 3, et 4 en parallèle, dans une seule unité de temps, tandis que selon le diagramme de la figure 6.1, 3ut sont nécessaires pour exécuter ces mêmes opérations.

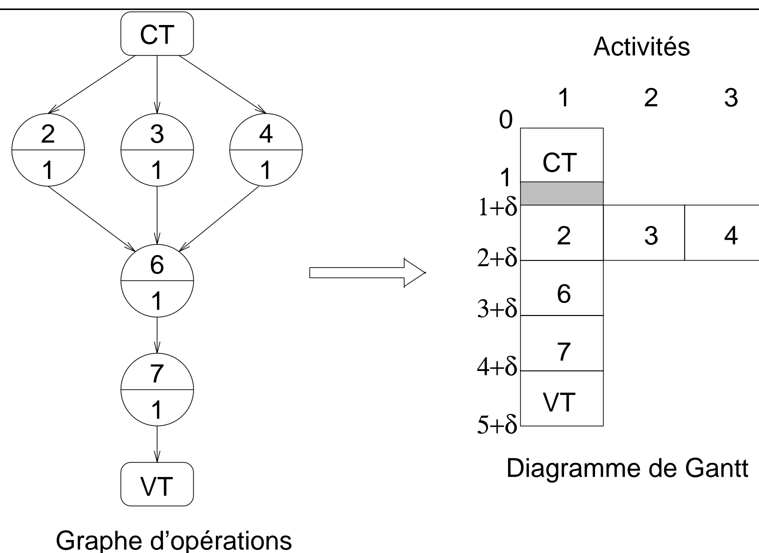


FIG. 6.2 - *Diagramme de Gantt d'une transaction avec parallélisme*

Dans une situation idéale, l'exécution des opérations 2, 3 et 4 en parallèle serait trois fois plus performante que l'exécution séquentielle de ces opérations. Cependant la gestion du parallélisme entraîne toujours un certain *coût* qui doit être pris en compte. Dans notre approche, cela se traduit par le temps de création des activités parallèles, auquel if faut ajouter le temps pour les synchroniser ce activités lorsqu'elles se terminent. La surcharge est représentée dans le diagramme par la zone hachurée. Le temps d'exécution de la transaction correspondant au Diagramme de Gantt de la figure 6.2 est donc de $5ut + \delta$, où δ représente le temps de la surcharge du parallélisme.

Les transformations que nous effectuons sur le code des transactions génèrent des graphes du type "fork/join", dits graphes de tâches séquentielles-parallèles [Gel89].

Les opérations séquentielles sont suivies de blocs d'opérations parallèles qui sont suivis d'opérations séquentielles et ainsi de suite. Le temps d'exécution d'une transaction parallèle est ainsi défini par le temps nécessaire à l'exécution de chacune de ses opérations séquentielles ajouté du temps pris pour exécuter les blocs d'opérations parallèles.

Le temps d'exécution d'un bloc est fonction du coût de la création de ses activités et du temps correspondant à l'exécution d'une opération du bloc. Le temps d'exécution d'un bloc est donc calculé par :

$$Tb = \delta + Topp$$

où $Topp$ est le temps d'exécution d'une opération du bloc.

Dans un graphe ayant n opérations séquentielles et m blocs d'opérations parallèles, nous modélisons le temps d'exécution d'une transaction parallèle par :

$$TP = Ti + \sum_{i=1}^{n} Tops_i + \sum_{j=1}^{m} Tb_j + Tv$$

où $Tops_i$ est le temps d'exécution d'une opération séquentielle;

Tb_j est le temps d'exécution d'un bloc d'opérations

La figure 6.3 montre le Diagramme de Gantt d'une transaction ayant 9 opérations dont 3 sont parallélisées dans un premier bloc et 4 le sont dans un second bloc. Ainsi le graphe présente 2 opérations séquentielles ($n = 2$) et 2 blocs d'opérations parallèles ($m = 2$). Supposons une surcharge de 10% du temps d'exécution de chaque bloc. Le temps d'exécution de la transaction serait ainsi calculé par :

$$TP = 1 + 2 + 2 * (0.1 + 1) + 1 \quad = \quad 6,2ut$$

Lorsqu'on compare le temps d'une version séquentielle de la transaction de la figure 6.3 (11ut) avec celui de la transaction parallèle, on voit que celle-ci est 56% (11/6,2) plus rapide que la transaction séquentielle. Autrement dit, la transaction parallèle exécute les mêmes opérations presque 2 fois plus vite que sa version séquentielle, malgré une surcharge de 10% du temps d'exécution dans chaque bloc.

Le degré maximum de parallélisme qu'une transaction peut atteindre est déterminé par le bloc d'opérations contenant le plus grand nombre d'activités parallèles. Dans la transaction de la figure 6.3 ce degré vaut 4.

6.3 Les gains apportés par la parallélisation

Les gains en performance d'une transaction parallélisée par rapport à une transaction séquentielle dépendent principalement des deux aspects suivants :

- Le nombre de blocs d'opérations générés et leur degré de parallélisme ;

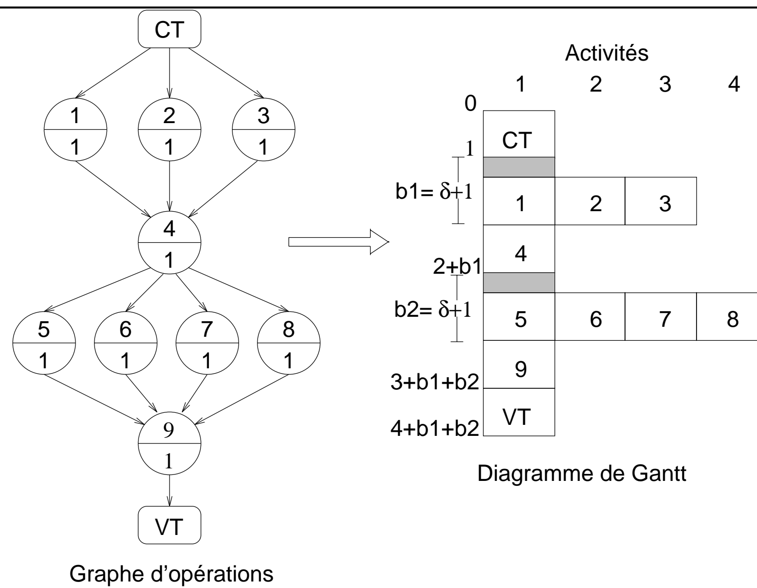


FIG. 6.3 - *Le temps d'exécution d'une transaction parallèle*

- La surcharge de la gestion du parallélisme vis à vis des opérations à exécuter dans un bloc.

Dans la suite, nous allons détailler chacun de ces aspects vis à vis des caractéristiques des transactions et des opérations.

6.3.1 Nombre de blocs et degré de parallélisme

La parallélisation des opérations d'une transaction est déterminée à la fois par l'ordonnancement des opérations—comme nous avons expliqué dans le Chapitre 3 par l'intermédiaire du graphe du plan d'exécution parallèle (GPES)— et par la relation de compatibilité entre ces opérations. Par rapport au nombre de bloc d'opérations parallèles, nous pouvons générer trois types de graphes différents : graphe de parallélisation totale, graphe sans parallélisation et graphe de parallélisation partielle.

6.3.1.1 Graphe de parallélisation totale

Un graphe de parallélisation totale est construit quand toutes les opérations d'une transaction sont *compatibles*. Il correspond à un bloc d'opérations parallèles, présentant un degré de parallélisme équivalent au nombre d'opérations dans le bloc de la transaction.

Le temps d'exécution est *presque* indépendant du nombre d'opérations. Ce temps n'est pas complètement indépendant car le nombre d'activités parallèles modifie la valeur de la surcharge d'un bloc. Toutefois on élimine de la formule de calcul du temps

d'exécution des transactions parallèles, un élément important : la partie qui calcule le temps d'exécution des opérations séquentielles.

Lorsque la parallélisation est totale, le temps d'exécution de la transaction est donc calculé par :

$$TP = Ti + \delta + Topp + Tv$$

Si l'on reprend la transaction de la figure 6.3, où $Topp$ était 1, celle-ci aurait un seule bloc d'opérations. Nous pouvons ainsidoubler la valeur de la surcharge puisqu'on double le nombre d'activités à créer dans le bloc, le temps d'exécution serait donné par :

$$TP = 1 + 0.2 + 1 + 1 \quad = \quad 3.2ut$$

Nous pouvons noter que, dans ces conditions, la version parallèle de la transaction effectue les mêmes opérations en 29% du temps de la version séquentielle. Autrement dit, la transaction parallèle est presque 4 fois plus rapide que sa transaction séquentielle correspondante.

6.3.1.2 Graphe sans parallélisation

Une transaction dont les opérations ne sont pas compatibles n'est pas transformée par l'algorithme de parallélisation. Le graphe généré est ainsi équivalent à celui d'une transaction séquentielle. Il s'agit d'un graphe sans parallélisation donc sans bloc d'opérations parallèles : toute la transaction sera exécutée par une seule activité. Bien évidement, dans ce cas, il n'y pas de gain en performance dans l'exécution de la transaction. Comme le code de la transaction n'est pas modifié, le temps d'exécution reste donc strictement équivalent à celui de la transaction séquentielle.

6.3.1.3 Graphe de parallélisation partielle

Entre la parallélisation totale et l'exécution séquentielle, les transformations d'une transaction peuvent apporter des gains de performance significatifs par la parallélisation d'une partie des opérations. Dans ce cas, le graphe transformé possède b blocs d'opérations parallèles, avec $b < n$, n étant le nombre d'opérations de la transaction. Ces transformations ne dépendent pas uniquement de la relation de compatibilité, mais également de la *séquence des opérations* dans la transaction. Il faut que les opérations soient à la fois compatibles et consécutives pour que l'algorithme de transformation puisse construire les blocs d'opérations parallèles.

Une transaction avec une forte concentration d'opérations compatibles aura probablement un GPEP avec la plupart de ses opérations ordonnancées pour une exécution parallèle. Dans le meilleur des cas, toutes les opérations sont assemblées dans un bloc parallèle, ce que nous amène au cas de la parallélisation totale. Dans le pire des cas,

chaque opération p est suivie d'une opération qui n'est pas compatible avec p. Par conséquent, aucun bloc d'opérations n'est créé, malgré la compatibilité des opérations. Nous retrouvons ainsi le cas des graphes sans parallélisation.

Le graphe de la transaction de la figure 6.3 à la page 165 est un exemple de parallélisation partielle. Considérons maintenant cette transaction avec l'opération 4 ordonnancée après l'exécution de l'opération 8. Si les opérations représentées par les sommets 1 à 3 et 5 à 8 sont toutes compatibles, le graphe généré et son Diagramme de Gantt sont ceux de la figure 6.4.

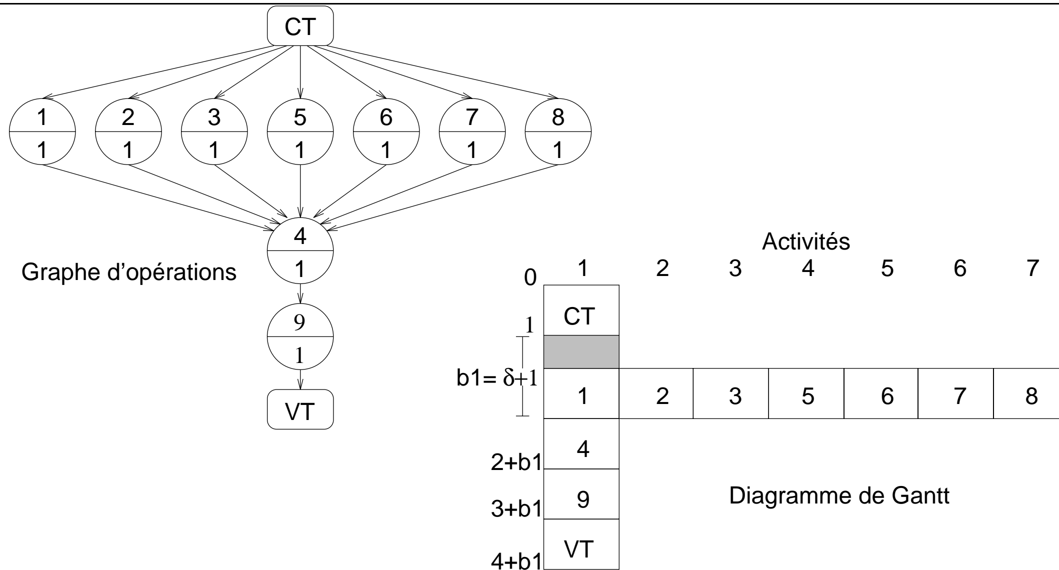


FIG. 6.4 - *Exemple d'un graphe de parallélisation partielle*

A travers l'exemple de la figure 6.4, nous pouvons noter que le nombre de blocs a été réduit mais le nombre d'opérations parallèles s'est maintenu constant. Autrement dit, le degré de parallélisme du seul bloc du graphe de la figure 6.4 est équivalent à la somme des degrés de parallélisme des blocs du graphe de la figure 6.3. On peut conclure que les gains en performance ne viennent pas directement du nombre de blocs d'opérations parallèles, mais surtout du degré de parallélisme de l'ensemble de blocs de la transaction.

6.3.2 La surcharge

Jusqu'à présent, nous avons utilisé un taux de surcharge qui dépendait uniquement du nombre d'activités des blocs d'opérations. Dans cette section, nous allons identifier les éléments qui déterminent le coût de la gestion des activités vis à vis du modèle de parallélisation que nous proposons. Notre objectif ne consiste pas à mesurer la valeur de la surcharge des blocs d'opérations, car celle-ci dépend avant tout de l'architecture

de la machine parallèle dont on utilise pour exécuter les transactions. Etant donné les hypothèses de départ, nous cherchons à :

- Identifier dans quelle mesure le coût de la surcharge n'est pas compensé par le parallélisme dans les blocs d'opérations.

- Analyser les gains en performance lorsque le temps d'exécution des opérations n'est plus constant.

6.3.2.1 Coût de la surcharge

La surcharge de la gestion du parallélisme dans un bloc d'opérations prend en compte le coût de création des activités et le temps nécessaire pour signaler leur terminaison. La création des activités et la signalisation de terminaison entraînent des envois de messages de contrôle entre les processeurs. Le nombre de messages est déterminé par le nombre d'opérations d'un bloc puisque, en principe, chaque opération doit être exécutée par une activité.

Une des opérations est effectuée par l'activité principale. Ainsi, dans le calcul du nombre de messages, on doit considérer le nombre d'opérations d'un bloc moins une opération. Cela se justifie par le fait que l'activité principale n'est pas créée puisqu'elle existe déjà. En plus, cette activité n'envoie pas de message de signalisation car il s'agit de l'activité en charge de la synchronisation des autres activités du bloc.

Par exemple, un bloc de 7 opérations, comme celui de la figure 6.4, génère 12 messages de contrôle, dont 6 correspondent à la création des activités et les 6 autres correspondent à la signalisation de terminaison d'activité.

