A Logical Foundation for Deductive Object-Oriented Databases

Mengchi Liu
Department of Computer Science
University of Regina
Canada
mliu@cs.uregina.ca

Gillian Dobbie, Tok Wang Ling
School of Computing
National University of Singapore
Singapore
{dobbie, lingtw}@comp.nus.edu.sg

Abstract

Over the past decade, a large number of deductive object-oriented database languages have been proposed. The earliest of these languages had few object-oriented features, and more and more features have systematically been incorporated in successive languages. However, a language with a clean logical semantics that naturally accounts for all the key object-oriented features, is still missing from the literature. Two features that are currently missing are the encapsulation of rule-based methods in classes, and non-monotonic behavioral inheritance with overriding, conflict resolution and blocking. This paper introduces the syntax of a language with these features. It then defines a class of databases, called well-defined databases, that have an intuitive meaning and develops a direct logical semantics for this class of databases. The semantics is based on the well-founded semantics from logic programming. The work presented in this paper establishes a firm logical foundation for deductive object-oriented databases.

1 Introduction

The objective of deductive object-oriented databases is to combine the best of the deductive and object-oriented approaches, namely to combine the logical foundation of the deductive approach with the modeling capabilities of the object-oriented approach. Based on the deductive object-oriented database language proposals as well as the work in object-oriented programming languages and data models, it is becoming clear that the key object-oriented features in deductive object-oriented databases include object identity, complex objects, typing, rule-based methods, encapsulation of methods, overloading, late binding, polymorphism, class hierarchy, multiple behavioral inheritance with overriding, blocking, and conflict handling. However, a clean logical semantics that naturally accounts for all the features is still missing from the literature. In particular the encapsulation of rule-based methods in classes, and non-monotonic multiple behavioral inheritance have not been addressed properly so far.

In object-oriented programming languages and data models, methods are defined using functions or procedures and are encapsulated in class definitions. They are invoked through instances of the classes. In deductive object-oriented databases, we use rules instead of functions and procedures. By analogy, methods in deductive object-oriented databases should be defined using rules and encapsulated in class definitions. Such methods should be invoked through instances of the classes as well. However, most existing deductive object-oriented database languages, including F-logic [9], IQL [1], Datalog^{meth} [2], ROL [12], Datalog^{++} [8], do not allow rule-based methods to be encapsulated in the class definitions. The main difficulty is that the logical semantics is based on programs that are sets of rules. If rules are encapsulated into classes, then it is not clear how to define their semantics. Several proposals such as Datalog^{meth} and Datalog^{++} provide encapsulation but use rewriting-based semantics which do not address the issue directly. The authors of [3] address encapsulation but do not include other important object-oriented features, like inheritance.

Non-monotonic multiple behavioral inheritance is a fundamental feature of object-oriented data models such as O_2 [5] and Orion [10]. The user can explicitly redefine (or override) the inherited attributes or methods and stop (or block) the inheritance of attributes or methods from superclasses. Ambiguities may arise when an attribute or method is defined in two or more superclasses, and the conflicts need to be handled (or resolved). Unfortunately, a logical semantics for multiple inheritance with overriding, blocking and conflict-handling has not been defined. The main difficulty is that the inherited instances of a superclass may not be well-typed with respect to its type definition because of overriding and blocking. Most deductive object-oriented database languages, including F-logic[1], LOGRES [4], LIV-

[1] F-logic however supports indeterminate non-monotonic default value

ING IN LATTICE [7], COMPLEX [6], only allow monotonic multiple structural inheritance, which is not powerful enough. Some deductive object-oriented languages such as Datalog^{meth} only support non-monotonic single inheritance by allowing method overriding. One extreme case is IQL, which does not support multiple inheritance at the class level at all. Instead, it indirectly supports it at the instance level via the union type so that inherited instances of a superclass can still be well-typed with respect to its type definition which is the union of the type for its direct instances and the type for its non-direct instances. ROL has a semantics that accounts for non-monotonic multiple structural inheritance with overriding and conflict-handling in a limited context, but without blocking. Datalog^{++} takes a quite different approach towards non-monotonic inheritance. It disallows the inheritance of conflicting attributes and methods, like in C++. It provides mechanisms for the user to block the inheritance of attributes and methods. However, it only provides an indirect, rewriting-based semantics for such non-monotonic inheritance.

