Technical Report: TR-96-05, February 26, 1996

A Logic Abstract Machine for Active Object-Oriented Databases

Carboni, Marilisa Giannotti, Fosca Manco, Giuseppe Pedreschi, Dino

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

We show how a fragment of the logical data language LDL++ can be used to define an abstract machine supporting the essence of active/deductive object-oriented databases. We exhibit a compilation into the mentioned logic language of the basic features of the object model, including the schema definition language, the query language with multiple roles, the basic update operations, and a form of active rules. The target logic language, which is essentially Datalog extended with non-determinism and a form of stratified negation, has been chosen with a twofold aim. On one side, it should provide an abstract implementation level, where the object model is declaratively reconstructed and its semantics clarified. On the other side, the proposed compilation should form the basis of a realistic implementation, as LDL++ can be efficiently executed, and supports real side effects.

ACM–CR Subject Classification: D.1.5; D.1.6; H.2.3; I.2.3

Keyword and phrases: Logic Database Languages, Object-Oriented Data Models, Updates, Stratified Logic Programs, Non-determinism, Active Databases

A Logic Abstract Machine for Active Object-Oriented Databases *

M. Carboni[1], F. Giannotti[1], G. Manco[1] and D. 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

We show how a fragment of the logical data language $\mathcal{LDL}$++ can be used to define an abstract machine supporting the essence of active/deductive object-oriented databases. We exhibit a compilation into the mentioned logic language of the basic features of the object model, including the schema definition language, the query language with multiple roles, the basic update operations, and a form of active rules. The target logic language, which is essentially Datalog extended with non-determinism and a form of stratified negation, has been chosen with a twofold aim. On one side, it should provide an abstract implementation level, where the object model is declaratively reconstructed and its semantics clarified. On the other side, the proposed compilation should form the basis of a realistic implementation, as $\mathcal{LDL}$++ can be efficiently executed, and supports real side effects.

Keywords. Logic Database Languages, Object-Oriented Data Models, Updates, Stratified Logic Programs, Non-determinism, Active Databases.

1 Introduction

This paper tries to identify an abstract machine capable of supporting the mechanisms of an active/deductive object-oriented data model. This project is rather ambitious: we are interested in finding an appropriate abstraction level for a twofold purpose.

1. On one side, such a machine should provide an abstract implementation level, where the object model is declaratively reconstructed, and its integration with actions and deductions clarified.

*Work partially supported by the EC-US Cooperative Activity Project ECUS-033 – Deus Ex Machina.

2. On the other side, the machine should provide the basis of a realistic implementation, at least at a prototype level.

In this paper, we consider a natural extension of Datalog with a single unary function and a non-deterministic construct, which corresponds to a fragment of the logical data language $\mathcal{LDL}$++ [AOTZ93], as a candidate intermediate language for constructing the mentioned abstract machine. Next, we consider a small language embodying the essential aspects of an active/deductive object data model, and provide a *compilation* of this language into the mentioned fragment of $\mathcal{LDL}$++. During this experiment, we found encouraging evidence that the adopted logic language is adequate to meet purposes 1 and 2 above.

On one side, it is high-level enough to allow a formal specification of the crucial aspects, such the design of the object store, the inheritance mechanisms, the deductive query processing, the semantics of active rules, and the integration of these various competing paradigms in a coherent framework—from this point of view, it is helpful that the semantics of the adopted fragment of $\mathcal{LDL}$++ is assigned in purely declarative terms [SZ90, ZAO93].

On the other side, the proposed compilation is low-level enough to form the basis of a realistic (prototype) implementation, as $\mathcal{LDL}$++ can be efficiently executed by means of a fixpoint procedure with real side-effects to support updates.

The static component of the considered object model, i.e., its data definition and query language, can be seen as a fragment of *F-logic* [BK93], and provides mechanisms for multiple inheritance, multiple roles, virtual objects, views, and methods (derived attributes). The dynamic component of the considered object model, i.e. the basic update and object-migration operations, is inspired by the analogous features of the Fibonacci language [ABGO93], but comprises also a simple form of active rules, or *condition-action* rules.

The object model is compiled into an extension of deductive databases with

- non determinism, used to realize object identifiers, in a way similar to [Zan89], and the non-deterministic semantics of active rules, and
- a form of stratification, used to realize updates, object-migration and, in general, actions against the database.

The code obtained as a result of the compilation can be understood in declarative terms, thus providing a formal interpretation of the source object-oriented program. But more importantly, such a code can be executed by the ordinary deductive, fixpoint-based computation integrated with efficient support for updates by means of side effects. A major advantage of this opportunity is the fact that the architecture of the abstract machine supporting deductive databases is left unaltered and, as a consequence, the available optimization techniques, such as magic sets, are directly applicable.

