COMPILING UPDATES IN DEDUCTIVE OBJECT-ORIENTED DATABASES

Internal Report C95-49

21 Dicembre 1995

M. Carboni
V. Foddai
F. Giannotti
D. Pedreschi

Compiling Updates in Deductive Object-Oriented Databases

Marilisa Carboni[1], Valeria Foddai[2], Fosca Giannotti[1], and Dino Pedreschi[2]

[1] CNUCE Institute of CNR
Via S. Maria 36, 56125 Pisa, Italy
e-mail: F.Giannotti@cnuce.cnr.it

[2] Dipartimento di Informatica, Univ. Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: pedre@di.unipi.it

Abstract

Deductive database languages exhibit an evident dychotomy in the way they support queries and transactions. Query answering is based on declarative semantics and fixpoint-based (bottom-up) evaluation. Transactions are based on procedural semantics and top-down evaluation, as for instance in the logic database language $\mathcal{LDL}$ [NT88]. This paper presents a technique to compile updates onto standard logic programs to be evaluated with the usual bottom-up evaluation mechanism. The compilation is based on the concept of XY-stratification [AOZ93, AOTZ93] which is a syntactic property of non-monotonic recursive programs. XY-stratified programs use stage arguments to integrate control on state transition within the deduction process. The proposed compilation is then extended to update operations on deductive object-oriented databases, by providing natural declarative accounts of object identifiers, classes, ISA hierarchies, multiple inheritance, and dynamic binding.

1 Introduction

Logic database languages use a declarative style to represent both knowledge and operations on databases. To coherently model the application domain, a deductive database should also express its dynamically changing aspects. As a matter of fact, updates are a primary concern of any database language.

On the one hand, deductive databases naturally support powerful and declarative query languages, and queries can be efficiently executed using a bottom-up, fixpoint-based procedure. Also, sophisticated optimizations such as magic sets are available to capture the advantages of top-down execution, when needed. On the other hand, deductive databases traditionally suffer from limitations in describing the dynamic and transactional aspects of database systems.

Most proposals in the literature augment deductive databases with a procedural semantics to implement the control mechanisms needed to support updates (see [Mon93] as a source of references.) Often the semantics of this procedural component is accomodated in some logic capable to deal with dynamic. This is the case of transaction logic [BK93], dynamic logic [MW86] or some modal logics. For instance, in the logic database language $\mathcal{LDL}$ [NT88] transactions are special rules with updates, which are evaluated top-down and their semasntics is given by

*Work partially supported by the EC-US Cooperative Activity Project ECUS-033 - DEUS EX MACHINA.

dynamic logic. This combination of declarative and procedural semantics has as a major disadvantage the fact that the architecture of the abstract machine supporting deductive databases is deeply altered. As a consequence, the available optimization techniques are no longer directly applicable.

We propose, in a more conservative way, to leave the simple declarative framework unaltered. This is achieved by means of a transformation of updates and transactions into sets of clauses which

- reflect the intuitive meaning of state changes in a declarative way, and
- can be efficiently executed using the ordinary bottom-up, fixpoint-based evaluation of deductive databases.

We first apply this transformation technique to a generalization of $\mathcal{LDL}$ transactions, i.e., clauses containing *update predicates* in their body. The subgoal preceding the updates is the precondition, and the subgoal following the updates is the postcondition of the transaction. Pre/postconditions are used to preserve database integrity: in particular, the updates are actually performed only if the postcondition is fulfilled, otherwise the transaction has no effect.

The notion of $\mathcal{LDL}$ transaction provides a form of integration of query and updates, in the sense that a uniform notation is available. However, different evaluation methods are used for query and transaction clauses. The proposed compilation allows to execute transaction clauses in a bottom-up way, similarly to queries. The transformation is based on the notion of XY-stratification [AOZ93], i.e., a syntactic property of programs which properly extends ordinary stratification. A remarkable characteristic of XY-stratified programs is that they can be executed by an iterated fixpoint procedure, even if they are recursive, non monotonic programs. XY-stratification is defined in terms of *stage arguments*, i.e., predicate arguments which record the stage of the computation, and allow to control state changes.

The *Statelog*$^{+-}$ proposal in [LL93] is directly related to our work: here, a different formalization of state change leads to similar results, e.g. the capability of computing the perfect model of programs with updates. The main differences with our approach are that we do not admit states as first class values in the language, and that our focus is on compilation of updates aimed at providing efficient executions.

The compilation technique is then generalized to support the basic mechanisms of a deductive object-oriented database model, by providing natural declarative accounts of object identifiers, classes, ISA hierarchies, multiple inheritance, and dynamic binding. We adopt a modification of the approach in [Zan89], and use non-determinism to model uniqueness of oid's. To this purpose, the *choice* construct of $\mathcal{LDL}$ is used, which was formerly introduced in [KN88] and later refined in [GPSZ91, CGP93] so that it can be used within recursive programs. By means of the combined used of XY-stratified negation and non-determinism we obtain a natural model of the update operations which fulfills the oid-uniqueness and hierarchical constraints.

The plan of the paper follows. Section 2 gives a formal account of XY-stratification. Section 3 introduces compilation of updates onto XY-stratified programs in four stages: simple updates, composition of updates, simple and nested transactions. Two different semantics for composition of updates are explored: parallel and sequential evaluation. Section 4 presents the extension of the compilation technique to object-oriented databases. The proposed compilations are illustrated by means of simple examples. Finally, Section 6 contains a few concluding remarks.

2 XY-stratified programs

The basic concept of our approach is the notion of XY-stratification, i.e. a syntactic property of non-monotonic recursive $Datalog_{1s}$ programs [1]. The class of programs identified by such property, named XY-programs, captures the expressive power of the inflationary fixpoint semantics.

The basic idea is that recursive predicates have a special argument named *stage* which is an integer. There are two different ways to use the stage: in rules which do not increment the stage (X-rules), and rules that increment the stage by one (Y-rules). An XY-stratified program allows recursion only when there is an increment of the stage. If there exists a reordering of the rules of the predicates which induces a XY-stratification, then it is possible to apply an iterated fixpoint procedure which distinguishes the application of X-rules by the application of Y-rules. Such procedure computes the perfect model associated to the set of recursive predicates.

