An Effective Deductive Object-Oriented Database Through Language Integration

Maria L. Barja, Norman W. Paton, Alvaro A.A. Fernandes, M. Howard Williams, Andrew Dinn

Department of Computing and Electrical Engineering,
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS, Scotland, UK
e-mail: <marisa,norm,alvaro,howard,andrew>@cee.hw.ac.uk
phone: +44-31-449-5111 ; fax: +44-31-451-3431

Abstract

This paper presents an approach to the development of a practical deductive object-oriented database (DOOD) system based upon the integration of a logic query language with an imperative programming language in the context of an object-oriented data model. The approach is novel, in that a formally defined data model has been used as the starting point for the development of the two languages. This has enabled a seamless integration of the two languages, which is the central theme of this paper. It is shown how the two languages have been developed from the underlying data model, and several alternative approaches to their integration are presented, one of which has been chosen for implementation. The approach is compared with other examples of language integration in a database context, and it is argued that the resulting system overcomes a number of important challenges associated with the development of practical deductive object-oriented database systems.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

1 Introduction

Designers of novel database systems and languages are often faced with situations in which compromises have to be made in the context of seemingly conflicting goals. In the case of data models, the addition of new constructs makes the model more expressive, but also increases its complexity. In the case of query languages, increases in the power of a language often complicate its underlying semantics or impact upon the nature/practicality of effective optimisation (e.g. the addition of update mechanisms to Datalog requires extensions to its fixpoint semantics, and the associated addition of control structures may limit opportunities for optimisation [CGT90]). As a result of this tension, database programming languages have been proposed which have very different areas of strength and weakness, and thus individual languages are suited to particular data management tasks, but not to others.

One strategy which has been adopted to overcome the weaknesses of particular approaches for certain tasks has been to integrate different programming paradigms for use with databases. The nature of such systems is considered further in section 2, where a number of examples are presented, but the broad idea is that an integrated system may be greater (in some sense) than the sum of its constituent parts. This paper presents a database programming system in which a logic query language ROLL (Rule Object Logic Language) is integrated with an imperative programming language ROCK (Rule Object Computation Kernel) in the context of a common underlying data model (OM). The overall architecture of the system is presented in figure 1. The resulting system supports a range of standard object-oriented mechanisms for structuring both data and programs, while allowing different and complementary programming paradigms to be used for different tasks, or different parts of the same task.

The focus in this paper is on the integration of the logic query language with the imperative manipulation language, rather than on the association of the logic query language with the underlying data model (as described in [FWP94]) or the relationship between the imperative language and the underlying data model (as described in [FBPW93]). These components, ROLL, ROCK and OM are introduced in section 3 to give a context for the consideration of the integration issues presented in section 4.

After the integration has been presented, a valid question which remains unaddressed is the effectiveness of the resulting system as a DOOD. In section 5 the integrated system (ROCK & ROLL) is discussed in the context of the issues raised by [Ull91], where it is argued that there are significant obstacles which must be overcome by any system which is to be fully deductive and fully object-oriented. We believe that ROCK & ROLL successfully addresses the issues raised by Ullman, in a way which yields a practical DOOD system. Conclusions are presented in section 6.

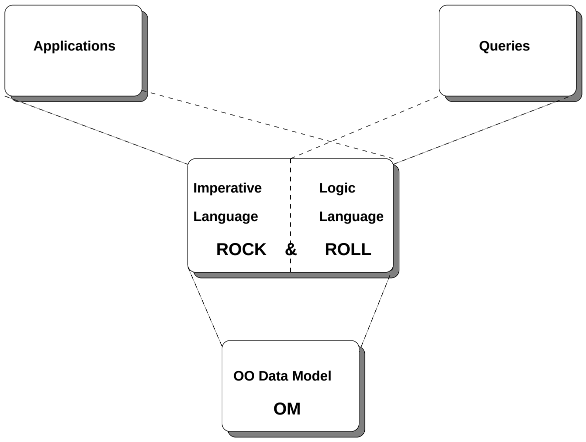

Figure 1: Relationship between the principal components in the architecture.

2 Related Work

Many proposals have been made for systems which integrate different languages for manipulating persistent data. Indeed, the notion of *impedance mismatch* in both its aspects (type system mismatch and evaluation strategy mismatch) was put forward to characterise certain ways in which an 'integration' may be less than seamless. The impedance mismatch is likely to manifest itself most strongly where the languages which are being used together were developed independently (e.g. when SQL is embedded in C), although most recent proposals that exploit integration involve languages which were designed with subsequent integration in mind. The following are criteria against which an integration can be judged:

- *Evaluation strategy compatibility.* This criterion is part of the standard impedance mismatch, and indicates whether the evaluation strategies used by the two languages are compatible.
- *Type system uniformity.* This criterion indicates whether all values which are first-class citizens in one language are first-class citizens in the other.
- *Type checker capability.* This criterion indicates whether or not strong compile-time type checking is carried out across the interface between the two languages.
- *Syntactic consistency.* This criterion indicates the degree to which the syntaxes of the integrated languages are consistent. The syntaxes are sure not to be identical, but in a syntactically consistent integration a similar syntax should be used on both sides of the interface to perform similar tasks (e.g. compare two strings, invoke a method).
- *Bidirectionality.* This criterion indicates whether or not it is possible for each language to call the other, or whether one is essentially embedded in the other.