Many proposals in literature enhance the standard semantics of deductive databases, in order to support the object-oriented paradigm, but a reference model is missing, which supports static and dynamic aspects in a uniform and simple framework. There are proposals which tackle only static (structural) aspects of an object-oriented data model. This is the case, for example, of the LOGIN proposal [AKN86], in which a new unification algorithm is defined to deal with structural inheritance, or of its extension F-logic [KLW93], in which a model-theoretic semantics is given in terms of F-structures, and a notion of object-identifier is modeled by way of existential quantification. As far as the dynamic aspects are concerned, there are proposals which provide a procedural semantics, as in the case of [Mon93, BGM95], and other proposals provide ad-hoc model theoretic semantics related to modal logic [BK93, MW86, Man89, Che91]. In the former case, a poor integration of the various aspects is achieved, whereas in the latter a close relationship to real computing is missing. Moreover, it is difficult to combine the structural and dynamic aspects of the object data model with deduction. For instance, the F-logic combined with transaction logic leads to a rather complicated semantic model, due to the need of combining F-structures with path-structures. Also, it is not clear how to handle more sophisticated dynamic features, such as role-dynamics and object-migration, which have been recognized as important issues in the object-oriented paradigm [ABGO93, RS90, Su91, WdS95].

2 XY-Stratification and Nondeterminism

The basic tool used in this paper is a fragment of the $\mathcal{LDL}++$ language, consisting of *Datalog* augmented with a non-deterministic *choice* construct [GPSZ91] and a form of stratification called XY-stratification [AOTZ93].

Choice goals are used to non-deterministically select subsets of answers to queries which obey a specified FD constraint. For instance, the rule

$$st\_ad(St, Ad) \leftarrow major(St, Area), faculty(Ad, Area), choice((St), (Ad)).$$

assigns to each student a unique advisor from the same area, as the choice goal restricts the *st_ad* relation to obey the FD $(St \rightarrow Ad)$. Choice programs have a natural declarative reading, which can be formalized using stable-model semantics [SZ90], and can be efficiently executed by enforcing the FD constraints during the bottom-up computation [GPSZ91]. To the end of this paper, the choice construct is used for two purposes: to assign unique identifiers to objects, and to specify the non-deterministic semantics of active rules.

XY-stratification has been introduced in [AOTZ93] as a syntactically decidable class of programs extending stratified programs. Recursive predicates use a special argument, called *stage* which is a natural number used to count the stages of the computation, and to enforce local stratification of recursive programs. Two kinds of rules are allowed:

- X-rules, where all stage arguments are the same variable, say J, and
- Y-rules, where a stage $s(J)$ is used in the head goals, and either $s(J)$ or J are used in body goals.

Recursion is allowed only in Y-rules, on decreasing stages.

As an example, consider the problem of computing the maximum of an array $a[0, n-1]$ of integers, represented as a relation

$$a(0, x_0), \ldots, a(n-1, x_{n-1})$$

An XY-stratified program which solves the problem is the following:

$$max(0, X) \leftarrow a(0, X). \qquad (m_1)$$
$$max(s(J), Y) \leftarrow max(J, X), a(s(J), Y), Y > X. \qquad (m_2)$$
$$del(s(J), X) \leftarrow max(J, X), a(s(J), Y), Y > X. \qquad (m_3)$$
$$max(s(J), X) \leftarrow max(J, X), \neg del(s(J), X). \qquad (m_4)$$

The intuitive meaning of this program is quite obvious: at each stage J, $max(J, X)$ holds if X is the maximum of the array $[0, J]$. Notice the use of the *del* relation to specify when the current maximum value changes from a stage to the next one. This interpretation is formalized by the notion of perfect model, characterized by an intended fixpoint procedure which computes one stratum at a time starting from stage 0.

Although useful to understand programs, this abstract procedural view is hopelessly inefficient, due to the enormous amount of copying needed at each stage. However, a smarter procedural interpretation allows us to avoid copying, based on the observation that new values can be produced starting from values in the current stage only, and therefore the old stages may be discarded. According to this interpretation, the stage argument is represented by a single variable which is incremented at each execution of the Y-rules. The only real computation needed, when the body of rules (m_2) and (m_3) is true, corresponds to inserting a new fact $max(Y)$ and deleting the old fact $max(X)$, in order to obtain the correct extension of the relation max at the current stage. In other words, the effect of Y-rules can be implemented in terms of real side-effects, thus making it viable to use XY-stratified programs for database programming. In our example, the execution of the proposed program boils down to the efficient iterative computation of the following Pascal-like program:

$max := a[0]$;
for $j := 1$ **to** $n - 1$ **do**
  if $max < a[j]$ **then** $max := a[j]$

For this reasons, XY-stratified programs play a central role in the reconstruction of the dynamics aspects of the object model, as discussed later.

