TOWARDS A DEDUCTIVE OBJECT-ORIENTED DATABASE LANGUAGE*

Serge Abiteboul†
Institut National de Recherche en Informatique et Automatique,
Domaine de Voluceau-Rocquencourt,
78153 Le Chesnay, France
(abitebou@inria.inria.fr)

December 30, 1993

Abstract

A language for databases with sets, tuples, lists, object identity and structural inheritance is proposed. The core language is logic-based with a fixpoint semantics. Methods with overloading and methods evaluated externally providing extensibility of the language are considered. Other important issues such as updates and the introduction of explicit control are discussed.

1 INTRODUCTION

The success of the relational database model [19, 38, 27] is certainly due to technological advances such as fast query processing or reliable concurrency control. However, we believe that a major factor in that success has been the existence of simple-to-use languages allowing the definition and manipulation of data. This has to be remembered while considering future generations of database systems.

Object-oriented database systems are now being developed, e.g., [15, 12, 22, 39, 36]. An object-oriented approach [24] is used to answer the needs of a much wider variety of applications. Most of the effort seems to be devoted to providing powerful languages for writing complex applications. This is in contrast with (what we believe was) the main reason for the success of the relational model.

Deductive databases [23, 17] are also being considered. There the focus is on "declarative" languages, thereby keeping with the philosophy of relational languages. However, most

*A preliminary version appeared in proceedings of First International Conference on Deductive Object-Oriented Databases, Kyoto, Japan, 1989
†Work supported by the Programme de Recherches Coordonnées BD3.

of the effort has been devoted to optimization issues for toy languages (such as Datalog) which do not seem to fulfil the needs of a wider range of applications than those covered by relational systems.

We believe that these two directions are not incompatible. Indeed, the present paper is an attempt to integrate features of object-oriented and deductive databases. Object-orientation serves as the basis for enlarging the range of applications that are considered. The logic-programming aspect centers around the use of a (to a large extent) declarative language. Complicated procedural aspects as found in Prolog are not incorporated. (Database languages based on resolution have been proposed, e.g., [35, 32].)

We are *not* proposing a general-purpose database programming language. (For this, see, for instance, [20, 21].) The goal is to obtain a simple-to-use, restricted database language that can be used to program a wide range of simple applications.

Let us define some features of the "goal" language. The language should allow for (1) the definition of data, (2) the manipulation of data (queries and updates) and (3) the specification of simple applications including window management. More precisely, the language should present the following features:

1. Query/update capabilities in the style of SQL should be provided, i.e., easy ways to extract and modify data,
2. Simple applications should be specifiable in the language. In particular, it should be possible to control a window manager from *within* the language,
3. The traditional separation between schema and instance should be maintained although a type system richer than that of the relational model must be provided,
4. The language should also be extensible, i.e. it should be easy to introduce new data types to fit the needs of specific applications, e.g., graphics.

Although it is rather ambitious to pursue all these goals, an important keyword is *simplicity*. It implies that for instance, we will be satisfied with little customizing, e.g., a particular menu-based interface is offered and its redefinition is not in the scope of the language.

Let us come to our proposal. An important aspect is the choice of data structure: the data model should be rich enough to describe reasonable applications. On the other hand, it should be rather simple to be understandable by nonsophisticated users. We use here the model of [5], i.e., a simple model with sets, tuples, object identity and structural inheritance. This model can be viewed as a least generalization of the models of [1, 31]. We make two additions to it. First, we introduce a list constructor. Such constructor was avoided in [5] for "safety" reasons, but is certainly useful for many applications. Also, we consider "methods". The various features that can be found in the data structure are standard in database models. (See, for instance, the survey on Semantic Database Models of Hull and King [25] or recent proposals on object-oriented database models, e.g., [12, 22, 33, 34].)

The language that we propose is rule-based. An important aspect is that it is (to a large extent) "declarative". All the user can do is specify statements of the forms:

$$\begin{array}{llll} if & < condition > & then & < assertion >; \\ while & < condition > & do & < assertion > . \end{array}$$

These statements can be viewed as *deduction rules* like those in Datalog, or as *triggers*. For simplicity, we choose a fixpoint semantics. In the fixpoint semantics, rules are applied repeatedly until a fixpoint is reached, i.e., until the database state cannot be modified by an application of a rule. More elaborate control will be provided but the semantics of the core language is a straightforward fixpoint semantics.

Let us comment on the choice of the language. The situation of the database field resembles that of the sixties. A set of functionalities that the systems should provide is slowly emerging. As in the sixties, the short-term solution is to use procedural languages. A challenge is to find an analogue of SQL for this new generation of systems, i.e. a simple language that satisfies the requirements and is easy to use. We don't claim that we provide such a language here: (i) the language that we propose is certainly much too complicated; and (ii) syntactic sugaring should be added to make it more user friendly.

There have already been proposals of high-level *query* languages for accessing object-oriented databases, e.g., [18, 16, 29]. We believe that high-level interfaces to future database systems should not be exclusively based on queries. The functionalities of the systems having been enlarged, the functionalities of the high-level interface *also* have to be reconsidered.

The language that we propose is "declarative". We don't claim that a *purely* declarative language (whatever this means) has to be the solution. Indeed, SQL is *not* declarative. However, we insist that the language should be high-level and should encourage understanding data manipulation at an abstract level. The use of rules forces us to do so.

While designing the language, strong typing is emphasized. Typing is clearly one of the main achievements of modern programming languages. This aspect has been somewhat underestimated or misunderstood in databases. We believe that it is important both for detecting many programming errors (as used in programming languages) but also for improving performances (as standard in databases). Although typing is essential, we don't want to put on the user the burden of specifying the types of everything, and in particular of rule variables. It is therefore necessary to provide type inference. As we shall see, this is not a tough requirement since the types of database objects are known.

The structural part of the model is considered in Section 2 and the core language in Section 3. These two sections mainly revisit previous proposals. The paper's contributions can be found in remaining sections. Overloading is considered in Section 4. The introduction of explicit control is discussed in Section 5. Section 6 deals with updates. Window management is briefly considered in Section 7. The last section is a conclusion.

2 THE MODEL

In this section, we present the model by examples. The model is that used in IQL [5] extended with (i) methods and (ii) list types. The language is discussed in the next section.

2.1 Classes and Relations

In the present paper, we use as running example, the *Bill of Materials* [7].

Example 2.1 A portion of the schema is given by:

$$
\begin{array}{lll}
\textit{class part}: & \textit{equal basepart} \ \vee \ \textit{compositepart} & \textit{and} \\
 & [name: string]; & \\
\textit{class basepart}: & \textit{isa part} & \textit{and} \\
 & [price: num, mass: num]; & \\
\textit{class cheapbasepart}: & \textit{isa basepart}; & \\
\textit{class compositepart}: & \textit{isa part} & \textit{and} \\
 & [assemblycost: num, massincrement: num]; & \\
 & & \\
\textit{class supplier}: & [name: string, address: country]; & \\
 & & \\
\textit{relation suppliedby}: & [p: part, s: supplier]... &
\end{array}
$$

An instance of the above schema is shown in Figure 1. Classes and relations are defined here by extension, and represented with big boxes. Dashed lines are used for relations. The small boxes with an integer in it denote objects. The reference to an object is done with a "#". For instance, there are two objects in the class *compositepart*, #1 et #2. There are 6 tuples in relation *suppliedby*, one of them being $[\#1, \#6]$. Arrows lead from oid's to their values. For instance, the value of the #1 is $[aa, 124, 50]$. Dashed arrows are used for methods that will be considered later. □

A *class* is a name for a set of *object identifiers* that have values of the corresponding *type*. (There should be no confusion between classes and types, a class is just a name for a set of oid's whereas a type specifies a particular structure.) A *relation* is a name for a set of *values* of the corresponding type. Note that we have redundancy in the model:

Why have both classes and relations?