Table 1 indicates the extent to which a number of systems satisfy these criteria, the specific systems being described briefly in what follows. The symbol $\approx$ in Table 1 indicates that the criterion is partially satisfied.

Coral++ [SRSS93] is essentially an interface from the deductive database system Coral to C++, whereby C++ objects can be accessed from Coral programs using some conservative extensions to the Coral language. Because two independently developed systems have been combined, the integration is not particularly seamless.

Embedded-SQL [Dat90] is the principal means by which complete applications are developed using the relational model, the query language SQL being embedded in an imperative language such as C. This is essentially a mechanism for allowing imperative programs to access and update relational databases rather than an attempt at a smooth integration.

Glue-Nail [PDR91] is a combination of the logic query language NAIL! with the imperative programming language Glue in the context of an essentially relational data model. In common with

Table 1 – Characterising Seamlessness of Language Integration					
System	Evaluation strategy compatibility	Type system uniformity	Type checker capability	Syntactic consistency	Bidirectionality
Coral++	$\times$	$\times$	$\times$	$\approx$	$\times$
Embedded-SQL	$\times$	$\times$	$\times$	$\times$	$\times$
Glue-Nail	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
PFL	$\surd$	$\approx$	$\surd$	$\surd$	$\surd$
Raleigh	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
ROCK & ROLL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

ROCK & ROLL, such an approach avoids complicating the semantics of the logic language with update facilities, but also allows modification of the underlying database. The fact that Glue was designed specifically as an update language for NAIL! means that the impedance mismatch can be minimised.

PFL [PS91] is a lazy functional programming language with an embedded query language construct influenced by logic query languages.

Raleigh [KR91] is an imperative database programming language with a functional flavour, into which can be embedded query language statements based upon `select` and `for each` constructs.

The process of integrating two intrinsically different languages inevitably leads to the introduction of a *paradigm mismatch*. The paradigm mismatch concerns the efficacy of the integration, and introduces two further issues which are relevant to the utility of an integration:

- *Complementarity*, which indicates whether an integrated system is likely to be significantly more useful than its individual components.
- *Reorientation*, which indicates how straightforward it is for a programmer to switch between the two paradigms in a single application.

These measures are considerably more subjective than those considered earlier in this section in connection with the impedance mismatch (which is principally concerned with the way in which an integration is engineered, rather than its desirability), but fully as important to the users of a system. For example, the integration of Glue with NAIL! in Glue-Nail increases the functionality of the NAIL! system by allowing updates while preserving the semantics of NAIL! for query processing, and thus the two languages can be considered to be complementary. It is less clear, however, that the integration of a logic query language with a functional programming language in PFL has led to significant gains – for example, neither language supports updates or I/O. The measurement of costs/benefits associated with reorientation remains an issue for future examination.

The work presented here is also related to other research on deductive object-oriented databases. Consideration of ROCK & ROLL as a DOOD is deferred until section 5, a more comprehensive review of approaches to the design of DOODs being given in [FPWB92].

3 ROCK & ROLL

This section describes the components of ROCK & ROLL – the data model, imperative programming language and logic query language. Each of these components can be considered to be quite conventional – an important theme of the work presented here is that it is possible to smoothly integrate these three components without sacrificing the characteristic virtues of any of them, and without introducing radical new concepts as part of the integration process.

3.1 The Data Model – OM

A brief informal account of the model is now given with the purpose of providing an introduction to the terminology used later in the paper. Atomic values and compound data items are called *primary objects* and *secondary objects* respectively. Each object is assigned an *object type*, and every secondary object has a unique object identifier. The *object-oriented model* informally described below has been formalised in [FWP93] as a class of first-order theories [Men87] called *object theories* (OTs) of which every legal database state is a logical model. This section gives an informal overview of the concepts used to model an application domain both structurally and behaviourally.

A type definition can describe references of two kinds. The first kind of reference definition is used to model the *properties* of the type. This results, for each type, in a possibly empty set of type names which are the attributes of the type. This is slightly unconventional, in that the type name given for each property both names the property and specifies which values it may hold. As an example of some property definitions, the following code fragment indicates that a `roadSegment` has two properties, a `startJunction` and an `endJunction`.

```
type roadSegment
    properties:
        startJunction, endJunction;
    ...
end-type
```

The second kind of reference definition, which is referred to as the *construction* of the type, is used to distinguish the fundamental structural characteristic of a type from its other stored properties. A type can be structured by association, sequentiation or aggregation, which support the modelling of sets, lists and tuples, respectively. For example, the following type definition indicates that a `polygon`, while having the attribute `area`, has as its construction a sequence of `segment` objects (represented by square brackets):

```
type polygon
    properties:
        area;
    [ segment ]
    interface:
      perimeter(): real;
    ...
end-type
```

An object type may also declare a behavioural interface. For example, the definition of `polygon` includes the specification of the signature of the method `perimeter` which returns a `real` result. The actual definition of a method is specified in the class which corresponds to the type – every type is associated with a single class which has the same name as the type. The intention is that the type specifies all that a user of the type needs to know in order to use the type, while method code and other implementation details are specified in the class (for examples of type and class definitions see sections 3.2 and 4.2.2). Methods are defined in a context which supports overloading, overriding and late binding.

