Contributions à l'étude de la tension entre cohérence et confidentialité et du classement d'objets selon leur histoire dans les bases de données

Xavier Delannoy

THESE

présentée par

Xavier DELANNOY

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier – Grenoble 1

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : Informatique

Contributions à l'étude de la tension entre cohérence et confidentialité & du classement d'objets selon leur histoire dans les bases de données

Thèse soutenue devant la Commision d'Examen le :

12 Septembre 1997

MM	C.	CHRISMENT	Président - Rapporteur
	L.	LAKHAL	Rapporteur
	M.	SIMONET	Directeur
Me	A-M.	SIMONET	Co-Directeur
MM	C.	DELOBEL	
	C.	DEL VIGNA	Examinateurs
	J.	DEMONGEOT	

Thèse préparée au laboratoire Techniques de l'Imagerie, de la Modélisation et de la Cognition (TIMC) de Grenoble

Remerciements

Au moment de présenter ce travail, qu'il me soit tout d'abord permis de remercier Monsieur Claude Chrisment, Professeur à l'Université de Toulouse de me faire l'honneur de présider cette Commission d'Examen. Qu'il me soit aussi permis de le remercier ainsi que Monsieur Lotfi Lakhal, Professeur à l'Université de Clermont-Ferrand, d'avoir accepté d'être les Rapporteurs de mon travail.

Madame Ana-Maria Simonet et Monsieur Michel Simonet m'ont offert le cadre scientifique de cette thèse. Ils m'ont conseillé et encouragé tout au long de ce travail. Je les en remercie particulièrement et espère avoir été digne de la confiance qu'ils m'ont spontanément témoignés en ayant accepté d'être respectivement co-Directeur et Directeur de ma thèse.

Ce travail doit aussi beaucoup à Monsieur Claude Del Vigna. Toujours disponible et patient, il m'a conseillé dans un esprit de rigueur scientifique. Nos nombreuses discussions, parfois vives, mais toujours stimulantes, ont, je le pense, contribué à former mon esprit à l'exercice de la recherche et ont nourrit ma réflexion sur les bases de données. Je l'en remercie vivement.

Mes remerciements vont aussi à Messieurs Claude Delobel, Professeur à l'Université Paris Sud, et Jacques Demongeot, Professeur à l'Université de Grenoble, pour avoir accepté d'être Examinateurs.

Je tiens aussi à remercier Monsieur Phillipe Courrège pour ses apports sur certains aspects formels ainsi que Madame Suzanne Graf et Messieurs Jim Melton et Simon Wiseman.

Je n'oublie pas, enfin, mes collègues de l'équipe Osiris et plus généralement du laboratoire TIMC pour leur présence et leur soutien cordial.

Grenoble, le 12 Septembre 1997.

Résumé et Mots-Clés

Cette thèse est composée de deux contributions à l'étude des bases de données :

(*i*) la première contribution porte sur l'amélioration de la compréhension, par l'étude formelle, de la tension entre les fonctionnalités de cohérence et de confidentialité. Cette tension permet, dans certaines situations, d'utiliser les contraintes d'intégrité (cohérence) pour révéler des secrets (confidentialité) et donc réaliser des fraudes. L'étude fixe tout d'abord un cadre général de recherche en donnant une définition formelle des notions de secret, révélation et fraude. Puis, une occurrence particulière, et originale, de tension est formalisée selon une méthode inspirée des méthodes de programmation. Cette occurrence s'est avérée liée aux treillis de Galois.

(*ii*) la deuxième contribution porte sur la spécification et l'implémentation d'une fonctionalité originale : le classement d'objets selon leur histoire. A cette fin, l'étude répond successivement aux trois questions : qu'est-ce que l'histoire d'un objet, comment exprimer des propriétés sur l'histoire des objets, et comment les vérifier efficacement ? L'expression est réalisée par des formules de logique temporelle et la méthode de vérification repose sur la traduction de ces formules en expressions régulières puis en automates d'états finis. L'implémentation réalisée utilise cette méthode de classement pour classer *a posteriori* des objets du langage prototypique NewtonScript.

Mots-clés : Bases de Données, Contraintes d'Intégrité, Sécurité, Canal Caché, Treillis de Galois, Logique Temporelle, Classement, Langages Prototypiques.

This thesis is made of two contributions to the sudy of databases :

(*i*) the first contribution concerns the improvement of the understanding, by means of formal study, of the tension between the functionalities of consistency and confidentiality. This tension allows us in some situations to use integrity constraints (consistency) to reveal secrets (confidentiality) and therefore perform a fraud. To begin with, the study states a general framework by giving a formal definition of the notions of secret, revelation and fraud. Then, a particular and original occurrence of the tension is formalised following a method inspired from the programming methods. This occurrence proves to be connected with Galois lattices.

(*ii*) the second contribution concerns the specification and the implementation of an original functionality : classification of objects on the basis of their history. The study successively answers the three questions : what is the history of an object, how do we express properties on object histories, and finally how do we check them efficiently ? Expression is done by temporal logic formulas and checking relies on their successive translation to regular expression and then to finite state automaton. The implementation carried out uses this classification scheme to classify *a posteriori* objects of the prototype-based language NewtonScript.

Keywords : Databases, Integrity Constraints, Security, Covert Channel, Galois Lattices, Temporal Logic, Classification, Prototype-based Languages.

Table des Matières

Chapitre 2 : Vérification des Contraintes d'Intégrité

Conclusion 51

Deuxième Partie

De la *Tension* entre Cohérence et Confidentialité

—

Secrets Révélations et Fraudes, un Cadre Général et l'Etude d'un Cas Particulier

Troisième Partie

Classement d'Objets selon leur Historique

Logique Propositionnelle Temporelle,
Expressions Régulières et
Automates d'Etats Finis

Chapitre 11 : Implémentation

Introduction Générale

Les systèmes de gestion de base de données sont une catégorie de systèmes pour la programmation (*programming system*), au même titre que les compilateurs et les systèmes d'exploitation (Ullman, 1988, p 1). Ils offrent au programmeur un corpus de fonctionnalités pour la gestion de données persistantes.

Les travaux de recherche sur les bases de données concernent en général une fonctionnalité donnée. En revanche, l'étude de l'interaction de ces fonctionalités entre elles a été peu abordé. Elles peuvent pourtant être contradictoires dans certaines situations. C'est le cas des fonctionalités de cohérence et de confidentialité. (Mazumdar, Stemple, Sheard, 1988) parle d'une *tension* entre ces deux fonctionnalités.

La première contribution de cette thèse est d'apporter une meilleur compréhension, par l'étude formelle, de la tension entre cohérence et confidentialité. Pour cela, un cadre général, qui définit formellement les notions essentielles, est tout d'abord proposé. Puis, une occurrence particulière, et originale, de cette tension est formalisée selon une méthode inspirée des méthode de programmation. Cette étude reprend et étend (Delannoy, Del Vigna, 1998), (Delannoy, Del Vigna, 1997), (Delannoy, Del Vigna, 1996) et (Delannoy, 1996).

Le corpus de fonctionnalités qu'offrent les systèmes de gestion de base de données n'est pas figé. Ainsi, au-delà des fonctionnalités habituelles (langage de requête, cohérence, confidentialité, …) d'autres, plus originales, peuvent être envisagées, par exemple en vue de types particuliers d'applications.

La deuxième contribution de cette thèse est la spécification et l'implémentation d'une telle fonctionnalité : le classement d'objets dans des vues selon leur histoire. On répond, pour cela, successivement aux trois questions : qu'est-ce que l'histoire d'un objet, comment exprimer des propriétés sur l'histoire des objets, et comment les vérifier efficacement ? Les premiers éléments de ce travail ont été présentés dans (Delannoy, Simonet, Simonet, 1996).

Le point commun entre ces deux contributions est la place qu'y occupent les contraintes d'intégrité. Dans l'étude de la tension entre cohérence et confidentialité elles expriment les critères de cohérence. Dans celle du classement d'objet, leurs méthodes de vérification ont inspiré celle de vérification des propriétés sur l'histoire des objets. Les contraintes d'intégrité constituent en ce sens le cadre d'étude de la thèse et sont donc tout d'abord présentées. Cet état de l'art est basé sur (Delannoy, 1994).

Première Partie

Les Contraintes d'Intégrité dans les Bases de Données

—

Un Etat de l'Art

Introduction

Les bases de données stockent des données provenant du « monde réel » mais ne permettent pas d'assurer une biunivocité totale entre leur contenu et celui-ci. En revanche, elles permettent d'assurer que certaines *préconditions* à cette biunivocité sont réunies. La cohérence, parfois appelée intégrité sémantique, en est une. La sérialisation, ou intégrité transactionnelle, en est une autre.

Un état de la base de données est cohérent s'il vérifie les critères de cohérence définis sur la base de données. Ces critères s'expriment par des contraintes d'intégrité. Celles-ci sont exhibées lors de la conception de la base et traduisent des invariants sur les données (Mazumdar, Stemple, Shread, 1988). Par exemple, l'âge d'une personne ne peut dépasser 150.

S'assurer de la validité des contraintes d'intégrité permet donc, dans une certaine mesure, d'éviter que les données stockées soient aberrantes. Elles protègent par conséquent, en partie, des mises à jour erronées dues à l'inattention ou au manque d'information de l'utilisateur, ou encore à des programmes bogués.

Cette première partie présente un état de l'art sur les contraintes d'intégrité dans les bases de données.

Les contraintes d'intégrité sont étudiées depuis le milieu des années soixante dix, mais la plupart des systèmes actuels n'en offrent qu'un support limité. La nécessité d'un support aisé et efficace des contraintes d'intégrité a cependant été citée comme un des problèmes majeurs pour les systèmes de la prochaine génération (Plexousaki, 1993) (Jeusfeld, Jarke, 1991). L'amélioration, certes progressive, de leur prise en compte dans les versions successives de la norme SQL illustre bien cette tendance (comparaison entre (SQL, 1992) et (SQL, 1986)).

Il est généralement admis que l'étude des contraintes d'intégrité recouvre deux aspects : *l'expression* et la *vérification*.

Le premier chapitre traite de l'expression des contraintes d'intégrité. Après avoir rappelé les avantages attachés à leur expression déclarative, on en présente la typologie généralement admise. Elle distingue les contraintes statiques et les contraintes dynamiques. Les contraintes statiques permettent d'exprimer des critères de cohérence d'un état par des propriétés sur cet état, tandis que les contraintes dynamiques permettent aussi de prendre en compte les états passés. Puis, on aborde brièvement la spécification, au niveau des contraintes, des actions à entreprendre dans le cas où elles ne sont pas vérifiées (opérations de réparation).

Le deuxième chapitre, quant à lui, présente les différentes méthodes de vérification de contraintes d'intégrité. La question de la vérification se pose toujours en termes de performances. Comme dans le premier chapitre, on distingue successivement le cas des contraintes statiques puis celui des contraintes dynamiques. La vérification des contraintes statiques a été largement étudiée et on ne présente que les méthodes les plus significatives. En revanche, il existe comparativement peu de travaux sur la vérification des contraintes dynamiques.

Les contraintes d'intégrité ont principalement été étudiées dans le cadre du modèle relationnel. C'est seulement récemment que l'intérêt s'est porté sur les modèles objet. Et encore, s'agit-il en général simplement d'adapter des résultats élaborés dans le cadre du modèle relationel. C'est pourquoi, nous retenons ce dernier comme cadre d'etude. Certains paragraphes, néanmoins, contiennent des remarques relatives aux modèles objet.

Chapitre 1

Expression des Contraintes d'Intégrité

Ce chapitre traite de l'expression des contraintes d'intégrité. Après avoir rappelé les avantages attachés à leur expression déclarative, § 1*, on en présente la typologie généralement admise. Elle distingue les contraintes statiques,* § 2*, et les contraintes dynamiques,* § 3*. Puis, on aborde brièvement la spécification, au niveau des contraintes, des actions à entreprendre lorsqu'elles ne sont pas vérifiées,* § 4.

1. Expression déclarative des contraintes d'intégrité

Les contraintes d'intégrité sont habituellement exprimées déclarativement. Spécifier les contraintes déclarativement au lieu d'intégrer les vérifications appropriées dans les programmes présente de nombreux avantages (Chomicki, 1992) :

- le concepteur de la base peut se concentrer sur la détermination des contraintes plutôt que sur leur vérification.
- les programmes applicatifs sont plus simples et plus modulaires grâce à la factorisation des aspects relatifs à la vérification des contraintes. Les mêmes tests n'ont plus à être écrits dans les différents programmes utilisant la base.
- on est sûr que les mêmes contraintes sont vérifiées dans tous les programmes, même s'ils n'ont pas été écrits par le même programmeur.

- il est possible d'automatiser la vérification des contraintes, à l'instar des langages d'interrogation déclaratifs qui ont permis l'automatisation de l'optimisation de l'évaluation des requêtes[1].

- la modification des contraintes ne nécessite pas de modifier les programmes applicatifs. Cet argument, souvent cité, semble néanmoins devoir être nuancé. Soit, par exemple, un programme dont une instruction effectue une division arithmétique par une donnée x de la base de données. Le changement de la contrainte $x \neq 0$ en $x \neq 1$ implique que désormais, il est possible que l'exécution du programme engendre une division par 0. Il en découle que le programme doit être modifié pour tester la nullité de x avant le calcul de la division.

L'expression déclarative des contraintes d'intégrité est apparue avec le modèle relationnel. Celui-ci étant équivalent à la logique du premier ordre sans symbole de fonctions, les contraintes ont été naturellement exprimées dans ce formalisme. Seule l'expression déclarative des contraintes est considérée dans la suite.

La typologie classique des contraintes d'intégrité distingue les contraintes statiques et les contraintes dynamiques. On présente maintenant, dans cet ordre, ces deux catégories de contraintes.

2. Contraintes d'intégrité statiques

2.1 Définition

Les contraintes d'intégrité statiques permettent d'exprimer les critères de cohérence d'un état de la base de données par des propriétés sur cet état. Ce sont les contraintes les plus étudiées.

Dans les paragraphes qui suivent, on présente tout d'abord la manière générale d'exprimer les contraintes statiques, puis on étudie plus précisément celles fréquemment abordées dans la littérature.

2.2 Expression des contraintes d'intégrité statiques

Les contraintes d'intégrité statiques sont habituellement exprimées par des formules de la logique du premier ordre. (Kung, 1985) et (Morgenstern, 1984) constituent de rares exceptions. Le premier introduit une logique multitypée (multityped logic). Le deuxième décrit un langage déclaratif particulier pour l'expression de contraintes sur

[1] Remarquons que contraintes d'intégrité et langages de requêtes sont deux concepts relativement proches : une contrainte d'intégrité peut être vue comme une requête devant rendre un résultat vide.

des relations qui ne sont pas en première forme normale. Les contraintes faisant appel aux agrégats sortent aussi du cadre de la logique du premier ordre et sont présentées au § 2.7.

On donne maintenant quelques exemples de contraintes d'intégrité exprimées en logique du premier ordre sur le schéma relationnel **S** suivant[2]:

EMPLOYE(n°employé, âge, salaire, n°département)
POSSEDE(n°employé, compétence)
PROJET(n°projet, début_prévu, fin_prévue)
ASSIGNE(n°employé, n°projet)
REQUIERT(n°projet, compétence)
DEPEND(avant projet, après projet)
DEPARTEMENT(n°département, nom_département)

La relation **POSSEDE** associe aux employés leurs compétences. La relation **REQUIERT** associe aux projets les compétences requises. Enfin, la relation **DEPEND** décrit le graphe de dépendance des projets dans le temps.

La contrainte:

CI_1 : si l'âge d'un employé est supérieur à 40 son salaire doit être supérieur (strictement) à 20000.

s'exprime en logique du premier ordre de la manière suivante:

CI_1 : $\forall$ x,y,a,b (EMPLOYE(a,x,y,b) $\wedge$ x>40) $\Rightarrow$[3] y>20000

En SQL (SQL, 1992), les contraintes d'intégrité peuvent être déclarées lors de la création des tables. Pour la contrainte **CI_1**, l'instruction de création de la table **EMPLOYE** est alors:

```
CREATE TABLE EMPLOYE (
        n°employé INTEGER, âge INTEGER,
        salaire INTEGER, n°departement INTEGER,
   CONSTRAINT CI_1
     CHECK âge≤40 OR salaire>20000)
```

Alternativement, on peut aussi définir la contrainte hors du **CREATE TABLE** comme une simple assertion :

---

[2] Les attributs soulignés désignent les clés des relations, voir § 2.3.

[3] Dans toute la thèse on note l'implication logique par $\Rightarrow$ et l'équivalence logique par $\Leftrightarrow$. Le symbole $\rightarrow$ est réservé dans la partie 1 à l'expression des dépendances fonctionnelles et multivaluées et dans les parties 2 et 3 aux notations fonctionnelles.

```
CREATE ASSERTION CI_1
   CHECK
      ( SELECT COUNT(*)
        FROM EMPLOYE
           WHERE âge > 40 AND salaire ≤ 20000 ) = 0
```

On considère maintenant d'autres contraintes plus complexes:

CI_2 : la date prévue de début d'un projet ne peut avoir lieu plus tôt que celle de fin de n'importe quel autre projet dont il dépend. (Il s'agit d'une contrainte de type PERT[4])

CI_2 : ∀ x,y,deb_x,fin_y,a,b PROJET(x,deb_x,a) ∧ DEPEND(y,x) ∧ PROJET(y,b,fin_y) ⇒ (deb_x ≥ fin_y)

CI_3 : l'ensemble des compétences des employés travaillant pour un projet contient celles requises pour le projet.

CI_3 : ∀ x,z,a,b PROJET(z,a,b) ∧ REQUIERT(z,x) ⇒ (∃y ASSIGNE(y,z) ∧ POSSEDE(y,x))

Il est courant de regrouper les contraintes d'intégrité statiques en trois grandes catégories (Delobel, Adiba, 1982, pp 281-282):

- contraintes *individuelles*. Elles doivent être vérifiées par chaque tuple individuellement. **CI_1** en est un exemple. Il existe plusieurs types de contraintes individuelles. On les illustre sur la relation **EMPLOYE** : plage de valeurs (**0 < salaire < 20000**), liste de valeurs possibles (**nom_département ∈ {R&D,vente}**), lien entre attributs (**âge < salaire**), … Ces types peuvent être combinés comme dans **CI_1**.

- contraintes intra-relation. Elles portent sur un sous-ensemble des tuples d'une relation. La contrainte **CI_9** présentée au § 2.7, en est un exemple.

- contraintes inter-relations. Elles portent sur plusieurs relations. **CI_2** et **CI_3** en sont des exemples.

2.3 Contraintes de clé

[4] La méthode PERT (Program Evaluation and Review Technique) est une méthode d'ordonnancement des tâches. Elle est en particulier appliquée dans la planification des grands projets.

Une contrainte de clé porte sur un ou plusieurs attributs d'une relation appelés dès lors *clé de la relation*. Elle impose que la relation ne peut contenir deux tuples identiques pour ces attributs et que leurs valeurs ne peuvent jamais être à NULL. Il ne peut y avoir plus d'une contrainte de clé par relation. Sur le schéma **S** on a, par exemple, la contrainte de clé suivante:

CI_4 : n°employé est la clé de la relation EMPLOYE.

La convention habituelle consiste à souligner dans le schéma de la relation les attributs qui forment la clé. On peut aussi, bien sûr, exprimer une contrainte de clé par une formule logique. Par exemple pour **CI_4** :

$$\mathbf{CI\_4 : \forall\ x,y,z,t\ EMPLOYE(x,y,z,t) \Rightarrow ((\forall\ y',z',t'\ \ EMPLOYE(x,y',z',t') \Rightarrow (y=y' \wedge z=z' \wedge t=t')) \wedge (\neg NULL(x)))}$$

Certains modèles objets autorisent l'expression de contraintes de clé afin de ne pas perdre en fonctionnalité par rapport au modèle relationnel (Stein, Anderson, Maier, 1989). Ces contraintes portent alors sur un ou plusieurs attributs d'un type. Elles imposent que deux objets distincts de l'extension du type ne peuvent avoir les mêmes valeurs pour ces attributs. Si le modèle spécifie que l'extension d'un type contient les extensions de ses sous-types, alors la contrainte porte aussi sur ces objets.

2.4 Contraintes d'intégrité référentielles

Elles constituent une catégorie importante de contraintes d'intégrité, dont la définition est cependant entourée d'une certaine confusion, comme le montrent les modifications successives de la définition de la notion de référence de Codd (Markowitz, 1990) (Date, 1981). La définition la plus générale et la plus précise est celle de (Elmasri, Navathe, 1989, p 144).

Définition – Une relation R_1 *référence* une relation R_2 si les deux propriétés suivantes sont toujours vérifiées (voir figure 1 ci-dessous) :

(*i*) R_1 contient un sous-ensemble d'attributs noté Fk, dont les domaines sont ceux de la clé Pk de R_2.

(*ii*) Pour tout tuple t_1 de R_1, soit il existe un tuple t_2 de R_2 tel que $t_1[FK]=t_2[PK]$, soit $t_1[FK]=$ NULL.

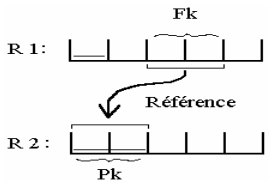

Figure 1 : Intégrité référentielle

Remarques:

- Fk est parfois appelée une clé étrangère. Elle peut avoir des attributs communs avec la clé de la relation R_1.
- R_1 est la relation référençante et R_2 la relation référencée. La notion de référence peut être découplée de celle de clé primaire (Date, 1981). La référence ne se fait alors pas forcément sur Pk, mais aussi sur d'autres attributs de R_2. On parle alors de dépendance d'inclusion. Les références sont donc un cas particulier des dépendances d'inclusion.
- les relations R_1 et R_2 peuvent être liées par plusieurs liens de référence différents.

Les références dans le modèle relationnel se font donc par valeur. Un tuple en référence un autre par une valeur qui identifie ce dernier de manière unique. Les contraintes d'intégrité référentielles permettent de garantir que la valeur en question désigne bien un tuple de la base. Ainsi, en reprenant les notations de la définition, l'insertion d'un tuple dans la relation R_1 sera refusée s'il n'y a pas de tuple de la relation R_2 dont la clé correspond à la valeur de la clé étrangère du tuple inséré.

Un exemple de contrainte d'intégrité référentielle sur le schéma **S** est :

CI_5 : pour tout tuple t de la relation EMPLOYE il existe un tuple de la relation DEPARTEMENT qui a pour valeur de clé celle de l'attribut n°département du tuple t.

CI_5 : $\forall$ x,a,b,c EMPLOYE(a,b,c,x) $\Rightarrow$ (($\exists$y DEPARTEMENT(x,y)) $\vee$ NULL(x))

Les contraintes d'intégrité référentielles font partie des normes SQL-1 (SQL, 1986) et SQL-2 (SQL, 1992). Dans SQL-2 il est en plus possible de spécifier des actions à entreprendre lorsqu'elles sont violées (on reviendra sur cette question au § 4).

(Date, 1981) enrichit la sémantique des contraintes d'intégrité référentielles en autorisant la référence à plusieurs relations. Pour cela, il introduit trois quantificateurs: *[Exactement-Un]*, *[Au-Moins-Un]*, *[Plusieurs]*. Supposons par exemple que la relation

DEPARTEMENT soit éclatée en plusieurs relations. La contrainte précédente (**CI_5**) peut alors par exemple devenir :

CI_5bis : pour tout tuple t de la relation EMPLOYE il existe dans *[Au-Moins-Un]* des relations DEPARTEMENT un tuple qui a pour valeur de clé celle de l'attribut n°département du tuple t.

Le modèle relationnel ne pouvant les exprimer directement, les contraintes d'intégrité référentielles sont une composante séparée, bien qu'essentielle, d'un schéma relationnel. Dans les modèles objets, au contraire, la notion de référence relève du modèle lui-même. Elle constitue une notion intrinsèque. Il faut qu'un objet B existe pour qu'un objet A puisse le référencer. Aussi n'est-il plus nécessaire de passer par « l'artifice » d'une valeur pour traduire le lien.

2.5 Dépendances

Un certain nombre de types de liens entre les données d'une base de données ont été formalisés. Ils sont regroupés sous le terme générique de dépendances. Puisqu'elles décrivent des invariants sur les données, les dépendances peuvent être considérées comme des contraintes d'intégrité, quoiqu'elles soient plutôt abordées dans la littérature en tant que composantes pour l'élaboration d'un "bon" schéma relationnel. On présente maintenant les principales dépendances. Les définitions sont issues de (Mannila, Räihä, 1992, pp 87-97) et (Delobel, Adiba, 1982, pp 345-384).

DEPENDANCES FONCTIONNELLES

Si R est une relation et X et Y deux sous-ensembles quelconques d'attributs de R, alors R: $X \rightarrow Y$ désigne une dépendance fonctionnelle sur R. Cette dépendance est vérifiée si et seulement si pour tout tuple t de R et tout tuple t' de R on a $t[X]=t'[X] \rightarrow t[Y]=t'[Y]$. Autrement dit, il existe une dépendance fonctionnelle entre X et Y si la connaissance d'une valeur de X détermine celle de Y de manière unique. (Mannila, Räihä, 1992, pp 120-121) étend la définition aux modèles objets.

Un exemple de dépendance fonctionnelle sur le schéma **S** est,

CI_6 : un employé ne travaille que dans un seul département.

CI_6 : EMPLOYE: n°employé → n°département

L'expression de cette dépendance en logique découle naturellement de la définition et s'écrit,

CI_6 : $\forall$ x,y,z,a,b EMPLOYE(x,a,b,y) $\wedge$ EMPLOYE(x,a,b,z) $\Rightarrow$ y=z

DEPENDANCES MULTIVALUEES

Les dépendances multivaluées ont été introduites par (Fagin, 1977). Si R est une relation et X, Y et Z des sous-ensembles d'attributs de R, alors R: $X \rightarrow\rightarrow Y$ désigne une dépendance multivaluée sur R. Cette dépendance est vérifiée si et seulement si pour chaque paire de tuples[5] (x,y,z) et (x,y',z') de la relation R, il existe deux autres tuples (x,y,z') et (x,y',z) dans R. Il découle de cette définition que $X \rightarrow\rightarrow Y$ implique $X \rightarrow\rightarrow Z$. Il en découle aussi que, dans une certaine mesure, les dépendances multivaluées sont une généralisation des dépendances fonctionnelles.

On illustre maintenant la définition sur un exemple. Considérons la relation **EMPLOYE (n°employé, prénom, n°responsable)**. Elle stocke, pour chaque employé, ses différents prénoms et les numéros de ses différents responsables. A priori, pour chaque employé, toute valeur de l'attribut **prénom** doit être associée à toutes les valeurs de l'attribut **n°responsable** auxquelles sont associés les autres noms de l'employé. Ainsi, si l'extension de la relation **EMPLOYE** contient les deux tuples (127, Jean, 128) et (127, Pierre, 129), alors elle doit aussi contenir les tuples (127, Jean, 129) et (127, Pierre, 128). Cette propriété s'exprime par la dépendance multivaluée,

CI_7 : EMPLOYE : n°employé $\rightarrow\rightarrow$ nom

L'expression de cette dépendance en logique découle naturellement de la définition et s'exprime par,

CI_7 : $\forall$ x,y,z,y',z' EMPLOYE(x,y,z) $\wedge$ EMPLOYE(x,y',z') $\Rightarrow$ EMPLOYE(x,y',z) $\wedge$ EMPLOYE(x,y,z')

DEPENDANCES DE JOINTURE (OU DE PRODUIT)

Soit R une relation de schéma U et $\{X_1, X_2, \ldots, X_k\}$ un ensemble de parties de U. La notation $\bowtie\{X_1, X_2, \ldots, X_k\}$ désigne une dépendance de jointure totale sur R. Elle est vérifiée si et seulement si $R = \Pi_{X1}(R) \bowtie \Pi_{X2}(R) \bowtie \ldots \bowtie \Pi_{Xk}(R)$.

Dans le cas où la dépendance est définie sur $U' \subset U$, on parle de dépendance de jointure partielle.

Les dépendances de jointure sont une généralisation des dépendances multivaluées ainsi que des dépendances hiérarchiques introduites par (Delobel, 1973). Ainsi, la dépendance multivaluée $X \rightarrow\rightarrow Y$ est équivalente à $\bowtie\{[X,Y],[X,Z]\}$ en notant Z le complément de X et Y par rapport à U.

---

[5] $x,x' \in X$; $y,y' \in Y$; $z,z' \in Z$.

2.6 Contraintes d'intégrité récursives

Il s'agit de contraintes d'intégrité nécessitant le calcul d'une fermeture transitive, i.e., de n jointures successives, où n est inconnu a priori. La contrainte suivante en est un exemple, dans le cas du schéma relationnel défini précédemment.

CI_8 : deux projets ne peuvent être mutuellement dépendants, directement ou indirectement.

Cette contrainte d'intégrité ne peut s'exprimer à l'aide d'une formule de la logique du premier ordre sans la définition préalable d'une relation **Succ(x,y)** égale à la fermeture transitive de la relation **DEPEND**. Sa définition en Datalog est la suivante :

```
Succ(x,y) :- DEPEND(x,y).
Succ(x,y) :- Succ(x,z),Succ(z,y).
```

La contrainte d'intégrité s'écrit alors:

CI_8 : $\forall$ x,y Succ(x,y) $\wedge$ $\neg$Succ(y,x)

2.7 Contraintes d'intégrité avec des agrégats

Les langages de requêtes des bases de données commerciales étendent souvent leur expressivité au-delà de la logique du premier ordre. Ainsi, ils permettent d'utiliser des agrégats dans les requêtes. Les agrégats usuels sont **SUM** (somme), **MIN** (minimum), **MAX** (maximum), **AVG** (moyenne), **COUNT** (comptage). Ils s'appliquent aux colonnes des relations (ou à des sous-ensembles) et fournissent une quantité unique (Ullman, 1989, p 175). Ces agrégats peuvent aussi être utilisés dans l'expression des contraintes d'intégrité (Ceri, Widom, 1990) (Jarke et *al.*, 1990) (Weber, Stucky, Karszt, 1983) pour en augmenter la puissance sémantique.

L'expression des contraintes d'intégrité peut, elle aussi, utiliser les agrégats. Par exemple, l'expression de la contrainte suivante requiert l'utilisation de l'agrégat **AVG** :

CI_9 : le salaire moyen des employés de plus de 60 ans doit être supérieur à 20000.

Cette contrainte s'exprime en SQL de la manière suivante:

```
CI_9 : CREATE ASSERTION CI_9
       CHECK
          ( ( SELECT AVG(salaire)
                FROM  EMPLOYE s
                  WHERE  s.âge > 60 ) > 20000 )
```

2.8 Contraintes sur les opérations élémentaires

Certains travaux tels que (Cremers, Domann, 1983) (Weber, Stucky, Karszt, 1983) et (Date, 1981) ont proposé que l'expression des contraintes d'intégrité soit directement liée aux opérations élémentaires des transactions (insertion, mise à jour, suppression). En pratique, ces travaux expriment des préconditions ou des postconditions à ces opérations.

Une grammaire simplifiée d'un langage élémentaire permettant de spécifier de telles contraintes est par exemple :

<contrainte> ::= BEFORE <opération> | AFTER <opération>

<opération>::= INSERTING <nom_tuple> FROM <relations> CHECK <condition>
| UPDATING <exp> FROM <relations> CHECK <condition>
| DELETING <nom_tuple> FROM <relations> CHECK <condition>

On donne maintenant trois exemples de contraintes exprimées au niveau des opérations élémentaires, toujours sur le schéma **S** du § 2.2 :

```
CI_10a : AFTER UPDATING  t.âge  FROM EMPLOYE t  CHECK t.âge<100

CI_10b : BEFORE INSERTING  t  FROM EMPLOYE CHECK  t.salaire>0

CI_10c : AFTER UPDATING  t.compétence  FROM  REQUIERT t CHECK
  ( EXIST   ( SELECT *
              FROM POSSEDE s
              WHERE s.compétence=t.compétence )  )
```

Ce type de contraintes trouve un prolongement naturel dans les modèles objets par les contraintes sur le contexte d'activation des méthodes (précondition) et sur leur contexte de validation (postcondition). (Martin, Adiba, Defude, 1992) étudie en détail ce type de contraintes. Un autre prolongement est celui évoqué, mais non étudié, par (Medeiros, Jomier, Cellary, 1992). Il suggère l'expression de contraintes sur l'ordre d'exécution des opérations, en particulier des méthodes dans le cas des bases de données objets.

Exprimer des contraintes au niveau des opérations élémentaires des transactions conduit implicitement à considérer que ces opérations ont un devoir de cohérence[6]. Or,

[6] On rappelle que les contraintes d'intégrité expriment des critères de cohérence.

la cohérence se définit au niveau des états de la base de données et à ce titre n'est concernée que par le résultat final des transactions. Il est donc illégitime d'appeler contraintes d'intégrité des contraintes exprimées au niveau des opérations élémentaires, sauf bien sûr, dans le cas de transactions mono-opération. Dans ce cas, en effet, toute opération élémentaire constitue une transaction.

Les contraintes exprimées au niveau des opérations ne pouvant être considérées, dans le cas général, comme des contraintes d'intégrité, on ne les aborde pas dans la suite.

3. Contraintes d'intégrité dynamiques

3.1 Définition

Les contraintes d'intégrité dynamiques, parfois aussi appelées contraintes d'intégrité temporelles, permettent d'exprimer des critères de cohérence d'un état E de la base de données par des propriétés sur cet état et ceux qui le précèdent[7].

Il en découle que, en présence de contraintes dynamiques, les états passés de la base contraignent partiellement les états suivants. Autrement dit, les contraintes dynamiques contraignent l'évolution des données de la base.

Les contraintes d'intégrité dynamiques trouvent leur utilité dans toutes les situations où des propriétés liées au temps doivent être prises en compte : suivi de projet, finance, attribution de crédit, …

Contrairement aux contraintes statiques, les contraintes dynamiques ont été assez peu étudiées. Les principaux travaux les concernant sont ceux de Jan Chomicki (Chomicki, Niwinski, 1995) (Chomicki, 1994) (Chomicki, 1992) et ceux de l'équipe de Udo Lipeck, dont les principales publications sont (Gertz, Lipeck, 1995), (Gertz, Lipeck, 1993), (Lipeck, Feng, 1988) et (Ehrich, Lipeck, Gogolla, 1984). Il existe quelques autres travaux comme (Dumas, 1996) et (Schwiderski, Hartmann, Saake, 1994).

Par ailleurs, le support déclaratif des contraintes dynamiques ne figure pas dans la norme SQL-2 (SQL, 1992) ni dans la future norme SQL-3 (SQL, 1994). Sabrina (Gardarin, Valduriez, 1989, p 189) est le seul système commercial à en proposer un support, certes limité aux contraintes de transition.

[7] Si la propriété ne porte que sur l'état E et celui qui le précède, alors la contrainte est dite de transition. Si la propriété ne porte que sur E, alors la contrainte est statique. Les contraintes statiques constituent donc un cas particulier de contraintes dynamiques.

3.2 Expression des contraintes d'intégrité dynamiques

Les contraintes d'intégrité dynamiques sont généralement exprimées par des formules de logique temporelle *interprétées sur la séquence des états de la base* $\sigma=(E_1, E_2, \ldots, E_i, \ldots)$. On note par $(\sigma,i) \models F$ le fait que la formule F est vérifiée dans l'état i de σ.

Le formalisme de la logique temporelle est présenté dans la troisième partie de la thèse. On se limite ici à une présentation rapide et intuitive. De manière informelle, une formule temporelle est composée de prédicats élémentaires auxquels on applique des opérateurs temporels, des connecteurs logiques traditionnels ($\vee, \neg, \ldots$) et des quantificateurs ($\forall, \exists$) (Manna, Pnueli, 1991, p 186). Les deux principaux opérateurs temporels sont l'opérateur modal universel, noté $\Box$, et l'opérateur modal existentiel, noté $\Diamond$. La logique temporelle existe selon deux versions symétriques : passée et future. Dans la version passée (resp. future) la sémantique associée à $\Box$ est « toujours dans le passé » (resp. « toujours dans le futur ») et celle associée à $\Diamond$ est « au moins une fois dans le passé » (resp. « au moins une fois dans le futur »). Autrement dit, pour la logique temporelle du passé on a :

$(\sigma,i) \models \Box F$ ssi $(\sigma,i) \models F$ pour tous les états k de σ tels que $0 \leq k \leq i$.

$(\sigma,i) \models \Diamond F$ ssi $(\sigma,i) \models F$ pour au moins un état k de σ tels que $0 \leq k \leq i$.

Les travaux sur les contraintes dynamiques introduisent souvent d'autres opérateurs temporels comme l'opérateur SINCE (« depuis »), l'opérateur UNTIL (« jusqu'à ») ou encore l'opérateur $\circ$ (« dans l'état d'avant »). La liste n'est bien sûr pas exhaustive et tout opérateur est utilisable pourvu que l'on puisse en donner une interprétation.

Les contraintes dynamiques portant sur les états passés de la base, il est naturel de les exprimer en logique temporelle du passé. C'est l'option retenue par Chomicki et c'est aussi celle que l'on retient dans la suite. Lipeck en revanche les exprime en logique temporelle du futur. (Chomicki, 1992) conjecture néanmoins qu'à toute contrainte dynamique exprimée en logique temporelle du futur on peut faire correspondre une contrainte dynamique équivalente exprimée en logique temporelle du passé. Notons que cette conjecture est établie dans le cas restreint de la logique propositionnelle.

On présente maintenant quelques exemples de contraintes d'intégrité dynamiques sur le schéma $\mathbf{S_d}$ suivant :

EMPLOYE(n°employé, statut, salaire)

DETACHEMENT(n°employé)

CI_11 : le salaire d'un employé ne peut pas diminuer.

CI_11 : ∀ x,y,sal1,sal2 EMPLOYE(x,y,sal1) ∧ ◊EMPLOYE(x,y,sal2) ⇒ sal1 ≥ sal2

CI_12 : avant de devenir cadre, un employé doit d'abord avoir été ouvrier et une fois qu'il est cadre il ne peut plus changer de statut.

CI_12 : ∀ x,y,z □(EMPLOYE(x,"cadre",y) ⇒ ◊EMPLOYE(x,"ouvrier",z)) ∧ □(◊EMPLOYE(x,"cadre",y) ⇒ EMPLOYE(x,"cadre",y))

CI_13 : un employé ne peut pas être détaché plus de trois fois.

CI_13 : ¬(∃ x) DETACHEMENT(x) ∧ ◊(DETACHEMENT(x) ∧ ◊ DETACHEMENT(x))

Quoique assez puissant, le pouvoir expressif de la logique temporelle est néanmoins limité. Par exemple, comme le fait remarquer (Wolper, 1981), il n'est pas possible d'exprimer que certaines séquences de valeurs doivent se répéter tous les n états. Cela conduit à envisager l'expression des contraintes dynamiques dans d'autres formalismes que celui de la logique temporelle « classique ». (Plexousakis, 1995), qui utilise une logique temporelle basée sur les intervalles, est un premier exemple de travaux en ce sens.

