A Logical Query Language for an Object-Oriented Data Model*

Alvaro A.A. Fernandes, Norman W. Paton, M. Howard Williams
Department of Computing & Electrical Engineering
Heriot-Watt University
Edinburgh, UK

Abstract

We report on our experience in designing and prototyping ROLL, a logical query language under which object-oriented databases conforming to the model defined in [17, 18] can be seen as *deductive object-oriented databases*. The main contribution of ROLL is to demonstrate the possibility of pursuing for the object-oriented case the same research strategy used in the last decade to extend relation databases with deduction. This paper briefly describes the rationale behind our approach to the field and the DOOD project that is underway at Heriot-Watt. After an introduction to ROLL's underlying data model by means of a simple example, the paper focusses on the implementation of ROLL in a prototype which has been built to experiment with both the language and the model. Certain techniques used in the prototype are described, and ROLL is contrasted with some well-known proposals of deductive languages for object-oriented databases. An indication of future work is given with some conclusions.

1 Introduction

It has long been recognized that the principal benefits offered by deductive and by object-oriented technologies to database systems are distinct and complementary. Deductive databases offer declarative query expression and rule-based programming, and systems are underpinned by a theoretical framework developed from first-order logic. Object-oriented databases (OODBs) offer rich data modelling constructs, and integrate programs and data to provide an effective development platform for new application areas. Furthermore the weaknesses of deductive databases are exactly the strengths of their object-oriented counterparts, and vice-versa – deductive databases are normally associated with the relational model which lacks the expressiveness of the object-oriented models; object-oriented database systems are normally developed without much regard to a theoretical framework, and commonly provide limited support for queries or deductive rules. Any database system which smoothly integrates the two paradigms, without sacrificing the characteristic virtues of either, is likely to present significant opportunities for application-developers in both novel and conventional domains.

The deductive object-oriented database (DOOD) project at Heriot-Watt is developing a system which supports object-oriented and deductive facilities. The approach adopted in this project differs from alternative proposals [14] in using a formally defined object-oriented data model as the starting point for the development of a DOOD. The data model [17, 18], which is outlined briefly in Section 2, supports the mandatory data model features identified in [7], and has been significantly influenced by the structural abstractions supported by many semantic data models. This model has been used as the basis for the design of two languages – an imperative data manipulation language [8] which is used to describe complete applications and as the sole means of performing updates, and a logical query language, which is the subject of this paper. The architecture of the complete system is described in more detail in [16]. The management facilities which implement that data model and the manipulation language are collectively referred to as ROCK (Rule Object Computational Kernel). The logic language itself is referred to as ROLL (Rule Object Logic Language).

ROCK is currently in an advanced stage of implementation using the EXODUS extensible database system [25], with the implementation of ROLL soon to follow. ROLL has been prototyped using MegaLog [9]. This stand-alone, interactive prototype uses the meta-interpretation technique described in Section 4 and has served two purposes – it has enabled experimentation with querying databases conforming to the data model described in Section 2 by means of a logical query language and it has assisted with the theoretical formulation of the logic language, providing a test ground for ideas. This paper describes the current syntax of the logic language under development at Heriot-Watt, and outlines how it has been prototyped using MegaLog.

*In N.W. Paton and M.H. Williams *Rules in Database Systems: Proceedings of the 1st International Workshop, Edinburgh, Scotland (30 August–1 September 1993)*, pages 234–250, 1994, Springer-Verlag, Workshops in Computing Series. ISBN 0-387-19846-6

The paper is structured as follows: Section 2 gives a flavour of the data model on which the logic language acts; the language is itself introduced in Section 3, and illustrated using a number of examples; Section 4 describes the prototype implementation; Section 5 contrasts this work with alternative approaches to the development of query languages for DOODs; Section 6 outlines future directions, and presents some conclusions.

2 The Data Model

This section informally describes the main structural and behavioural constructs used in the data model. *Primitive types* are domain-independent, and their instances, called *primary objects*, are named by *values*. For instance, `string` and `integer` are primitive types; `"scotcars"` and `18` are primary objects. *Object types* are domain-dependent, and can be user-defined if their instances are named by (proper) `object identifiers`, otherwise their instances are named by values, e.g. `company` and `companyName` are object types; `!1` is an object identifier naming an instance of `company`; `"scotcars"` is a value naming an instance of `companyName`. (The use of '`!`' to syntactically mark an object identifier is arbitrary.)

A hardware and software environment implies the existence and the well-typedness of primary objects, i.e. they are assumed to conform to an adequate structural and behavioural representation. Users assign a secondary object to one or more object types, whereby it acquires a structure and a behaviour to conform to. Object identity is strictly supported, detaching the existence of an object from its contingent *state*.

The state of an object consists only of the *references* it makes to other objects. References are of two kinds: the first models *properties*; the other, *construction*. Property references are stipulated through an abstraction mechanism called *attribution*; conceptually, they appear in an object's state as a type-heterogeneous type-indexed collection of object-names. Construction references characterize the object as *complex* and are stipulated by one of three abstraction mechanisms: *association* (stipulating that an object's state will contain a type-homogeneous non-indexed collection of object-names, a set of object identifiers in short); *sequentiation* (in this case, a type-homogeneous $\mathcal{N}$-indexed collection, somewhat like a list if $\mathcal{N}$ is the set of natural numbers); and *aggregation* (in this case, a type-heterogeneous type-indexed collection, somewhat like a record). Attribution and aggregation become distinct by the account they are given in the presence of multiple inheritance, aggregation being the more restrictive.