Schema diagrams for the data model can be constructed using the following notation. Secondary object types are represented using rectangles, primary object types using ellipses, and operations using rectangles with rounded corners. Labelled directed edges represent modelling features thus: ○ – attributes, △ – specialisation, ⊗ – aggregation, ⊛ – association, and ⊚ – sequentiation. This notation is used to describe part of a geographic database in figure 2. In this example, a `polygon` is constructed from a sequence of `segment` objects, has a stored attribute `area`, and the operations `adjacent`, `connected`, and `perimeter`. The type `landParcel` is a subtype of `polygon`. This schema will be used in examples throughout the paper.

3.2 The Imperative Programming Language – ROCK

The database programming language allows both persistent and transient data to be created and manipulated in a uniform way. ROCK is based on the data model described in section 3.1, and is a strongly typed imperative object-oriented database programming language.

ROCK can be regarded as the conjunction of a data definition language for schema declarations, and a data manipulation language that allows operations to be performed on persistent or transient objects. The types which can be defined using the data definition language are exactly those which are supported by the data model described in section 3.1. The data manipulation language provides constructs for processing such data, and thus is suitable for the development of complete applications. The facilities supported are comparable to those of other object-oriented database programming languages, and include:

- the object creation operator `new`
- assignment
- I/O operations (`read, write, ...`)
- control structures such as selection (`if then else`), iteration (`while, foreach`) and blocks (`begin end`). The `foreach` construct provides for iteration over the instances of a class or over the elements of an association or sequentiation.

All control structures have a mode of operation in which they return an object, or a set of objects in the case of the iterative constructs.

Operations on objects are classified into two groups: built-in (or system generated) and methods (or user-defined). The model of computation adopted in both cases is the messaging one, where the symbol "@" is the message sending operator. The message recipient is an *object expression*, that is, an expression in the language which evaluates to an object. For example, the following expression assigns to the variable `s` the result of sending the message `get_startJunction` to the `roadSegment` `rs`.

```
s := get_startJunction@rs
```

Message sends can be nested, and inside a method, messages can also be sent to *self* and *super*. Support for encapsulation is strict, i.e. the structure of objects can only be accessed through operations, whether system-generated or user-defined.

- System-generated operations

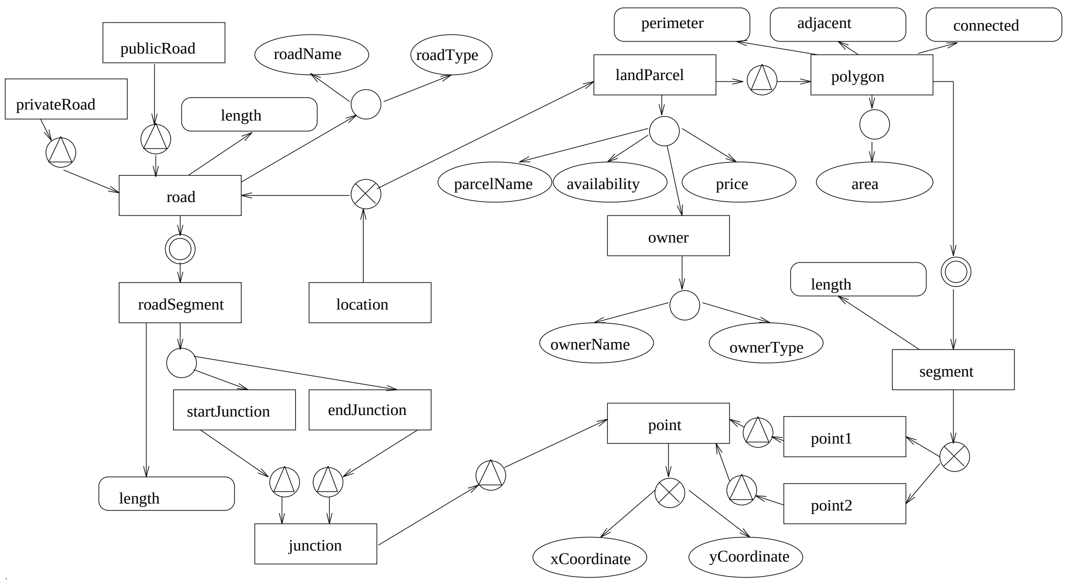

Figure 2: Schema diagram for a fragment of a geographic database

For each property attributed to a type and for each coordinate in an aggregate construction, the compiler generates a pair of methods whose names are those of the corresponding property/coordinate prefixed respectively by `get_` and `put_`. These methods allow the object referenced by a property/coordinate to be retrieved or updated, and their visibility is determined by that of the corresponding property/coordinate.

- User defined operations

  User defined operations are given in the form of methods. Methods have a visibility which is either *public* or *private*. Private methods are not accessible from outside the definition of a class, while public methods are. The signatures of public methods are defined in the interface part of the corresponding type definition.

To illustrate class and method definition, consider the definition of the class `polygon`, associated with the type `polygon` defined in section 3.1:

```
class polygon
    perimeter(): real
    begin
        var per: real;
        foreach s in self do
            per := per + length@s;
        per
    end
    ...
end-class
```

The method **`perimeter`** calculates the perimeter of the polygon by summing the result of the `length` method on each of the segments in the sequence from which the polygon is constructed – the **`foreach`** loop iterates over all the **`segment`** objects in this sequence; the type of `s` is inferred.

