Contribution aux problèmes de contrôle de concurrence et de reprise dans les bases de données à objets

José Martinez

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

THÈSE

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

Spécialité : INFORMATIQUE
Formation Doctorale : INFORMATIQUE

CONTRIBUTION AUX PROBLÈMES DE CONTRÔLE DE CONCURRENCE ET DE REPRISE DANS LES BASES DE DONNÉES À OBJETS

par

José MARTINEZ

soutenue le 2 décembre 1992 devant le jury composé de :

M. CARREZ Christian, Professeur, CNAM, Paris	Rapporteur
M. COGIS Olivier, Professeur, Université Montpellier II	Examinateur
M. FERRIÉ Jean, Professeur, Université Montpellier II	Directeur de Thèse & Président
M. GIRAULT Claude, Professeur, Université Paris VI	Rapporteur
M. PONS Jean-François, Maître de conférences, Université Montpellier II	Examinateur

---

M. CARREZ Christian, Professeur, CNAM, Paris	Rapporteur
M. GIRAULT Claude, Professeur, Université Paris VI	Rapporteur
M. MIRANDA Serge, Professeur, Université de Nice Sophia-Antipolis	Rapporteur

Remerciements

Qu'il me soit ici permis d'adresser des remerciements, et tout d'abord aux rapporteurs et membres du jury. J'emprunterai pour cela les commodités de l'ordre alphabétique.

Merci donc à M. Christian Carrez pour l'intérêt qu'il a porté à cette thèse et pour avoir accepté d'être rapporteur. L'attention soutenue qu'il a apportée à son travail a permis d'affiner les détails de cette thèse.

Je remercie M. Olivier Cogis de m'avoir fait l'amitié d'être examinateur. Qu'il sache, de plus, que ses conseils et son encadrement en tant que tuteur pédagogique au département informatique de l'IUT de Montpellier ont été tout autant appréciés que son attention lors de l'examen de ce travail.

Je dois à M. Jean Ferrié une grande estime pour m'avoir accepté dans son équipe, dirigé dans mes recherches, encouragé dans les moments difficiles, tout autant que critiqué et conseillé sur le fond et sur la forme de mes travaux.

Un grand merci est dû à M. Claude Girault qui, dans son emploi du temps plus que surchargé, a su trouver les moments nécessaires à son travail laborieux de rapporteur. Qu'il trouve ici l'expression de ma reconnaissance.

Je ne saurais que remercier M. Serge Miranda d'avoir répondu à ma demande de faire partie des rapporteurs malgré l'impossibilité d'assister à la soutenance dans laquelle il s'est trouvée. Sa lecture et ses commentaires m'ont été précieux pour parachever l'expression finale de cette thèse.

Enfin, je remercie M. Jean-François Pons qui, étant examinateur mais au-delà membre de l'équipe de recherche, a toujours su trouver l'aspect anecdotique, voire humoristique, qui se cache derrière les apparences froides de notre domaine de recherche.

Mes remerciements envers l'équipe ne seraient pas complets si j'oubliais Mlle Michèle Cart et M. Carmelo Malta ainsi que les nouveaux thésards.

Plus largement encore, je remercie tous ceux que je ne peux citer par manque de place mais, qui par discussion, qui par menu service, ont rendu plus agréables encore ces années d'études.

Enfin, j'adresse de tendres remerciements à Nathalie, sans oublier Guillaume qui monopolise plus qu'il ne partage notre quotidien depuis quelques mois.

Table des matières

Introduction **1**
Cadre de l'étude et résultats . . . 3
Cadre de l'étude . . . 3
Résultats . . . 4
La gestion transactionnelle . . . 5
Les bases de données à objets . . . 8
Les buts des bases de données . . . 8
Les concepts « objets » . . . 9
Les bases de données à objets . . . 11
Organisation de l'ouvrage . . . 14

I Un état de l'art **17**
I.1 Introduction . . . 19
I.2 Théorie de la sérialisabilité . . . 20
I.2.1 Modèle de base de données . . . 20
I.2.1.1 Types de données abstraits transactionnels . . . 20
I.2.1.2 Relations entre opérations . . . 23
I.2.2 Modèle de transactions . . . 28
I.2.3 Histoires . . . 30
I.2.3.1 Histoires sérialisables . . . 31
I.2.3.2 Histoires recouvrables . . . 33
I.3 Contrôle de concurrence par verrouillage . . . 37
I.3.1 Verrouillage à deux phases strict . . . 37
I.3.2 Verrouillage hiérarchique . . . 38
I.4 Reprise par journalisation . . . 42
I.4.1 Principe de la journalisation et hypothèses . . . 42
I.4.1.1 Objets et opérations logiques . . . 44
I.4.1.2 Écritures non retardées et atomicité . . . 44
I.4.1.3 Écritures non forcées et durabilité . . . 45
I.4.2 L'algorithme « *REDO/UNDO* » ou « *repeating history* » . . . 46
I.4.2.1 Structures de données . . . 46
I.4.2.2 Traitement normal . . . 47
I.4.2.3 Phase « *REDO* » . . . 47
I.4.2.4 Phase « *UNDO* » . . . 48
I.5 Transactions multi-niveaux . . . 50
I.5.1 Sérialisabilité multi-niveau . . . 50
I.5.2 Contrôle de concurrence multi-niveau . . . 52
I.5.3 Reprise multi-niveau . . . 52

I.5.3.1 Journalisation au niveau 0 . . . 53
I.5.3.2 Journalisation aux niveaux supérieurs . . . 53
I.5.3.3 Reprise multi-niveau sur rejet . . . 54
I.5.3.4 Reprise multi-niveau sur panne . . . 54
I.6 Conclusion . . . 56

II Limites de la commutativité **57**
II.1 Introduction . . . 59
II.2 Le modèle . . . 61
II.2.1 Opérations sur les types de données abstraits . . . 61
II.2.2 Composition (C1) . . . 61
II.2.3 Inversibilité (C2) . . . 62
II.2.4 Commutativité de l'état (C3) . . . 62
II.2.5 Commutativité des vues (C4) . . . 63
II.2.6 Remarques sur le modèle . . . 64
II.3 Limites pour la concurrence . . . 68
II.3.1 Quelques « *folk-theorems* » . . . 68
II.3.2 Discussion . . . 70
II.3.3 Restriction aux ensembles bornés . . . 73
II.3.4 Conclusion partielle . . . 75
II.4 Avantages pour la reprise . . . 76
II.4.1 Transitivité . . . 76
II.4.2 Isolation . . . 76
II.4.3 Indépendance vis-à-vis du rejet . . . 77
II.5 Moyens d'échapper aux limites . . . 83
II.5.1 Indépendance . . . 83
II.5.2 Non-déterminisme . . . 87
II.5.3 Flou . . . 87
II.5.4 Commutativité mathématique . . . 88
II.5.5 Conclusion partielle . . . 89
II.6 Comparaison . . . 91
II.7 Conclusion . . . 94

III Contrôle de concurrence sur le graphe d'héritage **97**
III.1 Introduction . . . 99
III.2 Le modèle à objets . . . 100
III.2.1 Les données . . . 100
III.2.1.1 Les classes . . . 100
III.2.1.2 Le graphe d'héritage . . . 100
III.2.1.3 Les instances . . . 102
III.2.1.4 Les sous-graphes . . . 102
III.2.2 Les accès aux objets . . . 103
III.3 Le contrôle des accès concurrents . . . 104
III.3.1 Les types d'accès . . . 104
III.3.1.1 Le verrouillage d'un sous-graphe . . . 104
III.3.1.2 Les différents types de verrous . . . 108
III.3.2 Les modes d'accès . . . 110
III.3.3 Le verrouillage implicite, intentionnel et hiérarchique . . . 111

III.3.4 Les verrous . . . 112
III.3.4.1 La construction des verrous . . . 112
III.3.4.2 Les noms des verrous . . . 113
III.3.5 Le protocole . . . 114
III.4 Les modifications . . . 118
III.4.1 Les créations et destructions d'instances . . . 118
III.4.2 Les modifications du graphe d'héritage . . . 120
III.4.2.1 Les problèmes de cohérence . . . 120
III.4.2.2 Un nouveau protocole . . . 122
III.4.2.3 Critique de la solution . . . 127
III.4.3 Les lectures et modifications de méthodes . . . 128
III.5 Comparaisons . . . 129
III.5.1 Comparaison avec le protocole d'ORION . . . 129
III.5.2 Comparaison avec la proposition pour O_2 . . . 133
III.6 Conclusion . . . 135

IV Automatisation d'un contrôle de concurrence applicatif 137
IV.1 Introduction . . . 139
IV.2 Un modèle à objets . . . 140
IV.2.1 Le modèle de données . . . 140
IV.2.2 Le mécanisme d'envoi de messages . . . 142
IV.3 Quatre problèmes . . . 145
IV.4 Calcul des vecteurs d'accès transitifs . . . 149
IV.4.1 Définitions préliminaires . . . 149
IV.4.2 Compilation des méthodes . . . 152
IV.4.3 Un algorithme . . . 155
IV.4.4 Conclusion partielle . . . 159
IV.5 Verrouillage dans un graphe d'héritage . . . 160
IV.5.1 Une extension du protocole primitif . . . 160
IV.5.1.1 Rappel . . . 160
IV.5.1.2 Extension . . . 161
IV.5.1.3 Critique de cette extension . . . 164
IV.5.2 Le protocole le plus simple . . . 164
IV.5.2.1 Des vecteurs d'accès aux modes d'accès . . . 164
IV.5.2.2 Le protocole pour accéder aux instances . . . 165
IV.5.2.3 Modifications des méthodes . . . 167
IV.6 Avantages des vecteurs d'accès . . . 170
IV.6.1 Avantages . . . 170
IV.6.2 Propositions pour une implémentation . . . 173
IV.6.3 Comparaison avec des travaux antérieurs . . . 174
IV.7 Conclusion . . . 177

Conclusion 179
Résumé et conclusions des travaux . . . 181
Perspectives . . . 183

A Notations utilisées 185
A.1 Symboles mathématiques . . . 186
A.2 Notations sur les graphes . . . 187

B Calcul des vecteurs d'accès transitifs **189**
B.1 L'algorithme . . . 190
B.2 Preuve de la boucle interne . . . 192
B.2.1 Preuve de correction partielle . . . 192
B.2.2 Preuve de correction totale . . . 196
B.2.3 Complexité . . . 197
B.3 Idée de la preuve de la boucle externe . . . 200
B.4 Indications pour une implémentation efficace et incrémentale . . . 201
B.4.1 Efficacité . . . 201
B.4.2 Modifications incrémentales . . . 202

C Calcul des classes frontière **205**
C.1 L'algorithme . . . 206
C.2 Idée de la preuve . . . 208
C.3 Complexité . . . 209

Contribution aux problèmes de contrôle de concurrence et de reprise dans les bases de données à objets

Résumé

Le contrôle des accès concurrents à des données partagées et manipulées simultanément par plusieurs utilisateurs, ainsi que la reprise lors de pannes, sont des composantes essentielles de tout système de gestion de données actuel, notamment des bases de données.

Nous mettons en évidence les limites inhérentes au critère le plus important du domaine : la commutativité des opérations sur les types de données abstraits. Ces limites nous confortent dans l'utilisation de techniques relativement simples dans les deux travaux ultérieurs.

Ensuite, nous proposons une technique de contrôle de concurrence adaptée aux bases de données à objets car s'appuyant sur le graphe d'héritage. Cette proposition s'avère être une généralisation de la seule méthode implémentée : le protocole d'ORION.

Combinant les limites théoriques de la commutativité à la notion de méthodes, qui encapsulent le comportement des objets, nous proposons une technique simple, et dans une large mesure suffisante, pour déterminer la commutativité entre méthodes. Cette dernière proposition combine les avantages de la technique précédente à ceux obtenus, involontairement, par la décomposition en première forme normale dans les bases de données relationnelles.

Mots clés

Gestion transactionnelle, sérialisabilité, commutativité, contrôle de concurrence, reprise, bases de données à objets.

A contribution to concurrency control and recovery in object-oriented databases

Abstract

Controlling concurrent accesses to shared data manipulated simultaneously by several users, as well as recovering from rejects and failures, are essential components of any current data system, especially databases.

We highlight the inherent limits of the main criterion of the field : commutativity of operations on abstract data types. These limits convince us to rely on simple techniques, especially in the forthcoming works.

Next, we propose a concurrency control technique fitted to object-oriented databases since it uses the inheritance graph. This proposition generalizes the only one which, to our knowledge, has been implemented : the protocol of ORION.

Combining theoretical limits of commutativity with the notion of methods, which encapsulate the behaviour of objects, leads us to a simple, yet sufficient, technique for determining commutativity of methods. This last proposal combines the advantages of the previous proposition with the ones obtained, indirectly, by the first normal form constraint in relational databases.

Keywords

Transaction management, serializability, commutativity, concurrency control, recovery, object-oriented databases.

« — Qu'est-ce qu'une idée ?
— C'est une image qui se peint dans mon cerveau.
— Toutes vos pensées sont donc des images ?
*— Assurément ; car les idées les plus abstraites ne sont que les filles de tous les objets que j'ai aperçus. Je ne prononce le mot d'*être *en général que parce que j'ai connu des êtres particuliers. Je ne prononce le nom d'*infini *que parce que j'ai vu des bornes, et que je recule ces bornes dans mon entendement autant que je le puis ; je n'ai d'idées que parce que j'ai des images dans la tête.*
— Et quel est le peintre qui fait ce tableau ?
— Ce n'est pas moi, je ne suis pas assez bon dessinateur ; c'est celui qui m'a fait, qui fait mes idées.
— [...] *Il est bien triste d'avoir tant d'idées, et de ne savoir pas au juste la nature des idées. »*

(Voltaire, Le dictionnaire philosophique, Idée, 1765)

À ceux que j'aime.

Introduction

« *Above all, it is vital to recognize that completely guaranteed behavior is impossible and that there are inherent risks in relying on computer systems in critical environments. The unforeseen consequences are often the most disastrous.*[1] »

(Neumann, 1986)

[1] « Avant toute chose, il est fondamental de reconnaître qu'un comportement totalement garanti est illusoire et qu'il y a des risques intrinsèques à s'en remettre à des systèmes informatisés dans des environnements critiques. Les conséquences qui n'ont pas été anticipées sont souvent les plus désastreuses. »

CADRE DE L'ÉTUDE ET RÉSULTATS

Cadre de l'étude

On ne demande plus seulement aux systèmes actuels d'exécuter des tâches automatisées, mais de toujours les exécuter correctement, même s'il survient un événement inattendu et grave tel un atterrissage de tête de lecture sur disque magnétique, ou bien une coupure de courant, ou encore une erreur de programmation. En bref, on demande aux systèmes actuels d'assurer leur propre fiabilité et sécurité. Les précautions nécessaires peuvent s'envisager sur plusieurs plans : logiciel ou matériel, préventif ou curatif [Abbot 90].

Dans le domaine des bases de données, ce sont des objectifs importants qui se traduisent par au moins trois aspects [Delobel & Adiba 82] :

- *contraintes d'intégrité* : la base de données doit être maintenue dans un état consistant (respect des prédicats) sinon cohérent (reflet de l'état réel du monde) ;
- *autorisations d'accès* : seules des personnes autorisées doivent pouvoir manipuler certaines données ;
- *gestion transactionnelle* : la manipulation simultanée de données partagées par plusieurs utilisateurs doit être contrôlée afin de ne pas introduire d'incohérences dues à la concurrence. La gestion transactionnelle incorpore également la tolérance aux pannes.

Le domaine abordé en base de données est déjà plus restreint que celui énoncé dans le paragraphe introductif. Notre propre étude se limite encore à la gestion transactionnelle.

Une autre tendance de ses dernières années est la généralisation de la notion d'*objet* dans toutes les branches de l'informatique. Deux courants coexistent, qui utilisent les mêmes concepts dans des buts différents : l'intelligence artificielle et le génie logiciel.

Les problèmes de l'intelligence artificielle concernent essentiellement la modélisation des connaissances et des approches diverses ont vue le jour : « *frames* » et mécanismes de *réflexes* (si-besoin, si-modifié, etc.) comme dans YAFOOL, acteurs et continuation, etc. [Stefik & Bobrow 86] dresse un tableau des aspects de la programmation à objets dans le domaine de l'intelligence artificielle.

Le génie logiciel a tout d'abord introduit les *modules* puis les *types de données abstraits* dans un souci de clarté durant le développement de grands programmes (dizaines de milliers, voire millions, de lignes de code) [Appelbe & Ravn 84]. À partir de ces derniers ont été dérivés les *classes* [Wegner 90] par la prise en compte d'une relation d'*héritage*, une spécialisation entre objets. L'ancêtre Simula répondait à ce genre de préoccupations.

La gestion transactionnelle et l'approche à objets, au sens du génie logiciel, se sont retrouvées au sein d'une nouvelle génération de bases de données : les *bases de données à objets*. C'est dans ce cadre que s'élabore notre propos.

Résultats

Nos apports concernent le domaine de la gestion transactionnelle quand celle-ci fait appel aux objets typés, et donc, en particulier, quand elle s'applique aux bases de données à objets.

Notre première contribution est de montrer que l'emploi d'objets typés, et des relations de commutativité entre leurs opérations, au lieu d'objets non interprétés, c'est-à-dire munis seulement des deux opérations très générales que sont *Lire* et *Écrire* et de la relation de compatibilité, ne recule les limites de la concurrence que dans une faible mesure. Autrement dit, le critère de commutativité exhibe un comportement tout à fait analogue à celui de la plus simple compatibilité ; nous l'avons caractérisé et le discutons.

Ceci dit, nous montrons également que le critère de commutativité offre un intérêt certain pour la reprise dès lors que les exécutions sont soumises à la technique du verrouillage (ou, plus généralement, à toute méthode qui assure des exécutions strictes). Cet avantage, étranger à d'autres auteurs, leur fait soit se poser des questions maintenant dépourvues de sens, soit implémenter des systèmes effectuant des contrôles inutiles.

Ces limites étant fixées, notre deuxième contribution est de proposer un contrôle de concurrence adapté aux bases de données à objets. Nous y mettons en évidence une forme de commutativité qui, à la lumière des résultats précédents, est largement suffisante. Elle offre, néanmoins, de nombreux avantages : sa détermination est entièrement automatisée, le contrôle à l'exécution est aussi efficace que celui de la simple compatibilité et les opérations inverses n'ont pas à être fournies. Enfin, et surtout, elle se révèle nécessaire pour autoriser des exécutions concurrentes possibles dans les bases de données relationnelles.

Avant de présenter l'organisation générale de l'ouvrage, nous effectuerons dans les deux sections suivantes un rapide survol des concepts relatifs à la gestion transactionnelle puis aux bases de données à objets.

LA GESTION TRANSACTIONNELLE

Une transaction est un programme qui possède la propriété d'être une unité de consistance (en tout cas, supposée telle), c'est-à-dire qu'à partir d'un état consistant de la base de donnée, elle dérive un nouvel état consistant, quand elle est exécutée seule. Par une induction triviale, une suite de transactions exécutées séquentiellement sur une base de données initialement consistante aboutit nécessairement à un nouvel état consistant de cette base.

D'ailleurs, le critère d'exécution correcte d'un ensemble concurrent de transactions est généralement la *sérialisabilité* : une exécution concurrente des transactions doit être équivalente, du point de vue des effets sur la base de données et des résultats obtenus par les transactions, à une exécution série quelconque. Afin d'assurer cette propriété *syntaxique* sur les exécutions parallèles, on associe à une transaction quatre propriétés, normalisées par l'ISO (« *International Standardization Office* »), dites A. C. I. D. : atomicité, consistance, isolation et durabilité.

atomicité On paraphrase couramment la propriété d'atomicité en disant qu'une transaction a un effet « tout ou rien » : soit elle se termine normalement et tous ses effets doivent être validés dans la base, soit elle est rejetée, pour une raison quelconque, et aucun de ses effets partiels ne doit être visible de l'extérieur (autre transaction ou utilisateur). En aucun cas, il ne doit apparaître qu'une transaction ne s'est exécutée qu'en partie puisque la propriété de consistance d'une transaction n'est pas nécessairement assurée pour tout préfixe de son exécution.

consistance La propriété de consistance est supposée vérifiée *a priori* et son respect reste à la charge du programmeur. Les contraintes d'intégrité sont des mécanismes qui permettent de détecter à l'exécution, mais en partie seulement, les transactions qui ne la vérifieraient pas.

isolation De par le critère de sérialisabilité, une transaction ne doit pas apercevoir les effets partiels des transactions concurrentes ; seuls les effets issus de transactions validées (ou éventuellement précédentes dans l'ordre de sérialisation) doivent (ou peuvent) être pris en compte.

durabilité Tous les effets d'une transaction validée doivent persister, même en cas de panne matérielle.

Si le respect de la propriété de consistance d'une transaction est en grande partie à la charge du programmeur, celui des trois autres contraintes est assuré par le système grâce à trois com-

posantes qui ne sont pas totalement indépendantes : contrôle de concurrence (isolation), reprise (atomicité) et reprise sur panne (atomicité et durabilité).

contrôle de concurrence Le contrôle de concurrence est chargé d'empêcher les transactions d'utiliser de manière intempestive les données modifiées par les transactions concurrentes non encore validées. On distingue deux grandes familles de contrôle de concurrence : *continues* et *par certification*. Les méthodes continues, ou pessimistes, effectuent un contrôle lors de chaque accès à une donnée partagée afin de détecter les conflits *a priori*, tandis que les méthodes par certification, ou optimistes, attendent la fin de la transaction pour vérifier *a posteriori* si des conflits se sont produits. Dans les deux familles de méthodes, si les conflits entraînent que les transactions en cause ne peuvent pas être sérialisées alors il faut choisir une ou plusieurs victimes qui seront rejetées. Les effets engendrés sur la base par les victimes sont annulés avant que celles-ci ne puissent tenter de s'exécuter de nouveau.

Si le but essentiel du contrôle de concurrence est de garantir l'isolation, les rejets qu'il génère sont gérés par la composante suivante qui assure l'atomicité.

reprise Les effets des transactions rejetées doivent être défaits. Une transaction peut avoir été rejetée pour diverses raisons :

- son exécution a déclenché une erreur logique comme une division par zéro, la violation d'une contrainte d'intégrité ou une tentative d'accès à des données avec des droits insuffisants ;
- l'arrêt de l'exécution a été demandé par l'utilisateur qui avait lancé la transaction ;
- le rejet a été demandé par le contrôle de concurrence car la transaction ne pouvait plus être sérialisée avec les transactions concurrentes.

Défaire les opérations déjà exécutées par une transaction peut être vu comme l'exécution d'une transaction de compensation qui exécuterait des opérations inverses. On peut donc se poser la question de savoir si cette transaction de compensation doit être contrôlée comme les autres et si elle peut être rejetée à son tour.

reprise sur panne Il peut y avoir d'autres causes, plus radicales, au rejet des transactions, mais également à leur restauration, gérées par cette troisième composante. Elle prend ainsi en charge une partie de la contrainte d'atomicité mais surtout celle de durabilité.

Il peut survenir des événements qui entraînent l'arrêt immédiat du fonctionnement d'une machine : une erreur de parité mémoire, du processeur, une coupure d'alimentation, etc. Dans ce cas, au redémarrage du système, toutes les transactions en cours à l'instant de la panne sont à rejeter. Quant aux transactions validées, elles doivent être refaites, car il est possible que certains de leurs effets n'aient pas encore été transférés de la mémoire centrale vers la base de

données. Comme le contenu de la mémoire centrale est volatile, ces deux travaux ne peuvent être réalisés qu'à partir d'informations redondantes stockées sur un support de données stable, c'est-à-dire exempt de risques de pannes. La stabilité d'un tel support est actuellement réalisée par duplication de supports physiques[2] (disques « miroirs », ce qui permet de diminuer autant que l'on veut la probabilité d'une panne totale du système (sauf inondations catastrophiques ou autres séismes [Gray 78]). L'union de l'état courant de la base de données et de la mémoire stable doit toujours permettre de remettre la base dans son état validé le plus récent.

Il existe une autre panne matérielle grave qui est la destruction du support physique contenant la base de données (le plus souvent atterrissage de têtes de lecture/écriture sur disques magnétiques). Comme corollaire de ce genre d'accident, la mémoire stable doit contenir la totalité de l'information permettant de restaurer la base validée sur un nouveau support vierge. La mémoire stable n'étant pas infinie, ceci est assuré par archivages périodiques sur bandes magnétiques.

Partant du cadre très général visant à améliorer globalement et dans plusieurs directions la fiabilité des systèmes informatiques, nous nous sommes recentrer progressivement et seulement vers l'amélioration des performances transactionnelles, ce qui n'en diminue ni l'importance ni la difficulté. Voyons maintenant le cadre dans lequel nous allons devoir assurer cette gestion.

[2]Il existe également des mémoires dédiées, dites *Safe-RAM*, qui n'assurent toutefois qu'un rôle de tampon afin d'éviter le goulot d'étranglement dû aux temps d'entrées/sorties. À notre connaissance, elles ne sont pas encore utilisées dans les bases de données commerciales.

LES BASES DE DONNÉES À OBJETS

Les tous premiers travaux entrepris autour de cette thèse visaient à fournir un contrôle de concurrence adapté au système de gestion de base de données à objets O_2. Une partie de ceux-ci ont fait l'objet d'un rapport de recherche [Cart & Ferrié 89b] et d'une publication [Cart & Ferrié 90]. Ceci explique que les hypothèses que nous avons faites sur un modèle de base de données à objets soient très proches de celles du système O_2. Toutefois, les principes généraux sont assez largement respectés car ayant fait l'objet d'un manifeste [Atkinson et al. 89], même si de nouvelles directions sont proposées [Kim 90].

Nous allons rappeler les buts principaux des systèmes de gestion de bases de données, avant de préciser les spécificités, et avantages, des versions à objets.

Les buts des bases de données

Nous suivrons [Bancilhon 88] lorsqu'il soutient que le but des bases de données, en général, est de gérer des *quantités volumineuses* de données *persistantes*, *sûres* et *partagées*. Ces quatre points définissent le cadre, grossier, des recherches sur les bases de données.

quantités volumineuses de données Les quantités de données sont trop importantes pour tenir dans une mémoire centrale conventionnelle, et cette caractéristique devrait prendre de plus en plus d'importance tant les connaissances humaines d'une part, et les besoins de gestion informatisée d'autre part, entraînent une demande croissante [Stonebraker & Kemnitz 91].

persistance Ensuite, les données doivent persister, naturellement, d'une session de travail à la suivante, contrairement à ce qui se passe dans des applications de bureautique où les données doivent être sauvegardées dans des fichiers à la demande de l'utilisateur. Autrement dit, les demandes de sauvegardes sont remplacées par des demandes, explicites ou implicites, de suppressions.

sécurité Aussi, les données doivent être sûres, c'est-à-dire ne pas être détruites à cause d'une erreur, aussi bien matérielle que logicielle.

partage Enfin, les données doivent être partagées de manière contrôlée lorsque plusieurs utilisateurs travaillent simultanément.

D'autres aspects jouent un rôle important dans la conception des bases de données ;. Les problèmes essentiels qui ont surgis récemment sont liés à l'apparition de nouvelles applications pour lesquelles l'utilisation de bases de données relationnelles pose plus de problèmes d'implémentation qu'elle ne permet de résoudre l'application elle-même. Ces problèmes sont : l'« *impedance mismatch* », le développement des *interfaces* avec l'utilisateur et la *productivité* du programmeur.

« *impedance mismatch* » Il y a de réelles difficultés à coupler une base de données relationnelle fournissant les quatre caractéristiques énoncées ci-dessus, mais dont le langage d'interrogation n'est pas complet, à un langage traditionnel (Pascal, C, COBOL, etc.) qui implémente vraiment l'application.

Ce problème essentiel a donné lieu à des fusions entre le domaine des bases de données et la programmation logique ainsi qu'entre bases de données et programmation à objets. L'approche à objets se révèle particulièrement attrayante puisque le paradigme des objets regroupe dans une seule entité la notion de donnée et celle de procédure.

interfaces L'interface avec l'utilisateur est l'une des partie d'une application la plus délicate et la plus difficile à programmer. Il faut réunir les ingrédients de clarté, intuition, facilité, rapidité. Or, la programmation à objets s'est montrée particulièrement efficace pour développer des interfaces conviviales (cf. [Plateau et al. 89]).

productivité Si la productivité des programmeurs est accrue par l'utilisation de la programmation à objets pour le développement d'interfaces, elle l'est également pour le développement du reste des applications. Ce dernier point reste toutefois du domaine de la recherche pour ce qui est du « *programming on the very large* » [Meyer 88].

Les concepts « objet »

Les principaux concepts de l'approche à objets, sur lesquels les chercheurs se rejoignent, sont l'*encapsulation*, l'*identité d'objet*, les *classes*, l'*héritage*, la *surcharge* et la *liaison dynamique*.

encapsulation Les notions de données et de procédures sont regroupées au sein d'une entité portant le nom d'objet. Un objet est composé d'un ensemble de données, invisibles depuis l'extérieur. Chaque objet dispose d'une interface composée d'un ensemble de procédures, seul moyen de manipuler les données de cet objet [Snyder 86].

identité d'objet Chaque objet possède une identité propre, distincte de sa valeur [Khoshafian & Copeland 86]. Cette notion d'identité doit être artificiellement introduite dans les bases de données relationnelles (encore que ORACLE l'incorpore en partie).

La notion d'identité d'objet entraîne l'existence de deux relations d'égalité : l'égalité des identités et celle des valeurs.

classe La notion de classe bien que voisine de celle de type de données abstrait, regroupe à la fois celle de moule (pouvant être un type proprement dit), à partir duquel sont créées les instances (les variables), et celle d'extension, c'est-à-dire que la classe connaît l'ensemble de ses instances, jouant ainsi un rôle analogue à celui de la relation dans le modèle relationnel.

Illustrons la différence entre classe et type. Les objets de type RECTANGLE peuvent être utiles à plusieurs personnes dans une base de données dédiée à la CAO. Toutefois, parler de l'ensemble des rectangles existant dans le système n'a pas grand sens. En revanche, connaître l'extension de la classe COMPOSANTÉLECTRONIQUE semble naturel.

Dans O_2, à la création d'une nouvelle classe, une clause « *with extension* » peut être spécifiée. Le système gère alors automatiquement l'insertion des instances créées dans un objet ensembliste de même nom que la classe.

héritage L'héritage est l'aspect fondamental de l'approche à objets [Wegner 90]. L'héritage permet à des objets proches les uns des autres par leur structure, de partager les opérations manipulant leurs parties communes. Ceci rejoint un principe très général qui énonce qu'il faut effectuer les tâches au niveau où elles sont le plus appropriées.[3]

Par exemple, il est gênant d'écrire séparément pour des employés et des étudiants des opérations communes aux deux catégories ; il suffit de les coder dans la classe plus générale des personnes.

On distingue au moins trois formes d'héritage : *substitution*, *inclusion* et *contrainte* [Atkinson et al. 89].

L'héritage par substitution apparaît dans les systèmes où les objets héritent à partir du moment où ils offrent toutes les opérations fournies par leurs super-classes et en ajoutent. L'héritage n'est basé que sur le comportement observable, c'est-à-dire les propriétés fonctionnelles, et non sur la structure de l'objet. C'est le cas de POOL-I, par exemple [America & Van der Linder 90].

L'héritage par inclusion est quant à lui basé sur la structure des objets. Une sous-classe hérite d'une super-classe en ajoutant de nouveaux champs et/ou de nouvelles méthodes.

L'héritage par contrainte est une forme particulière d'héritage par inclusion où une classe hérite d'une super-classe sans ajouter champs ou méthodes, mais en contraignant les valeurs de

[3] le fameux principe de subsidiarité du traité de Maastricht ou du nouveau catéchisme !

certains champs. Par exemple, les classes HOMME et FEMME héritent de Personne en restreignant la valeur du champ « *Sexe* » à « *Masculin* » et « *Féminin* » respectivement.

surcharge et liaison dynamique Il existe des cas dans lesquels des opérations de même sémantique (de même nom) ne doivent pas s'exécuter de la même façon. Par exemple, une figure géométrique possède une opération pour se dessiner sur un écran graphique. Un carré ou un cercle sont des figures géométriques particulières qui doivent donc pouvoir s'afficher. Toutefois, le code de l'opération est spécifique à chaque classe d'objet graphique.

La surcharge alliée à la liaison dynamique sont des solutions élégantes pour résoudre le problème de l'utilisation polymorphe d'ensembles d'objets, par exemple, afficher un plan d'architecte composé d'objets graphiques différents. [Meyer 86] présente très concrètement les problèmes qu'entraînent l'absence d'héritage et de liaison dynamique avec le langage Ada qui n'offre que la surcharge et l'encapsulation.

Autour de ces concepts de base, de nombreuses variations ont vu le jour, non seulement dans la façon de les implémenter, mais également sur les champs d'application. [Wegner 90] effectue un tour d'horizon particulièrement vaste (81 pages !).

Les bases de données à objets

Les nouvelles applications, telles que la bureautique, la CAO [Barthès 88], la cartographie, etc., nécessitent des fonctionnalités de bases de données car les quantités d'informations à manipuler sont volumineuses, doivent persister, être manipulées par plusieurs utilisateurs simultanément et ne pas subir de dégradations. Ces données sont essentiellement des objets dits complexes, dans le sens où ils sont fortement structurés, c'est-à-dire composés de sous-objets, et cependant manipulés comme un tout.

L'emploi d'une base de données relationnelle se révèle alors inadéquat, à cause de l'obligation de représenter les données sous première forme normale. Le processus de normalisation entraîne les désagréments suivants [Chrisment & Zurfluh 89] :

- création d'un nombre important, et souvent même excessif, de relations décomposant inutilement les objets complexes ;
- d'où, il faut pratiquer de nombreuses jointures pour évaluer une requête portant sur un seul objet.

Il y a donc un gouffre entre la notion d'objets complexes et la représentation de ceux-ci dans la base. Des extensions du modèle relationnel tentent d'améliorer les capacités de stockage de ces derniers, comme le modèle relationnel NF^2 (pour « *Non First Normal Form* ») [Pistor & Andersen 86]. D'autres extensions du modèle relationnel tentent de résoudre les manques de ce dernier en introduisant la notion de champs procéduraux [Stonebraker et al. 87], ou même de classes comme citoyens de second ordre [Exertier 91].

Toutefois, fusionner bases de données et approche à objets a été la voie favorisée pour offrir aux nouvelles applications les avantages des bases de données ainsi que la structuration en objets complexes. En effet, ces dernières combinent la complétude des langages de programmation aux fonctions d'une base de données [Bloom & Zdonik 87] [Bertino & Martino 91]. Notons tout de même que de nouvelles extensions visent à réintroduire la notion de relation dans les modèles à objets [Rumbaugh 87] [Andrews & Harris 87] [Hwang & Lee 91].

Si les recherches dans le cadre des bases de données à objets tentent de concilier ces deux approches, elles poursuivent également d'autres buts. Quelques « règles d'or » ont été énoncées [Atkinson et al. 89]. Certaines sont présentées comme obligatoires : *objets complexes*, *identité d'objet*, *encapsulation*, types ou *classes*, *héritage* de types ou classes, *surcharge* et *liaison dynamique*, *complétude*, *extensibilité*, *persistance*, gestion de la *mémoire secondaire*, *contrôle de concurrence*, *reprise*. Détaillons les caractéristiques qui n'ont pas encore été décrites.

objets complexes Les objets doivent pouvoir se (dé-)composer à l'aide de constructeurs, au minimum les constructeurs ensemble, liste et n-uplet.

Similairement à la double égalité liée à l'identité d'objet, la notion d'objet complexe entraîne celle de copie superficielle (seul l'objet racine de l'objet complexe est dupliqué) ou de copie profonde (tous les objets composant l'objet complexe sont récursivement dupliqués, et il faut donc faire attention à d'éventuels circuits). De plus, à l'égalité de l'identité ou de la valeur d'un objet, s'ajoute celle d'égalité profonde, de manière similaire à la copie profonde.

complétude SQL, le langage d'interrogation des bases de données relationnelles, n'est pas complet, dans le sens où il ne permet pas d'exprimer tous les calculs ; il lui manque la récursivité ou bien un opérateur de fermeture transitive. Les langages des bases de données à objets doivent être complets.

extensibilité De nouveaux types ou classes doivent pouvoir être ajoutés au système sans que l'on puisse faire de différence avec ceux pré-définis. Ceci ne concerne pas les constructeurs de types (ensemble, liste, n-uplet, etc.).

persistance La persistance des objets doit être complètement indépendante de leur nature ou de la façon dont ils sont utilisés. Le modèle de persistance privilégié dans les bases de données à objets est celui de l'*atteignabilité* : toute instance atteignable à partir de racines nommément désignées comme persistantes est elle-même persistante.

mémoire secondaire La mémoire secondaire doit être gérée de manière tout à fait transparente pour le programmeur d'application. La gestion d'index, le regroupement d'objets sur

disque et autres aspects de la gestion des échanges entre la mémoire secondaire et la mémoire centrale doivent rester à l'intérieur d'un noyau.

Les aspects qui nous intéressent plus particulièrement sont bien évidemment le contrôle de concurrence et la reprise. Cet exposé aura seulement eu pour but de montrer que le cadre de travail dans lequel nous allons faire nos propositions est loin d'être figé, d'où la difficulté de dresser des hypothèses de travail. Nous finirons d'ailleurs cette thèse sur cet aspect-là. Les hypothèses, aussi minimales que possibles, que nous ferons sur le modèle de données seront exposées non pas en bloc dans un chapitre spécifique, mais réparties dans les introductions des chapitres III et IV, en fonction de leurs besoins respectifs, quitte à être redondant.

ORGANISATION DE L'OUVRAGE

Avant de présenter sommairement le plan de notre thèse, faisons un rapide rappel historique.

Ces travaux ont été menés en collaboration avec M. Malta Carmelo. Un prototype de système gérant des types de données abstraits concurrents et recouvrables, utilisant le critère de commutativité des opérations comme propriété locale d'atomicité et le verrouillage à deux phases strict pour maintenir la sérialisabilité des transactions, a été implémenté. Une partie du prototype est décrite dans [Malta & Martinez 91b]. Nous avons mis en pratique des idées intuitives qui nous semblaient intéressantes. La formalisation et la validation théorique de ses idées n'ont été faites que plus tard. Elles ont débouchées sur un résultat ressenti également de façon intuitive : le critère de commutativité s'avère, *pour des objets arbitraires*, d'un bien faible intérêt [Malta & Martinez 92a]. Ceci nous a amené à étendre le prototype en intégrant, au-delà d'un moyen de description fin de la commutativité entre opérations, une alternative automatisée et suffisante pour les types de données abstraits ayant la forme de n-uplets [Malta & Martinez 92c], les plus couramment décrits dans une application quelconque. Tous les travaux, comparaisons et conclusions concernant le prototype sont exposés dans [Malta 93].

D'autre part, durant le stage de D. E. A., nous avions commencé des travaux sur le contrôle de concurrence dans les bases de données à objets, destinés en partie à la base de données O_2. Les premiers résultats [Aldebert & Martinez 89] ont été généralisés dans [Malta & Martinez 90] [Malta & Martinez 91a]. La notion de classe étant très proche de celle de type de données abstrait, l'application précédente aux n-uplets a été étendue aux systèmes à objets. La prise en compte de la commutativité entre méthodes [Malta & Martinez 93] se heurte aux problèmes supplémentaires de l'héritage, de la surcharge des méthodes et de la liaison dynamique.

Notre exposé se décompose en quatre chapitres.

Le chapitre I est un état de l'art [Malta & Martinez 92b] qui ne se veut absolument pas exhaustif mais qui présente les résultats du domaine sur lesquelles nous nous appuyons. Nous traitons de l'utilisation des types de données abstraits et de leurs opérations au lieu d'objets non interprétés et des seuls accès en lecture et écriture. Les résultats classiques de la sérialisabilité demeurent acquis après cette extension. Nous présentons ensuite les deux techniques les plus connues de contrôle de concurrence et de reprise, à savoir le verrouillage et la journalisation. Ce chapitre se termine par le passage du modèle des transactions plates au modèle des transactions multi-niveaux.

Le chapitre II est purement théorique, bien qu'agrémenté de multiples exemples, et souligne les limites d'un critère qui a connu son heure de gloire : la commutativité. Il répond aussi à la question que se sont posés certains auteurs quant à la normalité des opérations inverses. Ce chapitre discute plus profondément les résultats de [Malta & Martinez 92a] et ajoute une comparaison avec les travaux de [Weihl 89a].

Le chapitre III présente une méthode de contrôle de concurrence adaptée aux bases de données à objets car exploitant le graphe d'héritage. Il reprend [Aldebert & Martinez 89], [Malta & Martinez 90] et [Malta & Martinez 91a] et ajoute une comparaison avec, d'une part, le protocole d'ORION [Garza & Kim 88] dont il se révèle être, dans une très large mesure, une généralisation, et, d'autre part, avec la proposition pour O_2 de [Cart & Ferrié 90].

Le chapitre IV poursuit les deux chapitres précédents. Il s'appuie sur les travaux de [Malta & Martinez 92c] et les adapte à l'approche à objets en tenant compte des trois caractéristiques supplémentaires que sont l'héritage, la surcharge et la liaison dynamique. Une comparaison avec le contrôle de concurrence dans les bases de données relationnelles met en valeur son importance.

En conclusion, nous nous livrons à un bilan de nos travaux et y exposons des voies de recherche, expérimentales ou théoriques, qui nous semblent prometteuses.

Le lecteur qui est immédiatement intéressé par les résultats et conclusions de ces travaux peut se rendre directement à la conclusion où il trouvera un résumé général.

Chapitre I

Un état de l'art

Résumé du chapitre

La théorie de la *sérialisabilité* s'est enrichie d'un développement récent et commun à toutes les branches de l'informatique : la notion de *types de données abstraits*. La prise en compte de ceux-ci dans le domaine transactionnel permet d'accroître la concurrence entre opérations en considérant la sémantique des opérations, qui ne sont plus de simples lectures ou écritures. Cette sémantique se traduit par le remplacement du critère de *compatibilité* par celui de *commutativité* des opérations. Nous ajoutons à cette présentation un critère plus faible : la *recouvrabilité relative*.

Nous sommes ensuite beaucoup plus pratique puisque nous décrivons les deux techniques les plus employées pour assurer la gestion transactionnelle : *verrouillage* et *journalisation*.

Nous finissons ce très bref état de l'art en étendant aux *transactions multi-niveaux* la sérialisabilité, le contrôle de concurrence et la reprise, préalablement présentés dans le cadre des transactions plates.

Mots clés

Transactions, sérialisabilité, commutativité, recouvrabilité relative, contrôle de concurrence, verrouillage à deux phases strict, reprise, journalisation, transactions multi-niveaux.

I.1 INTRODUCTION

Les systèmes transactionnels doivent assurer deux fonctions importantes : le *contrôle de concurrence* et la *reprise*. Le but du contrôle de concurrence est d'empêcher (ou de gérer) les interférences qui ne peuvent manquer de se produire lorsque plusieurs utilisateurs manipulent simultanément des données partagées. Il assure l'isolation (I de ACID) des transactions. Le but de la reprise est d'assurer que les transactions sont effectuées intégralement ou pas du tout. Elle assure l'atomicité et la durabilité (A et D de ACID) des transactions.

Contrôle de concurrence et reprise ont fait l'objet d'une attention particulière dans le domaine des bases de données. Toutefois et de plus en plus, ces préoccupations s'introduisent dans les systèmes d'exploitation [Kumar & Stonebraker 89], les systèmes de communications [Bernstein et al. 90] ou les systèmes temps-réel [Huang et al. 91] [Ulusoy & Belford 91].

Dans ce chapitre, nous présentons la théorie, maintenant classique, de la *sérialisabilité*. ous rappelons ses résultats dans le cadre des transactions plates. Nous étendons les notations pour prendre en compte la *commutativité* des opérations typées au lieu de la plus ancienne *compatibilité* des seules opérations *Lire* et *Écrire*. Nous avons également intégré le critère plus récent de *recouvrabilité relative* [Badrinath & Ramamritham 87].

Une seconde partie présentera deux techniques pratiques de mise en œuvre du contrôle de concurrence et de reprise : le *verrouillage* et la *journalisation*.

Enfin, dans une dernière partie, nous généralisons tous les résultats au modèle des *transactions multi-niveaux* [Beeri et al. 88].

I.2 THÉORIE DE LA SÉRIALISABILITÉ

Les résultats fondamentaux de la théorie de la sérialisabilité ne sont pas remis en cause par l'introduction des types de données abstraits (ADT pour « *Abstract Data Type* »). En effet, les opérations typées permettent d'exploiter la commutativité de celles-ci, alors que les seules opérations *Lire* et *Écrire* ne permettent d'exploiter que leur compatibilité. La différence de termes est trompeuse car ils recouvrent la même propriété. Nous conserverons cependant ces deux termes pour souligner l'évolution.[1]

I.2.1 Modèle de base de données

Notre modèle de base de données est constitué d'un ensemble d'objets, instances d'ADT. L'exploitation des spécifications attachées aux opérations d'un ADT autorise une plus grande concurrence. Toutefois, afin d'être exploités dans un cadre transactionnel, les ADT doivent posséder quelques caractéristiques additionnelles : atomicité et, parfois, existence d'opérations inverses. De plus, pour exploiter le parallélisme latent sur les ADT, nous définissons trois relations entre couples d'opérations : commutativité, recouvrabilité relative et dépendance.

I.2.1.1 Types de données abstraits transactionnels

Définition I.1 (Type de données abstrait) *Un type de données abstrait (ADT) est un couple formé d'un ensemble de données et d'un ensemble d'opérations. Il décrit une structure de données ainsi que les seules opérations applicables sur cette structure.*

Les types de données abstraits implémentent un concept essentiel de la programmation moderne : l'*encapsulation*. Il est loisible de changer la structure d'un ADT indépendamment de ses spécifications et de son interface. Ainsi, la nouvelle version d'un même ADT pourra être utilisée sans avoir à modifier les programmes existants qui utilisaient son prédécesseur.

Exemple I.1 *Les cours de structures de données contiennent plusieurs ADT classiques : piles, files, arbres binaires, B-arbres, etc. Les ADT* PILE *et* FILE *sont notoirement connus pour posséder deux implémentations : une organisation séquentielle et une organisation chaînée. La première est plus simple à implémenter, alors que la seconde offre l'avantage de ne pas borner, autrement que par la place mémoire disponible, le nombre d'éléments que peut contenir une*

[1] Cette distinction nous est parfaitement propre puisque, suivant les auteurs, la signification de ces deux termes varie.

*pile ou une file. Si les instances de ces ADT ne sont accessibles que par l'intermédiaire de leurs opérations (*Mettre, Retirer, Premier, Plein, Vide, Vider*) alors la structure sous-jacente peut être modifiée à tout moment.*

La notion d'ADT se généralise à n'importe quelle structure.

Définissons un ADT COMPTEENBANQUE*. Sa structure sera un n-uplet composé du nom du détenteur du compte, d'un numéro de compte, d'un historique des dernières opérations effectuées, d'un débit et d'un crédit. Les opérations qui permettront de manipuler un compte en banque seront, entre autres,* Débiter, Créditer *et* Solde*. Une implémentation possible est donnée avec l'algorithme I.1.*

L'exploitation des ADT dans les systèmes transactionnels nécessite l'introduction de deux contraintes supplémentaires : l'*atomicité* de leur exécution et la disponibilité d'*opérations inverses*.

Définition I.2 (Opérations atomiques sur un ADT transactionnel) *Les opérations sur un ADT à usage transactionnel doivent être* atomiques*.*

Une opération est atomique s'il n'est pas possible de détailler les étapes de son exécution. Par conséquent, si deux opérations sont demandées « simultanément » sur une instance commune, les effets de l'exécution concurrente doivent toujours être équivalents à l'une des deux exécutions série possibles.

Ce pré-requis peut être obtenu soit par l'*indivisibilité* de l'exécution de chaque opération, une contrainte plus forte que l'atomicité, soit par l'utilisation des transactions multi-niveaux qui font l'objet de la section I.5.

Définition I.3 (Opérations inverses sur un ADT transactionnel) *Les opérations sur un ADT à usage transactionnel doivent parfois disposer d'opérations inverses.*

Cette contrainte assure que tous les effets d'une transaction pourront être défaits en cas de rejet si *le modèle de transaction a adopté la technique des mises-à-jour immédiates.*

Notez qu'une même opération peut avoir différentes opérations inverses qui dépendent des paramètres d'entrée, et surtout des paramètres de sortie.

Exemple I.2 *L'opération* Multiplier *sur l'ADT* ENTIER *possède trois inverses qui varient avec le paramètre d'entrée, le multiplicateur. Si le multiplicateur vaut zéro, alors l'opération inverse est une écriture aveugle qui devra recopier l'ancienne valeur de l'instance car l'opération directe est elle-même une mise à zéro aveugle. Si le multiplicateur vaut un, alors la multiplication se ramène à l'opération identité, ainsi que son inverse. Autrement, il s'agit d'une multiplication effective dont l'opération inverse sera une division.*

Algorithme I.1 Le type de données abstrait COMPTEENBANQUE

```
abstract data type COMPTEENBANQUE is
structure
    Nom : string ;
    NuméroCompte : string ;
    Débit : real ;
    Crédit : real ;
    DernièresOpérations : list of OPÉRATIONBANCAIRE ;
operations
    Débiter (Montant : real ; Intitulé : string) return boolean is
        var
            Op : OPÉRATIONBANCAIRE ;
        begin
            if Montant > Solde
            then return false
            else
                begin
                    new Op init Intitulé, Montant ;
                    DernièresOpérations.InsérerDernier(Op) ;
                    Débit := Débit + Montant ;
                    return true
                end
            end ;
        Créditer (Montant : real ; Intitulé : string) is
            var
                Op : OpérationBancaire ;
            begin
                new Op init Intitulé, Montant ;
                DernièresOpérations.InsérerDernier(Op) ;
                Crédit := Crédit + Montant
            end ;
        Solde return real is
            begin
                return Crédit - Débit ;
            end ;
        EditerRelevéBancaire is
            ...
    end ;
```

L'opération Insérer sur l'ADT ENSEMBLE *ne peut quant à elle se contenter des seuls paramètres d'entrée pour déterminer l'opération inverse. Ce n'est que lorsque l'opération directe a été exécutée que l'on sait si l'insertion a été effective ou si elle était inutile (l'élément à insérer se trouvant déjà dans l'ensemble). On pourra alors associer comme opération inverse soit l'identité, soit Supprimer.*

I.2.1.2 Relations entre opérations

L'exploitation des ADT permet d'échapper aux seules opérations *Lire* et *Écrire* s'appliquant indifféremment sur tout type de données. Le critère de compatibilité est rebaptisé commutativité sur les opérations d'un ADT.

La commutativité est une relation liant deux opérations d'un même ADT si, et seulement si, elles sont complètement indépendantes l'une de l'autre, dans un sens que nous allons préciser.

Deux autres relations entre couples d'opérations sont présentées : recouvrabilité relative et dépendance. Contrairement à la commutativité, ces relations sont ordonnées. La recouvrabilité relative est un critère plus faible que la commutativité. Elle impose un ordre de validation des transactions et requiert une gestion plus lourde des rejets. La dépendance, en étant la plus permissive de toutes ces relations, se heurte au problème des rejets en cascade, intolérables pour certaines applications. Avant d'introduire ces relations, nous donnons une formalisation des spécifications associées aux opérations sur un ADT.

Spécifications d'un ADT Nous reprenons la formalisation des spécifications des opérations sur un ADT de [Goodman & Shasha 85].

Définition I.4 (Spécifications d'un ADT) *Soit A un ADT admettant un ensemble OP d'opérations et un ensemble S des valeurs possibles des instances de A, alors nous définissons les spécifications de A comme un quintuplet :*

$$\textit{spécifications}(A) = S \times OP \times P_e \times P_s \times S$$

où :

- *P_e représente les paramètres d'entrée des opérations ;*
- *P_s représente les paramètres de sortie.*

La première valeur d'une instance représente la valeur initiale, tandis que la seconde valeur est celle obtenue après application, hors concurrence, de l'opération.

Commutativité La commutativité est le principal critère exploité dans la littérature sur les systèmes transactionnels.

Définition I.5 (Commutativité des opérations d'un ADT) *Soient deux opérations op_1 et op_2 appartenant à l'interface d'un même ADT, alors nous définissons le prédicat commute(op_1, op_2) qui est vrai si, et seulement si, les opérations op_1 et op_2 appliquées à n'importe quelle valeur*[2] *d'une instance de cet ADT, dans les deux ordres d'exécution possible, avec les mêmes paramètres d'entrée, aboutissent au même état final de l'instance et renvoient des paramètres de sortie identiques.*

Plus formellement, soient op_1 et op_2 deux opérations sur un même ADT A, alors :

$$\begin{array}{l} \textit{commute}(op_1, op_2) \\ \leftrightarrow \\ \forall s \in S, \\ \quad \left\{ \begin{array}{l} (s, op_1, pe_1, ps_1, s_1), (s_1, op_2, pe_2, ps_2, s'), \\ (s, op_2, pe_2, ps_2, s_2), (s_2, op_1, pe_1, ps_1, s') \end{array} \right\} \subseteq \textit{spécifications}(A). \end{array}$$

La valeur du prédicat peut tirer profit des valeurs des paramètres d'entrée et de sortie, voire même de l'état. On parle alors de *commutativité conditionnelle*.

[Weihl 88] introduit deux formes de commutativité conditionnelle sur les ADT : la commutativité en arrière et la commutativité en avant. Les deux formes vérifient la définition I.5. Ce qui les différencie est l'hypothèse qui est faite sur la valeur à partir de laquelle les opérations sont appliquées.

Pour la commutativité en arrière, la seconde opération s'applique sur l'état de l'instance obtenu après application de la première. On dispose donc de $(s, op_1, pe_1, ps_1, s_1)$ et $(s_1, op_2, pe_2, ps_2, s')$.

Pour la commutativité en avant, les deux opérations se font à partir de la dernière valeur validée. On dispose donc de $(s, op_1, pe_1, ps_1, s_1)$ et $(s, op_2, pe_2, ps_2, s_2)$.

En pratique, la commutativité en arrière s'emploie dans un modèle de transactions avec mises-à-jour immédiates, tandis que la commutativité en avant est exploitée dans un modèle de transactions avec mises-à-jour différées jusqu'à la validation. La différenciation entre ces deux formes de commutativité s'impose du fait que les paramètres de sortie informent sur l'état de l'objet. Cette différenciation est par conséquent impossible à faire si la commutativité n'exploite que les paramètres d'entrée et a fortiori aucun paramètre.

Comme on peut s'en rendre compte à partir des tables I.1 et I.3, les relations de commutativité en avant et en arrière sont incomparables. [Nakajima & Tokoro 90] propose un protocole multi-version afin d'exploiter conjointement ces deux formes de commutativité. Cela revient à exploiter l'historique des opérations non validées sur l'instance. De nombreuses autres propositions ont été faites autour de cet ADT particulier [O'Neil 86] [Ng 89] [Broessler & Freisleben

[2] Une opération qui n'est pas définie dans un état donné de l'ADT ne modifie pas cet état et renvoie un paramètre qui signale l'erreur ou, mieux, déclenche une exception.

(1) \ (2)	$D\acute{e}bit_{ok}^{m_2}$	$D\acute{e}bit_{\neg ok}^{m_2}$	$Crédit^{m_2}$	$Solde_{r2}$
$D\acute{e}bit_{ok}^{m_1}$	*oui*	non	non	non
$D\acute{e}bit_{\neg ok}^{m_1}$	non	*oui*	non	*oui*
$Crédit^{m_1}$	non	non	*oui*	non
$Solde_{r1}$	non	*oui*	non	*oui*

TAB. I.1 – Relation de commutativité en arrière de l'ADT COMPTEENBANQUE

89].

[Weihl 89b] récidive en proposant la commutativité en arrière *à droite*. Cette dernière n'est plus symétrique. Elle lève l'hypothèse faite par la commutativité en arrière sur l'état initial de l'objet. L'exemple I.3 illustre les différences entre ces formes de commutativité sur l'ADT COMPTEENBANQUE.

Exemple I.3 *Soit l'ADT* COMPTEENBANQUE *de l'exemple I.1. La relation de commutativité en arrière est donnée en table I.1, où la première opération exécutée est celle donnée en ligne. On supposera que les paramètres d'entrée* m_1 *et* m_2 *sont strictement positifs. Les paramètres de sortie de l'opération Débit ont donné lieu à deux entrées dans les tables afin d'éviter les expressions conditionnelles et ainsi faciliter les comparaisons. De manière générale, les paramètres d'entrée sont écrits en exposants et les paramètres de sortie en indices.*

La non-commutativité d'une opération de débit réussie avec une opération de débit ratée mérite une explication. Supposons que l'on exécute la séquence « Débit$_{ok}^{8}$ *; Débit*$_{\neg ok}^{20}$ *», alors la valeur initiale se trouvait dans l'intervalle* $[8, 28[$*. Exécutée dans l'autre ordre, les résultats ne seront pas les mêmes dans le cas où la valeur initiale se trouverait dans l'intervalle* $[20, 28[$*.*

L'exécution dans l'ordre opposé nous aurait permis de conclure que la valeur initiale se trouvait bien dans la différence des deux premiers intervalles, c'est-à-dire $[8, 20[$*. Mais, comme les deux compositions ne sont pas définies sur le même domaine, les opérations ne commutent pas.*

La commutativité en arrière à droite consiste tout simplement à ne plus faire le raisonnement précédent et à considérer qu'une estimation de l'état initial est effectivement connue d'après les résultats des opérations. Dans ce cas, exécuter les opérations dans des ordres différents conduit à différencier les états initiaux possibles et donc à autoriser certains ordres. Sur l'exemple discuté ci-dessus, « Débit$_{ok}^{8}$ *; Débit*$_{\neg ok}^{20}$ *» ne sera toujours pas autorisé, alors que « Débit*$_{\neg ok}^{20}$ *; Débit*$_{ok}^{8}$ *» est accepté par la commutativité en arrière à droite. La table I.2 se révèle donc asymétrique et plus permissive que la table I.1.*

Finalement, La table I.3 donne la relation de commutativité en avant pour les opérations sur l'ADT COMPTEENBANQUE*. Il n'y a pas d'ordre d'exécution privilégié entre les opérations puisqu'elles sont toutes deux appliquées à partir du même état initial de l'instance.*

On vérifie bien que commutativités en arrière et commutativité en avant sont incomparables.

(1) \ (2)	$D\acute{e}bit_{ok}^{m_2}$	$D\acute{e}bit_{\neg ok}^{m_2}$	$Cr\acute{e}dit^{m_2}$	$Solde_{r2}$
$D\acute{e}bit_{ok}^{m_1}$	*oui*	non	*oui*	non
$D\acute{e}bit_{\neg ok}^{m_1}$	*oui*	*oui*	non	*oui*
$Cr\acute{e}dit^{m_1}$	non	*oui*	*oui*	non
$Solde_{r1}$	non	*oui*	non	*oui*

TAB. I.2 – Relation de commutativité en arrière à droite de l'ADT COMPTEENBANQUE

(1) \ (2)	$D\acute{e}bit_{ok}^{m_2}$	$D\acute{e}bit_{\neg ok}^{m_1}$	$Cr\acute{e}dit^{m_2}$	$Solde_{r2}$
$D\acute{e}bit_{ok}^{m_1}$	non	*oui*	*oui*	non
$D\acute{e}bit_{\neg ok}^{m_1}$	*oui*	*oui*	non	*oui*
$Cr\acute{e}dit^{m_1}$	*oui*	non	*oui*	non
$Solde_{r1}$	non	*oui*	non	*oui*

TAB. I.3 – Relation de commutativité en avant de l'ADT COMPTEENBANQUE

Nous reverrons ces formes de commutativité à la lumières des résultats du chapitre II.

Dépendances de recouvrabilité relative La recouvrabilité relative a été introduite par [Badrinath & Ramamritham 87] [Badrinath & Ramamritham 92]. Contrairement à la commutativité, excepté la commutativité en arrière à droite, l'ordre d'exécution est déterminant.

Définition I.6 (Recouvrabilité relative des opérations d'un ADT) *Soient deux opérations op_1 et op_2 d'un même ADT, exécutées dans cet ordre, alors op_2 est relativement recouvrable par rapport à op_1, noté relativement_recouvrable(op_2, op_1), si les paramètres de sortie de op_2 ne dépendent pas de l'exécution préalable de op_1 sur l'instance partagée.*

Plus formellement, soient op_1 et op_2 deux opérations sur un même ADT A, alors :

$$\begin{array}{l} \textit{relativement\_recouvrable}(op_1, op_2) \\ \leftrightarrow \\ \forall s \in S, \\ \quad \left\{ \begin{array}{l} (s, op_1, pe_1, ps_1, s_1), (s_1, op_2, pe_2, ps_2, s'), \\ (s, op_2, pe_2, ps_2, s_2) \end{array} \right\} \subseteq \textit{spécifications}(A). \end{array}$$

Remarquons qu'il existe deux cas limites. Premièrement, les opérations qui ne renvoient pas de paramètres de sortie sont relativement recouvrables par rapport à toute autre. Deuxièmement, toute opération est relativement recouvrable par rapport à une opération qui ne modifie pas l'état de l'objet. Nous appellerons cette seconde forme « *recouvrabilité simple* » car elle n'a aucun effet sur la reprise, ce qui n'est pas le cas de la forme générale.

Notez que la recouvrabilité relative peut, tout comme la commutativité, être conditionnelle.

(1) \ (2)	$D\acute{e}bit_{ok}^{m_2}$	$D\acute{e}bit_{\neg ok}^{m_2}$	$Cr\acute{e}dit^{m_2}$	$Solde_{r2}$
$D\acute{e}bit_{ok}^{m_1}$	*oui*	non	*oui*	non
$D\acute{e}bit_{\neg ok}^{m_1}$	*oui*	*oui*	*oui*	*oui*
$Cr\acute{e}dit^{m_1}$	non	*oui*	*oui*	non
$Solde_{r1}$	*oui*	*oui*	*oui*	*oui*

TAB. I.4 – Relation de recouvrabilité relative de l'ADT COMPTEENBANQUE

La recouvrabilité est un critère plus faible que la commutativité, comme le souligne la propriété suivante, obtenue directement d'après les définitions.

Propriété I.1 ([Badrinaht & Ramamritham 87]) *Soient op_1 et op_2 deux opérations sur un même ADT, alors :*

$$\textit{commute}(op_1, op_2)$$
$$\rightarrow$$
$$(\textit{relativement\_recouvrable}(op_1, op_2) \wedge \textit{relativement\_recouvrable}(op_2, op_1)).$$

La réciproque n'est pas vraie puisque, en particulier, les opérations sans paramètres de sortie recouvrent toutes les autres mais, bien sûr, ne commutent pas nécessairement avec la totalité. L'exemple suivant montre que, en pratique, et malgré la propriété I.1, certaines formes de commutativité en avant ne sont pas acceptées par la recouvrabilité relative !

Exemple I.4 *La table I.4 donne la relation de recouvrabilité relative des opérations de l'ADT* COMPTEENBANQUE. *(Rappelons que la première opération exécutée est celle donnée en ligne.) L'opération Crédit, qui n'admet aucun paramètre de sortie, est relativement recouvrable par rapport à toute autre ; toute opération est relativement recouvrable vis-à-vis de Solde qui ne modifie pas l'objet.*

On vérifie sans peine que cette relation est plus permissive que la relation de commutativité en arrière à droite donnée en table I.2. En revanche, un débit réussi suivi d'un débit raté n'est pas accepté par la recouvrabilité relative car le second débit pourrait réussir si le premier venait à être annulé ; la commutativité en avant accepte les deux exécutions car l'état initial est connu, ce qu'il manque comme connaissance à la recouvrabilité relative pour pouvoir conclure favorablement. De même, un crédit réussi suivi d'un débit réussi est refusé par la recouvrabilité relative mais accepté par la commutativité en avant.

Tout comme pour la commutativité, il est donc possible de définir une recouvrabilité relative *en avant*. Cet aspect fait partie des voies de recherches.

Autres dépendances et conflits Quand aucune propriété de plus ou moins grande indépendance entre opérations ne peut être exhibée, il s'établit des dépendances.

op_1	op_2	critère maximal vérifié
Lire_{val_1}	Lire_{val_2}	commutativité
Lire_{val_1}	$\mathit{Écrire}^{val_2}$	recouvrabilité relative (plus précisément, recouvrabilité simple)
$\mathit{Écrire}^{val_1}$	Lire_{val_2}	dépendance
$\mathit{Écrire}^{val_1}$	$\mathit{Écrire}^{val_2}$	recouvrabilité relative (et « règle de Thomas »)

TAB. I.5 – Les opérations *Lire* et *Écrire* face aux trois critères

Définition I.7 (Conflit entre opérations exécutées sur une instance d'un ADT) *Soient deux opérations op_1 et op_2 d'un même ADT, exécutées dans cet ordre sur une instance partagée, alors op_2* dépend *de op_1 si elles ne commutent pas. On dit aussi, sans tenir compte de l'ordre, que op_1 et op_2 sont* en conflit*.*

Plus formellement, soient op_1 et op_2 deux opérations sur un même ADT, exécutées dans cet ordre, alors :

- *dépend_op*$(op_2, op_1) \leftrightarrow$ *commute*(op_1, op_2) *;*
- *conflicte_op*$(op_1, op_2) \leftrightarrow$ *dépend*$(op_1, op_2) \vee$ *dépend*(op_2, op_1).

Avec l'ancienne compatibilité des opérations *Lire* et *Écrire*, nous pouvons dresser le récapitulatif de la table I.5.

Remarquons que le couple d'opérations *Lire* / *Écrire* vérifie une propriété plus faible que la recouvrabilité relative qui est celle de recouvrabilité simple au sens de [Bernstein et al. 87]. Les opérations qui sont simplement recouvrables sont celles qui ne modifient pas la valeur de l'objet ; leurs opérations inverses étant l'identité, le rejet des transactions ne demande aucune précaution particulière.

Remarquons aussi que le couple *Écrire* / *Écrire* possède une propriété d'absorption que nous n'avons pas mise en lumière. *Écrire* est une opération absorbante et ces dernières offrent la possibilité de choisir arbitrairement l'ordre de sérialisation, donc d'inverser, si besoin, l'ordre réel d'exécution. Cette possibilité, connue sous le nom de « règle de Thomas » [Thomas 79], est généralisée dans [Pons 86].

Tous ces critères, locaux aux objets et à leurs opérations, sont utilisés par le niveau supérieur, celui des transactions, pour assurer la sérialisabilité des exécutions concurrentes.

I.2.2 Modèle de transactions

Définition I.8 (Transaction) *Une transaction est un programme qui se traduit par une séquence d'opérations sur des instances d'ADT. Une transaction se termine soit par une validation, soit par un rejet. En cas de validation, tous ses effets sont rendus permanents et visibles aux autres transactions. En cas de rejet, tous ses effets sont annulés, c'est-à-dire que le système se retrouve dans l'état où il aurait dû être si cette transaction ne s'était jamais déroulée, le*

terme « système » recouvrant aussi bien la base de données que les transactions concurrentes.

Une transaction est une unité de consistance (C de ACID), c'est-à-dire (que l'on suppose) qu'à partir d'un état consistant de la base de données, elle dérive un nouvel état consistant si elle est exécutée seule.

Certaines opérations de la transaction peuvent s'exécuter en parallèle.

Plus formellement, une transaction T est un couple composé d'une famille de triplets d'opérations effectuées sur des instances d'ADT pour le compte de la dite transaction et d'une relation d'ordre partiel sur ces opérations :

$$T = (OP_T; \prec_T)$$

où :

$$OP_T = ((T, op_i, x_i))_{i=1}^{n} \cup ((T, fin_{n+1}, \perp))$$

tel que :

$$(fin_{n+1} = \textit{validation}) \vee (fin_{n+1} = \textit{rejet})$$

et :

$$\forall i : 1 \leq i \leq n, op_i \prec_T fin_{n+1}$$

L'ordre partiel permet de modéliser le parallélisme intra-transaction. Nous supposerons que ce parallélisme interne n'a pas de répercussions sur la correction des exécutions ou, plus simplement, que les transactions sont séquentielles, c'est-à-dire que $\prec_T$ est un ordre total. Autrement, il faut s'en remettre au modèle des *transactions emboîtées* [Moss 82] [Moss 85] que nous n'abordons pas ici.

Le nom de la transaction est introduit pour prendre en compte les exécutions concurrentes dans les définitions suivantes.

La définition entraîne bien qu'une transaction ne peut pas être simultanément validée et rejetée, et que cet événement ne peut être que le tout dernier de la transaction.

Une validation ou un rejet consiste à valider ou rejeter les opérations faites sur chaque instance manipulée par la transaction ; il y en a au plus autant que le nombre d'opérations. Cependant, nous nous contentons ici d'une terminaison sous forme de macro avec le symbole indéfini en lieu et place de celui des instances accédées.

Exemple I.5 *Une exécution réussie de la transaction de transfert de compte à compte, donnée avec l'algorithme I.2, donnera l'exécution parallèle « (T, Débiter$^m_{true}$, CompteDébiteur) $\prec_T$ (T, validation, $\perp$) et (T, Créditer^m, CompteCréditeur) $\prec_T$ (T, validation, $\perp$) ».*

Algorithme I.2 Une transaction de transfert bancaire

```
transaction Transfert (CompteDébiteur, CompteCréditeur : COMPTEENBANQUE ;
                       Montant : real) return boolean ;
var
   DébitRéussi : boolean ;
begin
   parallel
      DébitRéussi := CompteDébiteur.Débiter(Montant, "Transfert") ;
      CompteCréditeur.Créditer(Montant, "Transfert")
   end parallel ;
   if DébitRéussi
   then commit /* validation */
   else rollback /* (auto-)rejet */
   end if ;
   return DébitRéussi
end ;
```

I.2.3 Histoires

Quand plusieurs transactions s'exécutent concurremment, leurs opérations sur les instances partagées peuvent s'enchevêtrer. Une histoire est donc au minimum l'union des exécutions des différentes transactions, mais ne se limite pas à cela. La relation d'ordre d'une histoire contient plus de couples que la simple union des relations d'ordre des transactions. Les couples supplémentaires sont ajoutés lors de l'apparition de conflits, notion associée à celle de dépendance dans la définition I.7.

Définition I.9 (Conflits entre transactions) *Par hypothèse, toutes les opérations exécutées sur une même instance, et qui ne commutent pas, sont totalement ordonnées (cf. définition I.2).*

Nous disons qu'une transaction T_2 dépend d'une transaction T_1 si elles ont exécutées des opérations conflictuelles sur une instance commune et que l'opération de T_2 suit celle de T_1.

Plus formellement, soient T_1 et T_2 deux transactions concurrentes, alors :

$$
\begin{aligned}
&\textit{dépend\_trans}(T_2, T_1)\\
&\leftrightarrow\\
&\exists (T_1, op_1, x_1) \in OP_{T_1},\\
&\exists (T_2, op_2, x_2) \in OP_{T_2},\\
&\quad (T_1 \neq T_2) \wedge\\
&\quad (x_1 = x_2) \wedge\\
&\quad \textit{dépend\_op}(op_2, op_1).
\end{aligned}
$$

Il suit de cette définition que les opérations sur des instances distinctes commutent toujours, dans la mesure où ces instances sont vraiment indépendantes. (La propriété d'objets indépen-

dants n'est pas nécessairement trivialement vérifiée ; nous aurons l'occasion d'en reparler.)