The *behaviour* of objects is determined by the *interfaces* of the object types they have been assigned to. An interface declares *operations* by associating *operation names* with *operation signatures*. Operation names can be *overloaded* by being declared on two or more distinct object types. The notion of *object class* is introduced, in one-to-one relationship with object types, to detach an interface from the *operation implementation*, called *method*, that realizes it. Users are only concerned with operation interfaces, whereas it is up to designers to write the methods which implement the interfaces using the language components the database is provided with, viz. ROCK & ROLL. This detachment protects users from changes designers may need to make in the implementation. Also, the state of an object is *encapsulated*, i.e. users can only manipulate it by invoking operations contained in the interface of the types it is assigned to. In this paper, only access operations, i.e. queries, are considered. A more comprehensive overview is found in [16].

Both structural and behavioural (multiple) *inheritance* are supported, potential conflicts being ruled out by well-formedness conditions on the schema. Inheritance can be presented as either *specialization*, *generalization* or *partition*, the latter differing from generalization by implicitly implying a non-intersection constraint on the extension of the types involved. Aspects are also catered for: an object type t' is thought of as an aspect of some object type t if t is best modelled by a distinct, typically unrelated, interface even though the structural characterization t' is best modelled as identical to that of t. The presence of inheritance and overloading of operations requires support for *overriding* and *late-binding*.

The schema in Figure 1 is used in examples throughout the paper. The notation used in the diagram is briefly described as follows. A *circle* represents the source from which names of instances of object types are drawn. (In Figure 1, not all arrows into circles are drawn.) A *horizontal oval* represents an object type whose instances are named by values; a *full-line rectangle* one whose instances are named by proper object-identifiers; a *dashed-line rectangle* represents an aspect; a *round-cornered rectangle* represents an operation. A *labelled directed edge* has as label one out of seven symbols: ◯ for *attribution*, ⧇ for *generalization*, ◈ for *partition*, ⟁ for *specialization*, ⊗ for *aggregation*, ⊛ for *association*, and ⊚ for *sequentiation*. The last three abstraction mechanisms may be specified in a *cohesive* version (not used in Figure 1) by enclosing the characteristic symbol within a square, rather than a circle. The semantic import of cohesion is to define an existence dependency between an object and the object it refers to, i.e. deleting the referred object may imply the deletion of the referring one. An *unlabelled directed edge* drawn as a *full-line* leads from an object type to the source of its objects' names; drawn as a *dashed-line* it leads from/to object types that are arguments/results of operations. Finally, in edges labelled with association, sequentiation or aggregation symbols, a *double-headed full-line arrow* stipulates the constraint that each instance of the type being

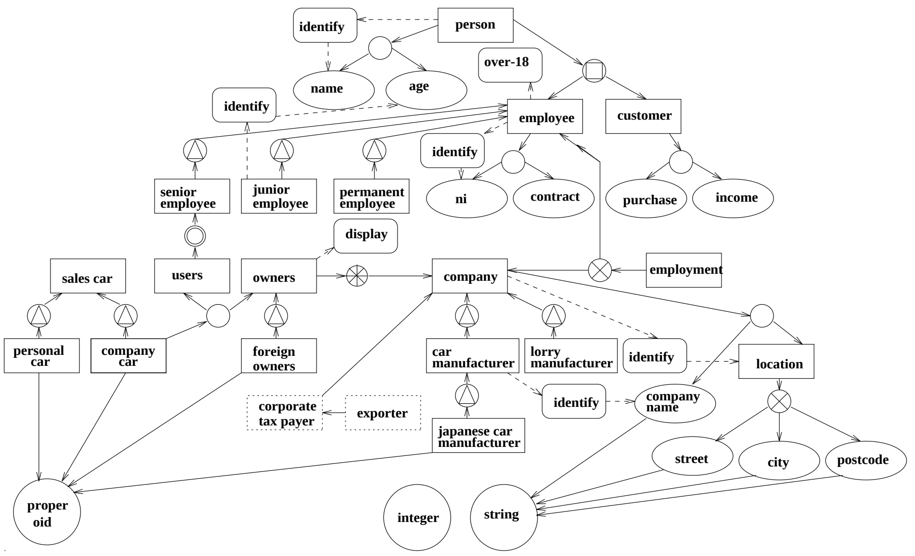

Figure 1: Company Schema Diagram

pointed to must be referenced by, possibly many, instances of the type where the arrow comes from; a *single-headed full-line arrow* the constraint that each instance must be referenced and at most once. Versions of the latter two not occurring in the diagram use dashed lines to drop the mandatory aspect of the reference.

3 The Logic Language

This section introduces ROLL, a logical query language for databases conforming to the model overviewed in Section 2. The ROLL syntax presented here is reminiscent of [19]. Syntactic sugar will be used to integrate it with ROCK [16].

A ROLL **term** is either a variable, a constant or a keyword (denoting built-in operations, as clarified later). The alphabet contains no function symbols, therefore a term is *ground* iff it is a constant or a keyword. The set of all terms is $Term = Cons \cup Kwords \cup Var$. The set of all ground terms, called the *Herbrand universe*, is denoted by H_U. Since $Cons$ and $Kwords$ are finite, so is H_U. A ROLL **atom** has the following general structure α`[`β`(`γ`)=>`δ`]`. The Greek letters are place-holders: α is filled by an object-denoting ROLL term; β is filled by a ROLL term denoting an operation in the interface to a class of α; γ is filled by a comma-separated sequence (of object-denoting ROLL terms) denoting the argument-tuple of β; δ is filled by a ROLL term denoting the result of applying the operation β with arguments γ to the object α. When β takes no arguments or returns no results, `(`γ`)` or `=>`δ, respectively, can be dropped. An atom is *ground* iff α and β are ground, and γ and δ are either ground or missing. Note that a number of built-in predicates (BIPs) exist that are written in infix notation. For instance, the value-equality operation is '`==`'. Thus `!1 == !2` is true iff the object whose name is `!1` is value-equal to the object whose name is `!2`. From ROLL atoms, ROLL literals and clauses are defined as for deductive databases [10]. A ROLL clause is a Horn clause. Again, **ROLL facts, rules and goals** are special forms of Horn clauses, as expected. The usual characterization of an *intensional* and an *extensional part* of databases and the logical interpretation to variables in queries are also applicable.