The model of persistence defines the class as the unit of persistence, and therefore provides for the persistence not only of data but also of code. The type definition corresponding to a persistent class persists along with the class, as do its instances. Persistent class and type definitions do not need to be rewritten in a program which uses them in order to verify consistency – it is only necessary for a program to name the persistent environments in which the appropriate information is stored. Other classes and types referenced from a persistent class or its associated type must also be persistent. The support for persistence is uniform, that is, the degree of persistence of the objects manipulated by a program does not affect the ways in which the objects can be manipulated.

3.3 The Logic Language ROLL

ROLL is a Horn clause language. Familiarity with the latter subclass of first-order languages is assumed at the level of [CGT90].

The *ROLL alphabet* consists of (typed) logic variables, (typed) constant symbols, and predicate symbols, but does not include function symbols. The set of *constant symbols* is the set of names of primitive objects, i.e. values (e.g. `5`, `"Edinburgh"`, `true`). Note that there is no need for object identifiers to appear as constant symbols in ROLL expressions, as specific objects are either passed into a ROLL expression from ROCK, or are retrieved from the extensional database.

The set of *predicate symbols* is the set of operation names declared by operation interfaces. It follows that ROLL queries abide by strict encapsulation.

A ROLL *term* τ is either a ROLL *constant* or a *(logical) variable*. A ROLL *atom* has the form:

$\beta(\tau_1, \ldots, \tau_n)@\alpha$

or

$\beta(\tau_1, \ldots, \tau_{n-1})@\alpha == \tau_n$

where β is an $(n+1)$-ary ROLL predicate symbol, each τ_i, $1 \leq i \leq n$, is a ROLL term, and α is a ROLL term appearing as the $(n+1)$-th argument of β.

The first form is used when the operation β is implemented in ROLL, which requires no distinction to be made between input and output parameters. A ROLL atom of the above form is read as "send the *message* β with the arguments $\tau_1, \ldots, \tau_n$ to the object α". An operation interface $\beta : T_1 \times \ldots \times T_n$ is assumed to be defined, such that each τ_i denotes an instance of T_i', where $T_i' \leq T_i$, the type subsumption relationship being expressed in the schema.

The second form is used when β is implemented in ROCK, where the (optional) result is explicitly distinguished from any input parameters. An operation interface $\beta : T_1 \times \ldots \times T_{n-1} \rightarrow T_n$ is assumed to exist, such that each τ_i denotes an instance of T_i', where $T_i' \leq T_i$. In this case, the result of evaluating the ROCK method β with the given parameters is unified with τ_n.

If $n = 0$ then $\beta@\alpha =_{def} \beta()@\alpha$. The above forms exist to support the integration of ROCK & ROLL, as discussed further in section 4.

A ROLL *literal* L is a ROLL atom A or a negated ROLL atom $\neg A$. A *ROLL clause* is a disjunction of literals of which at most one is positive. A clause containing a single literal is called a *unit clause*. The notion of a *ROLL fact* (a positive unit clause) though well-defined [FWP93], is not relevant in the context of the ROCK & ROLL system because the asserted facts that describe the state of an object at any point in time are encapsulated and are only made available in computations as responses to message-sends. A *ROLL rule* is a clause with exactly one positive literal and with at least one negative literal. The positive literal is referred to as the *head*, the set of negative literals is referred to as the *body*. Finally, a *ROLL query* is a clause containing negative literals only. The usual convention is followed of rewriting a clause as a reverse implication, i.e. *head* '`:-`' *body*, and replacing conjunctions by commas.

3.3.1 Example Query

The following is an example of a ROLL query which retrieves as bindings for `P` the `polygon` objects with an `area` less than `5` and a `perimeter` greater than `100`. The operator `==` denotes unification, which is used in this case to unify the result of a ROCK message send with a ROLL variable.

```
get_area@P == Area, perimeter@P == Perim,
Area < 5, Perim > 100.
```

This example uses a number of facilities which bear explanation: ROLL queries and methods are strongly typed, and a type inference system is used to infer types for logic variables – for example, that the logic variable `P` is associated with the type `polygon` is inferred from the fact that the methods `get_area` and `perimeter` are applicable to objects of type `polygon` (the type inference algorithm used is comparable to that of SML [MTH90]); in this example there is no explicit iteration over the instances of `polygon` – the ROLL query evaluator optimises a query, and plants iterators within the evaluation graph wherever there is no other way of obtaining a binding for a logic variable associated with an object type.

3.3.2 Example Rules

The following example rules implement the operation `connected` as a relationship indicating which polygons are directly or indirectly connected to each other. The operation `adjacent` uses a rule to define a binary relationship between `polygons`, indicating that two polygons are adjacent if they have a common segment. The rules for `connected` define the transitive closure of `adjacent`. The rule head is defined as shown to reflect the object-oriented structuring of the clause base, as discussed further in section 4.2.2.

```
adjacent(PolyB)@PolyA :-
     PolyA <> PolyB,
     get_member@PolyA == Segment,
     get_member@PolyB == Segment.

connected(PolyZ)@PolyA :-
     adjacent(PolyZ)@PolyA.
connected(PolyZ)@PolyA :-
     adjacent(PolyB)@PolyA,
     connected(PolyZ)@PolyB.
```

In practice, although the notion of a *ROLL program* as a finite set of ROLL clauses, is well-defined [FWP93], it is not relevant in the context of the ROCK & ROLL system because, in the integrated language, ROLL programs are defined in the context of classes as methods, using the syntactic form described in section 4.2.2.

