ELISA BERTINO, GIOVANNA GUERRINI AND LUCA RUSCA

OBJECT EVOLUTION IN OBJECT DATABASES

ABSTRACT: Application environments that object-oriented database management systems support are characterized by a highly evolving nature. Two different forms of evolution can be distinguished for object-oriented databases: evolution of schema and evolution of instances. This paper deals with evolution of instances in the context of the Chimera object-oriented deductive data model. In particular, problems related to object migration, dynamic object classification and multiple class direct membership are discussed.

1 INTRODUCTION

There are many aspects related to evolution in object-oriented databases. Not all of them have been investigated in sufficient depth. Generally speaking, one can distinguish between evolution of schemas - for example, modifying a class definition - and of instances - for example, the migration of an instance from one class to another. In the latter kind of evolution, an instance modifies its own structure while maintaining the same identity. In this paper we discuss instance evolution in the context of the Chimera object-oriented data model [11, 18]. However, though developed with reference to the Chimera data model, the discussion is applicable to any object-oriented database system. Chimera[1] is an object-oriented, deductive, active data model developed as part of ESPRIT Project Idea P6333. Chimera provides all concepts commonly ascribed to object-oriented data models, such as: object identity, complex objects and user-defined operations, classes, inheritance; it provides capabilities for defining deductive rules, that can be used to define views and integrity constraints, to formulate queries, to specify methods to compute derived information; it supports a powerful language for defining triggers.

The object-oriented model introduces different kinds of evolution for objects. In addition to modifications to the values of an object's attributes (state evolution), other kinds of evolution are possible, by which individual objects can modify their own structure and behavior, while

[1] A Chimera is a monster of Greek mythology with a lion's head, a goat's body, and a serpent's tail; each of them represents one of the three components of the language.

maintaining their own identity constant. Whereas state evolution finds its relational counterpart in modifications to the values of the attributes of a tuple, other kinds of evolution exist which are specific to object systems. In particular, the structure and/or behavior of an object can be modified because of:

- migrations of the object to different classes;
- dynamic addition of classes, even not related by inheritance, to the object, thus leading to multiple class direct membership[2];
- specialization of the object, leading to exceptional instances.

It should be noted that the migration of an object to a new class is different from adding a new class to the object. In the first case, the class to which the instance belonged is lost, whereas in the second case it is not. In the latter of these two options, an object must be able to be a direct member of several classes at the same time. The specialization of an object allows an object to have additional features (attributes and methods) in addition to those of the classes it belongs to. The specialized object is also called an exceptional instance.

These kinds of evolution are not as yet supported by many systems, as they introduce problems for both implementation and consistency. Chimera supports both object migration and dynamic addition of classes, leading to multiple class direct membership. By contrast, Chimera does not support exceptional instances, which is a peculiarity of the O_2 object-oriented database system [14].

Moreover, Chimera supports derived (or predicative) classes. That is, classes whose extents are not explicitly manipulated; rather those classes are implicitly populated in that a population predicate is associated with the class specifying sufficient and necessary conditions for an object to belong to the extent of the class. In models supporting derived classes, it is difficult to ensure that an object belongs to a unique most specific class, because it depends on the population predicates being disjoint. In such a situation, when a new object is inserted in the database, the object may be classified as instance of several most specific classes and the user may even not realize this fact. Note that also modifications to the values

[2]An object belonging to a class C is a *direct member* of C if it does not belong to any subclass of C. An object is a *member* of a class C if it is a direct member of C or is a direct member of some subclass of C.

of an object's attributes may result in the addition of one or more classes to the object or in the removal of the object from the extent of one or more classes.

Another important aspect concerning instance evolution is that when an object is able to migrate to different classes, or to dynamically acquire and loose classes, appropriate constraints must be imposed to ensure that correct evolutions are defined. Semantically meaningful migrations depend from the application domain. One option is to specify special integrity constraints [36]. Such constraints include:

- specifying a class as *essential*

  a class C is essential if an object which is a member of C cannot at a subsequent point in time migrate to another class and stop belonging to the set of members of C. This means that migrations of an object which is a member of C are confined to the inheritance hierarchy having C as root. Note that an object can have several essential classes, if the model has multiple inheritance.

- specifying a class as *exclusive*

  a class is exclusive if an object that belongs to this class as a direct member cannot belong at the same time to other classes. This constraint can be refined by introducing the notion of exclusiveness of one class with respect to another.

The fact that a class is essential does not imply it is exclusive. An essential class C can be added to an object O, even if the object already has essential classes. The only constraint is that O cannot later loose class C. Conversely, an object can loose an exclusive class.

In this paper we elaborate on all the aspects concerning object evolution in Chimera, presenting also a survey of instance evolution capabilities provided by other systems and discussing relevant open research issues. The paper is organized as follows. The remainder of this section introduces an example that will be used to illustrate the various kinds of object evolution throughout the paper. Section 2 discusses the main issues related to object migration, whereas Section 3 deals with implicit object migration, that is, with state-based dynamic object classification and derived classes. Section 4 is devoted to multiple class direct membership, while Section 5 presents some additional examples. Finally, Section 6 surveys the forms of instance evolution supported in other object-oriented systems, and Section 7 concludes the paper.

1.1 An Example

Figure 1 illustrates a portion of a database schema handling data related to teams in a national football championship. The graphic representation is similar to that used in [7]. Each class is represented by a rectangle, divided in a number of slots, representing the attributes and the methods of the class. Attribute names are in plain text, while method names are in italics. With each rectangle a name is moreover associated (bold), representing the name of the class. Two kinds of arc are used: plain arcs represent aggregation relationships between classes, whereas bold arcs represent inheritance relationships. Thus, a plain arc from an attribute a of a class C to a class C' denotes that C' is the domain of attribute a in class C. By contrast, a bold arc from class C to class C' denotes that C is a subclass of (that is, inherits from) C'.

A team is characterized by a name, which identifies the team, and by a division, in which the team plays (teams are organized according to a certain number of divisions, e.g. from first division to fifth division, each corresponding to a different championship). A group can moreover be associated with a team. Indeed, divisions can be organized in different groups. In general the first division consists of a single group, whereas lower divisions are organized around different groups (e.g. corresponding to different regional areas of the country). For each team its current score is recorded.

Teams are partitioned in professional and non-professional teams. For each professional team, the capital and the fiscal registration number are recorded, whereas for non-professional teams it is recorded whether or not the team has an associated under-20 team (it is supposed that each professional team has one). A subclass `First Position Team` of the `Team` class is also defined in the schema. This subclass contains the teams that are leading their championship (e.g. their group or their division). With each first position team, the number of matches from which it is leading the championship is associated.

Finally, the schema includes class `Professional Player`. Each professional player is characterized by a name, a role (e.g., goalkeeper, defender, mildfielder, forward) and a salary. Moreover, a professional player plays in a professional team, thus class `Professional Player` has an attribute `Plays_in` with domain `Professional Team`.

