A Deductive Object-Oriented Database For Data Intensive Application Development [1]

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

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

Abstract. This paper outlines an approach to the development of a deductive object-oriented database system. The approach presented uses a formally defined object-oriented data model as the starting point for the development of a logic query language and an imperative database programming language. These languages can be used independently – the logic language for expressing queries or defining rule-based applications, the imperative language for manipulating the database – or together to support a flexible system for data-intensive application development.

1 Introduction

This paper presents the architecture of a deductive object-oriented database system (DOOD) currently under development at Heriot-Watt University, and indicates how such an architecture supports the development of large-scale data-intensive applications.

The software development life-cycle traditionally involves a sequence of steps which take the development team through problem analysis, high-level design, detailed design, and implementation. Regardless of the tools and techniques which are used within a particular organisation to support analysis and design, it is likely that implementation using conventional database systems will not follow naturally from the documents which emerge from the design process. The database system often fails to integrate smoothly with the design process in two major areas. Firstly, the structural features of the application, which are described during design using a semantic data model such as the E/R model, have to be mapped into the less expressive structure representation facilities of the data model. Secondly, the behavioural features of the application, which are described using a process modelling system such as data flow diagrams, have to be mapped onto the constructs of a conventional programming language which embeds calls to the database system.

Database researchers have adopted a number of approaches which seek to ease the development of applications which make use of database systems. For example, database programming languages [7] remove the dichotomy between the constructs of a database system and the language used to program the application, and object-oriented databases [18] provide database support for certain modelling constructs which are reminiscent of those supported by semantic data models. However, database programming languages such as PS-algol [8] and its descendants lack high-level data modelling facilities. Furthermore, most commercial object-oriented database systems are based upon object-oriented programming languages such as C++, and thus support low-level data description mechanisms instead of the more abstract models associated with database design tools. Thus while such systems ease the implementation phase of certain data-intensive applications, they do not significantly reduce the dichotomy between database design and implementation tools. A broader review of recent proposals for database systems, such as that given in section 2, indicates that it is normal to find at least one area of compromise concerning the accessibility of the data, the richness of the data model, or the expressive power of the data manipulation language.

This paper describes an approach to the development of a deductive object-oriented database system which includes an object-oriented data model which is appropriate for use as a semantic data model. This data model is fully integrated with both an imperative database programming language and a logic query language, and thus helps to bridge the gap between the design and implementation processes. The object management facilities which are supported by the data model of [15] and the manipulation language of [9] are collectively known as ROCK (Rule

[1] In M. Worboys and A.F. Grund(eds.) *Advances in Databases: Proceedings of the 11th British National Conference on Databases (BNCOD 11)* Keele, UK. Springer-Verlag LNCS 696, pages 176-198, July 1993. ISBN 0-387-56921-9.

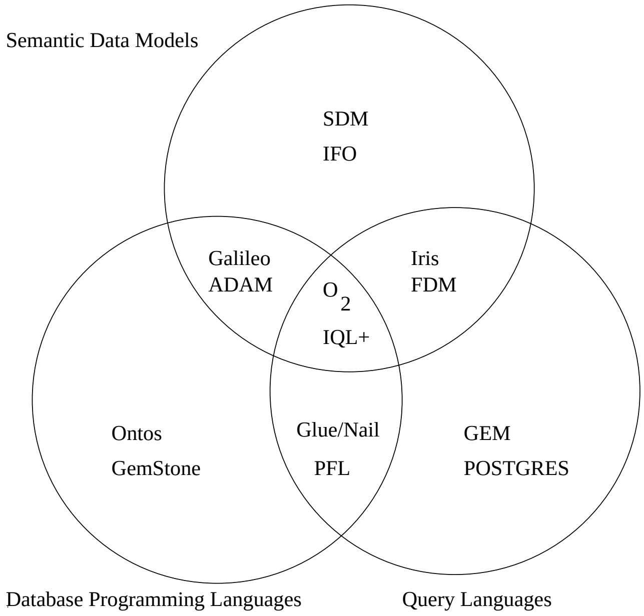

Figure 1: A Classification of Recent Database Systems

Object Computational Kernel). The logic language, which can either be used interactively or in an embedded form, is known as ROLL (Rule Object Logic Language).

This paper outlines an architecture in which a semantically rich data model can be fully integrated with a computationally complete programming language and a logical query language. The strategy is outlined in section 3, and an overview of the associated components is presented in section 4. Section 5 describes the consequences of the architecture for the development of large-scale, data-intensive applications, and section 6 presents some general conclusions. In section 4 acquaintance with object-oriented and deductive notions is assumed at the level of [18] and [11], respectively.

2 Related Work

This section compares a number of alternative approaches to the development of database systems which are intended to support application development, and in particular those approaches which utilise some form of object-oriented data model. One of the major goals of the DOOD project at Heriot-Watt is to integrate within a uniform formalism a range of facilities which have traditionally been supported by distinct systems. These distinct systems – semantic data models, database programming languages, and query languages – have been the focus of considerable attention by database researchers in recent years, as has their integration. However, an examination of the literature shows that many of the most prominent proposals for third generation database systems provide solutions which fail to cleanly integrate the three categories of system mentioned above. Figure 1 presents a classification of a number of systems according to the three categories introduced above, and identifies key areas of overlap which have been addressed by particular proposals. Such a classification process is clearly inexact, but should serve to identify different areas of emphasis among various research groups. The classification has been restricted to two systems in each category for reasons of limited space. The different assignments of systems to areas in the diagram are discussed in what follows.

