The ROL Deductive and Object-Oriented Database System

Mengchi Liu

Technical Report CS-96-02
March, 1996

Department of Computer Science
University of Regina
Regina, Saskatchewan, CANADA
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0316-3

The ROL Deductive and Object-Oriented Database System

Mengchi Liu

Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2

Fax: (306) 585-4745
Phone: (306) 585-4700
Email: mliu@cs.uregina.ca

Abstract

This paper describes ROL, a deductive and object-oriented database language which has been implemented. ROL integrates important features of object-oriented and deductive database systems. It supports object identity, complex objects, classes, class hierarchy, multiple inheritance with overriding and blocking, and schema. It also supports structured values such as functor objects and sets, treats them as first class citizens, and provides powerful mechanisms for representing both partial and complete information on sets. It is an extension of pure valued-oriented deductive systems such as Datalog and LDL and subsumes them as special cases. It supports important integrity constraints such as domain, key, referential, functional dependency, and cardinality in a uniform framework. Furthermore, it has a logical semantics that cleanly accounts for all of its object-oriented and value-oriented features.

1 Introduction

Deductive and object-oriented databases are two important extensions of the traditional database technology. Deductive databases extend the expressive power of traditional databases by means of recursion and declarative querying. Examples of such languages are Datalog [16], LDL [33] and CORAL [35]. Object-oriented databases extend the data modeling power of traditional databases by means of object identity, complex objects, classes, class hierarchy, inheritance and schema. Examples of such languages are Iris [20], Exodus [15], Orion [25], O_2 [27] and Jasmine [22]. However, both extensions have shortcomings. Deductive databases lack powerful data modeling mechanisms, while object-oriented databases lack logical semantics and declarative query languages. In the past few years, a lot of efforts have been made to integrate deductive and object-oriented databases to gain the best of the two approaches. A number of deductive object-oriented database languages have been proposed, such as O-logic [32], revised O-logic [24], C-logic [17], IQL [3], IQL2[1], F-logic [23], LOGRES [13], LLO [31], LOL [12], Datalog$^{\text{method}}$ [4], DLT [8], Gulog [19] and Rock & Roll [9].

Object identity is useful for supporting object sharing and update management. However, using object identifiers for every object is burdensome even in pure object-oriented databases and being able to use structured values as well is important as discussed in [3, 10, 14, 27]. In deductive object-oriented databases, using object identifiers for every object is problematic and pure value-oriented approach is argued better in this regard [37].

In this paper, we describes the ROL language that has been implemented. The name stands for Rule-based Object Language. The ROL language integrates in a uniform framework important features of object-oriented and value-oriented deductive approaches. It supports not only object identity but also structured values such as functor objects and sets. The user can use object identifiers for objects when it is better to do so or directly use structured values for objects when using object identifiers for such objects is unnecessary or problematic. Object sharing in ROL can be achieved not only by object identity but also by using rules. Indeed, ROL can be used as a pure object-oriented deductive database programming language, or as a pure value-oriented deductive database programming language. Most importantly, ROL allows both value-oriented and object-oriented features to be used together to take the advantages of both approaches.

In ROL, values, object identifiers, functor objects and sets are treated uniformly as objects so that functional dependencies can be represented directly and more generally than in functional data models [21, 36]. It builds-in other important integrity constraints such as domain, key, referential and cardinality in a simple and uniform framework. It also supports other important object-oriented features such as classes, class hierarchy, multiple inheritance with overriding and blocking, and schema.

ROL provides powerful set representation mechanisms that combine LDL and F-logic set treatments so that information about sets can be represented partially or completely. It has a uniform notion of schema for objects represented both extensionally and intensionally. It supports schema queries, and facts and rule queries in addition to traditional database queries. It also supports declarative updates of schema, facts and rules. It has a well-defined logical semantics that cleanly accounts for all of its object-oriented and value-oriented features [29].

This paper is organized as follows. Section 2 describes objects and attributes in ROL and discusses the flexibility of data modeling in ROL. Section 3 explains classes, class hierarchy and inheritance with overriding and blocking in ROL. Section 4 focuses on rules of ROL. Section 5

introduces ROL program structure. Sections 6 shows how to query schema, objects, facts and rules. Section 7 describes how ROL system supports updates on schema, facts and rules. Section 8 gives a brief comparison with related proposals, which highlights the novel features of ROL. Section 9 concludes the paper.

