Deductive Object-Oriented Database Languages: A Survey

Mohammad Ashrafuzzaman

Supervisor: Anthony J. Kusalik

Abstract

The deductive object-oriented database (DOOD) model is an integration of the object-oriented data modeling paradigm and rule-based deduction to provide a single powerful framework for intelligent database systems of the future. The database language of a DOOD system should specifically be based on a logic to facilitate use of deductive database technology and also to provide the theoretical foundation for the underlying data model. In this paper, we survey the languages proposed for deductive object-oriented databases and, in particular, present a more detailed account of two DOOD languages—F-logic and Orlog. Each of these languages is introduced by a short description of its syntax and semantics, and then its data manipulation mechanisms are described.

1 Introduction

Attempts to provide a powerful platform for intelligent databases of the next generation have given rise to the new field of deductive object-oriented databases (hereafter DOOD). Object-oriented databases (hereafter OOD) and deductive databases (hereafter DD) both extend the power of relational model, albeit, in two orthogonal directions. Among other advantages, both OOD and DD offer the power of Turing machine in their query languages. But these two formalisms are not without limitations. The research on DOOD model aims at overcoming limitations in OOD and DD by combining the descriptive power of object-oriented data model with the manipulation power of deduction.

Object-oriented databases have evolved from application of object-oriented (hereafter OO) programming concepts in databases. Although there is an acknowledged lack of consensus with regard to a formal OO data model, an OOD system can be characterized by the following features [16, 5].

- *Object and Object Identifier:* Any real-world entity is uniformly modeled as a distinct object, and each object is associated with a unique identifier, or *oid*. Objects contain attributes and methods. *Complex objects* are built from simpler ones by applying constructors (*i.e.*, tuples, sets, bags, lists, arrays, etc.) to them.

- *Attributes and Methods:* Attributes are named data that describe the properties of an object, and methods are functions (program codes) that operate on the attributes of an object. Attributes specify the state of an object, whereas methods embody its behavior. The state and behavior encapsulated within an object are accessed or invoked from outside the object only through explicit message passing. *Encapsulation* enables better data abstraction and controlled information hiding.

- *Class and Instance:* A class is a grouping of all the objects which share the same set of attributes and methods. An object is an instance of the class which it belongs to. For example, object *john* can be an instance of class *gradStudent*.

- *Inheritance:* Classes are organized into a hierarchy. Classes lower in the hierarchy inherit the properties of those higher up. This hierarchy naturally induces (super)class/subclass associationship among classes. For instance, class *gradStudent* can be a subclass of class *student*, where an instance of class *gradStudent*, such as *john*, inherits the properties of class *student*. A class can be subclass to only one (super)class, implying *single inheritance*; or subclass to more than one (super)class, implying *multiple inheritance*.

- *Late Binding:* Given a method call on a variable, sometimes the appropriate method to be executed can only be determined at run time based on the object's actual class. This is known as *late* or *dynamic binding*. Method names can also be *overloaded* by defining method of the same name in more than one class. Overloading is resolved at compile time based on method signature. Methods can be *overridden*, too, by redefining implementation of its operation according to the type.

Although OO data model offers significantly increased data modeling capabilities, there are two major problems with this approach. One is the lack of logical or mathematical foundations for both the data model and the data manipulation language. The second one is the absence of a declarative query language, which, in one sense, has moved the database evolution back to pre-relational stage. Moreover, there is a school of thought arguing that OO systems can not have a declarative query language [23]. Examples of OOD systems include O_2, Ontos, Orion, ODE and Exodus (for references see [5]).

Deductive databases, stemming from a combination of relational database and logic programming paradigm, have also gained immense popularity among database researchers. This approach is based on predicate logic as a data model, often following a Prolog style of using Horn-clause, or if-then, rules to define predicates or relations. DD has a firm mathematical foundation inherited from logic programming and offers a declarative, more expressive and potentially complete query language and additional data structuring capabilities. Moreover, the same language can be effectively used for both programming and database manipulation. But deductive databases rely on a flat data model and do not support data abstraction and object identity. Examples of DD languages are Datalog [10], $\mathcal{LDL}$ [21] and CORAL [22].

Recent works [1, 15, 13, 11, 26, 4, 9] show that the drawbacks in both the approaches can be eliminated by combining these two different and potentially orthogonal paradigms into a common framework. Moreover, an efficient integration would yield a powerful new data model, DOOD, that would effectively incorporate the features of both the object and the deductive paradigm.

In this paper, we attempt a limited survey of the theoretical languages proposed for deductive object-oriented databases. We study the logics associated with the proposals and to what extent they support the object-oriented features without sacrificing declarative-ness.

The rest of the paper is organized as follows. In section 2, we introduce a toy application that would be used for illustrative purposes. Section 3 presents the general characteristics of a DOOD language. In sections 4 and 5, two of the major DOOD languages, *F-logic* [15] and *ORLOG* [13], are presented. Section 6 briefly discusses other DOOD languages, and finally we conclude in section 7 indicating directions of future research in this area.