Query Language Emphasis: Such systems, e.g. POSTGRES [29], GEM [31], are likely to operate on a data model which is either not independently defined, or which is lacking in certain modelling facilities commonly supported by semantic data models. The implementation of data-intensive applications is likely to require the embedding of the query language in a conventional programming language, thereby introducing an impedance mismatch. POSTGRES is considered to belong to this category because its principal advocate adheres to the manifesto suggestion that all access to the database is obtained through the query language [29], and because it is hard to imagine using the POSTGRES data model as a stand-alone modelling formalism.

Semantic Data Model Emphasis: Such systems, e.g. IFO [4], SDM [19], major on the description of the static properties of an application, and as such are amenable to use for conceptual modelling, independent of a particular implementation platform. There may be few, if any, behaviour description mechanisms.

Programming Language Emphasis: Such systems commonly start with an existing programming language, and minimise modifications to the language by making persistence orthogonal to the type of data being processed. Facilities for structuring data are thus directly descended from programming language mechanisms, and are often at a lower level than those of semantic data models. Typical examples are PS-Algol [8], Ontos [6] and GemStone [24], which are descended from S-Algol, C++ and Smalltalk respectively.

Programming Language plus Query Language: Such systems can result either from embedding a query formalism within a programming language (e.g. deductive query statements within the functional language PFL [26]), or by merging a data manipulation language with an existing query language (e.g. Glue/Nail [25]).

Programming Language plus Semantic Data Model: Where the type system of a database programming language includes data modelling constructs akin to those of a semantic data model, rather than lower level structuring mechanisms, systems in this category result. Galileo [5] is included in this category, while GemStone and Ontos are not, because certain abstractions are explicit in the Galileo type system which have to be implemented by programmers in GemStone or Ontos. ADAM [18] is included because it supports a computationally complete programming language and has a data model which has been extended to support semantic modelling facilities.

Semantic Data Model plus Query Language: The addition of a query language to a semantic data model permits the structures of the model to be directly accessed and manipulated. However, such languages are not normally computationally complete, and thus have to be embedded in some imperative programming language for application development. Examples in this category include the query languages DAPLEX [28] and OSQL [10], which have been proposed for use with the Functional Data Model and Iris respectively.

Fully Combined Approach: Few systems smoothly integrate all three categories mentioned above. The O_2 [12] database system supports an independently specified data model, a number of programming language interfaces, and a query language. The approach described in this paper aims to yield a more expressive data model, a more powerful deductive query language, and a smoother integration of the three components. The system referred to as IQL+ [3] is an alternative approach to providing a deductive object-oriented database, by successively extending Datalog with concepts such as identity and inheritance for structuring the database, and explicit control to support updates. The principal limitations of IQL+ are the limited power of its data model, and the rather basic database programming facilities. The approach outlined in this paper does not introduce procedural mechanisms into the logic as in IQL+, but rather uses distinct languages for expressing deductive rules and for manipulating data, following in the footsteps of Glue/Nail [25]. This should result in much more standard and powerful data manipulation facilities, without complicating the semantics of the logic component by introducing extra-logical features.

The IQL+ approach to extending Datalog with object-oriented facilities is typical of a number of approaches summarised by [17]. An alternative approach adopted by [20] is to develop new logics, although current proposals in this area are a long way from yielding implemented systems. A summary of approaches to the development of DOODs is given in [13].

3 Architecture

The architecture we propose is built upon the data model defined in [15]. Conceptually, the model is defined as a formalism which specifies a mathematical structure corresponding to the database schema and which, for reasoning with such structures, provides:

- *Modelling axioms* which formalise the semantics of the abstraction mechanisms used in the model. When combined with the structures, these yield a logic theory of objects.
- *Operation axioms* which formalise the semantics of the operations which act on structures defined by the model. When taken with the structures, these yield an algebraic theory of objects.

Differentiating between the logical and algebraic theories gives rise to two design strands: the first combines the logic theory of objects with a proof procedure to yield an object-oriented logic language; the second associates the syntax and type system of a data manipulation language with the algebraic theory to yield an object-oriented database programming-language.

In this context, a deductive object-oriented database system can be envisaged which allows a user to interact with an object database through the two languages mentioned above. Crucially, in this approach the same set of objects determining a database must be taken as the Herbrand universe in the logical context, and as the carriers of the algebraic theory, as shown in Figure 2.

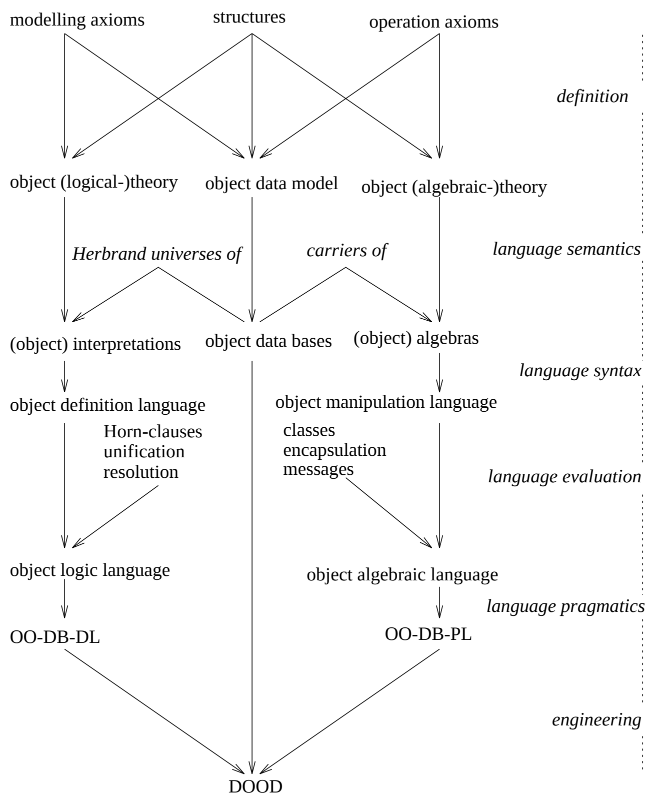

Figure 2: An Approach to DOODs

Some important shortcomings of DOOD proposals that have appeared so far can be explained by the fact that their sets of objects cannot be taken in both the above ways. Further, we have argued elsewhere [13] that an independently-defined data model is instrumental in achieving this.

