The Formalisation of ROCK & ROLL: A Deductive Object-Oriented Database System

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

Department of Mathematical Sciences,
Goldsmiths College New Cross,
London SE14 6NW, UK,
email: a.fernandes@gold.ac.uk

Department of Computing and Electrical Engineering†,
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS, Scotland, UK
e-mail: <marisa,howard>@cee.hw.ac.uk

Department of Computer Science††,
University of Manchester,
Oxford Road, Manchester M13 9PL, UK.

Abstract. This paper describes the formalisation of the deductive object-oriented database system ROCK & ROLL. This is a system which integrates the deductive and object-oriented paradigms in a way that is both clean and consistent, and that has a sound theoretical foundation. The system uses a formally defined object-oriented data model as a foundation for both a logic query language and an imperative data manipulation language in such a way that impedance mismatches are minimised. This paper introduces the facilities offered by ROCK & ROLL, and indicates how their formalisation has been achieved.

1 Introduction

The deductive and the object-oriented approaches to database management have risen to prominence in the database research community because each has distinctive characteristics that are suitable for the retrieval or management of persistent data. The deductive approach supports the declarative expression of queries and rules in the context of data models which are essentially relational and a formal framework which builds upon first order logic. The object-oriented approach provides comprehensive mechanisms for structuring both programs and the data to which they relate, often in the context of the widely used imperative programming paradigm. However, most representatives of each of these categories of system lack certain of the characteristic strengths of the other. For example, few deductive databases provide modelling mechanisms such as identity and inheritance, and many object-oriented databases have no mechanism for declarative query expression, and lack formal underpinnings.

The complementary nature of the deductive and the object-oriented paradigms has led to considerable interest in their integration, whereby it is hoped that new systems will emerge which use the strengths of each of the paradigms to overcome the weaknesses of the other. It is clear, however, that seamless integration of the two paradigms is not straightforward, and several different strategies have been adopted [18]. This paper focuses upon a new approach, represented by the deductive object-oriented database system ROCK & ROLL[1], which has been designed and implemented at Heriot-Watt. ROCK & ROLL has three tightly integrated components:

- An object-oriented data model (*OM*), which provides a rich collection of structural abstractions, such as generalisation, association, sequentiation and aggregation, with which to describe the domain of discourse.
- A deductive query language (*ROLL*), which allows queries and rules to be defined in a wholly declarative manner, and which operates over an extensional database described by *OM*.

[1] ROCK stands for Rule Object Computation Kernel, and ROLL for Rule Object Logic Language

- An imperative programming language (*ROCK*), which allows for the definition, creation and manipulation of *OM* databases.

The relationship between these components is depicted graphically in figure 1. The factor which is crucial to the success of the overall system is the way in which the two languages, ROCK and ROLL, can be used together to implement the solution to a problem – ROLL queries and rules can be invoked from ROCK, and side-effect free ROCK methods can be invoked from ROLL [8]. This allows the advantages of the deductive and the object-oriented paradigms to be exploited together in a single system, an approach which is presently being evaluated in a geographic application [1].

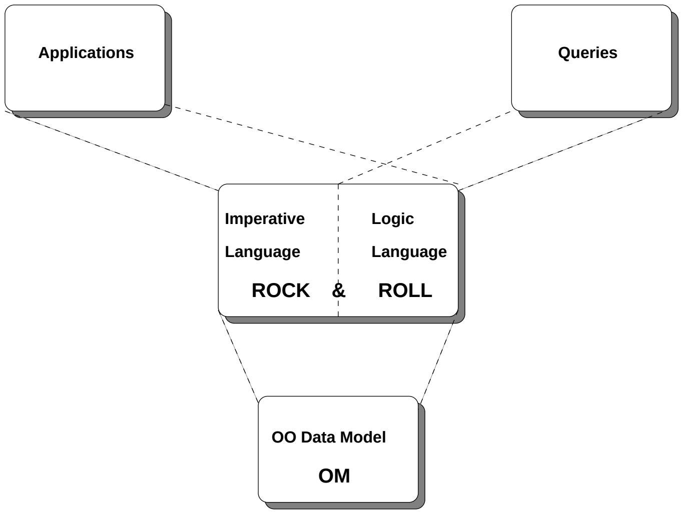

Figure 1: Relationship between the principal components in ROCK & ROLL.

One of the recognised strengths of the deductive approach to data management which is almost always missing from object-oriented databases is a formal definition of the semantics of the system. The formal underpinnings of deductive systems have been useful in directing the design of extensions to core deductive languages such as Datalog, and have supported extensive work on optimisation of deductive rule bases. For a deductive object-oriented database (DOOD) such as ROCK & ROLL to become widely accepted as an effective example of its kind, it is important for the formal foundations of the deductive approach to be retained. This paper shows how the semantics of ROCK & ROLL has been formally specified. This is a substantial task, as ROCK & ROLL is a multi-component system, with all of the components requiring to be formally specified, namely the data model OM, the logic language ROLL, and the imperative language ROCK. Furthermore, to be useful in directing the implementation of the overall system, the scope of the formalisation must be sufficiently great to address both the compilation and execution of ROCK & ROLL programs.

The paper is structured as follows. Section 2 provides some pointers to related work on DOODs and on specification of database systems. Section 3 gives an informal introduction to the facilities of ROCK & ROLL. Sections 4 and 5 describe how OM, ROLL and ROCK have been formally specified. Section 6 presents some conclusions.