The next subsections introduce syntax and semantics of XY-stratified programs following the presentation of [AOZ93].

Syntax

Definition 2.1 Given a program P, a set of rules of P defining a maximal set of mutually recursive predicates will be called a recursive clique of P. □

Definition 2.2 Given a recursive clique Q, the first arguments of the recursive predicates of a rule r in Q will be called the stage argument of r. □

The usage of stage arguments is for counting as in the recursive definition of integers: *nil* stands for zero and *s(I)* stands for *I+1*.

Definition 2.3 Let Q be a recursive clique and r be a recursive rule of Q. Then r is called an

- **X**-rule if all the stage argument of r are equal to a simple variable, say J, which does not appear anywhere else in r;
- **Y**-rule if (i) some positive goal of r has stage argument a simple variable J, (ii) the head of r has stage argument s(J), (iii) all the remaining stage arguments are either J or s(J) and (iv) J does not appear anywhere else in r. □

Definition 2.4 A recursive clique Q will be said to be an XY-clique when all its recursive rules are either **X**-rules or **Y**-rules. □

Priming: an atom $p'(t)$ is called the *primed version* of an atom $p(t)$.
Given an XY-clique Q, its version primed is constructed by priming certain occurrences of recursive predicates in recursive rules as follow:

- **X**-rules: all occurrences of recursive predicates are primed;
- **Y**-rules: the head predicate is primed, and so is every goal with stage argument equal to that of the head.

Definition 2.5 An XY-clique Q will be said to be XY-stratified when

[1] *$Datalog_{1s}$* is a simple extension of *Datalog* which admits a single unary function symbol $s(.)$. This language has been used for temporal reasoning in [CHO93].

- the primed version of Q is non recursive
- all exit rules have as stage argument the same constant. □

where a rule is an *exit*-rule if all predicates in its body are not defined in the clique.

Definition 2.6 A program is XY-stratified if every recursive rule that contain a negated recursive goal in its body belong to an XY-stratified clique. □

The dependence graph for a primed clique provides a very simple syntactic test to check whether a program is XY-stratified: it contains no cycles, thus there exists a topological sorting of the nodes of Q' which obeys stratification, and such that the unprimed predicate names precede the primed ones.

Semantics

We can partition the atoms in the Herbrand Base B_Q of the original program Q into classes according to their predicate name and their stage arguments as follow:

- there is a distinct class, say σ_0, containing all instances of non recursive predicates in Q, without a stage argument;
- all atoms with the same recursive predicate name and the same number of function symbols s in the stage argument belong to the same equivalence class $\sigma_{n,p}$, with n denoting the number of s function symbols in the stage argument of p.

The partition Σ of B_Q constructed in this way can be totally ordered, by letting σ_0 be the bottom stratum in Σ , and then letting $\sigma_{n,p} \prec \sigma_{m,q}$ if

- $n < m$, or
- if $n = m$ but p' precedes q' in the primed sorting of the clique.

The totally ordered Σ so constructed will be called *stage layering* of B_Q.

Theorem 2.1 *Each XY-stratified clique Q can be locally stratified according to a stage layering of B_Q. Then for every instance r of each rule in Q, the head of r belongs to a layer strictly higher than the layers for the goals in r (strict stratification).*

Since the stratification is strict, in computing the iterated fixpoint, the saturation for each stratum is reached in one step. Therefore, the compiler can reorder the rules according to the primed sorting of their head names; then having derived all atoms with stage value J a single pass through the rules of Q ordered according to the primed sorting computes all the atoms with stage value $s(J)$. To formalize the iterated fixpoint procedure for XY-stratified programs, we introduce the following notions.

- Let p' the k-th predicate name in the primed sorting,
- Let T_k the immediate consequence operator for the recursive rules in Q defining p.
- The composite consequence operator Γ_Q will be defined as follows:

  $$\Gamma_Q(I) = T_n(T_{n-1} \ldots (T_1(I)) \ldots)$$

  where I is an interpretation over Q's Herbrand Base B_Q, and $n \geq 1$.

- Let T_0 the immediate consequence operator for the *exit*-rules. By the second condition of XY-stratification, all atoms in $T_0(\emptyset)$ share the same stage argument. However, additional atoms with the same stage value might be obtained by firing the **X**-rules. Therefore, if p_k is the k-th predicate name in the primed sorting, we define $T_k^{\mathbf{X}}$ the immediate consequence operator for the **X**-rules with head name p_k, if any such rules exists, and the identity transformation otherwise. We can define the composite consequence operator for the **X**-rules, $\Gamma_Q^{\mathbf{X}}$ as follow: $\Gamma_Q^{\mathbf{X}} = T_n^{\mathbf{X}}(T_{n-1}^{\mathbf{X}} \ldots (T_1^{\mathbf{X}}(I)) \ldots)$ Thus, the ground atoms with the same stage argument as the *exit*-rules are $\Gamma_Q^{\mathbf{X}}(T_0(EDB))$

Theorem 2.2 *Let Q be a XY-stratified clique, with composite consequence operator Γ_Q and composite consequence operator for the* **X**-*rules* $\Gamma_Q^{\mathbf{X}}$*, then*

- *Q is locally stratified,*
- *the perfect model of Q is $M_Q = \Gamma_Q^{\omega}(M_{nil})$, where $M_{nil} = \Gamma_Q^{\mathbf{X}}(T_0(\emptyset))$ e T_0 is the immediate consequence operator for the exit-rules of Q.*

Thus the perfect model of an XY-stratified clique can be constructed as in the case of positive programs. Computation of XY-stratified programs proceeds similarly to that of stratified programs: all the non recursive predicates in the recursive XY-clique must be saturated before the recursive rules in the clique are computed.

