Objects with Multiple Most Specific Classes*

Elisa Bertino [1] *Giovanna Guerrini* [2]

[1] Dipartimento di Scienze dell'Informazione
Università di Milano - Milano, Italy
bertino@hermes.mc.dsi.unimi.it

[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova - Genova, Italy
guerrini@disi.unige.it

Abstract. In most object-oriented data models objects must belong to a single most specific class. This exclusive link between an instance and a class is often not adequate to model real-world situations. In this paper, we present an approach where objects can belong to several most specific classes. We formally address how the resulting conflicts can be handled, both for structural and behavioral components of objects. In particular, we formalize a notion of context and characterize the state of an object in terms of the set of its most specific classes. Moreover, we discuss two different dispatching approaches, one allowing context-dependent behavior, the other one ensuring behavior identity.

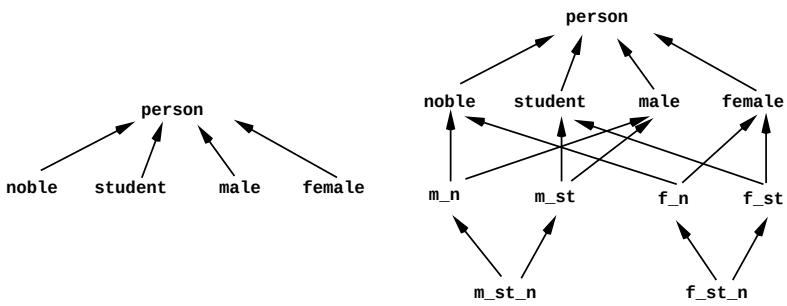
1 Introduction

One of the advantages of object-oriented data models [3] compared to other data models is that they support a direct representation of real-world domains. These models can represent the structure as well as the behavior of real-world entities. Most object-oriented data models are based on the notion of class; objects with the same structure and behavior are grouped into classes, which are in turn organized in inheritance hierarchies. In most models, real-world entities are represented as instances of the most specific class in which they can be classified[3]. Thus, real-world entities are partitioned into a set of disjoint classes.

However, when modeling application domains, objects are often found that naturally belong to several most specific classes. Consider as an example, a class *person*, whose instances can be classified along orthogonal dimensions, such as *students*, *nobles*, *males* and *females* (the corresponding hierarchy is illustrated in Figure 1(a)). According to the intuitive semantics, a person can be both a student, a noble and a female at the same time. Thus, the object representing this person does not have a unique most specific class, rather it has a set of most specific classes.

* Work partially supported by the EEC under ESPRIT Project 6333 IDEA.

[3] An object o can belong to a set of classes S. We call an element of S that has no subclass in S a *most specific* class of object o.



(a) (b)

Fig. 1. (a) Example of class hierarchy, extended in (b) with some meaningful subclasses

Although the above situation can be easily represented in a model with multiple inheritance by defining a subclass (say *noble_student_female*) of all the involved classes, this solution may lead to a lot of artificial subclasses, sometimes referred to as *intersection classes* [15]. Referring to the hierarchy above, the meaningful subclasses of the *person* class are six (see Figure 1(b)). 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 [14]. Indeed 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 object reference is.

Moreover, there are systems supporting 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 such models, it is difficult to ensure that an object belong to a unique most specific class, because it depends on the population predicates being disjoint. Referring to the above example, we may consider male, female and student as derived classes, provided that class person has two attributes `sex` and `profession`. In this case classes male, female and student are defined as derived subclasses with population predicates, respectively, `sex` $='$ `male`$'$, `sex` $='$ `female`$'$ and `profession` $='$ `student`$'$. In such a situation if a person is inserted in the object database, the insertion may result in an object belonging to several most specific classes and the user may even not realize this fact.

In our model, we take a different approach in that the extents of classes, that are not related via a subclass relationship, are not necessarily disjoint. Thus, in our model, 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, discarding the more common stricter (but "safer") approach of requiring an explicit subclass to exist in case of a non-empty intersection. 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[4].

In this paper we address the issues arising from multiple direct membership in the context of the Chimera data model [6, 12]. A way of solving 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[5]. 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[6], 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. Indeed, each object reference in each Chimera expression is assigned a single static type. We formalize the notion of context of an object reference in terms of static types. Then, we discuss what the state of an object is (in terms of attributes and attribute domains) depending on the set of its most specific classes. 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 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.

Therefore, the main contributions of this work can be summarized as follows: *i)* formalization of the problem of objects belonging to several most specific

[4] 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.