4 Integrating ROCK & ROLL

Many imperative database programming languages provide support for the development of complete data intensive applications, but lack comprehensive facilities for the expression of declarative queries. This is unfortunate, as it is clear that the advanced applications with which such systems are often associated do benefit from the presence of declarative query languages. Other systems emphasise the query language and thus tend to provide less fully integrated application development facilities, which is also a loss, as databases which are commonly accessed through query interfaces are often also manipulated by complex programs.

In the light of this, the integration of the imperative and logic languages is clearly important to the effectiveness of the overall system – it is not sufficient to support two distinct language interfaces, it is necessary to make both languages available to application programmers so that the most appropriate approach can be selected for each part of a complex task. As the two languages have been derived from a common data model, and therefore have a common type system, the process of integrating them without introducing the impedance mismatch has been considerably eased. The integration involves the following aspects:

1. Embedding of ROCK in ROLL. To keep the approach sound, only side-effect free methods can be invoked from within the logic language .

2. Embedding ROLL in ROCK. The logic language can be used from within the imperative one to query data as well as for method definition. ROCK can embed any expression in the logic language ROLL as long as an appropriate type can be inferred for that expression.

4.1 Embedding ROCK in ROLL

ROLL may invoke any ROCK method as long as the method is side-effect free. A method is defined to be side-effect free if it does not update, directly or indirectly, any non-local data. Any method which may itself update non-local data or which is overridden by a method which may update non-local data is considered to have side-effects. The classification of a method according to this criterion is determined at compile time.

ROCK object names may be referenced, prefixed by an exclamation mark (`!<varname>`), within a ROLL query. In this setting, ROCK object names denote logical constants. Examples of such references are given in section 4.2.

4.2 Embedding ROLL in ROCK

There are two ways in which ROLL can be used from ROCK. One is to express queries, and the other is to define methods. Neither query formulation nor method definition are allowed to break encapsulation.

4.2.1 Querying

A ROLL query expression belongs to the syntactic class of object expressions of ROCK, and can therefore be used in any context in which an object expression is appropriate. Hence, it is possible to pose a query on the result of another query, and to query both persistent and non-persistent data.

Queries invoked from ROCK are written enclosed within square brackets. A query comprises two parts separated by a vertical bar: an optional projection expression, and a query body. The query body is a ROLL goal. There are three different forms of query depending on the nature of the projection:

1. [*ROLL goal*]

   This form of query, with an empty projection, results in a boolean value being returned. The result is `true` if a proof can be found of the ROLL goal, otherwise the result is false. For example, the following query returns `true` if the `segment` object referenced by the ROCK variable `s` starts at a `point` with coordinate values (0,0):

```
[get_point1@!s == Point,
    get_xCoordinate@Point == 0,
    get_yCoordinate@Point == 0]
```

2. [**any** *proj* | *ROLL goal*]

   In this non-deterministic form of query, *proj* returns a single value which is either a single object (primary or secondary) or an aggregation of such objects. If the ROLL goal retrieves multiple values, one is chosen as the result. For example, in the following statement, a `road` of `roadType` `Motorway` is assigned to the ROCK variable `res1`.

```
var res1 : road;
...
res1 := [any Road |
        get_roadType@Road == "Motorway"];
```

   In the following example, the declaration and the assignment are combined in a single statement, in which `res1` is declared as a variable, the type of which is inferred from the logic language expression.

```
var res1 := [any Road |
            get_roadType@Road == "Motorway"];
```

It is possible to project aggregate results as shown in the following example, where **res2** is inferred to have a type which is an aggregate of a **road** and a **roadName**.

```
var res2 := [any <Road,Name> |
             get_roadType@Road == "Motorway",
             get_roadName@Road == Name];
```

3. [{ *proj* } | *ROLL goal*]

   In this case, all the projected objects which satisfy the goal are collected in an association. For example, the following query assigns to **res3** the set of **polygon** objects which have an **area** greater than 100:

```
var res3 := [ {Poly} | get_area@Poly == Area,
                        Area > 100 ];
```

A query must satisfy the condition that all variables which occur in the projection also occur in the goal that constitutes the body of the query.

4.2.2 Method definition using ROLL

In the integrated system, methods can be defined using the logic language as well as the imperative one. Method definitions in ROCK include a complete specification of formal parameters, specifically names and types of input parameters, plus the type of the result. However, this conflicts with the way in which rules are defined in logic languages, where there is no explicit distinction between input and output parameters – a single rule can be called with different binding patterns, thereby allowing greater flexibility. A central aim for the integration of ROCK and ROLL has been to retain the static type checking of ROCK without sacrificing the call-time binding flexibility of ROLL. The following approaches were considered for supporting the definition of methods using ROLL.

1. The most straightforward approach to supporting methods written in ROLL is to use the querying mechanism described in section 4.2.1 in the body of a standard ROCK method. For example, the following code fragment defines a method **owners** attached to **road** which, given an **availability**, returns the set of **owner** objects associated with the **landParcel**s with the given **availability** which are located beside the **road** to which the message has been sent:

```
class road
   ...
   owners(theAvail: availability): {owner}
   begin
      [{Owner} |
         get_road@Loc == self,
         get_landParcel@Loc == Land,
         get_availability@Land == !theAvail,
         get_owner@Land == Owner
      ];
   end
end-class
```

   This approach is directly supported by features described earlier in this paper, and has the advantage that there is no additional syntax associated with the definition of ROLL methods. However, there is no means of calling the ROLL method with different parameters bound – for example, it is not possible to run the method 'backwards' to find the **road** objects given an **owner** and an **availability** because the method signature specifies which values are to be supplied and which returned. Thus a different method has to be defined for each binding pattern, leading to a proliferation of method definitions which are only implicitly identified as being logically equivalent.