This paper provides a direct well-defined declarative semantics for a deductive object-oriented database language with encapsulated rule-based methods and non-monotonic behavioral inheritance with overriding, conflict resolution and blocking. In order to keep the setting simple, we omit some well understood features that don't affect the semantics described, e.g. set-valued attribute values, and we focus on a static database rather than a dynamic database, (see [13] for the semantics of updates to the database). In the language, methods are declared in the class definitions, and the methods are invoked through instances of the classes. We introduce a special class, *none*, to indicate that the inheritance of an attribute or method in a subclass is blocked i.e. it won't be inherited from its superclasses. We provide a very flexible approach to conflict resolution. Our mechanism consists of two parts. One part, the default part is similar to the method used in Orion, namely a subclass inherits from the classes in the order they are declared in the class definition. The other part allows the explicit naming of the class the attribute or method is to be inherited from. Therefore, a subclass can inherit attribute or method definitions from any superclasses. We then define a class of databases, called well-defined databases, that have an intuitive meaning and develop a direct logical semantics for this class of databases. The semantics naturally accounts for method encapsulation, multiple behavioral inheritance, overriding, conflict handling and blocking, and is based on the well-founded semantics [16] from logic programming. We define a transformation that has a limit, $\mathcal{I}^*$ for well-defined databases, and prove that $\mathcal{I}^*$, if it is defined, is a minimal model of the database.

inheritance. The value inherited depends on which inheritance step is done first at run time.

This paper is organized as follows. We introduce the syntax and semantics of the language using an example in Section 2. In Section 3 the class of well-defined databases and the semantics of well-defined databases are defined, and the main results are presented. Section 4 concludes the paper, reiterating our results and comparing this work with related work. Due to space limitation, the paper is quite terse and we have omitted proofs. They are included in [14].

2 Example

Our language in fact supports many of the important object-oriented features in a rule-based framework with a well-defined declarative semantics. In this section, we introduce and demonstrate concepts that are important in the paper. A more extensive description of the syntax can be found in [14].

The schema in Figure 1(a) defines four classes, *person*, *employee*, *student*, and *wstudent* (working student). The class *person* has three attributes, *name*, *birthyear*, and *spouse*, and two methods: *married(person)* and *single()*. The attribute *birthyear* has a default value of 1945. Method *married(X)* is true if the person the method is applied to has a spouse, X, and method *single()* is true if the person is not married. Notice, that the semantics of negation are defined using extended negation as failure [11]. The class *employee* inherits all attribute declarations, default values and method declarations from class *person* unless they are blocked or overridden in class *employee*. We say that class *employee* is a direct subclass of *person* and *person* is a direct superclass of *employee*. New attributes can also be declared in subclasses. The attribute declarations for *name*, *birthyear*, and *spouse*, and the method declarations for *married(person)*, and *single()* are inherited but the default value of *birthyear* is overridden in *employee*, i.e., the default value for attribute *birthyear* is redefined to 1960. The class *student* also inherits from *person*. Two methods are declared in *student*, namely *extrasupport()* and *support()*.

The class *wstudent* inherits from two classes, *employee* and *student*. With multiple inheritance, there can be conflicting declarations i.e. default values, attributes and methods may be declared in more than one superclass. There is one possible conflict to be resolved in *wstudent*, default value *birthyear* is defined on both *employee* and *student*. There are two ways that conflicts can be resolved. A conflict resolution declaration indicates explicitly which class a property is to be inherited from e.g. *birthyear* $\lhd$ *student* indicates that the definition of *birthyear* and the default value 1970 are inherited from *student*. If there is a conflict and there is no conflict resolution declaration then the property is inherited from the superclasses in the order the superclasses are listed in the class declaration. Notice that

Key
$\Rightarrow$ *type declaration*
$\rightarrow$ *value declaration*
$\bullet\!\!\rightarrow$ *default value*
$\lhd$ *explicit inheritance*

```
class person [
    name ⇒ string;
    birthyear ⇒ integer;
    birthyear •→ 1945;
    spouse ⇒ person;
    married(person) {married(X) :- spouse → X}
    single() {single() :- ¬married(X)}
    ]
class employee isa person [
    birthyear •→ 1960;
    ]
class student isa person [
    birthyear •→ 1970;
    extrasupport() ⇒ integer {
        extrasupport() → 1000 :- married(X),
            student X;
        extrasupport() → 500 :- married(X),
            ¬student X;
        extrasupport() → 100 :- single()}
    support() ⇒ integer {
        support() → S :- extrasupport() → S1,
            S = 1000 + S1}
    ]
class wstudent isa employee, student [
    birthyear ◁ student;
    support() ⇒ none;
    extrasupport() ⇒ person {
        extrasupport() → X :- spouse → X}
    ]
```

(a) *Schema*

```
employee tom  [name → "Tom"; birthyear → 1963;
               spouse → pam]
student   sam [name → "Sam"]
wstudent pam  [name → "Pam"; spouse → tom]
```

(b) *Instance*

Figure 1. Sample Database

the method *support*() is blocked in *wstudent* (i.e. its return type is *none*), and the method *extrasupport*() in *wstudent* overrides the method *extrasupport*() in *student*. A method declaration in a subclass overrides a method declaration in a superclass if the methods have the same signature, independent of their return values. A method has the same signature as another method if the method has the same method label and the same arguments, e.g. *extrasupport*() in *student* has the same signature as *extrasupport*() in *wstudent*. While classes *employee* and *student* are direct superclasses of *wstudent*, *person* is an indirect superclass of *wstudent*.