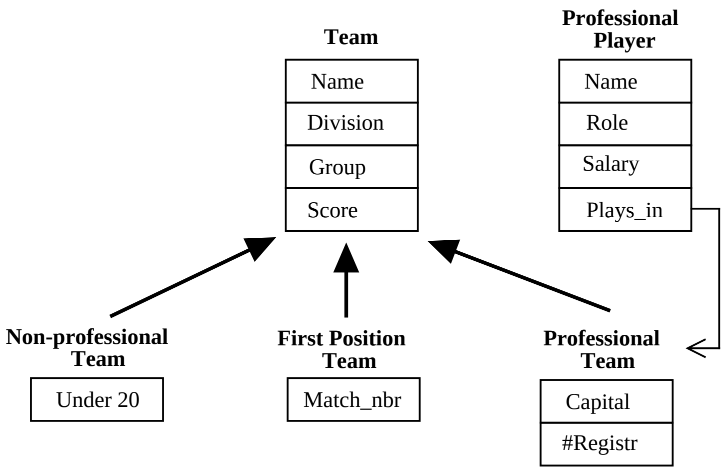

Figure 1. Database schema of our *Teams* example

2 OBJECT MIGRATION

In this section, we first discuss issues and approaches to support object migration (Subsection 2.1) and then we focus on the approach adopted in the Chimera data model (Subsection 2.2).

2.1 Issues and Approaches to Object Migration

Migration allows an object to become a direct member of a class which is different from the class from which the object has been created. Migration represents an important functionality for object evolution. In particular, migration allows an object to modify its features, attributes and methods, while retaining its identity.

Referring to our *Teams* example, the change of the status of a team from non-professional to professional is a meaningful evolution. However, despite such a change, teams maintain their identity, and their time-invariant properties, such as the name. In particular, if we restrict the teams playing in divisions one, two, and three to be professional teams, and teams in lower divisions to be non-professional teams, then, at the end of the championship, the teams which are promoted from the fourth to the third division become professional teams, whereas the teams that move from the third to the fourth division become non-professional teams.

This kind of evolution is not supported by many systems, because of implementation and consistency problems. Consistency problems, in particular, arise when an object O, which is a member of a class C, is referred by an object O' as the value of an attribute A, whose domain is C; in such case, the migration of O to a superclass of C violates the integrity constraint established by specifying the domain for A. In other words, object O', after the migration of O, will have as value for attribute A an object which is not a member (neither direct nor indirect) of the class domain of A. Referring to our *Teams* example, the migration of a professional team to the `Team` class causes that team to no longer be a legal value for attribute `Plays_in` of a professional player. The situation is similar to the one where the explicit deletion of a referred object is requested. Migration upward in the class hierarchy, indeed, can be seen as a "partial deletion" of the object. Because of those problems, in systems supporting migration, objects are restricted to migrate only into subclasses of the class to which they belong. Here, objects, in a sense, also keep the previous class to which they belonged, since they remain members of this class, despite becoming direct members of a new class.

Before discussing possible solutions for the consistency problems caused by migrations, let us recall that in object-oriented database systems there are two basic deletion policies. Under the first one, referred to as *explicit deletion view*, an object deletion statement is made available at user level[3]. By using such a command, object deletions are explicitly requested by users. Under the second policy, referred to as *garbage collection view*, users can only delete references from an object to another one. An object is then deleted (by the system) when it is no longer referred by any other object. In object systems with explicit deletion, there is the problem of dangling pointers, due to objects containing references to deleted objects. That problem is similar to the referential integrity problem arising in any data model with an explicit deletion operation. Several options to solve that problem are available in SQL [9] and have been revisited in an object-oriented context [5], ranging from forbidding the deletion of the referred object, to propagating (that is, cascading) the deletion to the referencing object, and to setting the reference to null. To avoid dangling references, Zdonik [36] proposes to keep a *tombstone ob-*

[3]Here and in what follows, the term user should be intended in the broader meaning of an actual user or an application.

ject in place of the deleted object. This solution overcomes the problem of dangling references, since each reference is either to the original object, or to its tombstone object. A main problem of this solution is that each method and query following references from an object to other ones must handle the case in which the referred objects have been deleted. Method code becomes more complicated, because very simple expressions, like the one denoting the value of an object attribute, must handle the exceptions generated by the fact that the object no longer exists (and its tombstone is found, instead).

The problem of upward migrations is similar. If an object O migrates from a class C to a class C', with C subclass of C', an object can exist with a reference to O as an object of class C. There are two different approaches to the migration of an object:

- *Global Type Modification*

  The class modification is performed directly on the object and causes a change in the object state, namely, the deletion of specific attributes defined in the classes from which the object has migrated. If there are other objects referring to object O as a member of C, they must be notified that O is no longer a member of C.

  The problem is similar to that of deletion discussed above and a similar approach can be used. A tombstone can for example be placed in the object to denote that this object used to be a member of class C, but that now the attributes related to C have been deleted. Whenever a method, or a query, tries to access the object's attributes specific to C, it must be prepared to receive a message (exception) denoting that those attributes are no longer available.

- *Local Type Modification*

  Under this approach, the migration operation does not modify the state of the object, rather it creates another *view* of the object. More specifically, upward migration does not delete the information related to the class from which the object migrates. Rather, it creates another view of the object. This view has as type the class to which the object has migrated. Thus, two different references to the object exist, with different types; the reference corresponding to the current object is the most general.

In a similar way, when dealing with downward migration, the attributes specific to the class to which the object has migrated are not added to the object state. Rather, a new view of the object is created. This view, having as type the lower class, has the additional attributes of the lower class, and shares the state with the original object. Under this approach, the portions of the object state that are no longer referred can be garbage collected.

2.2 Object Migration in Chimera

Object migration in Chimera can be *explicit*, through the invocation of migration commands by users, or *implicit*. Implicit migration arises because Chimera supports predicate classes. A predicate class is a class whose extent is implicitly specified by some predicates, called *population predicates*. All instances of a class C that verify the population predicate of some subclass of C are automatically migrated to this subclass. Because population predicates state conditions against object attribute values, changes to these values trigger automatic migrations. We discuss explicit migrations in the remainder of this subsection, whereas implicit migration is discussed in the following sections.

Chimera provides two operations, supporting upward and downward migrations, respectively. If an existing object in a given class is to be inserted into a more specific subclass or, inversely, moved back to a more general superclass, the OID of the object does not change. Only those attributes that exclusively belong to the more specific class have to be added or removed from the object state. As an example, referring to our *Teams* database, when a team becomes the first in its championship, it is specialized to class `First Position Team`, and a value for attribute `Match_nbr` must be provided. By contrast, when the team looses its leadership and it is generalized back to the `Team` class, the value for attribute `Match_nbr` is removed from the state of the object.

Specialization of an object to a subclass is performed by the `specialize` operation, which takes as input parameters two class names, C_1 and C_2, an object identifier O and a record term T. The result of the operation is to insert object O, initially belonging to class C_1, into class C_2 as well. Moreover, the state of O is extended by concatenating its old state (containing values for those attributes that are now inherited) with T, where T specifies values for those attributes that are specific to C_2. Note that O remains a member of C_1 due to the subclass relationship between

C_1 and C_2.

The inverse process is performed by the `generalize` operation, which takes only three parameters: two class names C_1 and C_2, and an identifier O. The result of the operation is to remove object O from class C_1, and to make it a direct member of the superclass C_2 of C_1, which O used to be a member of. Therefore, all attributes specific to C_1 are dropped. Thus, Chimera supports global type modifications. Referential integrity is enforced in Chimera as follows: whenever an object O is deleted from class C[4], the OID of the deleted object is dropped from all attribute values which refer to O in other objects O'; therefore:

- if C is the type of an atomic attribute of O' (either defined individually or a record component), its value is set to null;
- if C is the type of the element of an attribute of O' built by means of set or list constructors, then O is deleted from the set or list; this may result in producing an empty set or list.