Nous sommes maintenant en mesure de caractériser les exécutions concurrentes, appelées histoires.

Définition I.10 (Histoire concurrente) *Soient k transactions, T_1 à T_k, une histoire de leur exécution concurrente est :*

$$H = (OP_H; \prec_H)$$

où :

- $OP_H = \bigcup_{i=1}^{k} OP_{T_i}$;
- $\prec_H = \bigcup_{i=1}^{k} \prec_{T_i} \cup \{((T_1, op_1, x_1), (T_2, op_2, x_2)) \in OP_H \times OP_H | \textit{dépend\_trans}(T_2, T_1)\}$.

Exemple I.6 *Supposons que trois transactions, T_1, T_2 et T_3, exécutent respectivement* Transfert$(C_1, C_2, 500)$, Transfert$(C_1, C_3, 1\,500)$ *et* Transfert$(C_2, C_4, 750)$. *T_1 et T_2 valident alors que T_3 est rejetée, ce cas de figure pouvant se produire si C_2 est inférieur à 750. Une exécution concurrente possible est :*

« (T_3, Débiter$^{750}_{false}$, *C_2) $\prec_H$ (T_3,* Créditer^{750}, *C_4) $\prec_H$ (T_1,* Créditer^{500}, *C_2) $\prec_H$ (T_3, rejet, $\perp$) $\prec_H$ (T_1,* Débiter$^{500}_{true}$, *C_1) $\prec_H$ (T_2,* Débiter$^{1500}_{true}$, *C_1) $\prec_H$ (T_2,* Créditer^{1500}, *C_3) $\prec_H$ (T_2, validation, $\perp$) $\prec_H$ (T_1, validation, $\perp$) ».*

I.2.3.1 Histoires sérialisables

Définition I.11 (Graphe des dépendances d'une histoire) *À partir de la relation de précédence $\prec_H$ entre opérations d'une histoire H, on construit le graphe des dépendances de H, noté G_H, graphe quotient de $\prec_H$ par $\{T_1, \ldots, T_k\}$, l'ensemble des transactions validées ou actives de H. Autrement dit, les transactions rejetées ne sont pas prises en compte dans le graphe des dépendances.*

Plus formellement, soit H une histoire, alors :

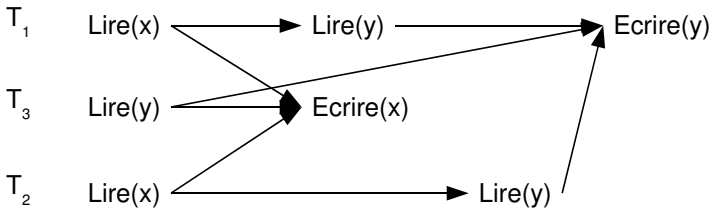
$$G_H = (\{T_1, \ldots, T_k\}, U)$$

avec :

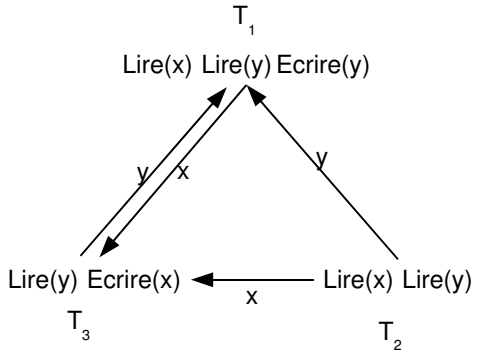
$$(T_i, T_j) \in U \leftrightarrow \exists((T_i, op_i, x), (T_j, op_j, x)) \in \prec_H .$$

Théorème I.2 (Histoire sérialisable [Bernstein et al. 87]) *Une histoire H est sérialisable si, et seulement si, G_H ne contient pas de circuit.*

Nous n'avons pas définie la sérialisabilité d'une exécution concurrente de façon classique en présentant d'abord les histoires séries, puis la notion d'équivalence d'histoires, et enfin l'équivalence d'une histoire sérialisable à une histoire série quelconque. Il est évident, *a posteriori*,



(a) $\prec_H$



(b) G_H

FIG. I.1 – Un exemple d'histoire concurrence et de graphe des dépendances associé

qu'une histoire série, c'est-à-dire dont toutes les transactions se succèdent sans jamais interférer, possède un graphe de dépendances sans circuit. Inversement, si G_H est sans circuit, n'importe quelle extension linéaire donne une exécution série qui aurait eu les mêmes effets sur la base et sur les transactions que l'exécution concurrente des mêmes transactions.

Exemple I.7 *Considérons l'exécution $\prec_H$ de la figure I.1, ne contenant que des opérations* Lire *et* Écrire *car elles génèrent davantage de conflits : T_3 dépend de T_1 et T_2 car elle écrit x qui a été préalablement lu par T_1 et T_2.*[3] *Ensuite, T_1 dépend de T_2 et T_3 pour la même raison sur y. Le graphe quotient G_H contient alors un circuit.*

À partir de G_H, on se rend compte que l'exécution concurrente ne peut être équivalente à une exécution série. T_2 s'exécute bien avant T_1 et T_3, mais ces deux dernières pourraient ne pas avoir le même comportement dans une exécution série. En effet, la lecture de y par T_3 peut *conditionner l'écriture de x, et inversement, l'écriture de y par T_1* peut *découler de la lecture de x. Par conséquent, cet exemple met en évidence que le critère syntaxique de sérialisabilité considère, implicitement, que chaque opération est une fonction de toutes celles qui la précèdent.*

Il faut noter que l'aspect syntaxique du critère de sérialisabilité entraîne que chaque opération d'une transaction est fonction de toutes celles qui la précèdent. La prise en compte des objets typés permet de limiter cette hypothèse très forte. En effet, créditer un compte consiste

[3] Il s'agit d'une dépendance de « recouvrabilité simple », une forme particulière de dépendance qui n'entraîne pas de rejet en cascade.

à lire la valeur, à l'incrémenter, puis à écrire la nouvelle valeur. L'écriture ne dépend que de la lecture préalable et non de toutes celles déjà faites par la transaction. De même, cette lecture n'influe pas sur les opérations ultérieures. Lever les limitations de la sérialisabilité au-delà des objets typés, nécessite de considérer la sémantique des transactions, ce qui est bien plus difficile. ([Shasha et al. 92] propose un découpage syntaxique d'une transaction en une séquence de transactions équivalente. Toutefois, cela nécessite de connaître l'ensemble des transactions susceptibles de s'exécuter en concurrence dans des intervalles de temps donnés.)

I.2.3.2 Histoires recouvrables

Dans une histoire, des transactions peuvent être rejetées pour plusieurs raisons, que nous détaillerons en section I.4. Réaliser le rejet d'une transaction doit être possible sans avoir de répercussions sur les transactions déjà validées, dont il a été garanti que leurs effets seraient permanents ! Il est également préférable que le rejet d'une transaction n'entraîne pas celui d'autres transactions, quand bien même elles seraient encore actives. Ce dernier phénomène est connu sous le terme de « rejets en cascade ». Ceux-ci peuvent se produire dès que l'on laisse s'établir des dépendances entre transactions vivantes.

Dans [Boksenbaum et al. 86], le graphe des dépendances est l'outil privilégié de la comparaison de plusieurs méthodes de contrôle de concurrence. Nous caractérisons également les diverses formes d'histoires recouvrables grâce aux graphes de dépendances qu'elles engendrent.

Les histoires recouvrables forment un sous-ensemble des histoires sérialisables. Elles doivent vérifier des conditions supplémentaires.

Définition I.12 *Appelons T^* le sous-ensemble de $\{T_1, \ldots, T_k\}$ des transactions qui ont validées dans H, et T^+ son complémentaire.*

Définition I.13 (Histoire recouvrable) *Une histoire H est recouvrable s'il n'y a pas d'arcs dans GH depuis une transaction de T^+ vers une transaction dans T^*.*

Ceci signifie que l'on n'autorise pas des dépendances à s'établir depuis une transaction vivante vers une transaction validée. Si la transaction vivante venait à être rejetée cela invaliderait les résultats et les effets d'une transaction validée, que l'on ne peut plus rejeter. La base de données se retrouverait donc dans un état inconsistant.

Définition I.14 *Une histoire H évite les rejets en cascade si aucune transaction de T^+ ne possède d'arcs sortants.*

La définition I.13 n'interdit que les arcs issus de transactions vivantes vers des transactions validées. Mais, des dépendances peuvent se créer entre transactions vivantes. Si une transaction T_2 dépend de T_1, alors, en cas de rejet de T_1, T_2 devra être également rejetée puisqu'elle s'est

exécutée, avec optimisme, sur la base de données modifiée par T_1, qui n'était pas encore définitive. En interdisant ces dépendances dans la définition I.14, on assure que le rejet d'une transaction ne concernera que cette unique transaction. (Notez que les dépendances qui s'établissent entre un lecteur suivi d'un écrivain n'engendrent pas de rejets en cascade puisque la valeur à partir de laquelle T_2 a travaillé était consistante car non modifiée par T_1 (cf. exemple I.7) ; ceci relève de ce que nous avons appelé « recouvrabilité simple » en section I.2.1.2.)

Entre les histoires recouvrables et les histoires sans rejets en cascade, grâce à la recouvrabilité relative, on peut introduire les histoires recouvrables sans rejets en cascade des transactions mais avec rejet en cascade temporaire des opérations !

Définition I.15 (Histoire relativement recouvrable) *Une histoire H est relativement recouvrable si tous les arcs issus de transactions appartenant à T^+ ne sont dus qu'à des dépendances de recouvrabilité relative.*

Les dépendances en général, et celles de recouvrabilité relative en particulier, imposent un ordre de validation qui soit celui de sérialisation. Une transaction de T+ n'est autorisée à passer dans T^* que si tous ses arcs entrants sont issus de transactions appartenant déjà à T^*. (Certaines méthodes autorisent des ordres de validation différents car elles maintiennent des informations annexes sur les instances qui vont permettre de détecter tardivement les incohérences : par exemple, la méthode optimiste par intervalles d'estampilles de [Boksenbaum et al. 84] permet à des lecteurs de valider après des écrivains qui ont modifiés les valeurs lues auparavant.) De plus, l'établissement de dépendances entraîne *nécessairement* celui du risque de rejets en cascade. Toutefois, la définition de la recouvrabilité relative permet de limiter le rejet en cascade aux objets communs seulement.

En effet, supposons qu'une dépendance de recouvrabilité relative se crée entre T_1 et T_2. Si T_1 est rejetée, alors T_2 devrait être rejetée puis redémarrée. Mais, grâce à la propriété de recouvrabilité relative, le rejet peut se limiter aux objets manipulés en commun par T_1 et T_2. De plus, le redémarrage va renvoyer les mêmes résultats aux opération de T_2 qui sont re-exécutées. Par conséquent, les décisions prises par T_2 au cours de son exécution ne seront pas remises en cause. T_2 ne se rend donc pas compte qu'elle a été en partie et momentanément rejetée.

Plus précisément, le protocole de rejet d'une transaction T, avec mises-à-jour immédiates, est le suivant [Badrinath & Ramamritham 92] :

1. sur tous les objets manipulés par T, récupérer la liste des opérations exécutées par les transactions concurrentes et appliquer toutes les opérations inverses, dans l'ordre chronologique inverse, jusqu'à la première opération exécutée par T incluse ;
2. re-exécuter toutes les opérations qui ont été défaites, dans l'ordre chronologique, sans répéter celles appartenant à T.

(1) \ (2)	$Dépiler_{ok,x_2}$	$Dépiler_{\neg ok}$	$Empiler^{x_2}$	$Sommet_{ok,x_2}$	$Sommet_{\neg ok}$
$Dépiler_{ok,x_1}$	si $x_1 = x_2$	non	*oui*	si $x_1 = x_2$	non
$Dépiler_{\neg ok}$	$\perp$	*oui*	*oui*	$\perp$	*oui*
$Empiler^{x_1}$	non	$\perp$	*oui*	non	$\perp$
$Sommet_{ok,x_1}$	*oui*	$\perp$	*oui*	*oui*	$\perp$
$Sommet_{\neg ok}$	$\perp$	*oui*	*oui*	$\perp$	*oui*

TAB. I.6 – Relation de recouvrabilité relative de l'ADT PILE

Exemple I.8 *Dans le cas de l'ADT* COMPTEENBANQUE*, la recouvrabilité relative se ramène à la commutativité en arrière à droite jointe à la « recouvrabilité simple ». Nous introduisons alors l'ADT* PILE *dont les opérations et la relation de recouvrabilité relative sont données en table I.6. (Les intersections notées « $\perp$ » correspondent à des exécutions série impossibles, donc des cas à ne pas considérer.)*

Soit une transaction T_1 qui exécute Empiler^x *suivie d'une transaction T_2 qui exécute* Empiler^y *sur une même instance de pile. L'opération* Empiler *ne renvoyant pas de paramètre de sortie est relativement recouvrable par rapport à toute autre, d'où l'exécution concurrente est autorisée. Supposons que T_1 soit rejetée, alors son rejet sur l'instance commune va consister à exécuter* Dépiler$_y$ *puis* Dépiler$_x$ *lors de la phase de rejet en cascade local, puis à refaire* Empiler^y*. L'état de la pile est bien restauré dans l'état où elle aurait dû être si T_1 ne s'était jamais exécutée et la transaction T_2 n'a pas eu à subir un rejet brutal.*

Théorème I.3 ([Bernstein et al. 87]) *L'ensemble des histoires sans rejets en cascade est inclus dans celui des histoires relativement recouvrables, qui est lui-même inclus dans celui des histoires recouvrables, et enfin dans celui des histoires sérialisables.*

Tout ceci concernait la théorie. En pratique, diverses méthodes existent pour le contrôle de concurrence et pour la reprise. Les interactions entre ces deux composants sont d'ailleurs subtiles [Weihl 89b].

Pour ce qui concerne le contrôle de concurrence, on distingue les méthodes *continues* des méthodes *par certification*. Parmi les premières, dites également *pessimistes*, se trouvent le verrouillage [Eswaran et al. 76] et l'estampillage. Parmi les méthodes par certification, dites aussi *optimistes*, on trouve les ensembles d'objets [Kung & Robinson 81] ou les intervalles d'estampilles [Boksenbaum et al. 84]. Les méthodes à base de multi-version forment une classe plus générale dans le cas de la compatibilité ; l'emploi d'opérations typées dans ce cadre reste une voie de recherche.

En ce qui concerne la reprise, là aussi de nombreuses méthodes ont été proposées. [Verhofstad 78] dresse une liste de diverses techniques de restauration des données d'une base après une panne, mais elles ne sont pas toutes utilisées dans le cadre transactionnel (programmes

de récupération, sauvegardes incrémentales, journalisation, pages « ombrées »,[4] fichiers différentiels, etc.). [Verhofstad 78] note déjà que la technique qui devrait s'imposer est celle de la journalisation [Gray 78] [Kohler 81].

Dans les deux sections suivantes, nous n'allons présenter que les deux méthodes les plus connues, et généralement celles implémentées dans les systèmes commerciaux : le verrouillage pour le contrôle de concurrence et la journalisation pour la reprise. Des simulations, sous les hypothèses d'un système centralisé, de transactions plates et courtes, et d'opérations de lecture et d'écriture sur pages physiques, donnent ce couple favori dans la majorité des cas [Agrawal & DeWitt 85] [Agrawal et al. 87].

[4]La page est l'unité d'échange entre la mémoire centrale et le disque. Le mécanisme des pages « ombrées » consiste à maintenir deux versions d'une même page : la dernière version validée et, éventuellement, la version en cours de modification. Lors d'une validation, il se produit une bascule entre ces deux pages.

I.3 CONTRÔLE DE CONCURRENCE PAR VERROUILLAGE

Le mécanisme de contrôle de concurrence le plus communément implémenté est le *verrouillage*. Il s'agit d'une méthode qualifiée de *continue* car chaque accès à un objet engendre un contrôle. Ce contrôle se traduit par la pose d'un verrou sur l'objet auquel veut accéder la transaction. Si l'accès est autorisé, la transaction peut continuer à s'exécuter, autrement le contrôle de concurrence soit fera patienter la transaction jusqu'à ce qu'elle puisse se poursuivre, soit la rejettera.

[Eswaran et al. 76] a montré qu'en l'absence de connaissance sur la sémantique des transactions, le *verrouillage à deux phases* (2PL pour « *2-Phase Locking* ») assure la consistance. De plus, le verrouillage à deux phases *strict* évite les rejets en cascade.

Verrouiller tous les objets d'une base peut consommer une importante partie des ressources du système (place mémoire et temps processeur). Afin de limiter ces pertes, le verrouillage hiérarchique a été introduit [Gray 78].

Le cas du rejet se produit quand les transactions se sont exécutées, ou risquent de s'exécuter, de telle manière qu'il n'existe pas d'ordre de sérialisation. En terme du graphe des dépendances de la définition I.11, cela signifie qu'il y a, ou qu'il risque de se former, un circuit dans ce graphe. Ce problème est connu sous le terme d'« *interblocage* » et il existe deux solutions pour le résoudre : la *prévention* ou la *guérison*. Nous ne traiterons pas ce problème et renvoyons le lecteur à l'étude de [Knapp 87].

I.3.1 Verrouillage à deux phases strict

Le *protocole de verrouillage à deux phases strict*, étendu aux ADT, est le suivant :

1. À chaque objet de la base de données est associé un verrou. Le verrou contient la liste des opérations exécutées, appartenant à des transactions actives, et la liste des opérations en attente, appartenant à des transactions suspendues (ou bloquées).
2. Chaque fois qu'une transaction veut exécuter une opération sur un objet, le verrou de l'objet doit être examiné. Si la nouvelle opération n'est en conflit avec aucune opération dans les différentes listes, elle est autorisée à s'exécuter. Par abus de langage, nous dirons que la transaction a obtenu, ou posé, le verrou. Dans le cas contraire, la nouvelle opération est insérée dans la liste des opérations en attente et sa transaction est bloquée.
3. À la fin d'une transaction, au moment où elle valide ou lorsqu'elle est rejetée, cette dernière ôte ses opérations de tous les verrous associés aux objets qu'elle a manipulés, per-

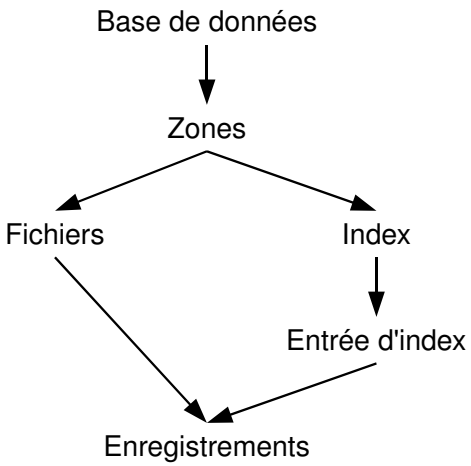


FIG. I.2 – Exemple de graphe des granules

mettant ainsi à des opérations bloquées, ou plus exactement à leurs transactions, d'être éventuellement redémarrées.

Théorème I.4 ([Eswaran et al. 76]) *Si toutes les transactions observent le protocole énoncé ci-dessus, alors (1) les exécutions sont sérialisables et (2) elles évitent les rejets en cascade.*

I.3.2 Verrouillage hiérarchique

D'une part, considérer tous les objets à un même niveau ne représente pas bien la réalité. Dans un système réel, les objets sont structurés, *physiquement ou logiquement*, en une hiérarchie de granules. Une relation se décompose en plusieurs fichiers ; les fichiers contiennent des enregistrements ; les relations peuvent être indexées ; les fichiers se trouvent répartis dans des zones du disque ; plusieurs relations forment une base de données ; etc. La figure I.2 donne un exemple de graphe des granules, comme on peut en trouver dans [Gray 78], [Korth 82], [Korth 83], [Bernstein et al. 87], [Korth et al. 88] ou [Garza & Kim 88].

Quand nous soulignions que le graphe des granules peut refléter une organisation physique, cela signifie que le graphe de la figure I.2 peut être utilisé indépendamment du modèle de données de la base elle-même (hiérarchique, réseau, relationnelle ou objet). Une instanciation possible de ce graphe des granules est donnée en figure I.3. Notez que tous les arcs n'ont pas à être nécessairement instanciés, par exemple si un fichier n'est pas muni d'index.

D'autre part, considérer que les objets appartenant à une base de données sont immuables ne reflète également pas la réalité. Les objets peuvent être détruits et créés et il faut éviter le problème des objets dits « fantômes ».

Le verrouillage hiérarchique poursuit donc deux buts principaux :

(i) minimiser le nombre de verrous à obtenir ;

(ii) éviter le problème des objets « fantômes ».

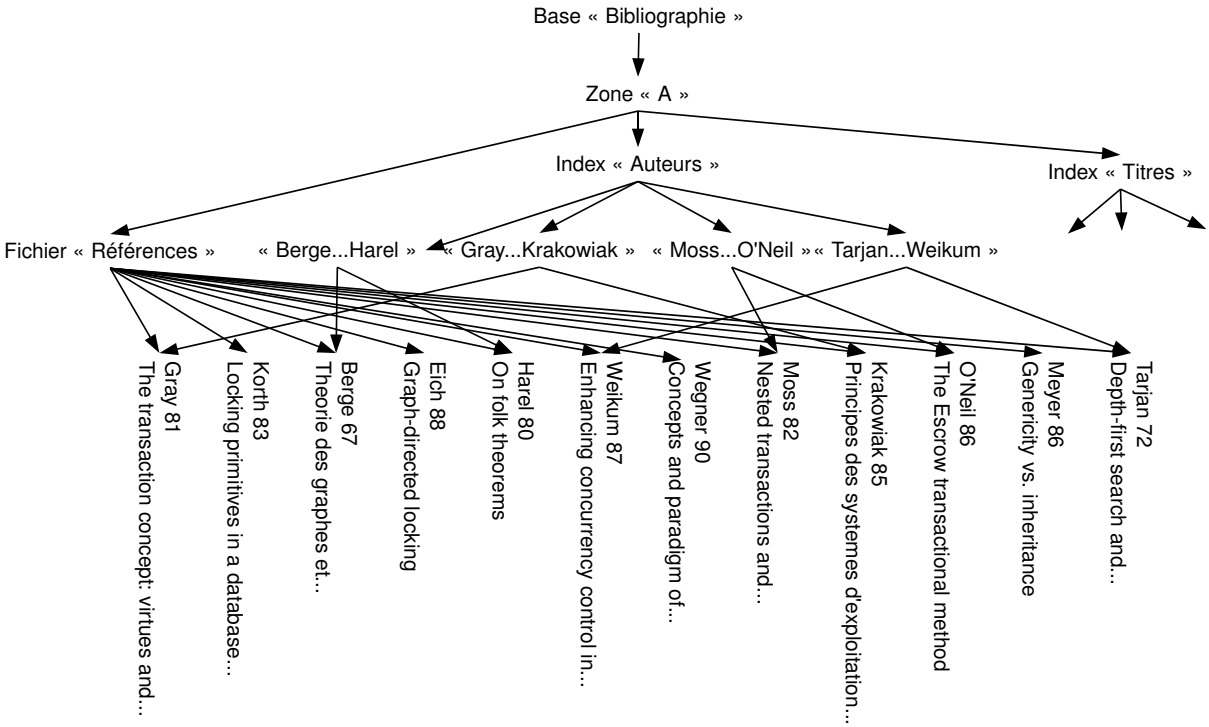


FIG. I.3 – Une instanciation du graphe des granules de la figure I.2

minimiser le nombre de verrous Si une transaction doit accéder à la totalité des articles d'un fichier (ou même à une grande majorité seulement), alors il est intéressant de ne poser qu'un seul verrou sur le fichier au lieu d'un par enregistrement. Les enregistrements seront dits verrouillés *implicitement*. Il faudra alors détecter les incompatibilités avec les enregistrements verrouillés *explicitement*.

Les transactions qui manipulent beaucoup d'objets privilégieront donc le verrouillage implicite, ou hiérarchique, afin de diminuer la surcharge d'avoir à poser un verrou sur chaque objet, au détriment d'un parallélisme potentiel. Les transactions manipulant peu d'objets verrouilleront explicitement chaque objet ce qui permettra de faire s'exécuter concurremment un grand nombre de « petites » transactions.

objets « fantômes » Supposons qu'une transaction veuille effectuer un parcours total sur un ensemble d'enregistrements vérifiant un certain prédicat. Ces enregistrements vont être verrouillés par la transaction. Supposons maintenant qu'une transaction concurrente crée un nouvel enregistrement qui vérifie ce prédicat-là. Il faut d'une façon ou d'une autre détecter l'incompatibilité.

Le *verrouillage prédicatif* a été introduit par [Eswaran et al. 76] pour résoudre ce problème. Mais, en pratique, il a été abandonné car tester des prédicats, même sous une forme simplifiée, se révèle coûteux en temps. Un graphe de granules peut être considéré comme un ensemble de prédicats pré-définis et très simples, par exemple en figure I.3, « l'ensemble des enregistrements

qui appartiennent au fichier Références' », ou plus finement, « l'ensemble des enregistrements qui sont référencés par la même entrée d'index Gray...Krakowiak' ».

Le *protocole de verrouillage hiérarchique*, étendu aux ADT, est le suivant :

1. On distingue la pose d'un verrou intentionnel et celle d'un verrou explicite.
2. Les verrous intentionnels sont toujours compatibles entre eux ; dans tous les autres cas, les opérations sont en conflit conformément à leur relation de commutativité.
3. Avant d'obtenir un verrou explicite sur un granule, la transaction doit obtenir des verrous intentionnels sur une majorité de granules parents. Avant d'obtenir un verrou intentionnel sur un granule, la transaction doit posséder le verrou intentionnel sur au moins un granule parent. Par induction, les verrous sont donc posés depuis la racine du graphe des granules jusqu'aux granules verrouillés explicitement. (Ceci est la version non biaisée du protocole ; on peut proposer des versions biaisées, qui favorisent certaines opérations ayant une relation de commutativité particulière [Korth 83].)
4. Les verrous doivent être relâchés dans l'ordre inverse de la pose, c'est-à-dire des feuilles vers la racine (ou dans n'importe quel ordre si le relâchement ne survient que lors de la terminaison de la transaction).

Notez bien que le protocole de verrouillage hiérarchique ne remplace pas le protocole de verrouillage à deux phases, mais s'utilise en combinaison avec celui-ci.

Théorème I.5 ([Gray 78]) *Le verrouillage hiérarchique dans un graphe de granules est équivalent au verrouillage explicite sur les nœuds puits du graphe.*

D'un point de vue pratique, pour que le verrouillage hiérarchique se révèle efficace, il faut trouver pour chaque type d'opération la bonne coupure dans l'instanciation du graphe. La bonne coupure est celle qui minimisera le nombre de verrous à obtenir et autorisera encore tout le parallélisme potentiel. (Évidemment, la coupure minimale, en terme de verrous à obtenir, est celle qui se restreint à la racine du graphe des granules, mais elle limite le parallélisme de manière draconienne.)

Exemple I.9 *En figure I.3, pour accéder à tous les enregistrements d'un fichier, la bonne coupure est de verrouiller le nœud fichier ainsi que les nœuds index explicitement. Tous les enregistrements (ainsi que toutes les entrées dans les index) sont alors verrouillés implicitement. Ce sont donc les nœuds « Références », « Auteurs » et « Titres » qui seront verrouillés explicitement, tandis que « A » et « Bibliographie » seront verrouillées intentionnellement. Remarquez que si les fichiers sont souvent indexés sur plusieurs champs, il peut être intéressant d'insérer dans le graphe des granules un nœud entre zone, d'une part, et fichier et index, d'autre part. Accéder à tous les enregistrements d'un fichier pourra alors être contrôlé sur un seul granule.*

Pour insérer une nouvelle référence, la bonne coupure est assez particulière. Il s'agit de verrouiller explicitement le nouvel enregistrement ainsi que toutes les entrées d'index où il va être référencé. Les nœuds « Références », « Auteurs », « Titres », « A » et « Bibliographie » seront verrouillés intentionnellement. Cette coupure permettra de créer, détruire, accéder ou modifier concurremment la plus grande partie des autres enregistrements du même fichier. Par exemple, insérer la référence « Abbott 90 » nécessite de poser un verrou explicite sur l'enregistrement créé ainsi que dans l'entrée d'index « Berge...Harel » (et similairement pour le titre).

I.4 REPRISE PAR JOURNALISATION

L'autre composante essentielle d'un système transactionnel est la reprise. Le but de la reprise est d'assurer la consistance de la base de données que l'un des événements suivants pourrait violer :

- destruction de la base de données par détérioration du support physique (atterrissage des têtes de *lecture*/écriture sur le disque) ;
- *panne* de la machine entraînant un arrêt du système (erreur de parité mémoire, erreur de processeur, ou plus simplement coupure d'alimentation) ;
- *interruption* de la transaction par l'utilisateur ;
- *rejet* de la transaction par le gestionnaire de transaction (rejet dû à la résolution d'un interblocage, à un dépassement de temps imparti) ;
- interruption de la transaction par le système à cause d'une *erreur* logicielle (division par zéro, accès à des données sans droits suffisants, violation d'une contrainte d'intégrité) ;
- *dépassement du temps maximal* de terminaison de la transaction dans des systèmes temps-réel ;
- etc.

Le mécanisme de reprise le plus souvent implémenté est la journalisation. *System R* combine la journalisation avec le mécanisme des pages « ombrées ». Malgré cela, et de l'avis même des concepteurs, le mécanisme de journalisation seul est suffisant et bien plus simple à mettre en œuvre [Gray et al. 81].[5] Nous présentons donc la journalisation en avant ou *WAL* (pour « *Write-Ahead Logging* ») [Gray 78].

I.4.1 Principe de la journalisation et hypothèses

Tout d'abord, nous faisons l'hypothèse qu'un fichier particulier, le *journal*, résiste à *toutes* les pannes. Ceci peut être obtenu en le dupliquant autant de fois que voulu sur des disques différents pour diminuer la probabilité que diverses pannes n'endommagent simultanément toutes les copies (généralement deux sont utilisés, d'où le terme de « disques miroirs »).

Observant les résultats de [Reuter 84], nous choisissons une méthode de reprise qui, vis-à-vis de quatre critères, fait les hypothèses les plus faibles :

[5] Le mécanisme des pages « ombrées » avait été repris depuis un système mono-utilisateur où il assurait parfaitement la tolérance aux pannes. Mais, son utilisation dans un système concurrent est redondant dès que le granule de verrouillage est inférieur à la page du disque.

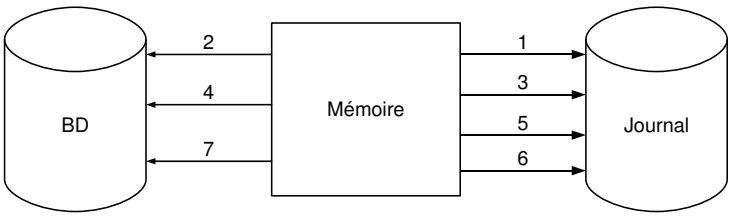


FIG. I.4 – Principe de la journalisation

(i) *propagation non atomique* : l'état instantané de la base n'est pas nécessairement consistant, c'est-à-dire que l'on ne force pas une transaction à propager toutes ses modifications de manière atomique dans la base de données ;

(ii) *écritures non différées* : les modifications d'une transaction vivante peuvent être répercutées sur la base de données avant sa validation ou même son rejet ;

(iii) *écritures non forcées* : les modifications d'une transaction validée peuvent ne pas être encore présentes dans la base mais seulement en mémoire centrale ;

(iv) *points de reprise flous* : on n'impose pas des points de reprises à des moments particuliers, notamment à la validation de chaque transaction.

Le principe de la journalisation WAL consiste à écrire des informations redondantes sur le journal avant toute écriture dans la base de données. Sur le schéma de principe de la figure I.4, on distingue les étapes suivantes :

1. *début de transaction* : un enregistrement indiquant le début d'une transaction est écrit dans le journal au démarrage d'une transaction ;[6]
2. *chargement des données* : la transaction accède directement en lecture à la base de données ;
3. *assurer l'atomicité* : avant d'écrire dans la base la valeur d'un objet modifié par une transaction non validée, des informations permettant de compenser cette écriture doivent être notées dans le journal ;
4. *écritures non retardées* : des données modifiées sont écrites prématurément dans la base ;
5. *assurer la durabilité* : avant de valider, une transaction doit noter dans le journal les informations permettant de refaire les modifications faites aux objets de la base de données ;[7]
6. *fin de transaction* : un enregistrement de fin de transaction est écrit dans le journal ;
7. *écritures non forcées* : les modifications d'une transaction validée peuvent être reportées dans la base après sa validation.

[6] Ce point accélère l'algorithme de reprise mais n'est pas indispensable.

[7] Mais aussi de *défaire* si les transactions sont réparties sur plusieurs sites, à cause de la validation à deux phases qui peut échouer.

Le terme « *WAL* » traduit donc le fait que l'état du journal est toujours en avance sur celui de la base de données : (3) précède (4) et (5) précède (7). Ce protocole de journalisation découle des hypothèses faibles faites auparavant. Avec des hypothèses plus fortes, les étapes (4) et/ou (7) auraient été éliminées : (4) si les écritures sont retardées à la validation, (7) si les écritures sont forcées à la validation. Détaillons quelques aspects de nos hypothèses.

I.4.1.1 Objets et opérations logiques

Il doit y avoir adéquation entre le granule de verrouillage du contrôle de concurrence et le granule de reprise [Haerder & Reuter 83]. Comme nos opérations portent sur des ADT, il est indispensable que la reprise se fasse sur des objets logiques. D'autre part, comme nous utilisons le critère de commutativité des opérations, plusieurs opérations peuvent s'exécuter en concurrence sur une même instance. On ne peut donc pas s'en remettre à la méthode plus simple de la journalisation des images avant et/ou après, c'est-à-dire de la valeur de l'instance avant et après exécution d'une opération.

En effet, supposons que la valeur initiale d'un compte en banque soit de 5 000 F. Deux transactions concurrentes exécutent un crédit de 3 000 F et 500 F, dans cet ordre. La transaction T_1 aura comme image-avant 5 000 et comme image-après 8 000. La transaction T_2 aura comme image-avant 8 000 et comme image-après 8 500. Supposons que T_2 valide et que T_1 soit rejetée. Aucune des informations journalisées ne permet de remettre la base dans un état cohérent, c'est-à-dire avec un compte en banque valant 5 500 F. (Même la méthode de la différence bit à bit entre image-avant et image-après ne fonctionne pas.) En journalisant des opérations, et en supposant qu'au moment de la panne la base contienne les deux mises-à-jour, il suffit d'exécuter une décrémentation de 3 000 F, ce qui donne bien le dernier état validé.

Comme le sous-entend la dernière phrase de l'exemple, la prise en compte des opérations au lieu de la simple restauration de valeurs pose un problème supplémentaire : celui de l'*idempotence* des opérations. Les opérations sur un ADT n'étant pas naturellement idempotentes, c'est au mécanisme de reprise de faire en sorte que les opérations et les opérations inverses ne soient exécutées qu'une seule fois, quand l'objet est dans l'état où elles sont définies.

I.4.1.2 Écritures non retardées et atomicité

La politique des mises-à-jour immédiates sur les instances partagées nécessite de gérer des opérations inverses.[8]

La mémoire centrale est bien plus limitée que la mémoire de masse. Au fur et à mesure que des données sont lues depuis le disque, le gestionnaire de pages leur trouve une place en mémoire centrale. Quand la mémoire centrale est saturée, le gestionnaire de pages doit choisir

[8] Avec un modèle d'exécution où les mises-à-jour sont différées, ce problème sera troqué pour un autre, peut-être plus difficile.

une page à renvoyer sur le disque. Il existe, dans un système d'exploitation, un mécanisme qui permet de rendre inéligible une page. Nous faisons l'hypothèse que contrôle de concurrence et reprise n'utilisent pas ce mécanisme pour bloquer en mémoire, jusqu'à la validation, les pages modifiées.

Dans ce cas, des pages contenant des instances modifiées par des transactions actives peuvent être renvoyées sur disque à tout instant. L'état physique de la base n'est donc pas le dernier état validé. L'atomicité (A de ACID) des transactions n'est pas assurée *de facto* et, en cas de panne, les effets des transactions qui étaient actives à cet instant-là doivent être défaits. Il faut donc enregistrer dans le journal les opérations inverses.

I.4.1.3 Écritures non forcées et durabilité

Lors de la validation d'une transaction, certaines pages contenant des instances que cette transaction a modifiées peuvent ne pas avoir été encore écrites sur disque, soit parce que la mémoire centrale n'a pas encore été saturée, soit parce que il s'agit d'une donnée utilisée en quasi permanence et qu'elle n'est donc jamais choisie pour retourner sur le disque. Si l'on ne veut pas pénaliser les performances des validations, la stratégie à adopter consiste à ne pas forcer l'écriture des instances modifiées.

Par conséquent, l'état physique de la base ne correspond pas toujours au dernier état validé, mais pour une raison différente du paragraphe précédent : tous les effets d'une transaction validée ne sont pas dans la base de données. En cas de panne, les effets de certaines transactions validées doivent être refaits pour garantir la durabilité (D de ACID) des transactions validées. Il faut donc enregistrer dans le journal les opérations directes.

En reprenant les différentes conséquences découlant des hypothèses choisies, nous devons :

(i) offrir un granule de reprise égal à celui du contrôle de concurrence, c'est-à-dire l'objet au lieu de la page physique (l'unité d'échange avec le disque restant la page, il y a donc un problème à résoudre) ;

(ii) journaliser des opérations et assurer l'idempotence de l'application de ces opérations (avec les images avant et après, l'idempotence est obtenue *de facto*) ;

(iii) journaliser des opérations inverses pour défaire les transactions non validées (rejetées ou encore actives au moment d'une panne) dont les effets partiels ont été propagés dans la base ;

(iv) journaliser des opérations directes pour refaire les effets de transactions validées dont tous les effets n'avaient pas encore été reportés au moment de la panne.

Cette version de la journalisation, qui fait les hypothèses les plus faibles, a été implémentée chez IBM sous le nom d'ARIES [Mohan et al. 89] [Mohan et al. 92]. C'est cette version que nous présentons dans la section suivante.

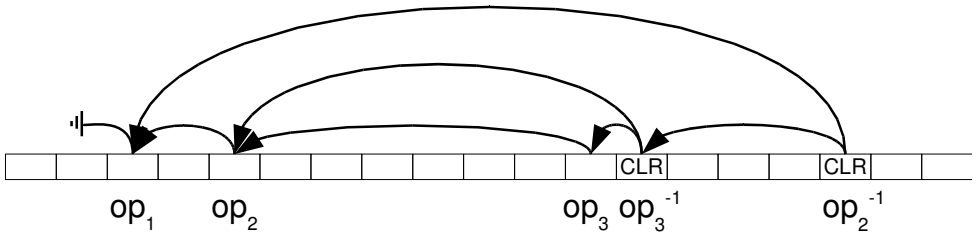


FIG. I.5 – Chaînage des enregistrements d'une transaction dans le journal

I.4.2 L'algorithme « *REDO/UNDO* » ou « *repeating history* »

Les termes « *REDO/UNDO* » ou « *repeating history* » traduisent bien le fonctionnement de l'algorithme. Celui-ci se décompose en deux phases. La première phase de la reprise, « *REDO* », consiste à remettre l'état physique de la base de données dans l'état logique où elle se trouvait au moment de la panne. La seconde phase, « *UNDO* », peut alors s'exécuter sur une base cohérente vis-à-vis du journal. Avant de détailler ces deux phases, nous décrirons les structures de données utilisées et les traitements en phase d'exécution normale.

I.4.2.1 Structures de données

Les enregistrements du journal sont référencés par un numéro unique, faisant parti d'une suite strictement croissante, qui indique leurs positions relatives dans le journal. Nous appellerons ce numéro le *LSN* (pour « *Log Sequence Number* »). Les enregistrements d'une même transaction sont chaînés entre eux. (Logiquement, on pourrait considérer que chaque transaction possède son propre journal.)

Le LSN sert à résoudre le problème de l'idempotence des opérations directes. Pour régler ce même problème pour les opérations inverses, [Mohan et al. 92] écrit des enregistrements dans le journal chaque fois qu'une opération inverse est effectuée. Ce type d'enregistrement, appelé un CLR (pour « *Compensation Log Record* »), contient un pointeur qui permet de sauter toutes les opérations qui ont déjà été compensées.

Les chaînages sont illustrés en figure I.5. Tous les enregistrements d'une transaction, y compris ceux concernant les opérations inverses sont chaînés en arrière (les traits simples). Chaque enregistrement correspondant à l'exécution d'une opération inverse pointe directement sur le premier enregistrement de la dernière opération directe qui n'a pas encore été compensée (les traits gras).

D'autre part, dans la base de données, chaque page contient le LSN de la dernière opération qui a affecté l'*un* des objets qu'elle contient.

I.4.2.2 Traitement normal

Quand une opération modificatrice est appliquée à un objet, celui-ci se trouvant dans une page que l'on supposera déjà en mémoire centrale, (1) l'objet doit être verrouillé,[9] (2) la page qui le contient est rendue non éligible pour un retour inopiné sur disque, (3) elle est verrouillée momentanément en mode écriture,[10] (4) l'opération est exécutée, (5) un enregistrement est ajouté dans le journal, (6) le LSN de cet enregistrement modifie celui qui est associé à la page, (7) le verrouillage de la page est levé, (8) elle est de nouveau rendue éligible pour être éventuellement renvoyée sur disque.[11]

Quand une opération de lecture est appliquée à un objet, les mêmes étapes, à l'exception des points (5) et (6), sont exécutées. Dans l'étape (3), la page est verrouillée en mode lecture seulement.

À la terminaison d'une transaction, un enregistrement indiquant la réussite ou l'échec de celle-ci est écrit dans le journal.

I.4.2.3 Phase « *REDO* »

Lors d'un redémarrage, le journal est parcouru dans le sens chronologique. Chaque enregistrement correspondant aussi bien à une opération directe qu'à une opération inverse est traité. La page référencée par l'enregistrement du journal est chargée en mémoire, si elle ne s'y trouve pas déjà. Si le LSN de cette page est inférieur au LSN du journal, alors on sait que l'opération n'a pas été répercutée dans la base ; elle est donc ré-appliquée dans un état cohérent, puisque le journal est parcouru dans le sens chronologique. Donc, l'idempotence des opérations directes et inverses est garantie durant cette phase par l'existence des LSN. La ré-exécution de certaines opérations n'engendre pas l'écriture de nouveaux enregistrements dans le journal.

À la fin de cette phase, l'état de la base de données est de nouveau celui dans lequel elle se trouvait au moment de la panne.[12]

Durant cette phase, le système collectera également la liste des transactions qui ont été interrompues par la panne, c'est-à-dire celles dont il n'apparaît pas d'enregistrement de terminaison dans le journal, ainsi que le LSN du dernier enregistrement correspondant à la dernière opération directe qui n'a pas encore été défaite. Toutes ces transactions vont être rejetées par la phase suivante.

[9] Nous renvoyons le lecteur à [Mohan et al. 89] pour le problème de la création d'un nouvel objet dans une page. En effet, il est impossible d'obtenir un verrou sur cet objet avant de l'avoir créé !

[10] Si l'objet a la taille d'une ou plusieurs pages, ce verrouillage « court » est inutile.

[11] Le blocage d'une page en mémoire n'étant que très temporaire n'est donc pas en contradiction avec l'hypothèse de non retardement des écritures que nous avons retenue.

[12] modulo les pages modifiées qui n'ont pas eu le temps d'être journalisées.

I.4.2.4 Phase « *UNDO* »

Pendant cette phase, le journal est de nouveau traversé, mais dans le sens chronologique inverse cette fois. Les enregistrements qui n'appartiennent pas à la liste des transactions à rejeter, établie préalablement, sont simplement ignorés. Autrement, lorsqu'un CLR est rencontré, son pointeur vers l'opération précédant celle qu'il a compensé devient le nouveau LSN de la dernière opération à défaire de la transaction. Lorsqu'un enregistrement non-CLR est rencontré, l'opération inverse est appliquée,[13] ce qui nécessitera d'écrire un enregistrement CLR à la fin du journal, et le nouveau LSN de la dernière opération à défaire de la transaction devient celui de l'opération précédente.

À partir de l'illustration en figure I.5, imaginons que nous nous trouvions juste après l'écriture de op_3 quand une panne se produit. Au redémarrage, cette transaction est retenue parmi celles qui doivent être rejetées puisqu'elle n'a pas écrit d'enregistrement de terminaison. Le LSN de la dernière opération est celui de op_3. Lors du parcours anti-chronologique, lorsque le système parvient à cet enregistrement, il se rend compte qu'il s'agit d'un enregistrement non-CLR qui doit donc être compensé : l'opération op_3^{-1} est appliquée, le CLR marquant cette compensation est enregistré dans le journal, le pointeur du CLR pointe sur l'opération de la même transaction précédant op_3, c'est-à-dire op_2, et enfin, le LSN de la nouvelle dernière opération est également positionné à celui de op_2. Si le système retombe en panne, lors du redémarrage suivant, l'opération op_3 ne sera plus défaite puisque le dernier enregistrement est un CLR qui permet de passer directement à op_2. Le chaînage arrière des CLR assure donc l'idempotence des opérations inverses pendant la phase « *UNDO* ».

De nombreuses améliorations des performances, et autres « détails », viennent compliquer la présentation synthétique que nous avons donné de cet algorithme de reprise. La plus importante omission est la présence de points de reprises dans le journal. Ceux-ci évitent d'avoir à parcourir la totalité du journal à chaque redémarrage, mais entraînent une phase préalable d'analyse qui, entre autre, détermine à quel enregistrement du journal doit débuter la phase suivante, le « *REDO* ».

L'idée essentielle est qu'il existe des méthodes de reprise et surtout de reprise sur panne permettant au contrôle de concurrence d'exploiter la sémantique des ADT et de leurs opérations, c'est-à-dire de travailler à un granule inférieur à celui d'une page sur le disque. En cela, ARIES semble jouir d'un certain renom puisque plusieurs publications récentes l'emploient [Kuo 92] [Franklin et al. 92] [Jhingran & Khedkar 92]. Cependant, pour les environnements à objets, cette méthode présente une faiblesse que nous illustrons dans l'exemple suivant.

[13] Notons ici une faiblesse, importante, de la méthode des mises-à-jour immédiates : les opérations inverses doivent toujours réussir. Autrement dit, si l'exécution d'une opération peut entraîner une erreur logique, comme une division par zéro, cela est tout à fait exclu pour les opérations inverses puisque les effets de transactions rejetées ne pourraient plus être défaits. En cela, la technique des images avant et après offre l'atout incontestable de la simplicité et donc de l'absence de telles erreurs.

Exemple I.10 *Les ADT sont souvent des objets clos. Les instances de classes dans les systèmes à objets sont plus souvent liées entre elles par des liens de composition. Par exemple, une instance père référence tous ses enfants au travers d'une variable d'instance «* Enfants : list of PERSONNE *». Inversement, chaque enfant possédera une variable «* Père : PERSONNE *».*

Supposons que l'on effectue une opération modificatrice op_1 sur une instance père qui va, durant son déroulement, invoquer une opération op'_1 sur les instances des enfants. Pendant ce temps, une transaction concurrente exécute une opération modificatrice op_2 sur l'un des enfants, qui va engendrer un appel à une méthode op'_2 sur le père. Nous supposerons aussi que op_1 et op'_2 ainsi que op'_1 et op_2 commutent.

En observant le protocole en huit points exposés dans le paragraphe « traitement normal » ci-dessus, le système va aboutir à un interblocage indétectable *par ARIES.*

En effet, l'instance père est verrouillée par op_1, puis la page contenant cette instance est verrouillée, momentanément, en mode exclusif puisque op_1 est une opération modificatrice. Simultanément, l'instance enfant est verrouillée par op_2 et la page la contenant, supposée différente de celle du père, est verrouillée en mode exclusif également. Le drame est joué. L'opération op'_1 verrouille d'abord l'instance enfant (op'_1 et op_2 commutent), puis sa page, en mode exclusif ou partagé, cela n'a pas d'importance. Symétriquement, le père et sa page sont verrouillés. Chaque transaction est bloquée sur les verrous courts des pages alors qu'aucun interblocage n'est apparu au niveau des instances. Comme ARIES ne détecte pas les interblocages au niveau des verrous de pages, les transactions sont interbloquées sans espoir de guérison !

Cet exemple met en évidence, d'une part, la nécessité de prendre en compte des opérations qui travaillent sur plusieurs pages et, d'autre part, le besoin de détecter les interblocages à ce niveau. En fin de compte, le niveau des pages mériterait d'être traité tout comme le niveau des instances. Le modèle des transactions multi-niveaux apporte une solution, partielle toutefois.

I.5 TRANSACTIONS MULTI-NIVEAUX

Le principe des transactions multi-niveaux se retrouve déjà en essence dans *System R* [Beeri et al. 88]. Sa généralisation a été menée en grande partie par Weikum [Weikum 87] [Weikum 91] dans le cadre du projet DASDBS [Schek et al. 90]. La reprise est étudiée dans [Weikum et al. 90] et les premières performances dans [Badrinath & Ramamritham 90] [Hasse & Weikum 91]. D'autres auteurs ont également étudié les transactions multi-niveaux sous différents aspects : [Moss et al. 85] [Moss et al. 86] et [Beeri et al. 89] pour des modélisations formelles, [Cart et al. 90a] pour un contrôle de concurrence entièrement optimiste, [Cart et al. 90b] pour une structuration correcte des objets imbriqués.

Une fois de plus, nous n'allons pas procéder à un tour d'horizon exhaustif, mais nous limiter à l'extension des résultats des trois premières sections de ce chapitre. Cette présentation reprend essentiellement [Weikum 91] et [Weikum et al. 90]. Un autre récapitulatif est proposé dans [Cart et al. 89], tandis qu'une comparaison entre le modèle de transactions plates et multi-niveaux apparaît dans [Cart & Ferrié 89a]. Enfin, dans le cours des chapitres suivants, nous aurons l'occasion de souligner certains aspects, ici omis, au travers des exemples.

I.5.1 Sérialisabilité multi-niveau

Nous avons vu, en section I.3.2, l'utilisation de graphes de granules pour représenter des groupements logiques ou physiques d'objets. Les transactions multi-niveaux reposent également sur une structuration hiérarchique des objets. Les objets sont décomposés en niveaux d'abstraction, de telle sorte que la structure des objets d'un niveau donné se décrive *uniquement* en objets du niveau inférieur, disjoints [Martin 87] ou indépendants [Cart et al. 90b].

La figure I.6 est extraite de [Weikum 91]. Elle représente une exécution concurrente multi-niveau des transaction T_1 et T_2 qui effectuent respectivement *Transfert*$(x, z, 100)$ et *Transfert*$(y, z, 220)$ (cf. algorithme I.2). Au niveau des *objets bancaires*, les opérations utilisées sont *Débit* et *Crédit*. Les comptes sont représentés au niveau d'abstraction inférieur par des *n-uplets relationnels*. Les opérations *Débit* et *Crédit* sont traduites en sélection et mise-à-jour, « *Select* » et « *Update* ». Enfin, les n-uplets relationnels sont stockés dans des *pages physiques*, les seules opérations pour les manipuler étant *Lire* et *Écrire*.

On remarquera que l'exécution des transactions sur les pages, au niveau 0, n'est pas sérialisable ; il existe des circuits vis-à-vis des transactions T_1 et T_2 dans le graphe de dépendances car x et y sont stockés dans la même page p. Au niveau 1, celui des n-uplets, les transactions

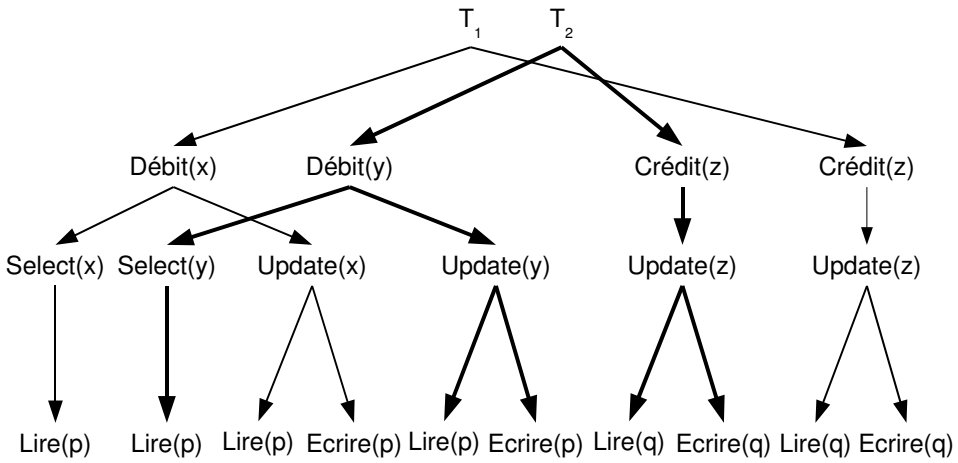


FIG. I.6 – Un exemple d'exécution dans un modèle de transactions à trois niveaux

sont sérialisables dans l'ordre $T_2 \prec T_1$, à cause des deux derniers « *Update* ». Au niveau 2, les transactions sont sérialisables dans n'importe quel ordre grâce à l'auto-commutativité de l'opération *Crédit* (cf. table I.1).

Comme les effets perceptibles pour les transactions se situent au niveau le plus élevé, nous considérerons que T_1 et T_2 sont bien sérialisables dans les deux ordres. Toutefois, pour pouvoir faire abstraction des niveaux inférieurs, il faut assurer que les opérations *Débit* et *Crédit* sont atomiques. Assurer l'atomicité par un traitement en exclusion mutuelle est inacceptable : les opérations des niveaux inférieurs devront pouvoir être enchevêtrées. Ces enchevêtrements devront être contrôlés afin de ne pas introduire d'anomalies : ils devront être sérialisables ! Les opérations d'un certain niveau deviennent donc des transactions au niveau inférieur. Ainsi, de proche en proche, en arrive-t-on à la *théorie de la sérialisabilité multi-niveau*. Nous renvoyons le lecteur à [Weikum 91] pour la présentation formelle de cette théorie. Nous n'en retenons que le résultat essentiel.

Théorème I.6 ([Weikum 91]) *Les transactions multi-niveaux sont sérialisables si elles sont sérialisables* en respectant l'ordre des conflits *à chaque niveau.*

D'après ce théorème, certaines exécutions multi-niveaux sérialisables ne seront pas acceptées car elles ne seront pas sérialisables niveau par niveau. [Weikum 91] en donne un exemple simple. Une transaction T_1 met à jour un objet x qui se trouve dans une page p, tandis qu'une transaction concurrente T_2 lit l'objet x et un objet y qui se trouve dans la même page p. Les exécutions sur les objets sont sérialisables mais on peut trouver des exécutions sur les pages qui, bien que ne créant pas d'anomalies, ne sont pas sérialisables. Par exemple, T_1 et T_2 commencent par lire la page p, puis T_1 l'écrit, donc T_2 précède T_1. Or, T_2 vient ensuite relire cette même page, pour extraire y. Il se produit un interblocage au niveau des pages correspondant à un pseudo conflit, l'ordre de sérialisation étant effectivement $T_2 \prec T_1$.

Toutefois, cette restriction concernant la forme de sérialisabilité n'est pas trop limitative si l'on considère l'argument de monotonie donné par [Yannakakis 84] : la classe des exécutions sérialisables qui vérifient de plus l'hypothèse de monotonie est exactement la classe des exécutions sérialisables respectant l'ordre des conflits.

I.5.2 Contrôle de concurrence multi-niveau

Le théorème I.6 a l'énorme avantage de rendre les niveaux indépendants. Il permet de concevoir indépendamment, niveau par niveau, des contrôles de concurrence adaptés. La solution la plus simple consiste à implémenter un verrouillage à deux phases strict à chaque niveau.

Chaque *opération d'un niveau* $i > 0$ donne naissance à une (sous-)*transaction au niveau* $i-1$. Le fait que les transactions du niveau $i-1$ soient sérialisées assure que l'opération du niveau i est atomique, sans que cela soit réalisé par de l'exclusion mutuelle, trop pénalisante.

Dans sa version stricte, les verrous obtenus par une transaction de niveau i sur les objets de niveau i seront relâchés à la fin de cette transaction, c'est-à-dire à la fin de l'opération de niveau $i+1$. D'une certaine manière, la sérialisabilité des transactions racines échappe ainsi à la classe des histoires 2PL. On notera que l'indépendance des niveaux s'étend aussi à l'indépendance des interblocages : si un interblocage se produit entre deux transactions, il ne peut pas mettre en jeu des objets de niveaux différents.

Un petit problème reste à résoudre. Comme au niveau $i-1$, une transaction peut être rejetée, l'opération qui l'a générée doit prendre une décision : se rejeter elle-même, essayer une alternative ou tout simplement recommencer si le rejet est dû à la guérison d'un interblocage. Dans le dernier cas, pour assurer que l'opération ne sera pas indéfiniment reportée, on peut utiliser une technique à base d'estampille pour assurer la convergence [Rosenkrantz et al. 78].

Une variante intéressante est de fournir un contrôle de concurrence optimiste [Kung & Robinson 81] au niveau 0, celui des pages physiques. En effet, les protocoles optimistes s'avèrent meilleurs que les approches pessimistes dans le cas où les transactions sont courtes car alors il se produit peu de conflits. C'est typiquement le comportement des transactions du niveau 0 dans un modèle multi-niveau. Mais, on peut encore pousser la sophistication plus loin. Des protocoles non-2PL peuvent être introduits dans certains niveaux suivant la structure de données des objets et/ou le comportement des transactions de ce niveau.

La reprise multi-niveau présentée dans la section suivante suppose que le verrouillage 2PL est employé à tous les niveaux.

I.5.3 Reprise multi-niveau

Cet unique théorème I.6 permet aussi d'associer à chaque niveau une technique de reprise différente, mais compatible avec le contrôle de concurrence.

En section I.4, nous avons vu que pour assurer la durabilité d'une transaction, des informations doivent être journalisées pour refaire les effets des transactions validées ; ceci se traduit, lors d'un redémarrage, par une phase « *REDO* ». Pour assurer l'atomicité d'une transaction, des informations doivent être enregistrées pour défaire les effets des transactions rejetées ; ceci donne lieu à une phase « *UNDO* ».

Dans le modèle des transactions multi-niveaux, il est possible de n'effectuer une passe « *REDO* » que pour le niveau 0, puis seulement des passes « *UNDO* » pour les niveaux 1 à n successivement.

Les transactions multi-niveaux, déjà dotées d'un contrôle de concurrence à chaque niveau d'abstraction, sont également munies d'un journal à chaque niveau. Cependant, il y a, dans la proposition de [Weikum et al. 90], une différence notable entre le niveau 0 et les niveaux supérieurs. Le niveau 0 seul assure la durabilité des transactions de tout niveau. Les niveaux supérieurs n'assurent que l'atomicité. Donc, le niveau 0 utilise un journal après et les niveaux supérieurs des journaux avant. De plus, le niveau 0 utilise des images-après, donc une journalisation physique, alors que les niveaux supérieurs utilisent des opérations inverses, c'est-à-dire une journalisation logique.

I.5.3.1 Journalisation au niveau 0

La méthode retenue pour la journalisation du niveau 0 est celle du DB-Cache [Elhardt & Bayer 84]. Vis-à-vis des quatre critères de [Reuter 84], elle est non atomique, *différée*, non forcée, avec points de reprise flous. Le fait que les écritures du niveau 0 soient différées n'a, dans le modèle multi-niveau, aucune incidence fâcheuse. En effet, les transactions du niveau 0 sont très courtes et ne manipulent que quelques pages. Ainsi, au regard des transactions de niveaux supérieurs, la politique globale reste celle du non retardement des écritures.

L'atomicité des transactions de niveau 0 est assurée par le fait que toutes les écritures sont retardées, donc rien n'a été propagé dans la base de données en cas de rejet.

La durabilité des transactions de niveau 0 est assurée par l'écriture *atomique*, lors de la validation, de toutes les images-après des pages modifiées. On enregistre avec chaque image-après l'identifiant de la transaction qui a effectué cette modification. En outre, la dernière page positionne un indicateur booléen. Ainsi, une séquence de pages modifiées ne sera répercutée sur la base que si elle a pu être journalisée en totalité. Ceci est une caractéristique essentielle fournie par la méthode du DB-Cache.

I.5.3.2 Journalisation aux niveaux supérieurs

Les journaux des niveaux supérieurs sont tous des journaux logiques dans lesquels ne sont enregistrées que des opérations inverses et les indicateurs de validation.

Considérons une transaction T de niveau i, avec $i > 0$, qui exécute une opération op sur l'objet x. L'opération op donne elle-même naissance à une sous-transaction ST. Après exécution de op il faut (1) journaliser le quadruplet $(T, op^{-1}, OID(x), ST)$ dans le journal de niveau i, puis (2) valider ST.[14] Une transaction de niveau $i-1 \neq 0$ valide en écrivant l'enregistrement (ST, EOT) dans le journal de niveau $i-1$.[15] Nous avons vu, dans le paragraphe précédent, comment une transaction de niveau 0 valide. Nous verrons plus loin pourquoi une opération doit enregistrer le nom de la sous-transaction qu'elle a générée.

I.5.3.3 Reprise multi-niveau sur rejet

En cas de rejet d'une transaction de niveau 0, il suffit d'éliminer de la mémoire centrale les pages modifiées. (Nous rappelons que la méthode du DB-Cache les fixe en mémoire jusqu'à la validation de la transaction.)

Pour rejeter une transaction de niveau supérieur, il suffit d'appliquer les opérations inverses en parcourant le journal de niveau i en sens inverse. Les opérations inverses sont traitées comme des opérations normales, c'est-à-dire que leurs inverses sont enregistrées dans le journal et que leur exécution se décompose en sous-transactions gérées comme des transactions normales. L'indicateur de fin de transaction sera écrit comme pour une transaction normale en fin de rejet.

I.5.3.4 Reprise multi-niveau sur panne

En cas de panne, et lors du redémarrage, les modifications des transactions du niveau 0 sont refaites, dans l'ordre chronologique, pendant la passe « *REDO* ». Comme il s'agit d'écrire des images-après, l'idempotence est obtenue *de facto*. À la fin de cette phase, la base de données se retrouve dans l'état logique dans lequel elle était au moment précis de la panne. La similitude avec le principe du « *repeating history* » d'ARIES est évidente.

Il faut alors défaire tous les effets des transactions non validées au moment de la panne. Avec le DB-Cache, le niveau 0 est atomique sans avoir à appliquer une phase « *UNDO* ». Les journaux des niveaux supérieurs sont traités séquentiellement de 1 à n. À chaque niveau, le journal est parcouru dans l'ordre chronologique inverse. Chaque fois qu'un enregistrement est rencontré, si son identifiant de transaction ne correspond pas à celui d'une transaction validée (pour laquelle on a trouvé son EOT), l'opération inverse est appliquée. Comme dans le cas du rejet, l'opération inverse est exécutée comme une opération normale.

Une difficulté supplémentaire par rapport au simple rejet est que la panne a pu survenir entre le moment où l'inverse était écrit dans le journal et le moment où la transaction de niveau inférieur aurait dû valider. C'est-à-dire que l'enregistrement $(T, op^{-1}, OID(x), ST)$ est présent dans le journal de niveau i, mais (ST, EOT) n'est pas présent dans le journal de niveau $i-1$.

[14] Seule l'« adresse » de l'objet, l'OID, est utile et surtout pas sa valeur.
[15] EOT pour « *End Of Transaction* ».

Par conséquent, les opérations inverses de ST au niveau $i-1$ ont été exécutées et ont déjà annulées op au niveau i ! C'est pour assurer l'idempotence de op^{-1} que le paramètre ST a été introduite dans les enregistrements des journaux. Lors du parcours du journal de niveau $i-1$, on construit l'ensemble V_{i-1} des transactions validées du niveau $i-1 : V_{i-1} = \{ST | (ST, EOT) \in \text{Journal}_{i-1}\}$. Cet ensemble est transmis à la reprise du niveau i qui pourra ainsi déterminer si l'opération inverse d'une transaction non validée doit vraiment être exécutée. Une opération inverse de niveau i, qui a généré une sous-transaction ST au niveau $i-1$, ne doit être exécutée que si ST a validé, c'est-à-dire si $ST \in V_{i-1}$. Notez que V_i est aussi utilisé au niveau i pour ne *pas* défaire les effets des transactions validées.

Nous finirons cette partie en faisant remarquer que si le modèle des transactions multi-niveaux règle élégamment le problème de l'accès à plusieurs pages, il subsiste un problème vis-à-vis de l'exemple I.10. En effet, seules les opérations qui manipulent *directement* plusieurs pages sont correctement traitées. Dans l'exemple I.10, une opération entraîne la manipulation de plusieurs pages mais par l'intermédiaire d'autres opérations sur d'autres instances. Les aller-retours entre niveaux d'abstractions n'ont, à notre connaissance, pas encore faits l'objet de recherches dans la littérature. Les seules allusions qui sont faites à ce sujet se limitent à dire que les systèmes implémentés pourvoient à ce problème par des techniques *ad hoc*.

I.6 CONCLUSION

Dans ce chapitre nous ne tenions pas, et surtout ne pouvions pas, faire un état de l'art exhaustif.

Nous avons commencé par revoir la théorie de la sérialisabilité en tenant compte des types de données abstraits et de la sémantique de leurs opérations. Les objets typés permettent d'exploiter la commutativité des opérations au lieu de la simple compatibilité des modes *Lire* et *Écrire*. (Nous avons également évoqué un critère plus faible que la commutativité : la recouvrabilité relative.) Les résultats classiques de la théorie de la sérialisabilité ne sont pas invalidés car ils reposent sur la définition des conflits, complémentaire du critère de compatibilité auparavant, et maintenant complémentaire de celui de commutativité.

Ensuite, nous avons présenté les deux principales techniques utilisées pour assurer l'isolation, l'atomicité et la persistance des transactions : le contrôle de concurrence par verrouillage et la reprise par journalisation. Seuls les grands principes ont été décrits, le lecteur pouvant retrouver dans la bibliographie les « détails » auxquels toute implémentation doit faire face.

Dans la dernière section, concernant le modèle des transactions multi-niveaux, nous avons vu qu'un unique théorème permettait de généraliser les résultats des sections précédentes sans aucun problème : chaque niveau doit assurer, indépendamment, contrôle de concurrence et reprise avec n'importe quelles méthodes qui préservent l'ordre des conflits. La solution la plus simple est d'adopter le 2PL strict à chaque niveau. Une variante intéressante est d'utiliser un protocole optimiste au niveau le plus bas, celui des pages physiques.

Dans les chapitres suivants, nous raisonnerons toujours sur des objets logiques, c'est-à-dire que nous nous en remettons à un système à au moins deux niveaux pour pouvoir effectivement implémenter nos propositions (bien qu'une faille, évoquée à la fin des sections 4 et 5, ouvre une voie de recherche).

Chapitre II

Limites de la commutativité

Résumé du chapitre

Ce chapitre présente quelques propriétés de la commutativité, critère proposé pour augmenter la concurrence dans les systèmes transactionnels. Ces propriétés nous permettent de comprendre plus complètement ses avantages et surtout ses limites.

Le principal résultat, *négatif*, de ce chapitre est que la commutativité est sujette, pour des objets arbitraires, aux mêmes limitations intrinsèques que la plus classique compatibilité entre lectures et écritures.

Cependant, la commutativité offre certains avantages intéressants pour la reprise, qui n'ont pas été utilisés dans la littérature et les implémentations. Nous illustrons ces différents points de vue sur des types de données abstraits bien connus.

Nous faisons alors un tour d'horizon des moyens d'obtenir un plus haut degré de commutativité, ou, à défaut, de concurrence.

Finalement, nous comparons nos résultats à ceux de [Weihl 89a], les seuls, à notre connaissance, qui traitent des limites de la sérialisabilité d'un point de vue théorique.

Mots clés

Théorie, sérialisabilité, compatibilité, commutativité, types de données abstraits, contrôle de concurrence, reprise.

II.1 INTRODUCTION

Dans les systèmes multi-utilisateurs, les données partagées sont amenées à être utilisées de façon concurrente [Bernstein et al. 87]. Ces accès se font à l'intérieur d'un programme qui se dénomme une *transaction*. Une transaction est une unité de consistance, c'est-à-dire qu'il s'agit d'un programme qui, partant d'un état des données consistant, en dérive un nouvel état consistant [Gray 81]. Cependant, cette condition n'est pas suffisante quand plusieurs transactions s'exécutent simultanément. Pour augmenter le taux d'utilisation du système, les exécutions parallèles doivent être autorisées ; mais, pour assurer la consistance de la base de données, ces exécutions doivent être équivalentes à une exécution séquentielle. Ce critère de correction des exécutions concurrentes est la *sérialisabilité*.

Les interactions entre transactions se produisent au travers de l'utilisation commune de certains objets de la base ; certaines interactions sont autorisées (comme la lecture simultanée de la valeur d'un objet par plusieurs transactions), alors que d'autres sont prohibées car elles pourraient conduire à un état incohérent (par exemple, une transaction ne peut pas lire une valeur qui vient d'être modifiée par une transaction encore en cours d'exécution). Les interactions prohibées sont appelées des *conflits* et génèrent des *dépendances* entre transactions. Les premiers conflits ont été définis entre des modes d'accès de sémantique très large, typiquement des lectures et écritures, voire même seulement des écritures [Kedem & Silberschatz 83]. Ce critère s'appelle la *compatibilité*. Cependant, la compatibilité s'est très vite révélée être un très mauvais critère pour les données utilisées intensivement [O'Neil 86]. Ainsi, un autre critère a été introduit : la *commutativité*. La commutativité prend en compte la sémantique des accès aux objets dans le but de diminuer le nombre de conflits qui apparaissent avec la seule compatibilité, et qui ne sont donc que des *pseudo conflits* [Schwarz & Spector 84].

Pour nous donner une idée intuitive des avantages de la commutativité par rapport à la compatibilité, prenons l'exemple de l'ADT RÉPERTOIRE [Schwarz & Spector 84].

Exemple II.1 *La structure d'un répertoire est une liste d'entrées contenant des noms différents et des données associées à ces noms, comme un répertoire de fichiers dans un système d'exploitation. Parmi les opérations nous ne retiendrons que* Créer *et* Supprimer *une entrée (un fichier dans un système d'exploitation). Chacune de ces opérations modifie le répertoire, donc elles sont toutes deux des accès en écriture à l'objet. Par conséquent,* Créer *et* Supprimer *sont en conflit et ne peuvent pas s'exécuter concurremment avec le critère de compatibilité.*

La commutativité permet d'avoir une vue affinée de ces opérations, et, grâce à la prise

en compte de leur sémantique, conclut que Créer *et* Supprimer *ne sont en conflit que si elles manipulent la même entrée. Il est alors évident que le nombre de conflits qui surviendront à l'exécution peut être considérablement réduit.*[1]

Si les avantages de la commutativité ont été souvent illustrés par des exemples adéquats, en revanche, ses éventuelles limites n'ont fait l'objet, à notre connaissance, d'aucune investigation théorique. Seuls les travaux de [Weihl 89a], qui ne concernent pas directement la commutativité, font mention de limites à la concurrence sur les ADT. Nous ne faisons pas ici allusion aux études, par simulation ou modèle analytique, du comportement des transactions vis-à-vis de charges du système, de leur taille, de probabilités de conflits, ou encore de protocoles particuliers [Agrawal & DeWitt 85] [Agrawal et al. 87] [Carey & Livny 89] [Franaszek & Robinson 85] [Thomasian 91] [Thomasian & Ryu 91].