The instance in Figure 1(b) contains three objects with oids *tom*, *sam*, and *pam*. In the database instance, each object is associated with a class and attributes are assigned values. For example, object *tom* is a direct instance of *employee*, and the value of its attribute *name* is *"Tom"*. The value of attribute *birthyear* is 1963, i.e. the default 1960 in *employee* is not used. The value of its attribute *spouse* is object identifier *pam*. We say that *employee* is the primary class of object *tom*, and object *tom* is an non-direct instance of *person*. The *birthyear* of *sam* is 1970, i.e. the default in class *student* is used because a value for attribute *birthyear* is not provided in object *sam*. The value of attribute *birthyear* is not given in object *pam*, nor in class *wstudent*. The default value 1970 is inherited from *student* because there is a conflict resolution declaration in *wstudent*.

We can ask the following queries on the sample database in Figure 1. The queries demonstrate how methods are encapsulated in classes, i.e. a method is declared in a class and invoked through instances of the class.

1. Find the birthyear of Sam.

   $?\text{-}\ student\ O[name \rightarrow \text{"}Sam\text{"};\ birthyear \rightarrow X]$

   The default value of *birthyear* for instances in class *student* is returned, $X = 1970$.

2. Find what support Sam gets.

   $?\text{-}\ student\ O[name \rightarrow \text{"}Sam\text{"};\ support() \rightarrow X]$

   The *support*() method in class *student* invokes the *extrasupport*() method. The *extrasupport*() rules in turn invoke the *married*(*person*) and *single*() methods defined in class *person*. As Sam has no spouse, Sam is not married, so Sam is single, and the third rule for *extrasupport*() is used. The *extrasupport*() that Sam receives is 100, so $X = 1100$ is returned.

3. Find what support Pam gets.

   $?\text{-}\ wstudent\ O[name = \text{"}Pam\text{"};\ support() \rightarrow X]$

   This method *support*() is blocked on *wstudent*, an error message indicating that this method is undefined is returned.

4. Find all students whose extra support is not 500.

   $?\text{-}\ student\ O\ [extrasupport() \rightarrow X], X <> 500$

   This query returns the oids of all the objects that belong to class *student* or subclasses of *student* whose value for method extrasupport is not 500. The answer is $\{O = sam\}$.

We make the following observations. Two kinds of classes are distinguished: value classes and oid classes. There are two special value classes, `none` and `void`. Class `none` is used to indicate that the inheritance of an attribute or method from a superclass is blocked in a subclass. Class `void` has only one value, namely `nil`, which is returned by a method if no other value is returned. Like in C++ and Java, we have a special variable, `This`, that is used to refer to the current object. Variables are represented throughout the paper using uppercase alphabetic characters.

A *schema* K is a set of class declarations, which can be represented abstractly as a tuple $K = (C, isa, \alpha, \delta, \mu, \chi)$ where C is a finite set of oid classes, isa is a finite set of superclass declarations, α is a finite set of attribute declarations, δ is a finite set of default value declarations, μ is a finite set of method declarations, and χ is a finite set of conflict resolution declarations. For simplicity, we assume that there is no abbreviation in α, δ, and μ. We write $\alpha(c)$, $\delta(c)$, $\mu(c)$, $\chi_\alpha(c)$ and $\chi_\mu(c)$ for the sets of attribute, default value, method, attribute conflict resolution declarations, and method conflict resolution declarations in α, δ, μ, and χ for the class c respectively. We impose constraints on the schema to capture the intended semantics of multiple inheritance with overriding, blocking and conflict handling. An *instance* I is a set of object declarations, that can be represented as a tuple $I = (\pi, \lambda)$ where π is a set of ground oid membership expressions called *oid assignments* and λ is a set of ground positive attribute expressions called *attribute value assignments*. A *database* $\mathcal{DB}$ consists of two parts: the schema K and the instance I, which can be represented abstractly as a tuple $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ where $K = (C, isa, \alpha, \delta, \mu, \chi)$ and $I = (\pi, \lambda)$. See Figure 1. A *query* is a sequence of expressions prefixed with ?-.

3 Semantics

In this section, we define the semantics of a database and queries. First we give the meaning of the schema and instance of the database, then we identify a class of databases, called well-defined databases, and finally, we define the meaning of the rule based methods of well-defined databases, based on the meaning of the schema and instance. The semantics of a database is based on the well-founded semantics except in this case the semantics of the rule-based methods must take into account the meaning of the schema and the instance of the database.

