DRIVER : Une couche objet pour les bases de données relationnelles

Franck LEBASTARD
CERMICS
INRIA
06902 Sophia-Antipolis Cedex
France

Résumé

Ce rapport décrit les caractéristiques générales du système DRIVER[1], couche objet virtuelle persistante permettant d'utiliser dans un même formalisme objet choisi aussi bien l'information contenue dans les bases de données relationnelles que la connaissance d'un système de plus haut niveau, comme notre générateur de Systèmes Experts SMECI. Un schéma de correspondances, défini par l'utilisateur, associe aux données des bases connectées une représentation objet qui permet de les manipuler et de les exploiter de façon absolument transparente dans l'environnement du système expert, notamment dans le cadre du raisonnement. DRIVER permet également d'apporter la persistance aux objets de l'environnement, au gré de l'utilisateur.

DRIVER : An object layer for relational databases

Abstract

This report describes the specifications of DRIVER, a persistent virtual object layer, that permits to use, in a same chosen object formalism, both the information contained in relational databases and the knowledge of a higher-level system, such as our expert system shell SMECI. A user-defined mapping assigns an object representation to data of connected bases; it permits to handle and to utilize them exactly as other objects in the expert system environment, for example during reasoning. DRIVER can also supply some environment objects with persistency, according to user's wishes.

1. **D**onnées **R**elationnelles et **I**nterface **V**irtuelle pour l'**E**xpertise et le **R**aisonnement

1 Caractéristiques générales

Notre souci initial était de permettre au Générateur de Systèmes Experts SMECI [Sme90] d'accéder aux bases de données relationnelles et de les exploiter dans le cadre du raisonnement.

Nombreuses sont en effet les applications SMECI souhaitant pouvoir orienter leur raisonnement par la consultation de larges banques de données. La possibilité d'interroger une base permet par exemple de connaître les matériaux produits dans une région pour y minimiser le coût d'une construction, de consulter les normes techniques pour que cette construction soit conforme aux règles de l'art, de déterminer les fondations en fonction du site, de sélectionner sur catalogue les poutres d'acier utilisables selon le dimensionnement effectué par le système... On constate que les utilisations en conception sont innombrables. Elles le sont également dans d'autres domaines quand la base de données est utilisée comme support pour le partage d'informations "up-to-date". Un système expert spéculateur peut par exemple tirer profit de la consultation en temps réel des fluctuations boursières pour gérer des porte-feuilles de façon optimale. Un pilote automatique peut moduler l'allure et le cap d'un navire en fonction du trafic et de la météo tenus à jour dans une base de données commune, etc...

Une autre de nos préoccupations était d'apporter la persistance aux objets manipulés par SMECI qui présentent un intérêt pour une utilisation ultérieure, en particulier l'ensemble de ses bases de connaissances et le produit de ses analyses.

Il peut en effet être utile et très efficace de pouvoir retrouver les solutions proposées à des problèmes posés lors de sessions précédentes. Les applications raisonnant d'abord par analogie sur les cas déjà rencontrés avant d'éventuellement poursuivre en utilisant des codes de calcul plus complexes et plus coûteux en temps sont des exemples de systèmes demandeurs de persistance.

1.1 Un pont entre deux mondes

Nos efforts se sont concrétisés au travers du système DRIVER [Leb90a, Leb90b, Leb93] dont l'une des caractéristiques principales est de définir une couche objet pour les bases de données relationnelles.

DRIVER permet de présenter les données relationnelles sous forme d'objets complexes et les rend directement accessibles pour exploitation dans l'environnement objet du client[1]. C'est d'autre part ce même client qui précise le modèle objet dans lequel les données relationnelles sont exprimées. De cette façon, les objets sont particulièrement adaptés à

1. Environnement Le_Lisp pour l'instant puisque notre prototype a été développé dans ce langage.

l'usage qu'il souhaite en faire et peuvent être manipulés dans l'environnement de la même façon que ceux qui ne sont pas issus d'une base de données. Tous peuvent s'intégrer harmonieusement dans la même couche objet.

Par ailleurs, DRIVER permet également de rendre persistants dans une base de données relationnelle des objets de l'environnement qui ne représentent pas déjà des données d'une telle base.

Dans le premier cas, le système associe une représentation objet à des structures relationnelles. À l'inverse, dans le second, il associe une représentation relationnelle à des structures objet. À chaque fois sont utilisées de semblables correspondances mettant en rapport les deux représentations, correspondances en fait indépendantes du sens dans lequel s'effectue la transaction. Elles constituent le *schéma de correspondances* (figure 1) qui permet la traduction et le transfert des données d'un monde dans l'autre lors des échanges gérés par DRIVER entre la base et l'environnement objet.

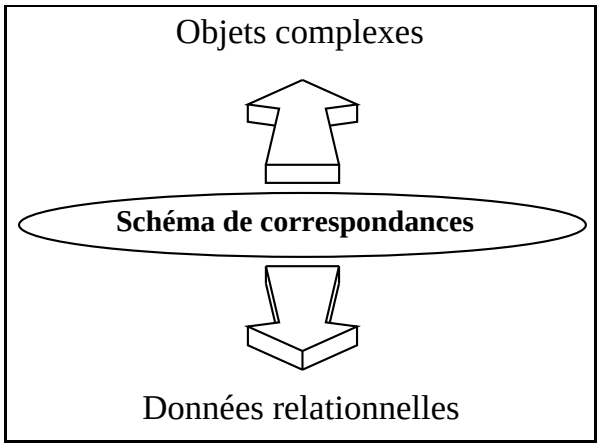

FIG. 1 - *Un schéma de correspondances entre deux représentations*

Un schéma de correspondances contient trois catégories d'informations : celles relatives aux deux représentations de données, plus celles qui assurent leur rapprochement.

Sa construction peut être effectuée de différentes manières. En général, l'une des deux représentations existe déjà. Reste donc à définir l'autre et à indiquer les liens qui les unissent.

La solution la plus simple peut consister à générer automatiquement les éléments manquants du schéma. DRIVER dispose de telles fonctionnalités qui permettent la prise en compte et l'exploitation immédiates d'une table relationnelle dans l'environnement et l'attribution tout aussi rapide de la persistance à une classe [Leb92].

Cette possibilité peut n'être considérée que comme une solution à court terme. Elle est en effet assez brutale. Les éléments générés mécaniquement ne peuvent être que stéréotypés, ils manquent en plus de lisibilité. Il est en général préférable de décrire plus finement

ses correspondances, notamment pour choisir une représentation objet pour des données relationnelles. Une des raisons principales est que seul l'utilisateur connaît la sémantique de certains attributs, en particulier ceux qui constituent des références vers d'autres tables. Le système n'a aucun moyen de la déterminer. Par souci de clarté, le terme "attribut" fera toujours référence dans notre propos à une colonne de table relationnelle tandis que le terme "champ" se rapportera à la propriété d'une classe d'objets.

