A Logical Foundation for Deductive Object-Oriented Databases

MENGCHI LIU
Carleton University and Wuhan University
GILLIAN DOBBIE
University of Auckland
and
TOK WANG LING
National University of Singapore

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. This article takes us another step towards solving this problem. Two features that are currently missing are the encapsulation of rule-based methods in classes, and nonmonotonic structural and behavioral inheritance with overriding, conflict resolution and blocking. This article introduces the syntax of a language with these features. The language is restricted in the sense that we have omitted other object-oriented and deductive features that are now well understood, in order to make our contribution clearer. 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 article establishes a firm logical foundation for deductive object-oriented databases.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classification—*object-oriented languages*; H.2.1 [**Database Management**]: Logical Design—*data models; schema and subschema*; H.2.3 [**Database Management**]: Languages—*data description languages (DDL); database (persistent) programming languages; query languages*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*deduction (e.g., natural, rule-based); logic programming; nonmonotonic reasoning and belief revision*;

M. Liu's work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grants RPGIN 193552-00 and EQPEQ 229966-00.

Authors' addresses: M. Liu, School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Ont. K1S 5B6, Canada; email: mengchi@scs.carleton.ca; G. Dobbie, Department of Computer Science, University of Auckland, Private Bag 92019, Auckland 1020, New Zealand; email: gill@cs.auckland.ac.nz; T. W. Ling, Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543; email: lingtw@comp.nus.edu.sg.

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*representation languages*

General Terms: Languages, Theory

Additional Key Words and Phrases: Declarative semantics, deductive databases, nonmonotonic multiple inheritance, object-oriented databases, rule-based languages

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. In the past decade, a large number of deductive object-oriented database languages have been proposed, such as O-logic [Maier 1986], revised O-logic [Kifer and Wu 1993], C-logic [Chen and Warren 1989], F-logic [Kifer et al. 1995], IQL [Abiteboul and Kanellakis 1998], LOGRES [Cacace et al. 1990], LLO [Lou and Ozsoyoglu 1991], COMPLEX [Greco et al. 1992], ORLOG [Jamil and Lakshmanan 1992], LIVING IN LATTICE [Heuer and Sander 1993], Datalog^{meth} [Abiteboul et al. 1993], CORAL++ [Srivastava et al. 1993], Noodle [Mumick and Ross 1993], DTL [Bal and Balsters 1993], Gulog [Dobbie and Topor 1995], Rock & Roll [Barja et al. 1995], ROL [Liu 1996], Datalog^{++} [Jamil 1997], ROL2 [Liu and Guo 1998; Liu 1999], Chimera [Guerrini et al. 1998], and DO2 [Ling and Lee 1998]. These proposals can be roughly classified into two kinds: loosely-coupled and tightly coupled. The first kind mainly uses or extends Datalog-like language as a query language for object-oriented databases. This is not a satisfactory approach as the resulting language/system consists of two clearly distinct parts with no unifying semantics. Typical examples of this kind are: IQL, Rock & Roll, CORAL++, and Chimera. The other approach is more fundamental in which new unifying logics are proposed to formalize the notions underlying object-oriented databases. Typical examples of this kind are: revised O-logic, C-logic, F-logic, ORLOG, and ROL.

Based on these proposals as well as the work in object-oriented programming languages and data models, such as GemStone [Butterworth et al. 1991], ONTOS [Soloviev 1992], O_2 [Deux et al. 1991], Orion [Kim 1990], Iris [Fishman et al. 1987], ObjectStore [Lamb et al. 1991], ODMG-93 [Cattell 1996], ODMG 2.0 [Cattell and Barry 1997], 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 structural and behavioral inheritance with overriding, blocking, and conflict handling. However, a clean logical semantics that naturally accounts for all these features is still missing from the literature. In particular, the following two important issues have not been addressed properly so far. One is rule-based methods and the encapsulation of these methods in classes. The other is nonmonotonic multiple structural and behavioral inheritance.

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 databases, we use rules instead of functions and procedures. By analogy, methods in deductive object-oriented databases can 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 [Kifer et al. 1995], IQL [Abiteboul and Kanellakis 1998], Datalog^{meth} [Abiteboul et al. 1993], ROL [Liu 1996], Datalog^{++} [Jamil 1997], 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 that do not address the issue directly. Bugliesi and Jamil [1994] address encapsulation but do not include other very important object-oriented features, like inheritance, in their language.

Nonmonotonic multiple structural and behavioral inheritance is a fundamental feature of object-oriented data models such as O_2 [Deux et al. 1990] and Orion [Kim 1990]. 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). Most systems use the superclass ordering to solve the conflicts. 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 revised O-logic [Kifer and Wu 1993], F-logic,[1] LOGRES [Cacace et al. 1990], LIVING IN LATTICE [Heuer and Sander 1993], COMPLEX [Greco et al. 1992], and Chimera [Guerrini et al. 1998] only allow monotonic multiple structural inheritance, which is not powerful enough. Some deductive object-oriented languages such as Datalog^{meth} only support nonmonotonic 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 nondirect instances. ROL [Liu 1996] has a semantics that accounts for nonmonotonic multiple structural inheritance with overriding and conflict-handling in a limited context, but without blocking. Datalog^{++} [Jamil 1997] takes a quite different approach towards nonmonotonic 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 nonmonotonic inheritance.

[1]F-logic however supports indeterminate nonmonotonic default value inheritance. The value inherited depends on which inheritance step is done first at run time.

This article provides a direct well-defined declarative semantics for a deductive object-oriented database language with encapsulated rule-based methods and nonmonotonic structural and 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 defined, for example, set-valued attribute values, and we focus on a static database rather than a dynamic database, that is, we do not consider 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, that is, it won't be inherited from its superclasses. We provide a very flexible approach to conflict resolution. Our mechanism consists of two parts. The first, and 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 second 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 structural and behavioral inheritance, overriding, conflict handling and blocking, and is based on the well-founded semantics [Gelder et al. 1991] from logic programming. However, our semantics differ from well-founded semantics in a number of ways. We introduce typing and the concept of a well-typed database, our definition of "satisfaction" is more complex, and our model has two parts, a two-valued part representing the extensional database and a three-valued part representing the intensional database. 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.

This article is organized as follows: We introduce the language using an example in Section 2. Section 3 introduces the syntax of the language. In Section 4, the class of well-defined databases and the semantics of well-defined databases are defined, and the main results are presented. Section 5 concludes the article, reiterating our results and comparing this work with related work.

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 a style similar to F-logic [Kifer et al. 1995]. In particular, it supports object identity, complex objects, typing, rule-based methods, encapsulation of methods, overloading, late binding, polymorphism, class hierarchy, and multiple structural and behavioral inheritance with overriding, blocking, and conflict handling. In this section, we introduce and demonstrate concepts that are important in the article.

Figure 1 shows a sample database. Notice that $\Rightarrow$ is used when types of attributes or methods are declared, $\rightarrow$ is used when values are assigned to

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

```
class person [
        $name \Rightarrow string;$
        $birthyear \Rightarrow integer;$
        $birthyear \bullet\!\!\rightarrow 2002;$
        $homephone \Rightarrow integer;$
        $spouse \Rightarrow person;$
        $emergencyContact \Rightarrow integer;$
        $emergencyContact \bullet\!\!\rightarrow 911;$
        $age() \Rightarrow integer$ {
              $age() \rightarrow A$ :- $birthyear \rightarrow B, A = 2002 - B$}
        $contactNumber() \Rightarrow integer$ {
              $contactNumber() \rightarrow X$ :- $homephone \rightarrow X$}
        $married\_to(person)$ {$married\_to(X)$ :- $spouse \rightarrow X$}
        $single()$ {$single()$ :- $\neg married\_to(X)$}
        ]
class employee isa person [
        $birthyear \bullet\!\!\rightarrow 1960;$
        $workphone \Rightarrow integer;$
        $salary \Rightarrow integer;$
        $salary \bullet\!\!\rightarrow 2000;$
        $homephone \Rightarrow none;$
        $contactNumber() \Rightarrow integer$ {
              $contactNumber() \rightarrow X$ :- $workphone \rightarrow X$}
        ]
class student isa person [
        $birthyear \bullet\!\!\rightarrow 1970;$
        $emergencyContact \Rightarrow employee;$
        $major \Rightarrow string;$
        $major \bullet\!\!\rightarrow$ "CS";
        $extrasupport() \Rightarrow integer$ {
              $extrasupport() \rightarrow 1000$ :- $married\_to(X), student\ X;$
              $extrasupport() \rightarrow 500$  :- $married\_to(X), \neg student\ X;$
              $extrasupport() \rightarrow 100$  :- $single()$}
        $support() \Rightarrow integer$ {
              $support() \rightarrow S$ :- $extrasupport() \rightarrow S_1, S = 1000 + S_1$}
        ]
class wstudent isa employee, student [
        $birthyear \lhd student;$
        $support() \Rightarrow none;$
        $extrasupport() \Rightarrow person$ {
              $extrasupport() \rightarrow X$ :- $spouse \rightarrow X$}
        ]
```

(*a*) *Schema*

```
employee tom   [$name \rightarrow$ "Tom"; $birthyear \rightarrow 1963$; $spouse \rightarrow pam$, $salary \rightarrow 5000$]
student  sam   [$name \rightarrow$ "Sam"; $major \rightarrow$ "CS"; $homephone \rightarrow 8751834$]
wstudent pam   [$name \rightarrow$ "Pam"; $spouse \rightarrow tom$; $major \rightarrow$ "CS"]
```

(*b*) *Instance*

Fig. 1. Sample database.

attributes or methods, $\bullet\!\!\rightarrow$ is used to define default values, and $\lhd$ defines which class a method or attribute is to be inherited from.

The schema in Figure 1(a) defines four classes, *person*, *employee*, *student*, and *wstudent* (working student).

The class *person* has five attributes, *name*, *birthyear*, *homephone*, *spouse*, and *emergencyContact*, and four methods with signatures: *age*(), *contactNumber*(), *married_to*(*person*), and *single*(). The attribute declaration for *name* gives the attribute label, *name*, and the class of the value of the attribute, *string*. The attribute *birthyear* has a default value of 2002, while the attribute *emergencyContact* has a default value of 911. The method declaration for *married_to*(*person*) has two parts: the method type and the method rule. The method type *married_to*(*person*) states the method label, *married_to*, the class of its argument, *person*, and the class of the result it returns, which is nothing in this case. The method rule is the statement in parentheses and is made up of a head and a body. The part before the :– is the head and the other part is the body. The method *age*() returns a person's age, method *contactNumber*() returns a person's *homephone* number, method *married_to*(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.