2 Related Work

2.1 Formalisation of Database Systems

Database research has built extensively upon formal descriptions of data models, languages and processes. This subsection makes no attempt to be comprehensive in scope, but does identify some representative examples of where database systems have benefited from formalisation.

Models: Many data models have formal roots, and the relational model in particular has been studied from both model theoretic and proof theoretic perspectives [33]. It is often lamented that object-oriented data models lack formal descriptions, but several models have been described formally, including O_2 [28] and IFO [3]. A variety of different approaches have been taken to the formal description of data models; the one adopted in this paper is axiomatic, for which a precedent exists in [10], but with very different guiding principles.

Languages: The potential for analysis and optimisation of programs that access databases has led to considerable exploitation of formally defined languages. For example, in the context of the relational model, relational algebra queries can be rewritten in a way which is likely to lead to more efficient evaluation [35]. In the deductive database community [14], alternative and equivalent semantics for logic rules have pointed the way towards alternative evaluation strategies, supported a range of optimisation techniques, and provided a gateway to more advanced issues such as integrity constraints [15] and incomplete information [37].

It is quite rare for formal semantics to be given to database programming languages. Most imperative database programming languages have evolved from existing programming languages, which are extended to provide support for database functionality such as some form of persistence, e.g. E [34], CO_2 [16]. It is commonly the case that neither the base languages nor the extensions are formally specified. The functional database language Galileo [4] integrates features to support semantic modeling capabilities into the programming language Standard ML [29]. Galileo can be considered to be formally defined to the extent to which it overlaps with ML (which has a semantics defined using *Structural Operational Semantics* [31, 22]). To our knowledge, the extensions have not been formally defined. The Persistent Functional Language PFL, which extends functional programming towards deduction, has a semantics defined using a a denotational approach [32]. This definition is based on defining the semantics of the kernel language which is gradually extended as new features are introduced into the language. The relationship between different database formalisms is discussed in detail by [21].

Systems/Processes: As sophisticated software systems, databases are a rich domain for the application of formal methods. As representative examples, [30] presents a formal specification of an object management system, [5] reasons about the termination of active rules using relational algebra, and [36, 12] indicate how active database systems can be formalised using Denotational Semantics and Object-Z, respectively.

Despite the widespread interest in formal database systems, relatively few implementations of database systems are fully formally specified. One of the contributions of this paper is to show that such a process is both practical and useful – the formalisations of the different parts of ROCK & ROLL were developed before the system was implemented, and thus underpinned the implementation process.

2.2 Formalisation of Deductive Object-Oriented Databases

In principle, there are a number of different ways in which a DOOD can be constructed, each of which is likely to lead to an alternative approach to formalisation. For example, a deductive language could be extended with object-oriented features, an object-oriented database could be extended with deductive features, or a DOOD could be developed from scratch.

The first of these, namely the extension of deductive languages with object-oriented features, has been extensively explored, with representative examples being IQL+ [2] and Logres [11]. Such an approach proceeds along similar lines to the extension of Datalog with negation or updates, and the extension of the underlying theory, which is commonly based upon fixpoints, goes hand-in-hand with the design of the additional constructs. Such approaches generally support some notion of identity and inheritance, and lead to the development of rule-based database programming languages in which complete applications can be implemented.

The second of these, namely the extension of an OODB with deductive features has been less frequently addressed; this is at least partly because of the limited number of formally defined object-oriented data models. ConceptBase is one example, in which the modelling formalism Telos is associated with a logic language for expressing queries and methods [23]. The approach taken in ROCK & ROLL is not out of place in this category, in that the logic language is defined in the context of an object-oriented data model defined as a set of first-order axioms.

The third of these, namely the development of a DOOD system independent of earlier proposals, leaves most scope for a radical approach to the problem. One such strategy is represented by the development of F-logic [24], a new first-order logic with a higher-order flavour. Such an approach provides scope for elegant new proposals to be made, but reduces the extent to which existing implementation technologies can be applied to DOODs.

3 Introduction to ROCK & ROLL

This section provides a brief informal introduction to ROCK & ROLL by way of an example concerning the storage of information on hierarchies of parts. In OM the structure of an object and the interface of every method associated with an object are determined by the *type* of the object. For example, the type `part` in figure 3 has the stored properties `name` and `number`, and the ROCK methods `compute_mass` and `compute_cost`.

```
type part:
    properties:
        name: string,
        number: int;
    ROCK:
        compute_mass(): real,
        compute_cost(): real;
end-type

type base_part:
    specialises: part;
    properties:
        mass    : real,
        imported: bool,
        supplier: string,
        cost    : real;
    ROCK:
        compute_mass(): real;
        compute_cost(): real;
end-type

type composite_part:
    specialises: part;
    properties:
        assembly_cost : real,
        mass_increment: real;
    [ part ];
    ROCK:
        compute_mass(): real,
        compute_cost(): real,
    ROLL:
        is_component(part);
end-type
```

Figure 2: Type definitions for parts example

The private properties of a type and the code implementing methods are given in the *class* definition. Every type is associated with a class of the same name, the idea being that the *type* specifies all that a user needs to know in order to use the type, and that the *class* contains details of the way in which the type is implemented. The class `part` provides default implementations of the methods introduced on the corresponding type:

```
class part
     public:
         compute_mass(): real begin 0 end
         compute_cost(): real begin 0 end
end-class
```

Subtypes of `part` allow a distinction to be drawn between base parts and composite parts, as represented by `base_part` and `composite_part` in figure 3. In `composite_part` two additional features are illustrated: a `composite_part` is defined to be constructed from a list of `part` objects (denoted by `[ part ]`), and the method `is_component` is introduced which is implemented using ROLL rather than ROCK; a distinction is drawn in the specification of method interfaces between ROCK and ROLL methods because they are invoked differently – further details are given in [8].