Le schéma de correspondances peut également être construit à l'aide d'un éditeur graphique prévu à cet effet [BM92]. Le néophyte est alors pris par la main et guidé pas à pas. À chaque étape, les choix proposés tiennent compte de ce qui a déjà été fait, rendant impossible l'introduction d'incohérences ou d'incompatibilités.

L'utilisateur avancé préfèrera directement décrire son schéma à partir de l'interface fonctionnelle [Leb92]. C'est la méthode la plus précise et la plus efficace. Un contrôle très strict de chaque déclaration est effectué, assurant ainsi la validité du schéma à chaque étape de sa construction. L'éditeur graphique dont on a précédemment parlé repose en réalité sur l'interface fonctionnelle. C'est une aide qui permet de construire simplement un schéma sans connaître le nom des primitives et leurs arguments.

1.2 Une couche objet virtuelle persistante

Les données relationnelle dans la base sont interprétées comme des objets potentiels qui résultent des associations définies dans le schéma de correspondances. Ces objets potentiels sont appelés les *objets relationnels*. DRIVER permet l'accès immédiat aux objets relationnels depuis l'environnement. Cet accès peut par exemple être réalisé en construisant en mémoire les images des objets relationnels sélectionnés. Ces images sont appelées les *objets jumeaux* (cf. §3.4.2). Cependant, si le client le souhaite, l'accès peut être directement effectué dans la base de données sans construction dans l'environnement des jumeaux correspondants.

Du fait du couplage fort qu'il réalise avec les SGBDR, DRIVER élargit l'ensemble des objets directement accessibles et manipulables dans l'environnement en adjoignant aux objets spécifiquement de l'environnement les objets représentants de données relationnelles. L'ensemble de ces objets constituent la *couche objet virtuelle de* DRIVER. Comme le système fournit en outre toutes les fonctionnalités nécessaires pour administrer la persistance et l'attribuer à tel ou tel objet ou à telle ou telle classe, on qualifie également cette couche objet virtuelle de persistante (figure 2).

DRIVER donne accès aux objets relationnels par une fonction qui permet de les filtrer. Ainsi, il est possible de sélectionner dans la base des objets relationnels vérifiant certains critères et de les exploiter ensuite au sein de la couche objet, par exemple dans le cadre du raisonnement d'un système expert.

Le filtre est un prédicat à n variables où chacune d'entre elles représente une instance d'une classe précisée, décrite dans le schéma de correspondances. Le corps du prédicat est un ensemble de contraintes liant généralement ces n variables et pouvant être des com-

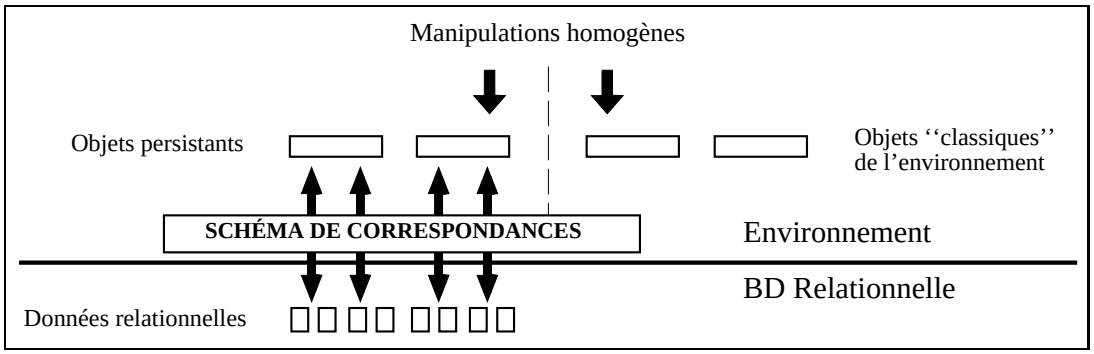

FIG. 2 - *La couche objet virtuelle*

paraisons entre objets, entre champs, entre champs et valeurs. Les substitutions solutions sont donc des n-uplets d'objets relationnels. Les objets filtrés ne sont pas construits dans l'environnement. Sont retournées à la place les références des objets relationnels concernés qui permettent de construire ensuite des "objets-liens" vers ces objets relationnels.

Ces "objets-liens" vers la base de données sont appelés des *défauts d'objet*. Ils rendent dans l'environnement la manipulation d'un objet relationnel qui a été filtré strictement identique à celle d'un simple objet de même classe en mémoire. Toutes les méthodes associées à sa classe sont également valables pour les deux catégories d'objets si bien qu'un utilisateur final peut complètement ignorer, lorsqu'il manipule un objet, si ce dernier est une représentation de données relationnelles ou un simple objet de l'environnement sans rapport avec la base. Les défauts d'objet sont plus précisément décrits en §3.3.

Au besoin, par exemple pour consulter ou mettre à jour la valeur d'un champ, il est possible, à partir du défaut d'objet, de construire dans l'environnement l'objet jumeau de l'objet relationnel correspondant.

On a vu que la couche objet pouvait comprendre des classes définies pour représenter des structures relationnelles. Elle comprend également celles à qui on a associé une représentation relationnelle pour pouvoir écrire leurs instances dans la base. Par souci de simplification, on qualifiera de *persistantes* l'ensemble de ces classes qui ont pour point commun d'être décrites dans un schéma de correspondances. Une classe persistante peut offrir la persistance à chacune de ses instances. Les informations propres à la classe sont elles-même persistantes grâce à un *méta-schéma* .

Les classes persistantes s'intègrent complètement aux hiérarchies de classes définies dans l'environnement (figure 3). Toute classe peut posséder une ou plusieurs sous-classes persistantes.

Une classe qui possède au moins une classe ancêtre persistante a des instances qui peuvent prétendre à la persistance. En effet, les instances d'une classe sont a fortiori instances des classes ancêtres. La classe hérite de toute qualité qui n'est pas redéfinie, notamment de la persistance. Malgré tout, seuls les champs hérités et décrits dans le schéma sont persistants. Les autres, en particulier les champs locaux, ne le sont pas.