Referring to our *Teams* example, the change of status from non-professional to professional team is performed by the following operations. First, the non-professional team is generalized to class `Team` (loosing attribute `Under_20`) through a `generalize` command, then the team is specialized to class `Professional Team` (specifying a value for attributes `Capital` and `#Registr`) through a `specialize` command.

Chimera supports multiple inheritance. However the constraint is imposed that for multiple inheritance a common ancestor must exist. Therefore a class C can be defined as a subclass of classes C_1 and C_2 only if a class C' exists from which both C_1 and C_2 inherit from. In Chimera the existence of a common root of the entire class hierarchy is not imposed. Rather the hierarchy is partitioned into multiple strongly connected components. Each strongly connected component is characterized by a single node without incoming edges: this node is called *root* of the strongly connected component. Thus, a class can inherit from multiple classes only if the classes belong to the same strongly connected component, that is, if they have a common ancestor. Therefore, we may think of the set of all classes as partitioned in m distinct hierarchies $\mathcal{H}_1, \ldots, \mathcal{H}_m$, corresponding to the m strongly connected components of the class hierarchy. In Chimera, moreover, an object cannot migrate over different hierarchies. This is reflected by the fact that the only migration

[4]Note that "deleted from class C" also means "generalized to a superclass of C".

primitives supported are generalization to a superclass and specialization to a subclass. Thus, in Chimera, the root of the hierarchy to which the object belong is an *essential* type, with the meaning discussed earlier (see Section 1).

3 OBJECT CLASSIFICATION AND PREDICATE CLASSES

A predicate (or derived) class has all the properties of a usual class, including a name, a set of superclasses, a set of attributes and a set of methods. In addition, a predicate class has a population predicate. A predicate class represents the subset of the members of its superclass(es) that also satisfy the predicate. Whenever an object is a member of the superclasses of the predicate class, and the population predicate evaluates to true on the object, the object is automatically considered a member of the predicate class. An object member of a predicate class has all the attributes and methods of this class. If the object state later changes and the population predicate no longer evaluates to true, the object is excluded from the predicate class. The population predicate can test the value or the state of an object, thus supporting a form of implicit classification based on attribute values in addition to explicit classification based on types supported by traditional classes. Predicate classes support indeed a form of automatic, dynamic classification of objects, based on their run-time value, state, or other user-defined properties. In traditional object-oriented models various kinds of static type-based classifications of objects using classes and inheritance are supported; by contrast, the specialization of an object depending on the value of one of its attributes is not allowed.

Referring to our *Teams* example, class `First Position Team` can be expressed as a derived class. Its population predicate requires, for an object to be member of `First Position Team`, that the object belongs to the `Team` class, and that its score is greater than (or equal to) that of each other team in the same championship (that is, in the same group and division). Thus, an object migration between classes `Team` and `First Position Team` is induced by a simple modification to attribute `Score` of an object (not necessarily the one that migrates). Moreover, if we consider that any team playing in division one, two or three must be a professional team, whereas each team playing in a lower division must be a non-professional team, then also classes `Non-professional Team` and

`Professional Team` can be defined as derived classes, whose population predicates depend on the value of the `Division` attribute.

In models supporting predicate classes, it is difficult to ensure that an object is a direct member of a unique class, since it depends on population predicates being disjoint. Thus, derived classes lead to the need of supporting a form of multiple class direct membership. The following section discusses how multiple class direct membership is supported in Chimera.

Predicate classes are also the base of *views* in object-oriented data models. View mechanisms for object-oriented databases based on derived classes are presented in [22, 26, 27, 29, 30]. In [29] multiple class direct membership is simulated by *surrogate objects*, that is, each view instance has a special attribute whose value is the identifier of its base object. In [26] the simulation is based on the *object-slicing* approach: the storage structure of a class (or view) object is dispersed through a hierarchy of implementation objects linked to a conceptual object which is a dictionary storing associations of implementation object identifiers and their respective classes.

A form of predicate classes is supported by object-oriented languages with classification facilities [35]. In those languages two kinds of class-like constructs are introduced: primitive concepts, used for explicit classification of objects, and defined concepts, used for implicit property-based classification. An object is member of a primitive concept only when explicitly stated, whereas an object is member of a defined concept whenever its attributes satisfy certain restrictions. Only a few kinds of restrictions are allowed, such as checking for an attribute being an instance of a particular class, being within an integer range, or being an element of some fixed set. In return, the system automatically computes subsumption relationships among concepts (i.e., when a concept "inherits" from another). An object in Yelland's system may be a member of several independently defined concepts. The system creates internal combination subclasses, and uses a single combination subclass to record that an object is a member of several independent concepts simultaneously.

When methods are associated with predicate classes, method dispatching depends not only on the dynamic type of an argument, but also on its dynamic value or state. Among languages supporting predicate classes, Cecil [12] is the only one for which a dispatching mechanism has been developed. Cecil is based on multiple-dispatching. In Cecil, methods are defined by specifying a name, the formal parameters and an implementation. Each formal parameter can, optionally, be associated with an *argument specializer* $\mathtt{obj}_i$, specifying that the method is defined only if the actual parameters (that is, the message arguments) are descendants of object $\mathtt{obj}_i$[5]. For non-specified formal parameters (that is, without argument specializer), any value is legal as actual parameter. Argument specializers are the mean for associating the (multi-)method with specialized objects.

In Cecil methods are dispatched as follows. First, methods applicable to the message, that is, methods with the same name and number of arguments of the message, and whose argument specializers are ancestors of corresponding actual parameters, are determined. Applicable methods are then ordered by their specificity: a method $\mathtt{m}_1$ is more specific than a method $\mathtt{m}_2$ if any argument specializer of $\mathtt{m}_1$ is a descendant of the corresponding argument specializer of $\mathtt{m}_2$ and at least one of the argument specializers of $\mathtt{m}_1$ is a proper descendant of (that is, different from) the corresponding argument specializer of $\mathtt{m}_2$. If a unique most specific applicable method does not exist, an "ambiguous message" error is generated and the message is not dispatched. Cecil does not make use of any ordering on objects or on arguments to solve ambiguities in an automatic way.

Another aspect that must be carefully handled in a system supporting derived classes is type checking. Predicate classes require a new kind of type checking taking into account that the interface exported by an object depends on the current state of the object, as shown by the following example. Referring to the *Teams* example, consider a variable `X` declared of type `First Position Team`. If the value of `X.Score` of the object referenced by `X` is modified, the object may not any longer verify the predicate of class `First Position Team`. As a consequence, variable `X` would reference an object which not consistent with the type of `X`. To avoid such problem, different solutions can be adopted. Two of them, namely

- disallowing a variable to be declared with a type corresponding to a derived class;

- disallowing updates on the attributes appearing in the population predicate

[5]Cecil does not support the notion of class, thus inheritance relationships are specified at the object level.

are conservative solutions. If the population predicate can only be falsified by updates on the object on which it is evaluated[6] that solutions can be refined by allowing a variable to be declared with a type corresponding to a derived class but disallowing updates on the attributes appearing in the population predicate to be applied to that variable. Such an approach prevents, by static checks, a variable of type T from referencing at run-time an object which is not an instance of the class corresponding to T. This approach emphasizes the type checking view.