It can be seen from figure 3 that both `base_part` and `composite_part` redefine the methods `compute_mass` and `compute_cost` defined on the superclass `part`. Such redefinition of methods is termed *overriding*, and *dynamic binding* is used to select a method definition for execution in response to a specific invocation, depending on the type of the recipient.

```
class base_part
    public:
        compute_mass(): real
        begin
            get_mass()@self
        end

        compute_cost(): real
        begin
            get_cost()@self
        end
end-class

class composite_part
    public:
        compute_mass(): real
        begin
            var tot := 0.0;
            foreach p in self do
            begin
                tot := tot + compute_mass()@p;
            end
            tot + get_mass_increment()@self
        end

        compute_cost(): real
        begin
            ...
        end

        is_component(part)
        begin
            is_component(P1)@Self :-
                isin(P1)@Self;
            is_component(P1)@Self :-
                isin(P2)@Self, is_component(P1)@P2:composite_part;
        end
end-class
```

Figure 3: Class definitions for parts example

Example method definitions are presented in figure 3. From the definition of `compute_mass` on `composite_part`, it can be seen that ROCK supports method invocation (the message sending operator is `@`), variable definition, iteration and assignment; it also supports input/output, conditionals, and other features which would be expected in a conventional strongly-typed imperative object-oriented programming language.

The one ROLL method defined in the example application is `is_component`. The clauses of `is_component` can be read thus: a part `P1` is a component of the part `Self` if `P1` is in the list from which `Self` is constructed; a part `P1` is a component of the part `Self` if `P2` is in the list from which `Self` is constructed and `P2` is a `composite_part`, and `P1` is a component of `P2`. The invocation of ROLL methods from ROCK is considered further in section 5.5.

All ROCK & ROLL programs are statically type checked, with a type inference system minimising the need for explicit association of types with variables. A more comprehensive example of the application of ROCK & ROLL is presented in [1], where it is shown how the facilities summarised above can be applied in the domain of geographic information systems.

4 Specifying the Data Model and the Logic Language

This section shows how the object-oriented data model OM has been formalised as a set of first-order axioms which constrain the set of valid database states, and how this formal description of the data model can be used to induce a logic query language for objects. The process indicates that familiar object-oriented constructs can be made accessible to a logic language without complicating the semantics of that language, which are essentially those of Datalog [14].

The output from this section is a necessary prerequisite for the formalisation of the complete ROCK & ROLL system, in that it describes formally what data can be stored extensionally in OM, and how intensional information can be inferred from databases structured in that way. The picture is completed in section 5, where it is shown how the programming language environment of ROCK & ROLL has been formally specified.

4.1 The Data Model (OM)

This section describes how an OODB can be presented as an axiomatic system of first-order logic, culminating in the characterisation of the OODB as a domain-specific *object theory* (OT), i.e. a set of axioms in an applied first-order language. Compared to the relational data model, which is a first-order theory with minimal application-independent modelling axioms, an OODB has a rich semantics that requires a significant number of axioms to characterise both schema-level and instance-level concepts.

For this purpose we use a language called OLL (for Object-Logic Language). OLL is an applied language of first-order logic without function symbols. The set of predicate symbols of OLL is denoted by $\mathcal{P}$, the set of individual constants by $\mathcal{C}$, and the set of terms by $\mathcal{T} = \mathcal{C} \cup \mathcal{V}$, where $\mathcal{V}$ is a countably infinite set of individual variables. The set $\mathcal{C}$ of individual constants may be partitioned into up to six sorts:

- T, whose elements denote *object types.*
- P, whose elements denote *primitive types.*
- M, whose elements denote *operations.*
- O, whose elements denote *(secondary) objects.*
- V, whose elements denote *primary objects* (i.e. *values*).
- N, whose elements denote *indices* into sequences.

The axioms formally specify the relationships which may exist between these fundamental constituents of an object-oriented data model. In writing down OLL axioms the following notational conventions are used. Letters taken from the beginning of the lower-case alphabet, e.g., a, b, c, d, e, f, denote metalanguage variables, and as such act as placeholders for OLL constants. Letters taken from the end of the lower-case alphabet, e.g., u, v, w, x, y, z denote OLL variables.

If $x_1, \ldots, x_n$ are all the free variables in a quantifier-free formula F, the **universal closure** (resp. **existential closure**) $(\forall\, x_1) \ldots (\forall\, x_n)(F)$ (resp. $(\exists\, x_1) \ldots (\exists\, x_n)(F)$) of F is usually written $\forall(F)$ (resp. $\exists(F)$). A sequence of quantifiers of the same kind $(\chi\ x_1) \ldots (\chi\ x_n)$, where χ is either $\forall$ or $\exists$, is usually written $(\chi\ x_1 \ldots x_n)$.