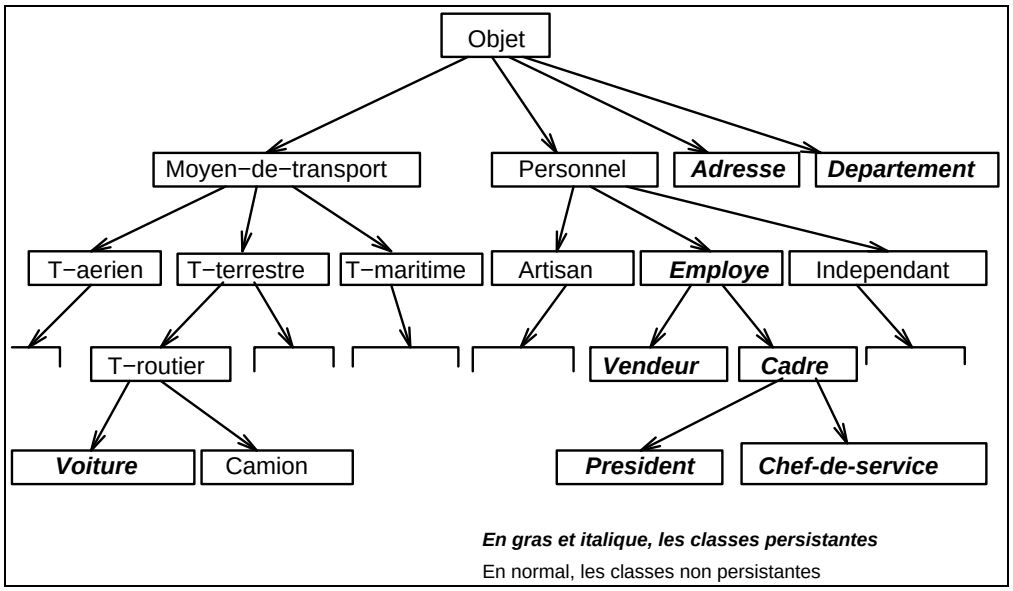

FIG. 3 - *Exemple de hiérarchie mêlant classes persistantes ou non*

Quand on définit une sous-classe persistante à une classe non persistante, les champs hérités ne sont généralement pas persistants, car sans correspondance relationnelle. Heureusement, il est possible de contourner cet obstacle en décrivant leurs correspondances dans le schéma de façon délocalisée, au niveau de la sous-classe.

1.3 Des représentations optimales

Il existe nombre de manières de représenter sous forme d'objets un ensemble de données relationnelles, comme il existe nombre de manières d'exprimer sous forme relationnelle un ensemble de classes.

La façon pertinente d'exprimer dans le monde objet une information contenue dans un attribut dépend de la sémantique qu'on lui prête et de l'utilisation qu'on va en faire.

Ainsi, pour telle application, tel programmeur donne comme correspondance à notre attribut un champ atomique qui permet l'accès à des données brutes, par exemple des entiers. Dans une autre ou avec un autre programmeur, le même attribut peut servir à définir une jointure[2] qui est la correspondance d'un champ de type "objet". Un champ de type "objet" ou "ensemble d'objets" (lien entre objets) a en effet pour correspondance naturelle une jointure qui est un lien entre deux tables. Autre exemple : lorsque cette jointure est définie entre deux tables principales (correspondantes de deux classes) et est, par exemple, l'expression d'une relation 1:N, on a le choix entre lui associer un champ de type "objet" dans l'une ou définir un champ de type "ensemble d'objets" dans la seconde. On peut même lui associer les deux, moyennant certaines restrictions d'usage.

2. Au sens SGBD.

Le choix existe également pour affecter une représentation relationnelle à un ensemble de classes. Une classe et ses sous-classes sont associées à une table appelée *table principale* et la plupart des correspondances des champs sont définies sur elle. Il est néanmoins tout-à-fait autorisé de choisir la correspondance d'un champ sur une autre qui devient de ce fait *table secondaire* de la hiérarchie de classes précédente.

Par ailleurs, une jointure, définie pour accéder à une information, un ensemble d'informations, à un objet ou à un ensemble d'objets, peut être définie de multiples façons possibles.

Ces exemples donnent un aperçu de l'ensemble des choix qui s'offrent au concepteur d'un schéma de correspondances qui tente de coupler une représentation à l'autre. On peut en outre faire la remarque que ce dernier n'est nullement tenu, lorsqu'il définit un schéma, d'y décrire la totalité des structures auxquelles il a accès, que ce soit dans une représentation ou dans l'autre.

Quand son but est d'attribuer la persistance à un ensemble de classes, nul n'est besoin de décrire les autres. Il lui est également inutile de définir au sein d'une classe à rendre persistante, la correspondance de champs dont l'écriture dans la base de données ne présente pour lui aucun intérêt.

Quand il souhaite associer une représentation objet à des données relationnelles qu'il a besoin d'exploiter, il est également libre de ne décrire de la base que les structures qui l'intéressent et, bien sûr, de ne définir que leurs correspondances.

La structure des classes qui en résultent et la forme sous laquelle les données choisies s'expriment dans les objets sont ainsi adaptées aux besoins réels de l'utilisateur ou du système raisonneur.

Si un schéma semble optimal pour résoudre un certain problème, une autre formulation de ce même problème peut faire apparaître d'autres façons de le résoudre et laisser émerger un autre schéma possible. Une information à gérer dans la représentation objet peut y être exprimée de multiples façons et dans des champs de types différents. C'est la manière par laquelle cette information va être exploitée qui doit guider dans le choix de sa représentation. Cette flexibilité de la représentation objet associée aux données relationnelles permet de l'adapter et de la faire évoluer en fonction des besoins qui changent.

1.4 Exploitation de plusieurs bases

Il est possible de tirer profit de l'accès à plusieurs bases de données en définissant pour chacune d'entre elles un schéma de correspondances. Chaque schéma décrit alors comment chaque base est exploitée.

Lorsque plusieurs schémas ont été définis dans le même environnement, chacun d'entre eux y explicite un certain nombre de classes persistantes qui s'intègrent dans la hiérarchie des classes.

Une exploitation avancée de ces possibilités de connexions multiples est ébauchée en §4.

2 Gestion de la persistance

2.1 Granularité de persistance

Avec DRIVER, une information contenue dans un objet est dite persistante si elle a été déclarée persistante (!) et s'il existe un schéma de correspondances dans lequel est décrite sa correspondance relationnelle.

Le seul fait de définir une correspondance ne suffit pas à rendre persistantes les données concernées. En effet, dans un environnement comme celui d'un système expert, l'information évolue beaucoup et il n'est pas toujours souhaitable de conserver l'ensemble des données transposables dans la base grâce à une correspondance. Il vaut mieux avoir la possibilité d'attribuer cette persistance de façon plus précise en sélectionnant explicitement ce qui doit être retenu.

C'est le cas avec DRIVER. D'une manière générale, le schéma de correspondances permet de décrire la correspondance relationnelle de classes et de champs. Tout objet instance d'une classe décrite dans un schéma peut prétendre à la persistance. Deux objets appartenant à une même classe peuvent être différenciés parce que l'un possède cette qualité et l'autre non. Il appartient à l'utilisateur de l'accorder à ceux qui lui semblent présenter un intérêt pour une réutilisation ultérieure.

Au sein d'une même classe rendue persistante peuvent être combinés champs persistants ou non. À nouveau, il est possible de sélectionner, cette fois au cœur même des objets persistants, les informations pouvant prétendre à la pérennité. Les champs non persistants sont tout simplement ignorés pendant les écritures ou lectures d'objets dans la base.

Ces champs non persistants ne doivent pas forcément être déclarés dans le schéma. Ils n'y ont leurs places que si le schéma est également mis en œuvre pour définir dans le modèle objet client les classes auxquelles ils appartiennent. Dans ce cas, leur absence dans la description de la classe dans le schéma se traduirait par leur absence dans la définition de la classe dans le modèle objet utilisé. Quand le schéma n'est pas utilisé pour cela, quand, par exemple, les classes préexistent au schéma, nul n'est besoin de déclarer à DRIVER les champs qu'on ne désire pas rendre persistants.

