DESIGN AND IMPLEMENTATION OF ROCK & ROLL: A DEDUCTIVE OBJECT-ORIENTED DATABASE SYSTEM

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

Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh EH14 4AS, UK

January 10, 1995

Abstract — This paper presents an approach to the development of a deductive object-oriented database system, describing the key design decisions and their consequences for implementation. The approach is novel, in that it integrates an object-oriented database system manipulated using an imperative programming language (ROCK) with a logic language for expressing queries and methods (ROLL). The integration is made seamless by deriving both the imperative and logic languages from a single formally defined data model, thereby avoiding impedance mismatches when they are integrated.

Key words: Deductive Object-Oriented Databases, Deductive Databases, Object-Oriented Databases, Database Programming, Logic Programming.

1. INTRODUCTION

The two most prominent of the emerging approaches to database programming build upon the deductive and the object-oriented paradigms. Both of these approaches have significant strengths. Deductive databases (DDBs) have a formal basis in first order logic, support expressive declarative querying, and benefit from substantial experience with query optimisation strategies. Object-oriented databases (OODBs) support rich facilities for representing both structural and behavioural information, are normally associated with computationally complete programming languages, and are suitable for use in a range of applications where relational database systems have been found to be inappropriate. However, both of these categories of system also have weaknesses. Deductive databases lack expressive data structuring mechanisms, and normally support limited facilities for update or I/O. Object-oriented database systems are rarely based upon formal semantic models, and often lack declarative query languages. It would thus seem to be the case that deductive databases and object-oriented databases have complementary areas of strength and weakness, and that any system which can combine the two paradigms in a way which uses their respective strengths to overcome their respective weaknesses would be an effective platform for a range of novel and conventional applications. Such an endeavour has been recognised as worthwhile [2, 43, 27], but is not straightforward without sacrificing certain of the characteristic strengths of either category of system.

This paper presents the design and implementation of a deductive object-oriented database (DOOD) which is built upon a formally defined data model described in section 3.1. This model has been formally specified as a set of axiomatic first-order theories, from which has been derived a corresponding set of Horn clauses. The object manager described in section 3.2 implements this data model, although the rule-based formalism is in fact implemented as a set of persistent C++ classes. Two languages have been derived from this model, an imperative database programming language described in section 4, and a logic query language described in section 5. The imperative programming language is used to provide a syntax for defining object classes, for writing application code as methods or free standing programs, and as the sole means of changing the state of the database. The query language is a conventional first-order logic language which can be used to define rules and to express queries over extensional databases which conform to the axioms that constrain valid database states. The semantics of the logic language stem from a syntactic mapping onto function-free Horn clause programs presented in [23]. The integration of the logic and imperative languages is presented in section 6, where it is shown how, and with what restrictions, each language can call the other. Conclusions are presented in section 7.

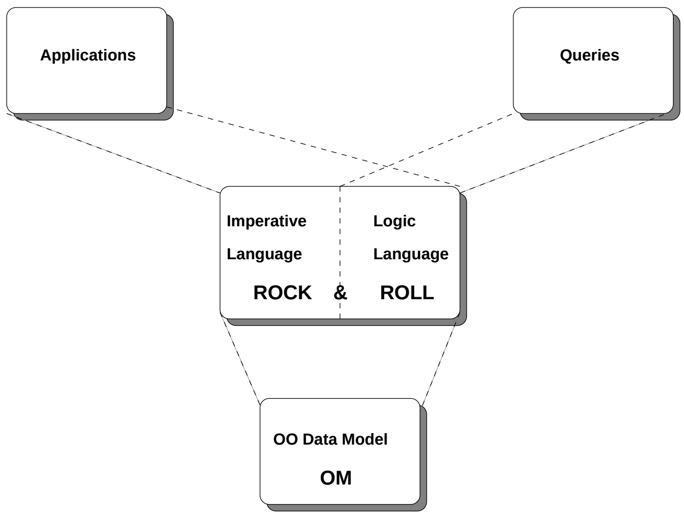

Fig. 1: Relationship between the principal components in the architecture.

The object manager and imperative programming language are referred to as OM and ROCK (Rule Object Computation Kernel) respectively. The logic language is known as ROLL (Rule Object Logic Language). The relationship between the principal components is depicted diagrammatically in figure 1. This architecture is designed to support the synergy of the different components: the logic language can be used to support those parts of an application which are suitable for declarative rule-based expression; the imperative language can be used to support updates, input/output, and the expression of algorithms normally written in a procedural manner; both languages operate in the context of an expressive object-oriented data model which has facilities comparable to those supported by a number of semantic data models.

A significant strength of the overall system is the conventionality of its individual components – it has not proved to be necessary to alter the defining principles of the component paradigms or to introduce any complex new concepts in order to achieve their integration. Thus the approach adopted can be seen as a judicious blending of existing OODB, DDB and database programming language technologies, to yield a system which provides effective support for advanced applications. Novelty is thus more manifest in the methodology adopted than in the individual components which have been implemented, although new results are presented regarding such issues as optimisation, type inference and language integration.

The applicability of the ROCK & ROLL system has been evaluated in the context of a geographic database system application [1]. In this domain, the comprehensive modelling facilities of OM have been used to support the structuring of complex geographic phenomena, the imperative language ROCK has been used for expressing geometric algorithms and for populating the database, and the logic language ROLL has been used to express derived relationships between extensionally defined concepts. The absence or dilution of any one of these components would have significantly impacted on the effectiveness with which this application could be supported.

To give a flavour of the overall approach, a definition of the type `segment` is presented in figure 2. The `type` definition indicates that `segment` is an aggregate of two `point`s and that it has two public methods `print` and `follows` which are implemented in ROCK and ROLL respectively. The `class` definition gives code for the methods `print` and `follows` – `print` invokes the `print` method of the two `point` objects from which the `segment` is constructed, and `follows` indicates that one segment follows another if the second point of one is the first point of the other. Both methods are statically type checked, can be inherited or overridden, and can be invoked from the same program or interactively. The integration of the two languages is discussed in detail in section 6.

```
type segment
    <point1:point, point2:point>
    interface:
        ROCK: print();
        ROLL: follows(segment);
end-type;

class segment
    public:
        print()
        begin
            print@get_point1; write "-->"; print@get_point2; write nl;
        end

        follows(segment)
        begin
            follows(OtherSegment)@ThisSegment :-
                get_point2@ThisSegment == get_point1@OtherSegment;
        end
end-class
```

Fig. 2: Type and class definition for `segment`.

The platform which has been used to implement ROCK & ROLL is the EXODUS extensible database system [13], and in particular the programming language E [42]. E is essentially a persistent C++ system, and has been chosen because it directly supports the linked storage of persistent objects, database indexing and transactions, thereby removing the need for such facilities to be implemented from scratch for ROCK & ROLL.

2. CONTEXT

2.1. Related Work

Research into deductive object-oriented databases (DOODs) has been conducted using a range of different strategies. The most radical departure from earlier work on deductive databases is characterised by F-Logic [28], where a new logic language is proposed incorporating such object-oriented features as inheritance and a notion of identity. The proposal is technically quite complex, and it is far from clear how it could be implemented and taken as the basis of a practical database system. For example, not only the language does not have all the functionality required for application development, there is also no mention of a strategy, such as embedding or the use of external functions, for achieving that functionality. In comparison with that of Datalog (and hence with ROLL), the metatheory of F-Logic is significantly more complex to grasp. More importantly, many desirable properties of Datalog are lost. For example, the low complexity of the unification algorithm is threatened by the presence of set terms. Also, the feasibility of a theorem-proving approach is reduced because inheritance is modelled by inference rules, which may adversely impact on the efficiency of the proof procedure by introducing multiple choices as to which rule to apply at each inference step. Finally, F-Logic is only structurally object-oriented. There is no attempt to tackle the behavioural aspects that make object-orientation a distinctive paradigm.

A less radical approach is to extend the semantics of Datalog with facilities associated with object-oriented databases (OODBs). This is the course taken in IQL+ [2] and Logres [11, 12]. This general approach has the advantage that it can directly reuse earlier work on the design and implementation of deductive relational databases, but tends to lead to systems with limited semantic data modelling facilities and logic language semantics complicated by the introduction of updates.

For example, IQL+ (i.e., the version of IQL in [2]) does not model sequences, does not cater for multiple inheritance and has no behavioural inheritance. Its predecessors, IQL [4] and COL [3] are more restricted, the latter being classifiable as non-first-normal-form relational, rather than object-oriented. The latest version of IQL has only been given the fixpoint semantics of COL. Since the language makes use of functions to model sets and tuples, and hence departs from Datalog in a crucial way, it is not immediately clear what form the model-theoretic and procedural semantics of the language will take. Although the metatheory of this language family is much closer to that of Datalog than the metatheory of F-Logic, some loss of desirable properties can still be detected, the most important of which are the lack of guaranteed termination which results from the reintroduction of function symbols, and the lack of alternative equivalent semantics.