3.1 Semantics of Schema and Instance

Encapsulation is dealt with in this subsection; each attribute, default value and method that are applicable to a class are identified. In order to determine which attributes, default values, and methods are applicable to a class, it is necessary to consider inheritance with overriding, blocking and conflict handling. Recall that $\alpha(c)$, $\delta(c)$ and $\mu(c)$ are the sets of attribute declarations, default value and method declarations respectively that are *defined* on c. In this section, we define $\alpha^*(c)$, $\delta^*(c)$, and $\mu^*(c)$, the attribute declarations, default value and method declarations that are *applicable* to class c, taking inheritance, overriding, conflict resolution and blocking into account.

In [14], we define `difference operators` that find the attribute declarations (default value declarations, method declarations respectively) that are defined on one class and not redefined on another class. Consider the database in Figure 1. The difference between the sets of attribute declarations for *person* and *student* is:

$\alpha(person) - \alpha(student) = \{$
$\quad person\ [name \Rightarrow string],$
$\quad person\ [birthyear \Rightarrow integer],$
$\quad person\ [spouse \Rightarrow person],$
$\}$

The result is the attribute declarations in *person* that are not redefined in *student*. In Figure 1 the difference between the default attribute declarations for *person* and *student* is:

$\delta(person) - \delta(student) = \emptyset$

This is not surprising because the default value for *birthyear* is redefined in *student*.

The following definition outlines an algorithm to find the `applicable declarations` $\alpha^*(c)$, $\delta^*(c)$, $\mu^*(c)$, which are the sets of declarations that are implicitly or explicitly declared on c with the blocked declarations removed, and the name of the class to which they apply changed. For example, consider the class *wstudent* in Figure 1. The algorithm produces:

$\alpha^*(wstudent) = \{$
$\quad wstudent[name \Rightarrow string],$
$\quad wstudent[birthyear \Rightarrow integer],$
$\quad wstudent[spouse \Rightarrow person],$
$\}$
$\delta^*(wstudent) = \{$
$\quad wstudent[birthyear \bullet\!\!\rightarrow 1970],$
$\}$

Overriding with Conflict Handling and Blocking The semantics of multiple inheritance with overriding, conflict handling and blocking are defined using the difference operators as follows:

1. If a class does not have any superclasses, then there is no inheritance, overriding, conflict resolution or block-

ing. The declarations in the class are the only ones that apply to the class.

That is, if there does not exist a class c' such that $c\ isa\ c'$, then

$\alpha^*(c) = \alpha_{bci}(c) = \alpha(c)$

$\delta^*(c) = \delta_{bci}(c) = \delta(c)$

$\mu^*(c) = \mu_{bci}(c) = \mu(c)$; otherwise

2. if $c\ isa\ c_1, ..., c_n$, then

   (a) we extend the sets of declarations to include new declarations due to explicit conflict resolution declarations

   $\alpha_{bc}(c) = \alpha(c) \cup \{c''[l \Rightarrow c_r] \mid \exists\, c[l \lhd c'] \in \chi_\alpha(c)$ and $c''[l \Rightarrow c_r] \in \alpha_{bci}(c')\}$

   $\delta_{bc}(c) = \delta(c) \cup \{c''[l \bullet\!\!\rightarrow o] \mid \exists\, c[l \lhd c'] \in \chi_\alpha(c)$ and $\exists\, c''[l \bullet\!\!\rightarrow o] \in \delta_{bci}(c')$ and $\neg\exists\, c[l \bullet\!\!\rightarrow o'] \in \delta(c)\}$

   $\mu_{bc}(c) = \mu(c) \cup \{M \mid \exists\, c[m(c_1, \ldots, c_n) \lhd c'] \in \chi_\mu(c)$ and $M \in \mu_{bci}(c')$ such that $signature(M) = m(c_1, ..., c_n)\}$

   (b) we extend the sets of declarations to include declarations that are inherited from both direct and indirect superclasses using the difference operator

   $\alpha_{bci}(c) = \alpha_{bc}(c) \cup (\alpha_{bci}(c_1) - \alpha_{bc}(c)) \cup \ldots \cup ((\ldots((\alpha_{bci}(c_n) - \alpha_{bci}(c_{n-1})) - \alpha_{bci}(c_{n-2})) - \ldots - \alpha_{bci}(c_1)) - \alpha_{bc}(c))$

   $\delta_{bci}(c)$ and $\mu_{bci}(c)$ are defined analogously.