Il peut être parfois utile de supprimer momentanément la persistance d'un champ. Cette opération peut facilement être réalisée avec DRIVER, par exemple pendant la phase de mise au point de l'application ou encore pour augmenter localement les performances du système pendant certaines opérations de transfert de données entre l'environnement et la base. Le même type d'opération est également possible avec les classes et se traduit par une gestion encore plus précise de la persistance qui supporte ainsi la notion d'exception. Tel champ d'ordinaire persistant ne l'est plus dans certaines conditions d'exécution de l'application.

Enfin, il peut être parfois utile de décrire, au niveau d'une sous-classe persistante, la correspondance de champs définis dans une classe mère. Ceci est particulièrement vrai quand cette classe mère n'est pas persistante.

Quand, par exemple, le modèle objet utilisé présente une classe racine unique et que cette classe possède un champ `nom`, toutes les classes qui seront ensuite définies hériteront de ce champ `nom`. S'il est souhaitable de rendre l'une de ces classes persistante, il peut être tout aussi souhaitable d'également rendre le champ `nom` persistant sans pour autant devoir attribuer la persistance à la classe racine.

DRIVER permet ainsi de décrire au niveau d'une classe la correspondance de champs non locaux qui seront gérés comme les autres en ce qui concerne la persistance, mais qui seront ignorés lorsque la classe sera définie dans le modèle objet client en utilisant le schéma de correspondances.

2.2 Persistance des classes et des correspondances

Le schéma de correspondances contient des informations relatives aux classes et aux relations qu'il aide à rapprocher et relatives aux correspondances établies entre ces classes et relations. Ces trois types d'information sont aisément séparables pour effectuer des opérations spécifiques à chacune d'entre elles comme définir des classes dans le modèle objet utilisé ou créer des relations dans la base.

DRIVER assure la sauvegarde des informations structurelles qui lui sont nécessaires en gérant la persistance des schémas de correspondances.

Cette persistance des schémas et de l'ensemble des informations qu'ils contiennent est assurée grâce à un *méta-schéma* . Ce schéma associe des relations système à l'ensemble des classes système de DRIVER qui participent à la description des schémas de correspondances. Ces relations forment de par leur nature une sorte de méta-base, un *dictionnaire de données complexes.*

Il peut arriver que DRIVER soit instancié avec un modèle objet dont il ne puisse pas gérer l'ensemble des concepts, tel celui de méta-classe. Les informations sur les classes contenues dans un schéma de correspondances sont alors insuffisantes pour prendre en compte l'ensemble des caractéristiques du modèle objet. Apporter la persistance aux schémas de correspondances ne suffit donc pas pour rendre persistantes toutes les informations structurelles du modèle objet client.

Si des caractéristiques particulières de ce modèle objet se sont pas implicitement gérées par DRIVER et que leur absence ne permet pas de reconstruire correctement l'ensemble des informations structurelles, il est tout à fait possible de définir un schéma de correspondances en complément du méta-schéma, associant ces traits qui ne sont pas gérés à des tables "système" supplémentaires.

3 Transactions

Un objet persistant modifié ou créé dans l'environnement n'est pas immédiatement mis à jour dans la base. Cette phase d'écriture et de répercussion dans la base de modifications effectuées en mémoire doit être explicitement provoquée par un ordre de validation de *transaction.*

3.1 Transaction DRIVER et transaction SGBD

Une *transaction* DRIVER est une session propre à un ensemble d'objets persistants pendant laquelle toute modification de l'un d'entre eux n'est effective que dans la couche objet virtuelle à laquelle ils appartiennent. Tous les autres clients de la base, en particulier des données relationnelles correspondant à ces objets, continuent de les "voir" comme si elles n'avaient pas été modifiées. Les mises-à-jour ne sont visibles pour les autres qu'après validation de la transaction.

La notion de transaction existe aussi au niveau du SGBD et implique l'ensemble des données d'une base. C'est une session d'accès à une base, propre à un utilisateur. Elle se valide par exemple en SQL au moyen de la commande COMMIT.

Toute requête adressée au SGBD dans le cadre d'une telle transaction verrouille les données concernées et les rend provisoirement inaccessibles pour les autres transactions. Ce verrouillage existe quel que soit le type de la requête, qu'elle soit une sélection, une insertion, une mise-à-jour ou une suppression de n-uplet. Selon le SGBD sont verrouillées les seules données concernées, les pages[3] qui les contiennent ou les relations entières. Tout client voulant accéder dans sa propre transaction à des données verrouillées par un autre est mis en attente. Il reste bloqué jusqu'à ce que ces données soient à nouveau disponibles, jusqu'à ce que le "verrouilleur" valide sa transaction.

La transaction DRIVER ne se confond pas avec la transaction SGBD. Cet attachement aurait pu être préjudiciable pour toutes les applications nécessitant le partage de données. En fait, la validation de transaction SGBD est entièrement gérée par DRIVER qui l'effectue à bon escient, de façon totalement indépendante de celle de son propre système de transactions qui s'applique aux objets et uniquement sur demande.

Une transaction DRIVER implique soit l'ensemble des objets persistants de la couche virtuelle, soit un groupe d'objets qui est précisé à la validation. Sa granularité est d'un objet, c'est-à-dire que la plus petite transaction peut ne concerner qu'un objet. Si cet objet en référence d'autres directement ou indirectement par l'intermédiaire de champs de type objet, il est possible d'inclure automatiquement ces objets référencés dans la transaction. Dans ce cas, ils sont également mis à jour dans la base lors de la validation.

Le fait que la transaction DRIVER n'implique pas forcément l'ensemble des objets persistants de la couche virtuelle est intéressant pour des raisons d'efficacité. De cette manière, à chaque phase de mise-à-jour de la base n'est concernée qu'une partie choisie de la couche objet, ceci se traduisant notamment par des gains en performances appréciables. Comme nous l'avons déjà souligné, la transaction SGBD, elle, implique à chaque fois l'ensemble des données contenues dans une base.

Enfin, les validations des deux types de transaction ne coïncident pas et sont effectuées selon des fréquences différentes. En DRIVER, une validation de transaction SGBD est faite[4]

3. La page est une entité de stockage propre aux BD
4. Par défaut. Voir plus loin le rôle de l'indicateur système driver-commit-after-reading.

après chaque opération de transfert de données entre la couche objet et la base. Si un objet ou un groupe d'objets est chargé en mémoire, une validation SGBD libère aussitôt les relations verrouillées par la lecture. Si un champ d'objet est directement lu dans la base, il est immédiatement suivi d'une validation analogue.