4.1.1 Schema and Instance Axioms

The first step is to specify a set of axioms which describe schema level and instance level information. These axioms are ground, resulting from full instantiation of axiom schemes, and include the following subsets:

1. *abstraction axioms* which partition object types into two sets: application-dependent (which use object identifiers to name their instances) and application-independent (which have values belonging to a primitive type in the environment as instances). The following are the abstraction axioms:

   An a is an application-dependent abstraction.

   $$application\_dependent(a) \tag{1}$$

   Intuitively, when a is an application-dependent type, the instances of a are identified by proper object identifiers. This implies that instances of a have structural and behavioural characteristics that are specific to the application domain under consideration.

   An a is an application-independent abstraction, named by an instance of the primitive type b.

   $$application\_independent(a, b) \tag{2}$$

   Intuitively, when a is an application-independent type, the instances of a are identified as values of the primitive type b. This implies that instances of a have whatever structural and behavioural characteristics the environment provides for the primitive type b.

2. *inheritance-specification axioms* which declare inclusion relationships, and hence define inheritance paths, between object types. For example:

   An object type a specializes an object type b.

   $$specialization\_of(a, b) \tag{3}$$

   Intuitively, every instance of a is an instance of b. Similar axioms define *generalization-of.*

3. *structural axioms* which come in two groups:

   (a) *property-type axioms* which declare that instances of some object type are referred to by instances of another as properties:

   An a has a b-property.

   $$has(a, b) \tag{4}$$

   Intuitively, an instance of a refers in its state to an instance of b as its property b. Note that type names are used to name properties rather than introducing yet another set of symbols, traditionally known as attribute names. Type aliases can be used to tidy up name clashes if necessary.

   (b) *construction-type axioms* which declare that instances of some object type are referred to by instances of another as construction elements.

   An a is built as an association of bs.

   $$association\_of(a, b) \tag{5}$$

   Intuitively, an instance of a is constructed as an unordered set of instances of b. More precisely, an instance of a contains in its state a non-indexed, type-homogeneous collection of references to instances of b.

   An a is built as a sequentiation of bs.

   $$sequentiation\_of(a, b) \tag{6}$$

   Intuitively, an instance of a is constructed as an ordered set of instances of b. More precisely, an instance of a contains in its state a N-indexed, type-homogeneous collection of references to instances of b, where N, the set of indices, is isomorphic to a finite segment of the natural numbers.

   A further axiom, not given here, is used to describe *aggregaton*s.

4. *interface axioms* which declare operation interfaces that are available to interact with instances of an object type. For example: operation b in object-type a has an interface consisting of argument-types $c_1, \ldots, c_n, n \geq 0$.

   $$op\_interface(a, b, c_1, \ldots, c_n) \tag{7}$$

5. *type-assignment axioms* which establish an object as an instance of some type. For example:

   a is an application-dependent object whose object type is b.

   $$object(a, b) \tag{8}$$

   Intuitively, a is an instance of the application-dependent type b. This assignment implies that the structure and the behaviour of the object a are those defined for the type b in the application-domain.

   a is an application-independent object whose primitive type is b.

   $$value(a, b) \tag{9}$$

   Intuitively, a is an instance of the application-independent abstraction b. This assignment implies that the structure and the behaviour of the value a are those defined for the type b in the environment.

6. *property value-assignment axioms* which establish a property reference from one object to another:

   a references c as its property b.

   $$property(a, b, c) \tag{10}$$

   Intuitively, the current value of object a's b-property is the (possibly primary) object c.

7. *construction element-assignment axioms* which establish a construction reference from one object to another. For example:

   a references c as an element in its construction by association.

   $$member(a, c) \tag{11}$$

   Intuitively, the (possibly primary) object c is a *member* of object a, or c *is in* a.

   a references c as the b-th element in its construction by sequentiation.

   $$term(a, b, c) \tag{12}$$

   Intuitively, the (possibly primary) object c is the *b-th term* of object a, or c *is b-th in* a.

   A further axiom, not given here, defines how an aggregation is constructed from its members.

4.1.2 Semantic Completion Axioms

These are axioms which define the semantics of the modelling relationships used in OM, and include (among others) the following subsets:

1. *semantic-relationship axioms* which characterise the extension of semantic relationships from the ground axioms presented above. For example:

   If x specializes y or y generalizes x then an x is a y.

   $$\forall\,(specialization\_of(x, y) \lor generalization\_of(y, x) \Rightarrow is\_a(x, y)) \tag{13}$$

   Intuitively, the is_a relation abstracts away from the particular form of axiom used to convey the inclusion relationship between x and y.

   The is_a relation is reflexive over object types and transitive.

   $$\forall\,(application\_dependent(x) \Rightarrow is\_a(x, x)) \tag{14}$$
   $$\forall\,(is\_a(x, y) \land is\_a(y, z) \Rightarrow is\_a(x, z)) \tag{15}$$

2. *inheritance axioms* which define the conditions under which structural and behavioural inheritance take place. For example:

   If x is a y then if y has a z-property then so does x.

   $$(\forall\,xy)(is\_a(x, y) \Rightarrow (\forall\,z)(has(y, z) \Rightarrow has(x, z))) \tag{16}$$

3. a *type-extension axiom* which defines the essential link between a database schema and a database state.