Il faut noter que les activités sont indépendantes les unes des autres. Par conséquent, nous éliminons tous les retards dûs à la synchronisation entre les activités à cause du partage de données. De plus, dans une machine parallèle à mémoire partagée, la mise en œuvre des mécanismes de synchronisation de terminaison des activités peut être très efficace, correspondant à une mise à jour d'une position mémoire. On pourrait ainsi remplacer les messages de signalisation du coût de la surcharge par des mises à jour mémoire. Toutefois il faut retenir que ce coût sera toujours fonction du nombre d'activités à créer.

Lorsqu'on considère le temps de création C d'une activité comme constant et le temps d'accès mémoire comme uniforme, le coût de la surcharge d'un bloc de n opérations parallèles est donné par :

$$\delta = C * (n - 1)$$

où C sera donnée comme paramètre de la machine parallèle.

La figure 6.5 montre la composition de la surcharge du bloc d'opérations b1 du Diagramme de Gantt présenté dans la figure 6.4. Au temps d'exécution de chaque opération est ajouté le temps nécessaire à la création de l'activité qui exécute l'opération, sauf pour l'opération 1 qui sera exécutée par l'activité principale. La surchage du bloc b1, représentée par δ, est ainsi formée par la sommes des temps pris par la création des activités du bloc.

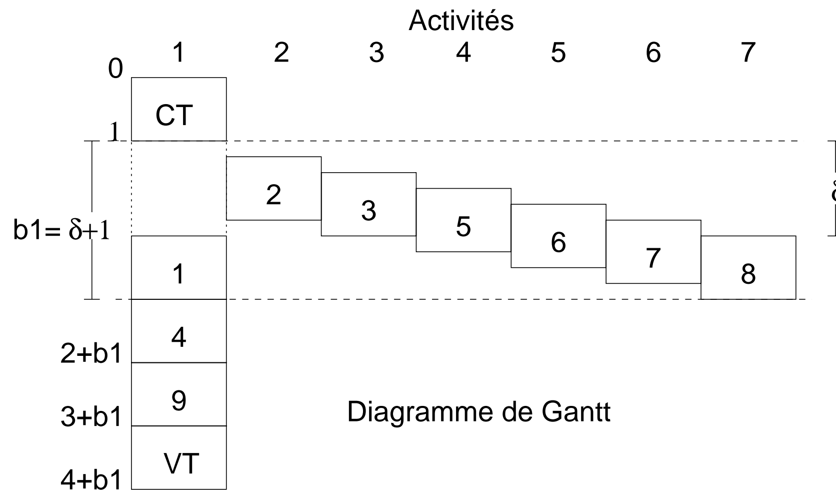


FIG. 6.5 - *Exemple détaillé de formation de la surcharge*

Le temps d'exécution d'un bloc est ainsi calculé par :

$$Tb = C * (n - 1) + Topp$$

Dans le but de simplifier le discours, nous allons parler de C comme le temps de création des activités. Cela ne bouleversera pas la compréhension du texte car la partie la plus importante de C est justement la création des activités. Le temps d'accès mémoire pour signaler la fin d'une activité a tendance à être beaucoup plus petit que celui utilisé pour la création d'une activité [FW78].

La figure 6.6 montre le temps d'exécution des 3 types de blocs d'opérations différents : un bloc de 3 opérations, un bloc de 7 opérations et un bloc de 10 opérations. Les pointillés donnent le temps d'exécution des blocs dans le cas où le temps d'exécution des opérations est constant et sa valeur est 0.5ut. Les lignes donnent le temps d'exécution des blocs lorsque le temps d'exécution des opérations est constant et égal à 1ut. Les points mis en évidence dans les courbes de temps indiquent l'instant où le temps d'exécution d'un bloc d'opérations est équivalent au temps d'une exécution séquentielle des mêmes opérations.

Les courbes de la figure 6.6 montrent trois aspects importants :

- Au fur et à mesure que le nombre d'opérations d'un bloc augmente, le coût de

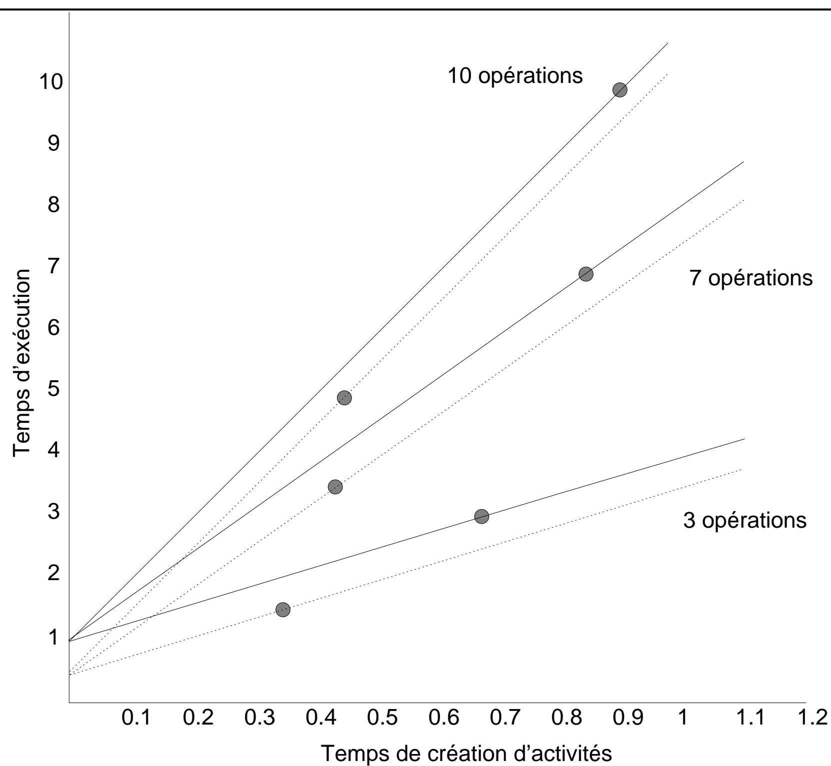


FIG. 6.6 - *Temps d'exécution des blocs d'opérations*

création des activités devient de moins en moins important. Cela est montré à la fois par les courbes des blocs d'opérations de 0.5ut, mais aussi par celles des blocs d'opérations de 1ut. Par exemple, selon les courbes des blocs d'opération de 1ut, quand on parallélise 3 opérations, on peut avoir un temps de création d'activités pouvant aller jusqu'à 60% du temps d'exécution d'une opération et encore gagner en performance. Pour le bloc de 10 opérations, le temps de création d'une activité peut aller jusqu'à 85% du temps d'exécution d'une opération. A nouveau l'exécution parallèle est plus performante que l'exécution séquentielle.

- Les courbes montrent également que l'on gagne plus lorsque les blocs contiennent plus d'opérations. Par exemple, si l'on considère le temps de création d'activités comme étant 0.2ut et le temps des opérations 1ut, l'exécution du bloc ayant 3 opérations est deux fois plus rapide que l'exécution séquentielle (exécution parallèle en 53% du temps de l'exécution séquentielle). L'exécution du bloc ayant 7 opérations est à peu près deux fois et demi plus performante que l'exécution séquentielle (35% du temps) et l'exécution du bloc ayant 10 opérations est plus de trois fois plus rapide que l'exécution séquentielle (30% du temps).

- Les gains en performance sont sensibles aux temps d'exécution des opérations. Cela est montré par le décalage des points d'équivalence entre les exécutions séquentielle et parallèles de blocs ayant le même nombre d'opérations. Considèrons les courbes des blocs de 7 opérations. Lorsque les opérations s'exécutent en 0.5ut, le temps de création d'une activité doit être limité à 40% du temps d'exécution d'une opération pour qu'on puisse avoir des gains en performance. En revanche, si l'on double le temps d'exécution d'une opération, on peut supporter une surcharge de création d'activité jusqu'à plus de 80% et encore avoir de gains en performance. Les aspects liés au temps d'exécution des opérations sont détaillés dans la section suivante.

Nous abordons ici le problème du rapport entre la surcharge produite par un bloc de n opérations et la surcharge produite par l'exécution en parallèle des mêmes opérations dans deux blocs de p et q opérations ($n = p + q$). Reprenons l'exemple de la figure 6.4, où nous avons remplacé deux blocs de 3 et 4 opérations par un seul de 7 opérations. Si l'on considère le coût de création d'une activité comme étant 0.5ut, le temps d'exécution du bloc de 7 opérations est donné par :

$$Tb_7 = 0.5 * (7 - 1) + 1 \quad = \quad 4ut$$

En revanche, le temps d'exécution de deux blocs de 3 et de 4 opérations est donné par :

$$Tb_{3+4} = 0.5 * (3 - 1) + 1 \quad + \quad 0.5 * (4 - 1) + 1 \quad = \quad 4,5ut$$

La transaction ayant un seul bloc est un peu plus performante que celle ayant deux blocs pour les raisons suivantes :

- Le rassemblement des opérations dans un seule bloc parallélise une opération de plus, puisque dans deux blocs cette opération est exécutée par l'activité principale.

- Le gain de temps ne correspond pas au temps d'exécution de l'opération parallèle supplémentaire, à cause de la surcharge produite par une activité parallèle de plus.

Dans le cas où le temps d'exécution des opérations est constant, le rapport entre le temps pris par un bloc de n opérations et celui nécessaire pour exécuter deux blocs de p et q opérations ($n = p + q$) est donné par l'expression suivante :

$$Tb_n = Tb_p + Tb_q - T_{op} + C$$

Ainsi, si le temps d'exécution d'une opération est beaucoup plus grand que celui nécessaire à la création d'une activité, le gain en performance produit par le rassemblement des opérations devient significatif. Nous reviendrons sur ces aspects dans la section 6.4.

6.3.2.2 Temps d'exécution des opérations

L'objectif de cette section consiste à analyser la surcharge produite par le parallélisme dans les blocs d'opérations lorsque le temps d'exécution des opérations est variable. Nous cherchons à identifier le niveau de surcharge acceptable dans les blocs de façon à ce que l'exécution des opérations en parallèle soit plus efficace que l'exécution séquentielle.

Le temps d'exécution d'une opération dépend largement de la *complexité du code* qui la met en œuvre. Dans les cas les plus simples, l'opération n'effectue qu'une instruction élémentaire, comme par exemple la mise à jour d'un attribut de l'état de l'objet. En général, les *opérations simples* sont peu coûteuses en terme de temps d'exécution. En revanche, l'opération peut être la racine d'un graphe de contrôle et son exécution représente un temps significatif par rapport au temps d'exécution de la transaction. De plus, certaines opérations, dites complexes, peuvent être très coûteuses en terme de temps à cause de la présence, par exemple, d'une boucle dans leur code.

Il est évident qu'une analyse détaillée de la complexité du code des opérations est en dehors des objectifs de cette étude. Toutefois, nous pouvons identifier les conditions dans lesquelles, étant donné une surcharge du parallélisme, notre approche produit des exécutions parallèles plus performante que les exécutions séquentielles correspondantes. Pour cela, trois aspects doivent être étudiés :

- Le temps d'exécution d'un bloc par rapport au temps d'exécution de ses opérations.
- Le coût de la création d'une activité par rapport au temps d'exécution d'une opération.
- L'ordonnancement des opérations dans un bloc.

Considérons le graphe d'une transaction avec parallélisme donné dans la figure 6.7. Par exemple, l'opération 6 prend 3ut pour s'exécuter tandis que l'opération 9 est exécutée en 0,5ut. Nous désignons l'opération la plus coûteuse d'un bloc, celle qui présente le temps d'exécution le plus grand parmi les temps d'exécution des opérations du bloc. Il s'agit de l'opération 3 dans le graphe de la figure 6.7 qui prend 8ut pour s'exécuter.

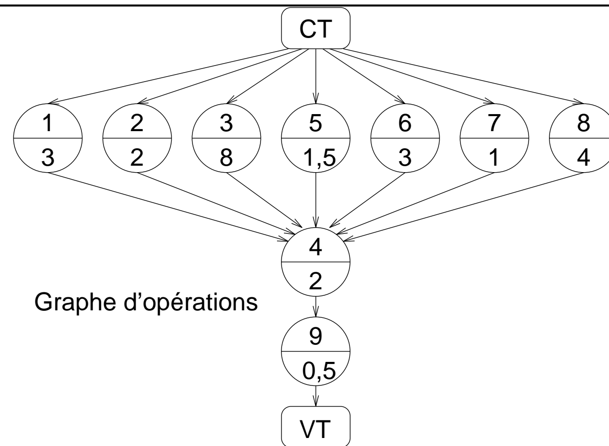


FIG. 6.7 - *Graphe d'une transaction avec temps d'exécution des opérations*

Le Diagramme de Gantt qui correspond au graphe de la figure 6.7 est montré dans la figure 6.8. Tb donne le temps d'exécution du bloc d'opérations parallèles. Le temps d'exécution de la transaction est calculé comme :

$$TP = Ti + Tb + Top_4 + Top_9 + Tv$$

Lorsque le temps d'exécution des opérations est variable, le temps d'exécution d'un bloc devient fortement dépendant de son opération la plus coûteuse. Cela est dû à la barrière de synchronisation présente à la fin de chaque bloc d'opérations parallèles. L'opération qui suit le bloc ne peut démarrer qu'après la terminaison de toutes les activités du bloc.

Dans le diagramme de la figure 6.8, l'opération 3 qui s'exécute en 8ut détermine la durée du bloc qui sera équivalente au temps d'exécution de cette opération ajouté du temps pour créer l'activité correspondante à 3. Les autres activités du bloc se terminent toutes *avant* l'activité qui exécute l'opération 3. A ce point l'activité principale démarre les opérations 4 et 9 de manière séquentielle et puis valide la transaction.

Comme les activités d'un bloc sont créées de manière séquentielle par l'activité principale, leur *ordre de création* met en cause le calcul du temps d'exécution du bloc. Par exemple, si l'activité qui exécute l'opération 3 était créée en dernier, le temps d'exécution Tb serait plus grand que celui donné par le diagramme de la figure 6.8.

Dans le cas le plus favorable, la création de l'activité la plus coûteuse se fait en premier et le temps d'exécution du bloc est calculé par :

$$Tb = C + max\{Topp\}$$

Par contre, si cette activité est créée en dernier, le temps d'exécution du bloc est le plus grand et sera donné par :

$$Tb = C * (n - 1) + max\{Topp\}$$

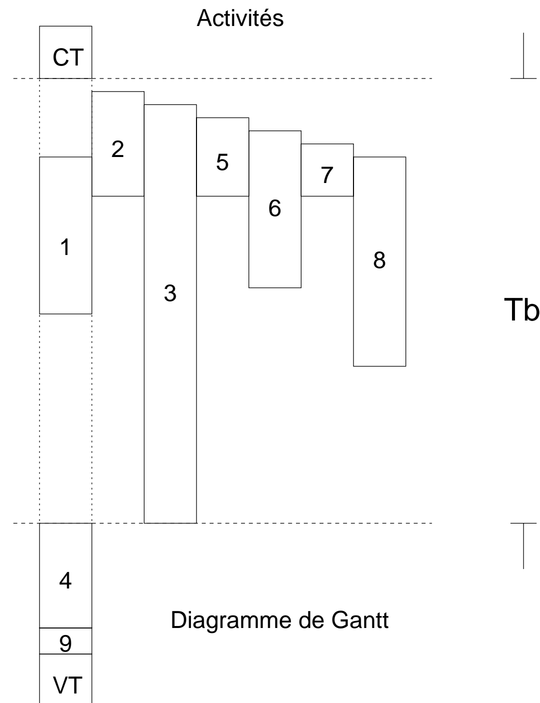


Fig. 6.8 - *Temps d'exécution pour les opérations*

Considérons pour le diagramme de la figure 6.8, le temps de création d'activité C comme étant 0,2ut. Dans le meilleur cas, Tb sera 8,2ut (0,2+8) et dans le cas le plus défavorable Tb sera 9,2ut (0,2*6+8). Si l'on considère maintenant un temps de création d'activité beaucoup plus grand (C = 1,5ut), Tb dans le meilleur cas se rapproche de 9,5ut (1,5+8) et dans le pire est d'environ 17ut (1,5*6+8). Nous pouvons noter que dans le cas où le temps de création d'activité est grand par rapport au temps d'exécution des opérations du bloc, l'ordre de création devient plus déterminant dans le calcul du temps du bloc.