3.3 Evolution d'un ensemble de contraintes dynamiques

L'ajout et la modification de contraintes statiques d'une base de données ne posent pas de problèmes particuliers. Il suffit simplement que la nouvelle contrainte soit vérifiée par l'état courant de la base ; il s'agit d'ailleurs là d'une condition nécessaire, mais non suffisante, pour qu'elle ne soit pas incohérente par rapport aux autres contraintes.

Dans le cas des contraintes dynamiques le problème est plus complexe, car celles-ci portent sur plusieurs états de la base. (Chomicki, 1992) entrevoit deux sémantiques possibles. Soit la contrainte mise à jour ou insérée doit être vérifiée dans l'état courant, soit l'état courant est pris comme état origine de σ pour son interprétation. La deuxième sémantique est plus réaliste, car elle ne conditionne pas l'ajout ou la modification de contraintes au passé de la base de données ; hypothèse qui rend tout changement d'autant plus difficile que la base est ancienne.

4. Spécification d'opérations de réparation

La cohérence étant définie au niveau des états de la base, la violation d'une contrainte signifie que le nouvel état n'est pas cohérent et qu'il ne doit donc pas être accepté. Par conséquent, il faut annuler la transaction qui a conduit à cet état. La transaction est donc l'unité de cohérence des systèmes de gestion de bases de données (Nicolas, 1982).

Il est néanmoins possible d'envisager de spécifier dans les contraintes des opérations dites de réparation, permettant de rétablir la cohérence en cas de violation. On parle alors d'une sémantique active pour la violation, par opposition à une sémantique passive (Urban, Karadimce, Nannapaneni, 1992). Ces opérations doivent nécessairement être exécutées dans la transaction qui a violé la contrainte.

La possibilité de définir des opérations de réparation doit être utilisée avec précaution car elles peuvent avoir des effets de bord. Il y a deux raisons principales à cela : (*i*) le risque de violation d'autres contraintes lors de l'exécution des opérations de réparation et donc l'exécution en cascade et non maîtrisée d'opérations de réparation, (*ii*) l'ordre d'exécution non déterminable des opérations de réparation dans le cas de la violation de plusieurs contraintes. Nous reviendrons sur ces deux problèmes dans le cadre de l'étude de la vérification des contraintes d'intégrité par les triggers, voir chapitre 2, § 1.2. Les triggers permettent, en effet, une mise en œuvre simple des opérations de compensation.

La question de la génération automatique des opérations de réparation a été abordée par (Markowitz, 1991) et (Casanova, Tucherman, Furtado, 1988) pour les contraintes d'intégrité référentielles et les dépendances d'inclusion, par (Morgenstern, 1984) pour une classe particulière de contraintes inter-relations et par (Urban, Karadimce, Nannapaneni, 1992) pour certaines contraintes dans les bases de données objets.

Il est possible de spécifier des opérations de réparation en SQL dans le cas des contraintes référentielles. Par exemple, pour la contrainte **CI_5** on peut spécifier qu'en cas de violation, l'employé dont le département n'existe pas dans la relation **DEPARTEMENT** soit supprimé, ce qui s'exprime, en SQL, au niveau de l'attribut **n°département** de la relation **EMPLOYE** par :

```
CONSTRAINT ref_dep
REFERENCE DEPARTEMENT(n°département)
ON DELETE CASCADE
```

Chapitre 2

Vérification des Contraintes d'Intégrité

On aborde maintenant la vérification des contraintes d'intégrité. Elle se pose toujours en termes de performances. Comme dans le premier chapitre, on distingue successivement le cas des contraintes statiques, § 1, *puis celui des contraintes dynamiques,* § 2. *La vérification des contraintes statiques a été largement étudiée et on ne présente que les méthodes les plus significatives. En revanche, il existe comparativement peu de travaux sur la vérification des contraintes dynamiques.*

1. Vérification des contraintes d'intégrité statiques

La vérification des contraintes d'intégrité statiques constitue un domaine de recherche ancien, qui a donné lieu à de nombreux travaux. Leur nombre rend difficile toute présentation exhaustive. Ce paragraphe expose les méthodes qui nous ont semblé les plus significatives et les plus générales. On présente tout d'abord les méthodes dites de simplification. Puis, on présente comment les triggers peuvent être utilisés pour vérifier des contraintes d'intégrité. Enfin, on présente une méthode basée sur l'axiomatique de Hoare ainsi qu'une autre utilisant le processeur de requêtes du système.

1.1 Méthodes de simplification de contraintes

1.1.1 Principe

Les méthodes de simplification permettent de déterminer, à l'issue de chaque transaction, une simplification des contraintes à vérifier. Pour une contrainte donnée, sa simplification peut consister, soit en une contrainte dérivée plus rapide à vérifier, soit en une réduction de son espace de vérification. La détermination de la simplification doit être rapide, de sorte que le gain espéré pour la vérification ne soit pas perdu par le calcul de la forme simplifiée.

1.1.2 Simplification dans le cas de transactions mono-opération

On présente la méthode de simplification de (Nicolas, 1982). Elle a donné lieu à de nombreux travaux ultérieurs. Les contraintes d'intégrité considérées doivent satisfaire la propriété de *portée limitée* (range restricted) et doivent être réécrites sous forme prénexe conjonctive[8]. Une formule de logique du premier ordre satisfait la propriété de portée limitée si :

- chaque variable quantifiée universellement apparaît dans au moins une formule atomique négative parmi toutes les disjonctions où la variable apparaît.
- pour chaque variable quantifiée existentiellement qui apparaît dans au moins une formule atomique, il y a une disjonction qui contient seulement des formules atomiques non négatives qui ont toutes la variable en argument.

En pratique, un grand nombre de contraintes d'intégrité vérifient la propriété de portée limitée. C'est le cas de **CI_1** qui se réécrit sous forme prénexe conjonctive,

CI_1 : $\forall$ x,y,a,b $\neg$EMPLOYE(a,x,y,b) $\vee$ x$\leq$40 $\vee$ y$\geq$20000.

La méthode de (Nicolas, 1982) permet de déterminer une contrainte simplifiée CI' pour toute contrainte CI vérifiant la propriété de portée limitée, dans le cas de transactions mono-opérations. Une transaction mono-opération est composée alternativement d'une des trois opérations : insertion, suppression ou modification d'un tuple d'une relation. On donne maintenant les différentes étapes pour obtenir CI' à partir de CI dans le cas d'une transaction effectuant une insertion sur la relation R.

(*i*) pour chaque occurrence négative de R dans CI, donc de la forme $\neg$R, déterminer la substitution σ_i du tuple inséré avec cette occurrence.

(*ii*) supprimer de l'ensemble des substitutions σ_i obtenues en *(i)*, celles qui sont subsumées par une autre. Une substitution σ en subsume une autre σ' ssi $\sigma \subset \sigma$'.

(*iii*) appliquer les substitutions de *(ii)* sur CI, et remplacer les littéraux instanciés par

[8] Dans cette forme on n'autorise plus les opérateurs logiques $\Rightarrow$ et $\Leftrightarrow$; les négations sont "distribuées" avec les lois de Morgan ; une variable n'est quantifiée qu'une fois ; les quantificateurs sont en tête de formule.

la valeur de vérité **faux**.

(*iv*) appliquer les règles d'absorption de la figure 2 ci-dessous le plus possible.

$\neg$ **vrai** $\rightarrow$ **faux**	$\neg$ **faux** $\rightarrow$ **vrai**
vrai $\vee$ L $\rightarrow$ **vrai**	**faux** $\vee$ L $\rightarrow$ L
vrai $\wedge$ L $\rightarrow$ L	**faux** $\wedge$ L $\rightarrow$ **faux**

Figure 2 : Règles d'absorption (L est un littéral quelconque)

A l'issue de l'étape (*iv*) on obtient la contrainte CI'.

Soit, par exemple, l'insertion $(118, 42, 250)$ dans la relation **EMPLOYE**. Pour **CI_1** on obtient l'unique substitution $\sigma = \{x/42, y/250\}$. Par application de (*iii*), **CI_1** se réécrit,

faux $\vee$ $(42 \leq 40)$ $\vee$ $(250 \geq 2000)$

et finalement, par application de (*iv*), on a,

CI_1' : faux.

L'insertion proposée viole donc la contrainte d'intégrité. On souligne que dans ce cas particulier la simplification conduit à une simple valeur de vérité. Par conséquent, la vérification de la contrainte **CI_1** s'effectue sans accès à la base de données. Ce n'est pas toujours le cas, certains prédicats pouvant ne demeurer que partiellement instanciés à l'issue de (*iv*).

1.1.3 Simplification dans le cas de transactions multi-opérations

La méthode de simplification parmi les plus référencées dans le cas de transactions multi-opérations est celle de (Hsu, Imielinski, 1985). Dans cette méthode, la forme simplifiée CI' d'une contrainte CI consiste en un arbre ET-OU qui donne un espace de vérification (voir définition ci-après) restreint pour CI.

L'algorithme de simplification n'est pas repris ici en détail, car il distingue un grand nombre de cas. On présente, en revanche, un exemple de simplification pour la contrainte **CI_14** ci-dessous,

CI_14 : il existe au moins un employé dont toutes les compétences sont de code inférieur à 10 (on suppose que les compétences de la relation POSSEDE sont codées par des nombres)

CI_14 : $\exists$x,y ($\exists$a,b,c EMPLOYE(x,a,b,c)) $\wedge$ POSSEDE(x,y) $\wedge$ y < 10

et une transaction T qui effectue des suppressions de tuples dans la relation **EMPLOYE** et des insertions dans la relation **POSSEDE**. On note par commodité r_1 l'ensemble des tuples de la relation **EMPLOYE**, ou portée de la relation **EMPLOYE**, Δr_1 l'ensemble des tuples supprimés par la transaction T dans r_1, r_2 l'ensemble des tuples de la relation **POSSEDE**, ou portée de la relation **POSSEDE**, et Δr_2 l'ensemble des tuples ajoutés à r_2 par la transaction T.

(Hsu, Imielinski, 1985) introduit la notion d'espace de vérification pour mesurer approximativement la complexité de la vérification d'une contrainte. Il est défini par le produit cartésien des portées des relations qui interviennent dans l'expression de la contrainte. Avec les notations précédentes, il vaut ici pour la contrainte **CI_14** et à l'issue de l'exécution de la transaction T, $(r_1 - \Delta r_1) \times (r_2 + \Delta r_2)$.

La méthode de simplification proposée permet de réduire l'espace de vérification. On note Vr_i la portée effective de la relation R_i de portée r_i. Elle est définie, pour une contrainte donnée, par l'union des sous-ensembles minimaux de r_i satisfaisant la contrainte. Ainsi, pour la relation mono-attribut $R(A) = ((1),(3),(5),(6))$ et la contrainte $\exists t \in r\ t.A > 4$, on a $Vr_1 = \{5\} \cup \{6\} = \{5,6\}$.

Dans le cas de la contrainte **CI_14** et de la transaction T, la simplification de l'espace de vérification conduit à l'arbre suivant :

ET ——— $(Vr_1 - \Delta r_1, r_2)$
|
└——— $(Vr_1 - \Delta r_1, \Delta r_2)$

Cet arbre signifie que, au lieu de vérifier **CI_14** sur l'espace de vérification $(r_1 - \Delta r_1) \times (r_2 + \Delta r_2)$ il suffit de la vérifier successivement sur les espaces de vérification $(Vr_1 - \Delta r_1) \times r_2$ et $(Vr_1 - \Delta r_1) \times \Delta r_2$.

Supposons que $|r_1| = 30$ tuples, $|r_2| = 10$ tuples, $|\Delta r_1| = 4$ tuples, $|\Delta r_2| = 3$ tuples et que $|Vr_1| = 15$ tuples. La taille de l'espace de vérification de **CI_14** est de

$$(|r_1| - |\Delta r_1|) * (|r_2| + |\Delta r_2|) = (30-4) * (10+3) = 338 \text{ tuple} \times \text{tuple},$$

mais devient après simplification

$$(|Vr_1| - |\Delta r_1|) * |\Delta r_2| + (|Vr_1| - |\Delta r_1|) * |r_2|$$
$$= (|Vr_1| - |\Delta r_1|) * (|r_2| + |\Delta r_2|) = (15-4) * 3 + (15-4) * 10 = 143 \text{ tuple} \times \text{tuple}.$$

La simplification réduit donc l'espace de vérification d'environ 58%.

Cette méthode de simplification présente un certain nombre de limitations :

- dans certains cas, aucune simplification n'est possible.
- le calcul de Vr_1, qui ne peut se faire qu'à l'exécution, doit être performant, afin de ne pas trop empiéter sur le bénéfice de la simplification.
- selon la « configuration », l'effet de la simplification peut être plus ou moins sensible. C'est le cas dans l'exemple précédent si $|r_1| = 20$ tuples au lieu de 30 et que $|Vr_1|$ demeure égal à 15. En effet, on a alors

$$(|r_1| - |\Delta r_1|) * (|r_2| + |\Delta r_2|) = (20-4) * (10+3) = 208 \text{ tuple} \times \text{tuple},$$

La méthode de simplification conduisant toujours à un espace de vérification de 143 tuple×tuple, la réduction de cet espace n'est plus que de 31%.

- les informations prises en compte sur les transactions sont sommaires et limitent donc l'étendue de la simplification. En particulier, contrairement à (Nicolas, 1982), la méthode n'exploite pas les paramètres des transactions.

1.1.4 Autres méthodes de simplification

(Olivé, 1991) (Bry, Decker, Manthey, 1988) et (Kowalski, Sadri, Soper, 1987) ont étendu les méthodes de simplification au cas des bases de données déductives relationnelles. (Jeusfeld, Jarke, 1991) est le seul travail, qui, à notre connaissance, s'intéresse au cas des bases de données déductives à objets complexes. Il utilise une description logique de ces modèles basée sur leur traduction dans un modèle sémantique.
Tous ces travaux sont inspirés de (Nicolas, 1982) et ne considèrent que des transactions mono-opération.

1.2 Utilisation des triggers

1.2.1 Principe des triggers

La notion de trigger, ou déclencheur, a été introduite dans les années soixante dix pour les bases de données relationnelles (Astrahan, 1976). Un trigger permet d'exécuter automatiquement des actions en réponse à des événements et à des conditions. Il a trois composantes :

- *l'Evénement* (**E**). Il constitue le critère de déclenchement du trigger. On distingue trois catégories d'Evénements :
  - ceux basés sur le changement du contenu de la base de données. Ils sont généralement liés aux opérations élémentaires sur les données (création,

destruction, mise à jour, lecture) ou aux transactions (début, validation, annulation).

○ ceux liés au temps. Par exemple : tous les jours à cinq heures, deux jours après un événement donné, …

○ ceux qui proviennent de l'environnement extérieur de la base de données. Par exemple, l'arrivée d'un courrier électronique.

La liste des Evénements envisageables est très vaste. Chaque système n'en implante qu'un sous-ensemble. Il est par ailleurs possible de définir des événements composites, construits récursivement à partir des trois types précédents (conjonction d'Evénements, disjonction d'Evénements, …). L'Evénement n'est pas une composante facultative du trigger.

- *la Condition* (**C**). Elle est composée d'une formule, le plus souvent exprimée en logique du premier ordre. Par conséquent, la logique est souvent utilisée comme fondement des langages d'expression des Conditions. Lorsqu'un Evénement déclenche le trigger, la Condition est évaluée. Si elle est vérifiée, la partie Action du trigger (voir ci-après) est alors exécutée. La Condition est une composante facultative du trigger.

- *l'Action* (**A**). Les langages d'expression des Actions peuvent être déclaratifs ou procéduraux. La plupart des bases de données commerciales utilisent le langage SQL, étendu avec des opérateurs qui permettent d'accéder à l'ancienne et à la nouvelle valeur d'un tuple modifié. Dans le système Oracle, les Actions peuvent aussi être spécifiées en PL/SQL qui est un sur-ensemble procédural de SQL, voir partie 2, chapitre 6, § 3. L'Action est une composante facultative du trigger.

Les trois composantes d'un trigger s'exécutent toujours dans l'ordre Evénement, Condition et Action. On les désigne parfois sous le terme de règles (**E**,**C**,**A**). Le couple (**C**,**A**) constitue une règle de production traditionnelle. La vérification de la Condition et l'exécution de l'Action constituent *l'exécution du trigger*.

Hors de cette définition générale, les mises en œuvre des triggers diffèrent par le modèle d'exécution retenu. Un même système peut en proposer plusieurs. Un modèle d'exécution est un triplet de la forme : (*mode de transaction, mode de synchronisation, mode d'exécution)*.

Le mode de transaction spécifie que le trigger est exécuté dans la même transaction que celle qui l'a déclenché (modalité *Imbriqué*) ou dans une autre transaction (modalité *Séparé*). Le mode de synchronisation spécifie si l'exécution du trigger interrompt celle du programme qui l'a déclenché (modalité *Synchrone*) ou si les deux s'exécutent en parallèle (modalité *Asynchrone*). Enfin, le mode d'exécution spécifie que l'exécution

du trigger est déclenchée dès que survient son événement déclencheur (modalité *Immédiat*) où à la fin de la transaction qui a déclenché le trigger (modalité *Différé*).

Le modèle d'exécution généralement retenu dans les systèmes de gestion de bases de données commerciaux est : *(Imbriqué, Synchrone, Immédiat)*.

Certains prototypes de recherche proposent des modèles plus complexes. C'est le cas de NAOS (Collet, 1996) et aussi de Starburst (Widom, Cochran, Lindsay, 1991) (Widom, 1996) qui propose les trois modèles : *(Imbriqué, Synchrone, Immédiat)*, *(Imbriqué, Synchrone, Différé)* et *(Imbriqué, Asynchrone, Immédiat)*.

L'exécution de triggers pose deux problèmes inhérents à leur nature :

- *conflit*. Il y a conflit entre deux triggers lorsque leurs Evénements et leurs Conditions se recouvrent respectivement, au moins partiellement. La question est alors de déterminer un ordre pour leur exécution et sur quel état de la base le deuxième trigger doit être exécuté. Les solutions généralement proposées sont basées sur des mécanismes complexes de priorité (Picouet, Cheiney, 1992). Si pour tout couple de triggers d'un ensemble de triggers leur ordre d'exécution fournit le même résultat, l'ensemble vérifie la propriété de *confluence*. Il s'agit bien sûr d'une propriété souhaitable, mais elle ne peut être démontrée dans le cas général.
- *problème de l'exécution en cascade*. L'exécution d'un trigger peut elle-même déclencher un autre trigger et ainsi de suite. Outre le fait que cela rend difficile la compréhension globale du système, le risque est celui de bouclage infini. Un ensemble de triggers pour lequel il ne peut y avoir de tels bouclages vérifie la propriété de *terminaison*. Tout comme pour la propriété de confluence, il s'agit d'une propriété souhaitable, mais qui ne peut être démontrée dans le cas général.

(Aiken, Widom, Hellerstein, 1992) présente une méthode de preuve de la confluence et de la terminaison dans le cadre des triggers du système Starburst. Elle est basée sur la construction de graphes d'appel des triggers entre eux.

Il est communément admis que les triggers seront une des caractéristiques essentielles pour les systèmes de gestion de bases de données du futur (Urban, Karadimce, Nannapaneni, 1992). Ils sont la composante essentielle des bases de données actives (Dayal, Hanson, Widom, 1994) (Dittrich, Gatziu, Geppert, 1995).

La plupart des systèmes commerciaux offrent un support limité des triggers. C'est le cas d'Oracle et Ingres pour les bases de données relationnelles et de O_2 pour les bases de données objets. Les triggers ne font pas partie de la norme SQL-2 (SQL, 1992), mais feront partie de la norme SQL-3 (SQL, 1994) prévue pour 1998.

Nous avons exposé les caractéristiques essentielles des triggers. On présente dans le paragraphe suivant comment les triggers peuvent être utilisés pour la vérification des contraintes d'intégrité. Les triggers ont bien sûr de nombreuses autres applications telles que la gestion des vues, l'implantation de L4G (Aristote et *al.,* 1993), le contrôle des droits d'accès, le maintien des versions des données, …

1.2.2 Utilisation des triggers pour la vérification des contraintes d'intégrité

GENERATION DES TRIGGERS

La première étape est la détermination des Evénements des triggers. L'objectif est de déterminer, pour chaque contrainte d'intégrité, toutes les insertions, toutes les mises à jour et toutes les suppressions de tuples qui pourraient la violer et qui constitueront à ce titre les Evénements des triggers.

(Ceri, Widom, 1990) propose une méthode dans le cadre du modèle relationnel lorsque les contraintes d'intégrité sont des clauses de Horn. Les ensembles de triggers obtenus vérifient les propriétés de confluence et de terminaison. Les algorithmes proposés ne sont pas détaillés ici en raison des longs développements que leur complexité rendrait nécessaire. Les résultats obtenus, dans les cas simples, sont cependant relativement naturels. Pour la contrainte **`CI_1`**, ils donnent les trois Evénements suivants :

- l'insertion d'un tuple dans la relation **`EMPLOYE`**.
- la mise à jour de l'attribut **`âge`** d'un tuple de la relation **`EMPLOYE`**.
- la mise à jour de l'attribut **`salaire`** d'un tuple de la relation **`EMPLOYE`**.

Notons que la méthode proposée peut générer du bruit, i.e., des Evénements supplémentaires qui ne risquent pas de violer la contrainte.

(Medeiros, Pfeffer, 1991) étend cette méthode aux modèles objets. Dans ces modèles, la modification des objets est effectuée par les méthodes. Dès lors, les événements sont de la forme *(Type_Objet_Receveur, Méthode)* au lieu de *(Relation, {insertion, mise_à_jour, suppression})* dans le cadre du modèle relationnel.

Après avoir déterminé les Evénements des triggers, il reste à déterminer leurs Conditions et leurs Actions.

La Condition est donnée par l'expression logique de la contrainte à vérifier, exprimée négativement. La Condition peut donc être générée automatiquement à partir de la spécification déclarative de la contrainte.

Dans la composante Action du trigger, on spécifie la sémantique retenue pour la violation. Dans la cas d'une sémantique passive, l'Action correspond simplement à l'ordre d'annulation de la transaction. Dans le cas d'une sémantique active, l'Action contient les opérations de réparation, voir chapitre 1, § 4.

On donne pour conclure l'expression des trois triggers nécessaires à la vérification de la contrainte **CI_1** dans la syntaxe, pour l'instant provisoire, de la future norme SQL-3 (SQL, 1994) et pour une sémantique de violation passive:

```
CREATE TRIGGER TR_1
AFTER INSERT
ON TABLE EMPLOYE
WHEN(âge > 40 AND salaire ≤ 20000)
   Abort Transaction
FOR EACH ROW
```

```
CREATE TRIGGER TR_2
AFTER UPDATE OF salaire
ON TABLE EMPLOYE
WHEN(âge > 40 AND salaire ≤ 20000)
   Abort Transaction
FOR EACH ROW
```

```
CREATE TRIGGER TR_3
AFTER UPDATE OF âge
ON TABLE EMPLOYE
WHEN (âge > 40 AND salaire ≤ 20000)
   Abort Transaction
FOR EACH ROW
```

L'instruction **Abort Transaction** ne figure pas dans la version provisoire de la norme SQL-3, dans son édition d'août 1994, mais selon (Melton, Simon, 1993) une instruction de cette nature devrait être introduite.

DECLENCHEMENT DES TRIGGERS

Les contraintes d'intégrité sont vérifiées à la fin des transactions. L'exécution des triggers est donc reportée à la fin de celles-ci. Par conséquent, le modèle d'évaluation doit retenir la modalité *différé* pour le mode d'exécution, voir § 1.2.1.

Soit, par exemple, la transaction **T = { UPDATE EMPLOYE SET âge = 45 WHERE n°employé = 127; UPDATE EMPLOYE SET âge = 35 WHERE n°employé = 127;}** qui modifie l'âge de l'employé 127 à 45 puis 35. On suppose que le salaire initial de cet employé est de 15000.

Chacune des deux instructions déclenche une occurrence du trigger **TR_3**. A la fin de la transaction ces deux occurrences sont exécutées. La dernière mise à jour de la transaction rétablissant une valeur de l'âge en accord avec le salaire, les Conditions des deux occurrences ne sont pas vérifiées. Leurs Actions ne sont donc pas exécutées. Par conséquent, la transaction est acceptée.

Dans cet exemple, la même Condition est évaluée deux fois. On peut éviter cette duplication en ne considérant, pour le déclenchement des triggers, que l'effet « net » des transactions. Il est obtenu par l'application des quatre règles suivantes (Ceri, Widom, 1990) : (*i*) si un tuple est mis à jour plusieurs fois, seul le résultat composé est considéré, (*ii*) si un tuple est mis à jour puis effacé, seule la suppression est considérée, (*iii*) si un tuple est inséré puis mis à jour, on considère l'insertion directement avec les valeurs de la mise à jour, et enfin (*iv*) si un tuple est inséré puis supprimé, il n'est pas pris en compte.

Cette présentation montre que les triggers permettent de vérifier simplement les contraintes d'intégrité. La simplicité apparente de cette méthode de vérification est cependant obtenue en reportant le problème de l'optimisation de la vérification des contraintes d'intégrité sur celui de l'optimisation de l'exécution de triggers, qui est un problème tout aussi ardu.

1.3 Utilisation de l'axiomatique de Hoare

(Gardarin, Melkanoff, 1979) propose une méthode basée sur l'axiomatique de Hoare (Hoare, 1969) pour s'assurer de l'invariance des assertions dans un programme. Une contrainte est considérée comme un invariant et une transaction comme un programme. Il en résulte que si une transaction admet une contrainte donnée comme invariant, elle ne peut la violer. On est alors ramené au problème, bien connu en programmation, d'établir qu'une assertion est un invariant pour un programme.

Cette méthode n'est pas, à proprement parler, une méthode de vérification de contraintes d'intégrité. En effet, elle laisse le soin au programmeur d'écrire, dans les transactions, les tests pour que les contraintes ne soient pas violées. En revanche, elle vérifie la correction de ces tests, i.e. qu'ils évitent la violation des contraintes.

Soit, par exemple, la base de données composée de la relation **EMPLOYE** du schéma **S** défini au chapitre 1, § 2.2, et sur laquelle seule la contrainte **CI_1** est définie. Pour chaque transaction qui insère ou met à jour un tuple t, la méthode permet de vérifier que la transaction inclut le code suivant :

if (t.âge > 40 and t.salaire ≤ 20000) then Annuler_Transaction endif.

La mise en oeuvre de cette méthode est complexe et présente certaines limitations, en particulier,

- l'ajout ou la modification d'une contrainte réclame que les preuves de toutes les transactions soient recalculées ;
- la complexité des transactions est limitée. Elles ne peuvent contenir qu'une seule structure conditionnelle **if ... then ... else endif** , qu'une seule structure itérative **pour tout ... faire ... fin pour** et, de surcroît, ces structures ne peuvent pas être imbriquées.

1.4 Utilisation du processeur de requêtes

Ce paragraphe présente la méthode de (Stonebraker, 1975). Il s'agit d'une des plus anciennes méthodes de vérification de contraintes d'intégrité. Quoiqu'elle ait été implantée sur le système Ingres, elle n'a pas été intégrée au produit commercial, probablement pour des raisons de performances.

Le principe général est d'ajouter des tests aux opérations élémentaires (insertion, mise à jour, suppression) qui modifient la base de données. Ils permettent de n'autoriser l'exécution de ces opérations que si elles ne violent pas les contraintes d'intégrité.

Considérons la mise à jour suivante[9] qui divise par deux le salaire de tous les employés du département 15 :

```
UPDATE EMPLOYE
   SET salaire = salaire / 2
   WHERE n°département = 15
```

Supposons que la seule contrainte portant sur la relation **EMPLOYE**, et donc susceptible d'être violée par cette mise à jour, soit **CI_1** (voir page 17). Les algorithmes de la méthode conduisent alors à modifier l'opération de mise à jour précédente sous la forme :

```
UPDATE EMPLOYE
   SET salaire = salaire / 2
      WHERE n°département = 15
         AND ( âge ≤ 40 OR salaire > 2 * 20000 )   // test issu de CI_1
```

[9] (Stonebraker, 1975) exprime les opérations sur la base dans le langage QUEL. On utilise ici le langage SQL plus connu.

Sous cette forme, la mise à jour n'est réalisée que pour les tuples sur lesquels elle ne conduit pas à violer la contrainte **CI_1**. Les même algorithmes génèrent aussi la requête suivante qui délivre l'ensemble des tuples qui n'ont pu être mis à jour, car cela aurait conduit à violer la contrainte.

```
SELECT *
   FROM EMPLOYE
      WHERE n°département=15
         AND âge>40 AND salaire≤2*20000
```

La vérification des contraintes d'intégrité par cette méthode se ramène finalement à l'évaluation de clauses **WHERE** complexes. Les questions d'évaluation et d'optimisation des contraintes sont donc déléguées à l'optimiseur de requêtes dont la performance conditionne alors celle de la méthode.

1.5 Quelques autres méthodes de vérification

• (Bernstein, Blaustein, Clarke, 1980) génère et stocke dans la base de données des informations calculées à partir de celle-ci et les utilise pour vérifier efficacement les contraintes. La difficulté réside dans la définition judicieuse des informations calculées, car le coût de leur maintien (il faut les recalculer lors des mises à jour) ne doit pas dépasser le gain en rapidité qu'elles permettent d'espérer. C'est probablement la raison pour laquelle les auteurs ne considèrent que des types très restreints de contraintes d'intégrité pour lesquelles des informations calculées pertinentes ont pu être définies.

• (Abiteboul, Vianu, 1985) introduit la notion de schéma transactionnel. Un schéma transactionnel spécifie de manière procédurale les transitions d'états acceptables de la base de données eu égard aux contraintes d'intégrité. Il fournit pour cela un certain nombre d'opérations minimales sur la base qui ne violent pas les contraintes. Les contraintes d'intégrité considérées sont cependant limitées à quelques types particuliers (dépendances fonctionnelles, d'inclusion, …).

• (McCune, Henschen, 1989) utilise un démonstrateur de théorèmes. La méthode considère des transactions mono-opération et les transactions multi-opérations pour lesquelles chaque opération (autre qu'une lecture) peut être exprimée à l'aide des règles de transition données par les auteurs.

• (Qian, Wiederhold, 1986) et (Qian, Smith, 1987) proposent une méthode de simplification dite sémantique dans le sens où elle utilise des connaissances

« externes » pour simplifier les contraintes. Ces connaissances peuvent être de nature variée :

- organisation physique des données : index, localité des données, …
- statistiques sur la base.
- propriétés de certaines opérations, vues existantes, informations dérivées, facteur de blocage, …

Notons pour terminer, qu'afin de limiter l'ampleur de l'étude, on n'a pas présenté la vérification des contraintes d'intégrité dans les bases de données distribuées, voir par exemple (Gupta, Widom, 1993), et dans les bases de données déductives, voir par exemple (Olivé, 1991), (Jeusfeld, Jarke, 1991), (Küchenhoff, 1991) et (Bry, Decker, Manthey, 1988).

2. Vérification des contraintes d'intégrité dynamiques

2.1 Présentation du problème

Les contraintes d'intégrité dynamiques permettent d'exprimer des critères de cohérence d'un état de la base de données par des propriétés sur cet état et ceux qui le précèdent. La vérification de ces critères requiert donc a priori l'accès à l'historique des états de la base. Cela a deux inconvénients : (*i*) l'espace de stockage nécessaire risque vite de devenir prohibitif et (*ii*) le coût de la vérification des contraintes croit avec la taille de l'historique de la base.

Les approches de vérification de contraintes dynamiques dites *a-historiques* (history-less) s'affranchissent de la nécessité de stocker exhaustivement l'historique, d'où leur dénomination. Leur principe général est de ne conserver du présent que ce qui est nécessaire pour les vérifications ultérieures de contraintes. Elles comportent toujours trois parties :

- la première partie consiste en la construction des structures de conservation des informations pertinentes du passé, ou structures auxiliaires, à partir des contraintes dynamiques. Il peut s'agir, par exemple, de graphes (voir § 2.2), ou de relations (voir § 2.3). A chaque contrainte correspond une (ou plusieurs) de ces structures.
- la deuxième partie se déroule tout au long de la vie de la base. Elle consiste en la mise à jour du contenu des structures auxiliaires en fonction des opérations effectuées sur la base.
- enfin, la troisième partie consiste en la vérification des contraintes à l'aide des structures auxiliaires.

Dans la suite de ce paragraphe on présente les deux principales méthodes a-historiques pour la vérification de contraintes dynamiques.

2.2 Méthode des graphes de transition

La méthode des graphes de transition est proposée par l'équipe de Udo Lipeck. Elle est principalement décrite dans (Gertz, Lipeck, 1995) (Gertz, Lipeck, 1993) (Lipeck, Feng, 1988) et (Lipeck, Saake, 1987). (Ehrich, Lipeck, Gogolla, 1984) constitue la première étape significative dans l'élaboration de la méthode.

Le principe consiste à construire pour chaque contrainte dynamique un graphe de transition qui décrit les séquences d'états admissibles des objets concernés par la contrainte.

Pour une formule temporelle F, le graphe de transition d'état (ou, plus simplement, graphe) est obtenu par compositions successives. On commence donc par établir les graphes des composantes élémentaires des formules et on les *compose* jusqu'à obtenir le graphe de F. La méthode garantit que les graphes obtenus sont déterministes. Pour une formule non temporelle P, la figure 5 de la page suivante donne le graphe associé. Pour f_1 et f_2 deux formules temporelles, ou non, elle donne le graphe de $f_1 \vee f_2$ et $\Diamond f_1$. Pour les graphes des autres opérateurs voir par exemple (Lipeck, Feng, 1988).

On illustre maintenant la méthode de construction par composition en l'appliquant à la formule $F = P \vee \Diamond Q$. La première étape consiste à déterminer les graphes des deux composantes élémentaires P et Q de F. S'agissant de composantes non temporelles, on applique le premier résultat de la figure 5. Les deux graphes auxquels on aboutit sont représentés figure 3.

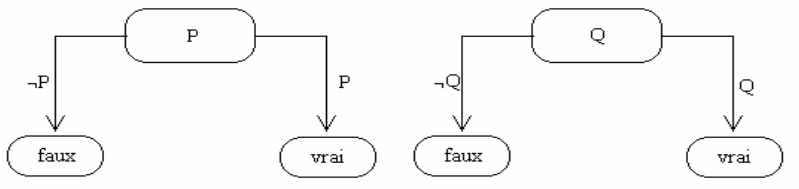

Figure 3 : Graphes de transition pour P et Q

A partir du graphe de Q et de celui donné dans la figure 5 pour l'opération $\Diamond$, on construit le graphe de $\Diamond Q$ qui est représenté figure 4.

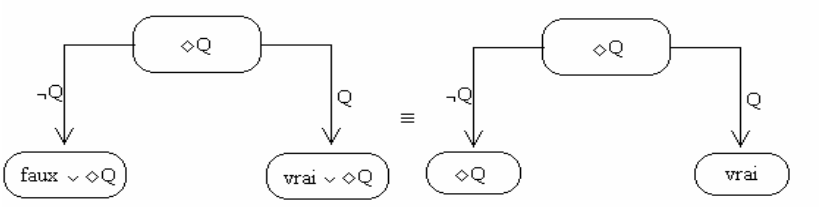

Figure 4 : Graphe de transition pour ◊Q

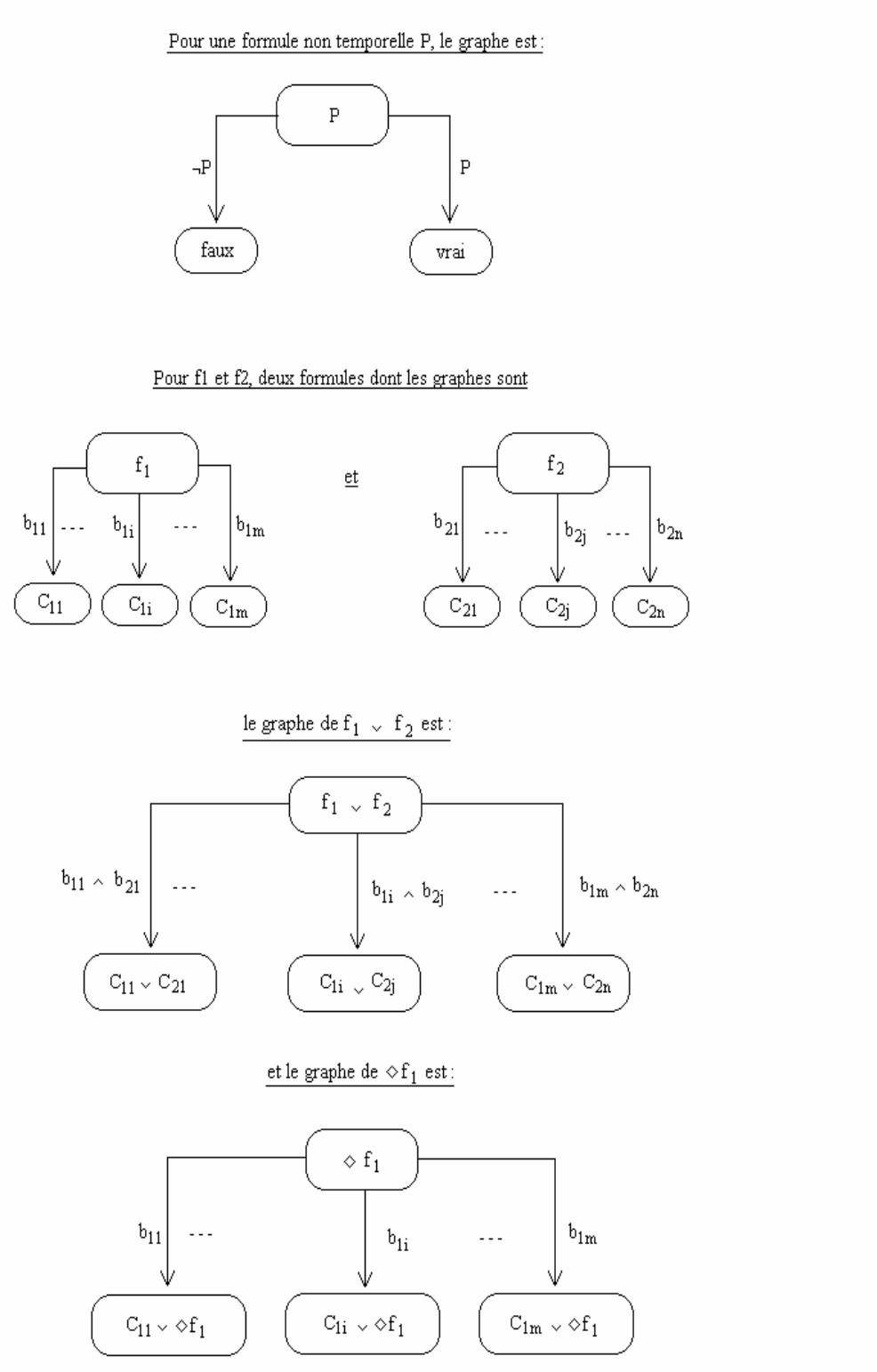

Figure 5 : Graphes de transition pour P, $f_1 \vee f_2$ et $\Diamond f_1$.