L'insertion ou la mise-à-jour de données dans la base **se termine** également par une validation SGBD. Toute opération d'écriture d'un objet ou d'une information contenue dans un objet commence en DRIVER par une lecture des données correspondantes dans la base. Cette lecture, par contre n'est pas immédiatement suivie d'une validation SGBD et se traduit ainsi concrètement par le verrouillage des données consultées. Elles peuvent ensuite être mises à jour sans risque d'intervention intempestive d'un autre client. L'écriture est close par la validation de transaction SGBD qui rend visibles de tous les éventuelles modifications de la base.

L'accès ou la modification d'informations contenues dans la base est ainsi non bloquant pour les autres. Les validations de transaction SGBD qui accompagnent la plupart des accès de DRIVER à la base de données garantissent des temps de verrouillage du SGBD extrêmement courts, ce qui est très utile lors de l'exploitation d'une base partagée.

Si l'application DRIVER est seule à utiliser la base ou si l'on souhaite adopter une politique plus sûre de gestion des accès concurrents, il est possible d'éviter les fréquentes validations SGBD qui sont faites en changeant la valeur d'un l'indicateur système (driver-commit-after-reading). Ceci peut se traduire par une efficacité accrue du système car DRIVER n'effectue plus de validation de transaction SGBD qu'au moment où il valide la transaction DRIVER. Dans ce cas, les transactions SGBD et DRIVER sont confondues.

En conclusion, le mécanisme de transaction au niveau de la couche objet apporte un confort important dans la manipulation des objets persistants. Ceux-ci peuvent ainsi être modifiés, complétés, affinés sans subir les lourdeurs liées à l'écriture en temps réel sur disque.

De façon générale, le mécanisme des transactions permet de protéger l'intégrité et la sûreté des données qui restent cohérentes au sein de la base.

3.2 Maintien de la cohérence

Un certain nombre de modifications effectuées sur des objets peuvent être cohérentes dans leur ensemble et incohérentes si elles sont faites partiellement. De telles modifications sont réalisées de façon sûre et validées conjointement par une même transaction alors qu'une mise-à-jour incrémentale aurait provoqué une incohérence momentanée au sein du SGBD.

Les transactions DRIVER comportent en plus des transactions SGBD un certain nombre de caractéristiques spécifiques qui vont dans le sens d'une plus grande sécurité pour les données contenues dans la base.

Une validation de transaction permet de mettre à jour dans la base un groupe d'objets persistants. Les modifications effectuées dans l'environnement sur les objets persistants

peuvent ainsi être validées de façon incrémentale, par exemple, après contrôle de leur validité.

Quand la base est partagée, un même environnement peut contenir simultanément des objets persistants dont la fréquence de validation dans la base doit varier de l'un à l'autre. On peut par exemple distinguer les objets qu'on sait uniquement manipulés par notre environnement et qui n'ont pas besoin d'être validés très fréquemment. À l'inverse, il y a ceux qui contiennent des informations partagées entre plusieurs systèmes et qui doivent être validés dès qu'ils sont modifiés. Grâce aux transactions DRIVER, ces différents types d'objets peuvent être gérés de façon optimale en minimisant les accès au SGBD.

En cas d'incohérence dans la couche virtuelle, au sein d'un groupe d'objets persistants précisément identifié, il est possible, en annulant la transaction en cours pour ces objets, de revenir en arrière et de leur redonner une valeur cohérente. Cette valeur est celle de leurs objets relationnels respectifs dans la base. Ils reflettent donc à nouveau très précisément les données relationnelles qu'ils représentent dans l'environnement. Le groupe est ainsi replacé dans un état qui a précédemment été reconnu valide.

Il est intéressant de remarquer que seul ce groupe est revenu à des valeurs antérieures. Le reste de la couche objet reste inchangé, en particulier les autres objets persistants modifiés qui n'ont pas encore été validés dans la base.

3.3 Défauts d'objet et objets relationnels

DRIVER permet une manipulation identique des objets de l'environnement et des *objets relationnels* au sein de la couche objet virtuelle. Rappelons que les objets relationnels sont les objets **potentiels** contenus dans la base de données qui résultent des correspondances effectuées.

Les objets relationnels sont atteints au moyen d'une primitive qui permet de les filtrer. Cependant, cette primitive ne renvoie pas directement les objets en question, mais leurs références. Ces dernières peuvent être confiées à une autre primitive qui délivre des objets système DRIVER qui "représentent" les objets relationnels correspondants dans l'environnement en l'absence d'objets jumeaux. Ces objets système sont les *défauts d'objet*.

Un défaut d'objet constitue un lien vers l'objet relationnel qu'il représente en mémoire (figure 4). Il contient toutes les informations nécessaires pour retrouver dans la base les données relationnelles associées et leur donner une représentation objet. En particulier, il connaît un schéma de correspondances, la classe principale de l'objet représenté et sa valeur de clé qui l'identifie de manière unique dans la base. Par le schéma de correspondances, il accède à la connaissance permettant de faire passer les données d'une représentation dans l'autre et à un canal de communication par lequel le dialogue avec la base est possible.

À la demande et de façon transparente, le défaut d'objet est capable d'interroger la base et d'accéder aux informations souhaitées contenues dans l'objet relationnel qu'il représente.

En fait, il se manipule comme un objet classique de la couche objet. Pour l'utilisateur, l'objet qu'il met en œuvre "*est*" l'objet relationnel, car le défaut d'objet se comporte

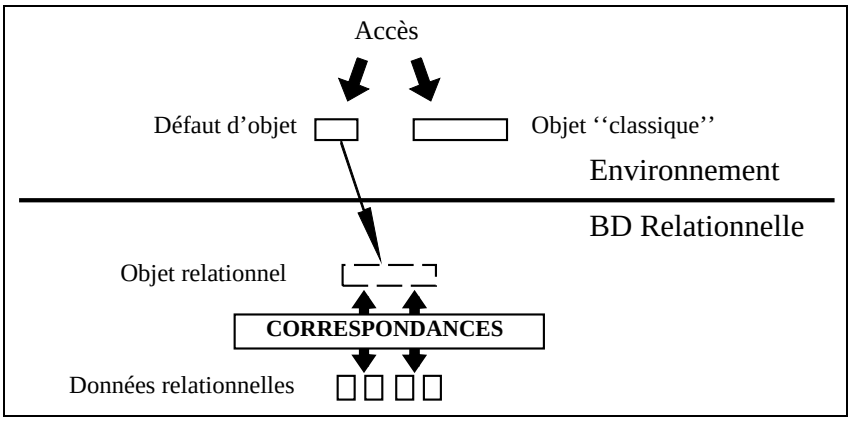

FIG. 4 - *Défaut d'objet et objet relationnel*

comme s'il était celui qu'il représente. Les champs définis pour la classe de l'objet relationnel sont normalement accessibles et peuvent être simplement consultés et mis à jour. Les méthodes sont également applicables. Le défaut d'objet répond correctement aux envois de message qui lui sont adressés et qui sont en réalité destinés à l'objet relationnel.