The class *employee* inherits from *person*. The word *isa* indicates that a class inherits from another class. We say that class *employee* is a direct subclass of *person* and *person* is a direct superclass of *employee*. That is, the class *employee* inherits all attribute declarations, default values and method declarations from class *person* unless they are blocked or overridden in class *employee*. New attributes and default values for attributes can also be declared in subclasses, for example, the attributes *workphone* and *salary* with default value of 2000 are declared in *employee*. The attribute declarations for *name*, *birthyear*, *spouse* and *emergencyContact*, and the method declarations for *age*(), *married_to*(*person*), and *single*() are inherited but the attribute *homephone*, the default value of *birthyear* and method *contactNumber*() are overridden in *employee*. When a default value, an attribute or a method is redefined in a class, we say that the new declaration in the subclass overrides the definition in the superclass, for example, the default value for attribute *birthyear* is redefined to 1960, and an employee's contact number is their *workphone* number. The attribute *homephone* is blocked in *employee*, that is, the attribute is redefined with a return class of *none*. When an attribute or method is blocked in a class, it is not inherited and is undefined on instances of that class. It is also undefined on subclasses of that class, unless it is redefined in the subclass.

The class *student* inherits from *person*. Because attribute *emergencyContact* is redefined with a return class of *employee*, the default value is not inherited from *person*. Two methods are declared in *student*, namely *extrasupport*() and *support*(). The value of the method *support*() for a *student* is the value of their *extrasupport*() plus 1000. The value of their *extrasupport*() is 1000 if they are married to a *student*, 500 if they are married and their spouse is not a *student*, and 100 if they are single.

The class *wstudent* inherits from two classes, *employee* and *student*. With multiple inheritance, there can be conflicting declarations, that is, default values, attributes and methods may be declared in more than one superclass. There are three possible conflicts to be resolved in *wstudent*, attribute *emergencyContact*, method *contactNumber*() and default value *birthyear* are defined on both *employee* and *student*. There are two ways that conflicts can be resolved. A conflict resolution declaration indicates explicitly from which class a property is to be inherited, for example, $birthyear \lhd student$ 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, for example, attribute *emergencyContact*, and method *contactNumber*() are inherited from *employee*. Notice that the method *support*() is blocked in *wstudent*, 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 but different return values. A method has the same signature as another method if the method has the same method label and the same arguments, for example, *extrasupport*() in *student* has the same signature as *extrasupport*() in *wstudent*. The method rule in *wstudent* states that the *extrasupport*() of an instance of *wstudent* is the *spouse* of *wstudent*. This overrides the *extrasupport*() method defined in class *student*. 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, that is, the default 1960 in *employee* is not inherited. 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 indirect instance of *person*. The *birthyear* of *sam* is 1970, that is, 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, that is, a method is declared in a class and invoked through instances of the class.

(1) Find the *birthyear* and *age* of Tom.

$$?\!-\ employee\ O\ [name \rightarrow \text{“}Tom\text{”};\ birthyear \rightarrow Y;\ age() \rightarrow Z]$$

In this query, we are asking for the value of one of the attributes and one of the methods for the object with name "Tom." The *age*() method is inherited from the *person* class. The answer is $\{O = tom,\ Y = 1963,\ Z = 39\}$.

(2) Find the birthyear and contact number for Sam.

$$?\!-\ student\ O\ [name \rightarrow \text{“}Sam\text{”};\ birthyear \rightarrow X;\ contactNumber() \rightarrow Y]$$

The default value of *birthyear* for instances in class *student* is returned, $X = 1970$. The method *contactNumber*() is inherited from *person*, $Y = 8751834$.

(3) Find what support Sam gets.

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

The *support*() method in class *student* invokes the *extrasupport*() method. The *extrasupport*() rules in turn invoke the *married_to*(*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.

(4) Find what support Pam gets.

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

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

(5) Find what extra support Pam has.

$$?\!-\ wstudent\ O\ [name = \text{“}Pam\text{”};\ extrasupport() \rightarrow X]$$

The method *extrasupport*() in *student*() is overridden by *extrasupport*() in *wstudent*. The value returned is the oid of the *spouse* of Pam, namely *tom*.

(6) Find all the people in the database.

$$?\!-\ person\ O$$

The answer to this query includes the oids of all the objects in the class *person* and all the subclasses of *person*. The answers are $O = tom$; $O = sam$; $O = pam$.

(7) Find all students whose extrasupport is not 500.

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

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 $O = sam$.

3. SYNTAX

In this section, we define the formal syntax. We use examples from Section 2 to clarify our definitions. We distinguish between values that belong to built-in values classes and objects that belong to user-defined object identifier (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. We also differentiate between attributes and methods. Attributes have a fixed value while the value of a method is derived at runtime. 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 article using uppercase alphabetic characters.

We assume the existence of the following pairwise disjoint sets:

(1) a set of value class names $\mathcal{B} = \{integer, string, void, none\}$;
(2) a set $\mathcal{C}$ of oid class names;
(3) a set $\mathcal{A}$ of attribute labels;
(4) a set $\mathcal{M}$ of method names with arity $n \geq 0$;
(5) a set $\mathcal{D}$ of values which is the union of the set $\mathcal{I}$ of integers, the set $\mathcal{S}$ of strings, and the set $\{nil\}$;
(6) a set $\mathcal{O}$ of object identifiers (oids);
(7) a set $\mathcal{V}$ of variables including *This*.

Example 3.1. In the example in Section 2, *person*, *employee*, *student* and *wstudent* are oid class names, *name*, *birthyear*, *homephone*, *spouse*, and *emergencyContact* are examples of attribute labels, *contactNumber*, *married_to* and *single* are examples of method names, *tom*, *sam* and *pam* are object identifiers, and S, S_1 and X are some of the variables.

Oid classes denote collections of objects that share common structural and behavioral properties. The structural properties are represented in terms of attributes and default attribute values whereas the behavioral properties are represented in terms of methods. Two kinds of classes are distinguished: value classes and oid classes. The collections of instances that value classes denote are fixed with built-in semantics (see the definition of π^* in Section 4.2). The collections that oid classes denote depend on the user-defined instances.

Definition 3.1. The *classes* are defined as follows:

(1) elements of $\mathcal{B}$ are *value* classes;
(2) elements of $\mathcal{C}$ are *oid* classes.

In order to define what a database is, we introduce the following auxiliary notions.

Definition 3.2. A *term* is either a value, an oid, or a variable. An *oid term* is either an oid or a variable. A term is *ground* if it has no variables.

For example, "*Tom*", *tom*, and X are terms, where *tom* is a ground oid term, and "*Tom*" is a ground value term.

In the following definition, we distinguish between simple and composite expressions where composite expressions are composed of simple expressions.

Definition 3.3. The *expressions* are defined based on terms as follows:

(1) Let c be a class and O an oid term. Then $c\ O$ is a *simple* expression, called a positive *oid membership* expression.
(2) Let O be an oid term, l an attribute label, and O' a term. Then $O.l \rightarrow O'$ is a *simple* expression, called a positive *attribute* expression. When O is *This*, we can simply use $l \rightarrow O'$.

(3) Let O be an oid term, m an n-ary method name with $n \geq 0$, and $O_1, \ldots, O_n$, O_r terms. Then $O.m(O_1, \ldots, O_n) \rightarrow O_r$ is a *simple* expression, called a positive *method* expression. When O is *This*, we can simply use $m(O_1, \ldots, O_n) \rightarrow O_r$; when O_r is *nil*, we can simply use $O.m(O_1, \ldots, O_n)$; when O is *This* and O_r is *nil*, we can simply use $m(O_1, \ldots, O_n)$.

(4) Let E be a positive oid membership, attribute or method expression. Then $\neg E$ is a *simple* expression, called *negative* oid assignment, attribute or method expression, respectively.

(5) Let $c\ O$ be a positive oid membership expression, $O.V_1, \ldots, O.V_n$, $\neg O.U_1, \ldots, \neg O.U_m$ attribute or method expressions with $n \geq 0, m \geq 0$. Then $c\ O\ [V_1; \ldots; V_n; \neg U_1; \ldots; \neg U_m]$ and $O\ [V_1; \ldots; V_n; \neg U_1; \ldots; \neg U_m]$ are *composite* expressions. When $n = 0$, $O\ [\neg U_1; \ldots; \neg U_m]$ is called a composite negative expression; when $m = 0$, it is called a composite positive expression.

(6) Arithmetic comparison expressions are defined using terms in the usual way.

Example 3.2. The following are examples of various expressions:

Positive oid membership expressions:	*person tom*, *person X*
Negative oid membership expressions:	$\neg$*person tom*, $\neg$*person X*
Positive attribute expressions:	$P.name \rightarrow$ "Tom," $spouse \rightarrow P$
Negative attribute expressions:	$\neg P.name \rightarrow$ "Tom," $\neg spouse \rightarrow P$
Positive method expressions:	$P.married\_to(tom)$, $support() \rightarrow S$
Negative method expressions:	$\neg P.married\_to(X)$, $\neg support() \rightarrow S$
Composite expressions:	$person\ P\ [name \rightarrow X; \neg single()]$
Arithmetic comparison expressions:	$A = 2002 - B,\ S = A * 500.$

Note that our language supports the omission of the special variable *This* to simplify programming as in C++ and Java.

An expression without abbreviation is *ground* if every term in it is ground. For example, *person tom* is a ground expression, but *person X* is not a ground expression.

3.1 Schema Syntax

In a database schema, we define the class hierarchy using superclass declarations, attributes of classes using attribute declarations, default attribute values using default value declarations, methods of classes using method declarations and conflict resolution declarations are used as part of the conflict resolution mechanism.

Definition 3.4. If c and $c_1, \ldots, c_l$ are oid classes with $l \geq 0$, then c *isa* $c_1, \ldots, c_l$ is a *superclass declaration* that declares c is a *direct subclass* of $c_1, \ldots,$ and c_l and $c_1, \ldots,$ and c_l are *direct superclasses* of c.

The following is an example of superclass declaration:

wstudent isa employee, student,

where *wstudent* is a direct subclass of *employee* and *student*, and *employee* and *student* are direct superclasses of *wstudent*.

Definition 3.5. Let c be an oid class, c' be a class other than *void*, and l be an attribute label. Then $c\ [l \Rightarrow c']$ is an *attribute declaration* for the class c. The special case where c' is *none*, that is, $c\ [l \Rightarrow none]$, is an *attribute blocking declaration* for the class c.

For example, $person\ [name \Rightarrow string]$ is an attribute declaration for the class *person*, and $employee\ [homephone \Rightarrow none]$ is an attribute blocking declaration for the class *employee*.

It is more and more common to see software systems provide various defaults to ease the use of the systems. A default attribute value holds on an instance of a class unless a value is specifically assigned to that attribute for that instance. Oracle supports default attribute values, F-logic [Kifer et al. 1995] and Datalog^{++} [Jamil 1997] also support default values (by a different name though). In our language, we use a syntax similar to F-logic for default values.