Another approach endeavours to reuse earlier work on deductive databases by developing a mapping from a DOOD language onto an existing deductive database system which is used to provide both a semantics for the mapped language and an implementation. Examples are the Bertino-Montesi proposal [10], OOLP+ [18], and OIL [46]. This is a practical route to the development of DOOD systems, but tends to be associated with similar drawbacks to the extended Datalog approach.

Of these three approaches, the work described here is most closely related to the mapping approach, in that the logic language has been formally defined by devising a mapping onto Datalog in the context of an axiom set which characterises the data model. However, in our approach the data model is a separately identifiable component supporting a wide range of semantic modelling constructs (see section 3.1), and the mapping is from a logic language which has not been complicated by the introduction of updates or control. Furthermore, broader applicability is obtained by the mismatch-free integration with an imperative programming language. This approach results in functionality that is at least as powerful as that provided by embedding of external-function calls, with none of the associated impedance mismatches.

The need for an update mechanism in the logic language has been obviated by using a separate language component for data manipulation. The approach of integrating different language components in a database context has been explored before. In Glue/Nail [38] it is shown how an imperative language Glue can be used to perform manipulation of the data stored in the deductive relational database Nail. Although this approach has been influential in our work, the research presented here is significant in that it brings together deductive, object-oriented and imperative approaches via a common theoretical model, has a more comprehensive imperative language and provides static type checking with type inference. More recently, Peplom[d] [19] has also taken this two-language-component approach.

Further examples of integration include PFL [40], in which a deductive query language is embedded in a lazy functional programming language, and Coral++ [43] in which C++ data structures are made accessible from the deductive database Coral. Our approach differs from PFL in the nature of the underlying data model and in the use of an imperative language alongside the deductive component. The principal difference between ROCK & ROLL and Coral++ stems from the fact that only in the former does a single underlying data model specify all the types that can be accessed by the logic language and manipulated by the imperative language. This uniformity is lacking in Coral++, where the types which can be manipulated by the logic language and the imperative language overlap, but are not identical. As such, Coral++ can be seen as a coupling of two language components, rather than an integration. Issues related to language integration arising in this context are discussed in detail in [7].

The DOOD system presented here builds directly on earlier work on deductive databases [14] and on database programming languages [6]. The logic language is conventional, in that it is both first order and function free, and furthermore has not been extended with facilities for updates or control. The database programming language is a conventional imperative persistent programming language, with both extensional and intensional information stored in the database. The integration of the two systems uses type inference to enable type-safe embedding, adapting earlier work on type inference [35] for use in the deductive object-oriented context. The logic language is optimised using an extension of static filtering [29], in which this technique has been revised for use in the context of a richer data model. By building upon such a wide range of earlier research activities, it has been possible to exploit prior theoretical and practical experience to yield a system which is both pragmatic and formally sound. A more extensive review of approaches to the design of DOODs is given in [22].

2.2. ROCK & ROLL as a DOOD

ROCK & ROLL embodies a number of design decisions specifically meant to preempt certain semantic hurdles that have slowed progress in achieving an extension of the deductive paradigm with object-oriented notions. The two most important of those decisions are:

1. To anchor ROCK & ROLL on an object-oriented model of structure *and* behaviour that underpins, but is wholly independent of, its sublanguages. This decision guarantees that multiple language components are inherently amenable to mismatch-free integration.
2. To formalise the model as a system of first-order logic. This decision guarantees that a clear path exists with which to instill object-oriented notions into a logical query language.

Other pragmatic decisions are also essential in order to make ROCK & ROLL a realistic option for advanced database applications, among which: the definition and manipulation sublanguages are serviced by a storage layer tightly coupled to them in the context of orthogonal persistence; all manipulation (including logical querying) is statically type-checked, and a type-inference mechanism takes most of the burden associated with this functionality; programmers retain as much control as they want and if all control is delegated, i.e. if code is purely declarative, optimisation is not hindered by impure features.

These decisions made it possible both to steer the implementation effort away from trouble spots and to cover more ground than most comparable research projects. This implementation effort has been concluded very recently and is being evaluated using an application in the area of geographical information systems [1]. It is already possible to report that the resulting system, besides being endowed with comprehensive and well-understood theoretical foundations, exhibits more functionality that many alternative proposals, as follows:

- ROCK & ROLL provides a full object-oriented model of structural knowledge. This includes rich mechanisms for the definition of collection types and complex types whose values possess an identity distinct from their state. Thus, ROCK & ROLL offers more comprehensive modelling facilities than deductive languages using complex values, based on non-first normal-form relations such as HILOG [15] and Nested Datalog [16], or based on constructor functions such as COL [3], C-Logic [17] or O-Logic [30].
- ROCK offers full support for explicit object creation, liberating ROLL from the need to become involved in the special semantics of object creation that have to be tackled in LIVING IN A LATTICE [25], O-Logic, and OIL [46].
- By axiomatizing their semantics, both collection types and complex types are handled within the logic without the complications of grouping constructs, as present in $\mathcal{LDL}$ [37], and without relinquishing finiteness of interpretation domains caused by the introduction of function symbols to denote constructors, a problem faced by both F-Logic (and predecessors) and by IQL+ (and predecessors).
- ROCK & ROLL provides a full object-oriented model of behavioural knowledge, in which there is a clear distinction between the interface to and the implementation of a type. Few proposals tackle this problem, of which ConceptBase [26] and LLO [33] are well-known exemplars. This is a necessary feature for a system to be classified as behaviourally object-oriented, and has origins in the notion of abstract data types. In a DOOD context, it implies devising ways of distinguishing within the database theory the declaration that instances of a type will respond to certain operations and the implementation, by deductive rules or otherwise, of

those operations. This gives rise to the notion of encapsulation, and is a significant departure from deductive databases over a relational data model.

- ROCK & ROLL supports overriding and dynamic (or late) binding of methods. Any method can be implemented using ROCK. However, if a method is dominated by data retrieval, it will probably be best implemented in ROLL, insofar as it would undergo optimisation (once, at compile time) with a potential gain in efficiency that would be harder to achieve using ROCK. A distinctive feature of ROCK & ROLL is that a method implemented in ROLL abides by encapsulation, and may be overridden and late-bound just as if it had been written in ROCK. Attempts to model one or more of encapsulation, overriding and dynamic binding in F-Logic and in IQL+ are not as intuitive and not as integrated as in ROCK & ROLL.

- Most proposals for DOODs ignore the provision of facilities for control resembling those of imperative programming languages. Exceptions to this are IQL+ and Logres. ROCK also provides a comprehensive set of control facilities. ROLL does not lose declarativeness, and the evaluation of ROLL-coded methods is not disturbed by the fact that ROCK evaluation is under control of the programmer. In this way, maximal opportunities are preserved both with respect to optimisation and to freedom in the choice of whether to evaluate an invocation bottom-up or top-down

- With respect to effecting state transitions, a well-known idea used both in the various languages defined by [5] and in Logres is to introduce transition-effecting annotations in the heads of rules. However, handling state transition directly in the rules makes them non-declarative. This precludes many optimisation opportunities and incurs the penalty of possibly non-terminating programs. In ROCK & ROLL, state transitions are performed with full imperative control; retrieval of data in a single state can be done imperatively or declaratively, at the discretion of programmers.

- ROCK & ROLL provides full structural and behavioural inheritance (possibly multiple) for subtyping and subclassing, allowing, on the one hand, the specification of constraints on the inclusion relationship between sub- and superclass, and, on the other, catering for overriding of overloaded methods through late-binding. This is more expressive than any other DOOD proposal, and has again been achieved through axiomatization, at practically no additional cost in terms of a more complex semantics (as, for a contrast, is the case with F-Logic).

- ROCK & ROLL provides static type-checking and SML-like type inferencing [35] for *both* language components in *both directions* of embedding – type inferences flow from ROCK to ROLL, and vice-versa. No other proposal for DOODs (compare, e.g. IQL+ and Coral++) is nearly as flexible or comprehensive in its treatment of typing.

- There is no type-system or evaluation-strategy impedance mismatch in ROCK & ROLL: every ROCK defined type can be used in a ROLL query, and vice-versa; also, the result of evaluating a method call in ROCK can be passed to a ROLL query, and vice-versa. In both cases no intermediate type-conversion is required of the programmer. No other DOOD proposal offers this functionality. Some provide no foreign-language interface (e.g. F-Logic), some (e.g. IQL+) provide hooks that may incur impedance mismatches, yet others (e.g. ConceptBase and Coral++) provide some smoothing out of conflicts, but only unidirectionally.