On regroupe alors les graphes de P et de ◊Q. Soit $E_P=\{P,\neg P\}$ l'ensemble des étiquettes du graphe de P et $E_{\Diamond Q}=\{Q,\neg Q\}$ celui du graphe de ◊Q. D'après le graphe donné dans la figure 5 pour l'opérateur ∨, celui de P∨◊Q comporte quatre arcs dont les étiquettes sont les formules conjonctives sur les composantes du produit cartésien $E_P \times E_{\Diamond Q}$. On a donc les étiquettes : P∧Q, P∧¬Q, ¬P∧Q, ¬P∧¬Q.

On procède de manière similaire pour obtenir le contenu des noeuds du graphe. Tous les noeuds obtenus se simplifient à la valeur de vérité **vrai**, sauf le quatrième qui contient ◊Q. On lui substitue alors le graphe obtenu à l'étape précédente pour ◊Q. Enfin, selon l'usage de la méthode on numérote les noeuds différents de **vrai**. La figure 6 ci-dessous représente le graphe de transition pour F = P∨◊Q avant et après simplification et numérotation.

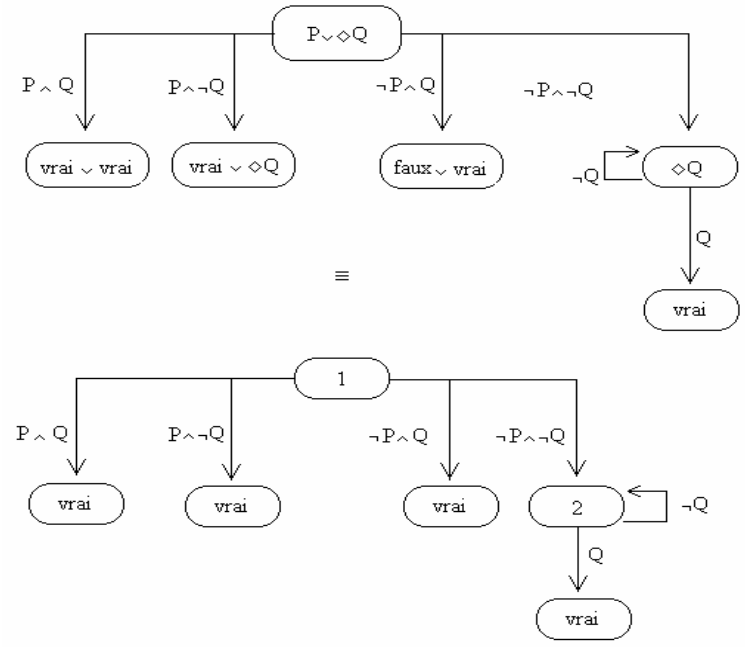

Figure 6 : Graphe de transition pour F = P∨◊Q

Cet exemple est relativement simple. La formule de la contrainte **CI_12**, voir chapitre 1, § 3.2, est plus complexe. Son graphe de transition est donné à la figure 7 (avec e ∈ **EMPLOYE**).

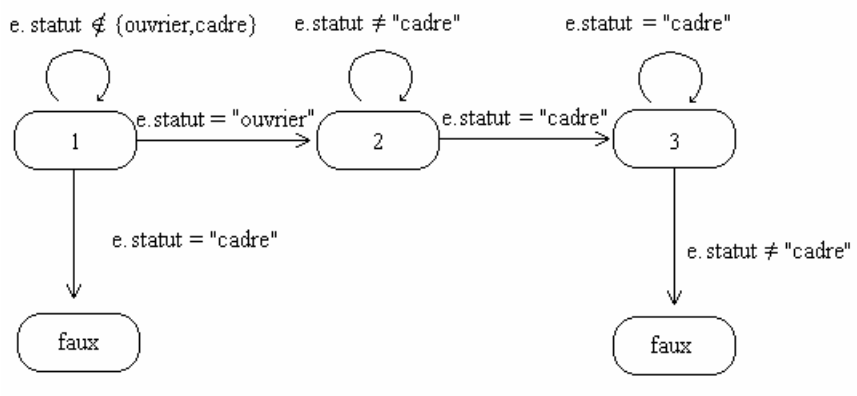

Figure 7 : Graphe de **`CI_12`**

La simplicité des graphes obtenus, même pour des formules complexes, conduit l'équipe de Lipeck à considérer la spécification des contraintes dynamiques directement sous forme de graphes de transition comme une alternative possible, dans certains cas, à leur expression en logique temporelle.

Les phases de monitoring et d'évaluation se réduisent, dans le cas de cette méthode, à un simple parcours des graphes de transition (déclenché à chaque fin de transaction) et au stockage de la position de chaque objet dans ces graphes.

Par exemple, un objet employé dans l'état 2 du graphe de la figure 6 reste dans cet état tant que son statut est différent de « cadre ». Une transaction modifiant son statut à « cadre » le déplace dans l'état 3. Dès lors, toute transaction qui modifie le statut en autre chose que « cadre » conduit au parcours de l'arc aboutissant à **`faux`** et qui signifie la violation de la contrainte (on retrouve bien la sémantique de la contrainte qui impose qu'une fois cadre, un employé le reste).

Le monitoring peut être programmé par des triggers dont l'expression est obtenue à partir des graphes de transition (Gertz, Lipeck, 1993).

2.3 Méthode des relations auxiliaires

La méthode des relations auxiliaires est présentée par Jan Chomicki dans (Chomicki, 1992), puis développée en détail dans (Chomicki, Niwinski, 1995) et (Chomicki, 1994). Dans cette méthode, les structures auxiliaires ne sont pas des graphes comme dans le cas de la méthode du paragraphe précédent, mais des relations auxiliaires dont les tuples conservent les informations pertinentes sur le passé. Ces relations auxiliaires, déterminées à partir des contraintes dynamiques, sont ajoutées au

schéma de la base et forment le schéma auxiliaire. Elles sont mises à jour lors des modifications de la base.

Considérons la contrainte **CI_13** du chapitre 1, § 3.2. Elle donne lieu à trois relations auxiliaires : (*i*) deux relations unaires $r_1(x) = \Diamond$ **DETACHEMENT(x)** et $r_2(x) =$ **◊(DETACHEMENT(x) ∧ ◊DETACHEMENT(x))** définies par ses deux sous-formules temporelles et (*ii*) une relation 0-aire r_C correspondant à la contrainte dans son intégralité. Ces trois relations composent le schéma auxiliaire de la base.

Par application d'un certain nombre de règles de réécriture on associe à ces trois relations des formules dérivées qui ne font intervenir que l'état courant (suffixe « curr ») et l'état suivant (suffixe « next ») de la base:

pour r_1 : $r_1^{next}(x) = r_1^{curr}(x) \vee$ **DETACHEMENT**$^{curr}(x)$

pour r_2 : $r_2^{next}(x) = r_2^{curr}(x) \vee$ **DETACHEMENT**$^{curr}(x) \wedge r_1^{curr}(x)$

pour r_3: $r_C^{next} = \neg(\exists x)$ (**DETACHEMENT**$^{next}(x) \wedge r_2^{next}(x)$)

A chaque nouvel état de la base de données, ces formules dérivées sont évaluées. L'état courant est celui de la base avant la transaction et l'état suivant celui en fin de transaction. Lorsqu'un de ces tuples rend une formule vraie, il est inséré dans la relation associée à la formule. Les règles de réécriture garantissent que la taille des relations ne dépend pas de la longueur de l'historique de la base, mais seulement des valeurs remarquables rencontrées.

On démontre alors que la contrainte C est vraie dans un état donné de la base si, dans cet état, r_C contient le tuple vide. Il suffit donc de vérifier en fin de transaction que r_C^{next} contient ce tuple.

Cette brève présentation de la méthode est simplifiée. En particulier, pour la première insertion dans une relation associée à une formule dérivée, ce n'est pas r^{next} mais une autre forme, r^{first} qui est évaluée. La nécessité de devoir considérer le premier état séparément se justifie par le fait qu'il est le seul à ne pas avoir de prédécesseur.

La méthode de Chomicki est reconnue plus générale que celle de Lipeck qui impose des limites très strictes sur les quantifications existentielles dans les formules. En contrepartie, elle nécessite le stockage de plus d'informations sur le passé. Cela semble indiquer que les méthodes de vérification des contraintes d'intégrité dynamiques peuvent se comparer sur la base du compromis : « plus général, moins a-historique ».

Conclusion

Les contraintes d'intégrité permettent d'exprimer des critères de cohérence sur la base de données. Dans cette première partie on a présenté un état de l'art sur les contraintes d'intégrité. On a successivement traité de leur expression puis de leur vérification.

L'étude de l'expression des contraintes d'intégrité nous a tout d'abord conduits à rappeler les avantages de leur expression déclarative, puis à en présenter la typologie généralement admise. Celle-ci distingue les contraintes statiques et les contraintes dynamiques. Les contraintes statiques permettent d'exprimer des critères de cohérence d'un état de la base de données par des propriétés de cet état. Elles sont généralement exprimées en logique du premier ordre. Les contraintes dynamiques, quant à elles, permettent d'exprimer des critères de cohérence d'un état de la base de données par des propriétés sur cet état et les états qui le précèdent. Elles sont généralement exprimées en logique temporelle.

La vérification des contraintes d'intégrité constitue un domaine de recherche ancien, qui a donné lieu à de nombreux travaux. Il devrait continuer à connaître un essor important avec l'étude du transfert aux modèles objets des méthodes de vérification élaborées pour le modèle relationnel. Le nombre des travaux rend difficile toute présentation exhaustive de ces méthodes. Nous avons néanmoins tenté de prendre en compte cette diversité, dans la mesure du possible, sans pour autant nuire au caractère synthétique de l'étude. On a considéré principalement quatre types de méthodes de vérifications de contraintes d'intégrité, chacune ayant ses avantages et inconvénients :

- les méthodes de *simplification*. Elles ne constituent certes pas à proprement parler des méthodes de vérification, mais elles permettent de simplifier les contraintes à vérifier.
- les *triggers*. Ils constituent une méthode très générale qui reporte cependant la question de l'optimisation de la vérification des contraintes sur celle de l'optimisation de l'exécution des triggers, laquelle est tout aussi ardue.

- les méthodes qui ne *s'appliquent qu'à des types restreints de contraintes*, soit du fait même de leur principe, soit parce que l'extension au cas général s'est avérée peu performante ou trop complexe.
- les méthodes *a-historiques* pour la vérification des contraintes d'intégrité dynamiques. Elles évitent le stockage et la prise en compte de tout l'historique de la base à chaque vérification de la contrainte, garantissant de ce fait un coût de vérification limité et constant.

Ces méthodes vérifient les contraintes indépendamment les unes des autres. Toutefois, plusieurs contraintes peuvent comprendre des "morceaux" communs. Des méthodes plus globales devraient alors permettre de les identifier afin de ne les évaluer qu'une fois et, ce faisant, conduire à une nouvelle forme d'optimisation de la vérification.

Une question corollaire à la définition d'un ensemble de contraintes d'intégrité, ou plus généralement de règles logiques, est celle de leur non contradiction. Cet aspect n'a pas été traité car il est relativement peu abordé dans le domaine des bases de données, contrairement à celui de l'intelligence artificielle, voir par exemple (Pipard, 1988), (Rousset, 1987) et (Bry, Manthey, 1986). Une des raisons est que, en général, l'ensemble des contraintes d'intégrité définies sur une base de données évolue peu.

Deuxième Partie

De la *Tension*

entre

Cohérence et Confidentialité

—

Secrets, Révélations et Fraudes

un Cadre Général

et

l'Etude d'un Cas Particulier

Introduction

Dans les bases de données, la cohérence et la confidentialité assurent la sécurité des données stockées. Comme on l'a vu dans la première partie, la cohérence, par les contraintes d'intégrité, vise à lier, dans un rapport « autant que possible biunivoque » (Courrège, 1965), les données de la base et le « monde réel ». La confidentialité, quant à elle, limite l'accès aux données. Ces deux fonctionnalités, généralement considérées comme importantes et, dès lors, perçues comme positives, peuvent cependant être contradictoires. En effet, les contraintes d'intégrité sont susceptibles d'être utilisées pour contourner les limitations d'accès aux données. Cela conduit à un dilemme du type « le plus cohérent, le moins secret ». (Mazumdar, Stemple, Sheard, 1988) désigne cette contradiction par le terme de *tension*.

Cette tension a son origine dans le fait que le vérificateur de contraintes d'intégrité d'un système de gestion de base de données ouvre nécessairement un *canal caché* (*covert channel*) entre l'utilisateur et le contenu de la base.

Une composante d'un système ouvre un canal caché lorsque son fonctionnement produit un effet de bord qui peut être utilisé comme un moyen de communication.

Dans le cas du vérificateur de contraintes d'intégrité, le canal caché trouve son origine dans le fait qu'en répondant « oui » ou « non » aux mises à jour sur la base, il donne des informations, positives ou négatives, sur son contenu. Il ouvre donc un canal de communication entre le contenu de la base de données et l'utilisateur.

Considérons par exemple la relation **EMPLOYE[nom, âge, ancienneté]**. Une contrainte d'intégrité typique sur cette relation est : **ancienneté ≤ âge**. Supposons alors que la valeur de l'attribut **âge** soit secrète. Pour connaître l'**âge** de l'employé Jean, un fraudeur peut procéder de la manière suivante.

Tout d'abord, il tente de mettre à jour successivement l'ancienneté de Jean pour toutes les valeurs d'ancienneté possibles. A chaque valeur proposée, le vérificateur d'intégrité évalue la contrainte. La valeur est acceptée si la contrainte est vérifiée, rejetée dans le cas contraire. Supposons que l'intervalle [1,48] soit l'ensemble des

valeurs acceptées. Ce sont les seules valeurs possibles pour l'ancienneté de Jean. La valeur 49, étant exclue, est donc $>$ à l'**âge** tandis que 48, incluse, est $\leq$ à l'**âge**. Il en résulte la double inégalité $48 \leq$ **âge** < 49. Dès lors, en vertu des propriétés de la relation $\leq$ sur les entiers naturels, **âge** $= 48$. Ainsi, l'âge de Jean, pourtant secret, a pu être obtenu par le fraudeur. La tension entre cohérence et confidentialité n'est pas la seule manière de frauder. Il en existe d'autres. Par exemple, la méthode des *traqueurs* (*trackers*)[10] (Denning, Schlöer, 1977), pour les bases de données statistiques, utilise le principe du tiers exclu sur des agrégats statistiques.

On souligne, pour les distinguer, les éléments constitutifs de la fraude de l'exemple précédent. D'une part, les tentatives successives de mises à jour conduisent à l'intervalle [1,48] et, d'autre part, la connaissance que l'on a des propriétés de la relation $\leq$ permet d'inférer que **âge** $= 48$. Ces éléments ne sont pas de même nature. Le premier est, pour ainsi dire, interne au modèle relationnel, en ce sens qu'il ne réclame aucune autre connaissance que de savoir faire une mise à jour. Le second réclame de connaître les propriétés mathématiques de $\leq$. Il est externe au modèle relationnel. Ce dernier, en effet, stocke et accède à des relations indépendamment de leurs propriétés. Ainsi stocke-t-il de la même manière une relation d'ordre ou une relation d'équivalence. Autrement dit, un système de gestion de bases de données relationnel ne « sait » pas ce qu'il stocke. La formalisation proposée dans ce travail ne s'intéresse qu'à l'aspect interne des fraudes. En cela, il se distingue de (Mazumdar, Stemple, Sheard, 1988) qui peut, parce qu'il utilise un démonstrateur de théorèmes, introduire, en plus de ce que contient une base, des axiomes qui favorisent les fraudes.

La tension entre la cohérence et la confidentialité est connue depuis longtemps, mais ses mécanismes, peu étudiés, ne sont pas, à notre connaissance, bien élucidés. L'objectif de ce travail est d'en améliorer la compréhension par une étude formelle. On peut distinguer deux niveaux de compréhension, celui de la question *qu'est-ce qu'une fraude ?* et celui de *quelles sont les situations susceptibles de générer une (des) fraude(s) ?* L'étude propose une réponse à la première question en donnant une base formelle à la phrase de tous les jours « Il y a fraude lorsqu'un secret est révélé ». En revanche, il ne prétend pas éclairer la seconde dans toute sa généralité. Les possibilités de fraude apparaissent tellement vastes qu'une formalisation générale est encore très prématurée dans le cadre de cette thèse. Le risque serait celui d'une proposition générale se révélant, très rapidement, ne pas l'être. Lorsque les promoteurs de l'actuel billet de 50 frs, celui à l'effigie de Saint Exupéry, ont assuré qu'il était absolument infalsifiable, ils n'avaient pas pensé aux premières photocopieuses couleur ! Dès lors, au regard de la seconde question, le champ de ce travail est volontairement limité. Il s'intéresse à une situation – on dira une *configuration* – particulière qui est la source d'une fraude. En revanche, il vise à

[10] L'annexe A présente la méthode de fraude par les traqueurs utilisée dans les bases de données statistiques.

explorer autant que possible tous ses aspects. Le pari est que des études limitées mais « fouillées » permettront, à terme, des approches plus ambitieuses.

L'étude se situe à l'intérieur du modèle relationnel, choisi pour son expressivité, sa simplicité formelle et son élégance. Elle se développe à partir du parti-pris de « dépouiller » le fraudeur de toute connaissance autre que relationnelle. Autrement dit, les fraudeurs autorisés ne connaissent que le langage SQL. Sont donc exclues des connaissances externes du type « R est un ordre » ou « R est réflexive » ou encore du type théorème. Sont aussi exclues les connaissances relatives aux clés des relations. L'étude ne tient pas compte de l'existence de clés, sauf celles, triviales, faites de tous les attributs. Ce parti-pris de l'austérité est essentiel pour atteindre l'objectif de cerner comment certains mécanismes, parmi les plus primitifs, mis en oeuvre dans un système de gestion de base de données – et *eux seuls*, sans apport extérieur –, permettent une fraude.

Si la configuration envisagée ici, bien que réaliste, est limitée, la méthode pour l'étudier relève d'une généralité plus grande et, ce faisant, peut se révéler applicable à d'autres configurations et d'autres occurrences de fraude. Elle s'inscrit, en effet, dans la tradition des méthodes en programmation, qui distinguent les niveaux de spécification, de programmation et, enfin, d'étude de la complexité. Les niveaux correspondants sont ici formel, de faisabilité et d'efficace.

L'objet du niveau formel est d'expliciter la propriété qui, en substance, fonde la fraude. Les fraudeurs sont souvent des Monsieur Jourdain. Ils appliquent, par intuition, une propriété que, par ailleurs, ils ignorent. L'objet de la faisabilité est de montrer que cette propriété peut effectivement être calculée dans la configuration considérée, compte tenu des droits d'accès. Enfin, le dernier niveau, celui de l'efficace, vise à répondre à la question : quelles sont les conditions les meilleures pour le fraudeur ? ou, autrement dit, quand obtient-il les meilleurs résultats ?

Le plan de cette partie est le suivant :

- le chapitre 3 fixe le cadre de l'étude en donnant une définition formelle aux termes de secret, révélation et fraude. Puis il présente le modèle de sécurité dans lequel l'étude se situe, ainsi que la notion de canal caché.
- le chapitre 4 décrit la configuration particulière de droits d'accès et de contraintes d'intégrité considérée. Le secret **S** dont elle permet une fraude est présenté. Le chapitre se termine par une discussion sur la pertinence de cette configuration.
- le chapitre 5 établit, dans le cadre fonctionnel des correspondances, la propriété qui explique la fraude du secret **S**. Il relève de ce que l'on a appelé précédement le niveau formel.

- le chapitre 6 explique le lien entre cette propriété et le canal caché. Il constitue l'étude de faisabilité.
- le chapitre 7 étudie formellement la propriété, la relie aux treillis de Galois, et établit la « géographie » des cas où la fraude conduit à une connaissance exacte de ce qui est secret.
- le chapitre 8, enfin, étudie l'efficace de la fraude. Il s'intéresse à une classe particulière de contraintes d'intégrité, dites manichéennes, particulièrement favorables au fraudeur.

Chapitre 3

Cadre de l'Etude

Ce chapitre fixe le cadre de l'étude. Il présente, § 1*, un exemple de la fraude étudiée et propose,* § 2*, une définition formelle des notions de révélation, de secret et de fraude. Enfin, il présente le modèle de sécurité considéré,* § 3*, ainsi que la notion de canal caché,* § 4.

1. Un exemple

On présente dans ce paragraphe un exemple de la fraude que l'on étudie. La définition précise de la configuration qui permet cette fraude est reportée au chapitre suivant.

Soit la relation **COMBINAISON[avion, mission, missile]** qui stocke les combinaisons autorisées d'avions, de missions et de missiles. La contrainte d'intégrité $\forall$v,x,y **COMBINAISON(v,x,y)** $\rightarrow$ **B(x,y),**

basée sur la relation **B[mission, missile]**, assure que pour chaque tuple de **COMBINAISON** le missile est adéquat pour la mission. La figure 1 ci-dessous donne une extension de la relation **COMBINAISON** et une extension de la relation **B**. On suppose que les valeurs de l'attribut **mission** de la relation **COMBINAISON** sont secretes (en italique dans la figure).

mirage	*interception*	mica
rafale	*interception*	mica
rafale	*bombardement*	apache
jaguar	*bombardement*	apache

Relation **COMBINAISON**

interception	apache
interception	mica
interception	super530
bombardement	apache
reconnaissance	magic

Relation **B**

Figure 1 : Exemple d'extensions pour les relations **COMBINAISON** et **B**

On note M l'ensemble des valeurs de l'attribut **mission** associées à l'avion rafale. Ici, M = { interception, bombardement}. De par les privilèges sur la relation **COMBINAISON**, l'ensemble M est secret. Toutefois, un fraudeur, s'il peut mettre à jour l'attribut **mission** de la relation **COMBINAISON**, peut révéler M de la manière suivante.

Tout d'abord, il exécute la requête

τ(m)←UPDATE COMBINAISON SET missile=m WHERE avion=« rafale »

pour toutes les valeurs m de l'attribut **missile**. La transaction **τ(m)** vise à modifier l'attribut **missile** dans *tous* les tuples où « rafale » est l'avion. Si m = « mica » le vérificateur d'intégrité refusera la requête puisque le couple (bombardement,mica) n'appartient pas à la relation **B**, de même la refusera-t-il pour m = « super530 » ou m = « magic ». Il l'acceptera, en revanche, pour m = « apache ».

De ces rejets et de ces accords, le fraudeur déduit que $M \times \{apache\} \subset \mathbf{B}$. Dès lors, M est inclus dans le plus grand sous-ensemble K de missiles tels que $K \times \{apache\} \subset \mathbf{B}$. L'ensemble K ne dépend que de la relation **B**, pas de la relation **COMBINAISON**. Il vaut, dans l'exemple, {interception, bombardement}. Puisque la relation **B** n'est pas secrète, le fraudeur peut calculer K qui, de ce fait, constitue une révélation de M. Dans l'exemple, K = M : la révélation obtenue par le fraudeur est exacte. Ce n'est pas toujours le cas. En effet, pour l'avion « jaguar » le procédé précédent conduit au même K, à l'intérieur duquel « interception » devient indésirable.

L'étude formelle de la fraude que l'on vient de présenter est l'objet de ce travail.

2. Notions de révélation, de secret et de fraude

L'exemple précédent a utilisé les termes « révélation », « secret » et « fraude » dans leur sens commun. On les formalise maintenant à partir de la notion de requête, démarche, en quelque sorte, naturelle compte tenu que les requêtes sont le seul moyen pour accéder aux données d'une base.

Le cadre formel retenu est celui du modèle relationnel. On suppose le lecteur familier de ce modèle. On commence par préciser les notations utilisées.

2.1 Notations relationnelles

Un univers **U** est un ensemble de domaines. Un schéma sur **U** est une famille ordonnée de domaines, notée entre crochets. Si X et Y sont des éléments de **U** alors [X,Y] et [X,Y,X] sont des schémas sur **U**. Tout élément d'un schéma S, i.e. un indice associé à un domaine, est un attribut. On note R[S] une relation sur le schéma S, et Dom(A) le domaine de l'attribut A de S. Dans la suite, un nom de relation fait référence aussi bien à la relation elle-même qu'à son extension.

On note R($\rightarrow$A) la projection de R sur A, i.e. le sous-ensemble de Dom(A) qui regroupe les valeurs de l'attribut A qui apparaissent dans les tuples de R. Cette projection est la partie *active* du domaine de l'attribut A. Cette notation peut être étendue pour un ensemble d'attributs.

2.2 Notion de révélation

Soit **U** un univers. On note **K**(**U**) l'ensemble des relations sur **U**. Pour n'importe quel état bd de la base de données BD, on note $\underline{\mathbf{K}}$(bd) toutes les relations calculables sur bd avec l'algèbre relationnelle. Ainsi, $\underline{\mathbf{K}}$(bd) $\subset$[11] **K**(**U**). Une *requête relationnelle* **q**, sur la base de données BD, ou simplement une *requête* sur BD, est une expression d'algèbre relationnelle sur *seulement* les relations de BD. Elle définit une fonction qui associe à tout état bd de BD une relation **q**(bd) dans $\underline{\mathbf{K}}$(bd).

Etant donné **Pr**(E_1,E_2) un prédicat d'arité 2 sur **K**(**U**), une fonction **rv** qui associe à tout état bd d'une base de données une relation dans **K**(**U**) est une **Pr**-révélation de la requête **q** si, pour tout état bd, on a **Pr**(**q**(bd),**rv**(bd)). Une **Pr**-révélation **rv** de la requête q est *relationnelle* s'il existe une requête relationnelle Q telle que, pour tout état bd, Q (bd) = **rv**(bd).

Quand le prédicat **Pr** est le prédicat égalité « = » sur **K**(**U**) (resp. l'inclusion ensembliste « $\subset$ », l'inclusion inverse « $\supset$ »), la révélation est *exacte* (resp. *par le haut*, *par le bas*). Un autre cas, enfin, est celui où **Pr**(E_1,E_2) $\equiv E_1 \cap E_2 \neq \emptyset$, pour lequel on propose le terme de révélation *modérée*. Dans toute l'étude, les révélations considérées seront par le haut et exactes.

Soit une Pr-révélation **rv** de **q**. On appelle **silence** de **rv** la fonction **q** – **rv** et **bruit** la fonction **rv** – **q**. Si bd désigne l'état de la base, (**q** – **rv**) (bd) regroupe les tuples de **q** non révélés par **rv**, de même (**rv** – **q**) (bd) les tuples de **rv** qui n'appartiennent pas à **q**. Par définition, le silence d'une révélation par le haut est vide. De même, le bruit d'une révélation par le bas est vide. En revanche, le bruit et le silence d'une révélation modérée ne sont pas vides.

Exemple – Soit E une relation mono-attribut et **q** la requête `SELECT * FROM E`. Si **rv** est une révélation par le haut de **q**, alors, pour tout état bd de la base, **q**(bd)$\subset$**rv**(bd). Ainsi, pour un état bd_0 de la base pour lequel E = ((1),(2)), noté plus simplement E = {1,2}, si par exemple, **rv**(bd_0) = {0,1,2,3}, alors **bruit**(bd_0) = {0,3} et **silence**(bd_0) = $\emptyset$. La figure 2, ci-dessous, représente graphiquement la disposition du

[11] Dans la suite, le symbole « $\subset$ » signifie inclusion ou égalité, ce qui est parfois noté par « $\subseteq$ ».

bruit et du silence pour des exemples de **rv**(bd_0) selon que **rv** est une révélation par le haut, par le bas, modérée et exacte.

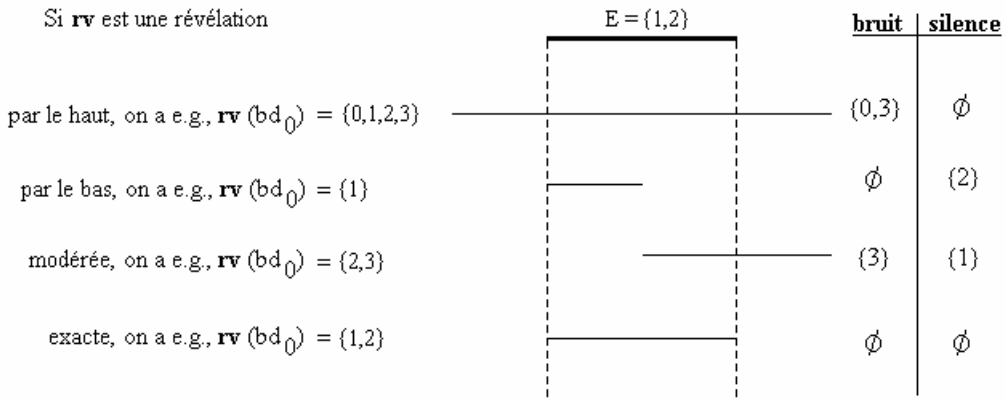

Figure 2 : Exemples de révélations par le haut, par le bas, modérée et exacte de E

La définition proposée pour la notion de révélation généralise celle proposée par (Mazumdar, Stemple, Shread, 1988), qui ne couvre que le cas des révélations par le bas. En revanche, elle ne formalise pas tous les cas de révélations. Par exemple, découvrir qu'une relation binaire est réflexive, ou qu'elle est un ordre, ou encore qu'elle possède une clé constituent des révélations de propriétés, en quelque sorte des révélations « méta », qui ne sont pas formalisées dans la définition proposée.

2.3 Notions de secret et de fraude

On appelle *secret* sur une base de données, toute requête relationnelle **q** que le système de gestion de bases de données refuse d'évaluer compte tenu des droits d'accès définis sur cette base. Une **Pr**-*fraude* d'un secret **q** est une **Pr**-révélation *calculable* **rv** de **q**. L'adjectif « calculable » n'est pas utilisé ici dans le sens qu'il a dans la théorie de la calculabilité. Il signifie « disposer d'un moyen d'évaluation (un algorithme) de la requête **rv** en dépit des droits d'accès ». Ce moyen peut être quelconque. Si la **Pr**-révélation **rv** n'est pas un secret, il suffit de la soumettre au système de gestion de bases de données pour qu'elle soit évaluée. Si elle est un secret, le moyen devra naturellement être différent.

Dans l'exemple du § 1, le secret est la requête **`SELECT mission FROM COMBINAISON WHERE avion = «Rafale»`** qui calcule l'ensemble M. La fonction qui associe à tout état de la base l'ensemble noté K est une révélation du secret par le haut.

Cette fonction est une fraude par le haut du secret puisqu'on dispose, via la transaction **τ(m)**, d'un moyen de l'évaluer.

Lorsqu'une requête est non calculable, c'est un secret, puisqu'il n'y a aucun moyen de l'évaluer, en particulier celui de la soumettre au système de gestion de bases de données. On souligne que l'inverse est faux : un secret peut être calculable. Par exemple, tout secret **q** est, trivialement, une =-révélation de lui-même. La requête ne peut être évaluée par le système de gestion de bases de données puisque c'est un secret. Mais, si l'on connaît, par ailleurs, un moyen de l'évaluer en dépit des droits d'accès, alors le secret devient une =-fraude de lui-même. C'est un *secret de Polichinelle* (!), i.e. un secret qui est sa propre =- fraude.

En résumé, les notions de révélation, de secret et de fraude s'articulent entre elles comme le schématise la figure ci-dessous.

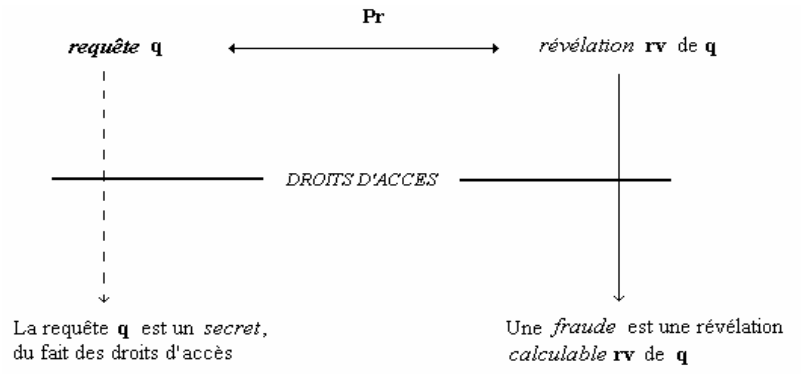

Figure 3 : Liens entre les notions de secret, de révélation et de fraude

Pour qu'il y ait fraude, il faut d'abord qu'il y ait secret, c'est à dire une requête **q** dont les droits d'accès interdisent le calcul. Ensuite, moyennant de s'être donné un prédicat **Pr**, il faut avoir établi qu'une requête **rv** était une **Pr**-révélation de **q**. Enfin, il faut disposer d'un moyen de calculer **rv**. La révélation **rv** est alors une **Pr**-fraude du secret **q**.

3. Modèle de sécurité

On présente maintenant le modèle de sécurité considéré, qui, par les droits d'accès qu'il permet de spécifier, induit des secrets. Selon (Bancilhon, 1977), un modèle de sécurité est défini par la réponse aux quatre questions suivantes : (*i*) qu'est-ce qui est protégé ? (*ii*) contre qui/quoi met-on une protection ? (*iii*) comment la protection est-elle maintenue ? (*iv*) que signifie le verbe « protéger » ?

Dans ce travail, la réponse à ces questions est :

(*i*) des données de bases de données relationnelles,

(*ii*) protégées contre les utilisateurs effectuant des opérations pour lesquelles ils n'ont pas les privilèges nécessaires,

(*iii*) par l'acceptation ou le rejet de ces opérations,

(*iv*) et où « protéger » signifie autoriser ou non l'exécution des opérations en fonction de l'identité de l'utilisateur et de règles spécifiant, pour chaque donnée, les privilèges de chaque utilisateur.

La réponse (*iv*) positionne le modèle de sécurité retenu dans le cadre des modèles de sécurité dits discrétionnaires (*discretionary models of security*) (Castano et *al.*, 1994, p 39). Ce type de modèle est général et, contrairement à d'autres, comme les modèles multi-niveaux[12] qui requièrent que données et utilisateurs soient classifiés, il est utilisable dans la majorité des situations (Castano et *al.*, 1994, p 81, p 137). C'est pourquoi les principaux systèmes de gestion de bases de données l'utilisent.

Dans notre étude, les deux seules opérations considérées sur une base de données sont la lecture et la mise à jour. A ces deux opérations sont associés les privilèges READ et UPDATE dont on donne maintenant les définitions, issues de la norme SQL-2 (SQL, 1992) :

- privilège READ : accordé sur l'attribut A d'une relation R, ce privilège permet la *lecture* de la valeur de A pour tout tuple de R. En l'absence du privilège READ sur un attribut A on dira que A est *secret*.

- privilège UPDATE : accordé sur l'attribut A d'une relation R, ce privilège permet la *mise à jour* de la valeur de A pour tout tuple de R.

Par exemple, pour la relation **EMPLOYE** de l'introduction, avec le privilège READ sur **nom** et le privilège UPDATE, mais pas READ, sur **âge**, on peut exécuter avec succès la commande SQL

```
UPDATE EMPLOYE SET âge = 30 WHERE nom = "Jean"
```

tandis que les commandes

```
SELECT âge FROM EMPLOYE
```

et

```
SELECT nom FROM EMPLOYE WHERE âge > 25
```

seront toutes deux rejetées.

---

[12] Le principe des modèles de sécurité multi-niveaux est présenté brièvement dans l'annexe B.

En SQL-2, le privilège UPDATE peut être accordé indépendamment du privilège READ. Quoique surprenante, cette indépendance est assurée pour des raisons pratiques (Melton, 1995) et sera maintenue dans SQL-3 (SQL, 1994). Les principaux systèmes de gestion de base de données, ainsi que le système R (Griffiths, Bradford, 1976), l'assurent aussi.

En revanche, certain modèles théoriques de sécurité tels que le modèle Action Entité (Bussolati, Fugini, Martella, 1983) (Fugini, Martella, 1984) (Castano et *al.*, 1994) ou le modèle de (Bertino, Weigand, 1994) dans le cas des bases de données objets, imposent que le privilège UPDATE ne puisse être accordé indépendamment du privilège READ.

4. Notion de canal caché

4.1 Définition

La notion de canal caché (*covert channel*) a été initialement introduite dans (Lampson, 1973). Comme on l'a déjà mentionné dans l'introduction, une composante d'un système ouvre un canal caché lorsque son fonctionnement produit un effet de bord qui peut être utilisé comme un moyen de communication. Le système en question n'est pas forcément un système informatique et par conséquent, la notion de canal caché n'est pas liée à ce domaine. Dans la suite, on l'applique bien sûr à ce domaine.

Il découle de la définition précédente qu'un canal caché ne constitue une menace pour la sécurité d'un système informatique que lorsque l'information qu'il est susceptible de transférer peut l'être vers un récepteur qui n'y a pas normalement accès.

Selon la nature de la composante du système qui ouvre le canal caché, on distingue (DoD, 1985) les canaux cachés de stockage (*storage channels*) et les canaux cachés temporels (*timing channels*).

Dans le cas d'un canal caché de stockage, la composante utilisée est une zone de stockage sur laquelle un processus émetteur écrit directement ou indirectement et qu'un processus récepteur lit. Un nom de fichier peut constituer une telle zone de stockage. Par exemple, en lui donnant le nom **`mission_127`** le processus émetteur indique qu'il existe une mission dont le numéro est 127.

Dans le cas d'un canal caché temporel, la composante utilisée est une ressource du système (par exemple le processeur) dont le processus émetteur module l'utilisation, de manière telle que le temps de réponse observé par le processus récepteur en soit affecté. A partir des variations observées, ce dernier en déduit l'information

transmise[13]. (He, Gligor, 1992) note que les canaux cachés temporels ont été relativement peu étudiés, contrairement aux canaux cachés de stockage.

Quel que soit le type de canal caché considéré, son exploitation requiert un certain degré de synchronisation entre le processus émetteur et le processus récepteur. (Landwehr et *al.*, 1994) fait remarquer que la frontière entre ces deux types de canaux cachés n'est pas délimitée avec précision. Tous deux requièrent un certain degré de synchronisation entre les processus émetteurs et récepteurs et utilisent une composante du système qui est partagée par les deux processus. En pratique, le type du canal caché est déterminé en fonction de la manière dont on le détecte. S'il peut être détecté par l'étude des flux d'informations (*information flow analysis*), des spécifications, ou du code du système lui-même, alors il s'agit d'un canal caché de stockage.