An alternative solution is to regard population constraints as other constraints and thus to check them run-time. This approach does not ensure that a variable of type T, with T corresponding to a derived class, always references a member of the class corresponding to T. Rather a check is performed at run-time to detect whether the variable references an object that meets the population constraint. If not, an error is raised. This approach requires some type checking at run-time and thus it is potentially less efficient.

Note however, that the two above solutions are not mutually exclusive. They can be combined to obtain a good compromise between semantic richness and efficiency. For example, a variable can be allowed to be declared of a type corresponding to a derived class, and updates on the attributes appearing in the population predicate can also be applied to that variable, but run-time checks for that variable (and only for that one) must be performed. More sophisticated solutions, based on flow analysis of application code [13], can also be investigated.

In Cecil, the relationships among predicate classes are specified explicitly by the programmer through inheritance declarations and `disjoint` and `cover` declarations. These declarations are used in type checking. In Cecil, if two predicate classes might both be acquired by an object, either one must be known to be more specific than the other, or they must have disjoint method names. In other words, the checker needs to know: when one predicate class *implies* another, when two predicate classes are *mutually exclusive*, and when a group of predicate classes is *exhaustive*. Since in Cecil population predicates can contain arbitrary user-defined code, the system is not able to infer implication, mutual exclusion and exhaustiveness by examining the population predicates associated with the various predicate classes. Consequently, it must rely on explicit user declarations to determine the relationships among predic-

[6] This is not always true, e.g. if the population predicate makes use of aggregate operators.

ate classes. The system dynamically verifies that these declarations are correct. To state that one population predicate implies another, the `isa` declaration is used. Mutual exclusion among a group of classes can be declared through the `disjoint` specification. This specification has the effect of stating that the predicate classes will never have simultaneous common members, that is, at most one of their population predicates will evaluate to true at any given time. Finally, the `cover` declaration asserts that a group of predicate classes exhaustively covers the possible states of some other class.

4 MULTIPLE CLASS DIRECT MEMBERSHIP

As we have seen in the previous section, when state-based dynamic object classification is supported, an object can be classified into different classes, even not related by the inheritance hierarchy. Thus, a modification of an object attribute may result in the dynamic addition of classes to the object, leading to multiple class direct membership. Referring to our *Teams* example, members of class `Team` can be classified along orthogonal dimensions, such as `Non-professional`, `Professional`, `First Position Team`. According to the intuitive semantics, a team can be both a professional team and a first position team at the same time. Thus, the object representing this team is classified both in class `Professional Team` and in class `First Position Team` and it does not have a unique most specific class, rather it has a set of most specific classes.

Although the above situation can be easily represented in a model with multiple inheritance by defining a subclass (say `First_Position_Professional_Team`) of all the involved classes, this solution may lead to a lot of artificial subclasses, sometimes referred to as *intersection classes* [28]. Referring to the hierarchy above, the meaningful subclasses of the `Team` class are shown in Figure 2. Thus, this approach can lead to a combinatorial explosion of sparsely populated classes, whose sole purpose is to allow an instance to have multiple most specific classes, without adding new state or behavior. Another problem with the multiple inheritance approach is that it only provides a single behavioral context for an object [25]. Name conflicts among features in the superclasses are solved once for ever in the subclass definition (for example by imposing an order on superclasses, or with an explicit qualification mechanism) and the selected feature is the only one always considered whatever the context of the

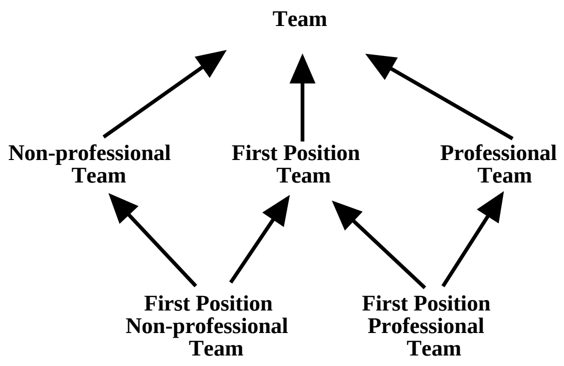
Team
Non-professional Team
First Position Team
Professional Team
First Position Non-professional Team
First Position Professional Team

Figure 2. Class hierarchy of our *Teams* example, enriched with some meaningful subclasses

object reference is. Thus, reducing multiple class direct membership to multiple inheritance, as proposed by Stein [31] and Chambers [12], does not account for any context dependence nor for object dependent class ordering.

If multiple class direct membership is supported, two classes C_1 and C_2, with a common superclass C, may have a non-empty intersection even when neither C_1 is a subclass of C_2, nor C_2 is a subclass of C_1. However, when objects belong to several most specific classes, conflicts among different definitions may arise. Indeed, if an object has several most specific classes, the object takes the union of the features of all the classes to which it belongs. Such conflicts resemble conflicts due to multiple inheritance. However, conflicts due to multiple inheritance can be detected at compile time whereas objects may become and cease to be an instance of a class at run time, thus the situation is more complicate for multiple class direct membership.

Referring to our *Teams* database, a team can be at the same time both a professional team and a first position team. The team behaves differently according to the different contexts from which it is accessed, e.g. if the context from which the team object is accessed is `Professional Team`, the attribute `Match_Nbr` of the object is not visible.

Some restrictions are imposed on multiple class direct membership in Chimera. An object can belong to several most specific classes only if the classes belong to the same strongly connected component, that is, if they have a common ancestor. The sets of OIDs in different strongly connected components are therefore disjoint. A class in a hierarchy $\mathcal{H}_i$ is therefore *exclusive* with respect to all classes in hierarchy $\mathcal{H}_j$, with $i \neq j$.

In the following subsections we discuss how name conflicts can be solved for models supporting multiple class direct membership, and how semantic constraints can be expressed to impose that two classes have no common instances.

4.1 Name Conflicts for Multiple Class Direct Membership

In this section we address the problem of name conflicts due to multiple class direct membership. This topic has been dealt with in [4], while an extensive discussion on dispatching can be found in [6].

An approach to solve conflicts is to impose that each object, though having several most specific classes, has a single *preferred* class. The binding between an object and its preferred class can be either *fixed* or *context-dependent.* A fixed binding only depends on the set of most specific classes of the object[7]. By contrast, a context-dependent binding also depends on the expression in which the object reference is contained. In a fixed preferred class approach, the preferred class can be determined by imposing a total ordering on classes, or by allowing each object to specify an ordering on the classes to which it belongs[8], or finally by specifying a reference class for each feature in the instance (with an explicit qualification mechanism). Context-dependent preferred class approach leads to a more more flexible language and models both context-dependent access restrictions and context-specific behavior.

In our approach, the context-dependent preferred class is determined by the static type of the object in the expression containing the object reference. Each object reference in each Chimera expression is assigned a single static type. The notion of context of an object reference can be characterized in terms of static types. We analyze conflicts arising in attribute accesses and method invocations. As far as attribute access is concerned, we can disambiguate each access by taking into account only the context of the object reference. By contrast, when considering

[7]A fixed binding does not mean that the binding is immutable for the object lifetime, because an object may acquire and loose classes dynamically.

[8]A reasonable ordering could be the one determined by the acquisition order of classes, in such a way that the most recently acquired behavior prevails (as in Fibonacci [3]).

method dispatching, if we want to ensure a notion of *most specific behavior*, the context alone is not enough to properly dispatch the method. Thus, we propose and compare two different dispatching approaches: the first approach ensures context-dependent behavior, the other one ensures behavior identity.