3 Compiling Updates in Deductive Databases

3.1 Simple updates

Updates are often classified according to two different semantics: *weak* updates and *strong* updates. Strong updates are those which allow to delete atoms only if they are in the current database state and allow to insert atoms only if they are not. In the case of weak updates no precondition is checked. Consider, for example, the database instance $\{ p(a), p(b), q(a) \}$. Under the weak semantics the insertion of $p(b)$ or the deletion of $q(b)$ does not change the database, although they are allowed. Under the strong semantics a failure is reached.

We consider here simple updates of the form $+p(a)$, $-p(a)$, corresponding respectively to insertion and deletion of an EDB predicate. As a consequence, view updates are not considered in this paper. Also, we refer here to the weak semantics, although we briefly sketch later how strong semantics might be dealt with.

The idea is to associate with every n-ary EDB predicate p two new $(n+1)$-ary predicate symbols: p_{stage} and p_{del} where the extra argument is the the *stage argument* in the first position. The stage argument in p_{stage} will model the various state transitions of the EDB predicate p performed by the updates. p_{del} will play the role of a delete list, keeping tracks of the tuples to be removed from p.

An update predicate $\pm p(a)$ is then compiled into an XY-stratified program which defines the predicates p_{stage} and p_{del}. The following definition shows the code which updates are compiled to. For generality of exposition, we deal here with updates $\pm p(a)$ where the tuple a may contain variables. The compilation is therefore parametric with respect to a query Q such that $vars(a) \subseteq vars(Q)$, which provides the actual tuples to be inserted in or removed from p.

Definition 3.1 Let p be an EDB predicate, a a tuple and Q a query such that $vars(a) \subseteq vars(Q)$. The code realizing the deletion $-p(a)$ with respect to the query Q, denoted $\mathcal{T}[Q](-p(a))$, is the following:

$r_1: \ p_{stage}(nil, x) \leftarrow p(x).$ $\{exit\text{-rule}\}$
$r_2: \ p_{del}(s(nil), a) \leftarrow Q, p_{stage}(nil, \_).$ $\{deletion\text{-rule}\}$
$r_3: \ p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x).$ $\{copy\text{-rule}\}$

□

Definition 3.2 Let p be an EDB predicate, a a tuple and Q a query such that $vars(a) \subseteq vars(Q)$. The code realizing the insertion $+p(a)$ with respect to the query Q, denoted $\mathcal{T}[Q](+p(a))$, is the following:

$r_1: \ p_{stage}(nil, x) \leftarrow p(x).$ $\{exit\text{-rule}\}$
$r_2: \ p_{stage}(s(nil), a) \leftarrow Q, p_{stage}(nil, \_).$ $\{insertion\text{-rule}\}$
$r_3: \ p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x).$ $\{copy\text{-rule}\}$

□

In both definitions, r_1 is an exit rule which initializes p_{stage}. r_2 is a Y-rule. In case of deletion r_2 records in p_{del}that tuple a has to be deleted; in case of insertion r_2 adds to p_{stage} the new tuple a in the next stage. Finally, in both definitions, the Y-rule r_3 is the *copy*-rule, which allows to copy to the next stage of predicate p_{stage} all tuples which have not been canceled. Notice that, r_3 acts as the *frame axiom* which states: whenever something is true in some stage and it is not explicitly deleted, then it will also be true in the next stage. In the insert and delete rules, the query Q plays the role of providing the actual tuples to inserted/removed. If a is a tuple of constants, then Q is not needed.

It is simple to show that programs $\mathcal{T}[Q](\pm p(a))$ are XY-stratified. It is therefore meaningful to consider their perfect model computed with the iterated fixpoint procedure.

Notice that the new extension of p after the update, denoted by p', is given by the set of tuples with the maximum stage computed by the the corresponding fragment (insert or delete) of p_{stage}:

$r_0: \ p'(x) \leftarrow p_{stage}(s(nil), x).$

In fact, in the case of single updates, the maximum stage is simply $s(nil)$, as the fixpoint is reached after two iterations. A more complex situation will arise when considering composition of updates.

Consider now the perfect model M of the program formed by rule r_0 and $\mathcal{T}[Q](\pm p(a))$. By our construction the extension of p' in M represents the effect of the update on the extension of p. In this sense, the above simple translation of updates into rules is a declarative reconstruction of an operational semantics based on state transition.

It is worth noting that the fragments in definitions 3.2 and 3.1 realize weak updates. The code for strong updates differs only for the insert and delete rules, which have to check for absence (resp., presence) of the tuple to be inserted (resp., removed).

$p_{del}(s(nil), a) \leftarrow Q, p_{stage}(nil, a).$ $\{deletion\text{-rule}\}$
$p_{stage}(s(nil), a) \leftarrow Q, \neg p_{stage}(nil, a).$ $\{insertion\text{-rule}\}$

3.2 Composition of updates

In this section, we consider compositions of updates denoted by $u_1, \ldots, u_n$, $(n > 1)$ with reference to two different semantics: parallel and sequential evaluation of updates. According to the parallel semantics, also referred to as *non-immediate semantics*, updates are computed in two phases. During the first phases updates are collected and, in the second, they are executed all together without affecting each other.

According to the sequential semantics, also referred to as *immediate semantics*, updates are executed as soon as they are encountered. The presence of updates in a rule with immediate semantics leads to evaluate a query in a sequence of database states. Insertions and removals are immediately triggered when a body rule is satisfied, thus a single query can be evaluated on different states.

Parallel Semantics

According to this semantics, the updates $u_1, \ldots, u_n$ are evaluated concurrently without affecting each other. Therefore the code realizing parallel composition is obtained by the simple union of the programs of the single updates.

Definition 3.3 Let $u_1, \ldots, u_n$ be a composition of updates, and Q a query such that $vars(u_1, \ldots, u_n) \subseteq vars(Q)$. Then the code realizing the parallel semantics of the composition is the following:

$$\mathcal{T}_{par}[Q](u_1, \ldots, u_n) = \mathcal{T}[Q](u_1) \cup \ldots \cup \mathcal{T}[Q](u_n).$$

□

Observe that as a consequence of the union operator, the exit rules and the copy rules in the programs of the single updates occur only once in the final program.