La suppression des canaux cachés d'un système n'est pas toujours possible. Par exemple, lors de la phase de certification B2[14] du système d'exploitation Multics, des canaux cachés ont été découverts et n'ont pu être supprimés du fait de la perte en fonctionnalité que cela occasionnait (Loepere, 1989). Dans d'autres cas, l'élimination des canaux cachés conduit à une baisse des performances du système (DoD, 1985). L'étude de leur suppression doit donc se faire sur le mode du compromis.

4.2 Application aux bases de données

La notion de canal caché a été appliquée aux bases de données par Simon Wiseman dans (Wiseman, 1989) et (Wiseman, 1990). Wiseman considère les mécanismes de confidentialité dans les bases de données. Eux-mêmes sont la source de canaux cachés. Ainsi, en répondant que l'on ne dispose pas des droits suffisants pour accéder à une donnée, le système de confidentialité révèle la présence de cette donnée dans la base de données. En assimilant la réponse du système à une écriture dans une zone partagée par le système de confidentialité et l'utilisateur, on peut identifier ce cas à celui des canaux de stockage.

L'annuaire électronique du Minitel s'avère être une illustration simple et concrète du cas étudié par Wiseman. Interrogé sur le numéro de téléphone d'un abonné de la Liste Rouge pour lequel on connaît le nom et la ville de résidence, le Minitel répond : « Il existe des réponses en Liste Rouge : France Télécom s'est engagé à ne pas les communiquer ».

---

[13] Un autre exemple utilisant les défauts de page (page fault) est décrit à http://www.lilli.com/timing-chn.html.

[14] Il s'agit d'un niveau de certification de sécurité du département américain de la défense, voir par exemple (DoD, 1985).

Ce message ne délivre pas le numéro de téléphone, mais confirme qu'il existe bien un abonné sous ce nom et habitant cette ville. La combinaison (nom, ville) existe donc dans la base, ce qui constitue une information sur son contenu. On peut l'exploiter pour déterminer par tâtonnements la ville de résidence d'un abonné sur Liste Rouge, en soumettant successivement différentes combinaisons (nom, ville). La méthode est d'autant plus efficace que le nom de l'abonné est peu répandu.

Le canal caché ouvert par le mécanisme élémentaire de confidentialité de l'annuaire électronique du Minitel permet donc de cerner la ville de résidence, voire l'adresse, d'un abonné, même sur Liste Rouge.

Dans la suite de cette étude on considère le canal caché ouvert par le vérificateur d'intégrité du système de gestion de bases de données. Comme on l'a vu en introduction, son acceptation ou son refus des mises à jour donne indirectement, d'où effet de bord, des informations sur le contenu de la base de données. Comme pour les systèmes de confidentialité étudiés par Wiseman, et sous les même hypothèses, on peut assimiler le canal caché ouvert par le vérificateur d'intégrité à un canal caché de stockage.

Chapitre 4

La Configuration et le Secret

Ce chapitre décrit, § 1*, la configuration particulière de droits d'accès et de contraintes d'intégrité considérée dans cette étude. Le secret qu'elle engendre et dont elle permet une fraude par le haut, est ensuite présenté,* § 2*. Le chapitre se termine,* § 3*, par une discussion sur la pertinence de cette configuration.*

1. Configuration

La configuration considérée est celle de toute base de données avec une relation R[V,X,Y] et une relation B[X,Y] telles que :

(*i*) l'attribut X de R est secret (pas de privilège READ) et l'attribut Y de R peut être mis à jour (privilège UPDATE). Les autres attributs ne sont pas secrêts.

(*ii*) la contrainte d'intégrité suivante est définie

$$C \equiv \forall v \in V, \forall x \in X, \forall y \in Y,\ R(v,x,y) \rightarrow B(x,y).$$

La contrainte d'intégrité C est dite *2-aire* par référence au fait que la partie droite de l'implication est composée d'une relation d'arité 2, ou relation 2-aire[15]. Cette contrainte s'exprime aussi en Datalog par la clause **panique** :−R(v,x,y), *not* B(x,y) qui dérive **panique**, d'arité 0, si la contrainte n'est pas vérifiée (Harinarayan, Gupta, 1995). Elle impose que la projection R(→(X,Y)) soit un sous ensemble de B. On note **r**(C) la relation 2-aire B qui figure du côté droit de l'implication. Par défaut, toute paire d'attributs X et Y d'une relation R[V,X,Y] est contrainte par sa *contrainte 2-aire naturelle*, i.e. la contrainte 2-aire C telle que **r**(C)=Dom(X)×Dom(Y). De plus, si plusieurs contraintes 2-aires sont définies sur les mêmes attributs X et Y d'une relation R, la proposition suivante montre que leur conjonction est encore une contrainte 2-aire.

[15] On insiste sur la différence entre relation 2-aire et relation binaire. Une relation binaire est définie sur un ensemble N comme un sous ensemble du produit N×N. Une relation 2-aire est définie sur un schéma [S,T] de longueur 2 comme tout sous ensemble du produit S×T. Les deux notions sont proches et il est aisé de passer de l'une à l'autre. Une relation 2-aire sur [S,T] peut être vue comme une relation binaire sur $S \cup T$ et, réciproquement, une relation binaire sur N peut être vue comme une relation d'arité 2 sur [S,T], avec $S = r^{-1} N$ et $T = r^{-1} N$.

En conséquence, on peut considérer que tout couple d'attributs est contraint par une seule contrainte 2-aire.

Proposition 4.A – Si C_1 et C_2 sont des contraintes 2-aires sur les attributs X et Y d'une relation R[V,X,Y], alors $C_1 \wedge C_2$ est une contrainte 2-aire sur les attributs X et Y. De plus, $\mathbf{r}(C_1 \wedge C_2) = \mathbf{r}(C_1) \cap \mathbf{r}(C_2)$.

preuve – On pose $B_1 = \mathbf{r}(C_1)$ et $B_2 = \mathbf{r}(C_2)$. Alors,

$[\forall v \in V, \forall x \in X, \forall y \in Y \quad R(v,x,y) \rightarrow B_1(x,y)] \wedge [\forall v \in V, \forall x \in X, \forall y \in Y \; R(v,x,y) \rightarrow B_2(x,y)]$

$\Leftrightarrow \forall v \in V, \forall x \in X, \forall y \in Y \;\; [R(v,x,y) \rightarrow B_1(x,y)] \wedge [R(v,x,y) \rightarrow B_2(x,y)]$

$\Leftrightarrow \forall v \in V, \forall x \in X, \forall y \in Y \;\; R(v,x,y) \rightarrow [B_1(x,y) \wedge B_2(x,y)]$

$\Leftrightarrow \forall v \in V, \forall x \in X, \forall y \in Y \;\; R(v,x,y) \rightarrow [B_1 \cap B_2](x,y)$

□

La disjonction de contraintes 2-aires n'est pas une contrainte 2-aire. Pour simplifier les notations, on supprime dans la suite, sauf mention contraire, les quantificateurs universels dans l'expression des contraintes d'intégrité.

2. Secret

Dans la configuration considérée, la requête **`SELECT X FROM R WHERE V=v`**, ne peut être exécutée puisque X est secret – voir point (*i*). Elle constitue donc un secret, que l'on note **S**(v) dans la suite.

L'exemple du chapitre 3, § 1, est une occurrence de cette configuration. La relation **`COMBINAISON`** joue le rôle de la relation R tandis que les attributs **`mission`** et **`missile`** jouent respectivement le rôle des attributs X et Y de R. Cet exemple a montré l'existence, dans le cadre de la configuration considérée, d'une fraude par le haut du secret **S**(v). Dans les chapitres 5 et 6 suivants, on établit formellemement cette fraude.

3. Discussion

La configuration considérée peut sembler limitée et irréaliste. On a déjà dit dans l'introduction pourquoi le champ de l'étude était volontairement limité. On souligne cependant que :

- la configuration peut être reproduite, comme on le verra au chapitre 6, § 3, dans la norme SQL-2 (SQL, 1992) et, avec elle, dans tous les systèmes qui l'implémentent ;

- les relations 2-aires sont fréquentes en pratique (les ordres, la différence, les relations de « voisinage » en sont des exemples) et donc aussi les contraintes 2-aires. De plus elles couvrent un large éventail de contraintes, allant des contraintes statiques aux contraintes de transition, voir chapitre 1, § 3. Les deux exemples suivants montrent comment des contraintes d'intégrité classiques se ramènent à des contraintes 2-aires.

Soient R[V,X], S[V,Y] et B[X,Y] des relations sur **U** et la contrainte $R(v,x) \wedge S(v,y) \rightarrow B(x,y)$. Si RS[V,X,Y] est la vue égale à la jointure de R et S, alors la contrainte précédente peut être remplacée par la contrainte 2-aire $RS(v,x,y) \rightarrow B(x,y)$ sur la vue RS. On souligne que l'inverse est faux. Ainsi, si R[X,Y,Z] vérifie la contrainte $R(V,X,Y) \rightarrow B(x,y)$, il n'est pas nécessairement vrai que les deux vues $R(\rightarrow(V,X))$ et $R(\rightarrow(V,Y))$ vérifient $R(\rightarrow(V,X))(v,x) \wedge R(\rightarrow(V,Y))(v,y) \rightarrow B(x,y)$.

De la même manière, si V est une clé pour la relation R[V,X], considérons la contrainte, ou *règle de transition* (Delannoy, 1996), $R(v,x) \wedge \circ R(v,x') \rightarrow B(x,x')$ dans laquelle $\circ$ est l'opérateur temporel[16] signifiant « dans l'état précédent » (Manna, Pnueli, 1991, p 192). On montre aisément que cette règle de transition peut être remplacée par la contrainte 2-aire $\underline{R}(v,x,x') \rightarrow B(x,x')$ où la relation $\underline{R}$[V,X,X] est égale à l'ensemble des tuples (v,x,x') tels que (v,x) est le tuple de clé v dans l'état courant de la relation R et (v,x') le même tuple de clé v dans l'état précédent de la relation R.

- il est important de garder à l'esprit que l'on traite de fraude. Or, on ne peut pas penser la fraude dans les catégories mentales de l'honnêteté. Car, ce que l'honnêteté aurait tendance à considérer commme irréaliste est en fait le « pain béni des fraudeurs ». Ainsi, spontanément, on n'imagine pas un administrateur de base de données recruté pour accorder des privilèges à des espions ! Mais si l'administrateur gère plusieurs centaines d'utilisateurs et de tables, il est fort probable qu'il accorde, un jour, des privilèges par erreur. Il en est de même des bogues et des programmeurs : les programmeurs sont supposés écrire des programmes, pas des bogues, mais les bogues surviennent pourtant. La fraude se glisse dans ces manquements. Même statistiquement peu fréquents, ils n'en sont pas moins réels et peuvent concerner des situations sensibles (applications militaires, commerciales, …).

---

[16] Pour une présentation formelle de la logique temporelle voir le chapitre 10, § x et § y.

Chapitre 5

Une Révélation par le Haut du Secret S(v)

Dans ce chapitre, on établit, § 2, *une propriété sur les relations et on montre,* § 4, *comment elle conduit à une révélation relationnelle par le haut du secret* **S**(v). *Pour l'exposé de la propriété, on se place, par commodité formelle, dans le cadre fonctionnel des correspondances, que l'on introduit,* § 1. *On établi à cette occasion un certain nombre de propriétés sur les correspondances qui seront utilisées dans les chapitres suivants.*

1. Notion de correspondance

1.1 Définition

Les correspondances sont définies dans (Bourbaki, 1954, p 72). Nous les utilisons dans une acception légèrement plus restrictive. Soient X et Y des ensembles. Une *correspondance* f *de* X *dans* Y, notée $f : X \multimap Y$, est une fonction $X \to 2^Y$, où 2^Y désigne l'ensemble des parties de Y, telle que :

(5.1a) $\forall x \in X,\ f\,x \neq \varnothing$ et (5.1b) $Y = \bigcup_{x \in X} f\,x$[17].

Etant donnée une correspondance $f : X \multimap Y$, X est son *ensemble de définition* **def**(f), Y son *ensemble de valeurs* **val**(f). L'assertion « la correspondance f est *définie sur* X (resp. *valuée sur* Y) », ou plus simplement, « f est $X \multimap Y$ » implique que

[17] Dans la suite, lorsqu'il n'y a pas d'ambiguïté, les parenthèses sont omises dans les notations fonctionnelles. Ainsi, on écrira f x pour f(x).

def(f) = X (resp. **val**(f) = Y). Une correspondance $f: X \multimap Y$ s'étend facilement à une fonction $2^X \to 2^Y$: pour tout sous-ensemble E de X, l'*image* f E de E par f est le sous-ensemble $\bigcup_{x \in E} f\,x$ de Y. Donc, la propriété (5.1b) se réécrit $Y = f\,X$.

La restriction de toute fonction $g: X \to 2^Y$ au sous-ensemble $X' = \{ x \in X \mid g\,x \neq \varnothing \}$ définit une correspondance $X' \multimap g\,X'$, notée **cor**(g) : $\forall x \in X'$, **cor**(g) (x) = g x.

On souligne que, du fait de (5.1), tous les x dans X sont « connectés » à au moins un y de Y et *vice versa*. Aussi, contrairement à une fonction $X \to 2^Y$, une correspondance ne peut pas être étendue canoniquement à un sur-ensemble de X, ou à un sur-ensemble de Y[18]. Dans la terminologie des bases de données cela revient à dire que X et Y sont des *domaines actifs (active domain)* et que les correspondances sont *indépendantes des domaines* (*domain-indépendant*).

Une correspondance $f: X \multimap Y$ est une *correspondance singleton* si $\forall x \in X$, $|\,f\,x\,| = 1$. C'est une *correspondance partition* si la famille $(f\,x\,;\,x \in X)$ de sous-ensembles de Y est une partition[19] de Y. Il existe une *correspondance vide* unique, notée [∅], avec **def**([∅]) = val([∅]) = ∅. Enfin, la correspondance $f: X \multimap Y$ telle que $\forall x \in X$, $f\,x = Y$, est nommée *correspondance produit* de X dans Y et notée [X×Y].

1.2 Correspondances Associées

Si $f: X \multimap Y$ est une correspondance, alors la fonction $g: Y \to 2^X$ définie par $\forall y \in Y$, $g\,y = \{ x \in X \mid y \in f\,x \}$ est la correspondance définie sur Y et valuée dans X. Elle est appelée la *correspondance inverse* $f^{\sim}$ de f.

Considérons maintenant la fonction $g: X \to 2^Y$ définie par $\forall x \in X$, $g\,x = Y - f\,x$. La correspondance **cor**(g) est appelée la *correspondance complément* f^{C} de f. Elle est définie sur $X^{C} \subset X$ et valuée sur $Y^{C} \subset Y$. Plus simplement, $f^{C}: X^{C} \multimap Y^{C}$. Notons que, en général, $f \neq f^{CC}$. Par exemple, pour [X×Y] on a $[X \times Y]^{CC} = [X \times Y]^{C} = [\varnothing]$.

Exemple – Soit la correspondance $f: \{1,2,3\} \multimap \{a,b\}$ définie par $f\,1 = \{a,b\}$, $f\,2 = f\,3 = \{b\}$. Alors, f^{C} est définie sur {2,3}, valuée dans {a} et $f^{C}\,2 = f^{C}\,3 = \{a\}$.

---

[18] L'extension canonique d'une fonction $g : X \to 2^Y$ à $g' : X' \to 2^{Y'}$, avec $X \subset X'$ et $Y \subset Y'$, est donnée par : $g'x = g\,x$ si $x \in X$ et $g'x = \varnothing$ sinon.

[19] (Berge, 1959, p 9) : une famille $(A_i\,;\, i \in K)$ est une *partition* de A si (*i*) $\forall\, i \in K$, $A_i \neq \varnothing$, (*ii*) $\forall\, i \in K$, $\forall\, j \in K$, $i \neq j \Rightarrow A_i \cap A_j = \varnothing$ et (*iii*) A est égale à l'union des A_i's. Donc, $\forall\, i \in K$, $\forall\, j \in K$, $i \neq j \Rightarrow A_i \neq A_j$.

Enfin, soit $g : X \to 2^Y$ la fonction définie par $\forall x \in X$, $g\,x = f\,x - f\,(X - \{x\})$. Elle est obtenue en enlevant de $f\,x$ tous les y qui appartiennent à un $f\,x'$, $x \neq x'$. La correspondance **cor**(g) est appelée la *réduction* $f^{\mathbf{R}}$ de f. Elle est définie sur $X^{\mathbf{R}} \subset X$ et valuée sur $Y^{\mathbf{R}} \subset Y$. Plus simplement, $f^{\mathbf{R}} : X^{\mathbf{R}} \multimap Y^{\mathbf{R}}$.

Exemple – La réduction $f^{\mathbf{R}}$ de la correspondance de l'exemple précédent est $\{1\} \multimap \{a\}$, avec $f^{\mathbf{R}}\,1 = \{a\}$.

1.3 Correspondances et relations binaires

Les correspondances sont naturellement liées aux relations binaires sur un ensemble. Le lemme 5.A, ci-dessous, explicite ces liens. Une relation binaire sur un ensemble N est un sous ensemble du produit $N \times N$. La relation **I** est la relation diagonale $\{ (x,x) \mid x \in N \}$. Si r est une relation binaire sur N, r^{-1} désigne son inverse, i.e. $\{ (x,y) \mid (y,x) \in r \}$. Si r et s sont des relations binaires sur N, $r \circ s$ désigne leur composition, i.e. $\{(x,y) \mid \exists\, z \in N,\ (x,z) \in r$ et $(z,y) \in s\}$. Enfin, pour tout sous-ensemble E de N, r E est l'image de E par r, i.e. $\{y \in N \mid \exists\, x \in E \quad (x,y) \in r\}$.

Toute relation binaire r sur N définit une correspondance $\mathbf{cor}(r) : r^{-1}N \multimap r\,N$ telle que $\forall x \in r^{-1}N$, $\mathbf{cor}(r)\,(x) = \{ y \in N \mid (x,y) \in r \}$. Notons que, si r est l'ordre *strict* 123 sur $N = \{ 1,2,3\}$, alors **cor**(r) est $\{1,2\} \multimap \{2,3\}$ alors que, si r est l'ordre *large* 123 sur N, **cor**(r) est $N \multimap N$. Réciproquement, à toute correspondance $f : X \multimap Y$, on associe, de manière naturelle, une relation binaire **bin**(f) sur $X \cup Y$. Deux correspondances sont égales si et seulement si leurs **bin** sont égaux.

Lemme 5.A – Soit $f : X \multimap Y$ une correspondance et *diff* la relation différence sur $X \cup Y$, i.e[20] *diff* $= (X \cup Y) \times (X \cup Y) - \mathbf{I}$. Alors,

(5.2a) $\mathbf{bin}(f^{\sim}) = \mathbf{bin}(f)^{-1}$ □;

(5.2b) $\mathbf{bin}(f^{\mathbf{C}}) = X \times Y - \mathbf{bin}(f)$ □ ;

(5.2c) $\mathbf{bin}(f^{\mathbf{R}}) = \mathbf{bin}(f) - \mathit{diff} \circ \mathbf{bin}(f)$ □.

1.4 Propriétés utiles

Les lemmes suivants sont utiles pour les démonstrations des chapitres suivants.

Lemme 5.B – Soit $f : X \multimap Y$ une correspondance.
Alors, $\forall A \subset X$, $\forall B \subset X$, $f\,A - f\,B \subset f\,(A - B)$ □.

---

[20] Par exemple si $X = \{1\}$ et $Y = \{a\}$ alors la relation différence sur $X \cup Y$ est *diff* $= ((1,a) , (a,1))$.

Lemme 5.C – Soit $f : X \multimap\!\!\langle Y$ une correspondance :

(5.3a) $f^{\sim\sim} = f$;

(5.3b) f est une correspondance partition ssi son inverse $f^{\sim}$ est une correspondance singleton □ ;

(5.3c) Si f est une correspondance partition, alors $\forall E \subset X$, $f^{\sim} f E = E$ □.

preuve – (5.3a) Soit $x \in X$. $y \in f^{\sim\sim} x \Leftrightarrow y \in \{ y' \in Y \mid x \in f^{\sim} y' \} \Leftrightarrow x \in f^{\sim} y \Leftrightarrow x \in \{ x' \in X \mid y \in f x' \} \Leftrightarrow y \in f x$.

□

Lemme 5.D – Une correspondance partition définie et valuée sur le *même* ensemble X, i.e. telle que $f : X \multimap\!\!\langle X$, est une correspondance singleton et définit une permutation σ sur X telle que $\forall x \in X$, $f x = \{ \sigma x \}$ □.

Lemme 5.E – Soit $f : X \multimap\!\!\langle Y$ une correspondance :

(5.4a) si $f^{\mathbf{C}}$ est $X \multimap\!\!\langle Y$, alors $f^{\mathbf{CC}} = f$;

(5.4b) $f^{\sim \mathbf{C}} = f^{\mathbf{C} \sim}$;

(5.4c) $f^{\mathbf{R}}$ est une partition correspondance et $f^{\mathbf{R}} = f^{\mathbf{RR}}$ □.

preuve – (5.4a) Puisque $f^{\mathbf{C}}$ est $X \multimap\!\!\langle Y$, on a d'après (5.2b), $\mathbf{bin}(f^{\mathbf{CC}}) = X \times Y - \mathbf{bin}(f^{\mathbf{C}}) = X \times Y - (X \times Y - \mathbf{bin}(f)) = \mathbf{bin}(f)$. *(5.4b)* Puisque $f^{\sim}$ est $Y \multimap\!\!\langle X$, alors on a $\mathbf{bin}(f^{\sim \mathbf{C}}) = Y \times X - \mathbf{bin}(f^{\sim}) = (X \times Y - \mathbf{bin}(f))^{-1} = \mathbf{bin}(f^{\mathbf{C} \sim})$.

□

2. Fermeture associée à une correspondance

Ce paragraphe introduit une propriété des correspondances dont le § 4 montre comment elle conduit à une révélation par le haut du secret **S**(v). L'exposé est uniquement formel.

On commence par rappeler qu'une *fermeture* sur un ensemble A, ordonné par une relation d'ordre $\leq$, est une fonction $h : A \to A$, voir (Aigner, 1979, p 167), telle que :

(5.5a) $\forall x \in A, \forall y \in A, x \leq y \Rightarrow h x \leq h y$ (h est *isotone*) ;

(5.5b) $\forall x \in A, x \leq h x$ (h est *extensive*) ;

(5.5c) $\forall x \in A, h x = h h x$ (h est *idempotente*).

Si $f : X \multimap\!\!\langle Y$ est une correspondance, on note $f^{\mathbf{ker}}$ la fonction $2^X \to 2^Y$ définie par

(5.6a) $f^{\mathbf{ker}} \varnothing = Y$;

(5.6b) $\forall E \subset X,\ E \neq \varnothing,\ f^{\mathbf{ker}}\, E = \bigcap_{x \in E} f\, x.$

Une conséquence immédiate de la définition est que $f^{\mathbf{ker}}(A \cup B) = f^{\mathbf{ker}}\, A \cap f^{\mathbf{ker}}\, B$. Cette égalité est aussi vérifiée quand A ou B sont vides. De plus on a $A \subset B \Rightarrow f^{\mathbf{ker}}\, A \supset f^{\mathbf{ker}}\, B$ ($f^{\mathbf{ker}}$ est *antitone*). Le lemme suivant donnent une définition alternative de $f^{\mathbf{ker}}$.

Lemme 5.F – Si f est une correspondance $X \multimap Y$, alors

(5.7) $f^{\mathbf{ker}}\, E = \{\, y \in Y \mid f \sim y \supset E \,\}.$

preuve – $y \in f^{\mathbf{ker}}\, E \Leftrightarrow \forall x \in E,\ y \in f\, x$
$\Leftrightarrow \forall x \in E,\ x \in f \sim y$
$\Leftrightarrow f \sim y \supset E$
$\Leftrightarrow y \in \{\, y' \in Y \mid f \sim y' \supset E \,\}.$

□

La proposition suivante donne une expression utile de $f^{\mathbf{ker}}$.

Proposition 5.G – Si f est une correspondance $X \multimap Y$, alors

(5.8) $\forall E \subset \mathbf{def}\,(f^{\mathbf{C}}),\ f^{\mathbf{ker}}\, E = Y - f^{\mathbf{C}}\, E.$

preuve – Si $E \subset \mathbf{def}\,(f^{\mathbf{C}})$, alors pour tout x dans E, $f^{\mathbf{C}}$ est définie, donc $f\, x$ s'écrit $Y - f^{\mathbf{C}}\, x$. En conséquence, $f^{\mathbf{ker}}\, E = \bigcap_{x \in E} (Y - f^{\mathbf{C}}\, x) = Y - \bigcup_{x \in E} f^{\mathbf{C}}\, x = Y - f^{\mathbf{C}}\, E.$

□

L'opérateur « **ker** » s'applique naturellement à la correspondance inverse $f \sim$ d'une correspondance f. On considère alors la composée $f \sim^{\mathbf{ker}} f^{\mathbf{ker}}$ ($f \sim^{\mathbf{ker}}$ après $f^{\mathbf{ker}}$). Il s'agit d'une fonction de 2^X dans 2^X.

Théorème 5.H (Barbut, Monjardet, 1970, pp 16-20) – Soit $f : X \multimap Y$ une correspondance, la fonction $f \sim^{\mathbf{ker}} f^{\mathbf{ker}}$ est une fermeture sur l'ensemble 2^X des parties de X ordonné par l'inclusion.

preuve – *(isotone)* puisque $f^{\mathbf{ker}}$ et $f \sim^{\mathbf{ker}}$ sont antitones. *(extensive)* $x \in E \Rightarrow \forall y \in f^{\mathbf{ker}}\, E,\ y \in f\, x \Rightarrow \forall y \in f^{\mathbf{ker}}\, E,\ x \in f \sim y \Rightarrow x \in \bigcap_{y \in f^{\mathbf{ker}} E} f \sim y \Rightarrow$

$x \in f \sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E.$

(idempotente) D'après l'extensivité de $f \sim^{\mathbf{ker}} f^{\mathbf{ker}}$, on a

$f \sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E \ \subset\ f \sim^{\mathbf{ker}} f^{\mathbf{ker}} (f \sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E).$

De même, d'après celle de $f^{\mathbf{ker}}\, f\sim^{\mathbf{ker}}$, on a $f^{\mathbf{ker}}\, E \subset f^{\mathbf{ker}}\, f\sim^{\mathbf{ker}}(f^{\mathbf{ker}}\, E)$. Or, $f\sim^{\mathbf{ker}}$ est antitone, donc

$f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E \supset f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\, f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E$. D'où l'idempotence.

□

La propriété d'extensivité de $f\sim^{\mathbf{ker}} f^{\mathbf{ker}}$ conduit à une révélation par le haut du secret **S**(v) (voir § 4 ci-après). Elle s'écrit :

(5.9) $\forall E\subset X,\ E\subset f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E$.

Lorsque le sous ensemble E de X est un singleton, l'inclusion (5.9) peut être « améliorée ». Elle conduit à une meilleure – au sens de l'inclusion – révélation par le haut du secret **S**(v), uniquement dans le cas où on est assuré que l'extension de **S**(v) est un singleton (voir § 4 ci-après). On introduit pour cela l'opérateur « **Xker** ».

Si f est une correspondance $X \multimap\!\!\langle Y$ et E un sous ensemble de X, on définit $f^{\mathbf{Xker}}\, E = \{\, y\in Y \mid f\sim y = E\,\}$. Les opérateurs « **ker** » et « **Xker** » diffèrent par le « = » utilisé dans « **Xker** » en lieu et place du « ⊃ » de « **ker** » (voir lemme 5.G).

Proposition 5.I – Pour toute correspondance $f: X \multimap\!\!\langle Y$ et pour tout x de X on a la double inclusion:

(5.10) $\{x\} \subset f\sim^{\mathbf{Xker}} f^{\mathbf{ker}}\,\{x\} \subset f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\,\{x\}$.

preuve – $\forall x\in E,\ f\sim^{\mathbf{Xker}} f^{\mathbf{ker}}\,\{x\} = f\sim^{\mathbf{Xker}} f\, x = \{\, x'\in X \mid f\sim\ \sim x' = f\, x\,\} = \{\, x'\in X \mid f\, x' = f\, x\,\}$. Donc, $\{x\} \subset f\sim^{\mathbf{Xker}} f^{\mathbf{ker}}\,\{x\}$. Quant à l'inclusion $f\sim^{\mathbf{Xker}} f^{\mathbf{ker}}\,\{x\} \subset f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\,\{x\}$, elle est une conséquence du remplacement, dans « **ker** », de « ⊃ » par « = ».

□

3. Exemple

Soit la correspondance $f:\{1,2,3\} \multimap\!\!\langle \{a,b,c,d\}$ définie par

$f\,1 = \{a,b,d\},\ f\,2 = \{b\},\ f\,3 = \{b,c\}$.

La correspondance $f\sim:\{a,b,c,d\} \multimap\!\!\langle \{1,2,3\}$ est alors égale à

$f\sim a = \{1\},\ f\sim b = \{1,2,3\},\ f\sim c = \{3\},\ f\sim d = \{1\}$.

Selon la propriété (5.9), l'ensemble $f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\, E$ est un sur-ensemble de E. Ainsi, prenant $E = \{1,3\}$, on a :

$f\sim^{\mathbf{ker}} f^{\mathbf{ker}}\,\{1,3 \;= f\sim^{\mathbf{ker}}(f\,1 \cap f\,3)$

$\}$ $= f \sim^{\mathbf{ker}} \{b\}$

$= f \sim b = \{1,2,3\} \supset E.$

De même, selon la propriété (5.10), l'ensemble $f \sim^{\mathbf{Xker}} f^{\mathbf{ker}} E$ est un sur-ensemble de E, dès lors que E est un singleton. Par exemple, en prenant $E = \{2\}$, on a $f \sim^{\mathbf{Xker}} f^{\mathbf{ker}} \{2\} = \{2\}$ et $f \sim^{\mathbf{ker}} f^{\mathbf{ker}} \{2\} = f \sim^{\mathbf{ker}} f\,2 = f \sim^{\mathbf{ker}} \{b\} = \{1,2,3\}$. On fait remarquer que la condition « E est un singleton » est essentielle pour l'inclusion (5.10). Ainsi, $f \sim^{\mathbf{Xker}} f^{\mathbf{ker}} \{1,3\}$ vaut $\{2\}$ et n'est pas un sur-ensemble de $\{1,3\}$.

4. Révélation par le haut

4.1 Application de (5.9) et (5.10) aux relations

Soit une relation 2-aire B[X,Y]. On désigne par $B(X\rightarrow Y)$[21] la correspondance **cor**(B) définie sur $B(\rightarrow X)$, à valeurs dans $B(\rightarrow Y)$, i.e.

$\forall\ x \in B(\rightarrow X),\ \ B(X\rightarrow Y)\,(x) = \{\, y \in B(\rightarrow Y) \mid (x,y) \in B(\rightarrow (X,Y)) \,\}.$

L'inclusion (5.9), appliquée à cette correspondance, conduit à

(5.11) pour tout sous-ensemble E de $B(\rightarrow X)$, $E \subset B(X\rightarrow Y) \sim^{\mathbf{ker}} B(X\rightarrow Y)^{\mathbf{ker}} E$,

ou encore, compte tenu que $B(X\rightarrow Y) = B(Y\rightarrow X)\sim$, que

(5.12) pour tout sous-ensemble E de $B(\rightarrow X)$, $E \subset B(Y\rightarrow X)^{\mathbf{ker}} B(X\rightarrow Y)^{\mathbf{ker}} E$.

Considérons alors une requête **req** dont l'extension est, quel que soit l'état de la base, un sous-ensemble de $B(\rightarrow X)$. L'inclusion (5.12) s'applique alors et s'écrit

$\mathbf{req} \subset B(Y\rightarrow X)^{\mathbf{ker}} B(X\rightarrow Y)^{\mathbf{ker}} \mathbf{req}$.

Il en résulte que la fonction

(5.13) $B(Y\rightarrow X)^{\mathbf{ker}} B(X\rightarrow Y)^{\mathbf{ker}}$

est une révélation par le haut de toute requête **req** dont l'extension est un sous ensemble de $B(\rightarrow X)$. C'est le cas du secret **S**(v) dans la configuration considérée, voir chapitre 4, § 1. En effet, la contrainte $R(v,x,y) \rightarrow B(x,y)$ fait que l'extension du secret **S**(v), i.e. l'extension de la requête **`SELECT X FROM R WHERE V=v`**, est nécessairement un sous-ensemble de $B(\rightarrow X)$. Par conséquent, la fonction (5.13) est une révélation par le haut du secret **S**(v).

De même, si l'extension d'une requête **req** est, quel que soit l'état de la base, un sous-ensemble singleton de $B(\rightarrow X)$, l'inclusion (5.10) s'écrit :

---

[21] Le symbole $\rightarrow$ n'est *pas* lié aux dépendances fonctionnelles.

(5.14) $\mathbf{req} \subset B(Y\rightarrow X)\ \mathbf{Xker}\ B(X\rightarrow Y)\ \mathbf{ker}\ \mathbf{req} \subset B(Y\rightarrow X)\ \mathbf{ker}\ B(X\rightarrow Y)\ \mathbf{ker}\ \mathbf{req}$.

Il en résulte que la fonction

(5.15) $B(Y\rightarrow X)\ \mathbf{Xker}\ B(X\rightarrow Y)\ \mathbf{ker}$

est une révélation par le haut de toute requête singleton **req**, meilleure, au sens de l'inclusion, que la fonction (5.13). Pour les mêmes raisons que précédement, (5.15) constitue une révélation par le haut du secret **S**(v), si on est, par ailleurs, assuré que l'extension du secret est toujours un singleton.

4.2 Lien avec la division relationnelle

On montre enfin, que les révélations (5.13) et (5.15) sont relationnelles. On établit pour cela que les opérateurs « **ker** » et « **Xker** » sont liés à la division relationnelle. Soient B[X,Y] et T[X] des relations sur un univers **U**. On note :

$B\ \mathbf{DIV}_X\ T$ la relation $\{ y \in B(\rightarrow Y) \mid \forall x \in X,\ T(x) \Rightarrow B(x,y) \}$ et

$B\ {}_{=}\mathbf{DIV}_X\ T$ la relation $\{ y \in B(\rightarrow Y) \mid \forall x \in X,\ T(x) \Leftrightarrow B(x,y) \}$.

De la même manière, si E[Y] est une relation sur **U**, on note :

$B\ \mathbf{DIV}_Y\ E$ la relation $\{ x \in B(\rightarrow X) \mid \forall y \in Y,\ E(y) \Rightarrow B(x,y) \}$ et

$B\ {}_{=}\mathbf{DIV}_Y\ E$ la relation $\{ x \in B(\rightarrow X) \mid \forall y \in Y,\ E(y) \Leftrightarrow B(x,y) \}$.

L'opération **DIV** est la division relationnelle. On appelle *équi-division* l'opération ${}_{=}\mathbf{DIV}$. Elle s'exprime à partir des opérateurs relationnels par

$$B\ {}_{=}\mathbf{DIV}_X\ T = B\ \mathbf{DIV}_X\ T - [B - T\times B(\rightarrow Y)]\ (\rightarrow Y)$$

où $[B - T\times B(\rightarrow Y)]\ (\rightarrow Y)$ est l'ensemble des $y \in Y$ associés dans R à d'autres valeurs de X que celles de T.

Proposition 5.J – Pour tout sous-ensemble E de $B(\rightarrow X)$ et tout sous-ensemble F de $B(\rightarrow Y)$,

(5.16) $B(X\rightarrow Y)\ \mathbf{ker}\ E = B\ \mathbf{DIV}_X\ E$ (5.18) $B(Y\rightarrow X)\ \mathbf{ker}\ F = B\ \mathbf{DIV}_Y\ F$

(5.17) $B(X\rightarrow Y)\ \mathbf{Xker}\ E = B\ {}_{=}\mathbf{DIV}_X\ E$ (5.19) $B(Y\rightarrow X)\ \mathbf{Xker}\ F = B\ {}_{=}\mathbf{DIV}_Y\ F$

preuve - (5.14) $B(X\rightarrow Y)\ \mathbf{ker}\ E = \underset{x\in E}{I} B(X\rightarrow Y)(x)$

$= \{ y \in B(\rightarrow Y) \mid \forall x \in E,\ y \in B(X\rightarrow Y)(x) \}$

$= \{ y \in B(\rightarrow Y) \mid \forall x \in E,\ (x,y) \in B \} = B\ \mathbf{DIV}_X\ E.$

On procède de manière identique pour *(5.15)*, *(5.16)* et *(5.17)*.

□

Compte tenu de cette proposition, les révélations (5.13) et (5.15) se réécrivent respectivement B $\mathbf{DIV}_Y$ (B $\mathbf{DIV}_X$ **req**) et $=\mathbf{DIV}_Y$ (B $\mathbf{DIV}_X$ **req**). Ce sont donc des requêtes relationnelles et, par conséquent, des révélations relationnelles.

Pour simplifier, on ne considère dans la suite que la révélation (5.13) qui correspond au cas général.

Chapitre 6

Fraude du Secret S(v)

Le chapitre précédent a établi une révélation relationnelle du secret **S**(v). *Dans ce chapitre, on montre,* § 1*, qu'elle est une fraude du secret* **S**(v)*, i.e. qu'elle est calculable. Le chapitre se termine par une reprise,* § 2*, de l'exemple du chapitre 3 selon le formalisme introduit, puis par la description,* § 3*, de sa programmation sous Oracle.*

1. Canal caché[22] et fraude

Compte tenu des droits d'accès de la configuration considérée, voir chapitre 4, § 1, la révélation relationnelle (5.13) $B(Y \rightarrow X)$ **ker** $B(X \rightarrow Y)$ **ker** **S**(v) du secret **S**(v) est elle-même un secret puisque son évaluation réclame que soit évalué le secret **S**(v). On montre dans ce paragraphe qu'elle est cependant calculable, indirectement, dans le cadre de la configuration et qu'elle constitue, par conséquent, une fraude du secret **S**(v).

Soit la transaction **τ(y) ← UPDATE R SET Y=y WHERE V=v**. Elle porte sur les attributs V et Y de la relation R. Elle vise à modifier à y la valeur de l'attribut Y de *tous* les tuples de R pour lesquels R.V = v. Compte-tenu des droits d'accès, cette transaction peut être exécutée. En revanche, du fait de la contrainte 2-aire $R(v,x,y) \rightarrow B(x,y)$, les modifications visées ne seront acceptées par le vérificateur d'intégrité que si $\forall x \in R(V \rightarrow X)\ (v)$, $(x,y) \in B$.

[22] Voir chapitre 3, § 4.

Le canal caché ouvert par le vérificateur d'intégrité informe donc des valeurs acceptables de y. Pour une valeur donnée de v, il se formalise comme une fonction de Y dans {**vrai**,**faux**}, que l'on note $\mathbf{CANAL}_V$. Si la transaction $\tau(y)$ est acceptée, $\mathbf{CANAL}_V(y)$ a la valeur **vrai**, sinon il a la valeur **faux**.