2. An alternative (discarded) approach, which continues to use the standard ROCK method definition facilities, but which avoids the duplication of ROLL code in different methods involves the definition of rules separately from the methods which use them. For example, the following code fragment implements the same method as that given above, except that the rule is defined in the context of a flat clause base:

```
owners(Road,Owner,Avail) :-
   get_road@Loc == Road,
   get_landParcel@Loc == Land,
   get_availability@Land == Avail,
   get_owner@Land == Owner.
...
class road
   ...
   owners(theAvail: availability): {owner}
   begin
      [{Owner} |
         owners(self,Owner,!theAvail)];
   end
end-class
```

   The advantage of this approach is that the same rule may be invoked from different methods and with different parameters bound. However, it has a number of disadvantages: there is likely to be a proliferation of ROCK method definitions which are essentially wrappers for ROLL rules; the clause base is not organised using the object-oriented facilities of the model; private attributes and methods are inaccessible to rules which are defined separately from classes; and an additional persistence scheme is required for rules, as they are no longer defined in particular classes.

3. The preferred approach, and that which has been implemented (along with (1) which comes for free), supports the flexible invocation of rules, but introduces a revised syntax for method definition specifically for ROLL rules which does not indicate which parameters are for input and which are for output. For example, the method associated with the same ROLL goal as defined in (1) and (2) can be specified thus:

```
class road
   ...
   owners(owner,availability)
   begin
      owners(Owner,Avail)@Road :-
         get_road@Loc == Road,
         get_landParcel@Loc == Land,
         get_availability@Land == Avail,
         get_owner@Land == Owner.
   end
end-class
```

This method is then always invoked from within the query structure specified in section 4.2.1. As the method definition does not distinguish between input and output arguments, the query form specifies which method parameters will be supplied when the method is invoked and which parameters will be collected as bindings to form the solution to the query. For example, the following statement assigns to `res3` the set of `owner` objects with the `availability` object `theAvail` associated with the `road` object `theRoad`:

```
var res3 := [{Owner} |
            owners(Owner,!theAvail)@!theRoad)];
```

To illustrate an alternative call pattern, the following statement assigns to `res4` the set of `road` objects associated with the `owner` object `theOwner` and the `availability` object `theAvail`:

```
var res4 := [{Road} |
            owners(!theOwner,!theAvail)@Road)];
```

This approach enables the clause base to be organised in an object-oriented manner, facilitates overloading and overriding of ROLL method definitions, permits access to the private properties of a class from ROLL methods, and supports flexible call-time binding.

4.3 Assessing the Integration

In section 2 a number of criteria are presented by which an integration of two languages can be assessed. In ROCK & ROLL, approaches (1) and (3) from the previous section have been implemented. As indicated in Table 1 (in section 2), the integration in ROCK & ROLL satisfies the five criteria associated with the engineering aspects of an integration, namely evaluation strategy compatibility, type system uniformity, type checker capability, syntactic consistency and bidirectionality. Clearly a paradigm mismatch remains, but it can be argued that the two languages are highly complementary, one supporting the declarative, rule-based retrieval of information, and the other supporting a procedural approach to data retrieval, algorithm definition, input/output, and updates. The effectiveness of the combination is being evaluated in the context of a geographic information systems application [AWP93].

5 ROCK & ROLL as a DOOD

This section examines the extent to which ROCK & ROLL can be considered to be an effective deductive object-oriented database. This is done by showing how certain issues raised in [Ull91] are addressed in ROCK & ROLL, and by indicating how the approach taken compares with other proposals for DOODs.

5.1 Overcoming Ullman's Objections

The following issues are raised in [Ull91] as presenting significant obstacles to the development of a DOOD:

5.1.1 Identity of objects obtained by different proofs.

In a deductive relational database (DRDB), an intensional `path` relation can be defined which is the transitive closure of `arc` thus:

```
path(X,Y) :- arc(X,Y)
path(X,Y) :- path(X,Z), arc(Z,Y)
```

Where the `arc` relation is finite, the derived `path` relation will also be finite, even where the `arc` relation contains cycles. This is because the `path` relation is built as a set of values, which leads to different derivations of a path being treated as a single `path` tuple. It is argued by Ullman that where object identity is treated seriously by a DOOD, each result derived for the `path` relation must be given a unique identifier, leading to multiple objects which represent the same path, and thus to the derivation of an infinite number of paths whenever `arc` contains cycles. This, however, is only a problem when the intermediate results of a derivation are represented as objects.

In ROCK & ROLL, path can be represented as a method as follows, where arc is represented as an object attribute:

```
path(Y)@X :- get_arc(Y)@X
path(Y)@X :- path(Z)@X, get_arc(Y)@Z
```

No new objects are created during execution of this method, rather pairs of existing objects which are reachable through the arc attribute are collected within the rule evaluator. After rule execution has completed a single collection object is created to hold all the solutions to the query. For example, the following statement assigns to res a new object which is an association of the objects which are reachable from the ROCK variable x according to the definition of path:

```
res := [{Y} | path(Y)@!x]
```

5.1.2 Dynamic Typing and Ad-Hoc Queries.

In a DRDB, the definition of an intensional relation has the effect of adding a new 'type' to the schema. For example, the rule defined above introduces a new (intensional) relation path. The new path relation can subsequently be used directly by other definitions because the standard relational operators are directly applicable to it. Ullman argues that the increased complexity of type systems in object-oriented databases, and the possibility of understanding an object in terms of its behaviour rather than its structure, makes it impractical to allow the incremental addition of rules which effectively produce new types as a result.