As an example, let us consider the update of an attribute of a tuple; it can be modeled with the parallel composition of the deletion of the old tuple and the insertion of the modified tuple Let p be an EDB predicate and a and b tuples. The code realizing the update of tuple a into tuple b is the following:

$r_1: \; p_{stage}(nil, x) \leftarrow p(x).$ $\{exit\text{-rule}\}$
$r_2: \; p_{stage}(s(nil), b) \leftarrow Q, p_{stage}(nil, \_).$ $\{insertion\text{-rule}\}$
$r_3: \; p_{del}(s(nil), a) \leftarrow Q, p_{stage}(nil, \_).$ $\{deletion\text{-rule}\}$
$r_4: \; p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x).$ $\{copy\text{-rule}\}$

Observe that the parallel composition of complementary updates $+p(a), -p(a)$, according the specified semantics, results in performing the insertion $+p(a)$.

Sequential Semantics

According to this semantics, the updates $u_1, \ldots, u_n$ are evaluated sequentially so that the updates on the same predicate affect each other. Therefore, the code realizing the single update in the sequential composition is now dependent on the set of updates on the same predicate which have already been performed.

Observe that the sequential composition differs from the parallel one only for updates of the same predicate. Given a composition of updates $u_1, \ldots, u_n$, we can rearrange it as a parallel composition of sequential composition of updates on the same predicate, without affecting the effect of the overall composition. As a consequence, it suffices to restrict ourselves to consider only sequences of updates on the same predicate.

Definition 3.4 Let p be an EDB predicate, a a tuple and Q a query such that $vars(a) \subseteq vars(Q)$. Let $u_0, \ldots, u_n$ be a composition of updates over the same EDB predicate p.

- The code realizing the insertion $u_i = +p(a)$ $(i \in [0, n])$ with respect to the query Q, denoted $\mathcal{T}_i[Q](+p(a))$ is the following:

$$\begin{array}{lll} r_1: & p_{stage}(s^{2i}(nil), x) \leftarrow p_{stage}(s^{2i-1}(nil), x). & \{exit\text{-rule}\} \\ r_2: & p_{stage}(s^{2i+1}(nil), a) \leftarrow Q, p_{stage}(s^{2i}(nil), \_). & \{insert\text{-rule}\} \\ r_3: & p_{stage}(s^{2i+1}(nil), x) \leftarrow p_{stage}(s^{2i}(nil), x), \neg p_{del}(s^{2i+1}(nil), x). & \{copy\text{-rule}\} \end{array}$$

- The code realizing the deletion $u_i = -p(a)$ $(i \in [0, n])$ with respect to the query Q, denoted $\mathcal{T}_i[Q](-p(a))$, differs only for rule r_2:

$$\begin{array}{lll} r_2: & p_{del}(s^{2i+1}(nil), a) \leftarrow Q, p_{stage}(s^{2i}(nil), \_). & \{delete\text{-rule}\} \end{array}$$

Notice that, for $i = 0$, the clause r_1 is

$$p_{stage}(s^0(nil), x) \leftarrow p_{stage}(s^{-1}(nil), x).$$

We stipulate that $p_{stage}(s^{-1}(nil), x)$ stands for $p(x)$. Since $s^0(nil) = nil$, the clause r_1 becomes the ordinary exit-rule of definition 3.2 and 3.1:

$$p_{stage}(nil, x) \leftarrow p(x).$$

□

Notice that there are two differences with the simple updates of definitions 3.1 and 3.2: a different exit rule r_2 has been added; and the code is parametric with respect to the number of occurrences of updates on the same predicate. Rule r_2 records the result of the last update on the same predicate. After i updates on predicate p, $s^{2i}(nil)$ is the maximum stage argument of p_{stage}.

The composition of updates according to the sequential semantics is given by the following definition.

Definition 3.5 Let $u_1, \ldots, u_n$ be a composition of updates over the same EDB predicate p, and Q a query such that $vars(u_1, \ldots, u_n) \subseteq vars(Q)$. Then the code realizing the parallel semantics of the composition is the following:

$$\mathcal{T}_{seq}[Q](u_1, \ldots, u_n) = \mathcal{T}_1[Q](u_1) \cup \ldots \cup \mathcal{T}_n[Q](u_n).$$

□

As an example, let us consider again the update of an attribute of a tuple; it can be modeled with the deletion of the old tuple and the insertion of the new tuple.

Let p be an EDB predicate and a and b tuples. The code realizing the update of tuple a into tuple b according to sequential semantics is the following:

$$\begin{array}{lll} r_1: & p_{stage}(nil, x) \leftarrow p(x).. & \{exit\text{-rule}\} \\ r_2: & p_{del}(s(nil), a) \leftarrow Q, p_{stage}(nil, \_). & \{deletion\text{-rule}\} \\ r_3: & p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x). & \{copy\text{-rule}\} \\ r_4: & p_{stage}(s(s(nil)), x) \leftarrow p_{stage}(s(nil), x). & \{exit\text{-rule}\} \\ r_5: & p_{stage}(s(s(nil)), b) \leftarrow Q, p_{stage}(s(nil), \_). & \{insertion\text{-rule}\} \\ r_6: & p_{stage}(s(s(nil)), x) \leftarrow p_{stage}(s(nil), x), \neg p_{del}(s(s(nil)), x). & \{copy\text{-rule}\} \end{array}$$

3.3 Simple Transactions

In this section we tackle the problem of integrating updates and queries. We propose how to integrate the two modalities of interacting with the deductive database in a unique framework which can be executed by a fixpoint evaluation. Such framework defines the concept of a transaction, which will be introduced gradually: simple transactions and nested transactions.

A simple transaction is a single rule of the form:

$$h \leftarrow pre, u_1, \ldots, u_m, post.$$

where $u_1, \ldots, u_m$ is a composition of updates, pre and $post$ are queries, and the predicate symbol in the head h, called a *transaction* predicate, is a fresh predicate symbol, which does not occur anywhere else in the program. pre is called the precondition of the transaction, and $post$ the postcondition. It is worth noting that preconditions and postconditions play the role of integrity constraints, and the interesting case is when the same predicate is involved both in pre/postconditions and in updates.