3 An Active/Deductive Object-Oriented Data Model

We consider here a set of mechanisms supporting the relevant aspects of active object-oriented databases, concentrating on those which received a certain consensus in the literature (see for instance [Abi92a, AHV95].)

- Object identity: objects in any class are identified by a unique *oid*.
- Multiple roles: an object belonging to a class can be viewed as playing a role in that class. The set of classes in which an object lives, at a given stage, specifies the set of roles of that object. Objects can dynamically change their roles.
- Multiple structural inheritance: the structure of objects propagates down from classes to subclasses; this form of inheritance is monotonic in the sense that any additional attribute or method specified for a subclass is added to the structure inherited from the superclass(es).

$atom$	::=	$pred\_name(term) \mid oid : class\_name[attrib\_val]$
$clause$	::=	$atom \leftarrow query. \mid class\_name$ **isa** $class\_name. \mid class\_name[attrib\_def].$
$query$	::=	$\mathtt{true} \mid atom \mid query \wedge query$
$attrib\_val$	::=	$attrib\_name \rightarrow term \mid mthd\_name@term \mid attrib\_val; attrib\_val$
$attrib\_def$	::=	$attrib\_name \Rightarrow attrib\_type \mid attrib\_def; attrib\_def$
$attrib\_type$	::=	$\mathtt{string} \mid \mathtt{int} \mid class\_name$

Figure 1: Abstract syntax for the object-oriented deductive model.

- Virtual objects and views: new classes can be derived from others by means of (deductive) rules, by specifying their extension on the basis of pre-existing classes.
- Methods: objects may have derived attributes, whose values are computed on the basis of pre-existing objects.
- Basic update operations: primitives for object creation, deletion and role migration are provided, whose effects propagates to the whole schema by means of inheritance and derivation rules.
- Active rules: a simple form of *condition-action* rules is used to specify the dynamics. Active rules describe how (sequences of) basic update operations are triggered on the of basis of conditions.

For limitation of space, we choose *not* to discuss some other relevant aspects, listed below.

- Overriding inheritance and late binding: as a consequence, we assume that attributes and methods are not redefined in subclasses. However, these features can be appropriately accommodated in our approach with minor modifications.
- Structured attribute types: sequence and tuple types are not considered here, although they can be easily modeled using complex objects.
- Static type checking: this is actually an orthogonal issue to our approach, and is outside the scope of this paper.
- ECA rules: a thorough treatment of event-triggered active rules is outside the scope of the paper, although the techniques presented in this paper are amenable to be extended with a compilation for ECA rules, such as that in [Zan95].
- Transactions: a systematic treatment of transactions in logic databases without active rules has been carried out in [CFGP95]. We plan to combine the techniques for transaction and active rules in a unifying framework.

$person[name \Rightarrow string]$. $department[mngr \Rightarrow employee; assistant \Rightarrow person]$.

$student[affiliation \Rightarrow school]$. $employee[salary \Rightarrow integer; affiliation \Rightarrow department]$.

$researcher[aPaper \Rightarrow article]$. $school[about \Rightarrow string; aStudent \Rightarrow student]$.

employee **isa** *person*. *school* **isa** *department*.
reseacher **isa** *employee*. *student* **isa** *person*.

$$O : employee[affiliation \rightarrow D; name \rightarrow N] \quad \leftarrow \quad \begin{array}{l} D : department[assistant \rightarrow O] \\ \wedge\ O : person[name \rightarrow N]. \end{array} \qquad (r_1)$$

$$O : employee[boss@M] \quad \leftarrow \quad \begin{array}{l} O : employee[affiliation \rightarrow D] \\ \wedge\ D : department[mngr \rightarrow M]. \end{array} \qquad (r_2)$$

Figure 2: An example schema.

The syntax of the reference object-oriented (deductive) database language is shown in Fig. 1. The language is closely related to many proposals such as [KLW93, Abi92b, AKN86, BJ94]. Figure 2 shows a schema definition in this language. A schema is composed by definitions of classes and (deductive) rules.

A class can be defined in three different ways:

- by explicitly defining the set of its attributes such as, for instance, in the definition of the class *department*;
- by specialization from another class as, for instance, in the definition of the class *school* as a subclass of *department*;
- by derivation from other classes by means of conditions as, for instance, in the definition of rule (r_1), which can be read as follows: all persons which are assistants of any department can be viewed also as employees.