We believe that this facilitates programming. Classes are *object-based* whereas relations are *value-based* and both features are of practical use. Objects are important for a variety of reasons: object sharing, representation of cyclic structures, identity during database evolution, etc. On the other hand, a pure object-based model leads to a cumbersome language and the explicit handling of duplicate elimination [5]. This is why we also have

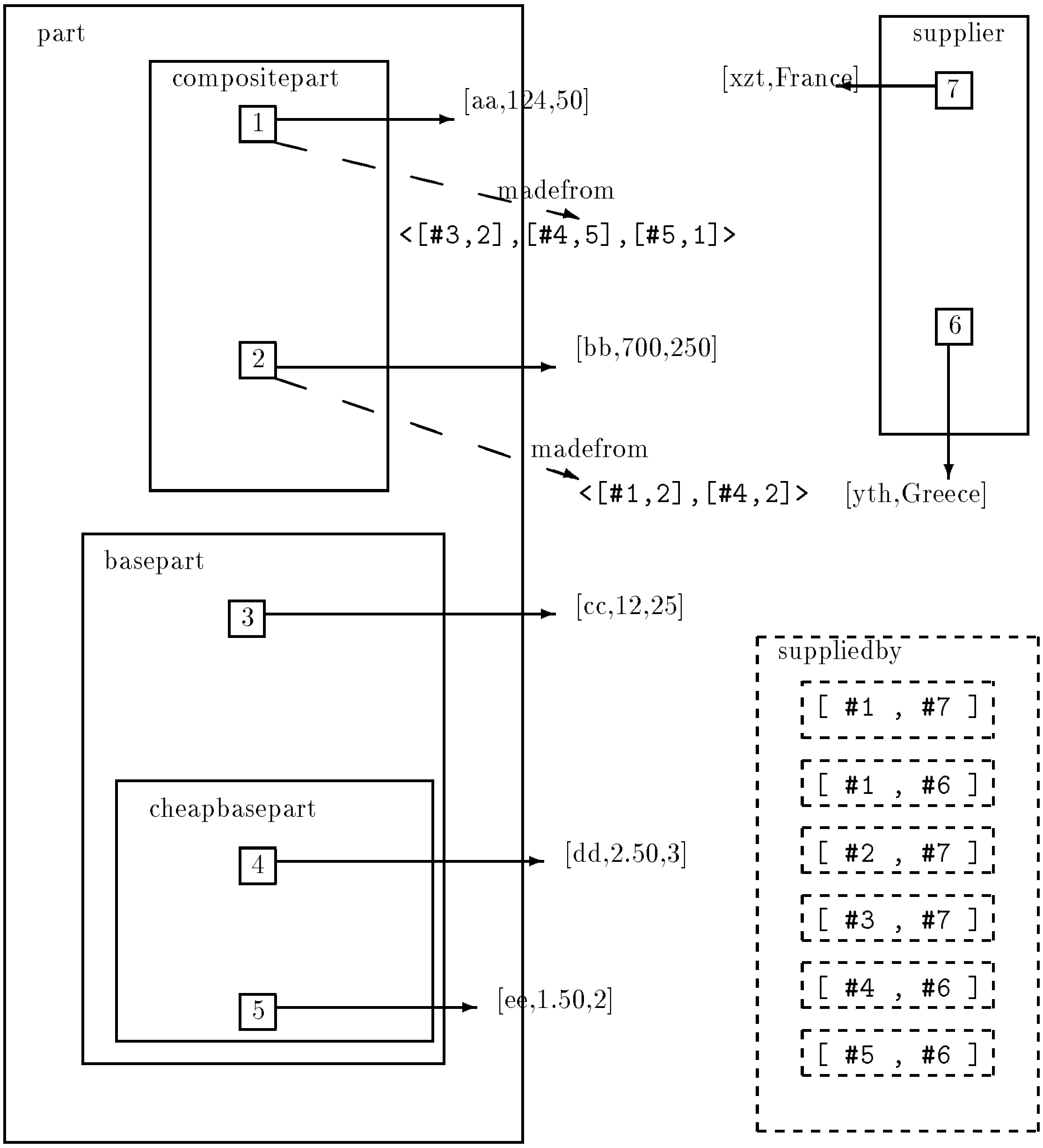

Figure 1: An instance

relations. (See [37] for a discussion of values vs. objects and [5] for more arguments for having both).

Relationships between classes are given by the schema. For instance, in the example, it is stated that a *cheapbasepart* is a *basepart* and that a *part* is either a *basepart* or a *composite part*.

The syntax of types is as in IQL. It is obtained using a given finite set of class names **P**. The set of *type expressions*, called *types*(**P**), is given by the following abstract syntax, where τ is a type expression, P an element of **P** and $k \geq 0$:

$$\tau = \emptyset \mid D \mid P \mid [A_1 : \tau, \ldots, A_k : \tau] \mid < \tau > \mid \{\tau\} \mid (\tau \vee \tau) \mid (\tau \wedge \tau)$$

(where $A_1, ..., A_k$ are from a particular set of symbols called attributes) and

$$D = int \mid num \mid real \mid string \mid bool...$$

The symbols [ ] are used for tuples, { } for finite sets and < > for finite lists. As usual $\vee$ indicates the union of types and $\wedge$ their intersection.

To give semantics to types, we first have to describe the semantics of classes. The semantics of classes is given by an oid assignment [5], i.e. an assignment of oid's to classes in **P**, using the *isa* relation on **P**. A disjoint oid assignment π gives for each class P the set of oid's that are "strictly" in P, i.e., in P and in no subclass of P.

The relation $\leq$ used in the definition is the reflexive, transitive closure of the *isa* relation specified by the schema.

Definition 2.2 A *disjoint oid assignment* π for **P** is a function mapping each name in **P** to a *finite* set of oid's such that $P \neq P'$ implies $\pi(P) \cap \pi(P') = \emptyset$ (where $P, P' \in \mathbf{P}$). The *inherited oid assignment* $\bar{\pi}$ is defined by:

$$\bar{\pi}(P) = \cup\{\pi(P') \mid P' \in \mathbf{P}, P' \leq P\} \ \text{ (for each } P). \ \Box$$

The oid assignment π in the example is:

$$\begin{array}{ll} \pi(part) = \{\ \} & \pi(compositepart) = \{\#1, \#2\} \\ \pi(basepart) = \{\#3\} & \pi(cheapbasepart) = \{\#4, \#5\} \\ \pi(supplier) = \{\#6, \#7\} & \end{array}$$

The inherited oid assignment $\bar{\pi}$ in the example is:

$$\begin{array}{ll} \bar{\pi}(part) = \{\#1, \#2, \#3, \#4, \#5\} & \bar{\pi}(compositepart) = \{\#1, \#2\} \\ \bar{\pi}(basepart) = \{\#3, \#4, \#5\} & \bar{\pi}(cheapbasepart) = \{\#4, \#5\} \\ \bar{\pi}(supplier) = \{\#6, \#7\} & \end{array}$$

Note in particular that

$$\bar{\pi}(basepart) = \pi(basepart) \cup \pi(cheapbasepart).$$

In the context of inheritance, a subtlety is that the same class symbol (e.g., *basepart*) is used to denote the objects that are strictly in that class ($\pi(basepart)$) and those that are in that class or in more specialized classes ($\bar{\pi}(basepart)$).

The use of *equal* (e.g., in the definition of *part*) allows to specify that a class has no proper extension. For instance, a *part* is either a *basepart* or a *compositepart*. Thus $\pi(part) = \emptyset$ and

$$\bar{\pi}(part) = \pi(compositepart) \cup \pi(basepart) \cup \pi(cheapbasepart).$$

The distinction between *specialization* or subsetting (*cheapbasepart isa basepart*) and *generalization* or union ($part = basepart \ \vee \ compositepart$) is in the spirit of [3]. This is perhaps nonstandard in object-oriented databases but we believe that it is an important distinction in describing applications.

The semantics of constructed types is now defined using the semantics of classes. For instance, the semantics of the type [*p:part,s:supplier*] is the set of pairs with a *p*-field containing an object in $\bar{\pi}(part)$ and an *s*-field containig an object in $\bar{\pi}(supplier)$. The declaration of the relation *suppliedby* indicates that the symbol *suppliedby* stands for a finite set of such pairs.

Let us now consider the structure of objects. A subtlety comes from the abbreviated form of type declarations in the schema. Consider for instance the declaration of *basepart*. A *basepart* has a *price* and a *mass*; but (as a *part*) it also has a *name*. The value of a *basepart* is therefore a triple. A formal definition of the types of object values can be found in [5]. We will just assume here that a type is associated with each class based on schema declarations. In the example,

- the type of *basepart* and *cheapbasepart* are: [*name:string, price:num, mass:num*];
- the type of *compositepart* is: [*name:string, assemblycost:num, massincrement:num*];

and the types of *part* and *supplier* are as given in the schema.