In the context of ROCK & ROLL, two points can be raised to counter Ullman's case. The first is that the definition of a new ROLL method does not add any new type to the system. For example, the above definition of the ROLL method path expands the signature of the type to which it is attached, but the number of types remains the same. The second is that the way in which such a method is used determines which new types are required, and that alternative approaches are available. For example, if the ROLL method is only used directly from other ROLL methods then the intermediate result of the method does not need a new type to be defined, as discussed in point (1) above. If the ROLL method is called from ROCK, then it is indeed necessary to assign the result of the method to a typed variable. If the typed variable is local to the computation, then its type can be inferred directly within the context where it is used, and there is no need for a global type. For example, the following program declares the variable res, assigns to it the result of an invocation of the path method, and prints the value of the pos attribute of each object in res:

```
var res := [{Y} | path(Y)@!x];
foreach o in res do write get_pos@o, nl;
```

By contrast, if the result of the derivation is to persist, and must be associated with some user-defined behaviour, then it is indeed necessary to define a new type with properties, an interface, etc. The extension of this type can then be constructed from the results of ROLL methods.

This flexible response to Ullman's case means that there is little or no overhead associated with the definition of types for temporary values, and that the programmer has the option of defining comprehensive types and classes for structuring the results of a ROLL method if this is what is required.

5.1.3 Optimising declarative queries and rules.

Queries and rules in DRDBs can be optimised using a range of techniques, effective optimisation being important to the practicality of a declarative language. Ullman suggests that optimisation in DRDBs is practical because of the generic and well understood semantics of the algebraic operations which are applicable to relations, and that the presence of user-defined operations (methods) in OODBs precludes effective optimisation.

In ROCK & ROLL, methods written using ROLL are logic programs on a par with Datalog. Indeed, the semantics of ROLL is defined by a mapping onto Datalog in the context of the set of first-order axioms which describe the data modelling constructs of OM [FWP93]. This means that ROLL programs can be optimised using variations on the techniques proposed for use with Datalog. The optimisation of ROLL using a technique based upon static filtering [KL90] is described in [BFP+94]. In this approach, ROLL queries and methods are represented using a graph structure through which constraints can be pushed in a bid to minimise the size of intermediate data sets. This essentially overcomes Ullman's objection by allowing the optimiser to look inside, and indeed to optimise, methods which are implemented using ROLL. If a ROLL method invokes a ROCK method, then the ROCK method is treated as a black box which, given a message recipient and an appropriate number of parameters, returns a result.

5.2 Comparison With Other DOOD Proposals

Proposals have been made for DOODs with widely different starting points and emphases [FPWB92], only a few of which are mentioned here because of limited space.

The most fundamental departure from earlier work on deductive databases is characterised by F-logic [KL89], where a new logic is proposed incorporating such object-oriented features as inheritance and a notion of identity. However, it is far from clear how such a logic can be implemented efficiently, and the semantics provided for the language make no attempt to allow for state change, as required in practical database systems.

An alternative approach builds more directly upon earlier work on DRDBs by extending the semantics of Datalog with constructs such as identity and inheritance [Abi90, CCCR+90]. Such proposals benefit from the extensive research on the design and implementation of DRDBs, but generally have limited semantic data modelling facilities, and logic language semantics complicated by the introduction of updates. Similar strengths and weaknesses can be obtained by an alternative approach in which a DOOD language is implemented by mapping it onto an existing deductive database system [BM92].

The above proposals significantly play down the role of an independently defined data model as the basis from which languages can be defined with different operational semantics. Other such proposals are LLO [LO91] and ORLOG [JL93]. Both are inspired by F-logic into postulating a higher-order syntax, which provides undeniably elegant solutions in certain cases, but is difficult for occasional users to fully grasp. The higher-order approach also incurs the penalty of a less than intuitive declarative semantics. In the case of LLO this implies few clues as to how an operational semantics can be implemented as the basis for query evaluation, a problem that ORLOG bypasses by rewriting into LDL [NT89].

ConceptBase [JJR90], a powerful KBMS built around the Telos language, also advocates deductive and object-oriented layers in a single system architecture. In particular, the variant of Telos used in ConceptBase has its expressiveness restrained to that of Datalog like ROLL. However, in ConceptBase methods can only be written using deductive rules, and therefore the range of applications that can be implemented using only ConceptBase is much more restricted than one can implement in ROCK & ROLL.

6 Conclusions

This paper has described the integration of a logic query language with an imperative programming language in the context of a semantically expressive object-oriented data model. It has been shown how the resulting system satisfies a number of criteria by which interfaces between languages can be judged, and how the integrated system overcomes certain challenges posed in [Ull91] to the developers of DOODs.

The system which results from the integration retains the strengths of its components and adds to these strengths the following benefits which derive from their co-existence:

- Data described using the data model can be processed or analysed using two different and complementary programming styles.
- The facility to manipulate data using an imperative programming language has been obtained without sacrificing the effectiveness of the query language.
- The semantics of the logic language has not been complicated by the introduction of update statements, and yet the database can be effectively manipulated using fully integrated facilities.
- The impedance mismatch, relating to both type system and evaluation strategy, has been overcome.