Les questions auxquelles ce chapitre apporte une réponse sont :

– « La commutativité n'étant certainement pas une panacée, quelles sont ses limites ? »
– « Mais alors, quels sont aussi ses avantages ? »
– « Enfin, où se montre-t-elle le plus efficace ? »

Pour y répondre, nous allons présenter quelques « *folk-theorems*[2] » [Harel 80] relatifs à la commutativité des opérations sur les ADT. Les résultats sont essentiellement dus à notre choix de modèle formel ; nous avons adopté une approche fonctionnelle des opérations sur les ADT, qui valorise la notion d'ensembles de départ et d'arrivée. Ce choix étant fait, les résultats sont alors d'une simplicité déconcertante.

Tout d'abord, nous introduisons notre modèle et mettons en évidence quatre conditions que les opérations sur un ADT doivent satisfaire. Ensuite, nous prouvons que pour des objets arbitraires, la commutativité n'est pas beaucoup plus puissante que la compatibilité. Nous prouvons également quelques propriétés avantageuses de la commutativité, la dernière d'entre elles ayant des implications pratiques intéressantes pour la reprise qui n'ont pas été exploitées dans la littérature. Finalement, nous faisons un tour d'horizon des moyens que l'on peut employer pour échapper, autant que faire se peut, aux limites de la commutativité. Toutes contournent le problème en exploitant des propriétés supplémentaires ou en affaiblissant les conditions fixées.

[1] règle des 20 % de données accédées par 80 % des transactions !
[2] théorèmes « bien connus » (?)

II.2 LE MODÈLE

Nous étudierons les propriétés de la commutativité sur les ADT en utilisant un formalisme fonctionnel de leurs opérations.

II.2.1 Opérations sur les types de données abstraits

Chaque opération sur un ADT s'exprime comme une ou plusieurs fonctions. Ce premier point est fondamental, et réalisable en pratique [Malta & Martinez 91b] [Malta 93]. La nécessité de cette approche sera davantage compréhensible à partir de la section II.2.3. Mais, illustrons-là sur un exemple.

Exemple II.2 *Sur l'ADT* ENSEMBLE*, considérons l'opération* Insérer(x)*. Nous distinguerons deux fonctions : la première sera définie de l'ensemble des* ENSEMBLE *contenant x dans lui-même, et la seconde depuis les ensembles ne contenant pas x vers ceux qui le contiennent. La première ne modifie pas la valeur de l'objet manipulé, alors que la seconde correspond à une insertion effective.*

Définition II.1 *Nous définissons $\mathcal{F} = (f_i)_{i=1}^n$ une famille de fonction avec :*

$$f_i : A_i \rightarrow B_i$$

et posons :

- $A_f = \bigcap_{i=1}^n A_i$;
- $B_{\mathcal{F}} = \bigcap_{i=1}^n B_i$.

II.2.2 Composition (C1)

Une famille est censée regrouper des fonctions qui commutent. En conséquence, ces fonctions doivent être composables dans n'importe quel ordre. Pour autoriser toutes les compositions, une condition est nécessaire sur leurs ensembles de départ et d'arrivée.

Définition II.2 *Soit $\mathcal{F}$ une* famille de fonctions deux à deux commutatives*, alors $\mathcal{F}$ doit vérifier la condition de composition (C1) :*

$$\begin{aligned} &\forall i : 1 \leq i \leq n, \\ &\forall j : (1 \leq j \leq n) \wedge (j \neq i), \\ &\quad B_j \subseteq A_i. \end{aligned}$$

$A_i \to B_i$	$\{a\}$	$\{b\}$	$\{c\}$
$\{a\}$	Id Id^{-1}	$f_1 : \{a\} \to \{b\}$ $f_1^{-1} : \{b\} \to \{a\}$	$f_2 : \{a\} \to \{c\}$ $f_2^{-1} : \{c\} \to \{a\}$
$\{b\}$	$f_3 : \{b\} \to \{a\}$ $f_3^{-1} : \{a\} \to \{b\}$	Id Id^{-1}	$f_4 : \{b\} \to \{c\}$ $f_4^{-1} : \{c\} \to \{b\}$
$\{c\}$	$f_5 : \{c\} \to \{a\}$ $f_5^{-1} : \{a\} \to \{c\}$	$f_6 : \{c\} \to \{b\}$ $f_6^{-1} : \{b\} \to \{c\}$	Id Id^{-1}

TAB. II.1 – Exemple de fonctions associées à une affectation

II.2.3 Inversibilité (C2)

Les effets d'une opération peuvent devoir être défaits, soit à la suite d'un rejet de la transaction correspondante, soit après une panne du système. Nous imposons donc que chaque fonction ait une fonction inverse à gauche.

Définition II.3 *Soit $\mathcal{F}$ une famille de fonctions deux à deux commutatives, alors $\mathcal{F}$ doit admettre $\mathcal{F}^{-1} = (f_i^{-1})_{i=1}^n$ une* famille de fonctions inverses à gauche *:*

$$f_i^{-1} : B_i \to A_i$$

c'est-à-dire telle que le couple $\langle \mathcal{F}, \mathcal{F}^{-1} \rangle$ vérifie la condition d'inversibilité (C2) :

$$\begin{aligned} &\forall i : 1 \leq i \leq n, \\ &\forall x \in A_i, \\ &\quad f_i^{-1} \circ f_i(x) = x. \end{aligned}$$

Cette condition peut toujours être satisfaite, même pour les opérations qui ne sont que des écritures aveugles. Dans ce cas, il faut restreindre les A_i à des singletons et avoir par conséquent $|A|^2 - |A| + 1$ fonctions, A étant l'union des A_i.

Exemple II.3 *En théorie, si A est un ADT admettant trois valeurs, disons a, b et c, nous pouvons considérer l'opération d'affectation comme sept fonctions constantes, présentées en table II.1. Chaque fonction possède bien une fonction inverse à gauche parfaitement définie.*

En pratique, il ne s'agit ni plus ni moins que de la méthode de reprise avec images avant [Gray 78] [Gray 81] [Kohler 81].

II.2.4 Commutativité de l'état (C3)

La définition suivante introduit la première partie de la seule condition qui est explicitement requise dans la plupart des articles.

Définition II.4 *Soit $\mathcal{F}$ une famille de fonctions deux à deux commutatives, alors $\mathcal{F}$ doit vérifier la condition de commutativité de l'état (C3) :*

$$\begin{array}{l}\forall i : 1 \leq i \leq n, \\ \forall j : (1 \leq j \leq n) \wedge (j \neq i), \\ \forall x \in A_i \cap A_j, \\ \quad f_j \circ f_i(x) = f_i \circ f_j(x).\end{array}$$

Nous ne tenterons pas de formaliser plus avant la notion d'équivalence d'états puisque l'égalité est déjà une relation d'équivalence et n'a donc, à notre sens, qu'à être définie convenablement pour chaque ADT. ([Weihl 88] en donne une définition mettant en œuvre l'ensemble des exécutions futures.)

II.2.5 Commutativité des vues (C4)

Dans les définitions précédentes, seuls les paramètres d'entrée des opérations sur un ADT sont, implicitement, exprimées. Nous exprimons explicitement les paramètres de sortie.

Définition II.5 *Soit $\mathcal{F}$ une famille de fonctions deux à deux commutatives, alors $\mathcal{F}$ doit admettre $\mathcal{F}^T = (f_i^T)_{i=1}^n$ une famille de fonctions définie comme suit :*

$$f_i^T : B_i \rightarrow E_i$$

où f_i^T n'est également définie qu'à gauche. $f_i^T \circ f_i(x)$ est la vue que la transaction T obtient après application de l'opération associée à f_i sur la valeur courante de l'objet x.

Cette formalisation des paramètres de sortie est correcte car chaque fonction possède une fonction inverse à gauche. Supposons en effet que la forme suivante soit une formalisation plus directe d'une opération sur un ADT :

$$op : \begin{array}{ccc} A & \rightarrow & B \times E \\ x & \mapsto & (op_1(x), op_2(x)) \end{array}$$

alors, f correspond directement à op_1, et f^T à $op_2 \circ op_1^{-1}$, s'il n'existe pas de raccourci.

Nous donnons ci-dessous une dernière condition, en général présentée conjointement à la précédente, concernant l'invariance des paramètres de sortie.

Définition II.6 *Soit $\mathcal{F}$ une famille de fonctions deux à deux commutatives, alors chaque couple $(\mathcal{F}, \mathcal{F}^T)$ doit vérifier la condition d'indépendance des vues (C4) :*

$$\begin{aligned}
&\forall i : 1 \leq i \leq n, \\
&\forall j : (1 \leq j \leq n) \wedge (j \neq i), \\
&\forall x \in A_i \cap A_j, \\
&\quad f_i^T \circ f_i \circ f_j(x) = f_i^T \circ f_i(x).
\end{aligned}$$

Notons tout de suite que les opérations non déterministes ne sont pas prises en compte [Hesselink 88]. L'extension au non-déterminisme n'est pas directe : d'une part, la sérialisabilité peut être perdue, mais d'autre part on n'évite pas l'existence d'opérations non déterministes. Si l'on peut montrer que les valeurs non déterministes renvoyées par certaines opérations ne changent pas le comportement des transactions, alors tous les résultats s'appliquent également à celles-ci.

Exemple II.4 *Un exemple d'opération non déterministe impossible à éliminer est fourni par la gestion dynamique de la mémoire. La création d'une nouvelle instance dans un système à objets, lui affecte un identifiant unique, son OID (pour «* Object IDentifier *»), par exemple un numéro de page sur disque concaténé à un numéro d'entrée dans la page. Les OID sont affectés de manière non déterministe puisque les emplacements libres dépendent du nombre de transactions concurrentes créant ou détruisant des instances. Cependant, on peut affirmer que le comportement d'une transaction n'est pas troublé par des détails d'aussi bas niveau puisque la seule chose que l'on demande à un OID est d'être unique.*

II.2.6 Remarques sur le modèle

De nombreuses remarques peuvent être faites sur cette formalisation.

$\mathcal{F}$ est une famille de fonctions plutôt qu'un ensemble car une fonction ne commute pas nécessairement avec elle-même. Avec une famille, et si c'est le cas, il suffit de dupliquer l'opération auto-commutative.

Exemple II.5 *La figure II.1 donne le graphe de la relation de compatibilité des verrous associées au protocole de verrouillage hiérarchique de [Gray 78].*

Les familles canoniques qui en résultent sont :

- $\mathcal{F}_1 = (IS, IS, S, S)$;
- $\mathcal{F}_2 = (IS, IS, IX, IX)$;
- $\mathcal{F}_3 = (IS, IS, SIX)$;
- $\mathcal{F}_4 = (X)$.

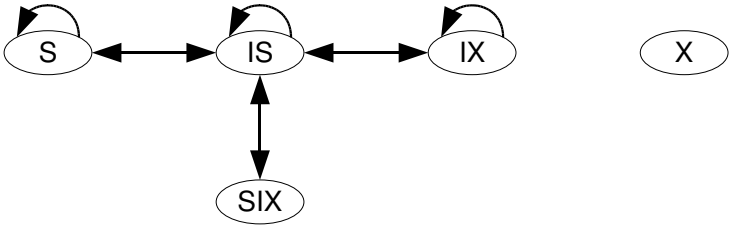


FIG. II.1 – Graphe de la relation de compatibilité des verrous hiérarchiques

Elles représentent les cliques maximales du graphe, les sommets réflexifs étant dupliqués. Toutes les familles obtenues en dupliquant un nombre quelconque de fois les sommets réflexifs sont également des familles de fonctions deux à deux commutatives.

Implicitement, chaque opération est exécutée pour le compte d'une transaction différente. Cette hypothèse simplifie le modèle. De plus, une séquence de fonctions exécutée pour une même transaction est réductible, par composition, à une seule fonction. Cette réduction peut même, en l'absence d'une possibilité de rejet partiel de la séquence, permettre plus de parallélisme. Il suffit pour cela que certaines opérations de la séquence ne commute pas avec les opérations concurrentes d'autres transactions mais que ce soit le cas pour la séquence complète.

Exemple II.6 *Un exemple trivial consiste pour une transaction à défaire d'elle-même une opération antérieure. Si une transaction exécute* Supprimer(x) *alors qu'elle a préalablement invoqué un* Insérer(x) *réussi, cette séquence se résume à la fonction identité, et commute trivialement avec n'importe quelle autre opération.*

Sur un compteur, si la transaction exécute Incrémenter(10) *et plus tard* Décrémenter(3)*, cette séquence se ramène à la seule opération* Incrémenter(7).

Pour des fonctions plus complexes, de telles équivalences seront vraisemblablement trop difficiles à établir.

De façon à simplifier le modèle, et parce qu'il ne s'agit que d'une expression mathématique des opérations sur les ADT qui n'a pas de lien direct avec une implémentation, toutes les valeurs possibles de paramètres d'entrée génèrent autant de fonctions [Roesler & Burkhard 87].

Exemple II.7 *Il n'y a pas seulement deux opérations* Insérer *(une effective et une inutile), comme présenté dans l'exemple II.2, mais une infinité de couples :* Insérer$^{x_1}_{ok}$, Insérer$^{x_1}_{\neg ok}$, Insérer$^{x_2}_{ok}$, Insérer$^{x_2}_{\neg ok}$*, etc., où l'exposant représente le paramètre d'entrée et l'indice celui de sortie.*

Est également implicite le fait que la commutativité que nous étudions est conditionnelle. Ceci est dû, en partie seulement, au fait que les paramètres d'entrée génèrent autant de fonctions qu'il y a de valeurs possibles. Indirectement, ceci est également dû à la condition C2 sur l'existence de fonctions inverses à gauche. En pratique, il arrive souvent que l'opération inverse ne

soit déterminée qu'après l'exécution de son opération directe ; cela signifie que les paramètres de sortie sont aussi exploités.

Exemple II.8 *Au moment où l'opération* Insérer(x) *est appliquée sur une instance de l'ADT* ENSEMBLE*, on ne sait pas encore s'il s'agit d'une insertion effective ou inopérante. La détermination de l'opération inverse ne pourra se faire qu'après l'exécution de l'opération directe, grâce au paramètre de sortie (ok ou* $\neg$ *ok).*

La condition C2 sur l'existence d'opérations inverses n'est nécessaire que pour les méthodes de contrôle de concurrence qui effectuent des mises-à-jour immédiates sur l'objet partagé. Dans le cas des méthodes utilisant espace de travail et listes d'intentions, comme la commutativité en avant [Weihl 88], l'absence de cette condition permet d'accepter davantage de fonctions commutatives (cf. exemple II.9). Toutefois, les difficultés d'implémentation sont plus grandes avec ce genre de méthodes qu'avec les mises-à-jour immédiates : (1) chaque transaction doit maintenir sa propre version de l'objet et/ou la liste des opérations effectuées, (2) lors de la validation, les opérations doivent être répétées sur l'objet ainsi que dans les espaces de travail des transactions concurrentes.

De toute façon, cette condition n'intervient que dans la section II.4.3. Donc, les limites de la commutativité, présentées en section II.3, s'appliquent aux deux formes de mises-à-jour.

Exemple II.9 *Sur un ADT* BOOLÉEN*, considérons la mise à vrai et la mise à faux.*

Avec mises-à-jour immédiates, nous avons besoin de la condition C2 pour défaire l'opération en cas de rejet. Nous obtenons alors les fonctions :

- Mettre$^{vrai}_{ok} : \{faux\} \rightarrow \{vrai\}$;
- Mettre$^{vrai}_{inutile} : \{vrai\} \rightarrow \{vrai\}$;
- Mettre$^{faux}_{ok} : \{vrai\} \rightarrow \{faux\}$;
- Mettre$^{faux}_{inutile} : \{faux\} \rightarrow \{faux\}$.

Et, nous obtenons les familles canoniques suivantes :

- $\mathcal{F}_1$ = (Mettre$^{vrai}_{ok}$) ;
- $\mathcal{F}_2$ = (Mettre$^{vrai}_{inutile}$, Mettre$^{vrai}_{inutile}$) ;
- $\mathcal{F}_3$ = (Mettre$^{faux}_{ok}$) ;
- $\mathcal{F}_4$ = (Mettre$^{faux}_{inutile}$, Mettre$^{faux}_{inutile}$).

Avec mises-à-jour différées, l'existence d'une opération inverse n'est pas requise puisque la valeur dans la base de l'objet est toujours la dernière valeur validée. Les opérations directes seront répétées lors de la validation. Donc, les fonctions directes ne doivent plus être injectives. Aussi, pouvons-nous nous contenter de deux fonctions :

- Mettre$^{vrai} : \{faux, vrai\} \rightarrow \{vrai\}$;
- Mettre$^{faux} : \{faux, vrai\} \rightarrow \{faux\}$.

D'où, nous avons seulement deux familles canoniques :

– $\mathcal{F}_1 = (\textit{Mettre}^{vrai}, \textit{Mettre}^{vrai})$;
– $\mathcal{F}_2 = (\textit{Mettre}^{faux}, \textit{Mettre}^{faux})$.

Avec mises-à-jour immédiates, une seule transaction peut effectivement modifier la valeur du booléen. Avec mises-à-jour différées, les affectations concurrentes d'une même valeurs sont possibles car elles sont idempotentes.

L'exemple du booléen s'étend directement à celui de l'ADT ENSEMBLE *(cf. algorithme II.1) qui est donc plus commutatif avec mises-à-jour différées qu'avec mises-à-jour immédiates, comme l'a noté [Weihl 88].*

La commutativité que nous avons définie n'englobe pas la totalité de la commutativité en arrière *à droite* de [Weihl 88]. Nous détaillerons cet aspect en section II.6.

Notre formalisme ne prend en compte que les ADT. C'est pourquoi nous avons implicitement identifié la notion d'instance à celle de variable. Par conséquent, les résultats présentés ci-dessous sont directement applicables aux transactions multi-niveaux où les objets sont composés d'objets du niveau d'abstraction sous-jacent. À l'opposé, ils ne peuvent pas être appliqués tels quels aux objets composés de références à d'autres objets.

Enfin, les familles que nous avons définies sont sérialisables parce qu'elles sont exemptes de conflits. Plus exactement, elles décrivent l'état instantané d'une exécution stricte où l'objet n'existe qu'en un seul exemplaire (ceci exclut donc les techniques à base de multi-version).

Dans les deux sections suivantes, nous allons respectivement présenter les limites et les avantages de la commutativité comme critère d'accès concurrent à des données partagées, sous les conditions C1 à C4 et donc toutes les hypothèses que nous venons de parcourir.

II.3 LIMITES POUR LA CONCURRENCE

Tout d'abord, mettons les inconvénients en lumière. Ils concernent le parallélisme qui est bien moins accru qu'on ne peut le croire à partir de l'exemple II.1 (cf. exemple II.19 et algorithme II.1).

II.3.1 Quelques « *folk-theorems* »

Nous commencerons par quelques lemmes techniques dérivés de la contrainte sur les ensembles de départ et d'arrivée des fonctions commutatives, c'est-à-dire la condition C1.

Lemme II.1 *Soit $\mathcal{F}$ une famille vérifiant C1, avec $n \geq 2$, alors :*

$$\begin{aligned} &\forall i : 1 \leq i \leq n, \\ &\quad B_i \subseteq A_{\mathcal{F}\setminus(f_i)}. \end{aligned}$$

Preuve II.2 *Immédiate par définition de la condition C1.*

Lemme II.3 *Soit $\mathcal{F}$ une famille vérifiant C1, avec $n \geq 2$, alors :*

$$\begin{aligned} &\forall i : 1 \leq i \leq n, \\ &\forall j : (1 \leq j \leq n) \wedge (j \neq i), \\ &\quad B_i \cap B_j \subseteq A_{\mathcal{F}}. \end{aligned}$$

Preuve II.4 *$B_i \subseteq A_{\mathcal{F}\setminus(f_i)}$ et $B_j \subseteq A_{\mathcal{F}\setminus(f_j)}$ [lemme II.1] d'où $B_i \cap B_j \subseteq A_{\mathcal{F}\setminus(f_i)} \cap A_{\mathcal{F}\setminus(f_j)}$.*

Ces lemmes nous amènent directement à un tout premier et bien simple théorème.

Théorème II.5 *Soit $\mathcal{F}$ une famille vérifiant C1, avec $n \geq 2$, alors :*

$$B_{\mathcal{F}} \subseteq A_{\mathcal{F}}.$$

Preuve II.6 *Évident à partir du lemme II.3.*

Le théorème II.5 résume que la condition C1, *nécessaire*, entraîne une relation très restrictive entre les ensembles de départ et d'arrivée d'une famille de fonctions deux à deux commutatives. Cette structure très forte n'est pas du tout étonnante dès que l'on réalise qu'elle résulte d'intersections et d'inclusions d'ensembles. Malgré sa désarmante simplicité, c'est la raison inévitable

qui fait de la commutativité un bien faible progrès pour des objets *arbitraires*. Intrinsèquement, ce théorème n'a pas d'autre intérêt, mais nous en verrons toutes les implications lorsqu'il est connecté au théorème suivant.

Pour démontrer le théorème II.9, nous avons besoin d'un lemme fondamental nécessitant la condition C3 sur la commutativité de l'état. Ce lemme sera étendu dans la section suivante car il constitue un avantage bien connu de la commutativité.

Définition II.7 *Nous écrivons $\pi(\{1,\ldots,n\})$ l'ensemble des permutations de $\{1,\ldots,n\}$.*

Lemme II.7 *Soit $\mathcal{F}$ une famille vérifiant C1 et C3, alors :*

$$\begin{aligned}&\forall i : 1 \leq i \leq n,\\ &\forall (k_1,\ldots,k_{n-1}) \in \pi(\{1,\ldots,n\} \setminus \{i\}),\\ &\forall x \in A_{\mathcal{F}},\\ &\quad \alpha \circ f_i(x) = f_i \circ \alpha(x).\end{aligned}$$

avec $\alpha = f_{k_{n-1}} \circ \cdots \circ f_{k_1}$.

Preuve II.8 *Procédons par récurrence sur le nombre d'opérations dans la famille. La propriété est triviale pour $n = 1$. Supposons qu'elle soit vraie pour $n-1$ et montrons qu'elle le reste pour n : $\alpha \circ f_i(x) = \beta \circ f_j \circ f_i(x) = \beta \circ f_i \circ f_j(x)$ [condition C3], or $A_{\mathcal{F}} \subseteq A_j$, $f_j(x) \in B_j$ et $B_j \subseteq A_{\mathcal{F}\setminus(f_j)}$ [lemme II.1], donc $\beta \circ f_i(f_j(x)) = f_i \circ \beta(f_j(x))$ [hypothèse de récurrence] $= f_i \circ \alpha(x)$.*

Théorème II.9 *Soit $\mathcal{F}$ une famille vérifiant C1 et C3, alors :*

$$\begin{aligned}&\forall x \in A_{\mathcal{F}},\\ &\quad f_n \circ \cdots \circ f_1(x) \in B_{\mathcal{F}}.\end{aligned}$$

Preuve II.10 *Procédons par récurrence sur le nombre d'opérations dans $\mathcal{F}$. La propriété est vérifiée par définition pour $n = 1$.*

Montrons qu'elle est vraie pour $n = 2$: $\mathcal{F} = (f_1, f_2)$, alors $f_1(f_2(x)) \in B_2$, $f_2(f_1(x)) \in B_1$, or $f_1 \circ f_2(x) = f_2 \circ f_1(x)$ [condition C3], donc $f_1 \circ f_2(x) \in B_1 \cap B_2$.

Supposons maintenant qu'elle soit vraie pour $n-1$ est montrons qu'elle l'est également pour n : $f_n \circ \cdots \circ f_1(x) = f_n \circ \alpha(x) = \alpha \circ f_n(x)$ [lemme II.7]. En utilisant le même raisonnement que pour $n = 2$, $\alpha(x) \in B_{\mathcal{F}\setminus(f_n)}$, et $B_{\mathcal{F}\setminus(f_n)} \subseteq A_{\mathcal{F}} \subseteq A_n$ [lemme II.3], donc $f_n(\alpha(x)) \in B_n$. Réciproquement, $f_n(x) \in B_n$, et $B_n \subseteq A_{\mathcal{F}\setminus(f_n)}$ [lemme II.1], donc $\alpha(f_n(x)) \in B_{\mathcal{F}\setminus(f_n)}$. D'où, $f_n \circ \alpha(x) \in B_{\mathcal{F}\setminus(f_n)} \cap B_n$.

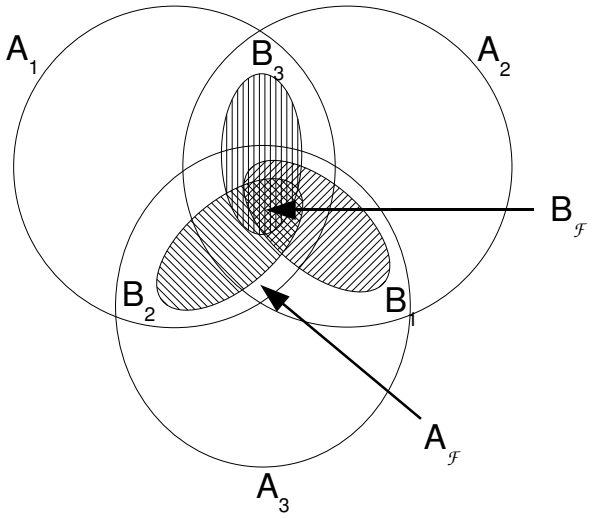


FIG. II.2 – « Théorème » sur les limites de la commutativité

Le théorème II.9 exprime le fait que la valeur finale résultant de l'application d'une famille de fonctions deux à deux commutatives appartient à $B_{\mathcal{F}}$. Il est connu qu'une relation ne préserve pas l'intersection : $\mathcal{R}(E \cap F) \subseteq \mathcal{R}(E) \cap \mathcal{R}(F)$ [Berge 67][3] [Bouvier 82]. Ce théorème étend cette propriété à une composition de fonctions *vérifiant* C1 et C3 : $f_n \circ \cdots \circ f_1 \left(\bigcap_{i=1}^{n} A_i\right) \subseteq \bigcap_{i=1}^{n} f_i(A_i)$.

II.3.2 Discussion

Nous sommes maintenant en mesure de discuter toutes les implications, néfastes, du théorème II.9. Il a deux interprétations : l'une quand $n \geq 2$ et une seconde pour $n = 1$. Elles ont un point en commun : toutes deux limitent la concurrence.

Quand deux transactions au moins s'exécutent concurremment sur le même objet, c'est-à-dire quand $n \geq 2$, le théorème II.5 s'applique également. Par conséquent, nous en déduisons que la commutativité d'une famille de fonctions deux à deux commutatives implique la *convergence*, ou tout au moins la monotonie, dans la suite de valeurs que peut prendre l'objet puisque la valeur initiale doit se trouver dans $A_{\mathcal{F}}$ et que la valeur finale appartiendra à $B_{\mathcal{F}}$, un sous-ensemble. En fait, il y a même double convergence puisque le nombre croissant d'opérations restreint simultanément $A_{\mathcal{F}}$. La figure II.2 en donne une « démonstration » graphique pour $n = 3$. (Cette illustration est aisément construite informellement à partir de la seule condition C1.)

C'est cette convergence des valeurs successives qui permet de mettre en œuvre le mécanisme des valeurs déduites dans notre prototype [Malta & Martinez 91b] [Malta 93]. Elle garantit qu'aucune opération concurrente ne remettra en cause les valeurs déduites, c'est-à-dire obtenues sans exécution de l'opération, des paramètres de retour de l'opération en cours. Par exemple, si

[3] Présenté, en page 4, sous la forme : $\Gamma\left(\bigcap_{i=1}^{n} A_i\right) \subseteq \bigcap_{i=1}^{n} \Gamma(A_i)$ où Γ est une application de X dans X et les A_i des sous-ensembles de X.

l'opération *Vide* a été demandée sur une instance de l'ADT PILE et que celle-ci a retourné la valeur « vrai », alors une opération *Dépiler* arrivant ultérieurement pourra signaler, sans même s'être exécutée mais en se fiant aux résultat retourné par *Vide*, qu'elle a échouée.

Ce qui est loin d'être intuitif dans cette proposition est que la vue que les transactions ont de l'exécution des opérations ne joue aucun rôle. En effet, même si la condition C2 sur les fonctions inverses amène à considérer, en partie, les paramètres de sortie, rien n'oblige à ce que ces valeurs soient transmises aux transactions. Il eut été plus simple de comprendre que chaque fois qu'une opération renvoie une partie de l'état de l'objet, cette vue ne devant plus être remise en cause, elle limite les modifications ultérieures.

Exemple II.10 *L'opération Insérer(x) sur l'ADT* ENSEMBLE *doit se décomposer en deux fonctions (cf. exemples II.2, II.7 et II.8). Toutefois, le paramètre de sortie, ok ou $\neg$ ok, s'il est nécessaire au système pour savoir comment éventuellement défaire l'opération, n'a pas nécessairement à être transmis à la transaction.*

C'est le rôle principal de l'interface à deux niveaux présentée dans [Malta & Martinez 91b] [Malta 93] : récupérer tous les paramètres de sortie nécessaires à la détermination de l'opération inverse quand ils ne font pas parti de l'interface publique de l'ADT.

Si l'on reconsidère l'exemple II.1 sur l'ADT RÉPERTOIRE, il n'est pas évident de convaincre quelqu'un que la commutativité soit si restrictive ! Bien sûr, cet exemple introductif a été choisi avec soin. Mais, même cet ADT-là ne peut pas échapper aux limites énoncées.

Exemple II.11 *Ajoutons l'opération Cardinal à l'ADT* RÉPERTOIRE *; elle retourne le nombre d'entrées utilisées. Il s'agit d'une opération assez restrictive puisque aucune insertion ou suppression effectives ne peuvent se faire en concurrence.*

Si nous ajoutons encore l'opération Vider, consistant à supprimer toutes les entrées, nous nous rendons compte que cette dernière est même exclusive.

En fait, les objets pour lesquels on peut espérer que la convergence sera assez, voire très lente, peuvent toujours posséder des opérations qui l'accélèrent dans des proportions considérables. La clef de la compréhension de ce résultat réside dans le fait qu'il s'applique aux opérations qui manipulent les objets *comme un tout*. Ainsi, dans l'exemple précédent, l'opération *Cardinal* renvoie-t-elle une valeur qui reflète une propriété globale de l'objet alors que les opérations *Créer* et *Supprimer* ne s'attachent qu'à l'un des éléments constituant le répertoire. Nous y reviendrons en section II.5.1.

La seconde interprétation du théorème II.9, c'est-à-dire quand il n'y a qu'une seule transaction, n'est pas plus enthousiasmante. C'est le seul cas dans lequel la valeur d'un objet peut être substantiellement modifiée car la condition C1 n'est pas effective. En d'autres termes, les

Vide	$\textit{Vide}_{oui}$	$: \mathbb{P}^0 \rightarrow \mathbb{P}^0$
	$\textit{Vide}_{non}$	$: \mathbb{P}^+ \rightarrow \mathbb{P}^+$
Vider	$\textit{Vider}_{oui}$	$: \mathbb{P}^+ \rightarrow \mathbb{P}^0$
	$\textit{Vider}_{inutile}$	$: \mathbb{P}^0 \rightarrow \mathbb{P}^0$
Dépiler	$\textit{Dépiler}_{vide}$	$: \mathbb{P}^0 \rightarrow \mathbb{P}^0$
	$\textit{Dépiler}_{dernier}$	$: \mathbb{P}^+ \rightarrow \mathbb{P}^0$
	$\textit{Dépiler}_{oui}$	$: \mathbb{P}^+ \rightarrow \mathbb{P}^+$
Empiler	$\textit{Empiler}_{premier}$	$: \mathbb{P}^0 \rightarrow \mathbb{P}^+$
	$\textit{Empiler}_{oui}$	$: \mathbb{P}^+ \rightarrow \mathbb{P}^+$

TAB. II.2 – Fonctions associées aux opérations de l'ADT PILE

fonctions qui sont définies d'un ensemble A vers un ensemble B disjoint sont exclusives. C'est un corollaire du théorème II.5.

Corollaire II.11 *Soit $\mathcal{F}$ une famille de fonctions vérifiant C1 et C3, et $f : A \rightarrow B$ une fonction de $\mathcal{F}$ telle que $A \cap B = \emptyset$, alors :*

$$A_{\mathcal{F}} \neq \emptyset \rightarrow |\mathcal{F}| = 1.$$

Preuve II.12 *Procédons par contradiction et choisissons $f \in \mathcal{F}$ avec $|\mathcal{F}| > 1$ et $A \cap B = \emptyset$, alors, par définition, $A_{\mathcal{F}} \subseteq A$ and $B_{\mathcal{F}} \subseteq B$, ce qui contredit clairement le théorème II.5, sauf quand $A_{\mathcal{F}} = \emptyset$.*

La condition C1 et ce corollaire nous donnent une idée pour détecter tous les couples de fonctions qui ne peuvent pas commuter, indépendamment de leur sémantique. Ils sont utiles car le nombre de fonctions est bien plus grand que le nombre d'opérations (cf. table II.2) et que les fonctions associées à une opération sont généralement exclusives les unes des autres.

Exemple II.12 *Développons entièrement l'ADT* PILE*. Nous distinguons neuf fonctions, associées à quatre opérations :* Vide*,* Vider*,* Empiler *et* Dépiler*. Ces fonctions, données en table II.2, sont définies sur les ensembles $\mathbb{P}^0$ et $\mathbb{P}^+$ représentant respectivement le singleton contenant la pile vide et l'ensemble de toutes les piles non vides.*

Les raisons des choix faits en table II.2 sont de deux ordres. Premièrement, la condition C2 impose l'unicité de l'opération inverse, or Vider_{oui} *et* $\text{Vider}_{inutile}$ *ont des inverses différents.*

Deuxièmement, certaines différenciations permettent d'augmenter, très légèrement bien sûr, le parallélisme, comme $\text{Empiler}_{premier}$ *et* Empiler_{oui} *car une unique fonction les regroupant aurait été exclusive.*

Dans la table II.3, tous les couples d'opérations qui ne vérifient pas la condition C1 sont marqués avec i comme impossible. Ensuite, les trois fonctions exclusives sont marquées en ligne et colonne avec i'. Enfin, le couple ($\text{Dépiler}_{dernier}$*,* $\text{Empiler}_{premier}$*) est marqué i'' car l'intersection de leurs ensembles de départ, aussi bien que celle de leurs ensembles d'arrivée, $\mathbb{P}^+$ et $\mathbb{P}^0$,*

		Vide		*Vider*		*Dépiler*			*Empiler*	
		oui	non	oui	inutile	vide	dernier	oui	premier	oui
Vide	oui		i	i’			i’	i	i’	i
	non	i		i’	i	i	i’		i’	
Vider	oui	i’	i’	i’	i’	i’	i’	i’	i’	i’
	inutile		i	i’			i’	i	i’	i
Dépiler	vide		i	i’			i’	i	i’	i
	dernier	i’	i’	i’	i’	i’	i’	i’	i”	i’
	oui	i		i’	i	i	i’		i’	
Empiler	premier	i’	i’	i’	i’	i’	i”	i’	i’	i’
	oui	i		i’	i	i	i’		i’	

TAB. II.3 – Relation de non-commutativité des fonctions sur l’ADT PILE

est vide. Ainsi, les couples qui peuvent être pris en considération pour une étude de leur propriétés de commutativité est singulièrement réduit (divisé par plus de quatre dans cet exemple), ce qui simplifie d’autant le travail du concepteur.

Ce genre d’outil, en conjonction avec d’autres méthodologies d’extraction de la commutativité [Roesler & Burkhard 87] [Chrysanthis et al. 91] pourrait être développé et intégré dans un système.

II.3.3 Restriction aux ensembles bornés

Les seuls objets réellement représentables dans un ordinateur sont des éléments appartenant à des ensembles finis. Cette réflexion nous amène tout naturellement à appliquer les résultats obtenus au cas des ensembles bornés.

Exemple II.13 *En théorie, la multiplication par deux est une fonction définie de* $\mathbb{N}$ *vers* $\mathbb{N}/2$*,* $\mathbb{N}/2$ *étant strictement inclus dans* $\mathbb{N}$ *bien que de même cardinal.*

Dans un ordinateur, les entiers sont bornés par le nombre de bits alloués à leurs représentation. Supposons que le type natural d’Ada soit borné par $2^{16} - 1$*, alors l’opération de multiplication par deux doit se décomposer en deux fonctions :*

- $\mathit{Multiplier}^2_{\mathit{oui}} : \{0, \ldots, 32767\} \rightarrow \{0, 2, \ldots, 65532, 65534\}$ *:*
- $\mathit{Multiplier}^2_{\mathit{dépassement}} : \{32768, \ldots, 65535\} \rightarrow \{32768, \ldots, 65535\}$.

Le théorème suivant montre que l’espace des valeurs pouvant faire l’objet de modifications concurrentes vis-à-vis d’une famille de fonctions se restreint alors brutalement.

Théorème II.13 *Soit $\mathcal{F}$ une famille vérifiant C1 et C2, et telle que :*

$$\begin{aligned}&\forall i : 1 \leq i \leq n,\\ &|A_i| \in \mathbb{N}\end{aligned}$$

alors, si $|\mathcal{F}| = 2$:

$$(A_1 = B_2) \wedge (A_2 = B_1)$$

et si $|\mathcal{F}| \geq 3$:

$$\begin{aligned}&\forall i : 1 \leq i \leq n,\\ &\forall j : 1 \leq i \leq n,\\ &\quad A_i = B_j = A_{\mathcal{F}} = B_{\mathcal{F}}.\end{aligned}$$

Preuve II.14 *Si $|\mathcal{F}| = 2$, $B_2 \subseteq A_1$ et $B_1 \subseteq A_2$ [condition C1], or $|B_1| = |A_1|$ et $|B_2| = |A_2|$ [condition C2 et ensembles bornés], d'où $|B_2| \leq |A_1| = |B_1| \leq |A_2| = |B_2|$, puis $|A_1| = |A_2| = |B_1| = |B_2|$ et finalement la conclusion annoncée.*

Si $|\mathcal{F}| = 3$, nous avons $A_1 = B_2$ et $A_2 = B_1$ mais aussi $A_1 = B_3$ et $A_3 = B_1$ ainsi que $A_2 = B_3$ et $A_3 = B_2$ d'où, par transitivité, nous obtenons $A_1 = B_2 = A_3 = B_1 = A_2 = B_3$ et, par définition, $A_{\mathcal{F}} = B_{\mathcal{F}}$.

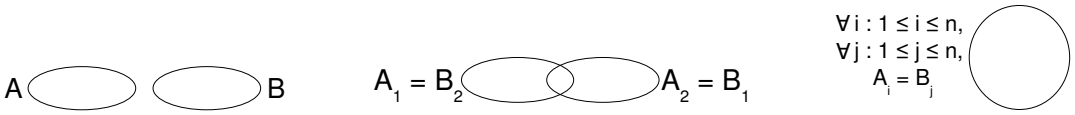
Supposons la propriété vraie pour $n \geq 3$ et montrons qu'elle le reste pour $n + 1$: nous avons $B_{n+1} \subseteq A_{\mathcal{F}}$ ainsi que $B_{\mathcal{F}} \subseteq A_{n+1}$ [condition C1 et hypothèse de récurrence], or, d'une part, $A_{\mathcal{F}} = B_{\mathcal{F}}$ [hypothèse de récurrence], et, d'autre part, $|A_{n+1}| = |B_{n+1}|$ [condition C2 et ensembles bornés], ce qui prouve la propriété.

La figure II.3 résume les différents phénomènes : opérations exclusives pour $|\mathcal{F}| = 1$, un cas anecdotique pour $|\mathcal{F}| = 2$, et le cas général pour $|\mathcal{F}| \geq 3$. Le second cas, où $|\mathcal{F}| = 2$, autorise une forme de commutativité marginale puisque applicable seulement entre deux opérations, à l'exclusion de toute tierce.

Exemple II.14 *Reprenons l'exemple de l'ADT* PILE *et posons $A_1 = \mathbb{P}^0 \cup \mathbb{P}^1$ et $A_2 = \mathbb{P}^1 \cup \mathbb{P}^2$, où $\mathbb{P}^1$ et $\mathbb{P}^2$ sont respectivement les ensembles des piles contenant un seul et deux éléments. Supposons que nous ayons les deux opérations concurrentes* Empiler(x) *et* Dépiler(x)*. Ces opérations peuvent donner lieu, sur la pile contenant l'unique élément x, à deux compositions, qui sont :*

- *Dépiler$_{x,oui}$ $\circ$ Empiler$^x_{oui}(\{x\})$;*
- *Empiler$^x_{premier}$ $\circ$ Dépiler$_{x,dernier}(\{x\})$.*

Nous somme fondés à croire que ces opérations commutent même si les fonctions associées sont différentes. Remarquons tout de même que les inverses restent les même pour chaque ordre d'application. En effet, les différenciations faites entre les fonctions jumelles ne sont pas indispensables pour déterminer les inverses correspondantes (cf. exemple II.12 pour la justification).



(a) $|\mathcal{F}| = 1$ (b) $|\mathcal{F}| = 2$ (c) $|\mathcal{F}| \geq 3$

FIG. II.3 – Limites de la commutativité sur les ensembles bornés

Dans le troisième cas, quand $|\mathcal{F}| \geq 3$, le phénomène de convergence disparaît pour être remplacé par un phénomène d'immobilité, plus limitatif encore.

Exemple II.15 *Sur l'exemple de l'ADT* PILE*, il est flagrant que nous avons distingué seulement deux domaines,* $\mathbb{P}^0$ *et* $\mathbb{P}^+$*, et par là deux prédicats. Il y a une unique clique de fonctions définies de* $\mathbb{P}^0$ *vers* $\mathbb{P}^0$*, une infinité de cliques où les fonctions sont définies de* $\mathbb{P}^+$ *vers* $\mathbb{P}^+$ *(une pour chaque valeur possible en sommet de pile), et quelques fonctions qui « font passer » de* $\mathbb{P}^0$ *à* $\mathbb{P}^+$ *ou inversement et sont donc exclusives.*

II.3.4 Conclusion partielle

Nous avons vu dans cette section quel est l'inconvénient majeur de la commutativité : un phénomène de *convergence* (cf. théorème II.9) ou d'*immobilité* (cf. théorème II.13). La racine du mal en est la condition C1 sur les ensembles de départ et d'arrivée, condition imposée par le sens que l'on donne à la propriété de commutativité entre opérations.

D'un point de vue moins pessimiste, nous avons montré que l'on pouvait utiliser certains résultats pour déterminer la *non*-commutativité des opérations. Cette direction nous semble toutefois d'application assez restreinte maintenant.

Enfin, nous avons vu, au travers de l'exemple II.1 de l'ADT RÉPERTOIRE, que la convergence pouvait être très lente pour certaines opérations sur certains objets. La commutativité vaut alors que l'on s'attarde sur ses avantages.

II.4 AVANTAGES POUR LA REPRISE

Parmi les avantages de la commutativité, les théorèmes II.15 et II.17 sont réellement connus depuis longtemps et font l'objet des deux premières sections. Quant à la troisième section, elle apporte un nouvel avantage répondant, dans la mesure du possible, aux interrogations d'autres auteurs.

II.4.1 Transitivité

Le théorème II.15 ne fait que préciser qu'une famille de fonctions deux à deux commutatives peut être composée dans n'importe quel ordre sans changer la valeur finale de l'objet. Il se résume à :

« Une relation deux à deux commutative est également transitive, abstraction faite de la réflexivité. »

Théorème II.15 *Soit $\mathcal{F}$ une famille vérifiant C1 et C3, alors :*

$$\begin{aligned}&\forall(k_1,\ldots,k_n)\in\pi(1,\ldots,n),\\&\forall x\in A_{\mathcal{F}},\\&\quad f_{k_n}\circ\cdots\circ f_{k_1}(x)=f_n\circ\cdots\circ f_1(x).\end{aligned}$$

Preuve II.16 *Procédons par récurrence sur la taille de la famille. La propriété est triviale pour $n=1$ et vérifiée par définition pour $n=2$. Supposons la propriété vraie pour $n-1$ et montrons qu'elle le reste pour n : posons $k_i=n$, alors $f_{k_n}\circ\cdots\circ f_{k_i}\circ\cdots\circ f_{k_1}(x)=\alpha\circ f_n\circ\beta(x)=f_n\circ\alpha\circ\beta(x)$ [lemme II.7] $=f_n\circ f_{n-1}\circ\cdots\circ f_1(x)$ [hypothèse de récurrence].*

II.4.2 Isolation

Le théorème suivant exprime que les paramètres de sortie ne sont pas sensibles à l'ordre d'application d'opérations commutatives, et plus précisément qu'ils ne sont pas sensibles au fait que d'autres opérations, commutatives, soient ou ne soient pas appliquées au préalable. Nous pouvons traduire ce théorème en :

« Le critère de commutativité assure l'isolation des transactions. »

Théorème II.17 *Soit $(\mathcal{F}, \mathcal{F}^T)$ un couple de familles vérifiant C1, C3 et C4, alors :*

$$\begin{array}{l}\forall i : 1 \leq i \leq n, \\ \forall x \in A_{\mathcal{F}}, \\ \quad f_i^T \circ f_n \circ \cdots \circ f_i \circ \cdots \circ f_1(x) = f_i^T \circ f_i(x).\end{array}$$

Preuve II.18 *Procédons par récurrence sur la taille de la famille. La propriété est trivialement vérifiée pour $n = 1$ et donnée par définition pour $n = 2$. Supposons-la vérifiée pour $n - 1 \geq 2$ et montrons qu'elle est encore vraie pour n : $f_n \circ \cdots \circ f_1(x) = f_i \circ f_j \circ \alpha(x)$ [théorème II.15], $\alpha(x) \in B_{\mathcal{F}\setminus(f_i,f_j)}$ [théorème II.9], et $B_{\mathcal{F}\setminus(fi,fj)} \subseteq A_i \cap A_j$ [lemme II.1 si $|B_{\mathcal{F}\setminus(fi,fj)}| = 1$, corollaire immédiat du lemme II.3 autrement], donc $f_i^T \circ f_i \circ f_j(\alpha(x)) = f_i^T \circ f_i(\alpha(x))$ [condition C4] $= f_i^T \circ f_i(x)$ [hypothèse de récurrence].*

De par l'absence de la condition C2 dans les hypothèses, ce théorème justifie aussi l'utilisation de la commutativité dans le cadre des méthodes où les opérations concurrentes de transactions distinctes sont réalisées dans des espaces de travail.

Il peut être vu comme la seconde partie du théorème II.15, tout comme la condition C4 peut être considérée comme la seconde partie de la condition C3.

II.4.3 Indépendance vis-à-vis du rejet

Nous en arrivons finalement à un avantage méconnu de la commutativité que nous paraphrasons pour en faciliter la lecture formelle : « La composition d'une famille de fonctions deux à deux commutatives avec un sous-ensemble de sa famille de fonctions inverses, où chaque fonction inverse n'est appliquée qu'après sa fonction directe correspondante, est équivalente à une composition où seule les fonctions qui n'ont pas été annulées sont présentes. »

Théorème II.19 *Soit $(\mathcal{F}, \mathcal{F}^{-1})$ un couple de familles vérifiant C1, C2 et C3, avec $\mathcal{F}' = (f_1, \ldots, f_n, f_{n+1}, \ldots, f_{n+m})$ tel que $\mathcal{F} = (f_1, \ldots, f_n)$, et, sans perte de généralité, $(f_{n+1}, \ldots, f_{n+m}) = (f_1^{-1}, \ldots, f_{m-1})$ avec $m \leq n$, alors :*

$$\begin{array}{l}\forall (k_1, \ldots, k_{n+m}) \in \pi(1, \ldots, n+m), \\ \forall x \in A_{\mathcal{F}}, \\ \quad (\forall i : 1 < i \leq n+m, \\ \qquad k_i \geq n+1 \rightarrow \begin{array}{r}\exists j : 1 \leq j < i, \\ k_i = n + k_j)\end{array} \\ \quad \rightarrow \\ \quad f_{k_{n+m}} \circ \cdots \circ f_{k_1}(x) = f_n \circ \cdots \circ f_{m+1}(x),\end{array}$$

la notation étant abusive quand $m = n$.

Preuve II.20 *Procédons par récurrence sur m. Le théorème II.15 prouve la propriété pour $m = 0$. Supposons cette propriété vérifiée pour $m \geq 0$ et montrons qu'elle le reste pour $m+1$: Choisissons le plus petit i tel que $k_i \geq n+1$, alors il existe $j < i$ tel que $k_i = n + k_j$; posons $k_j = p$, alors* $f_{k_{n+m}} \circ \cdots \circ f_{k_1}(x) = \alpha \circ f_{k_i} \circ \beta \circ f_{k_j} \circ \gamma(x) = \alpha \circ f_{p-1} \circ \beta \circ f_p \circ \gamma(x) = \alpha \circ f_{p-1} \circ f_p \circ \beta \circ \gamma(x)$ *[théorème II.15 car ni β, ni γ ne contiennent de fonctions inverses]* $= \alpha \circ \beta \circ \gamma(x) = f_n \circ \cdots \circ f_{m+1}(x)$ *[hypothèse de récurrence].*

Mais, nous pouvons surtout paraphraser ce résultat d'une façon beaucoup plus concise et intéressante :

> *« Une opération inverse peut s'appliquer à n'importe quel moment après application de son opération directe,* sans aucun contrôle *entre opérations directes et inverses ou entre opérations inverses. »*

On peut s'étonner de ce que ce théorème II.19 n'ait pas été exploité dans la littérature. (Cette remarque ne concerne pas le modèle des transactions plates avec les seuls modes d'accès *Lire* et *Écrire*, notamment le verrouillage 2PL. [Turc 89] précise même qu'en cas de redémarrage après panne, aucune structure utile au contrôle de concurrence, verrous, graphes, estampilles, ensemble d'objets, etc., n'a à être restaurée. Nous verrons dans l'exemple II.16 qu'une telle restauration peut toutefois permettre de lancer de nouvelles transactions en concurrence avec la fin de la reprise.)

Récemment encore, [Weikum 91] présente un cas, uniquement symbolique il est vrai, d'interblocage insoluble durant un rejet si une opération inverse est *moins commutative* que son opération directe. La solution implémentée dans le cadre du projet MONADS [Broessler & Freisleben 89] est de faire en sorte que les opérations directes et inverses aient exactement les mêmes restrictions de commutativité, autrement dit, deux opérations commutent si, et seulement si, elles commutent entre elles et avec leurs opérations inverses respectives. La seule réponse apportée par [Weikum 91], soutenant cette solution, est :

> « *Whereas this rule guarantees termination of (sub-)transactions, it restricts potential concurrency [...] On the other hand, one can conjecture that it is always possible to design inverse actions with a conflict relation that is no more restrictive than that of their primary actions.*[4] »

Pourtant, [Moss et al. 85] [Moss et al. 86] concluaient déjà en présentant la recherche d'un théorème approchant comme une voie de recherche :

> « *The relationship between forward conflict (between two actions) and backward conflict (between an action and an UNDO of another action) should also be addressed. Can we implement UNDOs in such a way that backward conflict occurs if*

[4] « Bien que cette règle garantisse la terminaison des (sous-)transactions, elle restreint la concurrence potentielle [...] Toutefois, on peut conjecturer qu'il est toujours possible de concevoir des opérations inverses ayant une relation de commutativité qui ne soit pas plus restrictive que celle de leurs opérations directes. »

and only if there is forward conflict? Also, to what extend can UNDOs be treated like ordinary actions? Can an ABORT or an UNDO be aborted or undone? What additional problems this present?[5] »

Quelles réponses pouvons-nous apporter à ces interrogations grâce au théorème II.19 ?

Tout d'abord, il établit le fait que les opérations inverses ne requièrent que l'atomicité de leur exécution, ce qui peut être obtenu par du verrouillage court (ou exclusion mutuelle) sur l'instance, et ne nécessite nullement un mécanisme de synchronisation plus général. Mais, comme les opérations inverses s'appuient sur le contrôle effectué par leurs opérations directes, il est indispensable de s'assurer que l'autorisation d'exécution concurrente obtenue par l'opération directe est encore effective au moment de l'application de l'opération inverse, ce qui n'est pas toujours réalisé.

Exemple II.16 *Lors d'un redémarrage du système consécutif à une panne, la mémoire centrale, qui contenait les verrous, est perdue. Si l'on veut pouvoir redémarrer le système tout en autorisant de nouvelles transactions à s'exécuter en parallèle, alors il faut reconstituer la table des verrous. En effet, le rejet des transactions interrompues va nécessiter l'exécution d'opérations inverses qui ne vont pas acquérir de verrous puisqu'elles s'en remettent à ceux de leurs opérations directes.*

ARIES [Mohan et al. 89] implémente une méthode de reprise sur pannes basée sur le paradigme du « repeating history *» (cf. chapitre I). Lors d'un premier parcours du journal, toutes les images après des objets modifiés sont restaurés dans la base de données. On profite tout naturellement de cette phase pour restaurer la table de verrous. À la fin du parcours, l'état physique, et logique, de la base correspond à l'état logique dans lequel elle se trouvait à l'instant de la panne. Un second passage, en sens inverse, va appliquer les opérations inverses pour les transactions non validées. La table des verrous ayant été reconstituée, il est permis aux utilisateurs de lancer de nouvelles transactions en concurrence avec la fin de la reprise.*

En revanche, bien que la méthode de reprise sur panne multi-niveaux de [Weikum et al. 90] (cf. chapitre I) se base sur le même principe du « repeating history *», on ne peut pas utiliser le théorème II.19. Seul le journal de niveau 0 (opérations* Lire *et* Écrire *sur les pages physiques) est parcouru dans le sens chronologique. À la fin de ce parcours, l'état physique de la base a été restauré à l'état logique au moment de la panne. Comme ARIES, la reprise va maintenant consister à parcourir le journal du niveau 1, contenant des opérations logiques inverses, dans le sens chronologique inverse. Toutefois, les verrous correspondants aux opérations directes n'ayant pas été restaurés, la poursuite de la reprise ne peut pas se faire en parallèle avec de*

[5] « La relation existant entre d'une part la commutativité entre opérations directes et d'autre part celle des opérations directes avec les opérations inverses devrait être étudiée. Pouvons-nous implémenter ces opérations inverses de telle sorte qu'il n'y ait conflit avec une opération inverse que s'il y a conflit avec son opération directe correspondante ? Dans le même ordre d'idée, jusqu'à quel point peut-on traiter les opérations inverses comme des opérations normales ? Un rejet peut-il être rejeté à son tour ? Quels seraient alors les nouveaux problèmes ? »

nouvelles transactions. Pour que cela soit possible, il faudrait dans la phase initiale parcourir séquentiellement tous *les journaux.*

Par conséquent, la réponse à la deuxième interrogation est que les opérations inverses ne doivent pas être traitées comme des opérations normales.

Enfin, comme elles ne peuvent pas être rejetées, les deux dernières questions de [Moss et al. 86] ne se posent plus.[6]

Revenons à la première question et précisons, car c'est une question qui vient naturellement à l'esprit, que le théorème II.19 n'implique pas que les opérations inverses commutent avec les opérations directes, ni entre elles. En fait, la question même n'a pas de sens si on ne la restreint pas, à cause de la condition C1 sur les ensembles de départ et d'arrivée des fonctions. Le corollaire attendu n'est vrai que dans le cas très particulier de fonctions bijectives sur des ensembles de départ et d'arrivée égaux.

Corollaire II.21 *Soit $(\mathcal{F}, \mathcal{F}^{-1})$ un couple de familles vérifiant C1, C2 et C3, avec $|\mathcal{F}| \geq 2$, alors :*

$$
\begin{aligned}
&\forall i : 1 \leq i \leq n, \\
&\forall j : (1 \leq j \leq n) \wedge (j \neq i), \\
&\forall x \in A_i \cap A_j, \\
&\quad (A_i = A_j) \wedge (f_i(A_i) = A_i) \wedge (f_j(A_j) = A_j) \rightarrow \\
&\quad f_i^{-1} \circ f_j(x) = f_j \circ f_i^{-1}(x) \wedge \qquad (1) \\
&\quad f_i^{-1} \circ f_j^{-1}(x) = f_j^{-1} \circ f_i^{-1}(x). \qquad (2)
\end{aligned}
$$

Preuve II.22 *Par hypothèse, $f_i(A_i) = B_i$, c'est-à-dire que f_i est surjective. D'autre part, f_i est nécessairement injective, autrement f_i^{-1} ne pourrait pas être une fonction [condition C2] (elle serait une fonction multivoque). Donc f_i est bijective. (1) et (2) sont alors des résultats qui suivent directement de l'égalité $B_i = A_i$.*

Vérifions que les fonctions inverses satisfont aux quatre conditions. (1) et (2) ne sont autre que la condition C3 (commutativité d'état). La condition supplémentaire $A_i = B_i$ (ainsi que $A_j = B_j$) est nécessaire pour que la condition C1 (inclusion réciproque des ensembles d'arrivée dans les ensembles de départ) soit vérifiée à la fois pour les fonctions inverses et directes. La condition C2 (existence d'une fonction inverse) est évidente puisque f_i est bijective. Enfin, la condition C4 (invariance des vues) est trivialement vérifiée puisque les fonctions inverses ne retournent aucune vue aux transactions.

[6] Dans le modèle multi-niveaux, une opération étant une transaction au niveau inférieur, cette dernière peut être rejetée. Il suffit de la redémarrer jusqu'à ce qu'elle valide, en utilisant éventuellement une méthode d'estampillage pour assurer le succès [Rosenkrantz et al. 78].

Dans *System R*, où les transactions multi-niveaux ne sont qu'à l'état pré-natal [Beeri et al. 88], les transactions qui procédent à leur phase de rejet ne doivent jamais être choisies comme victimes en cas d'interblocage. Pour être certain qu'il ne peut pas se produire d'interblocages qui ne mettraient en cause que des transactions en cours de rejet, il n'est autorisé qu'un seul rejet à la fois [Gray et al. 81] !

Donc, pour des fonctions bijectives, les fonctions inverses commutent entre elles et avec les fonctions directes. (Ce résultat s'étend facilement à toute une famille de fonctions commutatives.)

Exemple II.17 *Les rotations sont des fonctions bijectives :*

$$f_n : \begin{array}{ccc} [0, 2\pi[& \rightarrow & [0, 2\pi[\\ x & \mapsto & (x + n) \; mod \; 2\pi \end{array}$$

Les opérations d'incrémentation et de décrémentation sur un ADT COMPTEUR non borné *sont également bijectives. Mais, dès que l'on impose une borne inférieure ou supérieure, les opérations inverses ne commutent plus. Par exemple, sur l'intervalle* $[0, +\infty[$*, les incrémentations commutent toujours :* $\mathsf{Inc}^{10} \circ \mathsf{Inc}^{25}(5) = \mathsf{Inc}^{25} \circ \mathsf{Inc}^{10}(5)$ *mais ni* $\mathsf{Inc}^{25^{-1}}(5)$*, ni* $\mathsf{Inc}^{10^{-1}}(5)$ *ne sont définis puisque l'incrémentation n'est pas surjective de* $[0, +\infty[$ *dans* $[0, +\infty[$*.*

Pour que l'utilisation de la commutativité entre opérations directes et inverses puisse devenir effective, il va falloir programmer les ADT d'une manière toute adaptée.

Dans la proposition de [Moss et al. 86], quand une nouvelle opération arrive, on ne teste pas si elle commute avec les opérations directes qui la précède, mais si ces dernières commutent avec son opérations inverse. (On fera le rapprochement avec la recouvrabilité simple et relative vues dans le chapitre I.)

La proposition de [Turc 89] envisage non seulement la commutativité entre opérations directes et inverses, mais aussi entre opérations inverses, et aboutit par là à un critère moins restrictif, qui augmente donc le nombre d'exécutions sérialisables.

L'hypothèse de bijectivité est donc une contrainte bien plus forte que celles demandées par [Moss et al. 86] et [Turc 89]. Une première relaxation est, pour une fonction bijective, que sa fonction inverse admette un prolongement intéressant, comme dans l'exemple suivant. Notez bien que ceci modifie la condition C2 et, qu'en conséquence, nous venons de faire une entorse à notre définition de la commutativité.

Exemple II.18 *Reprenons l'exemple II.12 de l'ADT* PILE *et les fonctions* $\mathsf{Empiler}^n_{oui}$ *et* $\mathsf{Empiler}^m_{oui}$*. Si* $n \neq m$*, elles ne commutent pas. Mais, il est possible, bien qu'assez mal aisé, d'implémenter une fonction inverse qui commute.*

Une solution est d'empiler non seulement le paramètre mais aussi le nom de la transaction, ainsi une pile devient un suite de couples. L'opération $\mathsf{Empiler}^n_{oui}{}^{-1}$ *ne sera plus* $\mathsf{Dépiler}_{n,ok}$*; elle consistera à parcourir la pile du sommet vers la base en cherchant le nom de la transaction, puis à tasser le haut de la pile pour récupérer cet espace.*

Nous redoutons que de telles opérations deviennent extrêmement pénibles à programmer. Songez, par exemple, à l'implémentation de l'opération inverse d'un dépilement.

Nous venons de voir l'avantage essentiel du critère de commutativité : la relation de commutativité d'une fonction inverse ne peut pas être plus restrictive que celle de sa fonction directe. Dans le cas des fonctions bijectives, elle est exactement égale. Dans le cas des fonctions seulement injectives, la question n'a même pas de sens.

La discussion qui a suivi, sur la possibilité d'avoir des fonctions inverses qui aient une relation de commutativité moins restrictive que leurs fonctions directes, a montré, au travers de l'exemple II.18, que l'on pouvait contourner les limites annoncées en section II.3, au prix d'une modification des conditions. C'est tout le sujet de la section suivante.

II.5 MOYENS D'ÉCHAPPER AUX LIMITES

Nous avons prouvé que la commutativité possède quelques avantages, notamment pour annuler les effets des transactions rejetées, qui augmentent le parallélisme aussi bien entre transactions vivantes qu'entre transactions vivantes et rejetées.

Cependant, nous avons surtout montré que la commutativité n'est qu'une forme de supercompatibilité et souffre des mêmes inconvénients : si l'opération d'écriture est exclusive de toute autre avec le critère de compatibilité, les opérations exclusives ne sont pas éliminées par la commutativité ; si les opérations de lectures ne permettent pas de modifier la valeur d'un objet, les opérations commutatives ne permettent pas d'affaiblir le prédicat décrivant l'ensemble auquel appartient la valeur initiale de l'objet.

Essayons de dresser un tableau des moyens utilisés pour donner à la commutativité tout son intérêt. Nous en avons trouvé quatre : indépendance, non-déterminisme, flou et recours à la commutativité mathématique.

Un moyen plus radical encore est de recourir à des critères plus permissifs que la commutativité : recouvrabilité relative ou dépendances de visibilité, que nous avons intégrées à notre chapitre I sur l'état de l'art.

II.5.1 Indépendance

Les objets composés de sous-parties largement indépendantes peuvent profiter au maximum de la commutativité, *dans la mesure où les opérations ne s'intéressent qu'à quelques sous-parties* (cf. exemple II.11). De tels objets sont les ADT ENSEMBLE, FAMILLE et, moins généralement, LISTE. Des cas particuliers en sont l'ADT BOÎTEAUXLETTRES dans un système d'exploitation qui utilise à son tour l'ADT ANNUAIRE des utilisateurs connus. Tous ces exemples ne sont finalement que des cas particuliers de l'ADT RELATION.

Ce qui ce produit pour ce genre d'ADT est que l'indépendance des éléments autorise plusieurs opérations exclusives à s'exécuter concurremment ! En effet, les opérations sur ces ADT agissent à deux niveaux, comme nous l'illustrons sur l'exemple II.19.

Exemple II.19 *Soient les trois fonctions suivantes, définies sur l'ADT* ENSEMBLE *:*

- $f_1 = \mathit{Insérer}^{x_1}_{oui} : \mathcal{P}(X \setminus \{x_1\}) \rightarrow \complement_{\mathcal{P}(X)}\mathcal{P}(X \setminus \{x_1\})$;
- $f_2 = \mathit{Insérer}^{x_2}_{inutile} : \complement_{\mathcal{P}(X)}\mathcal{P}(X \setminus \{x_2\}) \rightarrow \complement_{\mathcal{P}(X)}\mathcal{P}(X \setminus \{x_2\})$;
- $f_3 = \mathit{Supprimer}^{x_3}_{oui} : \complement_{\mathcal{P}(X)}\mathcal{P}(X \setminus \{x_3\}) \rightarrow \mathcal{P}(X \setminus \{x_3\})$;

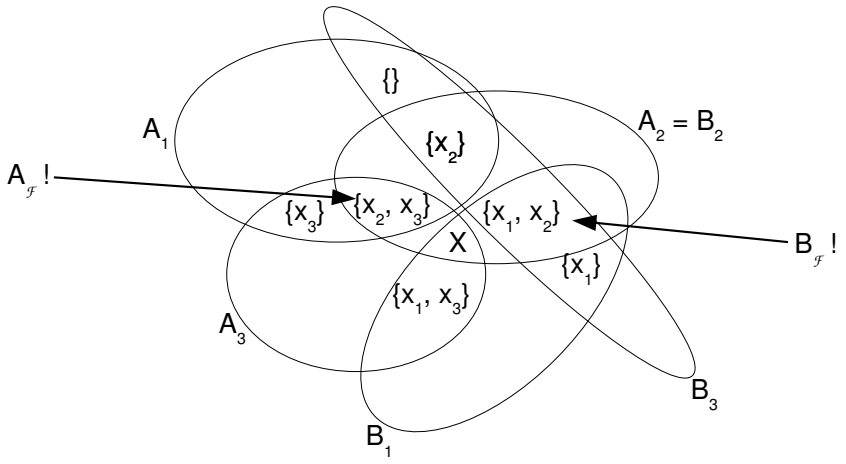


FIG. II.4 – Exemple d'opérations exclusives simultanées (!) sur l'ADT ENSEMBLE

où $X = \{x_1, x_2, x_3\}$.

De par le corollaire II.11, f_1 *et* f_3 *sont nécessairement des fonctions exclusives puisque toutes deux définies d'un ensemble vers son complémentaire. D'ailleurs, la figure II.4 représente les intersections des différents ensembles et* $B_{\mathcal{F}}$ *n'est effectivement pas inclus dans* $A_{\mathcal{F}}$ *!...*

Si les exécutions concurrentes sont possibles c'est parce que les fonctions doivent être réécrites, correctement, comme suit (cf. exemple II.9) :

- $f_1 = \textit{Insérer}^{x_1}_{oui} : \emptyset \rightarrow \{x_1\}$ *;*
- $f_2 = \textit{Insérer}^{x_2}_{inutile} : \{x_2\} \rightarrow \{x_2\}$ *;*
- $f_3 = \textit{Supprimer}^{x_3}_{oui} : \{x_3\} \rightarrow \emptyset$.

f_1 *et* f_3 *sont bien exclusives mais uniquement pour une valeur isolée de l'ensemble. L'exécution concurrente est l'union des exécutions concurrentes de tous les éléments pouvant appartenir à l'ensemble. Ainsi, ce que nous avons pris pour* $A_{\mathcal{F}}$ *et* $B_{\mathcal{F}}$ *sont en réalité l'union des ensembles de départ et d'arrivée, respectivement* $\{x_2, x_3\}$ *et* $\{x_1, x_2\}$.

L'opération Vider*, qui, elle, porte bien sur l'objet vu comme un tout, peut également être vue comme une macro composée d'une série de* Supprimer *(cf. algorithme II.1).*

Il ne faut donc pas s'étonner de ce niveau de commutativité. Il est même possible d'aller plus loin en constatant que les opérations sur les ensembles ne sont pas seulement commutatives mais surtout compatibles !

Pour prouver cette proposition, il suffit de trouver une implémentation qui s'accommode du seul critère de compatibilité des opérations de lecture et d'écriture. Une telle implémentation existe : elle s'appuie sur la fonction caractéristique d'un ensemble. En pratique, elle n'est réalisable que pour les ensembles discrets, raisonnablement bornés et de faible dispersion. La fonction caractéristique est un tableau de booléens. L'opération d'insertion met une position à vraie, et la suppression à faux. Le test d'appartenance est la lecture d'une entrée. Détermi-

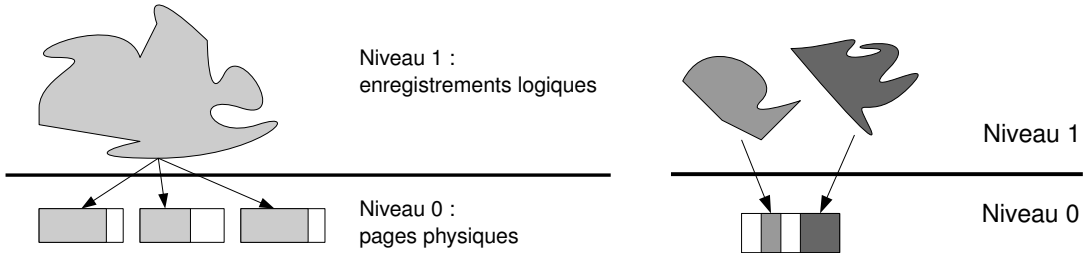
Algorithme II.1 Une implémentation compatible de l'ADT ENSEMBLE

```
abstract data type Ensemble generic type Element is
structure
    FonctionCaractéristique : array [range of Element] of boolean ;
operations
    Insérer (X : Element) is
        begin
            FonctionCaractéristique[X] := true
        end ;
    Supprimer (X : Element) is
        begin
            FonctionCaractéristique[X] := false
        end ;
    Appartient (X : Element) return boolean is
        begin
            return FonctionCaractéristique[X]
        end ;
    Cardinal return integer is
        var
            Total : integer ;
        begin
            Total := 0 ;
            for i in range of Element do
                if Appartient(i)
                then Total += 1 ;
            return Total
        end ;
    Vider is
        begin
            for i in range of Element do
                Supprimer(i)
            end ;
    end ;
```

ner le cardinal consiste à effectuer un parcours total en lecture, tandis que vider l'ensemble est un parcours total en écriture (cf. algorithme II.1). Si le granule sur lequel porte le contrôle de concurrence est le booléen, alors on retrouve bien la relation de « commutativité ».

L'utilisation de la commutativité sur les ADT de ce genre n'est donc justifiée que par la nécessité d'implémentations plus économiques, ou simplement faisables (par exemple, s'il s'agit d'ensembles de réels), que celle de l'algorithme II.1. En pratique, toutes les structures de recherches relèvent de cette analyse, c'est-à-dire essentiellement les B-arbres [Lehman & Yao 81] [Shasha & Goodman 88] [Huang 85] [Acosta 91] mais également des solutions alternatives comme les listes probabilistes [Pugh 90] ou les fichiers à adressage dispersé.