- Finally, ROCK & ROLL is fully implemented as a persistent-programming environment. Only two other DOOD proposals are known to us to be implemented, viz. ConceptBase and Coral++.

These contrasts demonstrate that ROCK & ROLL contributes a strategy for paradigm integration that is both more general and encompasses more functionality from each paradigm than proposed alternatives. In particular, it shows that trying to handle too much on one arm of the scale may be self-defeating, and that novel ways of using proven techniques can do much better than one would at first expect.

Methodologically speaking, ROCK & ROLL achieves its result by synergy. Its component parts in isolation are less daring than their counterparts in other proposals, but the whole is revealed to be harmonious and, most definitely, greater than its parts.

3. THE DATA MODEL (OM)

3.1. Design of the Data Model

This section gives an informal overview of the concepts used to model an application domain both structurally and behaviourally. The *object-oriented model* informally described below has been formalised in [23] as a class of first-order theories [34] called *object theories* (OTs). For each application domain D, an OT is an applied first-order theory whose language is fully determined by the constant symbols used to denote entities of interest in D. Predicate symbols are fixed for all OTs and denote semantic-modelling relationships as well as unacceptable configurations of declared facts. An OT A comprises three subsets: a set of *ground axioms* $A(D)$ that represent D at both schema- and instance-level, a set of *modelling axioms* M that characterise implicit information deducible from $A(D)$, and a set of *exception axioms* E that characterise unacceptable configurations within $A(D)$. An *object-oriented database* (OODB) A' can then be formally defined as an interpretation of an OT $A = A(D) \cup M \cup E$ such that all axioms in M are true in A' and all axioms in E are false in A'. Axioms in $A(D)$ are assumed to be true by definition.

As an example, the following axioms specify the origin and transitivity of the *is_a* relationship:

$$(\forall xyz)(specialization\_of(x,y) \vee generalization\_of(y,x) \vee partitioned\_by(y,x) \Rightarrow is\_a(x,y))$$

$$(\forall xyz)(is\_a(x,y) \wedge is\_a(y,z) \Rightarrow is\_a(x,z))$$

As described in detail in [23], any OT can be recast as a Horn theory [24], thus extending conventional OODBs with a logic-programming style of deduction that has the classical declarative, operational and fixpoint semantics of [21]. An informal description of the formalism is now given.

The model distinguishes between *primitive types* and *object types*. There is a fixed set of primitive types supported by the system, namely `integer`, `real`, `string` and `boolean`. An instance of a primitive type is called a *primary object*, and is denoted by its value.

An *object type* is defined by the user to model some aspect of an application domain. Examples of such types in a geographic information system are `landParcel`, `road` and `polygon`. An *object type* may be defined in one of two ways. Firstly, and somewhat trivially, an object type may be defined as taking primary objects as its instances, in which case the object type acts as an application-specific synonym for the primitive type. Secondly, and more interestingly, an object type may be used as a conceptual model of some part of the application domain. An instance of an *object type* is called a *secondary object*, and is named by a unique *object identifier* (oid). A secondary object is simply called an *object* in the rest of this section.

The *state* of an object comprises *references* of up to three kinds. First, and foremost, an object refers to, or is an instance of, one or more object types[†]. Second, depending on the structure determined by the types of which it is an instance, an object refers to objects, secondary or not, as values of its properties. Third, again depending on its types, an object may refer to objects, secondary or not, as its construction elements. As an example, consider an object named by the oid `!1`. It is assigned to object type `road`. As its `roadName` property it refers to `"M8"`. Finally, as its component elements it refers to `roadSegments !5`, `!12`, and `!3`, where the nature of the relationship with its components is described using the structural abstraction mechanisms described below.

The **basic abstraction mechanisms** available to model real-world entities in the application domain are:

- *Classification*, which declares that an object is an instance of some object type.

[†]An object is *directly* assigned a single type when it is created, and is associated with additional types by inheritance.

- *Specialisation*, *generalisation*, and *partition*, which are used to place object types into the abstract structure known as the *is-a hierarchy*. Specialisation and generalisation differ only in that specialisation places an object type in the hierarchy in terms of previously-defined supertype(s) whereas generalisation does so in terms of previously-defined subtype(s). Partition is a special case of generalisation with the additional constraint that the extensions of the subtypes are disjoint. Multiple inheritance from more than one supertype is also supported.

The **structural abstraction-mechanisms** available to model real-word entities in the application domain have the following characteristics:

- *Attribution* declares, for an object type t, which object types name properties of t. In other words, if t has a property t', the value of t' in the state of t is an instance of t'. For example, assume that `roadName` is a property of `road` and that the object identified by `!5` is an instance of `road`, then `!5` contains in its state a reference to the primary object identified by `"M8"`, as the value of its property `roadName`.

- *Association*, *sequentiation*, and *aggregation* determine how an instance of an object type is constructed from instances of other object types. An object type is said to be constructed from another when an instance of the former will contain in its state a number of construction references to instances of the latter. The three abstraction mechanisms are:

  - *association*, in which the constructed object references an unordered set of component objects (e.g. a `road` is constructed as an association of `roadSegments`).
  - *sequentiation*, in which the constructed object references an ordered collection of component objects indexed by the natural numbers (e.g. a `polygon` is constructed as a sequence of `Segments`).
  - *aggregation*, in which the constructed object references one or more component objects by the set of types which are the coordinates of the aggregation. As a result, aggregations have the same properties as records in programming languages (e.g. a `Segment` is constructed as an aggregation of two `Points`).

  The separation of the construction of an object from its attributes makes explicit a distinction between the fundamental characteristic of an object and its other structural features.

The single **behavioural abstraction-mechanism** available to model real-word entities in the application domain is called an *operation*. An operation has two detached components: an *interface* and an *implementation*, the latter expressed as code in ROCK or ROLL. An operation interface defines that an *operation name* abides by an *operation signature* in the context of an object type. This behavioural model allows for the *overloading* of operation names with respect to interfaces and implementations. The detachment of interfaces and implementations induces the notion of *object class*. To every object type there corresponds one and only one object class with identical name. An object class declares a single *operation implementation* or *method* for every operation interface declared in the object type corresponding to the class. Methods are defined in a framework which supports *overriding*, and *late-binding* is used to select the most specialised definition which is applicable to the message recipient. A most specific implementation is guaranteed to exist and to be unique by a well-formedness constraint on the declaration of object types. Concrete examples of type and class definitions are given in section 4.

Schema diagrams for the data model can be constructed using the following notation. Secondary object types are represented using rectangles, and primary object types using ellipses. Labelled directed edges represent modelling features thus: $\bigcirc$ – attribution, ⧇ – generalisation, ◈ – partition, △ – specialisation, $\otimes$ – aggregation, $\circledast$ – association, and $\circledcirc$ – sequentiation. An example which uses this notation to describe part of a geographic database is given in figure 3.

3.2. Implementation of the Data Model

The data model is supported in E by two principal class hierarchies which describe OM types and instances respectively. For example, the main E classes used to describe the intension of a OM

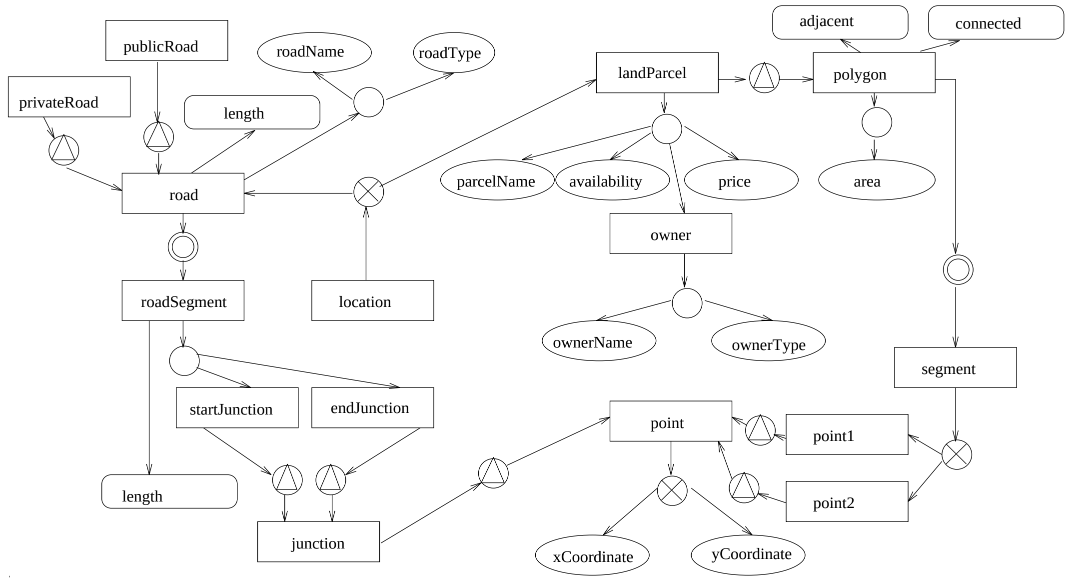

Fig. 3: Schema diagram for a fragment of a geographic database

database are presented in figure 4. The E class m_Type has attributes which store the name of the OM type, descriptions of its attributes, references to supertypes/subtypes, a list of instances, and a collection of indexes into the set of instances. There is also a collection of member functions for creating OM types, adding attributes, deleting indexes, etc.