As in the case of multiple updates, the parallel and sequential semantics of transactions behave differently, and therefore they are considered separately.

Parallel Semantics

In this case, each single update is constrained by the success of the precondition, so that they have to be evaluated before the execution of every update. Moreover, pre is needed to provide the actual tuples to be inserted/removed. Therefore, we use the code $\mathcal{T}_{par}[pre](u)$ of definition 3.3 instantiated on the precondition pre of the transaction.

Next, we have to take into consideration the postcondition. In fact, the success of the transaction, as well as the possibility of inferring facts of the transaction predicate h, is subject to the satisfaction of the query $post$. However, the evaluation of $post$ must take into account the effect of the updates on the extensional predicates. To this purpose, we use the following *derivation-rule* r_d for h:

$$r_d: \quad h \leftarrow pre, post'. \qquad \{derivation\text{-rule}\}$$

where $post'$ denotes the query $post$ evaluated with respect to the program modified by replacing every occurrence of an extensional predicate p in a rule with the predicate p', denoting the final extension of p after the updates. In the case of parallel semantics, p' can be simply defined as follows:

$$p'(x) \leftarrow p_{stage}(s(nil), x).$$

Finally, the code for a transaction $h \leftarrow pre, u_1, \ldots, u_m, post$ under a parallel semantics is obtained as follows, by cumulating the compilation of the parallel composition of updates with the derivation-rule r_d:

$$\mathcal{T}_{par}(h \leftarrow pre, u_1, \ldots, u_m, post.) = \mathcal{T}_{par}[pre](u_1, \ldots, u_m) \cup \{r_d\}.$$

As a simple example, consider the following $\mathcal{LDL}$ transaction on an EDB relation

$$emp(name, dept)$$

which transfers all employees of the toy department to the shoe department:

$$tr: \quad transf(x) \leftarrow emp(x, toy), -emp(x, toy), +emp(x, shoe).$$

According to the proposed compilation scheme, we obtain the following code for $\mathcal{T}_{par}(tr)$:

$r_1: \ emp_{stage}(nil, x, d) \leftarrow emp(x, d).$
$r_2: \ emp_{del}(s(nil), x, toy) \leftarrow emp(x, toy), emp_{stage}(nil, \_, \_).$
$r_3: \ emp_{stage}(s(nil), x, d) \leftarrow emp_{stage}(nil, x, d), \neg emp_{del}(s(nil), x, d).$
$r_4: \ emp_{stage}(s(nil), x, shoe) \leftarrow emp(x, toy), emp_{stage}(nil, \_, \_).$
$r_5: \ transf(x) \leftarrow emp(x, toy).$

Observe that, in absence of postconditions (which is precisely the case in $\mathcal{LDL}$), there is no need to exploit the updated EBD predicates to compute the derivation rule r_5. We next add a postcondition to the transaction, by requiring that no more than 20 employees can be associated with the shoe department:

$$tr': \quad transf(x) \leftarrow emp(x, toy), -emp(x, toy), +emp(x, shoe), count(emp(\_, shoe)) \leq 20.$$

According to the proposed compilation scheme, we obtain for $\mathcal{T}_{par}(tr')$ the same code as above, except from the derivation rule r_5, which now becomes:

$$r_5: \ transf(x) \leftarrow emp(x, toy), count(emp'(\_, shoe)) \leq 20.$$

where emp' is the updated version of emp, namely:

$$emp'(x, d) \leftarrow emp_{stage}(s(nil), x, d).$$

Sequential Semantics

In this case, the precondition must be re-evaluated before each update in the transactions, in order to take into consideration the effect of the preceding updates. To this end, we adapt the compilation of transactions with parallel updates by simply modifying how preconditions are compiled, in a way similar to postconditions in the parallel semantics. Generalizing a notion introduced earlier, given a precondition pre (or, analogously, a postcondition $post$), we denote by pre' the query pre evaluated with respect to the program modified by replacing every occurrence of an extensional predicate p in a rule with the predicate p', denoting the current extension of p. In the case of sequential semantics, the current stage of a relation after i updates can be retrieved using $s^{2i}(nil)$ as a stage argument. However, we can avoid counting how many times a relation p has been updated by defining p' as follows:

$$p'(x) \leftarrow p_{stage}(I, x), \neg p_{stage}(s(I), x).$$

which precisely identifies the current maximum stage I for p_{stage}. We can now consider the instantiation $\mathcal{T}_{seq}[pre']$ of the compilation of definition 3.5, thus obtaining that the insert and delete rules are instantiated with pre', as required.

Under the above definition of the updated predicates p', the same *derivation-rule* r_d for h adopted in the case of parallel semantics can be used:

$$r_d: \ h \leftarrow pre, post'. \qquad \{derivation\text{-rule}\}$$

Finally, the code for a transaction $h \leftarrow pre, u_1, \ldots, u_m, post$ under a sequential semantics is obtained as follows, by cumulating the translation of the sequential composition of updates with the derivation-rule r_d:

$$\mathcal{T}_{seq}(h \leftarrow pre, u_1, \ldots, u_m, post.) = \mathcal{T}_{seq}[pre'](u_1, \ldots, u_m) \cup \{r_d\}.$$

3.4 Nested transactions

In general, transactions are nested in the sense that transaction predicates may occur in the pre- or postconditions, although recursive calls to transaction predicates are not allowed. This is the case in $\mathcal{LDL}$, where moreover postconditions are not allowed. We do not explain here in detail how nested transaction are compiled for limitation of space. However, the idea of the compilation is the following. A set of nested, non recursive transaction predicates can be repeatedly unfolded, until a single rule is obtained. A this stage, a transformation scheme which closely follows that for sequential composition can be directly applied.