In the prototype described in Section 4, ROLL programs are given an operational semantics by a one-to-one mapping onto an internal representation that is then input to an SLDNF-resolution proof procedure, extended to cope with late binding. Querying with ROLL is now illustrated through excerpts from a transcript of an interactive session with the stand-alone prototype ROLL interpreter. The domain is that of Figure 1. Note that proper object identifiers (constant symbols of the form `!`n) would not appear explicitly in ROLL queries embedded in ROCK programs.

3.1 Schema-Level and Instance-Level Queries

First, a wide-ranging example of schema-level querying in ROLL. In the schema-level queries below all operations invoked are built-in, and their axiomatic semantics is captured at the level of the data model [18]. Intuitively, sending `ipo` to an object type t returns the object types t' which aggregate t, sending `has` to an object type t returns the object types t' that are attributes of t. Sending `isa` to an object type t returns the supertypes t' of t. Finally, sending `ici` to an object type t returns the object types t' which associate or sequentiate t. The query then reads: *which object types* `X`*, which aggregate* `city`*, are a property of subtypes* `Y` *of* `company`*, such that object types* `Z` *associate* `Y`*?*

```
roll ?- city[ipo => X], Y[has => X], Y[isa => company], Y[ici => Z].
[X = location, Y = carManufacturer, Z = owners]
another? ;
[X = location, Y = carManufacturer, Z = foreignOwners]
another? ;
[X = location, Y = company, Z = owners]
...
```

Using an `isa` atom with the addressee- and result-position free, and a interpreter-directive `all` to collect all answers and return them at once, it is simple to retrieve, as a set of <`Sub,Super`> pairs, the entire is-a hierarchy:

```
roll ?- all Sub[isa => Super].
...
[Sub = japaneseCarManufacturer, Super = japaneseCarManufacturer]
[Sub = japaneseCarManufacturer, Super = carManufacturer]
[Sub = japaneseCarManufacturer, Super = company]
[Sub = juniorEmployee, Super = juniorEmployee]
[Sub = juniorEmployee, Super = employee]
[Sub = juniorEmployee, Super = person]
...
```

Now, some examples of instance-level querying in ROLL. Again, most operations are built-in. Intuitively, sending `iof` to an object i returns the object types t it has been assigned to. Sending `.`t (a shorthand for `get(`t`)`) to an object i returns the object i' referred to by i as its t-property. The first query reads: *what is the name of employees* `E` *who are older than people named Mary?*

```
roll ?- E[.name => Ename], E[.age => Eage],
        Mary[.name => "mary"], Mary[.age => Maryage],
        Eage :> Maryage.
[E = !3, Ename = john, Eage = 30, Mary = !4, Maryage = 25]
another? ;
[E = !3, Ename = john, Eage = 30, Mary = !19, Maryage = 25]
...
```

Note the import of object identity in the fact that two distinct objects,`!4` and `!19`, with the same name and age, `"mary"` and `25`, respectively, play a part in answering the query. In the next example, sending `..`t (a shorthand for `ith(`t`)`) to an object i returns the object i' referred to by i as its t-coordinate. Sending `iin` to an object i returns the objects i' which refer to i as an association element. The query reads: *what is the name of companies* `C` *located in* `edinburgh` *that are owners?*

```
roll ?- C[.companyName => Name],C[iof => company], C[.location => L],
        L[..city => "edinburgh"], O[iof => owners], C[iin => O].
...
[C = !21, Name = scotplanes, L = !14, O = !16]
...
```

Finally, to retrieve all property-value pairs of all objects:

```
roll ?- all Object[.Property => Value].
[Object = !1, Property = companyName, Value = scotcars]
[Object = !1, Property = location, Value = !2]
[Object = !3, Property = age, Value = 30]
[Object = !3, Property = income, Value = 67890]
[Object = !3, Property = name, Value = john]
[Object = !3, Property = purchase, Value = 0]
...
```

The conciseness achieved in retrieving all subtype-supertype and all property-value pairs illustrates well the usefulness of logical variables in queries.

3.2 Overloading, Overriding and Late-Binding Methods

From the point of view of users, the features known as overloading, overriding and late-binding are among the most distinctive of the object-oriented paradigm. Currently, many well-known proposals for deductive object-oriented databases run into problems in their handling of the three features by being simplistic, or even wholly dismissive, as discussed in Section 5. The way ROLL accounts for all three features is intuitive from both the logic-programming and the object-oriented points of view, and gives ROLL querying a truly object-oriented look-and-feel. To illustrate the three features, consider the following query: *what are the identification* `Id` *and class* `C` *of the objects* `O` *?*

```
roll ?- O[identify => Id], O[iof => C].
...
[O = !1, Id = scotcars, C = company]
another? ;
[O = !3, Id = john, C = customer]
...
[O = !4, Id = 12345, C = seniorEmployee]
...
```

From Figure 1 it can be seen that `identify` is a domain-dependent user-defined operation. That it is overloaded can be seen from the excerpt above, where it is answered by objects belonging to three different types. Since operations are inherited and Figure 1 shows that `customer` and `seniorEmployee` lie along an inheritance path, overriding takes place: a `customer`, like `!3`, being identified by `name`; a `seniorEmployee`, like `!4`, by `ni`, the national-insurance number. The method available in the object's most specific class is the one selected by default, and there is one and only one such method by a well-formedness property imposed on the schema.