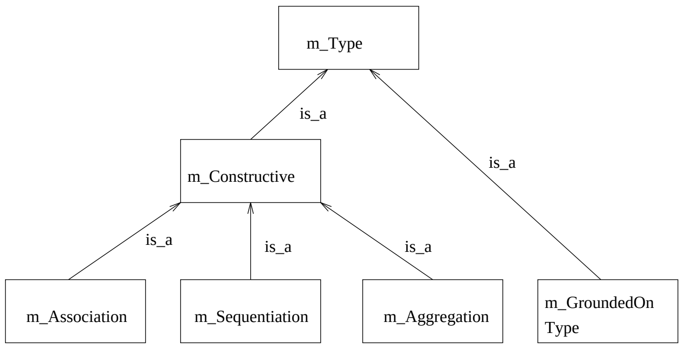

Fig. 4: E classes for describing OM types

Instances of OM types are described using a separate hierarchy of E classes which shadows the hierarchy in figure 4. The E class m_Instance has attributes which reference its m_Type, its attribute values, and the m_Instance objects which describe the OM object in different roles (for example, a `privateRoad` object which is also a `road` (because `privateRoad` is-a `road`) would be represented by two m_Instance objects).

To give a flavour of the interface to the object manager, the following E code fragment iterates over the instances of the type pointed to by T:

```
m_Type *T;
m_Instance *I;
```

```
...
m_Instance_Iter iterator = T->get_all_instances();
while (I = iterator.next()) {
    // operations on I
    ...
}
```

In the above, T is a reference to an OM object type definition, and I is a reference to an OM instance object. The E object iterator is an instance of the E class m_Instance_Iter, which enables iteration through the extension of a type, in this case T. In the while loop test, I is assigned a new instance of T in each iteration, as a result of the invocation of the next operation on iterator.

It can be seen from the above that the object model is essentially implemented as a collection of abstract data types which support primitive operations on OM types and instances. An example of how the object manager is invoked from the ROCK interpreter is presented in section 4.2.

4. ROCK

4.1. Design of ROCK

The database programming language provides an environment in which both persistent and transient data are created and manipulated in a uniform way. By integrating ROCK with the logic language ROLL described in section 5, this programming environment is further enriched with declarative querying and method definition facilities, as described in section 6. ROCK is based on the data model described in section 3, and is an imperative object-oriented programming language. It supports strong compile time type checking which offers advantages such as software integrity, consistency and efficiency.

ROCK can be regarded as the conjunction of a data definition language for schema declarations, and a data manipulation language that allows operations to be performed on the objects stored in the database. The types which can be defined using the data definition language are exactly those which are supported by the data model described in section 3.1. The data manipulation language provides powerful constructs for processing such data, and as such is the vehicle for the development of application code through support for the creation, manipulation and explicit deletion of objects. The facilities available for data manipulation include:

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

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

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

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

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

- System-generated operations

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

- User defined operations

  User defined operations are given in the form of methods. Methods have a visibility which is either *public* or *private*. Methods that are *private* to a class can only be invoked from other methods attached to that class. The signatures of public methods are defined in the interface part of the corresponding type definition.

To illustrate class and method definition as well as the use of built-in operations, consider the definition of classes `roadSegment` and `road`, whose structures are as defined by the types depicted in the schema diagram in figure 3. Both classes define a public method `length` which computes the length of the `roadSegment` and the length of the `road` (i.e. a set of `roadSegments`), as shown in figure 5.

4.1.1. Semantics

The semantics of ROCK is fully defined in [8] using *Structural Operational Semantics* [39, 45]. Structural operational semantics describes how each individual step of a computation takes place by defining a transition system whose steps describe the evaluation of programs. Inference rules are used to describe the axioms of a transition system, and propagation of information is achieved by pattern matching.

This formalism is used both for describing the *static semantics* of the language, as well as the *dynamic semantics* by syntax directed rules. The static semantics of the language is given by means of a set of *type rules*. These rules are used to derive semantic properties of phrases in the language, and form the basis of a type inferencing mechanism. They define how to check the type of a phrase as a result of type checking its components, as well as the compatibility of types. 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.

4.1.2. Model of Persistence

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

4.2. Implementation of ROCK

The creation, modification and retrieval of data is achieved by the ROCK interpreter invoking the object manager described in section 3.2.

Both the type checker and interpreter are structured in an object-oriented fashion. The implementation is based on a direct correspondence between classes in E and the syntactic constructs of the language [9]. Thus, the class hierarchy defined in E mirrors the relationship between syntactic categories defined for the language, where each E class defines its own methods for type checking and evaluation. In this way, each node of the tree can type check and evaluate itself – syntax trees

```
type roadSegment
    properties:
        startJunction, endJunction;
    interface:
        ROCK: length(): real;
        ...
end-type;

type road
    properties:
        roadName, roadType;
    [ roadSegment ];
    interface:
        ROCK: length(): real;
        ...
end-type;

class roadSegment
    length(): real
    begin
        var startj := get_StartJunction@self;
        var endj := get_EndJunction@self;
        var x_1 := get_xCoordinate@startj;
        var y_1 := get_yCoordinate@startj;
        var x_2 := get_xCoordinate@endj;
        var y_2 := get_yCoordinate@endj;
        sqrt((x_1 - x_2)**2 + (y_1 - y_2)**2)
    end
end-class

class road
    length(): real
    begin
        var len: real
        len := 0;
        foreach r in self do
            len := len + length@r;
        len
    end
end-class
```

Fig. 5: Type and class definitions showing method implementation.

are traversed in an object-oriented fashion, by sending a message to the root which in turn sends messages to its children and so on. For example, the `foreach` statement in ROCK can be used to iterate over the instances of a class thus:

```
foreach <iter> in <class> do <statement>
```

The `foreach` statement is represented as a node in the syntax tree, which is used to represent the structure of a program internally. This tree is modelled as a hierarchy of E classes, as depicted graphically in figure 6. Member functions associated with each node of the tree are then used to support type checking and evaluation of statements. For example, in figure 7, the member function TypeCheck() verifies that the class to be iterated over is in scope, and if so it records that the variable iter is of class classid. The type checking process then continues through the invocation of the TypeCheck member function on the statement.

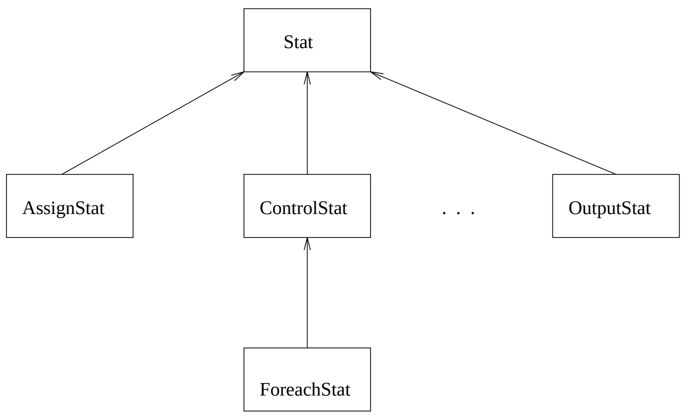

Fig. 6: Class hierarchy for representing programs internally.

The iteration process facilitated by `foreach` is implemented by the Eval member function also shown in figure 7. This member function calls the object manager to get_all_instances of the class classid, via an iterator called theinsts. The statement is then evaluated for every object instance in the `class`.

This implementation strategy has allowed a close correspondence to be maintained between the formal specification of the language and its implementation as object classes, and has the added advantage of facilitating the addition and removal of language constructs.