Il faut noter que même dans le cas le plus défavorable, l'exécution parallèle des opérations du bloc est plus performante que l'exécution séquentielle équivalente. Cette exécution serait réalisée en 22,5ut, donc moins vite que le résultat obtenu par l'exécution parallèle la moins efficace.

Le diagramme de la figure 6.8 est caractérisé par des petites surcharges par rapport au temps d'exécution des opérations. Lorsque la surcharge du bloc est grande par rapport à ses opérations, le temps d'exécution du bloc peut ne pas être déterminé par le temps d'exécution de l'opération la plus longue. Même dans le cas le plus avantageux où cette opération est la première à s'exécuter dans le bloc.

Par exemple, le Diagramme de Gantt de la figure 6.9 montre le temps d'exécution pour le bloc d'opérations de la figure 6.7 lorsque la surcharge est 1ut. L'opération 3 est

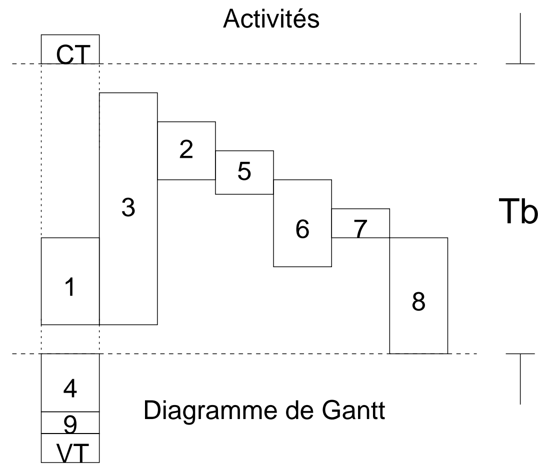


FIG. 6.9 - *Diagramme de Gantt pour un bloc ayant grande surcharge*

exécutée en premier, mais le temps d'exécution du bloc dépasse le temps d'exécution de cette opération. En effet, deux aspects retardent l'exécution du bloc :

- La dernière opération qui démarre, en l'occurrence l'opération 8, est la deuxième la plus coûteuse en terme de temps d'exécution des opérations du bloc. Cela fait que le temps pris pour créer l'activité en charge d'exécuter cette opération plus le temps d'exécution de l'opération elle-même est plus grand que le temps nécessaire à l'exécution de l'opération 3.

- L'exécution des opérations simples prend beaucoup de temps à cause de la relation surcharge/complexité de ces opérations et par conséquent repousse le déclenchement des opérations plus complexes.

A travers une réorganisation dans l'ordre de création des activités, le temps d'exécution d'un bloc peut être toujours proche du temps d'exécution de l'opération la plus coûteuse. Nous nous sommes donc confrontés au problème de *l'ordonnancement des opérations* du bloc [LER92]. Dans la suite, nous utilisons des Diagrammes de Gantt pour expliquer deux techniques différentes pour l'ordonnancement des opérations. Notre approche applique la deuxième technique au moment de la génération du code parallèle d'une transaction.

La première technique consiste à créer les opérations les plus coûteuses d'abord et en ordre décroissant de complexité [ACD74]. Les opérations les moins coûteuses sont alors démarrées en dernier. L'exécution de ces opérations aura lieu au moment de l'exécution des opérations les plus longues et, par conséquent, n'augmentera pas le temps d'exécution du bloc.

La figure 6.10 montre le Diagramme de Gantt du bloc d'opérations de la figure 6.7 avec un ordonnancement décroissant des opérations et la surcharge toujours à 1ut. On voit une tendance de concentration des exécutions parallèles au milieu du bloc d'opérations. Toutes les opérations sont exécutées en parallèle avec l'opération 3, qui en fait détermine le temps d'exécution du bloc.

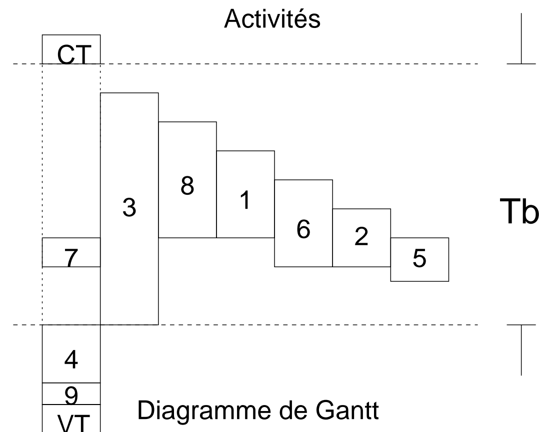


FIG. 6.10 - *La surcharge et l'ordonnancement décroissant des opérations*

La deuxième technique consiste à regrouper les opérations peu complexes dans une seule activité qui les exécutera de manière séquentielle [KL88]. Cette technique ne rend pas l'exécution du bloc plus efficace en terme de temps par rapport à la technique précédente, puisque le temps d'exécution d'un bloc ne peut être plus petit que le temps d'exécution de l'opération le plus long. Toutefois on obtient des résultats très proche voir équivalents en terme de temps d'exécution du bloc avec moins d'activités créées.

Le diagramme de la figure 6.11 montre le temps d'exécution pour le bloc de la figure 6.7 lorsque les opérations simples sont exécutées par une seule activité. L'ordre de création des activités dans le bloc est équivalent à l'ordre donné par le diagramme de la figure 6.8, sauf pour les opérations 2, 5 et 7 qui sont exécutées par l'activité principale.

Le temps d'exécution du bloc donné par le diagramme de la figure 6.11 est proche de celui donné par le diagramme de la figure 6.10. On perd 1ut qui correspond à la surcharge supplémentaire due à la création de l'activité pour exécuter l'opération 3 après la création de l'activité pour exécuter l'opération 1. Le temps d'exécution du bloc est de :

$$Tb = 1 * 2 + 3 \quad = \quad 5ut$$

Lorsqu'on compare Tb avec le temps de l'exécution séquentielle des mêmes opérations (22,5ut), on voit que l'exécution parallèle du bloc dans ces conditions est presque 5

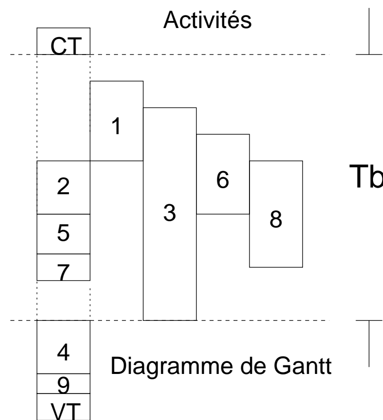


FIG. 6.11 - *Exemple de regroupement des opérations*

fois plus performante que l'exécution séquentielle.

Certes, il est souhaitable d'allouer des processeurs uniquement dans le but d'améliorer les performances. Toutefois, en ce qui concerne le temps d'exécution d'un bloc d'opérations parallèles, la création d'une activité pour les opérations ayant un faible temps d'exécution n'est pas l'aspect le plus important à considérer. Le diagramme de la figure 6.10 le montre. En effet, même si la surcharge de la création d'une activité est trop grande par rapport à la complexité de l'opération, l'exécution d'une telle opération dans une activité indépendante ne pénalise pas le temps d'exécution du bloc d'opérations. Cela à condition que l'opération la plus coûteuse démarre avant les opérations moins coûteuses.

D'une manière générale, les opérations les plus complexes des blocs définissent un *chemin critique* qui détermine le temps d'exécution de toute la transaction. Ainsi, nous pouvons gagner en performance par l'exécution le plutôt possible des opérations composantes de ce chemin.

Considèrons le graphe d'une transaction donné dans la figure 6.12. Sachant que les opérations les plus coûteuses de chaque bloc sont les opérations 3 et 8, le chemin critique de la transaction est composé des opérations CT, 2, 4, 8, 9 et VT. Le temps d'exécution de la transaction est montré dans le Diagramme de Gantt. Nous pouvons noter que le temps d'exécution de la transaction est très dépendant de l'exécution des opérations du chemin critique.

6.4 Evaluation des stratégies d'exécution

La parallélisation d'un ensemble d'opérations séquentielles peut suivre différentes stratégies qui déterminerons l'ordonnancement des opérations parallèles au moment

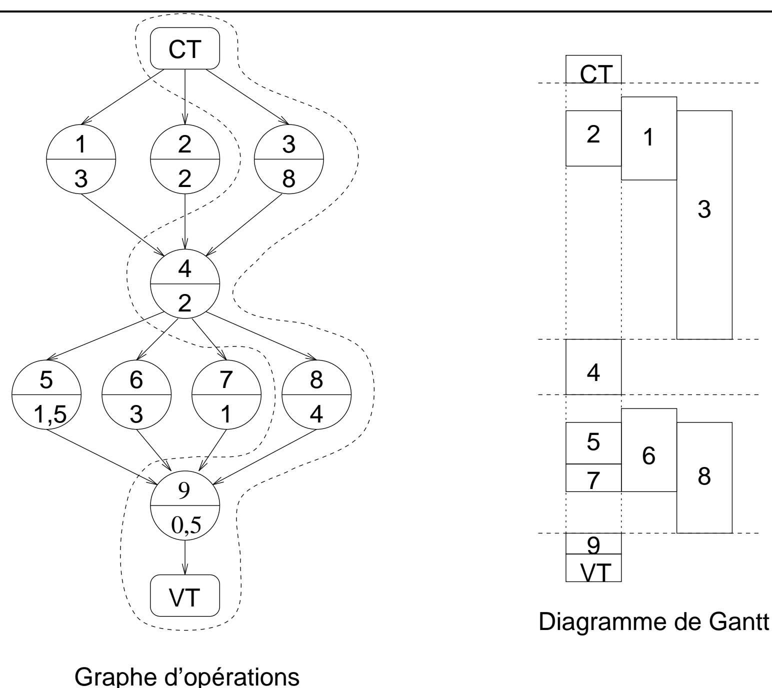


FIG. 6.12 - *Exemple de chemin critique*

de l'exécution. Ces stratégies sont donc utiles pour la transformation du code des transactions car des ordonnancements différents des opérations peuvent produit des résultats divers en terme de performance. Il s'agit d'identifier les aspects liés à la fois à la complexité du code des opérations et à l'ordonnancement des opérations d'un bloc.

Certains de ces aspects ont déjà été étudiés dans la section précédente vis à vis de la surcharge. Toutefois, dans cette section, nous cherchons à mieux comprendre le poids de chaque opération composante d'un bloc et ainsi pouvoir identifier le chemin critique d'une transaction. Nous fournissons ensuite des heuristiques qui l'on peut utiliser pour abréger le temps d'exécution des opérations du chemin critique. Comme résultat, le temps global d'exécution de la transaction est également abrégé.

6.4.1 Complexité des opérations

D'une manière générale, le temps d'exécution d'une opération ne peut être mesuré qu'*a posteriori*, c'est à dire, après l'observation de l'exécution de l'opération. Il est évident que le temps d'exécution des opérations simples, comme une mise à jour d'une variable ou une opération de multiplication de deux entiers, peut être connu étant donné une architecture matériel particulière. Cependant les opérations typées, telles que celles que nous considérons dans notre approche, sont complexes et, par

conséquent possèdent un temps d'exécution difficile, voir impossible, à déterminer de manière statique. La prise en compte des paramètres qui n'auront des valeurs qu'à l'exécution rend impossible le calcul au moment de la compilation des temps d'exécution des opérations.

La complexité des opérations peut être plus ou moins déterminée par une analyse détaillée de leur code. Pour cela, nous utilisons des *heuristiques* qui permettent de faire des *estimations* sur la complexité d'une opération [AG94]. Ainsi nous pouvons attribuer des poids qui représente le temps d'exécution d'une opération. Les poids sont utiles à l'ordonnancement des opérations d'un bloc à la fois pour définir l'ordre de création des activités et pour mettre en œuvre la technique de regroupement que nous avons expliqué dans la section 6.3.2.2.

- Premièrement, la présence, dans le code d'une opération, d'une *requête SQL* augmente de manière importante le temps d'exécution de l'opération et, par conséquent, son poids d'exécution. Sans prendre en compte la complexité de la requête ni les techniques mises en œuvre pour optimiser son exécution, l'exécution d'une opération ayant une requête comprend, en général, des parcours des collections d'objets à la fois persistants et/ou temporaires. Il s'agit donc d'opérations très coûteuses en terme de temps d'exécution.

- Le poids d'une opération est proportionnel à la taille du *graphe de flot des messages* dont l'opération est la racine. Si l'opération appelle d'autres opérations et ainsi de suite, le coût des envois de messages entre les objets fait que l'opération est coûteuse en terme de temps d'exécution. Cela est particulièrement évident lorsqu'on considère que certains objets qui reçoivent des messages doivent être récupérés sur mémoire secondaires avant d'effectivement exécuter la méthode correspondante au message. En revanche, une opération qui n'appelle pas d'autres opérations reçoit un petit poids en ce qui concerne la taille de son graphe de flot des messages

- Les *boucles* sont également coûteuses, surtout lorsque le nombre d'itérations est grand. Ce nombre n'est peut être pas connu au moment de la compilation. Toutefois, le fait qu'une boucle soit présente dans le code de l'opération entraîne déjà un poids d'exécution important.

- Enfin on prend en compte le *nombre d'instructions* qui forment le corps de l'opération. En fait, ce nombre ne signifie pas grand chose sauf s'il est très grand ou très petit. Dans le premier cas, il y a une forte chance que l'opération soit coûteuse, et par conséquent nous l'attribuons un grand poids d'exécution. Si ce

nombre est petit, il s'agit d'une opération simple pour laquelle l'ordonnancement ne créera pas d'activité. Nous détaillons à ces aspects dans la section suivante.

Il existe différentes possibilités de combiner les heuristiques citées ci-dessus. Par exemple, une opération qui est la racine d'un graphe de flot de messages dont une opération effectue une requête SQL doit être affectée d'un poids d'exécution au moins équivalent à la combinaison des deux critères d'estimation de la complexité du code. La combinaison des plusieurs de ces critères peut rendre complexe l'algorithme de calcul des poids des opérations.