Consider first the structural component of objects. For an object with multiple most specific classes, the state of the object, that is, the attributes of the object, and the proper domains for these attributes, must be determined. Roughly speaking, the state of an object belonging to several most specific classes is the union of all the attributes defined in these classes. However, the sets of attributes in those classes may not be disjoint. Thus, *name conflicts* may arise. To handle conflicts, we introduce the notion of *source* of an attribute. Intuitively, if an attribute belongs to the intersection of the attribute sets of two classes and it has in both classes the same source, that is, it is inherited by a common superclass, then the attribute is semantically unique, and thus the object must have a unique value for this attribute. If, by contrast, the attribute has different sources, the two attributes in the two classes have accidentally the same name, but represent different information, that must be kept separated. Thus, the object may have two different values for the two attributes (a renaming policy is applied).

Consider now the behavioral component of objects, that is, its methods. Each class in a type hierarchy may define a different implementation for the same method. For each method invocation on an object, an implementation must be chosen among the most specific ones. Note that different implementations may return different results or may perform different updates on data. According to one of the basic principles of object-orientation, when, because of subtype polymorphism [10], several method implementations are applicable to a method invocation, the implementation specified in the most specific class of the invocation receiver is executed, as it is the one that *most closely matches* the invocation. Thus, the most specialized behavior prevails, according to the classical late binding mechanism. However, in a model where an object is not characterized by a single most specific class, the choice of the method implementation that "most closely matches" the invocation is not obvious.

There are two different approaches to determine the implementation which most closely matches the invocation, among different implementations in different most specific classes of the object. The first approach, which we call **preferred class** approach, is based on the idea that each object has in each context a *preferred* class, among its most specific ones. Thus, each method invocation is dispatched choosing the implementation in the preferred class in the current context. This approach supports a context-dependent behavior, as the same method invocation may be dispatched differently, and thus may return different results and perform different updates, depending on the context where the method is invoked. The second approach, which we call **argument specificity** approach, does not determine the preferred class of an object to dispatch a method invocation, rather it makes use of the other actual arguments of the method call, thus considering the method as a multi-method [16].

In the following subsections we illustrate and compare these two approaches.

Preferred Class Dispatching Approach

According to this approach, a method invocation is dispatched by taking into account the context-dependent preferred class of the receiver object. As we have seen, in each Chimera expression, each object reference has a single static type. However static typing alone is not enough to select a preferred class for each object in each expression for method dispatching. Indeed, the static type of the object may not belong to the set of most specific classes of the object. Referring to our *Teams* example, consider an expression where an object reference has the static type `Team`; if at run-time the reference denotes an object belonging to both the class `Professional Team` and the class `First Position Team`, the context of the object reference does not help in choosing the preferred class.

In those cases, we must use a total order on classes. This order can be determined by the definition order of classes, eventually overridden by `before/after` clauses in class definitions. Alternatively, we may consider for each object a total order on its most specific classes, as the one determined by the acquisition order, in such a way that the most recently acquired class precedes the others in the order. We remark that such a total order, that may be considered too arbitrary and unpredictable by the user, is taken into account *only* when the context does not uniquely determine a preferred class for the object. The only alternative in these cases, apart from using that order, would be to simply not dispatch the message, because it is *ambiguous*.

The preferred class approach is based on both static and dynamic information. The static information consists of the static type of the expression, whereas the dynamic information consists of the set of the most specific classes of the object (such classes, in fact, can only be determined at run-time). The total order on classes can be fixed and thus known statically or can, by contrast, be object-dependent and thus known only at run-time.

The preferred class dispatching approach can be stated as follows.

> Let $o_1.\mathtt{m}(o_2, \ldots, o_n)$ be a method invocation. The method invocation is dispatched as follows:
>
> method `m` in class C is executed if C is the minimum, under the considered total order on classes[9], of the set of classes containing a definition for method `m` that are subclasses of the static type of object o_1.

The preferred class dispatching approach models context-dependent behavior. In particular, a given method invocation with a fixed set of parameters may produce different results (both in terms of results and data updates), though executed on the same database state, depending on the context of the receiver object reference in the expression containing the invocation.

Under the preferred class approach, any type correct method invocation can be dispatched. Moreover, when the preferred class dispatching strategy is used with a contravariant redefinition rule for method arguments, type correctness is ensured [6].

Argument Specificity Dispatching Approach

The second approach we consider does not take into account the preferred class of an object, rather it tries to determine the method implementation that most closely matches the invocation by taking into account the types of (all) the actual parameters of the invocation (in addition to the type of the receiver object). This approach is similar to multiple dispatching or multi-method approaches where the selection of the method to execute depends on the types of all the actual arguments of the invocation. In the preferred class approach only the type of the receiver object determines

[9]Note that the total order must be consistent with the subtype ordering on classes, thus, if C is a minimum with respect to the total order, a most specific behavior for object o_1 is certainly exhibited.

the method to execute and the other arguments only provide the actual values for the method arguments. However, they play no role in method selection. By contrast, in this approach, the method selection is based on the types of *all* arguments, the receiver as well as the other ones.

This approach can be regarded as fully dynamic, as opposed to the other, which is only partially dynamic. Indeed, in this approach dispatching is based only on run-time information, that is, the types of the actual parameters of the invocation. Moreover, whereas the previous approach models context-dependent behavior, the argument specificity approach ensures a notion of behavior identity. In particular, it ensures that a given method invocation, with a fixed set of actual parameters executed on a given database state, returns the same results and produces the same database state, regardless of the expression in which the method invocation is contained. Finally, note that we use multiple dispatching only for choosing an implementation among the ones in sibling classes, and never for choosing an implementation among the ones in a path in a given inheritance hierarchy. Chimera methods, indeed, are not really multi-methods [16] in that they are associated with classes. Thus, the "privileged receiver", though it is not the only one involved in dispatching, has higher priority with respect to other arguments, in that only the implementations in classes that are most specific for the receiver are considered as "candidates" for dispatching. Thus, the dispatching we propose here is not purely multiple in that we maintain a form of "privilege" for the receiver of the method: other arguments are taken into account only to choose among sibling implementations, in the different most specific classes of the receiver object.

To define the argument specificity dispatching rule a notion of *method specificity*, that is, an order on methods must be used. This order is based on the argument specificity (considered in the order from left to right[10]), and, when all the arguments are not comparable under the subtype relationship, on the total order of classes where the methods are defined. Such an order is exploited in choosing the method to be executed, among the applicable ones.

The following rule formalizes the argument specificity dispatching method.

> Let $o_1.\mathtt{m}(o_2, \ldots, o_n)$ be a method invocation, the method invocation is dispatched as follows:

[10]Note that this order corresponds to *argument order precedence* proposed in [2].

method `m` in class C is executed if it is the minimum, with respect to the method specificity order, in the set of methods applicable for the invocation.

The argument specificity approach ensures behavioral identity of a method invocation. According to this dispatching rule, the class to which a given message is dispatched does not depend on the context of the message receiver in the expression containing the invocation. Under the argument specificity approach, moreover, any type correct method invocation can be dispatched. The argument specificity approach, however, does not ensure type correctness [6].