Le défaut d'objet est petit; il occupe très peu de place en mémoire. À l'opposé, il peut représenter des objets relationnels de taille quelconque qui peuvent donc être a fortiori très gros. Par cette technique du défaut d'objet, DRIVER permet ainsi l'exploitation, depuis l'environnement, de bases de données importantes. La mémoire n'est pas surchargée puisque les données n'ont pas besoin d'être chargées pour être prises en compte. Inversement, il est également possible, de ce fait, de décentraliser au niveau des bases de données certains modules de connaissance, parfois peu utilisés, qui, jusqu'à présent, étaient gérés en mémoire.

3.4 Accès aux champs et objets jumeaux

Le défaut d'objet permet d'accéder de deux façons différentes aux informations contenues dans l'objet relationnel.

3.4.1 L'accès direct

La consultation ou la mise-à-jour d'un champ de l'objet relationnel peut se traduire par un accès à la base où sont directement lues ou écrites les données relationnelles correspondant à la valeur du champ. Chaque accès se traduit par l'envoi d'une requête au SGBD. Après l'opération, le défaut d'objet est toujours un défaut d'objet.

Ce type d'accès est intéressant à plus d'un titre. Il permet de consulter certaines informations contenues dans de gros objets sans alourdir la mémoire plus que ne le réclamerait l'application. Dans le cas d'objets partagés, on accède ainsi à des valeurs qui sont toujours à jour. Des données issues d'autres utilisateurs de la base peuvent ainsi être

immédiatement prises en compte.

Il présente cependant certains inconvénients. N accès à un objet se traduisent par N requêtes envoyées au SGBD, ce qui peut être lourd et manquer d'efficacité quand on travaille beaucoup avec les mêmes objets. C'est particulièrement vrai quand on sait que ces objets en question ne sont pas partagés avec d'autres clients du SGBD et que deux consultations successives d'un même champ ne risquent pas de renvoyer deux résultats différents. Enfin, un inconvénient de taille existe concernant la mise-à-jour. L'accès direct en écriture dans la base enfreint les règles du système des transactions DRIVER. Toutes les modifications effectuées ainsi ne peuvent pas être défaites par une annulation de transaction. L'usage de cette fonctionnalité en écriture doit donc être utilisée avec beaucoup de précautions.

3.4.2 Les objets jumeaux

L'autre alternative consiste à déclencher la construction d'un *objet jumeau* lors d'une tentative d'accès au champ d'un objet relationnel encore représenté par un défaut d'objet. L'objet jumeau est une image fidèle de l'objet relationnel. Seulement, contrairement à lui, il a l'avantage d'être un objet de l'environnement à part entière, héritant de toutes les propriétés du modèle objet client. À sa construction, il remplace purement et simplement le défaut d'objet si bien que toute connaissance de l'environnement qui référençait le défaut d'objet référence maintenant le jumeau. Une fois construit, le jumeau masque l'objet relationnel associé (figure 5). La primitive DRIVER d'accès aux objets relationnels renvoie d'ailleurs le jumeau s'il existe, au lieu du défaut d'objet.

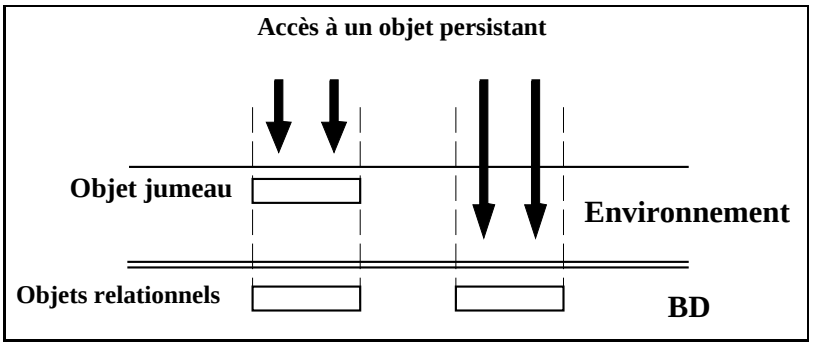

FIG. 5 - *Masquage de l'objet relationnel par l'objet jumeau*

Le jumeau est un objet de l'environnement ayant pour classe celle de l'objet relationnel. Quand il est construit, seules les valeurs atomiques et les ensembles sont chargés. Tous les objets relationnels auxquels il fait référence dans l'un de ces champs sont remplacés par des défauts d'objet. Là encore, si certains disposent déjà de leurs jumeaux, ce sont ces derniers qui sont retrouvés et utilisés.

Il était bien sûr possible de placer et construire directement dans les champs objet les jumeaux des objets relationnels référencés. Mais, pour peu qu'ils référencent à leur tour de nouveaux objets et que le réseau entre objets soit suffisamment étoffé, on aurait vite

fait de charger ainsi récursivement toute la base de données en mémoire. L'intérêt de la couche objet virtuelle s'en serait trouvé singulièrement amoindri.

Le principe des jumeaux s'intègre complètement à celui des transactions DRIVER. Quand un objet relationnel subit une tentative de modification, son jumeau est construit. C'est lui qui est mis à jour; les données relationnelles dans la base ne sont pas touchées. À ce moment, le contenu du jumeau ne correspond donc plus à celui de l'objet relationnel, mais cette différence n'est pas perceptible depuis l'environnement puisque le jumeau masque l'objet relationnel dès qu'il est créé. Tout nouvel accès sur l'objet relationnel est maintenant effectué sur le jumeau. Aucune requête n'est plus adressée au SGBD. À la validation de la transaction, toutes les modifications effectuées sur l'objet sont reportées sur l'objet relationnel, sur les données de la base. L'objet relationnel et le jumeau sont à nouveau strictement identiques. À l'inverse, si la transaction est annulée, tous les objets jumeaux sont remplacés par des défauts d'objet. Les modifications effectuées en mémoire sont ainsi perdues. L'objet jumeau disparaissant, le masquage de l'objet relationnel disparaît également.

Les avantages de cette gestion des accès aux objets relationnels sont assez importants. Tout d'abord, la préservation du système des transactions est une chose importante. Par ailleurs, en règle générale, le système gagne aussi en efficacité. Pour charger un objet (ou plusieurs) d'une classe comportant N champs de type "ensemble" ou "calcul multi–n-uplets", DRIVER effectue $1 + N$ requêtes au SGBD. Les applications comprenant de nombreuses définitions de champs de ces types bénéficient alors de la diminution du nombre de communications avec le SGBD.

Un autre avantage des objets jumeaux concerne les systèmes raisonneurs. Cet avantage est décrit dans la section suivante.

Il est toujours possible d'accéder directement à un objet relationnel dans la base de données, y compris quand son jumeau a été construit. Le fait que son représentant dans l'environnement soit un défaut d'objet ou un jumeau ne fait pas de différence à ce niveau; les primitives d'accès direct restent disponibles.

Cette possibilité est intéressante à plus d'un titre. On peut ainsi à tout moment surveiller l'évolution d'un paramètre dans une base de données partagée, quel que soit le représentant de l'objet relationnel dans l'environnement. Quand le jumeau est formé, il est également possible d'accéder directement à la valeur d'un champ dans la base pour la comparer avec celle contenue dans le jumeau. Enfin, l'accès direct à la base depuis l'objet jumeau permet aussi de gérer en mémoire la liste des valeurs successivement prises par un champ.