Le chemin critique d'une transaction est formé par les opérations séquentielles et par les opérations ayant les plus grand poids d'exécution dans les blocs parallèles. L'efficacité d'un ordonnancement des opérations d'un bloc dépend donc de la fiabilité du calcul des poids des opérations. Il faut noter que l'efficacité visée lors de l'utilisation des heuristiques présentées ci-dessus est résultat d'une optimisation de l'ordonnancement des opérations d'un bloc. Nous avons montré dans la section précédente que dans beaucoup de cas où le coût de la surcharge n'est pas trop grand par rapport à la complexité des opérations, l'ordonnancement parallèle des opérations est plus efficace que l'ordonnancement séquentiel. L'utilisation des heuristiques peut apporter plus de gains en termes de performance.

6.4.2 Ordonnancement des opérations

L'utilisation des poids des opérations permet d'identifier toutes les opérations que forment le chemin critique d'une transaction. Nous pouvons, à l'aide de différentes formes de structurer l'exécution des activités parallèles des blocs, obtenir plus d'efficacité dans l'exécution des opérations du chemin critique. Cependant cette structuration des activités doit respecter d'une part les contraintes posées par la compatibilité des opérations et d'autre part la structure tâches séquentielles-parallèles du graphe de la transaction. Ainsi nous pouvons assurer les propriétés de la parallélisation des transaction expliquées dans le Chapitre 3.

Dans notre approche, l'ordonnancement des opérations utilise l'heuristique qui consiste à vérifier si l'opération est petite en nombre d'instructions. Dans ce cas, l'algorithme de génération du code parallèle applique la technique de regroupement de sorte que ces opérations sont toutes exécutées par l'activité principale en parallèle avec les autres activités du bloc.

Nous avons présenté un exemple de regroupement des opérations dans la section 6.3.2.2 qui montrait la réduction de la surcharge. L'exemple a également montré l'efficacité de cette technique lorsqu'on dispose de peu de processeurs, ce que limite le degré de parallélisme dans les blocs. La figure 6.13 montre deux Diagrammes de

Gantt différents pour un bloc d'opérations. Les activités dans le diagramme A correspondent chacune à une opération, tandis que dans le diagramme B nous avons regroupé les petites opérations dans l'activité principale.

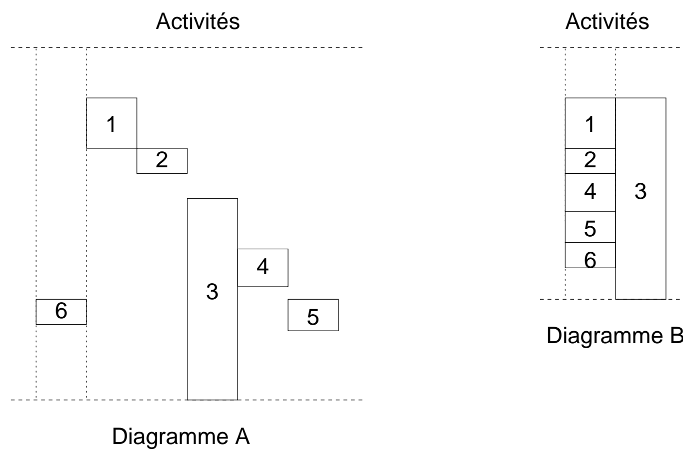


FIG. 6.13 - *Le regroupement des opérations*

Nous mettons en évidence deux aspects qui résultent de l'utilisation de la technique de regroupement :

- Nous obtenons des bonnes performances même dans des blocs ayant plusieurs opérations avec faible poids d'exécution. Il suffit qu'une des opérations du bloc soit complexe (grand poids d'exécution) pour que les gains soient significatifs. Cela n'est probablement pas vrai si l'on doit créer une activité pour chacune des opérations peu complexes du bloc. L'exécution séquentielle du bloc représentée dans la figure 6.13 aura un temps très proche de celui produit par une exécution parallèle selon le diagramme A. En revanche, le diagramme B montre clairement des gains en performance lorsque les opérations peu complexes sont exécutées par l'activité principale.
- Si l'on considère la notion d'accéleration ("speedup" en anglais), qui prend en compte le nombre de processeurs utilisé dans l'exécution parallèle [LER92], le regroupement rend l'exécution des opérations d'un bloc encore plus efficace. Cela est dû à des bons résultats de performance avec peu d'activités créées. Dans le diagramme B de la figure 6.13 le temps d'exécution du bloc est proche de celui nécessaire à l'exécution de l'opération la plus longue, et ceci avec uniquement deux activités créées.

Nous avons identifié deux autres techniques d'ordonnancement que nous n'utilisons pas pour l'instant dans notre approche mais qui serait envisageables de les utiliser

dans des avancements de notre travail. La première consiste à exécuter le plutôt possible dans les blocs l'opération la plus longue. Nous avons expliqué cette technique dans la section 6.3.2.2 vis à vis de la surcharge résultante de la gestion du parallélisme dans les blocs. Ici nous voudrions mettre en évidence l'apport de l'utilisation de cette technique avec celle du regroupement.

La figure 6.14 montre deux diagrammes sur lesquels nous avons appliqué le regroupement des opérations peu complexes. Le fait que les opérations 1 et 2, qui viennent avant l'opération 3, ne soient pas regroupées augmente la surcharge et retarde l'exécution de l'opération la plus longue dans le diagramme A. En revanche, dans le diagramme B, les opérations sont ordonnancées en ordre décroissante, éliminant la surcharge supplémentaire produite par le déclenchement de 1 et 2 avant 3. Il faut noter que dans tous les cas de figure l'exécution parallèle est plus performante que l'exécution séquentielle.

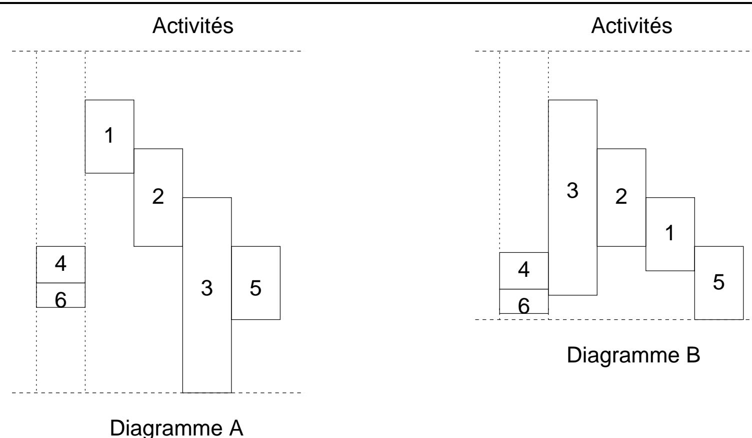


FIG. 6.14 - *Ordonnancement des activités et le regroupement*

La dernière technique d'ordonnancement que nous envisageons appliquer consiste à créer un "pool" d'activités au début de la transaction et à utiliser ces activités lors de l'exécution des blocs d'opérations. L'idée est en fait une adaptation de l'approche client-serveur avec la création dynamique des processus où le serveur est composé d'un "pool" de processus (voir la section "Parallélisme asynchrone" au Chapitre 2 ou [BST89] pour plus de détails). Dans ce cas, le déclenchement d'une opération est semblable à un *appel de procédure à distance* dans lequel le processus serveur est déjà créé.

Le nombre d'activité à créer au début de la transaction correspond au degré maximum de parallélisme atteint par la transaction. Autrement dit, le bloc le plus grand en terme de nombre d'activités parallèles détermine le nombre d'activités qui sont créées au début de la transaction. Noter que le création des activités est orthogonale

au poids des opérations, puisque à priori une activité exécute différentes opérations dans des blocs différents.

Considérons le graphe de transaction et son Diagramme de Gantt montrés dans la figure 6.15. La transaction démarre avec la création des activités qui vont être utilisées dans les blocs d'opérations. Le temps de démarrage de la transaction devient plus important car il entraîne la création des activités. Cependant, comme on peut noter dans le diagramme, la surcharge dans les blocs est plus petite, ce qui diminue le temps d'exécution de chaque bloc.

CT
b1
b2
VT
Graphe d'opérations
Activités
Ti
b1
b2
Tv
Diagramme de Gantt

FIG. 6.15 - *Ordonnancement avec un "pool" d'activités*

Dans le cas où la transaction possède plus d'un bloc, cette technique d'ordonnancement apporte plus de gain en performance en ce qui concerne le temps d'exécution de la transaction. Toutefois il s'agit d'une approche qui présente l'inconvénient de demander de ressource matériel — par la création des activités dans des processeurs — dont on n'aura besoin qu'ultérieurement. On risque ainsi de pénaliser d'autres transactions concurrentes ou d'autres applications qui peuvent être en attente de ressources matériel.

Dans des systèmes où l'application base de données est la seule à s'exécuter ou dans le cas où l'on dispose des ressources matériel potentiellement infinie, la création des activités au début de la transaction produit sans doute des gains en performance par rapport à l'approche création-mort des activités au sein des blocs d'opérations.

6.5 Les performances et les transactions emboîtées avec parallélisme

Jusqu'à présent, l'étude de performance s'applique surtout au modèle de parallélisation des transactions expliqué au Chapitre 3. Sachant que ce modèle est utilisé dans la parallélisation des applications lorsqu'on dispose d'un modèle de transactions emboîtées, les résultats de cette étude sont également valables pour le modèle de parallélisation des applications présenté au Chapitre 5.

Dans cette section, nous détaillons les aspects liés aux performances en prenant en compte la structuration hiérarchique des transactions emboîtées. Nous présentons tout d'abord les composants du calcul du temps d'exécution des transactions d'un arbre de décomposition avec parallélisme (voir section 5.3 pour plus de détails), puis nous montrons les gains en performances apportés par la parallélisation de ces transactions.

6.5.1 Temps d'exécution des transactions

Le temps d'exécution d'une transaction d'un arbre de décomposition dépend directement du modèle d'exécution qui met en œuvre le modèle de transactions emboîtées. Dans notre approche, le modèle d'exécution fourni du parallélisme horizontal aussi bien que du parallélisme vertical entre les transactions. De plus, les opérations d'une transaction peuvent être exécutées en parallèle dans des blocs d'opérations.

La figure 6.16 présente le graphe d'une transaction emboîtée, où nous représentons à la fois les opérations et les transactions filles (hexagones) de la transaction emboîtée. Chaque nœud d'une transaction fille représente en effet un nouveau graphe d'une transaction emboîtée pouvant avoir d'autres transactions et d'opérations parallèles. Au niveau des feuilles d'un arbre de décomposition, on retrouve les transactions parallèles dont nous connaissons déjà la formule que donne le temps d'exécution.

Dans les transactions emboîtées, les blocs parallèles peuvent contenir à la fois des opérations et des transactions, nommés blocs opérations/transactions (blocs OT). Dans le graphe de la figure 6.16, le bloc b1 possède des opérations et des transactions tandis que le bloc b2 ne possède que des transactions.

Le temps d'exécution d'un bloc OT est fonction du type du bloc :

- Pour les blocs ayant uniquement des opérations, on revient au calcul du temps d'exécution d'un bloc d'opérations parallèles.

- Pour les blocs ayant uniquement des transactions, le temps du bloc est déterminé par le temps d'exécution de chacune de ses transactions. En général,

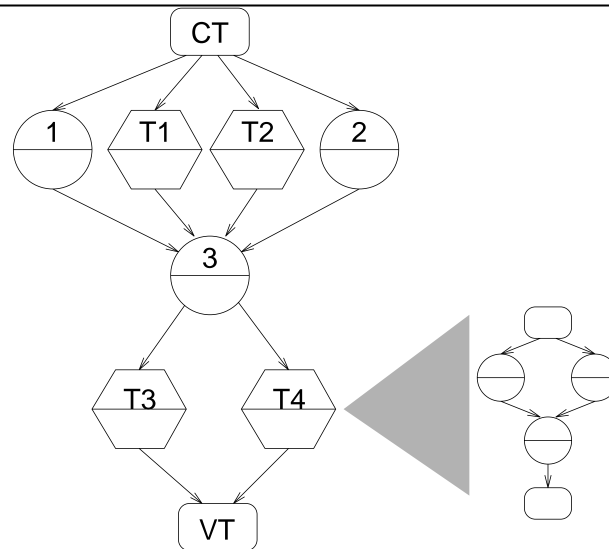


FIG. 6.16 - *Le graphe d'une transaction emboîtée*

la transaction présentant le temps d'exécution le plus grand définira le temps d'exécution du bloc ($max\{TEP\}$)[1].

- Pour les blocs ayant des opérations et des transactions, le temps d'exécution sera déterminé soit par $max\{Topp\}$ ou $max\{TEP\}$. Dans la plupart des cas, $max\{Topp\}$ est plus petit que $max\{TEP\}$, par conséquent celui-ci définira le temps d'exécution du bloc OT.

Pour une transaction emboîtée ayant n opérations séquentielles et m blocs OT, son temps d'exécution TEP est donné par :

$$TEP = Ti + \sum_{i=1}^{n} Tops_i + \sum_{j=1}^{m} Tot_j + Tv$$

où Tot_j est le temps d'exécution d'un bloc OT.

Nous pouvons noter que la formule donnant le temps d'exécution d'une transaction emboîtée est très semblable à celle qui calcule le temps d'exécution d'une transaction parallèle. La différence apparaît dans le calcul du temps d'exécution des blocs parallèles qui peut être cette fois-ci très dépendant du temps d'exécution des transactions composantes du bloc.

6.5.2 Gains en performance

Le calcul des gains en performance de l'exécution d'un arbre de décomposition doit tenir compte du gain apporté par la parallélisation de chaque transaction présente

1. TEP est un acronyme pour Temps d'une transaction Emboîtée avec Parallélisme.

dans l'arbre. Les gains dans l'exécution des transactions feuilles rentrent dans le cadre des transactions parallèles et ainsi ont été détaillés dans la section 6.3. Les gains en performance des feuilles sont répercutés vers leurs ancêtres. Cela produit un effet en cascade de sorte que l'exécution de tout l'arbre est plus efficace que l'exécution d'un arbre équivalent sans parallélisme.

En effet, les gains en performance de l'exécution d'une transaction emboîtée avec parallélisation par rapport à l'exécution d'une transaction emboîtée sans parallélisation sont produits par :

- l'exécution des opérations parallèles au sein d'une transaction de l'arbre de décomposition ;
- les gains apportés par l'exécution parallèle des transactions appartenant au sous-arbre d'une transaction.

La figure 6.17 montre le code d'une application utilisée dans le Chapitre 5 et son graphe correspondant. L'application possède deux transactions T1 et T2, dont T2 est fille de T1.