Cette règle d'indépendance devrait également s'appliquer au modèle des transactions multi-



(a) un objet volumineux couvre plusieurs pages

(b) plusieurs objets logiques co-stockés dans une même page

FIG. II.5 – Composition d'objets multi-niveaux

niveaux où la propriété d'objets emboîtés [Martin 87] et plus généralement d'objets indépendants [Cart et al. 90b] doit être vérifiée pour que les exécutions soient correctes. Nous conjecturons qu'il ne faut pas s'attendre à un gain substantiel de parallélisme si cette propriété n'est pas vérifiée entre deux niveaux adjacents. (C'est d'ailleurs ce qu'il ressort de quelques mesures effectuées sur des transactions plates, à deux et trois niveaux [Weikum 91] : le débit transactionnel avec le modèle à deux niveaux l'emporte sur les autres ; le modèle à trois niveaux est même plus mauvais que le modèle avec transactions plates !)

Exemple II.20 *Considérons un modèle de transactions à deux niveaux. Le niveau 0 est le niveau physique, composé d'objets qui sont les pages physiques du disque magnétiques, les seules opérations pour les manipuler étant* Lire *et* Écrire*. Le niveau 1 est celui des enregistrements logiques (n-uplets du modèle relationnel ou instances d'un modèle à objets), les opérations pouvant être quelconques.*

Les objets du niveau 1 peuvent être stockés sur plusieurs pages lorsqu'ils sont volumineux et alors la définition des objets emboîtés de [Martin 87] suffit à prouver leur indépendance (cf. figure II.5a).

Plus souvent, plusieurs enregistrements sont stockés dans une même page (cf. figure II.5b). Cependant, les pages sont indépendantes des enregistrements, au sens de [Cart et al. 90b], si les opérations du niveau 1 ne retirent aucune information sur les emplacements relatifs dans les pages. Par conséquent, deux opérations concurrentes sur une même page du disque commutent si les transactions du niveau supérieur accèdent à des enregistrements différents, car les enregistrements sont indépendants entre eux et que seules des contraintes liées soit à la compacité du stockage, soit au rapprochement de données souvent utilisées conjointement (« clustering *» [Benzaken & Delobel 89a] [Benzaken 90]), les ont placés dans la même page. Dans ce cas, les verrous du niveau 0 n'assurent que l'atomicité des opérations du niveau 0 et peuvent être relâchés prématurément, tandis que ceux du niveau 1 sont conservés afin d'assurer l'atomicité des transactions.*

Quand les objets ne sont pas naturellement composés de parties indépendantes, on peut alors essayer de forcer un tant soit peu les limites de la commutativité.

II.5.2 Non-déterminisme

Le non-déterminisme ne peut être utilisé, de façon transparente, que dans les couches basses du système (cf. exemple II.4), ou si les spécifications des transactions s'accommodent elles-mêmes d'une dose de non-déterminisme, comme les demandes d'impressions sur périphériques banalisés, l'ordre d'impression définitif n'ayant pas besoin d'être exactement l'ordre de validation des transactions. L'ADT FILEFAIBLE de [Schwarz & Spector 84] ainsi que l'ADT SEMI-FILE de [Weihl & Liskov 85] permettent d'obtenir ce résultat.

Exemple II.21 *[Schwarz & Spector 84] propose l'ADT* FILEFAIBLE*, dérivée d'un ADT extrêmement contraint, la* FILEFIFO*. En effet, le parallélisme que l'on peut attendre d'une* FILE-FIFO *est quasiment nul, du même ordre que celui de la* PILE*. En revanche, la* FILEFAIBLE *possède des opérations d'insertion et de retrait modifiées : les insertions peuvent se faire dans n'importe quel ordre, seules les insertions faites par une même transaction restent ordonnées ; l'élément à retirer en priorité est (1) le plus anciennement validé, ou sinon (2) le plus ancien inséré par la transaction elle-même, ou enfin (3) l'opération est bloquée jusqu'à ce qu'un élément validé apparaisse.*

On peut faire trois remarques à partir de cet exemple. Premièrement, cette dose de non-déterminisme revient à introduire une certaine indépendance entre les éléments de la FILE-FAIBLE : ils ne sont plus tous liés par une relation d'ordre strict. Deuxièmement, la programmation des opérations n'est plus indépendante du contrôle de concurrence et de la reprise, comme nous l'avions déjà noté pour l'exemple II.18. À titre comparatif, [Gray et al. 81] estime que l'écriture d'une opération et de son inverse est, subjectivement, 30 % plus difficile, et nécessite, objectivement, 20 % de code supplémentaire. Très certainement, la programmation d'opérations non déterministes est beaucoup plus difficile encore. Enfin, cet ADT est la version transactionnelle du célèbre « producteur-consommateur » ; il s'agit donc davantage de coopération que de compétition entre les transactions, une voie de recherche intéressante.

II.5.3 Flou

Le manque de précision dans les informations retournées par des opérations contribue à élargir les familles d'opérations commutatives.

L'information minimale qui peut être renvoyée par une opération de lecture est un booléen qui donne l'appartenance de l'instance à un ensemble de valeur. Plus le prédicat décrivant cet ensemble sera faible, plus d'opérations modifiantes pourront s'exécuter en parallèle.

En ce qui concerne les opérations modifiantes, la plus permissive est celle qui ne renverra aucune information sur l'état de l'objet.

Exemple II.22 *Un exemple d'opération de lecture booléenne est* Vide *définie pour l'ADT* PILE *et décomposée en deux fonctions :* Vide_{non} *et* Vide_{oui}*. La première n'autorise aucun écrivain concurrent puisque elle caractérise un ensemble à un seul élément,* $\mathbb{P}^0$*, d'où, au contraire, la seconde va autoriser des empilements ou dépilements concurrents qui maintiennent l'état de la pile dans son complémentaire,* $\mathbb{P}^+$*.*

Une écriture qui ne renvoie aucun paramètre de sortie est l'opération de rotation de l'exemple II.17.

Limiter la précision des paramètres de retour augmente, autant que faire se peut, la taille des ensembles d'arrivée afin de ralentir le phénomène de convergence.

II.5.4 Commutativité mathématique

Il existe un ADT particulier qui est grandement commutatif tout en ne présentant aucune propriété d'indépendance ; il reste tout de même unique, à notre connaissance. Il s'agit de l'ADT COMPTEUR popularisé sous le nom de « *BankAccount* ». Ce type d'ADT a été introduit [O'Neil 86] afin d'éliminer les points de contention dans certaines applications où les opérations d'incrémentation et de décrémentation de valeurs numériques sont extrêmement fréquentes (systèmes bancaires, systèmes de réservation de places, gestion de stocks, etc.). Le très haut degré de commutativité obtenu sur cet ADT est redevable aux propriétés de commutativité mathématique sur les nombres, entiers ou réels.

Exemple II.23 *Si la valeur d'un compte en banque est de 10 000 F et que le montant du compte doive se trouver dans l'intervalle* $[0; 100\,000]$ *alors les opérations Débit(5 000), Crédit(15 000), Débit(3 000), Crédit(150), Débit(350), Crédit(2 800), Crédit(4 000) commutent car la somme des débits ne dépasse pas 10 000 F et que celle des crédits n'atteint pas 90 000 F.*

Pour obtenir un tel niveau il faut étendre la définition de la commutativité en prenant en compte l'état de l'objet (ce qui entraîne l'accès exclusif et solidaire pendant les phases de contrôle et d'exécution) et des *séquences* entières d'opérations [Ng 89].

Toutefois, ces extensions, très désirables, n'empêchent pas cet ADT de souffrir du phénomène de convergence.

Exemple II.24 *Supposons que notre compte en banque ait une valeur de 35 000 F. Il peut initialement varier négativement de 35 000 F et positivement de 65 000 F sur l'intervalle* $[0; 100\,000]$*. Supposons maintenant que l'on cherche à savoir s'il y a au moins 25 000 F sur ce compte. Les débits concurrents sont dorénavant limités à une somme maximale de 10 000 F. De même, toute opération de débit ou de crédit limite les valeurs possibles des débits ou crédits concurrents arrivant ultérieurement.*

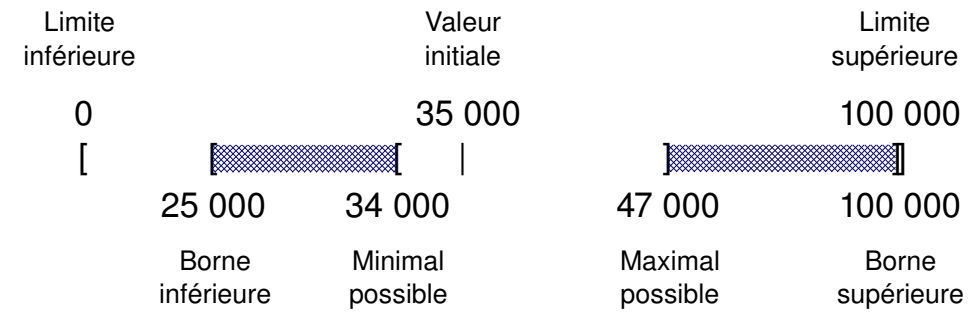


FIG. II.6 – Phénomène de convergence sur l'ADT COMPTEUR

Au fur et à mesure, et en l'absence de validations, la marge de manœuvre diminue jusqu'à devenir nulle. La figure II.6 représente ce phénomène. (Elle donne aussi une idée d'implémentation.) Outre la valeur initiale de l'objet, on distingue trois intervalles imbriqués : le plus grand fixe les valeurs possibles de l'objet ; le second donne l'amplitude maximale des modifications que l'on peut apporter à la valeur de l'objet et est fixé par la valeur maximale des paramètres des fonctions booléennes demandant si la valeur courante du compte en banque est supérieure ou inférieure à une valeur donnée ; enfin, l'intervalle le plus interne donne l'espace des variations possibles après validations ou rejets quelconques de toutes les opérations d'incrémentation et de décrémentation.

Une séquence d'opérations ayant pu mener à cet état est :

Débit$_{oui}^{1000}$ *; Supérieur*$_{oui}^{10000}$ *; Crédit*$_{oui}^{5000}$ *; Supérieur*$_{oui}^{25000}$ *; Crédit*$_{oui}^{7000}$.

Les zones hachurées marquent la marge de manœuvre dont dispose encore l'objet pour pouvoir répondre commutativement aux opérations se présentant ultérieurement.

II.5.5 Conclusion partielle

À bien y regarder, tous les moyens mis en œuvre pour proposer une forme de commutativité intéressante, du point de vue de la concurrence obtenue, sont essentiellement *ad hoc*.

L'indépendance des éléments dans un ensemble est naturellement exploitée. Ceci conduit à un statut ambivalent des opérations, certaines étant limitées à un élément de l'ensemble, alors que d'autres prennent en compte tous les éléments (elles peuvent être considérées comme des macros).

Le non-déterminisme tire parti de spécifications peu contraignantes sur le comportement d'une transaction. On n'impose plus la condition C4 sur l'invariance des paramètres de sortie.

L'exploitation d'opérations qui renvoient des paramètres de sortie les plus évasifs possible est le seul moyen de limiter le phénomène de converge sans modifier les conditions. Cela consiste à agrandir, autant que faire se peut, les ensembles de départ et d'arrivée des fonctions. Ces opérations doivent être effectivement utilisées par les programmeurs d'applications, ce que l'on ne peut pas imposer.

Enfin, la commutativité sur les nombres a été sensiblement étendue, la définition initiale autorisant finalement peu de concurrence (comme on pourra s'en persuader en appliquant les tables I.1, I.2 et I.3 à l'exemple II.23).

De plus, précisons que le critère de recouvrabilité relative [Badrinath & Ramamritham 87], présenté dans le chapitre précédent, ne requiert pas la condition C3 et affaiblit les condition C1[7] et C4 car les fonctions sont appliquées dans un ordre précis.[8]

Qu'en est-il de la commutativité sur les types de données les plus fréquemment décrits que sont les n-uplets ? Par essence, les champs d'un n-uplet sont peu indépendants entre eux. Il existe des dépendances fonctionnelles, telles que « *NuméroSécuritéSociale* » détermine « *Nom* », « *DateNaissance* », etc., ou encore d'autres contraintes sous forme prédicative plus générale. Ces dépendances impliquent que quand un champ est modifié, les champs qui lui sont liés le seront vraisemblablement, ce qui limitera rapidement le nombre d'opérations concurrentes autorisées.

Cette grande catégorie de types est utilisée dans les systèmes à objets pour décrire la structure des instances d'une classe. C'est aussi la structure d'une ligne de relation dans le modèle relationnel. Cependant, les méthodes de contrôle de concurrence dans les bases de données relationnelles n'ont pas, à notre connaissance, de granule de contrôle de concurrence inférieur au n-uplet ([Eswaran et al. 76] l'envisage mais *System R* ne l'implémente pas). Cet état de fait peut être lié aussi bien à des problèmes de performances (trop grand nombre de verrous à gérer si l'on choisit les champs des n-uplets comme granule de verrouillage) ou à l'obligation d'interpréter toutes les requêtes afin de déterminer les champs qu'elles mettent en jeu. Dans un système offrant les ADT, cette connaissance est disponible, dès la compilation, par l'intermédiaire des opérations qui manipulent les instances. Nous pouvons en tirer profit pour augmenter le parallélisme ; c'est ce que nous ferons dans le chapitre IV.

[Badrinath & Ramamritham 88] propose une technique d'analyse combinant une partie entièrement automatisée avec des prédicats fournis par les spécifications des opérations. Grâce aux résultats de ce chapitre, nous préconisons de franchir un pas... en arrière, en limitant la commutativité des opérations sur l'ADT N-UPLET à la simple compatibilité des accès aux différents champs (cf. [Malta 93] et chapitre IV).

[7] Elle devient : $\forall i : 1 < i \leq n, A_{i-1} \cup B_{i-1} \subseteq A_i$. L'inclusion de A_{i-1} dans A_i est nécessaire à cause des rejets.

[8] Néanmoins, ce critère devrait présenter ses propres limites. Tout comme la figure II.2 peut être tracée à partir de la condition C1, la condition C1 de la recouvrabilité relative permet de tracer le diagramme associé. On aperçoit alors un phénomène de divergence, complètement opposé à celui de la commutativité. Les ensembles de départ successifs sont « croissants », d'où les paramètres de retour des opérations ne peuvent être que de plus en plus flous. Pour maximiser le parallélisme, il faut donc que les opérations se présentent dans un bon ordre.

II.6 COMPARAISON

À notre connaissance, seul [Weihl 89a] montre que la concurrence sur un ADT posséde des limites. Pour cela, il introduit trois critères locaux, qui sont l'*atomicité dynamique*, l'*atomicité statique* et l'*atomicité hybride*. Ces divers critère locaux assurent la sérialisabilité. Toutes les méthodes à base de verrouillage, et certaines optimistes, relèvent du critère d'atomicité dynamique ; celles à base d'estampillage relèvent de l'atomicité statique ; enfin, les méthodes à base de multi-versions relèvent de l'atomicité hybride. Nous n'allons traiter que du critère local d'atomicité dynamique.

La formalisation des opérations d'un ADT est faite par un automate non déterministe. Les états de l'automate sont les états de l'objet ; les transitions sont des opérations munies de leurs paramètres d'entrée et de sortie. Toute séquence d'opérations dérivable à partir d'un état initial est une *séquence série* sur l'objet. L'ensemble des séquences série forme les spécifications de l'ADT.

Le critère d'atomicité dynamique ramène la caractérisation globale des exécutions sérialisables à un critère local. Une exécution est sérialisable dans un ordre donné si, en chaque objet, chaque ordre partiel connu de l'objet donne une séquence série. L'atomicité dynamique se caractérise ainsi : T_1 précède T_2 si, et seulement si, il existe une opération invoquée par T_2 qui se termine après la validation de T_1. On se rend compte que chaque objet n'a qu'une vue partielle de cette relation d'ordre. Si chaque objet assure que les exécutions concurrentes qui ont lieu sur lui-même sont sérialisables dans n'importe quel ordre compatible avec la vue partielle qu'il a de l'ordre global, alors les transactions sont sérialisables.

Par exemple, l'histoire donnée en table II.4 sur une instance de l'ADT ENSEMBLE ne fournit qu'une seule dépendance : $T_2 \prec T_3$.[9] Cette histoire est sérialisable dans l'ordre $T_1 \prec T_2 \prec T_3$. Mais, l'atomicité dynamique exige que sur l'objet x, l'histoire soit également sérialisable dans les ordres $T_2 \prec T_1 \prec T_3$ et $T_2 \prec T_3 \prec T_1$, ce qui n'est pas le cas. Donc, bien que l'histoire soit sérialisable, le critère local d'atomicité dynamique la rejette.

[Weihl 89a] montre que ce critère local est optimal, comme les deux autres. Si on l'affaiblit, les transactions globales ne seront plus nécessairement sérialisables. Donc, ce critère, ainsi que les deux autres, *fixe* une *limite à la concurrence* que l'on peut obtenir sur un ADT.

Toutefois, on remarque assez rapidement que ce critère ne tient pas compte de l'état de

[9] Weihl distingue les invocations des opérations et leurs retours, ce qui correspond *grosso modo* à nos fonctions f et f^T respectivement.

T_1	T_2	T_3
x.appartient(3)		
	x.insérer(3)	
	ok	
faux		
		x.appartient(3)
	x.valider	
		vrai
x.valider		
		x.valider

TAB. II.4 – Une histoire concurrente

T_1	T_2	T_3
x.empiler(5)		
ok	x.empiler(8)	
	ok	
	x.valider	
x.valider		
		x.dépiler
		5 ou 8 ?

TAB. II.5 – Un blocage définitif dû au critère d'atomicité dynamique

l'objet (notre condition C3) et ne peut donc être dénommé commutativité. Illustrons ce point sur l'ADT PILE avec l'histoire donnée en table II.5. Globalement, elle est sérialisable dans les deux ordres possibles (donc l'ordre de sérialisation peut être différent de l'ordre de validation) ; localement, le critère d'atomicité dynamique est également vérifié. Pourtant, en ce qui concerne l'état de l'objet, cette exécution est parfaitement indéterminée puisque plus aucune transaction ultérieure ne va pouvoir dépiler ! En effet, soit T_3 dépile la valeur 5, mais cela contredit l'ordre $T_1 \prec T_2 \prec T_3$, soit elle dépile 8, et c'est alors l'ordre $T_2 \prec T_1 \prec T_3$ qui ne donne pas une séquence série.

Dans une implémentation, le critère d'atomicité dynamique doit donc être renforcé. C'est ce que [Weihl 88] nomme *atomicité dynamique en ligne*. Ce nouveau critère impose deux contraintes supplémentaires : dès qu'une opération est exécutée, elle doit pouvoir valider à tout instant ; tous les ordres de sérialisation doivent conduire au même état final de l'objet. Nos conditions C1 à C4 vérifient le critère d'atomicité dynamique en ligne. Pour sa part, [Weihl 88] distingue la commutativité en arrière et la commutativité en avant. Nous les avons présentées assez en détail dans le chapitre I.

La commutativité en avant n'utilise pas la condition C2 puisqu'elle ne nécessite pas l'existence d'opérations inverses, les mises-à-jour étant différées jusqu'au moment de la validation. L'absence de la condition C2 permet de fusionner des fonctions qui, autrement, auraient dû être distinguées (cf. exemple II.9).

La commutativité en arrière nécessite de rendre plus complexe la condition C1, alors qu'elle ne semble avoir qu'un intérêt très limité. En effet, elle apporte un peu plus de commutativité à l'ADT COMPTEUR. Or, cet apport reste très faible par rapport aux propositions qui ont été citées en section II.5.3. De plus, [Weihl 89b] récidive en proposant une commutativité en arrière *à droite* ! Pour cela, le critère d'atomicité dynamique en ligne est abandonné au profit de l'« *équi-efficacité* » ! Cette troisième forme de commutativité n'est plus symétrique.

En définitive, [Weihl 89a] n'introduit pas les limites de la commutativité, mais celles, plus générales, de la concurrence vis-à-vis de certains critères locaux. Le critère qui embrasse la commutativité est celui d'*atomicité dynamique*. Il est optimal et impose donc une limite à la concurrence. Toutefois, ce critère est totalement illusoire car s'il assure la sérialisabilité des transactions, il peut aussi entraîner un blocage définitif d'un objet, comme le montre notre exemple sur l'ADT PILE, et donc d'une partie du système. Ainsi, en pratique, [Weihl 89b] impose-t-il des critères plus forts, l'*atomicité dynamique en ligne* puis l'*équi-efficacité*, et propose-t-il plusieurs formes de commutativité.

Pour notre part, nous avons montré les limites d'une forme de commutativité qui recouvre la commutativité en avant et, en partie seulement, la commutativité en arrière. Mais, la section II.5.3 montre, informellement, le cas particulier de convergence sur l'ADT COMPTEUR et finalement les limites de toutes les formes de commutativité.

II.7 CONCLUSION

La commutativité, critère principal de contrôle des accès concurrents à des données partagées, a fait l'objet d'un grand nombre de publications. Illustré par des exemples classiques, il semble être largement plus permissif que la compatibilité des simple modes d'accès *Lire* et *Écrire*.

Le résultat principal de ce chapitre est d'avoir montré que la commutativité est sujette à un phénomène de convergence qui étend le comportement du critère de compatibilité. Un accès en écriture est exclusif avec la compatibilité ; les opérations exclusives ne sont pas éliminées par la commutativité. Les accès en lecture ne permettent pas de modifier la valeur de l'objet ; les opérations commutatives ne permettent pas d'affaiblir le prédicat décrivant l'ensemble auquel appartient la valeur initiale de l'objet.

Le seul point intéressant de la commutativité (outre les avantages des histoires strictes, étudiées dans le chapitre I) est qu'elle possède une propriété agréable pour la reprise des transactions rejetées dans le cadre des méthodes avec mises-à-jour immédiates puisque l'exécution des opérations inverses n'a pas à être contrôlée. En fait, c'était la vérification de cette propriété qui nous motivait initialement. C'est parce que les contraintes sur les ensembles de départ et d'arrivée nous semblaient essentielles pour en faire une démonstration rigoureuse que nous avons opté pour une formalisation fonctionnelle, alors que celles de [Goodman & Shasha 85] et [Bernstein et al. 87] sont plutôt relationnelles, et que celle de [Weihl 88] est donnée sous forme d'automate. Ce choix étant fait, et en élaborant les premiers lemmes concernant ces ensembles, nous avons inévitablement abouti à ces si fâcheuses limites. Ceci dit, l'absence de variété dans les exemples illustrant différents articles pouvait déjà nous les faire pressentir.

Si les travaux de [Weihl 89a] montrent l'existence de telles limites, diverses et incomparables (atomicité dynamique, statique et hybride), ils ne caractérisent pas leur nature, et en particulier ne mettent pas en évidence le phénomène de convergence. De plus, la mise en œuvre du critère d'atomicité dynamique est totalement illusoire puisque pouvant conduire à des blocages définitifs de certains objets : les transactions ultérieures ne pourront plus lire leur état. D'autre part, les diverses formes de commutativité, notamment la commutativité en arrière à droite, semble n'avoir d'intérêt que pour un seul ADT bien particulier, l'ADT COMPTEUR. Une voie de recherche serait donc de faire des hypothèses plus faible pour englober le comportement de la commutativité en arrière à droite, mais surtout d'étudier celui de la recouvrabilité relative (en arrière et en avant !).

Ces limites étant posées, nous avons rapidement présenté quelques techniques qui augmentent la concurrence sur les ADT au prix d'une modification des conditions C1 à C4. Ceci nous permet de donner quelques règles empiriques :

- utiliser la commutativité pour les objets composés de parties logiquement indépendantes, comme l'ADT ENSEMBLE, où le phénomène de convergence apparaît indépendamment dans chaque composante ;
- s'en remettre à des techniques bien plus simples, peut être même à la compatibilité, pour les objets structurés sous forme de n-uplet ;
- prévoir un ADT COMPTEUR (voire plusieurs variantes) pour traiter le cas des incrémentations et décrémentions concurrentes fréquentes.

De plus, certaines applications, comme les systèmes de réservation de places, se contentent du non-déterminisme de certaines opérations. La programmation qui en résulte peut alors devenir complexe. Peut-on proposer un ADT non déterministe générique (ou une base minimale de tels ADT) ?

Dans le domaine des bases de données à objets où les classes et leurs méthodes sont fréquemment ajoutées, retirées ou modifiées, les limites de la commutativité doivent nous convaincre qu'une analyse très simple de la commutativité des méthodes devrait donner d'aussi bons résultats, sinon meilleurs (en termes de surcharge de travail à l'exécution), qu'une technique plus compliquée. Le prochain chapitre ne traitera pas immédiatement de la commutativité des méthodes ; nous nous en tiendrons à la compatibilité. Cependant, dans le chapitre IV, nous présenterons une technique d'analyse simple permettant de déduire automatiquement la commutativité de celles-ci.

Chapitre III

Contrôle de concurrence sur le graphe d'héritage

Résumé du chapitre

Dans ce chapitre, nous proposons une méthode de contrôle de concurrence adaptée aux bases de données à objets. L'approche à objets introduit un nouveau modèle de données, nettement plus riche que celui du modèle relationnel, ainsi que de nouveaux modes d'accès, caractérisés par l'envoi de messages au lieu de requêtes. Nous utilisons cette richesse et la sémantique associée pour introduire un protocole de contrôle de concurrence à base de verrouillage. Les qualités de notre protocole sont : la possibilité d'introduire autant de modes d'accès (généralistes) que voulu ; la création automatique des verrous correspondants ; la faculté de traiter le problème des objets dits « fantômes » lors des créations et destructions ; la possibilité de gérer les modifications concurrentes du graphe d'héritage.

Mots clés

Bases de données à objets, graphe d'héritage, contrôle de concurrence, verrouillage, protocole 2PL strict.

III.1 INTRODUCTION

Les réalisations actuelles dans le domaine des bases de données à objets sont nombreuses [Gardarin & Valduriez 90] : IRIS [Fishman et al. 87], GemStone [Maier et al. 86] [Penney & Stein 87], ODE [Agrawal & Gehani 89], ORION [Banerjee et al. 87a], O_2 [Lécluse et al. 88], *Trellis/Owl* [Schaffert et al. 86], V-Base [Andrews & Harris 87], etc. Ces systèmes gèrent des données dont les structures sont plus complexes que celles du modèle relationnel. Ce dernier offre des structures fixes, n-uplets et relations, et fournit des opérations pré-définies pour les manipuler : sélection, projection et jointure. Les systèmes à objets vont plus loin. Ils permettent à l'utilisateur de définir conjointement de nouvelles structures de données et des opérations spécifiques les manipulant.

Nous proposons dans ce chapitre une technique de contrôle des accès concurrents adaptée aux systèmes à objets. Tout comme le modèle permet de définir de nouvelles opérations, notre méthode permet d'étendre l'ensemble des modes d'accès, et d'en dériver automatiquement les éléments nécessaires au contrôle, ne laissant au réalisateur qu'une vue conceptuelle des modes d'accès. Notre technique est plus générale et plus adaptable que la proposition analogue d'ORION [Garza & Kim 88].

Ce chapitre est organisé de la façon suivante. Tout d'abord, nous introduisons notre modèle de données et présentons dans leur généralité les modes d'accès qui y sont associés. Ensuite, nous faisons un bref rappel du modèle transactionnel classique. L'étude du verrouillage, technique de contrôle de concurrence utilisée par notre méthode, y est décomposée en trois parties qui interviennent chacune dans l'étude des verrous et du protocole. Le protocole est ensuite étendu aux modifications : création, destruction et modifications de méthodes, puis créations et destructions d'instances, enfin créations et destructions des classes et des arcs d'héritage. Nous finirons ce chapitre en comparant notre proposition à celle d'ORION et plus succinctement à une autre proposition destinée à O_2.

III.2 LE MODÈLE À OBJETS

Dans un modèle à objets, les données et les traitements sont intimement liés : leurs créations se font conjointement. Autour de ce concept, de nombreux modèles se sont développés : les types de données abstraits issus du génie logiciel, les « *frames* » issus de l'intelligence artificielle, etc. [Masini et al. 89]. Nous nous intéressons ici aux environnements à base de classes et d'héritage issus de Smalltalk [Goldberg & Robson 83], sans retenir la notion de méta-classes. O_2 et ORION en sont deux exemples typiques.

III.2.1 Les données

III.2.1.1 Les classes

Une *classe* est un *type de données abstrait* (ADT). Elle possède des propriétés qui sont d'une part la description d'une structure de données (ou type), et d'autre part la définition d'opérations manipulant ce type. La caractéristique principale du type est d'être *totalement* invisible depuis l'extérieur de la classe ; c'est l'*encapsulation* des données. Parmi les opérations, certaines seulement sont visibles de l'extérieur ; elles constituent l'interface de la classe et sont appelées des *méthodes*.

Mais, une classe a quelque chose de plus qu'un ADT : elle connaît l'ensemble de ses instances, appelée *extension*. En ce sens, elle joue un rôle analogue à celui de la relation dans le modèle relationnel de données.

III.2.1.2 Le graphe d'héritage

Les classes sont liées entre elles par une relation d'ordre partiel. Cette relation est représentée par un graphe sans circuit muni d'une racine : le graphe d'héritage. Si le graphe est une arborescence, on parle d'*héritage simple* ; s'il s'agit d'un latticiel, on parle alors d'*héritage multiple*.

Si deux classes, c_1 et c_2, sont en relation, on appelle c_1 la *super-classe* de c_2 et c_2 la *sous-classe* de c_1. Dans les deux cas d'héritage, simple ou multiple, une classe peut avoir plusieurs sous-classes ; en revanche, il n'y a que dans l'héritage multiple qu'une classe peut avoir plus d'une super-classe.

Le graphe d'héritage possède une racine, dénommée couramment « OBJECT », qui est prédéfinie et ne peut être ni modifiée, ni supprimée.

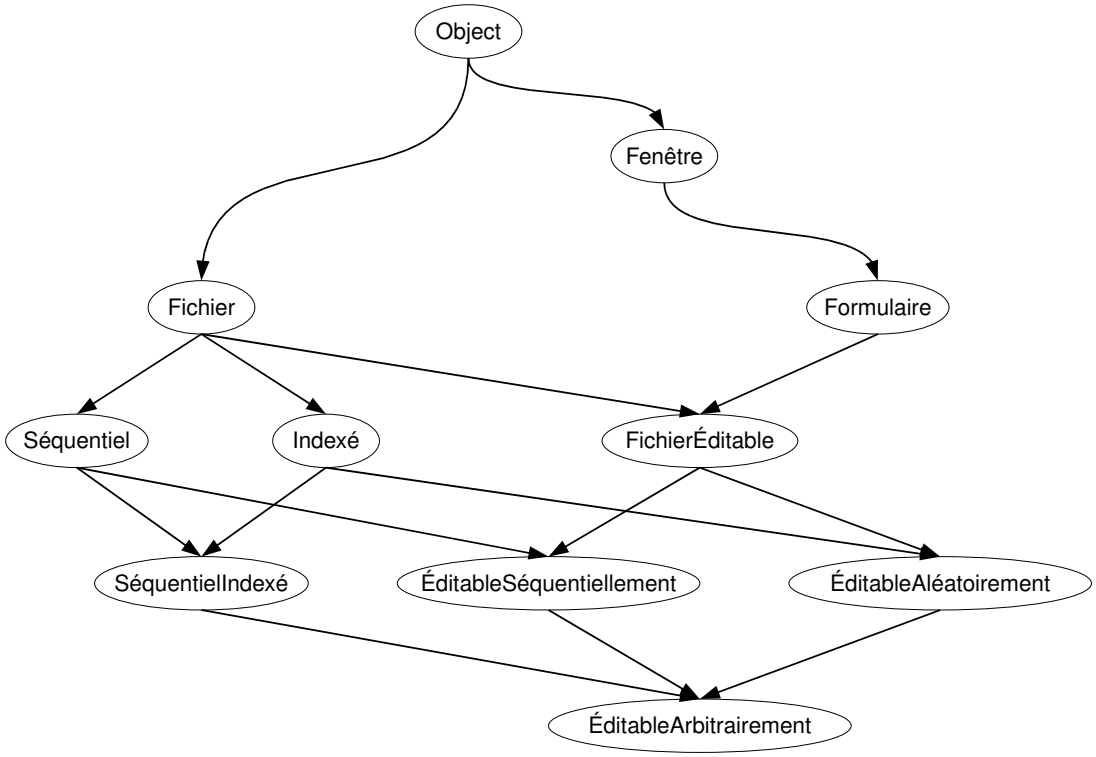


FIG. III.1 – Exemple de graphe d'héritage multiple

Nous introduisons une définition formalisée du graphe d'héritage qui sera utilisée dans la suite pour prouver la correction de notre protocole (cf. également annexe A pour les notations utilisées).

Définition III.1 (Graphe d'héritage) *Le* graphe d'héritage *est $G = (X, U) = (X, \Gamma)$ où X est l'ensemble des classes, et U la relation d'héritage, un ensemble de couples de $X \times X$.*

La relation U est un ordre partiel. Γ est la relation multivoque des successeurs d'un sommet, équivalente à U. On utilisera aussi Γ^{-1} qui représente les prédécesseurs.

On note $\lambda = (s_0, \ldots, s_n)$ et on appelle λ un chemin de G si, et seulement si :

$$\{(s_0, s_1), (s_1, s_2), \ldots, (s_{n-1}, s_n)\} \subseteq U.$$

Une racine d'un graphe est un sommet pour lequel il existe toujours un chemin vers n'importe quel autre sommet du graphe. G est muni d'une racine, notée r. Notez que dans un graphe sans circuit il ne peut y avoir au plus qu'une seule racine.

Notez que par analogie avec les représentations graphiques d'arbres, nous avons choisi d'orienter les arcs d'héritage des super-classes vers les sous-classes.

Exemple III.1 *Sur la figure III.1, nous avons représenté une hiérarchie assez complexe, mettant en œuvre plusieurs fois l'héritage multiple.*

Il y a une hiérarchie portant sur des classes de fichiers. Des fichiers peuvent être créés, ouverts, fermés, détruits. Les fichiers séquentiels peuvent être parcourus du début vers la fin. Les enregistrements des fichiers indexés sont accessibles par l'intermédiaire de clés d'accès. Enfin, les fichiers séquentiels-indexés combinent les deux modes d'accès.

En classe descendante des fenêtres, il existe une classe FORMULAIRE *qui permet de créer des fiches de saisie à l'écran. Nous en dérivons une hiérarchie parallèle et connectée deux à deux à celle des fichiers. Ainsi, un fichier séquentiel-indexé éditable sera parcouru en cliquant sur des icônes d'un formulaire ou en déroulant dans un menu les clés d'accès, etc.*

La relation d'héritage entraîne l'enrichissement des propriétés d'une classe par ajout de celles de ses super-classes. La structure de données d'une classe est composée de la somme des définitions des types de ses super-classes et de la sienne. Il s'en suit une relation de sous-typage, *isomorphe* à l'héritage [Lécluse & Richard 89a]. La classe hérite également de l'ensemble des méthodes utilisables par ses super-classes et définit elle-même de nouvelles méthodes qui manipulent sa structure de données.

III.2.1.3 Les instances

Les classes ne sont pas seulement des types de données abstraits mais également des regroupements d'objets de plus bas niveau : les *instances*. Ces dernières sont en fait les véritables objets du système. Les classes n'en ont le plus souvent que quelques caractéristiques telle que la possibilité de répondre à un ensemble restreint et pré-défini de messages. L'exemple typique est le cas du message « *new* » qui, envoyé à une classe pour créer une nouvelle instance, devrait être défini dans une méta-classe.

Une instance est une variable de même type que celui défini dans sa classe. Nous parlerons alors d'une *instance propre* de la classe, et inversement de la *classe propre* d'une instance.

III.2.1.4 Les sous-graphes

Une instance est également d'un même sous-type que celui de chacune des super-classes de sa classe propre, et, plus généralement, de toutes ses classes ancêtres dans le graphe d'héritage. Les instances d'une classe peuvent être manipulées par les méthodes définies dans une super-classe. Elles ne peuvent donc être différenciées des instances propres de la super-classe. On les appellera *instances générales*.

À chaque classe du graphe d'héritage on associe de manière unique l'ensemble des classes descendantes. Dans la suite du chapitre, nous appellerons, sans risque de confusion, ces ensembles des *sous-graphes*. Les instances générales d'une classe sont les instances du sous-graphe associé à cette classe.

Définition III.2 *Un* sous-graphe de descendants*, abrégé ici en* sous-graphe,[1] $G_{r_1} = (X_1, U_1)$ *de G est composé de* X_1*, un sous-ensemble de* X *formé de l'ensemble des descendants dans* X *d'un sommet* r_1*, et de* U_1*, la restriction de* U *à* X_1*. On définit plus formellement :*

$$X_1 = \{x \in X | \exists \lambda = (r_1, \ldots, x) \textit{ un chemin dans } G\}.$$

r_1 *est l'unique racine du sous-graphe* G_{r_1}.

La notion de sous-graphe n'est autre que celle de *domaine* d'instanciation [Banerjee et al. 87a].

III.2.2 Les accès aux objets

Les objets sont manipulés par *envoi de messages*. Cet envoi spécifie l'objet sur lequel s'applique l'opération, le nom de la méthode invoquée et des paramètres.

Différentes formes d'accès sont envisageables. On peut accéder :

- à une instance ;
- à une classe (plus exactement à ses propriétés) ;
- aux classes d'un sous-graphe ;
- aux classes d'un sous-graphe et à leurs instances.

accès à une instance L'accès à une instance est l'accès privilégié par l'envoi de messages.

accès à une classe L'accès à une classe en lecture est nécessaire lors de l'utilisation de pseudo messages, comme « *new* » qui récupère le type afin de créer une nouvelle instance.

accès aux classes d'un sous-graphe et à leurs instances L'accès à une classe C pour la modification de son type entraîne un accès en écriture à toutes les classes du sous-graphe de racine C mais également à toutes les instances de ce sous-graphe.

accès aux classes d'un sous-graphe L'accès en écriture à une classe pour la modification de ses méthodes n'entraîne que l'écriture des classes du sous-graphe dont elle est racine, sans accès aux instances.

D'autre part, le système doit également accéder à des méta-données tels que pointeurs, tables internes, index, etc. Ces méta-données peuvent être traitées séparément ou intégrées, de façon transparente pour l'utilisateur, aux données du système, notamment aux classes.

[1] Le terme approprié, en théorie des ensembles ordonnés, est *section finissante*.

III.3 LE CONTRÔLE DES ACCÈS CONCURRENTS

Le contrôle des accès concurrents est un élément important lié à la notion de transaction [Gray 78]. Une transaction est un programme qui, exécuté seul, maintient la consistance de la base de données. Cette condition n'est pas suffisante pour assurer la consistance de l'état final dès que plusieurs transactions s'exécutent concurremment. C'est le contrôle de concurrence qui est chargé d'assurer l'atomicité à la concurrence. Le critère utilisé est la *sérialisabilité* : l'exécution concurrente d'un ensemble de transactions doit être équivalente à une exécution série.

Il existe dans la littérature trois grandes catégories de contrôle de concurrence : l'estampillage et le verrouillage, toutes deux pessimistes, et la certification, optimiste [Boksenbaum et al. 86]. Nous avons retenu le verrouillage car il s'agit d'une technique éprouvée, assurant, semble-t-il, les meilleures performances [Agrawal & DeWitt 85] [Agrawal et al. 87].

Les objets sur lesquels portera le verrouillage sont les classes et les instances. Pour tenir compte des formes d'accès énumérées en section III.2.2, le contrôle de concurrence ne se contentera pas de traiter les objets indépendamment les uns des autres. Les accès sont décomposés en *type*, *mode* et *intention*. Le type de l'accès permet de sélectionner les objets, classes et instances, susceptibles d'être utilisés. Le mode d'accès correspond à la sémantique de l'opération qui va manipuler soit les classes, soit les instances, soit classes et instances sélectionnées. Enfin, l'intention permet de limiter le nombre de verrous à obtenir si toutes les instances sélectionnées sont utilisées. Les trois sections suivantes étudient successivement ces trois composantes.

III.3.1 Les types d'accès

La sémantique d'inclusion de l'héritage permet d'utiliser le graphe d'héritage pour déterminer les sous-ensembles d'objets sélectionnés. Les deux seuls types de sous-ensembles auxquels nous accordions un sens sont soit une classe et ses instances propres, soit un sous-graphe et les instances générales de sa racine.

Le verrouillage d'un sous-graphe constitue le cas général.

III.3.1.1 Le verrouillage d'un sous-graphe

Un sous-graphe étant un objet logique, notre verrouillage va reposer sur celui de certaines classes. Nous allons présenter deux propriétés (cf. corollaires III.3 et III.9) mettant en avant des classes particulières qui assurent le verrouillage d'un sous-graphe.

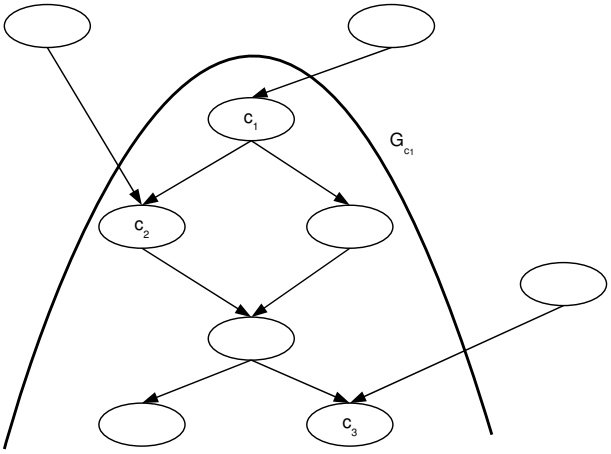


FIG. III.2 – Classes frontière d'un sous-graphe

Définition III.3 (Classe frontière) *On appelle* classe frontière *de* $G_{r_1} = (X_1, G)$ *un sommet de* X_1 *qui possède au moins un prédécesseur n'appartenant pas à* X_1*. Par convention, on pose que* r_1 *est un sommet frontière, ce qui est toujours vérifié sauf pour* r*, racine de* G*. Plus formellement, on décrit l'ensemble des classes frontière de* G_{r_1} *par :*

$$F(G_{r_1}) = \{x \in X_1 | \exists y \in \Gamma^{-1}(x), y \notin X_1\} \cup \{r_1\}$$

Exemple III.2 *Le sous-graphe* G_{c_1} *de la figure III.2 possède trois classes frontière,* c_1*,* c_2 *et* c_3*, qui constituent en quelque sorte les « points d'entrée » dans* G_{c_1}*.*

Insistons sur le fait qu'une classe n'est classe frontière que par rapport à une autre classe, racine d'un sous-graphe, et non de manière absolue.

Définition III.4 (Chemin d'accès) *On appelle un* chemin d'accès *à un sous-graphe* G_1*, un chemin quelconque joignant la classe racine du graphe d'héritage à une classe quelconque de* G_1*.*

On choisit arbitrairement un chemin d'accès à G_1 *et on appelle* classe de chemin *de* G_1 *toute classe appartenant à ce chemin et n'appartenant pas à* G_1*.*

Les sous-graphes ne sont pas des objets indépendants comme les classes et les instances. Deux sous-graphes peuvent partager tout ou partie de leurs classes (et instances). Plus précisément, avec l'héritage multiple, deux graphes peuvent posséder une intersection stricte, c'est-à-dire une intersection non vide et qui n'est ni l'un ni l'autre des deux sous-graphes ; dans les deux cas d'héritage, simple ou multiple, un sous-graphe peut être entièrement inclus dans un autre.

La notion de classes frontière nous permet d'exhiber une propriété pour l'intersection stricte et une autre pour l'inclusion.

Définition III.5 *Deux sous-graphes, G_1 et G_2, ont une* intersection stricte *quand :*

- $X_1 \cap X_2 \neq \emptyset$;
- $X_1 \cup X_2 \neq X_1$;
- $X_1 \cup X_2 \neq X_2$.

Définition III.6 *Soient $G_{r_1} = (X_1, U_1)$ et $G_{r_2} = (X_2, U_2)$ deux sous-graphes possédant une intersection stricte, alors on définit :*

$$d : \begin{array}{ccc} X_1 \cap X_2 & \rightarrow & \mathbb{N} \\ x & \mapsto & l_1 + l_2 \end{array}$$

avec $\lambda_1 = (r_1 = t_0, \ldots, t_{l_1} = x)$ et $\lambda_2 = (r_2 = s_0, \ldots, s_{l_2} = x)$ des chemins de longueur maximale *dans G_{r_1} entre r_1 et x et similairement dans G_{r_2}.*

Propriété III.1 *Soient $G_{r_1} = (X_1, U_1)$ et $G_{r_2} = (X_2, U_2)$ deux sous-graphes possédant une intersection stricte, alors :*

$$\begin{array}{l} \forall x \in X_1 \cap X_2 \text{ avec } d(x) \text{ minimal}, \\ \quad \exists y \in \Gamma^{-1}(x), \\ \quad\quad y \notin X_1 \wedge \\ \quad \exists y' \in \Gamma^{-1}(x), \\ \quad\quad y' \notin X_2. \end{array}$$

Preuve III.2 *Procédons par contradiction et, sans perte de généralité, supposons que, x ayant été choisi avec $d(x)$ minimal, $\forall y \in \Gamma^{-1}(x), y \in X_1$, alors :*

(i) si $\forall y \in \Gamma^{-1}(x), y \notin X_2$ alors $x \notin X_1 \cap X_2$ car G_2 est un sous-graphe de descendants, donc contradiction ;

(ii) si $\exists y \in \Gamma^{-1}(x), y \in X_2$ alors $y \in X_1 \cap X_2$ et $d(y)$ est défini, avec $d(y) \leq d(x) - 2$, or $d(x)$ est minimal, donc contradiction.

(iii, iv) les deux derniers cas se ramènent aux deux premiers.

Nous arrivons au premier corollaire qui nous importe pour notre protocole de verrouillage dans un graphe d'héritage.

Corollaire III.3 *Si deux sous-graphes ont une intersection stricte, alors il existe au moins une classe commune qui est classe frontière de chacun des deux sous-graphes.*

Preuve III.4 *Les sommets minimaux, au sens ci-dessus, sont des sommets frontière.*

Exemple III.3 *Les sous-graphes G_1 et G_2 de la figure III.3 possèdent une intersection stricte. La classe c_1 est simultanément classe frontière de G_1 et de G_2.*

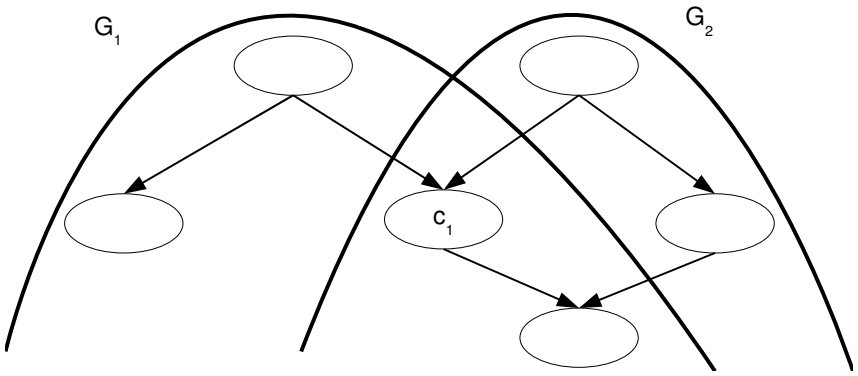


FIG. III.3 – Intersection stricte entre deux sous-graphes

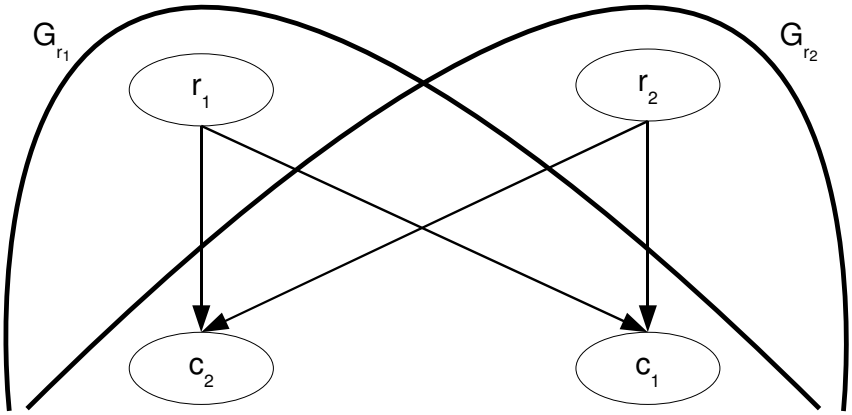


FIG. III.4 – Contre-exemple à un théorème de nécessité

Si le corollaire III.3 démontre que la notion de classes frontière est suffisante pour détecter les intersections strictes de sous-graphes, elle n'est pas nécessaire.

Exemple III.4 *Sur la figure III.1, les sous-graphes issus des classes* FICHIERÉDITABLE *et* INDEXÉ *ont deux classes frontière en commun.*

Plus généralement, dès qu'apparaît la forme d'héritage représentée en figure III.4, il y a plus d'une classe frontière commune dans l'intersection (ce qui peut entraîner des interblocages si elles ne sont pas verrouillées dans le même ordre par les transactions accédant à G_{r_1} et G_{r_2}).

Passons maintenant à la démonstration d'un second corollaire qui nous permet de traiter le cas de l'inclusion de sous-graphes.

Lemme III.5

$$\forall \lambda = (s_1 = t_0, \ldots, t_n = s_2) \textit{ avec } n > 0,$$
$$G_{s_2} \subset G_{s_1}.$$

Preuve III.6 *Supposons que $G_{s_1} \subseteq G_{s_2}$, alors $s_1 \in G_{s_2}$, or G_{s_2} est un sous-graphe de descendants, donc il existe $\lambda' = (s_2 = t'_0, \ldots, t'_m = s_1)$ et avec λ un circuit dans G, une contradiction.*

Propriété III.7

$$\begin{aligned}
&\forall \lambda = (s_1 = t_0, \ldots, t_n = s_2) \textit{ avec } n > 0,\\
&\forall \mu = (s_1 = r_0, \ldots, r_m = s_2) \textit{ avec } m > 0,\\
&\forall t_i : 1 \leq i \leq n,\\
&\exists r_j : 1 \leq j \leq m,\\
&\quad r_j \in G_{t_i} \wedge\\
&\quad r_{j-1} \notin G_{t_i}.
\end{aligned}$$

Preuve III.8 *Soient λ, μ et $i : 1 \leq i \leq n$, alors à partir du cas particulier $s_2 \in G_{s_2}$, où $s_2 = r_m$ et $G_{s_2} = G_{t_n}$, et sachant que $G_{t_n} \subseteq G_{t_i}$, nous en déduisons que $\exists r_j : 0 \leq j \leq m, r_j \in G_{t_i}$.*

Choisissons donc le plus petit j tel que $r_j \in G_{t_i}$. Si $j = 0$ alors, comme $r_0 = s_1 = t_0$, nous obtenons $t_0 \in G_{t_i}$, contredisant le lemme III.5 puisque $i \geq 1$. Donc $j > 0$ et, par choix minimal, $r_{j-1} \notin G_{t_i}$.

Le second corollaire nous permet d'achever la preuve de correction globale de notre protocole.

Corollaire III.9 *Soient G_1 et G_2 deux sous-graphes avec $G_1 \subseteq G_2 \subseteq G$, alors sur tout chemin d'accès à G_1, il existe au moins un sommet frontière de G_2.*

Preuve III.10 *Il suffit de spécialiser le sommet s_1 de la propriété III.7 au sommet r, racine du graphe G. Les r_j sont des sommets frontière. Enfin, $r = r_0$ est, par définition, un sommet frontière.*

Exemple III.5 *Sur la figure III.5, G_1 est un sous-graphe inclus dans G_2. Quel que soit le chemin choisi entre r, racine du graphe d'héritage et une classe quelconque de G_1, c_1 ou c_2 sont des classes frontière de G_2 qui appartiennent à l'un des chemin.*

Corollaire III.11 *Le corollaire III.9 s'applique également aux cas d'un sous-graphe réduit à une classe isolée et incluse dans un sous-graphe de descendant.*

Preuve III.12 *Évident.*

III.3.1.2 Les différents types de verrous

Pour verrouiller un sous-graphe et par là sélectionner les classes et les instances qui le composent, il faut poser des verrous sur les classes particulières mises en évidence dans la section précédente. Ces classes ne se comportent pas toutes de la même façon. Vis-à-vis d'un sous-graphe donné, nous distinguons :

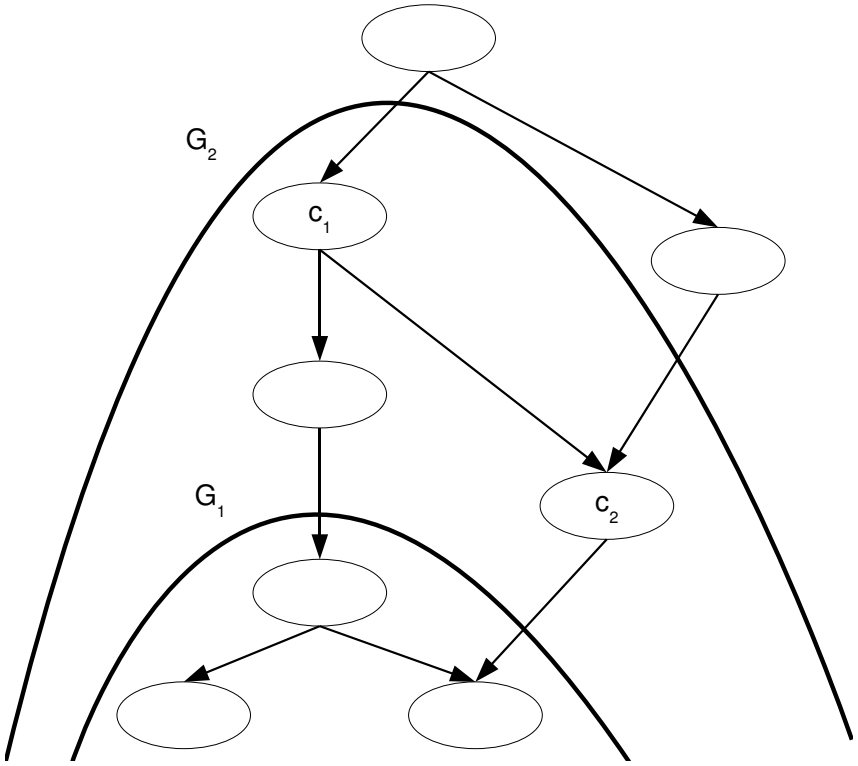


FIG. III.5 – Inclusion de sous-graphes

	Chemin	Classe	Frontière
Chemin	*oui*	*oui*	non
Classe	*oui*	non	non
Frontière	non	non	non

TAB. III.1 – Compatibilité C_T des types de verrous sur les classes

- des *classes frontière*, s'il s'agit d'un sous-graphe de descendants, sur lesquelles on pose des *verrous de frontière* ;
- une *classe isolée*, s'il s'agit d'un sous-graphe réduit à cette classe, sur laquelle on pose un *verrou de classe* ;
- des *classes de chemin*, sur lesquelles on pose des *verrous de chemin*.

Définition III.7 *Nous introduisons un prédicat de compatibilité entre types de verrous :*

$$C_T : \mathit{TYPES} \times \mathit{TYPES} \rightarrow \{oui, non\}$$

où $\mathit{TYPES} = \{\mathit{Chemin}, \mathit{Classe}, \mathit{Frontière}\}$.

La relation est donnée en extension en Table III.1.

Les compatibilités apparaissant en table III.1 ne peuvent être pleinement comprises qu'après lecture du protocole (cf. section III.3.5). Toutefois, à partir des figures III.2 à III.5 et des définitions, le lecteur s'imagine déjà certainement quelle réflexion a mené à cette relation.

Les verrous de chemin sont rendus compatibles entre eux car deux sous-graphes *distincts* peuvent partager tout ou partie de leur chemin d'accès. Par exemple, les sous-graphes de classes racines c_1 et c_2 de la figure III.4 peuvent tous deux choisir le chemin d'accès passant par la classe r_2 ; il n'y a cependant aucune raison de rendre des accès à ces sous-graphes incompatibles au niveau de r_2. Plus trivialement, tout chemin aboutissant finalement à la classe « OBJECT », racine du graphe d'héritage, le choix contraire aurait supprimer tout parallélisme.

Les verrous de classe et de chemin sont rendus compatibles car le sous-graphe (ou la classe isolée) auquel aboutit un chemin d'accès peut passer par une classe isolée. Les objets manipulées au niveau du sous-graphe et à celui de la classe isolée sont distincts.

Enfin, le verrou de frontière est incompatible avec tout autre car il existe toujours des données communes : existence d'une intersection stricte si ce sont deux verrous de frontière, inclusion stricte s'il s'agit d'un verrou de frontière et d'un verrou de chemin, partage d'une classe quand il y a un verrou de frontière et un verrou de classe.

Passons maintenant à la seconde composante, classique : les modes d'accès.

III.3.2 Les modes d'accès

[Korth 83] a montré qu'il est souhaitable d'associer à chaque opération un mode d'accès (cf. chapitre IV) ; autrement, un mode d'accès représente un ensemble d'opérations. Par exemple, le mode d'accès *E*, pour écrire une instance, regroupe l'ensemble des opérations différentes qui sont des écritures particulières. Un mode d'accès véhicule donc une sémantique liée à l'ensemble des opérations qu'il regroupe.

Toutes les opérations ne peuvent pas se faire en parallèle sans risquer de violer la sérialisabilité ; il en est de même pour les modes d'accès. Chaque mode d'accès résume l'ensemble des restrictions de ses opérations. Ces restrictions se représentent sous la forme d'un prédicat de compatibilité.

Définition III.8 *Nous introduisons un prédicat de compatibilité entre modes d'accès :*

$$C_M : \mathit{MODES} \times \mathit{MODES} \rightarrow \{oui, non\}$$

Pour illustrer notre propos, nous définissons quatre modes d'accès : *L*, *E*, *LD* et *MD*. Les modes *L* et *E* portent sur les instances ; ils représentent respectivement la lecture et l'écriture d'une instance. Le mode *LD* s'appliquant aux classes, autorise la lecture de la définition (variables d'instance, variables de classe éventuellement, méthodes). Enfin le mode *MD*, de modification de la définition d'une classe, s'étend aux classes et aux instances d'un sous-graphe. Notez bien que nous avons ainsi mélangé des modes d'accès s'appliquant soit à une instance,

	L	*E*	*LD*	*MD*
L	*oui*	non	*oui*	non
E	non	non	*oui*	non
LD	*oui*	*oui*	*oui*	non
MD	non	non	non	non

TAB. III.2 – Compatibilité C_M de quelques modes d'accès

(a) verrouillage en lecture explicite

(b) verrouillage en lecture implicite

FIG. III.6 – Du verrouillage explicite au verrouillage implicite

soit à une classe, soit à un sous-graphe. Leur relation de compatibilité est donnée en table III.2. Cette table pourra s'agrandir d'autres modes ; ce sera fait en section III.4.

Passons à la troisième et dernière composante.

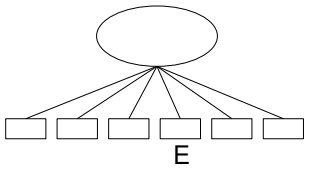
III.3.3 Le verrouillage implicite, intentionnel et hiérarchique

Nous avons vu (cf. section III.3.1) que le verrouillage d'un sous-graphe est utile et que celui d'une classe n'en est qu'un cas particulier. Il nous faut encore traiter le cas du verrouillage des instances. Celui-ci est rendu délicat par la nature des modes d'accès qui peuvent s'appliquer globalement à un ensemble de classes et d'instances, comme *MD*, ou à une instance isolée, comme *E*.

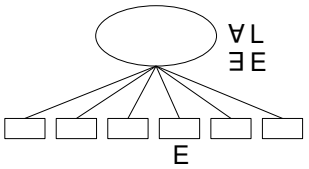
Il n'est pas envisageable de toujours verrouiller individuellement chaque instance, lors d'un accès en mode *MD* par exemple. Afin de minimiser le nombre de verrous à poser, le verrouillage du sous-graphe, classes chemin et frontière, impliquera le verrouillage implicite de toutes les instances du sous-graphe (cf. figure III.6).

Il faut alors pouvoir détecter une incompatibilité avec une transaction qui accéderait à une instance de ce sous-graphe en mode *E*. Pour résoudre ce problème, la transaction devra verrouiller l'instance bien sûr, mais également la classe (propre) de cette instance en mode *E* intentionnel. Donc, l'utilisation du verrouillage implicite nécessite celui du verrouillage intentionnel (cf. figure III.7).

Il est également possible que l'accès *E* soit répété sur l'ensemble des instances de la classe, ou d'un sous-graphe, ou tout au moins une majorité. Dans ce cas, il est intéressant d'éviter le verrouillage explicite et successif de chaque instance, en verrouillant directement la classe, ou le sous-graphe, en mode *E* hiérarchique. En fait, il s'agit d'un verrouillage hiérarchique limité



(a) verrouillage en écriture explicite



(b) verrouillage en écriture explicite et en lecture implicite

FIG. III.7 – Nécessité du verrouillage intentionnel avec le verrouillage implicite

	Explicite	Implicite
Explicite	*oui*	non
Implicite	non	non

TAB. III.3 – Compatibilité C_I des intentions d'accès aux instances

à deux niveaux : instances et extensions de classes (cf. verrous *IS* et *S*, *IX* et *X* et protocole hiérarchique de [Gray 78], présenté dans le chapitre I).

Définition III.9 *Nous introduisons un prédicat de compatibilité entre les intentions d'accès :*

$$C_I : \mathit{INTENTIONS} \times \mathit{INTENTIONS} \rightarrow \{oui, non\}$$

où $\mathit{INTENTIONS} = \{\mathit{Explicite}, \mathit{Implicite}\}$.

La relation est donnée en extension en table III.3.

L'intention d'accès est équivalente à un booléen, C_I représentant la relation « et logique ».

Les trois composantes que nous avons détaillées, type, mode et intention, sont toutes utiles pour caractériser les accès concurrents. Nous allons les combiner afin d'obtenir les verrous de notre proposition.

III.3.4 Les verrous

On distingue deux genres de verrous : les verrous que l'on pose sur les classes et les verrous que l'on pose sur les instances. Les verrous sont des éléments complexes. Pour simplifier leur écriture, nous leur donnerons des noms.

III.3.4.1 La construction des verrous

Définition III.10 *Un verrou sur une classe est un élément de :*

$$V_{\mathit{CLASSE}} = \mathit{TYPES} \times \mathit{MODES} \times \mathit{INTENTIONS}.$$

On définit le prédicat de compatibilité entre verrous de classes comme la disjonction *des trois prédicats portant sur ses composantes :*

$$C_{VC} : \begin{array}{ccc} V_{CLASSE} \times V_{CLASSE} & \rightarrow & \{oui, non\} \\ ((t_1, m_1, i_1), (t_2, m_2, i_2)) & \mapsto & C_T(t_1, t_2) \vee C_M(m_1, m_2) \vee C_I(i_1, i_2). \end{array}$$

Le type d'accès du verrou est utilisé afin de sélectionner l'ensemble des objets, classes et instances, susceptibles d'être utilisés par la transaction qui accède au sous-graphe.

La sémantique du mode d'accès indique, de façon générale, la façon dont les objets sélectionnés sont manipulés, et, de manière plus particulière, quels sont les objets utilisés : classes et/ou instances.

Si un mode d'accès implique que toutes les instances sélectionnées sont utilisées, alors la valeur de la composante intentionnelle est inutile (plus exactement, elle doit être positionnée à « Explicite » pour ne pas introduire de pseudo incompatibilité) ; autrement, « Explicite » nécessitera le verrouillage explicite des instances effectivement utilisées, tandis que « Implicite » autorisera la transaction à accéder à toutes les instances (soit de la classe, soit du sous-graphe), sans procéder à des demandes de verrous sur celles-ci.

La formulation sous forme *disjonctive* traduit l'*indépendance* des trois composantes d'un verrou. Il suffit que les transactions n'accèdent pas aux mêmes sous-ensembles d'objets, ou, si c'est le cas, que les modes d'accès soient compatibles, ou enfin, les deux premiers cas étant incompatibles, que les deux transactions verrouillent explicitement les instances.

Un verrou sur une instance se limite à $V_{\text{INSTANCE}} = \mathit{MODES}$. Il est présenté dans [Aldebert & Martinez 89], trois types de verrous sur les instances, similaires aux trois types de verrous sur les classes, répondant au problème du verrouillage des objets composites dans ORION [Garza & Kim 88] [Kim et al. 87] [Kim et al. 89]. Cette proposition augmente le parallélisme au détriment du nombre de verrous à acquérir.

III.3.4.2 Les noms des verrous

Pour alléger l'écriture, nous donnons aux verrous sur les classes des noms récapitulant typographiquement les trois composantes : *TYPES*, *INTENTIONS*, *MODES*. Les trois types de verrous, « Chemin », « Classe » et « Frontière », seront respectivement représentés par les symboles $\downarrow$, $\forall$ et ε (le symbole vide). Le mode intentionnel, « Explicite » ou « Implicite », sera représenté par les symboles $\exists$ et $\forall$. L'ambiguïté due à la double utilisation du symbole $\forall$ sera levée par le mode du verrou. Chaque mode sera directement représenté par son nom.

Les noms de verrous seront obtenus par concaténation du symbole du type, éventuellement du symbole de l'intention, et du nom du mode. Parmi les modes, on peut distinguer les modes s'appliquant à des instances (*L* et *E*), à des classes (*LD*) ou à des sous-graphes (*MD*). Le symbole

	Chemin		Classe		Frontière		Instance
Intention	Explicite	Implicite	Explicite	Implicite	Explicite	Implicite	
L	$\downarrow \exists L$	$\downarrow \forall L$	$\exists L$	$\forall L$	$\forall\exists L$	$\forall\forall L$	*L*
E	$\downarrow \exists E$	$\downarrow \forall E$	$\exists E$	$\forall E$	$\forall\exists E$	$\forall\forall E$	*E*
LD	$\downarrow$ *LD*		*LD*		$\forall$*LD*		
MD		$\downarrow$ *MD*				$\forall$*MD*	

TAB. III.4 – Tous les verrous obtenus à partir des modes d'accès *L*, *E*, *LD* et *MD*

d'intention n'est utilisé que pour les modes s'appliquant à des instances.

Par exemple, avec le mode d'accès *E* à une instance, on obtient six verrous qui se posent sur des classes : $\downarrow \exists E$,[2] $\downarrow \forall E$, $\forall\exists E$, $\forall\forall E$,[3] $\exists E$ et $\forall E$. Avec le mode d'accès *LD* à une classe, on obtient trois verrous s'appliquant aux classes : $\downarrow$ *LD*, $\forall$*LD* et *LD*. Avec le mode d'accès *MD* à un sous-graphe, on obtient seulement deux verrous : $\downarrow$ *MD* et $\forall$*MD*. Le verrou *MD* ne peut exister de par la sémantique même de l'opération.

Quant aux noms des verrous d'instances, ce sont ceux du mode : le nom *L* pour le mode *L*, le nom *E* pour le mode *E*.

On retrouve en table III.4 les différents noms de verrous formés à partir des modes *L*, *E*, *LD* et *MD*. On obtient, à partir de quatre modes d'accès, dix-neuf verrous. L'intérêt évident de cette construction des verrous est de ne pas nécessiter une matrice de compatibilité de 361 intersections, mais de se limiter à trois prédicat modestes, C_M, C_T et C_I, et deux disjonctions.

La création des verrous ne constitue que la première partie de cette étude. Il nous faut maintenant détailler le protocole esquissé en section III.3.1 à partir des définitions de classes frontière et de chemin.

III.3.5 Le protocole

Notre protocole s'expose en quatre points :

1. Une transaction respecte le verrouillage à deux phases (2PL) [Eswaran et al. 76] (cf. chapitre I). Avant d'accéder à quelque objet logique que ce soit, la transaction doit au préalable obtenir un verrou, implicite ou explicite, sur cet objet.
2. Pour accéder à un ensemble d'objets, une transaction doit :
   (a) poser un verrou sur toutes les classes frontière du sous-graphe auquel (de la classe isolée à laquelle) elle veut accéder, et,
   (b) poser un verrou sur toutes les classes se trouvant sur un chemin quelconque, de

[2]Ce symbole doit se lire : « Dans une classe strictement descendante de celle-ci, il existe au moins une instance utilisée en écriture ».

[3]Ce symbole doit se lire : « Dans toutes les classes descendantes, au sens large, de celle-ci, toutes les instances sont utilisées en écriture ».

préférence le plus court,[4] entre une classe frontière du sous-graphe (de la classe) et la racine du graphe d'héritage.

Rappelons que la classe frontière (isolée) ne fait pas partie du chemin d'accès (cf. définition III.4) car le verrou de frontière (de classe) est davantage restrictif que celui de chemin (cf. table III.1).

3. Si le verrouillage du sous-graphe (de la classe) n'implique pas un verrouillage implicite de toutes les instances et que certaines sont utilisées, alors il faudra verrouiller chaque instance séparément avant de l'utiliser.
4. Le protocole 2PL que nous utilisons est strict, c'est-à-dire que les verrous ne sont relâchés qu'à la validation de la transaction. L'ordre de libération des verrous peut alors être quelconque.

Les points 2 et 3 de ce protocole ont été traduits en procédures avec l'algorithme III.1. Ce protocole n'impose aucun ordre dans la pose des verrous, que ce soit entre les points 2 et 3 aussi bien qu'à l'intérieur du point 2. Néanmoins, pour éviter certains interblocages, il est souhaitable de choisir un ordre de pose des verrous sur les classes qui soit global à toutes les transactions du système (de haut en bas et de gauche à droite, dans l'ordre lexicographique des noms de classes, etc.).

Nos verrous s'appliquent à des objets logiques, instances et classes. La représentation de ceux-ci dans des pages physiques (cf. exemple II.20) nécessite un protocole adapté. Il s'agit du protocole issu de *System R* d'IBM utilisant des « verrous longs » sur les objets logiques et des « verrous courts » sur les pages physiques, qui se généralise aux transactions multi-niveaux [Beeri et al. 88] vues dans le chapitre I (cf. surtout [Cart & Ferrié 90] pour une proposition adaptée à O_2 ainsi que la section III.5.2 qui effectue une rapide comparaison).

Exemple III.6 *Illustrons le protocole sur la figure III.8. Soient deux transactions concurrentes : T_1 accède en mode écriture aux instances générales de c_1 ; T_2 accède en mode lecture à certaines instances de c_4. Nous avons alors un exemple de conflit lecture/écriture entre T_1 qui va verrouiller implicitement les instances générales de c_1, donc les instances de c_4, et T_2 qui va verrouiller explicitement certaines instances de c_4. La transaction T_1 va devoir poser les verrous écrits à gauche des classes r, c_1 et c_3 de l'exemple, tandis que T_2 posera les verrous notés à droite des classes r, c_2, c_3 et c_4. Ensuite, la transaction T_2 devra poser des verrous L sur chaque instance à laquelle elle accédera réellement.*

Détaillons maintenant cet exemple, en choisissant de poser d'abord les verrous sur le graphe d'héritage, de haut en bas et de gauche à droite (des algorithmes plus précis sont indiqués en section III.4.2.2), puis sur les instances si nécessaire :

[4] Ceci minimise le nombre de verrous à acquérir. Malheureusement, le chemin le plus court peut être très coûteux à obtenir ; cela ne sera possible que s'il est calculé pendant la phase de compilation et que les modifications du graphe d'héritage ne sont pas autorisées dans la phase d'exécution.