Proposition 6.A– $\mathbf{CANAL}_V$ est la fonction caractéristique[23] de l'ensemble

(6.1) $B(X\rightarrow Y)\ \mathbf{ker}\ \mathbf{S}(v)$.

preuve – Soit $y \in Y$.

$$\begin{aligned}\mathbf{CANAL}_V(y) = \mathbf{vrai} &\Leftrightarrow \forall x \in R(V\rightarrow X)(v),\ (x,y) \in B\\ &\Leftrightarrow \forall x \in R(V\rightarrow X)(v),\ y \in B(X\rightarrow Y)(x)\\ &\Leftrightarrow y \in \bigcap_{x \in R(V\rightarrow X)(v)} B(X\rightarrow Y)(x)\\ &\Leftrightarrow y \in B(X\rightarrow Y)\ \mathbf{ker}\ (R(V\rightarrow X)(v)).\end{aligned}$$

□

Il résulte de cette proposition que l'extension de la requête (6.1) est l'ensemble des valeurs y de Y pour lesquelles la transaction $\tau(y)$ est acceptée. Dès lors, on peut la calculer en soumettant successivement au système la transaction $\tau(y)$ pour tous les $y \in Y$.

On souligne que, puisque par (5.16) la formule (6.1) est de la forme $B\ \mathbf{DIV}_X\ E$, le résultat est nécessairement inclus dans la projection $B(\rightarrow Y)$. Autrement dit, si y appartient à $Y - B(\rightarrow Y)$, alors la transaction $\tau(y)$ sera nécessairement refusée. Aussi, au lieu de soumettre $\tau(y)$ pour tous les $y \in Y$, le fraudeur pourra soumettre seulement, s'il les connaît, les y de $B(\rightarrow Y)$, afin de réduire l'espace de recherche.

Or, du fait des droits d'accès, (6.1) est la partie non calculable par le système de gestion de bases de données de la révélation par le haut (5.13) de **S**(v),. La proposition ci-dessus lève cet « obstacle » et la révélation (5.13) devient calculable. Par conséquent, dans la configuration considérée, la révélation (5.13) est une fraude par le haut du secret **S**(v).

En conclusion la formule (5.13) et la proposition 6.A, établissent formellement la fraude par le haut du secret **S**(v).

2. Un exemple

[23] Etant donné un ensemble E, la fonction caractéristique $f: E\rightarrow\{0,1\}$ d'un sous ensemble F de E est la fonction définie par : $\forall x \in E,\ x \in F \Leftrightarrow f\,x = 1$.

On reprend maintenant l'exemple du chapitre 3, § 1, que l'on traite avec le formalisme introduit. On rappelle que le secret est la requête

S(rafale) ← **SELECT mission FROM COMBINAISON WHERE avion = rafale**.

D'après (5.13)[24], la requête ci-dessous est une révélation par le haut de ce secret :

B(missile → mission) $\sim^{\mathbf{ker}}$ **B(mission → missile)** $^{\mathbf{ker}}$ **S**(rafale).

Compte tenu des droits d'accès, cette révélation est un secret. Un fraudeur peut cependant l'évaluer en calculant, via le canal caché, la sous requête **B(mission → missile)** $^{\mathbf{ker}}$ **S**(rafale). Il soumet pour cela au système la transaction

τ(m) ← **UPDATE COMBINAISON SET missile = m WHERE avion = «rafale»**

successivement pour toutes les valeurs m de l'attribut **missile** de la relation B. Seul m = apache est accepté.

Une révélation par le haut de **S**(rafale) est donc

B(mission → missile) $^{\mathbf{ker}}${apache}

$= \mathbf{B} =_{\mathbf{DIV}_{\mathbf{missile}}}$ {apache}

= {interception, bombardement}.

L'évaluation de cette révélation par le haut s'avère, dans ce cas particulier, être égale à celle de son secret.

3. Calcul de la fraude en SQL

On a indiqué au chapitre 4, § 3, que la configuration dans laquelle on se place est réalisable sur tout système de gestion de bases de données satisfaisant à la norme SQL-2 (SQL, 1992). La contrainte 2-aire de l'exemple précédent se déclare, lors de la création de la table **COMBINAISON**, par la commande SQL :

```
CREATE TABLE COMBINAISON
   ( avion  CHAR(15), mission  CHAR(15), missile  CHAR(15),
     CONSTRAINT 2aire CHECK
     (  (misson,missile) IN (
                ('Interception' , 'apache'),
                ('Interception' , 'mica'),
                ('Interception' , 'super530'),
                ('Bombardement' , 'apache'),
                ('reconnaissance' , 'magic'),
```

[24] On note que (5.15) n'est pas applicable car l'attribut `avion` n'étant pas une clé pour la relation `CONFIGURATION`, la requête `S(rafale)` peut renvoyer plus d'une mission.

```
            )
        )   );
```

Le calcul de la sous requête $\mathbf{B}(\mathbf{mission} \rightarrow \mathbf{missile})$ $^{\mathbf{ker}}$ $\mathbf{S}$(rafale) par le canal caché peut être automatisé sous Oracle, par un programme en PL/SQL dont la figure 4 ci-dessous donne le code. Le langage PL/SQL, propre au système Oracle, est un langage de manipulation de tables relationnelles dans lequel il est possible de programmer des boucles, d'effectuer des tests conditionnels et de définir des variables et des fonctions. L'accès aux données de la base est réalisé par des commandes SQL.

```
DECLARE
  avion CHAR(15);
  CONSTRAINT_VIOLATION EXCEPTION;
  PRAGMA EXCEPTION_INIT(CONSTRAINT_VIOLATION,-2290);

-- // Fonction de calcul de la transaction τ(m). Elle lève une exception en cas de
     violation de la contrainte 2-aire.
 FUNCTION TRANSACTION(un_avion CHAR(15), m CHAR(15)) RETURN BOOLEAN
  BEGIN
      UPDATE MISSION SET missile=m WHERE avion=un_avion;
      RETURN TRUE;
    EXCEPTION
      WHEN CONSTRAINT_VIOLATION THEN RETURN FALSE;
  END TRANSACTION;

-- // Début du programme
BEGIN
  avion='rafale';
  FOR m IN ['apache','mica','super530','magic']
    LOOP
      IF TRANSACTION(avion,m)=TRUE
         THEN INSERT INTO missile_possibles VALUES (m);
      END IF;
  END LOOP;

COMMIT;
```

Figure 4 : Calcul de (6.1) en PL/SQL

La partie en grisé dans la figure précédente contient la boucle **FOR ... END LOOP** qui exécute successivement la transactions $\tau(\mathbf{m})$ pour les valeurs **m** de $\mathbf{B}(\rightarrow \mathbf{missile})$, c'est à dire ici **apache, mica, super530** et **magic**. Elle renvoie **vrai** si la mise à jour à été acceptée par le vérificateur d'intégrité de la base. Dans ce cas, la commande **INSERT** stocke la valeur **m** dans la relation **missile_possibles**. A l'issue de

l'exécution du programme cette table contient l'extension de la requête : **B**(**mission** → **missile**) **ker** **S**(rafale).

Chapitre 7

Ensembles Exactement Révélés

On a montré précédemment qu'en présence de la contrainte $R(V,X,Y) \rightarrow B(X,Y)$ *la requête* $q_1 \leftarrow f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}$ `[select X from R where V=v]` *est une fraude du secret* $q_0 \leftarrow$ `[select X from R where V=v]`. *La question «* q_0 *et* q_1 *ont-elles la même extension ? » est pertinente pour un fraudeur, puisque, lorsque la réponse est positive, la fraude conduit à une connaissance exacte de l'extension du secret. La réponse dépend des extensions courantes des relations* R *et* B. *Elle est positive pour certaines extensions, négative pour d'autres. Cependant, quelles que soient les extensions de* q_0 *et* q_1*, elles sont toujours des sous-ensembles de* X. *Comme* $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}$ *s'applique à tout sous-ensemble de* X *et retourne un sous-ensemble de* X*, la question ci-dessus peut être reformulée sous la forme « étant donné un sous-ensemble* E *de* X, E *est-il égal à* $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}$ E *? » et ne dépend plus que de* B*, ou de sa correspondance associée* f.

Dans ce chapitre, on montre que les sous-ensembles E *pour lesquels la réponse est positive sont liés aux treillis de Galois,* § 1. *On étudie ensuite,* § 2*, leur géographie, à savoir leur répartition parmi tous les sous-ensembles de* X. *On montre, qu'ils sont compris entre deux frontières que l'on caractérise. Enfin, on introduit,* § 3*, la notion d'ensemble critique pour une correspondance.*

1. Le treillis ex(f)

Etant donnée une correspondance $f: X \multimap Y$, un sous-ensemble E de X est exactement révélé s'il est un fermé pour $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}$, i.e. si $E = f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, E$. On note **ex**(f) l'ensemble des sous-ensembles exactement révélés par f. Il n'est jamais vide puisque X est fermé.

Lemme 7.A – Si $f: X \multimap Y$ est une correspondance,

(7.1a) $\mathbf{ex}(f) = \{f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, E,\ E \subset X\}$; (7.1b) $\mathbf{ex}(f) = \{f^{\sim \mathbf{ker}}\, F,\ F \subset$

Y}.

preuve – (7.1a) Car $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}$ est idempotent. *(7.1b)* (Barbut, Montjardet, 1970, p 13) D'après (7.1a), les éléments de **ex**(f) sont de la forme $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, E$, $E \subset X$, donc de la forme $f^{\sim \mathbf{ker}}\, F$, $F \subset Y$. Inversement, soit $E = f^{\sim \mathbf{ker}}\, F$, $F \subset Y$. Alors $E \in \mathbf{ex}(f)$. En effet, $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, E$, ou encore $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, f^{\sim \mathbf{ker}}\, F$, est égal à $f^{\sim \mathbf{ker}}\, F$, donc à E, puisque, d'une part,

$$f^{\sim \mathbf{ker}}\, F \;\subset\; f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, f^{\sim \mathbf{ker}}\, F, \text{ car } f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}} \text{ est extensive}$$

et, d'autre part,

$$f^{\sim \mathbf{ker}}\, F \;\supset\; f^{\sim \mathbf{ker}}\, (\, f^{\mathbf{ker}}\, f^{\sim \mathbf{ker}}\, F\,) \text{ car } f^{\sim \mathbf{ker}} \text{ est antitone et } f^{\mathbf{ker}}\, f^{\sim \mathbf{ker}} \text{ extensive.}$$

□

On désigne, à partir de maintenant, la fonction $f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}$ par $\mathbf{rev}_f$, ou plus simplement **rev** lorsqu'il n'y a pas d'ambiguïté.

Exemple – Soit, $f : \{1,2,3\} \longrightarrow \langle \{a,b,c\}$ définie par $f\,1 = \{a,b,c\}$, $f\,2 = \{b\}$ et $f\,3 = \{c\}$. La figure 5 donne **rev** E pour tous les sous-ensembles E de {1,2,3}. Les sous-ensembles exactement révélés sont {1}, {1,2}, {1,3} et {1,2,3}. Donc **ex**(f) = { {1},{1,2},{1,3}, {1,2,3} }.

Sous-ensembles E de X	**rev** E	$f^{\mathbf{ker}}\, E$
∅	{1}	{a,b,c}
{1}	{1}	{a,b,c}
{2}	{1,2}	{b}
{1,2}	{1,2}	{b}
{3}	{1,3}	{c}
{1,3}	{1,3}	{c}
{2,3}	{1,2,3}	∅
{1,2,3}	{1,2,3}	∅

Figure 5 : Les fonctions **rev** et $\mathbf{f^{ker}}$ pour {1,2,3}

On montre maintenant comment l'ensemble **ex**(f) est lié au treillis de Galois de f. Soit une correspondance $f : X \longrightarrow \langle Y$. On considère l'ensemble **gal**(f) défini par

$$(7.2) \quad \mathbf{gal}(f) = \{\, (f^{\sim \mathbf{ker}}\, f^{\mathbf{ker}}\, E,\; f^{\mathbf{ker}}\, E),\; E \subset X \,\}.$$

Chaque élément de **gal**(f) est un couple (A,B), $A \subset X$, $B \subset Y$. La relation $\leq$ sur **gal**(f), définie par $(E_1,F_1) \leq (E_2,F_2)$ ssi $E_1 \subset E_2$ et $F_1 \supset F_2$, est un ordre sur **gal**(f) qui lui confère une structure de treillis. Le treillis **gal**(f) est le *treillis* de *Galois* de f. Chaque élément de **gal**(f) est appelé *rectangle maximal* de f. On remarque que l'ensemble **ex**(f)

est égal à la projection de **gal**(f) sur sa première composante.

Pour la correspondance f de l'exemple précédent, on a **gal**(f) = { ({1},{a,b,c}), ({1,2},{b}), ({1,3},{c}), ({1,2,3},∅) }, voir les deux colonnes de droite de la figure 5. Son treillis de Galois est représenté figure 6.

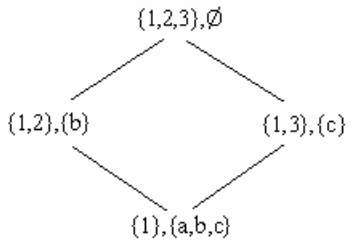

Figure 6 : Treillis de Galois associé à la correspondance f

En tant qu'ensemble de couples, **gal**(f) est bidimensionnel (chaque couple a deux composantes). Il est utilisé, en tant que tel, dans de nombreuses applications comme les analyses hiérarchiques, la sociométrie, la modélisation de l'information. Des références sur ses applications figurent dans (Guénoche, 1990) et (Yahia, Lakhal, Ciccheti, 1996). Dans notre cas, on s'intéresse à l'ensemble **ex**(f), donc uniquement à la première « dimension » du treillis **gal**(f).

Le théorème suivant montre que l'ensemble **ex**(f) ordonné par l'inclusion est aussi un treillis, les symboles Λ et V désignant respectivement l'*infimum* et le *supremum*.

Théorème 7.B (Aigner, 1979, p 167) – Si f est une correspondance X—⟨Y, l'ensemble ordonné **ex**(f), ordonné par $\subset$, est un treillis. De plus, pour tout $E_1 \in$ **ex**(f), $E_2 \in$ **ex**(f),

(7.3a) $E_1 \Lambda E_2 = E_1 \cap E_2$;

(7.3b) $E_1 \mathrm{V} E_2 = f^{\sim \mathbf{ker}\, f^{\mathbf{ker}}}(E_1 \cup E_2)$. □

Il résulte de ce théorème que l'intersection de deux ensembles exactement révélés est exactement révélée, mais pas leur union.

2. Le sous-ensemble min-ex(f)

On appelle *min-ex* de f : X—⟨Y, noté **min-ex**(f), le plus petit sous-ensemble de X exactement révélé par f. C'est donc aussi le plus petit élément de **ex**(f). Par le théorème 7.B, le **min-ex**(f) est l'intersection de tous les éléments de **ex**(f). Pour la correspondance de la figure 5, **min-ex**(f) = {1}.

Corollaire – Un sous-ensemble de X qui n'est pas un sur-ensemble de **min-ex**(f) ne peut pas être exactement révélé par une correspondance $f : X \multimap Y$. □

Ainsi, dans le treillis 2^X, seul le sous-treillis $\{E \mid \textbf{min-ex}(f) \subset E \subset X\}$ est concerné par les révélations exactes. Ceci est illustré par la figure 7 : le grand polygone est le treillis 2^X, celui de taille intermédiaire est le sous-treillis au-dessus de **min-ex**(f) et celui en pointillé représente le treillis **ex**(f). La fonction **rev** projette le treillis 2^X dans le treillis **ex**(f).

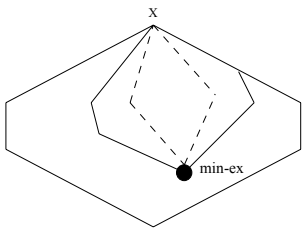

Figure 7 : Treillis 2^X et **ex**(f)

La proposition ci-dessous explicite l'extension de **ex**(f). Un élément x de X est *couvrant* pour la correspondance $f : X \multimap Y$ si $f\,x = Y$. On note **couv**(f) l'ensemble des éléments couvrants pour f.

Proposition 7.C – Pour toute correspondance $f : X \multimap Y$, **min-ex**(f) = **couv**(f).

preuve – (⊂) D'après la définition de **couv**(f), $f\,\textbf{ker}\,\textbf{couv}(f) = Y$, donc $\textbf{rev}\,\textbf{couv}(f) = f \sim \textbf{ker}\, Y$. D'après (5.7), $f \sim \textbf{ker}\, Y = \{ x \in X \mid f\,x \supset Y \}$. Puisque $f\,x \subset Y$, $f \sim \textbf{ker}\, Y = \{ x \in X \mid f\,x = Y \} = \textbf{couv}(f)$. Donc, $\textbf{rev}\,\textbf{couv}(f) = \textbf{couv}(f)$, ou encore $\textbf{couv}(f) \in \textbf{ex}(f)$. Il en résulte que $\textbf{min-ex}(f) \subset \textbf{couv}(f)$.

(⊃) D'après (5.7), $\forall F \subset Y$, $f\,\textbf{ker}\,F = \{ x \in X \mid f\,x \supset F \}$. Donc, $f \sim \textbf{ker}\, F \supset \textbf{couv}(f)$. Appliquée à $F = f\,\textbf{ker}\,\textbf{min-ex}(f)$, la dernière inclusion conduit à $\textbf{rev}\,\textbf{min-ex}(f) \supset \textbf{couv}(f)$, donc à $\textbf{min-ex}(f) \supset \textbf{couv}(f)$, puisque **min-ex** est exactement révélé.

□

Selon cette proposition, si $E \in \textbf{ex}(f)$, alors E contient tous les éléments couvrants pour f. Dans la correspondance f de la figure 5, le seul élément couvrant est 1. D'après la proposition 7.C, $\textbf{min-ex}(f) = \{1\}$. De plus, d'après le corollaire de 7.B, aucun sous-ensemble de {1} - seulement ∅ est candidat - ne peut être exactement révélé, ce qui est le cas, puisque $\textbf{rev}\,\varnothing = \{1\}$.

Proposition 7.D – Si $f : X \multimap Y$ est une correspondance, alors $\forall E \subset X$, $\forall C \subset$ **min-ex**(f), **rev** $E =$ **rev** $(E \cup C)$.

preuve – **rev** $(E \cup C) = f^{\sim\,\mathbf{ker}}(f^{\mathbf{ker}}E \cap f^{\mathbf{ker}}C)$. Si $C \subset$ **min-ex**(f), alors $f^{\mathbf{ker}}C = Y$ puisque **min-ex**(f) = **couv**(f). Donc, $f^{\mathbf{ker}}E \cap f^{\mathbf{ker}}C = f^{\mathbf{ker}}E$.

□

Corollaire – Soit $f : X \multimap Y$ une correspondance,

(7.4a) $\forall C \subset$ **min-ex**(f), **rev** $C =$ **min-ex**(f) ;

(7.4b) **min-ex**(f) = **rev** $\varnothing$ □.

preuve – *(7.4a)* En posant $E =$ **min-ex**(f) dans 6.D.

□

La proposition 7.D ci-dessus établit que la révélation de tout sous-ensemble E de X reste la même si E est augmenté des éléments couvrants de la correspondance. D'après le corollaire (7.4b) et puisque **rev** est isotone[25], la révélation de tout sous-ensemble E de X contient tous les éléments couvrants de f. Ils sont à l'évidence parasites dans la révélation de E quand E n'en contient initialement aucun.

3. Les ensembles critiques

En « descendant » le treillis 2^X, on s'est arrêté à **min-ex**(f). En le « remontant » on s'arrête aux sous-ensembles critiques. Un sous-ensemble C de X est *critique* pour une correspondance $f : X \multimap Y$, si $f^{\mathbf{ker}}C \neq f^{\mathbf{ker}}X$ et si aucun sur-ensemble de C ne possède cette propriété. On désigne par **critique**(f) l'ensemble des sous-ensembles de X critiques pour f. Par définition, $X \notin$ **critique**(f). On note que **critique**(f) peut être vide. Les ensembles critiques sont utilisés dans le chapitre suivant.

Proposition 7.E – Soit $f : X \multimap Y$ une correspondance, alors **critique**(f) $\subset$ **ex**(f).

preuve – Soit $C \in$ **critique**(f). On montre $C \in$ **ex**(f). Pour cela, on établit que $C =$ **r**ev C, plus simplement, puisque **rev** est extensive, que **rev** $C \subset C$. D'après (5.7), **rev** $C = \{ x' \in X \mid f x' \supset f^{\mathbf{ker}}C \}$. Si $x \in$ **rev** C, alors $f x \supset f^{\mathbf{ker}}C$ et $f x \cap f^{\mathbf{ker}}C = f^{\mathbf{ker}}C$. Autrement dit, $f^{\mathbf{ker}}(C \cup \{x\}) = f^{\mathbf{ker}}C$. Il en résulte que $x \in C$, sinon C ne serait pas critique.

□

[25] Voir chapitre 5, § 2, théorème 5.H.

D'après cette proposition, les sous-ensembles critiques pour f sont exactement révélés par f.

Corollaire – Soit $f : X \multimap Y$ une correspondance.

(*i*) Si $C \in$ **critique**(f), alors tout sur-ensemble E de C, non égal à C et à X, ne peut pas être exactement révélé et, de plus, **re**v $E = X$.

(*ii*) Soit $E \subset X$ et $\neq X$. Si E ne contient aucun sous-ensemble critique pour f, alors **rev** $E = X$ □.

preuve – (*i*) - Si $E \supset C$ et $E \neq C$, alors $f^{\mathbf{ker}} E = f^{\mathbf{ker}} X$ et **rev** $E =$ **rev** $X = X$.

□

Selon la proposition 7.E et son corollaire, **critique**(f) est l'ensemble des sous-ensembles maximaux de X qui sont exactement révélés par f. Cependant, tous les sous-ensembles de X ne sont pas nécessairement inclus dans un sous-ensemble critique. D'après l'énoncé (*ii*) du corollaire, leur **rev** vaut X.

Exemple – Soit la correspondance $g : \{1,2,3\} \multimap \{a,b,c\}$ avec $g(1) = \{a,c\}$, $g(2) = \{b,c\}$ et $g(3) = \{c\}$. On a $g^{\mathbf{ker}} \{1,2,3\} = \{c\}$ Donc, **critique**$(g) = \{ \{1\},\{2\} \}$. On remarque que le sous-ensemble {3} n'est inclus dans aucun des sous-ensembles critiques pour g.

Lemme 7.F – $\forall f : X \multimap Y$, $\varnothing \notin$ **critique**(f).

preuve – Si $\varnothing \in$ **critique**(f), alors $\forall x \in X$, {x} n'est pas critique, i.e. $f\,x = f^{\mathbf{ker}} X$. Donc, $Y = f^{\mathbf{ker}} X$. Or, d'après (5.7a), $f^{\mathbf{ker}} \varnothing = f^{\mathbf{ker}} X$. D'où, $\varnothing \notin$ **critique**(f).

□

Chapitre 8

Les Correspondances Manichéennes

Comme on l'a vu précédemment, pour certains sous-ensembles E *de* X $E = f^{\sim}\,\mathbf{ker}\,f\,\mathbf{ker}\,E$*, ce qui correspond au cas où le fraudeur obtient une connaissance exacte de* E*. Le sous-ensemble* E *étant inconnu du fraudeur par hypothèse, celui-ci ne peut savoir a priori quand il est dans une telle situation. Dans ce chapitre on caractérise,* § 5*, une classe de correspondances pour laquelle le fraudeur peut, en général, savoir si la connaissance de* E *qu'il obtient est exacte. Ces correspondances sont dites correspondances manichéennes. Au préalable, on caractérise, comme un cas particulier, les correspondances dites pires,* § 3*, et exactes,* § 4*. Le chapitre se clôt,* § 6*, sur un exemple.*

1. Efficace des correspondances

Du point de vue d'un fraudeur, les meilleures correspondances $f : X \multimap Y$ sont celles telles que $\forall\, E \subset X$, **rev** $E = E$, autrement dit, il n'y a jamais de bruit. Elles sont dites *exactes* (voir chapitre 3, § 1.2).

A leurs antipodes, une correspondance est dite *pire* lorsque $\forall\, E \subset X$, **rev** $E = X$. La seule information que l'on obtient dans ce cas est que les valeurs secrètes appartiennent au domaine actif, ce qui ne constitue pas une information très intéressante.

Entre ces deux situations extrêmes, on considère les correspondances pour lesquelles $\forall\, E \subset X$, **rev** E est soit E soit X. Autrement dit, soit elles révèlent exactement, soit elles révèlent au pire. On les appelle correspondances *manichéennes*. Les correspondances exactes et les correspondances pires en sont des cas particuliers.

Lorsqu'un fraudeur obtient un sur-ensemble F d'un ensemble de données E qui lui est inaccessible, il lui est essentiel de savoir si, par chance, $E = F$, i.e. si F ne contient pas de bruit. L'intérêt pratique des correspondances manichéennes est qu'elles permettent, dans certains cas, de répondre à la question : le bruit est-il vide?

En effet, si la correspondance $X \multimap Y$ est manichéenne, la réponse est « oui » si **rev** E est différent de X, puisque dans ce cas, par définition, $\mathbf{rev}\, E = E$. En revanche, lorsque $\mathbf{rev}\, E = X$, il n'est pas possible de répondre à la question puisque ce cas peut survenir aussi bien lorsque $E = X$ (auquel cas, la réponse serait « oui ») que lorsque $E \neq X$ (auquel cas la réponse serait « non »).

Dans le § 2 de ce chapitre on examine le rôle central de **critique**(f) pour les correspondances manichéennes. La proposition 8.G du § 5, caractérise les correspondances manichéennes. Auparavant, on caractérise, comme résultat préliminaire, les correspondances exactes et pires.

2. Le rôle de critique(f)

La proposition suivante lie la notion d'ensemble critique à celle de correspondance manichéenne.

Proposition 8.A – Si $f : X \multimap Y$ est une correspondance manichéenne, alors

(8.1) $\mathbf{ex}(f) = \{ E \subset X \mid \exists\, C \in \mathbf{critique}(f) : E \subset C \}$.

preuve – (⇒) Soient $C \in \mathbf{critique}(f)$ et $E \subset C$. On a $\mathbf{rev}\, E \subset \mathbf{rev}\, C$. Donc, d'après 7.E, $\mathbf{rev}\, E \subset C$. De plus, $C \neq X$ et, f étant manichéenne, **rev** E est soit égale à E, soit égale à X. Donc on a nécessairement $\mathbf{rev}\, E = E$ ou $E \in \mathbf{ex}(f)$. *(⇐)* L'inclusion réciproque est immédiate.

□

Selon cette proposition, si f est une correspondance manichéenne, alors l'ensemble **ex**(f) des sous-ensembles de l'ensemble de définition **def**(f) de f exactement révélés est entièrement déterminé par l'ensemble **critique**(f). Il est en effet formé de tous les sous-ensembles de **def**(f) qui sont inclus dans au moins un sous-ensemble critique de f.

Corollaire 1 – Si $f : X \multimap Y$ est une correspondance,

(8.2a) f pire $\Leftrightarrow \mathbf{critique}(f) = \varnothing$ □;

(8.2b) f exacte $\Leftrightarrow \mathbf{critique}(f) = \{C \subset X$ tels que $| C | = | X | - 1\}$ □;
($| C |$ et $| X |$ désignent, respectivement, les cardinaux de C et de X)

Corollaire 2 – Si $f : X \multimap Y$ est une correspondance,

(8.3a) si f est manichéenne et non pire alors **min-ex**(f) = ∅ ;

(8.3b) $| X | = 1 \Rightarrow$ f est pire.

preuve – (8.3a) Si f n'est pas pire, **critique**(f) n'est pas vide, d'après (8.2a). D'après 7.F, elle ne contient pas ∅, et donc contient $C \neq \emptyset$. D'après (8.1), $\emptyset \in$ **ex**(f), d'où, **min-ex**(f) = ∅. *(8.3b)* Puisque $| X | = 1$, le seul sous-ensemble critique possible est ∅ qui par ailleurs ne peut être critique (7.F). Donc, **critique**(f) = ∅.

□

3. Correspondances pires

Proposition 8.B – La correspondance produit [X×Y] est l'unique correspondance pire $X \multimap Y$.

preuve – $f : X \multimap Y$ est pire $\Leftrightarrow X =$ **critique**(f) $= \emptyset \Leftrightarrow \forall x \in X$, f x = Y
$\Leftrightarrow f = [X \times Y]$.

□

En termes relationnels, cette proposition signifie que la contrainte 2-aire $R(v,x,y) \rightarrow B(x,y)$, avec B = [X×Y], n'exprime aucun lien entre X et Y. D'après la proposition 8.B, les révélations qu'elle permet sont toutes pires. Le résultat rejoint l'intuition « pas de contrainte, pas de révélation ».

4. Correspondances exactes

Dans ce paragraphe, la proposition 8.D caractérise les correspondances exactes. Une condition suffisante est d'abord présentée dans le lemme 8.C. Puis, la proposition 8.E examine le cas particulier où la correspondance est définie et valuée sur le même ensemble.

Lemme 8.C – Une correspondance $f : X \multimap Y$ est exacte si son complément $f^{\mathbf{C}}$ est une correspondance partition définie sur X et valuée sur Y.

preuve – Soit $E \subset X$. Comme $f^{\mathbf{C}}$ est $X \multimap Y$, $f^{\mathbf{C}} \sim$ est $Y \multimap X$. On a alors
$f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E = X - f^{\mathbf{C}} \sim (Y - f^{\mathbf{C}} E)$, d'après 5.G
$\Rightarrow f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E \subset X - (f^{\mathbf{C}} \sim Y - f^{\mathbf{C}} \sim f^{\mathbf{C}} E)$, d'après 5.B,
$\Rightarrow f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E \subset X - (X - E)$, d'après (5.3c) pour $f^{\mathbf{C}}$,
$\Rightarrow f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E \subset E \Rightarrow f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E = E$, d'après (5.9).

□

La condition posée par ce lemme n'est pas nécessaire pour qu'une correspondance soit exacte. Considérons, par exemple, la correspondance g : {i,ii,iii}—⟨ {1,2,3,4,5} définie par g i = {3,4,5}, g ii = {1,5} et g iii = {1,2,3,4}. Le calcul montre que tout sous-ensemble de l'ensemble de définition **def**(g) de g est exactement révélé. Par exemple,

$\mathbf{rev}_g\{i,iii\} = g \sim \mathbf{ker}\,(g\,i \cap g\,iii) = g \sim \mathbf{ker}\,\{3,4\} = g \sim 3 \cap g \sim 4 = \{i,iii\}$.

La correspondance g est donc exacte. Or, son complémentaire $g^{\mathbf{C}}$, représenté figure 8, n'est pas une correspondance partition et, ce faisant, ne vérifie pas la condition du lemme. En effet, l'intersection $g^{\mathbf{C}}\,i \cap g^{\mathbf{C}}\,ii$ n'est pas vide puisqu'elle contient l'élément 2.

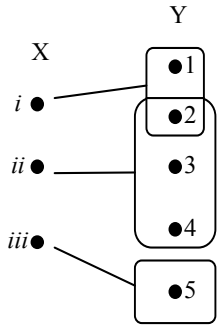

Figure 8 : La correspondance $g^{\mathbf{C}}$

La proposition suivante étend le lemme par une condition nécessaire et suffisante.

Proposition 8.D – Soit f : X—⟨ Y une correspondance. Les propositions suivantes sont équivalentes :

(8.4a) f est une correspondance exacte ;

(8.4b) $\forall\, E \subset X,\ \forall\, F \subset X,\ E \neq F \Rightarrow f^{\mathbf{ker}}\,E \neq f^{\mathbf{ker}}\,F$;

(8.4c) la réduction $f^{\mathbf{C\,R}}$ du complément $f^{\mathbf{C}}$ de f est définie sur X ($\mathbf{def}\,(f^{\mathbf{C\,R}}) = X$).

preuve – La preuve est circulaire.

((8.4a) ⇒ (8.4b)) $\forall\, E \subset X,\ \forall\, F \subset X,\ E \neq F \ \Rightarrow\ \mathbf{rev}\,E \neq \mathbf{rev}\,F \ \Rightarrow\ f^{\mathbf{ker}}\,E \neq f^{\mathbf{ker}}\,F$.

((8.4b) ⇒ (8.4c)) On démontre tout d'abord que $f^{\mathbf{C}}$ est définie sur X, puis que $f^{\mathbf{C\,R}}$ est définie sur X.

Si $f^{\mathbf{C}}$ n'est pas définie sur X, alors il y a un x dans X tel que $Y - f\,x = \varnothing$ ou, sinon, $f\,x = Y$. Alors, $\forall\, E \subset X$, $f^{\mathbf{ker}}\,(E \cup \{x\}) = f^{\mathbf{ker}}\,(E - \{x\}) \cap f\,x = f^{\mathbf{ker}}\,(E - \{x\})$. Donc, $E \cup \{x\}$ et $E - \{x\}$ ont le même $f^{\mathbf{ker}}$, ce qui contredit (8.4b).

Supposons maintenant que $f^{\mathbf{C\,R}}$ n'est pas définie sur X. Il existe un x dans X tel que $f^{\mathbf{C}}\,x - f^{\mathbf{C}}\,(X - \{x\}) = \varnothing$ ou, sinon, $f\,x \supset f\,(X - \{x\})$, ce qui induit $f\,x = f\,X$. Alors, $f\,x = Y$, ce qui contredit $f^{\mathbf{C}}$ définie sur X.

((8.4c) ⇒ (8.4a)) Soit $E \subset X$. Pour établir $\mathbf{rev}\, E = E$, on prouve d'abord que la correspondance $f^{\mathbf{C\,R\,C}}$ est définie sur X et qu'elle est exacte. On prouve alors que

(1) $\quad f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E \subset f^{\mathbf{C\,R\,C}} \sim^{\mathbf{ker}} f^{\mathbf{C\,R\,C\,ker}} E.$

Finalement, puisque $f^{\mathbf{C\,R\,C}}$ est exacte, l'inclusion **(1)** conduit à $f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E \subset E$ et d'après 5.9, à $\mathbf{rev}\, E = E$.

Soit $Y^{\mathbf{C\,R}}$ l'ensemble des valeurs de $f^{\mathbf{C\,R}}$. D'après (8.4c), $f^{\mathbf{C\,R}}$ est $X \leftarrow\!\!\langle\, Y^{\mathbf{C\,R}}$. Comme toute réduction, la correspondance $f^{\mathbf{C\,R}}$ est une correspondance partition et donc son complément $f^{\mathbf{C\,R\,C}}$ est aussi une correspondance $X \leftarrow\!\!\langle\, Y^{\mathbf{C\,R}}$. Donc, d'après (5.4a), $f^{\mathbf{C\,R}} = f^{\mathbf{C\,R\,C\,C}}$. En d'autres termes, $f^{\mathbf{C\,R}}$ est le complément de $f^{\mathbf{C\,R\,C}}$. Donc d'après le lemme 8.C, $f^{\mathbf{C\,R\,C}}$ est exacte.

On prouve maintenant **(1)**. Comme $f^{\mathbf{C\,R}}$ est définie sur X, $f^{\mathbf{C}}$ l'est aussi définie sur X. Alors, d'après 5.G, $f^{\mathbf{ker}} E = Y - f^{\mathbf{C}} E$, ce qui induit

(2) $\quad f^{\mathbf{ker}} E \supset Y^{\mathbf{C\,R}} - f^{\mathbf{C}} E$, puisque $Y^{\mathbf{C\,R}} \subset Y$.

Clairement, $Y^{\mathbf{C\,R}} - f^{\mathbf{C}} E = Y^{\mathbf{C\,R}} - f^{\mathbf{C\,R}} E$ et, puisque $f^{\mathbf{C\,R}}$ est le complément de $f^{\mathbf{C\,R\,C}}$, $Y^{\mathbf{C\,R}} - f^{\mathbf{C}} E = Y^{\mathbf{C\,R}} - (f^{\mathbf{C\,R\,C}})^{\mathbf{C}} E$.

Rappelons que $f^{\mathbf{C\,R\,C}}$ est une correspondance $X \leftarrow\!\!\langle\, Y^{\mathbf{C\,R}}$. Alors, d'après 5.G, $Y^{\mathbf{C\,R}} - (f^{\mathbf{C\,R\,C}})^{\mathbf{C}} E = f^{\mathbf{C\,R\,C\,ker}} E$.

Finalement, on déduit de **(2)** que $f^{\mathbf{ker}} E \supset f^{\mathbf{C\,R\,C\,ker}} E$ et

(3) $\quad f \sim^{\mathbf{ker}} f^{\mathbf{ker}} E \subset f \sim^{\mathbf{ker}} f^{\mathbf{C\,R\,C\,ker}} E.$

L'expression du côté droit du signe « $\subset$ » est de la forme $f \sim^{\mathbf{ker}} A$ avec $A = f^{\mathbf{C\,R\,C\,ker}} E$. L'ensemble A est un sous-ensemble de $Y^{\mathbf{C\,R}}$. Alors, **(3)** prouve **(1)** si $\forall\, A \subset Y^{\mathbf{C\,R}}$, $f \sim^{\mathbf{ker}} A \subset f^{\mathbf{C\,R\,C}} \sim^{\mathbf{ker}} A$. Ceci est établi ci-après par une séquence de réécritures :

$x \in f \sim^{\mathbf{ker}} A$
$\Leftrightarrow\ x \in \{ z \in X \mid f z \supset A \}$, d'après (5.7),
$\Leftrightarrow\ x \in \{ z \in X \mid f z \cap Y^{\mathbf{C\,R}} \supset A \}$, puisque $A \subset Y^{\mathbf{C\,R}}$,
$\Leftrightarrow\ x \in \{ z \in X \mid Y^{\mathbf{C\,R}} - (Y^{\mathbf{C\,R}} - f z) \supset A \}$
$\Leftrightarrow\ x \in \{ z \in X \mid Y^{\mathbf{C\,R}} - (Y^{\mathbf{C}} - f z) \supset A \}$
$\Leftrightarrow\ x \in \{ z \in X \mid Y^{\mathbf{C\,R}} - f^{\mathbf{C}} z \supset A \}$
$\Leftrightarrow\ x \in \{ z \in X \mid Y^{\mathbf{C\,R}} - f^{\mathbf{C\,R}} z \supset A \}$
$\Leftrightarrow\ x \in \{ z \in X \mid f^{\mathbf{C\,R\,C}} z \supset A \}$
$\Leftrightarrow\ x \in f^{\mathbf{C\,R\,C}} \sim^{\mathbf{ker}} A$, d'après (5.7).

□

Pour le complément $g^{\mathbf{C}}$, représenté à la figure 8, de la correspondance g, $g^{\mathbf{C\,R}}$ est définie sur $\{i,ii,iii\}$ puisque $g^{\mathbf{C\,R}}\ i = \{1\}$, $g^{\mathbf{C\,R}}\ ii = \{3,4\}$ et $g^{\mathbf{C\,R}}\ iii = \{5\}$. Par conséquent, selon la proposition 8.D, la correspondance g est exacte.