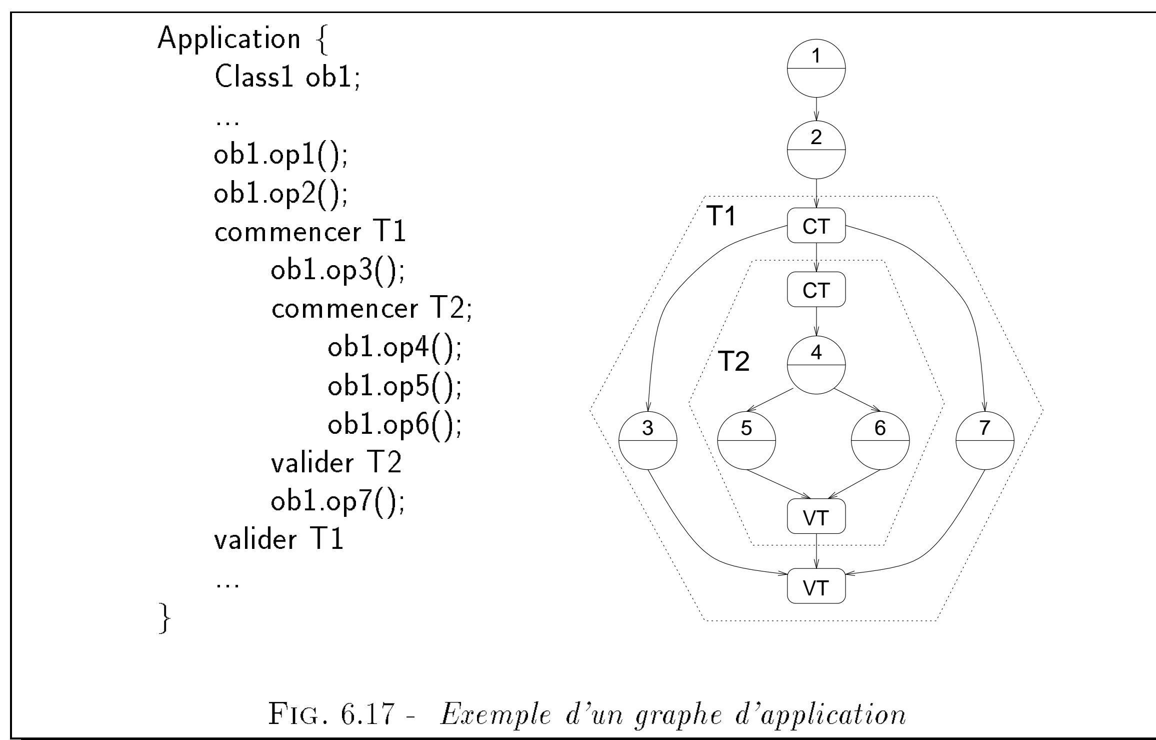


Fig. 6.17 - *Exemple d'un graphe d'application*

La transaction T1 exécute l'opération 3, crée la transaction T2 et puis exécute l'opération 7. Ces deux opérations et la transaction T2 forment un bloc OT, comme montre le graphe de la figure 6.17. T2 exécute l'opération 4 et puis crée un bloc OT composé des opérations 5 et 6.

La figure 6.18 présente deux Diagrammes de Gant qui représentent l'exécution de l'application de la figure 6.17. Le diagramme A montre l'exécution dans le cas où il n'y pas de parallélisation des transactions tandis que le diagramme B tient compte de la parallélisation de T1 et de T2.

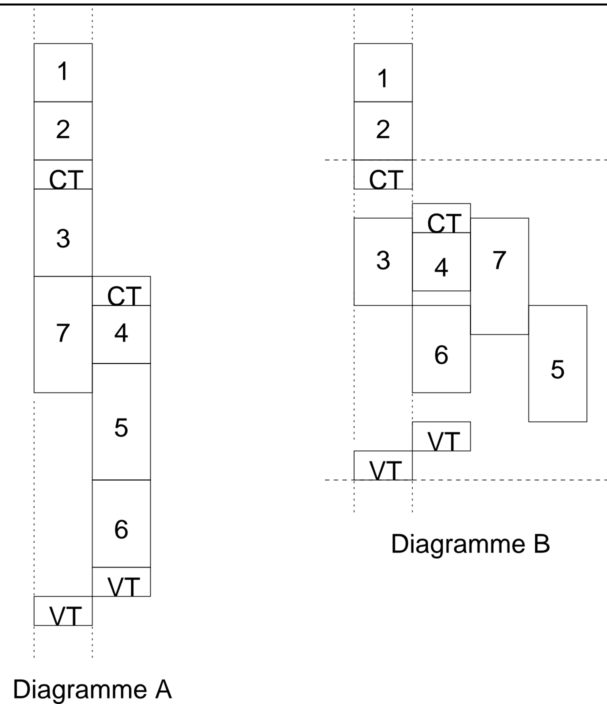


Fig. 6.18 - *Diagrammes de Gantt d'une application avec transactions emboîtées*

Dans le diagramme A, T1 exécute 3, crée la transaction T2 puis exécute l'opération 7 en parallèle avec T2. Cela est possible car nous considérons un modèle de transactions emboîtées fournissant du parallélisme vertical indépendamment de la parallélisation des transactions (voir Chapitre 5 pour plus de détails). Toutes les opérations de T2 sont exécutées de manière séquentielle, après quoi T2 passe à la validation et puis T1.

En revanche dans le diagramme B, les opérations de T1 et de T2 sont exécutées en parallèle. Il faut noter que le fait d'exécuter 5 et 6 en parallèle améliore davantage le temps d'exécution de T2, et par conséquent le temps d'exécution de sa transaction mère T1. De plus, les opérations 3 et 7 sont également exécutées en parallèle pouvant donc rendre plus efficace l'exécution de T1.

L'observation détaillée du diagramme B montre deux aspects importants à considérer :

- Le temps d'exécution d'une transaction emboîtée T est très sensible au temps d'exécution des transactions appartenant au sous-arbre dont T est la racine.

Cela est dû principalement à la sémantique du modèle de transactions emboîtées qui détermine la dépendance de validation des transactions mères par rapport aux transactions filles. Par exemple, T1 valide après la validation de T2 dans les deux diagrammes de la figure 6.18.

- Le fait de lancer des opérations de manière asynchrone permet le déclenchement le plutôt possible des transactions filles. Cela améliore l'exécution d'une transaction emboîtée car ses transactions filles peuvent donc valider aussi plutôt.

6.6 Conclusion

Dans ce chapitre nous avons présenté une étude théorique sur les gains en performance dans l'exécution des transactions apportés par notre modèle de parallélisation. Cette étude est une première démarche pour la compréhension des aspects liés à la structuration des graphes des transactions et à l'évaluation du coût provenant de la gestion du parallélisme. En général, une étude de ce type est nécessaire à la prise de mesures de performances dans le cadre d'une implantation de notre modèle sur une machine parallèle.

Nous avons montré, par l'intermédiaire des diagrammes présentant le comportement dynamique des transactions, que sous certaines conditions de surcharge et caractéristiques des opérations, notre approche apporte des gains significatifs en performances. Ces conditions sont fonction à la fois de l'architecture matérielle sur laquelle on peut exécuter les transactions parallèles et de l'application dont dépend le degré de compatibilité des opérations.

A travers des analyses nous avons déterminé des limites de surcharge vis à vis des blocs d'opérations qui l'on doit respecter pour obtenir des gains en performance. Ces limites sont en fait déterminés tout d'abord par la relation entre le coût du parallélisme et la structuration du graphe transactionnel et puis par la relation entre ce coût et la complexité des opérations :

- La structure du graphe d'une transaction généré par nos transformations donne le nombre de blocs d'opérations parallèles et le nombre d'opérations dans chacun de ces blocs. Nous avons montré qu'on gagne beaucoup lorsque la transaction est entièrement parallélisée et sont graphe consiste d'un seule bloc d'opérations et qu'on a rien a gagner si le graphe ne présente aucun bloc. Dans le cas d'une parallélisation partielle, nous avons montré qu'il vaut mieux avoir moins des blocs avec plus d'opérations que avoir plusieurs blocs ayant peu d'opérations. Cela est particulièrement vrai car le coût de la surcharge devient de plus en plus important au fur et à mesure que le nombre d'opérations d'un bloc diminue.

- Nous avons également constaté que le coût de la création d'une activité par rapport à l'opération qu'elle doit exécuter est fonction de la complexité de cette opération. Nous avons montré que dans le cas où l'ordre de création des activités suit l'ordre décroissant de la complexité des opérations, la création d'une activité pour une opération simple a une forte chance de ne pas mettre en cause le temps d'exécution d'un bloc d'opérations. En revanche, si l'on ne tient pas compte de l'ordre de création des activités, l'utilisation de la technique de regroupement des opérations peu complexes produit des temps d'exécution proche de celui déterminé par le chemin critique de la transaction, tout en utilisant peu de ressources matérielles.

Cette étude s'est intéressé également au problème de l'ordonnancement des opérations parallèles dans un graphe transactionnel. Nous avons montré que l'utilisation d'heuristiques pour déterminer le poids d'exécution des opérations et la prise en compte de ces poids dans l'ordonnancement, peut optimiser l'exécution parallèle des opérations d'un bloc et, par conséquent, améliorer davantage le temps d'exécution des transactions.

Nous avons décrit certaines techniques d'optimisation de l'ordonnancement des opérations parallèles dont une est présente dans notre approche. Cela a permis d'une part évaluer la technique utilisée en ce qui concerne l'apport en terme d'ordonnancement des opérations d'un bloc et d'autre part de montrer d'autres techniques qui peuvent être utilisées.

Nous avons considérer les performances des applications lorsqu'on utilise les techniques de parallélisation dans le cadre d'un modèle de transactions emboîtées. Dans ce cas, les gains en performance sont dûs à l'exécution parallèles des opérations dans les transactions emboîtées et, plus important, à l'effet en cascade des gains en performances des transactions filles vers leurs ancêtres d'une arbre de décomposition.

Il est évident que la prise de mesures dans le cadre des simulations de notre approche sur une machine parallèle pourrait fournir à la fois des nouvelles variables dans la définition de la surcharge et des confirmations pratiques de cette étude. En attendant le portage de notre prototype d'expérimentation sur une telle machine, nous espérons que cette étude ait élucidé certains aspects liés au gains de performance apportés par notre approche.

Chapitre 7

Conclusion

7.1 Les résultats

Dans cette thèse, nous nous sommes intéressé à l'exploitation du parallélisme dans le cadre des systèmes de bases de données à objets. Etant donnés les nombreux travaux sur le traitement parallèle de requêtes, nous avons considéré d'autres niveaux d'abstraction, notamment le parallélisme dans les transactions et dans les applications. Notre travail est donc complémentaire à celui sur la parallélisation des requêtes.

Nous avons tout d'abord fait une étude des différentes formes d'expression du parallélisme dans les langages de programmation procéduraux et à objets. Cette étude a mis en évidence les niveaux d'abstraction dans lequelles le parallélisme est exploité et les techniques de synchronisation mises en œuvre dans ces langages. La notion d'objet, en tant qu'unité d'encapsulation, s'impose comme le point de référence. Ainsi, nous avons analysé le parallélisme *entre* les objets aussi bien que celui *dans* les objets, qui se traduit par l'exécution parallèle de méthodes.

Le parallélisme dans un objet demande la synchronisation de ses méthodes. Cette idée n'est pas nouvelle et peut être comparable à l'exclusion mutuelle des procédures d'un moniteur. Cependant, l'abstraction de données permet de définir des relations sémantiques entre les méthodes afin d'obtenir plus de parallélisme que celui déterminé par l'exclusion mutuelle. Cette hypothèse est la base pour de nombreux travaux dans les domaines des langages à objets, systèmes répartis et systèmes de bases de données à objets. Toutefois, les techniques de spécification de ces relations et de mise en œuvre du parallélisme ne sont pas les mêmes.

Nous avons étudié l'exploitation du parallélisme par transformation de code dans le cadre de la parallélisation des programmes séquentiels et dans le traitement des requêtes. Dans les deux cas, les transformations sont fondées sur des propriétés du code et des données manipulées. Pour les programmes séquentiels, on parallélise les

itérations ou l'exécution des procédures selon certaines relations de dépendances de données. Pour les requêtes on parallélise des opérations, dont on connaît à priori le comportement, sur des relations (au sens des bases de données relationnelles).

Cette thèse associe les développements récents en matière de conflit entre opérations dans le cadre des types abstraits de données avec des techniques de parallélisation par transformation. Le code à transformer est celui d'une transaction composée d'opérations. Nous considérons d'abord un modèle de transactions classiques. Deux opérations compatibles peuvent être parallélisées. Les transformations génèrent du code qui produit l'exécution asynchrone des opérations sans introduit des problèmes liés aux non déterminisme. Nos techniques de création et de synchronisation des activités assurent d'une part l'isolation des activités et d'autre part l'existence d'une seule activité au début et à la fin d'une transaction.

Nous avons ensuite considéré un modèle de transactions emboîtées. Une étude de ce modèle a montré la difficulté à gérer à la fois des transactions et des activités parallèles. En intégrant notre approche pour la parallélisation et un modèle de transactions emboîtées multi-activités, nous exploitons le parallélisme dans une application sous deux formes différentes : d'abord explicitement par la création des sous-transactions et ensuite implicitement par l'exécution parallèle des opérations. Le modèle assure l'isolation à la fois des (sous-)transactions et des activités. Le parallélisme des opérations peut être introduit à n'importe quel niveau d'une transaction emboîtée.

Nous avons développé un prototype qui met en œuvre la parallélisation des transactions à l'aide du système O_2. Le prototype calcule la compatibilité des méthodes d'un schéma et construit des plans d'exécution pour les transactions d'une application qui utilise ces méthodes. Nous avons pris en compte la portabilité du prototype dans la mesure où l'architecture logique du prototype permet de l'isoler du système O_2.

Le modèle a été appliqué dans le cadre du système de règles NAOS. Nous avons utilisé la relation de compatibilité entre les méthodes pour définir une relation de compatibilité entre les règles. Notre approche construit des plans d'exécution pour les règles d'un cycle en prenant en compte la compatibilité et la priorité entre les règles. Les règles compatibles sont ordonnées pour une exécution parallèle, tandis que les autres règles sont exécutées de manière séquentielle en suivant leur priorité.

Nous avons effectué une étude de performances où nous décrivons les aspects qui déterminent les gains apportés par la parallélisation des transactions. Ainsi nous avons analysé nos choix concernant la structure des graphes des transactions parallèles et la surcharge résultant de la gestion du parallélisme pendant l'exécution de ces transactions. Nous prenons en compte le problème de l'ordonnancement dans les blocs d'opérations parallèles et considérons les les gains en performance dans le cadre des transactions emboîtées. L'étude a montré que, sous certaines conditions de surcharge

et caractéristiques des opérations, notre approche apporte des gains significatifs en performances.

Ensuite nous nous sommes intéressés au problème de l'ordonnancement dans les blocs d'opérations parallèles. Finalement, nous avons présenté les gains en performance lorsqu'on prend en compte la parallélisation des applications dans le cadre des transactions emboîtées.

7.2 Les perspectives

Il est évident que le parallélisme sera de plus en plus nécessaire aux futurs systèmes de bases de données à objets. Le travail de cette thèse n'est qu'un point de départ pour d'autres formes d'exploitation du parallélisme notamment dans des contextes autres que celui des requêtes. Cependant nous pensons qu'il s'agit d'un travail original étant donné le peu de travaux similaires.

Les perspectives sont vastes et concernent principalement (1) le passage d'une approche statique du contrôle de partage de données vers une approche dynamique ; (2) la mesure du gain en performance produit par notre modèle ; (3) l'implantation du modèle sur des machines multiprocesseurs où le vrai parallélisme peut être exploité.