   (c) we remove blocked declarations and change the class names in the sets of declarations

   $\alpha^*(c) = \{c\,[l \Rightarrow c'] \mid \exists\, c''\,[l \Rightarrow c'] \in \alpha_{bci}(c)$ and $c' \neq none\}$

   $\delta^*(c) = \{c\,[l \bullet\!\!\rightarrow o] \mid \exists\, c'\,[l \bullet\!\!\rightarrow o] \in \delta_{bci}(c)$ and $\neg\exists\, c''[l \Rightarrow none] \in \alpha_{bci}(c)\}$

   $\mu^*(c) = \{M' \mid \exists\, M \in \mu_{bci}(c)$, the type of M is $c'\,[m(c_1, \ldots, c_n) \Rightarrow c_r]c_r \neq none$, and M' is obtained from M by substituting c for $c'\}$

The symbols $\alpha_{bc}(c), \alpha_{bci}(c)$, etc. are used only in the definition of applicable declarations, and are not referred to anywhere else in this paper. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. Then $\alpha^* = \{\alpha^*(c) \mid c \in C\}$, $\delta^* = \{\delta^*(c) \mid c \in C\}$, and $\mu^* = \{\mu^*(c) \mid c \in C\}$. If α^*, δ^*, and μ^* are defined, then the *semantics* of the schema $K = (C, isa, \alpha, \delta, \mu, \chi)$ is given by α^*, δ^*, and μ^*.

We have dealt with non-monotonic inheritance within a database schema. We now describe the semantics of inheritance within an instance of a database, by introducing the notions of isa^*, π^*, and λ^*. We overload the *isa* notion so that if $c\ isa\ c_1, ..., c_n$, then $c\ isa\ c_i$ for $1 \leq i \leq n$. We define isa^* as the reflexive transitive closure of isa, that captures the general inheritance hierarchy. Note that $c\ isa^*\ c$.

We say that o is a *non-direct instance* of class c, (denoted by $c\ o \in \pi^*$) in database $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$, iff c is a value class, o is a value, and o is an element in the collection which c denotes, or c is an oid class, o is an oid, and there exists a c' such that $c'\ isa^*\ c$ and $c'\ o \in \pi$. The notion of π^* captures the semantics of instance inheritance; that is, if an oid is a direct instance of a subclass c, then it is a *non-direct* instance of c and the superclasses of c.

In the case, where there is a default value declaration for an attribute in a class, the instances of the class inherits the default value for the attribute. We extend the notion λ to λ^* to capture such intended semantics:

$\lambda^* = \lambda \cup \{o.l \rightarrow o' \mid c\ o \in \pi$ and $c\ [l \bullet\!\!\rightarrow o'] \in \delta^*$ and $\neg\exists o.l \rightarrow o'' \in \lambda\}$.

Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. If π^* and λ^* are defined, then the *semantics* of the instance $I = (\pi, \lambda)$ is given by π^*, λ^*.

It is possible to define a database that has no intuitive meaning. For example it is possible to define a database schema with a cycle in its class hierarchy or an attribute in a class that has two distinct default values, or a database instance where an object is an instance of more than one class, or an attribute has more than one value for an object. In [14], we discuss a number of constraints that can be used to guarantee an intended semantics of the database and queries on the database, we give properties that demonstrate that the set of expressions defined have the intended semantics, and define a well-defined database. In the following subsection, we are concerned only with well-defined databases, that is databases with an intuitive meaning.

A database instance does not have an intuitive meaning if an object is a direct instance of more than one class; or if an attribute has more than one value for an object.

3.2 Semantics of Databases and Queries

In this paper, we focus on static databases rather than dynamic databases i.e. databases where classes of oids and their attribute values remain the same. The semantics for dynamic databases can be found in [13]. The classes of oids and their attributes form our extensional database (EDB) in the traditional deductive database sense. The methods, however, are represented intensionally by method rules. They define our intensional database (IDB). In this section, we define the semantics of methods based on the well-founded semantics proposed in [16]. Our definition differs from [16] in the following ways. We are concerned with a typed language with methods rather than an untyped language with predicates. We introduce a *well-typed* concept and take typing into account when deducing new facts

from methods. The definition of satisfaction of expressions is simple in [16] and more complex in this paper because of the many kinds of expressions. Our definition reflects the fact that our model effectively has two parts, an extensional database (EDB) that models oid membership and attribute expressions, and an intensional database (IDB) that models method expressions. The EDB is a 2-valued model while the IDB is a 3-valued model. In the EDB, expressions are true if they're in the model otherwise they are false. In the IDB, method expressions are true if they are in the model, false if their complement belongs to the model, otherwise they are undefined. When a method expression is undefined, either the method isn't defined on the invoking object, or it isn't possible to assign a truth value to that expression. Every well-defined program has a total model, unlike in the well-founded semantics, where a program may have a partial model. In fact we prove that every well-defined program has a minimal model. We first define terminology that is needed later in this section.

Herbrand Base Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database. The Herbrand base $\mathcal{B}_{\mathcal{DB}}$ based on $\mathcal{DB}$ is the set of all ground simple positive method expressions formed using the method names in $\mathcal{DB}$ (without abbreviations).

The definition for compatible sets of expressions can be found in [16]. Consider the set

$E_3 = \{\neg tom[married()], tom[married()], \neg pam[married()], \neg sam[married()], pam[single()], \neg sam[single()]\}$

Because $\{\neg tom[married()], tom[married()]\} \in E_3$, and $E_3 \cap \neg E_3 \neq \emptyset$, the set is incompatible.