A popular question in the field is whether *multiple inheritance* is allowed. Multiple inheritance is the possibility for a class to have more than one direct superclasses (e.g., *James_Bond_car isa car and isa boat and isa submarine and isa plane*). We will not address this issue here although we tend to believe that so complicated features would probably have to be left out in simple models for "unsophisticated" users.

To conclude this subsection, we want to insist on the somewhat nonstandard way of specifying the semantics of types via some assignment of oid's. This allows to talk about recursive structures without introducing any recursive type constructor.

2.2 Methods

Methods are just functions satisfying certain typing constraints. These functions can be (i) stored extensionally or (ii) derived (i.e., given intentionally). (We will see a third case in

Section 3.6.)

Consider again the example. We may want to add the following declaration:

$$method\ madefrom : compositepart \rightarrow < [component : part, quantity : int] >;$$

The method *madefrom* is also stored. In the instance of Figure 1, the method *madefrom* applied to #2 returns $< [\#1, 2], [\#4, 2] >$.

It is clear that the same information can be represented by a relation, so

why have both methods and relations?

We believe that certain assertions/queries are easy to express in a relational manner whereas others are in essence functional and are not naturally described in relational terms. Compare for instance the two queries:

$$R(x,w) \leftarrow father(x,y),\ mother(y,z),\ boss(z,w),$$
$$\leftarrow R(x,john),$$

$$father(mother(boss(john)))\ ?$$

More motivations for having both methods and relations are given in [4]. Furthermore, as we shall see in Section 3.6, methods are also used to obtain an extensible language.

The essential aspect in using methods is inheritance. Specialization and generalization lead to different ways of inheriting "methods". For instance, if a method is defined on *basepart*, it is inherited by *cheapbasepart* ("downward inheritance"). On the other hand, if a method is defined on *basepart* and *compositepart*, it is inherited by *part* since a part is either a *basepart* or a *compositepart* ("upward inheritance"). Note that this can be artificially simulated in models lacking the *equal* primitive. To have a method m applying to all parts, one can (i) define m at *basepart* and *compositepart*, and (ii) define m at *part*. The definition of m at *part* is meaningless. However, it would be type incorrect to apply m to all the elements in a set of *part*s if m is not also defined at *part*.

Another important aspect of methods is *overloading*. To illustrate overloading, suppose that there is a method *cost* defined on *basepart*. That method can be "refined" on *cheapbasepart*. This means that we may have two type declarations for *cost*:

$$\begin{array}{lll} basepart & \rightarrow & [dollar : int], \\ cheapbasepart & \rightarrow & [dollar : int, cent : int]. \end{array}$$

Indeed, if *cost* is "computed", two different programs may be used for computing the cost in the two classes.

The first argument of a method (sometimes called *receiver*) plays a particular role in certain proposals. It is sometimes required that this argument alone should be sufficient to find the relevant signature *and* the code of the method. This usually comes together with viewing the various method declarations and codes as residing in classes. A motivation for this is to improve the performance both at compile time (type checking) and at runtime (dynamic binding). We prefer to disregard these motivations to have a simpler semantics. For the same reasons, we do not assume any "covariance[1]" conditions on the signatures of methods.

Overloading complicates the semantics of methods. Intuitively, such a semantics is defined given the partial order *isa* on the classes and a disjoint oid assignment π. The semantics of a method is then a function satisfying typing constraints. The impact of overloading is that the type of the output depends on the classes of the objects in the input.

Remark: It must seem at first glance that we have too many concepts whereas our goal is to have a simple model. Now, let us recapitulate the main features of the structural part of the model comparing them to the main features of the relational model:

1. We have relations as in the relational model.
2. Methods (up to now) is just a fancy way of talking about functional dependencies.
3. Classes and oid's are meant to capture referential constraints.
4. The more involved notion of type resembles that found in nested relation models (e.g., [26]) and complex object models (e.g., [1]).

Finally, some of these concepts will turn out to be essential to integrate new features in the following sections. Methods will be used to obtain extensibility and oid's are central for updates.

3 A RULE-BASED LANGUAGE

The language is built around the COL [2, 4] and IQL [5] languages. An implementation of COL is described in [9]. In this section, we are concerned with a core language roughly that of [2]. Fancier features are considered in the next sections. In particular, we assume here that methods are not overloaded (they are simply functions) and the control is provided exclusively by a fixpoint.

Two particular predicates belong to the language: equality ($=$) and ownership ($\ni$). (We also use their negation ($\neq, \not\ni$.) The following rule defines Q as the set of "all compositeparts supplied by two suppliers and having four subcomponents #23":

$$Q(x) \leftarrow composite(x), suppliedby(x,y), suppliedby(x,z), y \neq z, madefrom(x) \ni [\#23, 4].$$

[1]Let m be a method with signature $c_0 \rightarrow c_1$ that is refined in a subclass c_2 of c_0 with a new signature $c_2 \rightarrow c_3$. Covariance requires c_3 to be a subclass of c_1.

A rule has a "body" consisting of positive and negative literals (e.g., $y \neq z$ or $madefrom(x) \ni [\#23, 4]$) and a "head" consisting of a single positive literal (here, $Q(x)$). Note that the predicate $\ni$ applies both to sets to denote ownership, and to lists to denote the presence of some element in a list. Predicates are built using terms (e.g., y, z, $madefrom(x)$ or $[\#23, 4]$).

3.1 Terms

In the language, term constructors correspond to the various type constructors: if $t_1, ..., t_n$ are terms,

constants: constants, e.g., integers, are terms in the language; constant oid's are denoted by a "#", e.g., #245,

tuples: $[A_1 : t_1, ..., A_n : t_n]$ is a term,

sets: $\{ t_1, ..., t_n \}$ is a term,

lists: $< t_1, ..., t_n >$ is a term,

dereferencing: if t_1 is a term denoting an oid, $*t_1$ denotes the value of t_1,

projection: if t_1 denotes a tuple with an A-field, $t_1.A$ denotes its A-field,

cons: (for lists) $t_1 :: t_2$ is a term denoting the list consisting of t_1 followed by the list t_2,

variables: typed variables are terms (We assume the existence of a countable set of variables for each type.),

relations and classes: each relation and class name is a term,

methods: if m is a method and $t_1, ..., t_n$ are terms of appropriate types for m, $m(t_1, ..., t_n)$ is a term.

3.2 Literals and Facts

Literals are typed expressions obtained from terms using equality and membership predicates, e.g., $t = t'$, $t \ni t''$, and their negations. In $t \ni t''$, an alternative notation is $t(t'')$. For instance, *suppliedby(x,y)* can be equivalently written as $suppliedby \ni [x, y]$.

Using this, we can view an instance of the database as a set of *facts*, i.e., literals of various particular forms: e.g.,

1. for classes, *compositepart(#1)*
   stating that object #1 is a composite part;

2. for object values, $*(\#1) = [aa, 124, 50]$
   stating that object #1 has value $[aa, 124, 50]$;

3. for relations, *suppliedby*($\#1, \#7$)
   stating that the tuple $[\#1, \#7]$ is in relation *suppliedby*; and

4. for methods, $madefrom(\#2) = <[\#1, 2], [\#4, 2]>$
   stating that the value of $madefrom(\#2)$ is $<[\#1, 2], [\#4, 2]>$.

3.3 Rules

As mentioned above, rules are constructed in the standard way. A rule is an expression

$$body \leftarrow head$$

where *head* is a positive literal and *body* is a list of positive and negative literals.

For instance, the following rule defines Q' as the set of values of parts that are supplied by supplier $\#5$ and that have 4 wheels as first subcomponents:

$$Q'(*x) \leftarrow suppliedby(p : x, s : \#5), madefrom(x) = ([y, 4] :: t), y.name = \text{"}wheel\text{"}.$$

The type of Q' has to be:

$$T = [name : string, assemblycost : num, massincrement : num]$$

since the *madefrom* method applies only to *compositepart* and the value of a *compositepart* is of type T.