2 Objects and Attributes

ROL is a language that centers on objects and attributes. There are four kinds of objects in ROL:

(1) values such as integers 5, 8, reals 3.14, 1.0, and strings *'Smith', 'Mary'*, etc;
(2) object identifiers such as $smith, mary, s_1, p_1$;
(3) functor objects such as $family(bob, ann)$, $supplies(s_1, p_1)$, $parents(tom, \langle bob \rangle)$;
(4) sets which are partitioned into two kinds:
    (a) partial sets such as $\langle john, mary \rangle$, $\langle family(john, mary) \rangle$, $\langle parents(john, \langle smith \rangle) \rangle$;
    (b) complete sets such as $\{\}, \{john, mary\}, \{family(john, mary)\}$;

Complete sets are the normal sets while partial sets are special notation used to denote part of complete sets. We will show the difference using examples later.

Objects have attributes through which they can be related to each other. First of all, not only object identifiers, but also values, functor objects, and sets (partial or complete) can have attributes in ROL. The attribute values can be objects of any kinds. Following are several examples:

$3[factorial \rightarrow 6]$
$bob[name \rightarrow$ *'Bob'*$, age \rightarrow 25, parents \rightarrow \{pam, tom\}]$
$supplies(s_1, p_1)[quantity \rightarrow 200]$
$family(bob, ann)[children \rightarrow \{jim, pat\}, numChildren \rightarrow 2]$
$\{jim, pat\}[count \rightarrow 2]$

ROL supports object-oriented data modeling, value-oriented data modeling, and their combination. For example, to represent the facts that Bob and Ann have common children Jim and Pat, and live in the 123 King St, Regina, we can use three different ways in ROL:

Value-oriented approach This is the approach we use in languages such as LDL in which we don't have object identifiers.

$person($*'Bob'*$, \{$*'Jim', 'Pat'*$\}, address($*'123 King St', 'Regina'*$))$
$person($*'Ann'*$, \{$*'Jim', 'Pat'*$\}, address($*'123 King St', 'Regina'*$))$

The problem with this approach is the lack of support for object sharing. If they have another child or move, both facts must be updated. Also, *'Bob'* and *'Ann'* are subject to change which may result in database inconsistent.

Object-oriented approach This is the approach that object-oriented data models such as O_2 support.

$bob[children \rightarrow s, address \rightarrow a]$
$ann[children \rightarrow s, address \rightarrow a]$
$s[set \rightarrow \{jim, pat\}]$
$a[city \rightarrow'$ Regina$', street \rightarrow'$ 123 Castle Rd$']$

The object identifiers provide supports for object sharing and update management.

Their combination We can use the combination of both object-oriented and value-oriented approaches in ROL. That is, for structured values such as functor objects and sets, we don't associate separate object identifiers. Instead, they are identified by themselves. We can use rules for object sharing:

$bob[children \rightarrow \{jim, pat\}, address \rightarrow address$('Regina', '123 King St')$]$
$ann[children \rightarrow X, address \rightarrow Y] :- bob[children \rightarrow X, address \rightarrow Y]$

Follow the convention of Prolog, words starting with a capital letter are variables in ROL.

In some cases, pure object-oriented approach is not appropriate. For example, to represent the facts that Bob and Ann have common children Jim and Pat, and Ann also has her own child Liz. We can only use the third method in order to achieve object sharing:

$bob[children \rightarrow \{jim, pat\}]$
$ann[children \rightarrow \langle liz \rangle]$
$ann[children \rightarrow \langle X \rangle] :- bob[children \rightarrow \langle X \rangle]$

Here the complete set $\{jim, pat\}$ provides complete value for the attribute *children* of *bob*, while the partial set $\langle liz \rangle$ provides partial value of the attribute *children* of *ann*. The complete value $\{jim, pat, ann\}$ for the attribute *children* of *ann* is obtained by combining the partial value $\langle liz \rangle$ with the partial values $\langle jim \rangle$ and $\langle pat \rangle$ derived with the rule.

The notion $\langle X \rangle$ is called a partial set term. The partial set term functions differently depending on where it occurs in the rule. When in the body, it denotes part of a set. When in the head, it derives partial information about a set which can be combined with other partial information to obtain complete information about the set.

Many-to-many relationships can be handled in different ways in ROL. Let us consider the student/course relationships. One symmetric method is to use functor objects to represent these relationships and then use rules to derive all the courses a student takes and all the students enrolled in a course as follows:

$student\_course(smith, cs375)$
$student\_course(smith, cs355)$
$student\_course(mary, cs375)$
...
$Student[takes \rightarrow \langle Course \rangle] :- student\_course(Student, Course)$
$Course[enrollment \rightarrow \langle Student \rangle] :- student\_course(Student, Course)$

Updates to the student course information is as simple as in the relational database. Therefore, supporting both object identity and structured values gives the user the flexibility in modeling the real world.