Definition 3.6. Let c be an oid class, l an attribute label and o a ground term. Then $c\ [l \bullet\!\!\rightarrow o]$ is a *default value declaration* for the attribute l of class c.

For example, $person\ [birthyear \bullet\!\!\rightarrow 2002]$ is a default value declaration for the attribute *birthyear* of class *person*. If a person was born in 2002, then there is no need to explicitly specify his/her birthyear in the database.

Behavioral properties are given by class methods. A method consists of two parts, the method type and method rules.

Definition 3.7. Let c be an oid class, $c_1, \ldots, c_n$ classes other than *void* and *none*, c_r a class, and m an n-ary method name with $n \geq 0$. Then $c\ [m(c_1, \ldots, c_n) \Rightarrow c_r]$ is a *method type* for the method m of the class c. The *signature* of the method is $m(c_1, \ldots, c_n)$. The method type specifies the class c that the method is attached to, the name m of the method, the classes $c_1, \ldots, c_n$ of the arguments, and the class c_r of the result it returns. When c_r is *void*, we can simply write $c\ [m(c_1, \ldots, c_n)]$ instead.

Method rules are used to derive information that is not explicitly stored in the database.

Definition 3.8. A *method rule* of the class c is of the form $c\ [A :\!- L_1, \ldots, L_n]$ where the head A is a simple positive method expression, the body $L_1, \ldots, L_n$ with $n \geq 0$ is a sequence of expressions. The method name and its arity in A are also the *name* and *arity* of the rule.

A method declaration consists of a method type and a set of method rules.

Definition 3.9. Let $c\ [m(c_1, \ldots, c_n) \Rightarrow c_r]$ be a method type, $r_1, \ldots, r_k$ method rules that have name m and arity n with $n \geq 0$ and $k \geq 0$. Then $c\ [m(c_1, \ldots, c_n) \Rightarrow c_r\ \{r_1; \ldots; r_k\}]$ is a *method declaration* for the class c. Let M be a method declaration, we use $sig(M)$ to denote its signature. When $k = 0$, we can simply write $c\ [m(c_1, \ldots, c_n) \Rightarrow c_r]$. The special case where c_r is *none*, that is, $c\ [m(c_1, \ldots, c_n) \Rightarrow \mathit{none}]$, is a *method blocking declaration* for the class c.

Consider the following example of method declaration:

$$
\begin{array}{l}
\mathit{wstudent}\ [\\
\quad \mathit{extrasupport}() \Rightarrow \mathit{person}\ \{\\
\qquad \mathit{extrasupport}() \rightarrow X \mathrel{:-} \mathit{spouse} \rightarrow X\}\\
\quad].
\end{array}
$$

The method type is $\mathit{wstudent}\ [\mathit{extrasupport}() \Rightarrow \mathit{person}]$, the signature of the method is $\mathit{extrasupport}()$, and $\mathit{wstudent}\ [\mathit{extrasupport}() \rightarrow X \mathrel{:-} \mathit{spouse} \rightarrow X]$ is the method rule. The declaration $\mathit{wstudent}\ [\mathit{support}() \Rightarrow \mathit{none}]$ is a method blocking declaration for the class *wstudent*.

As we have shown, the inheritance hierarchy is defined using superclass declarations. Note that the order of $c_1, \ldots, c_l$ in a superclass declaration in our language is significant because the conflict resolution mechanism uses this ordering to determine the order of structural and behavioral inheritance if there are no explicit conflict resolution declarations. See the example in Section 2. Conflict resolution declarations can be used to name explicitly the class from which a particular attribute or method is to be inherited.

Definition 3.10. Let c, c' be oid classes, l an attribute and $m(c_1, \ldots, c_n)$ a method signature. Then $c\ [l \lhd c']$ is an *attribute conflict resolution declaration* for the class c; and $c\ [m(c_1, \ldots, c_n) \lhd c']$ is a *method conflict resolution declaration* for the class c.

For example, $\mathit{wstudent}\ [\mathit{birthyear} \lhd \mathit{student}]$ is an attribute conflict resolution declaration for the class *wstudent*.

Based on the above definitions, a class has five kinds of information: direct superclasses, attribute declarations, default value declarations, method declarations, and conflict resolution declarations.

Definition 3.11. Let $c\ \mathit{isa}\ c_1, \ldots, c_m$ be a superclass declaration, $c\ [C_1], \ldots, c\ [C_n]$ conflict resolution declarations, $c\ [A_1], \ldots, c\ [A_o]$ attribute declarations, $c\ [D_1], \ldots, c\ [D_p]$ default value declarations, and $c\ [M_1], \ldots, c\ [M_q]$ method declarations with $m \geq 0$, $n \geq 0$, $o \geq 0$, $p \geq 0$, and $q \geq 0$. Then

$$\mathit{class}\ c\ \mathit{isa}\ c_1, \ldots, c_m\ [C_1; \ldots; C_n; A_1; \ldots; A_o; D_1; \ldots; D_p; M_1\ \ldots\ M_q]$$

is a *class declaration*. When $m = 0$, we shall write as follows instead

$$\mathit{class}\ c\ [C_1; \ldots; C_n; A_1; \ldots, ; A_o; D_1, \ldots, D_p; M_1\ \ldots\ M_q]$$

Note that rules are encapsulated in classes rather than separated from classes as in other languages, like F-logic [Kifer et al. 1995] and ROL [Liu 1996].

Definition 3.12. A *schema* K is a set of class declarations, which can be represented abstractly as a tuple $K = (C, \mathit{isa}, \alpha, \delta, \mu, \chi)$ 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 note $\alpha(c)$, $\delta(c)$, $\mu(c)$, $\chi_\alpha(c)$, and $\chi_\mu(c)$ are 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.

In Section 4.2, we impose constraints on the schema to capture the intended semantics of multiple inheritance with overriding, blocking and conflict handling.

3.2 Instance Syntax

In a database instance, we define the objects and their attribute values using object declarations.

Definition 3.13. An *object declaration* is an expression of the form:

$$c\ o\ [l_1 \rightarrow o_1; \ldots; l_n \rightarrow o_n]$$

that has the ground oid membership expression $c\ o$ and ground positive attribute expressions $o.l_1 \rightarrow o_1, \ldots, o.l_n \rightarrow o_n$. It specifies that the oid class c is a *primary class* of the oid o, o is a *direct* instance of c, and o has values $o_1, \ldots, o_n$ for its attributes $l_1, \ldots, l_n$, respectively.

For example, *wstudent pam* [*name* $\rightarrow$ "*Pam*"; *spouse* $\rightarrow$ *tom*] is an object declaration, and *wstudent* is a primary class of *pam*, *pam* is a direct instance of *wstudent*, and *pam* has values "*Pam*" and *tom* for attributes *name* and *spouse*, respectively.

We require that each oid has a unique primary class. We also require object declarations to be well typed and consistent with respect to the schema. We discuss these issues in Section 4.2.

Definition 3.14. 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*.

3.3 Database and Query Syntax

A database consists of a schema and an instance, and a query is a sequence of expressions.

Definition 3.15. 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 $I = (\pi, \lambda)$.

Definition 3.16. A *query* is a sequence of expressions prefixed with ?–.

4. 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, we then 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.

Encapsulation is dealt with in Section 4.1; 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.

4.1 Semantics of Schema and Instance

In this section, we define the meaning of the schema and instance of a database. Recall that $\alpha(c)$, $\delta(c)$ and $\mu(c)$ are the sets of attribute declarations, default values and method declarations respectively that are defined on c. We also define $\alpha^*(c)$, $\delta^*(c)$, and $\mu^*(c)$, the attribute declarations, default values and method declarations that are applicable to class c, taking inheritance, overriding, conflict resolution and blocking into account.

Consider the schema in Figure 1. There are no attributes defined on *wstudent* so $\alpha(wstudent) = \emptyset$. However, the attributes that are applicable to objects in class *wstudent* are

$$
\begin{aligned}
&\alpha^*(wstudent) = \{\\
&\quad wstudent\ [name \Rightarrow string],\\
&\quad wstudent\ [birthyear \Rightarrow integer],\\
&\quad wstudent\ [spouse \Rightarrow person],\\
&\quad wstudent\ [workphone \Rightarrow integer],\\
&\quad wstudent\ [salary \Rightarrow integer],\\
&\quad wstudent\ [emergencyContact \Rightarrow integer],\\
&\quad wstudent\ [major \Rightarrow string]\\
&\}.
\end{aligned}
$$

Definition 4.2 specifies how $\alpha^*(c)$, $\delta^*(c)$ and $\mu^*(c)$ are calculated using difference operators to find declarations that are defined in one class but not in another class. We start by defining the difference operators. There are no restrictions on the relationship between the classes on which they operate. However, they are generally used to find declarations that are defined in a superclass and not redefined in the subclass. There are different operators to find the difference between attribute declarations, default value declarations and method declarations, respectively. For example, we would use the attribute declaration difference operator to find attributes that are declared in *person* and not redefined in *employee*. Note that in this definition and elsewhere in the article, the existential quantifier indicates that all variables in the expression are existentially quantified unless we state otherwise.

Definition 4.1. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. For each $c \in C$, let $\alpha_{\mathcal{DB}}(c)$ denote a set of attribute declarations, $\delta_{\mathcal{DB}}(c)$ a set of default value declarations, and $\mu_{\mathcal{DB}}(c)$ a set of method declarations for class c. Then for any $c_1, c_2 \in C$, the *difference* between sets $\alpha_{\mathcal{DB}}(c_1)$ and $\alpha_{\mathcal{DB}}(c_2)$ is defined as

$$\alpha_{\mathcal{DB}}(c_1) - \alpha_{\mathcal{DB}}(c_2) = \{c\ [l \Rightarrow d] \mid c\ [l \Rightarrow d] \in \alpha(c_1) \text{ and } \nexists\, c'\ [l \Rightarrow d'] \in \alpha(c_2)\}.$$

The *difference* between sets $\delta_{\mathcal{DB}}(c_1)$ and $\delta_{\mathcal{DB}}(c_2)$ is defined as

$$\delta_{\mathcal{DB}}(c_1) - \delta_{\mathcal{DB}}(c_2) = \{c\ [l \bullet\!\!\rightarrow o] \mid c\ [l \bullet\!\!\rightarrow o] \in \delta(c_1) \text{ and } \nexists\, c'\ [l \bullet\!\!\rightarrow o'] \in \delta(c_2) \text{ and } \nexists\, c''\ [l \Rightarrow c_r] \in \alpha(c_2) \text{ and } \nexists\, c'''\ [l \lhd c_r'] \in \chi(c_2)\}.$$

The *difference* between sets $\mu_{\mathcal{DB}}(c_1)$ and $\mu_{\mathcal{DB}}(c_2)$ is defined as

$$\mu_{\mathcal{DB}}(c_1) - \mu_{\mathcal{DB}}(c_2) = \{M_1 \mid M_1 \in \mu(c_1) \text{ and } \nexists\, M_2 \in \mu(c_2) \text{ such that } sig(M_1) = sig(M_2)\}.$$

Example 4.1. Consider the database in Figure 1. The difference between the sets of attribute declarations for *person* and *student* is:

$\alpha(person) - \alpha(student) = \{$
person [*name* ⇒ *string*],
person [*birthyear* ⇒ *integer*],
person [*homephone* ⇒ *integer*],
person [*spouse* ⇒ *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 really surprising because the default value for *birthyear* and the attribute *emergencyContact* are redefined in *student*.