A few remarks about deductive rules are in order. First, objects (in specific roles) are referred to by means of atoms of the form $oid : class[attrib\_val]$ whose declarative reading is: the object identified by *oid*, viewed in the role *class*, has attribute values as specified in *attrib_val*. Second, rules are a flexible mechanism to define virtual objects and views. Rule (r_1), for instance, allows to extend the extension of class *employee* with new derived objects. Similarly, views and derived (sub)classes may be constructed. Third, incomplete information is allowed, in the sense that attribute values may be partially specified in rule heads, as for instance, in rule (r_1) where the *salary* attribute is missing. In our approach, missing attribute values are filled in with *null* values. Finally, creation of virtual objects is subject to the constraint of the *oid* uniqueness, so that no new objects are created with *oid*'s already in use.

Rule (r_2) in Fig. 2 defines a method for the *employee* class. It specifies that *boss* is a derived attribute of any *employee* object O, which can be computed as the manager of the department where O works. To comply with structural inheritance, methods are propagated to subclasses.

$$
\begin{array}{lll}
update\_op & ::= & new(pred\_name(term), var) \mid extend(oid, pred\_name(term)) \mid \\
 & & drop(oid, pred\_name(term)) \mid delete(oid, pred\_name(term)) \mid \\
 & & modify(oid, pred\_name(term)) \\
rule & ::= & \texttt{on}\ condition\ \texttt{do}\ action \\
condition & ::= & query \\
action & ::= & update\_op
\end{array}
$$

Figure 3: Abstract syntax for update operations and active rules.

The syntax of the dynamic component of the reference model is shown in Fig. 3. We consider five basic update operations:

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

Role migration is supported by the *extend* and *drop* operations. Active rules are, in our setting, a simple form of production rules, triggered by conditions. At each stage, one (instance) of the rules with an enabled condition is non-deterministically selected, and the corresponding action—a basic update operation—is executed against the database. For instance, the following active rule represents an integrity trigger, which detects the illegal situation where an employee gets more than her boss, and takes an action to repair the inconsistency:

$$
\begin{array}{ll}
\texttt{on} & Emp : employee[salary \rightarrow ES, boss@Man] \wedge \\
 & Man : employee[salary \rightarrow MS] \wedge \\
 & ES > MS \\
\texttt{do} & modify(Emp, employee[salary \rightarrow MS])
\end{array}
$$

For reasons of simplicity, all subtleties concerning event-based rule semantics are not dealt with in this paper.

4 Compiling the Schema

In this section we use XY-stratified, non-deterministic Datalog programs to compile the reference object model, in a way inspired by [Zan89]. Informally, classes are represented by predicates which specify the class attributes augmented with two extra arguments denoting the stage and the *oid*. Therefore, we associate with every class p with n attributes, n ternary predicates $p_{class}(j, oid, f_i(x))$, $i = 1, \ldots, n$, where j is the stage, *oid* is the object identifier, f_i is the i-th attribute of the class p, and x is its value. Observe that the stage plays a role in modeling the dynamic aspects of the object model. In the compilation of the static part of the model, only X-rules are used, and therefore the stage is only used for deduction within the current state of the database.

The basis of our approach is an abstract object store where objects are represented 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 a 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 a real situation, we envisage that the system automatically completes the schema with intersection classes whenever these are not explicitly specified.

The role of *msv* is to activate the deduction process which populates the classes in the whole hierarchy. For each predicate q_{class} and for each attribute f of q the following clause is defined:

$$q_{class}(J, Oid, f(X)) \leftarrow \quad msv(J, Oid, q(Y)). \qquad \{\text{MSV Rule}\}$$

where Y is the tuple of variables corresponding to the attributes of the object Oid in class q, X is the variable corresponding to the attribute f in the tuple Y.

EXAMPLE 1 The code corresponding to the definition of the class *employee* in Fig. 2 is the following.

$$employee_{class}(J, Oid, affiliation(A)) \leftarrow msv(J, Oid, employee(N, A, S)).$$

$$employee_{class}(J, Oid, salary(S)) \leftarrow msv(J, Oid, employee(N, A, S)).$$

$$employee_{class}(J, Oid, name(N)) \leftarrow msv(J, Oid, employee(N, A, S)).$$

□

An ISA relation between two classes, q **isa** p, is modeled in the following way. For each attribute f in the superclass p the following rule is generated:

$$p_{class}(J, Oid, f(X)) \leftarrow q_{class}(J, Oid, f(X)). \qquad \{\text{ISA Rule}\}$$

which naturally states that each object of the subclass is also an object of the superclass.