3 Classes, class hierarchy and inheritance

Objects often share common attributes. In ROL, the user must organize objects using classes. Classes denote collections of objects that share common attributes. Corresponding to objects, four kinds of classes are distinguished:

(1) value classes which are classes for values such as
$integer, real, string, integer(15..30), string(\{\text{'Male', 'Female'}\})$
(2) object identifier classes which are classes for object identifiers such as
$student, course, supplier, part, person$;
(3) functor object classes which are classes for functor objects such as
$student\_course(student, course), supplies(supplier, part), family(person, person)$
(4) set classes which are the classes for sets such as
$\{person\}, \{person\}(0, 2), \{part\}, \{quantity(part, integer)\}$

In ROL, subranges of integers and reals, and enumeration of strings are supported. Furthermore, cardinality constraints on sets are also supported. For example, the set class $\{person\}(0, 2)$ denotes a collection of sets with at most two persons while $\{person\}(2, 2)$ denotes a collection of sets with exactly two persons.

An object in the collection denoted by a class is called an *instance* of the class. Instances of value classes are built-in.

Instances of set classes are automatically determined. For example, if the class *person* has instances *jim* and *pat*, then, $\{\}$, $\{jim\}$, $\{pat\}$ and $\{jim, pat\}$ are automatically instances of the set class $\{person\}$.

Object identifier classes and functor object classes may have two kinds of instances: immediate instances and non-immediate instances. We will discuss the non-immediate instances later. Immediate instances must be explicitly asserted. For example, $bob : student$ asserts object identifier *bob* to be an immediate instance of the class *person*, while $family(bob, ann)$ asserts itself to be an immediate instance of the class $family(person, person)$ provided that *bob* and *ann* both are instances of the class *person*.

In ROL, object identifiers and functor objects can only be immediate instances of exactly one class.

The attributes applicable to all instances of a class must be defined using attribute definitions. In the attribute definitions, we can directly specify functional dependency, key, referential and cardinality integrity constraints in the sense of the relational model. Following examples show some attribute definitions for classes *person*, $supplies(supplier, part)$, and $\{person\}$:

$$person[age \Rightarrow integer(1..125), gender \Rightarrow string(\{\text{'Male', 'Female'}\}, father \Rightarrow person,$$
$$mother \Rightarrow person, parents \Rightarrow \{person\}(0, 2), ancestors \Rightarrow \{person\}]$$
$$supplies(supplier, part)[quantity \Rightarrow integer(0..500)]$$
$$\{person\}[count \Rightarrow integer]$$

These attribute definitions say that *person*, $supplies(supplier, part)$, $\{person\}$ are keys, which single-valued or set-valued functionally determine their corresponding attributes. The class *supplier* and *part* are foreign keys in $supplies(supplier, part)$. For a person, the number of parents must be between 0 and 2.

As shown by the above examples, the attribute definitions in ROL can be made meaningful and specific and contain rich integrity constraints.

The attribute definitions of a class are used to constrain the possible attribute values of its instances represented extensionally or intensionally. It is allowed for attribute values of objects to be unknown.

Inheritance is a powerful mechanism for organizing data which allows the user to define classes in an incremental way by refining already existing ones.

In ROL, we can organize classes into class hierarchies by defining subclasses. Only subclasses of object identifier and functor object classes can be defined. Subclasses of value classes and set classes are derived.

A subclass inherits all attribute definitions of its superclasses unless it overrides or blocks them and can introduce additional attribute definitions. Following examples illustrate this.

$$student[age \Rightarrow integer(15..30), takes \Rightarrow \{course\}(0,5)]\ isa\ person$$
$$employee[age \Rightarrow integer(18..65), salary \Rightarrow integer]\ isa\ person$$
$$family(person, person)[children \Rightarrow \{person\}]\ isa\ taxunit.$$

The class *student* and *employee* inherit all attribute definitions of *person* except *age*. They both override (refine) the attribute definition for *age* and introduces additional attribute definition local to themselves.

In ROL, we use set inclusion semantics for subclasses. That is, an instance of a subclass is also an instance of its superclasses. For example, if *tom* and $family(bob, ann)$ are immediate instances of *student* and $family(person, person)$ respectively, then they are non-immediate instances of *person* and *taxunit* respectively. Therefore, an object identifier class can have functor objects as its non-immediate instances and a functor object class can also have object identifiers as its non-immediate instances.

The functor object $family(bob, ann)$ represents not only relationship between *bob* and *ann* but also an entity which may be used as attribute values or participate in other relationships.