2 A Programming Example

In this section we introduce a toy application, the *bill of materials* problem [3], and write program segments in (i) O_2, representing OO databases, and (ii) CORAL, a deductive database, for this application. To have an intuitive understanding of these programs the reader should have acquaintance with the OO programming style and SQL, and any logic programming language. Later we shall write programs for this problem in DOOD languages. This example will serve as the basis for comparing database languages from different paradigms.

2.1 The Bill of Materials Problem

The bill of materials of a certain product is a description of all the items that compose it, down to the lowest level of detail, where no further decomposition is possible. A component can appear several times within the bill of materials of a given main product, possibly at different depths in the subcomponent's tree. It represents a fragment of a manufacturing company's parts database, in particular, the way certain parts are manufactured out of other parts: the subparts that are involved in the manufacture of a part, the cost of manufacturing a part, the cost of manufacturing a part from its subparts, and the mass increment and decrement that occurs when subparts are assembled. Three tasks are defined on this database.

Task 1: Describe the database.

Task 2: Print the name, cost, and mass of every purchased part (base part) that costs more than $100.

Task 3: Print the total mass and total cost of a composite part.

2.2 Developing Bill of Materials in O_2

A class, *Part*, is defined with only one attribute, *name*. It has two subclasses: *Atomic_part* has three attributes, *name, price, and mass*; and *Derived_part* has these plus, *subparts*, which is a set of pairs of (part, quantity). The structure of all the classes except *Set_of_parts* is private. The *display_price* and *init* methods are public and can be accessed by any exterior methods. The method *display_price* calls the private method *price* and then displays the price, whereas *init* checks if a part with the same name already exists in the database. Task 2 is done by writing a simple query. For the third one, the price is calculated by the *price* method of the class *Part*. The program segment is shown in Figure 1.

Task: 1
```
add class Part type tuple(name: string)
        method price(): float,
                 init(): Part is public,
                 atomic(): Boolean is public,
                 display_price(): is public;

public read name in class Part;

add class Atomic_part inherits Part
     type tuple(price: float, mass: float)
        method price(): float,
                 atomic(): Boolean is public;

add class Derived_part inherits Part
     type tuple( incr_price: float,
               incr_mass: float,
               subparts:
                 set(tuple(part:Part, qty:integer)))
        method price(): float;

add class Set_of_parts type set(Part)
        method display() is public,
                 add_new_part() is public;

public * in class Set_of_parts;
add name set_of_parts: Set_of_parts;
execute co2 {set_of_parts = new( Set_of_parts);}
```

Task: 2
```
select part
from part in set_of_parts
where ([part atomic] and ([part price()] > 100))
```

Task: 3
```
body price(): float in class Atomic_part
co2 {return(self → price);}

body price(): float in class Derived_part
co2 { float p;
       o2 tuple(part: part, qty: integer) subpart;
       p = self → subparts)
       for (subpart in self → subparts)
       p+= ([subpart.part price]*subpart.qty);
       return(p);
       }

body display_price() in class Part
co2 { char * mes;
       int pid;
       mes = (char*) malloc(30);
       sprint(mes, "The price of a %s is %d",
                   self → name, [self price()]);
       [Dialoger message(mes)];
       }
```

Figure 1: An O_2 program for Bill of Materials problem.

2.3 Developing Bill of Materials in CORAL

Figure 2 shows the code of program written in CORAL. Programming in CORAL is very much similar to programming in a logic programming language, such as Prolog. The database is described by two predicative relations: *atomic_part*, which contains arguments for part-name, price, and mass of the part; and *derived_part*, which contains arguments for part-name, cost of manufacturing, mass increment, and the list of the names and quantities of the subparts. It shows that deductive databases are not as powerful as the OODs in modeling real-world data. The second task is implemented by a single goal rule. The last task is performed by a small program that evinces CORAL's ability to formulate complex queries in a simple declarative fashion. In O_2, we had to write a long and complicated code for this query.

Task: 1
```
atomic_part(Part, Price, Mass).
derived_part(Part, Price, MassIncrease, SubpartList).
```

Task: 2
```
task2(Part, Price, Mass):-
                    atomic_part(Part, Price, Mass),
                    Price > 100.
```

Task: 3
```
bom(Part, sum(<C>), sum(<M>)) :-
                    subpart_cost(Part, Subpart, C, M).
subpart_cost(Part, Part, Cost, Mass) :-
                    atomic_part(Part, Cost, Mass).
subpart_cost(Part, Subpart, Cost, Mass) :-
        derived_part(Part, Price, MassIncr, Subpart),
        bom(Subpart, TotalSubCost, TotalSubMass),
        Cost = Qty*TotalSubCost, Mass = Qty*TotalSubMass.
```

Figure 2: A CORAL program for Bill of Materials problem.

3 Characteristics of a DOOD Language

In this section, the position of a DOOD language among other database language classes is identified and the characteristics of a DOOD language are presented . The importance of DOOD class comes from the fact that it combines the benefits of deductive and OO paradigms. The deductive paradigm is specificational, high-level, extensible, modifiable, and easy to understand. The OO paradigm provides the richness of object structure and offers the potential of integration, through methods, with other languages.