4.2 Semantic Constraints for Multiple Class Direct Membership

In our model, an object can belong to several most specific classes. However, there are some classes that should not reasonably have common members, that is, no object must be member of those classes at the same time. For example, it is not reasonable (according to the usual interpretation) that an object be both a person and a car. In Chimera such kinds of constraints are modeled by partitioning the set of objects into different hierarchies, with disjoint extensions. Thus, an object can be a member of two most specific classes only if the classes belong to the same hierarchy, that is, if they have some "similarities". For example, persons and cars should be modeled by classes in different hierarchies.

Thus, the semantic constraint that an object cannot be an instance of two classes that "have nothing in common" can be modeled by hierarchies with disjoint extents. However, this approach is not sufficient to express all semantic constraints on multiple direct membership. Indeed, it might be reasonable that in the same hierarchy two classes exist that have no semantically meaningful common instances. As an example, classes `Non-professional Team` and `Professional Team` of our example are both subclasses of `Team`, and thus belong to the same hierarchy, but they should not have common instances.

These *exclusivity* constraints (e.g., class `Non-professional Team` is exclusive with respect to class `Professional Team`) can be expressed in Chimera as untargeted constraints, that is, as constraints that are not associated with any specific class[11]. Suppose that an exclusivity

[11]Conceptually, they could also been expressed as constraints targeted to the class

constraint between classes C_1 and C_2 must be expressed. Let C_i be the root of the hierarchy to which both C_1 and C_2 belong (if C_1 and C_2 belong to different hierarchies the extents are automatically disjoint). Then, the exclusivity constraint can be expressed by the following Chimera untargeted constraint (in denial form):

$$not\_excl(X) \leftarrow C_i(X), X\ in\ C_1, X\ in\ C_2.$$

Referring to the *Teams* database schema, an exclusivity constraint between classes `Non-professional Team` and `Professional Team` is expressed by the following rule:

```
improper_team(X) ← Team(X), X in Non-professional Team,
                   X in Professional Team.
```

Thus, any database state such that the extent of class `Non-professional Team` and the extent of class `Professional Team` are not disjoint, would violate the constraint. If the constraint is violated, the violation is reported to the user (together with the OID of the violating object, bound to variable `X`) and the user can decide how to solve it (e.g., by aborting the transaction, by deleting the object, and so on). These exclusivity constraints, like other Chimera constraints, can also be expressed as triggers [11], containing not only the condition that should not be violated but also the repairing action.

5 ADDITIONAL EXAMPLES

In this section we present few additional examples involving some form of object evolution.

5.1 Polygons, Squares, and Rectangles

Consider the database schema in Figure 3. Class `Polygon` has as an attribute `Vertices`, containing the points (pairs of real numbers) representing its vertices. Moreover, it has three methods, one for adding a vertex, one for displaying the polygon, and the last one for computing its

root of the hierarchy to which the two classes belong. However, the system would not be extensible, in that all the exclusivity constraints on the classes should be known at the time the root is defined.

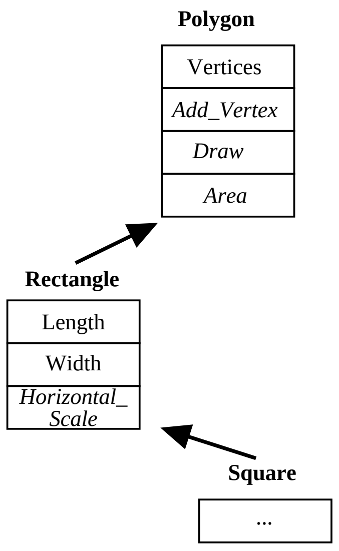

Figure 3. Database schema of our *Polygons* example

area. Subclass `Rectangle` of `Polygon` is a derived subclass. Its population predicate requires that the polygon has four vertices and that both the X and Y coordinates of the vertices are pairwise equal. The class has two derived attributes `length` and `width` (computed from the coordinates of the vertices) and a method `horizontal_scale`, which multiplies the width of the rectangle for a given factor (which is a method argument) and appropriately updates the vertices. Moreover, the class redefines methods `draw` and `area` of `Polygon`. Subclass `Square` of `Rectangle` is also a derived subclass. Its population predicate requires that the length and the width of the rectangle are equal. No matter which new attributes and methods the class it introduces, it redefines methods `draw` and `area` of `Rectangle`.

When a polygon is inserted in the database, it can be classified as a rectangle or as a square, depending on its geometric properties. If, for example, the polygon is a rectangle, a subsequent execution of method `add_vertex` causes the generalization of the object to class `Polygon`, since a polygon with five vertices is no longer a rectangle. Attributes length and width are discarded. By contrast, the execution of method `add_vertex`

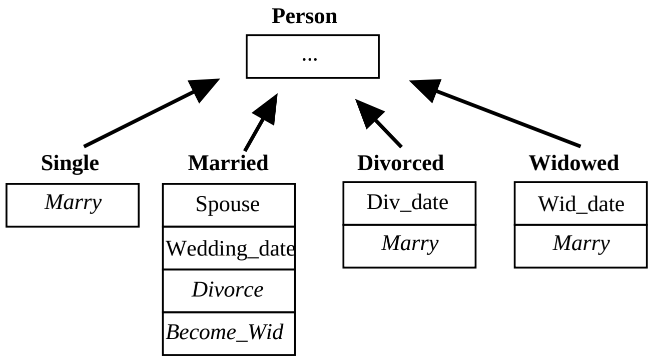

Figure 4. Database schema of our *Marital Status* example

on a right-angled triangle may cause the specialization of the triangle to class `Rectangle`, and the addition of attributes length and width. Similarly, the execution of method `horizontal_scale` on a rectangle may cause its specialization to class `Square`, whereas the execution of method `horizontal_scale` on a `Square` may cause its generalization to class `Rectangle`.

5.2 Marital Status

Consider the hierarchy in Figure 4, representing the partitioning of persons with respect to their marital status. Class `Person` has four different, disjoint, subclasses: `Single`, `Married`, `Divorced`, `Widowed`. For married persons the spouse and the wedding date are recorded, while for divorced and widowed persons the date in which they gain that status is kept. Obviously, a person can change its marital status during his/her life, thus object migration from a class to another is possible. Not all the migrations, however, are meaningful. Indeed, once married, a person can never return to the single marital status. Semantically meaningful migrations are those corresponding to methods attached to classes in the schema. Thus, the need of expressing dynamic constraints on object migrations may arise. Note that by using the constraints proposed in [36] we could only state that `Person` is an essential class, but no other restriction could be imposed. To express these kinds of restrictions, dynamic constraints on object migrations, like those discussed by Su in [32] and by Wieringa

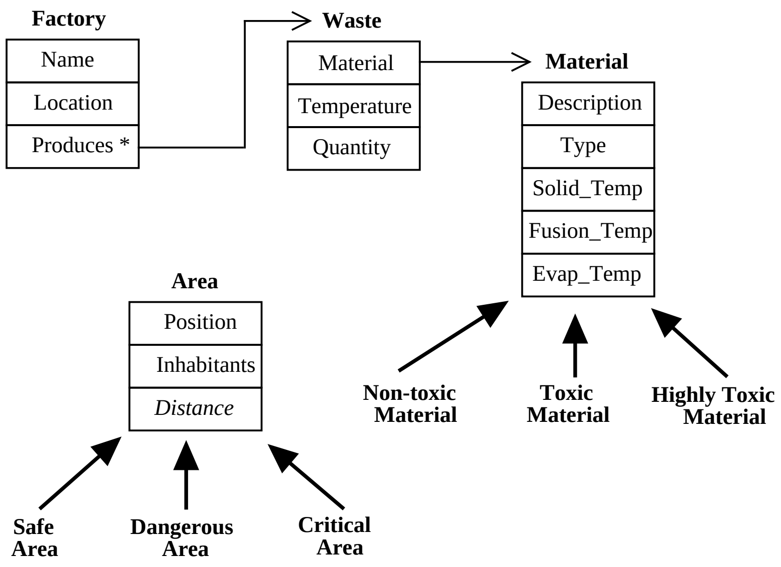

Figure 5. Database schema of our *Pollution Control* example

et al. in [34], should be employed.