The motivation for postulating the design of two languages is to keep the deductive language as pure as possible, leaving state-transition and control as the province of the imperative language. This simplifies the formal treatment of properties of logically-expressed queries, integrity constraints and derived data, while not necessarily imposing any mismatch in the embedding of the deductive language in the imperative one (as demonstrated by [25]). The logic language can be used in an interactive mode for expressing queries, for defining methods stored in the database, or as individual statements embedded within the imperative programming language. The resulting system can thus be placed in the intersection of all three categories in Figure 1, without the need to compromise on any of the three aspects.

4 Model and Languages

The previous section presented a strategy for the integration of an object-oriented data model with a logic query language and an imperative manipulation language without describing any of the individual components. This section introduces the data model and its associated languages.

4.1 The Data Model

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. Both atomic values and whatever abstractions are used to model a particular application domain are considered as *objects*. Atomic values are called *primary objects*, and all other abstractions are called *secondary objects*. Each object is assigned to an *object type*, and thereby conforms to the structure associated with the object type. Every object has a unique, immutable *identifier* the existence of which is completely detached from its *state*.

Structures comprise references of two kinds. The first kind of reference is used to model the *properties* of the object. This results, for each object, in a possibly empty set of pairs of the form ⟨*object-type, object-identifier*⟩. The second kind of reference is used when the object is complex, to model its *construction* from other objects. The result of this, for each such object, is a structure that takes one of three forms: a set of object identifiers, or a set of pairs of the form ⟨*index, object identifier*⟩, or a set of pairs of the form ⟨*object-type, object-identifier*⟩, which are the representations of abstraction mechanisms referred to as *association* (or set), *sequentiation* (or list) and *aggregation* (or tuple), respectively.

The *behaviour* of an object is determined by its response to *requests* for *operations* to be performed. Such operations are defined on an object class. From the point of view of users, an operation is viewed in terms of its *interface*, i.e. the *operation name* to be invoked in a request and the *operation signature* which dictates the types of the arguments and result which a request must carry. Designers are concerned, in addition, with *operation implementations*, generally called *methods*. Operations *encapsulate* the state of an object.

Invariants in the model are of two different kinds: domain-independent and domain-dependent. The first kind is established by means of *modelling axioms* and *operation axioms* which are embodied in any management system implementing the model. The second kind is handled within the rule-based language introduced in section 4.2. To give a flavour of the model, an example schema is given in Figure 3. The diagramming notation is as follows:

Nodes A **horizontal oval** represents an object type whose instances are primary objects. A **full-line rectangle** represents an object type whose instances are secondary objects. A **round-cornered rectangle** represents an operation. A **vertical oval** represents the source of the identifiers in the grounding of an object type.

Directed Edges An **unlabelled directed edge** represents the grounding of an object type by pointing to a primitive type, to the set of proper oids or to another object type. Note that some groundings have not been indicated (e.g. all those into integer) to keep the diagram less cluttered.

A **labelled directed edge** has one of seven symbols as its label: a directed edge labelled with a $\bigcirc$-*symbol* indicates an *attribution*; a directed edge labelled with a $\boxdot$-*symbol* indicates a *generalisation*; a directed edge labelled with a $\diamond$-*symbol* indicates a *partition*; a directed edge labelled with a $\triangle$-*symbol* indicates a *specialisation*; a directed edge labelled with a $\otimes$-*symbol* indicates an *aggregation*; a directed edge labelled with a $\circledast$-*symbol* indicates an *association*; a directed edge labelled with a $\circledcirc$-*symbol* indicates a *sequentiation*;

The last three have a *cohesive* version in which the characteristic symbol is enclosed within a square, instead of a circle, as described in 4.3.1.

Constraint Arrows in Constructive Edges A **double-headed arrow** represents the fact that multiple references to the same object of the type that is pointed to are admissible in the constructed objects. A **single-headed arrow** represents the fact that an object cannot be referenced more than once.

A **dashed-line arrow** represents the fact that some object of the type that is pointed to may not be referenced by any of the constructed objects. A **full-line arrow** represents the fact that every object of the type pointed to must be referenced.

4.2 The Logic Language

This section describes the logic language (ROLL) which can be used to query databases described by the model presented in the previous subsection. The syntax presented here bears some resemblance to that of the Stony Brook object logics [20], but is likely to be revised in the light of experience, and to ease its integration with the programming language introduced in the next section.

A ROLL **term** is either a variable, a constant or a keyword. There are no function symbols, so 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 is called the *Herbrand universe*, denoted by H_U. Since both $Cons$ and $Kwords$ are finite, it follows that so is H_U.

The general form of a ROLL **atom** is $\ulcorner\alpha\texttt{[}\beta\texttt{(}\gamma\texttt{)} \texttt{ => } \delta\texttt{]}\urcorner$, where the Greek letters are place holders as follows: α is a ROLL term which represents an object; β is a ROLL term which represents the name of an operation defined on the class of α; γ is a sequence of terms which represent the arguments of β; δ is a term which represents the result of applying the operation β with arguments γ to the object α. If β requires no arguments or if it returns no results then $\ulcorner\texttt{(}\gamma\texttt{)}\urcorner$ can dropped or $\ulcorner \texttt{ => } \delta\urcorner$ can be dropped or both. An atom is *ground* iff α and β are ground, and γ and δ are either ground or missing.