As multiple attribute inheritance is allowed, it is possible that a subclass inherits several attribute definitions. They all constrains the instances of the subclass. Consider the following example:

$$workstudent\ isa\ student, employee.$$

The class *workstudent* inherits attribute definitions of *student* and *employee* directly defined or inherited. It thus inherits two attribute definitions for the *age* attribute. Therefore, the attribute *age* of instances of *workstudent* is constrained by both $integer(15..30)$ and $integer(18..65)$ which are equivalent to $integer(18..30)$. In other words, constraints on a subclass are the conjunction of constraints on its superclasses. If this is not what is intended, then the user has to override the inherited attribute definitions by introducing a new one:

$$workstudent[age \Rightarrow integer(15..50)]\ isa\ student, employee.$$

This mechanism is the similar to the one used in O_2.

In ROL, the user can also block the attribute inheritance from superclasses. For example, the class *orphan* can be defined as a subclass of *person*. But it doesn't make sense for it to inherit $father$ and *mother* attributes from *person*. The inheritance of the attributes $father$ and *mother* should be blocked for *orphan* and all of its subclasses. This can be done in ROL by using the built-in class *none* as follows:

$$orphan[father \Rightarrow none, mother \Rightarrow none]\ isa\ person$$

$french\ isa\ person$
$french\_orphan\ isa\ french, orphan$

Types have played an extremely important role in development and study of programming languages and database systems. In logic programming languages, the benefits of introducing types have been increasingly recognized [2, 34]. However, the circular reference in their data structures prevents them from having a type system similar to that of traditional programming languages. In ROL, we provide a solution to that. A functor in ROL is allowed to be associated with exactly one functor object class. For example, if the functor $family$ is used for the class $family(person, person)$, then we cannot have another class such as $family(\{person\})$. As a result, we can simply use the functor to denote the class associated with it and define a functor object class recursively. For example, we can have:

$tree(node, tree, tree)$
$list(integer, list)$

Every functor object class has a built-in instance nil. Therefore, we can have following well-typed functor objects:

$tree(a, nil, nil)$
$tree(b, tree(a, nil, nil), tree(c, nil, nil))$
$list(1, nil)$
$list(1, list(2, list(3, nil)))$

As shown by the above examples, ROL is a typed language and supports rich structured data. If objects or its attribute values are not well-typed with respect to their class definitions, typed errors will occur during the run time. The type checking mechanism of ROL provides a means to detect the illegal description of objects.

4 Rules

ROL is a rule-based declarative language so that information about objects can be represented not only extensionally but also intensionally.

In general, we may use variables in places of objects and thus obtain terms in ROL. Corresponding to objects, four kinds of terms can be used in rules: value terms, object identifier terms, functor terms, and set terms. Sets term can be partial set terms of the form $\langle term_1, ..., term_n \rangle$ with $n \geq 1$ or complete set terms of the form $\{term_1, ..., term_n\}$ with $n \geq 0$.

Rules express intensional information about objects: their classes and/or their attribute values. In ROL, arithmetic and set-theoretic comparison expressions are built-in, and negation is supported. They can be used in rule body. Consider the following rules:

$X[parents \rightarrow \langle Y \rangle] \text{ :- } X[father \rightarrow Y]$
$X[parents \rightarrow \langle Y \rangle] \text{ :- } X[mother \rightarrow Y]$
$X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[parents \rightarrow \langle Y \rangle]$
$X[ancestors \rightarrow \langle Y \rangle] \text{ :- } X[ancestors \rightarrow \langle Z \rangle], Z[parents \rightarrow \langle Y \rangle]$
$X[trueancestors \rightarrow \langle Y \rangle] \text{ :- } X[ancestors \rightarrow \langle Y \rangle], not\ X[parents \rightarrow \langle Y \rangle]$
$family(Y, Z)[children \rightarrow \langle X \rangle] \text{ :- } X[father \rightarrow Y, mother \rightarrow Z]$
$X[numChildren \rightarrow Y] \text{ :- } X[children \rightarrow S], S[count \rightarrow Y]$

$$\{X\}[count \rightarrow 1] \text{ :- } X : person$$

$$S[count \rightarrow X] \text{ :- } S_1[count \rightarrow Y], S_2[count \rightarrow Z], X = Y + Z, S = S_1 \cup S_2, S_1 \cap S_2 = \{\}$$

The first two rules say that if Y is the father or the mother of X, then Y is one of the parents of X. The next two rules specify how to derive the values of attribute *ancestors* of X recursively. The value of attribute *trueancestors* of X can be derived using negation. The next rule involves a functor object term. It says that if X has father Y and mother Z, then the family denoted by $family(Y, Z)$ has X as one of its children. The next rule says that if X has children S and S has Y members, then the *numChildren* of X is Y. The last two rules specify how to compute the number of elements in sets, which show that traditional database aggregate operations such as *total*, *count*, etc. can be represented directly in ROL.