5. ROLL

ROLL is the logic language of the ROCK & ROLL DOOD system. As such, ROLL is not intended to be used in isolation for supporting complete data intensive applications. As a result, a comparison of ROLL with other deductive database languages could lead to the conclusion that ROLL is somewhat spartan. This is because ROCK is the sole vehicle for performing certain tasks in ROCK & ROLL, whereas in single-language systems (such as $\mathcal{LDL}$ [37] or Coral[41]) the logic language must support all the requirements of an application. In ROCK & ROLL certain tasks can be performed by both ROCK and ROLL, but the area of overlap is restricted to the retrieval of data from the database. ROLL contains no facilities for building new objects, changing existing objects, or expressing control. The advantage of this approach is that the semantics of ROLL is straightforward compared with certain other deductive database languages, which makes ROLL easier to learn and implement. ROLL could certainly be extended with additional features by following the example set in earlier deductive relational database systems, but this would serve only to extend the range of functionalities supported by both ROCK and ROLL. It is possible that,

```
class ForeachStat: public ControlStat {
    IterIdent iter;
    ClassId classid;
    Stat *statement;
    ...
public:
    void TypeCheck();
    void Eval();
    ...
}

void ForeachStat :: TypeCheck() {
    ...
    if ((ClassesStack -> Find(classid))
        ObjNameStack -> Push(iter, classid);
    else TError(this, "Undeclared class");

    statement -> TypeCheck();
    ...
}

void ForeachStat :: Eval() {
    ...
    m_Instance* instance;
    m_AttVal* obj = ValuesStack -> Find(iter);
    m_Instance_Iter theinsts = Get_m_Type(classid) -> get_all_instances();
    while (instance = theinsts.next())
    {
        obj -> PutValue(instance);
        statement -> Eval();
    }
    ...
}
```

Fig. 7: Example type checking and evaluation member functions for ROCK

in future, certain extensions will be made to ROLL which reduce its dependence upon ROCK for complex applications, but decisions as to which extensions, if any, are most appropriate await further experimentation with the present system.

5.1. Design of ROLL

ROLL is a Horn clause language. Familiarity with the latter subclass of first-order languages is assumed at the level of [14].

The domain-dependent subset of the *ROLL alphabet* consists of constant symbols and predicate symbols. The ROLL alphabet is function-free. The set of *constant symbols* is the set of values of primitive types (e.g. `5`, `"Edinburgh"`, `true`). Note that there is no need for object identifiers to appear as constant symbols in ROLL expressions, as specific objects are either passed into a ROLL expression from ROCK, or are retrieved from the extensional database.

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

A ROLL *term* τ is either a ROLL *constant* or a *(logical) variable*. A ROLL *atom* has one of the following forms:

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

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

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

The second form is used when β is implemented in ROCK, where the (optional) result is explicitly distinguished from any input parameters. An operation interface $\beta : T_1 \times \ldots \times T_{n-1} \rightarrow T_n$ is assumed to exist, such that each τ_i denotes an instance of T_i', $T_i' \leq T_i$. In this case, the result of evaluating the ROCK method β with the given parameters is unified with τ_n. If $n = 0$ then $\beta@\alpha =_{def} \beta()@\alpha$. The above forms exist to support the integration of ROCK & ROLL, as discussed further in section 6.

A ROLL atom can be negated, using the metalogical symbol '`~`', hence negation is allowed, either in queries or in rule bodies. The semantics for negation that is implemented in the ROCK & ROLL system is stratified semantics. Alternatives such as those discussed in [44], e.g. well-founded models, stable models, modularly stratified semantics, are also admissible in the context of ROLL but, for the purposes of implementation, are either plagued by anomalies or cannot be computed in acceptable time (for details on both these problems, see [44]).

A *ROLL clause* is a disjunction of literals of which at most one is positive. A clause containing a single literal is called a *unit clause*. The notion of a *ROLL fact* (a positive unit clause) though well-defined [23], is not relevant in the context of the ROCK & ROLL system because the asserted facts that describe the state of an object at any point in time are encapsulated and are only made available in computations as responses to message-sends. A *ROLL rule* is a clause with exactly one positive literal and with at least one negative literal. The positive literal is referred to as the *head*, the set of negative literals is referred to as the *body*. Finally, a *ROLL query* is a clause containing negative literals only. The usual convention is followed of rewriting a clause as a reverse implication, i.e. *head* '`:-`' *body*, and replacing disjunctions by commas.

5.1.1. Example Query

The following is an example of a ROLL query which retrieves as bindings for `L` all available `landParcel` objects connected to the particular `landParcel` represented by `!y`[†]. The operator `==` denotes unification.

```
connected(!y)@L, get_availability@L == "yes"
```

5.1.2. Example Rules

The following example rules implement the `connected` relation referred to in the above query. The rule for `adjacent` defines a binary relation between `polygons`, while `connected` defines the transitive closure of this adjacency relation.

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

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

In practice, although the notion of a *ROLL program* as a finite set of ROLL clauses, is well-defined [23], it is not relevant in the context of the ROCK & ROLL system because, in the integrated language, programs are defined in the context of classes as methods, using the syntactic form described in section 6.2.1.

5.2. Implementation of ROLL

Although ROLL operates over a more expressive data model than conventional deductive database languages, it is based on first-order Horn clauses. This allows the implementation of ROLL to build upon earlier work on query optimisation and evaluation in deductive relational databases, as surveyed in [14]. The presence of more comprehensive modelling constructs than in earlier deductive database languages impacts on the optimisation and evaluation processes thus:

Optimisation: Increased information is available on the types of values that can be stored in variables, and access paths to objects are navigational rather than value based.

Evaluation: Object-oriented behavioural features such as overriding impact upon the flow of information at runtime, and value-based joins are largely replaced by navigational exploration of the object base.

While different optimisation strategies (e.g. magic sets, query-subquery) could potentially be adapted for use with ROLL, the strategy adopted is based upon static filtering [29], adapted in a way that supports new kinds of constraints on logic variables. The evaluation strategy that has been implemented executes queries bottom-up using a graph structure which is essentially an object algebra.

As different evaluation strategies seem to be be most suitable for different types of query, we have also designed a top-down evaluation strategy based on an extended form of the Warren Abstract Machine (WAM) [20]. As this has not yet been implemented, it is not described further here.

[†]Note that the `!` indicates that a ROCK program variable `y` is supplied as input to the query which generates bindings for the logic variable `L`.

5.2.1. Bottom-Up Evaluation of ROLL

The principal evaluation strategy for ROLL is bottom-up, and enables the use of bulk operations which trawl the database or operate over highly interconnected sets of objects. It is based around a modified form of *Processing Tree* as defined by [32]. This is a form of query execution plan which represents query evaluation by means of dataflows. Processing Trees are constructed from *System Graphs* [29] which are an internal form used to represent a parsed query and the underlying ROLL rules which implement it.

An approach based on *Static Filtering* [29] is used to perform global optimisation of Processing Trees prior to execution. In Static filtering, constraints appearing within a query are propagated down through the Processing Tree. In certain circumstances a constraint appearing in a calling rule may be moved into the called rule, allowing it to be applied earlier. A typical example is where a binding from a query argument is propagated down the graph as a binding constraint. This allows the combinatorial explosion of goal solutions to be nipped in the bud.

After static filtering, a second stage of Processing Tree optimisation attempts to reorganise nodes within a given section of the tree. A heuristic search is used to consider alternative orderings for bulk operations in the processing tree. A cost estimate is used to compare alternative arrangements for each tree section and this cost is used to control the search process. This part of the optimisation process works from the bottom of the tree upwards, using the best cost estimate for subordinate tree sections as inputs to the cost function for the parent tree section.

The stages in the ROLL compilation process are as follows:

- Parsing of methods and queries.
- Type-checking of methods and queries.
- Construction of system graphs for each method and query.
- Verifying stratification of negated goals in system graphs.
- Construction of processing tree from each system graph.
- Global optimization of each processing tree.
- Local optimization of each processing tree.

5.2.2. System Graphs

A System Graph is a form of rule-call graph rooted at a query. Nodes in successive levels of the tree alternate between *goal nodes* and *constraint nodes*. In the context of RDBs, goal nodes correspond to intensional predicates or stored relations. Constraint nodes represent rules implementing intensional predicates; they are linked above to the goal they define and below to the goals they call; they also mention *constraints*, restrictions on goal solutions such as equalities or bindings of arguments.

ROLL System Graphs can use relationships defined by the object model either as constraints or as primitives which generate bindings. For example, an attribute relation between two goal arguments serves to constrain solutions. If only one argument is generated by another goal then the attribute constraint generates bindings for its other argument. If neither argument appears in other goals then a type generator goal is added, producing candidate bindings for one of the arguments.

The example query and rules in section 5.1 give rise to the system graph in figure 8. Ovals represent goal nodes and boxes constraint nodes. Where necessary, goal node outputs are labelled with aliases of the form ConstVar/GoalVar which relabel goal variables to the appropriate constraint variable. The type generator for *PolyA* at the bottom of the tree drives the *get_member* attribute constraint producing bindings for the other variables. The base clause for the recursive definition of *connected* does not constrain its inputs, hence the corresponding constraint predicate is *true*.