Recall how overriding, conflict resolution, and blocking are dealt with. An attribute declaration, default value, or method declaration that redefines a declaration in a superclass overrides the declaration in the superclass. Conflicts can be resolved in one of two ways, either implicitly due to the ordering of the superclasses in the class declaration, or explicitly by stating the class the property is to be inherited from. To block the inheritance of a declaration from a superclass in a subclass, we redefine the attribute or method with a return class *none*. In other words, to block the inheritance of an attribute l or method $m(c_1, \ldots, c_n)$ in subclass c, we include $c\ [l \Rightarrow none]$ or $c\ [m(c_1, \ldots, c_n) \Rightarrow none]$, respectively, in the class declaration of c.

Definition 4.2 defines the attribute declarations, default values, and method declarations that are applicable to a class taking inheritance with overriding, conflict handling and blocking into account. There are two cases considered in the definition, namely when a class inherits from no superclasses, and when a class inherits from at least one superclass. In the first case, no inheritance takes place, so the declarations that are defined on the class are the only ones that apply to that class. In the second case, we must consider inheritance, overriding, conflict resolution and blocking. We consider each of these factors separately in the four subparts of the definition. Each part extends the previous part of the definition. The expressions α_{bc} (respectively, δ_{bc}, μ_{bc}), and α_{bci} (respectively, δ_{bci}, μ_{bci}) are introduced to clarify the definition and only α^*, δ^*, and μ^* are referred to outside this section in this article.

Expressions α_{bc}, δ_{bc}, and μ_{bc} are the resulting sets of declarations when explicit conflict resolution is considered, α_{bci}, δ_{bci}, and μ_{bci} are the resulting sets when inheritance, overriding and implicit conflict resolution have been considered, and α^*, δ^*, and μ^* are the sets of resulting rewritten declarations.

We provide some examples, before the formal definition.

Example 4.2. For the class *person* in Figure 1 that inherits from no other classes, the properties that are applicable to class *person* are those that have been explicitly declared on *person*:

$$
\begin{aligned}
&\alpha^*(person) = \alpha_{bci}(person) = \alpha(person) = \{\\
&\quad person\ [name \Rightarrow string],\\
&\quad person\ [birthyear \Rightarrow integer],\\
&\quad person\ [homephone \Rightarrow integer],\\
&\quad person\ [spouse \Rightarrow person],\\
&\quad person\ [emergencyContact \Rightarrow integer]\\
&\}.
\end{aligned}
$$

Now consider a class that inherits from at least one other class.

Example 4.3. Consider the class *employee* in Figure 1, that has two attribute declarations and one attribute blocking declaration.

$$
\begin{aligned}
&\alpha_{bc}(employee) = \{\\
&\quad employee\ [workphone \Rightarrow integer],\\
&\quad employee\ [salary \Rightarrow integer],\\
&\quad employee\ [homephone \Rightarrow none]\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&\delta_{bc}(employee) = \{\\
&\quad employee\ [birthyear \bullet\!\!\rightarrow 1960],\\
&\quad employee\ [salary \bullet\!\!\rightarrow 2000]\\
&\}.
\end{aligned}
$$

Consider also the class *wstudent* that has one attribute conflict resolution declaration. The conflict resolution declaration states that the definition of *birthyear* is to be inherited from *student*, which in turn has inherited the definition from *person*. Due to the conflict resolution declaration, the default value for *birthyear* is inherited from *student*.

$$
\begin{aligned}
&\alpha_{bc}(wstudent) = \{\\
&\quad person\ [birthyear \Rightarrow integer]\\
&\}
\end{aligned}
$$

$\delta_{bc}(wstudent) = \{$
  $student\ [birthyear \bullet\!\!\rightarrow 1970]$
$\}.$

The sets $\alpha_{bci}(c)$, $\delta_{bci}(c)$, $\mu_{bci}(c)$ are the sets of declarations that are declared explicitly on c or inherited from the superclasses of c, taking overriding and conflict resolution into account.

Example 4.4. Consider the class *wstudent* in Figure 1, that inherits from *employee* and *student*.

$\alpha_{bci}(wstudent) = \{$
  $person\ [birthyear \Rightarrow integer],$
  $employee\ [workphone \Rightarrow integer],$
  $employee\ [salary \Rightarrow integer],$
  $employee\ [homephone \Rightarrow none],$
  $person\ [emergencyContact \Rightarrow integer],$
  $person\ [name \Rightarrow string],$
  $person\ [spouse \Rightarrow person],$
  $student\ [major \Rightarrow string]$
$\}$

$\delta_{bci}(wstudent) = \{$
  $person\ [emergencyContact \bullet\!\!\rightarrow 911],$
  $employee\ [salary \bullet\!\!\rightarrow 2000],$
  $student\ [birthyear \bullet\!\!\rightarrow 1970],$
  $student\ [major \bullet\!\!\rightarrow \text{“}CS\text{”}]$
$\}.$

In other words, $\alpha_{bci}(wstudent)$ is the set of attributes that are derived from the set $\alpha_{bc}(wstudent)$, and the attributes that are defined on *employee* and not redefined in *wstudent*, and the attributes that are defined on *student* and not redefined in *employee* or *wstudent*. Similarly for $\delta_{bci}(wstudent)$.

The sets $\alpha^*(c)$, $\delta^*(c)$, $\mu^*(c)$ 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.

Example 4.5. Consider the class *wstudent* in Figure 1:

$\alpha^*(wstudent) = \{$
  $wstudent\ [name \Rightarrow string],$
  $wstudent\ [birthyear \Rightarrow integer],$
  $wstudent\ [spouse \Rightarrow person],$
  $wstudent\ [workphone \Rightarrow integer],$
  $wstudent\ [salary \Rightarrow integer],$
  $wstudent\ [emergencyContact \Rightarrow integer],$
  $wstudent\ [major \Rightarrow string]$
$\}$

$\delta^*(wstudent) = \{$
$\quad wstudent\ [emergencyContact \bullet\!\!\rightarrow 911],$
$\quad wstudent\ [salary \bullet\!\!\rightarrow 2000],$
$\quad wstudent\ [birthyear \bullet\!\!\rightarrow 1970],$
$\quad wstudent\ [major \bullet\!\!\rightarrow \text{“}CS\text{”}]$
$\}.$

Definition 4.2. The semantics of multiple inheritance with overriding, conflict handling and blocking are defined using the difference operators as follows:

(1) If there does not exist a class c' such that $c\ isa\ c'$, then

$$\alpha^*(c) = \alpha_{bci}(c) = \alpha(c)$$
$$\delta^*(c) = \delta_{bci}(c) = \delta(c)$$
$$\mu^*(c) = \mu_{bci}(c) = \mu(c).$$

In other words, if a class does not have any superclasses, then there is no inheritance, overriding, conflict resolution or blocking. The declarations in the class are the only ones that apply to the class.

(2) Otherwise, if $c\ isa\ c_1, \ldots, c_n$, then

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

$$\alpha_{bc}(c) = \alpha(c) \cup \{c''\ [l \Rightarrow c_r] \mid \exists\, c\ [l \lhd c'] \in \chi_\alpha(c) \text{ and } c''\ [l \Rightarrow c_r] \in \alpha_{bci}(c')\}$$
$$\delta_{bc}(c) = \delta(c) \cup \{c''\ [l \bullet\!\!\rightarrow o] \mid \exists\, c\ [l \lhd c'] \in \chi_\alpha(c) \text{ and } \exists\, c''\ [l \bullet\!\!\rightarrow o] \in \delta_{bci}(c') \text{ and } \nexists\, c\ [l \bullet\!\!\rightarrow o'] \in \delta(c)\}$$
$$\mu_{bc}(c) = \mu(c) \cup \{M \mid \exists\, c\ [m(c_1, \ldots, c_n) \lhd c'] \in \chi_\mu(c) \text{ and } M \in \mu_{bci}(c') \text{ such that } sig(M) = m(c_1, \ldots, c_n)\}.$$

(b) we extend the sets of declarations to include declarations that are inherited from both direct and indirect superclasses using the difference operator in Definition 4.1

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

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

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

$$\alpha^*(c) = \{c\ [l \Rightarrow c'] \mid \exists\, c''\ [l \Rightarrow c'] \in \alpha_{bci}(c) \text{ and } c' \neq none\}$$
$$\delta^*(c) = \{c\ [l \bullet\!\!\rightarrow o] \mid \exists\, c'\ [l \bullet\!\!\rightarrow o] \in \delta_{bci}(c) \text{ and } \nexists\, c''\ [l \Rightarrow none] \in \alpha_{bci}(c)\}$$
$$\mu^*(c) = \{M' \mid \exists\, M \in \mu_{bci}(c), \text{ the type of } M \text{ is } c'\ [m(c_1, \ldots, c_n) \Rightarrow c_r]\ c_r \neq none, \text{ and } M' \text{ is obtained from } M \text{ by substituting } c \text{ for } c'\}.$$

Definition 4.3. 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 $\mu^* = \{\mu^*(c) \mid c \in C\}$.

Definition 4.4. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. 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 nonmonotonic inheritance within a database schema. We now define 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 $c\ isa\ c_1, \ldots, c_n$, 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 $c\ isa^*\ c$.

Definition 4.5. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database, c a class and o an object. Then o is a *nondirect instance* of c in $\mathcal{DB}$, denoted by $c\ o \in \pi^*$, if and only if one of the following holds:

(1) c is a value class, o is a value, and o is an element in the collection that c denotes.

(2) 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 *nondirect* instance of c and the superclasses of c.

For example, if $wstudent\ pam \in \pi$ and $wstudent\ isa^*\ person$, then we have $person\ pam \in \pi^*$. In other words, π contains direct instances while π^* contains both direct and nondirect instances.

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:

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

Definition 4.6. 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^*.