D'autres formes de contrôle de partage de données entre les activités sont envisagées, notamment des approches dynamiques. Nous avons vu au le chapitre 5, que l'isolation des transactions par rapport aux activités est difficilement assurée, principalement lorsqu'on utilise des techniques dynamiques différentes pour isoler séparément les transactions et les activités. Une étude sur les relations entre les techniques dynamiques serait à faire.

Compte tenu du type d'application cible, le contrôle de partage doit être effectué par le système de manière implicite et automatique. Nous pensons que les techniques actuelles de contrôle de partage de données sont plus appropriées au niveau système et ne doivent pas être disponibles au niveau application. Cet aspect doit être pris en compte dans une étude sur les techniques dynamiques de contrôle de partage.

Par ailleurs, l'approche statique présente peu de surcharge en temps d'exécution. Cette surcharge serait beaucoup plus importante lorsqu'on utilise un approche dynamique. Cela nous amène au deuxième point des perspectives que consiste à définir un modèle de coût qui puisse montrer les gains en performance apporter par les activités parallèles. On pourrait donc mieux évaluer la surcharge de la gestion du parallélisme dans les transactions qui utilise à la fois un contrôle de partage statique et dynamique.

Finalement, nous envisageons de produire des nouvelles versions de notre prototype, notamment sur des prototypes de recherches de SGBD à objets sur des "clusters" Unix. Nous envisageons donc dans un premier temps de valider notre approche sur

une machine parallèle. Nous pourons ainsi évoluer notre modèle pour l'utiliser dans d'autres contextes, notamment celui des gestionnaires d'objets parallèles [ODV94]. L'objectif consiste à produire un système liant l'exploitation du parallélisme par transformation avec les systèmes à objets répartis tout en assurant une interface de programmation simple et convenable au programmeur d'applications bases de données.

Bibliographie

[ABD+92] Atkinson (M.), Bancilhon (F.), DeWitt (D.), Dittrich (K.), Maier (D.) et Zdonik (S.). – The object-oriented database system manisfesto. *In : Building an Object-Oriented Database System - The Story of O2*, éd. par Bancilhon (F.), Delobel (C.) et Kanellakis (P.), chap. 1, pp. 3–20. – Morgan Kaufmann, 1992.

[ABLN85] Almes (G.T.), Black (A.P.), Lazowska (E.D.) et Noe (J.D.). – The Eden system: A technical review. *IEEE Transactions on Software Engineering*, vol. 11, n° 1, janvier 1985, pp. 43–59.

[AC93] Adiba (M.) et Collet (C.). – *Objets et Bases de Donneés : Le SGBD O_2*. – Hermès, 1993.

[ACD74] Adam (T.), Chandy (K.) et Dickson (J.A.). – Comparison of list schedulers for parallel processing systems. *Communications of the ACM*, vol. 17, décembre 1974, pp. 685–690.

[AE92a] Agrawal (D.) et El Abbadi (A.). – A non-resctrictive concurrency control for object-oriented databases. *In : Proc. of the 3rd Int. Conf. on Extending Data Base Technology*. pp. 469–482. – Vienna, mars 1992.

[AE92b] Agrawal (D.) et El Abbadi (A.). – Transaction management in database systems. *In : Database Transactions Models for Advanced Applications*, éd. par Elmagarmid (A.K.), chap. 1, pp. 1–31. – Morgan Kaufmann, 1992.

[AG94] Almasi (G.S.) et Gottlieb (A.). – *Highly Parallel Computing*. – Redwood city, Ca, Benjamin Cummings, 1994, 2 édition.

[Agh86] Agha (G.). – *Actors: A Model of Concurrent Computation in Distributed Systems*. – The Mit Press, 1986.

[AK87] Allen (J.R.) et Kennedy (K.). – Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, vol. 9, n° 4, octobre 1987, pp. 3–43.

[AKPW83] Allen (J.R.), Kennedy (K.), Porterfield (C.) et Warren (J.). – Conversion of control dependence to data dependence. *In : Proc. of the 10th Annual ACM Symposium on Principles of Programming Languages*, pp. 177–189. – Austin, janvier 1983. ACM Press.

[Ame87] America (P.). – POOL-T: A parallel object-oriented language. *In : Object-Oriented Concurrent Programming*, éd. par Yonezawa (A.) et Tokoro (M.), pp. 199–220. – The Mit Press, 1987.

[And91a] Andrews (G.R.). – *Concurrent Programming: Principles and Practice.* – Benjamin Cummings, 1991.

[And91b] Andrews (G.R.). – Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, vol. 23, n° 1, mars 1991, pp. 49–90.

[AS83] Andrews (G.R.) et Schneider (F.B.). – Concepts and notations for concurrent programming. *ACM Computing Surveys*, vol. 15, n° 1, mars 1983, pp. 3–43.

[AT88] Aksit (M.) et Tripathi (A.). – Data abstraction mechanism in SINA/st. *In : Proc. of the 1988 Object-Oriented Programming Systems Languages and Applications*, pp. 267–275. – San Diego, septembre 1988.

[BAC+90] Boral (H.), Alexander (W.), Clay (L.), Copeland (G.), Danforth (S.), Franklin (M.), Hart (B.), Smith (M.) et Valduriez (P.). – Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, n° 1, mars 1990, pp. 4–24.

[Bad89] Badrinath (B.R.). – *Concurrency Control in Complex Information Systems: A Semantics-Based Approach.* – Amherst, Thèse de PhD, University of Massachusetts, septembre 1989.

[Ban91] Banatre (J-P.). – Modèles et langages pour le parallèlisme. *In : Construction des Systèmes d'Exploitation Répartis*, éd. par Balter (R.), Banatre (J-P.) et Krakowiak (S.), chap. 3. – INRIA, 1991.

[Bar78] Barth (J.M.). – A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, vol. 21, n° 9, septembre 1978, pp. 724–736.

[BBDW83] Bitton (D.), Boral (H.), DeWitt (D.J.) et Wilkinson (W.K.). – Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, vol. 8, n° 3, septembre 1983, pp. 342–353.

[BCL93] Bergsten (B.), Couprie (M.) et Lopez (M.). – DBS3: A parallel database system for shared store. *In : Proc. of the 2nd Int. Conf. on Parallel and Distributed Information System.* pp. 260–262. – San Diego, janvier 1993.

[BDK92] Bancilhon (F.), Delobel (C.) et Kanellakis (P.) (édité par). – *Building an Object-Oriented Database System - The Story of O2.* – Morgan Kaufmann, 1992.

[BGS93] Bacon (D.F.), Graham (S.L.) et Sharp (O.J.). – *Compiler Transformations for High-Performance Computing.* – Technical Report n° UCB/CSD-93-781, Berkeley, Univ. of California, 1993.

[BHG87] Bernstein (P.A.), Hadzilacos (V.) et Goodman (N.). – *Concurrency Control and Recovery in Database Systems.* – Addison-Wesley Publisher, 1987.

[Boo94] Booch (G.). – *Object-Oriented Analysis and Design with Applications.* – Benjamin Cummings, 1994, 2 édition.

[BR88] Badrinath (B.R.) et Ramamrithan (K.). – Synchronizing transactions on objects. *IEEE Transactions on Computers*, vol. 37, n° 5, mai 1988, pp. 541–547.

[BR92] Badrinath (B.R.) et Ramamrithan (K.). – Semantic-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, vol. 17, n° 1, mars 1992, pp. 163–199.

[Bra93] Braunl (T.). – *Parallel Programming: An Introduction.* – Prentice-Hall, 1993.

[BSCD91] Borla-Salamet (P.), Chachaty (C.) et Dageville (B.). – Compiling control into database queries for parallel execution management. *In : Proc. of the 1st Int. Conf. on Parallel and Distributed Information System.* pp. 253–275. – Miami, décembre 1991.

[BST89] Bal (H.E.), Steiner (J.G.) et Tanenbaum (A.S.). – Programming language for distributed computing systems. *ACM Computing Surveys*, vol. 21, n° 3, septembre 1989, pp. 261–322.

[CABK88] Copeland (G.), Allexander (W.), Boughter (E.) et Keller (T.). – Data placement in Bubba. *In : Proc. of the 1988 ACM SIGMOD Int. Conf. on Management of Data.* pp. 99–108. – Chicago, juin 1988.

[CAD+94] Cattel (R.), Atwood (T.), Dubl (J.), Ferran (G.), Loomis (M.) et Wade (D.). – *The Object Database Standart: ODMG-93.* – Morgan Kaufmann, 1994.

[Car89] Caromel (D.). – A general model for concurrent and distributed object-oriented programming. *ACM Sigplan Notices*, vol. 24, n° 4, avril 1989, pp. 102–107.

[Cat91] Cattel (R.). – *Object Data Management.* – Addison-Wesley Publisher, 1991.

[CC91] Chin (R.S.) et Chanson (S.T.). – Distributed object-based programming systems. *ACM Computing Surveys*, vol. 23, n° 1, mars 1991, pp. 91–124.

[CC96] Collet (C.) et Coupaye (T.). – Architecture and implementation of the NAOS active rule system. *In : Submitted to 12th IEEE Int. Conf. on Data Engineering.* – New Orleans, Louisiana, février 1996.

[CCS94] Collet (C.), Coupaye (T.) et Svensen (T.). – NAOS efficient and modular reactive capabilities in an object-oriented database system. *In : Proc. of the 20th Int. Conf. on Very Large Data Bases*, pp. 132–143. – Santiago, Chile, septembre 1994.

[CD89] Chen (R.C.) et Dasgupta (P.). – Linking consistency with object/thread semantics: An approach to robust computation. *In : Proc. of the 9th Int. Conf. on Distributed Computing Systems*, pp. 121–128. – Newport Beach, juin 1989.

[CF92] Cart (M.) et Ferrié (J.). – Integrating concurrency control. *In : Building an Object-Oriented Database System*, éd. par Bancilhon (F.), Delobel (C.) et Kanellakis (P.), chap. 20, pp. 463–485. – Morgan Kaufmann, 1992.

[CFR89] Cart (M.), Ferrié (J.) et Richy (H.). – Contrôle de l'exécution de transactions concurrentes. *Technique et Science Informatiques*, vol. 8, n° 3, 1989, pp. 225–240.

[CHCA94] Collet (C.), Habraken (P.), Coupaye (T.) et Adiba (M.). – Active rules for the GOODSTEP software engineering platform. *In : Proc. of the 2nd*

Int. Workshop on Database and Software Engineering. – Sorrento, Italy, mai 1994.

[Chr91] Chrysanthis (P.K.). – *ACTA: A Framework for Modeling and Reasoning about Extended Transactions.* – Amherst, Thèse de PhD, University of Massachussetts, 1991.

[CL91] Corradi (A.) et Leonardi (L.). – PO constrains as tools to synchronize active objects. *Journal of Object-Oriented Programming*, vol. 4, n° 3, octobre 1991, pp. 41–53.

[CLYY92] Chen (M-S.), Lo (M.), Yu (P.S.) et Young (H.C.). – Using segmented right-deep trees for the execution of pipelined hash joins. *In : Proc. of the 18th Int. Conf. on Very Large Data Bases*, pp. 15–26. – Vancouver, août 1992.

[CM95] Collet (C.) et Machado (J.). – Optimization of active rules with parallelism. *In : Proc. of the Int. Workshop on Active and Real-Time Database Systems.* – Skovde - Sweden, juin 1995.

[CM96] Collet (C.) et Machado (J.). – A parallel execution model for object-oriented database applications. *In : Submitted to 5th Int. Conf. on Extending Database Technology.* – Avignon, mars 1996.

[Cos93] Costa (C.M.). – *Environnement d'Exécution Parallèle : Conception et Architecture.* – Grenoble, Thèse de PhD, Université Joseph Fourier, octobre 1993.

[CRR91] Chrysanthis (P.K.), Raghuram (S.) et Ramamritham (K.). – Extracting concurrency from objects: A methodology. *In : Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data.* pp. 108–117. – Denver, Colorado, juin 1991.

[CW85] Cardelli (L.) et Wegner (P.). – On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, vol. 17, n° 4, décembre 1985, pp. 471–522.

[DA82] Delobel (C.) et Adiba (M.). – *Bases de Données et Systèmes Relationnels.* – Dunod, 1982.

[Dec93] Dechamboux (P.). – *Gestion d'Objets Persistants : du Langage de Programmation au Système.* – Grenoble, Thèse de PhD, Université Joseph Fourier, février 1993.

[Deu91] Deux (O.). – The O2 system. *Communications of the ACM*, vol. 34, n° 10, octobre 1991, pp. 34–48.

[DG85] DeWitt (D.J.) et Gerber (R.). – Multiprocessor hash-based join algorithms. *In : Proc. of the 11th Int. Conf. on Very Large Data Bases*, pp. 151–164. – Stockholm, août 1985.

[DG90] DeWitt (D.J.) et Gray (J.). – Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, vol. 19, n° 4, décembre 1990, pp. 104–112.

[DG92] DeWitt (D.J.) et Gray (J.). – Parallel database systems: The future of high performance database systems. *Communications of the ACM*, vol. 35, n° 6, juin 1992, pp. 85–98.

[DGS88] DeWitt (D.J.), Ghandeharizadeh (S.) et Schneider (D.). – A performance analysis of the Gamma database machine. *In : Proc. of the 1988 ACM SIGMOD Int. Conf. on Management of Data*. pp. 350–360. – Chicago, mai 1988.

[DGS+90] DeWitt (D.J.), Ghandeharizadeh (S.), Schneider (D.), Bricker (A.), Hsiao (H.) et Rasmussen (R.). – The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, n° 1, mars 1990, pp. 112–124.

[DLA88] Dasgupta (P.), LeBlanc Jr. (R.J.) et Appelbe (W.F.). – The Clouds distributed operating system. *In : Proc. of the 8th Int. Conf. on Distributed Computing Systems*. pp. 2–9. – San Jose, juin 1988.

[DLR91] Delobel (C.), Lécluse (C.) et Richard (P.). – *Bases de Données: des Systèmes Relationnels aux Systèmes à Objets*. – InterEditions, 1991.

[DNB93] DeWitt (D.J.), Naughton (J.F.) et Burger (J.). – Nested loops revisited. *In : Proc. of the 2nd Int. Conf. on Parallel and Distributed Information System*. pp. 230–250. – San Diego, janvier 1993.

[DNSS92] DeWitt (D.J.), Naughton (J.F.), Schneider (D.A.) et Seshadri (S.). – Practical skew handling in parallel join. *In : Proc. of the 18th Int. Conf. on Very Large Data Bases*, pp. 27–40. – Vancouver, août 1992.

[EGLT76] Eswaran (K.), Gray (J.), Lorie (R.) et Traiger (I.). – The notion of consistency and predicate locks in a database system. *Communications of the ACM*, vol. 19, n° 11, apr 1976, pp. 624–632.

[Elm92] Elmagarmid (A.K.) (édité par). – *Database Transactions Models for Advanced Applications.* – Morgan Kaufmann, 1992.

[EMS91] Eppinger (J.L.), Mummert (L.B.) et Spector (A.Z.). – *Camelot and Avalon: A Distributed Transaction Facility.* – Morgan Kaufmann, 1991.