Quand l'objet jumeau a été construit, l'accès direct dans la base de données à la valeur d'un champ peut déclencher sur ce jumeau des *réflexes de mise-à-jour* qui sont inhibés par défaut.

En consultation, si la valeur du champ lue dans la base est différente de celle se trouvant dans l'objet jumeau, la différence indique que les données relationnelles correspondantes

ont été modifiées par un autre utilisateur de la base depuis la construction du jumeau. Dans ce cas, DRIVER peut ne pas se contenter de rendre la nouvelle valeur du champ. Il peut également mettre à jour le champ dans l'objet jumeau.

En écriture, quand on accède directement à la base pour modifier le champ d'un objet relationnel, on peut également mettre à jour le champ correspondant de l'objet jumeau si sa valeur diffère de celle qu'on affecte dans la base.

Ces réflexes présentent un intérêt pour les applications où la base est partagée et où il est important d'avoir dans l'environnement des objets qui reflètent au maximum l'état de cette base de données.

Ils présentent cependant certains inconvénients importants qui empêchent de les laisser actifs par défaut. Leur principe est incompatible avec le système de transactions de DRIVER. Il est troublant par exemple qu'un accès en lecture dans la base se traduise automatiquement par l'écriture d'un champ dans l'environnement.

Parfois, des objets sont dépendants les uns des autres du fait de l'existence d'une relation liant certains de leurs champs. Si la consultation dans la base d'un champ prenant part à cette relation se traduisait par sa mise-à-jour brutale dans l'environnement, la cohérence du groupe d'objets pourrait être anéantie.

3.4.3 Objets jumeaux et raisonnement

Certains générateurs de systèmes experts comme SMECI disposent d'objets réversibles. Voyons en quoi les jumeaux présentent un réel intérêt pour ce type de système.

Retraçons d'abord rapidement le principe général de SMECI.

Le processus de conception, sur lequel est basé le raisonnement de SMECI, consiste à assembler des objets élémentaires qui peuvent être soit entièrement connus à l'avance, soit connus de manière partielle. Les paramètres inconnus sont alors déterminés par l'expert à l'aide de règles. A l'aide de la connaissance dont il dispose, SMECI génère en parallèle les différentes solutions compatibles avec les spécifications et contraintes données au départ [Nev86].

Le moteur travaille sur des états qui représentent la solution en cours d'élaboration. Un état est donc caractérisé par les objets du processus de conception, plus ou moins précis selon l'état d'avancement de la solution.

Le moteur développe ainsi un arbre d'états (figure 6) où chaque branche correspond à une variante envisagée, où les nœuds le long de chaque branche sont les états de plus en plus précis par lesquels est passé le moteur. Une fois cet arbre complet, la feuille de chaque branche est soit une solution (feuille $n^o 402$ de notre exemple), soit un état qui a été reconnu incohérent par une règle de contradiction (feuilles $n^{os} 76, 79$, etc...), soit un état bloqué sur lequel aucune règle ne peut s'appliquer.

Au cours du raisonnement, lorsque, pendant la construction de l'arbre des états, le moteur reconnaît son état courant comme étant une feuille, il revient en arrière d'état en état jusqu'au premier où il lui reste au moins une variante à explorer. Il choisit la plus prometteuse par rapport à une fonction d'évaluation et la développe à son tour.

Quand on souhaite exploiter des objets relationnels dans le cadre du raisonnement apparaît le problème posé par la réversibilité des objets de l'environnement.

Si l'on suppose l'absence du mécanisme des objets jumeaux se poserait le problème de la non-réversibilité des données relationnelles dans la base. En effet, les SGBD relationnels ne gèrent pas en standard de liste, administrée par datation ou numéro de version, des valeurs successivement prises par un attribut au cours du temps. En cas de retour-arrière de SMECI, apparaîtrait la difficulté de remettre à jour, dans l'état à partir duquel repart le raisonnement, les objets relationnels qui ont été directement modifiés dans la base au cours de l'exploration de la branche abandonnée. Une différence essentielle s'établirait au sein de la couche objet virtuelle entre les objets SMECI présents dans l'environnement qui sont réversibles et les objets relationnels qui sont incapables, en cas de retour-arrière du système, de retrouver leurs valeurs originelles.

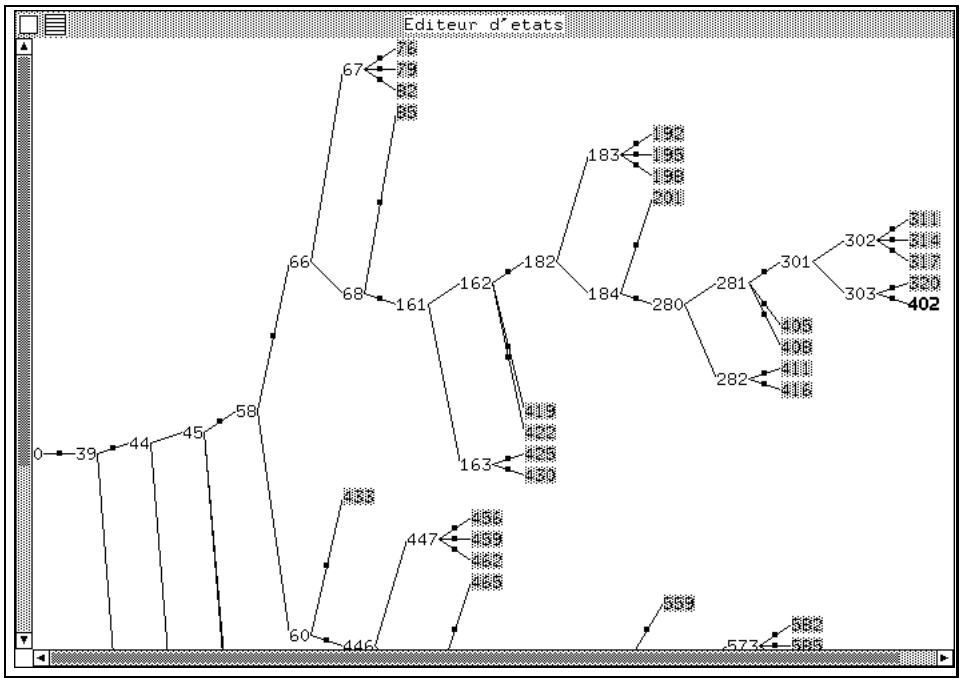

FIG. 6 - *Exemple d'arbre d'états de* SMECI

Les objets jumeaux évitent cette distinction et conservent à la couche objet son homogénéité. Quand un objet relationnel est consulté par le système expert, DRIVER génère automatiquement le jumeau qui est un objet SMECI à part entière et le lui substitue pour la suite du raisonnement. Le jumeau comme tout objet SMECI est réversible. Il peut évoluer au fil du raisonnement et retrouver ses anciennes valeurs en cas de retour-arrière, sans risquer de perturber la cohérence de la base de données. Si le moteur revient en arrière vers un état antérieur à la création de ce jumeau, le masquage disparaît et c'est l'objet relationnel qui est à nouveau éventuellement unifié dans la poursuite du raisonnement.