EXAMPLE 2 The code corresponding to the definition of *employee* as a subclass of *person* in Fig. 2 is the following.

$$person_{class}(J, Oid, name(N)) \leftarrow employee_{class}(J, Oid, employee(N, A, S)).$$

□

By the 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.

The above rules can be systematically generated, by gathering from the schema the following static information:

- for each definition $p[f_1 \Rightarrow t_1; \ldots; f_n \Rightarrow t_n]$ we associate to the label p the set of attribute-type pairs $\{\langle f_1, t_1\rangle, \ldots; \langle f_n, t_n\rangle\}$;
- for each definition p **isa** q, we extend the set of pairs associated to p with those associated to q, and repeat transitively this operation w.r.t. the ISA hierarchy;
- for any pair of not ISA-related classes p and q the class $p \cap q$ is defined, whose attributes are the (disjoint) union of the attributes of p and q.

We now provide a compilation for deductive rules. We consider rules where all atoms are either of the form $oid : p[f \rightarrow t]$ or of the form $oid : p[f@t]$. Clearly, an atom $oid : p[f_1 \rightarrow t_1; \ldots; f_n \rightarrow t_n]$ in the body of a rule is equivalent to a conjunction of atoms $oid : p[f_1 \rightarrow t_1] \wedge \ldots \wedge oid : p[f_n \rightarrow t_n]$, as well as such a predicate in the head of a rules corresponds to a set of rules with identical body. The compilation function $\mathcal{T}[\![.]\!]$ is defined as follows.

$$\mathcal{T}[\![A_0 \leftarrow A_1 \wedge \ldots \wedge A_n]\!] = \{a_0 \leftarrow a, a_1, \ldots, a_n \mid \langle a_0, a\rangle \in \mathcal{H}[\![A_0]\!],\ \mathcal{B}[\![A_i]\!] = a_i\ (i = 1, \ldots, n)\}$$

where $\mathcal{H}[\![.]\!]$ and $\mathcal{B}[\![.]\!]$ are defined as follows:

- $\mathcal{H}[\![Oid : p[f \rightarrow t]]\!] = \{\langle p_{class}(J, Oid, f(t)), \neg msv(J, Oid, q(\_))\rangle \mid q \text{ is a subclass of } p\}$
- $\mathcal{H}[\![Oid : p[m@t]]\!] = \{\langle q_{class}(J, Oid, m(t)), q_{class}(J, Oid, \_)\rangle \mid q \text{ is a subclass of } p\}$;
- $\mathcal{B}[\![Oid : p[f \rightarrow t]]\!] = \mathcal{B}[\![Oid : p[f@t]]\!] = p_{class}(J, Oid, f(t))$.

Intuitively, each method is propagated downwards to the subclasses, as specified by the $\mathcal{H}$ function, which produces a conclusion for any subclass of the given one. Moreover, the $\mathcal{H}$ function yields a check of the existence of the object Oid in the class where the method is defined—in a way which recalls the use of `self` in o-o languages. Conversely, the conclusions of derivation rules are only propagated upwards, by means of the ISA-rules. It is needed, however, to check that virtual objects are created only if not already existing: this is performed by the $\mathcal{H}$ function by introducing a negated use of the msv relation.

EXAMPLE 3 The code for the rule (r_1) of Fig. 2 is the following.

$$
\begin{aligned}
employee_{class}(J, O, affiliation(D)) &\leftarrow \neg msv(J, Oid, employee(\_)),\\
&\quad department_{class}(J, D, assistant(O)),\\
&\quad person_{class}(J, O, name(N)).\\
employee_{class}(J, O, name(N)) &\leftarrow \neg msv(J, Oid, employee(\_)),\\
&\quad department_{class}(J, D, assistant(O)),\\
&\quad person_{class}(J, O, name(N)).\\
employee_{class}(J, O, affiliation(D)) &\leftarrow \neg msv(J, Oid, researcher(\_)),\\
&\quad department_{class}(J, D, assistant(O)),\\
&\quad person_{class}(J, O, name(N)).\\
employee_{class}(J, O, name(N)) &\leftarrow \neg msv(J, Oid, researcher(\_)),\\
&\quad department_{class}(J, D, assistant(O)),\\
&\quad person_{class}(J, O, name(N)).
\end{aligned}
$$

□

EXAMPLE 4 The code for the rule (r_2) of Fig. 2 is the following.

$$
\begin{aligned}
employee_{class}(J, O, boss(M)) &\leftarrow employee_{class}(J, O, \_),\\
&\quad employee_{class}(J, O, affiliation(D)),\\
&\quad department_{class}(J, D, mngr(M)).\\
researcher_{class}(J, O, boss(M)) &\leftarrow researcher_{class}(J, O, \_),\\
&\quad employee_{class}(J, O, affiliation(D)),\\
&\quad department_{class}(J, D, mngr(M)).
\end{aligned}
$$

□

5 Compiling Updates and Role Migration

We are now ready to compile updates and role migration within our object-oriented model. Notice that the previous translation was independent from the stage argument of each class predicate. Now, in order to modify the current state of the database (i.e., to add, delete or modify the objects), we need to operate on the stage argument, in a way similar to that used in [CFGP95, SZ95].