6.0.1 Current Position

The ROCK & ROLL system is being implemented using EXODUS, and in particular the persistent C++ system E [CDG+90]. A functionally complete implementation of ROCK & ROLL has been completed, and work is underway on the implementation of an optimiser for ROLL queries and rules.

6.0.2 Acknowledgements

The work that resulted in this paper has been funded by the UK Science and Engineering Research Council through the IEATP programme, and their support is duly acknowledged. The object manager was implemented by Alia Abdelmoty. We would also like to thank 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 as the industrial partners in the project.

References

[Abi90] S. Abiteboul. Towards a Deductive Object-Oriented Database Language. *Data & Knowledge Engineering*, 5:263–287, 1990.

[AWP93] A.I. Abdelmoty, M.H. Williams, and N. W. Paton. Deduction and Deductive Databases for Geographic Data Handling. In David Abel and Beng C. Ooi, editors, *Advances in Spatial Databases: Third International Symposium, SSD '93*, LNCS 692, pages 443–464. Springer-Verlag, 1993.

[BFP+94] M.L. Barja, A.A.A. Fernandes, N.W. Paton, M.H. Williams, A. Dinn, and A.I. Abdelmoty. Design and Implementation of ROCK & ROLL: A Deductive Object-Oriented Database System. In *submitted for publication*, 1994.

[BM92] E. Bertino and D. Montesi. Towards a Logical-Object Oriented Programming Language for Databases. In Alain Pirotte, Claude Delobel, and Georg Gottlob, editors, *Advances in Database Technology - EDBT'92*, LNCS 580, pages 168–183. Springer-Verlag, 1992.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-based Programming Paradigm. In *Proc. ACM SIGMOD Conf.*, pages 225–236, 1990.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, CA 94303-9953, 1990. Morgan Kaufman Publishers, Inc.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[Dat90] C.J. Date. *Introduction to Database Systems, Volume 1 (5th Edition).* Addison-Wesley, 1990.

[DKM91] C. Delobel, M. Kifer, and Y. Masunaga. *Deductive and Object-Oriented Databases (Second International Conference DOOD'91, Munich).* Springer-Verlag, Berlin, 1991.

[FBPW93] A.A.A. Fernandes, M.L. Barja, N.W. Paton, and M.H. Williams. A Deductive Object-Oriented Database for Data Intensive Application Development. In *Proc. 11th British National Conference on Databases*, LNCS 696, pages 176–198. Springer-Verlag, 1993.

[FPWB92] A. A. A. Fernandes, N. W. Paton, M. H. Williams, and A. Bowles. Approaches to Deductive Object-Oriented Databases. *Information and Software Technology*, 34(12):787–803, December 1992.

[FWP93] A.A.A. Fernandes, M.H. Williams, and N.W. Paton. An Axiomatic Approach to Deductive Object-Oriented Databases. Technical Report TR93002, Department of Computing and Electrical Engineering, Heriot-Watt University, April 1993.

[FWP94] A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A Logical Query Language for an Object-Oriented Data Model. In N.W. Paton and M.H. Williams, editors, *Proceedings of First International Workshop on Rules in Database Systems*, pages 234–250. Springer-Verlag, 1994.

[JJR90] M. Jarke, M. Jeusfeld, and T. Rose. Software Process Modelling as a Strategy for KBMS Implementation. In *[KNN90]*, pages 531–550. 1990.

[JL93] M. Hasan Jamil and Laks V.S. Lakshmanan. Realizing Orlog in $\mathcal{LDL}$. In *[Mum93]*, pages 45–59, 1993.

[KL89] M. Kifer and G. Lausen. F-logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proc. ACM SIGMOD Conf.*, pages 134–146, 1989.

[KL90] M. Kifer and E. Lozinskii. On Compile-Time Query Optimization In Deductive Databases By Means Of Static Filtering. *TODS*, 15(3):385–426, 1990.

[KNN90] W. Kim, J-M Nicolas, and S. Nishio, editors. *Deductive and Object-Oriented Databases (First International Conference DOOD'89, Kyoto).* North-Holland, 1990.

[KR91] M.H. Kay and P.J. Rivett. An Overview of the Raleigh Object-Oriented Database System. *ICL Technical Journal*, 7:780–798, 1991.

[LO91] Y. Lou and Z.M. Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD Conf.*, pages 198–207, 1991.

[Men87] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole, 3rd edition, 1987.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Mum93] I.S. Mumick, editor. *Proc. Workshop on Combining Declarative and Object-Oriented Databases*, Washington, DC, May 1993.

[NT89] S.A. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, 1989.

[PDR91] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A Deductive Database System. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD Conf.*, pages 308–317, 1991.

[PS91] A. Poulovassilis and C. Small. A Functional Programming Approach to Deductive Databases. In G.M. Lohman, A. Sernadis, and R. Camps, editors, *Proc. 17th VLDB*, pages 491–500. Morgan Kaufmann, 1991.

[SRSS93] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding Object-Orientation to a Logic Database Language. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. 19th VLDB*, pages 158–170. Morgan Kaufmann, 1993.

[Ull91] Jeffrey D. Ullman. A Comparison Between Deductive and Object-Oriented Database Systems. In *[DKM91]*, pages 263–277. 1991.