4.2 Constraints on a Database

Using the syntax introduced, it is possible to define a database that has no intuitive meaning. For example it is possible to define a database with a cycle in its class hierarchy, or an attribute in a class that has two distinct default values. In this section, we discuss a number of constraints that can be used to guarantee an intended semantics of the database and queries on the database. The following definitions introduce constraints on the database schema.

Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. Recall that 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, χ is a finite set of conflict resolution declarations, μ is a finite set of method declarations, π is a set of ground oid membership expressions, and λ is a set of ground positive attribute expressions.

Definition 4.7. A set S of *superclass declarations* is *well defined* if and only if there do not exist distinct c and c' such that $c\ isa^*\ c'$ and $c'\ isa^*\ c$.

Informally, a set of superclass declarations is well defined if there are no cycles in the inheritance hierarchy. For example, the following set is not well defined:

$$\{wstudent\ isa\ employee,\ wstudent\ isa\ student,\ student\ isa\ wstudent\}.$$

When there are cycles in the inheritance hierarchy, α^*, δ^* and μ^* cannot be defined by Definitions 4.2 and 4.3.

Definition 4.8. The consistency constraint for attribute declarations is defined as follows:

—A set S of *attribute declarations* is *consistent* if and only if there do not exist $c\ [l \Rightarrow c'] \in S$ and $c\ [l \Rightarrow c''] \in S$ such that $c' \neq c''$.
—A set S of *attribute declarations* is *consistent with respect to* χ if and only if there do not exist $c\ [l \Rightarrow c_r] \in S$ and $c\ [l \lhd c_l] \in \chi$.

Informally, a set of attribute declarations is consistent if each attribute has only one return class. For example, the following set is not consistent.

$$\{person\ [spouse \Rightarrow integer],\ person\ [spouse \Rightarrow string]\}.$$

A set of attribute declarations is consistent with respect to a set of conflict resolution declarations if there does not exist an attribute declaration and a conflict resolution declaration for the same attribute. Consider now an example where $S = \{wstudent\ [homephone \Rightarrow string]\}$ and $wstudent\ [homephone \lhd student] \in \chi$, then S is not consistent with respect to χ.

Definition 4.9. A set S of *default value declarations* is *consistent* if and only if there do not exist $c\ [l \bullet\!\!\to o] \in S$ and $c\ [l \bullet\!\!\to o'] \in S$ such that $o \neq o'$.

Informally, a set of default value declarations is consistent if there is at most one default value for each attribute in a class. For example, the following set is not consistent.

$$\{student\ [birthyear \bullet\!\!\to 1979],\ student\ [birthyear \bullet\!\!\to 1980]\}.$$

Definition 4.10. The consistency constraint for method declarations is defined as follows:

—A set S of *method declarations* is *consistent* if and only if there do not exist two distinct method declarations $c\ [M] \in S$ and $c\ [M'] \in S$ that have the same signature.
—A set S of *method declarations* is *consistent with respect to* χ if and only if there do not exist $c\ [M] \in S$ and $c\ [sig(M) \lhd c'] \in \chi$.

Informally, a set of method declarations is consistent if each method has only one return class. This definition is similar to the definition for attribute declarations above, except in this case, we must consider the signature of the methods.

For example, $\{person\ [married\_to(person)], person\ [married\_to(person) \Rightarrow person]\}$ is not consistent because both methods in the set have the same signature, but the set $\{person\ [married\_to()], person\ [married\_to(person)]\}$ is consistent because the two methods in the set do not have the same signature. A set of method declarations is consistent with respect to a set of conflict resolution declarations if there does not exist a method declaration and a conflict resolution declaration for the same method. Consider now an example where $S = \{wstudent\ [support() \Rightarrow integer]\}$ and $wstudent\ [support() \lhd student] \in \chi$, then S is not consistent with respect to χ. The set $S = \{wstudent\ [support(integer) \Rightarrow integer]\}$ is consistent with respect to χ because the method $support(integer)$ declared in S does not have the same signature as $support()$ in χ.

Definition 4.11. The consistency constraint for conflict resolution declarations is defined as follows:

—A set S of *attribute conflict resolution declarations* is *consistent* if and only if there do not exist $c\ [l \lhd c'] \in S$ and $c\ [l \lhd c''] \in S$ such that $c' \neq c''$.

—A set S of *method conflict resolution declarations* is *consistent* if and only if there do not exist $c\ [m(c_1, \ldots, c_n) \lhd c'] \in S$ and $c\ [m(c_1, \ldots, c_n) \lhd c''] \in S$ such that $c' \neq c''$.

—Let S be a set of conflict resolution declarations, such that $S = S_\alpha \cup S_\mu$ where S_α is a set of attribute conflict resolution declarations and S_μ is a set of method conflict resolution declarations. The set S is *consistent* if and only if both S_α and S_μ are consistent.

Informally, a set of conflict resolution declarations is consistent if each attribute or method in the set is inherited from only one class. For example, the following set is not consistent:

$$\{wstudent\ [birthyear \lhd student],\ wstudent\ [birthyear \lhd person]\}.$$

Definition 4.12. The definition of well-defined attribute (respectively, method) declarations is defined as follows:

—An attribute blocking declaration $c\ [l \Rightarrow none] \in \alpha(c)$ is *well defined* in $\mathcal{DB}$ if and only if (1) $c\ isa\ c_1, \ldots, c_l$; (2) $\exists\, c_i\ [l \Rightarrow c'] \in \alpha^*(c_i)$ for $1 \leq i \leq l$; and (3) $\nexists\, c_j\ [l \Rightarrow c''] \in \alpha^*(c_j)$ for $1 \leq j \leq i$.

—A method blocking declaration $c\ [m(c'_1, \ldots, c'_n) \Rightarrow none] \in \mu(c)$ is *well defined* in $\mathcal{DB}$ if and only if (1) $c\ isa\ c_1, \ldots, c_l$; (2) $\exists\ M \in \mu^*(c_i)$ for $1 \leq i \leq l$ such that $sig(M) = m(c'_1, \ldots, c'_n)$; (3) $\nexists\, M' \in \mu^*(c_j)$ for $1 \leq j \leq i$ such that $sig(M) = m(c'_1, \ldots, c'_n)$.

—A set S of *attribute declarations* (respectively, method declarations) is *well defined* in $\mathcal{DB}$ if and only if each attribute (respectively, method) blocking declaration is well defined in $\mathcal{DB}$.

Informally, a blocking declaration is well defined if it blocks an attribute (or method) that has been defined on or inherited by a parent class. For example, if *student isa person* and $person\ [name \Rightarrow string] \notin \alpha^*(person)$, the blocking declaration $student\ [name \Rightarrow none]$ is not well defined.

Definition 4.13. The definition of well defined conflict resolution declarations is defined as follows:

—An *attribute conflict resolution declaration* $c\ [l \lhd c']$ is *well defined* in $\mathcal{DB}$ if and only if (1) c *isa* c'; and (2) $\exists\, c'\ [l \Rightarrow c''] \in \alpha^*(c')$.

—A *method conflict resolution declaration* $c\ [m(c_1, \ldots, c_n) \lhd c']$ is *well defined* in $\mathcal{DB}$ if and only if (1) c *isa* c'; and (2) $\exists\, M \in \mu^*(c')$ such that $sig(M) = m(c_1, \ldots, c_n)$.

—A set S of *conflict resolution declarations* is *well defined* in $\mathcal{DB}$ if and only if each conflict resolution declaration is well defined in $\mathcal{DB}$.

Informally, a conflict resolution declaration is well defined if the attribute or method is defined on the class it is to be inherited from, and a set of conflict resolution declarations is well defined if every conflict resolution declaration in the set is well defined. For example, if *student* [*birthyear* $\Rightarrow$ *integer*] $\in \alpha^*$(*student*) and *wstudent isa student*, then *wstudent* [*birthyear* $\lhd$ *student*] is well defined. However, if *birthyear* was not defined on student and was not inherited from any of the superclasses of student, then *wstudent* [*birthyear* $\lhd$ *student*] would not be well defined.

Definition 4.14. The definition of well typed default value declarations is defined as follows:

—A default value declaration $c\ [l \bullet\!\!\to o]$ is *well typed* in $\mathcal{DB}$ if and only if (1) there exists an attribute declaration $c\ [l \Rightarrow c'] \in \alpha^*(c)$; and (2) $c'\ o \in \pi^*$.

—A set S of default value declarations is *well typed* in $\mathcal{DB}$ if each default value declaration in S is well typed in $\mathcal{DB}$.

Informally, a default value declaration is well typed if the default value of the attribute matches the return class of the attribute, and a set of default value declarations is well typed if every default value declaration in the set is well typed. For example, if *student* [*birthyear* $\Rightarrow$ *integer*] $\in \alpha^*$(*student*), then *student* [*birthyear* $\bullet\!\!\to$ 1970] is well typed, but *student* [*birthyear* $\bullet\!\!\to$ "1970*AD*"] is not well typed.

Based on the previous definitions, we have the following property: The properties demonstrate that the sets of expressions defined in this section have the intended semantics.

PROPOSITION 4.1. *Let* $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ *be a database. If isa is well defined*, α, δ, μ, *and* χ *are consistent,* α *and* μ *are consistent with respect to* χ, *then* α^*, δ^*, μ^* *are defined and consistent.*

PROOF. When *isa* is well defined, it is straightforward that α^*, δ^*, μ^* are defined by Definitions 4.2 and 4.3.

For consistency, consider first α^*. Suppose α^* is not consistent, then for some c, there exists $c\ [l \Rightarrow c'] \in \alpha^*$ and $c\ [l \Rightarrow c''] \in \alpha^*$ such that $c' \neq c''$. For any class c, $\alpha(c)$ is consistent, so an expression must be added in the evaluation of $\alpha^*(c)$ to make the above true.

Suppose that c has no superclasses, then $\alpha^*(c) = \alpha(c)$, so $\alpha^*(c)$ is consistent.

Suppose next that c has superclasses $c_1, \ldots, c_n$, then

(1) because $\alpha(c)$ is consistent with respect to $\chi_\alpha(c)$, $\alpha_{bc}(c)$ cannot contain two expressions $c_1\ [l \Rightarrow c_2]$, $c'_1\ [l \Rightarrow c'_2]$ such that $c_1 \neq c'_1$,
(2) $\alpha_{bci}(c)$ cannot contain two expressions $c_1\ [l \Rightarrow c_2]$, $c'_1\ [l \Rightarrow c'_2]$ such that $c_1 \neq c'_1$ by the definition of the difference operator, and
(3) no new expressions are added when $\alpha^*(c)$ is derived from $\alpha_{bci}(c)$.

Thus, α^*, must be consistent.

A similar case can be made for δ^* and μ^*. □

The following three definitions introduce constraints on the database instance. A database instance does not have an intuitive meaning if an object is a direct instance of more than one class; if an attribute has more than one value for an object; or if attribute values of objects are not well typed.