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

[6] 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 [2]).

classes, which has not been addressed on a formal basis yet; *ii)* proposal and comparison of two different dispatching approaches for messages sent to objects with multiple most specific classes.

The remainder of this paper is organized as follows. In Subsection 1.1 we survey related works. We introduce the relevant characteristics of the Chimera object-oriented data model in Section 2. Section 3 discusses issues related to the inheritance hierarchy. The following sections are specifically devoted to object handling with multiple most specific classes, being Section 4 concerned with structural aspects and Section 5 with behavioral ones (dispatching). Finally, Section 6 concludes the paper.

1.1 Related works

Several papers have addressed the relevance of a more flexible binding between objects and classes, but only few of them elaborate on how to solve name conflicts or how to handle dispatching when objects are members of several most specific classes.

The restriction that an object must be associated with a single most specific type was first pointed out in the framework of the Iris OODBMS [9]. 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.

Stein [16] discusses the motivations for supporting multiple direct membership, but her approach reduces an instance of multiple classes to an instance of a (virtual) subclass-by-multiple-inheritance of its classes. She did not discuss which policy is adopted for conflict resolution. Moreover, no context-dependent behavior can be supported by her approach.

The notions of context-dependent access restriction and context-dependent behavior are by contrast stressed in object data models supporting roles [2, 11, 14]. In these models, two different hierarchies are provided: a class (type) hierarchy and a role hierarchy. These different hierarchies make the model more complex, in that one must choose which features must be modeled as a class and which as a role. Moreover in these approaches, it is not always enforced that (one of) the most specific behavior(s) of the object is exhibited. Among these works, only the work by Albano et Al.[2] deals with the problem of method dispatching for objects with multiple roles. Dispatching is based on the following rules: *i)* the most specific behavior prevails (unless a strict interpretation of messages is explicitly required); *ii)* the most recently acquired behavior prevails. Thus, this approach to dispatching shows some similarities with our preferred class dispatching approach (cf. Section 5.1).

Another approach relaxing the strict object-most specific class link has been proposed by Sciore [15]. 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 a message is sent to an object, it either directly responds to the message or it delegates the message to its

parents. The behavior observed thus depends on the arrangement of the object hierarchy. However, this approach lacks object identity semantics and strong typing.

2 Reference object model

Chimera [6] is a novel database language designed with the aim to integrate active and deductive rules with object-oriented features[7]. Chimera integrates an object-oriented data model, a declarative query language based on deductive rules and an active rule language for reactive processing. The Chimera model is a rather classical object-oriented data model with operations and inheritance; in addition to objects, the model also supports the notion of value classes (extensionally populated) and of views (populated by means of passive rules). The signature of objects describes not only the state (attributes) and the operations for accessing and manipulating object instances, but also integrity constraints and triggers; in addition, class attributes, operations, and constraints collectively apply to classes. The Chimera language integrates object operations (used for manipulating objects), passive rules (used for expressing derived classes and views) and active rules (used for expressing database triggers). In what follows we recall only the aspects of the Chimera formal model relevant for this work. We refer the reader to [12] for a complete definition of the model.

The set of Chimera types $\mathcal{T}$ (that are collection of values) is defined as the union of value types ($\mathcal{VT}$) and object types ($\mathcal{OT}$). Object types are class names and their instances are object identifiers. Value types are defined as follows. They can be either basic domains or structured types built by applying the set, list or record constructors. Named types (possibly constrained) and value classes are also value types.

Definition 1 *[12] The set of Chimera value types $\mathcal{VT}$ is inductively defined as follows*

- *the predefined basic value types (integer, real, bool, character) are value types*
- *if T is a value type or an object type then $list-of(T)$ and $set-of(T)$ are value types*
- *if $T_1, \ldots, T_n$ are value types or object types and $a_1, \ldots, a_n$ are distinct attribute names in $\mathcal{AN}$, then $record-of(a_1 : T_1, \ldots, a_n : T_n)$ is a value type*
- *if T is a value type and Tn is a type name, and Tn is declared as a name for T through a declaration $Tn : T$, then Tn is a (named) value type*
- *if T is a value type, Tn is a type name, and R is a set of rules, and Tn is declared as $Tn : T$ constraint R, then Tn is a (constrained) value type*
- *if T is a value type, Tn is a type name, and S is a set of values, and Tn is declared as $Tn : T$ extent S, then Tn is a value type, which we call value class.* □

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

We recall from [12] the notion of extension, that is, the set of legal values of each Chimera type. The extension of a type depends on the explicit extent for such types, like value classes and object classes, that have one. To model these extents we introduce a function $\pi = (\pi_V, \pi_O)$, such that π_V assigns a set of values to each value class (*value assignment*), and π_O assigns a set of object identifiers to each object class (*oid assignment*)[8].