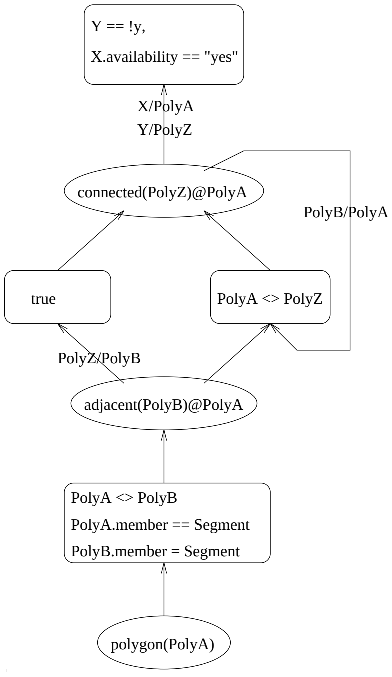

Fig. 8: Example system graph for ROLL rule.

5.2.3. Query Execution via Processing Trees

Processing Trees can be viewed as a query execution plan for a ROLL query. They are implemented using the E class *pt_Tree* which stores a linked set of Processing Tree Nodes. Each Node is an (indirect) instance of the abstract E class *pt_Node*. Subclasses of *pt_Node* represent particular database operations. The hierarchy of node classes is detailed in figure 9.

Query evaluation is based upon a dataflow execution model. *Binding tuples* which represent solutions to query goals flow upwards from the leaves of the processing tree to the root. Each tuple is an instance of the E class *Tuple*, a vector of references to OM objects. Nodes receive tuples from input nodes lower down the tree and combine them or filter them, as appropriate, passing on valid solution tuples to nodes further up the tree. An example processing tree is detailed in figure 10, implementing the system graph given in figure 8.

Leaf nodes in the processing tree are either *pt_EnumScans*, which generate tuples containing individual objects (query arguments or constants mentioned in the rules), *pt_TypeScans* which generate tuples containing all instances of a given type or *pt_NegTypeScans* which are like TypeScans but avoid generation of instances for types which specialise a rule (see also fixed feed nodes below). In the example tree the bottom left node is a *pt_TypeScan* producing singleton tuples containing instances of the class *polygon*.

Join nodes are used to combine solutions from subgoals. All joins are binary, combining pairs of input tuples to generate an output tuple. A cascade of joins is used to combine solutions from multiple goals. A join is either a *pt_XProduct* which merely concatenates its input tuples, or a *pt_EquiJoin* which also imposes an equality constraint on its input tuples' arguments. Join nodes keep copies of all input tuples so that they can remove duplicates, avoiding unnecessary propagation of repeated solutions. In the example tree a *pt_Equijoin* combines tuples from the *adjacent* and *connected* relations where the second element of the left tuple equals the first element of the right tuple.

pt_Constraint nodes filter their input stream removing tuples which fail to satisfy constraints expressed in the rules. The simplest constraints are implemented by *pt_Equal* and *pt_Alias* nodes,

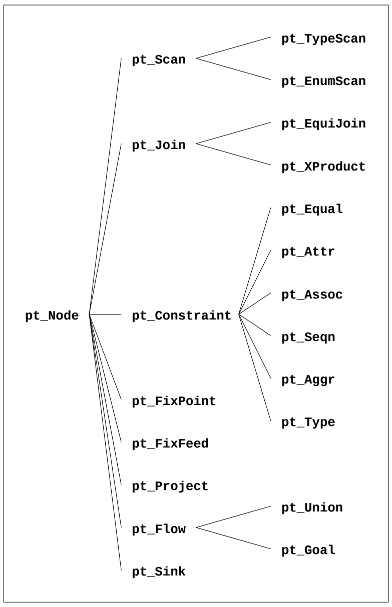

Fig. 9: E Class hierarchy for internal representation of processing trees.

which arise from unifications in rule bodies. They only propagate tuples which, respectively, have either a particular value for a given entry or the same value in two entries. In the example tree an assignment constraint at the top of the tree enforces the constraint that query solutions have a given value ($!y$) for the second argument. Other nodes, (e.g. *pt_Attr* nodes) ensure that tuple entries are related according to structure relations defined in the OM data model (e.g. that one is an attribute of the other). *pt_Type* nodes are used to remove tuples whose entries are not members of a subtype. This is necessary where a subgoal produces solutions of a generic type and it is known from type inference in the calling rule that the type of the goal argument is a subtype. Conversely, *pt_NegType* constraint nodes are required in the presence of overriding to filter out tuples containing instances of subtypes.

pt_FixPoint nodes are used to propagate solutions to recursive queries back down the graph for recursive combination. *pt_FixFeed* nodes occur as leaf nodes in place of recursive subgoals in the tree. They are fed with solutions as they arrive at the corresponding fixed point node. Fixed point nodes retain copies of all input tuples in order to avoid propagating repeat solutions to their feed nodes. This avoids unnecessary work and also allows the fixed point computation to be terminated when propagation of solutions to feed nodes results in no new solutions arriving at the fixed point node. In the example tree the *pt_FixPoint* node in the middle of the graph feeds new solutions to the *connected* relation back to the *pt_Fixfeed* node in the bottom right of the tree for recursive combination with solutions to *ancestor*.

pt_Project nodes project a subset of entries from their input tuples and pass these on as outputs. This allows intermediate references required for constraint checking to be dropped when they are no longer required. It has the added benefit of reducing the number of tuples stored in join and fix point nodes since solution tuples which differ before projection may be equal after projection. In the example graph project nodes occur above the tree sections corresponding to each of the original rules, projecting out tuple entries which correspond to the formal parameters of each rule.

pt_Union nodes are used to merge solutions arising from different rules. They merely propagate tuples arising from their inputs without processing them. *pt_Goal* nodes do not perform processing

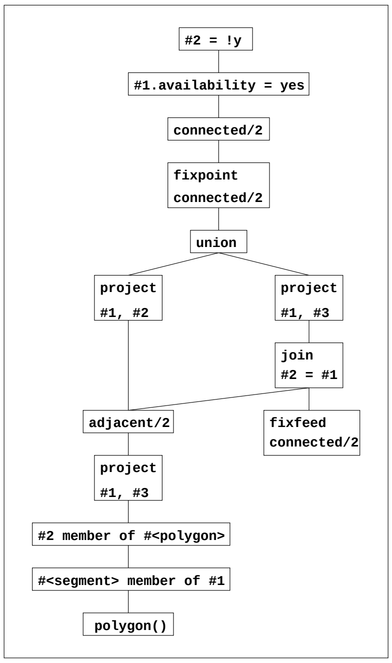

Fig. 10: Example processing tree for ROLL rule.

either, passing on their input tuples directly to their outputs – their function is to mark boundaries in the tree during optimisation.

5.2.4. Processing Tree Optimization

Optimization of processing trees proceeds in two stages. *Global optimization* is analogous to the static filtering optimization of [29]. Constraint nodes appearing in one section of the tree are moved *through* join and goal nodes to subordinate constraint sections, thereby performing the associated selection operations early. The propagation algorithm ensures that constraints propagated through a goal are valid in the presence of multiple invocations of the goal, including recursive invocations. If necessary, alternative propagated constraints are added in parallel rather than in series, implementing a disjunctive constraint.

For example, the equality constraint $\#2 =!y$ in the example processing tree in figure 10 can be propagated through the *connected*/2 goal (note that the index is remapped in accordance with the projection). This constraint remains valid when propagated through the recursion, so it is legitimate to add it to each branch below the project nodes. Since all solutions output by the *connected*/2 node will satisfy the assignment, it may be removed from the top of the tree. The constraint on the value of the *availability* attribute cannot be propagated since it does not remain valid when pushed down through the recursive call. The Equal node does not propagate further down through the *adjacent*/2 goal since the invocation via the join node is unconstrained. The tree resulting from global optimization is shown in figure 11.

Local optimization partitions the tree into sections bounded above by a goal node (or the top node for the query) and below by directly invoked subordinate goal nodes. A heuristic search is used to reorder join nodes in each section and to move constraint nodes down into the join tree. The search is guided by a cost function which estimates a cost for the current tree section using the cost estimate for subordinate sections and estimates for the number of solutions processed by the

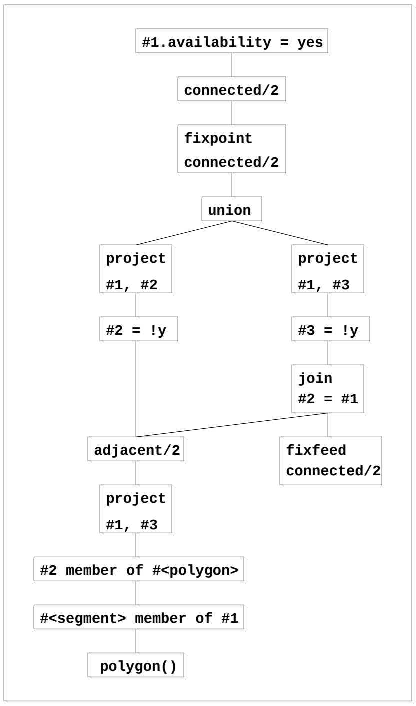

Fig. 11: Example processing tree after global optimization

current section's join and constraint nodes. Since global optimization will have identified where it is possible to propagate constraints into subordinate tree sections, it is only necessary for the search process to consider local orders. This partitioning of the optimization process limits the danger of a combinatoric explosion in the search process.