In general, the body of a rule consists of a set of positive and negative literals. The head consists of a positive literal of a particular form:

1. for classes, $R(x)$ where R is a class name and x a term of type R.

2. for set-valued objects, $*(x) \ni y$ where x is a term of type R for some class R and y a term of type T if $\{T\}$ is the type of the values of objects in R.

3. for monovalued objects, $*(x) = y$ where x is a term of type R for some class R and y a term of type T if T is the type of the values of objects in R.

4. for relations, $R(x_1, ..., x_n)$ where R is an n-ary relation, and each x_i is a term of appropriate type.

5. for set-valued methods, $F(x_1, ..., x_m) \ni x_{m+1}$ where F is an m-ary set-valued method and each x_i is of appropriate type.

6. for monovalued methods, $F(x_1, ..., x_m) = x_{m+1}$ where F is an m-ary monovalued method and each x_i is of appropriate type.

Note the analogy between the possible forms for heads of rules and the facts. This should not come as a surprise since derived facts will be obtained from heads of rules.

It is important to remember that all terms are typed. For instance, $t_1 :: t_2$ is meaningful only if the type of t_1 is some τ, and that of t_2 is $< \tau >$. As mentioned in the introduction, the system is responsible for infering the types of variables. However, it is useful to be able to explicitly type some variables in rules to remove ambiguity. Therefore, for each term t and each type τ, $t : \tau$ indicates that the term t has τ for type.

Remark: Named fields are used as in databases. However, attribute names can be omitted in programs. The types of the field may remove ambiguity; otherwise, the ordering given at definition-time is used by default.

3.4 Derived Methods

We next consider "derived methods". (Overloading will be considered in the following section, so the "methods" of the current section are nothing but functions.) Derived functions/methods are illustrated using two examples. In the first one, the method is set-valued, whereas in the second it is single-valued.

Example 3.1 A fact concerning a set-valued object is expressed using the ownership predicate. Consider the classical *nest* operations [26] of complex objects. The relation *suppliedby* is of type $[part, supplier]$. The parts supplied by each supplier are grouped using a method *supplies* of type $supplier \rightarrow \{part\}$ as follows:

$$supplies(x) \ni y \leftarrow suppliedby(y,x).$$

The set $supplies(x)$ is defined as the set of all parts y such that $supplies(x) \ni y$ can be derived. The above interpretation of the set-valued functions therefore follows the Closed World Assumption. More precisely, a closed-world interpretation assumes a fact to be false unless it is explicitly stated that it holds. Thus, if one cannot derive that for some a, b, $supplies(a) \ni b$, we deduce that $supplies(a) \not\ni b$. This yields the exact value of $supplies(a)$, i.e., the set of all b's such that one can derive that $supplies(a) \ni b$.

The next rule organizes the same information in a relation S of type $[supplier, \{part\}]$:

$$S(x, supplies(x)) \leftarrow supplier(x).$$

Objects can also be set-valued. For instance, let P be a class of type $\{ int \}$, the rule:

$$*x \ni 12 \leftarrow P(x)$$

states that the value of each object in class P contains the integer 12.

In the example, we can see some of the forms that can be taken by facts and rule heads:

- the head $supplies(x) \ni y$ allows to derive facts concerning set-valued methods such as $supplies(a) \ni b$;
- the head $S(x, supplies(x))$ allows to derive facts concerning relations such as $S(\#6, \{\#1, \#4, \#5\})$; and
- the head $*x \ni 12$ allows to derive facts concerning values of objects such as $*\#24 = 35$.

$\square$

Example 3.2 While set-valued objects are defined with the ownership predicate, single-valued objects are defined with the equality predicate. We next consider a method *livesin* which maps a supplier's name to the country of this supplier. Thus the method *livesin* has type *string* to *string*. This method is defined by the rule:

$$livesin(x.name) = x.country \leftarrow supplier(x).$$

(For instance, using $x = \#7$, one can derive *livesin*("xzt")= *"France".*)

The use of single-valued functions raises the issue of *consistency*. Suppose that we add the rule:

$$livesin(x.name) = \text{"France"} \leftarrow supplier(x), suppliedby(\#2, x).$$

If part #2 is supplied by some non-french supplier, we derive conflicting facts. The consistency issue is studied in [4] where it is shown, in particular, that *compile-time* consistency checking is undecidable for COL.

Note that set-valued functions do not lead to similar problems since the derivation of facts $F(a) \ni b_1$,...,$F(a) \ni b_n$ only implies that $F(a) \supseteq \{b_1, ..., b_n\}$ and is not incompatible with the derivation of any other fact on F since we derive only *positive* facts. $\square$

3.5 Stratification and Semantics

In this section, we briefly mention how the semantics of such programs is defined using a "stratification" restriction on programs. Due to space limitations, the presentation is quite brief. We refer to [6] for a formal presentation of Datalog with negation and stratification and to [2] for an extension to COL which forms the basis of the semantics that is used here.

There is no obvious way to interpret programs with negation, when rules can be recursive. The semantics is defined for a subclass of programs, namely "stratified programs". Problems arise when there are recursive loops through negation. To solve the problems, recursion through negation is disallowed in stratified programs (e.g., see [6]). Intuitively, this consists in prohibiting the pathological cases. The intuition is that, in "good" programs, $\neg B$ can be used only if B cannot be deduced any more.

It turns out that set-valued objects also potentially induce non-monotonicity as shown by the rules:

$$\begin{array}{lcl} bored(x) & \leftarrow & hob(x) = \{\}, \\ hob(x) \ni tv & \leftarrow & bored(x). \end{array}$$

If John has no hobby, then $hob(John) = \{\}$ by the *Closed World Assumption*, so John is bored. Therefore, $hob(john) \ni tv$ is deduced. The problem is very similar to the problem of recursion through negation. Indeed, the notion of stratification is extended [2, 13] in order to also prohibit such cases of recursion.

Formal treatments of stratification for our context can be found in [2, 4].

The notion of stratification may seem a bit obscure at first glance. However, the constraint is reasonably easy to explain syntactically. Then, all the user has to know is that once the program has passed the test of stratification, its semantics is the "natural" semantics. No deep technique (such as resolution in Prolog) is required in order to understand the meaning of "good" programs.

From an implementation point of view, the semantics of the program can be computed using a sequence of fixpoints. More precisely, the program is split into "strata". The semantics of each stratum is obtained by applying all its rules to saturation. This is done for the first stratum, for the second, etc.

The saturation of a stratum can be realized using some simple algebraic operators under "safety" conditions on the rules. These conditions ensure the finiteness of relations and methods which are generated and will imply, in particular, that only fully instantiated methods are invoked. (See [2] or [5].)

To conclude this section, let us reconsider the problem of consistency. Since compile-time consistency is too complex, we will have to do runtime consistency checking. During the fixpoint computation, an inconsistency may be detected (e.g., if *livesin("John")="France"* and *livesin("John")="Usa"* are both derived). The computation is interrupted and the user is notified of the cause of the inconsistency.

3.6 Externals

We argued in the introduction that the language does not have to be general purpose. However, a *modern* database language should be extensible. The strong types that are present here facilitate this task. Intuitively, particular methods, called *externals*, can be used in rules. These methods are interpreted *outside* the core language in a host language. Some externals are provided by the systems. (They correspond to the aggregate functions of relational systems.) Users can also write their own externals in a host programming language. An experiment with CAML as a host language for COL is reported in [9] together with a discussion of the typing and safety problems that are raised.

We illustrate the use of externals with two examples.

Example 3.3 The cost before tax of a composite part can be defined as follows:

$$beforetaxcost(x) = sum(assemblycost(x), total(madefrom(x))) \leftarrow compositepart(x).$$

This rule uses external functions sum and $total$ (defined in the next example). □

The second example illustrates the use of an iteration over the elements of a list.

Example 3.4 Let $X =< l_1; ...; l_n >$ be of type $< [quantity : int, component : part] >$. Suppose that we want to compute the sum

$$\sum_{i=1}^{n}(l_i.quantity \times cost(l_i)).$$

To do that, we can use the rule:

$$total(X) = iterate(0, sum, map(X, mult(2nd, cost(1st))) \leftarrow .$$

Here $2nd$ and $1st$ are methods that take a pair as input and project it on the first or second coordinates, respectively. (It is trivial to write them in the language.) The method $mult$ is an external multiplication function. The external functional map applies a function (here $mult(2nd, cost(1st))$) to all the elements of a list (here X). Finally, the external functional $iterate$ iterates a certain function (here sum) over the elements of a list (here $map(...)$) starting from some initial value (here 0).