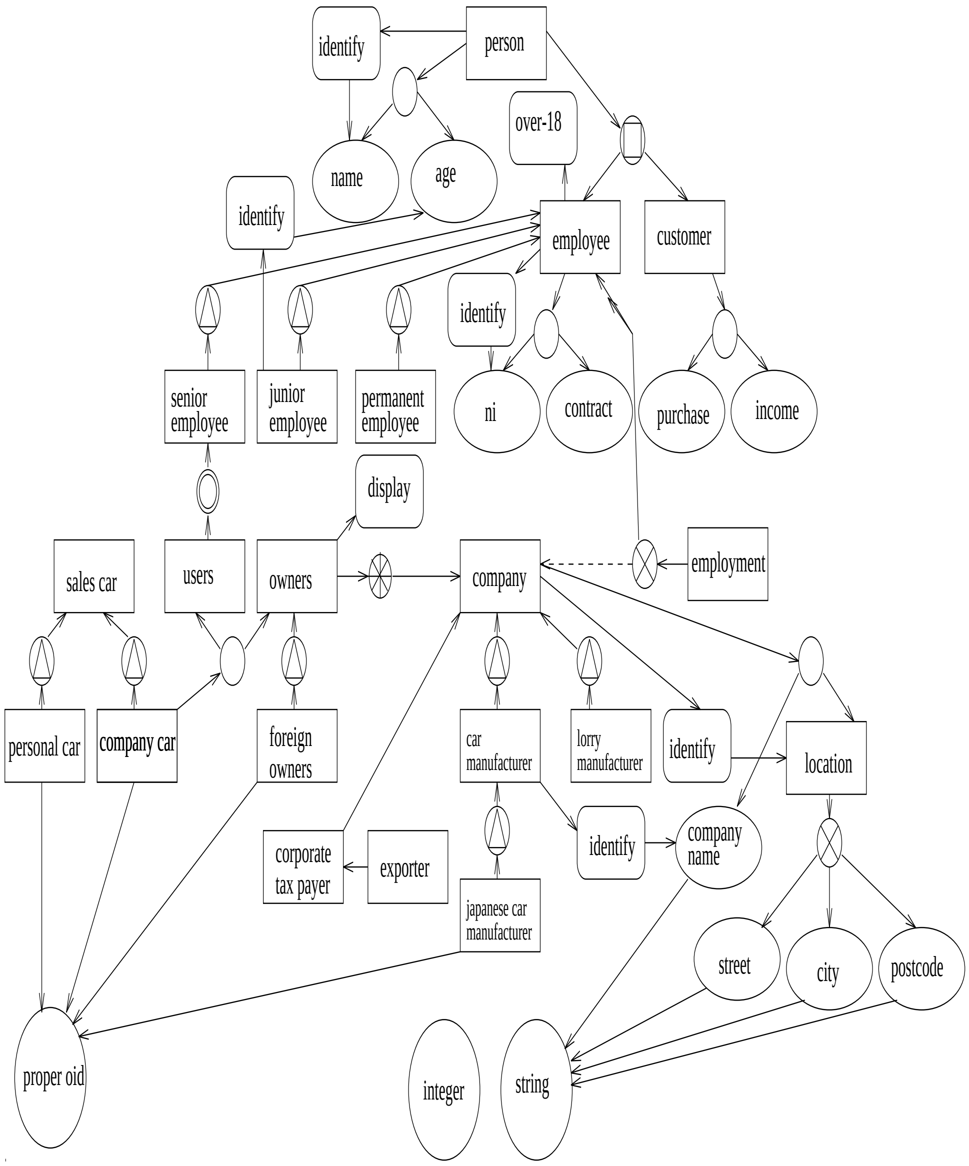

Figure 3: Company schema diagram

A ROLL **literal** is an atom $\ulcorner \alpha$`[`$\beta(\gamma)$ `=>` δ`]`$\urcorner$ or a negated atom $\ulcorner$ `~(`α`[`$\beta(\gamma)$ `=>` δ`])`$\urcorner$. A literal is *ground* iff it is a ground atom or a negated ground atom. A literal is *positive* iff it is an atom, a literal is *negative* iff it is a negated atom.

A clause is a finite set of literals $\{L_1, \ldots, L_n\}$, which is interpreted as $L_1 \vee \ldots \vee L_n$. As usual in logic programming, there is a direct correspondence between clauses and first-order predicate logic formulae. A clause is *ground* iff all its literals are ground. A clause is *positive* iff all its literals are positive. A clause is *negative* iff all its literals are negative. A clause which contains a single literal is called a *unit clause*. A clause which contains at most one positive literal is called a *Horn clause*. A ROLL **clause** is a Horn clause.

Facts, rules and goals are forms of Horn clauses. A ROLL **fact** is a positive unit ROLL clause. A ROLL **rule** is a ROLL clause with exactly one positive literal and at least one negative literal. A ROLL **goal** is a negative ROLL clause.

There are also a number of operators in ROLL which are written as infix operators. For instance, the value-equality operation is '`==`'. Thus '`!1 == !2`' is true iff the object whose identifier is !1 is value-equal to the object whose identifier is !2.

ROLL clauses and variables are interpreted logically. More specifically, in the context that concerns us here, ROLL goals are queries to be evaluated against the intensional and extensional databases using resolution-based automated theorem-proving. In the demonstration system that we have implemented, an operational semantics of ROLL programs (i.e. of sets of ROLL clauses) is obtained by mapping them onto an internal representation which is used as input to an SLDNF-resolution proof procedure, extended to cope with late binding, as described in [14].