However, the effect of the basic update operations is *virtual*, in the sense that the real update will take place only if the update is in the action part of an active rule enabled for execution, in the sense explained later. Therefore, the update operations record their potential effects in a *virtual update* relation *vupd*:

$$vupd(j, oid, new\_msv, old\_msv)$$

denoting the fact that, at stage j, the old *msv* of object *oid* is modified as specified in *new_msv*. As a consequence of this design decision, the translation of the five update operations *new*, *extend*, *drop*, *delete* and *modify* is assigned by means of (deductive) rules which derive entries for the *vupd* relation. Such a translation is parametric to a query Q which provides a pre-condition for the execution of the operation. We write $\mathcal{T}[\![op]\!](Q)$ to denote the code for a generic update operation *op* w.r.t. the query Q.

The code $\mathcal{T}[\![new(p[a], Oid)]\!](Q)$ realizing the *new* operation is composed by the following rule. Notice that, if a is a partial specification of the attributes of p then a' is the complete list of the attributes of p, where the attributes unspecified in a are assigned to *null*.

$$\begin{array}{l} vupd(s(J), Oid, p(a'), null) \leftarrow Q, ide(Oid), \neg msv(J, Oid, \_), \\ \quad choice((Oid), (p(a))), choice((p(a)), (Oid)). \end{array} \qquad \{\text{NEW Rule}\}$$

In the NEW-rule, the relation *ide* 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 p. The *choice* operator allows us to non deterministically select, for each tuple, exactly one of the unused identifiers. The newly created object is inserted in the relation of virtual updates, where the *null* value in the fourth position specifies that no previous *msv* is present for the new Oid. Observe that the NEW-rule is applicable also when the database is empty, i.e., when no fact $msv(0, \_, \_)$ is present—thus reflecting the fact that the database can be loaded by a sequence of *new* operations.

Example 5 The operation $new(employee[name \rightarrow smith], Oid)$ generates the following NEW-rule:

$$\begin{array}{l} vupd(s(J), Oid, employee(smith, null, null), null) \leftarrow ide(Oid), \neg msv(J, Oid, \_), \\ \quad choice((employee(smith, null, null)), (Oid)), choice((Oid), (student(smith, null, null))). \end{array}$$

This rule, together with the MSV-rules of example 1 has the effect of inserting the new object in class *employee*. Finally, the ISA-rules of example 2 insert the same object in class *person*. □

The code $\mathcal{T}[\![delete(oid)]\!](Q)$ realizing the *delete* operation is the following:

$$vupd(s(J), oid, null, X) \leftarrow Q, msv(J, oid, X) \qquad \{\text{DELETE Rule}\}$$

Here, the value *null* in the third position of the virtual update relation denotes the fact that object *oid* is discarded.

The code $\mathcal{T}[\![extend(oid, q[a])]\!](Q)$ realizing the *extend* operation is composed by the following three sets of rules, where a' is the set of attributes of q specified in a and completed with *null*:

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

$$vupd(s(J), oid, q(a'), r(c)) \leftarrow Q, msv(J, oid, r(c)). \qquad \{\text{SPECIALIZE Rules}\}$$

2. For each subclass r of q:

$$vupd(s(J), oid, q(a), r(a, b)) \leftarrow Q, msv(J, oid, r(a, b)). \quad \{\text{GENERALIZE Rules}\}$$

3. For each class r not ISA-related with q:

$$vupd(s(J), oid, q \cap r(a', b), r(b)) \leftarrow Q, msv(J, oid, r(b)). \qquad \{\text{EXTEND Rules}\}$$

Notice that each set of rules takes two different (virtual) actions: an insertion rule, which inserts a new msv, and a deletion rule, which deletes the old msv. The SPECIALIZE-rules model the specialization of the object *oid* from superclass r to subclass p. The associated deletion rule deletes the previous msv of the object *oid*. The GENERALIZE-rules remove the msv of object *oid* if it belongs to a subclass of p, and generate the new msv in p. The EXTEND-rules deal with the possibility of inserting the object *oid* in a class p while its current msv belongs to class r which is not ISA-related with p. In this case the new msv of the object belongs to the intersection class of p and r.