   If x is an instance of y then x is also an instance of z for every object type z that is a supertype of y.

   $$(\forall\,xy)(object(x, y) \Rightarrow (\forall\,z)(is\_a(y, z) \Rightarrow object(x, z))) \tag{17}$$

4. *object-state axioms* which define how the state of an object conforms to the definition of the object types of which the object is an instance. For example:

   If an object x is an instance of y then for every z that is a property-type of y there is an instance w of z that is referenced by x as its property z.

   $$(\forall\,xy)(instance\_of(x, y) \Rightarrow (\forall\,z)(has(y, z) \Rightarrow (\exists\,w)(property(x, z, w) \land instance\_of(w, z)))) \tag{18}$$

5. *index-ordering axioms* which define properties of indices within sequentiations, essentially that they form a chain.

4.1.3 The Axioms in Clausal Form

The set of axioms presented in the previous sections are given as formulae in first order logic. However, it is convenient to convert the set of OOL formulas from section 4.1 into the corresponding Horn theory, using the algorithm of [27]. For reasons of space, the clausal version of the axioms is not presented here. However, an example, axiom (13) translates into the following rules:

$$is\_a(x, y) \leftarrow specialization\_of(x, y)$$
$$is\_a(x, y) \leftarrow generalization\_of(y, x)$$
$$is\_a(x, y) \leftarrow partitioned\_by(y, x)$$

4.2 The Logic Language (ROLL)

This section indicates how a logic query language can be derived from the clausal version of the axiomatically defined data model presented in section 4.1. As the set of Horn clauses which represents OM consists of logic rules, it can already be held that a deductive database exists. For example, from the facts:

$$specialization\_of(composite\_part, part)$$
$$has(part, name)$$

and the rules corresponding to axioms (13) and (16), it can be deduced that:

$$is\_a(composite\_part, part)$$
$$has(composite\_part, name)$$

In this context, it is straightforward for application-specific rules to be defined. For example, the following rule identifies *newparts* as *part* objects with a *number* greater than 1000 [2]:

$$\begin{aligned} newparts(P) \leftarrow\ & object(P, part), \\ & property(P, number, Num), \\ & Num > 1000. \end{aligned}$$

The logical machinery so far gives support for object identity, inheritance, complex objects, and the expression of derived data by means of deductive rules. However, the behaviour-based view of objects that is at the heart of the object-oriented paradigm is still absent. Up to this point, the model can be claimed to be structurally but not behaviourally object-oriented.

Incorporating Logic Methods: The presence of methods is described in the axiom set by axiom (7). For example, if there exists an operation interface

$$op\_interface(base\_part, total\_cost, real)$$

then the corresponding implementation of such a method can be defined using *op_implementation* as follows:

$$\begin{aligned} op\_implementation(base\_part, total\_cost, Part, Value) \leftarrow\ & object(Part, base\_part), \\ & property(Part, cost, Cost), \\ & property(Part, mass, Mass), \\ & Value = Mass/Cost. \end{aligned}$$

This allows logic rules to be defined which are associated with types, and thus starts to provide the functionality associated with methods. There are, however, a number of deficiencies: the syntax used is cumbersome, and there is no support for *encapsulation*, *overloading*, *overriding* or *dynamic binding*. These concepts are supported through the introduction of the user-level logic language ROLL.

Message Sending and Encapsulation: Encapsulation is lacking in the context described so far because all information associated with an OM database is equally accessible to clausal OLL programs or methods. Since the user language ROLL is strictly distinct from clausal OLL, encapsulation can be enforced by restricting ROLL atoms to the form

$$m(\theta_1, \ldots, \theta_n)@o$$

[2] The Prolog convention of capitalizing logical variables is adopted from now on.

where m is the name of an operation with an interface specified using $op\_interface$, o is the object to which the message is sent, and $\theta_1, \ldots, \theta_n$ are the arguments of the operation, conforming in type and number to those specified in $op\_interface$. As an example, the following ROLL atom can be read as "send the message $total\_cost$ to the $part$ P with the argument V":

$$total\_cost(V)@P$$

Supporting Overriding and Late Binding: The notions of overriding and late binding allow the decision as to which implementation of an operation should be invoked in response to a message-send to be deferred until run-time. Overriding occurs when there may be definitions of a method m attached to both types T_1 and T_2, where $is\_a(T_2, T_1)$. Late binding occurs in this context when a message is sent to a variable v of type T_1, as v may hold objects which are instances of type T_1 or of type T_2. The decision as to which definition of m is used is deferred until run-time, when the type of the object assigned to v is known, a process known as late binding.

To support overriding, the axioms of OM allow multiple definitions of $op\_implementation$ associated with different types along an inheritance path to provide implementations for a single $op\_interface$.

To support late binding, a rewriting step is required when a method is invoked from ROLL. When a ROLL atom associated with an overridden operation is to be matched by the rule evaluator against the clause base, the definition of $op\_implementation$ which is selected is the most specific one associated with the type of the message recipient. That such a method exists is guaranteed by a well-formedness axiom of OM.

4.3 Summary

This section has outlined how an expressive object-oriented data model can be described as an object theory, in a manner which allows a Horn clausal version of the model to be queried by a logic language with conventional syntax and semantics. The conventional nature of the language permits evaluation techniques associated with deductive relational databases to be applied in the context of extensional databases structured using OM.

Behavioural features which are characteristic of the object-oriented paradigm, such as encapsulation and late binding are supported through the introduction of the user language ROLL. ROLL has a syntax which reflects the object-oriented structure of the data model over which it acts, and has a semantics defined by mapping ROLL programs into clausal OLL, which is essentially Datalog in its syntax and semantics (extended in the standard way with negation and built-in predicates).