5.3 Pollution Control

Consider the hierarchy in Figure 5, representing information about factories and their wastes, to control the safety with respect to pollution in certain areas. In particular, for each factory its location is recorded, as well as the set of wastes it produces. Wastes are materials, produced in a certain quantity (e.g., per day) and at a certain temperature. Materials are characterized by a number of physical properties and are classified in non-toxic, toxic and highly toxic. For each area the position and the number of inhabitants are recorded. A method to compute the distance of the area from a given location is also provided. Areas are classified as safe, dangerous or critical, depending on the number of inhabitants, and on the distance from factories producing certain quantities of highly toxic or toxic wastes.

Thus, there are several database operations that may cause a reclassification (and thus a class migration) of a certain area: the creation of a new factory, the change of the factory location, the update of the kind, or of the quantity (or of the temperature, if it influences the toxicity) of the wastes produced by a factory.

6 OBJECT EVOLUTION IN OTHER SYSTEMS AND MODELS

In this section, we first discuss evolution in object systems, other than Chimera, namely in O_2, GemStone and Iris. We then briefly discuss approaches to role modeling. Roles can be used, among other things, to provide alternative and/or additional views of objects and thus can be used to support object evolution.

6.1 Object Evolution in OODBMSs

In O_2 [14] instances can be specialized. This enables attributes and methods to be added and redefined for an individual object. In redefining an attribute or a method, the new definition must be compatible with the definition given by the class. In this respect, rules governing subtype definitions are used. For example, the domain of an attribute in an instance can be specialized with respect to the domain specified for the attribute in its class. However, the domain used in the instance must be a subclass of the domain specified in the class.

The only kinds of instance evolution supported in GemStone [8] are state evolution and migration between classes. GemStone allows state evolutions of objects to be controlled, since the state of objects can be "frozen" by using the **`immediateInvariant`** message which, when sent to an object, does not allow any further modifications to the state of the object. Note that it can also be specified that all the instances of a class are non-modifiable by using a flag (`instanceInvariant`) which appears in a class definition. In this case, an instance can only be manipulated during the transaction in which it was created. Once the transaction commits, the instance can no longer be modified. In GemStone, instances can only migrate between two given classes if:

- the new class is a subclass of the old class;
- the new class has no instance attributes in addition to those of the superclass (however, it can have additional instance methods and additional class attributes);

- the new class has the same storage format as the old class;
- the new class has exactly the same attribute domains as the old class.

Therefore, instances can migrate only under very restrictive conditions. The purpose of these restrictions is to allow instance migration only when there are no modifications on the involved instances. The message for requiring the migration of an object is `changeClassTo`. This message has the class to which the object must migrate as its only argument.

Types in the Iris system [15] can be added to and removed from an object. The object can thus migrate from a type T to a type T'; type T' has just to be added to the object and and type T deleted from the object. A type T' can moreover be added to an object without loosing the previous type to which the object belonged; to do this, the type T' has to be added to the object. Thus, in Iris objects can acquire and loose types dynamically with the instructions `ADD TYPE TO` and `REMOVE TYPE FROM`. The `ADD TYPE` statement also specifies the values to be assigned to the properties of the type which is added. Thus in Iris, arbitrary types can be added to an object. Iris, however, does not support context-dependent behavior since the entire set of types of an object is visible in every context. To avoid conflicts, two different types of an object must not have different methods with the same name.

Table 1 summarizes the various forms of object evolution supported by existing object-oriented database systems. As it can be seen from the table, most systems support very limited forms of evolution. In particular, note that Ode and Orion do not provide any form of object evolution in addition to state modification.

6.2 Role Models

Object models with roles support evolving objects, that is, objects migrating among classes, and objects that cannot be exclusively classified in a single class. In such object models, two different hierarchies are provided: a class (type) hierarchy and a role hierarchy. The role hierarchy is a tree of special types, called *role types*. The root of this tree defines the time-invariant properties of an object. The other nodes represent properties (types) that the object may acquire and loose during its lifetime. At any point in time, an entity is represented by an instance of

	Chimera	GemStone	Iris	O_2	Ode	Orion
Reference	[11]	[8]	[15]	[14]	[1]	[19]
State modification	YES	YES*	YES	YES	YES	YES
Explicit object migration	YES	Limited	YES	NO	NO	NO
Dynamic state-based classification	YES	NO	NO	NO**	NO	NO
Multiple class direct membership	YES	NO	YES***	NO	NO	NO
Exceptional instances	NO	NO	NO	YES	NO	NO

* As discussed earlier, in GemStone, an object state can be modified provided that the `instanceInvariant` flag has not been set to True in the class of the object.

** A view model has, however, been proposed for O_2 [29].

*** In Iris, arbitrary types can be added to an object. To avoid conflicts, however, two different types of an object cannot have different methods with the same name.

Table 1. Object evolution in existing OO data models

the root type and an instance of every role type whose role it currently plays. When an entity acquires a new role, a role-specific instance of the appropriate role type is created; when it abandons a role, the role-specific instance is destroyed. Thus, the role concept supports the dynamic nature of entities and their non-exclusive classification. Moreover, entities can exhibit role-specific behavior and roles can be used to restrict access to a particular context. The main drawback of models with roles compared to those allowing an object to be a direct member of multiple classes is that in a model with roles the different hierarchies (role and class ones) highly increases the complexity of the model. Such complexity impacts both the system architecture and the application development. For example, users must choose which features to model as classes and which as roles.

A first approach based on role hierarchies has been proposed by Sciore [28]. In his approach, real-world entities are modeled as object hierarchies where inheritance is determined on a per-object basis, thus merging class-based and prototype-based approaches. When an object receives a message, it either directly replies to the message or delegates the message to its parents. The observed behavior thus depends on the organization of the object hierarchy. A similar approach is proposed in [20]. Richardsons and Schwartz [25] have introduced the concept of *aspect* to model roles in strongly typed object-oriented database systems. More recently, Wieringa et al. in [33] have pointed out that objects may reference a particular role of an object and not only the object itself. The relevance of roles in object-oriented analysis has been stressed by Pernici [24], Papazoglou [23], and Martin and Odell [21]. The Fibonacci object data model [3] and the model proposed in [17] are quite similar. In both a role hierarchy can be associated with a root class; an object in this class can play any role belonging to the hierarchy. In both models, messages are dispatched according to the roles the object plays (though differently).