Ground method expressions are required to be well-typed with respect to the appropriate class declarations. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database, and $\psi = o.m(o_1, ..., o_n) \rightarrow o_r$ or $\psi = \neg o.m(o_1, ..., o_n) \rightarrow o_r$ a ground method expression. Then ψ is *well-typed* in $\mathcal{DB}$ iff the following hold:

1. there exists a class c such that $c\ o \in \pi$; and
2. there exists a method in $\mu^*(c)$ with the method type $c\ [m(c_1, ..., c_n) \Rightarrow c_r]$ such that $c_i\ o_i \in \pi^*$ for $1 \leq i \leq n$ and $c_r\ o_r \in \pi^*$.

A set of ground method expressions is *well-typed* in $\mathcal{DB}$ iff each ground method expression is well-typed in $\mathcal{DB}$. Methods can return values. However, for the same arguments, a method should return only one value. We formalize this using the notion of consistency. A set of ground method expressions are *consistent* iff there do not exist $o.m(o_1, ..., o_n) \rightarrow o_r \in S$ and $o.m(o_1, ..., o_n) \rightarrow o'_r \in S$ such that $o_r \neq o'_r$.

Interpretation Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database. A *partial interpretation* of $\mathcal{DB}$ is a tuple $\mathcal{I} = (\pi, \lambda, \mathcal{S})$ where $\mathcal{S}$ is a compatible, consistent, and well-typed set of method expressions in $\mathcal{DB}$, and each atom in $\mathcal{S}$ is an element of the Herbrand base. A *total interpretation* is a partial interpretation that contains every well-typed method atom of the Herbrand base or its negation. For an interpretation $\mathcal{I} = (\pi, \lambda, \mathcal{S})$, π and λ form an extensional database whereas $\mathcal{S}$ forms an intensional database.

Note that $\mathcal{S}$ contains both positive and negative expressions, and different interpretations of $\mathcal{DB}$ have the same extensional database but different intensional databases. A *ground substitution* θ is a mapping from variables to oids and values. It is extended to terms and expressions in the usual way.

Satisfaction Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database and $\mathcal{I} = (\pi, \lambda, \mathcal{S})$ an interpretation of $\mathcal{DB}$. The notion of satisfaction of expressions, denoted by $\models$, and its negation, denoted by $\not\models$, are defined as follows.

1. The satisfaction of ground positive and negative oid membership expressions, ground positive and negative attribute expressions, and ground arithmetic comparison expressions are defined in the usual way.
2. For a ground positive method expression ψ, $\mathcal{I} \models \psi$ iff $\psi \in \mathcal{S}$; $\mathcal{I} \not\models \psi$ iff $\neg\psi \in \mathcal{S}$.
3. For a ground negative method expression $\neg\psi$, $\mathcal{I} \models \neg\psi$ iff $\neg\psi \in \mathcal{S}$; $\mathcal{I} \not\models \neg\psi$ iff $\psi \in \mathcal{S}$.
4. For a ground composite expression $\psi = c\ o[V_1; ...; V_n]$,
   - $\mathcal{I} \models \psi$ iff $\mathcal{I} \models c\ o$, $\mathcal{I} \models o.V_i$ for $1 \leq i \leq n$;
   - $\mathcal{I} \not\models \psi$ iff $\mathcal{I} \not\models c\ o$ or $\mathcal{I} \not\models o.V_i$ for some i with $1 \leq i \leq n$.

   For a ground composite expression $\psi = o[V_1; ...; V_n]$,
   - $\mathcal{I} \models \psi$ iff $\mathcal{I} \models o.V_i$ for every $1 \leq i \leq n$;
   - $\mathcal{I} \not\models \psi$ iff $\mathcal{I} \not\models o.V_i$ for some i with $1 \leq i \leq n$.
5. For a method rule $r = c[A :- L_1, ..., L_n]$, $I \models r$ iff for each ground substitution θ,
   - $\mathcal{I} \models \theta A$; or
   - $\mathcal{I} \not\models \theta A$ and for each ground method rule with head θA there exists an L_i with $1 \leq i \leq n$ such that $\mathcal{I} \models \neg\theta L_i$; or
   - there exists an L_i with $1 \leq i \leq n$ such that neither $\mathcal{I} \models \theta L_i$, nor $\mathcal{I} \not\models \theta L_i$.

In other words, $\mathcal{I} \models \psi$ means that ψ is true in $\mathcal{I}$; $\mathcal{I} \not\models \psi$ means that ψ is false in $\mathcal{I}$; if neither $\mathcal{I} \models \psi$, nor $\mathcal{I} \not\models \psi$, then ψ is unknown in $\mathcal{I}$.