[EN89] Elmasri (R.) et Navathe (S.). – *Fundamentals of Database Systems.* – Benjamin Cummings, 1989.

[FLMW89] Fekete (A.), Lynch (N.), Merritt (M.) et Weihl (W.). – Commutativity-based locking for nested transaction. *In : Proc. of the 3rd Int. Workshop on Persistent Object Systems.* pp. 319–340. – Newcastle, 1989.

[FLW92] Fekete (A.), Lynch (N.) et Weihl (W.). – Hybrid atomicity for nested transaction. *In : Proc. of the 4th Int. Conf. on Database Theory.* pp. 216–230. – Berlin, 1992.

[Fly72] Flynn (M.J.). – Directions and issues in architecture and language. *IEEE Computer*, vol. 13, n° 10, octobre 1972, pp. 948–960.

[FOW87] Ferrante (J.), Ottenstein (K.J.) et Warren (J.D.). – The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, vol. 9, n° 3, juillet 1987, pp. 319–349.

[FS92] Feldman (Y.) et Shapiro (E.). – Spatial machines: A more realistic approach to parallel computation. *Communications of the ACM*, vol. 35, n° 10, octobre 1992, pp. 61–73.

[FW78] Fortune (S.) et Wyllie (J.). – Parallelism in random access machines. *In : Proc. of the 12th Annual ACM Symposium on Theory of Computing*, pp. 114–118.

[GD90a] Ghandeharizadeh (S.) et DeWitt (D.J.). – Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. *In : Proc. of the 16th Int. Conf. on Very Large Data Bases*, pp. 481–492. – Brisbane, août 1990.

[GD90b] Ghandeharizadeh (S.) et DeWitt (D.J.). – A multiuser performance analysis of alternative declustering strategies. *In : Proc. of the 6th IEEE Int. Conf. on Data Engineering.* pp. 466–475. – Los Angeles, février 1990.

[Gel89] Gelenbe (E.). – *Multiprocessor Performance.* – John Wiley & Sons, 1989.

[GOO94] GOODSTEP Team. – The GOODSTEP Project: General Object-Oriented Database for SofTware Engineering Processes. *In : Proc. of the Asia-Pacific Software Engineering Conference*. pp. 410–420. – Tokyo, Japan, 1994.

[GOP92] Gorlen (K.E.), Orlow (S.M.) et Plexico (P.S.). – *Data Abstraction and Object-Oriented Programming in C++*. – John Wiley & Sons, 1992.

[GR93] Gray (J.) et Reuter (A.). – *Transaction Processing: Concepts and Techniques*. – Morgan Kaufmann, 1993.

[Gra81] Gray (J.). – The transaction concept: Virtues and limitations. *In : Proc. of the 7th Int. Conf. on Very Large Data Bases*. pp. 144–154. – Cannes, France, septembre 1981.

[Gra94] Graefe (G.). – Volcano: An extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, n° 1, février 1994, pp. 120–135.

[HH91] Hadzilacos (T.) et Hadzilacos (V.). – Transaction synchronization in object bases. *Journal of Computer and System Sciences*, vol. 43, n° 1, août 1991, pp. 2–24.

[HKM+94] Haines (N.), Kindred (D.), Morrisett (J.G.), Nettles (S.M.) et Wing (J.M.). – Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, vol. 16, n° 6, novembre 1994, pp. 1719–1736.

[Hon92] Hong (W.). – Exploiting inter-operation parallelism in XPRS. *In : Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*. pp. 19–28. – San Diego, juin 1992.

[HR93] Härder (T.) et Rothermel (K.). – Concurrency control issues in nested transaction. *The VLDB Journal*, vol. 2, n° 1, janvier 1993, pp. 39–74.

[HW88] Herlihy (M.) et Weihl (M.). – Hybrid concurrency control for abstract data types. *In : Proc. of the 7th ACM Symposium on Principles of Database Systems*, pp. 201–210. – Austin, mars 1988.

[KGBW90] Kim (W.), Garza (J.F.), Ballou (N.) et Woelk (D.). – Architecture of the Orion next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, n° 1, mars 1990, pp. 109–123.

[Kim90] Kim (W.). – *Introduction to Object Oriented Databases.* – MIT Press, 1990.

[Kit94] Kitajima (J.P.F.W.). – *Modèles Quantitatifs d'Algorithmes Parallèles.* – Grenoble, Thèse de PhD, Institut National Polytechnique de Grenoble, octobre 1994.

[KL88] Kruatrachue (B.) et Lewis (T.). – Grain size determination for parallel programming. *IEEE Software*, vol. 5, n° 1, janvier 1988, pp. 23–32.

[KL89] Kim (W.) et Lochovsky (F.H.) (édité par). – *Object-Oriented Concepts, Databases, and Applications.* – Addison-Wesley Publisher, 1989.

[KL90] Kafura (D.) et Lee (K.H.). – ACT++: Building a concurrent C++ with actors. *Journal of Object-Oriented Programming*, vol. 3, n° 2, mai 1990, pp. 25–37.

[KMV+90] Krakowiak (S.), Meysembourg (M.), Van (H. Nguyen), Riveill (M.), Roisin (C.) et de Pina (X. Rousset). – Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, vol. 3, n° 3, septembre 1990, pp. 11–22.

[KO90] Kitsuregawa (M.) et Ogawa (Y.). – Bucket spreading parallel hashing: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC). *In: Proc. of the 16th Int. Conf. on Very Large Data Bases*, pp. 210–221. – Brisbane, août 1990.

[Kor83] Korth (H.). – Locking primitives in a database system. *Journal of the ACM*, vol. 30, n° 1, janvier 1983, pp. 55–79.

[Kra91] Krakowiak (S.). – Rappels sur les systèmes de communication et les machines parallèles. *In: Construction des Systèmes d'Exploitation Répartis*, éd. par Balter (R.), Banatre (J.-P.) et Krakowiak (S.), chap. 2. – INRIA, 1991.

[LDH+89] Lorie (R.), Daudenarde (J-J.), Hallmark (G.), Stamos (J.) et Young (H.). – Adding intra-transaction parallelism to an existing DBMS: Early experience. *Database Engineering*, vol. 8, 1989, pp. 2–8.

[LER92] Lewis (T.G.) et El-Rewini (H.). – *Introduction to Parallel Computing.* – Prentice-Hall, 1992.

[Les93] Lester (B.P.). – *The Art of Parallel Programming.* – Prentice-Hall, 1993.

[Lis88] Liskov (B.). – Distributed programming in Argus. *Communications of the ACM*, vol. 31, n° 3, mars 1988, pp. 300–312.

[LKB87] Livny (M.), Khoshafian (S.) et Boral (H.). – Multi-disk management algorithms. *In : Proc. of the 1987 SIGMETRICS Conference.* – Banff, mai 1987.

[LKK85] Lee (G.), Kruskal (C.P.) et Kuck (D.J.). – An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Transactions on Computers*, vol. 34, n° 10, octobre 1985, pp. 927–933.

[LLOW91] Lamb (C.), Landis (G.), Orenstein (J.) et Weinreb (D.). – The Objectstore database system. *Communications of the ACM*, vol. 34, n° 10, octobre 1991, pp. 50–63.

[LST91] Lu (H.), Shan (M.) et Tan (K.). – Optimization of multi-way join queries for parallel execution. *In : Proc. of the 17th Int. Conf. on Very Large Data Bases*, pp. 549–560. – Barcelona, septembre 1991.

[LTS90] Lu (H.), Tan (K.) et Shan (M.). – Hash-based join algorithms for multiprocessor computers with shared memory. *In : Proc. of the 16th Int. Conf. on Very Large Data Bases*, pp. 198–209. – Brisbane, aug 1990.

[MCD+93a] Machado (J.C.), Collet (C.), Defude (B.), Dechamboux (P.) et Adiba (M.). – The parallel PEPLOM execution model. *In : Proceedings of the 31st Annual ACM Southeast Conference.* – Birmingham - USA, avril 1993.

[MCD+93b] Machado (J.C.), Collet (C.), Defude (B.), Dechamboux (P.) et Adiba (M.). – Parallelism in an object-oriented database programming language. *In : Anais do VIII Simpósio Brasileiro de Banco de Dados.* pp. 358–368. – Campina Grande, mai 1993.

[MM93] Malta (C.) et Martinez (J.). – Automating fine concurrency control in object-oriented databases. *In : Proc. of the 9th IEEE Int. Conf. on Data Engineering*, pp. 253–260. – Vienna, avril 1993.

[MNC+89] Masini (G.), Napoli (A.), Colnet (D.), Léonard (D.) et Tombre (K.). – *Les Languages à Objets.* – InterEditions, 1989.

[Mos85] Moss (J.E.B.). – *Nested Transactions: An Approach to Reliable Distributed Computing.* – Cambridge, Ma, The Mit Press, 1985.

[MRW+93] Muth (P.), Rakow (T.C.), Weikum (G.), Broessler (P.) et Hasse (C.). – Semantic concurrency control in object-oriented database systems. *In : Proc. of the 9th IEEE Int. Conf. on Data Engineering*, pp. 233–242. – Vienna, avril 1993.

[Nie87] Nierstrasz (O.M.). – Active objects in Hybrid. *In : Proc. of the 1987 Object-Oriented Programming Systems Languages and Applications*, pp. 243–253. – Orlando, octobre 1987.

[ODV94] Özsu (M.T.), Dayal (U.) et Valduriez (P.). – *Distributed Object Management.* – Morgan Kaufmann, 1994.

[OL92] Omiecinski (E.) et Lin (E.). – The adaptive-hash join algorithm for a hypercube multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, n° 3, 1992, pp. 334–349.

[Omi91] Omiecinski (E.). – Performance analysis of a load balance relational hash join algorithm for a shared memory multiprocessor. *In : Proc. of the 17th Int. Conf. on Very Large Data Bases*, pp. 375–385. – Barcelona, septembre 1991.

[Omi95] Omiecinski (E.). – Parallel relational database systems. *In : Modern Database Systems*, éd. par Kim (W.), chap. 24, pp. 494–512. – ACM Press, 1995.

[OV91] Özsu (M.T.) et Valduriez (P.). – *Principles of Distributed Database Systems.* – Prentice-Hall, 1991.

[Pap86] Papadimitriou (C.H.). – *The Theory of Database Concurrency Control.* – Computer Science Press, 1986.

[Pap92] Papathomas (M.). – *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* – Thèse de PhD, Université de Genève, 1992.

[Per87] Perrot (R.H.). – *Parallel Programming.* – Addison-Wesley Publisher, 1987.

[PW86] Padua (D.) et Wolfe (M.). – Advanced compiled optimizations for supercomputers. *Communications of the ACM*, vol. 29, n° 12, décembre 1986.

[RB87] Roesler (M.) et Burkhard (W.). – Concurrency control scheme for shared objects : A peephole approach based on semantics. *In : Proc. of the 7th Int. Conf. on Distributed Computing Systems*, pp. 224–231. – Berlin, septembre 1987.

[SC90] Shekita (E.) et Carey (M.J.). – A performance evaluation of pointer-based joins. *In : Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data*. pp. 300–311. – Atlantic City, NJ, mai 1990.

[SD89] Schneider (D.A.) et DeWitt (D.J.). – A performance evaluation of four parallel join algorithms in a sharing-nothing multiprocessor environnement. *In : Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data*. pp. 110–121. – Portland, Or, juin 1989.

[SD90] Schneider (D.A.) et DeWitt (D.J.). – Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. *In : Proc. of the 16th Int. Conf. on Very Large Data Bases*, pp. 469–480. – Brisbane, aug 1990.

[SK91] Stonebraker (M.) et Kemnitz (G.). – The Postgres next-generation database management systems. *Communications of the ACM*, vol. 34, n° 10, octobre 1991, pp. 78–92.

[SKPO88] Stonebraker (M.), Katz (R.), Patterson (D.) et Ousterhout (J.). – The design of XPRS. *In : Proc. of the 14th Int. Conf. on Very Large Data Bases*, pp. 318–330. – Los Angeles, Ca, août 1988.

[Ste87] Stein (L.A.). – Delegation is inheritance. *In : Proc. of the 1987 Object-Oriented Programming Systems Languages and Applications*, pp. 138–155. – Orlando, octobre 1987.

[Str92] Stroustrup (B.). – *The C++ Programming Language*. – Addison-Wesley Publisher, 1992.

[SZ89] Skarra (A.H.) et Zdonik (S.B.). – Concurrency control in object-oriented databases. *In : Object-Oriented Concepts, Databases, and Applications*, éd. par Kim (W.) et Lochovsky (F.H.), chap. 16, pp. 395–421. – Addison-Wesley Publisher, 1989.

[Tan92] Tanenbaum (A.S.). – *Modern Operating Systems*. – Prentice-Hall, 1992.

[Tec94] Technology (O_2). – *The O_2 User Manual - Version 4.4*, janvier 1994.

[TNW91] Tokoro (M.), Nierstrasz (O.) et Wegner (P.) (édité par). – *Object-Based Concurrent Computing.* – Genève, Springer Verlag, juillet 1991.

[Val87] Valduriez (P.). – Join indices. *ACM Transactions on Database Systems*, vol. 12, n° 2, juin 1987, pp. 218–246.

[Val90] Valiant (L.). – A bridging model for parallel computation. *Communications of the ACM*, vol. 33, n° 8, août 1990, pp. 103–111.

[WB87] Wolfe (M.) et Banerjee (U.). – Data dependence and its application to parallel processing. *Int. Journal of Parallel Programming*, vol. 16, n° 2, 1987, pp. 137–179.

[WDJ91] Walton (C.B.), Dale (A.G.) et Jenevein (R.M.). – A taxonomy and performance model of data skew effects in parallel join. *In : Proc. of the 17th Int. Conf. on Very Large Data Bases*, pp. 537–548. – Barcelona, SP, septembre 1991.

[WDYT91] Wolf (J.L.), Dias (D.M.), Yu (P.S.) et Turek (J.). – Comparative performance of parallel join algorithms. *In : Proc. of the 1st Int. Conf. on Parallel and Distributed Information System*, pp. 78–88. – Miami, décembre 1991.

[Weg87] Wegner (P.). – Dimensions on object-based language design. *In : Proc. of the 1987 Object-Oriented Programming Systems Languages and Applications*, pp. 168–182. – Orlando, octobre 1987.

[Wei88] Weihl (W.). – Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, vol. 37, n° 12, décembre 1988, pp. 1488–1505.

[Wei89a] Weihl (W.). – The impact of recovery on concurrency control. *In : Proc. of the 8th ACM Symposium on Principles of Database Systems*, pp. 259–269. – Philadelphia, Pe, mars 1989.

[Wei89b] Weihl (W.). – Theory of nested transactions. *In : Distributed Systems*, éd. par Mullender (S.), pp. 262–290. – ACM Press, 1989.

[WFH+93] Wing (J.M.), Faehndrich (M.), Haines (N.), Kietzke (K.), Kindred (D.), Morrisett (J.G.) et Nettles (S.). – *Venari/ML Interfaces and Examples.* – CMU-CS-93-123, Pittsburgh, Carnegie Mellon Univ., mar 1993.

[WKH92] Wyatt (B.B.), Kavi (K.) et Hufnagel (S.). – Parallelism in object-oriented languages: A survey. *IEEE Software*, vol. 9, n° 11, novembre 1992, pp. 56–66.