The complete axiom set and a more formal description of the mapping which characterises the semantics of ROLL is presented in [19]. The logic language ROLL is described more fully in [17].

5 Specifying the Compiler and the Imperative Language

Section 4 has outlined the formal description process that indicates which structures can be described using OM and how additional information can be deduced from these structures using ROLL. There is a sense in which this represents the semantics of a complete deductive object-oriented database system, in that what results from implementing such a formalism directly is a system which supports both deduction and object-orientation [20]. However, the above steps in the formalisation process are not sufficient to provide a practical system. For example, there is no description of how updates or I/O are performed, and there is no consideration of issues such as type checking or interfaces to programming languages.

To overcome these limitations, the overall approach taken in ROCK & ROLL has been to integrate an imperative language with the logic language and database formalised in section 4 within the architectural framework outlined in figure 1. This increases the functionality of the system by facilitating I/O and updates, and adds to its flexibility by allowing complementary paradigms to be used together in a single system. It also means that the complete ROCK & ROLL system takes on many of the characteristics of database programming languages, such as compile time type checking, which are often lacking in deductive database systems. This requires that the formalisation process be extended to cover the compilation of ROCK & ROLL programs and the execution of ROCK statements, as described below.

The semantics of ROCK have been defined using *Structural Operational Semantics* (SOS) [31, 22], an approach in which both static and dynamic semantics can be specified precisely by syntax directed rules. Following this approach, the semantics of a language is given by a set of inference rules of the form:

$$\frac{P_1, P_2, ..., P_n}{C}$$

where $P_1, ..., P_n$ are the premises which if evaluated to true convey the conclusion C. The following subsections outline how the static and dynamic semantics of ROCK programs have been specified using SOS, and how a type inference mechanism for ROLL can be developed within the same framework; the complete rule set is given in [6].

5.1 Syntax

The syntax of a programming language can be specified using a grammar, and implemented using a syntax analyser generator. The parser used in ROCK & ROLL has been implemented using such a tool, and the grammar is defined correspondingly. As the inference rules used in SOS are syntax directed, it has also been necessary to develop an abstract syntax for ROCK & ROLL, which consists of a number of syntactic classes (*expressions, declarations, statements*), each of which is partitioned into a number of groups, which in turn are ranged over by metavariables in the specification.

5.2 Static Semantics

The static semantics is defined by a set of type checking rules which indicate how to check the type of a construct as a result of checking the types of its components. As an example, the following rule indicates how to type check an assignment in the context of an environment $\mathcal{E}$ which stores information on declared types and variables:

$$\frac{v \in dom(\mathcal{E}.VEnv), \mathcal{E}.VEnv(v) = \tau, \mathcal{E} \vdash objexp \Rightarrow \tau_1, \mathcal{E}.TEnv \vdash \tau_1 \sqsubseteq \tau}{\mathcal{E} \ \vdash \ v{:=}objexp \Rightarrow \mathcal{E}} \tag{19}$$

The rule can be read as follows: $v{:=}objexp$ is a valid assignment in the environment $\mathcal{E}$ and does not change the environment if v has been declared ($v \in dom(\mathcal{E}.VEnv)$), the type of v is τ ($\mathcal{E}.VEnv(v) = \tau$), the type of the object expression $objexp$ is τ_1 ($\mathcal{E} \vdash objexp \Rightarrow \tau_1$), and τ_1 is type equivalent to τ, i.e. is the same as or a subtype of τ. There are 73 such rules used to define the static semantics of ROCK.

5.3 Dynamic Semantics

The dynamic semantics of the language takes the form of a set of *state transition rules* which precisely define the evaluation of expressions and statements in the language. The general form of a state transition rule is similar to that of a type checking rule, although the notion of a *store* $\mathcal{S}$ is required, which is a pair of finite mappings from object names to locations and from locations to objects. The context within which a program executes is known as the *dynamic environment* $\mathcal{X}$, which consists of the environment $\mathcal{E}$ and the store $\mathcal{S}$. The following rule expresses the dynamic semantics of assignment:

$$\frac{\mathcal{X} \ \vdash \ objexp \Rrightarrow (o, \mathcal{X}^1)}{\mathcal{X} \ \vdash \ \ v{:=}objexp \Rrightarrow \mathcal{X}^1 + \{\mathcal{S}_V(v) \mapsto o\}} \tag{20}$$

The rule can be read as follows: the evaluation of the assignment $v{:=}objexp$ within the dynamic environment $\mathcal{X}$ yields a new dynamic environment $\mathcal{X}^1$ which results from $\mathcal{X}$ by enriching the store with the value o for v if $objexp$ can be evaluated within $\mathcal{X}$ to yield the object o and the dynamic environment $\mathcal{X}^1$. There are 57 such rules used to define the dynamic semantics of ROCK.