For some given $X =< [a_1, b_1], ..., [a_n, b_n] >$ and a given i:

- $mult(2nd, cost(1st))([a_i, b_i]) = b_i \times cost(a_i)$,
- $map(X, mult(2nd, cost(1st))) =< (b_1 \times cost(a_1)), ..., (b_n \times cost(a_n)) >$,
- $total(X) = 0 + (b_1 \times cost(a_1)) + ... + (b_n \times cost(a_n))$.

Note two subtleties:

- The second argument of $iterate$ is a function. Thus $iterate$ is a functional and the system has to consider higher order types.
- The $total$ method succeeds only if the cost of all components is known. □

4 OVERLOADING METHODS

We illustrate overloading by reexamining the Bill of Materials example. Suppose that we add the following declarations (see Figure 2):

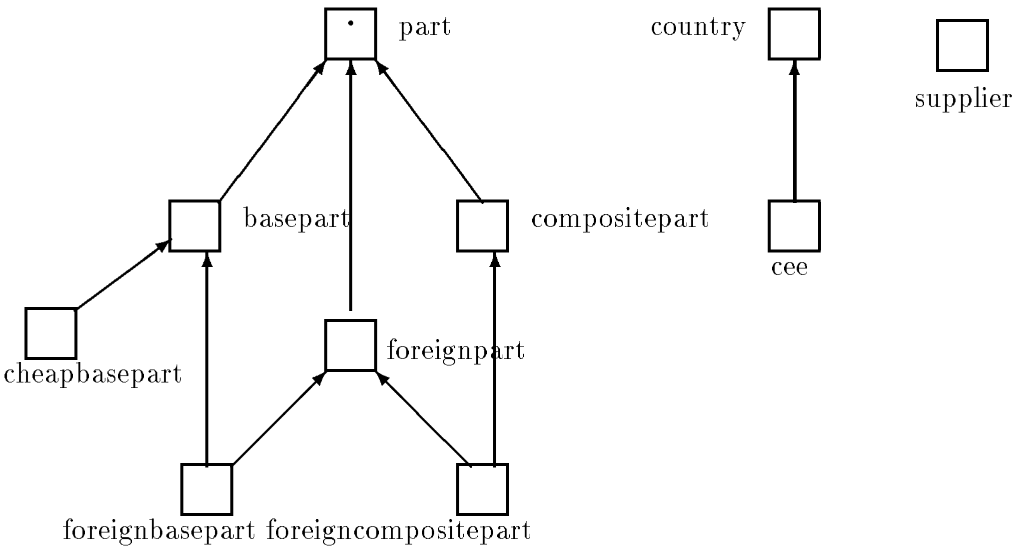

Figure 2: A class hierarchy

class country :	*string*,
class cee :	*isa country*,
class foreignpart :	*isa part and* [*madein* : *country*],
class foreigncompositepart :	*isa foreignpart and isa compositepart*,
class foreignbasepart :	*isa foreignpart and isa basepart*.

We now want to compute the cost of each part assuming that there is an import tax *tax* on imports from non-*cee* countries. (We use the *total* method as defined in Section 3.6.)

1. $beforetaxcost(x) = sum(assemblycost(x), total(madefrom(x))) \leftarrow compositepart(x)$,
2. $beforetaxcost(x) = x.price \leftarrow basepart(x)$,
3. $tfactor(x) = add(1, tax) \leftarrow country(x)$,
4. $tfactor(x) = 1 \leftarrow cee(x)$,
5. $cost(x) = beforetaxcost(x) \leftarrow$,
6. $cost(x) = mult(beforetaxcost(x), tfactor(x.madein)) \leftarrow foreignpart(x)$.

The *beforetaxcost* method is defined on *basepart* and *compositepart*. As a consequence, it is defined on *part*. The *tfactor* method is defined on *country* and refined on *cee* which is one case of overloading. The *cost* method is defined on *part* and refined on *foreignpart*.

Note that strictly speaking, for a *cee* country, we deduce two *tfactors*, i.e., 1 and $1 + tax$. The more refined deduction overrules the other one. For instance, we deduce facts such as:

$$tfactor(Greece : cee) = 1,$$
$$tfactor(Greece : country) = 1.27,$$
$$tfactor(Usa : country) = 1.27...$$

Some of them, e.g., $tfactor(Greece : country) = 1.27$ are overruled by others and just discarded.

This is perhaps the core of the integration between the logic-programming and object-oriented paradigms. From a logical point of view, deriving two contradicting facts is an inconsistency. This is indeed how conflicting facts are treated in general here. We make an exception in the context of overloading. This leads to introducing some precedence between derived facts. As a consequence, the logic is non-monotonic as shown in the following example.

Example 4.1 Suppose that we have a class *europe* and we add the rule:

$$cee(x) \leftarrow europe(x), tfactor(x) > 1.2$$

Using the previous rules, we can derive:

$$tfactor(Poland) = 1.27.$$

Using the new rule, this fact serves to derive: $cee(Poland)$. Thus $tfactor(Poland) = 1.0$ is derived overwriting a fact that has been previously used. □

This non-monotonicity resembles that introduced by negation. The solution that we adopt also resembles the solution that has been used for negation: we impose stratification conditions on programs to prohibit such cases.

To conclude this section, we consider an issue that is raised by the use of overloaded methods.

Consider the running example and suppose that Rule 4 is replaced by:

$$\text{(4-a) } tfactor(x) = 1 \leftarrow cee(x), x.name \neq \text{“}Spain\text{”}.$$

Then, the *tfactor* for Spain will be $1 + tax$. In other words, although there is a rule that deals precisely with *cee* members, the rule about arbitrary countries is still applicable to *cee* members.

Suppose now that rules are encapsulated in modules attached to classes. Then Rule (3) would be attached to the *country* class whereas Rule (4-a) would be attached to the *cee* class ass follows:

$$
\begin{array}{ll}
module & country \\
 & \ldots \\
 & tfactor(x) = add(1, tax) \leftarrow country(x), \\
 & \ldots \\
module & cee \\
 & \ldots \\
 & tfactor(x) = 1 \leftarrow cee(x), x.name \neq \text{"}Spain\text{"}, \\
 & \ldots
\end{array}
$$

Because of the encapsulation, only rule (4-a) would be active for deriving the *tfactor* of *cee* members. Thus the *tfactor* of Spain would not be $1 + tax$ but would be undefined.

The second approach that views rules as attached to modules seems preferable from a semantic point of view. It also clearly simplifies the implementation.

5 EXPLICIT CONTROL

Until now, we have considered that an application program is a set of rules that are applied until saturation. Even simple applications often require (in particular, for interacting with users) more explicit control. For instance, one may want to control which rules are enabled and which are not. There are two solutions. The first one is to adopt a more involved "declarative" semantics. The other one is to introduce new primitives for explicitly controlling the flow of execution of the rules. As mentioned above, we already have some forms of control (e.g., the stratified semantics enforces an order of execution of the rules). We next present more ways of introducing explicit control. Of course, this violates the original goal of having a "declarative" language. However, we feel here that philosophy (or religion) should not come on the way of simplicity and certain things are more easily expressed using explicit control.

We will now view an application program as a set of sets of rules organized into a transaction using primitives providing explicit control. We first examine some useful control primitives. We then revisit the semantics of programs.

5.1 Executing only once

It may be useful to force a rule to execute only once. Suppose that we want to increase the current tax by 3% if it is less than 0.15. To do this, one might want to use:

$$\text{(a) } tax = mult(tax, 0.03) \leftarrow tax < 0.15.$$

But, with the above semantics, tax is iteratively increased by 3% until it gets over 0.15 whereas we meant to increase it only once. (If the condition $tax < 0.15$ is not present, the rule would never terminate.) To specify that a rule must be applied only once, we use the keyword *once*. Here, we would use the rule:

$$\text{(b) } tax = mult(tax, 0.1) \leftarrow tax < 0.15, once.$$

Clearly, the semantics of *once* can easily be specified in the original language using some explicit control predicates, so the use of *once* can be viewed as some syntactic sugaring.