4 Compiling Updates in Deductive Object-Oriented Databases

In this section, we extend the proposed compilation of updates to support some basic mechanisms of object-oriented database models (see e.g. [AHV95]), including:

- objects and object identifiers;
- classes;
- ISA hierarchies and multiple inheritance;
- dynamic binding.

In a way similar to [Zan89], we use *non determinism* and *negation* to support object identifiers. The combination of XY-stratified negation and the non-deterministic *choice* construct allows us to model uniqueness of oid's. In our approach, both the notions of object *sameness* (identical oid's) and object *equality* (identical tuples) are supported, differently from the original proposal in [Zan89], where only equality is supported.

Informally, classes are represented by predicates which correspond to EDB relation augmented with two extra arguments denoting the stage and the *oid*. Therefore, we associate with every n-ary EDB predicate p a new $(n+2)$-ary predicate $p_{class}(j, oid, p(x))$ where j is a stage and *oid* is the object identifier of the tuple x from relation p.

We admit ISA hierarchies with multiple inheritance. An ISA relation between two classes, say $p\ ISA\ q$, is modeled by the following clause:

$$q_{class}(j, oid, q(x)) \leftarrow p_{class}(j, oid, p(x, y)) \qquad \{ISA\text{-rule}\}$$

which naturally states that each object of the subclass is also an object of the superclass. Observe that y denotes the set of extra attributes of the subclass, and that the *oid* is the same in both classes. The following simple hierarchy of classes is used as a running example throughout this section. Consider the following four relations:

$$pers(Name), \quad stud(Name, Major), \quad emp(Name, Salary), \quad stud\_work(Name, Major, Salary)$$

where *pers*, *stud*, *emp*, *stud_work* abbreviate respectively person, student, employee and student worker. The hierarchy is arranged as follows:

student ISA person
employee ISA person
student-worker ISA student, employee.

Pictorially:

$$\begin{tikzcd}
 & \textit{person} \arrow[dl] \arrow[dr] & \\
\textit{student} \arrow[dr] & & \textit{employee} \arrow[dl] \\
 & \textit{student-worker} &
\end{tikzcd}$$

The clauses which model this simple hierarchy are:

$r_1: \; pers_{class}(j, oid, pers(n)) \leftarrow stud_{class}(j, oid, stud(n, m))$
$r_2: \; pers_{class}(j, oid, pers(n)) \leftarrow emp_{class}(j, oid, stud(n, s))$
$r_3: \; stud_{class}(j, oid, stud(n, m)) \leftarrow stud\_work_{class}(j, oid, stud\_work(n, m, s))$
$r_4: \; emp_{class}(j, oid, emp(n, s)) \leftarrow stud\_work_{class}(j, oid, stud\_work(n, m, s))$

The basis of our approach is to represent objects as instances of the most specialized class they dynamically belong to. In other words each object is completely specified by its *most specialized version* (*msv* in short) which contains all attributes currently (at each stage) available for the object. To this purpose, we introduce a relation

$$msv(j, oid, q(x))$$

which denotes that, at stage j, the tuple x in class q is the most specialized version of object *oid*. In our approach we require that the *msv* of each object is unique, albeit possibly different at different stages. Such property is achieved, in our model, by requiring that the ISA hierarchy is closed under intersection, i.e., for any two classes r and q which are not ISA-related, the intersection class $r \cap q$ is present. Clearly, $r \cap q$ is a subclass of both r and q, and its attributes are the union of the attributes of r and q. In the example, *stud_work* is the intersection of classes *stud* and *emp*. In a real situation, we envisage that the system automatically completes the schema with intersection classes whenever these are not explicitly specified. Other possible forms of specialization, such as partition subclasses, are not considered here but they can be easily accommodated within our framework.

Finally, the role of *msv* is to activate the deduction process which populates the classes in the whole hierarchy. For each predicate q_{class} the following clause is defined:

$$q_{class}(j, oid, q(x)) \leftarrow msv(j, oid, q(x)). \qquad \{MSV\text{-rule}\}$$

By the above MSV-rule, the *msv* of each object is inserted in the appropriate class; then, by the ISA-rules each object is propagated up in the hierarchy to the superclasses. It is worth noting how the ISA-rules defining the hierarchy from one side provide the declarative specification of subclasses, and from the operational point of view they allow to populate the database—a dual reading which is typical of logic programs.

In our example the MSV-rules are:

$r_5: \; pers_{class}(j, oid, pers(n)) \leftarrow msv(j, oid, pers(n))$
$r_6: \; stud_{class}(j, oid, stud(n, m)) \leftarrow msv(j, oid, stud(n, m))$
$r_7: \; emp_{class}(j, oid, emp(n, s)) \leftarrow msv(j, oid, emp(n, s))$
$r_8: \; stud\_work_{class}(j, oid, stud\_work(n, s, m)) \leftarrow msv(j, oid, stud\_work(n, s, m))$

We are now ready to define updates within this object-oriented framework. We consider four operations:

- $new(q(a))$: creation of a new object in class q with attributes a.
- $insert(oid, q(a))$: insertion of an existing object oid in class q with attributes a.
- $delete(oid, q(a))$: deletion of an existing object oid from class q with attributes a.
- $modify(oid, q(a), q(b))$: modification of attributes a of an existing object oid in class q with attributes b.

As for simple updates, oid, a and b may contain variables. Therefore, the code that we propose is parametric with respect to a query Q such that $vars(a, b, oid) \subseteq vars(Q)$, which provides the actual tuples to be inserted in or removed from class q.

As consequence of the assumption that each object has a unique msv, we can model updates by simply updating the msv relation, as the MSV-rules and the ISA-rules accomplish the task of reflecting the updates on the whole database. The next four definitions show how the msv relation is used to the purpose of modeling the above four update operations.