5.4 Type Inference in ROLL

In ROCK & ROLL, the logic language can be used to express queries (embedded or free standing) and methods. To conform with the expectations of users of other deductive database languages, and to retain concise expression of queries and rules, it was decided to develop a type inference mechanism for ROLL. The rules for inferring types are expressed using inference rules, and are again illustrated by example. The following rules define how a logic variable ℓ is associated with a type:

$$\frac{\ell \in dom(\mathcal{E}.FEnv)}{\mathcal{E} \vdash \ell \Rightarrow (\mathcal{E}, \mathcal{E}.FEnv(l))} \tag{21}$$

$$\frac{\ell \notin dom(\mathcal{E}.FEnv), \ell \in dom(\mathcal{E}.LEnv),}{\mathcal{E} \vdash \ell \Rightarrow (\mathcal{E}, \mathcal{E}.LEnv(\ell))} \tag{22}$$

$$\frac{\ell \notin dom(\mathcal{E}.FEnv), \ell \notin dom(\mathcal{E}.LEnv)}{\mathcal{E} \vdash \ell \Rightarrow (\mathcal{E} + \{\ell \mapsto \gamma\}, \gamma)} \tag{23}$$

The first of these rules states that if ℓ has been previously encountered as an argument of the clause being type checked ($\ell \in dom(\mathcal{E}.FEnv)$) then the environment remains unchanged, and the type of ℓ is that recorded in the environment ($\mathcal{E}.FEnv(l)$). The second of these rules handles the case when ℓ is not an argument of the clause, but has been previously encountered within the goal ($\ell \in dom(LEnv)$), in which case the environment remains the same, and the type of ℓ is that recorded in the environment ($\mathcal{E}.LEnv(\ell)$). The third rule handles the case when this is the first time that ℓ has been encountered within the clause, in which case the environment is extended with an entry which associates ℓ with the generic type γ ($\mathcal{E} + \{\ell \mapsto \gamma\}$), and the type γ is returned as the type of ℓ. The type inference algorithm for ROLL is based upon 25 such rules, and on the application of substitutions when more specific types are inferred.

5.5 Integrating ROCK & ROLL

It has been shown above how OM, ROLL, ROCK (compilation and evaluation) and ROLL type inference have been formally specified. What has not been dealt with is the invocation of ROCK from ROLL, and vice versa. ROCK programs can invoke ROLL methods using an embedded set-comprehension notation, illustrated by the following embedded query, which returns the set of `composite_part` objects that have a part with `name` equal to `"Widget"` as a component:

```
s := [ {P} | is_component(W)@P, get_name@W == "Widget" ];
```

In this example, the set of bindings for the variable `P` which results from the evaluation of the ROLL query `is_component(W)@P, get_name@W == "Widget"` is assigned to the ROCK variable `s`. The type inference system infers that the embedded query yields a set of `composite_part` objects, and is able to verify that the variable `s` has been declared to be of appropriate type (or, alternatively to infer a type for the ROCK variable using the revised syntax `var s := ...`). The membership of the set which is assigned depends upon the extension of the database, and is computed by the ROLL evaluator based upon the semantics outlined in section 4.2.

The invocation of ROCK from ROLL can be illustrated using a slight change to the previous example, in which the result contains the set of `composite_part` objects which have a part with cost greater than `1000` as a component:

```
s := [ {P} | is_component(W)@P, compute_cost@W > 1000 ];
```

This illustrates the invocation of the ROCK method `compute_cost` from the embedded ROLL query. Type checking proceeds as before – the type inference rules know about embedded ROCK methods, and are able to use their presence to constrain the types of ROLL variables. When the query is evaluated, the embedded ROCK method is treated in the same way as a built-in predicate – control is passed to the ROCK interpreter which evaluates the message in conformance with the evaluation rules outlined in section 5.3. Two checks are made to guarantee the safety of this process – the ROCK compiler only allows methods which contain no side-effects as candidates for embedding in ROLL; the ROLL evaluator ensures that a ROCK method is only invoked after its message recipient and all its input parameters are known.

6 Conclusions

This paper has shown how the deductive object-oriented database system ROCK & ROLL has been formalised. In so doing a number of contributions have been made:

1. An object-oriented data model has been formalised as an applied theory of first order logic.
2. This formalisation of the data model has been used as the formal basis for the development of a logic language for objects.
3. Structural operational semantics have been used to specify the static and dynamic aspects of an imperative database programming language, and a type inference system for a logic language without recourse to further specification mechanisms.

The semantics presented above were all completed before the implementation of the corresponding component of the ROCK & ROLL system. The role of the formalisation in the implementations of the components has differed from component to component.

In the case of the data model and logic language, two implementations have been developed. In the first, presented in [20], the implementation strategy closely followed the formal model, being based upon a representation of the data model as Prolog clauses and a logic language implemented as a metainterpreter. This allowed a verification that the semantics supported effective deductive capabilities in an object-oriented context, but was not intended to result in a scalable implementation. In the second implementation [7, 17] using the persistent C++ system E/EXODUS [13], the data model is implemented as a set of persistent C++ classes, and the logic language uses a technique based upon static filtering [25]. In this case, for the model, the formalisation served the more limited role of indicating what should be describable in the model, and thus essentially presented postconditions for operations on the object store – all object bases created had to conform to the formal description provided by the model. In the case of the logic language, it was clear from the formal model that the logic language was essentially Datalog with negation in the context of a rich data model. This meant that optimisation and evaluation strategies developed for Datalog with negation were applicable to ROLL, and as a result, the ROLL optimiser has been able to build directly upon earlier work on the optimisation of deductive relational databases. Where the ROLL optimiser extends the original work on static filtering [25] for deductive relational databases, this is generally to exploit type constraints that have been derived from the axioms that describe the data model.

In the case of the ROCK compiler and interpreter, and the ROLL type inference system, the implementation has been derived directly from the specification in structural operational semantics [9]. Type checking and program evaluation are carried out by sending messages to C++ classes that are structured to reflect the abstract syntax upon which the type rules of the semantics is based. In essence, every syntactic class in the language is represented within the system as a C++ class, and the each of the type checking and state transition rules from the specification is implemented as a C++ member function for typechecking or evaluation. There is thus a very direct mapping between the semantics and the implementation in this case.