To stress the difference between rules with and without *once*, we may use two different syntaxes. The keyword *while* indicates an arbitrary rule and the keyword *if* indicates a rule with *once* in it. Rules (a) and (b) become:

$$\begin{array}{lllll} (a') & while & tax < 0.15 & do & tax = mult(tax, 0.1), \quad \text{and} \\ (b') & if & tax < 0.15 & then & tax = mult(tax, 0.1). \end{array}$$

We next consider queries. We will see that the difference between *if* and *while* rules will lead directly to the notions of integrity constraints and triggers.

Consider for instance the rule:

$$\leftarrow R(x, y), R(x, y), x \neq z.$$

Applied only once:

$$if\ R(x, y), R(x, y), x \neq z\ do\ nothing,$$

this is the traditional way of expressing queries in rule-based languages. This particular one can be interpreted as: "does R satisfies the functional dependency $1 \rightarrow 2$?"; or in more precise terms, "give me the tuples that exhibit a violation of the constraint *now*". Now suppose instead that we use the "while" construct:

$$while\ R(x, y), R(x, y), x \neq z\ do\ nothing.$$

The rule stays valid after application. Whenever in the following of the session a violation of the dependency occurs, the rule becomes active and detect that violation. The system will detect an inconsistency and print the "query" that is involved together with the answer to that query indicating the origin of the violation.

5.2 Sequencing

More elaborate control is also achieved via *sequencing*.

We use the following more elaborate syntax for programs:

program → < *set of if/while rules separated with commas* >
 | *begin program end*
 | *program* ; *program*

The semantics of a block is the semantics described above. The semicolon indicates sequencing.

5.3 Methods as Procedures

We finally consider the addition of procedures. Procedures are called only with fully instantiated arguments. These arguments may use "local" temporary relations and methods. The use of procedures clearly introduce the means of controlling the execution flow of a set of rules.

We next define the *total* function as a procedure.

$$
\begin{array}{lll}
let \;\; total & (X :< [quantity : int, component : part] >) \; be & \\
& begin & \\
& relation \; tail :< [quantity : int, component : part] >, & \\
& method \; tailcost :< [quantity : int, component : part] > \rightarrow num, & \\
& & \\
& \quad tail(X) \hfill \leftarrow & , \\
& \quad tail(t) \hfill \leftarrow & tail(a :: t), \\
& \quad tailcost(<>) = 0 \hfill \leftarrow & , \\
& \quad tailcost([y, z] :: t) = sum(mult(y, cost(z)), tailcost(t)) \;\leftarrow & tail([y, z] :: t), \\
& \quad total = tailcost(X) \hfill \leftarrow & \\
& end &
\end{array}
$$

Intuitively, a call to a procedure interrupts the normal fixpoint execution of a program. The rules of the procedure become active. When these rules reach a fixpoint, two cases occur:

- the result (here *total*) has been given a value, and this is the return value.
- it has not; the return value is viewed as undefined.

Thus from a formal point of view, a procedure is seen exactly like a method that is extensionally or intentionally defined or like an external method. Note that again *total* is defined only if the *cost* is known for every component.

We next present another example dealing with list manipulations.

Example 5.1 Clearly, using techniques like above a method *append* on lists can be defined. However, such a method would construct a new list by copying the two arguments. It is possible to avoid this copying. Indeed, suppose that we have two lists of integers. These lists can be represented using a class *element* of $[A : int, B : elem]$ as follows. Suppose that we have three objects o_1, o_2, o_3 of this class with $*o_1 = \perp$, $*o_2 = [12, o_3]$, $*o_3 = [5, o_1]$. Then o_1 represents the empty list, o_3 the list $[5]$, and o_2 the list $< 12; 5 >$. Suppose also that we have two objects o'_2, o'_3 of this class with $*o'_2 = [12, o'_3]$, $*o'_3 = [2, o_1]$. To obtain in o_2 the concatenation of the list represented by o_2 and that represented by o'_2, i.e., $< 12; 5; 12; 2 >$, it suffices to modify the value of o_3 to $[5, o'_2]$.

More generally, the following method takes two lists represented in this fashion as arguments and appends the second one to the first[2]:

$$
\begin{array}{lll}
let & \multicolumn{2}{l}{directappend(list1, list2)\ be} \\
 & \multicolumn{2}{l}{begin} \\
 & \multicolumn{2}{l}{relation\ inlist1 : elem,} \\
 & & \\
 & inlist1(list1) \quad \leftarrow & , \\
 & inlist1(z) \quad \leftarrow & inlist1(x), *x = [y, z], *z = t; \\
 & & \\
 & x = [y, list2] \quad \leftarrow & inlist1(x), *x = [y, z], \neg inlist1(z), \\
 & directappend = list1 \quad \leftarrow & \\
 & \multicolumn{2}{l}{end}
\end{array}
$$

5.4 The extended language

As mentioned above, a *declarative* way of writing programs is not a goal in itself. An application program consists in sets of rules (i.e., small pieces of "declarative" programs) organized into a transaction using primitives providing explicit control.

Programs are now more complicated and their semantics is more involved. In particular, three levels have to coexist:

1. the local level consisting of a set of rules with a fixpoint semantics;
2. the top level which is procedural and controls which rules are active and which are not; and
3. the external level which provides the extensibility via a host language.

This is further complicated by the fact that there are several causes for non-monotonicity at the local level: the presence of negation, of sets and the overloading. Stratification provides a uniform way of solving the problems raised by non-monotonicity.

Again this may seem unnecessarily complicated at first glance. However, the user shouldn't have to be aware of it to be able to program:

- the external level is only a way of providing new data types and the programmer of simple applications does not deal with it;
- the explicit control is quite natural and should not introduce more difficulty in programming; finally,
- the stratification condition will force to write simple programs easy to understand.

[2]The semicolon here introduces some explicit control in the program. It forces the first two rules to saturate before going to the second part of the program. The program assumes that both lists are non empty; the general case is only slightly more involved.

5.5 Persistence and Transactions

To conclude this section, we briefly consider the issue of persistence of data and rules. A symbol (relation, class or method) or a rule is persistent if it has to survive the end of the session/transaction. Temporary symbols and rules are automatically discarded at the end of the session. The programmer has to explicitly give the status (temporary/permanent) of rules and symbols. (A default can obviously be chosen.)

$$\begin{array}{llll} relation & R & :... & permanent \\ method & M & :... \rightarrow ... & temporary \\ rule & r_1 & :... & temporary \\ rule & & :... & permanent \end{array}$$

(Note in the example that rules can be given names.)

The previous discussion leads to a more general notion of *nested transaction* that will not be considered in its generality here.

6 UPDATES

The language should also allow for the insertion, deletion and modification of rules and facts. In this section, we focus primarily on data updates.

To be able to discuss updates, we have now to precisely distinguish between the database and views. Traditionally, in deductive databases, a predicate is a database predicate if it does not occur in the head of a rule. The implicit assumption is that rules are used exclusively to define views. Thus a predicate occurring in the head of a rule has to be a derived predicate, i.e., a view. We do not adopt this convention here since we use rules both for defining views and for updating the database.

The database consists of classes, relations and methods. Some of them are explicitly "stored" whereas others are only defined intentionally, i.e. are views. The "status" (base or views) must be given explicitly at definition time:

$$\begin{array}{llll} relation & R & view & :... \\ method & M & & :... \rightarrow ... \\ method & N & view & :... \rightarrow ... \\ class & S & & :... \end{array}$$

Views are computed only when it is necessary to acquire more knowledge. For instance, if P is a base relation, the rule

$$P(x, y) \leftarrow ...$$

is interpreted immediatly (new tuples are inserted in P); whereas it is not if P is a view. The interpretation may be delayed until, for instance, P is queried.

Depending on the status of the defined symbol, rules are therefore used for updates or view definition.

6.1 Insertion and Object Creation

We already saw that a rule such as $R(a, b) \leftarrow$ is interpreted as an insertion if R is a base relation. One can insert new facts concerning a method in the same manner. Insertion in a class is more involved since we are inserting oids and not values.

Object creation is controlled following the spirit of [8, 5]. We allow the presence of variables occurring in the head of a rule and not in the body assuming that they are class variables. These variables are interpreted as new objects.