Algorithme III.1 Procédures réalisant les points 2 et 3 du protocole

```
type
   MODES = (L, E, LD, MD, ...) ;
   VCLASSE = record
                Type : (Frontière, Classe, Chemin) ;
                Mode : MODES ;
                Intention : boolean
             end ;
   VINSTANCE = MODES ;

   procedure VerrouillerClasses (ClasseRacine : CLASSES ;
                                 SousGraphe : boolean) ;
                                 ModeAccès : MODES ;
                                 ToutesInstances : boolean) ;
   var
      Verrou : VCLASSE ;
   begin
      Verrou.Type := Chemin ;
      Verrou.Mode := ModeAccès ;
      Verrou.Intention := ToutesInstances ;
      for C in ClassesChemin(ClasseRacine, SousGraphe)
      loop
         VerrouillerClasse(C, Verrou) ;
      end loop ;
      if SousGraphe
      then
         Verrou.Type := Frontière ;
         for C in ClassesFrontière(ClasseRacine)
         loop
            VerrouillerClasse(C, Verrou)
         end loop
         else
            Verrou.Type := Classe ;
            VerrouillerClasse(ClasseRacine, Verrou)
         end if
      end ;

      procedure VerrouillerInstance (Instance : Instances ;
                                     ModeAccès : MODES) ;
      begin
         if not DéjàVerrouillée(ClassePropre(Instance),
                                ModeAccès, /* dans le même mode, */
                                true) /* verrouillage implicite ? */
         then
            VerrouillerClasses(ClassePropre(Instance),
                               false, /* uniquement la classe, */
                               ModeAccès,
                               false) ; /* et verrouillage explicite */
            /* Verrouiller effectivement l'instance */
         end if
      end ;
```

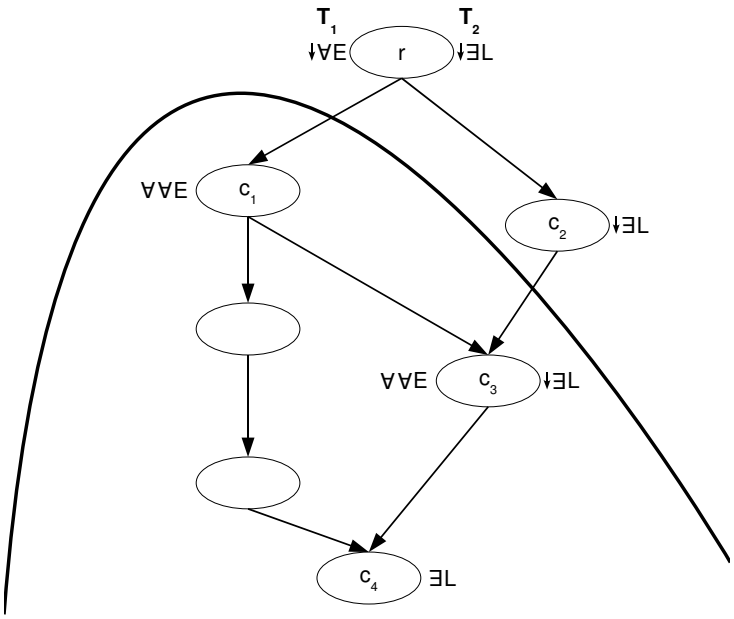


FIG. III.8 – Exemple de verrouillages concurrents dans un graphe d'héritage

1. T_1 *pose le verrou* $\downarrow \forall E$ *sur la classe* r *;*
2. T_2 *pose le verrou* $\downarrow \exists L$ *sur la classe* r*, les deux verrous sont compatibles car il s'agit de verrous de chemin (cf. table III.1) ;*
3. T_2 *pose un autre verrou* $\downarrow \exists L$ *sur la classe* c_2 *;*
4. T_1 *pose un verrou* $\forall\forall E$ *sur la classe* c_1*, racine et classe frontière du sous-graphe aux instances duquel* T_1 *veut accéder en totalité ;*
5. T_2 *pose un verrou* $\downarrow \exists L$ *sur la classe* c_3 *;*
6. T_1 *tente de poser un verrou* $\forall\forall E$ *sur la classe* c_3*, mais ce verrou étant incompatible avec le verrou de chemin de* T_2*,* T_1 *se bloque :*
7. T_2 *n'a plus qu'à poser un verrou de classe* $\exists L$ *sur la classe* c_4 *pour finir son verrouillage du graphe d'héritage.*
8. T_2 *peut maintenant tenter de poser les verrous L sur les instances auxquelles elle veut accéder.*

L'instant critique est le point 5. En effet, c_3 *est la première et seule classe commune pendant le verrouillage de* T_1 *et de* T_2*. La première transaction à verrouiller* c_3 *est donc celle qui sera libre de continuer alors que la seconde se bloquera.*

Si la transaction T_2*, au lieu d'accéder à certaines instances de* c_4*, avait voulu lire la définition de* c_4*, c'est-à-dire utiliser le mode d'accès LD (verrous* $\downarrow$ *LD sur* r*,* c_2 *et* c_3*, puis verrou LD sur* c_4*), alors les deux transactions auraient pu s'exécuter concurremment. Ce parallélisme-là n'est pas possible avec le protocole d'ORION.*

III.4 LES MODIFICATIONS

En ce point, notre protocole ne permet pas de traiter les modifications. Elles sont au nombre de trois : créations et destructions d'instances ; puis, créations, destructions et modifications de méthodes ; enfin, modifications du graphe d'héritage par ajout ou suppression de classes ou d'arcs d'héritage.

Les créations et destructions d'instances sont à envisager quand bien même les autres seraient interdites. Nous commencerons donc par elles.

III.4.1 Les créations et destructions d'instances

Ce problème peut être traité très simplement ou de façon plus délicate suivant le degré de connaissance que les classes ont de leurs instances. Une classe peut :

- ne pas connaître ses instances : c'est le cas par défaut dans O_2 ;
- connaître ses instances propres ;
- connaître ses instances générales : c'est le cas dans O_2 si l'on crée une classe avec la clause « *with extension* » [Lécluse & Richard 89b] ;
- connaître ses instances propres et ses instances générales : c'est le cas dans ORION [Banerjee et al. 87a].

Le traitement varie suivant que la classe connaît ou ne connaît pas certaines de ses instances.

aucune connaissance des instances Pour une classe qui ne joue pas le rôle d'une relation du modèle relationnel, aucune extension n'est nécessaire au protocole ! En effet, l'OID de l'instance créée ou détruite n'est connu que de la transaction qui a demandé cette opération. Toute affectation ultérieure de l'OID nécessite l'obtention d'un verrou *E*, exclusif, sur l'instance à laquelle appartient la variable d'instance qui recevra cette valeur. (Bien que nous n'en ayons pas tenu compte dans la présentation du modèle, ceci est également valable pour les instances nommées dans O_2 [Lécluse & Richard 89b].)

connaissance des instances propres ou générales Dans les trois autres cas, le verrouillage hiérarchique en mode exclusif sur la classe propre de l'instance résout, assez brutalement, le problème des objets « fantômes[5] » (cf. chapitre I). Il faut donc créer un nouveau mode s'appliquant à une classe, *CDI* pour « création et/ou destruction d'instances », incompatible avec *L*, *E*,

[5]Supposons qu'une transaction sélectionne tous les cadres d'une entreprise. Si, concurremment, un nouveau cadre est ajouté, alors nous obtenons un objet « fantôme » vis-à-vis de la première transaction.

MD et lui-même, mais pas avec *LD*. Ce mode peut s'appliquer à un sous-graphe en utilisant le verrou de frontière $\forall CDI$.

Le problème des objets « fantômes » est lié au verrouillage prédicatif. Même la forme restreinte de prédicat proposée initialement par [Eswaran et al. 76] n'a pas été retenue dans *System R* car tester la satisfiabilité de leurs conjonctions était déjà trop coûteux [Chamberlin et al. 81]. Le verrouillage hiérarchique de [Gray 78] offre alors un jeu de prédicats pré-définis sous la forme d'un latticiel de granules de verrouillage (niveaux n-uplet, relation, index, segment et base de données). Or, verrouiller une classe ou un sous-graphe est également une forme, très restreinte, de verrouillage prédicatif ; on sélectionne bien un ensemble d'instances vérifiant la propriété d'appartenir à une classe donnée.

Pour rendre notre solution moins abrupte quant à l'exclusivité qu'elle entraîne, on peut intégrer dans le modèle de données la notion de *classes virtuelles*. Une classe virtuelle est une sous-classe d'une classe normale qui n'a comme propriété supplémentaire que celle de regrouper le sous-ensemble des instances de sa super-classe qui satisfont un prédicat donné [Abiteboul 89] [Abiteboul & Bonner 91]. (Cette notion est analogue à celle de vue dans le modèle relationnel.) En considérant les classes virtuelles comme des classes normales pour le contrôle de concurrence, on pourra limiter l'exclusivité d'accès à la classe la plus particulière.

Exemple III.7 *Considérons la classe* PERSONNE. *Il y est défini un champs « Sexe : (Masculin, Féminin) ». On pourra alors créer deux sous-classes virtuelles, Homme et Femme, qui vérifieront respectivement les prédicats « Sexe = Masculin » et « Sexe = Féminin » (cf. aussi la forme d'héritage par contrainte présentée dans l'introduction).*

Lors de la création d'une nouvelle instance de la classe PERSONNE*, l'évaluation des prédicats permet de déterminer dans quelle classe virtuelle, et donc plus précise, on peut ajouter l'instance. Ainsi, la création de nouveaux hommes pourra se faire en concurrence avec une transaction qui calculerait l'âge moyen des femmes.*

Toutefois, de nouveaux problèmes se posent, que nous ne ferons qu'énumérer car le protocole de contrôle de concurrence adapté dépendra des réponses apportées sur plusieurs points :

(i) Les prédicats ne définissent pas nécessairement des ensembles disjoints (dans l'exemple III.7, si l'on ajoute la classe virtuelle ENFANT des instances vérifiant le prédicat « $\hat{A}ge \leq 10$ ») et détecter les intersections n'est pas décidable si l'on ne restreint pas leur forme.

(ii) Si l'on ne veut pas restreindre la forme des prédicats, alors on risque d'avoir des instances appartenant à plusieurs classes virtuelles (mais qui seront toujours instances d'une seule classe réelle).

(iii) Dans ce cas, doit-on autoriser le système à créer automatiquement de nouvelles classes virtuelles par héritage multiple, au risque d'une explosion combinatoire ?

(iv) De manière plus générale, peut-on créer des sous-classes virtuelles de classes virtuelles ? En supposant que l'on effectue la conjonction des prédicats se trouvant sur un chemin, cela risque d'entraîner un nombre important d'évaluation de prédicats, surtout en considérant le point suivant.

(v) À chaque exécution de méthode modifiante sur une instance d'une classe virtuelle, il faut éventuellement la faire migrer de sa classe virtuelle vers une autre et donc gérer automatiquement le protocole de création et destruction d'instances sur la classe d'origine et celle d'arrivée[6] (par exemple, dans une application médicale, si l'état de santé d'un patient devient « mauvais », son instance doit être transférée dans la classe PATIENTCRITIQUE [Estier & Guyot 91]).

(vi) Peut-on créer des sous-classes normales de classes virtuelles ? (Cela signifie soit que le prédicat devient une contrainte d'intégrité, soit que le système est autorisé à éliminer les variables d'instances définies dans la sous-classe normale pour transférer l'instance vers une autre classe virtuelle.)

Remarquez que les créations automatiques de classes virtuelles, contrairement à ce qui a été écrit en introduction de cette section, ne permettent plus de considérer les créations et destructions d'instances comme indépendantes des modifications du graphe d'héritage.

III.4.2 Les modifications du graphe d'héritage

Les opérations de modifications du graphe d'héritage posent de nouveaux problèmes pour le contrôle de concurrence, qui sont essentiellement liés au fait que les classes frontière et de chemin de certaines parties du graphe sont changées. Nous proposons toutefois une extension du protocole capable de traiter ces modifications de façon concurrente et sérialisable. Nous présentons auparavant les problèmes de cohérence posés en général par ces modifications.

III.4.2.1 Les problèmes de cohérence

Parmi les modifications du graphe d'héritage, et en procédant par ordre de difficulté croissante, on trouve :

- la création d'un arc d'héritage ;
- la destruction d'un arc d'héritage ;
- la création d'une classe ;
- la destruction d'une classe.

Si, dans O_2, les créations et suppressions de classes ne sont autorisées qu'en puits du graphe d'héritage et à la condition supplémentaire qu'elles ne possèdent plus aucune instance, au contraire, dans ORION et GemStone, toutes les modifications sont envisagées. La sémantique

[6] La phrase doit être mise au pluriel en cas de multi-instanciation sur les classes virtuelles.

de ces opérations est décrite dans [Banerjee et al. 87a] [Banerjee et al. 87b] [Penney & Stein 87]. Elles peuvent poser des problèmes de cohérence importants, et dans tous les cas un gros travail de restructuration des données.

création d'un arc d'héritage La création d'un nouvel arc d'héritage, entre une classe c_1 et sa nouvelle sous-classe c_2, entraîne la modification de la définition de la classe c_2 ainsi que de toutes ses classes descendantes. De plus, la modification de la définition d'une classe entraîne également celle de toutes ses instances. Par conséquent, ajouter un arc d'héritage entraîne un travail extrêmement lourd de restructuration de tout un sous-graphe. Celui-ci peut être ramené à la modification immédiate des classes et à l'installation de filtres qui ne transformeront les instances que lorsqu'elles seront utilisées [Banerjee et al. 87b].

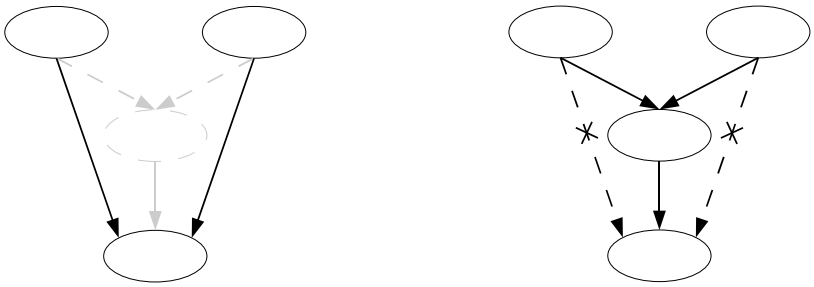
destruction d'un arc d'héritage La destruction d'un arc d'héritage entre deux classes peut avoir, quant à elle, des effets très importants sur la cohérence des données, omis par [Banerjee et al. 87b]. Illustrons ce propos sur un exemple. Soit une variable déclarée sur un domaine de classe racine c_1 : à tout instant, cette variable référence une instance d'une classe descendante de c_1 ; supposons qu'elle contienne une instance propre de la classe c_2, sous-classe directe de c_1. La suppression de l'arc d'héritage (c_1, c_2) déconnecte ces deux classes, et la variable référence alors une instance devenue illicite. Ceci met en évidence que les invariants sur le schéma de la base de données ne sont pas suffisants ; il faut incorporer des invariants sur les instanciations du schéma.

Il est dans ce cas de la responsabilité du programmeur de la transaction d'effectuer le travail de suppression des instances illégales dans toutes les variables qui ont été déclarées avec une classe anciennement ancêtre mais qui ne l'est plus. Ce travail est évidemment sujet à erreurs !

création d'une classe La création d'une classe peut se faire soit en puits du graphe d'héritage, soit plus généralement entre super-classes et sous-classes déjà existantes. La première façon ne pose aucun problème, alors que la seconde conjugue, avec la création de la classe proprement dite, la création et la suppression d'arcs d'héritage.

Il n'y a que si l'insertion préserve ou étend les domaines, c'est-à-dire si seuls les arcs de transitivité sont supprimés (cf. figure III.9), que la cohérence des instances de la base est directement préservée. (En fait, la cohérence est trivialement vérifiée pour les insertion en puits du graphe d'héritage.)

destruction d'une classe Enfin, la destruction d'une classe pose des problèmes de références perdues. En effet, si une variable était définie sur le domaine c_1 et que la classe c_1 est supprimée, alors toutes les variables d'instances qui avaient c_1 pour domaine doivent être redéfinies (en choisissant une super-classe de c_1, de préférence). Comme précédemment, pour la création



(a) avant (b) après

FIG. III.9 – Insertion non problématique d'une classe dans un graphe d'héritage avec seule suppression des arcs de transitivité

d'une classe, il est préférable que chaque sous-classe de c_1 hérite maintenant directement de chacune de ses super-classes.

Trois points ressortent de cette étude rapide :

(i) Les modifications du graphe d'héritage entraînent des modifications importantes dans les définitions des classes et donc dans la représentation des instances.

(ii) Les modifications ne peuvent pas toujours être automatiquement gérées par le système [Lerner & Habermann 90][Nguyen & Rieu 89]. En particulier, de nombreuses méthodes peuvent être invalidées par des suppressions de classes ; de nombreuses variables par celles d'arcs d'héritage [Zicari 89] [Zicari 91].

(iii) Ce sont les suppressions qui posent vraiment problème, la seule exception étant la suppression d'arcs de transitivité (encore que l'ordre d'héritage des super-classes, appelé multiplicité, puisse poser des problèmes de résolution des conflits).

Par conséquent, la modification d'un graphe d'héritage est un travail lourd qui devrait être réservé à la phase de réalisation d'une application. Effectivement, il s'avère que dans O_2, on distingue un mode développement, où la concurrence n'est pas admise, et un mode application, n'autorisant pas les modifications du graphe d'héritage et des méthodes.

III.4.2.2 Un nouveau protocole

Si l'on persiste à vouloir offrir des modifications concurrentes, alors il va falloir étendre le protocole.

Les modifications du graphe d'héritage entraînent celles des classes frontière de plusieurs sous-graphes ; il devient alors très difficile de vouloir maintenir dans chaque classe l'ensemble des classes frontière de son sous-graphe associé, calculées au préalable. Il est plus simple de parcourir à chaque fois le graphe des descendants avec l'algorithme donné en annexe C. De même, le chemin d'accès à un sous-graphe peut être modifié. L'algorithme le plus simple consiste à effectuer une remontée à partir de la classe racine du sous-graphe (de la classe isolée) en choi-

sissant n'importe quelle super-classe ; le chemin remontera irrémédiablement vers « OBJECT », racine du graphe d'héritage. Nous faisons donc l'hypothèse que chaque classe possède la liste de ses super-classes et celle de ses sous-classes. Notez qu'il s'agit de méta-données comme nous le signalions au dernier paragraphe de la section III.2.2.

Nous introduisons alors deux nouveaux modes, $L\Gamma$ et $M\Gamma$, qui vont nous permettre respectivement de lire et de modifier les listes des super-classes et de sous-classes. $L\Gamma$ et $M\Gamma$ ont respectivement la même relation de compatibilité que *LD* et *MD* mais sont d'un usage distinct.

Lors du calcul des classes frontière d'un sous-graphe, toutes les classes de ce dernier seront verrouillées avec le verrou $L\Gamma$, lecture de la relation d'héritage locale, ce qui permet d'assurer que la relation d'héritage dans les classes descendantes n'est pas en cours de modification. De plus, la détermination d'un chemin d'accès au sous-graphe ou à la classe étant devenue dynamique, là aussi il faudra poser le verrou $L\Gamma$ sur chaque classe rencontrée afin d'être certain d'emprunter un arc valide. Ces deux recherches et la pose des verrous $L\Gamma$ assurent la transaction que le sous-graphe sélectionné ne peut plus être modifié.

Quant au verrou $M\Gamma$, il ne sera posé que sur les classes origine et extrémité d'un arc d'héritage ôté ou ajouté.

Notez que nous n'utilisons que les verrous de classe ($L\Gamma$ et $M\Gamma$) et jamais ceux de frontière ($\forall L\Gamma$ et $\forall M\Gamma$) ou de chemin ($\downarrow L\Gamma$ et $\downarrow M\Gamma$).

Le protocole devient :

1. Verrouiller un sous-graphe (une classe isolée) dans un mode quelconque va consister, dans une première étape, à parcourir l'ensemble de ses classes pour déterminer dynamiquement les classes frontière (cf. algorithme en annexe C) en posant sur chaque classe (sur la classe isolée) un verrou $L\Gamma$ qui assurera que le graphe n'est pas, localement, en cours de modification. La seconde étape va consister à verrouiller le chemin d'accès au sous-graphe en effectuant une remontée à partir de la classe racine du sous-graphe (de la classe isolée), en posant chaque fois un verrou $L\Gamma$ et en choisissant un prédécesseur quelconque dans la liste des super-classes. Une fois ces deux premières étapes réussies, il suffit de convertir les verrous $L\Gamma$ sur le chemin d'accès en verrous de chemin dans le mode désiré, de convertir les verrous $L\Gamma$ sur les classes frontière du sous-graphe (la classe isolée) en verrou de frontière (de classe) dans le même mode, enfin de libérer les verrous $L\Gamma$ sur les autres classes descendantes.
2. Toute classe dont on veut modifier la liste de ses super-classes ou celle de ses sous-classes (ajout ou suppression d'arc d'héritage) doit être verrouillée avec le verrou $M\Gamma$, mais sans poser de verrous de chemin sur un chemin menant à cette classe.
3. Toute classe dont on veut modifier la liste de ses super-classes voit sa définition modifiée, d'où celles de ses classes descendantes et finalement de ses instances générales. Il

convient donc de verrouiller le sous-graphe issu de cette classe en mode *MD*, c'est-à-dire poser des verrous $\forall$*MD* sur les classes frontière et des verrous $\downarrow$ *MD* sur un chemin vers la racine du graphe d'héritage en utilisant le nouveau protocole exposé en 1 ci-dessus. (On peut alors supprimer le verrou $M\Gamma$ sur la classe racine du sous-graphe.)

On peut remarquer que les verrous $L\Gamma$ et $M\Gamma$ obéissent à un protocole biaisé [Korth 82] puisqu'ils ne sont pas utilisés comme les autres. De plus, ils ne respectent pas le verrouillage à deux phases strict. En fait, leur but est d'assurer l'atomicité du verrouillage d'un sous-graphe (d'une classe isolée), face uniquement aux modifications concurrentes du graphe d'héritage.

Pour fixer les idées, reprenons les quatre opérations de modification du graphe d'héritage. Nous exposerons également un exemple assez complexe.

création d'un arc d'héritage Un nouvel arc d'héritage est créé entre c_1, la super-classe, et c_2, sa nouvelle sous-classe. Les classes c_1 et c_2 sont tout d'abord verrouillées avec le verrou $M\Gamma$. Ensuite, le sous-graphe de classe racine c_2 est verrouillé en mode *MD* (en observant le nouveau protocole).

destruction d'un arc d'héritage On procède de la même façon que pour la création d'un arc. Cependant, c_1 peut être la seule super-classe de c_2. Il convient alors de rattacher c_2 à au moins une autre classe afin de conserver le graphe d'héritage connexe. On combine alors en une seule opération destructions et créations d'arcs.

création d'une classe On procèdera comme pour la création d'un arc d'héritage à ceci près que la classe racine du sous-graphe qu'il faut verrouiller en mode *MD* est la classe insérée. En revanche, il n'est pas nécessaire de la verrouiller avec $M\Gamma$. En effet, la classe n'existant pas auparavant, tous ses arcs entrant ou sortant sont de nouveaux arcs, c'est-à-dire qu'en respectant le point 1 du nouveau protocole, il est impossible pour une autre transaction d'aboutir à cette nouvelle classe sans avoir été au préalable bloquée sur une de ses super-classes ou de ses sous-classes.

Il nous faut ici tenir compte d'un élément que nous avons ignoré jusqu'à présent. Il existe un répertoire global qui contient la liste des classes du schéma, en particulier leurs noms externes. Il s'agit de contrôler dans ce répertoire que la classe créée ne possède pas un homonyme. Ceci relève de l'ADT ENSEMBLE et des propriété de commutativité de l'opération *Insérer*.

destruction d'une classe Nous faisons l'hypothèse, généralement vérifiée, que la classe racine du graphe d'héritage ne peut être détruite. (Ceci peut être imposé par l'utilisation d'une classe racine réservée au système, rendue invisible aux utilisateurs mais intervenant dans le protocole.) Lors de la destruction d'une classe, il se pose le même problème de connexité que lors de la suppression d'un arc d'héritage.

Du point de vue du contrôle de concurrence, chaque super-classe et chaque sous-classe doit être verrouillée avec $M\Gamma$. Puis, chaque sous-graphe associé à chaque sous-classe de la classe détruite doit être verrouillée en mode *MD*.

Ici encore il nous faut tenir compte du répertoire global et utiliser l'opération *Supprimer* sur celui-ci. On détectera ainsi les incompatibilités avec des tentatives d'accès à la classe supprimée. Cela veut surtout dire que les transactions devront se soumettre à exécuter l'opération Existe dans le répertoire des classes avant tout accès soit à celle-ci, soit à son sous-graphe (c'est-à-dire dans la procédure *VerrouillerClasses* de l'algorithme III.1)

Exemple III.8 *La figure III.10 présente quatre transactions concurrentes. Une première transaction modificatrice du graphe d'héritage, nommons-là T_1, veut transformer l'arc d'héritage (c_5, c_6) en (c_5, c_3). La classe c_6 se trouvant déconnectée, seul l'arc d'héritage (c_4, c_6) est alors introduit (le domaine de c_7 est donc restreint). Simultanément, une transaction T_2 voudrait accéder en lecture au sous-graphe de racine c_1, tandis qu'une transaction T_3 voudrait écrire les instances générales de c_7.*

T_1 verrouille alors (en caractères gras) c_3, c_4, c_5 et c_6 en mode $M\Gamma$, puis les sous-graphes de racine c_3 et c_6 en mode MD, enfin les verrous $M\Gamma$ sur c_3 et c_4 sont relâchés. Pour plus de clarté sur la figure, nous omettrons les verrous de chemin $\downarrow$ MD de la transaction T_1 sur les classes c_1, c_2, c_4, c_5 et la racine du graphe d'héritage.

Pour se prémunir contre l'oubli de verrouiller de nouvelles classes frontière (c_3 sera une nouvelle classe frontière du sous-graphe associé à c_1 si T_1 valide) ou le verrouillage inutile d'anciennes classes frontière, voire de classes qui n'appartiendraient plus au sous-graphe (c'est le cas de la classe c_6 qui ne sera plus descendante de c_7), les transactions T_2 et T_3 doivent effectuer un parcours de leur sous-graphe en verrouillant toutes les classes rencontrées avec le verrou $L\Gamma$. Ainsi, une incompatibilité est détectée par T_2 au niveau de la classe c_3 ; en effet, c_3 qui n'était pas jusqu'alors une classe frontière est en passe de le devenir. Similairement, T_3 est bloquée lors du verrouillage de la classe c_5 car la liste de ses sous-classes est en cours de modification ; par conséquent, T_3 n'est pas encore en mesure de connaître l'ensemble des classes descendantes de c_7.

La figure III.10 se complique encore par la présence d'une transaction T_4 qui voudrait créer une nouvelle classe, c_10, entre c_8 et c_9. Elle pose donc un verrou $M\Gamma$ sur c_8 et c_9, puis commence le verrouillage du sous-graphe issu de c_10 en mode MD en posant des verrous $L\Gamma$ comme l'exige le point 1 du nouveau protocole. T_4 sera finalement bloquée sur c_5 pour la même raison que T_3.

On remarquera que le chemin d'accès emprunté n'est pas indifférent : pour T_1, passer par c_4 n'est pas bloquant, alors que choisir c_7 mènera à un interblocage avec T_4.

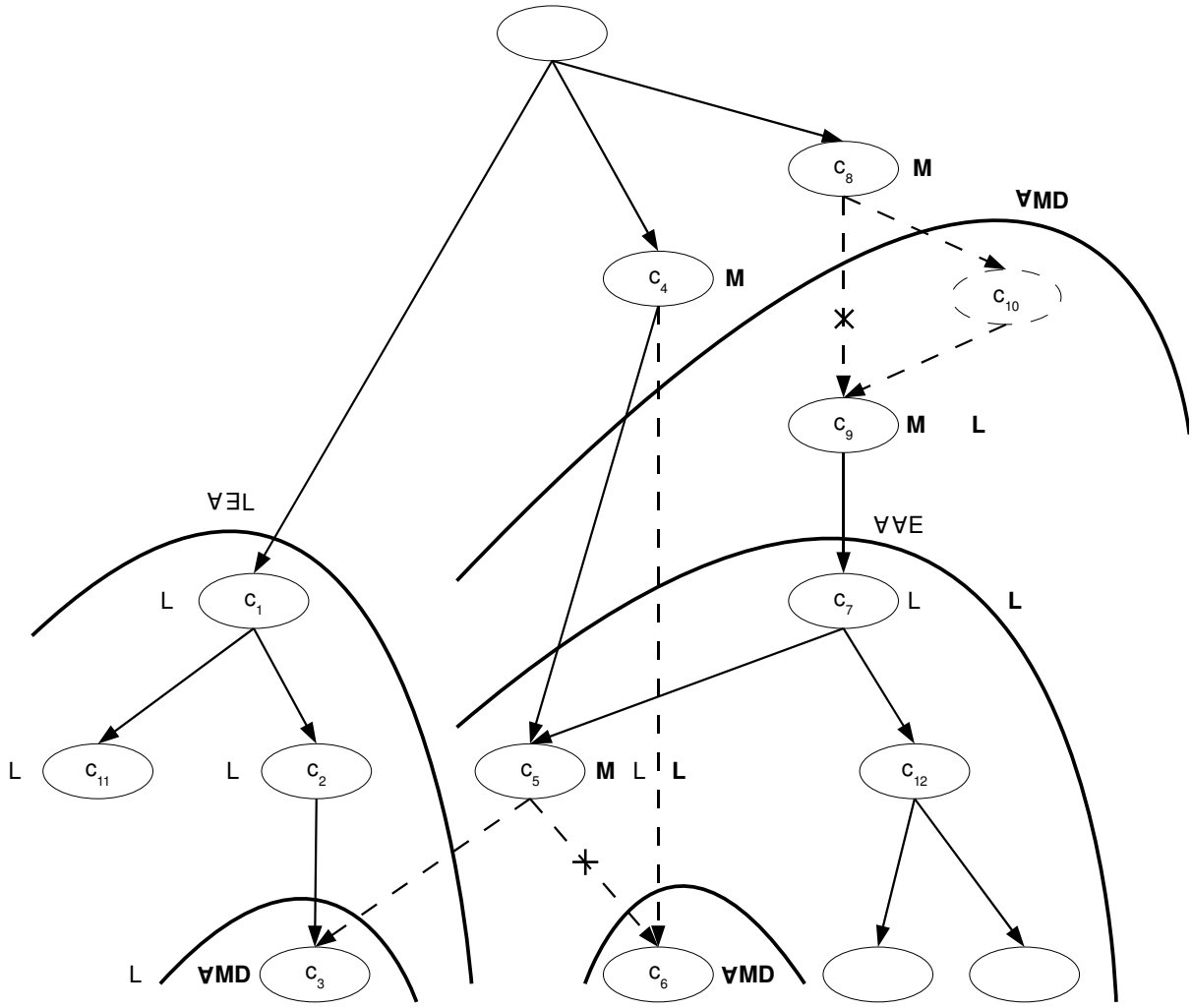


FIG. III.10 – Accès et modifications concurrents du graphe d'héritage

III.4.2.3 Critique de la solution

Nous avons proposé une solution au problème des modifications concurrentes dans un graphe d'héritage. Nous pouvons toutefois faire certaines réserves :

(i) Cette solution met en évidence le contrôle nécessaire sur les méta-données : ici la relation d'héritage représentée par les listes de sous-classes et super-classes en chaque classe. Ceci oblige à modifier sensiblement le protocole, notamment à fournir un algorithme pour déterminer à l'exécution les classes frontière d'un sous-graphe (cf. annexe C) et celles de chemin.

(ii) On se rend compte avec les cas de création ou destruction de classes que le verrouillage dans le seul graphe d'héritage ne suffit pas ; il faut procéder à un verrouillage dans le répertoire des classes. Toutefois, celui-ci est géré comme un ADT ENSEMBLE, donc avec des propriétés de commutativité importantes (cf. chapitre II). Néanmoins, ce contrôle supplémentaire constitue une surcharge.

(iii) L'intérêt principal de la première partie est perdu, à savoir que le verrouillage des classes frontière n'est plus suffisant puisque, en définitive, il faut maintenant verrouiller toutes les classes d'un sous-graphe, même si ce n'est qu'avec des verrous temporaires.[7]

(iv) Le problème essentiel, lié au parallélisme obtenu, est que la modification de la définition d'une classe bloque toute transaction qui voudrait accéder soit à une ou plusieurs classes incluses dans le sous-graphe en cours de redéfinition, soit à un sous-graphe qui inclurait ou aurait une intersection stricte avec celui qui est en cours de modification. Cela représente sinon la majorité, du moins un très grand nombre d'accès possibles. Par exemple, en figure III.8, supprimer l'arc d'héritage (c_2, c_3) interdit l'accès concurrent à tout sous-graphe du graphe d'héritage ainsi qu'aux classes c_3 et c_4. De même, on pourra vérifier sur la figure III.10 que l'on ne peut plus accéder qu'à un seul sous-graphe : celui de racine $c_1 1$! En effet, Les accès à $c_1 2$ seront bloqués lors du verrouillage du chemin d'accès.

En conséquence, si cette solution respecte la sérialisabilité, et en l'absence de critères sur la forme typique d'un graphe d'héritage, elle n'est pas satisfaisante du point de vue du parallélisme obtenu, notamment dans le cas de bases de données qui doivent être maintenues accessibles de manière quasi permanente.[8]

Il reste encore une forme de modification importante : celle des méthodes. Nous ne ferons que l'aborder dans ce chapitre.

[7] Comme nous l'avons dit, les verrous temporaires ne servent qu'à assurer l'atomicité du verrouillage d'un sous-graphe. Une solution plus brutale serait d'effectuer ce verrouillage en exclusion mutuelle.

[8] Par exemple, si le graphe d'héritage est peu profond et très large, alors les sous-graphes seront petits par rapport au graphe d'héritage et dans leur très grande majorité deux à deux disjoints. Dans ce cas, le parallélisme reste satisfaisant.

III.4.3 Les lectures et modifications de méthodes

Un appel de méthode correspond à un accès à une instance d'une classe C et à des accès virtuels aux classes ancêtres de C afin de résoudre dynamiquement la référence au code de la méthode à appliquer.

La solution proposée pour détecter les conflits est de ne pas verrouiller les classes lues lors des accès virtuels. En contrepartie, la modification ou la destruction d'une méthode dans une classe C nécessite le verrouillage du sous-graphe de racine C et pas seulement de la classe, avec un mode *MM* pour « modification de méthode ».

Le verrouillage du sous-graphe de racine C permet de détecter une incompatibilité avec tous les accès utilisant des instances générales de C, c'est-à-dire des instances susceptibles d'être appelées avec une méthode définie dans C. Cette précaution est nécessaire pour éviter qu'une transaction qui applique une méthode sur certaines instances générales de C, n'applique pour les premières instances la méthode avant modification et pour les suivantes la méthode après modification, et a fortiori pas du tout si elle est supprimée.

Les transactions qui modifient ou suppriment une méthode dans une classe C, sont pénalisées par des attentes dans le cas où des instances générales de C sont utilisées, même par d'autres méthodes. Cette limitation sera levée dans le chapitre suivant.

III.5 COMPARAISONS

Au moment où ce travail a été réalisé, les protocoles de contrôle de concurrence adaptés aux bases de données à objets n'étaient pas légion. G-BASE et V-Base étaient encore mono-utilisateur ; IRIS était implémenté au-dessus d'une base de données relationnelle et utilisait donc son contrôle de concurrence ; GemStone proposait un contrôle de concurrence optimiste « classique » sur les segments (pages consécutives) du disque ; O_2 se contentait d'utiliser le protocole hiérarchique à deux phases strict implémenté dans le système de gestion de fichiers sous-jacent : WiSS (« *Wisconsin Storage System* »), mais une étude était réalisée [Cart & Ferrié 89b] [Cart & Ferrié 90] pour trouver une méthode plus appropriée. En revanche, ORION proposait déjà plusieurs solutions : deux utilisant le graphe d'héritage, une pour gérer les objets composites.

Dans la section suivante, nous allons comparer notre protocole à celui d'ORION qui lui est le plus proche et montrer qu'il le généralise. Ensuite, nous effectuerons une comparaison plus sommaire avec la proposition de [Cart & Ferrié 90].

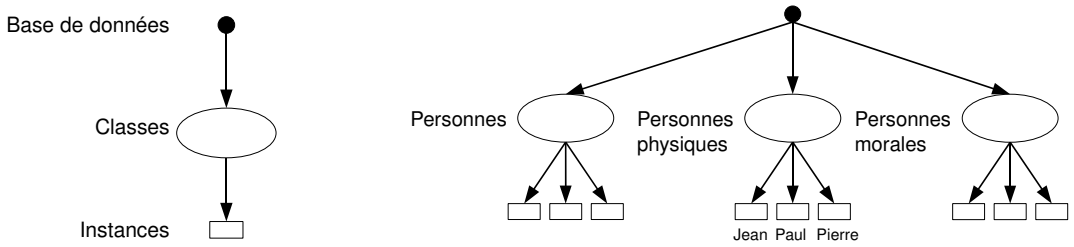
III.5.1 Comparaison avec le protocole d'ORION

Nous n'avons pas besoin de présenter le protocole d'ORION puisque notre proposition s'avère le généraliser. Les différences portent sur :

- la notion de classes frontière ;
- le choix d'un chemin jusqu'à un sous-graphe ;
- l'utilisation du seul graphe d'héritage ;
- le nombre des modes d'accès.

classes frontière La notion de classe frontière n'est pas présente dans le protocole d'ORION. Ce dernier utilise celle moins précise de classe ayant plus d'une super-classe. Par conséquent, le nombre de classes à verrouiller est généralement plus élevé.

Le verrouillage d'un sous-graphe n'est pas disponible pour tout mode d'accès ; seuls les modes *R* et *W*, correspondant respectivement à $\forall\forall L$ et $\forall\forall E$, ont ce privilège. Les modes $\forall\exists L$ et $\forall\exists L$ n'existent pas ; si l'on veut verrouiller un sous-graphe avec ces modes d'accès intentionnels, il faut alors verrouiller toutes les classes du sous-graphe.



(a) graphe des granules (b) une instanciation

FIG. III.11 – Graphe des granules générique et une instanciation possible

ORION	*IS*	*IX*	*S*	*X*	*SIX*	*R*	*W*	*IR*	*IW*		
	$\exists L$	$\exists E$	L	E		$\forall\forall L$	$\forall\forall E$	$\downarrow \exists L$	$\downarrow \exists E$	$\forall\exists L$	$\forall\exists E$
			$\forall L$	$\forall E$		$\forall LD$	$\forall MD$	$\downarrow \forall L$	$\downarrow \forall E$		
			LD					$\downarrow LD$	$\downarrow MD$		

TAB. III.5 – Correspondance entre les verrous d'ORION et les nôtres

chemin de sous-graphe Dans le protocole d'ORION, les classes de chemin doivent se trouver sur un chemin entre la racine du graphe d'héritage et celle du sous-graphe. Nous avons vu que la classe où aboutit le chemin peut être quelconque et ne se limite pas à la seule classe racine du sous-graphe.

graphe d'héritage Le protocole d'ORION se présente lui-même comme une extension du verrouillage hiérarchique de [Gray 78]. Nous ne partageons pas ce point de vue.

ORION exploite deux graphes : le graphe d'héritage et le graphe des granules. Ce dernier est extrêmement simple : il consiste en un nœud représentant la base de données, elle-même composée de classes, aboutissant finalement aux instances. Le graphe des granules générique et une instanciation sont donnés en figure III.11.

Dans le graphe des granules, toutes les classes se trouvent au même niveau, ce qui rend l'utilisation du graphe d'héritage indispensable pour certaines formes d'accès.

Toutefois, nous avons montré, par son absence, que l'exploitation du graphe des granules est inutile. En effet, le verrouillage hiérarchique entre les niveaux classes et instances est obtenu par la troisième composante d'un verrou de classe (cf. section III.3.3). Enfin, l'accès exclusif à toute la base de données, qui serait obtenu dans ORION en posant un verrou exclusif sur le nœud racine du graphe des granules, est obtenu avec notre protocole en posant le verrou $\forall MD$ sur la classe racine du graphe d'héritage. Il nous faut faire l'hypothèse supplémentaire que cette classe-là n'est pas modifiable. C'est généralement le cas.

modes d'accès ORION ne dispose que d'un jeu limité et *ad hoc* de verrous, neuf très exactement. La table III.5 donne la correspondance entre les verrous d'ORION et les nôtres.

On se rend compte que certains verrous jouent plusieurs rôles, suivant qu'ils sont posés sur

les instances ou sur les classes, mais aussi pour des modes d'accès qui devraient, sémantiquement, être différenciés.

Comme nous l'avons déjà dit, les verrous $\forall\exists L$ et $\forall\exists E$ n'existent pas. En revanche, ORION offre le verrou *SIX* sur les classes qui est la combinaison des deux verrous *S* et *IX*, c'est-à-dire accès en *lecture* à *toutes* les instances de la classe, doublé d'un accès en *écriture* à *certaines* instances, qui devront alors être verrouillées explicitement avec le verrou *X*. Ce mode d'accès combine deux niveaux : hiérarchique et implicite au niveau de la classe, explicite au niveau des instances.

Notre protocole n'autorise qu'un unique mode d'accès par verrou. Cependant, nous sommes en mesure d'introduire un mode d'accès approchant de *SIX*. Pour cela, ajoutons le mode d'accès *Esc* (pour escalade) qui correspond à un accès en lecture à une instance pouvant ultérieurement, et *si nécessaire*, se transformer en accès en écriture.[9] Ce mode d'accès est rendu compatible avec *L* et *LD*. Le mode *Esc* s'appliquant aux instances, on obtient le verrou *Esc* portant sur les instances, et les verrous $\forall Esc$ (analogue à *SIX*), $\exists Esc$ (un *IX* affaibli), $\forall\forall Esc$, $\forall\exists Esc$, $\downarrow \forall Esc$ et $\downarrow \exists Esc$ sur les classes. Le verrou $\forall Esc$ n'est pas exactement le verrou *SIX*. Comme *SIX*, toutes les instances (propres de la classe) sont utilisées implicitement en lecture, et certaines pourront être accédées en écriture, si l'on requiert explicitement le verrou *E* sur ces dernières. Contrairement à *SIX*, $\forall Esc$ est compatible avec $\forall L$ (équivalent de *S*) et il faudra effectuer une escalade sur la classe, avec le verrou $\exists E$, en plus de la pose des verrous *E* sur les instances.

La figure III.12 représente le graphe de Hasse des privilèges associés à nos verrous, construit à partir d'une table de compatibilité symétrique (le mode *Esc* pouvant être utilisé asymétriquement). Il s'avère que les modes *MM* et *CDI* ont exactement la même relation de compatibilité que *E* (mais cela peut changer si l'on introduit de nouveaux modes d'accès).

La figure III.13 permet de faire la comparaison avec ceux d'ORION, en n'oubliant pas que *SIX* et $\forall Esc$ ne sont pas strictement identiques. L'inclusion d'ORION est stricte si l'on fait abstraction de ce mode d'accès.

Toutefois, le lecteur ne doit pas se laisser abuser par les figures III.12 et III.13. Si les verrous de notre proposition sont plus fins, ils ne le sont pas assez pour permettre à un autre verrou d'exploiter un parallélisme potentiel !

La seule limitation du protocole d'ORION, vis-à-vis des modes d'accès envisagés, est que *une transaction qui lit la définition d'une classe empêche une autre transaction de modifier, en totalité ou en partie, les instances de cette classe* (cf. exemple III.6).

La table III.2, rendant *LD* et *E* compatibles, évite cette limitation. Pour pouvoir vraiment profiter d'un parallélisme accru, il faudra encore augmenter le nombre de modes d'accès. Dans

[9] Il a été initialement proposé dans le cadre des recherches sur *System R*, des mesures ayant montré que 97 % des interblocages étaient causés par les escalades de verrou du mode *L* au mode *E*. [Korth 83] généralise les escalades à n'importe quel couple de modes.

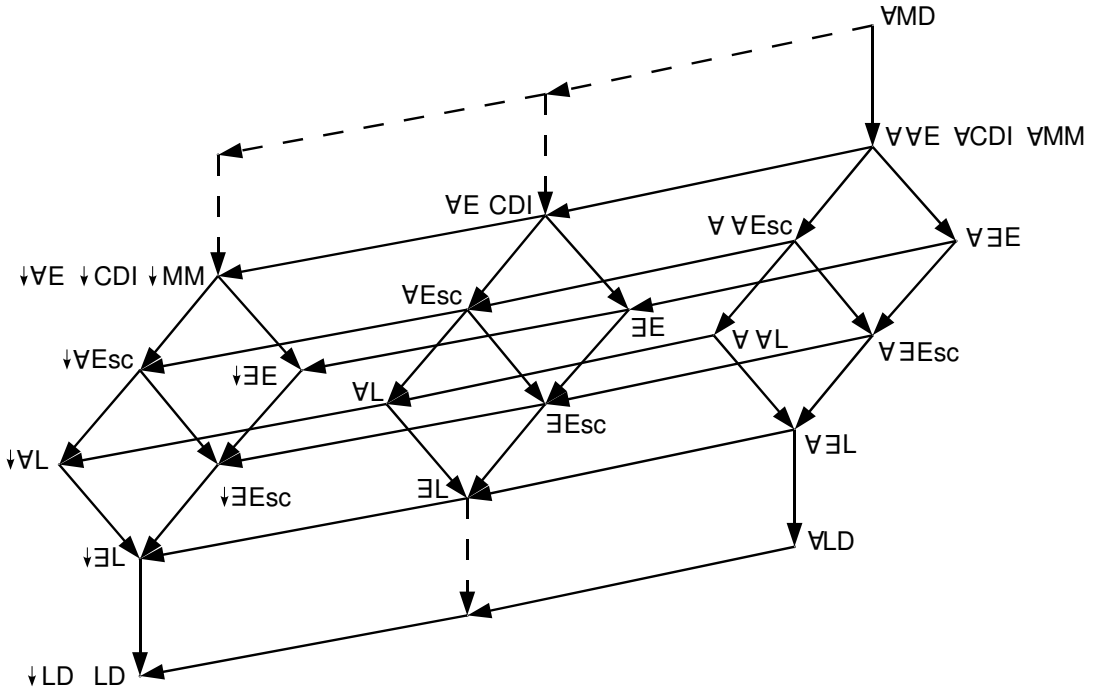


FIG. III.12 – Graphe des privilèges associés à nos verrous

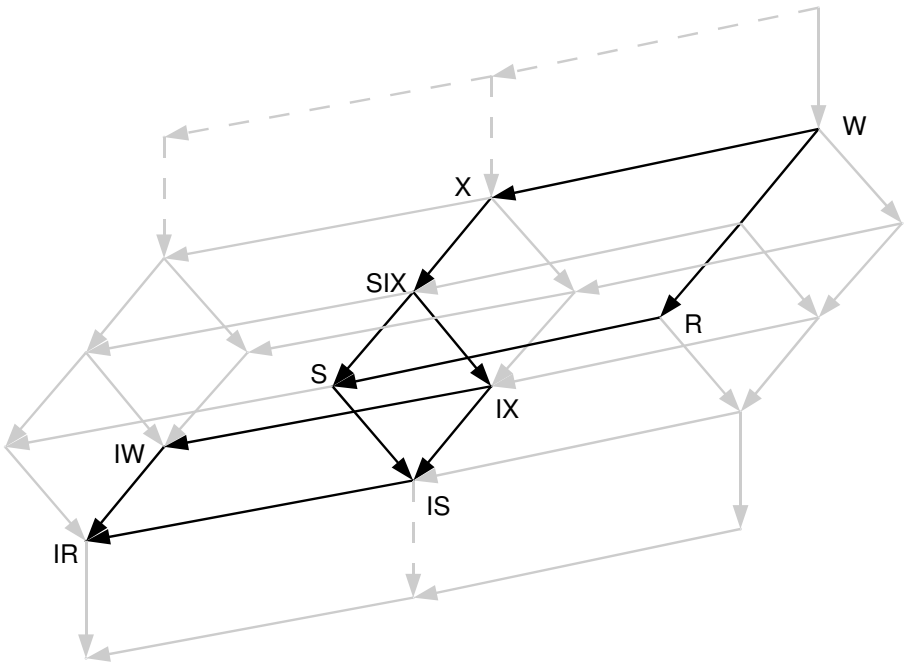


FIG. III.13 – Graphe des privilèges associés aux verrous d'ORION

ORION, les classes peuvent comporter des variables de classe, c'est-à-dire des variables partagées par toutes les instances de la classe ; on peut envisager de créer les modes de lecture et d'écriture de ces variables (LVC et EVC). La plus grande généralité de nos verrous le permet. Toutefois, avec l'ajout d'un trop grand nombre de modes, et comme le note fort justement [Korth 83] :

> « *The transaction writer (i.e., the user) cannot be expected to make optimal use of all these lock modes. Ideally, a transaction compiler would determine which modes of locks should be requested.*[10] »

Dans le chapitre suivant, nous réalisons cet idéal, dans une certaine mesure, en spécialisant les modes portant sur les instances à un mode par méthode.

Comme nous l'annoncions au début de cette section, le protocole d'ORION se révèle donc être un strict sous-ensemble de notre proposition, exception faite du verrou *SIX* : 8 (9) verrous d'ORION correspondent à 4 ($\approx$ 5) modes d'accès et se traduisent par 19 (26) verrous dans notre proposition.

III.5.2 Comparaison avec la proposition pour O_2

La proposition de [Cart & Ferrié 89b] [Cart & Ferrié 90] est beaucoup plus pratique que la nôtre et aborde de nombreux aspects relatifs au problème de la gestion des accès concurrents. Tout d'abord, les instances, que nous nous bornons à considérer comme des objets logiques formant un tout, sont décomposées en termes de représentations physiques. Ensuite, le protocole multi-niveaux, que nous ne faisons que citer, y est clairement décrit. Enfin, l'hypothèse d'*indépendance* entre objets multi-niveaux (cf. exemple II.20), pour une exploitation correcte de la commutativité entre méthodes, est exposé. Les autres aspects peuvent être rapidement comparés.

Les modes d'accès envisagés sont décomposés en lectures et écritures sur les instances et les classes, ce qui correspond à nos modes *L*, *E*, *LD* et *MD*. *L* et *E* sont ensuite étendus aux modes d'accès hiérarchiques de [Gray 78], en s'appuyant sur le graphe des granules déjà présenté pour ORION : *R*, *W*, *IR*, *IW*, *RIW*. Les modes d'accès présentés dans [Cart & Ferrié 90] sont simultanément des verrous. La différenciation entre verrous de classes et verrous d'instances permet toutefois d'éviter le problème du protocole d'ORION qui ne permet pas la modification d'instances en concurrence avec la lecture des définitions de leurs classes (cf. exemple III.6). Le parallélisme offert est donc exactement celui qu'il est possible d'obtenir avec les accès envisagés.

[10] « On ne peut pas s'attendre à ce que le rédacteur d'une transaction (c'est-à-dire l'utilisateur) fasse un usage optimal de tous ces modes de verrouillage. Idéalement, un compilateur de transactions devrait déterminer les verrous à requérir. »

La notion d'accès virtuels aux sous-classes (notion de sous-graphe) et aux super-classes d'un classe donnée est particulièrement bien mise en valeur dans [Cart & Ferrié 90]. Le choix adopté dans le protocole est de verrouiller explicitement *toutes* les classes d'un sous-graphe est aucune super-classe de la racine de ce dernier. Pour notre part, nous ne verrouillons que les classes frontière du sous-graphe, mais, en compensation, nous devons aussi verrouiller un chemin d'accès parmi les super-classes. (À noter que l'implémentation actuelle dans le prototype ORION utilise ce protocole [Kim et al. 90] et non celui décrit dans la section précédente. Il est dit que ce choix est arbitraire ! La facilité d'implémentation y est certainement pour quelque chose.)

Enfin, le problème des objets « fantômes », en ce qui concerne les classes, est traité succinctement. Nous avons vu en section III.4.2 qu'il s'agit d'un problème délicat, qui mériterait d'être étudié avec d'autres objectifs en vue, comme la coopération transactionnelle.

III.6 CONCLUSION

Après avoir présenté les caractéristiques minimales d'un système à objets, nous avons détaillé l'ensemble des accès envisageables aux données : accès à une ou plusieurs classes, accès simultanés ou indépendants à leurs instances.

Afin de traiter tous les cas envisagés, nous avons tout d'abord démontré deux propriétés suffisantes sur le graphe d'héritage qui permettent d'isoler les ensembles d'objets intéressants. Nous avons vu que ces ensembles d'objets sont soit une classe et ses instances propres, soit les classes descendantes d'une classe racine donnée et leurs instances. Ensuite, de manière plus classique, nous avons différencié plusieurs modes d'accès. Enfin, nous offrons la notion de verrouillage hiérarchique au niveau des instances, c'est-à-dire que le verrouillage de celles-ci peut être implicite. Ces trois points de vue nous ont amenés à une construction systématique des verrous qui sont des vecteurs composés d'un types d'accès, d'un modes d'accès et de sa nature hiérarchique.

Un premier protocole à base de verrouillage à deux phases strict est proposé, capable de répondre à toutes les formes d'accès présentées.

Une extension est ensuite apportée au protocole afin de prendre en compte les modifications concurrentes du graphe d'héritage. Le contrôle de concurrence y est considérablement alourdit, et, de plus, le respect du critère de sérialisabilité pénalise le parallélisme des transactions. Il est donc intéressant, lorsque cela est possible, de différencier deux modes de fonctionnement : un mode développement, où les modifications concurrentes sur le schéma de la base de données seraient autorisées, et un mode application, pour lequel on pourra utiliser la première version du protocole, plus efficace.

Nous comparons ensuite notre proposition à celle d'ORION et d'O_2. Vis-à-vis du protocole d'ORION, il s'agit quasiment d'une généralisation. Toutefois, le gain effectif de parallélisme est minime si l'on n'étend pas considérablement le nombre de modes d'accès. Par rapport à la proposition pour O_2, les protocoles s'avèrent incomparables. À l'actif de cette proposition, le parallélisme obtenu est très exactement celui que l'on peut obtenir en théorie.

À la fin de ce chapitre, nous avons encore des problèmes de parallélisme. Nous avons vu que l'introduction de classes dites virtuelles (cf. section III.4.1) augmentait le parallélisme dans le cas des créations et destructions d'instances. Mais, nous venons de répéter que les modifications concurrentes du graphe d'héritage sont extrêmement limitatives et devraient s'envisager dans un cadre coopératif qui constitue une voie de recherche. Nous allons voir dans le chapitre suivant d'autres limitations.

Chapitre IV

Automatisation d'un contrôle de concurrence applicatif

Résumé du chapitre

Les propositions qui n'exploitent que les modes d'accès *Lire* ou *Écrire* sur les instances conduisent à refuser des exécutions concurrentes permises dans les bases de données relationnelles ! Il faut donc absolument affiner la vue que l'on a des méthodes. Les bases de données à objets offrent l'opportunité d'accroître le parallélisme des transactions si l'on exploite la commutativité potentielle entre méthodes. Toutefois, il n'est pas possible de se contenter de tables de commutativité *ad hoc*, fournies explicitement par un programmeur, car les méthodes sont appelées à être fréquemment modifiées, ajoutées ou supprimées.

En conséquence, nous proposons dans ce chapitre une technique d'analyse « simple », justifiée par les résultats théoriques du chapitre II, qui autorise un contrôle de concurrence au niveau des variables d'instance plutôt que de l'instance dans sa totalité. Cette technique permet (1) de limiter le nombre de contrôles, (2) d'éviter certains interblocages et surtout (3) de rendre compatibles des méthodes écrivains entre elles et avec des méthodes lectrices. Elles offre également d'autres avantages : (4) le processus de détermination de la commutativité des méthodes se fait entièrement durant la phase de compilation, sans surcharge sensible, (5) le contrôle à l'exécution est aussi efficace qu'avec la simple compatibilité, (6) les opérations inverses n'ont pas à être fournies, et (7) il reste possible d'intégrer des objets munis de relations de commutativité plus intéressantes, comme l'ADT COMPTEUR.

Mots clés

Bases de données à objets, héritage, surcharge, liaison dynamique, algorithme incrémental, efficacité, commutativité, contrôle de concurrence, verrouillage à deux phases strict.

IV.1 INTRODUCTION

Parmi les propositions visant à fournir un contrôle de concurrence adapté aux bases de données à objets, la plupart n'envisagent que les accès en lecture ou écriture, c'est-à-dire la simple compatibilité, aux instances : [Garza & Kim 88], [Cart & Ferrié 90] ainsi que le chapitre précédent. Nous verrons en section IV.6.3, la proposition de [Agrawal & El Abbadi 92] qui traite les accès concurrents au niveau des variables d'instances. Nous allons montrer que ne considérer que les modes d'accès *Lire* et *Écrire* n'est pas suffisant dans les bases de données à objets. Il faut absolument fournir une forme de commutativité entre méthodes, comme plusieurs auteurs l'ont fait avec les types de données abstraits (ADT) [Schwarz & Spector 84] [Weihl 88]. Toutefois, le chapitre II a mis en évidence les limites de la commutativité ; une forme simple de commutativité sera donc suffisante. Celle que nous retenons est quasiment identique à celle de [Noe et al. 87], à la différence qu'elle est définie statiquement et non dynamiquement. Malgré sa simplicité, cette forme de commutativité va nous permettre de résoudre quatre problèmes qui, à notre connaissance, n'ont pas été signalés dans la littérature.

Tout d'abord, il n'est pas raisonnable de laisser à la charge d'un programmeur d'application la détermination de la commutativité entre *tout couple* d'opérations, ainsi que l'écriture systématique d'une opération inverse pour *chaque* opération. Ensuite, la factorisation du code conduit à un grand nombre de messages envoyés à l'instance courante elle-même, ce qui engendre, d'une part, plusieurs contrôles pour l'envoi effectif d'un message, et, d'autre part, des risques d'interblocage dus à des escalades de verrous. Enfin, avec les seuls mode d'accès *Lire* et *Écrire*, certaines exécutions concurrentes seront refusées alors qu'elles sont acceptées dans les bases de données relationnelles.

Présentons rapidement l'organisation de ce chapitre. Premièrement, nous détaillerons quelques concepts des systèmes à objets, ces concepts-là étant communs à la majorité des bases de données actuellement implémentées. (Cette partie pourra être sautée par quiconque connaît les rudiments de la programmation à objets.) Ensuite, nous présenterons, à partir d'un exemple, les quatre problèmes principaux qui rendent l'utilisation des simples modes d'accès *Lire* et *Écrire* aux instances insatisfaisante. Puis, les vecteurs d'accès directs seront définis et un algorithme incrémental efficace sera proposé pour construire les vecteurs d'accès transitifs, solution des problèmes évoqués. En section IV.5, nous traiterons du verrouillage dans un graphe d'héritage grâce aux vecteurs d'accès transitifs. Nous dresserons alors la liste de tous les avantages que l'on est en droit d'attendre de ceux-ci. Une proposition d'implémentation ainsi qu'une comparaison avec des travaux antérieurs seront données avant la conclusion.

IV.2 UN MODÈLE À OBJETS

Afin d'être applicable à une majorité de bases de données à objets, nous avons isolé ce qui nous semble être le plus grand diviseur commun des nombreux modèles dont nous avons eu connaissance : *classes*, *mono-instanciation* et *héritage multiple*. Dans cette présentation rapide, nous insisterons notamment sur le mécanisme d'*envoi de messages* qui met en œuvre *héritage*, *surcharge* et *liaison dynamique*.

IV.2.1 Le modèle de données

Le modèle le plus couramment décrit est celui du paradigme des classes. Il distingue instances et classes, mais n'introduit pas les méta-classes, mécanisme d'une grande généralité mais s'accordant mal à la compilation. Les instances appartiennent de manière propre à une et une seule classe ; on parle alors de *mono-instanciation*. Les classes sont liées par la relation d'héritage, simple ou multiple. Ces concepts de bases sont ceux introduits par Smalltalk [Goldberg & Robson 83] et on peut les retrouver dans ORION [Banerjee et al. 87a], O_2 [Lécluse et al. 88], GemStone [Butterworth et al. 91], *ObjectStore* [Lamb et al. 91] ou V-Base [Andrews & Harris 87].

IRIS [Fishman et al. 87] est un exemple de base de données à objets ne respectant pas ces critères puisqu'une instance peut appartenir simultanément à plusieurs classes incomparables dans le graphe d'héritage, ce que l'on nomme *multi-instanciation* [Rieu et al. 91]. G-BASE également [Roffé et al. 90], intégrant le mécanisme des méta-classes, n'est pas directement prise en compte par notre proposition.

Une classe est composée d'un n-uplet de *variables d'instance*, que nous appelons plus brièvement *champs*, et d'un ensemble de méthodes, *unique moyen* de manipuler les instances. Nous différencions les champs dont le type est un type de base tel que « *integer* » ou « *string* », des champs qui constituent des références à d'autres instances, c'est-à-dire qui ont été déclarés avec comme nom de type un nom de classe. Certaines bases de données à objets, comme O_2, permettent de créer des champs de type complexe, c'est-à-dire pouvant être à leur tour n-uplet (*tuple of [...]*), famille (ou « multi-ensemble » : *set of ...* !), ensemble (*unique set of ...* !) ou liste (*list of ...*).

Exemple IV.1 *Dans l'algorithme IV.1, nous avons mis en évidence ces différentes formes de déclarations. Les champs déclarés au niveau racine de la structure d'un n-uplet sont soit des*

Algorithme IV.1 Déclarations de types de classes en O_2C

```
class Personne
type tuple of (Nom : string,
               Adresse : tuple of (Numéro : integer,
                                   Voie : string,
                                   Ville : string,
                                   CodePostal : string,
                                   BureauDistributeur : string))

class PersonnePhysique inherits Personne
type tuple of (Prénoms : list of string,
               EtatCivil : tuple of (Sexe : (Masculin, Féminin),
                                     SituationFamiliale : (Célibataire,
                                                           Marié,
                                                           Veuf,
                                                           Divorcé),
                                     Père : PersonnePhysique,
                                     Mère : PersonnePhysique,
                                     Enfants : list of PersonnePhysique),
               Profession : string,
               Employeur : Personne)

class PersonneMorale inherits Personne
...
```

types de base comme « Nom : string », soit des références à d'autres instances tel que « Employeur : Personne », soit des champs complexes de type n-uplet, comme « Adresse », ou liste, comme « Prénoms », ou encore liste de références comme « Enfants » (qui n'est toutefois pas un champ racine).

Remarquez que la structuration en objets complexes peut susciter l'envie de prendre en compte les constructeurs de types [Hermann et al. 90]. Nous ne considérerons ici que les champs racines de la structure n-uplet. Il y a deux raisons à cela.

Tout d'abord, que le champ soit ou ne soit pas complexe, le résultat théorique fondamental du chapitre II et la grande dépendance entre champs d'un n-uplet ne nous encouragent certainement pas à affiner exagérément notre technique de recherche de la commutativité entre méthodes. Par exemple, le champ « *Adresse* » de l'algorithme IV.1 ne doit pas être décomposé ; si le programmeur a groupé les sous-champs c'est parce qu'il les considérait comme formant un tout indivisible. Il est possible de déménager pour la maison d'en face, plus souvent dans la même ville, mais généralement on sera amené à modifier solidairement l'ensemble de l'adresse.

Deuxièmement, les champs ensemblistes peuvent être composés de références, comme le champ « *Enfants* » dans l'algorithme IV.1 ; dans ce cas, le contrôle de concurrence sera différé aux instances référencées si et lorsqu'un message leur sera envoyé. Le cas particulier des objets

composites, objets liés par la relation « *est-une-partie-de* », a fait l'objet d'études spécifiques dans ORION ([Kim et al. 87] pour les objets composites disjoints et [Kim et al. 89] pour une extension aux objets composites non disjoints).

Les champs et les méthodes sont hérités par les sous-classes. Une sous-classe peut définir des champs supplémentaires ainsi que de nouvelles méthodes. De plus, les méthodes héritées peuvent être redéfinies dans une sous-classe, c'est-à-dire qu'un code spécifique sera fourni pour la méthode de même nom. C'est cette faculté de surcharger des méthodes de même nom qui complique notre technique et nécessite une présentation plus détaillée.

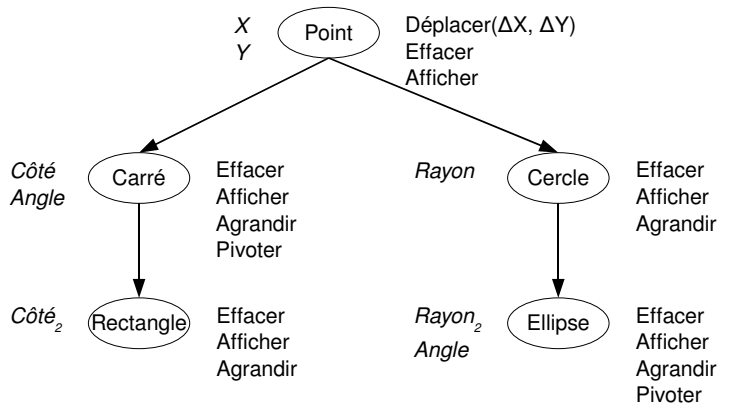
IV.2.2 Le mécanisme d'envoi de messages

Le paradigme de l'envoi de messages en programmation à objets consiste à envoyer des messages aux instances plutôt que de leur appliquer des procédures traditionnelles, et encore moins de manipuler directement leurs champs. Ce paradigme permet d'obtenir l'*encapsulation* des données dans la mesure où l'utilisateur d'une classe n'a plus à connaître la représentation interne d'une instance pour s'en servir. Mieux encore, sa structure peut être modifiée dans le temps sans que cela ait de répercussions sur les programmes qui l'importent, dans la mesure où les spécifications externes des méthodes ne sont pas changées. Toutefois, l'encapsulation est également obtenue par les types de données abstraits et n'est donc pas une conséquence du mécanisme d'envoi de message. Cependant, que les instances ne soient manipulées que par envoi de messages est primordial dans ce chapitre.

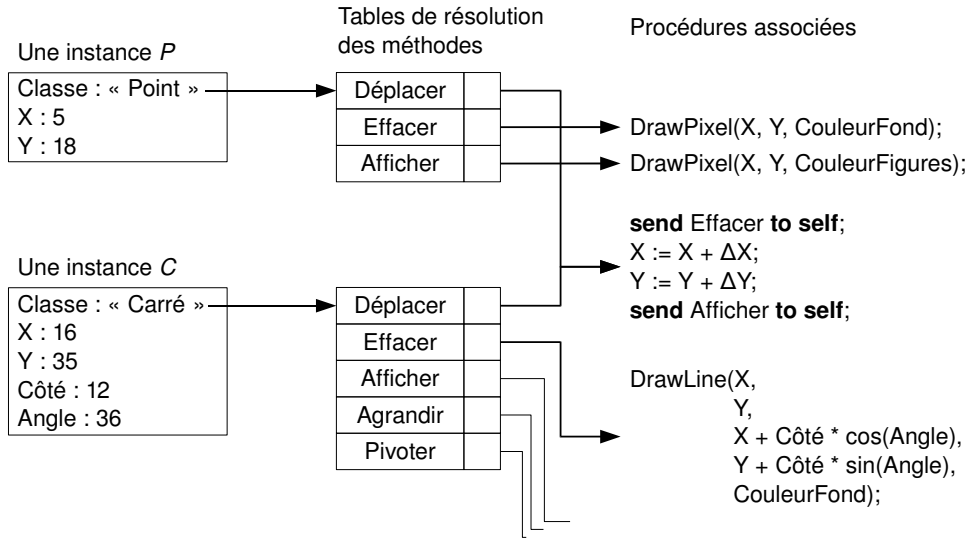
Un message est lié au moment de l'exécution à la méthode de même nom la plus spécifique. La méthode choisie dépendra de la classe à laquelle appartient l'instance. Une implémentation possible du mécanisme de *liaison dynamique* est illustrée en figure IV.1. Toute instance possède un champ « *Classe* »,[1] ajouté automatiquement par le système, qui désigne la classe propre de l'instance. Ce champ est utilisé pour retrouver la table de liaison dynamique ; il en existe une par classe. Cette table met en relation le nom du message avec le corps de la méthode de même nom définie dans la classe. Ainsi, le message *Effacer* n'appellera pas le même code suivant que l'instance à laquelle il a été envoyé est un point *stricto sensu* ou un carré, et cela même si le message *Effacer* est envoyé depuis le corps de la méthode *Déplacer*, commune aux deux classes.

Pour résumer, la différence, *essentielle*, entre « message » et « méthode » est que le premier se situe juste avant le second dans la chaîne des accès [Krakowiak 85], le mécanisme de liaison pouvant être celui que nous avons esquissé. Pour les différencier, les messages porteront un nom simple alors que les méthodes seront préfixées par un nom de classe, le message

[1] Dans Smalltalk et ORION, le « nom » de la classe n'est pas un champ de l'instance mais fait parti de son OID. Ce choix présente l'inconvénient de ne plus permettre, facilement, de changer la classe propre de l'instance.



(a) quelques classes « géométriques » (attributs à gauche, méthodes à droite)



(b) tables de résolution des liaisons dynamiques

FIG. IV.1 – Exemples de tables de résolution de la liaison dynamique

Afficher devenant, suivant l'instance à laquelle il est envoyé, la méthode POINT.*Afficher*, ou CARRÉ.*Afficher*, ou encore CERCLE.*Afficher*, etc.

L'instance courante porte un nom particulier dans le corps des méthodes ; il est souvent implicite mais il peut être nommément désigné par « *self* » en *Smalltalk* et O_2C, « *this* » en C++, « *me* » dans *Trellis/Owl*.

Notre technique d'extraction de la commutativité des méthodes ne nécessite pas une analyse très pointue du code source des méthodes. Nous n'avons besoin de porter notre attention que sur des instructions simples et les expressions, jamais sur les structures de séquencement (boucles et alternatives). Par conséquent, le corps d'une méthode peut-être vu comme un simple ensemble d'instructions (affectations et envois de messages) et d'expressions (conditions et paramètres effectifs).

Les seuls messages qui nous intéressent sont ceux qui sont envoyés à l'instance courante

elle-même, c'est-à-dire à « *self* ». Les messages envoyés à d'autres instances par l'intermédiaire d'un champ de type référence ne nous intéressent pas car le contrôle de concurrence sera alors délégué aux instances réceptrices de ces messages. Nous distinguons les messages auto-dirigés en deux catégories : les messages simples et les messages préfixés par un nom de classe ancêtre.

La première forme est la plus usuelle ; c'est celle utilisée dans le corps de la méthode *Déplacer* en figure IV.1 ou par m_1 en figure IV.2. Nous la noterons avec la syntaxe « *send* M *to* f » où M est le nom de méthode invoquée sur l'instance f.