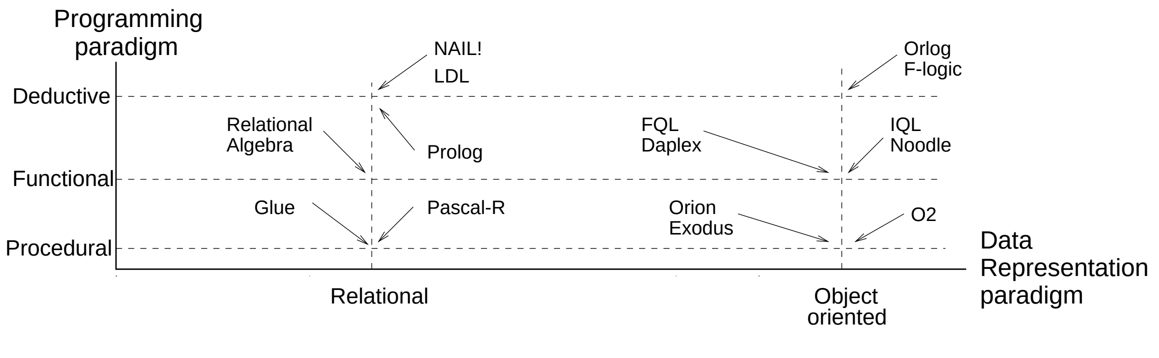

Figure 3: A Classification of Database Languages.

The database languages can be classified along two axes [15], as shown in Figure 3. The x-axis corresponds to data representation paradigm with relational at one extreme and object-oriented at the other. At the logical level in the relational paradigm, data is grouped by properties characterized by related attributes. For instance, information regarding the same person may be split among different relations, each describing different properties of person. At the logical level in the OO paradigm, data is grouped by objects. The vertical axis is for programming paradigm with deductive and procedural programming modes at the extremes, and the functional paradigm in between them. Together the axes classify six language types. Pascal-R and Glue are procedural relational languages. Deductive database languages, such as NAIL! or $\mathcal{LDL}$ are deductive relational languages. Relational algebra is functional relational, whereas FQL and Daplex (see [15] for references) are functional OO languages. Object-oriented database languages (e.g., O_2, Ontos, Orion, Exodus) are procedural OO languages. The sixth one is a new class that contains the DOOD languages.

The languages of the DOOD class are characterized by features from both the OOD and the DD formalisms. From OO paradigm they acquire object and object identity, complex data modeling constructs (set/list/bag), inheritance, encapsulation and dynamic binding. On the other hand, they inherit knowledge independence, increased power of interactive and ad-hoc query formulation in a declarative fashion, and support for the definition and design of knowledge management tools, from the deductive paradigm. *Knowledge independence* is the ability to handle generic knowledge about an application world (*i.e.*, derivation rules, integrity constraints) independently from the application program.

A DOOD model and language should specifically be based on a logic, for three reasons [24]. First, logic languages are inherently declarative. Second, this would enable the already developed deductive database technology (*i.e.*, deduction and integrity checking) to be directly used, and knowledge management tools (*i.e.*, consistency checker, explanation) to be effectively developed. Third and possibly the most important reason is that it would help to provide the logical foundation for the DOOD model. Logical foundation is significant because it (i) provides a clean declarative semantics for the underlying data model, (ii) enhances productivity by encouraging high level database programming, (iii) offers a hierarchical approach to implementation, and (iv) provides a basis for query optimization.

4 F-Logic

The proposal of *Frame-logic* or *F-logic* by Kifer and Lausen [14] characterizes the most fundamental departure from earlier works on deductive databases, as the new logic proposed incorporates such object-oriented features as object identity and inheritance. Over the years, F-Logic has been extended to encompass other OO features, such as the notions of complex object, inheritance, methods and schema in a natural way [15], and is considered as one of the best developed proposals of a formalism for deductive object-oriented database language. The work on this formalism has generated a whole suite of research in the field of DOOD.

4.1 Syntax and Semantics of F-logic

In this subsection, we present the syntax of F-logic and briefly discuss its semantics. F-logic has a sound and complete proof theory and a well-defined model theory. Readers are referred to [15] for details.

The alphabet of an F-logic language, $\mathcal{L}$, consists of:

- a set of *object constructors*, $\mathcal{F}$;
- an infinite set of *variables*, $\mathcal{V}$;
- auxiliary symbols, such as, (,),[,],$\rightarrow$,$\twoheadrightarrow$, $\bullet\!\!\rightarrow$, $\bullet\!\!\twoheadrightarrow$, $\Rightarrow$, $\Rrightarrow$, etc; and
- usual logical connectives and quantifiers, $\vee$, $\wedge$, $\neg$, $\forall$, $\exists$.

Object constructors play the role of function symbols in F-logic. Function symbols with arity 0 are the *constants*; symbols of arity ≥ 1 are used to construct larger terms out of the simpler ones. An *id-term* is a usual first-order term composed of function symbols and variables, as in predicate calculus. The set of all *ground* id-terms is denoted by $U(\mathcal{F})$, which is the *Herbrand Universe*. Conceptually, ground id-terms play the role of *logical oid*'s—a logical abstraction of the implementational concept of *physical object identity*. Among the auxiliary symbols, the double-shafted arrows, $\Rightarrow$ and $\Rrightarrow$ are used to specify types of attributes, while $\rightarrow$ and $\twoheadrightarrow$ specify values of attributes. The double-headed arrows, $\twoheadrightarrow$ and $\Rrightarrow$, are used in conjunction with set-valued attributes, while $\rightarrow$ and $\Rightarrow$ indicate scalar attributes. The arrows with a trailing bullet, $\bullet\!\!\rightarrow$ and $\bullet\!\!\twoheadrightarrow$, indicate that the values are inheritable.