Definition 4.15. A set S of oid assignments is *consistent* if and only if there do not exist $c\ o \in S$ and $c'\ o \in S$ such that $c \neq c'$.

Informally, a set of oids is consistent if each oid belongs directly to at most one class. For example, the set {*person tom*, *employee tom*} is not consistent.

Definition 4.16. A set S of attribute value assignments is *consistent* if and only if there do not exist $o\ [l \rightarrow o'] \in S$ and $o\ [l \rightarrow o''] \in S$ such that $o' \neq o''$.

Informally, a set of attribute assignments is consistent if each attribute has a single value. For example, the set {*tom* [*name* → "*Tom*"], *tom* [*name* → "*Thomas*"]} is not consistent.

Proposition 4.2. *Let* $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ *be a database. If* π, λ *and* δ^* *are consistent, Then* λ^* *is defined and consistent.*

Proof. Suppose λ^* is not consistent, then there exists $o\ [l \rightarrow o'] \in \lambda^*$ and $o\ [l \rightarrow o''] \in \lambda^*$ such that $o' \neq o''$. Because λ is consistent, there must be an expression added when λ^* is evaluated to make the above true. However, in the evaluation of λ^*, new expressions are added only when there does not exist an expression $o\ [l \rightarrow o'] \in \lambda$, or when there is a default value for instances of the class that the object belongs to. In the latter case, because π and δ^* are consistent by Proposition 4.1, only one expression will be added for object o. Thus, λ^* must be consistent. □

Based on Propositions 4.1 and 4.2, we have the following corollary.

Corollary 4.1. *Let* $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ *be a database. If isa is well defined,* α, δ, μ, *and* χ *are consistent,* α *and* μ *are consistent with respect to* χ, *then* $\alpha^*, \delta^*, \mu^*, \lambda^*$ *are defined and consistent.*

Definition 4.17. The definition of well typed attribute value assignments is defined as follows:

—Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. An attribute value assignment $o\ [l \rightarrow o']$ is *well typed* in $\mathcal{DB}$ if and only if

—there exists a class c such that $c\ o \in \pi$;

—there exists an attribute declaration $c\ [l \Rightarrow c'] \in \alpha^*(c)$; and

—$c'\ o' \in \pi^*$.

—Let $\mathcal{DB}$ be a database. A set S of attribute value assignments is *well typed* in $\mathcal{DB}$ if each attribute value assignment in S is well typed in $\mathcal{DB}$.

This constraint is similar to the well typed definition for default value declarations except that in this definition we must also account for the class of the oid. For example, *sam* [*workphone* $\rightarrow$ 8742911] is not well typed in the database in Figure 1, where *student sam* $\in \pi$ but *student* [*workphone* $\Rightarrow$ *integer*] $\notin \alpha^*(\textit{student})$.

Definition 4.18. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a database. Then $\mathcal{DB}$ is *well defined* if and only if

(1) *isa* is well defined;

(2) α is consistent, consistent with respect to χ and well defined;

(3) δ is consistent and well typed in $\mathcal{DB}$;

(4) μ is consistent, consistent with respect to χ, and well defined;

(5) χ is consistent and well defined in $\mathcal{DB}$;

(6) π is consistent; and

(7) λ is consistent and well typed in $\mathcal{DB}$.

A well defined database $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ incorporates all the constraints discussed above. Therefore, α^*, δ^* and μ^* provide intuitive semantics for the schema, and π^* and λ^* provide intuitive semantics for the instance.

In the following section, we are concerned only with well defined databases; that is, databases with an intuitive meaning.

4.3 Semantics of Databases and Queries

In this article, we focus on static databases rather than dynamic databases, that is, databases where classes of oids and their attribute values remain the same. The semantics for dynamic databases can be found in Liu [1998a]. 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 Gelder et al. [1991].

Our definition differs from Gelder et al. [1991] 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 Gelder et al. [1991] and more complex in our definition because we define the truth values for our 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, in which oid membership and attribute expressions are true if they're in the model; otherwise, they are false. The IDB is a 3-valued model, in which 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. The reason we use a 3-valued model for IDB is that we can infer both positive and negative method expressions using method rules. On the other hand, EDB only contains positive oid membership and attribute expressions so we just use a 2-valued model.

In the well-founded semantics, a program may have a partial model. This is not the case in our definition, in fact we prove that every well-defined program has a minimal model. We first define terminology that is needed later in this section.

Definition 4.19. For each simple method expression, ψ or $\neg\psi$, we say that ψ is an atom.

Definition 4.20. 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 method expressions formed using the method names in $\mathcal{DB}$ (without abbreviations).

Example 4.6. Consider the following well-defined database DB:

```
class person [
        spouse ⇒ person;
        married() {
                married() :– spouse → X;
                married() :– X.married(), X.spouse → This}
        single(){
                single() :– ¬married()}
        canDivorce(){
                canDivorce() :– married()}
        ]

person tom  [spouse → pam]
person pam
person sam.
```

Consider the following sets of expressions:

$$\begin{aligned} E_1 = \{&tom.married(),\ pam.married(),\ \neg sam.married(),\\ &\neg tom.single(),\ \neg pam.single(),\ sam.single(),\\ &tom.canDivorce(),\ pam.canDivorce(),\ \neg sam.canDivorce()\}\end{aligned}$$

$$E_2 = \{tom.spouse \rightarrow pam\}.$$

Then $E_1 \subset \mathcal{B}_{DB}$, but $E_2 \not\subset \mathcal{B}_{DB}$ because the expression in E_2 is not a method expression.

We are interested in compatible, consistent, and well typed subsets of the Herbrand Base. A set of method expressions is incompatible if it contains an atom and the complement of that atom.

Definition 4.21. For a set of method expressions S, we denote the set formed by taking the complement of each expression in S by $\neg S$.

—We say expression q is *incompatible* with S if $q \in \neg S$.

—Sets of expressions R and S are *incompatible* if some expression in R is incompatible with S; that is, if $R \cap \neg S \neq \emptyset$ or $\neg R \cap S \neq \emptyset$.

—A set of expressions is *incompatible* if it is incompatible with itself; otherwise, it is *compatible*.

Example 4.7. Consider the following set E_3:

$$E_3 = \{\neg tom.married(),\ tom.married(),\ \neg pam.married(),\ \neg sam.married(),\ pam.single(),\ \neg sam.single(),\ sam.canDivorce()\}.$$

It is incompatible because $\{\neg tom.married(), tom.married()\} \in E_3$, and $E_3 \cap \neg E_3 \neq \emptyset$. The set E_1 in Example 4.6 is compatible.

Ground method expressions are required to be well typed with respect to the appropriate class declarations.

Definition 4.22. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database, and $\psi = o.m(o_1, \ldots, o_n) \rightarrow o_r$ or $\psi = \neg o.m(o_1, \ldots, o_n) \rightarrow o_r$ a ground method expression. Then ψ is *well typed* in $\mathcal{DB}$ if and only if 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 $c\ [m(c_1, \ldots, c_n) \Rightarrow c_r]$ 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 $\mathcal{DB}$ if and only if 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.

Definition 4.23. A set of ground method expressions are *consistent* if and only if there do not exist $o.m(o_1, \ldots, o_n) \rightarrow o_r \in S$ and $o.m(o_1, \ldots, o_n) \rightarrow o'_r \in S$ such that $o_r \neq o'_r$.

The set E_1 in Example 4.6 is compatible, consistent, and well typed in the database DB.

Definition 4.24. 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.

Example 4.8. The following are two interpretations of the database DB in Example 4.6:

$$\begin{aligned} I_1 = (&\{person\ tom,\ person\ pam,\ person\ sam\},\\ &\{tom.spouse \to pam\},\\ &\emptyset)\end{aligned}$$

$$\begin{aligned} I_2 = (&\{person\ tom,\ person\ pam,\ person\ sam\},\\ &\{tom.spouse \to pam\},\\ &\{tom.married(),\ pam.married(),\ \neg sam.married(),\\ &\ \neg tom.single(),\ \neg pam.single(),\ sam.single(),\\ &\ tom.canDivorce(), pam.canDivorce(),\ \neg sam.canDivorce()\}.\end{aligned}$$

Definition 4.25. A *ground substitution* θ is a mapping from $\mathcal{V}$ to $\mathcal{O} \cup \mathcal{D}$. It is extended to terms and expressions in the usual way.

Definition 4.26. 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) For a ground positive oid membership expression ψ, $\mathcal{I} \models \psi$ if and only if $\psi \in \pi^*$; $\mathcal{I} \not\models \psi$ if and only if $\psi \notin \pi^*$.
(2) For a ground positive attribute expression ψ, $\mathcal{I} \models \psi$ if and only if $\psi \in \lambda^*$; $\mathcal{I} \not\models \psi$ if and only if $\psi \notin \lambda^*$.
(3) For a ground negative oid membership expression or attribute expression $\neg\psi$, $\mathcal{I} \models \neg\psi$ if and only if $\mathcal{I} \not\models \psi$; $\mathcal{I} \not\models \neg\psi$ if and only if $\mathcal{I} \models \psi$.
(4) For a ground positive method expression ψ, $\mathcal{I} \models \psi$ if and only if $\psi \in \mathcal{S}$; $\mathcal{I} \not\models \psi$ if and only if $\neg\psi \in \mathcal{S}$.
(5) For a ground negative method expression $\neg\psi$, $\mathcal{I} \models \neg\psi$ if and only if $\neg\psi \in \mathcal{S}$; $\mathcal{I} \not\models \neg\psi$ if and only if $\psi \in \mathcal{S}$.
(6) For a ground composite expression $\psi = c\ o\ [V_1; \ldots; V_n]$,
—$\mathcal{I} \models \psi$ if and only if $\mathcal{I} \models c\ o$, $\mathcal{I} \models o.V_i$ for $1 \le i \le n$;
—$\mathcal{I} \not\models \psi$ if and only if $\mathcal{I} \not\models c\ o$ or $\mathcal{I} \not\models o.V_i$ for some i with $1 \le i \le n$.
For a ground composite expression $\psi = o\ [V_1; \ldots; V_n]$,
—$\mathcal{I} \models \psi$ if and only if $\mathcal{I} \models o.V_i$ for every $1 \le i \le n$;
—$\mathcal{I} \not\models \psi$ if and only if $\mathcal{I} \not\models o.V_i$ for some i with $1 \le i \le n$.
(7) For a ground arithmetic comparison expression ψ, $\mathcal{I} \models \psi$ if and only if ψ holds in the standard arithmetic interpretation; $\mathcal{I} \not\models \psi$ if and only if ψ does not hold in the standard arithmetic interpretation.
(8) For a method rule $r = c\ [A :\!-\, L_1, \ldots, L_n]$, $I \models r$ if and only if 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 $1 \le i \le n$ such that $\mathcal{I} \not\models \theta L_i$; or
—there exists an L_i with $1 \le i \le n$ 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 $\mathcal{I}$.