Continuing with the Bill of Material example. Suppose that we are given a set *partnames* of names and we want to have a base part for each of these names. This is done using:

$$if\ partnames(y)\ then\ *x : basepart = [name : y].$$

For each y, a new oid x is created in the *basepart* class and is assigned the tuple $[name : y]$ for value. This is done only if there is no existing base part with this name. For the newly created object, the mass and price fields are undefined ($\perp$).

The created objects force us to reconsider the fixpoint semantics: a new object is created *only* if there is no existing object that can fulfil the specifications. In particular here, the above rule is interpreted:

$$\forall y \in partnames(\exists x \in basepart(x.name = y))$$

and if this is not the case for a particular y, a new *basepart* is created to fulfil the requirement. Note that this is rather easy to implement. However, this is potentially dangerous since it is now straightforward to write nonterminating programs using the creation of oid's.

Remark: Once an object has been created, we have three ways of naming it: e.g.,

$$\begin{array}{llll} if & true & then & currentobj_1 = mynewobject, \\ if & true & then & currentobj_2 = \#388, \\ if & x.name = hammer & then & currentobj_3 = x. \end{array}$$

In the first case, an object is named by a surrogate (i.e. a particular method with zero arguments.) In the second case, the oid is used directly. Finally, a property of the object is used to select it in the last case. Note that if several objects have the same name, this last rule will result in a runtime error.

6.2 Deletion

Deletion is performed using a negative literal in the head of a rule. Thus,

$$if\ T(x,6)\ then\ \neg T(x,6)$$

removes all tuples in relation T having 6 for second field.

To see another illustration of deletions and insertions, consider the status of relations. This status can be changed using the rules:

$$\begin{array}{lll} if & true & then \quad persistent(R), \\ if & true & then \quad \neg persistent(R). \end{array}$$

Rules are objects in a particular class *rule* and have certain properties such as *name* and *number*. A rule can be deleted and its status changed:

$$\begin{array}{llll} if & r.name = \text{"}ttt245b\text{"} & then & \neg rule(r), \\ if & true & then & \neg status(transitive\_closure\_R) \ni persistent, \\ if & r.number = \#25 & then & status(r) \ni persistent. \end{array}$$

The first rule deletes the rule with name "ttt245b". The second changes the status of the rule that has *transitive_closure_R* for surrogate; and the last one the status of the rule with rule number #25.

6.3 Modifications

Two kinds of modifications are considered:

1. modifying the value of an object,
2. modifying the value of a method.

The two cases are treated similarly. Note that a first assignment of value is just a particular form of such modifications (i.e., modifying from $\perp$ to some non-$\perp$). To see an example, suppose that we are given two relations

$$\begin{array}{l} BPart\ (name : string, price : num, mass : num), \\ CPart\ (name : string, assemblycost : num, massincrement : num, \\ \qquad subpart : string, quantity : int). \end{array}$$

We want to move this relational information to the schema defined above. To do that, we must create a new oid for each part and assign values to these newly created oid's. This is done with:

$$\begin{array}{llll} if & BPart(u,v,w) & then & *x : basepart = [u,v,w], \\ if & CPart(u,v,w,s,q) & then & T(u,v,w), \\ if & T(u,v,w) & then & *x : compositepart = [u,v,w], \\ if & CPart(u,v,w,s,q), x.name = u, y.name = s & then & madefrom(x) \ni [y,q]. \end{array}$$

An important aspect here is that new objects can be created without being assigned any value. This provides us with a weak form of incomplete information which is needed if complex cyclic structures have to be created and maintained [5].

The assignment of values to oid's raises issues similar to that encountered in [4] for defining a semantics in presence of single-valued functions. The problem is that a program may yield conflicting statements, e.g., $*myobj = 12$ and $*myobj = 13$. In [4, 5], single assignments were considered, i.e., once the value of a single-valued function had been derived, any attempt to modify it would raise an error [4] or be just discarded [5]. We need here to be able to modify values of objects in a somewhat less monotonic way. Therefore we distinguish between two cases:

- $*myobj = 12$ is a fact and a new assertion $*myobj = 13$ is derivable from the program, in which case the value of the object is updated;
- $*myobj = 12$ and $*myobj = 13$ are both derivable at the same time, in which case an error is raised.

Remark: The previous example stresses the fact that program are interpreted in a "temporal" perspective. The notion of *when* a fact holds is important. Thus the underlying logic is a *temporal* logic.

We next show an example to illustrate how this can be used:

Example 6.1 Suppose that we want to "increase the price of the base part *hammer* by 5% until it is over 10000". This is realized with the rule:

$$while\ *x = [\text{``}hammer\text{''}, y, z], y < 10000\ do\ *x = [\text{``}hammer\text{''}, mult(y, 1.05), z].\Box$$

6.4 Modifying the oid assignment

We saw how an oid can be created. Oid's may also be deleted from classes using negative literal in heads of rules. For instance, the rule "$\neg P(o) \leftarrow o.A = 5$" removes each oid o living strictly in P such that $o.A = 5$, but also each oid in all subclasses of P with that property. For instance, suppose that we have a subclass *south_cee* of *cee*. Then the rule:

$$if\ kingdom(x)\ then\ \neg cee(x).$$

(no political message intended) removes the kingdoms from $\pi(cee)$ and $\pi(south\_cee)$. The kingdoms stay in $\pi(country)$.

If one removes an oid from all classes, this oid "dies". All references to it are then viewed as $\perp$.

Finally, an object can be transferred from one class to another. Suppose that we want to "expel" the kingdoms from the common market but still want to consider them as countries. This is realized with the rule:

$$if\ cee(x), kingdom(x)\ then\ country(x).$$

This is yet another occurrence of the overloading of class names. In the right side of the rule, *cee* means the objects that are in the *cee* class or a subclass, whereas *country* in the head is intended to mean the objects that are strictly in the *country* class.

When an object is deleted, all occurrences of that object have to be "traced" in order to be removed. When an object is moved from one class to another, all references to that object must be found to test that they are valid (i.e., do not violate typing constraints once the object is moved to its new class).

Remark: Schema updates have also to be provided. They are certainly even more involved and will not be considered in the present paper (See, e.g., [11]).

7 THE WINDOW CLASS

To illustrate the language, we briefly consider window manipulation. It is important to be able to control the interface from the rule-based language. Again we stress the fact that we don't want here to have a powerful language for developing interfaces, but only to incorporate limited such features. In this section, we sketch how this can be achieved.

Let us continue with the Bill of Material example. Suppose that we want a table with the following kinds of information: for each part, the name, the set of suppliers and the quantity of each subcomponent. This table can be represented by a function *summary* defined as follows:

$$\begin{array}{llll}
if & true & then & summary(x : part, "name") = x.name, \\
if & suppliedby(x, y) & then & sup(x) \ni y.name, \\
if & true & then & summary(x : part, "suppliedby") = sup(x), \\
if & madefrom(x) \ni [y, z] & then & summary(x : compositepart, y) = z.
\end{array}$$

We now assume that a particular class *window* is given. To edit (or browse through) a relation or function, it suffices to use this predicate. For instance, the rule

$$if\ true\ then\ *gwindow{:}window = summary.$$

creates a new object *gwindow* in the *window* class and gives to this object *summary* for value. An object of the window class is assigned automatically certain properties. Window parameters are assigned to it by default, e.g.,

$$if\ window(x), undefined(uppercorner(x))\ then\ uppercorner(x) = [0, 0];$$

part	name	suppliedby	#1	#3	#4	#5
#1	aa	$\{ yth, xzt \}$		2	5	1
#2	bb	$\{ xzt \}$	2		2	
#3	cc	$\{ xzt \}$				
#4	dd	$\{ yth \}$				
#5	ee	$\{ yth \}$				

Figure 3: A table representation of *summary*

unless some explicit rules are entered by the user, e.g.,

$$\begin{array}{llll} if & *x = summary & then & uppercorner(x) = [40,40], \\ if & true & then & uppercorner(gwindow) = [40,40]. \end{array}$$

The user can also modify graphically the parameters of a window, and then "save" them, i.e., have the system automatically generate a rule like the explicit rules above.