Les formes préfixées apparaissent quand une méthode n'est pas redéfinie complètement mais comme une extension de la méthode surchargée, par exemple m_2 dans la classe c_2 (cf. figure IV.2). Dans ce cas, le nouveau code contient un appel à la méthode surchargée. Comme elles portent le même nom, il est nécessaire de pouvoir préciser à quelle classe appartient la version surchargée. Ce genre d'envoi de messages est typique des méthodes de création ou de destruction des instances (méthodes constructeur et destructeur en C++). La méthode qui crée une personne physique va tout d'abord appeler le constructeur de la classe PERSONNE, qui initialisera les champs hérités, puis initialisera ensuite les champs supplémentaires. Avec l'héritage multiple, il se produit des conflits sur les noms si la même méthode est définie dans plusieurs super-classes. Le préfixe est explicitement le nom de la classe ancêtre la plus proche dans le graphe d'héritage à partir de laquelle la méthode doit être recherchée. Ceci permet d'appeler un nombre quelconques de méthodes de même noms ainsi que des méthodes de noms différents, ou encore de traiter les exceptions d'héritage. (Smalltalk introduit un forme préfixée par le mot clé « *super* ». Elle permet d'appeler uniquement la méthode de même nom définie dans la super-classe directe. Cette forme n'étant pas assez générale, nous n'en tenons pas compte ici.) Nous noterons cette forme avec la syntaxe « *send* $C.M$ *to* f » où C est la classe ancêtre de la classe propre de f à partir de laquelle il faut rechercher la méthode M.

Affectations, utilisation d'un champ dans une expression et les deux formes de messages auto-dirigés sont donc les éléments que nous analysons pour construire les vecteurs d'accès directs puis transitifs.

IV.3 QUATRE PROBLÈMES

Dans cette section, nous allons détailler les quatre problèmes évoqués dans l'introduction, c'est-à-dire la difficulté d'avoir à fournir des tables de commutativité *ad hoc* pour chaque classe, la surcharge de verrouillage, les escalades de verrou et interblocages qui en découlent, enfin, les pspseudo conflits

commutativité des méthodes Un ADT peut, généralement, être implémenté une fois pour toutes. Il est alors intéressant de rechercher pour ses opérations une relation de commutativité *ad hoc* qui autorise le maximum de parallélisme avec une surcharge de travail acceptable du point de vue des performances. De tels ADTs sont l'ADT ENSEMBLE [Weihl 88], l'ADT RELATION (« AMAP ») [Weihl & Liskov 85], l'ADT RÉPERTOIRE [Schwarz & Spector 84], l'ADT PILE, l'ADT COMPTEUR [O'Neil 86], etc. De plus, si les relations de commutativité ne sont bâties que sur le comportement observables, modifier l'implémentation des opérations n'aura pas de répercussion sur ces relations.

Les classes ont un lien de parenté évident avec les ADT, mais elles ont aussi une différence de taille : la relation d'héritage. Par conséquent, deux méthodes de même nom, appartenant à des classes liées par l'héritage peuvent avoir des relations de commutativité différentes. Modifier une méthode peut avoir des répercussions pour plusieurs des sous-classes descendantes de celle où est définie la méthode. De plus, alors que l'on peut espérer de la part d'un ADT qu'il restera stable dans le temps, les classes et les méthodes sont appelées à être plus fréquemment créées, détruites ou modifiées. Dès lors, automatiser la détermination de la commutativité entre méthodes est un objectif primordial. De toute façon, il est inimaginable de laisser à la charge d'un programmeur d'application la responsabilité de fournir explicitement les propriétés de commutativité de *tout couple* de méthodes, pas plus qu'il n'est raisonnable de lui demander de fournir l'opération inverse de *toute* méthode pour traiter les rejets de transactions.

Il faut tout de même remarquer que nous n'éliminons pas complètement l'utilisation de relations de commutativité *ad hoc*. Celles-ci sont intéressantes pour des objets, généralement livrés avec le système, comme le type de base « *integer* » ou la classe COLLECTION, qui pourront avoir été programmés de façon à offrir le parallélisme maximal qu'ils autorisent.

Nous illustrerons les trois autres problèmes à partir de l'exemple de la figure IV.2. La hiérarchie qui y est représentée est plutôt simple mais instructive. Tout d'abord, en ce qui concerne les *variables d'instance*, que nous abrègerons en *champs*, on constate qu'elles peuvent être de

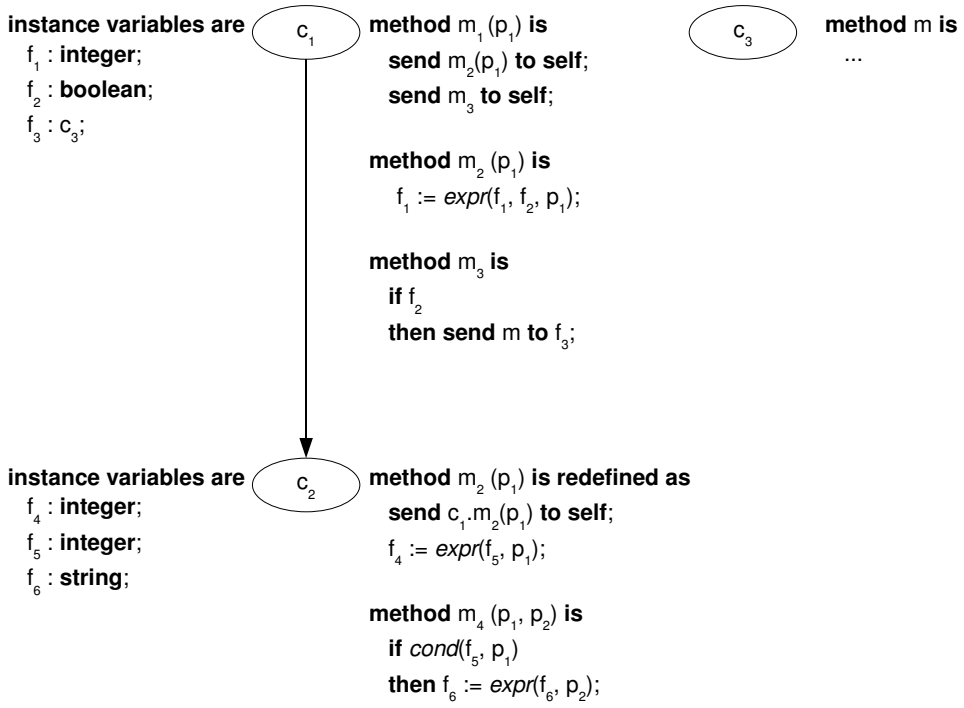


FIG. IV.2 – Quelques aspects de la programmation à objets

type prédéfini (« *integer* », « *boolean* », etc.) ou désigner une classe quelconque du graphe d'héritage et prendre comme valeur toute instance générale[2] de cette classe (ce que l'on appelle un *lien de composition*).

Pour ce qui est des méthodes, on peut remarquer que certaines sont très générales (comme m_1), les détails de l'implémentation étant sous-traités à d'autres méthodes (m_2 et m_3), ces dernières pouvant être surchargées dans des classes descendantes (comme m_2 dans la classe c_2) pour spécialiser l'algorithme. C'est ce que l'on appelle *factorisation du code*. Ceci se traduit par l'utilisation intensive de messages qui sont envoyées à l'instance courante elle-même (« *self* »). On remarque également qu'une méthode peut étendre le travail de celle qu'elle surcharge, comme m_2 dans la classe c_2. Dans ce cas, elle fait appel à sa version surchargée, avant, après ou pendant son traitement spécifique. Enfin, toutes les méthodes n'apparaissent pas aux même niveau dans le graphe d'héritage ; c'est ainsi que la méthode m_3 est spécifique aux instances générales de la classe c_2.

Tout cela peut sembler trivial, mais l'examen sous l'angle du contrôle de concurrence va démontrer que les propositions qui se contentent d'associer un mode d'accès *Lire* ou *Écrire* à une méthode sont insuffisantes. Dans ce cas, en classe c_1, les méthodes m_1 et m_3 seront classées comme lectrices, alors que la méthode m_2 sera un écrivain. Dans la classe c_2, les méthodes héritées m_1 et m_3 ne changeront pas de statut, la méthode surchargée m_2 ainsi que la nouvelle méthode m_3 seront classées comme écrivains. Les problèmes sont au nombre de trois :

[2]Rappelons que ce que nous avons baptisé instances générales d'une classe est l'union de l'ensemble des instances de cette classe ainsi que de toutes ses classes descendantes.

(i) une instance donnée peut être contrôlée un grand nombre de fois pour ce qui n'est virtuellement qu'un seul accès ;

(ii) ces contrôles répétés peuvent engendrer un nombre important d'interblocages dus à des escalades de verrou ;

(iii) des méthodes, dont l'une au moins est un écrivain, agissant sur des parties indépendantes sont inutilement mises en conflit.

Détaillons chacun de ces inconvénients.

trop de contrôles d'accès concurrents La factorisation du code engendre un nombre important d'envois de messages auto-dirigés, comme pour la méthode m_1. Cette forme de programmation étant l'une des forces de l'approche à objets, grâce à la possibilité de *surcharge* (« *overriding* ») et au mécanisme de *liaison dynamique* (« *late binding* »), il faut s'attendre à ce qu'elle soit souvent exploitée. Ceci ne doit pas devenir un inconvénient majeur pour le contrôle de concurrence. Si un contrôle est effectué à chaque envoi de message, alors appliquer la méthode m_1 à une instance propre de c_1 va nécessiter trois contrôles successifs sur cette même instance (pour m_1, m_2 et m_3), peut être plus sur des instances générales (dans le cas où les méthodes surchargées invoqueraient à leur tour des méthodes supplémentaires).

escalade de verrou et interblocage On cite souvent les résultats d'expériences sur *System R* qui montrent que 97 % des interblocages sont dus à des escalades de verrou du mode *Lire* au mode *Écrire* ; jusqu'à 76 % de ces interblocages pourraient être éliminés par la pré-annonce du verrou le plus exclusif [Korth et al. 88].

C'est typiquement ce qui se passe avec la méthode m_1. Elle acquiert le verrou *Lire* sur une instance puis envoie le message m_2 qui, tout du moins pour les instances des classes c_1 et c_2, demandera un verrou *Écrire*. Demander immédiatement le verrouillage de l'instance dans le mode le plus restrictif éviterait ce risque majeur d'interblocage.

pseudo conflits Il est vraisemblable qu'une méthode qui a été classée comme écrivain (respectivement lectrice) verra ses homologues qui la surchargent dans des classes descendantes conserver ce label. C'est le cas pour la méthode m_2. En revanche, de nouvelles méthodes peuvent être des écrivains mais sur des parties distinctes seulement, en particulier les champs ajoutés. La dichotomie entre lecteurs et écrivains de tient pas compte de cette réalité. C'est ici le cas avec les méthodes m_2 et m_3 dans la classe c_2 : m_3 est une méthode qui n'existe pas dans c_1 et qui ne manipule que les champs définis dans c_2. Les classer toutes deux comme écrivains les rend exclusives, ce qui est une perte de concurrence *déraisonnable*, comme nous allons l'exposer.

En effet, il est essentiel de remarquer que ce problème ne se pose pas dans les bases de données relationnelles. Supposons que l'on veuille représenter les classes c_1 et c_2 dans un schéma relationnel : elles deviendront des relations. En supposons que le champs f_1 soit désigné comme clef primaire de la relation c_1, la relation c_2 contiendra les champs définis dans la classe c_2 plus f_1 comme clef étrangère. Si une transaction doit accéder à tous les champs de la classe c_2, alors une équi-jointure sera réalisée entre les relations c_1 et c_2 sur le champ f_1. Par conséquent, les deux relations seront soumises au contrôle de concurrence. En revanche, si seuls les champs définis dans c_2 sont manipulés, alors seule la relation c_2 sera contrôlée ce qui permettra à une transaction concurrente qui n'accèderait qu'aux champs définis dans c_1 de s'exécuter en parallèle sur la relation c_1. Cette exécution concurrente est obtenue, dans une base de données relationnelle, sans avoir à considérer un granule de verrouillage inférieur au n-uplet. Au contraire, dans une base de données à objets, il devient impératif de verrouiller au niveau des champs pour autoriser ce genre de parallélisme.

Nous avons donc mis en évidence dans cette section quatre problèmes qui jusqu'à maintenant n'avaient pas été reportés dans la littérature. Notre proposition élimine ces quatre inconvénients grâce à une technique d'analyse assez simple du code source des méthodes et à un algorithme efficace de calcul sur des fermetures réflexo-transitives. Un mode d'accès est généré par méthode et par classe. Tout ce travail étant réalisé en phase de compilation, aucune perte de performance n'est endurée pendant l'exécution.

Très sommairement, la technique va consister à associer à chaque méthode un vecteur d'accès direct. Une classe est un produit cartésien des domaines de ses différents champs, ainsi c_1 définit-elle le domaine « integer $\times$ boolean $\times$ c » et c_2 celui de « integer $\times$ boolean $\times$ c $\times$ integer $\times$ integer $\times$ string ». À chaque méthode, dans chaque classe où elle est définie, nous allons associer un vecteur de la même dimension que le produit cartésien. Chaque valeur composant ce vecteur décrira le mode d'accès le plus restrictif qu'emploie la méthode sur le champ correspondant, ce mode d'accès se limitant à *Nul*, *Lire* ou *Écrire*. Par exemple, le vecteur d'accès direct de m_1 sera (*Nul*, *Nul*, *Nul*), celui de la m_2 dans la classe c_2 étant (*Nul*, *Nul*, *Nul*, *Écrire*, *Lire*, *Nul*). La commutativité entre méthodes est déterminée par comparaison des vecteurs d'accès. Plus exactement, deux vecteurs d'accès sont compatibles si les modes d'accès sont deux à deux compatibles sur les composantes du plus court des deux vecteurs ; seul *Écrire* est incompatible avec tous les autres modes. Ainsi les deux vecteurs ci-dessus sont-ils compatibles puisque les trois premières composantes sont toutes égales à *Nul*.

La reprise utilise également les vecteurs d'accès comme critère de projection afin d'extraire les champs modifiés par la méthode, ceux dont la composante est positionnée à *Écrire*.

Les vecteurs d'accès directs éliminent le premier et le quatrième inconvénient. Ils ne sont que des éléments de base qui vont nous permettre de calculer des *vecteurs d'accès transitifs*. Ces derniers résolvent les deux autres problèmes.

IV.4 CALCUL DES VECTEURS D'ACCÈS TRANSITIFS

Le calcul des vecteurs d'accès transitifs n'est pas complètement évident, même s'il ne s'agissait que d'ADT, c'est-à-dire d'approche « basée sur les objets » et non « à objets », d'après la terminologie de [Wegner 90], à cause des appels récursifs entre méthodes. Dans une approche réellement à objets, c'est-à-dire intégrant l'héritage, la tâche est compliquée par l'*héritage multiple*, la *surcharge* et la *liaison dynamique*. Nous devons donc définir nos « vecteurs » d'une façon moins évidente que ne l'a laissé supposer l'introduction.

IV.4.1 Définitions préliminaires

Tout d'abord, précisons que nous ne formaliserons pas la notion de code source d'une méthode puisque nous n'avons besoin de l'interpréter que pour détecter les affectations, les accès en lecture des champs et les messages, préfixés ou non, qui sont envoyés à l'instance courante. Donc, nous nous appuierons sur la compréhension informelle que le lecteur a du sens du code source.

Commençons par des définitions de base sur les classes et les modes d'accès classiques.

Définition IV.1 *À chaque classe, nous associons un couple composé d'un ensemble de champs et d'un ensemble de méthodes :*

$$CLASSES \rightarrow \mathcal{P}(CHAMPS) \times \mathcal{P}(METHODES)$$

où CHAMPS est l'ensemble des identifiants de champs tandis que METHODES est l'ensemble de noms de méthodes.

Les identifiants de champs sont des numéros uniques associés à chaque champ au moment de sa déclaration ; autrement dit, les identifiants servent à différencier dans l'ensemble des champs ceux qui auraient été déclarés avec le même nom externe dans des classes incomparables (cf. en figure IV.1 la variable d'instance « *Angle* » dans les classes CARRÉ et ELLIPSE). En revanche, l'intérêt de la surcharge, en vue du polymorphisme, est directement lié à l'exploitation d'un même nom de message pour désigner plusieurs méthodes de codes physiquement distincts.

Définition IV.2 *Nous utiliserons la notation $CHAMPS(X)$ pour extraire l'ensemble des champs apparaissant dans la structure X, qui pourra être une classe ou un vecteur d'accès.*

	Nul	*Lire*	*Écrire*
Nul	*oui*	*oui*	*oui*
Lire	*oui*	*oui*	non
Écrire	*oui*	non	non

TAB. IV.1 – Relation de compatibilité « classique »

De même, nous appellerons METHODES(C) l'ensemble des méthodes, héritées ou définies, d'une classe C.

Enfin, nous noterons respectivement DESCENDANTS(C) et ANCETRES(C) l'ensemble des classes héritant de C et C elle-même, et l'ensemble des classes dont C hérite ainsi que C elle-même.

Définition IV.3 *Nous appelons C_{MODES} la relation binaire de compatibilité définie sur MODES est donnée en extension en table IV.1, où MODES $= \{$Nul, Lire, Écrire$\}$ avec Nul $\prec$ Lire $\prec$ Écrire.*

La relation d'ordre sur *MODES* se déduit directement de la relation de compatibilité par inclusion des lignes (et des colonnes pour les tables asymétriques) [Korth 83]. Comme *MODES* est totalement ordonné, nous pourrons utiliser l'opérateur « sup » ($\vee$) existant dans un treillis, qui donne l'unique maximum commun (par exemple, *Lire* $\vee$ *Écrire* $=$ *Écrire*).

Nous allons maintenant donner la définition des vecteurs d'accès et une « algèbre » minimale sur ceux-ci. (Il aurait été possible d'étendre la structure de treillis aux vecteurs d'accès mais cela est parfaitement inutile pour la suite.)

Définition IV.4 *Le « vecteur » d'accès d'une méthode M dans une classe C est une famille de modes indexés par les champs de la classe :*

$$A_{C,M} = (m_{ch} \in \mathit{MODES})_{ch \in \mathit{CHAMPS}(C)}.$$

Exemple IV.2 *Illustrons cette notion avec les définitions de classes données avec l'algorithme IV.2. Intuitivement, les vecteurs d'accès directs, mais aussi transitifs puisque ces méthodes n'en invoquent aucune autre, sont :*

- $A_{\text{EMPLOYÉ,}\mathit{AugmenterSalaire}} = (\mathit{Écrire}_{Salaire})$;
- $A_{\text{PROGRAMMEUR,}\mathit{AugmenterSalaire}} = (\mathit{Écrire}_{Salaire})$;
- $A_{\text{PROGRAMMEUR,}\mathit{BonusProductivité}} = (\mathit{Nul}_{Salaire}, \mathit{Lire}_{LignesEcrites}, \mathit{Écrire}_{BonusLignes})$.

Remarquez que les deux méthodes AugmenterSalaire *et* BonusProductivité *sont toutes deux des écrivains, mais, qu'ayant été définies à des niveaux différents dans la hiérarchie d'héritage, la seconde ne modifie que des champs définis dans la classe* PROGRAMMEUR.

Algorithme IV.2 Un exemple de méthodes en O_2C

```
class Employé inherit PersonnePhysique with extension
    type tuple (Salaire : float)
    method AugmenterSalaire (Pourcentage : float)
end

class Programmeur inherit Employé with extension
    type tuple (LignesEcrites : integer,
                BonusLignes : float)
    method BonusProductivité (Limite : integer ;
                              Francs : float ;
                              Par : integer)
end

method body AugmenterSalaire (Pourcentage : float)
in class Employé
{
self->Salaire *= 1 + Pourcentage ;
}

method body BonusProductivité (Limite : integer ;
                               Francs : float ;
                               Par : integer)
in class Programmeur
{
if (self->LignesEcrites > Limite)
    self->BonusLignes = (self->LignesEcrites - Limite) % Par * Francs ;
}
```

Nous étendons l'opérateur « sup » ($\vee$) défini sur *MODES* aux vecteurs d'accès et présentons quelques propriétés algébriques nécessaires à l'algorithme de calcul des vecteurs d'accès transitifs de la section IV.4.3.

Définition IV.5 *Soient a' et a'' des vecteurs d'accès, nous étendons l'opérateur sup de la façon suivante :*

$$a' \vee a'' \in (m_{ch} \in \mathit{MODES})_{ch \in \mathit{CHAMPS}(a') \cup \mathit{CHAMPS}(a'')}$$

avec :

$$\begin{aligned} a' \vee a'' = \; & (m'_{ch} \vee m''_{ch})_{ch \in \mathit{CHAMPS}(a') \cap \mathit{CHAMPS}(a'')} \cup \\ & (m'_{ch})_{ch \in \mathit{CHAMPS}(a') \; \mathit{CHAMPS}(a'')} \cup \\ & (m''_{ch})_{ch \in \mathit{CHAMPS}(a'') \; \mathit{CHAMPS}(a')}. \end{aligned}$$

La fusion de vecteurs d'accès consiste donc à conserver tous les champs en ne gardant pour les champs communs que le mode d'accès le plus restrictif.

Exemple IV.3

$$\begin{array}{rccccc} & (\mathit{Nul}_{f_1}, & \mathit{Nul}_{f_2}, & \mathit{Nul}_{f_3}, & &) \\ \vee & (& & & \mathit{Écrire}_{f_4}, & \mathit{Lire}_{f_5}) \\ \vee & (\mathit{Écrire}_{f_1}, & \mathit{Lire}_{f_2}, & & &) \\ \vee & (& \mathit{Lire}_{f_2}, & \mathit{Lire}_{f_3}, & &) \\ \hline = & (\mathit{Écrire}_{f_1}, & \mathit{Lire}_{f_2}, & \mathit{Lire}_{f_3}, & \mathit{Écrire}_{f_4}, & \mathit{Lire}_{f_5}). \end{array}$$

L'algorithme de la section IV.4.3 nécessite la propriété suivante.

Propriété IV.1 *L'opérateur sup étendu aux vecteurs d'accès est idempotent, commutatif et associatif.*

Preuve IV.2 *Immédiat car sup sur MODES possède ces trois propriétés.*

Nous en terminerons avec cette section préliminaire en donnant la définition, attendue, de la compatibilité entre vecteurs d'accès : deux vecteurs d'accès sont compatibles si les modes sont compatibles sur chaque champ commun.

Définition IV.6 *La relation de compatibilité entre deux vecteurs, a' et a'', est la suivante :*

$$\begin{array}{l} a' \mathcal{C} a'' \\ \leftrightarrow \\ \forall ch \in \mathit{CHAMPS}(a') \cap \mathit{CHAMPS}(a''), \\ \quad C_{\mathit{MODES}}(m'_{ch}, m''_{ch}). \end{array}$$

IV.4.2 Compilation des méthodes

Pour déterminer la commutativité des méthodes, leurs codes sources sont analysés par le compilateur. Dans cette section, nous donnons les trois définitions qui correspondent aux instructions que nous avons retenues auparavant, c'est-à-dire l'affectation, l'accès à un champ dans une expression et les messages auto-dirigés, simples ou préfixés.

En fait, les définitions suivantes sont très exactement les spécifications des informations qui doivent être extraites par le compilateur. La définition IV.7 donne la valeur du vecteur d'accès direct d'une méthode. Les ensembles définis en IV.8 et IV.9 vont servir à construire, dans la section suivante, le graphe de résolution des messages associé à chaque classe, prérequis afin de calculer les vecteurs d'accès transitifs.

Définition IV.7 *Soit C une classe, alors à chaque méthode M définie dans cette classe, nous associons un* vecteur d'accès direct *$\mathit{VAD}_{C,M}$ tel que :*

(i) si M est héritée d'une super-classe C', alors :

$$\mathit{VAD}_{C,M} = \mathit{VAD}_{C',M} \vee (\mathit{Nul}_{ch})_{ch \in \mathit{CHAMPS}(C)} \ ;$$

(ii) autrement, si M est définie pour la première fois ou surchargée dans C, alors :
$\forall ch \in$ *CHAMPS*(C),

Écrire$_{ch} \in$ *VAD*$_{C,M} \leftrightarrow$ *il existe une affectation de la forme « ch :=* <expression> *» dans le code source de M ;*

Lire$_{ch} \in$ *VAD*$_{C,M} \leftrightarrow$ *il n'apparaît pas de telle affectation, mais « ch » est utilisé dans une expression quelconque (partie gauche d'affectation, condition, paramètre effectif ou destinataire d'un message) ;*

Nul$_{ch} \in$ *VAD*$_{C,M} \leftrightarrow$ *« ch » n'apparaît nulle part dans le code source de M.*

Exemple IV.4 *Avec les définitions de la figure IV.2, nous obtenons :*

$$\begin{array}{lcl} VAD_{c_1,m_1} & = & (Nul_{f_1}, \quad Nul_{f_2}, \quad Nul_{f_3}) \\ VAD_{c_1,m_2} & = & (\acute{E}crire_{f_1}, \quad Lire_{f_2}, \quad Nul_{f_3}) \\ VAD_{c_1,m_3} & = & (Nul_{f_1}, \quad Lire_{f_2}, \quad Lire_{f_3}) \end{array}$$

et :

$$\begin{array}{lcll} VAD_{c_2,m_1} & = & VAD_{c_1,m_1} \vee & (Nul_{f_1}, \; Nul_{f_2}, \; Nul_{f_3}, \; Nul_{f_4}, \; Nul_{f_5}, \; Nul_{f_6}) \\ & = & & (Nul_{f_1}, \; Nul_{f_2}, \; Nul_{f_3}, \; Nul_{f_4}, \; Nul_{f_5}, \; Nul_{f_6}) \\ VAD_{c_2,m_2} & = & & (Nul_{f_1}, \; Nul_{f_2}, \; Nul_{f_3}, \; \acute{E}crire_{f_4}, \; Lire_{f_5}, \; Nul_{f_6}) \\ VAD_{c_2,m_3} & = & VAD_{c_1,m_3} \vee & (Nul_{f_1}, \; Nul_{f_2}, \; Nul_{f_3}, \; Nul_{f_4}, \; Nul_{f_5}, \; Nul_{f_6}) \\ & = & & (Nul_{f_1}, \; Lire_{f_2}, \; Lire_{f_3}, \; Nul_{f_4}, \; Nul_{f_5}, \; Nul_{f_6}) \\ VAD_{c_2,m_3} & = & & (Nul_{f_1}, \; Lire_{f_2}, \; Lire_{f_3}, \; Nul_{f_4}, \; Lire_{f_5}, \; \acute{E}crire_{f_6}) \end{array}$$

En fait, distribuer les champs des instances dans plusieurs relations, comme nous l'avons fait en section IV.3, puis verrouiller séparément les relations, revient en quelque sorte à créer un vecteur d'accès grossier. Lorsqu'une requête est soumise, les relations utilisées sont verrouillées soit en mode exclusif, soit en mode partagé. Les relations qui n'interviennent pas dans l'élaboration de la réponse ne sont tout simplement pas verrouillées, ce qui correspond au mode d'accès *Nul*.

Définition IV.8 *Soit C une classe, alors, à chaque méthode M définie dans cette classe, nous associons un ensemble de messages auto-dirigés directs, $MD_{C,M}$ comme suit :*

(i) si M est héritée de C', alors :

$$MD_{C,M} = MD_{C',M} \; ;$$

(ii) autrement :

$$MD_{C,M} = \left\{ M' \in METHODES(C) \;\middle|\; \begin{array}{l} \textit{le message « send } M' \textit{ to self »} \\ \textit{apparaît dans le code source de } M \end{array} \right\}.$$

Exemple IV.5 *En figure IV.2, nous obtenons :*

$$\begin{aligned} MD_{c_1,m_1} &= \{m_2, m_3\} \\ MD_{c_1,m_2} &= \emptyset \\ MD_{c_1,m_3} &= \emptyset \end{aligned}$$

et :

$$\begin{aligned} MD_{c_2,m_1} &= MD_{c_1,m_1} \\ &= \{m_2, m_3\} \\ MD_{c_2,m_2} &= \emptyset \\ MD_{c_2,m_3} &= MD_{c_1,m_3} \\ &= \emptyset \\ MD_{c_2,m_3} &= \emptyset \end{aligned}$$

Ce sont ces ensembles de messages auto-dirigés directs qui serviront, à la *compilation*, à résoudre statiquement, *pour une classe*, les liaisons dynamiques qui, *pour les instances*, devront être résolues *à l'exécution*. Il faut pour cela attendre jusqu'à la définition IV.10.

Définition IV.9 *Soit C une classe, alors, à chaque méthode M définie dans cette classe, nous associons un ensemble de* messages auto-dirigés préfixés, *$MP_{C,M}$, comme suit :*

(i) si M est héritée de C' :

$$MP_{C,M} = MP_{C',M} \ ;$$

(ii) autrement :

$$MP_{C,M} = \left\{ (C', M') \;\middle|\; \begin{array}{l} C' \in \textit{ancêtres} C \wedge \\ M' \in METHODES(C') \wedge \\ \textit{le message « send } C'.M' \textit{ to self »} \\ \textit{apparaît dans le code source de } M \end{array} \right\}.$$

Exemple IV.6 *En figure IV.2, nous obtenons :*[3]

$$\begin{aligned} MP_{c_1,m_1} &= \emptyset \\ MP_{c_1,m_2} &= \emptyset \\ MP_{c_1,m_3} &= \emptyset \end{aligned}$$

et :

$$\begin{aligned} MP_{c_2,m_1} &= MP_{c_1,m_1} \\ &= \emptyset \\ MP_{c_2,m_2} &= \{c_1.m_2\} \\ MP_{c_2,m_3} &= MP_{c_1,m_3} \\ &= \emptyset \\ MP_{c_2,m_3} &= \emptyset. \end{aligned}$$

Il convient de noter que construire les vecteurs d'accès directs (*VAD*) ainsi que les deux ensembles de messages auto-dirigés (*MD* et *MP*) est une tâche aisée pour un compilateur. Ce genre de travail doit déjà être effectué pour traduire le code source en code objet ; il ne s'agit donc que de recupérer des informations contenues dans les tables du compilateur.

Les vecteurs d'accès directs donnés par la définition IV.7 peuvent se révéler tout à fait suffisants pour les ADT. Mais, en programmation à objets, nous ne pouvons pas nous en contenter à cause de la factorisation du code et des problèmes qu'elle engendre (cf. section IV.3). En pratique, chaque message devra faire l'objet d'un contrôle de concurrence et comme les messages auto-dirigés sont assez fréquents, le nombre de contrôle croîtra dans les mêmes proportions. Il serait donc bon de ne contrôler les accès concurrents à une instance que lors de l'invocation du message initial. Par conséquent, nous devons construire des vecteurs d'accès dits transitifs.

IV.4.3 Un algorithme

Dans cette section, nous ne ferons que décrire le paramètre d'entrée de l'algorithme puisqu'il réside essentiellement dans l'algorithme de détermination des composantes fortement connexes d'un graphe, résolu efficacement par [Tarjan 72].

À partir des informations extraites lors de la compilation de toutes les méthodes d'une classe C, nous construisons le graphe de résolution des liaisons dynamiques qui est applicable à toute instance propre de la classe C.

[3]Pour simplifier la lecture, nous écrivons « $c_1.m_2$ » au lieu de « (c_1, m_2) », bien que cela soit abusif quand m_2 est héritée et non définie dans c_1.

Définition IV.10 *Soit C une classe, alors $GR_C = (X, G)$ est son* graphe de résolution des liaisons dynamiques*, avec :*

- $X = (\{C\} \times \mathit{METHODES}(C)) \cup \bigcup_{M \in \mathit{METHODES}(C))} \mathit{MP}^*_{C,M}$;
- $\forall (C', M') \in X$,
  $\Gamma_{C',M'} = (\{C\} \times \mathit{MD}_{C',M'}) \cup \mathit{MP}_{C',M'}$;

où MP est la fermeture réflexo-transitive des appels préfixés.*

X est formé de l'ensemble des méthodes qui pourront être éventuellement exécutées lors de l'envoi d'un message à une instance propre de la classe C.

Γ est formé de l'ensemble des appels de méthodes qui lient les différentes méthodes pouvant être invoquées lors de l'envoi d'un message à une instance de la classe C. C'est dans la première partie de la formule, $\{C\} \times \mathit{MD}_{C',M'}$, que les liaisons dynamiques sont résolues, c'est-à-dire que les envois de messages recueillis dans les *MD* sont transformés, virtuellement, en appels de méthodes. Quant aux *MP*, ce sont déjà des appels de méthodes, les conflits dus à l'héritage multiple ayant été levés auparavant.

Exemple IV.7 *La figure IV.3 présente les deux graphes de résolution des liaisons dynamiques associés aux classes c_1 et c_2 de la figure IV.2. Détaillons la construction de Gc_2. Nous obtenons à partir des données construites dans les exemples IV.4, IV.5 et IV.6 :*

$$\begin{aligned}
\mathit{MP}^*_{c_2,m_1} &= \emptyset \\
\mathit{MP}^*_{c_2,m_2} &= \mathit{MP}_{c_2.m_2} \cup \mathit{MP}_{c_1.m_1} = c_1.m_2 \cup \emptyset \\
\mathit{MP}^*_{c_2,m_3} &= \emptyset \\
\mathit{MP}^*_{c_2,m_3} &= \emptyset
\end{aligned}$$

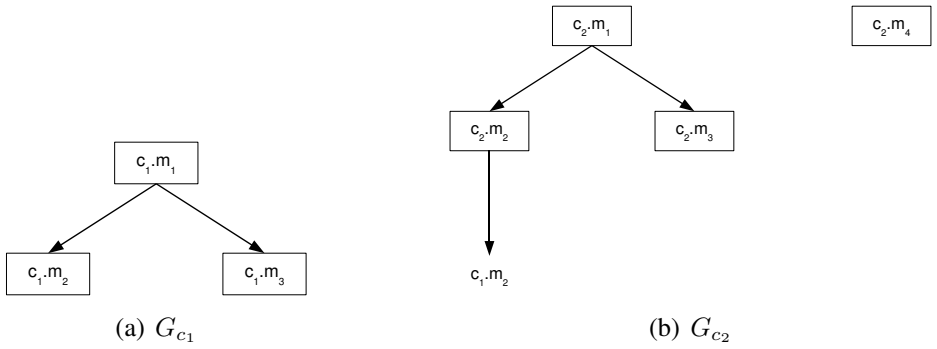
d'où :

$$X = \{c_2.m_1, c_2.m_2, c_2.m_3, c_2.m_3\} \cup \emptyset \cup \{c_1.m_2\} \cup \emptyset \cup \emptyset$$

puis :

$$\begin{aligned}
G_{c_2}.m_1 &= \{c_2.m_2, c_2.m_3\} \cup \emptyset \\
G_{c_2}.m_2 &= \emptyset \cup \{c_1.m_2\} \\
G_{c_2}.m_3 &= \emptyset \cup \emptyset \\
G_{c_2}.m_3 &= \emptyset \cup \emptyset \\
G_{c_1}.m_2 &= \emptyset
\end{aligned}$$

Nous obtenons donc bien le graphe G_{c_2} représenté figure IV.3. Nous avons reporté les équivalences de messages, c'est-à-dire que, par exemple, les messages $c_2.m_1$ et $c_1.m_1$ aboutissent à



(a) G_{c_1} (b) G_{c_2}

FIG. IV.3 – Graphes de résolution des liaisons dynamiques pour les instances propres des classes c_1 et c_2 de la figure IV.2

la même méthode (cf. section III.2.2 pour revoir la différence entre message et méthode).

Ces graphes mettent en évidence les différences d'exécutions quand le même message m_1 est envoyé à une instance propre de la classe c_1 ou de la classe c_2.

Une autre particularité de ces graphes est que, bien qu'étant particuliers à une classe, ils ne sont pas indépendants les uns des autres. Ici, la méthode $c_2.m_1$ n'est autre que $c_1.m_1$ et, par conséquent, elle est commune aux deux graphes. Au total, ces deux graphes ne partagent pas moins de trois méthodes.

Les vecteurs d'accès transitifs sont définis à partir des vecteurs d'accès direct et du graphe de résolution des liaisons dynamiques.

Définition IV.11 (Vecteur d'accès transitif) *Soient C une classe, M une méthode définie dans C, alors le vecteur d'accès transitif, $VAT_{C,M}$, est :*

$$VAT_{C,M} = VAD_{C,M} \vee \bigvee_{(C',M') \in G^*_{C,M}} VAD_{C',M'}$$

où G^* est la fermeture réflexo-transitive de G.

La définition de la valeur du vecteur d'accès transitif d'une méthode M dans une classe C est donc très simple : il s'agit du sup de tous les vecteurs d'accès directs correspondants aux méthodes qui peuvent être éventuellement exécutées quand le message M est envoyé à une instance propre de la classe C.

Il s'agit donc une approche pessimiste qui envisage uniquement l'exécution où toutes les méthodes seraient appliquées. Bien évidemment, la présence d'alternatives ne permet souvent pas à ce scénario de se réaliser. C'est cependant le prix à payer pour ne pas avoir à contrôler chaque appel de méthode. La solution proposée dans [Malta & Martinez 92c] [Malta 93] permettrait d'éliminer cet inconvénient, mais elle n'est réalisable efficacement que pour les ADT, l'héritage venant considérablement alourdir le coût de construction d'un vecteur d'accès réel

pendant l'exécution. De toute façon, ces vecteurs d'accès transitifs vont être utilisés pour des accès à des ensembles d'instances en section IV.5, d'où plusieurs exécutions différentes ont plus de chances d'aboutir à l'exploration de tous les cas de figure.

Exemple IV.8 *À partir des différents VAD donnés dans l'exemple IV.4 et des graphes de la figure IV.3, nous obtenons pour les instances propres de la classe c_1 :*

$$\begin{array}{lcll} VAT_{c_1,m_1} & = & VAD_{c_1,m_1} \vee VAD_{c_1,m_2} \vee VAD_{c_1,m_3} & = (\acute{E}crire_{f_1},\ Lire_{f_2},\ Lire_{f_3}) \\ VAT_{c_1,m_2} & = & VAD_{c_1,m_2} \vee VAD_{c_1,m_3} & = (\acute{E}crire_{f_1},\ Lire_{f_2},\ Nul_{f_3}) \\ VAT_{c_1,m_3} & = & VAD_{c_1,m_3} & = (Nul_{f_1},\ Lire_{f_2},\ Lire_{f_3}) \end{array}$$

tandis que pour les messages m_1 envoyés à une instance propre de la classe c_2, nous avons :

$$\begin{array}{lcl} VAT_{c_2,m_1} & = & (\acute{E}crire_{f_1},\ Lire_{f_2},\ Lire_{f_3},\ \acute{E}crire_{f_4},\ Lire_{f_5},\ Nul_{f_6}) \\ VAT_{c_2,m_2} & = & (\acute{E}crire_{f_1},\ Lire_{f_2},\ Nul_{f_3},\ \acute{E}crire_{f_4},\ Lire_{f_5},\ Nul_{f_6}) \\ VAT_{c_2,m_3} & = & (Nul_{f_1},\ Lire_{f_2},\ Lire_{f_3},\ Nul_{f_4},\ Nul_{f_5},\ Nul_{f_6}) \\ VAT_{c_2,m_3} & = & (Nul_{f_1},\ Nul_{f_2},\ Nul_{f_3},\ Nul_{f_4},\ Lire_{f_5},\ \acute{E}crire_{f_6}) \end{array}$$

Le calcul de tous les vecteurs d'accès transitifs d'un graphe peut être réalisé efficacement en un seul parcours du graphe utilisant une « descente en profondeur d'abord » ; la complexité en temps sera donc en $O(|X| + |G|)$, c'est-à-dire linéaire en la taille du graphe.

Les vecteurs d'accès transitifs sont calculés depuis les puits du graphe, où l'égalité « $VAT = VAD$ » est évidente, jusqu'aux sources. Chaque fois que le calcul d'un sommet du graphe est achevé, la valeur de son *VAT* est transmise à son prédécesseur dans l'ordre de la descente.

L'algorithme doit être quelque peu amélioré car, les méthodes pouvant s'appeler récursivement par l'intermédiaire d'autres méthodes, les graphes de résolution des liaisons dynamiques peuvent contenir, même si l'on peut supposer cet événement rare, des circuits. En faisant l'observation triviale que les *VAT* de tous les sommets se trouvant sur un circuit commun sont nécessairement égaux puisque leurs G^* sont identiques, nous pouvons encore effectuer les calculs des vecteurs d'accès transitifs en un seul parcours. La version de l'algorithme sera basée sur celui de [Tarjan 72] permettant de déterminer les composantes fortement connexes (cf. annexe A pour les définitions de graphes). Grâce à la propriété IV.1, nous pouvons calculer les dépendances cycliques (idempotence) dans n'importe quel ordre (commutativité et associativité).

Comme on peut s'en rendre compte dans la définition IV.10 et dans l'exemple IV.7, nous devons faire face à un autre problème lors de modifications du graphe de résolution des liaisons dynamiques. Il n'est pas complètement local à une seule classe puisque des méthodes décrites dans des classes ancêtres peuvent être impliquées, soit parce qu'elles n'ont pas été surchargées, soit parce qu'elles ont été explicitement utilisées par un message préfixé. Ceci augmente le nombre de vecteurs d'accès qui devront être recalculés lors de modifications. Il est préférable

de fournir une version incrémentale de l'algorithme. Les sommets qui peuvent être affectés par la modification du code d'une méthode $C.M$ sont ceux appartenant aux différents $G^{-1*}_{C,M}$ dans tous les graphes de résolution des liaisons dynamiques des classes où cette méthode soit n'a pas été surchargée, soit est invoquée directement par un message préfixé.

L'algorithme est donné en annexe B avec sa preuve, des indications pour une implémentation efficace et des initialisations pour des calculs incrémentaux.

Certains pourraient objecter que de tels graphes sont trop lourds à gérer. Nous ferons seulement remarquer qu'une structure de graphe encore plus complexe est mise en œuvre dans O_2 [Zicari 89] [Zicari 91]. Ce ne sont pas seulement les relations entre méthodes appartenant à des classes liées par héritage qui sont prises en compte mais tous les appels, comme le message « *send* m *to* f_3 » en figure IV.2. Ce vaste graphe s'appelle le graphe des dépendances entre méthodes. Ainsi, notre proposition peut même s'intégrer assez aisément à des travaux déjà réalisés.

IV.4.4 Conclusion partielle

Nous sommes maintenant à la fin de la première partie de ce chapitre IV. Nous avons réussi à associer un mode d'accès spécifique à chaque méthode, plutôt que de ne la classer que comme lectrice ou écrivain. De plus, nous avons exhibé un algorithme linéaire permettant de calculer ces modes d'accès spécifiques.

On peut reprocher aux vecteurs d'accès transitifs d'être trop pessimistes puisque les méthodes peuvent contenir des alternatives et qu'ils représentent donc l'union d'exécutions distinctes. Ceci est vrai pour des accès individuels aux instances. Toutefois, dans une base de données, ce sont plutôt les accès ensemblistes qui sont favorisés. Sur une population d'instances, on peut supposer que chaque cas d'exécution pourra se présenter.

Comment utiliser les vecteurs d'accès transitifs pour le verrouillage dans un graphe d'héritage est tout le sujet de la section suivante.

IV.5 VERROUILLAGE DANS UN GRAPHE D'HÉRITAGE

Dans le chapitre III, nous avons élaboré un protocole de verrouillage dans les graphes d'héritage pour lequel seuls les modes d'accès *Lire* et *Écrire* s'appliquent aux instances. En tenant compte de ce que nous affirmions, à savoir que nous pouvons définir autant de mode d'accès que voulu, la première idée est de considérer que chaque vecteur d'accès transitif est un mode d'accès. Mais, les choses ne sont pas aussi simples. En effet, les modes *Lire* et *Écrire* sont extrêmement généraux alors que les vecteurs d'accès transitifs sont spécifiques à une méthode dans une classe. On ne peut donc pas utiliser un vecteur d'accès transitif pour un sous-graphe, c'est-à-dire diverses méthodes, de même nom, dans un ensemble de classes.

Il semblerait que l'on doive encore étendre le protocole. Malheureusement, la solution qui en découle est trop lourde à mettre en œuvre. L'intérêt premier du protocole primitif, qui est de limiter le nombre de verrous qu'une transaction doit obtenir, ne compense pas les nouveaux inconvénients. (Nous avons déjà vu que les modifications concurrentes du graphe d'héritage entament cet avantage.)

Nous avons laissé le développement de cette solution dans la section qui suit afin de souligner ses inconvénients. Le protocole le plus simple s'impose en section III.5.2 et n'est pas sujet à ces problèmes.

IV.5.1 Une extension du protocole primitif

IV.5.1.1 Rappel

Avec l'héritage, il existe deux définitions des instances d'une classe : les instances propres et les instances générales. Les instances générales d'une classe sont l'union des instances propres de toutes ses classes descendantes et de ses instances propres. Par exemple, un carré ou un cercle sont des instances générales de la classe POINT si les classes CARRÉ et CERCLE en sont des sous-classes (cf. figure IV.1). Ainsi, dans un graphe d'héritage, nous sommes amenés à verrouiller non seulement une classe mais également le sous-graphe des classes descendantes de cette classe.

Nous avons décomposé un verrou sur une classe en un triplet composé d'un type de verrou, d'un mode d'accès et d'une intention. Un verrou sur une instance n'est quant à lui composé que du mode d'accès.

Le type de verrou peut prendre trois valeurs : classe, frontière ou chemin. La valeur « *classe* » indique que les accès de la transaction se limitent à la classe et/ou à ses instances propres. La valeur « *frontière* » indique que les accès se font à cette classe, à ses classes descendantes, ainsi qu'à ses instances générales. Enfin, la valeur « *chemin* » indique qu'il y a une classe strictement descendante qui est actuellement utilisée par la transaction.

Le mode d'accès indique de quelle façon les classes et/ou les instances sont manipulées.

Enfin, l'intention indique si toutes les instances sont utilisées ou seulement certaines.

Prenons un exemple pour clarifier ce bref rappel. Le verrou (Chemin, *Lire*, Implicite) posé sur une classe C signifie que la transaction est en train de lire toutes les instances d'au moins une classe strictement descendante de C ; les instances sont verrouillées implicitement en mode *Lire*. Le verrou (Frontière, *Écrire*, Explicite) indique que certaines instances sont modifiées dans toutes les classes descendantes de C, ainsi que C elle-même ; les instances effectivement modifiées doivent être individuellement verrouillées en mode *Écrire*.

Le protocole qui verrouille un sous-graphe se décompose en deux étapes. Premièrement, une transaction doit acquérir des verrous de frontière sur toutes les classes frontière du sous-graphe utilisé. Ensuite, elle doit acquérir des verrous de chemin sur des classes décrivant un chemin entre une classe quelconque du sous-graphe accédé et la classe racine du graphe d'héritage.

Un verrou de classe est un genre spécial de verrou de frontière où seule la classe racine est accédée. La notion de classes frontière d'un sous-graphe permet de ne verrouiller qu'un nombre suffisant de classes pour détecter toutes les incompatibilités. Par rapport à la classe racine d'un sous-graphe, les classes frontière sont celles du sous-graphe qui ont au moins une super-classe n'appartenant pas à ce sous-graphe. Par convention, la racine du graphe d'héritage est une classe frontière.

Deux verrous sont compatibles si, et seulement si, l'un des composants du triplet est compatible avec son homologue dans l'autre verrou. Les deux verrous de l'exemple précédent ne sont pas compatibles car ils sont incompatibles sur chaque composant, conformément aux relations données en tables III.1, III.2 et III.3. Mais, si nous remplaçons « Implicite » par « Explicite » dans le premier verrou, alors ils deviennent compatibles. En effet, les sous-graphes utilisés ne sont pas disjoints, pas plus que les modes d'accès compatibles, mais les instances utilisées doivent être explicitement verrouillées par chaque transaction. Par conséquent, la détection des incompatibilités est retardée jusqu'au moment de l'accès à des instances communes.

IV.5.1.2 Extension

Très simplement, l'extension consiste à changer le composant mode du triplet par un vecteur d'accès, c'est-à-dire un mode d'accès plus fin qu'une simple lecture ou écriture globale de

l'instance. Cependant, nous allons devoir proposer des versions adaptées des vecteurs d'accès pour les verrous de frontière et de chemin : *vecteurs d'accès de sous-graphe* et *vecteurs d'accès de chemin*. En fait, nous devons résoudre le problème de poser des verrous avec un vecteur d'accès qui contient des champs qui n'ont pas encore été déclarés, quand ce n'est pas la méthode elle-même qui est encore inconnue !

Exemple IV.9 *À partir de la figure IV.1, considérons une variable de type « list of Point ». Les instances se trouvant dans la liste ne sont pas nécessairement homogènes : il y aura des points, des cercles, des carrés, des rectangles ou des ellipses. Ces instances ne sont pas composées des mêmes champs et leurs méthodes de même nom ont des vecteurs d'accès incomparables. Ainsi* $VAT_{\text{CARRÉ},Afficher} = (Lire_X, Lire_Y, Lire_{Côté})$ *alors que* $VAT_{\text{CERCLE},Afficher} = (Lire_X, Lire_Y, Lire_{Rayon})$.

Le vecteur d'accès idéal associé à un sous-graphe serait celui qui collecterait tous les champs déclarés dans les classes descendantes.

Définition IV.12 (Vecteur d'accès complet à un sous-graphe) *Soient C une classe et M une méthode définie dans C, nous définissons le vecteur d'accès complet à un sous-graphe, $VACS_{C,M}$, comme suit :*

$$VACS_{C,M} = \bigvee_{C' \in DESCENDANTS(C)} VAT_{C',M}.$$

En pratique, nous allons avoir un problème avec les classes qui se situent près de la racine du graphe d'héritage si nous adoptons cette définition. Un vecteur d'accès complet à un sous-graphe sur la racine du graphe d'héritage décrira l'utilisation de quasiment tous les champs déclarés dans l'ensemble de la hiérarchie ! Nous devons limiter nos ambitions à des vecteurs d'accès plus praticables. Avec le modèle de données qui est le nôtre dans cette étude, chaque classe a en commun avec toute autre les champs déclarés dans leurs classes ancêtres communes.

Nous allons conserver le détail des informations sur les champs déclarés dans une classe commune C, mais compacter tous les champs qui sont déclarés dans des classes strictement descendantes de C sous un nom générique qui restera réservé au système, disons « $\downarrow$ ». Donnons un exemple avant la définition afin de la rendre plus lisible.

Exemple IV.10 *Avec la hiérarchie de la figure IV.1, le vecteur d'accès complet au sous-graphe de la méthode* POINT.*Déplacer aurait été :*

$$(Lire_X, Lire_Y, Lire_{Côté}, Lire_{\text{CARRÉ.}Angle}, Lire_{Côté2}, Lire_{Rayon}, Lire_{Rayon2}, Lire_{\text{ELLIPSE.}Angle}),$$

alors que le vecteur d'accès retenu sera plus réaliste :

$$(Lire_X, Lire_Y, Lire_\pi).$$

Définition IV.13 (Vecteur d'accès à un sous-graphe) *Soit C une classe, alors nous définissons le vecteur d'accès à un sous-graphe, $VAS_{C,M}$, de la façon suivante :*

$$VAS_{C,M} = VACS_{C,M}[CHAMPS(C)] \cup (m_{\downarrow})$$

avec :

$$m_{\downarrow} = \bigvee_{m' \in VACS_{C,M}][CHAMPS(C)[} m'$$

où $\downarrow$ est un nom de (pseudo) champ réservé au système.

Finalement, il nous faut fournir une autre forme de vecteurs d'accès pour les verrous de chemin. La raison est double. Premièrement, la méthode invoquée n'est certainement pas définie dans toutes les classes formant un chemin entre une classe du sous-graphe et la racine du graphe d'héritage. Deuxièmement, même si elle y est définie, son vecteur d'accès de sous-graphe est probablement plus restrictif que celui effectivement utilisé dans les sous-classes manipulées. L'utiliser directement limiterait donc inutilement le parallélisme possible.

En fait, un vecteur d'accès de chemin doit être calculé à partir du vecteur d'accès transitif ou de sous-graphe utilisé plus bas ; il ne dépend donc pas de la classe sur laquelle on pose le verrou, mais de la classe racine du sous-graphe (de la classe isolée) verrouillé(e).

Définition IV.14 (Vecteur d'accès de chemin) *Soit C une classe et M une méthode qui n'est pas nécessairement définie dans C, alors nous définissons le vecteur d'accès de chemin vis-à-vis d'une classe descendante C' où M est définie, $VAC_{C,M/C'}$, comme :*

$$VAC_{C,M/C'} = VAS_{C',M}[CHAMPS(C')] \cup (m_{\downarrow})$$

avec :

$$m_{\downarrow} = \bigvee_{m' \in VAS_{C',M}][CHAMPS(C)[} m'.$$

Cette définition est très proche de la précédente et peut être facilement calculée de proche en proche depuis la racine du sous-graphe jusqu'à la racine du graphe d'héritage. Dans chaque classe du chemin parcouru, certains champs sont « transférés » dans le champ générique $\downarrow$.

Dans la version étendue du protocole, le composant mode d'un verrou est maintenant un vecteur d'accès. À un verrou de classe, nous associons un vecteur d'accès transitif, à un verrou de frontière, un vecteur d'accès de sous-graphe, et à un verrou de chemin, un vecteur d'accès de chemin relatif à la classe racine du sous-graphe (ou à la classe isolée) utilisée par la transaction.

IV.5.1.3 Critique de cette extension

Une telle extension présente plus d'inconvénients que d'avantages. Les deux inconvénients sont la perte d'un parallélisme potentiel et la dégradation des performances à l'exécution.

perte de parallélisme potentiel Le champ générique $\downarrow$ est suffisant mais non nécessaire. Regroupant la valeur la plus restrictive des accès effectués sur un ensemble de champs, il assure que le vecteur d'accès de sous-graphe ou de chemin est correct, mais il ne pourra pas être optimal. Ainsi, $(Nul_X, Écrire_Y)$, $(Écrire_X, Nul_Y)$, $(Lire_X, Écrire_Y)$ et $(Écrire_X, Lire_Y)$ donnent-ils tous $(Écrire_\downarrow)$, d'où l'on perd la connaissance que les deux premiers vecteurs sont compatibles. Comme ce genre de problème fait parti de ceux que l'on veut supprimer (cf. section IV.3), il est assez mal venu de le conserver, même amoindrie, dans une solution.

dégradation des performances à l'exécution Les vecteurs d'accès de sous-graphe peuvent être calculés au préalable ; ils ne font que doubler l'espace nécessaire aux stockage des vecteurs d'accès. Il est plus difficile d'envisager que chaque classe conserve la liste des vecteurs d'accès de chemin (de l'ordre du carré du nombre de classes dans le cas le plus mauvais). Les calculer au moment de la pose des verrous pénalise les performances.

De toute façon, l'obtention d'un verrou comportant un vecteur d'accès sera forcément moins efficace que celui d'un verrou composé à partir d'un mode d'accès.

Non seulement la version de la section suivante évite ses inconvénients, mais elle est extrêmement plus simple.

IV.5.2 Le protocole le plus simple

La solution la plus simple, les implémenteurs ayant une prédilection pour ces dernières quand elles ne sont pas trop mauvaises, est de renoncer au verrouillage implicite du chapitre III et de [Garza & Kim 88] pour procéder comme [Cart & Ferrié 90], c'est-à-dire verrouiller toutes les classes d'un sous-graphe (ou une classe isolée) et plus aucune classe de chemin. Afin de ne pas dégrader les performances lors de l'acquisition de verrous, les vecteurs d'accès transitifs sont transformés en simples mode d'accès. Ce sont ces derniers qui seront utilisés dans le protocole. La troisième sous-section traite des modifications concurrentes dans les graphes de résolution des liaisons dynamiques.

IV.5.2.1 Des vecteurs d'accès aux modes d'accès

Nous avons nous-même objecté que les vecteurs d'accès sont des éléments qui demandent un temps de comparaison proportionnel au nombre de champs de la classe. Pour éliminer cet inconvénient, la transformation d'un vecteur d'accès en un mode d'accès est très simple.

	MA_{c_1,m_1}	MA_{c_1,m_2}	MA_{c_1,m_3}
MA_{c_1,m_1}	non	non	*oui*
MA_{c_1,m_2}	non	non	*oui*
MA_{c_1,m_3}	*oui*	*oui*	*oui*

TAB. IV.2 – Commutativité entre méthodes de la classe c_1

	MA_{c_2,m_1}	MA_{c_2,m_2}	MA_{c_2,m_3}	MA_{c_2,m_4}
MA_{c_2,m_1}	non	non	*oui*	*oui*
MA_{c_2,m_2}	non	non	*oui*	*oui*
MA_{c_2,m_3}	*oui*	*oui*	*oui*	*oui*
MA_{c_2,m_4}	*oui*	*oui*	*oui*	non

TAB. IV.3 – Commutativité entre méthodes de la classe c_2

Définition IV.15 *À chaque méthode M, dans chaque classe C, on associe un mode d'accès $MA_{C,M}$.*

La relation de compatibilité des modes d'accès se déduit de la relation de compatibilité des vecteurs d'accès transitifs donnée en définition IV.6 :

$$MA_{C,M}\mathcal{C}_{MA}MA_{C,M'} \leftrightarrow VAT_{C,M}\mathcal{C}VAT_{C,M'}$$

Ainsi, contrairement aux vecteurs d'accès de sous-graphe de la section IV.5.1.2, chaque mode d'accès est parfaitement identique à son vecteur d'accès associé quant aux accès concurrents qu'il autorise.

Une relation de commutativité sera ainsi créée pour chaque classe. L'exemple suivant nous donne celles associées aux classes de notre exemple principal.

Exemple IV.11 *À partir des vecteurs d'accès transitifs calculés dans l'exemple IV.8 et associés à la figure IV.2, nous obtenons les deux relations de commutativité données en Tables IV.2 et IV.3.*

IV.5.2.2 Le protocole pour accéder aux instances

Encore une fois, rappelons que nous nous en remettons aux protocole de verrouillage à deux phases dans sa version stricte [Eswaran et al. 76].

Les accès aux instances se décomposent en :

(i) accès à une instance isolée ;

(ii) accès, sinon à toutes, du moins à une majorité d'instances d'une classe ;

(iii) accès à quelques instances appartenant à un même sous-graphe ;

(iv) accès à toutes les instances générales d'une classe racine d'un sous-graphe.

Le verrouillage implicite au niveau des classes ne semble plus faisable, par conséquent, verrouiller une classe isolée ou verrouiller un sous-graphe ne différent essentiellement pas. Cependant, en ce qui concerne les instances, le verrouillage implicite est encore utile. Si une transaction accède à toutes les instances, propres ou générales, d'une classe, il est toujours intéressant de n'avoir à verrouiller que la classe, ou les classes du sous-graphe, en mode hiérarchique plutôt que chaque instance individuellement.

En conséquence, un verrou sur une instance sera toujours un mode d'accès. Mais, un verrou sur une classe ne sera plus un triplet (type, intention, mode) mais seulement un couple (intention, mode).

Nous allons illustrer les quatre accès possibles aux instances avec l'exemple de la figure IV.2.

accès à une instance Quand une transaction T_1 envoie le message m_1 à une instance propre de la classe c_1, le verrou MA_{c_1,m_1} est obtenu sur l'instance et le verrou (MA_{c_1,m_1}, Explicite) sur la classe c_1.

accès à toutes les instances propres d'une classes Si T_2 projette d'envoyer le message m_1 à toutes les instances propres de la classe c_1, aucun verrou n'aura à être demandé sur les instances, mais le verrou (MA_{c_1,m_1}, Implicite) devra être demandé sur la classe c_1. Comme le verrou détenu par T_1 est explicite alors que celui demandé par T_2 est implicite, la compatibilité des accès dépend des modes d'accès, qui sont incompatibles comme le montre la table IV.2. Donc, T_2 doit attendre jusqu'à la terminaison de T_1.

accès à quelques instances générales d'un sous-graphe Une autre transaction, T_3, envoie le message m_3 à certaines instances appartenant au domaine de c_1. Donc, les classes c_1 et c_2, ainsi que d'autres sous-classes de c_1 sont verrouillées avec $(MA_{c_?}, m_3, \text{Explicite})$. T_3 peut s'exécuter en concurrence avec T_1 et n'est pas bloquée derrière T_1 puisque le mode associé à la méthode m_1 est compatible avec ceux utilisés par T_1 et T_2 dans les deux tables IV.2 et IV.3 (et qu'on le suppose également compatible dans les autres classes descendantes de c_1).

accès à toutes les instances générales d'un sous-graphe Une dernière transaction, T_4, veut envoyer le message m_4 à toutes les instances générales de c_2. Le verrou $(MA_{c_?}, m_4, \text{Implicite})$ doit être obtenu sur toutes les classes descendantes de c_2. Aucune des transactions précédentes ne bloque T_4 qui peut donc s'exécuter en concurrence sur chaque instance générale de c_2 sans aucun contrôle supplémentaire.

Avec les seuls modes d'accès *Lire* et *Écrire*, soit T_1 et T_3 auraient pu s'exécuter concurremment car toutes les deux verrouillent explicitement les instances, soit T_1 et T_4 car elles n'accèdent à aucune instance commune.

Dans le schéma relationnel donné en section IV.3, T_1 verrouille un n-uplet de la relation c_1 en écriture ainsi que le n-uplet associé dans la relation c_2 (car f_1, la clef, a été modifiée). T_2 verrouille les deux relations en mode écriture. T_3 verrouille la relation c_1 en mode lecture. Enfin, T_4 verrouille la relation c_2 en mode écriture. Par conséquent, seuls T_3 et T_4 peuvent s'exécuter en concurrence. (Nous aurions pu avoir T_1 et T_3 si le champ modifié n'était pas la clef primaire de la relation c_1 et la clef étrangère de la relation c_2.)

Nous pouvons donc constater que les exécutions concurrentes autorisées sont *incomparables*. Cela provient du fait que les décompositions qu'engendrent, d'une part, l'héritage, et d'autre part, la contrainte de première forme normale, sont elles-mêmes incomparables.

Nous avons écrit, en section IV.4.2, que la décomposition sous première forme normale correspondait, du point de vue du contrôle de concurrence, à l'élaboration d'un vecteur d'accès grossier. L'utilisation des vecteurs d'accès permet donc profiter et d'étendre les exécutions concurrentes possibles dans les bases de données relationnelles. De plus, comme nous verrouillons dans un graphe d'héritage, nous profitons aussi des exécutions concurrentes permises dans celui-ci, c'est-à-dire d'une forme de verrouillage prédicatif (cf. section III.4.1). *Par conséquent, la technique des vecteurs d'accès transitif couvre toutes les exécutions concurrentes autorisées dans les bases de données relationnelles et par la proposition du chapitre précédent.*

Dans le chapitre précédent, nous avons discuté assez longuement des créations et destructions d'instances, ainsi que des modifications du graphe d'héritage. Le protocole de la section III.4.2.2 se simplifie puisque l'on n'a plus besoin de calculer classes frontière et classes de chemin. Nous n'allons ici traiter que les modifications concurrentes des méthodes d'une classe.

IV.5.2.3 Modifications des méthodes

La solution de la section III.4.3 concernant les modifications de méthodes entraînait un traitement exclusif. Nous allons apporter un peu plus de parallélisme.

Les modifications de méthodes et celles de champs de la classe sont liées par l'intermédiaire des vecteurs d'accès. *Supprimer* un champ va invalider plusieurs méthodes qui devront être modifiées. Modifier une méthode va non seulement changer son vecteur d'accès direct ainsi que son vecteur d'accès transitif, mais également tous les vecteurs d'accès transitifs des méthodes qui l'invoquent directement ou indirectement.

Afin d'offrir un parallélisme maximal entre méthodes, nous esquissons un protocole de verrouillage dans le graphe de résolution des liaisons dynamiques. Il utilise trois verrous s'appliquant sur les sommets de ce graphe.

Définition IV.16 *Nous introduisons trois modes d'accès aux sommets d'un graphe de résolution des liaisons dynamiques :* $MODES_{METHODES} = \{LVAT, MVAT, M\Gamma\}$. *Ces modes d'ac-*

	LVAT	*MVAT*	$M\Gamma$
LVAT	*oui*	non	*oui*
MVAT	non	non	non
$M\Gamma$	*oui*	non	non

TAB. IV.4 – Compatibilité des verrous sur les sommets des graphes de résolution des liaisons dynamiques

cès sont directement utilisés comme verrous, leur relation de compatibilité étant donnée en table IV.4.

Le protocole est le suivant :

1. Lorsqu'une méthode est invoquée par une transaction, le verrou *LVAT* (« Lecture du Vecteur d'Accès Transitif ») doit être obtenu sur le sommet associé à cette méthode dans le graphe de résolution des liaisons dynamiques.
2. Lorsqu'une méthode est modifiée, le verrou *MVAT* (« Modification du Vecteur d'Accès Transitif ») doit être posé sur le sommet associé à cette méthode ainsi que sur tous les sommets ancêtres dans le graphe de résolution des liaisons dynamiques puisque leurs vecteurs d'accès transitifs vont être modifiés. Cette phase se combine avec l'initialisation incrémentale de l'algorithme de calcul des vecteurs d'accès transitifs fourni en annexe B (cf. section B.4.2).
3. Enfin, si des méthodes anciennement invoquées ne le sont plus, ou si de nouvelles méthodes sont invoquées,[4] leurs sommets associés doivent être verrouillées avec $M\Gamma$ (« Modification des arcs du graphe de résolution des liaisons dynamiques »). ($M\Gamma$ joue exactement le même rôle que son homonyme en section III.4.2.2.)

Exemple IV.12 *En figure IV.4, une méthode est modifiée. Son sommet associé, représenté en grisé, est verrouillé avec le verrou MVAT, ainsi que toutes les méthodes qui, directement ou indirectement, l'invoque. Une méthode invoquée par l'ancienne version ne l'est plus par la nouvelle, et inversement, une nouvelle méthode est appelée (extrémité des arcs pointillés). Ces deux méthodes doivent être verrouillées avec $M\Gamma$ car la liste des méthodes qui l'invoque doit être mise à jour.*

Toutes les méthodes qui n'ont pas été verrouillées avec MVAT peuvent être utilisées par des transactions concurrentes car elles pourront acquérir le verrou LVAT.

Ce protocole doit être observé dans chaque classe où la méthode modifiée est utilisée. Il est, en grande partie, une transposition de celui de la section III.4.2.2. Il offre quelques avantages par rapport à une méthode où chaque sommet serait vu comme un objet distinct, verrouillé dans le mode requis lors de chaque accès.

[4] Il s'agit de la différence symétrique entre l'ancien et le nouvel ensemble des méthodes invoquées.

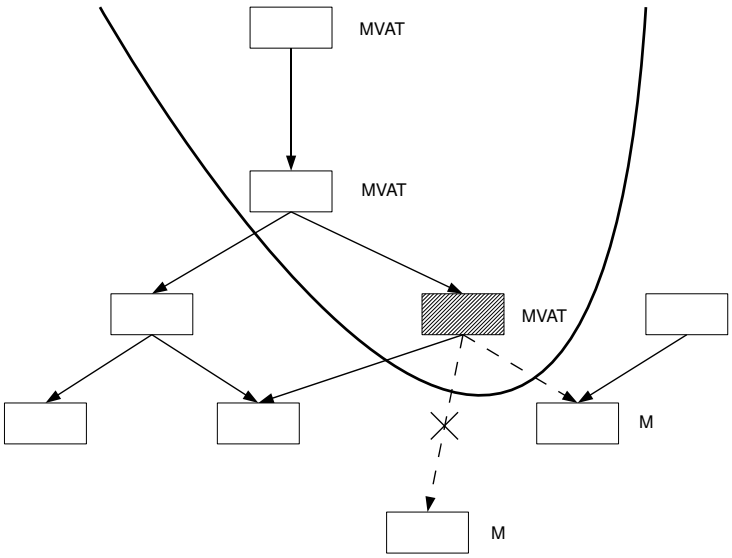


FIG. IV.4 – Verrouillage dans un graphe de résolution des liaisons dynamiques

Premièrement, pendant la phase de calcul des nouvelles valeurs de vecteurs d'accès transitifs, aucun verrou *LVAT* ne doit être demandé.

Ensuite, tous les verrous étant posés avant la phase de calcul, si une transaction concurrente qui tente de modifier une autre méthode est bloquée, elle n'a pour l'instant pas effectué de modifications. C'est ce qui assure que malgré que les verrous *LVAT* n'aient pas à être demandés pendant la phase de calcul, les lectures des vecteurs d'accès transitifs de sommets non verrouillés restent cohérentes.

Enfin, si les vecteurs d'accès transitifs ne sont pas utilisés directement mais servent à construire des modes d'accès, la table de compatibilité des modes d'accès constitue un objet qui serait verrouillé en mode exclusif. Grâce à ce protocole, nous savons que si la modification de cette table doit être atomique, il n'est point besoin de maintenir un verrou jusqu'à la fin de la transaction. La répercussion des modifications de certains vecteurs d'accès ne nécessite qu'une exclusion mutuelle.

Les cas d'insertion, de suppression ou de surcharge de méthodes se déduisent de celui de la modification. Dans une implémentation, il faudra gérer avec soin les sommets correspondants à des méthodes invalides. L'appel d'une méthode invalide entraînera le rejet de la transaction appelante, tout comme une division par zéro.

IV.6 AVANTAGES DES VECTEURS D'ACCÈS

Nous allons dans cette section présenter les différents avantages que l'exploitation des vecteurs d'accès transitifs procure. Nous ferons ensuite quelques propositions pour une implémentation efficace. Enfin, nous comparerons ce travail aux propositions qui s'en rapprochent.

IV.6.1 Avantages

Nous allons dresser un panorama des avantages que l'on est en droit d'attendre de l'exploitation des vecteurs d'accès. Il s'y trouve les objectifs annoncés dans l'introduction ainsi que quelques autres. Ce sont :

(i) un seul contrôle lors de l'accès à une instance ;

(ii) suppression d'une cause importante d'interblocages ;

(iii) parallélisme autorisé entre méthodes distinctes ;

(iv) tous les calculs sont effectués, efficacement, durant la phase de compilation ;

(v) les vecteurs d'accès permettent d'exploiter la commutativité entre méthodes aussi bien qu'avec les requêtes ;

(vi) les vecteurs d'accès peuvent servir à déclencher des « *triggers* » ou à contrôler des contraintes d'intégrité ;

(vii) la taille du journal, pour la reprise, est diminuée.