ROLL handles methods as an integral part of a domain definition, i.e. database designers include in the intensional part of the database a method definition to implement each domain-dependent operation. Such implementations are, if ROLL is the chosen language, ROLL programs. Given an operation μ and an object type t, assigning a ROLL program to the pair $<\mu, t>$ defines a method in the necessary way for overloading, overriding and late-binding to be catered for. Because of inheritance, methods assigned to other classes may be available at any single class. For instance, as depicted in Figure 1, the operation name is distinctly defined on five different classes, viz. `company`, `carManufacturer`, `person`, `employee`, `juniorEmployee`, and is available through inheritance on yet others, e.g. `permanentEmployee`. The three (of a total five) different implementations of `identify` used in the last example query are as follows:

```
method(identify,person,
      {X as person[identify => Y] <- X[.name => Y]}).

method(identify,employee,
      {X as employee[identify => Y] <- X[.ni => Y]}).

method(identify,company,
      {X as company[identify => Y] <- X[.companyName => Y]}).
```

In the above, the ROLL program implementing each method lies between the pair of curly brackets. Note that the *role* in which the object should respond to message is explicitly stated using the syntactic marker `as`. The way ROLL uses the *role* annotation in queries is addressed later. To late-bind methods to messages (ROLL goal-atoms, in the present case), it is necessary to extend the proof procedure to select the most adequate implementation taking into account the classes of the addressee. In ROLL's underlying model 'most adequate' is stipulated to mean 'most specific', of which there is one and only one, as previously pointed out. From this stipulation and the fact that `employee` is a more specific class than `person`, objects that happen to be instances of both `person` and `employee` respond to messages using the method defined in the latter class, as the last example query has shown.

Static type checking was not implemented in the prototype described in this paper. The result types of `identify` in the three excerpted answers above are `name`, `ni`, and `companyName`. The data model leaves it to designers to stipulate a common supertype, e.g. `identification`, for types requiring a guaranteed comparability. This requirement may propagate upwards till the definition of a *unique* most general class than any other, e.g. `objects`.

ROLL also allows for user-defined binding through the annotation of the addressee with a role. The next query shows how the outcome varies accordingly:

```
roll ?- X[identify => Y], X as person[identify => Z].
...
[X = !4, Y = 12345, Z = mary]
...
```

3.3 Recursion

As pointed out in [26], support for object identity introduces certain problems for DOODs, especially relating to the creation of objects by the proof procedure. However, the magnitude of this problem has sometimes been overestimated, as discussed in [14], and this section briefly presents a solution which is supported in ROLL.

To set the scene, note that in a Prolog procedure, such as `display/1` below, the intermediate arguments with which the procedure is recursively invoked have one and the same nature under the data model, viz. value-based tuples.

```
display([]).
display([H|T]):- write(H),nl,display(T).
```

However, under an object-oriented data model the intermediate argument with which the procedure is recursively called must be an identity-based object whose state is that resulting from the recursion step in which the call takes place. Thus a naive implementation of a `display` method in a DOOD will create a new (potentially persistent) object, whose state is to contain the remainder of the collection, every time a call is made to the recursive clause of `display`.

In ROLL such objects are given *volatile identities* which, intuitively, have an existence that is terminated at the end of the proof within which they were created. The particular value defined for these volatile identities is a unary Skolem function `void` (for volatile `oid`) on the identifier of the object from which the volatile one is derived in the recursion step. The implementation of recursion is discussed in Section 4. Two BIPs correspond to 'first' and 'rest' functions on type-homogeneous collections, i.e. on associations and sequentiations (in the former case, the 'first' element is chosen at random). They are written as `>->` and `>=>`, respectively. In what follows, assume that object `!16` is built by association, and contains references to objects `!18` and `!21` in the construction part of its state. The following queries illustrate the use of these two BIPs:

```
roll ?- !16 >=> Rest, Rest >-> First, First[.Prop => Value].
[Rest = void(!16), First = !21, Prop = companyName, Value = scotplanes]
another? ;
[Rest = void(!16), First = !21, Prop = location, Value = !14]
another? ;
no
roll ?- !16 >=> R, R >=> RestOfR, RestOfR >=> RestOfRestOfR.
no
```

The first query shows that remaindering the object named `!16` creates an object named `void(!16)` in whose construction the first object is named `!21`, the properties of which can be retrieved in the normal way. The second query shows that applying `>=>` three times to `!16` fails, because the cardinality of its construction is only 2.

The ROLL equivalent to Prolog's `display/1` makes use of the above BIPs. In the case of the example domain, `display` is implemented as a method assigned to class `owners`, whose instances are built as associations of `company` objects.

```
method(display,owners,
    {   X as owners[display] <- ~(X >-> H)
    #   X as owners[display] <-   X >-> H, H[write], X >=> T,
                                  T as owners[display] }).
```

Within the definition of `display`, the operator `#` is used to separate ROLL clauses in the program which implements it in the class `owners`. The built-in unary predicate `~` implements negation, as described further in Section 4. The BIP `write` prints out at top level the (name of the) object to which it is sent. Sending the query `!16[display]`, where `!16` is an instance of `owners` results in observable behaviour that is equivalent to the Prolog version.

4 Prototype

A prototype has been built on top of MegaLog [9] whose purpose is twofold:

Implementing the data model of [14] - Given a schema definition, a set of objects and a set of methods, the prototype provides facilities for storing and checking the schema for well-formedness and well-definedness, for storing and checking each object for well-typedness under the schema, taking into account each type the object is assigned to, and for compiling each method (as described further down in this section) and storing it with the other data. Schemas and objects are given a concrete representation that closely resembles the abstract set-theoretic notation of [14], while rules and methods are input as sets of ROLL clauses, the latter bound to the pair <*operation, type*> which identifies a particular implementation. Internally, all the above database elements are represented as first-order terms, and the database thus generated is made to persist using the secondary storage provided by MegaLog.