Example 4.9. Consider the database DB in Example 4.6. The following is part of the ground method rules:

$$\begin{array}{l}
tom.married() \text{ :– } tom.spouse \rightarrow tom \\
tom.married() \text{ :– } tom.spouse \rightarrow pam \\
tom.married() \text{ :– } tom.spouse \rightarrow sam \\
tom.married() \text{ :– } tom.married(), tom.spouse \rightarrow tom \\
tom.married() \text{ :– } pam.married(), pam.spouse \rightarrow tom \\
tom.married() \text{ :– } sam.married(), sam.spouse \rightarrow tom \\
tom.canDivorce() \text{ :– } tom.married().
\end{array}$$

We can verify that the interpretation I_2 of DB in Example 4.8 satisfies every ground method rule in DB while the interpretation I_1 does not satisfy most of the rules.

Definition 4.27. 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 method rule in μ^*.

Consider again Example 4.9, the interpretation I_2 is a model of DB while the interpretation I_1 is not.

Due to the typing and compatibility constraints as in ROL [Liu 1996], 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.

We now define unfounded sets. An unfounded set for a database with respect to an interpretation provides a basis for false method expressions in our semantics.

Definition 4.28. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database, $\mathcal{B}_{\mathcal{DB}}$ its Herbrand base, and $\mathcal{I} = (\pi, \lambda, S)$ be an interpretation. Then $U \subseteq \mathcal{B}_{\mathcal{DB}}$ is an *unfounded set* of $\mathcal{DB}$ with respect to $\mathcal{I}$ if each expression $\psi \in U$ satisfies the following condition:

For each ground method rule r of μ^* whose head is ψ, at least one of the following holds:

(1) There is some positive or negative expression L_i of the body such that $\mathcal{I} \not\models L_i$.

(2) Some positive expression of the body occurs in U.

Example 4.10. Consider the database DB in Example 4.6 and the following interpretation:

$$\begin{array}{l}
I = (\{person\ tom,\ person\ pam, person\ sam\}, \\
\qquad \{tom.spouse \rightarrow pam\}, \\
\qquad \emptyset).
\end{array}$$

The set $U = \{sam.married(), sam.canDivorce()\}$ is an unfounded set of DB with respect to I because every ground rule with head $sam.married()$ has an expression ψ in the body, such that $I \not\models \psi$, and the ground rule with the head *sam.canDivorce*() has *sam.married*() in U. The set $U' = \{tom.married()\}$ is not an unfounded set of *DB* with respect to I because neither (1) nor (2) (from Definition 4.26) holds for the method rules with head *tom.married*().

The greatest unfounded set 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.

Definition 4.29. The *greatest unfounded set with respect to* $\mathcal{I}$ (GUS) is the union of all sets that are unfounded with respect to $\mathcal{I}$.

We now continue to define the semantics of a database.

Definition 4.30. 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:

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

where

$$\mathcal{T}(\mathcal{I}) = \{\theta A \mid A \text{ :- } L_1, \ldots, L_n \text{ is a method rule in } \mathcal{DB} \text{ and there exists a ground substitution } \theta \text{ such that } \mathcal{I} \models \theta L_1, \ldots, \mathcal{I} \models \theta L_n\}$$

$\mathcal{U}(\mathcal{I}) = \neg G$, where G is the GUS of $\mathcal{DB}$ with respect to $\mathcal{I}$.

$\mathcal{W}(\mathcal{I}) = \mathcal{T}(\mathcal{I}) \cup \mathcal{U}(\mathcal{I})$.

Definition 4.31. 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 $\mathcal{I}^*$.

Example 4.11. Consider the database in Example 4.6.

$\mathcal{I}_0 = (\pi, \lambda, \emptyset)$.
$\mathcal{T}(\mathcal{I}_0) = \{tom.married()\}$,
$\mathcal{U}(\mathcal{I}_0) = \{\neg sam.married(),\ \neg sam.canDivorce()\}$
$\mathcal{W}(\mathcal{I}_0) = \mathcal{T}(\mathcal{I}_0) \cup \mathcal{U}(\mathcal{I}_0)$

$\mathcal{I}_1 = (\pi, \lambda,\ \mathcal{W}(\mathcal{I}_0))$
$\mathcal{T}(\mathcal{I}_1) = \mathcal{T}(\mathcal{I}_0) \cup \{tom.canDivorce(),\ pam.married(),\ sam.single()\}$,
$\mathcal{U}(\mathcal{I}_1) = \mathcal{U}(\mathcal{I}_0) \cup \{\neg tom.single()\}$,
$\mathcal{W}(\mathcal{I}_1) = \mathcal{T}(\mathcal{I}_1) \cup \mathcal{U}(\mathcal{I}_1)$

$$\begin{aligned}
&\mathcal{I}_2 = (\pi, \lambda,\ \mathcal{W}(\mathcal{I}_1))\\
&\quad \mathcal{T}(\mathcal{I}_2) = \mathcal{T}(\mathcal{I}_1) \cup \{pam.canDivorce()\},\\
&\quad \mathcal{U}(\mathcal{I}_2) = \mathcal{U}(\mathcal{I}_1) \cup \{\neg pam.single()\},\\
&\quad \mathcal{W}(\mathcal{I}_2) = \mathcal{T}(\mathcal{I}_2) \cup \mathcal{U}(\mathcal{I}_2)
\end{aligned}$$

$$\begin{aligned}
\mathcal{I}^* = (\pi,\ \lambda,\ &\{tom.married(),\ pam.married(),\ \neg sam.married(),\\
&sam.single(),\ \neg tom.single(),\ \neg pam.single(),\\
&tom.canDivorce(),\ pam.canDivorce(),\ \neg sam.canDivorce()\}).
\end{aligned}$$

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

THEOREM 4.1. *Let $\mathcal{DB}$ be a well-defined database. If $\mathcal{I}^* = (\pi, \lambda, S)$ is defined, then it is a model of $\mathcal{DB}$.*

PROOF. First, we prove that $\mathcal{I}^*$ is an interpretation, that is, S is compatible, consistent, and well typed in $\mathcal{DB}$. Assume that S is not compatible, then there is an expression A such that $A \in S$ and $\neg A \in S$. Now, $\neg A \in S$ only if for every ground rule $A :\!- L_1, \ldots, L_n$, there is an expression L_i for $1 \le i \le n$ such that $\mathcal{I}^* \not\models L_i$. Also, $A \in S$ if there is a ground rule $A :\!- L'_1, \ldots, L'_m$ such that $\mathcal{I}^* \models L'_i$ for $1 \le i \le m$. This is contradictory, so S is compatible. By Definition 4.30, S is consistent and well typed in $\mathcal{DB}$. Thus, $\mathcal{I}^*$ is an interpretation.

Next we prove that interpretation $\mathcal{I}^*$ is a model. Assume that $\mathcal{I}^*$ is not a model, then there is a method rule r in μ^* that is not satisfied in $\mathcal{I}^*$. By Definition 4.26, r has a ground instantiation $A :\!- L_1, \ldots, L_n$ such that $\mathcal{I}^* \not\models A$ and $\mathcal{I}^* \not\models \neg A$, and either

(1) $\mathcal{I}^* \models L_i$ for each $1 \le i \le n$, or
(2) there is an L_i with $1 \le i \le n$ such that $\mathcal{I}^* \not\models L_i$.

If $\mathcal{I}^* \models L_i$ for $1 \le i \le n$, then $A \in \mathcal{I}^*$ by Definitions 4.30 and 4.31. So Case (1) cannot occur. If there is an L_i with $1 \le i \le n$ such that $\mathcal{I}^* \not\models L_i$, then there are two possible situations to consider. One is that there is another ground rule $A :\!- L'_1, \ldots, L'_m$ such that $\mathcal{I}^* \models L'_i$ with $1 \le i \le m$. In this case, $A \in \mathcal{I}^*$. The other is for every other ground rule $A :\!- L''_1, \ldots, L''_o$, there is an expression L''_i, $1 \le i \le o$ such that $\mathcal{I}^* \not\models L''_i$. In this case, $\neg A \in \mathcal{I}^*$. Thus, both situations result in contradiction. So Case (2) cannot occur either. Thus, there are no ground method rules in μ^* that are not satisfied in $\mathcal{I}^*$. Therefore $\mathcal{I}^*$ is a model of $\mathcal{DB}$. □

Definition 4.32. 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 4.2. *Let $\mathcal{DB}$ be a well-defined database. If $\mathcal{I}^* = (\pi, \lambda, S)$ is defined, then it is a minimal model of $\mathcal{DB}$.*

PROOF. Suppose $\mathcal{I}^*$ is not a minimal model, then there is an expression ψ in S such that $N = (\pi,\ \lambda,\ S - \psi)$ is a model of $\mathcal{DB}$.

Suppose ψ is a positive method expression, then $\psi \in S$ because there is a rule $A :\!- L_1, \ldots, L_n$ in μ^* and a ground substitution θ such that $\psi = \theta A$ and

$\mathcal{I}^* \models \theta L_i$ for all $1 \leq i \leq n$. When ψ is removed, $N \not\models r$. So N is not a model of $\mathcal{DB}$.

Suppose ψ is a negative method expression, then $\psi \in S$ because for each ground rule r in μ^* with head ψ, there exists an expression L in the body such that $\mathcal{I}^* \not\models L$. When ψ is removed, $N \not\models r$. So N is not a model of $\mathcal{DB}$.

Since there exists no expression $\psi \in S$ such that $(\pi, \lambda, S - \psi)$ is a model of $\mathcal{DB}$, $\mathcal{I}^*$ is a minimal model. □

Definition 4.33. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database. The *semantics* of $\mathcal{DB}$ is represented by the limit $\mathcal{I}^*$ if it is defined.

Definition 4.34. Let $\mathcal{DB} = (C, isa, \alpha, \delta, \mu, \chi, \pi, \lambda)$ be a well-defined database, Q a query of the form ?–$L_1, \ldots, L_n$, 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 $\mathcal{I}^* \models \theta L_1, \ldots, \mathcal{I}^* \models \theta L_n$,

(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 ground expression ψ, if $\mathcal{I}^* \models \psi$, then ψ is true; if $\mathcal{I}^* \not\models \psi$, then ψ is false; and if neither $\mathcal{I}^* \models \psi$, nor $\mathcal{I}^* \not\models \psi$, then ψ is undefined.

Consider the database in Example 4.6 and the query:

?–*X.canDivorce*().