EXAMPLE 6 The operation $extend(oid, student[name \rightarrow greene; school \rightarrow oid'])$ generates the following rules.

$$vupd(s(J), oid, student(greene, oid'), person(greene)) \leftarrow \\ msv(J, oid, person(greene)).$$

$$vupd(s(J), oid, student \cap employee(greene, oid', X, Y), employeee(greene, X, Y)) \leftarrow \\ msv(J, oid, employee(greene, X, Y)).$$

$$vupd(s(J), oid, student \cap researcher(greene, oid', X, Y, Z), researcher(greene, X, Y, Z)) \leftarrow \\ msv(J, oid, researcher(greene, X, Y, Z)).$$

Notice that, to complete the compilation schema, the EXTEND-rules related to *department*, *school* and others should be generated. □

Observe that, by the uniqueness of the msv relation, for each *oid* we have that at most one rule is applicable. In fact, if either the student *oid* is not present or its current msv belongs to $student$ or to $student \cap employee$, then none of the above clauses is applicable and therefore the $extend$ operation is not executed.

The code $\mathcal{T}[\![drop(oid, q[a])]\!](Q)$ realizing the *drop* operation is composed by the following rules. For each subclass r of q (possibly $r = q$), and for each class p which is either a superclass of q or a superclass of r not ISA-related with q:

$$vupd(s(J), oid, p(c), r(a, b)) \leftarrow Q, msv(J, oid, r(a, b)). \qquad \{\text{DROP Rules}\}$$

The DROP rule 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, as we postulate that $p(c) = null$ in this case. 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. 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 $person$. Otherwise, if current msv of *oid* belongs to $student \cap employee$, then the new msv will end up in $employee$. 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 $student \cap employee$ directly, but it is needed to delete it both as a $student$ and as an $employee$.

The code $\mathcal{T}[\![modify(oid, q[b])]\!](Q)$ realizing the $modify$ operation is is composed by the following rules, one for each subclass r of q (possibly $r = q$):

$$vupd(s(J), oid, r(b'), r(a)) \leftarrow Q, msv(J, oid, r(a)). \qquad \{\text{MODIFY Rule}\}$$

where the tuple b' is obtained from b by completing the missing attributes as in r.

6 Compiling Active Rules

An instance of an active rule `on` Q `do` op is enabled at stage j if its condition Q is. Among all enabled instances of rules, one is selected non-deterministically, and the corresponding action executed. Since an action is an update operation, its execution turns its effect from virtual to real. This behavior is mirrored by the following SELECT rule, which select one of the possible updates for execution, recording the selection in the *real update* relation *rupd*

$$rupd(J, Oid, New\_msv, Old\_msv) \leftarrow vupd(J, Oid, New, Old), choice((J), (Oid)).$$

$$\{\text{SELECT Rule}\}$$

The next step is to record the real update on the object store, i.e., on the *msv* relation. As a 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. This is done by means of a *frame axiom*, or COPY rules, in the style of [CFGP95, SZ95, Zan95]. Intuitively, the alive objects at the current stage are the newly created non-null ones, and those of the previous stage which have not been explicitly deleted. The following rule states the above concept: relation *rupd* records the object which has been deleted, updated and/or migrated at the current stage.

$$msv(s(J), Oid, New\_msv) \leftarrow rupd(s(J), Oid, New\_msv, \_), New\_msv \neq null.$$
$$msv(s(J), Oid, Old\_msv) \leftarrow msv(J, Oid, Old\_msv), \neg rupd(s(J), Oid, \_, Old\_msv).$$

$$\{\text{COPY Rules}\}$$

Notice that, as already mentioned in Section 2, the COPY rules are not implemented as rewrite rules, but rather by means of real side effects.

We are now ready to define the complete translation of an active rule `on` Q `do` op, where the condition $Q = A_1 \wedge \ldots \wedge A_n$ is a query and the action op is an update operation. The code $\mathcal{T}[\![\mathtt{on}Q\mathtt{do}op]\!]$ is simply the code $\mathcal{T}[\![op]\!](\mathcal{B}[\![A_1]\!], \ldots, \mathcal{B}[\![A_n]\!])$ for the update operation op w.r.t. the translation of the query Q, augmented with the SELECT and COPY rules.