Implementing an interpreter for ROLL - A mapping was defined that, given a database element, returns an *image* of that element in the form of a set of CROLL clauses (CROLL, as explained below, is an extremely trimmed-down subset of pure Prolog). Based on this, the prototype acts as a user-level front-end which accepts ROLL goals, compiles them into CROLL goals, and, by means of a meta-interpreter, resolves them against the union of the *image* database (i.e. the database transformed into a set of CROLL clauses by the mapping previously alluded to) with the methods and rules previously compiled into CROLL. Here, both the image database and the methods and rules, are stored as clauses which, from the point of view of the meta-interpreter, comprise the stored program against which goals are evaluated.

The rationale behind this approach is consistent with our belief that the data model must be independently defined. Note that the two stages above are distinct. They are conceptually bound by an explicit mapping to effect the logical reconstruction of a database as a logic program. This makes it less important which concrete syntax and internal representation is chosen for either, because within reasonable bounds one syntax can be translated into the other and syntactic sugar can be easily added.

Some interesting features of the prototype are the static approach to inheritance (i.e. all propagation of properties and operations from super- to subtypes is computed at schema-installation time) and to the extension of semantic relationships (i.e. all semantic relationships have their closures computed and their entire extensions stored at schema-installation time). This approach has the main advantage that it eliminates the need for the prototype to handle properties of semantic relationships by means of inference rules. In other words, once the semantic relationships have had their closures computed and stored, a great number of goals that would otherwise require special consideration or the invocation of specific inference rules, can be solved in a single resolution step against the ground facts in the image database.

We now briefly describe how a database can be mapped into a set of clauses in a Horn language (precise details are found in [15].) The mapping amounts to a logical reconstruction of our model along the lines first suggested by Reiter [24] in the relational case. As mentioned before, due to the relative ease with which parsers and meta-interpreters can be written in Prolog, it was decided to recast a database as a set of clauses in a language referred to as *CROLL* (Compiled ROLL). Essentially, CROLL is a very restricted subset of pure Prolog. The alphabet which determines the language has the following main features:

- No function symbols, as usual in the database field.
- Constant symbols are those introduced by the schema and by the set of objects plus a number of *keywords*, some of them interpreted by the semantic relationships defined by the data model (e.g. `isa`, `iof`, `has`, etc.), others interpreted as BIPs (e.g. `equals`, which is the CROLL equivalent of the Prolog `is`).
- There is only one 4-ary predicate symbol `m` (for `m`essage).

The notions of term, literal, clause, fact, rule and goal are as in Prolog. CROLL atoms have the general form $\mathtt{m(}\alpha\mathtt{,}\beta'\mathtt{,}\gamma\mathtt{,}\delta\mathtt{)}$ corresponding to a ROLL atom of the form $\alpha\mathtt{[}\beta\mathtt{(}\gamma\mathtt{)=>}\delta\mathtt{]}$, as defined in Section 3. The nature of β' and its relationship with β can be explained as follows.

If β is a keyword, then β' has a one-to-one correspondence with β, and either identifies the invocation of a BIP (e.g. $\beta' = \mathtt{equals}$) which the meta-interpreter has support for, or else it acts as a discriminator between sets of clauses in the image database (i.e. a role similar to that of predicate symbols in determining the definition of a procedure by the sharing of a symbol in the heads of clauses). For instance, if $\beta = \beta' = \mathtt{isa}$, then the CROLL atom expresses a fact about the structural subsumption relationship. The mapping which yields the image database places identifying keywords at β' according to the kind of fact expressed by the corresponding clause as described in detail in [15]. Some instances of schema-level CROLL facts:

```
m(carManufacturer, isa, [], [carManufacturer]).
m(carManufacturer, isa, [], [company]).
```

If β is an operation name μ, then β' is a pair $<\mu, \theta>$, where θ is the type in which the method implementing μ is applicable. Note that, in a ROLL query, if the user refrains from specifying the desired binding, θ remains free after compilation into CROLL, and it is up to the meta-interpreter to provide the late-binding of θ to the most specific class of α for which μ is defined. For instance, (`arigeq` identifies the arithmetic greater-or-equal BIP) the method `over_18`

```
method(over_18,employee,
    {   X as employee[over_18] <- X[.age => Y],
                                  Y :>= 18 }).
```

is compiled into

```
m(X1, over_18 - employee, [], []) <-
   m(X1, iof, [], [employee]),
   m(X1, get, [age], [X4]),
   m(X4, arigeq, [18], []).
```

The mapping, informally and in broad terms, takes advantage of the fact that the extension of semantic relationships is stored by making each tuple[1] in a relationship correspond one-to-one with a CROLL fact. This captures as CROLL facts practically all the schema-level information. Similarly, at the instance level, each object is translated into a ground theory, i.e. a set of CROLL facts, each of which is distinguishable by keywords that correspond to the kinds of reference an object makes to other objects or to types. These correspond to the semantic relationships holding between an object and its assigned types, and between an object and the objects it references as a value of some property or as a component in its construction. The end result of this mapping is referred to as the *image* database. The image database is a subset of the CROLL Herbrand base, i.e. the extensional database over which the meta-interpreter is to act. As an example, consider the ground theory corresponding to the object !2 of type `location`, with no properties and a construction with three coordinates `street, city, postcode`:

```
m('!2', iof, [], [location]).
m('!2', abst, [], [aggregation]).
m('!2', cohe, [], [nonc]).
m('!2', ith, [street], ["grassmarket"]).
m('!2', ith, [city], ["edinburgh"]).
m('!2', ith, [postcode], ["eh12hj"]).
```