ROL subsumes Datalog and LDL as special cases. Following examples show that LDL grouping and set enumeration are supported in ROL.

$$parents(X, \langle Y \rangle) : -parent(X, Y)$$

$$book\_deal(\{X, Y, Z\}) : -book(X, Px), book(Y, Py), book(Z, Pz)$$
$$X \neq Y, X \neq Z, Y \neq Z, Px + Py + Pz < 100$$

5 Programs

A ROL program consists of three parts: a schema, facts, and rules. The schema is a set of classes and their attribute definitions as well as immediate subclass relationship definitions. The facts are the extensional information about objects. The rules are the intensional information about objects.

Following is a ROL program. It is taken from [7], where it is proposed as a task to test database languages.

Schema
$$part[name \Rightarrow string]$$
$$basepart[cost \Rightarrow integer(0..1000), mass \Rightarrow integer(0..10000)] \ isa \ part$$
$$compositepart[madefrom \Rightarrow \{quantity(part, integer)\}(1, 5),$$
$$assemblycost \Rightarrow integer(0..5000),$$
$$massincrement \Rightarrow integer(0..10000)] \ isa \ part$$
$$\{quantity(part, integer)\}[totalcost \Rightarrow integer, totalmass \Rightarrow integer]$$

Facts
$$p_1 : basepart[cost \rightarrow 20, mass \rightarrow 50]$$
$$p_2 : basepart[cost \rightarrow 10, mass \rightarrow 30]$$
$$p_3 : basepart[cost \rightarrow 15, mass \rightarrow 40]$$
$$p_4 : compositepart[madefrom \rightarrow \{quantity(p_2, 3), quantity(p_3, 2)\}]$$
$$p_5 : compositepart[madefrom \rightarrow \{quantity(p_1, 1), quantity(p_4, 2)\}]$$

$$
\begin{aligned}
\text{Rules}\quad & \{quantity(P,\ Q)\}[totalcost \to C, totalmass \to M] \text{ :- } quantity(P,\ Q), \\
& \qquad P[cost \to C_1, mass \to M_1], C = Q \times C_1, M = Q \times M_1 \\
& \{quantity(P,\ Q)\}[totalcost \to C, totalmass \to M] \text{ :- } quantity(P,\ Q), \\
& \qquad X[assemblycost \to C_1, massincrement \to M_1], \\
& \qquad C = Q \times C_1, M = Q \times M_1 \\
& S[totalcost \to C, totalmass \to M] \text{ :- } S = S_1 \cup S_2, S_1 \cap S_2 = \{\}, \\
& \qquad S_1[totalcost \to C_1, totalmass \to M_1], \\
& \qquad S_2[totalcost \to C_2, totalmass \to M_2], \\
& \qquad C = C_1 + C_2, M = M_1 + M_2 \\
& P[assemblycost \to C, massincrement \to M] \text{ :- } P[madefrom \to S], \\
& \qquad S[totalcost \to C, totalmass \to M]
\end{aligned}
$$

There are two kinds of parts: base parts and composite parts. Both *basepart* and *compositepart* are subclasses of *part* and inherit the *name* attribute from *part*. In order to express the quantity of each part used to manufacture a composite part, a functor object class $quantity(part, integer)$ is used. The set class $\{quantity(part, integer)\}$ has attributes *totalcost* and *totalmass*.

The facts in the program are about the base parts, their cost and mass, the quantity of parts used in manufacturing composite parts, and the way composite parts are made from other parts. The rules in the program tell how to compute the assembly cost and mass increment of composite parts.

To see how the assembly cost and mass increment of composite parts are derived using rules, let us consider the composite part p_4. Using the last rule and the fact that $p_4[madefrom \to \{quantity(p_2, 3), quantity(p_3, 2)\}]$, we just need to derive the *totalcost* of the set $\{quantity(p_2, 3), quantity(p_3, 2)\}$. With the second last rule, we can partition the set into $\{quantity(p_2, 3)\}$ and $\{quantity(p_3, 2)\}$ and directly derive their *totalcost* and *totalmass* with the first rule. We therefore derive p_4's cost 60 and mass 170. Similarly, we can derive p_5's cost 140 and mass 390.

This example shows that the task of finding the assembly cost and mass increment of a composite part can be done in ROL not only declaratively, but also object-orientedly.

6 Queries

Three kinds of queries can be issued in ROL: schema queries, object queries, and fact and rule queries. The ROL system will always give all answers at once (i.e., set-at-a-time).