Definition 2 *(Type extension) [12]. The extension of the type T under the assignment π (denoted as $[\![T]\!]_\pi$), is defined as follows:*

- $null \in [\![T]\!]_\pi$, $\forall T \in \mathcal{T}$
- $[\![D_i]\!]_\pi = ED_i$, *for D_i basic domain (ED_i is postulated)*
- *if* $c \in \mathcal{OT}$, $[\![c]\!]_\pi = \pi(c)$
- $[\![set-of(T)]\!]_\pi = 2^{[\![T]\!]_\pi}$, *where 2^S denotes the power set of the set S*
- $[\![list-of(T)]\!]_\pi = \{[v_1, \ldots, v_n] \mid n \geq 0, v_i \in [\![T]\!]_\pi \ \ \forall i, 1 \leq i \leq n\}$
- $[\![record-of(a_1 : T_1, \ldots, a_n : T_n)]\!]_\pi = \{(a_1 : v_1, \ldots, a_n : v_n) \mid a_i \in \mathcal{AN}, v_i \in [\![T_i]\!]_\pi \ \ \forall i, 1 \leq i \leq n\}$
- $[\![Tn]\!]_\pi = [\![type(Tn)]\!]_\pi$ *if Tn is an unconstrained named type and $type(Tn)$ is the type denoted by the name*
- $[\![Tn]\!]_\pi = \{v \mid v \in [\![type(Tn)]\!]_\pi$, *$v$ meets constr(Tn)*$\}$ *if Tn is a constrained value type and $constr(Tn)$ denotes its constraint*
- $[\![VC]\!]_\pi = \pi(VC)$ *if VC is a value class.* □

A class[9], in addition to having an extent, is characterized by a structural and a behavioral component. Given a class c, $A(c)$ denotes the set of attributes of that class. Furthermore, $dom(a, c)$, for $a \in A(c)$, denotes the domain of attribute a in class c. Similarly, for the behavioral component of classes, $M(c)$ denotes the set of methods of class c and $sign(m, c)$, for $m \in M(c)$, denotes the signature of method m in class c.

Let $\mathcal{C}$ denote the set of all classes. The user-defined ISA hierarchy is represented through a partial order $\leq_{ISA} \subseteq \mathcal{C} \times \mathcal{C}$ on classes. Moreover, $c <_{ISA} c'$ denotes the relation $c \leq_{ISA} c' \wedge c \neq c'$[10]. In the following, $ISA^*(c) = \{c' \mid c \leq_{ISA} c'\}$ denotes the set of c superclasses, and

$ISA(c) = ISA^*(c) \setminus \bigcup_{c' \ such \ that \ c <_{ISA} c'} ISA^*(c')$

denotes the set of direct superclasses of c.

Definition 3 *(Most specific class). Given a set of classes C, $ms(C)$ denotes the set $\{c \mid c \in C \wedge \nexists\ c' \in C$ such that $c' <_{ISA} c\}$. A class c is said a most specific class in C if $c \in ms(C)$.* □

Definition 4 *(Objects). An object is a triple $o = (i, v, CS)$, where*

- *i is the object identifier of o*

[8] Thus, if c is a class, $\pi(c)$ is the set of the identifiers of objects members of c.

[9] With the term *class* we refer to object classes, that is classes whose instances are objects, as opposed to value classes, that are collections of values.

[10] In what follows, given an order $\leq$, $<$ denotes the non-reflexive relation obtained from the order $\leq$, that is, $c_1 < c_2$ iff $c_1 \leq c_2$ and $c_1 \neq c_2$.

- *v is a value, called state of o*
- *$CS \subseteq \mathcal{C}$ is the set of most specific classes to which o belongs, that is* $CS = ms(\{c \mid c \in \mathcal{C} \wedge i \in [\![c]\!]\})$. □

Each object is required to be an instance of (at least) a class; therefore, $o \downarrow 3 \neq \emptyset$[11] must hold. Moreover, all the classes in $o \downarrow 3$ are most specific with respect to the others, that is $ms(o \downarrow 3) = o \downarrow 3$.