Model Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database and $\mathcal{I} = (\pi, \lambda, S)$ an interpretation of $\mathcal{DB}$. Then

$\mathcal{I}$ is a *model* of $\mathcal{DB}$ if $\mathcal{I}$ satisfies every ground method rule in μ^*.

Consider the following database:

$class\ person\ [$
$\quad spouse \Rightarrow person;$
$\quad married()\ \{$
$\quad\quad married() :- spouse \rightarrow X;$
$\quad\quad married() :- X.married(),$
$\quad\quad\quad X.spouse \rightarrow This\}$
$\quad single()\{single() :- \neg married()\}$
$\quad]$
$person\ sam[spouse \rightarrow pam]$
$person\ pam$

The following set is a model of this database:

$I = (\{person\ sam,\ person\ pam\},$
$\{sam.spouse \rightarrow pam\},$
$\{sam.married(),\ pam.married(),$
$\neg sam.single(),\ \neg pam.single()\}).$

Due to the typing and compatibility constraints as in ROL [12], it is possible that a database has no models. Also, a well-defined database may have several models. Our intention is to select a proper minimal model as the intended semantics of the database.

An unfounded set for a database with respect to an interpretation provides a basis for false method expressions in our semantics. The greatest unfounded set (GUS) is the set of all the expressions that are false in a database with respect to an interpretation and is used to provide the negative expressions when finding the model of a database. The definition for unfounded sets and greatest unfounded sets can be found in [16]. The greatest unfounded set is used in the definition of a model, i.e. a limit of the following transformation.

Transformation Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database. The *transformation* $T_{\mathcal{DB}}$ of $\mathcal{DB}$ is a mapping from interpretation to interpretation defined as follows.

$$T_{\mathcal{DB}}(\mathcal{I}) = \begin{cases} (\pi, \lambda, \mathcal{W}(\mathcal{I})) & \text{if } \mathcal{W}(\mathcal{I}) \text{ is well-typed and compatible} \\ undefined & \text{otherwise} \end{cases}$$

where

$\mathcal{T}(\mathcal{I}) = \{\theta A \mid A :- L_1, ..., L_n$ is a method rule in $\mathcal{DB}$
and there exists a ground substitution θ such that
$\mathcal{I} \models \theta L_1, ..., \mathcal{I} \models \theta L_n\}$
$\mathcal{U}(\mathcal{I}) = \neg G$ where G is the GUS of $\mathcal{DB}$ with respect to $\mathcal{I}$.
$\mathcal{W}(\mathcal{I}) = \mathcal{T}(\mathcal{I}) \cup \mathcal{U}(\mathcal{I})$

Model For all countable ordinals h the tuple $\mathcal{I}_h$ for database $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$, the limit of the transformation T_{DB} is defined recursively by:

1. For limit ordinal h, $\mathcal{I}_h = (\pi,\ \lambda,\ \cup_{j<h}\mathcal{W}(\mathcal{I}_j))$
2. For successor ordinal $k+1$, $\mathcal{I}_{k+1} = T_{\mathcal{DB}}(\mathcal{I}_k)$.

Note that 0 is a limit ordinal, and $\mathcal{I}_0 = (\pi,\ \lambda,\ \emptyset)$. This sequence reaches a limit $\mathcal{I}^*$.

We now prove that $\mathcal{I}^*$ is a model.

Theorem 3.1 Let $\mathcal{DB}$ be a well-defined database. If $\mathcal{I}^* = (\pi,\ \lambda,\ S)$ is defined, then it is a model of $\mathcal{DB}$. □

Minimal model Let $M = (\pi,\ \lambda,\ S)$ be a model of a database $\mathcal{DB}$. We say that model M is *minimal* if there does not exist an expression ψ in S such that $(\pi,\ \lambda,\ S - \psi)$ is still a model.

We now prove that for a well-defined database $\mathcal{DB}$, $\mathcal{I}^*$ is a minimal model of $\mathcal{DB}$ if it is defined.

Theorem 3.2 Let $\mathcal{DB}$ be a well-defined database. If $\mathcal{I}^* = (\pi,\ \lambda,\ S)$ is defined, then it is a minimal model of $\mathcal{DB}$. □

Semantics of Databases The *semantics* of a well-defined database $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ is represented by the limit $\mathcal{I}^*$ if it is defined.

Semantics of Queries Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database, Q a query of the form $?\text{–}L_1, ..., L_n$, and θ a ground substitution for variables of Q. Assume $\mathcal{I}^*$ is defined. Then the *answer* to Q based on $\mathcal{DB}$ is one of the following:

1. *true* if $\mathcal{I}^* \models \theta L_1, ..., \mathcal{I}^* \models \theta L_n$,
2. *false* if there exists an L_i with $1 \leq i \leq n$ such that $\mathcal{I}^* \not\models \theta L_i$, and
3. *unknown* otherwise.