6.1 Schema queries

Schema queries are used to retrieve information about classes and their attribute definitions. The user can retrieve information about immediate subclass relationships *isa* and general subclass relationships $isa*$ between object identifier classes and functor object classes, and general subclass relationships $eisa*$ (extended $isa*$) between arbitrary classes. Following are several examples where class and attribute variables are used:

$?- workstudent\ isa\ X$
$?- X\ isa*\ person, not\ X\ isa\ person$
$?- X\ eisa*\ \{person\}$
$?- student[A \Rightarrow X], A \neq age$
$?- X[name \Rightarrow \_\,]$

The first query asks for all immediate superclasses of the class *workstudent*. The second query asks for all non-immediate subclasses of *person*. The third asks for all subclasses of the set class $\{person\}$. The next one asks for all attribute definitions of *student* other than *age*. Only the attribute definitions that are directly defined or inherited but not overrided or blocked will be shown in the results. The last one asks for the classes that have an attribute *name*. The underscore character denotes anonymous variable which we do not care about its values as in Prolog.

6.2 Object Queries

The user can query the information about objects represented extensionally by facts or intensionally by rules. The results to such a query is always based on the final evaluated and combined information.

Following are several examples:

$?- X : person$
$?- liz[children \rightarrow \langle X \rangle], not\ bob[children \rightarrow \langle X \rangle]$
$?- liz[children \rightarrow S]$
$?- liz : P[A \rightarrow X], P[A \Rightarrow Y]$

The first query asks for all instances of the class *person*. The second one asks for children of *liz* but not of *bob*. The third asks for all the children of *liz*. The next one asks for the class of *liz* and its attribute values and the attribute definitions of its class as well. The last one asks for all persons with two or three children.

6.3 Fact and Rule Queries

The object queries allow us to obtain evaluated and combined information about objects that is represented intensionally or extensionally. Sometimes, we need to know how the information about objects is represented. That is, we need to query facts and rules.

As facts and rules are used to specify instances of classes and attribute values of objects, we can use class names and attribute names to query them. Following are some examples:

$?- ann[children]$
$?- [ancestors]$
$?- family[\ ]$

The first query asks for all facts and rules that are used to derive the values of the attribute *childrent* of *ann*. Based on the example in Section 2, we will get the following results where $f5$ and $r7$ are the fact and rule numbers associated with them in the ROL system:

$f5:\ \ ann[children \rightarrow \langle liz \rangle]$
$r7:\ \ ann[children \rightarrow \langle X \rangle] :- bob[children \rightarrow \langle X \rangle]$

The next query asks for all rules that are used to derive the values of attribute *ancestors*. Based on the rules shown in Section 4, the following results will be given by the ROL system:

$r10:\ \ X[ancestors \rightarrow \langle Y \rangle] :- X[parents \rightarrow \langle Y \rangle]$
$r11:\ \ X[ancestors \rightarrow \langle Y \rangle] :- X[ancestors \rightarrow \langle Z \rangle], Z[parents \rightarrow \langle Y \rangle]$

The last query asks for all rules that are used to derive the instances of the class $family$. We will have:

$$r13: \quad family(Y,Z)[children \rightarrow \langle X \rangle] \text{ :- } X[father \rightarrow Y, mother \rightarrow Z]$$

7 Updates

The ROL system supports schema, facts and rules updates. Update operations can be used in the body of rules or in queries.

For the schema, the user can insert or delete classes and their immediate subclass relationships, and insert, delete or modify attribute definitions of classes. Consider these examples:

$insert\ person[age \Rightarrow integer]$
$insert\ student[age \Rightarrow integer(15..30), takes \Rightarrow \{course\}]\ isa\ person$
$modify\ person[age \Rightarrow integer(0..100)]$
$delete\ student[age \Rightarrow \_]$
$delete\ course$

The first operation inserts a new attribute definition for *person* and if the class *person* doesn't exist in the system, then it also inserts this class. This operation will fail if *person* already has the attribute *age* defined. The second operation inserts the class *student* as an immediate subclass of *person* which refines the attribute definition *age* and introduces a new attribute definition *takes*. This operation will fail if the class *course* is not defined. The third operation modifies the attribute *age* defined on *person*. It will fail if this new constraint is violated by some of its instances. The next operation deletes the directly defined attribute *age* on *student* so that the inherited definition can be used instead. It will fail if some of its instances violate the inherited attribute definition. The last operation will delete the class *course* and $\{course\}$ as well as their instances. Furthermore, the attribute *takes* of *student* and its instances will also be deleted.