A number of ROLL queries are now given to illustrate the language in use. All queries range over the example '`company`' domain. The text given is taken from a prototype interactive implementation of the logic language. In the examples, object identifiers of the form ! < *integer* > are typed directly by the user, although such an activity is only permitted when using the language in an interactive mode – object identifiers cannot appear explicitly within compiled programs.

4.2.1 Schema-Level and Instance-Level Queries

The following are example schema-level ROLL queries:

```
roll ?- employee[isa => person].
yes
roll ?- city[ipo => location].
yes
roll ?- company[ici => X].
[X = owners] ;
[X = foreignOwners] ;
no
roll ?- seniorEmployee[ici => X].
[X = users] ;
no
roll ?- X[has => name].
[X = customer] ;
...
no
```

The first query returns true iff the object-type whose name is '`employee`' is a subtype of the object-type whose name is '`person`'. Intuitively, the object-type name '`person`' is the result of sending the message '`isa`' to the object-type name '`employee`'.

The second query is analogous, but sends the message corresponding to the *is-part-of* relationship rather than *is-a* as in the first. The third and fourth use the messages corresponding to the *is-collected-in* relationship, which is applicable to association and sequentiation abstractions, to find object-types which are collections of companies (in the case of the third) and of senior employees (in the case of the fourth). Finally, the fifth query uses the message corresponding to the *has-a* relationship to retrieve all object-types which have a '`name`' property.

The following are example instance level queries:

```
roll ?- !4[iof => employee].
yes
roll ?- Y[ith(city) => "edinburgh"].
[Y = !2] ;
[Y = !14] ;
```

```
no
roll ?- X[iin => !16].
[X = !18] ;
[X = !21] ;
no
roll ?- !4[.age => Y], Y :>= 18.
[Y = 25] ;
no
roll ?- !4[.name => Y], !13[.name => Z], Y :@> Z.
[Y = mary, Z = john] ;
no
roll ?- all !4[.X => Y].
[X = age, Y = 25]
[X = contract, Y = permanent]
[X = name, Y = mary]
[X = ni, Y = 12345]
yes
```

The first query returns true iff the object whose name is '`!4`' is an instance of the object-type whose name is '`employee`'. Intuitively, this is done by sending the message '`iof`' to the instance '`!4`'.

The second and third queries uses two other message names. The second query uses the *is-the* relationship between an object, the numeral or object-type-name that indexes a reference to an object in its construction (by sequentiation or aggregation) and the referenced object. Thus, the second query returns as '`Y`' the names of aggregate objects which have the object named '`"edinburgh"`' in their '`city`' component. The message in the third query corresponds to the *is-in* relationship between an object and the association to which it belongs. Thus, the third query returns the names of the objects which are referenced in the construction by association of the object whose name is '`!16`'.

The three remaining queries use the message '`.<propertyname>`' (which is simply a shorthand for '`get(<propertyname>)`') to gain access to objects referenced as properties of the addressee. The fourth query demonstrates the use of an arithmetical comparison operator '`:>=`', and the fifth that of a term comparison operator'`:@>`'. Finally, the sixth returns as property('`X`')-value('`Y`') pairs all property references of the object with name '`!4`'.

```
roll ?- !2[..street => Y].
[Y = grassmarket] ;
no
roll ?- !7[.. 1 => X].
[X = !4] ;
no
roll ?- all !2[..X => Y].
[X = street, Y = grassmarket]
[X = city, Y = edinburgh]
[X = postcode, Y = eh12hj]
yes
```

The three queries above illustrate how the elements in the construction of objects built by sequentiation and aggregation can be manipulated in ROLL. Analogously to '`get`' and '`.`', '`..`' is synonymous with '`ith`'. The first query returns the '`street`' coordinate of the object whose name is '`!2`' as the instantiation of '`Y`'. The second query returns the first element in the construction by sequentiation of the object whose name is '`!7`'. Finally, the third query returns as coordinate('`X`')-value('`Y`') pairs all construction references of the object (in this case built by aggregation) with name '`!2`'.

4.2.2 Overloading, Overriding and Late-Binding Methods

It is difficult to claim that a formalism falls under the object-oriented paradigm if the phenomena known as overloading, overriding and late-binding of methods are not accounted for in a clearly recognisable way. At present, this is a point where most ongoing research in the area of deductive object-oriented databases runs into problems. The following three groups of queries illustrate how these features are present in ROLL, and how the account given to them is intuitive from both the deductive and the object-oriented perspectives. We begin with overloading.

```
roll ?- !1[iof => X], !1[identify => Y], Y[iof => Z].
[X = company, Y = scotcars, Z = companyName] .
yes
roll ?- !4[iof => X], !4[identify => Y], Y[iof => Z].
[X = seniorEmployee, Y = 12345, Z = ni] .
yes
```

In the examples above, '`!1`' is an instance of '`company`', and '`!4`' is an instance of '`seniorEmployee`'. The crucial point to notice is that the operation '`identify`' associated with '`company`' objects is different from that associated with '`seniorEmployee`' objects, and thus the name '`identify`' is overloaded – its implementation in class '`company`' is different from that in class '`seniorEmployee`'.

At this point it is important to have a closer look at ROLL methods. As already pointed out, methods are an integral part of a domain definition, in that they are used in building the intensional database. The ROLL program that implements a method must be assigned to a pair $\langle \mu, t \rangle$, where μ is a method name and t is an object type name, so that overloading, overriding and late-binding can be catered for.