The intensional database comprises the set of compiled methods previously mentioned, as well as deductive rules used to characterize derived data and to define integrity constraints not directly captured by the modelling formalism. Note that deductive rules, like methods, are input as ROLL programs and undergo compilation into CROLL before being stored. The basic meta-interpreter is:

```
roll_solve(true).
roll_solve((A,B)):-
    roll_solve(A),
    roll_solve(B).
roll_solve(m(A,X,AL,RL)):-
    late_bind(m(A,X,AL,RL),m(A,Y,AL,RL)),  % ROLL extension
    croll(m(A,Y,AL,RL),C),
    roll_solve(C).
```

The extended features occur in the third clause, where `croll/2` is equivalent to Prolog's `clause/2`, i.e. it is used to retrieve CROLL clauses from MegaLog relations (this interaction with MegaLog uses a slightly different syntax). Recall that `X` may be a pair $<\mu, \theta>$. In this case, if θ is free then the call to `late_bind/2` results in the binding of θ, in `Y`, to the most specific class of which `A` is an instance that has μ defined for it. The call also ascertains that the arguments `AL` and the result `RL` are well-typed. If θ is bound then the call only serves to perform type-checking.

The basic meta-interpreter is extended to handle collected multiple-answers, BIPs and (Prolog-style) negation-as-failure, by the inclusion of the following as third, fourth and fifth clauses of the meta-interpreter:

```
roll_solve(findall(Vars,A,Answers)):-
    findall(Vars,roll_solve(A),Answers).    % collect multiple-answers
roll_solve(A):-
    roll_built_in(A).                       % handling of BIPs
roll_solve((not A)):-
    not roll_solve(A).                      % negation as failure
```

where example of clauses belonging to the definition of `roll_built_in/1` are:

```
roll_built_in(m(X,write,[],[]))                 :- write(X),nl.
roll_built_in(m(Answer,equals,[Expression],[])) :- Answer is Expression.
```

[1] This is set-theoretic, not relational database, terminology

which handle the output of ROLL terms and evaluation of arithmetic expressions, respectively. Note that the definition of `roll_built_in/1` essentially uses the host language (in this case, MegaLog Prolog) to implement the BIPs of the user-level language.

The last extension to the meta-interpreter is needed to handle recursion as described in Section 3. It requires that `roll_solve/1` and `roll_built_in/1` have their arity increased to 3. The two additional argument places are lists of CROLL clauses, corresponding to the volatile objects created during the proof by the use of the *first* and *rest* BIPs. The second argument of `roll_solve/3` comes instantiated with the volatile objects created thus far, while the third argument exits with the previous volatile objects plus any others created by the current proof step. The complete meta-interpreter with additional arguments and an additional sixth clause is:

```
roll_solve(true,U,U).
roll_solve((A,B),U,W):-
    roll_solve(A,U,V),
    roll_solve(B,V,W).
roll_solve(findall(Vars,A,Answers),U,V):-
    findall(Vars,roll_solve(A,U,V),Answers).
roll_solve(A,U,V):-
    roll_built_in(A,U,V).
roll_solve((not A),U,V):-
    not roll_solve(A,U,V).
roll_solve(A,U,U):-
    memberchk(croll(A,true),U).             % try volatile facts
roll_solve(m(A,X,AL,RL),U,V):-
    late_bind(m(A,X,AL,RL),m(A,Y,AL,RL),U),
    croll(m(A,Y,AL,RL),C),
    roll_solve(C,U,V).
```

i.e. an additional way of proving a CROLL goal is to resolve it against a fact about volatile objects created thus far in the proof.

5 Related Work

In this section our approach to deductive object-oriented languages is compared with some selected alternative proposals. First, note that the problem of integrating object-orientation and logic programming can be tackled either by extending an object-oriented system with some form of deduction, or by extending logic-programming languages with object-oriented concepts. All the proposals discussed here adopt the latter viewpoint. Further, one can bias the work towards programming-language issues or towards databases issues. Again, the present proposal as well as those discussed below are mainly concerned with the latter. Three main approaches can be used to develop a DOOD language against the background of previous research into deductive relational databases (DRDBs):

1. *extending the deductive language* - this characterizes work such as the COL/IQL/IQL+ family of languages [1, 4, 5], in which DATALOG [10], seen as the logic-programming language underlying DRDBs, is progressively extended with complex typed-terms, object identity, inheritance and so on.

2. *redesigning the deductive language* - this characterizes the work that resulted in the family composed of the C-/O-/F-logic languages [11, 19, 21]. Here, no data model is assumed in the background, rather there is the intention of expressing directly at the logic-language level all the features deemed necessary to model a domain in an object-oriented style.

3. *redesigning the data model* - this characterizes our work [16, 14]. Here, the underlying data model is defined independently, in a deliberate attempt to retrace, for the object-oriented case, the development path that led from the relational model to DATALOG. The idea is to obtain for classes, objects, and methods on one hand and clauses of a logic language on the other, the same sort of structural similarity that holds between relations and DATALOG clauses. This also has the advantage that imperative features can be handled by a separate language component, which can be seamlessly integrated with the logic language by virtue of their sharing a single type system.

The remainder of this section focusses on IQL+ and F-logic (as the latest members in their respective families) and contrasts these approaches with ROLL. However, it should be stressed that ROLL is but one part of a three-component architecture [16], while most other proposals for DOOD are based on single-component architectures.

IQL+ was proposed as a stand-alone language, powerful enough to tackle small-scale application development and resorting to external functions to extend its capabilities. Its data modelling constructs are less comprehensive than those available in our model (e.g. it does not cater for multiple inheritance; also, it has no means of expressing existence dependences required to inform update propagation of composite objects), but they satisfy usual requirements. One point of contrast with both ROLL and F-logic is that schema expressions in IQL+ are not part of the logic language, and hence they cannot be queried. IQL+ has a number of control structures to drive the computation of fixpoints. In certain respects, IQL+ resembles more a production-rule system than a logic programming one. No declarative semantics has been proposed to characterize the meaning of an IQL+ program.