For the facts, the user can insert new facts or delete part of existing facts about objects. Following are several examples:

$insert\ smith : student[age \rightarrow 25, takes \rightarrow \langle cs375, cs355 \rangle]$
$delete\ cs375[name \rightarrow \_]$
$delete\ cs375$
$delete\ X : student$

The first operation inserts the object identifier *smith* as an immediate instance of *student* with corresponding values for attribute *age* and *takes*. It will fail if *smith* is already an immediate instance of another class, the attribute values are not well-typed respect to the schema, *smith* already has a value for *age*, or the existing value for *takes* is not consistent with the new value. The second operation deletes the value of the attribute *name* of the object *cs*375. It will fail if the value is not known. The third operation deletes the object identifier *cs*375 and all of its attribute values from the system. It is also deleted from the value of the attribute *takes* of *smith* so that the referential integrity constraints are maintained. The last operation deletes all immediate instances of the class *student*.

Following example shows how to increase the salary of employees by 10%:

$$delete\ X[salary \rightarrow Y], Z = Y * 1.1, insert\ X[salary \rightarrow Z]$$

In the ROL system, the deletion of facts is always done before the insertion of facts in order to avoid inconsistency. An update can either succeed or fail. If it fails, it has no effect on the database. For the above example, the insertion of a new salary for an employee may fail if the salary is above the salary range. If so, the whole update will fail and all changes will be undone. Besides, the user can also explicitly specify the order of operations. For example

$$(insert\ john : student, delete\ smith : student)$$

tells the system to do insertion first and deletion second.

The user can also delete and insert facts and rules together. Consider the following example:

$$delete\ ann[children], insert\ ann[children \rightarrow S] : -tom[children \rightarrow S]$$

First, the fact $f5$ and the rule $r7$ shown in Section 6 obtained with the query $ann[children]$ will be deleted and the new rule will be inserted. If we just want to delete the rule $r7$ rather than $f5$, we can simply use:

$$delete\ r7$$

8 Comparison with Related Approaches

In this section, we briefly compare ROL with a number of reported deductive languages.

As ROL effectively integrates important features in deductive databases and object-oriented databases, none of the reported deductive languages has the same expressive and data modeling power as ROL.

Compared to deductive database language Datalog [16], COL [2], LDL [11], LPS [26], CORAL [35], and Relationlog [28] as well as logic programming language Prolog, ROL is unique as it supports object identity, complex objects, classes, class hierarchy and multiple inheritance with overriding and blocking, and functional dependency in a uniform framework. It subsumes Datalog and LDL as special cases.

Compared to deductive object-oriented database languages such as O-logic [32], revised O-logic [24], F-logic [23], LLO [31], DLT [8], and Gulog [18], as well as their predecessors LOGIN [5] and LIFE [6], ROL is unique as it supports structured values and the notion of schema.

Unlike deductive database languages such as Datalog, LDL and CORAL which have no notion of schema for objects represented intensionally by rules, ROL is a strongly typed language and has a uniform notion of schema for objects represented both extensionally and intensionally. Furthermore, it supports schema queries, and facts and rule queries in addition to traditional database queries. It also supports declarative updates of schema, facts and rules.

Compared to deductive languages that support sets such as COL, LPS, LDL, revised O-logic and F-logic, ROL is more expressive in terms of set representation as it combines LDL and F-logic set treatments by means of partial set terms and complete set terms.

As shown in Section 5, ROL can perform the database task proposed in [7] to test database programming languages not only declaratively, but also object-orientedly. None of other deductive languages can do the same. In LDL, the task can be done declaratively but not object-

orientedly. In COL, IQL, and Rock and Roll [9], the task cannot by done declaratively, but only imperatively by resorting to a low-level imperative program.

LOGRES [13] and IQL [3] also supports object identity, relationships and sets. However, they are strictly based on the Entity/Relationship model so that relationships can contain object identifiers but object identifiers cannot have relationships as attribute values and relationships cannot contain other relationships. ROL is more general and uniform than LOGRES and IQL.

In LDL and CORAL, the user has to provide all possible query forms so that the program can be compiled accordingly. For a predicate with n arguments, there are 2^n possible query forms to be specified and the user cannot issue a query that has no corresponding query form. This is burdensome from the user's point of view. In ROL, the user can issue any kind of queries as long as they are syntactically correct. The system dynamically rewrites rules relevant to the queries and uses bottom-up and top-down alternating technique to achieve the objective of set-at-a-time model of computation and the reasonable performance execution of the queries. The details about the implementation may be found in the paper [30] which is in preparation.

9 Conclusion

This paper has described ROL, a powerful and flexible declarative database programming language which effectively integrates important features of deductive databases and object-oriented databases. The formal logical semantics about the ROL language is described in [29].