4 Exploitation avancée des connexions multiples

Plusieurs schémas peuvent être définis, associant chacun une représentation objet à des données relationnelles.

4.1 Vers des objets de persistance distribuée

Aucune cohérence entre schémas n'est gérée par DRIVER. Lors de la construction d'un schéma, aucune vérification de compatibilité n'est effectuée entre les nouvelles déclarations et les autres schémas déjà existants.

Il est ainsi possible d'accorder la persistance à une classe dans un schéma en l'associant à une certaine table, puis de l'accorder dans un second à une sous-classe de la première en l'associant à une autre table. Cette redéfinition de la correspondance d'une classe au niveau d'une sous-classe est interdite en DRIVER au sein d'un même schéma.

Cette absence de contrôle peut être exploitée pour sauver par ce biais un objet en plusieurs parties, chacune dans une base différente. Dans le cas classe/sous-classe évoqué ci-dessus, les champs définis au niveau de la mère sont écrits dans une base, ceux qui sont locaux à la sous-classe sont écrits dans une autre. D'une façon plus générale, certains champs d'une classe peuvent être décrits dans un schéma, d'autres dans un autre, etc...Lors de chaque validation de transaction, selon le schéma courant sont écrits dans la base tels ou tels groupes de champs.

Ces objets relationnels *distribués* ont une identité propre dans chaque base. Chacune de leurs références est donc constituée d'un ensemble de couples schéma de correspondances - clé dans la base.

Le défaut d'objet, "objet-lien" vers un objet relationnel, n'est pas tout-à-fait adapté à référencer un tel objet relationnel distribué dans la version actuelle de DRIVER. En effet, il ne connaît qu'un seul schéma de correspondances et qu'une clé dans la base associée au schéma. Cependant, étendre le système serait aisé. L'accès à un champ d'un objet se ferait simplement en recherchant la classe de l'objet et le champ dans les schémas référencés par le défaut d'objet. Une fois le champ retrouvé[5], le schéma qui le contient déterminerait le canal de communication, la base et la référence de l'objet dans cette base. La suite de l'opération, notamment la construction de la requête resterait la même.

Dans la version actuelle de DRIVER, un défaut d'objet pourrait référencer un objet relationnel distribué à condition, lors de l'accès aux champs, de préciser au défaut d'objet le schéma dans lequel est décrite la correspondance du champ concerné. Un mauvais trait est que, lors du déclenchement de la construction d'un objet jumeau, seuls seraient chargés les champs de l'objet définis dans le schéma référencé par le défaut d'objet.

En l'état actuel des choses, ce genre de manipulation d'objets à persistance distribuée est fortement déconseillé à un utilisateur non avancé.

5. Ces idées n'ayant pas été implantées, les problèmes de conflit de nom de champ n'ont pas été étudiés.

4.2 Vers des représentations multiples contextuelles

Du fait de l'indépendance complète des schémas, aucune information issue d'un autre schéma ne peut être exploitée lors de la construction de l'un d'entre eux. Il est par exemple impossible de définir des champs de type objet destinés à recueillir des instances d'une classe déclarée dans un autre schéma. On peut cependant faire la remarque que ce genre d'opération n'aurait de sens que si les deux schémas étaient définis sur la même base. En effet, la correspondance d'un champ objet est une chaîne de jointures. Si les deux schémas étaient définis sur des bases différentes, des requêtes adressées au SGBD effectueraient des jointures entre tables de bases différentes, ce qui serait insensé.

Par contre, plusieurs schémas peuvent être définis pour une même base. Cette perspective peut être intéressante quand il s'agit de prendre en compte des données relationnelles dans l'environnement. On peut en effet définir plusieurs représentations objet pour de mêmes données, chacune étant exprimée dans un schéma différent.

Ces différentes représentations objet co-existent dans le même environnement. Leur intérêt est qu'elles sont toutes pertinentes pour une certaine tâche, dans un certain contexte. Elles permettent d'accéder à de mêmes données de façons multiples et optimales en fonction du contexte. De cette façon, les champs peuvent directement contenir l'information dans la forme la plus adaptée. Le filtrage des d'objets relationnels correspondants est lui-même plus efficace, les requêtes générées et adressées au SGBD étant optimisées.

Un autre intérêt de ces représentations objet multiples est la possibilité d'avoir simultanément dans l'environnement différents arbres d'héritage établis selon des critères différents et qui représentent de mêmes données. Les différentes classifications des êtres vivants sont des exemples de telles hiérarchies, indépendantes mais basées sur les mêmes données. La classification cladistique regroupe les êtres vivants selon leurs caractéristiques physiques communes tandis que la classification phénétique les regroupe en fonction de leurs similarités génétiques et hématologiques.

Là encore, les descriptions multiples de bases doivent être utilisées avec prudence. En effet, elles court-circuitent les vérifications effectuées lors de la construction d'un schéma, qui interdisent la représentation redondante d'informations dans l'environnement. Certaines données de la base peuvent par cette méthode se retrouver plusieurs fois en mémoire dans des objets de classes différentes. Si elles sont exploitées en simple consultation, elles ne sont la source d'aucun problème particulier. Si elles peuvent être modifiées et mises à jour dans la base, des incohérences peuvent être introduites.

Références

[BM92] S. Boyera and S. Maria. Une interface graphique pour DRIVER. Technical report, ESSTIN, Sophia-Antipolis (France), Juin 1992. 13 pages. Rapport de stage - rapport technique.

[Leb90a] F. Lebastard. DRIVER : A persistent virtual object layer for reasoning. In *ISMIS-90, 5th International Symposium on Methodologies for Intelligent Systems*, pages 74–81, Knoxville (Tennessee), October 1990.

[Leb90b] F. Lebastard. DRIVER : A persistent virtual object layer for reasoning. In *AAAI Workshop on Knowledge Base Management Systems*, page multiple, Boston (Massachusetts), July 1990.

[Leb92] F. Lebastard. DRIVER version 1.34, manuel de référence. Technical Report 92-7, CERMICS-INRIA, Sophia-Antipolis (France), Octobre 1992. 119 pages.

[Leb93] F. Lebastard. *DRIVER : Une couche objet virtuelle persistante pour le raisonnement sur les bases de données relationnelles.* Thèse de doctorat, INSA de Lyon/INRIA/CERMICS Sophia-Antipolis, Mars 1993. 380 pages.

[Nev86] B. Neveu. Une nouvelle génération de systèmes experts pour les domaines de conception. In *8ème Séminaire Tuniso-Français d'Informatique*, page multiple, Tunis (Tunisie), mai 1986.

[Sme90] Ilog, Gentilly (France). *SMECI Version 1.65, Manuel de référence*, Mai 1990. 470 pages.