A language of F-logic, an *F-language*, consists of a set of formulae constructed out of the alphabet symbols using the usual connectives and quantifiers. The simplest kind of formulas are called *molecular* F-formulas or *F-molecules*. An *F-molecule* is one of the following statements:

(i) An *is-a assertion* of the form $C::D$ or of the form $O::C$, where C, D, *and* O are id-terms.

(ii) An *object molecule* of the form O[a ';'-separated list of *method expressions*]. Here, O is an id-term denoting an object. A *method expression* can be either a *non-inheritable data expression*, an *inheritable data expression*, or a *signature expression*.

- *Non-inheritable data expressions* take one of the following two forms:
  - A non-inheritable *scalar* expression ($k \geq 0$): $ScalarMethod\ @\ Q_1,\ \ldots\ ,Q_k \rightarrow T$
  - A non-inheritable *set-valued* expression ($l,m \geq 0$): $SetMethod\ @\ R_1,\ldots,R_l \twoheadrightarrow \{S_1,\ldots,S_m\}$
- *Inheritable* scalar and set-valued data expressions are like non-inheritable expressions except that "$\rightarrow$" is replaced with "$\bullet\!\!\rightarrow$", and "$\twoheadrightarrow$" with "$\bullet\!\!\twoheadrightarrow$".
- *Signature expressions* also take two forms:
  - A *scalar* signature expression ($n,r \geq 0$): $ScalarMethod\ @\ V_1,\ldots,V_n \Rightarrow \{A_1,\ldots,A_r\}$
  - A *set-valued* signature expression ($s,t \geq 0$): $SetMethod\ @\ W_1,\ldots,W_s \Rrightarrow \{B_1,\ldots,B_t\}$

In the F-logic program shown in Figure 4, *atomic_part :: part* and *derived_part :: part* are *is-a assertions*, and the rules defining classes (*i.e., part, atomic_part, derived_part,* etc.) are *signature expressions*. The method *name* in class *atomic_part* is an example of *ScalarMethod*, while *subparts* in *derived_part* is that of a *SetMethod*.
Complex F-formulae are built out of simpler F-formulae by means of logical connectives and quantifiers:

- Molecular formulae are F-formulae;
- If φ and ψ are F-formulae, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$ are also F-formulae;
- If φ and ψ are F-formulae and X, Y are variables, then $\forall X \varphi$, $\exists Y \psi$ are F-formulae.

In F-logic, semantic structures are called *F-structures*. Given an F-language, $\mathcal{L}$, an *F-structure* is a tuple $\mathbf{I} = \langle\ U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\rightarrow}, I_{\bullet\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rrightarrow}\rangle$, where U is the domain of $\mathbf{I}$, $\prec_U$ is an irreflexive partial order on U, and $\in_U$ is a binary relation used to model class membership. By analogy with classical logic, U can be viewed as a set of all *actual* objects in a *possible world*, $\mathbf{I}$. Ground id-terms (the elements of $U(\mathcal{F})$) are interpreted by the objects in U via a mapping, $I_{\mathcal{F}}$, that interprets each k-ary object constructor, $f \in \mathcal{F}$, by a function $U^k \longrightarrow U$. For k=0, $I_{\mathcal{F}}(f)$ can be identified with an element of U. Thus, function symbols are interpreted the same way as in predicate calculus. The remaining six symbols in $\mathbf{I}$ denote mappings for interpreting each of the six types of method expressions in F-logic. First four of them are used to attach functions to methods and the remaining two attach types to methods.

4.2 F-logic Database

A database in F-logic is a set of formulae and a *query* is a statement of the form ?-Q, where Q is a F-molecule. The set of *answers* to this query with respect to an F-language program, $\mathbf{P}$, is the smallest set of F-molecules that (1) contains all instances q of Q that are found in the canonical model of $\mathbf{P}$, and (2) is closed under "$\models$".

Every F-language program or database can be split into three disjoint subsets, according to the type of information specified:

- The *IS-A hierarchy declaration* part of F-program consists of the rules whose head-literal is an is-a assertion. These declarations organise objects and classes into is-a hierarchy.
- The *signature declaration* part contains rules whose head literal is an object-molecule that is built out of signature molecules only. Signature declarations specify the types of arguments for each method and the type of the output they produce.

- The *object-base definition* part consists of rules whose head literals do not contain signatures or is-a expressions; *i.e*, rules whose heads are either predicates or object-molecules built exclusively out of data expressions.

An object-base definition specifies what each method is supposed to do. Object definitions may be explicit; *i.e.*, given as facts, or implicit; *i.e.*, specified by deductive rules. Collection of explicit rules corresponds to *extensional database* and the implicit rules form the *intensional database*, as in deductive databases [17]. See [15] for details of data modeling capabilities of F-logic.