Détaillons maintenant ces différents points positifs.

moins de contrôles Les vecteurs d'accès transitifs résolvent les problèmes évoqués en section IV.3, en particulier le second : ils diminuent la charge de travail impartie au système pour contrôler les accès concurrents, en éliminant les demandes d'accès répétées dues aux messages auto-dirigés. Ce point est détaillé dans la section IV.6.2 ci-dessous.

moins d'interblocages La pose de verrous à base de vecteurs d'accès transitifs constitue une pré-déclaration de l'ensemble des variables d'instance lues et écrites par la plus pessimiste exécution de la méthode, celle-ci pouvant être impossible à cause des alternatives. Il y a ici un parallèle évident avec des propositions où les transactions pré-déclarent des sur-ensembles d'objets lus et écrits [Korth 82] [Eich 88] [Dasgupta & Kedem 90] [Ulusoy & Belford 91]

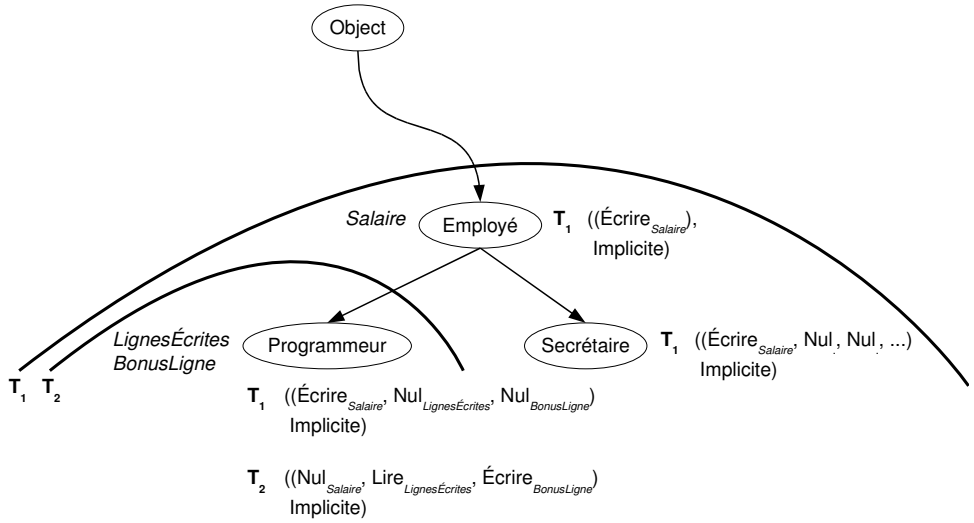


FIG. IV.5 – Deux transactions écrivains concurrentes dans un même sous-graphe

dans le but d'obtenir des exécutions non-2PL ou d'éviter les interblocages. Ici, nous évitons seulement les interblocages dus à l'exécution concurrente de méthodes sur une même instance.

plus de parallélisme Nous avons vu en section IV.5.2.2 que les vecteurs d'accès transitifs utilisés dans un graphe d'héritage permettaient de couvrir et d'étendre les exécutions concurrentes autorisés par les modes d'accès *Lire* et *Écrire* appliqués dans un graphe d'héritage et dans le schéma relationnel associé.

En pratique, cela se traduit par la possibilité d'exécuter plusieurs méthodes écrivains ou des méthodes écrivains et des méthodes lectrices sur des instances communes.

Exemple IV.13 *En figure IV.5, une transaction T_1 applique la méthode* AugmenterSalaire *à chaque employé (cf. exemple IV.2). Simultanément, T_2 applique le bonus annuel de productivité des programmeurs. Avec les seuls modes d'accès* Lire *et* Écrire*, les deux méthodes auraient été classées comme écrivains et l'exécution parallèle refusée car les verrous sur la classe* EMPLOYÉ *auraient été incompatibles.*

calculs peu coûteux Tous les calcul sont faits durant la phase de compilation des méthodes, donc il n'y a aucune surcharge à l'exécution (sauf en cas de modifications concurrentes). Les algorithmes sont tous linéaires en la taille des graphes de résolution des liaisons dynamiques.

requêtes L'encapsulation peut être violée par les langages de requête [Bancilhon et al. 89]. Si une analyse des requêtes est réalisée, alors des vecteurs d'accès pourraient être également construits avant l'évaluation et le parallélisme augmenté entre transactions soumettant des requêtes et transactions exécutant une application. Autrement, il suffit d'utiliser le vecteur positionnant tous les champs dans le mode *Lire*.

Cet avantage est perdu si l'on compacte les vecteurs pour obtenir des modes, comme cela est fait en section IV.5.2.1. Dans ce cas, toutes les méthodes écrivains seront incompatibles avec les requêtes. Notez qu'il faudra caractériser chaque méthode par un indicateur booléen indiquant s'il s'agit ou non d'une méthode lectrice.

« *triggers* » et contraintes d'intégrité La notion de « *trigger* », ainsi que celle de *règle*, est maintenant implémentée dans les bases de données : ODE [Andrews & Harris 87] [Agrawal & Gehani 89] [Gehani & Jagadish 91], ADAM [Diaz et al. 91], Starburst [Lohman et al. 91], Postgres [Stonebraker & Kemnitz 91]. Ce sont des réflexes qui, lorsqu'un événement se produit, déclenchent automatiquement une procédure. Ils peuvent servir plusieurs buts : autorisation d'accès, maintien de statistiques sur les accès, contrôle des contraintes d'intégrité [Beeri & Milo 91], gestion de vues, intégrité référentielle, protection, gestion de versions, déclenchements de transactions [Stonebraker & Kemnitz 91] ou de sous-transactions [Dayal et al. 90] [Dayal et al. 91].

Les deux utilisations les plus directes des vecteurs d'accès directs sont l'autorisation d'accès et le contrôle des contraintes d'intégrité.

En ce qui concerne les autorisations d'accès, le système pourra refuser l'invocation d'une méthode par un utilisateur si cette méthode manipule un champ qui doit lui être caché ou qu'il n'a pas le droit de modifier.

Chaque contrainte est composée d'une suite de conditions sur des champs de l'instance et de l'action à entreprendre si la condition est vérifiée. (Certaines contraintes sont des pré-conditions, d'autres des post-conditions.) Les vecteurs d'accès *directs* permettent de limiter le nombre de conditions qu'il faut tester.

diminution de la taille du journal Le problème du rejet des transactions et de la reprise sur panne est conceptuellement très simple : si l'on adopte la technique de journalisation des mises-à-jour, il s'agit de restaurer des images avant et/ou après de parties d'instances. Les parties d'instances à journaliser sont fournies par les vecteurs d'accès, c'est-à-dire par le positionnement d'un champ en mode *Écrire*. La taille du journal est réduite, ainsi que le temps de traitement des entrées/sorties. Nous renvoyons le lecteur à l'arbre des propositions de [Noe et al. 87] qui, implicitement, limite sa forme de commutativité à celle présentée ici, ainsi qu'à [Malta & Martinez 92b] [Malta 93].

Tous ces avantages ne pourront être atteint que si l'implémentation se révèle efficace. Voici quelques propositions.

IV.6.2 Propositions pour une implémentation

Dans une implémentation il faudra être capable de réaliser efficacement les contrôles de concurrence lors des accès aux données.

Nous avons vu que les informations définies en IV.5, IV.6 et IV.7, c'est-à-dire *VAD*, *MD* et *MP*, sont des informations propres à une méthode, c'est-à-dire un bloc physique de code. C'est données peuvent donc être stockées près du code source de la méthode. S'agissant d'informations qui ne sont utiles qu'au moment de la compilation, elles peuvent également être regroupées dans une structure de données séparée.

Ce qui nous importe bien davantage est de pouvoir retrouver très rapidement les vecteurs d'accès transitifs (ou le mode associé). Ces vecteurs-là ne sont pas liés à une méthode mais à un message. Il suffit donc de les stocker dans les différentes tables de résolution des liaisons dynamiques. Ainsi ces tables ne seront dorénavant plus constituées de couples (message, méthode), mais de triplets (message, méthode, vecteur d'accès transitif). Retrouver les vecteurs d'accès n'est donc pas plus long que de retrouver le code de la méthode associée à un message.

Enfin, pour éviter les contrôles répétés lors d'envois de messages auto-dirigés, nous reprenons l'idée d'interface à deux niveaux exposée dans [Malta & Martinez 91b] [Malta 93].

Dans une classe, il y a un certain nombre de méthodes qui font partie de l'interface publique de la classe, c'est-à-dire qui sont utilisables depuis l'extérieur, ainsi que des méthodes privées qui ne sont utilisées que par d'autres méthodes de la classe (ou parfois des classes descendantes) [Schaffert et al. 86]. Par exemple, sur l'ADT B-ARBRE, les méthodes publiques seront *Insérer* et *Supprimer*, tandis que les méthode *Partager* et *Fusionner* resteront privées car utilisées seulement par les deux premières.

Pour le contrôle de concurrence, chaque méthode de l'interface publique d'une classe sera décomposée en une version publique et une version privée. La version publique contiendra le code préliminaire qui effectue le contrôle de concurrence, puis un appel à sa version privée. Dans chaque classe, toute méthode donnera lieu à une méthode publique distincte car les vecteurs d'accès sont différents. La version privée de la méthode contiendra purement le code écrit par le programmeur, à quelques nuances près : les messages envoyés à des objets référencés par des variables d'instance invoqueront la version publique de la méthode associée ; en revanche, les messages envoyés à l'instance courante elle-même, « *self* », appellerons directement la version privée de la méthode.

Il sera alors intéressant de disposer de deux tables de résolution des liaisons dynamiques : une pour les méthodes publiques, où les vecteurs seront stockés, et une seconde, classique, pour les méthodes privés. (La table pour les méthodes privées sera également utilisée pour les objets locaux à une transactions, qui ne sont donc pas soumis au contrôle de concurrence.)

Message	Méthode	Vecteur d'accès transitif			
Déplacer	POINT.*Déplacer*	($\acute{E}crire_X$,	$\acute{E}crire_Y$,	$Lire_{Côté}$,	$Lire_{Angle}$)
Effacer	CARRÉ.*Effacer*	($Lire_X$,	$Lire_Y$,	$Lire_{Côté}$,	$Lire_{Angle}$)
Afficher	CARRÉ.*Afficher*	($Lire_X$,	$Lire_Y$,	$Lire_{Côté}$,	$Lire_{Angle}$)
Pivoter	CARRÉ.*Pivoter*	(Nul_X,	Nul_Y,	$Nul_{Côté}$,	$\acute{E}crire_{Angle}$)
Agrandir	CARRÉ.*Agrandir*	(Nul_X,	Nul_Y,	$\acute{E}crire_{Côté}$,	Nul_{Angle})

TAB. IV.5 – Table de résolution des liaisons dynamiques et des vecteurs d'accès

Exemple IV.14 *À partir de la hiérarchie de la figure IV.1, nous obtenons la table de résolution des liaisons dynamiques et des vecteurs d'accès pour la classe* CARRÉ *donnée en Table IV.5.*

Lorsque le message Déplacer *est envoyé à une instance de la classe* CARRÉ*, le corps de la méthode publique à appliquer est retrouvé ainsi que le vecteur d'accès transitif (ou le mode d'accès associé). Supposons qu'il s'agisse d'un accès à une instance isolée, alors le vecteur d'accès transitif est posé en tant que verrou sur l'instance puis utilisé pour verrouiller la classe propre de l'instance avec l'indicateur intentionnel positionné à Explicite. Si l'exécution n'est pas bloquée, la version privée de la méthode* POINT.*Déplacer commence à se dérouler. Le message* Effacer *est envoyé à « self ». S'agissant d'un appel auto-dirigé, une indirection dans la table jumelle des méthodes privées permet de lier le message à la version privée de la méthode* CARRÉ.*Effacer. Par conséquent, cet appel n'engendrera aucun contrôle.*

IV.6.3 Comparaison avec des travaux antérieurs

Historiquement, les vecteurs d'accès étaient déjà proposés par [Eswaran et al. 76] en conjonction avec le verrouillage prédicatif. Dans *System R*, le verrouillage prédicatif a été abandonné (car trop coûteux en temps, et remplacé par le verrouillage hiérarchique [Gray 78]) et les vecteurs d'accès seuls n'ont plus été retenus. Quelques raisons de cet abandon peuvent être que (1) il est coûteux de traiter chaque requête SQL, interprétativement, afin de construire les vecteurs d'accès associés à la requête sur chaque relation mise en jeu, (2) il est également coûteux, en temps, de verrouiller avec des vecteurs d'accès qui ont de longueurs variant avec celles des relations utilisées, et (3), comme nous l'avons vu en sections 3 et 4.2, décomposer en première forme normale fournit une forme grossière de vecteur d'accès.

Nous avons vu en section IV.3 que nous devions utiliser les vecteurs d'accès (ou toute autre forme de commutativité), autrement certaines exécutions concurrentes autorisées dans les bases de données relationnelles ne seraient pas acceptées dans les bases de données à objets. Grâce à l'encapsulation des données *et* des méthodes au sein d'une classe, les vecteurs d'accès sont calculés à la compilation et non durant l'utilisation de la base. Enfin, les vecteurs d'accès sont traduits en modes d'accès classiques [Korth 83], ce qui n'entraîne donc aucune surcharge à l'exécution par rapport à la simple compatibilité de *Lire* et *Écrire*. Donc, aucun des inconvé-

nients évoqués dans le paragraphe précédent ne demeure avec la technique des vecteurs d'accès dans les bases de données à objets. Il est même étonnant de constater que verrouillage prédicatif et vecteurs d'accès, qui avaient été abandonnés dans *System R*, réapparaissent dans notre proposition.

Nous avons déjà effectué une comparaison dans le chapitre III avec les méthodes de [Garza & Kim 88] et [Cart & Ferrié 90]. [Agrawal & El Abbadi 92] propose une méthode qui est proche de celle que nous préconisons ici. Nous n'allons présenter que l'aspect qui s'apparente à notre proposition, puis les inconvénients et avantages de cette proposition.

La méthode se ramène *grosso modo* à utiliser dans chaque classe deux ADT ENSEMBLE : un pour les méthodes, un second pour les champs (remarquez la correspondance immédiate avec la définition IV.1). Verrouiller un sous-graphe lors de l'envoi d'un message M à des instances de la classe C, consiste à demander le « verrou » *utiliser*(M) sur la classe C et sur toutes ses classes descendantes. Ensuite, des « verrous » *utiliser*(ch) sont demandés sur les mêmes classes pour chaque champ ch qui est manipulé par la méthode M. Quand tous les verrous ont été obtenus, la méthode peut s'exécuter.

inconvénients Visiblement, ce protocole présente quelques inconvénients :

(i) un contrôle est demandé à chaque envoi de message, y compris sur l'instance courante ;

(ii) chaque champ est verrouillé successivement, ce qui est plus long encore que la pose d'un verrou composé d'un vecteur d'accès (de plus, avec les appels auto-dirigés, un même champ peut être verrouillé plusieurs fois) ;

(iii) le problème des escalades de verrou ne se pose plus puisque on ne précise pas si les champs sont lus ou écrits, d'où des transactions lisant un même champ sont inutilement bloquées.

avantages En revanche, les modifications dans le graphe d'héritage sont extrêmement simplifiées. L'insertion ou la suppression de méthodes, ainsi que les deux mêmes opérations sur les champs, se contentent de verrouiller les classes avec les appels *supprimer*(M), *insérer*(M), ou *supprimer*(ch), *insérer*(ch).

Tout comme nous l'avions souligné dans le chapitre III, il est plus simple de procéder à des calculs dynamiques lorsque les modifications concurrentes du graphe d'héritage sont autorisées que de vouloir maintenir à jour, incrémentalement et concurremment, des structures de données globales. C'était le cas des classes frontière et de chemin dans le chapitre précédent ; c'est, en section IV.4.2, le cas des graphes de résolution des liaisons dynamiques, des vecteurs d'accès transitifs, de la table de compatibilité des modes d'accès associés et de l'indicateur d'invalidité.

Le choix entre cette méthode et la nôtre se déduit du taux de modifications. Si celui-ci est élevé, il est certain que la méthode de [Agrawal & El Abbadi 92] doit être adoptée. Si ce sont

les transactions non modificatrices qui l'emportent, alors il est justifié de traiter plus difficilement les mises-à-jour puisque le prix de cette complexité sera remboursé par de meilleures performances à l'exécution. En tout état de cause, la méthode de [Agrawal & El Abbadi 92] est parfaitement adaptée à un environnement interprété, ce qui est le cas d'ORION dont le langage de définition et de manipulation est Lisp [Kim 90]. La nôtre semble plus appropriée à un environnement compilé, comme O_2 et C.

IV.7 CONCLUSION

Dans ce chapitre, nous avons levé une limitation présente dans le chapitre précédent : différencier les modes d'accès aux instances en plus de deux catégories grossières, lecteurs et écrivains. Nous avons souligner que cela est indispensable si nous voulons autoriser certaines exécutions concurrentes permises dans les bases de données relationnelles.

Nous avons exposé une technique de détermination automatique d'une forme simplifiée de commutativité. Cette commutativité est automatiquement extraite du code source des méthodes par le compilateur sous forme de vecteurs d'accès, une version affinée des modes d'accès, tenant compte des particularités des systèmes à objets : encapsulation, héritage, surcharge et liaison dynamique. En particulier, nous fournissons un algorithme linéaire permettant de calculer les vecteurs d'accès transitifs, c'est-à-dire ceux qui ne se limitent pas à l'analyse du code direct d'une méthode, mais prennent en compte les codes de toutes les méthodes qui pourront être éventuellement exécutées par envois de messages imbriqués sur l'instance courante.

Les vecteurs d'accès transitifs résolvent les quatre problèmes soulignés en section IV.3. Tout d'abord, lors de l'envoi d'un message à une instance, il n'y a qu'un seul contrôle. Les envois de messages imbriqués, dus à la technique de programmation qui consiste à factoriser autant que possible le code, n'entraînent plus de surcharge. De plus, les vecteurs d'accès transitifs éliminent un grand risque d'interblocages, dû à l'appel d'une méthode écrivain par une méthode lectrice, ce qui demande une escalade de verrou. Enfin, le granule de verrouillage effectif étant les variables d'instance, ceci permet de reconnaître comme compatibles de méthodes qui auraient été exclusives au niveau de l'instance.

Tout ce travail se faisant en phase de compilation, puis les vecteurs d'accès transitifs étant compactés en modes d'accès classiques, les performances à l'exécution sont les mêmes que celles obtenues par l'utilisation des seuls modes *Lire* et *Écrire*.

Dans la seconde partie de ce chapitre, le verrouillage dans un graphe d'héritage a été revu pour incorporer les vecteurs d'accès. Comme les vecteurs d'accès sont spécifiques à une méthode dans une classe, alors que les modes d'accès étaient applicables dans n'importe quelle classe, nous avons dû renoncer au protocole du chapitre III qui avait l'intérêt de minimiser le nombre de verrous à obtenir. C'est le protocole qui consiste à verrouiller explicitement toutes les classes descendantes d'une classe racine donnée qui a été retenu. La première version est tout de même présentée afin de mettre en évidence deux inconvénient qui sont absents du protocole le plus simple. En cela, le choix « arbitraire » (*sic*) fait pour ORION [Kim et al. 90] (cf. section III.5.2) est justifié *a posteriori*.

Ce qui rend cette proposition attrayante est que l'ensemble de la technique est « facilement » implémentable. Ce point est essentiel dans le cadre des bases de données à objets où les méthodes sont appelées à être régulièrement modifiées, ajoutées ou supprimées. La totalité des calculs se fait pendant la phase de compilation des méthodes, et l'exécution peut être rendue efficace. Les vecteurs trouvent naturellement leur place dans la table de résolution des liaisons dynamiques.

Le seul problème avec les vecteurs d'accès transitifs (et les graphes de résolution des liaisons dynamiques) et qu'il s'agit d'informations globales à plusieurs méthodes et plusieurs classes. Maintenir incrémentalement et surtout concurremment de telles structures est la seule partie un peu difficile. Cela en vaut la peine quand les modifications ne sont pas excessivement fréquentes. Dans le cas contraire, l'approche « interprétative » de [Agrawal & El Abbadi 92] est sans doute meilleure, bien que pénalisant les performances. Une troisième solution, qui a notre préférence, tout du moins passagère, est de réaliser les modifications séparément, sur une copie des données. Lorsque les modifications de l'application sont (supposées car testées) correctes, alors elles sont répercutées sur la version de la base manipulée par les utilisateurs. Cette troisième solution allie l'avantage d'un traitement simplifié des modifications concurrentes du graphe d'héritage (cf. chapitre III) à une plus grande sécurité pendant les modifications vis-à-vis de l'intégrité des données.

Un avantage qui n'a pas été rapporté dans ce chapitre est que deux vecteurs d'accès compatibles permettent aux méthodes de s'exécuter sans aucun contrôle, même pas l'atomicité des opérations. Autrement dit, même si les deux méthodes modifient l'instance commune, elles peuvent se dérouler vraiment en parallèle. Aucune des méthodes de contrôle de concurrence et de reprise vues dans le chapitre I ne savent tirer profit de ce cette propriété, ce qui ouvre une voie de recherche.

Enfin, les vecteurs d'accès peuvent servir à d'autres tâches que le contrôle de concurrence, en particulier les autorisation d'accès, le contrôle des contraintes d'intégrité et, bien sûr, la reprise [Malta & Martinez 92b] [Malta 93].

Conclusion

RÉSUMÉ ET CONCLUSIONS DES TRAVAUX

Nos travaux ont porté sur le problème du contrôle de concurrence et de la reprise dans le cadre des bases de données à objets. Nous avons fait un bref rappel des résultats sur lesquels nous nous appuyons dans le chapitre premier. Toutefois, nous avons fait une réserve sur l'utilisation de ces résultats dans le chapitre premier. En effet, la relation de composition entre objets n'a jamais été prise en compte. Fort heureusement, dans nos travaux, et bien que n'ayons pas eu l'occasion de le souligner, ceci n'a pas de répercussions fâcheuses.

Nous nous sommes tout d'abord intéressés aux opérations typées, indépendamment du domaine des bases de données. Dans le chapitre II, nous avons vu que la commutativité, critère principal de contrôle des accès concurrents à des données partagées, est sujette à un phénomène de convergence qui étend le comportement du critère plus ancien de compatibilité. *Un accès en écriture est exclusif avec la compatibilité ; les opérations exclusives ne sont pas éliminées par la commutativité. Les accès en lecture ne permettent pas de modifier la valeur de l'objet ; les opérations commutatives ne permettent pas d'affaiblir le prédicat décrivant l'ensemble auquel appartient la valeur initiale de l'objet.*

Ces limites étant posées, et après un bref panorama sur les techniques employées pour les contourner, nous avons recommandé de s'en remettre à trois grands types d'objets. Il s'agit de l'ADT ENSEMBLE, où l'indépendance des éléments permet de circonscrire les limites de la commutativité à chaque élément, l'ADT COMPTEUR qui offre un degré de commutativité très élevé dès qu'il est programmé de manière appropriée, et l'exploitation de la simple compatibilité, éventuellement étendue aux champs, pour les objets structurés sous forme de n-uplet, les plus nombreux.

Dans les chapitres III et IV, nous nous sommes intéressés aux contrôle de concurrence dans les bases de données à objets. Dans le chapitre III, après avoir présenté les caractéristiques minimales d'un système à objets, nous avons détaillé l'ensemble des accès envisageables aux données : accès à une ou plusieurs classes, accès simultanés ou indépendants à leurs instances. Un premier protocole à base de verrouillage à deux phases strict a été proposé. Comparé à la proposition de [Garza & Kim 88] pour ORION, il s'avère en être une quasi généralisation (un seul mode d'accès, hybride, n'est pas réalisable dans notre proposition). Malgré cela, le gain effectif de parallélisme est minime si l'on n'étend pas considérablement le nombre de modes d'accès. Ce protocole sera étendu une première fois afin de prendre en compte les modifica-

tions concurrentes du graphe d'héritage. À la fin du chapitre III, des problèmes de parallélisme subsistaient donc.

Dans le chapitre IV, nous avons exposé une technique de détermination automatique de vecteurs d'accès, une version affinée des modes d'accès, tenant compte des particularités des systèmes à objets : encapsulation, héritage, surcharge et liaison dynamique. Les vecteurs d'accès transitifs introduisent une forme particulière de commutativité et éliminent quatre problèmes que la littérature n'avait jusqu'alors pas évoquée. Le problème le plus important est que les bases de données relationnelles offrent une forme de parallélisme que la proposition du chapitre III, ainsi que toutes celles qui ne considéreraient que les modes d'accès *Lire* et *Écrire* sur les instances, ne permet pas d'atteindre. Réciproquement, les protocoles à base de verrouillage dans les graphes d'héritage offre un parallélisme absent des bases de données relationnelles. Un troisième et dernier protocole est alors proposé. Il consiste à verrouiller toutes les classes descendantes d'une classe racine donnée avec le vecteur d'accès le plus approprié. Le choix « arbitraire » (*sic*) fait pour ORION [Kim et al. 90] se voit ainsi justifié. *Cette dernière proposition offre plus de concurrence que les propositions précédentes dans les graphes d'héritage et les schémas de données relationnels.* Les modifications, aussi bien de la définition d'une classe, que des méthodes, ou encore du graphe d'héritage, viennent toujours considérablement compliquer le protocole (à moins d'adopter la méthode de [Agrawal & El Abbadi 92], bien adapté aux environnements interprétés) et surtout pénaliser les performances. Cette considération justifie la différenciation faite dans O_2 entre un mode développement et un mode application.

Ce que l'on peut retenir de ces travaux est qu'il est indispensable de rechercher des formes de commutativité, mais que ces dernières doivent rester raisonnablement simples. Le cadre des bases de données à objets nous a permis de mettre en pratique cette idée et surtout de montrer qu'elle se révèle payante.

PERSPECTIVES

À l'issue de ces travaux, les perspectives de recherches futures sont nombreuses. La progression des chapitres amène tout naturellement à l'étape de l'implémentation. Mais, disséminées au long des pages, ont surgi quelques questions dont nous retenons les suivantes comme centres d'intérêt immédiats : caractérisation des exécutions avec le critère de recouvrabilité relative, exécutions multi-niveaux assouplies pour tenir compte des liens de composition entre objets, enfin, étude des transactions coopératives dans le cadre des modifications concurrentes dans le schéma de la base de données, graphe d'héritage et méthodes. Une toute dernière question, la plus générale, est : « Jusqu'à quel point peut-on adapter les méthodes aux modèles de données et de traitement ? »

limites de la recouvrabilité relative Nous avons montré les limites de la commutativité sur les types de données abstraits. Nous avons également souligné le fait que notre formalisme ne prenait pas en compte les restrictions commutatives de fonctions non commutatives. Ceci resterait donc à faire. Toutefois, il nous semble que les définitions de la commutativité en arrière, puis de la commutativité en arrière à droite introduites par [Weihl 88] [Weihl 89b] n'ont d'autre but que d'étendre le parallélisme sur le type de données abstrait très particulier qu'est un compteur muni des opérations d'incrémentation et de décrémentation (ainsi que ses variantes, comme l'ADT COMPTEENBANQUE du chapitre I).

Il nous semble plus intéressant de traiter directement des limites de la recouvrabilité relative, qui couvrent celles de la commutativité en arrière à droite. D'autre part, nous avons montré dans le chapitre I que, en pratique, la recouvrabilité relative pouvait être moins permissive que la commutativité en avant. Il serait intéressant d'envisager l'emploi de la recouvrabilité relative avec mises-à-jour différées, c'est-à-dire de recouvrabilité relative *en avant*.

transactions multi-niveaux assouplies Dans le chapitre I, nous avons présenté deux techniques de reprise. Nous avons cependant montré (cf. exemple I.10) que des hypothèses implicites étaient faites par leurs auteurs sur la forme des traitements. Dans les deux cas, une opération est supposée s'exécuter complètement sur un seul objet. Nous avons vu que la relation de composition entre objets, essentielle dans la définition des classes, ne s'accommode pas de cette vision des traitements : une méthode en cours d'exécution peut en invoquer une autre, sur une instance référencée *via* un lien de composition. Les transactions multi-niveaux n'apportent

qu'une solution partielle : elles autorisent les exécutions qui mettent en jeu plusieurs pages physiques pour un seul objet logique, mais non plusieurs pages pour plusieurs objets logiques.

Il nous semble nécessaire de lever cette hypothèse qui veut que les opérations d'un certain niveau d'abstraction soient toujours déroulées exclusivement en opérations d'un niveau inférieur : les opérations d'un niveau donné peuvent, au moins, se décliner en autres opérations du même niveau sur des instances différentes.

transactions coopératives D'une part, la résolution du problème des modifications concurrentes du graphe d'héritage d'une base de donnés à objets a surtout montré que la sérialisabilité aboutissait à des traitements hautement exclusifs. D'autre part, les modifications peuvent être menés en parallèle, sur une copie (virtuellement ?) distincte de la base. Le domaine de la coopération entre transactions a été essentiellement abordé dans le cadre des systèmes de CAO. Les modifications simultanées d'un graphe d'héritage et des méthodes offre un cadre de recherche sur les transactions coopératives qui reste interne au modèle : il s'agit d'une forme de CASE (pour « *Computer-Aided Software Engineering* ») coopératif ou comment développer une application au sein d'équipes de programmation.

quels modèles ? Nous avons mis en évidence, dans le chapitre IV, l'apport d'un contrôle qui s'appuie, d'une part, sur graphe d'héritage et, d'autre part, sur une connaissance syntaxique de la commutativité des méthodes : elle couvre la forme de verrouillage prédicatif offert par l'utilisation du graphe d'héritage, et étend le verrouillage des parties d'instances que la première forme normale impose dans les bases de données relationnelles. Par conséquent, la prise en compte du modèle de données et du modèle de traitement a visiblement eu un intérêt.

Les extensions au modèle de base de données à objets que nous avons citées sont nombreuses : objets composites [Kim et al. 87] [Kim et al. 89], versions, relations [Rumbaugh 87], méta-classes [Roffé et al. 90], multi-instanciation [Fishman et al. 87] [Rieu et al. 91], classes virtuelles [Abiteboul & Bonner 91]. Jusqu'où pourra-t-on apporter une solution spécifique qui ne devienne pas inextricable ? Devrait-on alors se contenter du contrôle explicite sur chaque objet logique ?

Annexe A

Notations utilisées

A.1 SYMBOLES MATHÉMATIQUES

$\forall$	quantificateur universel
$\exists$	quantificateur existenciel
$\leftrightarrow$	équivalence
$\rightarrow$	implication logique
$\wedge$	et logique
$\vee$	ou logique ; opérateur sup dans un treillis
$\neg$	non logique
$\perp$	valeur indéterminée
$\{\ldots\}$	ensemble
$\emptyset$	ensemble vide
$\mathcal{P}(X)$	ensemble des parties de l'ensemble X
$\complement_X Y$	complémentaire de Y dans X
$\in$	appartenance
$\notin$	non appartenance
$\subseteq$	inclusion large
$\subset$	inclusion stricte
$\cap$	intersection
$\cup$	union
$\setminus$	différence d'ensembles
$\times$	produit cartésien
$\rightarrow$	application
f	fonction
$\circ$	composition de fonctions
$f[X]$	restriction de f en X
$f]X[$	restriction de f dans le complémentaire de X
$=$	égalité
$\neq$	différence
$\leq$	inférieur ou égal
$<$	strictement inférieur
$\prec_X$	relation d'ordre dans l'ensemble X
$\geq$	supérieur ou égal
$>$	strictement supérieur

A.2 NOTATIONS SUR LES GRAPHES

En ce qui concerne les graphes, nous utilisons les notations de [Berge 67].

$G = (X, U)$ est un *graphe orienté* formé de *sommets* composant l'ensemble X et d'*arcs*, couples d'éléments de X, composant U.

Nous utilisons plus souvent la notation (X, Γ) où Γ est l'union des fonctions Γ définies pour chaque sommet de X. $\Gamma(x)$ donne l'ensemble des sommets *successeurs* de x. $\Gamma^{-1}(x)$ est la fonction réciproque qui donne l'ensemble des *prédécesseurs* d'un sommet.

U et Γ sont équivalents : $y \in \Gamma(x) \leftrightarrow (x, y) \in U$.

Si $U' \subseteq U$, alors (X, U') est le *graphe partiel* de (X, U). Si $X' \subseteq X$, alors $(X', U[X'])$ est le *sous-graphe* (X, U) induit par les sommets de X', obtenu en enlevant les arcs entrant ou sortant des sommets de $X \setminus X'$. $(X', U'[X'])$ est alors appelé un *sous-graphe partiel* de (X, U).

$\lambda = (t_0, \ldots, t_n)$ est un *chemin* de (X, U) si $\forall i : 1 \leq i \leq n, t_i \in X$ et $\forall i : 1 \leq i < n, (t_i, t_{i+1}) \in U$. λ est un *circuit* de (X, U) si c'est un chemin, que $t_0 = t_n$ et que $(t_n, t_0) \in U$. La longueur de λ est n. Par convention, nous poserons que $\lambda = (x)$ est un chemin et un circuit de longueur zéro.

Un sommet x est un *puits* s'il ne possède aucun *arc sortant*, c'est-à-dire si $\Gamma(x) = \emptyset$. Réciproquement, x est une *source* s'il ne possède aucun *arc entrant*, c'est-à-dire si $\Gamma^{-1}(x) = \emptyset$.

Un sommet x est une *racine* si, pour tout sommet y, il existe un chemin entre x et y.

Un graphe est *fortement connexe* si tous ses sommets sont des racines du graphe. Comme cette propriété est rarement vérifiée pour un graphe dans sa totalité, on définit les *composantes fortement connexes* d'un graphe comme les sous-graphe partiels maximaux pour cette condition.

La notion de *clique* est plus forte que celle de graphe fortement connexe. Une clique est un graphe fortement connexe dont tous les sommets sont liés par des chemins de longueur un, c'est-à-dire un arc. On restreint également cette propriété à des sous-graphes.

$\Gamma^*(x)$ est la *fermeture réflexo-transitive* de $\Gamma(x)$, l'ensemble des sommets descendants de x, c'est-à-dire tous les sommets y pour lesquels il existe un chemin entre x et y : $G^*(x) = \{y \in X | \lambda = (x, \ldots, y)$ est un chemin dans $(X, \Gamma)\}$. On définit de même $\Gamma^{-1*}(x)$. La première fermeture définit les *descendants* d'un sommet, alors que la seconde représente ses *ancêtres*.

Annexe B

Calcul des vecteurs d'accès transitifs

L'algorithme de calcul des vecteurs d'accès transitifs est basé sur celui des composantes fortement connexes d'un graphe orienté. Il existe plusieurs algorithmes pour calculer ces composantes. Certains sont inefficaces, en $O(|X|^3)$ pour celui de [Berztiss 75],[1] d'autres ne permettent pas des calculs incrémentaux [Aho et al. 83]. Nous utilisons celui de [Tarjan 72].

[1] Il consiste à calculer la fermeture réflexo-transitive du graphe par l'algorithme de Roy-Warshall, en $O(|X|^3)$. On ne conserve alors que les arcs (x_1, x_2) tels que (x_1, x_2) appartienne au graphe et (x_2, x_1) appartienne à la fermeture, en $O(|X|^2)$. Le graphe obtenu ne contient que les arcs appartenant à un circuit. Remarquez que les composantes fortement connexes ne sont pas directement calculées.

B.1 L'ALGORITHME

L'algorithme est tout simplement un parcours total du graphe de résolution des liaisons dynamiques, muni d'une boucle interne qui est une descente en profondeur d'abord. C'est cette dernière qui est le cœur de l'algorithme.

Les vecteurs d'accès transitifs sont calculés de haut en bas, depuis les puits (s'il en existe !), où l'égalité *VAT*(x) = *VAD*(x) est évidente, jusqu'aux racines (s'il y en a !). Chaque fois que le calcul d'un sommet du graphe est achevé, il devient « calculé » et la valeur de son *VAT* est transmise à son éventuel prédécesseur dans l'ordre de la descente.

Les méthodes pouvant s'appeler récursivement par l'intermédiaire d'autres méthodes, ce graphe peut contenir des circuits. En faisant l'observation triviale que les *VAT* de tous les sommets se trouvant sur un circuit commun sont nécessairement égaux puisque leurs Γ^* (l'ensemble des méthodes qu'ils peuvent éventuellement invoquer) sont identiques, nous pouvons encore calculer efficacement les valeurs des vecteurs d'accès transitifs de tout le graphe avec cet algorithme. La variation de parcours de graphe à utiliser est celle de [Tarjan 72] permettant de déterminer les composantes fortement connexes d'un graphe orienté. Grâce à la propriété III.1, nous pouvons calculer les dépendances cycliques (idempotence) dans n'importe quel ordre (commutativité et associativité).

L'algorithme B.1 est muni d'une initialisation non incrémentale. La version itérative de la descente est de [Bordat & Cogis 86].

Nous supposons que nous avons à notre disposition un processeur ensembliste et une instruction de choix non déterministe, « $x \leftarrow$ **choix**$(z \in Y | P)$ » qui affecte à la variable x une une valeur z *quelconque* appartenant à un ensemble Y *non vide* et vérifiant la propriété P.

L'algorithme utilise deux variables locales de type sommet : x et y. Il utilise également trois variables de type ensemble de sommets : atteint, traité et calculé. Les sommets du graphes sont valués par les deux fonctions *VAD* et *VAT* ainsi que par des fonctions qui ne sont utiles qu'au calcul des *VAT* : racine, descente, parcouru et circuit.

La preuve de cet algorithme n'est pas simple. Il faut démontrer que la totalité du graphe est prise en compte (théorème B.7), que ce sont bien les composantes fortement connexes qui sont isolées (théorème B.3), qu'elles le sont dans un ordre qui n'est pas quelconque (corollaire B.5), enfin que les vecteurs d'accès transitifs sont eux aussi correctement calculés (théorème B.9). Nous proposons une preuve académique utilisant les invariants de boucle.

Algorithme B.1 Calcul des vecteurs d'accès transitifs

```
atteint ← ∅ ;
traité ← ∅ ;
calculé ← ∅ ;
compteur_descente ← 0 ;
tant que X \ atteint ≠ ∅ faire /* parcours total du graphe (X, Γ) */
  x ← choix(z ∈ X \ atteint) ;
  atteint ← atteint ∪ {x} ;
  parcouru(x) ← ∅ ;
  VAT(x) ← VAD(x) ;
  circuit(x) ← {x} ;
  compteur_descente ← compteur_descente + 1 ;
  descente(x) ← compteur_descente ;
  racine(x) ← compteur_descente ;
  tant que atteint \ traité ≠ ∅ faire /* descente en profondeur */
    x ← choix(z ∈ atteint \ traité|descente(z) maximal) ;
    si Γ(x) \ parcouru(x) = ∅
    alors traité ← traité ∪ {x} ;
          si racine(x) = descente(x)
          alors calculé ← calculé ∪ circuit(x) ;
                VAT ← ∅ ;
                pour y ∈ circuit(x) faire
                  VAT ← VAT ∨ VAT(y) ;
                pour y ∈ circuit(x) faire
                  VAT(y) ← VAT ;
                si atteint \ traité ≠ ∅
                alors y ← choix(z ∈ atteint \ traité|descente(z) maximal) ;
                      VAT(y) ← VAT(y) ∨ VAT ;
                sinon y ← choix(z ∈ atteint \ traité|descente(z) maximal) ;
                      racine(y) ← min{racine(y), racine(x)} ;
                      circuit(y) ← circuit(y) ∪ circuit(x) ;
          sinon y ← choix(z ∈ Γ(x) \ parcouru(x)) ;
                parcouru(x) ← parcouru(x) ∪ {y} ;
                si y ∈ calculé
                alors VAT(x) ← VAT(x) ∨ VAT(y) ;
                sinon si y ∈ atteint
                alors racine(x) ← min{racine(x), racine(y)} ;
                sinon atteint ← atteint ∪ {y} ;
                      parcouru(y) ← ∅ ;
                      VAT(y) ← VAD(y) ;
                      circuit(y) ← {y} ;
                      compteur_descente ← compteur_descente + 1 ;
                      descente(y) ← compteur_descente ;
                      racine(y) ← compteur_descente ;
```

B.2 PREUVE DE LA BOUCLE INTERNE

Nous allons commencer par prouver la correction partielle puis totale de la boucle interne de cet algorithme, c'est-à-dire la descente en profondeur d'abord. Nous calculerons ensuite sa complexité.

B.2.1 Preuve de correction partielle

L'algorithme de descente en profondeur d'abord construit une arborescence dans le graphe initial. Cette arborescence est un sous-graphe partiel du graphe initial, contenant tous les nœuds descendants d'un nœud racine x.

Un exemple d'exécution en cours est donné en figure B.1, ce qui permet d'illustrer la plupart des invariants de la boucle. Les arcs de l'arborescence recouvrante sont tracés en gras, les arcs de retour (x_8, x_7), gauche-droite (x_3, x_2) et de transitivité sont tracés en trait simple, enfin, les arcs pointillé $(x_6, x_\perp)$ sont ceux qui n'ont pas encore été parcourus.

Propriété B.1 *Les propositions suivantes sont des invariants de boucle :*

A. $\mathit{traité} \subseteq \mathit{atteint}$

B. $\forall x \in \mathit{atteint}$,
$\mathit{parcouru}(x) \subseteq \Gamma(x)$;

C. $\forall x \in \mathit{traité}$,
$\mathit{parcouru}(x) = \Gamma(x)$;

D. $\forall x, y \in \mathit{atteint} \setminus \mathit{traité}$,
$\mathit{descente}(x) < \mathit{descente}(y)$
$\rightarrow$
$(x, \ldots, y)$ *est un chemin dans* $(\mathit{atteint} \setminus \mathit{traité}, \mathit{parcouru})$;

E. $\forall x \in \mathit{atteint} \setminus \mathit{traité}$,
$\mathit{circuit}(x) \setminus \{x\} \subseteq \mathit{traité} \setminus \mathit{calculé}$;

F. $\forall x \in \mathit{atteint} \setminus \mathit{calculé}$,
$\exists! y \in \mathit{atteint} \setminus \mathit{traité}$,
$x \in \mathit{circuit}(y)$;

G. $\forall x, y \in \textit{atteint} \setminus \textit{traité}$ *avec* $\textit{descente}(x) < \textit{descente}(y)$,
$\forall x' \in \textit{circuit}(x)$,
$\forall y' \in \textit{circuit}(y)$,
$\textit{descente}(x') < \textit{descente}(y')$;

H. $\forall x \in \textit{atteint} \setminus \textit{traité}$,
$\forall x' \in \textit{circuit}(x)$,
$(x, \ldots, x')$ *est un chemin dans* $(\textit{circuit}(x), \textit{parcouru})$;

I. $\textit{calculé} \subseteq \textit{traité}$ *;*

J. $\forall x, y \in \textit{parcouru}$,
$x, y \in \textit{atteint}$;

K. $\forall x \in \textit{atteint} \setminus \textit{traité}$,
$\forall x' \in \textit{circuit}(x)$,
$\textit{racine}(x) \leq \textit{racine}(x')$;

L. $\forall x \in \textit{atteint}$,
$\textit{racine}(x) \leq \min\{\textit{descente}(x') | x' \in \{x\} \cup (\textit{parcouru}(x) \setminus \textit{calculé})\}$;

M. $x \in \textit{atteint} \setminus \textit{traité}$ *avec* $\textit{descente}(x)$ *minimal*
$\rightarrow$
$\textit{racine}(x) = \textit{descente}(x)$;

N. $\forall x \in \textit{traité} \setminus \textit{calculé}$,
$\textit{racine}(x) < \textit{descente}(x)$;

O. $\forall x \in \textit{atteint} \setminus \textit{calculé}$,
$(x, \ldots, \textit{descente}^{-1}(\textit{racine}(x)))$ *est un chemin dans* $(\textit{atteint} \setminus \textit{calculé}, \textit{parcouru})$;

P. $\forall x \in \textit{atteint} \setminus \textit{calculé}$,
$\exists y \in \textit{atteint} \setminus \textit{traité}$,
$(x, \ldots, y)$ *est un chemin dans* $(\textit{atteint} \setminus \textit{calculé}, \textit{parcouru})$;

Q. $\forall x, y \in \textit{atteint} \setminus \textit{traité}$,
$x \neq y$
$\rightarrow$
$\textit{circuit}(x) \cap \textit{circuit}(y) = \emptyset$;

R. $\bigcup_{x \in \textit{atteint} \setminus \textit{traité}} \textit{circuit}(x) = \textit{atteint} \setminus \textit{calculé}.$

Preuve B.2 *Chaque invariant est supposé vérifié au début du corps de la boucle. Pour chacun, il suffit de montrer que, quelle que soit l'exécution (il y en a cinq différentes), l'invariant reste satisfait à la fin du corps de la boucle. Les démonstrations sont suffisamment directes pour que nous omettions leurs preuves. Quelques unes ont recours à des invariants qui la précède. Nous ne démontrerons que (P) qui seule ne suit pas ce schéma de preuve.*

(P) $\forall x \in$ *atteint* $\setminus$ *calculé*,
$\exists y \in$ *atteint* $\setminus$ *traité*,
$(x, \ldots, y)$ *est un chemin dans* (*atteint* $\setminus$ *calculé*, *parcouru*).

Élaborons une preuve constructive.

Soit $x \in$ *atteint* $\setminus$ *calculé, alors on peut toujours construire une suite* $\lambda_1, \lambda_2, \ldots, \lambda_i, \ldots$, *telle que* $\lambda_1 = (t_1 = x, \ldots,$ *descente*$^{-1}$(*racine*$(x)) = t_2)$ *est un chemin dans* (*atteint* $\setminus$ *calculé*, *parcouru*) *et* $\forall i > 1, li = (t_i, \ldots,$ *descente*$^{-1}$(*racine*$(t_i)) = t_{i+1})$ *est aussi un chemin dans* (*atteint* $\setminus$ *calculé*, *parcouru*) *[invariant O et induction triviale sur l'appartenance de chaque* t_i *à atteint* $\setminus$ *calculé]. Cette suite converge au sens où* *descente*$(t_1) \geq$ *descente*$(t_2) \geq$ *descente*$(t_3) \geq \ldots$ *[invariants A, L et N pour la décroissante, et atteint* $\setminus$ *calculé est fini pour la valeur limite]. Remarquez que la fonction descente étant bijective de atteint* $\setminus$ *calculé vers* $I \subseteq N$, *descente*$^{-1}$(*racine*$(x)) \neq x$. *Soit* $k \geq 1$ *le plus petit indice tel que* *descente*$(t_k) =$ *racine*(t_k), *alors* $t_k \in$ *atteint* $\setminus$ *traité, car* $t_k \in$ *atteint* $\setminus$ *calculé [invariant O] mais* $t_k \in$ *traité* $\setminus$ *calculé est contradictoire [invariant N]. Ainsi,* $\lambda = \lambda_1.\lambda_2.\ldots.\lambda_k$ *est un chemin dans* (*atteint* $\setminus$ *calculé*, *parcouru*) *tel que* $t_k \in$ *atteint* $\setminus$ *traité.*

Remarquez que les invariants A et I autorisent l'ordre dans lequel sont faits les tests d'appartenance, tandis que l'invariant M permet d'éviter le test d'existence d'un prédécesseur dans atteint $\setminus$ traité lorsque les conditions « parcouru$(x) = \Gamma(x)$ » et « descente$(x) \neq$ racine(x) » sont vérifiées.

Les invariants F, Q et R montrent que les différents ensembles circuits associés aux sommets de atteint $\setminus$ traité partitionnent atteint $\setminus$ calculé. Toutefois, ce partitionnement de atteint $\setminus$ calculé ne permet pas de mettre en évidence les composantes fortement connexes de (atteint, parcouru) sans avoir recours à une relation complexe et récursive (par exemple, en figure B.1, les quatre derniers circuits forment une composante fortement connexe de (atteint, parcouru)). Aussi, n'allons nous nous intéresser qu'à une partition prête à être insérée dans calculé.

Théorème B.3 *Soit* $x \in$ *atteint* $\setminus$ *traité avec :* *descente*(x) *maximal,* *racine*$(x) =$ *descente*(x) *et* *parcouru*$(x) = \Gamma(x)$, *alors* (*circuit*(x), *parcouru*) *est une composante fortement connexe de* (X, Γ).

Preuve B.4 *Il faut montrer que (1) tous les couples de sommets dans circuit*(x) *sont liés par un circuit, et (2) qu'il n'existe aucun autre sommet dans tout le graphe vérifiant cette relation.*

Pour (1), nous devons montrer que $\forall x', x'' \in$ *circuit*$(x), \exists (x', \ldots, x'', \ldots, x')$ *un circuit dans* (*atteint* $\setminus$ *calculé*, *parcouru*). *Pour cela, montrons plus simplement que* $\forall x' \in$ *circuit*(x), $\exists (x, \ldots, x', \ldots, x)$ *un circuit dans* (*atteint* $\setminus$ *calculé*, *parcouru*).

Soit $x' \in$ *circuit*(x) *et* $x' \neq x$, *alors* $x' \in$ *traité* $\setminus$ *calculé [invariant E] et* $(x', \ldots, y)$ *est un chemin dans* (*atteint* $\setminus$ *calculé*, *parcouru*) *avec* $y \in$ *atteint* $\setminus$ *traité [invariant P]. Si* $x' = x$, *alors, trivialement, on pose* $y = x$. *Or* $(y, \ldots, x)$ *est un chemin dans* (*atteint* $\setminus$ *traité*, *parcouru*)

[invariant D] donc dans $(\textit{atteint} \setminus \textit{calculé}, \textit{parcouru})$ *[invariant I] et* $(x, \ldots, x')$ *est un chemin dans* $(\textit{circuit}(x), \textit{parcouru})$ *[invariant H] donc dans* $(\textit{atteint} \setminus \textit{calculé}, \textit{parcouru})$ *[invariants A et E]. Finalement, par concaténation,* $(x', \ldots, y).(y, \ldots, x).(x, \ldots, x') = (x, \ldots, x', \ldots, x)$ *est un circuit dans* $(\textit{atteint} \setminus \textit{calculé}, \textit{parcouru})$.

Par conséquent, $\forall x', x'' \in \textit{circuit}(x), (x, \ldots, x', \ldots, x).(x, \ldots, x'', \ldots, x)$ *forme un circuit dans* $(\textit{atteint} \setminus \textit{calculé}, \textit{parcouru})$.

Pour (2), montrons que $\textit{circuit}(x)$ *est maximal, en choisissant, sans perte de généralité et comme l'étape précédente le permet,* x *comme sommet caractéristique :*

$$
\begin{aligned}
&\forall (x = t_0, \ldots, t_k = x) \textit{ circuit dans } (X, \Gamma) \textit{ avec } k \geq 1,\\
&\forall i : 1 < i \leq k,\\
&\quad t_{i-1} \in \textit{circuit}(x) \wedge\\
&\quad t_i \in \textit{circuit}(x) \wedge\\
&\quad (t_{i-1}, t_i) \in \textit{parcouru}.
\end{aligned}
$$

Procédons par contradiction et supposons donc que :

$$
\begin{aligned}
&\exists (x = t_0, \ldots, t_k = x) \textit{ un circuit dans } (X, \Gamma) \textit{ avec } k \geq 1,\\
&\exists i : 1 < i \leq k,\\
&\quad t_{i-1} \notin \textit{circuit}(x) \vee\\
&\quad t_i \notin \textit{circuit}(x) \vee\\
&\quad (t_{i-1}, t_i) \notin \textit{parcouru}.
\end{aligned}
$$

Il existe nécessairement $i, j : 0 < i \leq j < k$ *tel que* $t_{i-1} \in \textit{circuit}(x), t_i \notin \textit{circuit}(x)$ *[car* $x = t_0$*] et* $t_j \notin \textit{circuit}(x), t_{j+1} \in \textit{circuit}(x)$ *[car* $t_k = x$*]. Nous pouvons, sans perte de généralité, ne considérer que le cas* i.

Si $t_{i-1} \in \textit{circuit}(x)$ *et* $t_{i-1} \neq x$*, alors* $t_{i-1} \in$ *traité*$\setminus$*calculé [invariant E] et* $\textit{parcouru}(t_{i-1}) = \Gamma(t_{i-1})$ *[invariant C], sinon* $t_{i-1} = x$ *et* $\textit{parcouru}(x) = \Gamma(x)$ *[par hypothèse]. Dans les deux cas,* $(t_{i-1}, t_i) \in \textit{parcouru}$*, donc* $t_i \in$ *atteint [invariant J].*

Dans un premier temps, supposons que $t_i \notin$ *calculé, alors soit* $t_i \in$ *traité et* $\exists y \in \textit{atteint} \setminus \textit{traité}, t_i \in \textit{circuit}(y)$ *[invariant P], soit* $t_i \in \textit{atteint} \setminus \textit{traité}$ *et on pose* $t_i = y$*. On a nécessairement* $y \neq x$ *[invariant F], d'où* $\textit{descente}(y) < \textit{descente}(x)$ *[par hypothèse] et* $\textit{descente}(t_i) < \textit{descente}(x)$ *[invariant G]. Enfin, nous avons* $\textit{racine}(x) \leq \textit{racine}(t_{i-1})$ *[invariant K] et* $\textit{racine}(t_{i-1}) \leq \textit{descente}(t_i)$ *[invariant L]. Nous obtenons ainsi* $\textit{descente}(t_i) < \textit{descente}(x) = \textit{racine}(x) \leq \textit{racine}(t_{i-1}) \leq \textit{descente}(t_i)$*, c'est-à-dire une contradiction.*

Supposons maintenant que $t_i \in$ *calculé et procédons par contradiction en réutilisant le début de la démonstration de cette partie. Quand un circuit quelconque est ajouté à calculé, nous savons que tous ses sommets appartiennent à traité* $\setminus$ *calculé [invariant E], que tous les*

arcs issus de ceux-ci ont été parcourus [invariant C] et surtout qu'il n'existe aucun arc (a, b) *tel que* $a \in$ *circuit et* $b \notin$ *circuit. Par conséquent, une fois l'adjonction faite dans calculé, il ne peut pas exister d'arc* (a, b) *tel que* $a \in$ *calculé et* $b \notin$ *calculé. Or, si l'on suppose que* $t_i \in$ *calculé, alors nécessairement* $\exists j : i \leq j < k, t_j \in$ *calculé* $\wedge\, t_{j+1} \in$ *atteint* $\setminus$ *calculé [car* $t_k = x$*], ce qui est contradictoire.*

Pour finir la preuve, tout sommet d'un circuit du graphe passant par au moins un sommet de circuit(x) *appartenant également à circuit*(x)*, on ne peut alors pas avoir d'arc n'appartenant pas à parcouru [invariants C et E, et hypothèse parcouru*$(x) = \Gamma(x)$*].*

Corollaire B.5 *Les propriétés suivantes sont des invariants de la boucle :*

S. $\forall x \in$ *calculé*,
$\Gamma(x) \subseteq$ *calculé* ;

T. $\forall x \in$ *atteint*,
$$\textit{VAT}(x) = \textit{VAD}(x) \vee \bigvee_{y \in \textit{parcouru}^*(x) \cap \textit{calculé}} \textit{VAD}(y).$$

Preuve B.6 *Pour (S), voir la seconde partie de la preuve du théorème B.3.*

Pour (T), supposons la propriété vérifiée au début du corps de la boucle.

Si circuit(x) *est inséré dans calculé, alors les deux boucles «* pour *» calculent bien les valeurs définitives des VAT grâce au théorème B.3. Finalement, l'instruction conditionnelle «* VAT$(y) \leftarrow$ VAT$(y) \subseteq$ VAT(x) *» conserve l'invariance pour le sommet* y*.*

Si l'extrémité y *du nouvel arc parcouru se trouve dans calculé, alors l'instruction «* VAT$(x) \leftarrow$ VAT$(x) \subseteq$ VAT(y) *» conserve l'invariance [invariant S].*

Enfin, si un nouveau sommet y *est atteint, alors les deux instructions «* VAT$(x) \leftarrow$ VAD(x) *» et «* parcouru$(x) \leftarrow \emptyset$ *» assurent l'invariance.*

B.2.2 Preuve de correction totale

Les invariants sont bien vérifiés par l'initialisation générale, ainsi que par l'initialisation partielle constituée par les premières instructions de la boucle externe.

Théorème B.7 *Si la boucle admet les invariants comme pré-condition, alors (1) elle converge et (2) elle admet comme post-conditions :*

1. *calculé* $=$ *traité* $=$ *atteint* $\subseteq X$ *;*
2. $(\textit{atteint}, \textit{parcouru}) = (\textit{atteint}, \Gamma)$.

Preuve B.8 *(1) La boucle admet comme invariants traité* $\subseteq$ *atteint [invariant A] et parcouru* $\subseteq$ Γ *[invariant B]. Donc,* $|\textit{traité}| + |\textit{parcouru}| \leq |X| + |\Gamma|$ *est également un invariant de la boucle. Or, l'expression* $|\textit{traité}| + |\textit{parcouru}|$ *est un variant strictement croissant de la boucle,* $|\textit{traité}|$

étant augmenté de 1 exactement dans la première branche de l'alternative principale, tandis que |parcouru| est augmenté de 1 exactement dans la seconde branche, et le graphe (X, Γ) est fini.

(2) traité = atteint est immédiat [invariant A et condition d'arrêt : atteint \ traité = ∅].

Nous avons calculé ⊆ traité [invariant I]. Supposons calculé ≠ traité, alors ∃y ∈ traité \ calculé = atteint \ calculé, donc ∃x ∈ atteint \ traité [invariant F], ce qui est en contradiction avec traité = atteint.

(atteint, parcouru) = (atteint, Γ) est immédiat [corollaire B.5].

Théorème B.9 *Si la boucle admet comme pré-condition les invariants, alors elle admet comme post-condition que les valeurs des VAT de tous les sommets atteints sont bien les vecteurs d'accès transitifs des méthodes associées.*

Preuve B.10 *Restreignons l'invariant T à calculé. Une induction triviale sur le corollaire B.5 montre que parcouru$^*(x)$ ∩ calculé = calculé. De la post-condition calculé = atteint, nous obtenons alors :*

$$\forall x \in \textit{atteint},\quad \textit{VAT}(x) = \textit{VAD}(x) \vee \bigvee_{y \in \Gamma^*(x)} \textit{VAD}(y).$$

Ceci n'est autre que la définition IV.9 que nous avons donnée de la valeur des vecteurs d'accès transitifs.

B.2.3 Complexité

Théorème B.11 *Soit N le nombre de champs apparaissant dans un VAT, alors la complexité de la boucle interne est en $O(N \times (|X| + |\Gamma|))$.*

Preuve B.12 *Toutes les instructions de la boucle peuvent se fairent en $O(1)$ comme nous le montrerons en section B.4.1. Éliminons momentanément les deux boucles « pour ». Alors, la preuve de l'arrêt [théorème B.7] montre que la complexité de la descente en profondeur est en $O(|X| + |\Gamma|)$, c'est-à-dire linéaire en la taille du graphe, ce qui est un résultat bien connu [Aho et al. 83] [Bordat & Cogis 86].*

Les deux boucles « pour » ne modifient pas la complexité. Elles ont une complexité égale au nombre de sommets dans l'ensemble circuit concerné. Or, nous avons vu que les ensembles circuits associées aux sommets de atteint \ traité partitionnement atteint \ calculé [invariants F, Q et R]. Un circuit n'étant inséré qu'une et une seule fois dans calculé, la complexité de toutes les insertions est alors en $O(|X|)$.

Chaque fois qu'un sommet est atteint pour la première fois, l'instruction « VAT$(x) \leftarrow$ VAD(x) » est exécutée. Les valeurs VAT et VAD ne pouvant pas être partagées, la recopie

se fait en $O(N)$. Un sommet n'est introduit dans atteint qu'une et une seule fois. Si le sommet y, extrémité d'un nouvel arc parcouru, se trouve déjà dans calculé, alors l'instruction « VAT$(x) \leftarrow$ VAT$(x) \vee$ VAT(y) » est elle aussi en $O(N)$. Chaque arc n'étant parcouru qu'une fois et une seule, ceci confirme la complexité annoncée.

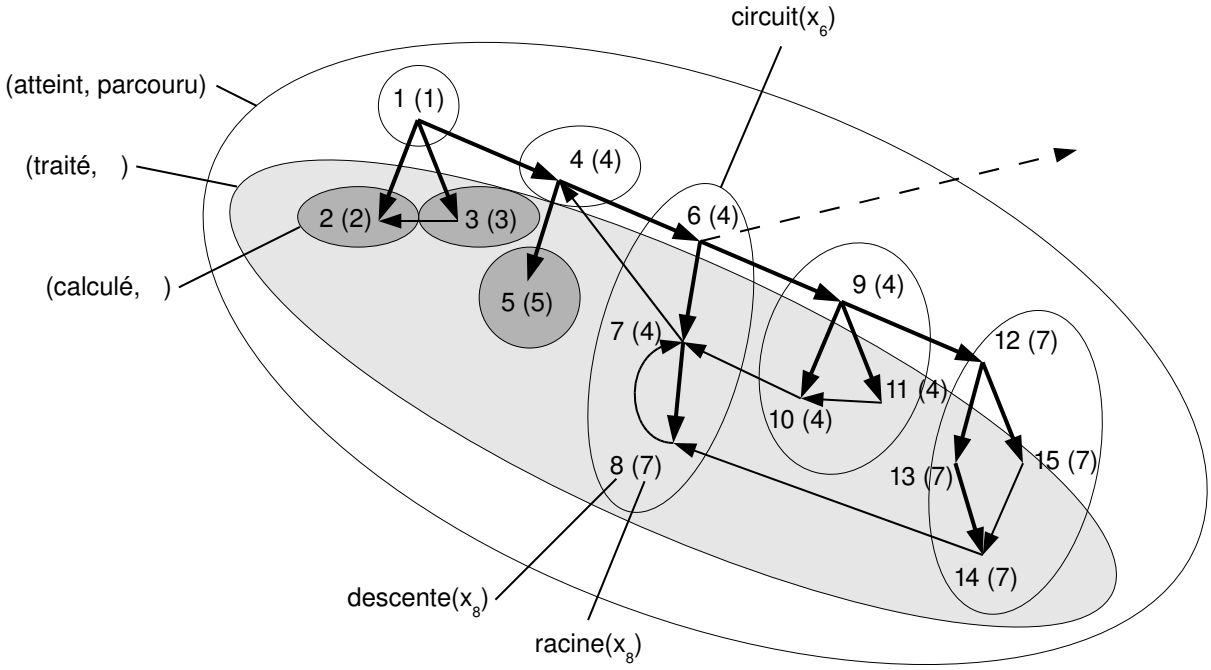


FIG. B.1 – Un exemple d'exécution en cours

B.3 IDÉE DE LA PREUVE DE LA BOUCLE EXTERNE

Les post-conditions données par les théorèmes B.7 et B.9 sont des invariants supplémentaires de la boucle externe. On remarque qu'ils sont vérifiés par l'initialisation générale.

La post-condition de la boucle externe est alors calculé $=$ traité $=$ atteint $= X$, d'où tout le graphe a été parcouru et tous les sommets ont vus leurs *VAT* calculés correctement.

Il est plus délicat de remarquer que la complexité de la boucle externe est la même que celle de la boucle interne. En effet, tous les sommets et arcs appartenant à calculé ne seront plus visités par les descentes successives.

B.4 INDICATIONS POUR UNE IMPLÉMENTATION EFFICACE ET INCRÉMENTALE

B.4.1 Efficacité

Toutes les opérations de l'algorithme B.1 sont réalisables en temps constant avec des structures de données appropriées. L'implémentation du graphe doit se faire sous forme chaînée. La structure d'un sommet est donnée en figure B.2.

$\Gamma(x)$ est une liste simplement chaînée de pointeurs sur les sommets successeurs. parcouru(x) est alors un pointeur sur la première cellule de $\Gamma(x)$ dont le sommet pointé n'a pas encore été atteint, ou nil quand parcouru$(x) = \Gamma(x)$. Le test « $\Gamma(x) \setminus$ parcouru$(x) = \emptyset$ » revient donc à tester la valeur nil. Les deux instructions consécutives « $y \leftarrow$ **choix**$(z \in \Gamma(x) \setminus$ parcouru$(x))$; parcouru$(x) \leftarrow$ parcouru$(x) \cup \{y\}$ » sont implémentées en avançant le pointeur parcouru(x) sur la cellule suivante.

L'invariant Q nous permet d'implémenter l'union de circuits en temps constant, puisqu'il n'y a pas de doublons à éliminer. Les circuits devront être représentés sous forme de listes simplement chaînées, comme Γ, mais avec en plus un pointeur direct sur la cellule de queue. L'instruction « circuit$(y) \leftarrow$ circuit$(y) \cup$ circuit(x) » sera donc réalisée, en temps constant, par les deux instructions suivantes : faire pointer la valeur successeur de la dernière cellule de circuit(y) sur la première cellule de circuit(x) ; rendre la valeur du pointeur de fin de liste de circuit(y) égale à celle de circuit(x). Notez que, comme l'on ne s'intéresse qu'aux ensembles circuits des sommets de atteint $\setminus$ traité, pour les autres sommets, les valeurs des pointeurs de tête et de queue peuvent être considérées comme indéfinies.

En corollaire de l'invariant D, nous avons la propriété que le sous-graphe partiel induit par atteint $\setminus$ traité définit un chemin dont les sommets sont ordonnés par la fonction descente. En figure B.1, le chemin instantané est $(x_1, x_4, x_6, x_9, x_{12})$. En pratique, cela signifie que la stratégie de descente en profondeur peut être implémentée en utilisant une pile. Le choix de x correspond donc à la lecture du sommet de la pile ; « traité $\leftarrow$ traité $\cup \{x\}$ » correspond à un dépilement ; enfin, « atteint $\leftarrow$ atteint $\cup \{y\}$ » est l'opération d'empilement. Ces trois opérations peuvent être implémentées en temps constant.

L'affectation « atteint $\leftarrow \emptyset$ » doit être remplaçée par un parcours de tous les sommets du graphe afin de mettre leur champ booléen correspondant à « *false* ». Les champs « *Traité* » et « *Calculé* » pourront n'être mis à « *false* » que quand « *Atteint* » est positionné à « *true* ». Cette boucle d'initialisation ne change pas la complexité de l'algorithme puisqu'elle n'est qu'en

Algorithme B.2 Initialisation après modifications de méthodes (*VAD* et/ou Γ)

atteint $\leftarrow X \setminus \bigcup_{x \text{ est un sommet modifié}} \Gamma^{-1*}(x)$;
traité $\leftarrow$ atteint ;
calculé $\leftarrow$ atteint ;

$O(|X|)$.

Enfin, et sans répercussions sur la complexité de l'algorithme, la boucle principale peut être écrite « *pour* $x \in X$ *faire* » en remplaçant l'instruction de choix par une alternative dont le test serait « *si non Atteint*(x) *alors* ».

B.4.2 Modifications incrémentales

Contrairement à l'algorithme de [Aho et al. 83],[2] celui-ci accepte, très simplement, une variation incrémentale.

Dans la version incrémentale de l'algorithme, les deux instructions d'initialisation « traité $\leftarrow \emptyset$; calculé $\leftarrow \emptyset$ » doivent être ôtée. En effet, le graphe est supposé correctement calculé. Le cycle des calculs est composé des deux étapes suivantes répétées autant de fois que voulu : modification du graphe, puis calcul incrémental, qui laisse de nouveau le graphe dans un état où tous les sommets du graphe font parti de l'ensemble calculé.

Les modifications du graphe peuvent consister en ajouts ou suppressions d'arcs, ajouts ou suppressions de sommets, ou modifications de *VAD*.

ajouts ou suppressions d'arcs, modifications de *VAD* L'ajout ou la suppression d'arcs, ainsi que les modifications de *VAD*, invalident les valeurs des *VAT* de tous les sommets prédécesseurs, directs ou indirects, des sommets dont on a changé les *VAD* ou les Γ.

On peut facilement retrouver ces sommets-là en effectuant une fermeture réflexo-transitive (par exemple, avec une descente en profondeur d'abord) sur les Γ^{-1} des sommets modifiés. L'initialisation est donnée avec l'algorithme B.2.

ajouts de sommets L'ajout de sommets est l'opération la plus simple qui soit puisqu'il suffit de positionner les champs « *Atteint* » des nouveaux sommets à « *false* » avant d'exécuter l'algorithme. Exécuter l'algorithme est nécessaire si plusieurs sommets sont introduits simultanément et que tous leurs arcs sortants ne sont pas dirigés que vers des sommets anciens. En

[2] Rappelons que cet algorithme procède en trois étapes : (1) parcours total du graphe en profondeur d'abord en numérotant les sommets suivant l'ordre de remontée (le squelette de l'algorithme est le même que celui de l'algorithme B.1), puis (2) renversement du graphe, c'est-à-dire renversement de tous ses arcs, enfin (3) nouveau parcours (pas nécessairement en profondeur d'abord) en choisissant chaque fois le sommet de numéro le plus grand dans $X \setminus$ atteint. La post-condition de la boucle interne (la descente) est que l'ensemble des sommets descendants du sommet choisi forme une composante fortement connexe. La complexité de cet algorithme est en $O(|X| + |\Gamma|)$ puisque chacune des trois étapes a cette même complexité. Remarquons déjà que la constante 3, que l'on ne doit pas oublier en pratique, pénalise cet algorithme.

Algorithme B.3 Initialisation en cas d'ajouts de nouvelles méthodes

$X \leftarrow X \cup \{\text{nouveaux sommets}\}$;
$\Gamma \leftarrow \Gamma \cup \bigcup_{x \text{ est un nouveau sommet}} \Gamma(x)$;

effet, dans ce cas, il risque de définir de nouveaux circuits entre eux. L'initialisation est donnée avec l'algorithme B.3.

suppressions de sommets Enfin, la suppression d'un sommet ne requiert aucun calcul si ce sommet était une source du graphe. Dans le cas contraire, d'autres sommets (c'est-à-dire méthodes) ont été invalidés puisque leurs Γ pointent sur un sommet disparu. Il faut alors procéder à des modifications récursives et l'on se retrouve dans le premier cas.

Remarquez que contrairement à l'algorithme de [Tarjan 72], la fonction descente n'est bijective que de atteint $\setminus$ calculé vers $I \subseteq N$. Sans cette relaxation, la version incrémentale ne pourrait être prouvée correcte.

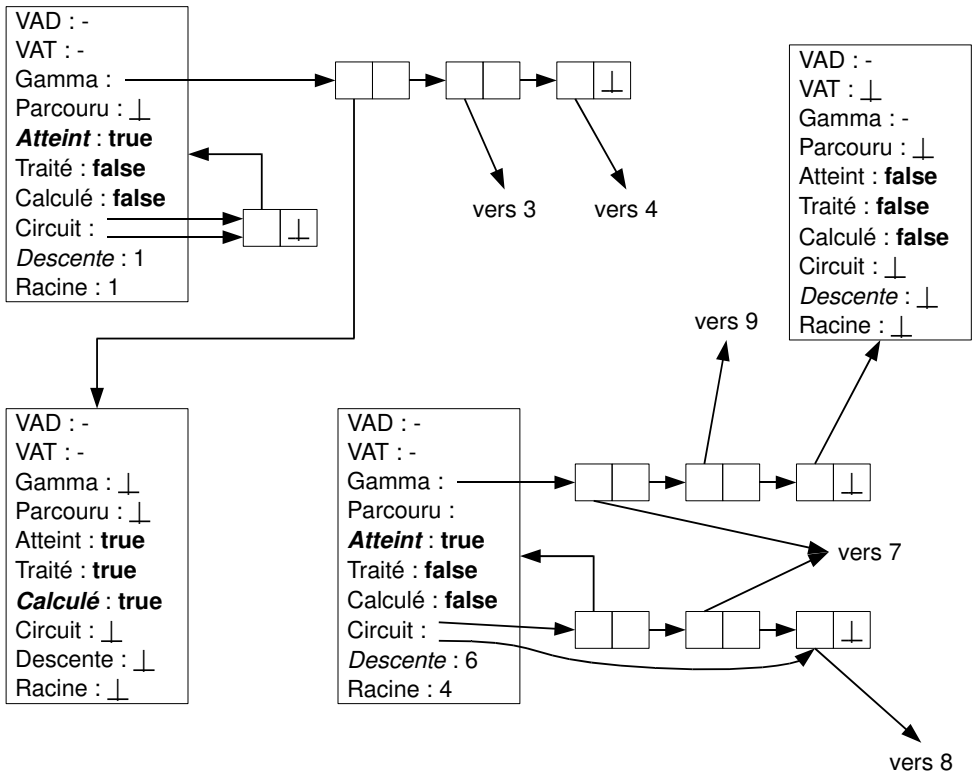


FIG. B.2 – Représentation en machine du graphe de la figure B.1

Annexe C

Calcul des classes frontière

C.1 L'ALGORITHME

Le graphe d'héritage est $G = (X, \Gamma)$. L'algorithme est une descente à partir d'un ou de plusieurs sommets $X_0 \subseteq X$, racines de l'union des domaines auxquels une transaction veut accéder. Cette extension, par rapport à la définition III.3 qui se limite à un unique sous-graphe, est justifiée par une caractéristique de Trellis/Owl [Schaffert et al. 86] : dans ce système, une variable peut être déclarée avec comme type une union de types (cf. exemple avec l'algorithme C.1).