The emphasis in data models supporting roles is on context-dependent behavior. In Fibonacci [3], the selection of the methods to be executed depends on the role receiving the message. Dispatching is based on the following basic principles: (*i*) the most specific behavior prevails (unless a strict interpretation of messages is explicitly required); (*ii*) the most recently acquired behavior prevails. In Fibonacci, messages are interpreted as follows. When a role receives a message, first it is checked whether a descendant of this role exists having a proper (that is, non-inherited) method to reply to this message. Descendants are considered in inverse temporal order, that is, the more recently acquired descendant is considered first. Subtyping rules ensure that the delegated role can safely replace the receiving role. If no descendant role is able to handle the message, an implementation for the message is then looked for among the methods of the receiver itself. If also this search fails, an implementation for the message is finally looked for in the ancestor role from which the property corresponding to the message has been inherited. If the method invocation is type correct, the last search will certainly succeed. Fibonacci also supports an alternative dispatch mechanism (referred to as *strict binding*) to force an object to exhibit the behavior of a certain role without keeping into account possible specializations of the role. Strict binding must be explicitly required, through a special operator, when a message is sent to a role.

In the role model of Gottlob et al. [17], a message that cannot be handled by any role instance is delegated to the more general instance in the role hierarchy. In that model, however, no priorities are used to select a subrole among a number of candidates; this approach can lead to improper behaviors. Consider, for example, the roles `Enterpreneur` and `Employee` of type `Person`, each defining a method `income`. The income of a person might not be reduced neither to its income as an employee nor to its income as an entrepreneur, nor to the one of the most recently acquired role. By contrast, the income of a person can be obtained as an aggregate of the incomes of all its roles. Aggregation is one choice, but it is not always the most meaningful one.

7 CONCLUSIONS

In this paper we have discussed the various kinds of object evolution that should be supported in object database systems. The required capabilities include the possibility for an object to change class, either through an explicit migration operation or through state-based dynamic object classification. These kinds of evolution introduce problems both for implementation and consistency. In particular, object evolution introduces problems with respect to type checking, since an object changes its type during its lifetime. Moreover, if multiple class direct membership is allowed, problems concerning name conflicts arise and forms of context-dependence may need to be used. Semantic problems related to object evolution can be handled through appropriate integrity constraints. Those issues have been discussed in the context of the Chimera data model. The forms of object evolution supported by other object-oriented data models have also been surveyed.

Affiliations
Elisa Bertino is with the Dipartimento di Scienze dell'Informazione, Università di Milano, Via Comelico, 39/41 - 20135 Milano, Italy. E-mail: `bertino@dsi.unimi.it`. Giovanna Guerrini and Luca Rusca are with the Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso, 35 - 16146 Genova, Italy. E-mail: {`guerrini, rusca`}`@disi.unige.it`.

REFERENCES

[1] R. Agrawal and N. Gehani. Ode (Object Database and Environment): The Language and the Data Model. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 36–45, 1989.

[2] R. Agrawal, L. G. De Michiel, and B. C. Lindsay. Static Type Checking of Multi-Methods. In A. Paepcke, editor, *Proc. Sixth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 113–128, 1991.

[3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An Object Data Model with Roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. Nineteenth Int'l Conf. on Very Large Data Bases*, pages 39–51, 1993.

[4] E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In W. Olthoff, editor, *Proc. Ninth European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 102–126, 1995.

[5] E. Bertino and G. Guerrini. A Composite Object Model. Technical Report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1996. Submitted for publication.

[6] E. Bertino, G. Guerrini, and L. Rusca. Method Dispatching in Object Data Models with Multiple Class Direct Membership. Technical Report DISI-TR-96-17, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1996. Submitted for publication.

[7] E. Bertino and L. D. Martino. *Object-Oriented Database Systems - Concepts and Architecture*. Addison-Wesley, 1993.

[8] R. Breitl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databasases, and Applications*, pages 283–308. Addison-Wesley, 1989.

[9] S.J. Cannan and G.A.M. Otten. *SQL - The Standard Handbook*. McGraw-Hill, 1992.

[10] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polimorphism. *Computing Surveys*, 17:471–522, 1985.

[11] S. Ceri and R. Manthey. Consolidated Specification of Chimera. Technical Report IDEA.DE.2P.006.01, ESPRIT Project 6333, November 1993.

[12] C. Chambers. Predicate Classes. In *Proc. Seventh European Conference on Object-Oriented Programming*, pages 268–296, 1993.

[13] A. Coen Porisini, L. Lavazza, and R. Zicari. Static Type Checking of Object-Oriented Databases. Technical Report 91-60, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1991.

[14] O. Deux et al. The Story of 0_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[15] D. H. Fishman et al. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. Addison-Wesley, 1989.

[16] R. Gabriel, J. White, and D. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):28–38, September 1991.

[17] G. Gottlob, M. Schrefl, and B. Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 1994.

[18] G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. To appear in *Journal of Intelligent Information Systems*, Kluwer Academic Publishers, 1997.

[19] W. Kim et al. Features of the ORION Object-Oriented Database System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databasases, and Applications*, pages 251–282. Addison-Wesley, 1989.

[20] G. Kniesel. Implementation of Dynamic Delegation in Srongly Typed Inheritance-Based Systems. Technical Report IAI-TR-94-3, Institut für Informatik, Universität Bonn, 1994.

[21] J. Martin and J. J. Odell. *Object-Oriented Analysis and Design.* Prentice Hall, 1992.

[22] A. Ohori and K. Tajima. A Polimorphic Calculus for Views and Object Sharing. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–266, 1994.

[23] M. P. Papazoglou. Roles: A Methodology for Representing Multifaced Objects. In *Proc. of the International Conference on Database and Expert Systems Applications*, pages 7–12, 1991.

[24] B. Pernici. Objects with Roles. In *Proc. of the ACM Conference on Office Information Systems*, pages 205–215, 1990.

[25] J. Richardson and P. Schwartz. Aspects: Extending Objects to Support Multiple, Indipendent Roles. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 298–307, 1991.

[26] E.A. Rundensteiner. A Methodology for Supporting Multiples Views in Object-Oriented Databases. In *Proc. Eighteenth Int'l Conf. on Very Large Data Bases*, pages 187–198, 1992.

[27] M. Scholl, C. Laasch, and M. Tresch. Views in Object-Oriented Databases. In *Proc. Second International Workshop on Foundations of Models and Languages for Data and Objects*, pages 37–58, 1990.

[28] E. Sciore. Object Specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.

[29] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Proc. Fourth Int'l Conf. on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, pages 81–94, 1994.

[30] M. Staudt, M. Jarke, M. Jeusfeld, and H. Nissen. Query Classes. In S. Tsur, S. Ceri, and K. Tanaka, editors, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, number 760 in Lecture Notes in Computer Science, pages 283–295, 1993.

[31] L. A. Stein. A Unified Methodology for Object-Oriented Programming. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 211–222. John Wiley & Sons, 1991.

[32] J. Su. Dynamic Constraints and Object Migration. In G. M. Lohman, A. Sernadas, and R.Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 233–242, 1991.

[33] R. Wieringa, W. de Jonge, and P. Spruit. Roles and Dynamic Subclasses: a Modal Logic Approach. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, 1994.

[34] R. Wieringa, W. de Jonge, and P. Spruit. Using Dynamic Classes and Role Classes to Model Object Migration. *Theory and Practice of Object Systems*, 1(1):61–83, Spring 1995. Special Issue: Selected Papers from ECOOP '94.

[35] P. Yelland. Experimental Classification Facilities for Smalltalk. In A. Paepcke, editor, *Proc. Seventh Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 235–246, 1992.

[36] S. Zdonik. Object-Oriented Type Evolution. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 277–288. Addison-Wesley, 1990.