6. INTEGRATING ROCK & ROLL

The integration of ROCK and ROLL overcomes both aspects of the impedance mismatch which commonly results from embedding one language in another:

- *Type system mismatch:* which is overcome because both languages share the same underlying data model, and have the same type system realised in OM. It is thus possible to perform strong type checking across the interface.
- *Evaluation strategy mismatch:* which is overcome because the data values retrieved by one language can be handled directly by the other, thereby removing the need for copying between formats or for special treatment of temporarily unassigned results.

We are left with a paradigm or stylistic mismatch which implies that the user has to have an understanding of the two different paradigms supported by the integrated system. This form of mismatch is unavoidable in any system which brings together distinct computational paradigms. The nature of the integration in ROCK & ROLL is that the user can switch between using the imperative language and the logic language without suffering from the conventional manifestations of the impedance mismatch, and without having to learn substantially different syntaxes for the two languages. However, it *is* necessary for programmers to change their way of thinking to conform to the style which suits the paradigm being used for any particular task. Thus ROCK & ROLL can be seen as facilitating the development of complete data-intensive applications in which different parts of the application are most naturally expressed using different computational styles.

The integration of the two languages is such that the imperative language can embed any expression in the logic language as long as the type inferred for that expression conforms to the type rules. The logic language can invoke any method defined in the imperative language as long as that method is *side-effect free*.

A type inference mechanism is used with ROLL queries – ROLL can be used both as an interactive query language and as an embedded language. The decision to use type inference for ROLL is motivated by:

- The need to minimise the amount of type definition syntax in an interactive query language, where conciseness of expression is an important practical consideration.
- The need to retain strong type checking in ROCK programs which invoke embedded ROLL queries.
- The need to have as similar a syntax as possible for the interactive and embedded forms of ROLL.
- The need to conform to the expectations of logic programmers using ROLL, who are not accustomed to entering type definitions in other logic programming languages.
- The need to identify type information for use in query optimisation.

It is possible to explicitly associate a type with a ROLL variable using the operator `:`, as in `L:landParcel`. This explicit association of a type with a ROLL variable is sometimes necessary to allow the compiler to infer a unique type for an expression, or can be used as a form of documentation.

The process of integration has two aspects: the embedding of the imperative language in the logic one and vice-versa.

6.1. Embedding ROCK in ROLL

ROLL may invoke any ROCK method as long as the method is side-effect free. A method is side-effect free if it does not update, directly or indirectly, any non-local data. Any method which may itself update any non-local data, or which is overridden by a method which may update non-local data, is considered to have side-effects. The classification of a method according to this criterion can be determined at compile time. There is no additional syntax associated with the invocation of a ROCK method from ROLL (examples are given in section 6.2.2).

6.2. Embedding ROLL in ROCK

6.2.1. Method definition using ROLL

In the integrated system, it is possible to define methods in the logic language as well as in the imperative one. ROCK requires signatures of methods to explicitly distinguish between input and output parameters. This conflicts with the fact that in logic programming no such distinction is required.

The solution adopted is to define ROLL methods within classes with the types of the parameters given by the signature of the method in the corresponding type definition. However, no distinction is made in the definition of a ROLL method between input and output parameters. Thus ROLL methods can be invoked with different binding patterns for their arguments, as illustrated in section 6.2.2.

Overloading and overriding of ROLL methods is allowed, and the process of binding a call to an implementation is handled in a way analogous to that for methods defined in the imperative language.

As an example, the ROLL method `adjacent` attached to the class `polygon` indicates that two polygons are adjacent to each other if they share a common segment, whereas `connected` is the transitive closure of `adjacent`.

```
class polygon
    public:
        adjacent(polygon)
        begin
            adjacent(OtherPoly)@ThisPoly :-
                OtherPoly <> ThisPoly,
                get_member@OtherPoly == get_member@ThisPoly;
        end

        connected(polygon)
        begin
            connected(OtherPoly)@ThisPoly :-
                adjacent(OtherPoly)@ThisPoly;
            connected(OtherPoly)@ThisPoly :-
                adjacent(IntermediatePoly)@ThisPoly,
                connected(OtherPoly)@IntermediatePoly;
        end
    ...
end-class
```

6.2.2. Querying

A query expression belongs to the syntactic class of object expressions of ROCK, and can therefore be used in any context in which an object expression is appropriate.

Queries are delimited by square brackets. A query expression can take one of three forms:

1. [*ROLL goal*]

   Returns `true` if the ROLL goal succeeds, else `false`.

2. [{ *Projected Variables*} | *ROLL goal*]

   All the projected variable bindings which satisfy the goal are collected in an association.

3. [**any** *Projected Variables* | *ROLL goal*]

   This form is a variant of the previous one, in which rather than collecting all solutions together as a set, a single solution is selected nondeterministically.

The following are examples of ROLL queries embedded in ROCK:

- Retrieve all the `landParcel`s which are `adjacent` to those owned by `ICI`.

  ```
  res1 := [ {Y} | adjacent(X)@Y, get_ownerName@get_owner@X == "ICI" ]
  ```

  In this example, the result `res1` must have been previously declared to be an association of `landParcel`s. The type inference system will verify that the result of the embedded ROLL statement is of appropriate type.

  The result of the ROLL query is a new object constructed by association. The persistence of this association is determined by `res1` – it will only persist if `res1` is an instance of a persistent class.

- Retrieve all the land parcels which are available in road `R7` with price less or equal to `10000`.

  ```
  var res2 := [ {LP} |
          get_roadName@get_road@Loc == "R7",
          get_landParcel@Loc == LP,
          get_availability@LP == "yes",
          get_price@LP == Price,
          Price <= 10000 ]
  ```

In this example, the type of `res2` is inferred to be an association of land parcels, as this is the type of the result of the embedded ROLL query. It is then possible for ROCK to manipulate the objects retrieved by the query. For example, the following ROCK code fragment increases by 10% the price of all the `landParcel`s retrieved:

```
foreach l in res2 do
    put_price(get_price@l*1.1)@l;
```

- Retrieve objects which are `adjacent` to various existing land parcels:

```
var p1:= new landParcel(....);
var p2:= new landParcel(....);
...

var res3:= [ {X} | adjacent(X)@!p1 ]

var res4:= [ adjacent(!p1)@!p2 ]

var res5 := [ {<Y,R>} | adjacent(X)@Y, ]
                        get_availability@X == "yes",
                        get_landParcel@Loc == Y,
                        get_road@Loc == R ]
```

In this example, the ROCK variables `p1` and `p2` are assigned to particular (newly created) `landParcel` objects.

The ROCK variable `res3` is then assigned the set of `landParcel`s which are adjacent to the object referenced by `p1`. The value of the ROCK variable `p1` is obtained by the ROLL query by embedding `p1` in the ROLL goal (an embedded variable is distinguished by the prefix `!`).

The ROCK variable `res4` is assigned either `true` or `false`, depending upon whether or not `p1` and `p2` are adjacent.

The ROCK variable `res5` is assigned a set of aggregates of type `landParcel` × `road` representing `roads` that have a `location` next to `landParcel`s that are `adjacent` to available `landParcel`s.

The examples which assign to `res3`, `res4` and `res5` illustrate how the ROLL method `adjacent` can be called using different binding patterns. In the example which assigns to `res3`, the message-recipient is bound but the argument is free. In the example which assigns to `res4` both the message recipient and the argument are bound. In the example which assigns to `res5`, both the message recipient and the argument are free. The optimiser can exploit such binding information, as shown in section 5.2.4.

In fact, it is also possible to use ROCK methods from within ROLL with different binding patterns, although in this case the evaluator must actually call the ROCK method with all input parameters and the message recipient bound.

A more detailed discussion on the nature of the integration of ROCK & ROLL is given in [7].

6.3. Implementation of Integration

The additional implementation work associated with the integration of ROLL and ROCK is quite modest, as each of the integrated components retains its principal features. The following are the principal implementation tasks not mentioned earlier in the paper, but it is noteworthy that much of the work associated with these tasks would be required to support a free-standing ROLL interface to a database described using OM and manipulated using ROCK:

1. *ROLL front-end:* The ROLL parser and type inference system have been implemented within the structural framework described for the ROCK compiler in section 4.2. At the design level,

a whole new set of semantic objects (e.g. *Goal, SubGoal, Clause* etc.) are introduced. For each of these semantic objects, a new E class is defined which implements the methods for type checking/inference.