Starting from the ISA hierarchy on classes and from the inheritance relationships on value types due to constrained value types and value classes, which we suppose expressed by an order $\leq_{ISA_V}$ [12], a subtype relation $\leq_T \subseteq \mathcal{T} \times \mathcal{T}$ is defined as follows.

Definition 5 *(Subtype relation) [12]. Given two types $T_1, T_2 \in \mathcal{T}$, T_2 is a subtype of T_1 (denoted as $T_2 \leq_T T_1$) iff at least one of the following conditions holds:*

- *$T_1 = T_2$ or T_1 and T_2 unconstrained value types with $type(T_1) = type(T_2)$*
- *$T_2 \leq_{ISA_V} T_1$ or $T_2 \leq_{ISA} T_1$*
- *$T_2 = set-of(T_2')$, $T_1 = set-of(T_1')$ and $T_2' \leq_T T_1'$*
- *$T_2 = list-of(T_2')$, $T_1 = list-of(T_1')$ and $T_2' \leq_T T_1'$*
- *$T_1 = record\text{-}of(a_1 : T_1', \ldots, a_n : T_n')$, $T_2 = record\text{-}of(a_1 : T_1'', \ldots, a_n : T_n'')$ and for each i, $1 \leq i \leq n$, $T_i' \leq_T T_i''$.* □

$\leq_T$ is a partial order. The notions of *least upper bound* (lub, $\sqcup$) and of *greatest lower bound* (glb, $\sqcap$) with respect to this order have been discussed in [12]. Note that since $(\mathcal{T}, \leq_T)$ is a DAG and is not a lattice, *lub*s and *glb*s do not always exist.

2.1 Static and dynamic types

Chimera is a strongly typed database language. When writing a Chimera expression, one must explicitly state the type of each variable used in the expression. This is accomplished in Chimera by using class formulas. Indeed, in Chimera class formulas are used as a typing mechanism. Class formulas are built from class or type names representing unary predicate symbols. Thus, to state in an expression that a variable X is of type T the class formula $T(X)$ is added to the expression. The class formula $T(X)$ has also the effect of stating that $X \in [\![T]\!]$, thus providing a domain for the evaluation of the expression.

The scope of a type assignment for a variable is that of an expression. Each Chimera expression must contain a class formula for each variable. Thus, each variable is associated with a unique type, with respect to which type checking is performed. Declaring a variable X of type T influences type checking in that the only features available for X are those of type T. That is, type checking of the expression, in which X appears, is done regarding X as a term of type T. We say that T is the *static type* of variable X, because it is the type with respect to which the static type checking of the expression containing X is done.

[11] $o \downarrow n$ denotes the n-th component of a tuple o.

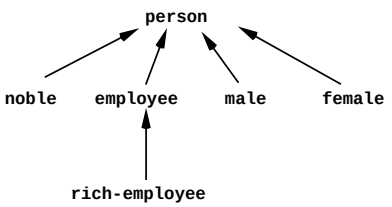


Fig. 2. Class hierarchy of Example 1

At execution time variable X is instantiated with a value of type T. Note that when T is an object type, the variable is instantiated with the identifier of an object member of the class identified by T. This means that the variable may also be instantiated with the identifier of an object which is a direct member of a subclass T' of that class. T' is said to be a *dynamic type* of variable X. However, since we consider objects possibly belonging to several most specific classes, the dynamic type of a variable denoting an object is, in general, not unique. Thus a variable is characterized by a set of dynamic types. The only constraint on those dynamic types is that they must have a common ancestor in the inheritance hierarchy. Moreover, the static type T of the variable denoting the object must be a subtype of this ancestor. Finally at least one dynamic type must be a subtype of the static one. These relationships are stated by the following rule.

Rule 1: Relationship between the static type and the dynamic types of objects

Let $\mathcal{C}$ be a set of classes with the partial order $\leq_{ISA}$. Let $t^s(X)$ be the static type of an object reference X. Let o be an object to which X is instantiated at run-time, and let $o \downarrow 3$ be the set of dynamic types of o. The following relationships hold:

1. $\exists c \in o \downarrow 3$ such that $c \leq_{ISA} t^s(X)$
2. $\exists c \in \mathcal{C}$ such that $\forall c' \in o \downarrow 3 \;\; c' \leq_{ISA} c \wedge t^s(X) \leq_{ISA} c$[12]. □

These relationships between static and dynamic types of an object hold for all kinds of object references, not only for object-denoting variables.