Among the functions specifying a window, one finds geometrical arguments (such as position, size), arguments such as colors and contour, read/write or read-only, and more important some arguments for selecting a presentation style. For instance, the summary function is presented by default in a functional manner. One can, using the rule

$$if\ true\ then\ style(gwindow) = table,$$

choose a table representation like in Figure 3.

To see another example, suppose that we want to implement a trigger that create a new window whenever the balance of a user becomes negative and display some information on the user. This can be realized using the rule:

$$while\ balance(x) < 0\ do\ *alarm : window = info(x).$$

A window proposes a computer aided edition of an object. Database objects (relations and functions) can be edited. As a lower level, the values of oid's can also be edited.

It is clearly desirable to have an interface as powerful as possible. For instance, it is important to be able to select an object in the interface and move it from window to window. Furthermore, most commands concerning windows can be made directly using the window manager or the menus of the windows (e.g., changing the geometric parameters or the style). One can therefore wonder why bother having window control functions *within* the language. We believe that it is necessary to be able to control the window manager from within the program if one is to code a complete application in the language. Examples of rules involving both the manipulation of data and of windows are:

- if the number of parts is greater than 40 then change the style to *table*,

- if an unnamed part is supplied by some supplier, then display the name of that part in a new window with a warning to the user,
- if a new part is inserted, create a window to edit the value of that part.

8 CONCLUSIONS

We emphasized the following features:

In the data structure:

1. **Types:** The concrete types used here are standard in databases. Efficiency is the main justification for the separation of database schema and instance, as well as, for the requirement of static type checking.
2. **Relations vs classes:** This dichotomy is essential for being able to express simple queries in a simple manner. We think that without this redundancy, the language would lead to difficulties in maintaining temporary results, eliminating duplicates, and unnatural use of indirection.
3. **Relations vs. Methods:** This redundancy allows to write more natural programs.
4. **Cyclicity and oid's:** There is some subtlety in the use of oid's to code cyclic structures, since these are represented in an acyclic manner using the dichotomy object/value.

In the language:

1. **Declarativity:** The user should be encouraged to program in a declarative manner. The means must also be available for introducing explicit control (e.g., sequentiality, transactions, procedures,...), but this should not be central to the language.
2. **Views:** they are naturally introduced in a rule-based approach. However, the concept of rule is overloaded in this context since besides defining views, rules are also used for programming (and in particular for updates), for queries, triggers and integrity constraints.
3. **Types:** Typing the language serves *both* for correctness and for efficiency.
4. **Basic query and update capabilities:** The means must be there to easily extract information from the database and modify its state.
5. **Invention of oid's:** A primitive for oid invention must be in the language. We believe that this is necessary, if unbounded structures are to be constructed. Invention is a powerful mechanism; it is crucial if arbitrary computations are to be simulated and it should be carefully restricted.

6. **Extensibility:** The language should be easily extensible. That extensibility can be achieved using *external* methods. The external methods belong to a general environment that may be provided by a general purpose object-oriented database systems. An important aspect of this extensibility is to offer basic window management from *within* the language.

7. **Inheritance, Methods, Overloading:** The motivations are all the usual motivations for using an object-oriented style of programming.

It should be mentioned again that this is a first attempt to integrate various approaches and show their compatibility. A lot of efforts has still to be done: (i) to simplify the language, (ii) to offer a SQL-like syntax or a syntax in the style of SQL, and (iii) to implement such a language. Our previous experience with implementing the COL language lead us to believe that this is not an insurmountable task.

Certain aspects have to be better understood before undertaking such an implementation:

1. the harmonization between the fixpoint semantics of the local level and the procedural one of the top level;

2. the implementation of updates and, in particular, of schema updates;

3. a more general notion of transaction.

Acknowledgements: We thank Michel Adiba, Won Kim and Eric Simon for comments on a draft of this paper. The material of this paper has been quite influenced by earlier works with Catriel Beeri, Stéphane Grumbach, Rick Hull, Paris Kanellakis and Victor Vianu.

References

[1] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects, INRIA Technical report, No 846, 1988.

[2] S. Abiteboul and S. Grumbach. COL: a Logic-based Language for Complex Objects. In *Proc. EDBT*, 271–293, 1988.

[3] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM TODS*, 12:525–565, 1987.

[4] S. Abiteboul and R. Hull. Data-functions, Datalog and Negation. In *Proc. ACM SIGMOD*, 143–153, 1988.

[5] S. Abiteboul and P. Kanellakis, Object Identity as a Query Language Primitive. In *Proc. ACM SIGMOD*, 1989.

[6] K. Apt, H. Blair, and A. Walker. Toward a Theory of Declarative Knowledge. In *Proc. of Workshop on Foundations of Deductive Database and Logic Programming*, pages 546–628, 1986.

[7] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[8] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Language. In *Proc. ACM PODS*, 240–250, 1988.

[9] S. Abiteboul, S. Grumbach, A. Voisard, and E. Waller. An Extensible Rule-based Language with Complex Objects and Data-functions. In *Proc. DBPL-II Workshop*, Oregon USA, 1989. To appear.

[10] F. Bancilhon. Object-Oriented Database Systems. In *Proc. ACM PODS*, 152–162, 1988.

[11] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. 1987.

[12] J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, and N. Ballou. Data Model Issues for Object-Oriented Applications. *ACM TOIS*, 5:1:3–26, 1987.

[13] C. Beeri and al. Sets and Negation in a Logic Database Language (LDL1). In *Proc. ACM PODS*, 21–37, 1987.

[14] F. Bancilhon and S. Khoshafian. A Calculus for Complex Objects. In *Proc. ACM PODS*, 53–60, 1986.

[15] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2, an Object-Oriented Database System. To appear. In *Proc. OODBS2 Workshop*, Badmunster RFA, 1988.

[16] F. Bancilhon, S. Cluet, and C. Delobel. Query Languages for Object-Oriented Database Systems. In *Proc. DBPL-II Workshop*, Oregon USA, 1989.

[17] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query-processing strategies. In *Proc. SIGMOD*, 1986.

[18] M.J. Carey, D.J. Dewitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD*, 413–423, 1988.

[19] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.

[20] Proceedings of the DBPL Workshop, Roskoff, 1988.

[21] Proceedings of the DBPL Workshop, Oregon, 1989.

[22] D. Fishman et al. Iris: an Object-Oriented Database Management System. *ACM TOIS*, 5:1:46–69, 1987.

[23] H. Gallaire and J. Minker and J.M. Nicolas, Logic and Databases: a Deductive Approach, *Computing Surveys*, 16:1, 154-185.

[24] A. Goldberg and D. Robson. *Smalltalk 80, the Language and Implementation.* Addison-Wesley, 1983.

[25] R. Hull and R. King. Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys*, Sept. 1987.

[26] Jaeschke, B., H.J. Schek, Remarks on the algebra of non first normal form relations, Proc. ACM SIGACT/SIGMOD Symposium on Principle of Database Systems, Los Angeles (1982) 124–138.

[27] P. Kanellakis. *Elements of Relational Database Theory.* Brown U. Technical Report, 1988. To appear as a chapter in the Handbook of Theoretical Computer Science.

[28] W. Kim. *A Foundation for Object-Oriented Databases.* Technical Report, MCC, 1988.

[29] S. Khoshafian. A Persistent Complex Object Database Language. *Data and Knowledge Engineering*, 1989

[30] G.M. Kuper. Logic Programming with Sets. In *Proc. ACM PODS*, 11-20, 1987.

[31] G.M. Kuper and M.Y. Vardi. A New Approach to Database Logic. In *Proc. ACM PODS*, 86–96, 1984.

[32] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proc. ACM PODS*, 1989.

[33] C. Lecluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. ACM PODS*, 1989.

[34] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proc. ACM SIGMOD*, 424-434, 1988.

[35] D. Maier A Logic for Objects. In *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, Washington USA, 1986.

[36] D. Maier, A. Otis, and A. Purdy. Development of an Object-Oriented Dbms. *Quarterly Bulletin of IEEE on Database Engineering*, 8, 1985.

[37] J.D. Ullman. Database Theory - Past and Future. In *Proc. ACM PODS*, 1–10, 1987.

[38] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[39] S. Zdonik. Object Management Systems for Design Environments. *Quarterly Bulletin of IEEE on Database Engineering*, 8, 1985.