4.3 Developing Bill of Materials in F-Logic

A program written in a DOOD language like F-Logic has the flavor of object-orientation blended into logic programming or deductive style, as shown in Figure 4. As the code for Task 1 shows, F-logic can describe data in an OO style. F-logic uses first-order terms to represent *oid*s and, unlike O_2, does not need system defined function *new* to generate new oids.

```
Task: 1
part[name ⇒ {string}; price ⇒ {float};
       init ⇒ {part}; atomic ⇒ {boolean};
       display_price ⇒ {}].

atomic_part[price ⇒ {float}; mass ⇒ {float}; atomic ⇒ {boolean}].

derived_part[incr_price ⇒ {float}; incr_mass ⇒ {float};
              subparts ⇒⇒{parts[part⇒ part; qty ⇒ {integer}]};
              price ⇒ {float}].

parts[part ⇒ part; qty ⇒ {integer}].

set_of_parts[parts ⇒⇒{part}; display⇒{}; add_new_part ⇒ {}].
atomic_part :: part.
derived_part :: part.
set_parts : set_of_parts.

Task: 2
X[] ← X: set_of_parts, X[atomic → true; price → Y], Y > 100.

Task: 3
Z[price → X] ← Z:atomic_part[price → X].
Z[price → X] ← Z:derived_part[subparts →→< Y : parts[part→ P;
                       qty → Q]>], P[price → Pr], X=sum(Pr*Q).
```

Figure 4: An F-Logic program for Bill of Materials problem.

The declaration *type tuple* in O_2 has no counterpart in F-logic . Attributes and methods are treated uniformly and are listed side by side with no particular order. F-logic does not have a logical semantics for encapsulation and treats everything as public. Task 2 is performed by a simple rule.

The program segment for Task 3 is short, simple and declarative. It shows dynamic binding also. While trying to find the price of an object, it uses the code according to the status of the object. If it is an atomic object then it uses the code for the *price* method defined for atomic objects, and so on. This choice of method is not made until execution time. It is clearly evident that F-logic can capture benefits of object-orientation without ceasing be to declarative.

In *price* method of Task 3, a *grouping function* on *parts*—that is, for a particular *derived_part* object the set of objects parts are available in rule body—is assumed. The goal *P[price → Pr]* finds the prices of the subparts that depend on their type (atomic or derived), and X contains the subtotal price of all the subparts of a derived part after grouping. Grouping is an extra-logical feature of F-logic. This feature comes with commercial programming tools. For example CORAL has this feature.

5 ORLOG

ORLOG, or *O*bject *R*elationship *Log*ic, is the logical language associated with the *Object Relationship*, or *OR*, data model proposed by Jamil and Laksmanan [13]. The OR model extends conventional semantic data models (SDM) by incorporating OO features. In addition to SDM and OO features, OR model incorporates the notions of *withdrawal, stratified constraints, values, relationships, method polymorphism,* etc. A notion of *admissible databases* has been introduced to capture the spirit of encapsulation. ORLOG has been implemented in $\mathcal{LDL}$ using a translational approach [12]. Like F-logic, ORLOG features a higher order syntax but provides a first-order declarative semantics and a sound logical foundation for the OR model.

5.1 Syntax and Semantics of ORLOG

The alphabet of language $\mathcal{L}$ of ORLOG consists of (1) an infinite set, $\mathcal{I}_{\mathcal{O}}$, of many-sorted symbols to be used as *oid*s, (2) an infinite set, $\mathcal{R}$, of typed predicate symbols, (3) an infinite set, $\mathcal{C}$, of many-sorted symbols for individual constants, (4) an infinite set, $\mathcal{M}$, of function symbols for method names, (5) an infinite set, $\mathcal{V}_{\mathcal{O}}$, of many-sorted id variables, (6) an infinite set, $\mathcal{V}_{\mathcal{B}}$, of many-sorted symbols for basic domain or individual variables, (7) a finite set, $\mathcal{T}_{\mathcal{B}}$, of symbols for basic type names, and (8) an infinite set, $T_O \subseteq \mathcal{I}_{\mathcal{O}}$, of symbols for objects or constructed type names. The alphabet also includes conventional logical connective and quantifier symbols.

Formulae in ORLOG are built from the set of terms $\mathbf{T}$ using logical connectives and quantifiers. The set $\mathbf{T}$ consists of:

- An *is-a* term, of the form P **Is-a** Q where P and Q are *oid*s, and is used to define subclass relationships or hierarchy associations among objects.
- An *object* term, or *o-term*, is of the form $P[E_1, \ldots, E_n], n \geq 1$, where P is an *oid*, and each E_i is an object value term. P is called the *descriptor* of the o-term.
- A *relationship* term, or *r-term* is of the form $R[E_1, \ldots, E_n], n \geq 1$, where R is a predicate symbol of arity n in $\mathcal{R}$, and each E_i is a relationship value term.
- A *constraint* term, or *c-term*, is of the form $P(T_1, \ldots, T_n), n \geq 1$, which represents typing constraints for methods, where each T_i is an object type term when P is an *oid*, and is a relationship type term when P is a predicate symbol in $\mathcal{R}$.
- A *participation* term, or *p-term*, is of the form $R \diamond [!]T_1, \ldots, [!]T_n$, where $R \in \mathcal{R}$ and $T_i \in \mathcal{T}_{\mathcal{O}}$, for $i = 1, \ldots, n$, and is used to represent relationship among classes.