The available methods in a particular class are determined by the computed operation set of the class, which supports inheritance as described in [15]. For instance, in the example domain (see Figure 3) the operation name '`identify`' appears in five $\langle \mu, t \rangle$ pairs, viz. ⟨ *identify, company* ⟩, ⟨ *identify, carManufacturer* ⟩, ⟨ *identify, person* ⟩, ⟨ *identify, employee* ⟩, ⟨ *identify, juniorEmployee* ⟩. To each of these pairs there corresponds a method implementation, two of which are given below:

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

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

The first of these can be read as: the method '`identify`' attached to the class '`person`' has as result '`X`' as defined by the clause '`person[identify => Y] <- X[.name => Y]`'. Thus the ROLL program which gives a definition to the method is enclosed in curly brackets. Some features present in the above ROLL programs are explained later in this document.

In order to cater for late-binding, the proof procedure must be extended to select that implementation of *identify* that best suits the addressee. The precise definition of what it is that best suits an addressee can be found in [15], and roughly amounts to selecting the implementation associated with the most specific class of the addressee. The absence of conflicting implementations at a given class is a well-definedness property of domain-definitions [15].

Recall that *identify* is defined for *juniorEmployee, employee* and *person*. These three classes are ordered from the most to the least specific as follows: $juniorEmployee \sqsubseteq employee \sqsubseteq person$ [15]. The following two queries illustrate how overriding takes place with respect to instances of *employee* and *person*.

```
roll ?- !3[identify => X].
[X = john] .
yes
roll ?- !4[identify => X].
[X = 12345] .
yes
```

In the above cases the (rather contrived) *identify* operation name has been late-bound to its implementation in the most specific class of the addressee for which the operation is available, viz. *person* for '`!3`', and *employee* for '`!4`'. The prototype implementation does not cater for static type checking, and hence the differing result types of *identify* in the examples above are permitted.

4.3 The Programming Language

The logic language described in the previous section can be considered to be the query language of the database system. Its role differs from that of SQL in relational databases in two important senses – it has no statements which support data manipulation, and it enables rule-based applications to be built within the database system itself.

The role of the DBPL in the DOOD architecture is to support data definition and manipulation at both the schema and the data levels, and to enable complete applications to be implemented without the hindrance of the impedance mismatch. This can be achieved by developing a programming language which shares the same type system as the logic language – in our case derived from the data model described in section 4.1, in accordance with the architecture given in Figure 2. This section gives a flavour of the programming language through a number of examples of data definition and manipulation statements.

The language is defined as a strongly typed imperative language with the intention of adopting static type-checking whenever possible. To relieve the user of the burden of having to specify the types of all variables and expressions, a type inferencing mechanism is used.

4.3.1 Object type declarations

Each object in the system can be categorised as belonging to either the set of primary or the set of secondary objects. Primary objects are instances of primitive types and secondary objects are instances of classes each of which has a structure that conforms to that given by an *object type* definition. The primitive types supported by the language are *int, real, string* and *bool.* Each of these primitive types is associated with a number of standard operators for manipulating objects of that type.

The programmer can define new types by renaming existing types or by aggregating a number of primitive and/or user defined types to build an object type declaration. An object type declaration consists of a name (the name of the object type being defined) and an object type definition.

An object type definition may consist of up to six components, namely a *grounding expression*, an *inheritance expression*, an *attributive expression*, a *constructive expression*, a *construction constraints expression*, and an *interface expression*.

The **grounding expression** determines whether the objects conforming to this type are primary objects or secondary ones. A type definition can have at most one grounding expression – if no grounding expression is present, the objects conforming to the type are secondary objects. If the objects in the grounding set are defined to be primary objects then the definition of the type includes only a grounding expression (the grounding set of an object-type contains the set of objects that conform to that object-type). Alternatively, the grounding set of the object type may be the same as that of another object type (objects may have the same structure, but the operations which determine their behaviour may be different). As an example of a grounding expression, *type name:string* indicates that *name* is grounded on *string.*

The **inheritance expression** allows for the specification of relationships between the grounding set of the object type being defined and that of other object types. A type may *specialise, generalise* or be *partitioned-by* other types.

The **attributive expression** declares the properties of the objects of the type being defined.

The **constructive expression** determines the mode of construction for complex objects. Three constructors exist, namely *association* (which builds a set), *sequentiation* (which builds a list) and *aggregation* (which builds a tuple). These constructors may be *non-cohesive* (in which case they are denoted by { }, [ ] and < > respectively) or *cohesive* (in which case they are denoted by {{ }}, [[ ]], and << >> respectively). A type definition can have at most one constructive expression and cannot both specify and inherit a constructive expression. The removal of an object from a collection of objects defined using a *cohesive* constructor leads to the deletion of the cohesively constructed object. As an example, the object type *location* in Figure 4 has a constructive expression which is an *aggregation* (see also Figure 3).

A **construction constraint** may be defined for object types involved in the construction and is composed of a *membership constraint* and a *correlation constraint.* The membership constraint determines whether or not it is required that every object of the constituent type occurs in the construction of some object of the type being defined. A membership constraint can take as value one of *exhaustive* or *sparse.* Correlation constraints allow the specification of whether or not any particular object of the constituent type can have one or more references to it. The value taken by a correlation constraint is one of *exclusive* or *shared.* The update semantics for cohesive constructors follow a similar approach to that of [22] with respect to the semantics of references to composite objects, where properties model *weak references*, non-cohesive constructors model *independent composite references* and cohesive constructors model *dependent composite references.* If we wish further to express whether any of these references is *exclusive* or *shared*, this is accomplished by defining *correlation constraints.*

The **interface expression** defines the set of operation interfaces of the operations which are publicly available. An operation interface consists of a name and a signature. The signature specifies the formal input parameters, their types, and optionally the type of the result that might be returned when the operation is invoked. An implementation for each one of the interfaces defined on an object type definition is given in the *class* declaration associated with each type declaration for which the interface expression is not empty. If no interface expression is given in an object type declaration a class is declared implicitly, and objects of this class can only be accessed and manipulated using the default behaviour.