Pour terminer ce paragraphe, on donne une caractérisation des correspondances exactes quand elles sont définies et valuées sur le même ensemble.

Proposition 8.E – Une correspondance $f : X \multimap\!\!\langle X$ est exacte ssi il existe une permutation σ de X telle que $\forall\, x \in X$, $f\,x = X - \{\sigma\, x\}$.

preuve – $f : X \multimap\!\!\langle X$ est exacte $\Leftrightarrow$ $f^{\mathbf{C\,R}}$ est définie sur X. Comme $f^{\mathbf{C\,R}}$ est une correspondance partition, $|\,X\,| \leq |\,f^{\mathbf{C\,R}}\,X\,|$. f est $X \multimap\!\!\langle X$, d'où, $f^{\mathbf{C\,R}}\,X = X$. Donc, d'après 5.D, $f^{\mathbf{C\,R}}$ définit une permutation σ sur X et $\forall\, x \in X$, $f^{\mathbf{C\,R}}\,x = \{\sigma\, x\}$. Or, $f^{\mathbf{C\,R}} = f^{\mathbf{C}}$. En effet, la définition de la réduction implique que :

$$f^{\mathbf{C\,R}}\,X = f^{\mathbf{C}}\,X \;-\; \bigcup_{x \in X} [f^{\mathbf{C}}\,x \cap f^{\mathbf{C}}\,(X - \{x\})].$$

Or, $f^{\mathbf{C\,R}}\,X = X$, $\Rightarrow$ $f^{\mathbf{C}}\,X = X$, $\Rightarrow$ l'union ci-dessus est vide, $\Rightarrow$ $\forall\, x \in X$, $f^{\mathbf{C}}\,x \cap f^{\mathbf{C}}\,(X - \{x\}) = \varnothing$. Alors, $f^{\mathbf{C\,R}}\,x = f^{\mathbf{C}}\,x$, $\Rightarrow f^{\mathbf{C}}\,(x) = \{\sigma\, x\}$ et $f\,x = X - \{\sigma\, x\}$.

□

Lorsque la permutation σ est l'identité sur X, la correspondance exacte associée $X \multimap\!\!\langle X$ est telle que $\forall\, x \in X$, $f\,x = X - \{x\}$ ou, en d'autres termes, f est la relation différence *diff*, ou $\neq$, sur X. Pour tout sous-ensemble E de X, $f^{\mathbf{ker}}\,E = X - E$ et $\mathbf{rev}\,E = X - (X - E) = E$. Cela correspond à l'intuition. En effet, considérons la contrainte 2-aire $R(v, x, y) \rightarrow \mathit{diff}\,(x, y)$. Elle impose que les deux dernières composantes de tout tuple de R appartiennent à X et sont différentes. Le processus de mises à jour successives, décrit au chapitre 6, § 1, conduit évidemment à des révélations exactes.

L'identification des correspondances exactes est importante dans le cadre de l'administration sécurisée des bases de données.

5. Correspondances manichéennes

Etant donnée une correspondance $f : X \multimap\!\!\langle Y$ et un sous-ensemble E de X, la *restriction* de f à E est la correspondance $f_E : E \multimap\!\!\langle f\,E$, égale à f sur E. On note $\mathbf{rev}_E$ la composition $(f_E)^{\sim}\,^{\mathbf{ker}}\,(f_E)^{\mathbf{ker}}$.

Lemme 8.F – Si $f : X \multimap\!\!\langle Y$ est une correspondance et $C \in \mathbf{critique}(f)$, alors, $\forall\, E, \neq \varnothing, \subset C$, $\mathbf{rev}_C\,E = \mathbf{rev}\,E$.

preuve – Soit $E \subset C$, $E \neq \varnothing$. D'après (5.7), $\mathbf{rev}\,E = \{\, x \in X \mid f\,x \supset f^{\mathbf{ker}}\,E \,\}$, ou sinon $\mathbf{rev}\,E = \{\, x \in C \mid f\,x \supset f^{\mathbf{ker}}\,E \,\} \cup \{\, x \in X - C \mid f\,x \supset f^{\mathbf{ker}}\,E \,\}$. Quand $x \in C$, $f\,x = f_C\,x$ et, de manière similaire, comme $E \subset C$ et $E \neq \varnothing$, $f^{\mathbf{ker}}\,E = (f_C)^{\mathbf{ker}}\,E$. Alors,

$\mathbf{rev}\, E = \mathbf{rev}_C\, E \cup \{\, x \in X - C \mid f\, x \supset f^{\mathbf{ker}}\, E \,\}$.

Or, l'ensemble $K = \{\, x \in X - C \mid f\, x \supset f^{\mathbf{ker}}\, E \,\}$ est vide, ce qui établit le lemme. En effet, si tel n'est pas le cas, posons $x' \in K$. On a $f\, x' \supset f^{\mathbf{ker}}\, E$, $\Rightarrow$ $f\, x' \supset f^{\mathbf{ker}}\, C$ ($E \subset C$ et $f^{\mathbf{ker}}$ est antitone), $\Rightarrow$ $f^{\mathbf{ker}}\, (C \cup \{x'\}) = f^{\mathbf{ker}}\, C$, $\Rightarrow$ $f^{\mathbf{ker}}\, (C \cup \{x'\}) \neq f^{\mathbf{ker}}\, X$. L'union $C \cup \{x'\}$ inclut strictement C puisque $x' \in X - C$, donc C n'est pas critique.

□

Le lemme pose que l'on obtient la même révélation pour E sous-ensemble du critique C si on la calcule à partir de f ou à partir de la restriction de f à n'importe lequel de ses sous-ensembles critiques C.

Proposition 8.G – Lorsque **critique**(f) $\neq \emptyset$, une correspondance $f : X \multimap Y$ est manichéenne si et seulement si elle satisfait les propriétés :

(8.5a) $\forall\, C \in$ **critique**(f), $|\, C \,| \geq 2 \Rightarrow$ la restriction f_C est exacte ;

(8.5b) si $\forall\, C \in$ **critique**(f), C est un singleton, alors **critique**(f) a au moins deux éléments.

preuve – *(⇐)* Soit $E \subset X$. On montre que **rev** $E = E$ ou $= X$. Cas1 : $E \neq \emptyset$. Si E est strictement inclus dans un ensemble critique C de f, alors $|\, C \,| \geq 2$. D'où f_C est exacte et, d'après 8.F, **rev** $E = E$. Si $E \in$ **critique**(f), alors, d'après 7.E, **rev** $E = E$. Autrement, d'après le corollaire de 7.E, **rev** $E = X$. Cas2 : $E = \emptyset$. On montre que **rev** $\emptyset = \emptyset$. Si on a $C \in$ **critique**(f) avec au moins 2 éléments x et x', alors $\{x\}$ et $\{x'\}$ sont exactement révélés par f (cas 1 ci-dessus), tout comme d'après 7.B, $\{x\} \cap \{x'\}$. D'où, **rev**$\emptyset = \emptyset$. Si tous les ensembles critiques de f sont des singletons, alors d'après (8.5b), ils sont au moins deux. Et pour la même raison, **rev** $\emptyset = \emptyset$.

(⇒) Soit $C \in$ **critique**(f) et $|\, C \,| \geq 2$. Tout sous-ensemble non vide de C étant, d'après 8.A, exactement révélé par f est, d'après 8.F, exactement révélé par f_C. De plus, l'argument utilisé dans la preuve de (⇐) établit que $\emptyset$ est exactement révélé par f_C. Finalement, si f_C est exacte, alors (8.5a). Supposons alors qu'il y a seulement un sous-ensemble critique et qu'il s'agit du singleton $\{x_0\}$. Nécessairement, x_0 est le seul élément dans X tel que $Y = f\, x_0$. D'où, **rev** $\emptyset = f^{\sim\, \mathbf{ker}}\, f^{\mathbf{ker}}\, \emptyset$ $= f^{\sim\, \mathbf{ker}}\, Y =$ **rev** $x_0 = \{x_0\}$ et f n'est pas manichéenne. On a alors, (8.5b).

□

6. Exemple

Soit la correspondance $f : \{a,b,c,d,e\} \multimap [1,8]$ définie par :

$f\, a = \{1,2,3,5\}$,
$f\, b = \{1,3,4,7\}$,

$f\,c = \{7,8\}$,
$f\,d = \{1,2,4,6\}$,
$f\,e = \{2,5,6\}$.

La figure 9 ci-dessous donne la révélation de chacun des sous-ensembles E de {a,b,c,d,e}.

	Sous-ensemble de E	**rev** E		Sous-ensemble de E	**rev** E
	∅	∅		{e}	{e}
	{a}	{a}		{a,e}	{a,e}
	{b}	{b}		{b,e}	{a,b,c,d,e}
	{a,b}	{a,b}		{a,b,e}	{a,b,c,d,e}
	{c}	{c}		{c,e}	{a,b,c,d,e}
	{a,c}	{a,b,c,d,e}		{a,c,e}	{a,b,c,d,e}
◆	{b,c}	{b,c}		{b,c,e}	{a,b,c,d,e}
	{a,b,c}	{a,b,c,d,e}		{a,b,c,e}	{a,b,c,d,e}
	{d}	{d}		{d,e}	{d,e}
	{a,d}	{a,d}	◆	{a,d,e}	{a,d,e}
	{b,d}	{b,d}		{b,d,e}	{a,b,c,d,e}
◆	{a,b,d}	{a,b,d}		{a,b,d,e}	{a,b,c,d,e}
	{c,d}	{a,b,c,d,e}		{c,d,e}	{a,b,c,d,e}
	{a,c,d}	{a,b,c,d,e}		{a,c,d,e}	{a,b,c,d,e}
	{b,c,d}	{a,b,c,d,e}		{b,c,d,e}	{a,b,c,d,e}
	{a,b,c,d}	{a,b,c,d,e}		{a,b,c,d,e}	{a,b,c,d,e}

Figure 9 : Révélations des sous-ensembles de {a,b,c,d,e}.

On observe que les révélations sont soit exactes, soit pires, i.e., égales à {a,b,c,d,e}. La correspondance f est donc manichéenne. Ses sous-ensembles critiques {a,b,d}, {a,d,e} et {b,c} sont marqués d'un « ◆ » dans la figure 9. D'après la proposition 8.G, la restriction de f à chacun de ses sous-ensembles critiques est une correspondance exacte. On le vérifie ci-dessous pour chacun d'eux. On rappelle que, d'après la proposition 8.D, une correspondance $f : X \multimap Y$ est exacte si et seulement si la réduction f^{CR} de son complément f^{C} est définie sur X.

- *Ensemble critique* {a,b,d}

  On note p la restriction de f à {a,b,d}. On a alors,

  $p\,a = \{1,2,3,5\}$, $p\,b = \{1,3,4,7\}$, $p\,d = \{1,2,4,6\}$.

Donc, la correspondance p est définie sur {a,b,d} à valeurs dans [1,7]. Autrement dit, p:{a,b,d}⟶⟨ [1,7]. Dès lors[26],

$p^{C}\,a = \{4,6,7\},\ p^{C}\,b = \{2,5,6\},\ p^{C}\,d = \{3,5,7\}$, et,

$p^{C\,R}\,a = \{4\},\ p^{C\,R}\,b = \{2\},\ p^{C\,R}\,d = \{3\}$.

Il en résulte que $p^{C\,R}$ est définie sur l'ensemble {a,b,d} tout entier, et donc que p est exacte.

- *Ensemble critique* {a,d,e}

  On note q la restriction de f à {a,d,e}. On a,

  $q\,a = \{1,2,3,5\},\ q\,d = \{1,2,4,6\},\ q\,e = \{2,5,6\}$,

  Donc on a q:{a,d,e}⟶⟨ [1,6]. Dès lors,

  $q^{C}\,a = \{4,6\}\ q^{C}\,d = \{3,5\},\ q^{C}\,e = \{1,3,4\}$ et,

  $q^{C\,R}\,a = \{6\},\ q^{C\,R}\,d = \{5\},\ q^{C\,R}\,d = \{1\}$.

  Il en résulte que $q^{C\,R}$ est définie sur l'ensemble {a,d,e} tout entier, et donc que q est exacte.

- *Ensemble critique* {b,c}

  On note r la restriction de f à {b,c}. On a,

  $r\,b = \{1,3,4,7\},\ r\,c = \{7,8\}$.

  Donc, r:{a,d,e}⟶⟨ {1,3,4,7,8}. Dès lors,

  $r^{C}\,b = \{8\},\ r^{C}\,c = \{1,3,4\}$ et $r^{C\,R}\,b = \{8\},\ r^{C\,R}\,c = \{1,3,4\}$.

  Il en résulte que $r^{C\,R}$ est définie sur l'ensemble {b,c} tout entier, et donc que r est exacte.

[26] On insiste sur le fait que le complémentaire doit être calculé par rapport à [1,7], qui est l'ensemble de valeurs de la restriction p, et non par rapport à [1,8] qui est celui de f.

Conclusion et Perspectives

Ce travail constitue une étape dans la compréhension, par la formalisation, de la tension entre cohérence et confidentialité. La formalisation a été réalisée selon une démarche inspirée des méthodes en programmation. Elle pose que derrière toute fraude existe une propriété qui la rend possible. Cette propriété a été identifiée pour une configuration particulière de droits d'accès et de contrainte d'intégrité, dont on a commenté l'intérêt pratique.

La limitation volontaire à une configuration particulière est apparue essentielle pour atteindre l'objectif visé. En effet, le domaine est vaste. Par ailleurs, l'absence de travaux similaires a limité le recul que l'on pouvait avoir sur la question, tout en obligeant à « fabriquer » les outils formels adéquats. La démarche utilisée est, cependant, suffisamment générale pour envisager son application à l'étude d'autres configurations.

Cette étude ouvre de nombreuses directions de recherche. Ainsi, le dernier chapitre, qui traite de l'efficace des fraudes, ouvre la voie à l'étude d'une caractérisation générale de la notion de « pouvoir de révélation » d'une relation 2-aire, pour l'instant limitée à la modalité exacte/pire. Autrement dit, quand une relation 2-aire est-elle meilleure qu'une autre au regard de la fraude ? La réponse à cette question est importante pour l'administration sécurisée des bases de données, puisqu'elle permet d'informer l'administrateur sur l'étendue potentielle des fraudes.

Une autre direction consiste à étendre l'étude aux modèles de sécurité multi-niveaux[27]. A partir du modèle de Jajodia et Sandhu (Jajodia, Sandhu, 1991), nous avons déjà établi une configuration, similaire à celle étudiée dans ce travail, qui conduit à la même fraude. C'est un premier résultat qui incite à penser que les modèles de sécurité multi-niveaux ne permettent pas, en dépit d'un meilleur contrôle des flux d'information, d'éviter la tension entre les fonctionnalités de cohérence et de confidentialité.

[27] Le principe des modèles de sécurité multi-niveaux est exposé brièvement dans l'annexe A.

Une dernière direction d'étude est celle de la prévention des fraudes. Celles-ci reposant sur l'émergence d'un canal caché, c'est de sa suppression dont il est question. On a déjà mentionné que la suppression des canaux cachés n'était pas toujours possible ou pouvait conduire à réduire les fonctionnalités du système (voir chapitre 3, § 4.1). Une solution à explorer est l'utilisation des résultats de (Dening, Dening, 1977) concernant le contrôle des flux d'information dans les programmes. On présente cette voie de recherche plus en détail dans le cadre de la configuration étudiée.

Dans le cas des contraintes d'intégrité 2-aires **panic** :- R(v,x,y), *not* B(x,y) , le vérificateur de contraintes du système peut être vu comme une procédure qui prend en paramètre d'entrée deux valeurs **x,y** et la relation **B** de la contrainte. Elle renvoie une réponse, positive ou négative, selon que (**x**,**y**) ∈ **B** ou ∉ **B**, dans une variable **check**. Une programmation de cette procédure est donnée figure 10, ci-dessous.

```
Procedure Vérificateur_Integrité(input:x,y,B; Output:check)
begin
check:= « non »;
if (x,y)∈B then check:= « oui »;
end
```

Figure 10 : Vérificateur de contraintes d'intégrité 2-aires

Dans la configuration étudiée, **x** est le seul paramètre d'entrée secret. **check** contient la réponse du vérificateur d'intégrité et n'est donc pas secrète. Le vérificateur de contraintes lit par conséquent des informations secrètes et non secrètes et retourne, à partir d'elles, une information non secrète. Il *déclasse* de ce fait, dans une certaine mesure, de l'information secrète. C'est là l'origine du canal caché.

Les résultats de (Dening, Dening, 1977) concernant le contrôle de flux d'informations classifiées dans les programmes peuvent être appliqués afin de rendre le déclassement « sûr » et donc de supprimer le canal caché. Ils conduisent à la procédure de la figure 11 (**U** est un paramètre identifiant l'utilisateur).

```
Procedure Vérificateur_Integrité(input:x,y,B,U; Output:check)
begin
check:= « non »;
if (x,y)∈B then
   if privilège(U,x)=READ AND privilège(U,y)=READ then check:= « oui »;
end
```

Figure 11 : Vérificateur *sécurisé* de contraintes d'intégrité 2-aires

Par rapport à la procédure précédente, elle ajoute une condition supplémentaire. Pour que **check** contienne « **oui** », il faut que l'utilisateur **U** ait le privilège READ sur **x** et **y**. Dès lors, un utilisateur sans ces privilèges reçoit toujours une réponse négative et n'obtient donc pas d'information sur la base, d'où la suppression du canal caché. En contre-partie, il ne peut plus faire de mises à jour sur la relation R. Il y a donc une diminution des fonctionnalités, conformément à ce qu'on a mentionné précédemment. Sous réserve d'études à mener, mettant en balance ce qu'on gagne et ce qu'on perd, l'utilisation de vérificateurs d'intégrité *sécurisés* peut constituer une solution pour la suppression du canal caché.

Troisième Partie

Classement d'Objets selon leur Histoire

—

Logique Propositionnelle Temporelle,
Expression Régulières et
Automates d'Etats Finis

Introduction

Les vues sont une fonctionnalité essentielle pour les bases de données. Elles permettent de présenter aux différents utilisateurs des données restructurées selon leurs besoins. Elles définissent ainsi, selon la terminologie de (Date, 1990, p 63), des fenêtres différentes sur les mêmes données et sont donc un moyen de personnalisation de l'accès aux données (Korth, Silberschatz, 1988, p 98). De nombreux travaux considèrent que ces fenêtres doivent pouvoir présenter des éléments supplémentaires par rapport au schéma de la base. C'est le cas par exemple d'une vue qui contient l'âge des employés alors que la base ne stocke que leur date de naissance. A ce titre les vues constituent une forme d'abstraction du schéma de la base de données.

Les vues permettent aussi l'indépendance logique. En effet, les programmes qui accèdent aux données uniquement par des vues ne perçoivent pas les éventuels changements du schéma, si les vues existantes sont conservées (Delobel, Lécluse, Richard, 1991, p 168). L'ensemble des vues d'un utilisateur compose la vue externe (*external view*), au sens du modèle Ansi-Sparc[28], qu'a cet utilisateur de la base de données.

Le modèle des P-types (Sales, 1984), est un modèle objet centré sur la notion de vue. Il a été conçu dans le cadre des types abstraits algébriques (Guttag, Horning, 1978). Dans ce modèle, les vues sont définies par des prédicats (ou assertions) sur les objets de la base. Un objet de la base appartient à une vue s'il vérifie les assertions qui la définissent. Il en découle que les objets ne sont pas affectés définitivement à une vue, mais au contraire changent de vues en fonction de leurs mises à jour. L'opération centrale dans la mise en œuvre du modèle des p-types est donc la détermination des vues auxquelles appartiennent les objets. Elle est désignée sous le terme de *classement des objets*.

[28] ANSI/X3/SPARC Study Group on Database Management Systems, Interim report, FDT (ACM SIGMOD bulletin) 7, No. 2, 1975.

On présente au chapitre 9 le modèle des p-types et le système Osiris qui l'implémente. La caractéristique essentielle de ce système est la mise en œuvre d'une structure de données particulière, appelée *espace de classement*, pour le classement des objets. L'utilisation de cette structure conduit à un coût de classement constant.

Le classement dans le système Osiris n'est envisagé que sur la base de l'état courant des objets. Il existe cependant des domaines dans lesquels il est nécessaire de classer les objets selon leur histoire. C'est le cas chaque fois que des « caractéristiques » liées à l'évolution des objets doivent être prises en compte. La mise en œuvre de ce type de classement est l'objet du chapitre 10. Il répond pour cela successivement aux trois questions : qu'est-ce que l'histoire d'un objet, comment exprimer des assertions sur l'histoire des objets et comment les vérifier efficacement ?

Le dernier chapitre, enfin, décrit l'implémentation réalisée de la méthode proposée pour le classement des objets selon leur histoire. Cette méthode, étudiée pour le système Osiris, est cependant générale et peut donc être appliquée à d'autres situations. L'implémentation réalisée l'utilise ainsi pour classer des objets d'un langage prototypique. L'utilisation de ces langages dans le domaine des bases de données étant inhabituel, on en présente, à cette occasion, le principe.

Chapitre 9

Le Modèle des P-types et le Système Osiris

Ce chapitre présente, § 1*, le modèle des p-types proposé par (Sales, 1984) et le système Osiris qui l'implémente,* § 2*. La présentation est basée sur (Simonet, Simonet, 1994) et (Simonet, Simonet, 1995).*

1. Le modèle des p-types

1.1 Notions fondamentales

FAMILLE D'OBJETS, VUES ET P-TYPES

Le modèle des p-types est un modèle de données objet centré sur la notion de vue. Il pose pour principe que les objets du monde réel peuvent être regroupés dans des familles d'objets disjointes. Les personnes, les voitures et les commandes sont des exemples de familles d'objets disjointes. Les objets d'une famille peuvent être perçus selon plusieurs points de vue, ou vues, par différentes catégories d'utilisateurs.

Dans le modèle des p-types, une famille d'objets est définie par un p-type. Un p-type comporte tout d'abord une vue racine, dite vue minimale, puis d'autres vues définies par spécialisation, simple ou multiple, de vues déjà définies. La vue minimale définit les propriétés (attributs et assertions) nécessaires qu'un objet doit satisfaire pour appartenir au p-type. L'extension du p-type est donc celle de la vue minimale. La spécialisation est réalisée par l'ajout de propriétés supplémentaires.

OBJETS

Un objet de la base de données est défini comme une instance d'un p-type. Il value les attributs du p-type et doit vérifier les assertions de la vue minimale. Il appartient également aux autres vues du p-type dont il vérifie les assertions. Classer un objet consiste à déterminer les vues dont il satisfait les propriétés.

Il en découle que les vues auxquelles appartient un objet peuvent changer en fonction des mises à jour qu'il subit. Il en découle aussi qu'un objet peut appartenir à une ou plusieurs vues.

Un objet ne peut changer de p-type. Il est identifié de manière unique dans l'extension du p-type par son Oid. Il doit toujours appartenir au moins à la vue minimale.

1.2 Exemple

La présentation du modèle des p-types au paragraphe précédent est simplifiée, mais elle constitue cependant un cadre conceptuel suffisant pour les chapitres suivants. On l'illustre maintenant par le commentaire de la déclaration du p-type **EMPLOYE** de la figure 1 page suivante.

Dans ce p-type tous les attributs sont définis au niveau de la vue minimale. Le p-type contient trois vues : **Minimale, Senior** et **Parent**. La vue minimale comporte une seule assertion : **âge ≤ 120**.

Une instance de ce p-type est par exemple l'objet (Dupont, 27, 3, 0, programmeur). On vérifie qu'il value tous les attributs du p-type et qu'il appartient à sa vue minimale. Il n'appartient par contre à aucune autre vue.

En revanche, l'objet (Durand, 62, 0, 22, directeur) appartient aux vues **Minimale** et **Sénior** du p-type. A l'issue de la mise à jour de l'attribut **nbr_enfants** de cet objet à 1, celui-ci appartient aussi à la vue **Parent**.

Un schéma de base de données contient en général plusieurs p-types. Les attributs d'un p-type peuvent alors référencer des objets d'autres p-types. Par exemple, si l'on avait défini un p-type **VOITURE**, on aurait pu ajouter l'attribut **possède : VOITURE** au p-type **EMPLOYE** de l'exemple.

```
P-type EMPLOYE

  Vue  Minimale
        nom : string;
        âge : unsigned integer;
        ancienneté : unsigned integer;
        nbr_enfants : unsigned integer;
        statut : string;
        âge ≤ 120;
   Fin Vue

   Vue Senior : spécialise Minimale
        âge ≥ 60;
        âge ≤ 65 ∧ nbr_enfants=0 → ancienneté ≥ 20;
   Fin Vue

   Vue Parent : spécialise Minimale
        nbr_enfants ≥ 1;
   Fin Vue

Fin P-type
```

Figure 1 : Le p-type **EMPLOYE**

2. Le système Osiris

2.1 Classement des objets

L'opération centrale dans la mise en œuvre du modèle des p-types est la détermination des vues auxquelles appartiennent les objets. Le coût de cette opération, appelée classement des objets, doit donc être faible.-

Le système Osiris constitue une implantation du modèle des p-types. Le classement des objets est réalisé par l'utilisation d'une structure de données particulière, appelée *espace de classement*.

L'espace de classement est une partition de l'espace des objets dont les blocs, appelés eq-classes, sont des classes d'équivalence pour la satisfaction des prédicats de domaine présents dans les assertions du p-type. On présente maintenant sa construction.

2.2 Construction de l'espace de classement

2.2.1 Prédicats de domaines

L'espace de classement d'un p-type est construit à partir des prédicats de domaine qui apparaissent dans ses assertions. Un prédicat de domaine sur un attribut A d'un p-type est de la forme $A \in D_i$ où D_i est un sous domaine du domaine Δ de l'attribut. Les Simple Predicates de (Oszu, Valduriez, 1991) sont un cas particulier de prédicats de domaine limité à la forme Attribut R Valeur, où $R \in \{<,>,\leq,\geq,=,\neq\}$.

Les assertions du p-type **EMPLOYE** défini au § 1.2 sont toutes composées de prédicats qui peuvent être mis sous la forme de prédicats de domaine. Par exemple, **nbr_enfants≥1** peut se réécrire sous la forme **nbr_enfants∈[1,∞[**.

2.2.2 Sous-domaines stables des attributs

La construction de l'espace de classement repose sur la notion de sous-domaine stable (SdS) que l'on présente maintenant.

Tout prédicat de domaine sur un attribut A partitionne son domaine en deux sous-domaines : D_i et $D\text{-}D_i$. Le produit des partitions engendrées par tous les prédicats de domaine sur A définit une partition de son domaine en blocs disjoints et complémentaires (Stanat, McAllister, 1977). Ces blocs sont appelés sous-domaines stables de l'attribut A, car il y a « stabilité » de la valeur de vérité de tous les prédicats de domaine lorsqu'un

attribut change de valeur en restant dans le même bloc de la partition. En effet, si P est l'ensemble des prédicats de domaine du p-type sur l'attribut A et S l'ensemble des sous-domaines stables de A, on a,

$$\forall s \in S,\ \forall a_1, a_2 \in s,\ \forall p \in P,\ p(a_1) = p(a_2)$$

par construction de la partition (Simonet, Simonet, 1989).

On détermine maintenant, par exemple, l'ensemble des sous-domaines stables de l'attribut **âge** du p-type **EMPLOYE**, voir § 1.2, figure 1. Cet attribut intervient dans les trois prédicats de domaine,

âge $\geq$ **60** (âge $\in [60,\infty[$), **âge** $\leq$ **120** (âge $\in [0,120]$) et **âge** $\leq$ **65** (âge $\in [0,65]$),

ce qui conduit aux trois partitions :

[0,59], [60,∞[,

[0,65], [66,∞[,

et [0,120], [121,∞[.

Le produit de ces partitions donne alors les sous-domaines stables de l'attribut âge :

SdS(âge) = {d_{11} : [0,59], d_{12} : [60,65], d_{13} : [66,120], d_{14} : [120,∞[}.

De la même manière, les sous-domaines stables des attributs **ancienneté**, et **nbr_enfants** sont :

SdS(ancienneté) = {d_{21} : [0,19], d_{22} : [20,∞[},

SdS(nbr_enfants) = {d_{31} : [0], d_{32} : [1,∞[}.

Les attributs **nom** et **statut** n'intervenant dans aucune assertion du p-type **EMPLOYE**, ils ne constituent pas des attributs classificateurs et leurs sous-domaines stables sont simplement leurs domaines respectifs.

2.2.3 L'espace de classement

Le produit cartésien des sous-domaines stables de tous les attributs classificateurs du p-type constitue l'espace de classement du p-type. D'après la définition des sous-domaines stables, les objets qui appartiennent aux mêmes n-uples de l'espace de classement satisfont les mêmes prédicats et donc les mêmes assertions. Ils appartiennent par conséquent aux mêmes vues. Les n-uples de l'espace de classement constituent donc des classes d'équivalence pour la satisfaction des prédicats de domaine des assertions du p-type et sont appelés eq-classes.

L'espace de classement du p-type **EMPLOYE** ainsi que les vues associées à ses eq-classes sont représentés par le diagramme en trois dimensions de la figure 2 ci-dessous[29].

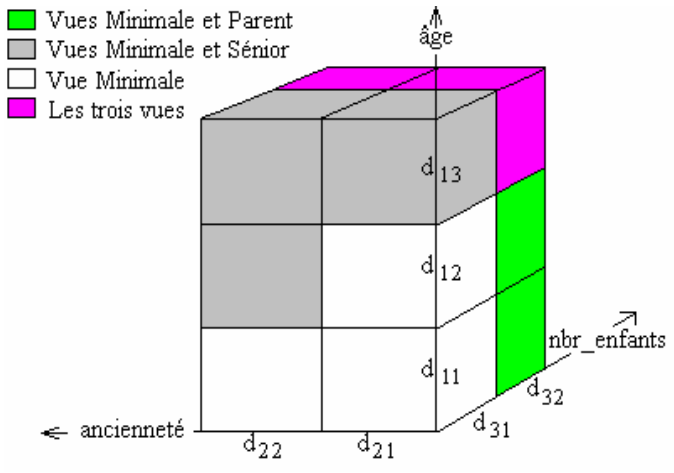

Figure 2 : Espace de classement du p-type **Employé**

Une modification dans un p-type (ajout, suppresssion, modification, d'une vue, d'une assertion, d'un attribut) nécessite une restructuration de l'espace de classement qui peut être plus ou moins importante.

2.3 Utilisation de l'espace de classement

Les vues auxquelles appartient l'objet sont déterminées par le positionnement de l'objet dans l'espace de classement. Pour cela, le système détermine les sous-domaines stables auxquels appartiennent les valeurs de ses attributs classificateurs. Par exemple, pour un objet du p-type **EMPLOYE** dont la valeur de l'attribut **ancienneté** est de 15, celle de l'attribut **âge** de 67 et celle de l'attribut **nbr_enfants** de 2, les sous-domaines stables correspondants sont d_{21}, d_{13}, et d_{32}. Ils déterminent l'eq-classe à laquelle appartient l'objet et donc les vues de l'objet. Ici, l'objet appartient aux trois vues définies sur le p-type **EMPLOYE**.

L'espace de classement n'est toutefois jamais représenté explicitement dans sa totalité par le système Osiris. En effet, le nombre d'eq-classes est une fonction exponentielle au nombre d'attributs classificateurs. En pratique, le classement est réalisé par un automate

[29] Pour simplifier, on a pas représenté les eq-classes contenant le SdS d_{14} qui correspond aux objets dont l'âge est supérieur à 120 et qui n'appartiennent donc pas au p-type.

à architecture connexionniste[30] dont la couche de sortie est constituée des vues du p-type et la couche d'entrée des sous-domaines stables des attributs (Bassolet, Simonet, Simonet, 1996). Le coût de classement est constant et est une fonction polynômiale du nombre d'attributs et du nombre d'assertions.

Tout objet est classé avant d'être inséré dans la base. Le classement fournit l'ensemble des vues que satisfait l'objet. Cet ensemble ne doit pas être vide, car tout objet doit satisfaire au moins les contraintes de la vue minimale.

Par ailleurs, les objets persistants sont indexés par les sous-domaines stables (Simonet, Simonet, 1996). Ainsi, seules sont représentées les eq-classes contenant au moins un objet, ce qui évite l'explosion combinatoire liée à la taille de l'espace de classement.

[30] Une architecture connexionniste est constituée d'un graphe orienté de cellules où chaque cellule calcule une activité à partir des activités de ses cellules d'entrée et communique son activité aux cellules connectées en sortie.

Chapitre 10

Classement d'Objets selon leur Histoire

Dans le système Osiris, les objets sont classés selon leur état courant. Il existe cependant des domaines dans lesquels il est nécessaire de les classer selon leur histoire. La mise en œuvre de ce type de classement est l'objet de ce chapitre. On répond successivement pour cela aux trois questions : qu'est-ce que l'histoire d'un objet, § 1*,? comment exprimer des propriétés sur l'histoire des objets,* § 2*,? et comment les vérifier,* § 3 *?*

1. Histoire d'un objet

Chaque mise à jour sur un objet en crée un nouvel état. La succession des mises à jour sur un objet O établit une *séquence finie d'états*, notée S(O), appelée *histoire* de l'objet O. Elle contient au moins l'état initial de l'objet (celui de son insertion dans la base de données). S(O) est isomorphe à un intervalle [1,n] de l'ensemble des entiers naturels. Dans cet intervalle, 1 représente l'état initial et n l'état courant de l'objet. Ces deux états bornent l'histoire de l'objet.

Le classement des objets selon leur histoire est réalisé sur la base d'assertions qui expriment des propriétés sur l'histoire des objets. Elles sont appelées des *assertions historiques*.

Exemple – Soit la vue **Monstre** définie par l'assertion historique :

(10.1) **Une fois la taille de 20 mètres atteinte, le poids n'est jamais descendu en dessous de 400 kilogrammes.**

Un objet O vérifie (10.1), et appartient donc à la vue **Monstre**, si le premier état de S(O) dans lequel l'attribut **taille** a une valeur ≥ à 20 n'est suivi que d'états dans lesquels la valeur de l'attribut **poids** est > à 400 kilogrammes. On note par ailleurs que, tout comme les assertions du chapitre précédent, les assertions historiques ne portent que sur un seul objet.

Une manière habituelle d'exprimer des propriétés sur des séquences est la logique temporelle. C'est dans ce formalisme que seront exprimées les assertions historiques. Il est déjà utilisé dans les bases de données pour exprimer les contraintes d'intégrité dynamiques, voir chapitre 1, § 3, ainsi que pour étendre l'algèbre relationnelle à l'interrogation des bases de données temporelles (Gabbay, Mc Brien, 1991).

2. Expression des assertions historiques

Dans la suite, on ne considère que la modalité « passé » de la logique temporelle. Par ailleurs, à l'instar du système Osiris, seule des formules propositionnelles sont considérées. Dès lors, on exprimera les assertions historiques en Logique Temporelle Propositionnelle du Passé (LTPP).

2.1 Logique temporelle propositionnelle du passé

Ce paragraphe présente brièvement la syntaxe et la sémantique de la LTPP. Il est basé sur (Turner, 1986, chapitre 6), (Thayse et *al.*, 1989, pp 15-34, 179-258), (Audureau, Enjalbert, Farinas del Cerro, 1990, chapitre 2) et (Manna, Pnueli, 1991, chapitre 3).

SYNTAXE

De manière informelle, la LTPP est similaire à la logique propositionnelle qu'elle étend par deux opérateurs « temporels » : un opérateur *universel* sur le passé noté $\Box$ (« toujours dans le passé ») et un opérateur *existentiel* noté $\Diamond$ (« au moins une fois dans le passé »). D'un point de vue technique, la LTPP ne présente pas de différences avec la logique modale.

Plus formellement, le langage de la LTPP est défini au moyen :

(*i*) d'un ensemble de propositions élémentaires Vp = { p,q,r, … } ;
(*ii*) des opérateurs logiques : $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$;
(*iii*) des symboles de parenthèse ;
(*iv*) de constantes logiques : **vrai** et **faux** ;
(*v*) des deux opérateurs temporels : $\Box$ et $\Diamond$;

(*vi*) de règles de formation :
  (*a*) les propositions élémentaires sont des formules ;
  (*b*) si A et B sont des formules alors $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $A \Leftrightarrow B$, $\Box A$, $\Diamond A$, (A) sont des formules ;
  (*c*) toute formule est obtenue par application de (*a*) et (*b*) un nombre fini de fois.

SEMANTIQUE

Il n'est pas possible d'adopter pour la LTPP une sémantique compositionnelle comme pour la logique « classique ». En effet, les opérateurs temporels $\Box$ et $\Diamond$ ne sont pas vérifonctionnels. Ainsi, si p est une proposition vraie, alors $\Box p$ peut être soit **vrai** soit **faux** selon « le passé ».

Les formules de LTPP sont interprétées selon des modèles d'interprétation de la forme $M = <W,R,m>$. Dans ce triplet, W désigne un ensemble de dates (ou instants)[31], R désigne une relation binaire sur W dite relation de précédence temporelle et $m : W \rightarrow P(Vp)$ une fonction de l'ensemble des dates dans l'ensemble des parties de Vp (voir le point (*i*) de la définition de la syntaxe de la LTPP, page précédente). Elle associe à chaque date l'ensemble des énoncés élémentaires qui sont vrais à cette date.

Etant donné un modèle M, une date $w \in W$ et une formule A, on définit la relation « M satisfait A à la date w » (notation $M,w \models A$) à l'aide des clauses suivantes :

$M,w \models p$	ssi	$p \in m(w)$;
$M,w \models \neg A$	ssi	non ($M,w \models A$) ;
$M,w \models A \wedge B$	ssi	$M,w \models A$ et $M,w \models B$;
$M,w \models \Box A$	ssi	quel que soit $w' \in W$, $w'Rw$ implique $M,w' \models A$;
$M,w \models \Diamond A$	ssi	il existe $w' \in W$ tel que $w'Rw$ et $M,w' \models A$.

La nature universelle de $\Box$ et la nature existentielle de $\Diamond$ rendent ces deux opérateurs duaux par la relation $\Diamond A \equiv \neg \Box \neg A$ (ou $\Box A \equiv \neg \Diamond \neg A$).

Tout comme en logique « classique » des propositions, la sémantique des opérateurs $\vee$, $\rightarrow$ et $\leftrightarrow$ peut être définie à partir de celle de $\neg$ et $\wedge$: $A \vee B \equiv \neg(\neg A \wedge \neg B)$, $A \Rightarrow B \equiv \neg(A \wedge \neg B)$, $A \Leftrightarrow B \equiv \neg(A \wedge \neg B) \wedge \neg(B \wedge \neg A)$.

[31] On ne s'intéresse pas dans ce travail à la question de la granularité des dates. A ce titre, voir par exemple (Fauvet, Canavaggio, Scholl, 1997).

2.2 Modèle pour les assertions historiques

Dans ce paragraphe on donne le modèle d'interprétation des assertions historiques. Au préalable, on introduit la notion de vecteur de vérité.