Definition 4.1 Let q be an EDB predicate, a a tuple and Q a query such that $vars(a, oid) \subseteq vars(Q)$. The code realizing the operation $new(q(a))$ with respect to the query Q, denoted $\mathcal{T}[Q](new(q(a)))$, is the following:

$$msv(s(j), oid, q(a)) \leftarrow Q, idc(oid), \neg msv(j, oid, \_), choice((j), (oid)). \{insertion\text{-rule}\}$$
$$msv(s(j), oid, X) \leftarrow msv(j, oid, X), \neg msv_{del}(s(j), oid, X). \qquad \{copy\text{-rule}\}$$

□

In the insertion rule, the relation idc is the domain of object identifiers. According to this rule, a new oid is chosen which is not already in use at stage j, and associated to tuple a in class q. The $choice$ operator allows us to non deterministically select, at each stage, exactly one of the unused identifiers. As usual, the copy rule passes to the next stage all msv's of objects which have not been canceled. Observe that the insertion rule is applicable when the database is empty, i.e., when no fact $msv(nil, \_, \_)$ is present—thus reflecting the fact that the database can be load by a sequence of new operations.

In our example the operation $new(stud(smith, physics))$ generates the following insertion rule:

$$msv(s(j), oid, stud(smith, phisics)) \leftarrow idc(oid), \neg msv(j, oid, \_), choice((j), (oid)).$$

This rule, together with the MSV-rule r_6 has the effect of inserting the new object in class $stud$. Finally. the ISA-rule r_1 inserts the same object in class $pers$.

Definition 4.2 Let q be an EDB predicate, a a tuple and Q a query such that $vars(a) \subseteq vars(Q)$. The code realizing the operation $insert(oid, q(a))$ with respect to the query Q, denoted $\mathcal{T}[Q](insert(oid, q(a)))$, is composed by three sets of rules.

1. For each superclass r of q the following rules are defined:

$$msv(s(j), oid, q(a)) \leftarrow Q, msv(j, oid, r(b)). \qquad \{insertion\text{-rule}\}$$
$$msv_{del}(s(j), oid, r(b)) \leftarrow Q, msv(j, oid, r(b)). \qquad \{deletion\text{-rule}\}$$

2. For each class r not ISA-related with class q, the following rules are defined:

$$msv(s(j), oid, q \cap r(a,b)) \leftarrow Q, msv(j, oid, r(b)). \qquad \{insertion\text{-rule}\}$$
$$msv_{del}(s(j), oid, r(b)) \leftarrow Q, msv(j, oid, r(b)). \qquad \{deletion\text{-rule}\}$$

3. $$msv(s(j), oid, X) \leftarrow msv(j, oid, X), \neg msv_{del}(s(j), oid, X). \qquad \{copy\text{-rule}\}$$

□

The insertion rule in the first set models the specialization of the object *oid* from superclass r to subclass q. The associated deletion rules delete the previous *msv* of the object *oid*. The insertion rule in the second set deals with the possibility of inserting the object *oid* in a class q while its current *msv* belongs to class r which is not ISA-related with q. In this case the new *msv* of the object belongs to the intersection class of q and r.

In our example the operation $insert(oid, stud(greene, math))$ generates the following insertion and deletion rules:

$$msv(s(j), oid, stud(greene, math)) \leftarrow msv(j, oid, pers(greene)).$$
$$msv_{del}(s(j), oid, pers(greene)) \leftarrow msv(j, oid, pers(greene)).$$
$$msv(s(j), oid, stud\_work(greene, math, X)) \leftarrow msv(j, oid, emp(greene, X)).$$
$$msv_{del}(s(j), oid, emp(greene, X)) \leftarrow msv(j, oid, emp(greene, X)).$$

Observe that, by the uniqueness of the *msv* relation, for each *oid* we have that at most one insertion rule is applicable, and analogously for the deletion rules. In fact, if either the student *oid* is not present or its current *msv* belongs to *stud* or to *stud_work*, then none of the above clauses is applicable and therefore the $insertion(oid, stud(greene, math))$ is not executed. As in the case of creation, due to the rules r_6 and r_1, the object is inserted in class *stud* and *pers* with the same identifier.

Definition 4.3 Let q be an EDB predicate which is not defined by multiple inheritance, a a tuple and Q a query such that $vars(a, oid) \subseteq vars(Q)$. The code realizing the operation $delete(oid, q(a))$ with respect to the query Q, denoted $\mathcal{T}[Q](delete(oid, q(a)))$, is composed by three sets of rules.

1. for each subclass r of q (possibly $r = q$) the following rule is defined

$$msv_{del}(s(j), oid, r(a,b)) \leftarrow Q, msv(j, oid, r(a,b)). \qquad \{deletion\text{-rule}\}$$

2. If q ISA p, i.e. q is not the root of the hierarchy, for each subclass r of q which is either a non-intersection class or the intersection of two subclasses of q the following rule is defined:

$$msv(s(j), oid, p(c)) \leftarrow Q, msv(j, oid, r(a,b)). \qquad \{insertion\text{-rule}\}$$

3. For each subclass r of q (possibly $r = q$) and for each class p not ISA-related with q, the following rule is defined:

$$msv(s(j), oid, p(c)) \leftarrow Q, msv(j, oid, p \cap r(a,b)). \qquad \{insertion\text{-rule}\}$$

4. $$msv(s(j), oid, X) \leftarrow msv(j, oid, X), \neg msv_{del}(s(j), oid, X). \qquad \{copy\text{-rule}\}$$

□

The deletion rule in the first set removes the *msv* of object *oid* if it belongs to a subclass of q or to q itself. To generate the new *msv* of *oid* three cases are considered. If q is the root class of the hierarchy, the *oid* is removed from the database as no insertion rule is defined. If q is the specialization of a unique class p, then the new *msv* of *oid* belongs to p. In any other case, by the hypothesis of closure under intersection of the hierarchy, the current *msv* of *oid* belongs to the intersection of a subclass of q and some relation p which is not ISA-related with q. In this case the new *msv* of *oid* belongs to p.