A semantic structure $\mathcal{S}$ of the language $\mathcal{L}$ of ORLOG is a 4-tuple $\langle \mathcal{D}, \prec_{\mathcal{O}}, I_d, I_s \rangle$, where $\mathcal{D}$ is the domain of interpretation, $\prec_O$ is a partial order on domain $\mathcal{D}$, which defines a recursive and acyclic subclass association on the objects in the domain, and I_d and I_s are two interpretation functions which assign meaning to the symbols in $\mathcal{L}$. Interested readers are referred to [13] for details.

5.2 ORLOG Database

An *ORLOG database*, or *program*, **D**, is a set of ORLOG Horn-clause formulae which includes object/relationship definitions, subclass definitions, type definitions, and method definitions. Objects, associations between objects, basic and complex values, and constraints are the primitive building blocks of an ORLOG database. Unlike F-logic, which treats them as objects, basic values in ORLOG are just values and do not participate in any hierarchy. ORLOG took the view that encapsulation should hide both structural and behavioral aspects of objects. The notion of private properties has been introduced to achieve structural encapsulation, and by hiding the details of implementation of methods behavioral encapsulation is achieved. The associations used in ORLOG are is-a (or specialisation-generalisation) hierarchy, aggregation (complex typing), and relationships. The *inheritance model* is different from other models due to its non-monotonic nature and other features.

5.3 Developing Bill of Materials in ORLOG

As evident from Figure 5, the ORLOG program for this particular example is very similar to the corresponding F-logic program. But it exemplifies one major difference between the two languages; *i.e.*, in inheritance modeling. In the rule body of each clause, ":"s are replaced by "::"s to account for inheritance in ORLOG. The main difference between ORLOG and F-logic regarding inheritance is that F-logic uses the idea of one step inheritance, which is a meta-logical notion, while inheritance is built into the semantics and proof theory of ORLOG. This implies that inheritance in F-logic is a programming technique, not part of the semantics, while in ORLOG it is in the semantics and does not matter how a programmer writes a program. As in F-logic, ORLOG treats everything as public.

```
Task: 1
part[name ⇒ {string}; price ⇒ {float};
      init ⇒ {part}; atomic ⇒ {boolean};
      display_price ⇒ {}].

atomic_part[price ⇒ {float}; mass ⇒ {float}; atomic ⇒ {boolean}].

derived_part[incr_price ⇒ {float}; incr_mass ⇒ {float};
            subparts ⇒⇒{parts[part⇒ part; qty ⇒ {integer}]};
            price ⇒ {float}].

parts[part ⇒ part; qty ⇒ {integer}].

set_of_parts[parts ⇒⇒{part}; display⇒{}; add_new_part ⇒ {}].
atomic_part :: part.
derived_part :: part.
set_parts : set_of_parts.

Task: 2
X[] ← X:: set_of_parts, X[atomic → true; price → Y], Y > 100.

Task: 3
Z[price → X] ← Z::atomic_part[price → X].
Z[price → X] ← Z::derived_part[subparts →→< Y : parts[part→ P;
                qty → Q]>], P[price → Pr], X=sum(Pr*Q).
```

Figure 5: An ORLOG program for Bill of Materials problem.

6 Other DOOD Languages

In this section, we briefly discuss specific features of a number of other DOOD languages.

Noodle [20] is a general declarative query language for object-oriented databases developed at the AT&T Bell Laboratories. This is not an all-purpose database manipulation language; rather it is intended to be integrated with a procedural data manipulation language, O++ of the ODE database system. Noodle is rule-based, relations are supported as objects with oids, but individual tuples of a relation do not have oids. Complex objects (functional and set-valued) are supported using Hilog-like constructs. Attributes of objects are referenced by name rather than position. Variables can range over all classes, relations, attributes, and objects.

$\mathcal{QUIXOTE}$ [26] has been developed for knowledge information processing applications in the Japanese Fifth Generation Computer System project. In this language, integration of logic and OO features is done using *subsumption constraints*, and knowledge classification and inheritance are incorporated by a *module* concept. $\mathcal{QUIXOTE}$ supports query processing for partial information databases, abduction of subsumption constraints and hypothetical reasoning. It also facilitates support of such database management system features as concurrency control, nested transaction, and persistence. In $\mathcal{QUIXOTE}$, an object consists of an oid and a set of properties. Each property can be considered as a method and the implementation of a method is written in the body of a rule. A *module* is a set of rules and each method is identified by a *module identifier* or *mid*. Inheritance is achieved by using submodule relationships with the help of *mid*s. Rules without a *mid* are inherited by all modules. A $\mathcal{QUIXOTE}$ *database* or *program* is defined as a triple (S,M,R) of a set S of subsumption relations among basic objects, a set M of submodule relations among *mid*s, and a set R of rules.