A preliminary version of ROL has been released and is available through anonymous ftp from *ftp.cs.uregina.ca/pub/rol/*. A functionally complete implementation of ROL has just been completed and will be made available publicly after further testing and debugging. Extensive examples can be found in the package. The work is underway on query optimization techniques, storage structures and indexing mechanisms in order to enhance the performance of the ROL system. Interfaces to C/C++ and X-window are also being developed.

References

[1] S. Abiteboul. Towards a deductive object-oriented database language. *Data and Knowledge Engineering*, 5(2):263–287, 1990.

[2] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. *ACM TODS*, 16(1):1–30, 1991.

[3] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–173, 1989.

[4] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 32–41, 1993.

[5] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *J. Logic Programming*, 3(3):198–215, October 1986.

[6] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of life. *J. Logic Programming*, 12, 1993.

[7] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, 1987.

[8] R. Bal and H. Balsters. A Deductive and Typed Object-Oriented Language. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Deductive and Object-Oriented Databases*, pages 340–359, Phoenix, Arizona, USA, December 1993. Springer-Verlag Lecture Notes in Computer Science 760.

[9] 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(3):185–211, 1995.

[10] C. Beeri. Formal Models for Object-Oriented Databases. In W. Kim, J.M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 405–430, Kyoto, Japan, December 1989. North-Holland.

[11] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set construction in a logic database language. *J. Logic Programming*, 10(3,4):181–232, April/May 1991.

[12] E. Bertino and D. Montesi. Towards a Logical Object-oriented Programming Language for Databases. In *Proc. Intl. Conf. on Extending Database Technology*, pages 168–183. Springer-Verlag, March 1992.

[13] F. Cacace, S. Ceri, S. Crepi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modelling with a rule-based programming paradigm. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 251–261, 1990.

[14] D. Calvanese and M. Lenzerini. Making object-oriented schemas more expressive. In *ACM Symp. on Principles of Database Systems*, pages 243–254, 1994.

[15] M. Carey, D. DeWitt, and S. Vanderberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1988.

[16] S. Ceri, G. Gottlob, and T. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[17] W. Chen and D. S. Warren. C-logic for complex objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 369–378, 1989.

[18] G. Dobbie and R. Topor. A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Deductive and Object-Oriented Databases*, pages 473–488, Phoenix, Arizona, USA, December 1993. Springer-Verlag Lecture Notes in Computer Science 760.

[19] G. Dobbie and R. Topor. On the Declarative and Procedural Semantics of Deductive Object-Oriented Systems. *Journal of Intelligent Information System*, 4:193–219, 1995.

[20] D. H. Fishman, B. B., H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):48–69, January 1987.

[21] B. C. Housel, V. Waddle, and S. B. Yao. The functional dependency model for logical database design. In *Proc. Intl. Conf. on Very Large Data Bases*, 1979.

[22] H. Ishikawa, F. Suzuki, F. Kozakura, A. Makinouchi, M. Miyagishima, Y. Izumida, M. Aoshima, and Y. Yamane. The model, language, and implementation of an object-oriented multimedia knowledge base management system. *ACM TODS*, 18(1):1–50, 1993.

[23] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, 1995.

[24] M. Kifer and J. Wu. A logic for programming with complex objects. *J. Computer and System Sciences*, 47:77–120, 1993.

[25] Won Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[26] G. M. Kuper. Logic programming with sets. *J. Computer and System Sciences*, 41:44–64, 1990.

[27] C. Lecluse and P. Richard. The O_2 database programming language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 411–422, Amsterdam, The Netherlands, 1989.

[28] M. Liu. Relationlog: A typed extension to datalog with sets and tuples (extended abstract). In *Proc. Intl. Logic Programming Symp.*, pages 83–97, Portland, Oregon, U.S.A., December 1995. MIT Press.

[29] M. Liu. A typed deductive object-oriented database language with sets. *Submitted for Publication*, May 1995.

[30] M. Liu and W. Yu. Implementation of the ROL rule-based object system. *In preparation*, 1996.

[31] Y. Lou and M. Ozsoyoglu. LLO: A deductive language with methods and method inheritance. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 198–207, 1991.

[32] D. Maier. A logic for objects. Technical Report CS/E-86-012, Oregon Graduate Center, Beaverton, Oregon, 1986.

[33] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[34] F. Pfenning, editor. *Types in Logic Programming.* MIT Press., 1992.

[35] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations and logic. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 238–250, 1992.

[36] D. W. Shipman. The functional extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, 4(4):297–434, December 1979.

[37] J. Ullman. A Comparison between Deductive and Object-Oriented Databases Systems. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases*, pages 263 – 277, Munich, Germany, December 1991. Springer-Verlag Lecture Notes in Computer Science 566.