In other words, for a method expression ψ, if $\mathcal{I}^* \models \psi$ then expression ψ is true, if $\mathcal{I}^* \models \neg\psi$ then expression ψ is false, and if $\mathcal{I}^* \not\models \psi$ and $\mathcal{I}^* \not\models \neg\psi$ then expression ψ is undefined. Let us consider an example with unknown answers. Consider the following database:

$class\ person\ [$
$\quad spouse \Rightarrow person;$
$\quad married()\ \{married() :- \neg single()\}$
$\quad single()\{single() :- \neg married()\}$
$\quad]$
$person\ sam[spouse \rightarrow pam]$
$person\ pam$

Then $\mathcal{I}^* = (\{person\ sam, person\ pam\},\ \{sam[spouse \rightarrow pam]\},\ \emptyset)$ is a three-valued model, in which the answers to the following queries are unknown.

$?\text{–}sam[married()]$
$?\text{–}sam[single()]$

There are two reasons why $\mathcal{I}^*$ may be undefined. One is that the inferred set of method expressions is not well-typed. The other is that it is not consistent. For the first problem, we could define another constraint on method rules using type substitution as in [13] to constrain the database. For the second problem, run-time checking is necessary.

4 Conclusion

Logical semantics have played an important role in database research. However, the object-oriented approach to databases was dominated by "grass-roots" activity where several systems were built without the accompanying theoretical progress. As a result, many researchers feel the area of object-oriented databases is misguided [9]. The deductive object-oriented database research, however, has taken quite a different approach. It has logical semantics as its main objective and started with a small set of simple features taken from the object-oriented paradigm such as F-logic [9], and gradually incorporates more and more difficult features that can be given a logical semantics such as ROL [12] and Datalog++ [8].

The main contribution of the paper is the addition of two outstanding object-oriented features to deductive object-oriented databases together with a direct logical semantics. The two outstanding features were rule-based methods and the encapsulation of these methods in classes, and multiple behavioral inheritance, with overriding, blocking, and conflict handling. We have shown that these object-oriented features which are believed to be difficult to address, can indeed be captured logically. We believe that the semantics given in this paper have a far reaching influence on the design of deductive object-oriented languages and even object-oriented languages in general. The language and semantics defined on the language form the theoretical basis for a practical query language. Indeed, the practical deductive object-oriented database language ROL2 [15] supports the theory discussed here.

Our work differs from the work of others in many ways. Most existing deductive object-oriented database languages do not allow rule-based methods to be encapsulated in the class definitions. Those that do, do not address the issue directly. Also, most existing deductive object-oriented database languages do not allow non-monotonic multiple behavioral inheritance. ROL does, but deals with conflict handling only in a limited context and doesn't have blocking. Datalog^{++} provides blocking and disallows the inheritance of conflicting properties. F-logic supports monotonic structural inheritance and indeterminate non-monotonic default value inheritance by allowing a database to have multiple possible models. For a class, not only its subclasses but also its elements can inherit its properties.

References

[1] S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[2] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and Rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 32–41, Washington, D.C., 1993.

[3] M. Bugliesi and H. M. Jamil. A Logic for Encapsulation in Object Oriented Languages. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, pages 213–229, Madrid, Spain, 1994. Springer-Verlag, LNCS 844.

[4] F. Cacace, S. Ceri, S. Crepi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 225–236, 1990.

[5] O. Deux and others. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[6] S. Greco, N. Leone, and P. Rullo.. COMPLEX: An Object-Oriented Logic Programming System. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344–359, 1992.

[7] A. Heuer and P. Sander. The LIVING IN A LATTICE Rule Language. *Data and Knowledge Engineering*, 9(4):249–286, 1993.

[8] H. M. Jamil. Implementing Abstract Objects with Inheritance in Datalog^{neg}. In *Proceedings of the International Conference on Very Large Data Bases*, pages 46–65, Athens, Greece, 1997.

[9] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM*, 42(4):741–843, 1995.

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

[11] T. W. Ling The Prolog not-predicate and negation as failure rule. In *New Generation Computing*, 8:1, (1990), pages 5–31.

[12] M. Liu. ROL: A Deductive Object Base Language. *Information Systems*, 21(5):431 – 457, 1996.

[13] M. Liu. Incorporating Methods and Encapsulation into Deductive Object-Oriented Database Languages. In *Proceedings of the 9th International Conference on Database and Expert System Applications (DEXA '98)*, pages 892–902.

[14] M. Liu and G. Dobbie and Tok Wang Ling. A Logic for Deductive Object-Oriented Databases. To be submitted.

[15] M. Liu and M. Guo. ROL2: A Real Deductive Object-Oriented Database Language. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER '98)*, pages 302–315, Singapore, Nov. 16-19 1998. Springer-Verlag LNCS 1507.

[16] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of ACM*, 38(3):620–650, 1991.