In our example the operation $delete(oid, stud(\_))$ translates to the following deletion and insertion rules:

$$\begin{array}{l} msv_{del}(s(j), oid, stud(n,m)) \leftarrow msv(j, oid, stud(n,m)). \\ msv_{del}(s(j), oid, stud\_work(n,m,s)) \leftarrow msv(j, oid, stud\_work(n,m,s)). \\ msv(s(j), oid, pers(n)) \leftarrow msv(j, oid, stud(n,m)). \\ msv(s(j), oid, emp(n,s)) \leftarrow msv(j, oid, stud\_work(n,m,s)). \end{array}$$

Observe that two cases are possible. If the current *msv* of *oid* belongs to *student*, then the new *msv* of *oid* will end up in *pers*. Otherwise, if current *msv* of *oid* belongs to *stud_work*, then the new *msv* will end up in *emp*. It is worth noting that the constraint that deletion cannot be applied to intersection classes is needed to guarantee the uniqueness of the *msv*. In the example, it is not allowed to delete a *stud_work* directly, but it is needed to delete it both as a *stud* and as an *emp*.

Definition 4.4 Let q be an EDB predicate, a and c tuples and Q a query such that $vars(a, c, oid) \subseteq vars(Q)$. The code realizing the operation $modify(oid, q(a), q(c))$ with respect to the query Q, denoted $\mathcal{T}[Q](modify(oid, q(a), q(c)))$, is composed by two sets of rules.

1. For each subclass r of q (possibly $r = q$) the following two rules are defined:

$$\begin{array}{ll} msv(s(j), oid, r(c,b)) \leftarrow Q, msv(j, oid, r(a,b)). & \{insertion\text{-rule}\} \\ msv_{del}(s(j), oid, r(a,b)) \leftarrow Q, msv(j, oid, r(a,b)). & \{deletion\text{-rule}\} \end{array}$$

2. $msv(s(j), oid, X) \leftarrow msv(j, oid, X), \neg msv_{del}(s(j), oid, X).$ $\{copy\text{-rule}\}$

□

In the case of the modify operation, the clauses from the first set retrieve the current *msv* of the object to be modified, from some subclass of q, delete the current *msv* and insert the modified *msv* in the same class with the same oid.

As a final remark on the compilation technique, observe that the compilation of compositions of updates and that of transactions is analogous to that presented in Section 3, and therefore omitted.

5 Final Remarks

We proposed in this paper a compilation of updates and transactions based on their declarative reconstruction in terms of XY-stratified programs. Despite its simplicity, the proposed compilation produces code that can be efficiently executed by a machine supporting bottom-up execution of XY-stratified programs, such as that of $\mathcal{LDL}$++, the successor of $\mathcal{LDL}$ [AOTZ93]. In particular:

- the stage arguments can be actually implemented as a single counting variable, global to the database, thus avoiding the overhead of the copy rules;
- the compilation technique directly support virtual updates, which can be actually executed after the transaction commits.

Various more general forms of transactions and other dynamic aspects of databases can be supported on the basis of the proposed technique, and are currently under investigation. These include recursive transactions and active rules, both in the purely deductive and the object-oriented case. Preliminary investigations show that the proposed compilation can be adapted to this extended framework.

References

[AHV95] S. Abitebul, R. Hull, V. Vianu. *Foundation of Databases*, Addison-Wesley Publishing Company (1995)

[AOTZ93] N. Arni, K. Ong, S.Tsur, C.Zaniolo. *$\mathcal{LDL}$ ++: A Second Generation Deductive Databases System*, MCC Technical Report, Austin, Texas (1993)

[AOZ93] N. Arni, K. Ong, C. Zaniolo. *Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$ ++ Approach*, In Proceeding of Deductive and Object-Oriented Databases – Third International Conference, Springer-Verlag, (Ed. S. Ceri, K. Tanaka, Shalom Tsur), LNCS, pagg. 204-221 (1993)

[BK93] A. J. Bonner and M. Kifer. *Transaction Logic Programming*, Technical Report CRSI-270, Computer System Research Institute, University of Toronto.(1993)

[CF94] M. Carboni, V. Foddai *Aspetti dinamici delle basi di dati deduttive*, Laurea Thesis. Dipartimento di Informatica, Università di Pisa. 1994 (in Italian)

[CHO93] M. Baudinet, J.Chomicki, P. Wolper, *Temporal Deductive Databases*, in *Temporal Databases*, eds. Tansel, Clifford, Gadia, Jajodia, Sagev, Snodgrass, Benjamin and Cummings (1993)

[CGP93] L. Corciulo, F. Giannotti and D. Pedreschi. Datalog with Non-deterministic Choice Computes *NDB-PTIME*. In: S. Ceri, T. Katsumi, and S. Tsur, eds., *Proc. of DOOD'93, Third Int. Conf. on Deductive and Object-oriented Databases*, Lecture Notes in Computer Science, Vol. 760 (Springer, Berlin, 1993) 49-65.

[GPSZ91] F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo. *Non-Determinism in Deductive Databases*, In Proceeding of Deductive and Object-Oriented Databases Second International Conference, Springer-Verlag, (Ed. C. Delobel, M. Kifer, Y. Masunga), LNCS, pagg. 129-146 (1991)

[KN88] R. Krishnamurthy, S. Naqvi. *Non-Deterministic Choice in Datalog*. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Morgan Kaufmann Pub., Los Altos (1988). pp. 416-424.

[LL93] G. Lausen, B. Ludäscher. *Updates by Reasoning about States*, Second Compunet Workshop on Deductive Databases, Athens (1993)

[Mon93] D. Montesi, *A Model for Updates and Transactions in Deductive Databases.* PhD. thesis, Dipartimento di Informatica, Università di Pisa (1993)

[MW86] S. Manchanda and D. S. Warren. A logic-based Language for Database Updates. In: J. Minker editor, *Foundations of Deductive Databases and Logic Programming*, (Springer-verlag, Berlin, 1986) 363-394.

[NT88] S. Naqvi, S. Tsur. *A Logic Language for Data and Knoledge Bases*, Computer Science Press, NewYork (1988)

[Zan89] C. Zaniolo. *Object-Identity and Inheritance in Deductive Databases- an evolutionary approach*, In Proceeding of Deductive and Object-Oriented Databases Conference, Kyoto (1989)