Soit H une assertion historique comportant n propositions élémentaires p_i. Il lui correspond 2^n *vecteurs de vérité* qui forment l'ensemble noté H_Σ. Ils sont obtenus par le produit cartésien $\overset{n}{\underset{i=1}{X}}(\text{vrai}, \text{faux})$, ou plus simplement $\overset{n}{\underset{i=1}{X}}(+,-)$. Par exemple, si H est composée de deux propositions élémentaires p_1 et p_2, il lui correspond les quatre vecteurs de vérité $H_\Sigma = \{ (+,+), (+,-), (-,+),(-,-) \}$.

Selon les propriétés élémentaires que vérifie l'objet O, il lui correspond un unique vecteur de vérité de H_Σ. Ainsi, s'il vérifie p_1, mais pas p_2, il lui correspond le vecteur de vérité $(+,-)$. A S(O) correspond donc canoniquement une séquence unique de vecteurs de vérité, notée $S_\Sigma(O)$.

Or, la valeur de vérité de H, dans l'état courant, ne dépend que des valeurs de vérité de ses propriétés élémentaires sur les différents états de S(O) et donc de $S_\Sigma(O)$. Il en découle que le modèle d'interprétation de H pour l'objet O est $S_\Sigma(O)$. Autrement dit, en reprenant les notations du paragraphe précédent, on a :

W : ensemble des états de l'objet O ;
R : relation d'ordre sur les états de l'objet O décrivant la séquence S(O) ;
m : fonction associant à tout état de l'objet O le vecteur de vérité de H_Σ correspondant.

Si la relation d'ultériorité R est réflexive, il n'y pas de différence entre passé et présent puisque tout état est son propre passé. $\Box$A (resp. $\Diamond$A) s'interprète alors « toujours (resp. au moins une fois) dans le passé, y compris dans le présent ». Considérer la relation d'ultériorité antiréflexive[32] permet, au contraire, de singulariser le présent. $\Box$A (resp. $\Diamond$A) s'interprète alors « toujours (resp. au moins une fois) dans le passé, indépendamment du présent ». Selon l'alternative retenue les opérateurs $\Box$ et $\Diamond$ possèdent des propriétés différentes. Ainsi, ils ne sont idempotents que si la relation d'ultériorité est réflexive.

Le modèle retenu pour l'interprétation des assertions historiques laisse libre le choix de la réflexivité ou de l'antiréflexivité de R. Dans la suite on considère la relation R, et donc le temps, antiréflexifs. Les résultats établis dans les paragraphes suivants

[32] R est antiréflexive ssi $\forall x \in X \neg(xRx)$.

s'adaptent néanmoins aisément au cas réflexif. Dans le cadre retenu, l'assertion historique (10.1) s'écrit,

(10.2) $\Box($**◊taille>20** $\Rightarrow$ **poids>400** $)\ \wedge$ **◊taille>20**.

Après avoir abordé l'expression des assertions historiques, le paragraphe suivant s'intéresse à leur évaluation. Les assertions historiques posent à ce titre deux problèmes : (*i*) chaque évaluation d'une assertion historique sur un objet O requiert l'accès à son histoire, ce qui a un coût élevé, et (*ii*) plus l'histoire est « longue » plus le coût augmente encore.

La méthode proposée dans la suite ne requiert pas l'accès à l'histoire de l'objet. Son coût est dès lors limité et constant. Elle s'apparente aux méthodes dites *a-historiques* proposées pour la vérification des contraintes d'intégrité dynamiques, voir chapitre 2, § 2. Son principe est de traduire les assertions historiques en expressions régulières puis en automates d'états finis. Chaque objet reçoit alors un pointeur sur l'automate de chaque assertion historique. Il est déplacé lors des mises à jour suivant l'arc de l'automate dont la condition est vérifiée. Si le noeud désigné est final, alors l'assertion historique correspondante est vérifiée.

3. Evaluation des assertions historiques

3.1 Expressions régulières

Soient Σ et $\Sigma' = \{\cdot, \cup, \cap, -, *, +, \varnothing, (,)\}$ deux alphabets disjoints. Un mot P sur $\Sigma \cup \Sigma'$ est une expression régulière sur Σ si et seulement si P est soit :

(*i*) un caractère de Σ ;
(*ii*) le caractère $\varnothing$;
(*iii*) de la forme $(Q \cdot R)$, $(Q \cup R)$, $(Q \cap R)$, $(Q - R)$, $(Q)^*$, ou $(Q)^+$, où Q et R sont elles-mêmes des expressions régulières sur Σ.

Dans la suite on omettra les parenthèses lorsque cela ne soulève pas d'ambiguïté.

Les règles suivantes associent à chaque expression régulière P un langage L(P) sur Σ:

(*i*) le langage décrit par $\varnothing$ est le langage vide ;
(*ii*) le langage décrit par $a \in \Sigma$ est le mot « a » ;
(*iii*) pour les expressions régulières Q et R sur Σ,

(*concaténation*) $L(Q \cdot R) = L(Q) \cdot L(R)$

(*union*) $L(Q \cup R) = L(Q) \cup L(R)$

(*intersection*) $L(Q \cap R) = L(Q) \cap L(R) = L(Q - (Q - R))$

(*différence*) $L(Q - R) = L(Q) - L(R)$

(*fermeture de Kleene*) $L(Q^*) = L(Q)^*$

$L(Q^+) = L(Q) \cdot L(Q^*)$.

Dans chacune des égalités ci-dessus, les symboles qui apparaissent à droite sont les opérateurs ensemblistes classiques ($\cup, \cap, -$) ou ceux particuliers sur les langages ($\cdot, *, ^+$).

Un langage L sur Σ est *régulier* s'il existe une expression régulière P telle que $L = L(P)$.

3.2 Assertions historiques et expressions régulières

Soit une assertion dynamique H. Certaines séquences d'éléments de H_Σ la rendent vraie. On note L(H) le langage sur H_Σ formé par ces séquences. Selon le théorème 10.E, ci-après, ce langage est régulier. Sa démonstration utilise les quatre lemmes que l'on présente maintenant.

On introduit les notations suivantes :

$[\pm]$ désigne l'ensemble de tous les vecteurs de vérité ;

$[+_i]$ désigne l'ensemble de tous les vecteurs de vérité pour lesquels p_i est vraie.

Lemme 10.A – $\forall$ i, $1 \le i \le n$, $L(p_i) = [\pm]^* \cdot [+_i]$.

Preuve – Pour qu'une séquence de vecteurs de vérité satisfasse l'assertion p_i, il faut et il suffit qu'elle se termine par un élément de $[+_i]$.

□

Lemme 10.B – $\forall$ F, $L(\Diamond F) = L(F) \cdot [\pm]^+$.

Preuve – Une séquence de vecteurs de vérité satisfait $\Diamond$F si et seulement si elle est égale à la concaténation d'une séquence satisfaisant F et d'une séquence non vide.

□

Lemme 10.C – $\forall$ F, $L(\neg F) = [\pm]^+ - L(F)$.

Preuve – Une séquence de vecteurs de vérité satisfait $\neg$F si et seulement si elle est de longueur ≥ 1 et n'appartient pas à L(F). On insiste sur le fait que la longueur de la

séquence doit être ≥ 1. En effet, aucune formule temporelle n'est vérifiée pour la séquence vide lorsque la relation d'ultériorité du modèle d'interprétation n'est pas réflexive (on rappelle que c'est l'hypothèse retenue). Cette condition explique que $L(\neg F)$ est égal à $[\pm]^+ - L(F)$ et non à $[\pm]^* - L(F)$.

□

Lemme 10.D – $\forall F, \forall G, L(F \vee G) = L(F) \cup L(G)$. □

Théorème 10.E – Pour toute assertion historique H, L(H) est un langage régulier.

Preuve – Toute formule H est, d'après le § 2.1, une composition finie de ses propositions élémentaires p_i, $1 \leq i \leq n$, à l'aide des opérateurs $\Diamond$, $\neg$ et $\vee$. Les opérateurs $\wedge$ et $\Box$ s'expriment en fonction des trois précédents (en particulier, on a déjà mentionné que $\Box \equiv \neg\Diamond\neg$). Or, pour tout i, $1 \leq i \leq n$, le langage $L(p_i)$ est régulier puisque, d'après le lemme 10.A, il est égal à la concaténation des langages réguliers $[\pm]^*$ et $[+_i]$. Par ailleurs, si le langage L(F) est régulier, alors le langage $L(\Diamond F)$ l'est aussi comme concaténation de langages réguliers (lemme 10.B). De même, le langage $L(\neg F)$ comme différence de langages réguliers (lemme 10.C). Enfin, si L(F) et L(G) sont réguliers, alors $L(F \vee G)$ l'est aussi comme union de langages réguliers (lemme 10.D).

□

En vertu du théorème 10.E, il existe pour toute assertion historique H une expression régulière R(H) qui décrit le langage L(H). Il résulte des lemmes 10.A, 10.B, 10.C et 10.D que R(H) est inductivement définie par :

(*i*) $R(p_i) = [\pm]^* \cdot [+_i]$;

(*ii*) $R(\Diamond F) = R(F) \cdot [\pm]^+$;

(*iii*) $R(\neg F) = [\pm]^+ - R(F)$;

(*iv*) $R(F \vee G) = R(F) \cup R(G)$.

On a alors, en particulier,

(*v*) $R(\Box F) = R(\neg\Diamond\neg F) = [\pm]^+ - ((([\pm]^+ - R(F)) \cdot [\pm]^+)$;

(*vi*) $R(F \wedge G) = R(F) \cap R(G)$.

Exemple 1 – Soit l'assertion historique $\Box(\Diamond(\text{âge}=18) \vee \text{âge}=18)$. Elle exprime que l'attribut **âge** doit avoir été initialisé à 18. En posant $p_1 \equiv (\text{âge}=18)$ on a,

$R(\Box(\Diamond p_1 \vee p_1))$

$= [\pm]^+ - (([\pm]^+ - R(\Diamond p_1 \vee p_1)) \cdot [\pm]^+)$

$= [\pm]^+ - (([\pm]^+ - (R(\Diamond p_1) \cup R(p_1))) \cdot [\pm]^+)$

$= [\pm]^+ - ((([\pm]^+ - ((R(p_1) \cdot [\pm]^+) \cup R(p_1))) \cdot [\pm]^+)$
$= [\pm]^+ - ((([\pm]^+ - (([\pm]^* \cdot [+_1] \cdot [\pm]^+) \cup ([\pm]^* \cdot [+_1]))) \cdot [\pm]^+)$

Exemple 2 – $R(\textbf{vrai}) = R(P \vee \neg P) = ([\pm]^* \cdot [+]) \cup ([\pm]^+ - ([\pm]^* \cdot [+]))$

$R(\textbf{faux}) = R(P \wedge \neg P) = ([\pm]^* \cdot [+]) \cap ([\pm]^+ - ([\pm]^* \cdot [+]))$

Un objet O vérifie l'assertion historique H dans l'état courant si $S_\Sigma(O) \in L(H)$ D'après le théorème 10.E, cela revient à vérifier que $S_\Sigma(O)$ est un mot du langage décrit par R(H). Vérifier une assertion historique pour un objet revient donc à vérifier qu'un mot appartient au langage décrit par une expression régulière. Une manière classique de procéder est d'utiliser des automates d'états finis.

3.3 Automates d'états finis

Un automate d'états finis, non déterministe, est un 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ où :

(*i*) Q est un ensemble fini d'états ;
(*ii*) Σ est un ensemble fini de symboles d'entrée ;
(*iii*) δ est une application de $Q \times \Sigma$ dans l'ensemble 2^Q des parties de Q ;
(*iv*) $q_0 \in Q$ est l'état initial ;
(*v*) $F \subset Q$ est l'ensemble des états finaux.

Un automate est *déterministe* si et seulement si, pour tout élément q de Q et tout élément a de Σ, l'ensemble $\delta(q,a)$ contient au plus un élément. Tout automate non déterministe peut être rendu déterministe. Dès lors, dans la suite, on ne considère que des automates déterministes.

Un *déplacement* dans M est une relation binaire sur $Q \times \Sigma^*$ notée $\mapsto$. Si $\delta(q,a)$ contient q', alors pour tout w de Σ on a le déplacement $(q,aw) \mapsto (q',w)$. Le langage reconnu par M est $L(M) = \{w \mid w \in \Sigma^* \text{ et } (q_0,w) \overset{*}{\mapsto} (q,\mathbf{e}) \text{ pour tout q dans F}\}$, où **e** désigne la chaîne vide.

On montre, voir (Aho, Ullman, 1972), qu'à toute expression régulière R correspond un automate d'états finis qui reconnaît le même langage. Il en découle qu'il existe pour toute assertion historique H un automate M qui reconnaît le langage L(H). Dès lors, un objet O vérifie l'assertion H si $S_\Sigma(O)$ est un mot du langage reconnu par M.

(Hopcroft, Ullman, 1979) décrit la construction de ces automates. On donne ci-dessous l'automate[33] correspondant à l'assertion historique (10.2).

[33] On présente au chapitre suivant, § 3.3, l'outil logiciel que l'on a utilisé pour les calculer.

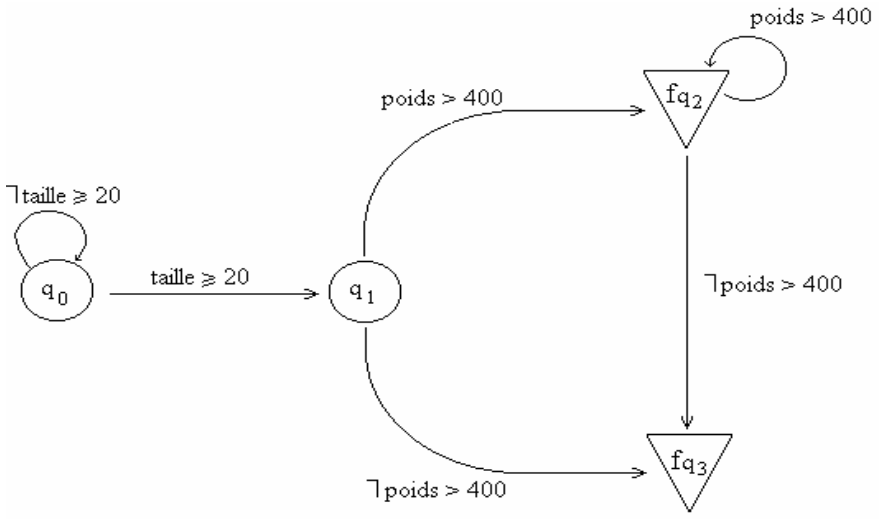

Figure 3 : Automate d'états finis pour l'assertion historique (10.2)

L'automate de la figure 3 comporte quatre états dont deux, fq_2 et fq_3, sont finaux. Il reconnaît les séquences de vecteurs de vérité de l'assertion historique (10.2) qui la rendent vraie. Les vecteurs de vérité sont $H_\Sigma = \{$(**taille≥20, poids>400**), (**taille≥20,¬ poids>400**),(¬**taille≥20,poids>400**),(¬**taille≥20,¬ poids>400**)$\}$. Par simplification, le label **taille≥20** sur l'arc entre les états q_0 et q_1 désigne tous les vecteurs de vérité pour lesquels cette proposition est vraie, c'est à dire (**taille≥20, poids>400**) et (**taille≥20, ¬ poids>400**). Il en est de même pour les autres arcs de l'automate.

La figure 4 ci-dessous représente l'automate d'états finis correspondant à l'assertion historique $\Box(\Diamond p_1 \vee p_1)$ de l'exemple 1 du paragraphe précédent.

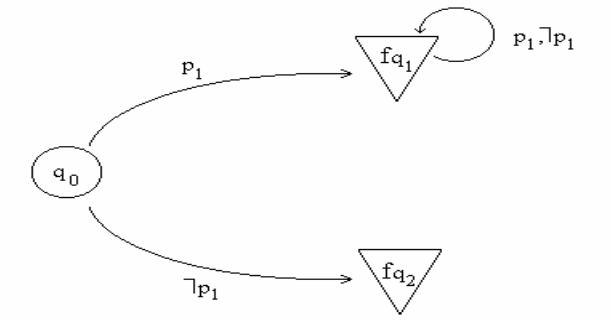

Figure 4 : Automate d'états finis pour l'assertion historique $\Box(\Diamond p_1 \vee p_1)$

Finalement, l'évaluation des assertions historiques par les automates d'états finis comporte trois étapes, à l'instar des méthodes de vérification pour les contraintes d'intégrité dynamiques (voir chapitre 2, § 2) :

(*i*) la première étape consiste en la construction des automates d'états finis des assertions historiques et l'ajout à chaque objet d'un pointeur sur chacun de ces automates. Ce pointeur est initialisé sur l'état initial des automates.

(*ii*) la deuxième étape se déroule tout au long de la vie des objets. A chaque modification d'un objet, on déplace les pointeurs sur les automates en fonction du vecteur de vérité qui correspond au nouvel état de l'objet. Comme les graphes considérés sont déterministes, il ne peut pas y avoir de choix entre plusieurs déplacements possibles. Selon la terminologie de (Lipeck, Feng, 1988), cette étape est appelée *monitoring* de l'objet .

(*iii*) la troisième, et dernière, étape consiste en la détermination du type de l'état auquel on aboutit. S'il s'agit d'un noeud final, cela signifie que l'histoire de l'objet vérifie l'assertion historique associée à l'automate. De plus, si cet état n'a aucun arc sortant, alors l'assertion s'évaluera à faux dans tous les états suivants.

Cette méthode d'évaluation des assertions historiques conduit naturellement à celle de classement des objets selon ce type d'assertion : un objet appartient à une vue définie par des assertions historiques s'il vérifie chacune d'elles.

Chapitre 11

Implémentation

Dans ce chapitre on décrit, § 3, l'implémentation réalisée de la méthode de classement d'objets selon des assertions historiques, décrite au chapitre précédent. Le § 1 en précise le cadre. L'originalité de cette implémentation est qu'elle classe des objets d'un langage prototypique. L'utilisation de ces langages dans le domaine des bases de données étant inhabituel, on en présente, § 2, le principe.

1. Cadre de l'implémentation

On a proposé, dans ce qui précède, une méthode pour le classement d'objets selon des assertions historiques. Dans ce qui suit on décrit l'implémentation que l'on a réalisée de cette méthode. Les objets qu'elle considère sont ceux du langage prototypique NewtonScript (Apple Corp., 1996 (1)) (Smith, 1994). La simplicité et l'extrême « dépouillement » de la notion d'objet sur laquelle ce langage est fondé en tant que langage prototypique a permis une implémentation elle-même simple et « naturelle ». L'implémentation n'a pas été réalisée sur le système Osiris pour des raisons contingentes, extérieures à cette thèse. Cependant, l'expérience acquise sur les langages prototypiques, et leur apport, peut constituer un premier élément d'une réflexion sur leur adéquation à la manipulation d'objets de ce système[34].

Il convient de souligner que l'implémentation réalisée constitue plus une étude de faisabilité qu'une application aboutie. Elle a cependant été fort utile comme support à la réflexion en permettant de mettre en évidence les aspects essentiels de la méthode. Comme on le verra par la suite, le classement est écrit en NewtonScript et la

[34] Cette réflexion devrait aussi tenir compte des langages à base de rôles. Pour une synthèse sur ces langages voir par exemple (Albano, Diotallevi, Ghelli, 1995).

génération des automates est réalisée par un assemblage de C++, de Web et de copier-coller.

Les langages prototypiques constituent une famille de langages dont l'utilisation est inhabituelle dans les bases de données. C'est pourquoi, on les présente dans le paragraphe suivant.

2. Les langages prototypiques

Les fondements des langages prototypiques se sont constitués progressivement et évoluent encore. Les langages prototypiques ont été principalement développés par la communauté de l'intelligence artificielle et celles des langages Lisp et Smalltalk. Le traité d'Orlando (Stein, Lieberman, Ungar, 1988) est une première synthèse de leurs caractéristiques.

Dans la suite, on présente brièvement leurs caractéristiques essentielles que l'on illustre par des exemples en NewtonScript. On présente aussi le gestionnaire de persistance NewtonStore, interfacé à NewtonScript.

NewtonScript est l'un des seuls langages prototypiques pour lesquels il existe une implémentation commercialisée[35]. A ce titre, il dispose d'un environnement de développement élaboré, le Newton Tool Kit (Apple Corp., 1996 (2)). Self (Ungar, Smith, 1987) et Kevo (Smith et *al*, 1987) sont deux exemples de langages prototypiques de recherche.

2.1 Objets sans classe

Les langages à base de classes (C++, Java, …) distinguent les notions d'instances et de classes. Une instance appartient à une classe, qui en détermine la structure et le comportement. La classe constitue donc une description des instances.

Les langages prototypiques, au contraire, ne présentent pas cette dichotomie. Il n'existe que des objets. Chaque objet a sa structure propre et contient son comportement, les deux pouvant évoluer simplement. Autrement dit, l'objet n'est pas lié à une description générique quelconque. De cette caractéristique découlent les deux principaux avantages reconnus aux langages prototypiques (Ungar, Smith, 1987) : simplicité et caractère concret. Le titre de (Ungar, Smith, 1987) est, en ce sens, tout à fait significatif : *The Power of Simplicity*.

[35] L'interpréteur NewtonScript n'est disponible que sur Newton (Smith, 1994) et c'est donc cette plate-forme matérielle que l'on a utilisée pour l'implémentation.

Les langages prototypiques sont simples, car la structure et le comportement des objets sont directement accessibles. Au contraire, dans les langages à base de classes, leur détermination implique le parcours de deux liens : le lien d'instantiation (a-kind-of) pour connaître la classe dont l'objet est instance et les liens de sous-classement (is-a) pour déterminer les super classes de cette classe.

Les langages prototypiques sont concrets car la programmation consiste à travailler directement sur les objets, plutôt qu'à travailler sur des classes, pour produire des effets indirects sur les objets. En termes plus imagés, Walter Smith suggère l'analogie suivante (Smith, 1995) : la programmation avec un langage à base de classes consiste à écrire le manuel pour le montage de la bicyclette, alors qu'avec un langage prototypique on monte effectivement la bicyclette.

En contrepartie, le partage de structures et de comportements similaires entre instances est moins aisé dans les langages prototypiques que dans les langages à base de classes.

2.2 Création d'objets sans classe

En l'absence de classes, les objets ne peuvent plus être créés comme instance de celles-ci. Les langages prototypiques offrent deux manières pour créer des objets. Dans la première les objets sont créés *ex-nihilo*. Dans la seconde on utilise une forme d'héritage.

2.2.1 Création *ex-nihilo*

Un objet créé *ex-nihilo* doit être intégralement spécifié. Le modèle de données de NewtonScript définit un objet comme un ensemble de couples (nom d'attribut, valeur). Deux attributs de même nom ne sont pas autorisés dans un même objet. Les valeurs des attributs ne sont pas typées. Elles peuvent être soit *(i)* un objet primitif (entier, chaîne de caractères, …) soit *(ii)* un autre objet, soit *(iii)* une fonction.

La figure 5, ci-dessous, donne la déclarations de trois objets en NewtonScript : **`UnePlanète`**, **`UneMission`** et **`UnVaisseau`**. L'objet **`UnVaisseau`** décrit le vaisseau « Entreprise » dont l'équipage est composé de 15 membres et dont la mission est décrite par l'objet **`UneMission`**. Cette mission a pour objectif le transport de minerai et « Jim Kirk » en est l'officier responsable. La destination est la planète « Talos » dont la position est calculée en fonction de la date.

```
UnePlanète := {nom: "Talos",
               position: func(date) begin ... /*code*/ ... end };
UneMission := {destination: UnePlanète, objectif: TransportMinerai,
               officier: "Jim Kirk"};
UnVaisseau := {nom: "Entreprise", mission: UneMission,
               équipage: 15};
```

Figure 5 : Exemple de création *ex-nihilo* d'objets en NewtonScript

L'envoi de messages en NewtonScript est réalisé par l'opérateur « . ». Ainsi, le chemin **UnVaisseau.mission.destination.position(20/07/2969)** retourne la position de la planète Talos à la date spécifiée.

Un objet peut subir trois types de modifications :

• *Mise à jour* de la valeur d'un attribut. Par exemple, l'instruction **UnVaisseau.équipage:="Expérimenté"** modifie la valeur de l'attribut **équipage** de **UnVaisseau**, qui contient désormais le niveau de qualification de l'équipage au lieu du nombre de ses membres. Notons que, les valeurs des attributs n'étant pas typées, elles ne sont pas tenues d'appartenir toujours au même type.
De la même manière il est possible de modifier l'attribut **position** de **UnePlanète** par l'instruction **UnePlanète.position:=func(date) begin ... /*nouveau code*/ ... end**.

• *Ajout* d'un attribut. Par exemple, l'instruction **UnVaisseau.organisme:="StarFleet"** ajoute un attribut **organisme** à **UnVaisseau** avec pour valeur « StarFleet ».

• *Suppression* d'un attribut. Par exemple, l'instruction **RemoveSlot(UnVaisseau, équipage)** supprime l'attribut **équipage** de l'objet **UnVaisseau**.

La structure libre des objets implique que l'on ne peut savoir *a priori* si un objet possède un attribut donné. L'opérateur **exists** permet de le vérifier. Par exemple, l'instruction **UnVaisseau.mission exists** renvoie la valeur booléenne **Vrai** car il existe bien un attribut **mission** dans l'objet **UnVaisseau**.

2.2.2 Prototypes et clones

La deuxième solution pour créer des objets dans un langage prototypique consiste à utiliser des objets prototypes. Tout objet peut être un prototype. Un objet est un prototype s'il sert de modèle pour la construction d'autres objets. Les objets (aussi appelés instances) ainsi obtenus peuvent à leur tour jouer le rôle de prototypes.

Le mécanisme qui permet de construire des objets à partir des prototypes est le clonage. Le clonage produit une copie en surface (*shallow copy*) du prototype. Il ne copie donc pas les objets référencés, à l'image des copy constructeurs par défaut du langage C++. En ce sens, le clone *hérite* du prototype.

Un objet dans le rôle de prototype se comporte, dans une certaine mesures comme une classe pour l'objet cloné et comme une super-classe pour les prototypes clonés à partir d'elle.

L'instruction NewtonScript suivante crée un clone de l'objet **UnePlanète** noté **UneAutrePlanète** : **UneAutrePlanète:=clone(UnePlanète)**. Ici, **UnePlanète** joue le rôle du prototype et **UneAutrePlanète** celui de l'objet obtenu par clonage.

Le clonage n'aurait pas beaucoup d'intérêt sans la possibilité de modifier les clones pour qu'ils diffèrent de leurs prototypes. La modification la plus simple est bien sûr celle de la valeur d'un attribut. Par exemple, **UneAutrePlanète.nom:=NimbusIII**. Plus généralement, toutes les modifications sur un objet évoquées en 1.3.1 sont possibles (mise à jour, ajout et suppression d'attributs).

Une fois le clonage effectué, le clone et le prototype évoluent indépendamment. Une modification sur le prototype ne modifie pas le clone et réciproquement.

Le clonage est une caractéristique essentielle des langages prototypiques. Certains langages en proposent des variantes. C'est le cas de NewtonScript. Ainsi, il est possible de définir dans ce langage des clones qui demeurent liés au prototype par l'utilisation de l'attribut réservé **_proto**. Considérons, par exemple, les instructions en NewtonScript reprises dans la figure 6 ci-dessous.

```
(1) UnAutreVaisseau := {_proto: UnVaisseau} ;
(2) UnVaisseau.nom := "Discovery" ;
(3) UnAutreVaisseau.vitesse := 15000 ;
```

Figure 6 : Clonage par l'attribut réservé _proto

L'instruction (1) crée un clone **UnAutreVaisseau** de **UnVaisseau**. A l'issue de l'exécution de cette instruction, la valeur de l'attribut **nom** de **UnAutreVaisseau** est Entreprise. Mais à l'issue de l'exécution de l'instruction (2) sa valeur est Discovery, alors que si **UnAutreVaisseau** avait été crée par **UnAutreVaisseau:= clone(UnVaisseau)** elle aurait continué d'être Entreprise. En revanche, l'instruction (3) ajoute un attribut **vitesse** à **UnAutreVaisseau** mais pas à **UnVaisseau**.

Les langages prototypiques présentent un certain nombre de caractéristiques avancées que l'on présente maintenant.

2.3 Caractéristiques avancées

2.3.1 Modification dynamique du comportement des objets

On illustre cette caractéristique sur un exemple inspiré de (Abadi, Cardelli, 1996, p 36).

Soit un objet **o1** contenant un attribut **contenu**. On souhaite maintenir une copie de la dernière valeur de cet attribut. Une solution classique est d'ajouter un attribut **précédent** dans **o1** ainsi qu'une fonction **set** qui, avant de mettre à jour cet attribut, copie son ancienne valeur dans l'attribut **précédent**.

On présente maintenant une solution différente, qui utilise la modification dynamique du comportement des objets. Il s'agit là d'une caractéristique avancée des langages prototypiques. La figure 7 ci-dessous donne la description en NewtonScript de l'objet **o1** tel qu'on le déclare alors.

```
(1) o1 :=  {contenu: 0};
(2) o1.get :=  func() return contenu;
(3) o1.set :=  func(value) begin
(4)                x:=self:get();
(5)                self.restaurer:=func() self.contenu:=x;
(6)                contenu:=value;
(7)                end;
(8) o1.restaurer := func() self.contenu:=0;
```

Figure 7 : Exemple de modification dynamique d'objet

L'instruction (1) crée l'objet **o1** avec un attribut **contenu**. L'instruction (2), quant à elle définit une fonction **get** pour **o1** qui retourne la valeur courante de l'attribut **contenu**. La fonction **set** définie de (3) à (7) est la fonction de mise à jour de l'objet **o1**. Elle copie tout d'abord la valeur actuelle de l'attribut **contenu** de **o1** dans la variable[36] **x** par l'instruction (4), puis modifie la fonction **restaurer** de l'objet **self**[37], c'est à dire **o1**, pour qu'elle retourne désormais **x** comme valeur. Ainsi, à chaque modification de l'attribut **contenu**, la fonction **set** modifie la fonction **restaurer** de sorte qu'elle

[36] En NewtonScript les variables n'ont pas à être déclarées avant leur utilisation.

[37] Le mot réservé **self**, similaire au **this** du C++, désigne l'objet courant qui est ici **o1**.

retourne désormais la valeur de l'attribut **contenu** qu'elle écrase. La fonction **set** effectue donc une modification dynamique du comportement de **o1**. L'instruction (6) effectue la mise à jour proprement dite et l'instruction (8) définit la fonction **restaurer** initiale.

2.3.2 Création de « classes »

Il est possible, et parfois souhaitable, de simuler des classes dans les langages prototypiques pour bénéficier des avantages généralement attachés à cette notion (principalement, le partage de structures et de comportements similaires). Dans la figure 8 ci-dessous on simule une classe **Personne** en NewtonScript.

```
Personne := {
   new: func()
      begin
         x:=clone(self);
         return x;
      end,
   nom: NIL,
   âge: NIL,
   salaire: func() âge*100  } ;
```

Figure 8 : Une classe Personne en NewtonScript

La « classe » **Personne** contient deux attributs **nom** et **âge**, et deux méthodes **salaire** et **new**. La méthode **new** permet de créer une nouvelle instance de la « classe » **Personne** par l'instruction **p=Personne.new()**, à comparer avec l'écriture en langage C++, **Personne *p= new Personne**.

2.3.3 Encapsulation

L'encapsulation est rarement implantée dans les langages prototypiques. Cette fonctionnalité ne s'oppose cependant pas aux concepts qu'ils mettent en jeu ou à leur esprit. Self devrait recevoir cette fonctionnalité dans l'avenir. NewtonScript ne l'implémente pas.

2.3.4 Caractéristiques avancées de NewtonScript

NewtonScript présente un certain nombre de fonctionnalités avancées, en plus des caractéristiques fondamentales des langages prototypiques. On ne fait ici que les citer : mécanisme d'exception, variables locales et globales, itérateurs sur les attributs des objets, fonctions globales, possibilité de rediriger toutes les références faites sur un objet vers un autre, manipulation de tableaux d'objets, paramétrage des chemins (`o1.(attribut)` désigne un attribut différent de `o1` selon la valeur de `attribut`), …

2.4 Utilisation des langages prototypiques

NewtonScript est l'un des seuls langages prototypiques véritablement utilisés pour le développement de logiciels. Les autres demeurent des langages de recherche. Deux raisons principales expliquent le fait que les langages prototypiques soient si peu répandus :

- Ils sont moins performants que les langages à base de classes. Selon (Smith, 1995) l'amélioration des performances est possible mais constitue un domaine de recherche. Il note à ce titre que les principales connaissances sur l'optimisation dans les langages à base de classes ne sont pas utilisables pour les langages prototypiques. Par ailleurs, en l'absence de classes, les optimisations sont plus complexes.

- Ils ont été présentés de manière très technique et essentiellement sous leurs aspects les plus avancés (voir § 2.3), en laissant de côté leurs autres caractéristiques pertinentes (Smith, 1987).

2.5 Le gestionnaire d'objets persistants NewtonStore

Le langage prototypique NewtonScript est interfacé avec le gestionnaire d'objets persistants NewtonStore (Apple Corp., 1996 (3), chapitre 11) (Smith, 1994). Il ne gère pas les accès concurrents, mais permet la programmation de séquences d'opérations atomiques (transactions).

Un objet NewtonScript est rendu persistant en le plaçant dans une « soupe ». Pour le programmeur, une soupe est simplement un label qui désigne un ensemble d'objets persistants. Le fait de placer un objet dans une soupe y place aussi tous les objets qu'il référence et ainsi de suite. Un même objet peut être placé dans plusieurs soupes, mais conserve bien sûr le même Oid.

Il est possible de copier un objet d'une soupe à une autre et de faire des requêtes sur les soupes pour récupérer des objets selon des critères donnés. Les requêtes renvoient des curseurs sur les objets satisfaisant leurs critères. Les requêtes ne retournent que des

objets existant déjà. Elles ne créent pas d'objets nouveaux comme il est possible de le faire en OQL (ODMG, 1993).

La programmation avec NewtonStore est tout à fait similaire à celle d'une base de données objet comme ObjectStore ou Poet.

Le fragment de code de la figure 9 ci-dessous présente des exemples d'utilisation de NewtonStore. Ils correspondent à des opérations classiques lorsque l'on programme avec des objets persistants.

```
(1) MaSoupe :=  GetUnionSoupAlways("SoupeEssai");
(2) objet := {nom: Kirk, prénom: Jim};
(3) MaSoupe.AddToDefaultStoreXmit(objet);
(4) curseur :=  MaSoupe.Query(
     { ValidTest: func(entry)
                  begin
                    return test(entry);
                  end; } );
(5) MapCursor(curseur,print(entry));
(6) curseur2 :=  MaSoupe.Query({words:["Spock"]});
(7) MapCursor(curseur2,func(entry)
                    begin
                      mise_à_jour(entry,paramètre);
                    end;  );
(8) MapCursor(curseur,Store("SoupeEssai"));
```

Figure 9 : Exemple d'utilisation de NewtonStore

La commande GetUnionSoupAlways de l'instruction (1) crée un objet particulier **MaSoupe** capable d'accéder, via des fonctions prédéfinies, aux objets de la soupe **SoupeEssai**. Cet objet est semblable à une variable FILE* du langage C dans le sens où il ne contient pas les données, mais permet d'y accéder. L'instruction (2) crée, alors un objet **objet** que l'instruction (3) sauve dans la soupe **SoupeEssai**. L'instruction (4) est une requête. Elle renvoie dans le curseur **curseur** tous les objets de la soupe **SoupeEssai** pour lesquels la fonction **test** s'évalue à **Vrai**. Ces objets sont affichés à l'aide de la commande **print** qui est appliquée sur chacun d'eux par la commande **MapCursor** de l'instruction (5). La requête de l'instruction (6) renvoie dans le curseur **curseur2** tous les objets qui contiennent le mot Spock, tous attributs confondus (hors fonctions et références à d'autres objets). L'instruction (7) exécute la commande **mise_à_jour** sur tous les objets de **curseur2**. Cette commande prend un paramètre

paramètre. Enfin, l'instruction (8) rend persistantes dans la soupe **SoupeEssai** les mises à jour faites sur les objets de **curseur**.

Après avoir présenté les langages prototypiques et en particulier le langage NewtonScript, on traite de l'implémentation du classement historique comme une extension à ce langage.

3. Implémentation du classement d'objets

Comme on l'a vu dans le paragraphe précédent, les objets des langages prototypiques, et en particulier ceux de NewtonScript, n'appartiennent pas à des classes. La méthode de classement présentée au chapitre précédent peut être utilisée pour affecter ces objets à des classes définies par des assertions historiques et restaurer ainsi, dans une certaine mesure, la notion de classe dans le langage NewtonScript.

Les classes sont cependant, dans ce cas, affectées *a posteriori* aux objets alors qu'elles le sont *a priori* dans les langages à base de classes et les systèmes de gestion de bases de données objets « classiques ».

La description de ce classement *a posteriori* d'objets du langage NewtonScript fait l'objet de ce paragraphe. Pour simplifier, on suppose que les objets considérés possèdent dans chacun de leurs états tous les attributs intervenant dans le classement. On s'interdit donc de supprimer des attributs classificateurs. On rappelle, voir chapitre 9, § 2.2, qu'un attribut est classificateur s'il intervient dans au moins une assertion de classement.

3.1 Un exemple de classement d'objets

Le classement des objets est réalisé par une fonction, notée **classement**, qui prend en argument l'objet à classer. Avant de détailler son implémentation, on présente dans ce sous-paragraphe un exemple d'interaction avec l'interpréteur NewtonScript augmenté de cette fonction.

Pour cet exemple, on définit la classe d'objets **Monstre**. Un objet appartient à cette classe s'il vérifie l'assertion historique (10.2), voir chapitre 10, § 2.2. Dans la suite, on appelle schéma de classement l'ensemble des classes selon lesquelles on classe les objets. Ici, le schéma de classement n'en contient qu'une.

La figure 10 donne l'interaction, avec l'interpréteur Newtonscript, commenté ci-après. Les réponses de l'interpréteur sont en italiques.

```
(1)  o1:={taille: 15, poids: 420, lieu: "LochNess"};
                {taille: 15, poids: 420, lieu: "LochNess"}

(2)  classement(o1);
                NIL

(3)  o1.classe();
                ["Pas de classe"]

(4)  o1.taille:=20;
                20

(5)  classement(o1);
                NIL

(6)  o1.classe();
                ["Pas de classe"]

(7)  o1.poids:=410;
                410

(8)  classement(o1);
                NIL

(9)  o1.classe();
                ["Monstre"]
```

Figure 10 : Un exemple de classement d'objets du langage NewtonScript

L'instruction (1) définit un objet **o1**. L'instruction (2) classe cet objet en utilisant la fonction **classement**. L'instruction (3) exécute la fonction **classe()** de l'objet **o1**. Cette fonction, ajoutée à l'objet **o1** par la fonction **classement**, retourne les classes auxquelles appartient l'objet. L'objet **o1** ne vérifiant pas la sous-formule $\Diamond$**taille**$\geq$**20** de la formule (10.2), il n'appartient pas à la vue **Monstre** et donc à aucune classe.