[YBS86] Yonezawa (A.), Briot (J-P.) et Shibayama (E.). – Object-oriented concurrent programming in ABCL/1. *In : Proc. of the 1986 Object-Oriented Programming Systems Languages and Applications*, pp. 258–268. – Portland, novembre 1986.

[YT87] Yonezawa (A.) et Tokoro (M.) (édité par). – *Object-Oriented Concurrent Programming*. – The Mit Press, 1987.

[ZG90] Zeller (H.) et Gray (J.). – An adaptive hash join algorithm for multiuser environments. *In : Proc. of the 16th Int. Conf. on Very Large Data Bases*, pp. 186–197. – Brisbane, août 1990.

[ZZBS93] Ziane (M.), Zaït (M.) et Borla-Salamet (P.). – Parallel query processing with zigzag trees. *The VLDB Journal*, vol. 2, n° 3, juillet 1993, pp. 277–301.

Annexe A

La Méthode Booch

Cette annexe présente les éléments de base de la méthode à objets de Booch [Boo94] permettant au lecteur une meilleure compréhension de la notation utilisée dans le chapitre 4.

La méthode de Booch partage la spécification d'un système en deux modèles : un modèle logique et un modèle physique. Le modèle logique permet de décrire les classes et leurs méthodes. Ce modèle permet également de présenter les itérations entre les instances des classes. Ainsi, dans le modèle logique on décrit les aspects structurels et dynamiques des classes du système. Le modèle physique permet de décrire l'organisation des classes dans les modules de implantation du système.

Le modèle logique permet de décrire et de définir le vocabulaire et les concepts du domaine d'un problème donné, au moyen d'une technique d'organisation et d'un langage graphique appropriés [Boo94]. Dans le modèle logique, on décrit à la fois les aspects structurels mais aussi les aspects dynamiques liés aux concepts du domaine. Pour cela, nous utilisons les *diagrammes de classes*, les *diagrammes d'objet* et les *diagrammes de catégories de classes*.

A.1 Diagramme de classes

Les diagrammes de classes sont utilisés pour montrer l'existence des classes et leurs relations avec une vision logique du système. Un diagramme de classes représente une abstraction du domaine du problème. Les deux éléments essentiels d'un diagramme de classes sont les *classes* et leurs *relations*.

A.1.1 Les classes

Une classe définit une structure et un comportement communs à tous ses objets. La figure A.1(a) présente la notation employée pour représenter les classes dans la

méthode. Dans cette notation, au-dessus de la ligne, on trouve le nom de la classe et au-dessous, les noms de ses attributs et de ses opérations.

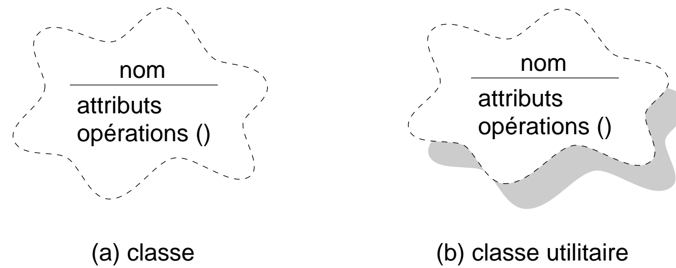


FIG. A.1 - *Les classes*

- Les attributs décrivent une propriété particulière des objets d'une classe. Chaque attribut possède un nom et un domaine. Dans un approche à objet, le domaine est spécifié par un nom de classe. On peut les représenter sous une des formes suivantes :
  - "A" montre uniquement le nom de l'attribut.
  - "A : C" montre le nom de l'attribut et son domaine (la classe C.
  - "A : C = E" montre le nom de l'attribut, son domaine et une expression dont l'évaluation donne une valeur par défaut à l'attribut dans une instance de la classe.
- Les opérations spécifient les services que sont accomplis pour la classe. La notation pour différencier les attributs des opérations est faite en utilisant des parenthèses après le nom de l'opération. Par exemple, "OP()" donne le nom de l'opération;

La méthode introduit un type spécial de classe, appelé *classe utilitaire*, qui sert à modéliser des librairies de classes fournissant des services aux autres classes du système. La notation utilisée pour représenter les classes utilitaires est montrée dans la figure A.1(b).

A.1.2 Relations entre classes

La méthode Booch propose quatre types différents de relations dont les notations sont présentées dans la figure A.2 :

- La relation *Association* représente une relation générale entre deux classes. Une telle relation doit être nommée avec un nom qui montre la nature de l'associa-

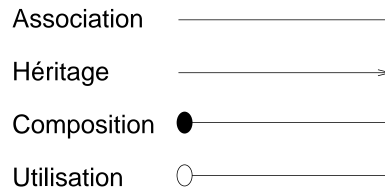


FIG. A.2 - *Les relations entre les classes*

tion. On doit fournir également la cardinalité de l'association.

- La relation *Héritage* représente une relation du type généralisation-spécialisation.
- La relation *Composition* représente une relation du type Composé/Composant.
- La relation *Utilisation* représente une relation du type client/serveur, où le serveur typiquement fournit des services au client.

La figure A.3 montre un exemple des relations entre classes. Les classes clas2 et clas3 sont sous-classes de clas1. Ainsi clas2 et clas3 ont une relation d'héritage avec clas1. Cette classe a une relation de composition avec la classe clas4 qui décrit le fait qu'une instance de clas1 possède plusieurs instances de clas4. Une association appelée faitRef établit une relation entre les classes clas2 et clas4. Finalement, la classe clas4 possède une relation d'utilisation avec la classe utilitaire util1 qui décrit le fait que des instances de clas4 envoient des messages aux instances d'util1.

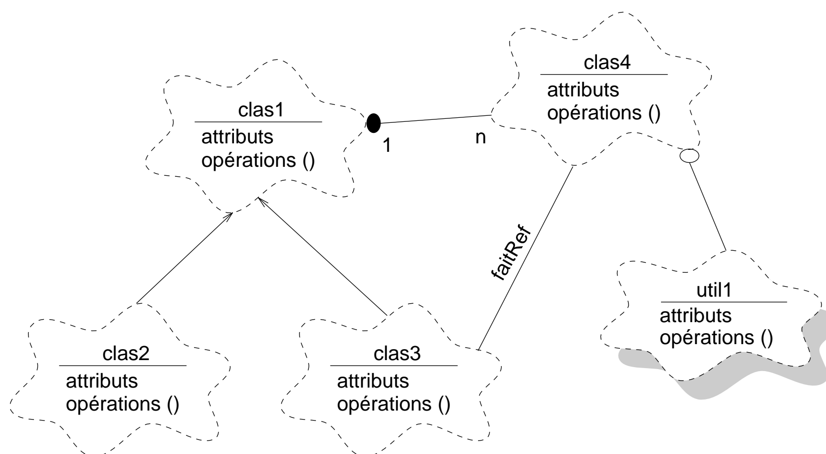


FIG. A.3 - *Les relations entre les classes*

A.2 Catégories de classes

Les catégories de classes sont utilisées pour structurer un système. Elles sont composées de classes et d'autres catégories de classes. La notation utilisée pour représenter les catégories de classes est donnée dans la figure A.4. La description d'une catégorie comprend son nom et les nom de ses classes composantes.

Une catégorie de classes peut utiliser d'autres catégories de classes : c'est à dire les services offerts par une classe d'une autre catégorie. Alors on dit qu'il existe une "relation d'utilisation" entre les deux catégories. La méthode utilise la notation employée pour spécifier les relations utilisation entre classes. La figure A.4 montre des exemples de spécification de catégories de classes et leurs relations.

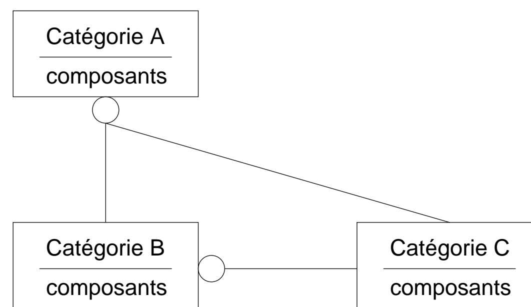


Fig. A.4 - *Les catégories de classes*

A.3 Diagramme d'objets

Les diagrammes d'objets sont utilisés pour montrer l'existence des objets et leurs relations dans le modèle logique du système. Un diagramme représente une photo instantanée dans le temps d'une certaine configuration d'objets. Les éléments essentiels d'un diagramme d'objets sont les *objets* et les *relations*.

A.3.1 Objets

Les objets possèdent un état, un comportement et une identité. Un objet est instance d'une classe. La structure et le comportement des objets similaires sont définis dans leur classe. Les termes *instance* et *objet* sont permutables. La spécification des objets suit la syntaxe de celle des attributs :

- "O" spécifie un objet de nom O d'une classe quelconque.

- ":C" spécifie un objet de la classe C
- "O : C" spécifie un objet de nom sf O de la classe C.

La figure A.5 présente la notation employée pour représenter les objets dans la méthode.

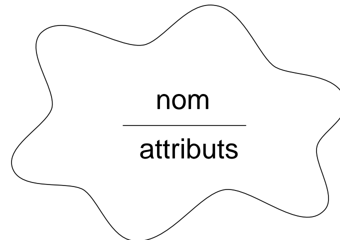


FIG. A.5 - *Les objets*

A.3.2 Relations entre objets

Les relations entre objets expriment la manière dont les objets sont liés les uns aux autres. Ces liens sont des instances des associations de la même manière que les objets sont des instances des classes. Les liens sont permis uniquement entre objets dont les classes sont "en relation".

A.3.2.1 Les messages

Un message est une demande d'exécution d'opération qu'un objet client exprime à un objet serveur. En général, le client connaît le serveur mais le serveur ne connaît pas le client. Les messages sont présentés comme des décorations des liens entre objets. Une flèche est utilisée comme symbole de synchronisation pour montrer la direction des messages séquentiels, où la sémantique est assurée par une séquence de contrôle. Nous montrons un exemple dans la figure A.6, où séquence est le numéro de séquence de l'envoi du message, message() dénote l'opération demandée et la flèche indique la direction de l'envoi du message.

A.4 Diagramme de modules

Les diagrammes de modules sont utilisés pour montrer l'allocation de classes et d'objets dans des modules du projet physique d'un système. Ainsi ces diagrammes

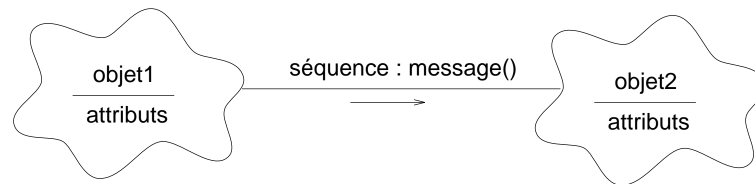


Fig. A.6 - *Les envois des messages*

donnent une vision de la structure modulaire du système. Les éléments essentiels d'un diagramme sont les *modules* et les *dépendances*.

A.4.1 Modules

La méthode propose deux types de modules (voir figure A.7) :

- *Le module spécification* représentent les fichiers qui contiennent les *déclarations* des classes d'un système. En C++, par exemple, ce sont des fichiers ".h".
- *Le module corps* représentent les fichiers qui contiennent les *définitions* des classes d'un système. En C++, par exemple, ce sont des fichiers ".cpp".

A.4.2 Dépendances

Les dépendances sont des relations entre modules que décrivent une *dépendance de compilation*. En C++, par exemple, les dépendances de compilation sont décrites par les directives "# include". La notation utilisée pour une dépendance est une flèche qui part d'un module vers un autre module.

La figure A.7 montre la notation pour représenter les modules. La flèche partant du module A vers le module B indique que le module A "dépend" du module B.

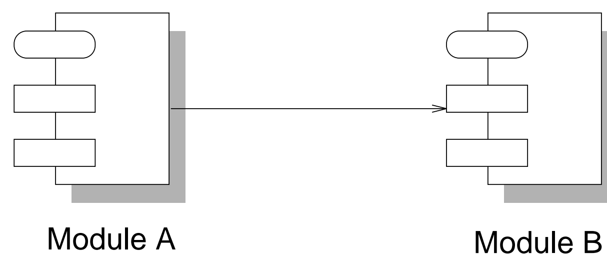


Fig. A.7 - *Les modules des classes*

Résumé : Cette thèse cherche à exploiter le parallélisme dans le contexte des systèmes de gestion de bases de données à objet à d'autres niveaux d'abstraction que celui d'une requête SQL, à savoir le parallélisme intra-transaction et le parallélisme intra-application. Notre approche pour parallélisation des transactions considère un modèle de transactions classiques, ou une transaction est une séquence d'opérations. Le parallélisme intra-transaction consiste à transformer le code d'une transaction pour pouvoir exécuter ses opérations en parallèle. Notre approche pour l'exploitation du parallélisme intra-application étend tout d'abord le modèle de parallélisme intra-transaction de manière à considérer la transaction comme unité de parallélisation. Deuxièmement nous avons considéré l'exploitation du parallélisme intra-application dans un contexte ou le modèle de transactions fournit des transactions emboîtées. Nous avons développé un modèle de parallélisation des applications où nous associons le parallélisme offert par le modèle des transactions emboîtées avec le parallélisme par transformation de notre approche de parallélisation des transactions.

Nous avons implanté un premier prototype qui met en œuvre le modèle de parallélisation des transactions. Pour cela, nous avons utilisé le système de bases de données à objet O2. Notre prototype introduit le parallélisme par la création et la synchronisation des activités parallèles au sein du processus client O2 qui exécute une application. Le système étant développé sur une machine monoprocesseur, les fonctions liées au parallélisme utilisent de processus légers. Nous avons appliqué ensuite notre modèle de parallélisation au système de règles NAOS. Notre approche considère l'ensemble de règles d'un cycle d'exécution, dites règles candidates, pour la parallélisation. Nous construisons un plan d'exécution pour les règles candidates d'un cycle qui détermine l'exécution séquentielle ou parallèle des règles.

Mots clés : bases de données, transactions, parallélisme, transformation de code

Abstract : This thesis defines an approach for exploring parallelism in object-oriented database systems outside of the context of SQL queries. We propose a technique for parallelazing transactions in a flat classical transaction model where a transaction is sequence of operations. Intra-transaction parallelism is accomplished by transforming transaction code definition in order to execute operations in parallel. Our approach for exploring parallelism inside applications first extends the intra-transaction parallelization model so that a transaction is considered as an unit of parallelization. We have then considered a nested transaction model for exploring parallelism inside applications. We developed a parallelization model for applications where we merge capabilities for parallel execution already given in nested transactions with our approach for transaction parallelization by transformation.

We implemented a prototype for the intra-transaction parallelization model, using the O2 object-oriented database system. The prototype introduces parallel execution

inside O2 transactions through creation and synchronization of threads inside an O2 client running an application. Our prototype runs in a monoprocessor Unix-like workstation and supports virtual parallelism. We also applied the transaction parallelization model to the NAOS Rule System. Our approach considers a set of rules of an execution cycle for parallelization. We build an execution plan for the rules of a cycle which defines sequential or parallel execution for the rules.

Keywords: databases, transactions, parallelism, code transaformation