Example 1 Referring to the inheritance hierarchy of Figure 2, consider a variable X declared of type `employee`. At run-time variable X can be instantiated with an object o such that $o \downarrow 3 = \{$`female`, `rich-employee`, `noble`$\}$. The above relationships between $t^s(X) =$ `employee` and $o \downarrow 3$ hold. ◇

The static type of an object is not only exploited to restrict access to a particular context, but may also determine a context-specific behavior. Dynamic types are used for choosing the appropriate method definition to be used on the

[12] Due to the properties of Chimera inheritance hierarchies (cf. Section 3) this is equivalent to the statement that each class in $o \downarrow 3$ and $t^s(X)$ belong to the same strongly connected component $\mathcal{H}_i$ of the inheritance DAG.

object denoted by X. Note that the chosen implementation must always be the implementation provided by one of the most specific classes of the object.

2.2 Object references and contexts

In this subsection we examine the possible forms that can be used in Chimera to denote an object and, for each of them, we show that it is possible in each expression to derive a unique *context* or static type for each expression denoting an object (object reference). The context of an object reference can be derived from the types declared for the variables in the expression and from schema information.

Conceptually, objects are referenced through their identifiers. However, according to the Chimera syntax, object identifiers are regarded as a system feature and they are not accessible by the user. Thus, objects can be denoted by variables, or they can be indirectly denoted by path expressions (containing attribute accesses and method invocations) starting from an object-denoting variable. Thus, an object reference can have one of the following forms:

- an object-typed variable;
- an attribute access $e.a$ where e is an object reference and a is an attribute name;
- a method invocation $e.m(e_1, \ldots, e_n)$ where e is an object reference, m is a method name and $e_1, \ldots, e_n$ are expressions denoting the parameters of the invocation and can be either object references or values.

For the sake of conciseness, we do not address here the detailed forms of values we support in Chimera (basic values, structured values with set, list and record constructors) nor the constructs of record and list field selection. The complete syntax of Chimera expressions is presented in [12].

Since a unique type T is explicitly associated with each variable in each expression (through the class formula mechanism), the static type or context of each variable is known to be T. To deduce the context of object references which are path expressions consisting of attribute accesses and method invocations, we use the following rule. The context derivation rule is expressed through two inference rules[13] to be instantiated letting the metavariable T varying on the set of object types $\mathcal{OT}$ and the metavariable e varying on the set of object references.

Rule 2: Context derivation rule

$$\frac{e : T \quad a \in A(T) \quad dom(a, T) = T'}{e.a : T'} \quad T \in \mathcal{OT}$$

$$\frac{e : T \quad m \in M(T) \quad sign(m, T) = T_1 \times \ldots \times T_n \rightarrow T' \quad \forall i, 1 \leq i \leq n, e_i : T_i}{e.m(e_1, \ldots, e_n) : T'} \quad T \in \mathcal{OT}$$

□

[13] The meaning of these inference rules is the following: if the conditions in the rule premises (the upper part of the rule) are satisfied, then the rule consequence (the lower part of the rule) can be inferred.

It can be easily shown that, if each value is associated with a unique type, the above rule assigns a unique type to each object reference. Since Chimera typing rules assign a unique type to each value, and since, as deduced above, the context of a variable is uniquely determined, the following result holds.

Proposition 1 *Given an expression, each object reference contained in the expression has a unique context in this expression.*

Example 2 Consider a class c such that $dom(a, c) = c_1$, $dom(b, c) = c_2$ and $sign(m, c) = c_1 \times c_2 \rightarrow c$. The context of the object reference $X.m(X.a, X.b).a$ in an expression containing the class formula $c(X)$ is c_1. $\diamond$

3 Inheritance hierarchies

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 $ISA^*(c_1) \cap ISA^*(c_2) \neq \emptyset$.

In Chimera inheritance hierarchies are DAGs characterized by a certain number of strongly connected components. Moreover, each strongly connected component of the DAG is characterized by a single node without incoming edges, which we call *root* of the strongly connected component. Indeed, we do not impose the existence of a common root (as an `object` class) of the entire hierarchy, rather the hierarchy is partitioned into multiple strongly connected components, and each component has a single root. Note that an object can belong to several most specific classes only if the classes belong to the same strongly connected component in the DAG, that is, if they have a common ancestor.