L'instruction (4) modifie à 20 la valeur de l'attribut **taille** de l'objet **o1**. Après classement par l'instruction (5), l'exécution de l'instruction (6) confirme que l'objet **o1** n'appartient toujours pas à la classe **Monstre**. En effet, comme on retient une sémantique non réflexive pour l'opérateur $\Diamond$, voir chapitre 10, § 2.2, celui-ci ne porte que sur les états passés et pas sur l'état présent. Ici, l'objet **o1** vérifie bien **taille**$\geq$**20**, mais pas $\Diamond$**taille**$\geq$**20**.

L'instruction (7) effectue alors une modification du poids, qui passe à 410. De nouveau, l'objet **o1** est classé, par l'instruction (8). Cette fois, l'objet vérifie $\Diamond$**taille**$\geq$**20** et $\Box$**(**$\Diamond$**taille**$\geq$**20**$\rightarrow$**poids**$>$**400)** et donc la formule (10.2). Par

conséquent, l'objet **o1** appartient maintenant à la classe **Monstre**, comme le confirme l'exécution de l'instruction (9).

On présente maintenant la programmation de la fonction **classement**.

3.2 La fonction de classement

La fonction **classement** est programmée en NewtonScript. Elle réalise le classement de l'objet **oc** passé en paramètre, en trois étapes successives.

Etape 1 – La première étape consiste à ajouter à l'objet **oc**, lors de son premier classement, (*i*) un attribut **listeclasses** dont la valeur sera la liste des classes auxquelles il appartient, (*ii*) la fonction **classe()**, utilisée dans l'exemple du paragraphe précédent et qui retourne le contenu de l'attribut **listeclasses**, ainsi que (*iii*) d'autres attributs nécessaires pour le monitoring (voir § 3.4). Ces composantes supplémentaires « polluent » l'objet **oc**. Pour éviter cela, la solution retenue consiste à créer un objet **global** qui factorise ces composantes. Lors du premier classement de chaque objet **oc**, un clone de l'objet **global** est alors créé puis attaché à **oc** par l'attribut réservé **_parent**. Par exemple, dans le cas de l'objet **o1** du paragraphe précédent, on obtient **o1:={_parent:CloneDeGlobal, taille:15, poids:420, lieu:"LochNess"}**.

L'utilisation de l'attribut réservé du langage **_parent** permet d'invoquer la méthode **classes** et de modifier la valeur des attributs du clone comme s'ils appartenaient à **oc** lui-même. Par exemple, l'instruction **oc.listeclasses:=["UnNomDeClasse", "UnAutreNomDeClasse"]** modifie la valeur de l'attribut **listeclasses** et **oc.classes** renvoie la valeur de cet attribut. Ainsi, l'utilisation de l'attribut **_parent** permet d'ajouter à l'objet **oc** de nouvelles caractéristiques sans le polluer de composantes supplémentaires

Etape 2 – Cette étape réalise le monitoring de l'objet **oc**. Elle positionne donc cet objet dans les différents automates d'états finis en fonction de sa position courante et des valeurs de ses attributs, comme on l'a vu au chapitre 10, § 3.3. La génération et la représentation des automates sont décrites au § 3.3. La mise en œuvre du monitoring est exposée au § 3.4.

Etape 3 – La dernière étape détermine les classes auxquelles appartient l'objet **oc** en fonction des assertions historiques qui s'évaluent à vrai. Un objet relève d'une classe s'il est positionné dans un état final de l'automate correspondant à la classe.

3.3 Génération des automates à partir des assertions historiques

La génération des automates utilisés pour vérifier les assertions historiques est réalisée selon trois étapes successives:

• chaque formule du schéma de classement est tout d'abord traduite en une expression régulière selon les règles de composition données au chapitre 10, § 3.2, au moyen d'un programme C++. Ce dernier permet la saisie de l'assertion historique en logique temporelle, moyennant quelques conventions simples : ◊ est noté **<>**, ∧ est noté **&**, ¬ est noté **-** et enfin ∨ est noté **+**. Par exemple, pour l'assertion historique (10.2) la chaîne de caractères passée en entrée du programme est

<>taille>=20 & -<>-(-<>taille>=20 + poids>400).

Le programme utilise une notation simplifiée pour les vecteurs de vérité : a=(**taille≥20,poids>400**), b=(**taille≥20,¬ poids>400**), c=(**¬taille≥20, poids>400**) et d=(**¬taille≥20**, **¬ poids>400**). Il fournit l'expression régulière :

[[[[a|b|c|d]* [a|b]] [a|b|c|d]+] & [[a|b|c|d]+ - [[[a|b|c|d]+ - [[[[a|b|c|d]+ - [[[a|b|c|d]* [a|b]] [a|b|c|d]+]]] | [[[a|b|c|d]* [a|c]]]]] [a|b|c|d]+]]]

• l'automate correspondant à cette expression régulière est obtenu en utilisant le Web de la société Rank Xerox. Ce Web propose, en effet, une page[38] où il est possible de saisir une expression régulière et d'obtenir en retour l'automate déterministe et minimal équivalent. Il nous a été très souvent utile. En effet, le calcul « à la main » de l'automate d'une formule temporelle est pratiquement infaisable passé une certaine taille. Nous tenons à souligner combien l'outil proposé est commode, utile et puissant.

Dans le cas de l'expression régulière ci-dessus, l'automate fourni est reproduit figure 11 ci-dessous (voir son dessin, chapitre 10, § 3.3).

```
4 states, 12 arcs, Circular.
Sigma: a b c d

s0:   a -> s1, b -> s1, c -> s0, d -> s0.
s1:   a -> fs2, b -> fs3, c -> fs2, d -> fs3.
fs2:  a -> fs2, b -> fs3, c -> fs2, d -> fs3.
fs3:  (no arcs)
```

Figure 11 : Automate obtenu pour (10.2) sur le Web de Rank Xerox

[38] Xerox Finite-State Compiler : http://www.rxrc.xerox.com/research/mltt/fst/fsinput.html

• enfin, l'automate est implanté en NewtonScript sous forme d'un tableau de triplets (D,C,A) qui décrivent chacun une transition. D est le numéro du noeud de départ, C la condition de l'arc exprimée sous forme d'une fonction renvoyant un booléen et A le numéro du noeud d'arrivée. Une structure annexe liste l'ensemble des noeuds finaux de l'automate.

Les automates sont stockés dans un tableau **Automate**. La figure 12 ci-dessous donne un fragment de la traduction en NewtonScript de l'automate de la figure 3 du chapitre 10. On rappelle que cet automate correspond à l'assertion historique (10.2) qui définit la classe **Monstre**.

```
Automate[1] := [
[0,func(objet) return NOT(objet.taille>=20) AND objet.poids>40,0],
[0,func(objet)
       return NOT(objet.taille>=20) AND NOT(objet.poids>40),0],
[0,func(objet) return objet.taille>=20 AND objet.poids>40,1],
[1,func(objet) return objet.taille>=20 AND NOT(objet.poids>40),1],
[1,func(objet) return objet.taille>=20 AND objet.poids>40,2],
..... ] ;
```

Figure 12 : Représentation d'un automate d'états finis en NewtonScript

3.4 Monitoring des objets classés

Le monitoring des objets classés requiert que soit stockée pour chaque objet sa position courante dans chaque automate. Pour cela, l'objet **global** est créé avec un attribut supplémentaire par automate. Chaque objet classé étant associé à un clone de l'objet **global**, il possède donc indirectement ces attributs. Ils stockent la position de l'objet dans les différents automates par un entier égal au numéro du noeud auquel a abouti l'objet dans l'automate. Lors du premier classement, il s'agit du numéro de l'état initial de l'automate.

Le monitoring d'un objet, pour un automate donné, est alors effectué par la fonction **classement** en déterminant le triplet (D,C,A) de la représentation en NewtonScript de l'automate telle que : (*i*) D est la position courante de l'objet et (*ii*) C s'évalue à vrai. Les automates étant déterministes, un seul triplet (D,C,A) est sélectionné. A désigne alors la nouvelle position de l'objet dans l'automate et est stocké comme tel dans l'objet en remplacement de l'ancienne position.

On souligne que l'évaluation des conditions C se réalise simplement en NewtonScript par l'utilisation de la fonction **`apply`** du langage. Par exemple, **`apply(Automate[1][1][2],oc)`** évalue la condition du premier triplet (D,C,A) de l'automate de la figure 12 pour l'objet **`oc`**.

Conclusions et Perspectives

Le système Osiris classe les objets selon leur état courant. Il est parfois nécessaire de les classer selon leur histoire. Dans cette partie, on a présenté une technique pour réaliser ce type de classement. Elle repose sur une méthode d'évaluation des assertions historiques dont le coût est faible et constant, quelque soit la « taille » de l'histoire. Cette méthode utilise une traduction de ces assertions en expressions régulières. Une assertion historique s'évalue alors à vrai pour un objet donné si son histoire est un mot du langage décrit par l'expression régulière correspondant à l'assertion.

L'expression des assertions historiques est réalisée en logique temporelle. Ce choix s'est imposé par son utilisation dans d'autres domaines des bases de données. Il conviendrait pourtant de s'interroger sur sa pertinence dans le cadre où on l'applique. En effet, certaines propriétés sur l'histoire des objets ne peuvent s'exprimer dans ce formalisme. C'est le cas, par exemple, de propriétés qui doivent être périodiquement vraies (Abiteboul, Herr, Van den Bussche, 1997). Il conviendrait aussi de s'interroger sur la notion d'état d'un objet. Quand considère-t-on qu'une modification sur un objet crée un nouvel état ? En particulier, la mise à jour d'un attribut d'un objet à sa valeur courante crée-t-elle un nouvel état de l'objet ?

L'implémentation de la méthode de classement d'objets selon leur histoire a été réalisée sur le langage prototypique NewtonScript. Les objets de ce langage n'étant pas des instances de classes, ils ne sont pas classés *a priori*. La méthode de classement a été utilisée pour les classer *a posteriori*, selon leur histoire.

Le paradigme qui consiste à classer les objets *a posteriori* selon des assertions, qu'elles soient historiques ou non, n'est pas habituel dans les bases de données. Dans celles-ci, les objets sont des instances de classes et donc classés *a priori*. Le paradigme de classement *a posteriori* nous semble constituer un élément de réponse à la manipulation de certaines formes de données semi-structurées, comme évoqué dans (Delannoy, 1997). La prise en compte de ce type de données fait l'objet d'une nouvelle direction de recherche dans les bases de données. Une première bibliographie sur ce

domaine est : (Abiteboul, 1997), (Buneman, 1997), (Fernandez, Popa, Suciu, 1997), (Buneman, Davidson, Suciu, 1996) et (Quass et *al.*, 1995).

Ces différents travaux s'intéressent à l'interrogation d'ensembles de données semi-structurées. L'expérience acquise sur les langages prototypiques et le classement *a posteriori* d'objets de ces langages, nous conduit à penser qu'il y a là les fondements d'un langage de programmation pour ce type de données.

En guise de Conclusion Générale ...

Cette thèse est faite de deux contributions. Toutes deux sont liées aux contraintes d'intégrité. On se sera toutefois rendu compte qu'elles s'intéressent à des sujets très différents puisque la première traite de sécurité et de fraude tandis que la deuxième traite du classement d'objets selon leur histoire. Chaque contribution s'est naturellement terminée par une conclusion et l'exposé de perspectives. Dès lors, l'exercice d'une conclusion générale se révèle un peu délicat car, s'il ne veut pas reprendre ce qui a déjà été énoncé dans les conclusions spécifiques, il se doit de recouvrir, par une généralité plus grande, les deux contributions. Autrement dit, écrire une autre thèse, par exemple de nature épistémologique. Aussi, plus modestement, on se limitera à présenter deux constatations générales qui s'imposent d'évidence à la fin de ce travail.

La première est celle de l'ouverture de la recherche en bases de données à d'autres domaines dont elle utilise les ressources et les résultats. Les deux contributions de cette thèse font ainsi appel à des résultats en sécurité (canal caché), en logique (logique temporelle), en programmation (langages prototypiques), en théorie des langages (expressions régulières, automates), et en mathématiques (correspondances et treillis de Galois).

La seconde est celle du rôle de l'approche formelle. Elle permet de révéler les composantes essentielles des questions traitées, de cerner les incompréhensions, de progresser dans la compréhension de ces questions et aussi de faire surgir de nouvelles interrogations. En particulier, dans l'étude de la tension elle a conduit à l'identification des correspondances manichéistiques et dans l'étude du classement d'objets selon leur histoire elle a permis d'établir le lien avec les expressions régulières.

Ces deux constatations pourraient être faites à propos de la programmation. Sa pratique et sa théorie ont abondamment utilisé les apports d'autres disciplines et ont conduit à une abondante littérature formelle. On fait remarquer que se trouve ainsi renforcé et, en quelque sorte conforté, le lien entre la programmation et les bases de données que la première phrase de cette thèse visait à poser d'emblée en définissant, à la suite d'Ullman, les systèmes de gestion de bases de données comme des outils de programmation de mémoires persistantes

ANNEXE A

La Méthode des Traqueurs

Les bases de données statistiques permettent d'effectuer des requêtes statistiques (moyenne, écart-type, …) sur les données stockées. Une méthode de fraude dans ce type de bases de données est celle des *traqueurs* (Schlörer, 1980).

Soit BDS une base de données statistiques et **D** une donnée de cette base qui soit la seule à vérifier la propriété **C**. Il en découle que l'exécution de la requête statistique **count(C)** sur BDS retourne 1. La connaissance de cette information permet, même si l'on a pas accès à **D**, de savoir si la donnée **D** vérifie aussi la propriété **X**. En effet, si la requête **count(C AND X)** retourne 1 alors nécessairement **D** vérifie la propriété **X**. Si, au contraire, elle retourne 0, alors elle ne la vérifie pas.

Une solution élémentaire pour empêcher ce type de fraude est d'interdire la réponse aux requêtes dont le résultat est inférieur à une certaine valeur entière positive **k**, appelée facteur de requête. Par exemple, **k**=5.

Les traqueurs permettent de contourner cette mesure. Reprenons l'exemple précédent. Du fait du facteur de requête, il n'est plus possible d'obtenir le résultat de la requête **count(C AND X)**. Supposons alors que l'on arrive à décomposer la condition **C AND X** sous la forme **A AND B** telle que **count(A)**$\geq$**k** et **count(A AND (NOT B))**$\geq$**k**. **A AND (NOT B)** est le traqueur de **C AND X** pour l'état courant de BDS. On a alors **count(C AND X)=count(A)-count(A AND (NOT B))**. Or, les deux termes de la soustraction sont calculables puisque chacun donne un résultat supérieur au facteur de requête **k**. On peut donc calculer **count(C AND X)** et donc déterminer si **D** possède la propriété **X**.

Annexe B

Modèles de Sécurité Multi-niveaux

Les modèles de sécurité *multi-niveaux* (multilevel models) permettent d'éviter certains flux d'informations illicites. Ils comblent là une faiblesse reconnue des modèles de sécurité discrétionnaires. Il existe de nombreux modèles de sécurité multi-niveaux. La plupart sont inspirés du modèle Bell & Lapaluda (Bell, Lapaluda, 1975), lui-même issu des règles de confidentialité mises en œuvre dans les organisations militaires bien avant l'apparition de l'informatique dans ces organisations.

Les modèles multi-niveaux contrôlent l'accès aux informations par une classification des données et des sujets qui y accèdent. Les sujets peuvent être des utilisateurs physiques ou des programmes. Les différents niveaux de classification sont partiellement ordonnés. Par exemple : Public (P) et Secret (S) avec $P < S$. Les deux règles fondamentales de la sécurité multi-niveaux sont : *(i)* un sujet ne peut accéder en lecture à une donnée que si son niveau de classification est identique ou supérieur à celui de la donnée, et *(ii)* un sujet ne peut mettre à jour une donnée que si son niveau de classification est identique ou inférieur à celui de la donnée.

Bibliographie

Abadi, M., Cardelli, L., *A Theory of Objects*, Monographs in Computer Science, Springer Verlag, 1996.

Abiteboul, S., *Querying Semi-Structured Data*, Proc. Int. Conf. ICDT, 1997.

Abiteboul, S., Herr, L., Van den Bussche, J., *Temporal versus First-Order Logic to Query Temporal Databases*, Int. Conf. Principles of Database Systems, 1996.

Abiteboul, S., Vianu, V., *Transactions and Integrity Constraints*, ACM SIGMOD Symp. on Principles of Database Systems, 1985.

Aho, A., Ullman, J., *The Theory of Parsing, Translation and Compiling*, *Volume 1 : Parsing*, Prentice Hall series in Automatic Computation, 1972.

Aigner, M., *Combinatorial Theory*, Grundlehren der mathematischen Wissenschaften 234, Springer Verlag, 1979.

Aiken, A., Widom, J., Hellerstein, J., *Behaviour of Database Production Rules : Termination, Confluence, and Observable Determinism*, Proc. Int. Conf. ACM SIGMOD, pp 59-68, 1992.

Albano, A., Diotallevi, M., Ghelli, G., *Extensible Objects for Database Evolution : Language Features and Implementation Issues*, Proc. Int. Work. on Database Programming Languages, pp 203-223, Septembre, 1995.

Apple Corp. , *The NewtonScript Programming Language (version 2)*, Newton System Group, 1996. (1)

Apple Corp. , *Newton Tool Kit User Guide*, Newton System Group, 1996. (2)

Apple Corp. , *Newton's Programmers Guide : System Software*, Newton System Group, 1996. (3)

Aristote (Adiba, M., Collet, C., Coupaye, T., Habraken, P., Machado, J., Martin, H. , Roncancio, C.,), *Triggers Systems: Different Approaches*, SUR007, Institut IMAG - Grenoble, Juin, 1993.

Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G. R., Traiger, I.L., Wade, B.W., Watson, V., *System R: Relational Approach to Database Management*, ACM transactions on database systems, Vol. 1, No. 2, Juin, 1976.

Audureau, E., Enjalbert, P., Farinas del Cerro, L., *Logique Temporelle : sémantique et validation de programmes parallèles*, collection études et recherches en informatique, Masson, 1990.

Bancilhon, F., Spyratos, N., *Protection of Information in Relational Databases*, Proc. Int. Conf. VLDB, 1977.

Barbut, M., Monjardet, B., *Algèbre et Combinatoire*, Paris, Hachette, 1970.

Bassolet, C-G., Simonet, A., Simonet, M., *Probabilistic Classification in Osiris, a View -based OO DBMS and KBMS*, Proc. Int. Work. on Database and Expert Systems Applications, Septembre, 1996.

Bell, D., Lapaluda, L., *Secure Computer Systems : Unified Exposition and Multics Interpretation*, Technical Report, MTR-2997, MITRE, Bedford, Massachusetts, 1975.

Berge, C., *Espaces Topologiques, Fonctions Multivoques*, Dunod, 1959.

Bernstein, P., Blaustein, B., Clarke, E.M., *Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data*, Proc. Int. Conf. VLDB, 1980.

Bertino, E., Weigand, H., *An Approach to Authorisation Modelling in Object Oriented Database Systems*, Data & Knowledge Engineering, Vol. 12, No. 1, Février, 1994.

Bourbaki, N., *Théorie des Ensembles*. Chap. I et II, Actualités scientifiques et industrielles, 1212, Herman, 1954.

Bry, F., Decker, H., Manthey, R., *A Uniform Approach to Constraint Satisfiability in Deductive Databases*, Proc. Int. Conf. on Extending Database Technology, 1988.

Bry, F., Manthey, R., *Checking Consistency of Database Constraints : a Logical Basis*, Proc. Int. Conf.VLDB, 1986.

Buneman, P., *Semistructured Data*, Int. Conf. On Principles of Database Systems, Invited tutorial, 1997. (*http://www.cis.upenn.edu/~db*)

Buneman, P., Davidson, S.B., Suciu, D., *Programming Constructs for Unstrucured Data*, Int. Work. on Database Programming Languages, 1996.

Bussolati, U., Fugini, M.G., Martella, G., *A Conceptual Framework for Security System Design*, Proc. IFIP World Conf., Septembre, 1983.

Casanova, M., Tucherman, L., Furtado, A.L., *Enforcing Inclusion Dependencies and Referential Integrity*, Proc. Int. Conf. VLDB, 1988.

Castano, S., Fugini, M., Giancarlo, M., Pierangela, S., *Database Security*, Addison Wesley, 1994.

Ceri, S., Widom, J., *Deriving Production Rules for Constraint Maintenance*, Proc. Int. Conf. VLDB, 1990.

Chomicki, J., *History-less Checking of Dynamic Integrity Constraints,* Proc. Int. Conf. on Data Engineering, pp 557-564, Février, 1992.

Chomicki, J., *History-less Checking of Temporal Integrity Constraints*, non publié, 1994.

Chomicki, J., Niwinski, D., *On the Feasibility of Checking Temporal Integrity Constraints*, Jour. of Computer and System Sciences, 1995.

Collet, C., *Bases de Données Actives : des Systèmes Relationnels aux Systèmes à Objets*, Thèse d'Habilitation à Diriger les Recherches, Laboratoire Logiciels, Systèmes et Réseaux – IMAG, 1996.

Courrège, P., *Un Modèle Mathématique des Structures Elémentaires de Parenté*, L'Homme, Vol. 5, pp 249-290, 1965.

Cremers, A.B., Domann, G., *AIM-An Integrity Monitor for the Database System Ingres*, Proc. Int. Conf. VLDB, 1983.

Date, C. J., *An Introduction to Database Systems*, Vol. 1, Addison-Wesley, 1990.

Date, C.J., *Referential Integrity*, Proc. Int. Conf. VLDB, 1981.

Dayal, U., Hanson, E., Widom, J., *Active Database systems*, dans Modern Database systems: The Object Model, Interoperability, and Beyond, édité par Won Kim, Addison Wesley, Reading, Massachusetts, Septembre, 1994.

Delannoy, X., *Understanding the Tension Between Transition Rules and Confidentiality*, Proc. British Nat. Conf. On Databases (BNCOD), Edimbourg, Lecture Notes in Computer Science 1094, Springer Verlag, pp 92-106, Juillet, 1996. *(http://curie.imag.fr/~delannoy /perso/publication.html)*

Delannoy, X., *Typing Persistent Objects a posteriori to Manage Semi-Structured Data*, Exposé donné à Ibex Object Systems Corp. (http://www.ibex.ch), Janvier, 1997.

Delannoy, X., Del Vigna, C., *Binary Integrity Constraints against Confidentiality*, Proc. Int. Conf. on Database and Expert Systems Applications (DEXA), Lecture Notes in Computer Science 1134, Springer Verlag, pp 264-275, Septembre, 1996. *(http://curie.imag.fr/~delannoy/perso/publication.html)*

Delannoy, X., Del Vigna, C., *Secret Data Exactly Unveiled with 2 - ary Constraints : the Manicheistic Situations*, Int. Work. on Database Theory, en conjonction avec la European Summer School in Logic, Language and Information (ESSLI), 1997. *(http://curie.imag.fr/~delannoy/perso/publication.html)*

Delannoy, X., Del Vigna, C., *Secrets, Unveilings and Fraud : a General Framework and the Study of a Special Case*, soumis à Int. Conf. on Data Engineering, 1998.

Delannoy, X., *La Cohérence dans les Bases de Données*, Rapport de recherche RR-936I, Université de Grenoble, IMAG-TIMC Lab., Novembre, 1994.

Delannoy, X., Simonet, A., Simonet M., *Database Views wtih Dynamic Assertions,* Proc. Int. Conf. on Object Oriented Information Systems (OOIS), Londres, Springer Verlag, Décembre, 1996.

Delobel, C., Adiba, M., *Bases de Données et Systèmes Relationnels*, Dunod, 1982.

Delobel, C., *Contributions théoriques à la conception d'un système d'information*, Thèse de Doctorat d'Etat, Université de Grenoble, 1973.

Delobel, C., Lécluse, C., Richard, P., *Bases de Données: des Systèmes Relationnels aux Systèmes à Objets*, InterEditions, 1991.

Denning, D., Denning, P., *Certification of Programs for Secure Information Flow*, Communication of the ACM, Vol. 20, No. 7, Juillet, 1977.

Dittrich, K., Gatziu, S., Geppert, A., *The Active Database Management System Manifesto: A Rule-base of ADBMS Features*, Proc. Int. Work. on Rules In Database Systems, Septembre, 1995.

DoD, Department of Defense, *US Department of Defense Trusted Computer System Evaluation Criteria*, Decembre, 1985. *Connu aussi sous le nom de « Orange Book ». (http ://www.pinsight.com :80/~royg/security/dod/orange_book/orange01.html)*

Dumas Menjívar, M., *Expression de Contraintes d'Intégrité dans un SGBD Temporel à Objets*, Rapport de DEA, Laboratoire Logiciels, Systèmes et Réseaux – IMAG, Grenoble, 1997.

Ehrich, H.D., Lipeck, U.W., Gogolla, M., *Specification, Semantics, and Enforcement of Dynamic Database Constraints,* Proc. Int. Conf. VLDB, 1984.

Elmasri, R., Navathe, S., *Fundamentals of Database Systems*, Addison Wesley, 1989.

Fagin, R., *Multivalued Dependencies and a New Normal Form for Relational Databases*, ACM Transactions on Database Systems, Vol. 2, No. 3, 1977.

Fauvet, M-C., Canavaggio, J-F., Scholl, P-C., *Tempos : un Modèle d'Historiques pour un SGBD Temporel à Objets*, Proc. Nat. Conf. Bases de Données Avancées, Grenoble, 1997.

Fernandez, M., Popa, L., Suciu, D., *A Structure-Based Approach to Querying Semi-Structured Data*, Int. Workshop on Database Programming Languages (DBPL), 1997.

Fugini, M.G., Martella, G., *ACTEN : A Conceptual Model for Security System Design*, Computers and Security, Elsevier (North Holland), Vol. 3, No. 3, 1984.

Gabbay, D., Mc Brien, P., *Temporal Logic and Historical Databases*, Proc. Int. Conf. VLDB, Septembre, 1991.

Gardarin, G., Melkanoff, M., *Proving Consistency of Database Transactions*, Proc. Int. Conf. VLDB, 1979.

Gardarin, G., Valduriez, P., *SGBD Relationnels : Analyse et comparaison des bases de données*, Eyrolles, 1989.

Gertz, M., Lipeck, U., *Deriving Integrity Maintaining Triggers from Transition Graphs*, Proc. Int. Conf.on Data Engineering, pp 22-29, 1993.

Gertz, M., Lipeck, U., *"Temporal" Integrity Constraints in Temporal Databases*. Recent Advances in Temporal Databases, Proc. Int. Work. on Temporal Databases, pp 77-92, 17-18 Septembre, 1995.

Griffiths, P., Bradford, W., *An Authorisation Mechanism for a Relational Database System*, ACM Transactions on Database Systems, Vol. 1, No. 3, pp 242-255, Septembre, 1976.

Guénoche, A., *Construction du Treillis de Galois d'une Relation Binaire*, Mathématiques, Informatique et Sciences Humaines, n° 109, pp 41-53, 1990.

Gupta, A., Widom, J., *Local Verification of Global Integrity Constraints in Distributed Databases*, Proc. Int. Conf. ACM SIGMOD, Mai, 1993.

Guttag, J., Horning, J., *The Algebraic Specification of Abstract Data Types*, Acta Informatica, 1978.

Harinarayan, V., Gupta, A., *Optimization Using Tuple Subsumption*, Proc. Int. Conf. on Database Theory (ICDT), Lecture Notes in Computer Science 893, Springer Verlag, pp 338-352, 1995.

He, J., Gligor, V., *Formal Methods and Automated Tool for Timing-Channel Identification in TCB Source Code*, Second European Symp. on Research in Computer Security, Lecture Notes of Computer Science 648, Springer Verlag, Novembre, 1992.

Hoare, J., *An Axiomatic Basis for Computer Programming*, CACM Vol. 12, No. 10, pp 576-580, Octobre, 1969.

Hopcroft, J., Ullman, J, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

Hsu, A., Imielinski, T., *Integrity Checking for Multiple Updates*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1985.

Jajodia, S., Sandhu, R., *Toward a Multilevel Secure Relational Data Model*, Proc. ACM SIGMOD Int. Conf. on Management of Data, pp 50-59, Mai, 1991.

Jarke, M., Mazumdar, S., Simon, E., Stemple, D., *Assuring Database Integrity,* Jour. of Database Administration, Vol. 1, No. 1, pp 391-400, 1990.

Jeusfeld, M., Jarke, M., *From Relational to Object Oriented Integrity Simplification*, Proc. Int. Conf. on Deductive and Object Oriented Databases, 1991.
Korth, H.F., Silberschatz, A., *Systèmes de Gestion de Bases de Données,* McGraw-Hill, 1988.

Kowalski, R., Sadri, F., Soper, P., *Integrity Checking in Deductive Databases*, Proc. Int. Conf. VLDB, 1987.

Küchenhoff, V., *On the Efficient Computation of the Difference Between Consecutive Database States*, Proc. Int. Conf. on Deductive and Object Oriented Databases (DOOD), 1991.

Kung, C., *A Temporal Framework for Database Specification and Verification*, Proc. Int. Conf. VLDB, 1984.

Lampson, B., *A Note on the Confinement Problem*, Communication of the ACM Vol. 16, No. 10, Octobre, 1973.

Landwehr, E., Bull, A., McDermott, J., Choi, W., *A Taxonomy of Computer Program Security Flaws*, ACM Computing Surveys, Vol. 26, No. 3, Septembre, 1994.

Lipeck, U., Feng, D., *Construction of Deterninistic Transition Graphs from Dynamic Integrity Constraints*, Proc. Int. Work. on Graph Theoric Concepts in Computer Science, Springer Verlag, 1988.

Lipeck, U., Saake, G., *Monitoring Dynamic Integrity Constraints Based on Temporal Logic,* Information Systems Vol.12, No.3, pp 255-269, 1987.

Loepere, K., *The Covert Channel Limiter Revisited*, Operating System Review, ACM Press, Vol. 23, No. 2, Avril, 1989.

McCune, W., Henschen, L., *Maintaining State Constraints in Relational Databases: A Proof Theoric Basis*, Jour. of the ACM, Vol. 36, No. 1, 1989.

Manna, Z., Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems - Specification*, Springer Verlag, 1991.

Mannila, H., Räihä, K., *The design of Relational Databases,* Addison Wesley, 1992.

Markowitz, V. , *Referential Integrity Revisited: An Object Oriented Approach*, Proc. Int. Conf. VLDB, 1990.

Markowitz, V., *Safe Referential Integrity Structures in Relational Databases*, Proc. Int. Conf. VLDB, 1991.

Martin, H., Adiba, M., Defude, B., *Consistency Checking in Object Oriented Databases: a Behavioral Approach*, Proc. Int. Conf. CIKM, 1992.

Mazumdar, S., Stemple, D., Shread, T., *Resolving the Tension between Integrity and Security Using a Theorem Prover*, Proc. Int. Conf. on Management of Data, Chicago, Illinois, SIGMOD Record Vol. 17, No. 3, pp 233-42, Septembre, 1988.

Medeiros, C.B., Jomier, G., Cellary, W., *Maintaining Integrity Constraints across Versions in a Database*, Universidade Estadual De Campinas, Relatorio Technico DCC-08/1992, 1992.

Medeiros, C.B., Pfeffer, P., *Object Integrity Using Rules*, Proc. Europ. Conf. on Object Oriented Programming, 1991.

Melton, J., Correspondance personnelle avec Jim Melton, Senior Architect of Standards for Sybase Corp. and Editor of the ISO SQL-92 and emerging SQL-3 standards, Decembre, 1995.

Melton, J., Simon, A., *Understanding the new SQL Standard - A complete Guide*, Morgan Kaufmann Publishers, 1993.

Morgenstern, M., *Constraint Equation: Declarative Expression of Constraints With Automatic Enforcement*, Proc. Int. Conf. VLDB, 1984.

Nicolas, J.M., *Logic for Improving Checking in Relational Databases*, Acta Informatica, Vol. 18, No. 3, 1982.

ODMG93, Cattel, R.G.G., *The Object Database Standard: ODMG93*, Morgan Kaufmann, 1993.

Olivé, A., *Integrity Constraint Checking In Deductive Databases*, Proc. Int. Conf. VLDB, 1991.

Ozsu, M. T., Valduriez, P., *Distributed Database Design : Fragmentation (Chap. 5.3)*, in Principles of Distributed Database Design, Prentice Hall, 1991.

Picouet P., Cheiney J.P., *SGBD et Triggers: produits d'aujourd'hui et de demain*, Les bases de données avancées : du modèle relationnel au modèle orienté objet, HERMES, 1992.

Pipard, E., *Détections d'Incohérences et d'Incomplétudes dans les Bases de Règles: le Système INDE*, Avignon, 1988.

Plexousakis, D., *Compilation and Simplification of Temporal Integrity Constraints*, Proc. Int. Work. on Rules in Database Systems, 1995.

Plexousakis, D., *Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases*, Proc. Int. Conf. VLDB, 1993.

Qian, X., Smith, D.R., *Integrity Constraints Reformulation for Efficient Validation*, Proc. Int. Conf. VLDB, 1987.

Qian, X., Wiederhold, G., *Knowledge Based Integrity Constraint Validation*, Proc. Int. Conf. VLDB, 1986.

Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Widom, J., *Querying Semistructured Heterogeneous Information*, Int. Conf. On Deductive and Object Oriented Databases, 1995.

Rousset, M-C., *Sur la Validité dans les Bases de Connaissances: le Systéme COVADIS*, Avignon, 1987.

Sales, A-M., *Types Abstraits et Bases de Données: Formalisation de la Notion de Partage et Analyse Statique des Contraintes d'Intégrité*, Thèse, Université Scientifique et Médicale de Grenoble, 1984.

Schlörer, J., *Disclosure from Statistical Databases : Quantitative Aspects of Trackers*, ACM Transactions on Database Systems, Vol. 6, No. 1, Mars, 1980.

Schwiderski, S., Hartmann, T., Saake, G., *Monitoring Temporal Preconditions in a Behaviour Oriented Object Model*, Data & Knowledge Engineering (14), pp 143-186, 1994.

A. Simonet, M. Simonet, *Les classes d'équivalence induites par des dépendances inter-attributs: fondements pour une représentation des objets partagés dans les bases de connaissances*, Rapport de Recherche, 769-I laboratoire Artemis, Grenoble, France, 1989.
Simonet, A., Simonet, M., *Objects with Views and Constraints : from Databases to Knowledge bases*, Proc. Int. Conf. on Object Oriented Information Systems (OOIS), 1994.

Simonet, A., Simonet, M., *OSIRIS : an Object Oriented System unifying Databases and Knowledge Bases*, KBKS 95 : Toward very Large Knowledge Bases, IOS Press, 1995.

Simonet, A., Simonet, M., *Classement d'Objets et Evaluation des Requêtes en Osiris,* 12ièmes Journées Bases de Données Avancées, Publié sous la direction de P. Bosc, Cassis, 27-30 Août, 1996.

Smith, W., *The Newton Application Architecture*, Proc. Int. Conf. IEEE Computer Conference, 1994.

Smith, W., *Using a Prototype-based Language for User Interface : The Newton Project's Experience*, Proc. Int. Conf. on Object Oriented Programming Systems, Language and Application, 1995.

Smith, R., Lentczner, M., Smith, W., Taivalsaari, A., Ungar, D., *Prototype-Based Languages : Object Lessons from Class-Free Programming*, Proc. Int. Conf. OOPSLA, 1987.

SQL *Database Language SQL (SQL3)*, ISO-ANSI Working Draft, ANSI TC X3H2, ISO/IEC JTC 1/SC 21/WG 3, Août, 1994.

SQL, *Database Language SQL*. Technical report ANSI X3.135-1986 and ISO/TC97/SC21/WG3N117, ANSI and ISO/TC97/SC21/WG3, 1986.

SQL, *Information Technology - Database Language SQL*, Technical Report, ISO/IEC/JTC1/SC21 N6789, ISO/IEC/JTC1/SC21, *IEC9075*, (and 1994 addendum), 1992.

Stanat, D., McAllister, D., *Discrete Mathematics in Computer Science*, Prentice Hall, 1977.

Stein, J., Anderson,T.L., Maier, D., *Mistaking Identity*, Proc. Int. Work. on Database Programming Languages, 1989.

Stein, L., Lieberman, H., Ungar, D., *A shared View of Sharing : the Treaty of Orlando*, in Object Oriented Concepts, Applications and Databases, W. Kim and F. Lochowsky, eds, Addison Wesley, 1988.
Stonebraker, M., *Implementation of Integrity Constraints and Views by Query Modification*, Proc. Int. Conf. ACM SIGMOD, 1975.

Thayse, A., Gribomont, P., Hulin, G., Pirotte, A., Roelants, D., Snyers, D., Vauclai r, M., Gochet, P., Wolper, P., Gregoire, E., Delsarte, P., *Approche logique de l'intelligence artificielle, Tome 2 : De la logique modale à la logique des bases de données*, Dunod, 1989.

Turner, T., *Logiques pour l'Intelligence Artificielle*, Manuels Informatiques Masson, Masson, 1986.

Ullman, J.D., *Principles of Database and Knowledge-Base Systems,* Computer Science Press, 1989.

Ungar, D., Smith, R., *Self : The Power of Simplicity*, Proc. Int. Conf. OOPSLA, 1987.

Urban, S.D., Karadimce, A.P., Nannapaneni, R.B., *The Implementation and Evaluation of Integrity Maintenance Rules in Object Oriented Databases*, Proc. IEEE Int. Conf. on Data Engineering, 1992.

Weber, W., Stucky, W., Karszt, J., *Integrity Checking in Database Systems,* Information Systems Vol. 8, No. 2, 1983.

Widom, J., Cochran, R., Lindsay, B., *Implementing set-oriented production rules as an extension to Starburst*, Proc. Int. Conf. VLDB, 1991.

Widom, J., *The Starburst Active Database Rule System*, IEEE Transactions on Knowledge and Data Engineering, 1996.

Wiseman, S., *Control of Confidentiality in Databases*, Computers and Security, Vol. 9, No. 6, Octobre, 1990.

Wiseman, S., *On the Problem of Security in Databases*, IFIP WG 11.3 Proc. Int. Work. on Database Security, Septembre 1989, Database Security, III : Status and Prospects, North Holland, pp 301-310, 1990.

Wolper, P., *Temporal Logic Can Be More Expressive*, Proc. Int. Conf. IEEE annual Symp. on Foundation of Computer Science, 1981.

Yahia, A., Lakhal, L., Ciccheti, R., *Building Inheritence Graphs in Objet Database Design*, Proc. Int. Conf. on Database and Expert Systems Applications (DEXA), Lecture Notes in Computer Science 1134, Springer Verlag, pp 11-28, Septembre, 1996.