The type inference algorithm defined for ROLL is based on:

- Using type information provided by the context.
- Assigning generic types to logic variables.
- Gradual derivation of specific types from more general ones.

The concept of generic type is implemented as an E class *PolyType* which references a type expression. This type is replaced as more specific types are inferred. The way *PolyTypes* are used is now described by way of an example. Assume the following query, which returns the set of polygons that are adjacent to polygons with an area greater than `10000`:

```
[ {X} | adjacent(X)@Y, get_area@Y >= 10000 ]
```

Initially, the type of the logic variables `X` and `Y` is assumed to be a PolyType, whose body is NULL. From the subgoal `adjacent(X)@Y`, it can be inferred that both `X` and `Y` have the type `polygon`, as this method is defined in the class `polygon`, and takes as parameter a `polygon`. Thus the PolyType associated with each of these variables is made to point to that type. If it were the case that the next subgoal allowed a more specialised type to be inferred for one or both of the variables, this more specific type would replace `polygon` in the PolyType reference.

2. *ROCK/ROLL evaluation:* The ROCK interpreter must be extended to invoke the ROLL evaluator on embedded ROLL queries, and the ROLL evaluator must be able to invoke the ROCK interpreter to handle ROCK calls from within ROLL queries. This process is eased by the fact that both languages build upon the same object manager, and thus the translation of values passing between the two interpreters is minimal and completely transparent to the programmer.

   The extensions to the ROCK implementation with respect to evaluation only relate to the creation of the corresponding objects (according to the projection expression of the query) from the set of bindings returned by ROLL. A query expression defines an *Eval* method, which invokes the ROLL engine to compute a set of bindings for each variable projected in the query, and collects the result into the appropriate structures.

7. CONCLUSIONS

7.1. *Current Position*

A complete implementation of ROCK & ROLL, as described in this paper, has been developed, such that the individual components consist of approximately the following number of lines of E (persistent C++) code: OM = 3,000; ROCK = 16,000; ROLL = 13,000. The system is available over the Internet – for more information see the following World Wide Web page: http://www.cee.hw.ac.uk/Databases.

7.2. *Summary*

This paper has presented a novel approach to the design and implementation of a deductive object-oriented database system. The overall design builds upon a semantically expressive object-oriented data model from which two languages have been derived, one an imperative manipulation language and the other a logic query language. It has also been shown how these two languages can be integrated without introducing classical impedance mismatches.

The resulting system retains the traditional strengths of both paradigms. In step with other object-oriented databases, application data can be described using a powerful data model and manipulated by a computationally complete programming language, thereby allowing a close integration of programs and the data on which they operate. Object-oriented facilities can be used for structuring and sharing both programs and data in a uniform context. In step with other deductive databases, both queries and rules can be expressed using a logic language which is firmly within the formal framework provided by first-order logic. The logic language is thus amenable to optimisation and evaluation using extensions to existing techniques developed for deductive relational databases.

Not only are traditional strengths maintained, widely recognised weaknesses of the respective paradigms have been overcome. The object-oriented aspects of the system benefit from a formally defined data model and logic query language, thereby allowing rule-based applications to be developed in the context of an object-oriented database. The deductive aspects of the system are enhanced by the expressive data model, by strong type checking of logic queries and methods, and by an integrated data manipulation language which allows updates, I/O and procedural computation without complicating the semantics of the logic language, which remains side-effect free.

Acknowledgements — The work that resulted in this paper has been funded by the UK Engineering and Physical Sciences Research Council through the IEATP programme, and their support is duly acknowledged. 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)*, pp. 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 S. Grumbach. COL: A Logic-Based Language for Complex Objects. In Joachim W.Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Advances in Database Technology - EDBT'88, International Conference on Extending Database Technology*, LNCS 303, pp. 271–293, Venice, Italy. Springer-Verlag (1988).

[4] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pp. 159–173, Portland,OR. ACM Press (1989).

[5] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages (Extended Abstract). In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 240–250, Austin,TX. ACM Press (1988).

[6] M.P. Atkinson and O.P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, **19**(1):105–190 (1987).

[7] 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)*, pp. 463–474. Morgan-Kaufmann (1994).

[8] M.L. Barja, N.W. Paton, and M.H. Williams. Design of an Object-Oriented Database Programming Language for a DOOD. Technical Report TR92016, Department of Computing and Electrical Engineering,Heriot-Watt University (1992).

[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] E. Bertino and D. Montesi. Towards a Logical-Object Oriented Programming Language for Databases. In Alain Pirotte, Claude Delobel, and Georg Gottlob, editors, *Advances in Database Technology - EDBT'92, 3rd International Conference on Extending Database Technology*, LNCS 580, pp. 168–183. Springer-Verlag (1992).

[11] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-based Programming Paradigm. In *Proc. ACM SIGMOD International Conference on the Management of Data*, pp. 225–236. ACM Press (1990).

[12] F. Cacace, S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. An Overview of the Logres System. In *[36]*, pp. 31–43 (1993).

[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*, pp. 474–499, CA 94303-9953. Morgan Kaufman Publishers, Inc. (1990).

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

[15] Q. Chen and W.W. Chu. HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases. In *[31]*, pp. 431–452 (1990).

[16] Q. Chen and G. Gardarin. Nested Datalog: A Rule Language for Complex Objects. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL. ACM Press (1988).

[17] W. Chen and D.S. Warren. C-Logic of Complex Objects. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 369–378, Philadelphia,PA. ACM Press (1989).

[18] M. Dalal and D. Gangopadhyay. OOLP: A Translation Approach to Object-Oriented Logic Programming. In *[31]*, pp. 593–606 (1990).

[19] P. Dechamboux and C. Roncancio. Peplom^d: An Object-Oriented Database Programming Language Extended with Deductive Capabilities. In Dimitri Karagiannis, editor, *Database and Expert System Applications - 5th International Conference, DEXA '94, Proceedings*, LNCS 856, pp. 2–14, Athens, Greece. Springer-Verlag, ISBN 3-540-58435-8 (1994).

[20] A. Dinn. Top-down evaluation of roll. Technical report, Department of Computing and Electrical Engineering, Heriot-Watt University (1993).

[21] M.H. Van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, **23**(4):733–742 (1976).

[22] 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 (1992).

[23] 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 (1993).

[24] J. Grant and J. Minker. Deductive Database Theories. *The Knowledge Engineering Review*, **4**(4):267–304 (1989).

[25] A. Heuer and P. Sander. The LIVING IN A LATTICE Rule Language. *Data & Knowledge Engineering*, **9**:249–286 (1992).

[26] M. Jarke, S. Eherer, R. Gallersdoerfer, M.A. Jeusfeld, and M. Staudt. ConceptBase - A Deductive Object Base Manager. Technical Report 93-14, Aachener Informatik-Berichte/RWTH Aachen (1993).

[27] 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*, pp. 134–146. ACM Press, A revised and extended version exists as [28] (1989).

[28] M. Kifer, G. Lausen, and J. 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), Revised June 1993 (1993).

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

[30] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 379–383, Philadelphia,PA. ACM Press (1989).

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

[32] R. Lanzelotte and P. Valduriez. Optimization of object-oriented recursive queries using cost controlled strategies. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 256–265. ACM Press (1992).

[33] Y. Lou and Z.M. Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD International Conference on the Management of Data*, pp. 198–207. ACM Press (1991).

[34] E. Mendelson. *Introduction to Mathematical Logic.* Mathematics Series. The Wadsworth & Brooks/Cole, Monterey,CA, 3rd edition, ISBN 0-534-06624-0 (1987).

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

[36] I.S. Mumick, editor. *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, Washington, DC (1993).

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

[38] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A Deductive Database System. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD International Conference on the Management of Data*, pp. 308–317. ACM Press (1991).

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

[40] A. Poulovassilis and C. Small. A Functional Programming Approach to Deductive Databases. In G.M. Lohman, A. Sernadis, and R. Camps, editors, *Proceedings of Very Large Data Bases Conf.*, pp. 491–500. Morgan Kaufmann (1991).

[41] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL-Control, Relations and Logic. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases*, pp. 239–250. Morgan Kaufman (1992).

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

[43] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding Object-Orientation to a Logic Database Language. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pp. 158–170. Morgan Kaufmann Publishers (1993).

[44] J.D. Ullman. Assigning an Appropriate Meaning to Database Logic with Negation. Technical report, Department of Computer Science, Stanford University, USA. (1994).

[45] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press (1993).

[46] C. Zaniolo. Object Identity and Inheritance in Deductive Databases – An Evolutionary Approach. In *[31]*, pp. 7–24 (1990).