EXAMPLE 7 The active rule:

$$\begin{array}{ll} \mathtt{on} & Emp : employee[salary \rightarrow ES, boss@Man] \wedge \\ & Man : employee[salary \rightarrow MS] \wedge \\ & ES > MS \\ \mathtt{do} & modify(Emp, employee[salary \rightarrow MS]) \end{array}$$

is translated to the following code:

$$\begin{array}{l} vupd(s(J), Emp, employee(N, MS, A), employee(N, ES, A)) \leftarrow \\ \quad employee_{class}(J, Emp, salary(ES)), \quad employee_{class}(J, Emp, boss(Man)), \\ \quad employee_{class}(J, Man, salary(MS)), ES > MS, msv(J, Emp, employee(N, ES, A)). \end{array}$$

and similarly for all subclasses of *employee*, augmented with the SELECT and COPY rules. □

7 Conclusions

In this paper, we showed how a fragment of the logical data language $\mathcal{LDL}$++ provides a flexible framework where the object-oriented, active and deductive aspects of databases can be compiled to in a uniform way. Our experiment can be seen as an attempt to design a *logic abstract machine*, providing an abstract implementation level where crucial issues such as object store management, deductive query processing and active rule semantics can be effectively tackled. Nevertheless, the abstract machine should be also close enough to actual implementations to yields, at least, efficient prototypes. A thorough experiment with the proposed compilation is currently in progress, and preliminary investigations show how the proposed method scales smoothly to important issues not covered in this paper, such as late binding and event-triggered actions.

References

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An Object Data Model with Roles. In *Proceedings of the 19th International Conference on Very Large Data Bases*, 1993.

[Abi92a] S. Abiteboul. Deductive and Object-Oriented Databases. Technical report, INRIA, 1992.

[Abi92b] S. Abiteboul. Towards a Deductive Object-Oriented Databases. In *Proceedings of the Int. Conf. on Deductive and Object-Oriented Databases*, pages 453–472. W. Kim and Nicolas, J.-M. and S. Nishio, 1992.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AKN86] H. Aït Kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-in Inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

[AOTZ93] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. $\mathcal{LDL}$++: A Second Generation Deductive Databases Systems. Technical report, MCC Corporation, 1993.

[BGM95] E. Bertino, G. Guerrini, and D. Montesi. Deductive Object Databases. In *Proocedings of ECOOP'95*, 1995.

[BJ94] M. Bugliesi and H. M. Jamil. A Logic For Encapsulation in Obiect-Oriented Languages. In Maria Alpuente, Roberto Barbuti, and Isidro Ramos, editors, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming*, volume 2, pages 161–175, September 1994.

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

[CFGP95] M. Carboni, V. Foddai, F. Giannotti, and D. Pedreschi. Declarative Reconstruction of Updates in Logic Databases: A Compilative Approach. In M. Alpuente and M. I Sessa, editors, *Proceedings of the GULP-PRODE joint conference on Declarative Programming- GULP-PRODE95*, pages 169–180, september 1995.

[Che91] W. Chen. Declarative Specification and Evaluation of Database Updates. In *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, number 566 in Lecture Notes in Computer Science, pages 147–166, 1991.

[GPSZ91] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Non-Determinism in Deductive Databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD91)*, Lecture Notes in Computer Science, pages 129–146. Springer-Verlag, Berlin, 1991.

[KLW93] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Programming. Technical Report 93/06, Department of Computer Science, SUNY at Stony Brook, June 1993. Also appeared in Journal of ACM.

[Man89] S. Manchanda. Declarative Expression of Deductive Database Updates. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1989.

[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*, pages 363–394. Springer-Verlag, Berlin, 1986.

[RS90] J. Richardson and P. Schwarz. Aspects: Extending Objects to Support Multiple, Indipenddent Roles. Technical Report 7657, IBM Almaden Research center, 1990.

[Su91] J. Su. Dynamic Constraints and Object Migration. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.

[SZ90] D. Saccà and C. Zaniolo. Stable Models and Non-determinism in Logic Programs with Negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 205–217, 1990.

[SZ95] V. S. Subrahmanian and C. Zaniolo. Relating Stable Models and AI Planning Domains. In *Proceeding of the International Conference on Logic Programming*, 1995.

[WdS95] R. Wieringa, W. de Jonge, and P. Spruit. Using Dynamic Classes and Role Classes to Model Object Migration. *Theory and Practice of Object Systems*, 1(1):173–196, 1995.

[Zan89] C. Zaniolo. Object Identity and Inheritance in Deductive Databases - An Evolutionary Approach. In *Int. Conf. on Deductive and Object-Oriented Databases, DOOD'89*, 1989.

[Zan95] C. Zaniolo. Active Database Rules with Transaction Cunscious Stable Model Semantics. In *4th Int. Conf. on Deductive and Object-Oriented Databases*, 1995.

[ZAO93] C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: The $\mathcal{LDL}$++ Approach. In *Int. Conf. on Deductive and Object-Oriented Databases, DOOD'93*, 1993.