Abiteboul et al. describe a language, $datalog^{meth}$ [1], where three different translations to Datalog capture the semantics of the language with monotonic, static and dynamic inheritance respectively. With monotonic inheritance there is no overriding and inherited values are accumulated. Static inheritance is like inheritance in procedural languages, overriding occurs independent of the value of the body of the clause. With dynamic overriding, a clause overrides only if the body of the clause is true. They introduce views into the language using virtual classes and show how dynamic inheritance can be simulated using static inheritance and views. They extend $datalog^{meth}$ to $datalog^{meth,flex}$ to reason about class hierarchy.

Gulog [11] was initially designed to investigate the interaction of deduction, inheritance, and overriding in the simplest possible context. Later, it was extended to incorporate set-valued methods and multiple inheritance, demonstrating the ease of such extentions. The language of Gulog is loosely based on LIFE [2] and F-logic. In Gulog, schema declarations are separate from data definitions. Objects and types (or classes) are distinguished and method overriding can be defined on specific instances of a subtype.

In **L&O** [18], another deductive object-oriented language, the semantics of inheritance and overriding is given indirectly by translating L&O programs to logic programs and, hence, it provides little insight into the relationship between inheritance, overriding and deduction. Query processing is also provided using a translation to Prolog.

In a recently proposed language [8], the notion of *inheritance by completion (i-completion)* is introduced in order to account for inheritance in stratification. Reasoning on the i-completion of programs in the proposed language, the authors show that *i-stratification*—a certain syntactic condition—is sufficient to guarantee the existence and uniqueness of the stable models of programs supporting inheritance and overriding.

Meseguer and Qian propose a semantic foundation for OOD based on a simple logic of change called *rewriting logic* and a language called **MaudeLog** that is based on that logic [19]. MaudeLog attempts an integration of query, update and general purpose programming language. One significant difference of MaudeLog from others is that its syntax is not higher-order. Higher-order features are provided through parameterization and module inheritance mechanism. Meta-data is dealt with using module hierarchies, parameterized modules, module expressions, and theory interpretations. Views are related to theory interpretations, and therefore, handled without any need for higher-order logics.

DTL, or *DataTypeLog* [4], is a typed logical query language designed to be used with databases specified in TM, a high-level specification language for OO database schemas. DTL is an extension of Datalog with complex objects, object identity and multiple inheritance. DTL incorporates predicative sets as first-class objects. The formal language of DTL, called FDTL, supports arbitrarily nested records, variant records, sets and lists. It is also possible to define static constraints of different granularity (*i.e.*, at the object, class or database level). DTL employs full static type-checking. Answers to DTL queries result in a set of homogenous elements, in the sense that these elements all have the same minimal type with respect to subtyping. This approach differs from that of F-Logic, where basically no distinction is made between types and instantiations of types. As in F-logic and ORLOG, the subclass relationship is represented by *is-a* hierarchy in DTL.

The type inconsistency problem is also addressed in **ERCLog** [25]. ERCLog is a stratified OO Datalog with negation, data functions and invention of new oids. ERCLog is the language associated with a data model, ERC+, based on the entity-relationship approach. ERC+ supports complex structured objects, object identities and multi-instantiations, in addition to supporting the basic ER concepts; *i.e.*, entity, relationship and attribute. One of the features that is unique in ERCLog is that types in this language are extended with cardinality constraints, and thus the mono- and multi-valued types become the special kinds of cardinality constrained types, and they are treated similarly. Objects are cardinality restricted set-valued. Since ERCLog explicitly manages the cardinalilties of the attributes of the objects and the links between objects, it allows the language static checking of the consistency of the program.

The logical rendition of encapsulation is not as simple as the concept of it. Both F-logic and ORLOG failed to accomodate it in their semantics. Bugliesi and Jamil have developed a language where encapsulation is built into the syntax, semantics and the proof theory of the language [7]. The major contribution of this work is that it shows the equivalence of the different characterizations—under proof theory, model theory and fixed-point theory—of the semantics of encapsulation. The logical foundation of encapsulation is studied by introducing a mapping from the language developed to a corresponding modal logic, and the language is shown to be sound and complete.

7 Conclusion

In this paper, we have attempted a limited survey of the new class of languages associated with deductive object-oriented data models. This survey indicates that two different approaches are taken by researchers to design logic languages which combine OOD and DD features. One is to extend existing deductive database languages, like Datalog, with object-orientation and the other one is to build new logical theories to encompass the DOOD features. It seems that the later approach leads to better integration.

Almost all the proposed languages have a higher-order syntax but a first-order semantics. This is because first-order logic is not exactly suitable to model complex objects and higher-order logic is better for object-oriented data models. But it is difficult to have a sound and complete proof theory for higher-order logic, and, therefore, higher-order logic is not computationally feasible. A logic with a higher-order syntax and a first-order semantics is a trade-off between these two constraints.

Attempts to integrate OOD and DD show the feasibility of such integration from a theoretical perspective, indicating a challenging research direction in the development of new generation of intelligent databases. The DOOD languages are only query language and are not capable of performing updates in databases. It would be interesting to investigate whether a logical update language is possible by combining DOOD language, like F-logic, with recently proposed logical theory for database programming, *Transaction Logic* [6]. Inheritance in F-logic needs further study, too. Above all, practical systems on DOOD model and employing DOOD languages should be developed to verify the achievements accomplished on theoretical playgrounds.