The ground substitutions $\{X = tom\}$ and $\{X = pam\}$ make the query true whereas the ground substitution $\{X = sam\}$ makes it false.

Let us consider an example with unknown answers.

Example 4.12. Consider the following database:

```
class person [
        spouse ⇒ person;
        married() {
                married() :– ¬single()}
        single(){
                single() :– ¬married()}
        ]

person sam [spouse → pam]
person pam.
```

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

?–*sam.married*()
?–*sam.single*().

There are two reasons why $\mathcal{I}^*$ may be undefined, according to Definitions 4.30 and 4.31. 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 Liu [1998b] to constrain the database. For the second problem, run-time checking is necessary.

5. 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 [Kifer et al. 1995].

The deductive object-oriented database research, however, has taken quite a different approach. It has logical semantics as its main objective and therefore starts with a small set of simple features taken from the object-oriented paradigm such as F-logic [Kifer et al. 1995], and gradually incorporates more and more difficult features that can be given a logical semantics such as ROL [Liu 1996] and Datalog++ [Jamil 1997].

The main contribution of this article 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 structural and behavioral inheritance, with overriding, blocking, and conflict handling. In the language defined in this article, methods are declared within class declarations, and the methods are invoked through instances of the classes. We have also given a semantics for multiple structural and behavioral inheritance with overriding, conflict resolution and blocking. We use a special class, *none*, to indicate that a method is blocked in a subclass. We provide a flexible conflict resolution mechanism that consists of two parts. One part allows the explicit naming of the class a property is be inherited from, using conflict resolution declarations. This part provides a lot of flexibility but it is difficult to check that all conflicts have been resolved using this mechanism alone. If there is no conflict resolution declaration, then the order of inheritance is determined by the order in which the superclasses are listed in the class declaration. This part alone provides little flexibility. Together, the two parts provide a flexible mechanism that guarantees that no conflicts will arise at runtime. The trickiest part in defining a semantics for nonmonotonic multiple inheritance was dealing with inheritance and conflict handling because the two concepts are inseparable. We define a class of databases, called *well-defined databases*, that have an intuitive meaning. The semantics of methods is based on well-founded semantics but differs from the well-founded semantics in a number of ways: we introduce typing and the concept of a well-typed database, the definition for satisfaction of expressions is more complex, and our model has two parts, a two-valued part that represents the extensional database and a three-valued part that represents the intensional database. We define a transformation that has a limit, $\mathcal{I}^*$ for well-defined databases, and prove that $\mathcal{I}^*$ is a minimal model of the database.

This article has shown that the object-oriented features that are believed to be difficult to address, can indeed be captured logically. We believe that the semantics given have a far reaching influence on the design of deductive object-oriented languages and even object-oriented languages in general as it is the first attempt to provide logical semantics for these common object-oriented features.

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. In contrast, we have provided a direct semantics for methods encapsulated in class definitions. Also, most existing deductive object-oriented database languages do not allow nonmonotonic multiple structural and behavioral inheritance. ROL does, but deals with conflict handling only in a limited context and doesn't have blocking. Datalog^{++} supports blocking but disallows the inheritance of conflicting properties. F-logic supports monotonic structural inheritance and indeterminate nonmonotonic 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.

We are not recommending the language presented in this article as a practical database query language, rather the language and semantics defined on the language would form the theoretical basis for a practical query language. Indeed, the implemented deductive object-oriented database language ROL2 [Liu and Guo 1998; Liu 1999] supports all the features discussed here.

We have provided a solid foundation in which other issues can be addressed. We briefly introduce one outstanding issue here, namely support for object migration. Object migration is another very important issue that has been ignored in most deductive object-oriented database languages to date, although it has been addressed more generally in the literature (e.g., Ling and Teo [1995]). The semantics of object migration could be studied in this setting.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous referees for their valuable comments and suggestions, which improved the technical content and the presentation of the article.

REFERENCES

ABITEBOUL, S. AND KANELLAKIS, P. C. 1998. Object identity as a query language. *J. ACM 45*, 5, 798–842.

ABITEBOUL, S., LAUSEN, G., UPHOFF, H., AND WALLER, E. 1993. Methods and rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C.). ACM, New York, pp. 32–41.

BAL, R. AND BALSTERS, H. 1993. A deductive and typed object-oriented language. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, S. Ceri, K. Tanaka, and S. Tsur, Eds. (Phoenix, Az.). Lecture Notes in Computer Science, vol. 760. Springer-Verlag, New York, pp. 340–359.

BARJA, M. L., FERNANDES, A. A. A., PATON, N. W., WILLIAMS, M. H., DINN, A., AND ABDELMOTY, A. I. 1995. Design and implementation of rock & roll: A deductive object-oriented database system. *Inf. Syst. 20*, 3, 185–211.

BUGLIESI, M. AND JAMIL, H. M. 1994. A logic for encapsulation in object oriented languages. In *Proceedings of International Symposium on Programming Languages, Implementations,* *Logics and Programs* (*PLILP '94*) (Madrid, Spain). Lecture Notes in Computer Science, vol. 844. Springer-Verlag New York, pp. 213–229.

BUTTERWORTH, P., OTIS, A., AND STEIN, J. 1991. The gemstone object database management system. *Commun. ACM 34*, 10, 64–77.

CACACE, F., CERI, S., CREPI-REGHIZZI, S., TANCA, L., AND ZICARI, R. 1990. Integrating object-oriented data modelling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Atlantic City, N.J.). ACM, New York, pp. 225–236.

CATTELL, R. G. G., Ed. 1996. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan-Kaufmann, Los Altos, Calif.

CATTELL, R. G. G. AND BARRY, D., Eds. 1997. *The Object Database Standard: ODMG 2.0*. Morgan-Kaufmann, Los Altos, Calif.

CHEN, W., AND WARREN, D. 1989. C-Logic for complex objects. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Philadelphia, Pa.). ACM, New York, pp. 369–378.

DEUX, O. ET AL. 1990. The Story of O_2. *IEEE Trans. Knowl. Data Eng. 2*, 1, 91–108.

DEUX, O. ET AL. 1991. The O_2 System. *Commun. ACM 34*, 10, 35–48.

DOBBIE, G. AND TOPOR, R. 1995. On the declarative and procedural semantics of deductive object-oriented systems. *J. Intel. Inf. Syst. 4*, 2, 193–219.

FISHMAN, D. H., BEECH, B., CATE, H. P., CHOW, E. C., CONNORS, T., DAVIS, J. W., DERRETT, N., HOCH, C. G., KENT, W., LYNGBAEK, P., MAHBOD, B., NEIMAT, M. A., RYAN, T. A., AND SHAN, M. C. 1987. Iris: An object-oriented database management system. *ACM Trans. Office Inf. Syst. 5*, 1, 48–69.

GELDER, A. V., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38*, 3, 620–650.

GRECO, S., LEONE, N., AND RULLO, P. 1992. Complex: An object-oriented logic programming system. *IEEE Trans. Knowl. Data Eng. 4*, 4, 344–359.

GUERRINI, G., BERTINO, E., AND BAL, R. 1998. A formal definition of the chimera object-oriented data model. *J. Int. Inf. Syst. 11*, 1, 5–40.

HEUER, A. AND SANDER, P. 1993. The living in a lattice rule language. *Data Knowl. Eng. 9*, 4, 249–286.

JAMIL, H. M. 1997. Implementing abstract objects with inheritance in datalog^{neg}. In *Proceedings of the International Conference on Very Large Data Bases* (Athens, Greece). Morgan-Kaufmann, Los Altos, Calif., pp. 46–65.

JAMIL, M. H. AND LAKSHMANAN, L. V. S. 1992. Orlog: A logic for semantic object-oriented models. In *Proceedings of the 1st International Conference on Information and Knowledge Management* (Baltimore, Md.). ACM New York, pp. 584–592.

KIFER, M., LAUSEN, G., AND WU, J. 1995. Logical foundations of object-oriented and frame-based languages. *J. ACM 42*, 4, 741–843.

KIFER, M. AND WU, J. 1993. A logic for programming with complex objects. *J. Comput. Syst. Sci. 47*, 1, 77–120.

KIM, W. 1990. *Introduction to Object-Oriented Databases*. The MIT Press, Cambridge, Mass.

LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The objectStore system. *Commun. ACM 34*, 10, 50–63.

LING, T. W. AND LEE, W. B. T. 1998. DO2: A deductive object-oriented database system. In *Proceedings of the 9th International Conference on Database and Expert System Applications* (*DEXA '98*) (Vienna, Austria). Lecture Notes in Computer Science, vol. 1460. Springer-Verlag, New York, pp. 50–59.

LING, T. W. AND TEO, P. K. 1995. Object migration in ISA hierarchies. In *Proceedings of the International Conference on Database Systems for Advanced Applications* (*DASFAA '95*). World Scientific Press, Singapore, pp. 216–225.

LIU, M. 1996. ROL: A deductive object base language. *Inf. Syst. 21*, 5, 431–457.

LIU, M. 1998a. 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*) (Vienna, Austria). Lecture Notes in Computer Science, vol. 1460. Springer-Verlag, New York, pp. 892–902.

LIU, M. 1998b. Relationlog: A typed extension to Datalog with sets and tuples. *J. Logic Prog. 36*, 3, 271–299.

LIU, M. 1999. Overview of the ROL2 deductive object-oriented database system. In *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages & Systems* (*TOOLS USA '99*) (Santa Barbara, Cailf.). IEEE Computer Society Press, Los Alamitos, Cailf., pp. 63–72.

LIU, M. AND GUO, M. 1998. ROL2: A real deductive object-oriented database language. In *Proceedings of the 17th International Conference on Conceptual Modeling* (*ER '98*) (Singapore). Lecture Notes in Computer Science, vol. 1507. Springer-Verlag, New York, pp. 302–315.

LOU, Y. AND OZSOYOGLU, M. 1991. LLO: A deductive language with methods and method inheritance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Denver, Colo.). ACM, New York, pp. 198–207.

MAIER, D. 1986. A logic for objects. Tech. Rep. CS/E-86-012. Oregon Graduate Institute, Beaverton, Ore.

MUMICK, I. S. AND ROSS, K. A. 1993. Noodle: A language for declarative querying in an object-oriented database. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases* (Phoenix, Az.). S. Ceri, K. Tanaka, and S. Tsur, Eds. Lecture Notes in Computer Science, vol. 760. Springer-Verlag, New York, pp. 360–378.

SOLOVIEV, V. 1992. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, O2. *SIGMOD Rec. 21*, 1, 93–104.

SRIVASTAVA, D., RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. 1993. CORAL++: Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Data Bases* (Dublin, Ireland). Morgan-Kaufmann, Los Altos, Calif., pp. 158–170.

Received August 2001; revised January 2002; accepted February 2002