A **method** is the code specifying the implementation of an operation. Objects of different classes may have different methods for the same operation. The choice of the method to be executed as a consequence of a request for an

```
type person:
  generalises: employee, customer;
  properties :
       name, age;
  interface :
       identify: name;
end-type
type employee:
  properties:
       ni, contract, salary;
  interface :
       identify: ni;
       increment_salary(incr: salary);
end-type
type customer:
  properties:
       purchase, income;
end-type
type location:
  construction: < street, city, postcode >;
end-type
type company:
  properties:
       location, company_name;
  interface :
       identify: location;
end-type
type employment:
  < (employee,exhaustive,shared),
    (company, sparse, exclusive) >;
end-type
```

Figure 4: Object type declaration examples

operation is performed by binding the operation to an actual implementation taking into account overriding and late binding.

Types are declared as a means of defining the structure of classes. A class is determined by (and has the same name as) the type which specifies its structure. Two kinds of method can be defined for a class: private methods and public methods. A private method can only be called from other methods attached to its class, or subclasses thereof. For private methods, both the interface and the implementation are specified within the class definition. Figure 5 presents class definitions corresponding to the object type definitions in Figure 4. The data manipulation language described below is the language used for method implementation.

4.3.2 Data Manipulation

The Data Manipulation Language is a language for specifying operations. It provides a syntax for defining operations on objects and for requesting that operations be performed on objects. The model of computation adopted is the messaging one according to [16] that is, the recipient of the message is a distinguished parameter within the request. The message sending operator is @. The message recipient is an object and a message can also be sent to *self* or to *super*.

The constructs supported by the Data Manipulation Language are roughly the ones supported by most imperative programming languages, i.e. assignment, control structures such as *while* and *if then else*, and I/O statements. A further control statement is the *foreach* construct which may be used to iterate over the elements of a set or a sequence, or over the instances of a class. All these constructs have a mode of operation in which they may return an object (or a set of objects in the case of *foreach*) and can therefore be assigned to an *object name*. In general, the language offers two main types of syntactic classes namely *object expressions* and *statements*. Object expressions return an object when they are executed, while statements do not return an object but instead provoke a change in the state of computation.

```
class person:
  public:
    identify: name begin get_name@self end
end-class
class employee:
  public:
    identify: ni begin get_ni@self end
    increment_salary(incr:salary)
       begin
          put_salary((get_salary@self) * (1 + incr / 100))@self;
       end
end-class
class company:
  public:
    identify: location begin get_company_name@self end
end-class
```

Figure 5: Class declaration examples

The system provides a set of built-in methods to manipulate objects i.e. their properties and construction. In order to ease the programmer's task by offering a simpler syntax to express computations, most of the built-in methods need not be written in a message sending style. Ultimately, all these expressions can be recast in the message sending style, and in fact their associated meaning is defined using this form. For example, each class whose construction is an association has a method *add_member* which takes as parameter an object which is to be added to the association. Such an operation can be invoked by (*add_member(a)@b*), or using the simpler and more intuitive syntax *b ++ a*.

For each property the system generates a pair of methods whose names are formed from the name of the respective property with the prefixes *get_* and *put_*. These methods allow the object referenced by a property to be retrieved or replaced, and can be made public or redefined in the public part of a class definition.

A class can be assigned to a persistent environment, and if so the type associated with the class, its methods and its instances are made to persist. A persistent class is only allowed to reference other persistent classes. Examples of how to make class and object type definitions persist and how to manipulate them are given in Figures 6 and 7 respectively. Figure 6 shows how an environment, which is a parameter to a program, can be associated with a class by naming the environment in the definition of the class. Figure 7 shows how such an environment is subsequently accessed, and includes an example of a *foreach* statement which awards a salary increase to all employees older than *18*.

```
program employees_schema(ENV)
begin
   ...
   class ENV.person:
      public:
         identify: name begin get_name@self end
   end-class
   ...
end
```

Figure 6: Defining a persistent class

4.4 Integrating The Languages

The logic language and the DBPL can be fully integrated in the following sense – the logic language can call any method written in the DBPL as long as that method does not (directly or indirectly) change any global data; the DBPL can invoke any logic language expression for which an appropriate type can be inferred. This latter restriction on the calling of a logic expression from the DBPL applies because it is not possible to infer a single result type for

```
program employees_management(ENV)
begin
    ...
    foreach y in employee do
          if get_age@y > 18 then increment_salary(5)@y;
    ...
end
```

Figure 7: Manipulation of a persistent environment

a number of different goal categories – for example, goals in which the method name is a variable.

The exact syntax to be used for embedding the logic language within the DBPL has yet to be finalised. However, the nature of the integration can be illustrated using the example in Figure 8, in which the program in Figure 7 has been modified to iterate over a set specified using the logic language.

```
program employees_management(ENV)
begin
    ...
    foreach y in [E | E[iof => employee], E[.age => A], A :> 18] do
          increment_salary(5)@y;
    ...
end
```

Figure 8: Integrating a logic query with the DBPL

.

5 Summary

5.1 Current Position and Future Plans

The data model and DBPL are fully specified in [15] and [9], and a prototype of the logic language has been implemented using MegaLog, as described in [14]. The data model and DBPL are currently being implemented using EXODUS [27], which will then be used to re-implement the logic language. The formalisation of the logic language is underway.