F-logic is, in a different way, a more complex proposal. Its avoidance of data modelling issues is, we believe, a serious obstacle to its acceptance as a DOOD language. While ROLL is strongly influenced by semantic data-modelling research, with its emphasis on the precise characterization of abstraction mechanisms to model data, F-logic conflates the schema- and instance-levels by using the same linguistic construct to express the membership relationship between an individual and its class and the inclusion relationship between a sub- and a superclass. Since all three languages are semantically first-order, all are similarly bound to interpret constants in the language as individuals in the world, which may imply some form of reification of types, for example. However, in both ROLL and IQL+ one is not short of means for distinguishing membership from inclusion. The F-logic proposal is closer to ours in its provision of a declarative and a procedural semantics for the language. F-logic also has no control mechanisms and does not handle updates. The proposal of F-logic comes with a proof procedure for the monotonic part of the language and a unification algorithm for F-terms. Both the unification algorithm (which has to handle set unification and a higher-order syntax) and the proof procedure (which uses a number of additional inference rules to handle inheritance and typing issues) pose non-trivial problems for efficient implementation.

There is a close analogy between ROLL and DATALOG, in that several desirable properties of DATALOG programs (e.g. concerning termination) hold for ROLL also. It can be seen from the meta-interpreter presented in Section 4 that giving a procedural semantics to ROLL programs based on an extended SLDNF proof procedure is straightforward. In general, the semantics of ROLL programs is as intuitive as those of DATALOG programs, a characteristic which, we believe, makes it very attractive as a query language for end-users. This is only possible because critical issues of control and updates are handled by a separate language component.

Neither IQL+ nor ROLL should present particular difficulties for implementation, although the style used for that task in each case would be probably different. An implementor for ROLL is able to benefit directly from the large body of work carried out in recent years in implementing DRDBs and, more generally, Horn-clause languages. In contrast, F-logic proposes implementation problems of its own to handle the more complex unification algorithm and proof procedure, unless a compilation approach similar to the one described in Section 4 is used, reducing the burden on the proof procedure.

The most important point of contrast between the three proposals, however, concerns the account for those core notions that characterize the object-oriented paradigm. Unless these notions are clearly and convincingly accounted for, doubts will always be cast on whether the end-result qualifies as *object-oriented*. Object identity and inheritance are well-catered for by most proposals. The two crucial areas of difference are *encapsulation* and the *overriding/-overloading/late-binding* trio.

F-logic considers methods to be essentially properties of objects. This leads to an account of encapsulation that resembles that of private views, i.e. programs are specialized to users according to their signatures for each class object. This is ingenious, but in no way a very faithful account of how encapsulation is construed in object-oriented systems. It conflates structural and behavioural inheritance and preempts the issue of late-binding by not acknowledging the existence of separate implementations (i.e. programs). IQL+ uses modules to account for encapsulation and treats overloading of methods as a non-monotonic feature, and, similarly to F-logic, also preempts the issue of late-binding.

The approach to giving a semantics to ROLL, viz. by rewriting ROLL into Horn clauses, has a close similarity to the one used for OIL [27] and OOLP+ [12] (OIL acquires a semantics by a rewriting into $\mathcal{LDL}$ and OOLP+ by a rewriting into Prolog), but otherwise both proposals are very different from ROLL, not least in their treatment of object as terms, as opposed to objects as theories as is the case with ROLL.

In contrast to all above proposals, ROLL is quite faithful to the object-oriented approach. It supports strict encapsulation by a clear-cut distinction between structure and behaviour: encapsulated access to the state of objects can be enforced simply by restricting legal queries to those that use operation names *only*. As for overriding, overloading and late-binding, we have given examples and explained the technique used to account for them in a way that is true to both paradigms.

6 Future Directions and Conclusions

ROLL has been completely formalized in [18], a task that involved defining the language from its foundations in the data model of [14] and showing how it extends the class of Horn programs with accounts for semantic relationships, methods, encapsulation, overriding, overloading and late-binding.

The surface syntax of ROLL will be defined in consonance with the imperative database-programming language of [8]. Our goal is to obtain an integrated language which is free from impedance mismatches. The single underlying type-system is instrumental in that respect: objects retrieved by ROLL expressions within the imperative language are instances of types which the imperative language itself has defined. The other dimension of mismatch, namely, the computation paradigm, should not be a difficult issue as both languages rely on message-sending as the means to drive computation. Also, the rich control structures of the imperative language can accommodate a great variety of styles. As indicated in Section 1, the DOOD system will be built on top of EXODUS. After the surface syntax of the logic-language component has been defined, an interpreter for it will be implemented. This will naturally imply the choice of efficient techniques to evaluate ROLL queries both as a stand-alone language and as a component.

Our experience in prototyping an interpreter for ROLL has convinced us that there are important advantages in taking a model-based approach to the problem of extending OODBs with deduction. The lack of a widely-agreed formal model for OODBs has only made things look more forbidding, which has led some to try and accommodate within the relational model the necessary extensions, and others to skip the modelling issue altogether. As described in this paper, the main contribution of ROLL is to demonstrate that by formalizing OODBs in terms that are close to the roots of logic-programming theory, one is able to retrace for DOODs the simple extension path that proved so successful for DRDBs.