To summarise, the ROCK & ROLL system is novel, in that it is a fully implemented DOOD based upon a semantically expressive data model and two languages, one deductive and the other imperative. It is also noteworthy in that it is a comprehensive database programming system that has been fully formalised prior to implementation. This formalisation has directed the design by constraining the logic language to support constructs that can be expressed naturally within the framework of first order logic, and by providing a clear specification of the data model prior to the design of the imperative language. The formalisation has also directed the implementation by yielding a logic language conventional enough to be implemented by standard optimisation and evaluation strategies, and by facilitating the development of a compiler and imperative language interpreter that directly support the formally specified behaviour.

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 EXODUS implementations of the object manager and logic languages have been carried out by Alia Abdelmoty and Andrew Dinn, respectively. We would also like to thank Prof. 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] A.I. Abdelmoty, N.W. Paton, M.H. Williams, A.A.A. Fernandes, M.L. Barja, and A. Dinn. Geographic Data Handling in a Deductive Object-Oriented Database. In D. Karagiannis, editor, *Proc. 5th Int. Conf. on Databases and Expert Systems Applications (DEXA)*, pages 445–454. Springer-Verlag, 1994.

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

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

[4] A. Albano. Conceptual languages: A comparison of ADAPLEX and Galileo and taxis. *Foundations of Knowledge Base Management*, pages 395–408, 1989.

[5] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 163–179. Springer-Verlag, 1994.

[6] Maria Luisa 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, December 1992.

[7] 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. *Information Systems*, 20:185–211, 1995.

[8] M.L. Barja, N.W. Paton, A.A.A. Fernandes, M.H. Williams, and A. Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 463–474. Morgan-Kaufmann, 1994.

[9] M.L. Barja, N.W. Paton, and M.H. Williams. Semantics Based Implementation of a Deductive Object-Oriented Database Programming Language. *J. Programming Languages*, 2(2):93–108, 1994.

[10] Michael L. Brodie. Axiomatic Definitions for Data Model Semantics. *Information Systems*, 7(2):183–197, 1982.

[11] Filippo Cacace, Stefano Ceri, Letizia Tanca, and Stefano Crespi-Reghizzi. Designing and Prototyping Data-Intensive Applications in the Logres and Algres Programming Environment. *IEEE Transactions on Software Engineering*, 18(6):534–546, June 1992.

[12] J. Campin, N.W. Paton, and M.H. Williams. A Structured Specification of an Active Database System. *Information and Software Technology*, 37(1):47–61, 1995.

[13] 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.

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

[15] S.K. Das and M.H. Williams. A path finding method for constraint checking in deductive databases. *Data & Knowledge Engineering*, 4:223–244, 1989.

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

[17] A. Dinn, N.W. Paton, M.H. Williams, A.A.A. Fernandes, and M.L. Barja. The Implementation of a Deductive Query Language Over an Object-Oriented Database. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Proc. 4th Intl. Conf. on Deductive Object-Oriented Databases*, pages 143–160. Springer-Verlag, 1995.

[18] 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.

[19] 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.

[20] 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, *Proc. 1st Int. Workshop on Rules in Database Systems*, pages 234–250. Springer-Verlag, 1994.

[21] Peter M.D. Gray. *Logic, Algebra and Databases*. Ellis Horwood, 1984.

[22] Matthew Hennesy. *The Semantics of Programming Languages*. John Wiley and sons, 1990.

[23] M. Jarke, R. Gallersdorfer, M.A. Jusfeld, and M. Staudt. ConceptBase – A Deductive Object Base for Meta Data Management. *J. Intelligent Information Systems*, 3:167–192, 1994.

[24] 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 International Conference on the Management of Data*, pages 134–146. ACM Press, 1989. A revised and extended version exists as [26].

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

[26] 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.

[27] Robert Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series (The Computer Science Library). Elsevier North Holland Inc., New York, NY, 1979. ISBN 0-444-00365-7.

[28] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an Object-Oriented Data Model. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL, 1988. ACM Press. Also in [38], pages 227-236.

[29] Robert Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[30] J. Murphy and J. Grimson. Formal Specification of a Persistent Object Management System. *Information and Software Technology*, 35(5):277–286, 1993.

[31] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Dept. of Computer Science, Aarhus, Denmark, 1981.

[32] A. Poulovassilis and C. Small. A Domain-theoretic Approach to Integrating Functional and Logic Database Languages. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. 19th Very Large Data Bases Conf.*, pages 416–428. Morgan Kaufmann, 1993.

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

[34] J. Richardson and M. Carey. Implementing Persistence in E. In J. Rosanberg and D. Koch, editors, *Persistent Object Systems*, pages 175–199. Springer-Verlag, 1989.

[35] J.M. Smith and P.Y.T. Chang. Optimising the performance of a Relational Algebra Database Interface. *Comm. ACM*, 18:568–578, 1975.

[36] J. Widom. A Denotational Senmantics for the Starburst Production Rule Language. *ACM SIGMOD Record*, 21(3):4–9, 1992.

[37] M.H. Williams and Q. Kong. Incomplete Information in Deductive Databases. *Data & Knowledge Engineering*, 3(3):197–220, 1988.

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