En considérant X_0 dans son ensemble, plutôt que les différentes racines individuellement, on peut faire une économie sur le nombre de classes frontière. Sur l'exemple de la figure C.1, $X_0 = \{r_1, r_2\}$ et $F(G_{r_1}) = \{r_1, sc_1\}$ tandis que $F(G_{r_2}) = \{r_2, sc_1, sc_2\}$. Les classes frontière de l'union des sous-graphes n'est pas l'union des classes frontière de chacun des sous-graphes, mais se limite à $\{r_1, r_2\}$.

On gagne également en temps de calcul puisque l'on ne parcout qu'une seule fois les sous-graphes partiels issus d'intersections de sous-graphes, voire d'inclusions (par exemple, sur la déclaration de l'algorithme C.1, MONITEUR $\subseteq$ ALLOCATAIRERECHERCHE $\subseteq$ ÉTUDIANT3ECYCLE).

L'algorithme C.2 consiste à calculer, en même temps que l'ensemble des descendants de X_0, une fonction « degré » en chaque sommet qui sera le nombre d'arcs entrant qui proviennent des classes qui ne sont pas descendantes de X_0. Il est alors évident que seuls les sommets ayant une valeur « degré » strictement positive sont des classes frontière.

Algorithme C.1 Déclaration d'union de types en Trellis/Owl

```
var EnseignantsIUT : set of (Professeur |
                             Etudiant3eCycle |
                             Moniteur |
                             Professionnel) ;
```

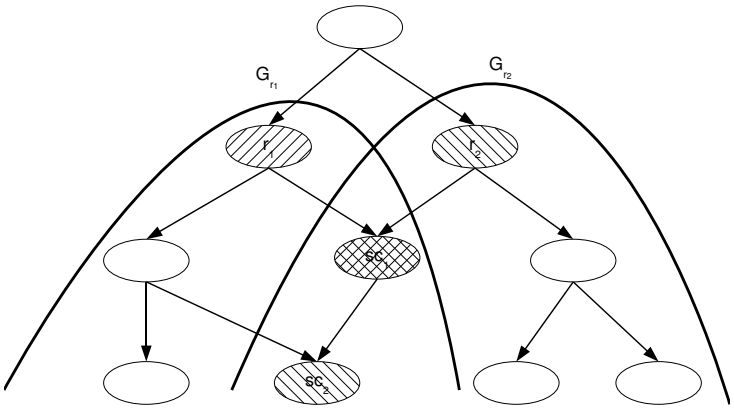


FIG. C.1 – Exemple de classes frontière

Algorithme C.2 Algorithme de calcul des classes frontière d'une union de domaines

atteint $\leftarrow X_0$;
pour $x \in X_0$ faire
  parcouru$(x) \leftarrow \emptyset$;
  degré$(x) \leftarrow |\Gamma^{-1}(x)|$;
traité $\leftarrow \emptyset$;
tant que atteint $\setminus$ traité $\neq \emptyset$ faire
  $x \leftarrow$ **choix**$(z \in$ atteint $\setminus$ traité$)$;
  si $\Gamma(x) \setminus$ parcouru$(x) = \emptyset$
  alors traité $\leftarrow$ traité $\cup \{x\}$;
  sinon $y \leftarrow$ **choix**$(z \in \Gamma(x) \setminus$ parcouru$(x))$;
    parcouru$(x) \leftarrow$ parcouru$(x) \cup \{y\}$;
    si $y \in$ atteint
    alors degré$(y) \leftarrow$ degré$(y) - 1$;
    sinon atteint $\leftarrow$ atteint $\cup \{y\}$;
      parcouru$(y) \leftarrow \emptyset$;
      degré$(y) \leftarrow |\Gamma^{-1}(y)| - 1$;
frontière $\leftarrow \emptyset$;
pour $x \in$ atteint faire
  si (degré$(x) > 0) \vee (|\Gamma^{-1}(x)| = 0)$
  alors frontière $\leftarrow$ frontière $\cup \{x\}$;

C.2 IDÉE DE LA PREUVE

L'invariant important de la boucle tant que de cet algorithme est :

$$\forall x \in \text{atteint},$$
$$\text{degré}(x) = |\Gamma^{-1}(x)| - |\{y \in \text{atteint} | x \in \text{parcouru}(y)\}|.$$

Comme nous disposons aussi des invariants A et C de l'annexe B, « traité $\subseteq$ atteint » et « $\forall x \in$ traité, parcouru$(x) = \Gamma(x)$ », nous avons la post-condition immédiate « atteint $=$ traité », mais surtout :

$$\forall x \in \text{atteint},$$
$$\text{degré}(x) = |\Gamma^{-1}(x) \setminus y \in \text{atteint} | y \in \Gamma^{-1}(x)|.$$

Or, atteint est l'ensemble des sommets descendants de X_0. Donc, de par la définition III.3, les classes frontière de l'union de sous-graphes associés à X_0 (c'est-à-dire atteint) sont bien celles qui ont une valeur « degré » strictement positive. Enfin, les sommets sans prédécesseurs, c'est-à-dire la classe racine (ou les classes sources) du graphe d'héritage, ont été déclarés classes frontière par convention. Ceci prouve la dernière boucle et la validité du résultat qui se trouve dans la variable « *frontière* ». Remarquons toutefois qu'une « descente en largeur d'abord » [Aho et al. 83] permettrait de déterminer pendant la descente les classes frontière.

C.3 COMPLEXITÉ

La complexité de cet algorithme est en $O(|X| + |\Gamma|)$ (cf. preuve de l'arrêt dans le théorème B.7). Il est bien évident qu'elle est très précisément en la taille de l'union des sous-graphes et non en celle de la totalité du graphe d'héritage. Il n'est donc pas moins efficace que celui d'ORION qui consiste à effectuer une descente et à verrouiller tout sommet qui possède plus d'une super-classe [Garza & Kim 88].

Bibliographie

[Abbott 90] ABBOTT, R. J. ; *Resourceful systems for fault tolerance, reliability, and safety* ; ACM Computing Surveys, vol. 22, n° 1, mars 1990, pp. 35-68

[Abiteboul & Bonner 91] ABITEBOUL, S., BONNER, A. ; *Objects and views* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Denver, Colorado, USA, mai 1991, pp. 238-247

[Abiteboul 89] ABITEBOUL, S. ; *Virtuality in object-oriented databases* ; VI^{es} Journées Bases de Données Avancées, Montpellier, France, septembre 1989, pp. 17-32

[Acosta 91] ACOSTA, F. ; *Les arbres balancés : spécification, performances et contrôle de concurrence* ; Thèse d'université, Université Montpellier II, Sciences et techniques du Languedoc, octobre 1991, 200 p.

[Agrawal & DeWitt 85] AGRAWAL, R., DEWITT, D. J. ; *Integrated concurrency control and recovery mechanism: design and performance evaluation* ; ACM Transactions On Database Systems, vol. 10, n° 4, décembre 1985, pp. 529-564

[Agrawal & El Abbadi 92] AGRAWAL, D., EL ABBADI, A. ; *A non-restrictive concurrency control for object-oriented databases* ; Proceedings of the 3rd Inf. Conf. on Extending Data Base Technology, Vienne, Autriche, 1992, pp. 469-482

[Agrawal & Gehani 89] AGRAWAL, R., GEHANI, N. H. ; *ODE (Object Database and Environment): the language and the data model* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Portland, Oregon, USA, vol. 18, n° 2, juin 1989, pp. 36-45

[Agrawal et al. 87] AGRAWAL, R., CAREY, M. J., LIVNY, M. ; *Concurrency control performance modeling: alternatives and implications* ; ACM Transactions On Database Systems, vol. 12, n° 4, décembre 1987, pp. 609-654

[Aho et al. 83] AHO, A. V., HOPCROFT, J. E., ULLMAN, J. D. ; *Data structures and algorithms* ; Addison-Wesley Pusblishing Company, Reading, Massachusetts, USA, janvier 1983, 427 p.

[Aldebert & Martinez 89] ALDEBERT, O., MARTINEZ, J. ; *Contrôle de concurrence dans les bases de données à objets* ; Mémoire de DEA Informatique, Université Montpellier II, Sciences et Techniques du Languedoc, juin 1989, 87 p.

[America & Van Der Linder 90] AMERICA, P., VAN DER LINDER, F. ; *A parallel object-oriented language with inheritance and subtyping* ; Proceedings of the ECOOP/OOPSLA Int. Conf., Ottawa, Canada, octobre 1990, pp. 161-168

[Andrews & Harris 87] ANDREWS, T., HARRIS, C. ; *Combining language and database advances in an object-oriented development environment* ; Proceedings of the 2nd ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, Orlando, Floride, USA, octobre 1987, pp. 430-440

[Appelbe & Ravn 84] APPELBE, W. F., RAVN, A. P. ; *Encapsulation constructs in system programming languages* ; ACM Transactions On Programming Languages and Systems, vol. 6, n° 2, avril 1984, pp. 129-158

[Atkinson et al. 89] ATKINSON, M. P., BANCILHON, F., DEWITT, D., DITTRICH, K., MAIER, D., ZDONIK, S. ; *The object-oriented database system manifesto* ; Proceedings of the 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japon, décembre 1989

[Badrinath & Ramamritham 87] BADRINATH, B. R., RAMAMRITHAM, K. ; *Semantics-based concurrency control: beyond commutativity* ; Proceedings of the 3rd IEEE Int. Conf. on Data Engineering, Los Angeles, USA, février 1987

[Badrinath & Ramamritham 88] BADRINATH, B. R., RAMAMRITHAM, K. ; *Synchronizing transactions on objects* ; IEEE Transactions On Computers, vol. 37, n° 5, mai 1988, pp. 541-547

[Badrinath & Ramamritham 90] BADRINATH, B. R., RAMAMRITHAM, K. ; *Performance evaluation of semantics-based multilevel concurrency control protocols* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Atlantic City, New Jersey, USA, mai 1990, pp. 163-172

[Badrinath & Ramamritham 92] BADRINATH, B. R., RAMAMRITHAM, K. ; *Semantics-based concurrency control: beyond commutativity* ; ACM Transactions On Database Systems, vol. 17, n° 1, mars 1992, pp. 163-199

[Bancilhon 88] BANCILHON, F. ; *Object-oriented database systems* ; Proceedings of the 7th Int. Symposium on the Principles Of Database Systems, Austin, Texas, USA, mars 1988

[Bancilhon et al. 89] BANCILHON, F., CLUET, S., DELOBEL, C. ; *A query language for an object-oriented database system* ; Proceedings of the 2nd Int. Workshop on Database Programming Languages, Salishan, Oregon, USA, juin 1989

[Banerjee et al. 87a] BANERJEE, J., CHOU, H.-T., GARZA, J. F., KIM, W., BALLOU, D. W. N., KIM, H.-J. ; *Data model issues for object-oriented applications* ; ACM Transactions On Information Systems, vol. 5, n° 1, janvier 1987, pp. 3-26

[Banerjee et al. 87b] BANERJEE, J., KIM, W., KIM, H.-J., KORTH, H. F. ; *Semantics and implementation of schema evolution in object-oriented databases* ; Proceedings of the ACM Int. Conf. on the Management Of Data, San Francisco, Californie, USA, mai 1987, pp. 311-322

[Barthès 88] BARTHÈS, J.-P. A. ; *Bases de données et CAO* ; 5e école d'automne de l'AFCET : bases de données, objets, connaissances, Port-Barcarès, novembre 1989

[Beeri & Milo 91] BEERI, C., MILO, T. ; *A model for active object-oriented database* ; Proceedings of the 17th Int. Conf. on Very Large Data Bases, Barcelone, Espagne, septembre 1991, pp. 337-349

[Beeri et al. 88] BEERI, C., SCHEK, H.-J., WEIKUM, G. ; *Multi-level transaction management: theoritical art or practical need?* ; Proceedings of the Int. Conf. on Extending Database Technology, Venise, Italie, 1988, pp. 134-154

[Beeri et al. 89] BEERI, C., BERNSTEIN, P. A., GOODMAN, N. ; *A model for concurrency in nested transactions* ; Journal of the ACM, vol. 36, n° 2, avril 1989, pp. 230-269

[Benzaken & Delobel 89a] BENZAKEN, V., DELOBEL, C. ; *Regroupement d'objets sur disque dans un système de bases de données orienté-objet* ; Ves Journées Bases de Données Avancées, Genève, Suisse, september 1989

[Benzaken 90] BENZAKEN, V. ; *Un modèle d'évaluation de stratégies de regroupement dans un système de bases de données orienté-objet* ; VIes Journées Bases de Données Avancées, Montpellier, France, septembre 1990, pp. 397-418

[Berge 67] BERGE, C. ; *Théorie des graphes et S applications* ; Dunod, Paris, France, 1967, 269 p.

[Bernstein et al. 87] BERNSTEIN, P. A., HADZILACOS, V., GOODMAN, N. ; *Concurrency control and recovery in database systems* ; Addison-Wesley Publishing Company, Reading, Massachusets, 1987

[Bernstein et al. 90] BERNSTEIN, P. A., HSU, M., MANN, B. ; *Implementing recoverable requests using queues* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Atlantic City, New Jersey, USA, mai 1990, pp. 112-122

[Bertino & Martino 91] BERTINO, E., MARTINO, L. ; *Object-oriented database management systems: concepts and issues* ; Computer, vol. 24, n° 4, avril 1991, pp. 33-47

[Berztiss 75] BERZTISS, A. T. ; *Data structures: theory and practice* ; Academic Press, New-York, USA, 1975, 586 p.

[Bloom & Zdonik 87] BLOOM, T., ZDONIK, S. B. ; *Issues in the design of object-oriented database programming languages* ; Proceedings of the 2nd ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, octobre 1987

[Boksenbaum et al. 84] BOKSENBAUM, C., CART, M., FERRIÉ, J., PONS, J.-F. ; *Certification by intervals of timestamps in distributed database systems* ; Proceedings of the 10th Int. Conf. on Very Large Data Bases, Singapour, août 1984

[Boksenbaum et al. 86] BOKSENBAUM, C., CART, M., FERRIÉ, J., PONS, J.-F. ; *Le contrôle de cohérence dans les bases de données réparties* ; Modèles et Bases de Données, n° 2, janvier 1986

[Bordat & Cogis 86] BORDAT, J.-P., COGIS, O. ; *Parcours et calculs dans les graphes sans circuit* ; Rapport de Recherche n° 11, Centre de Recherche en Informatique de Montpellier, Université des Sciences et Techniques du Languedoc, janvier 1986, 43 p.

[Bouvier 82] BOUVIER, A. ; *La théorie des ensembles* ; Collection "Que sais-je ?", Presses Universitaires de France, août 1982, 128 p.

[Broessler & Freisleben 89] BROESSLER, P., FREISLEBEN, B. ; *Transactions on persistent objects* ; Proceedings of the Workshop on Persistent Object Systems : their Design, Implementation, and Use, Newcastle, Australie, janvier 1989, pp. 19-35

[Butterworth et al. 91] BUTTERWORTH, P., OTIS, A., STEIN, J. ; *The GemStone object database management system* ; Communications of the ACM, vol. 34, n° 10, octobre 1991, pp. 64-77

[Carey & Livny 89] CAREY, M. J., LIVNY, M. ; *Parallelism and concurrency control performance in distributed database machines* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Portland, Oregon, USA, juin 1989, pp. 122-133

[Cart & Ferrié 89a] CART, M., FERRIÉ, J. ; *Contrôle de la concurrence et tolérance aux pannes dans les systèmes répartis* ; Actes du Séminaire Franco-Brésilien sur les Systèmes Informatiques Répartis, Florianopôlis-SC, Brésil, septembre 1989, pp. 239-246

[Cart & Ferrié 89b] CART, M., FERRIÉ, J. ; *GIP Altaïr, rapport final : contrôle de concurrence dans* O_2 ; rapport, CRIM & GIP Altaïr, mars 1989

[Cart & Ferrié 90] CART, M., FERRIÉ, J. ; *Integrating concurrency control into an object-oriented database system* ; Proceedings of the 2nd Int. Conf. on Extending Data Base Technology, Venise, Italie, mars 1990, pp. 363-377

[Cart et al. 89] CART, M., FERRIÉ, J., RICHY, H. ; *Contrôle de l'exécution de transactions concurrentes* ; Technique et Science Informatiques, vol. 8, n° 3, 1989

[Cart et al. 90a] CART, M., FERRIÉ, J., RICHY, H. ; *An optimistic multi-level concurrency control for nested typed objects* ; Proceedings of the 23rd Annual Hawaii Int. Conf. on System Sciences, vol. II, 1990, pp. 481-490

[Cart et al. 90b] CART, M., FERRIÉ, J., PONS, J.-F. ; *Objects modeling when using a multi-level transaction model* ; Proceedings of the ECOOP/OOPSLA Workshop on Transactions and Objects, Ottawa, Canada, octobre 1990

[Chamberlin et al. 81] CHAMBERLIN, D. D., ASTRAHAN, M. M., BLASGEN, M. W., GRAY, J. N., KING, W. F., LINDSAY, B. G., LORIE, R., MEHL, J. W., PRICE, T. G., PUTZOLU, F., SELINGER, P. G., SCHKOLNICK, M., SLUTZ, D. R., TRAIGER, I. L., WADE, B. W., YOST, R. A. ; *A history and evaluation of system R* ; Communications of the ACM, vol. 24, n° 10, octobre 1981, pp. 632-646

[Chrisment & Zurfluh 89] CHRISMENT, C., ZURFLUH, G. ; *Bases de données multimédia et approche orientée objet* ; 5e école d'automne de l'AFCET : bases de données, objets, connaissances, Port-Barcarès, 5-10 novembre 1989

[Chrysanthis et al. 91] CHRYSANTHIS, P. K., RAGHURAM, S., RAMAMRITHAM, K. ; *Extracting concurrency from objects: A methodology* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Denver, Colorado, USA, mai 1991, pp. 108-117

[Dasgupta & Kedem 90] DASGUPTA, P., KEDEM, Z. M. ; *The five color concurrency control protocol: non-two-phase locking in general databases* ; ACM Transactions On Database Systems, vol. 15, n° 2, juin 1990, pp. 281-307

[Dayal et al. 90] DAYAL, U., HSU, M., LADIN, R. ; *Organizing long-running activities with triggers and transactions* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Atlantic City, New Jersey, USA, mai 1990, pp. 204-214

[Dayal et al. 91] DAYAL, U., HSU, M., LADIN, R. ; *A transactional model for long-running activities* ; Proceedings of the 17th Int. Conf. on Very Large Data Bases, Barcelone, Espagne, septembre 1991, pp. 113-122

[Delobel & Adiba 82] DELOBEL, C., ADIBA, M. ; *Bases de données et systèmes relationnels* ; Editions DUNOD, Bordas, Paris, France, 1982

[Diaz et al. 91] DIAZ, O., PATON, N., GRAY, P. ; *Rule management in object-oriented databases: a uniform approach* ; Proceedings of the 17th Int. Conf. on Very Large Data Bases, Barcelone, Espagne, septembre 1991, pp. 317-326

[Eich 88] EICH, M. H. ; *Graph directed locking* ; IEEE Transactions On Software Engineering, vol. 14, n° 2, février 1988

[Elhardt & Bayer 84] ELHARDT, K., BAYER, R. ; *A database cache for high performance and fast restart in database systems* ; ACM Transactions On Database Systems, vol. 9, n° 4, décembre 1984, pp. 503-525

[Estier & Guyot 91] ESTIER, T., GUYOT, J. ; *Du local au global : un aller-retour* ; VIIes Journées Bases de Données Avancées, Lyon, France, septembre 1991, pp. 279-298

[Eswaran et al. 76] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., TRAIGER, I. L. ; *The notions of consistency and predicative locks in a database system* ; Communications of the ACM, vol. 19, n° 11, novembre 1976, pp. 624-633

[Exertier 91] EXERTIER, F. ; *Extension orientée objet d'un SGBD relationnel* ; Thèse d'université, Université Grenoble I, Joseph Fourier, décembre 1991, 145 p.

[Fishman et al. 87] FISHMAN, D. H., BEECH, D., CATE, H. P., CHOW, E. C., CONNORS, T., DAVIS, J. W., DERRETT, N., HOCH, C. G., KENT, W., LYNGBAEK, P., MAHBOD, B., NEIMAT, N. A., RYAN, T. A., SHAN, M. C. ; *Iris: an object-oriented database management system* ; ACM Transactions On Information Systems, vol. 5, n° 1, janvier 1987, pp. 48-69

[Franaszek & Robinson 85] FRANASZEK, P., ROBINSON, J. T. ; *Limitations of concurrency in transaction processing* ; ACM Transactions On Database Systems, vol. 10, n°1, mars 1985, pp. 1-28

[Franklin et al. 92] FRANKLIN, M. J., ZWILLING, M. J., TAN, C. K., CAREY, M. J., DEWITT, D. J. ; *Crash recovery in client-server EXODUS* ; Proceedings of the ACM Int. Conf. on the Management Of Data, San Diego, Californie, USA, juin 1992, pp. 165-174

[Gardarin & Valduriez 90] GARDARIN, G., VALDURIEZ, P. ; *SGBD avancés : bases de données objets, déductives, réparties* ; Editions Eyrolles, Série Architecture des systèmes d'informations, Paris, France, 1990

[Garza & Kim 88] GARZA, J. F., KIM, W. ; *Transaction management in an object-oriented database system* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Chicago, Illinois, USA, juin 1988

[Gehani & Jagadish 91] GEHANI, N., JAGADISH, H. V. ; *Ode as an active database: constraints and triggers* ; Proceedings of the 17th Int. Conf. on Very Large Data Bases, Barcelone, Espagne, septembre 1991, pp. 327-336

[Goldberg & Robson 83] GOLDBERG, A., ROBSON, D. ; *Smalltalk-80, the language and its implementation* ; Addison-Wesley, Reading, Massachusetts, 1983

[Goodman & Shasha 85] GOODMAN, N., SHASHA, D. ; *Semantically-based concurrency control for search structures* ; Proceedings of the 4th ACM Symposium on Principles Of Database Systems, USA, mars 1985, pp. 8-19

[Gray 78] GRAY, J. N. ; *Notes on database operating systems* ; An Advanced Course in Operating Systems, Springer-Verlag, New-York, 1978-1979

[Gray 81] GRAY, J. N. ; *The transaction concept: virtues and limitations* ; Proceedings of the 7th Int. Conf. on Very Large Data Bases, Cannes, France, 1981, pp. 144-154

[Gray et al. 81] GRAY, J. N., JONES, P. M., BLASGEN, M., LINDSAY, B. G., LORIE, R. A., PRICE, T., PUTZOLU, F., TRAIGER, I. L. ; *The recovery manager of the system R database manager* ; ACM Computing Surveys, vol. 13, n° 2, juin 1981, pp. 223-242

[Greenberg et al. 92] GREENBERG, S., ROSEMAN, M., WEBSTER, D., BOHNET, R. ; *Issues and experiences designing and implementing two group drawing tools* ; Proceedings of the 25th Annual Hawaii Int. Conf. on System Sciences, vol. IV, 1992, pp. 139-150

[Haerder & Reuter 83] HAERDER, T., REUTER, A. ; *Principles of transaction-oriented database recovery* ; ACM Computing Surveys, vol. 15, n° 4, décembre 1983, pp. 287-317

[Harel 80] HAREL, D. ; *On folk theorems* ; Communications of the ACM, vol. 23, n° 7, juillet 1980, pp. 379-389

[Hasse & Weikum 91] HASSE, C., WEIKUM, G. ; *A performance evaluation of multi-level transaction management* ; Proceedings of the 17th Int. Conf. on Very Large Data Bases, Barcelone, Espagne, septembre 1991, pp. 55-66

[Hermann et al. 90] HERMANN, U., DADAM, P., KÜSPERT, K., ROMAN, E. A., SCHLAGETER, G. ; *A lock technique for disjoint and non-disjoint complex objects* ; Proceedings of the 2nd Int. Conf. on Extending Data Base Technology, Venise, Italie, mars 1990, pp. 219-237

[Hesselink 88] HESSELINK, W. H. ; *A mathematical approach to nondeterminism in data types* ; ACM Transactions On Programming Languages and Systems, vol. 10, n° 1, janvier 1988, pp. 87-117

[Huang 85] HUANG, S.-H. S. ; *Height-balanced trees of order* (b, g, d) ; ACM Transactions On Database Systems, vol. 10, n°2, juin 1985, pp. 261-284

[Huang et al. 91] HUANG, J., STANKOVIC, J. A., RAMAMRITHAM, K., TOWSLEY, D. ; *Experimental evaluation of real-time concurrency control scheme* ; Proceedings of the 17th Int. Conf. on Very Large Data Bases, Barcelone, Espagne, septembre 1991, pp. 35-46

[Hwang & Lee 91] HWANG, S., LEE, S. ; *The object-oriented relationship system for managing complex relationships* ; Proceedings of the 2nd Int. Symposium on Database Systems For Advanced Applications, Tokyo, Japon, avril 1991, pp. 391-400

[Jhingran & Khedkar 92] JHINGRAN, A., KHEDKAR, P. ; *Analysis of recovery in a database system using write-ahead log protocol* ; Proceedings of the ACM Int. Conf. on the Management Of Data, San Diego, Californie, USA, juin 1992, pp. 175-184

[Kedem & Silberschatz 83] KEDEM, Z. M., SILBERSCHATZ, A. ; *Locking protocols: from exclusive to shared locks* ; Journal of the ACM, vol. 30, n° 4, octobre 1983, pp. 787-804

[Khoshafian & Copeland 86] KHOSHAFIAN, S. N., COPELAND, G. F. ; *Object identity* ; Proceedings of the 1st ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA, septembre 1986, pp. 406-416

[Kim 90] KIM, W. ; *Object-oriented databases: definition and research directions* ; IEEE Transactions On Knowledge And Data Engineering, vol. 2, n° 3, septembre 1990, pp. 327-341

[Kim et al. 87] KIM, W., BANERJEE, J., CHOU, H.-T., GARZA, J. F., WOELK, D. ; *Composite object support in an object-oriented database system* ; Proceedings of the 2nd ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, octobre 1987

[Kim et al. 89] KIM, W., BERTINO, E., GARZA, J. F. ; *Composite objects revisited* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Portland, Oregon, USA, juin 1989

[Kim et al. 90] KIM, W., GARZA, J. F., BALLOU, N., WOELK, D. ; *Architecture of the ORION next-generation database system* ; IEEE Transactions On Knowledge And Data Engineering, vol. 2, n° 1, mars 1990, pp. 109-124

[Knapp 87] KNAPP, E. ; *Deadlock detection in distributed databases* ; ACM Computing Surveys, vol. 19, n° 4, décembre 1987, pp. 303-328

[Kohler 81] KOHLER, W. H. ; *A survey of techniques for synchronization and recovery in decentralized computer systems* ; ACM Computing Surveys, vol. 13, n° 2, pp. 149-183

[Korth 82] KORTH, H. F. ; *Deadlock freedom using edge locks* ; ACM Transactions On Database Systems, vol. 7, n° 4, décembre 1982, pp. 632-652

[Korth 83] KORTH, H. F. ; *Locking primitives in a database system* ; Journal of the ACM, vol. 30, n° 1, janvier 1983, pp. 55-79

[Korth et al. 88] KORTH, H. F., KIM, W., BANCILHON, F. ; *On long-duration CAD transactions* ; Information sciences 46, 1988, pp. 73-107

[Krakowiak 85] KRAKOWIAK, S. ; *Principes des systèmes d'exploitation des ordinateurs* ; DUNOD Informatique, Bordas, Paris, France, 1985, 436 p.

[Kumar & Stonebraker 89] KUMAR, A., STONEBRAKER, M. ; *Performance considerations for an operating system transaction manager* ; IEEE Transactions On Software Engineering, vol. 15, n° 6, juin 1989

[Kung & Robinson 81] KUNG, H. T., ROBINSON, J. T. ; *On optimistic methods for concurrency control* ; ACM Transactions On Database Systems, vol. 6, n° 2, juin 1981, pp. 213-226

[Kuo 92] KUO, D. ; *Model and verification of a data manager based on ARIES* ; Proceedings of the 4th Int. Conf. on Database Theory, Berlin, Allemagne, octobre 1992

[Lamb et al. 91] LAMB, C., LANDIS, G., ORENSTEIN, J., WEINREB, D. ; *The ObjectStore database system* ; Communications of the ACM, vol. 34, n° 10, octobre 1991, pp. 51-63

[Lécluse & Richard 89a] LÉCLUSE, C., RICHARD, P. ; *The O_2 data model* ; GIP Altaïr Technical Report 37-89, octobre 1989

[Lécluse & Richard 89b] LÉCLUSE, C., RICHARD, P. ; *The O_2 database programming language* ; Proceedings of the 15th Int. Conf. on Very Large Data Bases, Amsterdam, Pays-Bas, août 1989

[Lécluse et al. 88] LÉCLUSE, C., RICHARD, P., VELEZ, F. ; *O_2, an object-oriented data model* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Chicago, Illinois, USA, juin 1988

[Lehman & Yao 81] LEHMAN, P. L., YAO, S. B. ; *Efficient locking for concurrent operations on B-trees* ; ACM Transactions On Database Systems, vol. 6, n°4, décembre 1981, pp. 650-670

[Lerner & Habermann 90] LERNER, B. S., HABERMANN, A. N. ; *Beyond schema evolution to database reorganization* ; Proceedings of the ECOOP/OOPSLA Int. Conf., Ottawa, Canada, octobre 1990, pp. 67-76

[Lohman et al. 91] LOHMAN, G. M., LINDSAY, B., PIRAHESH, H., SCHIEFER, K. B. ; *Extensions to StarBurst: objects, types, functions, and rules* ; Communications of the ACM, vol. 34, n° 10, octobre 1991, pp. 94-109

[Maier et al. 86] MAIER, D., STEIN, J., OTIS, A., PURDY, A. ; *Development of an object-oriented DBMS* ; Proceedings of the 1st ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, septembre 1986

[Malta & Martinez 90] MALTA, C., MARTINEZ, J. ; *Le contrôle des accès concurrents dans un environnement à objets* ; VIes Journées Bases de Données Avancées, Montpellier, France, septembre 1990, pp. 439-454

[Malta & Martinez 91a] MALTA, C., MARTINEZ, J. ; *Controlling concurrent accesses in an object-oriented environment* ; Proceedings of the 2nd Int. Symposium on Database Systems For Advanced Applications, Tokyo, Japon, avril 1991, pp. 192-200

[Malta & Martinez 91b] MALTA, C., MARTINEZ, J. ; *A framework for designing concurrent and recoverable abstract data types based on commutativity* ; Proceedings of the 6th Int. Symposium on Computer and Information Sciences, vol. I, Side, Antalya, Turquie, novembre 1991, pp. 189-198

[Malta & Martinez 92a] MALTA, C., MARTINEZ, J. ; *Limits of commutativity on abstract data types* ; Proceedings of the 3rd Int. Conf. on Information Systems and Management Of Data, Bangalore, Inde, juillet 1992, pp. 261-270

[Malta & Martinez 92b] MALTA, C., MARTINEZ, J. ; *Contrôle de concurrence et reprise : un état de l'art* ; Actes du 1er Colloque Africain sur la Recherche en Informatique, Yaoundé, Cameroun, octobre 1992, pp. 337-348

[Malta & Martinez 92c] MALTA, C., MARTINEZ, J. ; *Tuple-based abstract data types: full parallelism* ; Proceedings of the 7th Int. Symposium on Computer and Information Sciences, Kemer, Antalya, Turquie, novembre 1992, pp. 173-179

[Malta & Martinez 93] MALTA, C., MARTINEZ, J. ; *Automating fine concurrency control in object-oriented databases* ; Proceedings of the 9th IEEE Int. Conf. on Data Engineering, Vienne, Autriche, avril 1993

[Malta 93] MALTA, C. ; *Les systèmes transactionnels pour environnements d'objets : principes et mise en œuvre* ; Thèse d'université, Université Montpellier II, Sciences et techniques du Languedoc, à paraître

[Martin 87] MARTIN, B. E. ; *Modeling concurrent activities with nested objects* ; Proceedings of the 7th Int. Conf. on Distributed Computing Systems, Berlin, Allemagne, septembre 1987, pp. 432-439

[Masini et al. 89] MASINI, G., NAPOLI, A., COLNET, D., LÉONARD, D., TOMBRE, K. ; *Les langages à objets : langages de classes, langages de frames, langages d'acteurs* ; InterEditions, Paris, France, 1989

[Meyer 86] MEYER, B. ; *Genericity versus inheritance* ; Proceedings of the 1st ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA, septembre 1986, pp. 391-405

[Meyer 88] MEYER, B. ; *Object-oriented software construction* ; Prentice Hall International Ltd, Hemel Hempstead, Royaumes-Unis, 1988

[Mohan et al. 89] MOHAN, C., HADERLE, D., LINDSAY, B. G., PIRAHESH, H., SCHWARZ, P. M. ; *ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging* ; IBM Research Report, RJ 6649, 1989, 45 p.

[Mohan et al. 92] MOHAN, C., HADERLE, D., LINDSAY, B. G., PIRAHESH, H., SCHWARZ, P. M. ; *ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging* ; ACM Transactions On Database Systems, vol. 17, n° 1, mars 1992, pp. 94-162

[Moss 82] MOSS, J. E. B. ; *Nested transactions and reliable distributed computing* ; Proceedings of the 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, juillet 1982

[Moss 85] MOSS, J. E. B. ; *Nested transactions: an approach to reliable distributed computing* ; MIT Press Series in Information Systems, Michael Lesk, Editor, 1985, 160 p.

[Moss et al. 85] MOSS, J. E. B., GRIFFITH, N. D., GRAHAM, M. H. ; *Abstraction in concurrency control and recovery management* ; Technical report 85-51, Department of Computer and Information Sciences, University of Massachusetts, Amherst, décembre 1985, 24 p.

[Moss et al. 86] MOSS, J. E. B., GRIFFITH, N. D., GRAHAM, M. H. ; *Abstraction in recovery management* ; Proceedings of the ACM Int. Conf. on the Management Of Data, Washington D.C., USA, mai 1986, pp. 72-83

[Nakajima & Tokoro 90] NAKAJIMA, T., TOKORO, M. ; *Concurrency control and recovery on multiversion objects* ; Proceedings of the ECOOP/OOPSLA Workshop on Transactions and Objects, Ottawa, Canada, octobre 1990

[Ng 89] NG, T. P. ; *Using histories to implement atomic objects* ; ACM Transactions On Computer Systems, vol. 7, n° 4, novembre 1989, pp. 360-393

[Nguyen & Rieu 89] NGUYEN, G. T., RIEU, D. ; *Schema change propagation in object-oriented databases* ; Rapport de Recherche n° 1045, INRIA, Groupe de Recherche Grenoble, juin 1989, 11 p.

[Noe et al. 87] NOE, J. D., KAISER, J., KROGER, R., NETT, E. ; *The commit/abort problem in type-specific locking* ; Technical report 87-10-04, Department of Computer Science FR-35, University of Washington, Seattle WA 98195, USA, octobre 1987

[O'Neil 86] O'NEIL, P. E. ; *The Escrow transactional method* ; ACM Transactions On Database Systems, vol. 11, n° 4, décembre 1986, pp. 405-430

[Penney & Stein 87] PENNEY, D. J., STEIN, J. ; *Class modification in the GemStone object-oriented DBMS* ; Proceedings of the 2nd ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, octobre 1987

[Pistor & Andersen 86] PISTOR, P., ANDERSEN, F. ; *Designing a generalized NF^2 model with an SQL-type language interface* ; Proceedings of the 12th Int. Conf. on Very Large Data Bases, Kyoto, Japon, août 1986, pp. 278-285

[Plateau et al. 89] PLATEAU, D., CAZALENS, R., LÉVÊQUE, D., MAMOU, J.-C., POYET, B. ; *Building user interfaces with the LOOKS hyper-object system* ; GIP Altaïr Technical Report, mai 1989

[Pons 86] PONS, J.-F. ; *Algorithmes répartis de contrôle de cohérence : généralisation de la règle de Thomas* ; Rapport de recherche n° 27, CRIM, novembre 1986

[Posner & Baecker 92] POSNER, I. R., BAECKER, R. M. ; *How people write together* ; Proceedings of the 25th Annual Hawaii Int. Conf. on System Sciences, vol. IV, 1992, pp. 127-138

[Pugh 90] PUGH, W. ; *Skip lists: a probabilistic alternative to balanced trees* ; Communications of the ACM, vol. 33, n° 6, juin 1990, pp. 668-676

[Reuter 84] REUTER, A. ; *Performance analysis of recovery techniques* ; ACM Transactions On Database Systems, vol. 9, n° 4, décembre 1984, pp. 526-559

[Rieu et al. 91] RIEU, D., NGUYEN, G. T., CULET, A., ESCAMILLA, J., DJERABA, C. ; *Instanciation multiple et classification d'objets* ; VIIes Journées Bases de Données Avancées, Lyon, France, septembre 1991, pp. 51-69

[Roesler & Burkhard 87] ROESLER, M., BURKHARD, W. A. ; *Concurrency control scheme for shared objects: A peephole approach based on semantics* ; Proceedings of the 7th Int. Conf. on Distributed Computing Systems, Berlin, Allemagne, septembre 1987, pp. 224-231

[Roffé et al. 90] ROFFÉ, R., ANOTA, P., BOURGAIN, E. ; *G-BASE V4: the object-oriented DBMS for multimedia data and heterogeneous workstations* ; Proceedings of the 2nd Int. Conf. on Technology of Object-Oriented Languages and Systems, Paris, France, juin 1990, pp. 315-323

[Rosenkrantz et al. 78] ROSENKRANTZ, D. J., STERANS, R. E., LEWIS, P. M. ; *System level concurrency control for distributed database systems* ; ACM Transactions On Database Systems, Vol. 3, n° 2, juin 1978, pp. 178-198

[Rumbaugh 87] RUMBAUGH, J. ; *Relations as semantic constructs in an object-oriented language* ; Proceedings of the 2nd ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, USA, octobre 1987, pp. 466-481

[Schaffert et al. 86] SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., WILPOLT, C. ; *An introduction to Trellis/Owl* ; Proceedings of the 1st ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, septembre 1986

[Schek et al. 90] SCHEK, H.-J., PAUL, H.-B., SCHOLL, M. H., WEIKUM, G. ; *The DASDBS project: objectives, experiences, and future propects* ; IEEE Transactions On Knowledge And Data Engineering, vol. 2, n° 1, mars 1990, pp. 25-43

[Schwarz & Spector 84] SCHWARZ, P. M., SPECTOR, A. Z. ; *Synchronizing shared abstract types* ; ACM Transactions On Computer Systems, vol. 2, n° 3, août 1984, pp. 223-250

[Shasha & Goodman 88] SHASHA, D., GOODMAN, N. ; *Concurrent search structure algorithms* ; ACM Transactions On Database Systems, vol. 13, n° 1, mars 1988, pp. 53-90

[Shasha et al. 92] SHASHA, D., SIMON, E., VALDURIEZ, P. ; *Simple rational guidance ofr chopping up transactions* ; Proceedings of the ACM Int. Conf. on the Management Of Data, San Diego, Californie, USA, juin 1992

[Snyder 86] SNYDER, A. ; *Encapsulation and inheritance in object-oriented programming languages* ; Proceedings of the 1st ACM Int. Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA, septembre 1986, pp. 38-45

[Stefik & Bobrow 86] STEFIK, M., BOBROW, D. G. ; *Object-oriented programming: themes and variations* ; The AI magazine, janvier 1986, pp. 40-62

[Stonebraker & Kemnitz 91] STONEBRAKER, M., KEMNITZ, G. ; *The PostGres next generation database management system* ; Communications of the ACM, vol. 34, n° 10, octobre 1991, pp. 78-92

[Stonebraker et al. 87] STONEBRAKER, M., ANTON, J., HANSON, E. ; *Extending a database system with procedures* ; ACM Transactions On Database Systems, vol. 12, n° 3, septembre 1987, pp. 350-376

[Tarjan 72] TARJAN, R. E. ; *Depth-first search and linear graph algorithms* ; SIAM jounal of computing, vol. 1, n° 2, juin 1972, pp. 146-160

[Thomas 79] THOMAS, R. H. ; *A majority consensus approach to concurrency control* ; ACM Transactions On Database Systems, vol. 4, n° 2, juin 1979, pp. 180-209

[Thomasian & Ryu 91] THOMASIAN, A., RYU, I. K. ; *Performance analysis of two-phase locking* ; IEEE Transactions On Software Engineering, vol. 17, n° 5, mai 1991, pp. 386-401

[Thomasian 91] THOMASIAN, A. ; *Performance limits of two-phase locking* ; Proceedings of the 7th IEEE Int. Conf. on Data Engineering, Kobe, Japon, avril 1991, pp. 426-435

[Turc 89] TURC, S. ; *Formalisation de la correction des exécutions dans les systèmes transactionnels sujets aux pannes* ; Thèse d'Université, Université Montpellier II, Sciences et Techniques du Languedoc, mars 1989, 135 p.

[Ulusoy & Belford 91] ULUSOY, Ö., BELFORD, G. G. ; *An access control protocol for real-time database transactions* ; Proceedings of the 6th Int. Symposium on Computer and Information Sciences, vol. I, Side, Antalya, Turkey, novembre 1991, pp. 179-188

[Verhofstad 78] VERHOFSTAD, J. S. M. ; *Recovery techniques for database systems* ; ACM Computing Surveys, vol. 10, n° 2, juin 1978

[Wegner 90] WEGNER, P. ; *Concepts and paradigms of object-oriented programming* ; OOPS Messenger, vol. 1, n° 1, août 1990, pp. 7-87

[Weihl & Liskov 85] WEIHL, W. E., LISKOV, B. ; *Implementation of resilient, atomic data types* ; ACM Transactions On Programming Languages and Systems, vol. 7, n° 2, avril 1985, pp. 244-269

[Weihl 88] WEIHL, W. E. ; *Commutativity-based concurrency control for abstract data types* ; IEEE Transactions On Computers, vol. 37, n° 12, décembre 1988, pp. 1488-1505

[Weihl 89a] WEIHL, W. E. ; *Local atomicity properties: modular concurrency control for abstract data types* ; ACM Transactions On Programming Languages and Systems, vol. 11, n° 2, avril 1989, pp. 249-282

[Weihl 89b] WEIHL, W. E. ; *The impact of recovery on concurrency control* ; Proceedings of the 8th Int. Symposium on Principles Of Database Systems, Philadelphie, mars 1989, pp. 259-269

[Weikum 87] WEIKUM, G. ; *Enhancing concurrency control in layered systems* ; Proceedings of the Int. Conf. on High Performance Transaction Systems, Pacific Growe, septembre 1987

[Weikum 91] WEIKUM, G. ; *Principles and realization strategies of multilevel transaction management* ; ACM Transactions On Database Systems, vol. 16, n° 1, mars 1991, pp. 132-180

[Weikum et al. 90] WEIKUM, G., HASSE, C., BROESSLER, P., MUTH, P. ; *Multi-level recovery* ; Proceedings of the 9th Int. Symposium on Principles Of Database Systems, Nashville, Tennessee, USA, avril 1990, pp. 109-123

[Yannakakis 84] YANNAKAKIS, M. ; *Serializability by locking* ; Journal of the ACM, vol. 31, n° 2, avril 1984, pp. 227-244

[Zicari 89] ZICARI, R. ; *A framework for O_2 schema updates* ; GIP Altaïr Technical Report 38-89

[Zicari 91] ZICARI, R. ; *A framework for schema updates in an object-oriented database system* ; Proceedings of the 7th IEEE Int. Conf. on Data Engineering, Kobé, Japon, avril 1991, pp. 2-13

Table des figures

I.1 Un exemple d'histoire concurrence et de graphe des dépendances associé . . . 32
I.2 Exemple de graphe des granules . . . 38
I.3 Une instanciation du graphe des granules de la figure I.2 . . . 39
I.4 Principe de la journalisation . . . 43
I.5 Chaînage des enregistrements d'une transaction dans le journal . . . 46
I.6 Un exemple d'exécution dans un modèle de transactions à trois niveaux . . . 51

II.1 Graphe de la relation de compatibilité des verrous hiérarchiques . . . 65
II.2 « Théorème » sur les limites de la commutativité . . . 70
II.3 Limites de la commutativité sur les ensembles bornés . . . 75
II.4 Exemple d'opérations exclusives simultanées (!) sur l'ADT ENSEMBLE . . . 84
II.5 Composition d'objets multi-niveaux . . . 86
II.6 Phénomène de convergence sur l'ADT COMPTEUR . . . 89

III.1 Exemple de graphe d'héritage multiple . . . 101
III.2 Classes frontière d'un sous-graphe . . . 105
III.3 Intersection stricte entre deux sous-graphes . . . 107
III.4 Contre-exemple à un théorème de nécessité . . . 107
III.5 Inclusion de sous-graphes . . . 109
III.6 Du verrouillage explicite au verrouillage implicite . . . 111
III.7 Nécessité du verrouillage intentionnel avec le verrouillage implicite . . . 112
III.8 Exemple de verrouillages concurrents dans un graphe d'héritage . . . 117
III.9 Insertion non problématique d'une classe dans un graphe d'héritage avec seule suppression des arcs de transitivité . . . 122
III.10 Accès et modifications concurrents du graphe d'héritage . . . 126
III.11 Graphe des granules générique et une instanciation possible . . . 130
III.12 Graphe des privilèges associés à nos verrous . . . 132
III.13 Graphe des privilèges associés aux verrous d'ORION . . . 132

IV.1 Exemples de tables de résolution de la liaison dynamique . . . 143

IV.2 Quelques aspects de la programmation à objets 146
IV.3 Graphes de résolution des liaisons dynamiques pour les instances propres des classes c_1 et c_2 de la figure IV.2 157
IV.4 Verrouillage dans un graphe de résolution des liaisons dynamiques 169
IV.5 Deux transactions écrivains concurrentes dans un même sous-graphe 171

B.1 Un exemple d'exécution en cours 199
B.2 Représentation en machine du graphe de la figure B.1 204

C.1 Exemple de classes frontière 207

Liste des tableaux

I.1 Relation de commutativité en arrière de l'ADT COMPTEENBANQUE 25
I.2 Relation de commutativité en arrière à droite de l'ADT COMPTEENBANQUE . 26
I.3 Relation de commutativité en avant de l'ADT COMPTEENBANQUE 26
I.4 Relation de recouvrabilité relative de l'ADT COMPTEENBANQUE 27
I.5 Les opérations *Lire* et *Écrire* face aux trois critères 28
I.6 Relation de recouvrabilité relative de l'ADT PILE 35

II.1 Exemple de fonctions associées à une affectation 62
II.2 Fonctions associées aux opérations de l'ADT PILE 72
II.3 Relation de non-commutativité des fonctions sur l'ADT PILE 73
II.4 Une histoire concurrente 92
II.5 Un blocage définitif dû au critère d'atomicité dynamique 92

III.1 Compatibilité C_T des types de verrous sur les classes 109
III.2 Compatibilité C_M de quelques modes d'accès 111
III.3 Compatibilité C_I des intentions d'accès aux instances 112
III.4 Tous les verrous obtenus à partir des modes d'accès *L*, *E*, *LD* et *MD* 114
III.5 Correspondance entre les verrous d'ORION et les nôtres 130

IV.1 Relation de compatibilité « classique » 150
IV.2 Commutativité entre méthodes de la classe c_1 165
IV.3 Commutativité entre méthodes de la classe c_2 165
IV.4 Compatibilité des verrous sur les sommets des graphes de résolution des liaisons dynamiques 168
IV.5 Table de résolution des liaisons dynamiques et des vecteurs d'accès 174

Liste des algorithmes

I.1 Le type de données abstrait COMPTEENBANQUE 22
I.2 Une transaction de transfert bancaire 30
II.1 Une implémentation compatible de l'ADT ENSEMBLE 85
III.1 Procédures réalisant les points 2 et 3 du protocole 116
IV.1 Déclarations de types de classes en O_2C 141
IV.2 Un exemple de méthodes en O_2C 151
B.1 Calcul des vecteurs d'accès transitifs 191
B.2 Initialisation après modifications de méthodes (*VAD* et/ou Γ) 202
B.3 Initialisation en cas d'ajouts de nouvelles méthodes 203
C.1 Déclaration d'union de types en Trellis/Owl 207
C.2 Algorithme de calcul des classes frontière d'une union de domaines 207

Index

O_2C, 141, 143, 151

références bibliographiques

[Abbot 90], 3

[Abiteboul 89], 119

[Abiteboul & Bonner 91], 119, 184

[Acosta 91], 85

[Agrawal & DeWitt 85], 36, 60, 104

[Agrawal & El Abbadi 92], 139, 175, 176, 178, 182

[Agrawal & Gehani 89], 99, 172

[Agrawal et al. 87], 36, 60, 104

[Aho et al. 83], 189, 197, 202, 208

[Aldebert & Martinez 89], 14, 15, 113

[America & Van der Linder 90], 10

[Andrews & Harris 87], 12, 99, 140, 172

[Appelbe & Ravn 84], 3

[Atkinson et al. 89], 8, 10, 12

[Badrinaht & Ramamritham 87], 27

[Badrinath & Ramamritham 87], 19, 26, 90

[Badrinath & Ramamritham 88], 90

[Badrinath & Ramamritham 90], 50

[Badrinath & Ramamritham 92], 26, 34

[Bancilhon 88], 8

[Bancilhon et al. 89], 171

[Banerjee et al. 87a], 99, 103, 118, 121, 140

[Banerjee et al. 87b], 121

[Barthès 88], 11

[Beeri & Milo 91], 172

[Beeri et al. 88], 19, 50, 80, 115

[Beeri et al. 89], 50

[Benzaken 90], 86

[Benzaken & Delobel 89a], 86

[Berge 67], 70, 187

[Bernstein et al. 87], 28, 31, 35, 38, 59, 94

[Bernstein et al. 90], 19

[Bertino & Martino 91], 12

[Berztiss 75], 189

[Bloom & Zdonik 87], 12

[Boksenbaum et al. 84], 34, 35

[Boksenbaum et al. 86], 33, 104

[Bordat & Cogis 86], 190, 197

[Bouvier 82], 70

[Broessler & Freisleben 89], 25, 78

[Butterworth et al. 91], 140

[Carey & Livny 89], 60

[Cart & Ferrié 89a], 50

[Cart & Ferrié 89b], 8, 129, 133

[Cart & Ferrié 90], 8, 15, 115, 129, 133, 134, 139, 164, 175

[Cart et al. 89], 50

[Cart et al. 90a], 50

[Cart et al. 90b], 50, 86

[Chamberlin et al. 81], 119

[Chrisment & Zurfluh 89], 11

[Chrysanthis et al. 91], 73

[Dasgupta & Kedem 90], 170

[Dayal et al. 90], 172

[Dayal et al. 91], 172

[Delobel & Adiba 82], 3

[Diaz et al. 91], 172

[Eich 88], 170
[Elhardt & Bayer 84], 53
[Estier & Guyot 91], 120
[Eswaran et al. 76], 35, 37–39, 90, 114, 119, 165, 174
[Exertier 91], 11
[Fishman et al. 87], 99, 140, 184
[Franaszek & Robinson 85], 60
[Franklin et al. 92], 48
[Gardarin & Valduriez 90], 99
[Garza & Kim 88], 15, 38, 99, 113, 139, 164, 175, 181, 209
[Gehani & Jagadish 91], 172
[Goldberg & Robson 83], 100, 140
[Goodman & Shasha 85], 23, 94
[Gray 78], 7, 36–38, 40, 42, 62, 64, 104, 112, 119, 130, 133, 174
[Gray 81], 59, 62
[Gray et al. 81], 42, 80, 87
[Haerder & Reuter 83], 44
[Harel 80], 60
[Hasse & Weikum 91], 50
[Hermann et al. 90], 141
[Hesselink 88], 64
[Huang 85], 85
[Huang et al. 91], 19
[Hwang & Lee 91], 12
[Jhingran & Khedkar 92], 48
[Kedem & Silberschatz 83], 59
[Khoshafian & Copeland 86], 10
[Kim 90], 8, 176
[Kim et al. 87], 113, 142, 184
[Kim et al. 89], 113, 142, 184
[Kim et al. 90], 134, 177, 182
[Knapp 87], 37
[Kohler 81], 36, 62
[Korth 82], 38, 124, 170
[Korth 83], 38, 40, 110, 131, 133, 150, 174
[Korth et al. 88], 38, 147
[Krakowiak 85], 142
[Kumar & Stonebraker 89], 19
[Kung & Robinson 81], 35, 52
[Kuo 92], 48
[Lécluse & Richard 89a], 102
[Lécluse & Richard 89b], 118
[Lécluse et al. 88], 99, 140
[Lamb et al. 91], 140
[Lehman & Yao 81], 85
[Lerner & Habermann 90], 122
[Lohman et al. 91], 172
[Maier et al. 86], 99
[Malta 93], 14, 61, 70, 71, 90, 157, 172, 173, 178
[Malta & Martinez 90], 14, 15
[Malta & Martinez 91a], 14, 15
[Malta & Martinez 91b], 14, 61, 70, 71, 173
[Malta & Martinez 92a], 14, 15
[Malta & Martinez 92b], 14, 172, 178
[Malta & Martinez 92c], 14, 15, 157
[Malta & Martinez 93], 14
[Martin 87], 50, 86
[Masini et al. 89], 100
[Meyer 86], 11
[Meyer 88], 9
[Mohan et al. 89], 45, 47, 79
[Mohan et al. 92], 45, 46
[Moss 82], 29
[Moss 85], 29
[Moss et al. 85], 50, 78
[Moss et al. 86], 50, 78, 80, 81
[Nakajima & Tokoro 90], 24
[Ng 89], 24, 88
[Nguyen & Rieu 89], 122

[Noe et al. 87], 139, 172
[O'Neil 86], 24, 59, 88, 145
[Penney & Stein 87], 99, 121
[Pistor & Andersen 86], 11
[Plateau et al. 89], 9
[Pons 86], 28
[Pugh 90], 85
[Reuter 84], 42, 53
[Rieu et al. 91], 140, 184
[Roesler & Burkhard 87], 65, 73
[Roffé et al. 90], 140, 184
[Rosenkrantz et al. 78], 52, 80
[Rumbaugh 87], 12, 184
[Schaffert et al. 86], 99, 173, 206
[Schek et al. 90], 50
[Schwarz & Spector 84], 59, 87, 139, 145
[Shasha & Goodman 88], 85
[Shasha et al. 92], 33
[Snyder 86], 9
[Stefik & Bobrow 86], 3
[Stonebraker & Kemnitz 91], 8, 172
[Stonebraker et al. 87], 11
[Tarjan 72], 155, 158, 189, 190, 203
[Thomas 79], 28
[Thomasian 91], 60
[Thomasian & Ryu 91], 60
[Turc 89], 78, 81
[Ulusoy & Belford 91], 19, 170
[Verhofstad 78], 35, 36
[Wegner 90], 3, 10, 11, 149
[Weihl 88], 24, 63, 66, 67, 92, 94, 139, 145, 183
[Weihl 89a], 15, 57, 60, 91, 93, 94
[Weihl 89b], 25, 35, 93, 183
[Weihl & Liskov 85], 87, 145
[Weikum 87], 50
[Weikum 91], 50, 51, 78, 86
[Weikum et al. 90], 50, 53, 79
[Yannakakis 84], 52
[Zicari 89], 122, 159
[Zicari 91], 122, 159
2PL, 114

Abstract Data Type (ADT), 20
abstraction
  niveau (d'), 67
accès
  autorisation, 3
  chemin (d'), 105
  classe (à une), 103
  classes d'un sous-graphe (aux), 103
  concurrent, 104
  instance (à une), 103
  instances d'un sous-graphe (aux), 103
  mode (d'), 110, 113, 130, 159
  type (d'), 113
  virtuel, 128, 134
Ada, 11, 73
ADT, 100
  Étudiant3eCycle, 206
  Abstract Data Type, 20
  AllocataireRecherche, 206
  AMap, 145
  Annuaire, 83
  B-arbre, 173
  BoîteAuxLettres, 83
  Booléen, 66
  Carré, 143, 149, 160, 162, 174
  Cercle, 143, 160, 162
  Collection, 145
  ComposantÉlectronique, 10
  CompteEnBanque, 21, 22, 25–27, 35, 183
  Compteur, 81, 88, 89, 93–95, 137, 145, 181
  Ellipse, 149, 162
  Employé, 150, 171

Enfant, 119
Ensemble, 23, 61, 66, 67, 71, 83–85, 91, 95, 124, 127, 145, 175, 181
Entier, 21
Famille, 83
Femme, 11
FichierÉditable, 107
File, 20
FileFaible, 87
FileFIFO, 87
Formulaire, 102
Homme, 11
Indexé, 107
Liste, 83
Moniteur, 206
n-uplet, 90
Object, 100, 110, 123
PatientCritique, 120
Personne, 49, 119, 144
Pile, 20, 35, 71–75, 81, 87, 88, 92, 93, 145
Point, 143, 160, 162, 174
Programmeur, 150
Répertoire, 59, 71, 75, 145
Rectangle, 10
Relation, 83, 145
SemiFile, 87
algorithme
incrémental, 137
linéaire, 159
ancêtres (Γ^{-1*}), 187
arborescence, 100
arc, 187
entrant, 187
sortant, 187
ARIES, 45, 54, 79
associativité, 152, 158, 190
atomicité, 5, 6, 21, 44, 45, 53, 79
dynamique, 91, 93
ligne (en), 92, 93
hybride, 91
statique, 91
atteignabilité, 12
auto-commutativité, 51
autorisation d'accès, 3

base de données à objets, 97, 99, 137
bureautique, 11

C, 9, 176
C++, 143, 144
CAO, 11
cartographie, 11
certification, 104
champs, 140, 145
chemin, 101, 187
accès (d'), 105
sous-graphe (de), 130
circuit, 187
classe, 9, 10, 12, 100, 140
chemin (de), 105, 109
création (de), 121, 124
descendante, 102, 160
destruction (de), 121, 124
frontière, 105, 109, 129
isolée, 109
méta-, 100, 140
propre, 102, 142
virtuelle, 119
clique, 187
CLR (Compensation Log Record), 46
COBOL, 9
commutativité, 17, 23, 57, 59, 137, 152, 158, 190
arrière (en), 24, 92
droite (à), 25, 67, 93

auto-, 51
avant (en), 24, 66, 92
conditionnelle, 24
méthodologie d'extraction, 73
mathématique, 88
non-, 75
opérations typées (des), 19, 24
compatibilité, 19, 57, 59, 150
Compensation Log Record (CLR), 46
complétude, 12
composante fortement connexe, 158, 187
composite (objet), 113, 129, 142
conception assistée par ordinateur, 11
condition
commutativité de l'état (de), 63
composition (de), 61
indépendance des vues (d'), 64
inversibilité (d'), 62
conflit, 6, 59
ordre, 51, 56
pseudo, 59, 147
consistance, 5
base de données (d'une), 59, 104
constructeur de type, 141
contrôle de concurrence, 6, 8, 12, 17, 57, 97, 104, 137
contrainte d'intégrité, 3, 5, 6, 120, 172
convergence, 70, 71, 75, 88

DB-Cache, 53
déduction de valeur, 70
dépendance, 27, 59
cyclique, 158, 190
fonctionnelle, 90
descendants (Γ^*), 103, 187
descente
ordre, 158, 190
profondeur d'abord (en), 158
disque miroir, 7, 42
domaine, 103
donnée
base (de), 99
type abstrait (de), 10, 20, 57, 100
durabilité, 5, 6, 45, 53

écriture
non forcée, 45
non retardée, 44
efficacité, 137
encapsulation, 9, 12, 20, 100, 142, 171
End Of Transaction (EOT), 54
ensemble
arrivée (d'), 61
borné, 73
départ (de), 61
fini, 73
envoi de message, 103, 140, 142
EOT (End Of Transaction), 54
équi-efficacité, 93
équivalence d'état, 63
escalade de verrou, 147
espace de travail, 66, 77
estampillage, 104
exécution
concurrente, 59
parallèle, 59
séquentielle, 59
stricte, 67
exclusion mutuelle, 79
extensibilité, 12
extension, 100

factorisation de code, 146
famille de fonctions, 64
fantôme (objet), 39, 97, 118, 134
fermeture réflexo-transitive, 156, 157, 187

fiabilité, 3, 7
fonction
famille, 64
inverse à gauche, 62

G-BASE, 129, 140
généralisation, 15
GemStone, 99, 120, 129, 140
gestion transactionnelle, 3
graphe
dépendances entre méthodes (des), 159
granules (de), 38, 130
héritage (d'), 97, 100, 104, 130
Hasse des privilèges (de), 131
orienté, 187
partiel, 187
résolution des liaisons dynamiques (de), 156
sous-, 102, 160, 187

héritage, 9, 10, 12, 100, 137
arc (d')
création (d'un), 121, 124
destruction (d'un), 121, 124
classe (de), 12
contrainte (par), 10
graphe (d'), 97, 130
inclusion (par), 10
multiple, 100, 105, 140, 149
ordre, 122
relation (d'), 140
simple, 100
substitution (par), 10
type (de), 12
histoire, 31

IBM, 45, 115
idempotence, 44, 47, 152, 158, 190
identité d'objet, 9, 10, 12
image avant, 62
impedance mismatch, 9
inclusion, 68, 105
indépendance, 83, 89, 133
indivisibilité, 21
instance, 102, 140
générale, 102, 104, 128, 160
propre, 102, 104, 160
variable (d'), 140, 145
instanciation
domaine (d'), 103
graphe (d'un), 40
graphe de granules (d'un), 38, 130
mono-, 140
multi-, 120, 140, 184
schéma (d'un), 121
interblocage, 78, 115, 147, 171
interface, 173
deux niveaux (à), 71, 173
utilisateur, 9
intersection, 68
stricte, 105, 106
invariance, 89
paramètres de sortie (des), 63
IRIS, 99, 129, 140
isolation, 5, 6

journalisation, 17

langage
O_2C, 141, 143, 151
Ada, 11, 73
C, 9, 176
C++, 143, 144
COBOL, 9
Lisp, 176
Pascal, 9
Smalltalk, 100, 140, 142–144
SQL, 12

latticiel, 100
liaison dynamique, 9, 11, 12, 137, 140, 142, 147, 149, 154
lien de composition, 146
Lisp, 176
liste d'intentions, 66
Log Sequence Number (LSN), 46
LSN (Log Sequence Number), 46

mémoire, 64
  centrale, 7, 8, 44, 79
  masse (de), 44
  secondaire, 12
  stable, 7
message
  auto-dirigé
    direct, 153
    préfixé, 154
  envoi (de), 103, 140, 142
méta-classe, 100, 102, 140
méthode, 100, 140
  certification (par), 6
  continue, 6
  optimiste, 6
  pessimiste, 6
mise-à-jour
  différée, 24
  immédiate, 21, 24, 34, 44, 66
modèle
  formel, 60
  relationnel, 99, 100
    NF^2, 11
mode d'accès, 110, 113, 130
  spécifique à chaque méthode, 159
MONADS, 78
mono-instanciation, 140
monotonie, 70
multi-instanciation, 140, 184
multi-version, 67
multiplicité, 122

n-uplet, 90, 99, 140
niveau d'abstraction, 67
non-commutativité, 25, 73
non-déterminisme, 64, 87, 89
  opération, 64

O_2, 8, 10, 14, 15, 99, 100, 115, 118, 120, 122, 129, 133, 135, 140, 159, 176, 182
Object IDentifier, 64
ObjectStore, 140
objet
  base de données (à), 99
  complexe, 12, 141
  composite, 113, 129, 142
  fantôme, 39, 97, 118, 134
  identité (OID), 9, 10, 12
  notion (d'), 3
ODE, 99
OID, 9, 10, 12, 54, 64, 118, 142
opération
  absorbante, 28
  inverse, 21
  non déterministe, 64
ORACLE, 10
ordre
  chronologique, 34, 54
    inverse, 34, 54
  conflits (des), 51, 56
  descente (de la), 158, 190
  héritage (d'), 122
  histoire (d'une), 30
  libération (de), 115
  partiel, 29, 91, 100, 101
  sérialisation (de), 5, 28, 34, 37, 92
  total, 29

validation, 23
validation (de), 34, 87, 92
ORION, vii, 15, 99, 100, 113, 117, 118, 120, 129–135, 140, 142, 176, 177, 181, 182, 209

panne
reprise, 6, 79
paradigme
classes (des), 140
envoi de message (de l'), 142
objet (à), 9
repeating history (du), 79
parallélisme intra-transaction, 29
paramètre
entrée (d'), 21, 24, 63
sortie (de), 21, 24, 63
invariance, 63
partage, 8
Pascal, 9
performance, 104
transactionnelle, 7
persistance, 8, 12
polymorphisme, 149
prédécesseurs (Γ^{-1}), 101, 187
prédicat de compatibilité entre
intentions d'accès, 112
modes d'accès, 110
types de verrous, 109
verrous de classes, 113
productivité, 9
protocole, 114, 123, 168
2PL strict, 97
biaisé, 124
verrouillage à deux phases strict (de), 37
verrouillage hiérarchique (de), 40, 64
prototype, 70
pseudo conflit, 59, 147
puits, 187

racine, 101, 105, 187
recouvrabilité
relative, 17, 19, 26, 90
simple, 26, 28
redémarrage, 6, 79
règle, 172
Thomas (de), 28
rejet, 28
relation, 99, 100
compatibilité (de), 150
est-une-partie-de, 142
héritage (d'), 140
multivoque, 101
ordre (d'), 150
ordre partiel (d'), 100
sous-typage (de), 102
reprise, 6, 12, 17, 57
panne (sur), 6, 79
multi-niveau, 79
requête, 171

satisfiabilité, 119
section finissante, 103
sécurité, 3, 8
sémantique, 59, 72
sérialisabilité, 5, 17, 19, 57, 59, 64, 104, 110
SGBDO
G-BASE, 129, 140
GemStone, 99, 120, 129, 140
IRIS, 99, 129, 140
O_2, 8, 10, 14, 15, 99, 100, 115, 118, 120, 122, 129, 133, 135, 140, 159, 176, 182
ObjectStore, 140
ODE, 99
ORION, vii, 15, 99, 100, 113, 117, 118, 120, 129–135, 140, 142, 176, 177, 181,

182, 209
Trellis/Owl, 99, 206
V-Base, 99, 129, 140
SGBDR
ORACLE, 10
System R, 42, 50, 80, 90, 115, 119, 131, 147, 174, 175
Smalltalk, 100, 140, 142–144
sommet, 187
source, 187
sous-classe, 100
sous-graphe, 102, 187
classes descendantes (des), 160
partiel, 187
spécification, 23
SQL, 12
stabilité, 7
successeurs (Γ), 101, 187
super-classe, 100
surcharge, 9, 11, 12, 137, 140, 147, 149
système
communication (de), 19
exploitation (d'), 19
multi-utilisateur, 59
temps-réel, 19
System R, 42, 50, 80, 90, 115, 119, 131, 147, 174, 175

théorie, 57
Thomas
règle (de), 28
transaction, 5, 17, 28, 59, 104
deux niveaux (à), 86
emboîtée, 29
gestion, 3
interaction, 59
multi-niveau, 17, 19, 67, 86, 115
parallélisme intra, 29
performance, 7
plate, 19
rejet, 28
validation, 28
treillis, 150
Trellis/Owl, 99, 206
trigger, 172
type
accès (d'), 113
constructeur, 141
données abstrait (de), 10, 20, 57, 100

V-Base, 99, 129, 140
valeur déduite, 70
validation, 28
variable d'instance, 140, 145
vecteur d'accès
chemin (de), 162, 163
complet, 162
direct, 148, 152
sous-graphe (à un), 162, 163
sous-graphe (de), 162
transitif, 148, 157
verrou
chemin (de), 109
classe (de), 109, 112
court, 115
escalade (de), 147
frontière (de), 109
instance (d'), 113
long, 115
verrouillage, 97, 104
2PL, 37, 78
court, 79
deux phases (à), 37, 114
strict, 17, 37, 137
hiérarchique, 111, 118
protocole (de), 40, 64

implicite, 111
intentionnel, 111
objet composite (d'un), 113
prédicatif, 39
sous-graphe (de), 104
vue, 63

WAL (Write-Ahead Logging), 42
WiSS (Wisconsin Storage System), 129
Write-Ahead Logging (WAL), 42

Contribution aux problèmes de contrôle de concurrence et de reprise dans les bases de données à objets

Résumé

Le contrôle des accès concurrents à des données partagées et manipulées simultanément par plusieurs utilisateurs, ainsi que la reprise lors de pannes, sont des composantes essentielles de tout système de gestion de données actuel, notamment des bases de données.

Nous mettons en évidence les limites inhérentes au critère le plus important du domaine : la commutativité des opérations sur les types de données abstraits. Ces limites nous confortent dans l'utilisation de techniques relativement simples dans les deux travaux ultérieurs.

Ensuite, nous proposons une technique de contrôle de concurrence adaptée aux bases de données à objets car s'appuyant sur le graphe d'héritage. Cette proposition s'avère être une généralisation de la seule méthode implémentée : le protocole d'ORION.

Combinant les limites théoriques de la commutativité à la notion de méthodes, qui encapsulent le comportement des objets, nous proposons une technique simple, et dans une large mesure suffisante, pour déterminer la commutativité entre méthodes. Cette dernière proposition combine les avantages de la technique précédente à ceux obtenus, involontairement, par la décomposition en première forme normale dans les bases de données relationnelles.

Mots clés

Gestion transactionnelle, sérialisabilité, commutativité, contrôle de concurrence, reprise, bases de données à objets.

A contribution to concurrency control and recovery in object-oriented databases

Abstract

Controlling concurrent accesses to shared data manipulated simultaneously by several users, as well as recovering from rejects and failures, are essential components of any current data system, especially databases.

We highlight the inherent limits of the main criterion of the field : commutativity of operations on abstract data types. These limits convince us to rely on simple techniques, especially in the forthcoming works.

Next, we propose a concurrency control technique fitted to object-oriented databases since it uses the inheritance graph. This proposition generalizes the only one which, to our knowledge, has been implemented : the protocol of ORION.

Combining theoretical limits of commutativity with the notion of methods, which encapsulate the behaviour of objects, leads us to a simple, yet sufficient, technique for determining commutativity of methods. This last proposal combines the advantages of the previous proposition with the ones obtained, indirectly, by the first normal form constraint in relational databases.

Keywords

Transaction management, serializability, commutativity, concurrency control, recovery, object-oriented databases.