Acknowledgements The work that resulted in this paper has been funded by the Science and Engineering Research Council through the IEATP programme, and their support is duly acknowledged. We would also like to thank our colleagues Maria Luisa Barja and Alia Abdelmoty, and Dr. Keith G. Jeffery of Rutherford Appleton Laboratory for useful discussions on the subject of this paper, and Dr. J.M.P. Quinn representing ICL and Mr Neil Smith of Ordnance Survey for their contribution as the industrial partners in the project.

References

[1] Serge Abiteboul. Towards a Deductive Object-Oriented Database Language. *Data & Knowledge Engineering*, 5:263–287, 1990. Preliminary Version in [23], pages 453–472.

[2] Serge Abiteboul and Stéphane Grumbach. COL: A Language for Complex Objects Based on Recursive Rules. In *Proceedings of the Workshop on Database Programming Languages*, pages 253–276, Roskoff, 1987. Complete version in [4]. See also [3].

[3] Serge Abiteboul and Stéphane Grumbach. COL: A Logic-Based Language for Complex Objects. In Joachim W.Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Advances in Database Technology - EDBT'88, International Conference on Extending Database Technology*, LNCS 303, pages 271–293, Venice, Italy, March 1988. Springer-Verlag. ISBN 3-540-19074-0, ISBN 0-387-19074-0. See also [2, 4].

[4] Serge Abiteboul and Stéphane Grumbach. COL: A Logic-Based Language for Complex Objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, Frontier Series, chapter 21, pages 347–374. ACM Press/Addison-Wesley Publ. Co., New York, NY, 1990. Abstract in [2]. See also [3]. ISBN 0-201-50257-7.

[5] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 159–173, Portland,OR, June 1989. ACM Press. SIGMOD Record, 18(2), June 1989. This is a preliminary version of [6].

[6] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. Technical Report 1022, INRIA, April 1989. A preliminary version appears as [5].

[7] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley B. Zdonik. The Object-Oriented Database System Manifesto. In *[23]*, pages 223–240. 1990.

[8] Maria L. Barja, Norman W. Paton, and M.Howard Williams. Design of an Object-Oriented Database Programming Language for a DOOD. Technical Report TR92016, Department of Computing and Electrical Engineering,Heriot-Watt University, Edinburgh, Scotland, UK., December 1992.

[9] Jorge Bocca. Megalog: A Platform for Developing Knowledge-Base Management Systems. In *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pages 374–380, Tokyo, 1991.

[10] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990. ISBN 3-540-51728-6, ISBN 0-387-51728-6.

[11] Weidong Chen and David S. Warren. C-Logic of Complex Objects. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369–378, Philadelphia,PA, March 1989. ACM Press.

[12] Mukesh Dalal and Dipayan Gangopadhyay. OOLP: A Translation Approach to Object-Oriented Logic Programming. In *[23]*, pages 593–606. 1990.

[13] Claude Delobel, Michael Kifer, and Yoshifumi Masunaga. *Deductive and Object-Oriented Databases (Second International Conference DOOD'91, Munich).* Springer-Verlag, Berlin, 1991. ISBN 3-540-55015-1, ISBN 0-387-55015-1.

[14] Alvaro A. A. Fernandes, Norman W. Paton, M. Howard Williams, and Andrew Bowles. Approaches to Deductive Object-Oriented Databases. *Information and Software Technology*, 34(12):787–803, December 1992.

[15] Alvaro A.A. Fernandes. An Object-Oriented Logic by Examples. Technical Report TR92015, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland, UK., September 1992.

[16] Alvaro A.A. Fernandes, Maria L. Barja, Norman W. Paton, and M. Howard Williams. A Deductive Object-Oriented Database for Data Intensive Application Development. In *Advances in Databases: Proceedings of the 11th British National Conference on Databases (BNCOD 11)*, LNCS 696, pages 176–198, Keele, UK, July 1993. Springer-Verlag. ISBN 3-540-56921-9, 0-387-56921-9.

[17] Alvaro A.A. Fernandes, M. Howard Williams, and Norman W. Paton. A Formal Abstract Definition of Objects as a Data Modelling Primitive. Technical Report TR92003, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland, UK., April 1992. Revised June 1992.

[18] Alvaro A.A. Fernandes, M. Howard Williams, and Norman W. Paton. An Axiomatic Approach to Deductive Object-Oriented Databases. Technical Report TR93002, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland, UK., April 1993.

[19] Michael Kifer and Georg Lausen. F-logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 134–146, Portland,OR, June 1989. ACM Press. SIGMOD Record, 18(2), June 1989. A revised and extended version exists as [20].

[20] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 93/06, Department of Computer Science, State University of New York at Stony Brook (SUNY), April 1993. Revised June 1993.

[21] Michael Kifer and James Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–383, Philadelphia,PA, March 1989. ACM Press. Preliminary Report of [22].

[22] Michael Kifer and James Wu. A Logic for Programming with Complex Objects. Technical report, Department of Computer Science, State University of New York at Stony Brook (SUNY), 1989.

[23] Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors. *Deductive and Object-Oriented Databases (First International Conference DOOD'89, Kyoto).* Elsevier Science Press (North-Holland), Amsterdam, 1990. ISBN 0-444-88433-5.

[24] Raymond Reiter. Towards a Logical Reconstruction of Relational Database Theory. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–233. Springer-Verlag (Topics in Information System Series), New York, NY, 1984.

[25] Joel E. Richardson and Michael J. Carey. Implementing Persistence in E. In John Rosenberg and David Koch, editors, *Persistent Object Systems: Proceedings of the Third International Workshop*, pages 175–199. Springer-Verlag, London, 1989.

[26] Jeffrey D. Ullman. A Comparison Between Deductive and Object-Oriented Database Systems. In *[13]*, pages 263–277. 1991.

[27] Carlo Zaniolo. Object Identity and Inheritance in Deductive Databases – An Evolutionary Approach. In *[23]*, pages 7–24. 1990.