A central aim of the project is to evaluate the resulting system using an application in the area of geographic information systems. To this end work has been done to identify key areas of Geographic Data Management where the power of a DOOD yields significant benefits over conventional techniques [2, 1, 30].

5.2 Conclusions

This paper has presented an approach to the development of a deductive object-oriented database system in which a semantically rich object-oriented data model is used to underpin a logic query language and an imperative data manipulation language. The resulting system provides comprehensive support for semantic data modelling, user queries and application development. This integration of facilities provides improved database support for the software development process based upon an object-oriented view of design. The data model presented in section 4.1 supports the classical abstraction mechanisms supported by many semantic data models, and as such can be used during both design and implementation. Furthermore, method interfaces specified during the design phase can be directly implemented using the database programming language presented in section 4.3, and rule-based solutions specified using the logic language described in section 4.2.

To summarise, the principal benefits of the approach presented here are:

- There is no longer an emphasis on the data model, the query language or the programming language at the expense of other components, a feature of many other proposals identified in section 2.

- The database system provides facilities which reduce the size of the step which must be taken when moving from design to implementation.
- The logic language is powerful enough to support the expression of complete rule-based applications or components of applications, and is not limited to straightforward retrieval operations.
- The envisaged advantages of deductive object-oriented databases are supported – a theoretical foundation for object-oriented modelling and querying; rule-based applications are freed from the limitations of the relational model; OODBs can be queried using a rule-based language.

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 colleague 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 as the industrial partners in the project.

References

[1] Alia I. Abdelmoty, M. Howard Williams, and Norman 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, Singapore, June 1993. Springer-Verlag. ISBN 3-540-56869-7, 0-387-56869-7.

[2] Alia I. Abdelmoty, M. Howard Williams, and Michael Quinn. A Rule-Based Approach to Computerized Map Reading. *Information and Software Technology*, 35(10):587–602, October 1993.

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

[4] Serge Abiteboul and Richard Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[5] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 565–575, Barcelona, June 1991. Morgan Kaufmann Publishers.

[6] T. Andrews and C. Harris. *The Ontos Object Database*. Ontologic Corp., 1989.

[7] M.P. Atkinson and O.P. Buneman. Types and Persistence in Database Programming languages. *ACM Computing Surveys*, 19:105–190, June 1987.

[8] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17, 1981.

[9] 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.

[10] David Beech. A Foundation for Evolution from Relational to Object Databases. 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 251–270, Venice, Italy, March 1988. Springer-Verlag. ISBN 3-540-19074-0, ISBN 0-387-19074-0.

[11] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley Publ. Co., Wokingham, England, 1992. ISBN 0-201-56897-7.

[12] O. Deux et al. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[13] 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.

[14] 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.

[15] 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.

[16] Elizabeth Fong, William Kent, Ken Moore, and Craig Thompson. X3/ SPARC/ DBSSG/ OODBTG Final Report, September 1991.

[17] Georges Gardarin. Rule Languages for Relational and Object-Oriented Databases. In M.S. Jackson and A.E. Robinson, editors, *Aspects of Databases: The Proceedings of the Ninth British National Conference on Databases*, pages 1–16, Wolverhampton, July 1991. Butterworth-Heinemann Ltd.

[18] Peter M.D. Gray, Krishna G. Kulkarni, and Norman W. Paton. *Object-Oriented Databases: A Semantic Data Model Approach.* Prentice-Hall International(UK), Hertfordshire, 1992. ISBN 0-13-620203-3.

[19] Michael Hammer and Dennis McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[20] 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 [21].

[21] 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.

[22] Won Kim, Elisa Bertino, and Jorge F. Garza. Composite Objects Revisited. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 337–347, Portland,OR, June 1989. ACM Press. SIGMOD Record, 18(2), June 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] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *OOPSLA '86 Conference Proceedings Object-Oriented Programming: Systems, Languages and Applications*, pages 472–482, Portland,OR, September 1986. ACM Press.

[25] Geoffrey Phipps, Marcia A. Derr, and Kenneth A. Ross. Glue-Nail: A Deductive Database System. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on the Management of Data*, pages 308–317, Denver, Co, May 1991. ACM Press. SIGMOD Record, 20(2), June 1991.

[26] Alexandra Poulovassilis and Peter King. Extending the Functional Data Model to Computational Completeness. In François Bancilhon, Costantino Thanos, and Dennis Tsichritzis, editors, *Advances in Database Technology - EDBT'90, International Conference on Extending Database Technology*, LNCS 416, pages 75–91, Venice,Italy, March 1990. Springer-Verlag. ISBN 3-540-52291-3, ISBN 0-387-52291-3.

[27] 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.

[28] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981. Also in [32], pages 95-111.

[29] The Committee for Advanced DBMS Function. Database Systems: Achievements and Opportunities. *ACM SIGMOD Record*, 19(3):31–44, September 1990. The *Committee* is composed of Michael Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Brodie, Philip Bernstein, and David Beech.

[30] M.H. Williams, N.W. Paton, A. Abdelmoty, A.A.A Fernandes, and M.L. Barja. Applying Advanced Database Techniques to Geographic Information Systems. In M.J. Cardwell et al., editor, *JFIT Technical Conference Digest*, pages 221–228, University of Keele, March 1993. DTI/SERC.

[31] Carlo Zaniolo. The Database Language GEM. In *Proceedings of the 1983 ACM SIGMOD International Conference on the Management of Data*. ACM Press, 1983. Also in [32], pages 449-460.

[32] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, CA, 1990. ISSN 1046-1698, ISBN 1-55860-000-0.