References

[1] Abiteboul, S., Lausen, G., Uphoff, H., and Waller, E. Methods and rules. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data* (Washington DC, 1993), pp. 32–41.

[2] Ait-Kaci, H., and Podelski, A. Towards a meaning of LIFE. *Journal of Logic Programming 16* (August 1993), 195–234.

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

[4] Bal, R., and Balsters, H. A deductive and typed object-oriented language. In *Proceedings of the 3rd Int. Conf. on DOOD* (1993), S. Ceri, K. Tanaka, and S. Tsur, Eds., LNCS 760, Springer, pp. 340–359.

[5] Bancilhon, F., Delobel, C., and Kanellakis, P., Eds. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[6] Bonner, A. J., and Kifer, M. An overview of transaction logic. *Theoretical Computer Science 133*, 2 (October 1994), 205–266.

[7] Bugliesi, M., and Jamil, M. H. A logic for encapsulation in object oriented languages. In *Proceedings of the 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLIPL)* (Madrid, Spain, 1994), LNCS 844, Springer, pp. 215–229.

[8] Bugliesi, M., and Jamil, M. H. A stable model semantics for behavioral inheritance in deductive object oriented languages. In *Proceedings of the 5th Int. Conf. on Database Theory (ICDT)* (Prague, Czech Republic, January 1995), G. Gottlob and M. Y. Vardi, Eds., LNCS 893, Springer-Verlag, pp. 222–237.

[9] Cacace, F., Ceri, S., Crespi-Reghizzi, S., Tanca, L., and Zicari, R. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data* (Atlantic City, NJ, May 1990), pp. 225–236.

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

[11] Dobbie, G., and Topor, R. A model for sets and multiple inheritance in deductive object-oriented systems. In *Proceedings of the 3rd Int. Conf. on DOOD* (1993), S. Ceri, K. Tanaka, and S. Tsur, Eds., LNCS 760, Springer, pp. 473–488.

[12] Jamil, H. M., and Lakshmanan, L. V. S. Realizing ORLOG in $\mathcal{LDL}$. In *Proceedings of the ACM SIGMOD Workshop on Combining Declarative and Object-Oriented Databases held in conjunction with the ACM PODS and SIGMOD Conference* (Washington DC, May 1993), pp. 45–59.

[13] Jamil, M. H., and Laksmanan, L. V. S. ORLOG: A logic for semantic object-oriented models. In *Proceedings of the 1st Int. Conf. on Information and Knowledge Management (CIKM)* (Baltimore, USA, November 1992), pp. 584–592.

[14] Kifer, M., and Lausen, G. F-logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data* (Oregon, USA, 1989), ACM, pp. 134–146.

[15] Kifer, M., Lausen, G., and Wu, J. Logical foundation of object-oriented and frame-based languages. Tech. Rep. TR 93/06, CS, SUNY at Stony Brook, Stony Brook, NY, April 1993. Revised in March 1994 and accepted for Journal of the ACM, 1995.

[16] Kim, W. Object-oriented databases: Definition and research directios. *IEEE Transactions on Knowledge and Data Engineering 2*, 3 (September 1990), 327–341.

[17] Lloyd, J. W. *Foundations of Logic Programming*, 2nd edition ed. Springer Verlag, 1987.

[18] McCabe, F. G. *Logic and Objects*. Prentice Hall International, London, 1992.

[19] Meseguer, J., and Qian, X. A logical semantics for object-oriented databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (Washington DC, 1993), pp. 89–97.

[20] Mumick, I. S., and Ross, K. A. Noodle: A language for declarative querying in object-oriented database. In *Proceedings of the 3rd Int. Conf. on DOOD* (1993), S. Ceri, K. Tanaka, and S. Tsur, Eds., LNCS 760, Springer, pp. 360–374.

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

[22] Ramakrishnan, R., Srivastava, D., and Sudarshan, S. CORAL: Control, relations and logic. In *Proceedings of the 18th Conf. on Very Large Data Bases* (Vancouver, 1992), pp. 238–250.

[23] Ullman, J. D. A comparison between deductive and object-oriented database systems. In *Second Int. Conf. on DOOD* (1991), C. Delobel, M. Kifer, and Y. Masunaga, Eds., LNCS 566, Springer, pp. 263–277.

[24] Vieille, L. What is the true path to integrating object-oriented and declarative languages. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases* (Washington DC, May 1993), pp. 14–15. Position statement of panel discussion.

[25] Ye, X., Parent, C., and Spaccapietra, S. Cardinality consistencies of derived objects in DOOD systems. In *Proceedings of the 13th Int. Conf. on the Entity Relationship Approach* (Manchester, UK, December 1994).

[26] Yokota, K., Tsuda, H., and Morita, Y. Specific features of a deductive object-oriented database language $\mathcal{QUIXOTE}$. In *Proceedings of the ACM SIGMOD Workshop on Combining Declarative and Object-Oriented Databases held in conjunction with the ACM PODS and SIGMOD Conference* (Washington DC, May 1993), pp. 89–99.