The sets of oids in different strongly connected components are therefore disjoint. Consider a set of root classes $c_1, \ldots, c_m$, then Ext_i, for $i = 1, \ldots, m$ denotes the extent of c_i ($[\![c_i]\!]$) which is the extent of the whole strongly connected component rooted at c_i. We may think of the set of all classes $\mathcal{C}$ as partitioned in $\mathcal{H}_1, \ldots, \mathcal{H}_m$, that is, the m distinct hierarchies (corresponding to the m strongly connected components) we have on $\mathcal{C}$.

3.1 A total order on classes

Since an object may have several most specific classes in a hierarchy, a total order $\preceq_i$ of the classes in an inheritance hierarchy $\mathcal{H}_i$ must be provided. Thus it may be useful to choose which is the *preferred* class of an object and therefore to know which methods must be executed for the object. Thus, a total order, denoted as $\sqsubseteq$ and called *ISA-independent order*, is imposed on the classes in the hierarchy. This order may simply be based on the definition order, or it may be specified by the user with `before/after` declarations in class definitions. Alternatively this order may be obtained by considering the definition order as a default, which may be overridden by `before/after` declarations.

Note, however, that the ISA-independent order must be consistent with the ISA order $\leq_{ISA}$. The notion of consistency is formalized as follows.

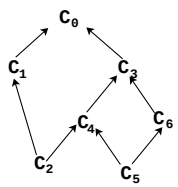


Fig. 3. Inheritance hierarchy $\mathcal{H}_0$ of Example 3

Definition 6 *The total order $\sqsubseteq$ on a set of classes $\mathcal{H}_i$ is said to be consistent with the (partial) ISA order $\leq_{ISA}$ on $\mathcal{H}_i$ if $\forall c_1, c_2 \in \mathcal{H}_i$ such that $c_1 \leq_{ISA} c_2$ does not exist any class $c \in \mathcal{H}_i$ such that the following conditions are satisfied*

- $c_1 \not\leq_{ISA} c$ *and* $c \not\leq_{ISA} c_1$,
- $c_2 \not\leq_{ISA} c$ *and* $c \not\leq_{ISA} c_2$,
- $c_2 \sqsubseteq c \sqsubseteq c_1$. □

We define now a total order on the hierarchy $\mathcal{H}_i$. This total order is obtained by merging the order induced by the ISA relationship with the ISA-independent total order $\sqsubseteq$. The order induced by the ISA relationship dominates the total order, in that for every two classes c_1 and c_2, if $c_1 \leq_{ISA} c_2$ holds, then $c_1 \preceq_i c_2$ holds. There is no need of relating in the order classes from different hierarchies, since they cannot have common instances (the extents of different hierarchies are disjoint).

Definition 7 *Let $\mathcal{H}_i$ be an inheritance hierarchy, $\sqsubseteq$ be an ISA-independent total order on $\mathcal{H}_i$, consistent with the ISA relation $\leq_{ISA}$. A total order on the classes in $\mathcal{H}_i$, denoted as $\preceq_i$, is defined as follows. Let $c_1, c_2 \in \mathcal{H}_i$, then $c_1 \preceq_i c_2$ if*

(i) $c_1 \leq_{ISA} c_2$, *that is, c_1 is a subclass of c_2, or*

(ii) $c_1 \not\leq_{ISA} c_2$, $c_2 \not\leq_{ISA} c_1$, *and $c_1 \sqsubseteq c_2$ (that is, neither c_1 is a subclass of c_2 nor vice-versa, but c_1 precedes c_2 in the $\sqsubseteq$ order).* □

Note that in the above definition we have only exploited the order $\sqsubseteq$ to define $\preceq_i$. However, there are different possible options for defining $\preceq_i$ as, for example, getting the ancestors of the two classes at the same level in the DAG and comparing by $\sqsubseteq$ those ancestors or considering the order on levels and comparing by $\sqsubseteq$ only classes at the same level.

Example 3 Consider the inheritance hierarchy of Figure 3. Suppose the ISA-independent order be denoted by the subscripts of class names, that is

$c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq c_3 \sqsubseteq c_4 \sqsubseteq c_5 \sqsubseteq c_6$

whereas the ISA order corresponding to the tree is

$c_2 \leq_{ISA} c_1 \leq_{ISA} c_0$ $\quad$ $c_3 \leq_{ISA} c_0$ $\quad$ $c_2 \leq_{ISA} c_4$

$c_5 \leq_{ISA} c_4 \leq_{ISA} c_3$ $\quad$ $c_5 \leq_{ISA} c_6 \leq_{ISA} c_3$

The ISA-independent order is consistent with the ISA order, and the classes are related in the total order $\preceq_0$ as follows

$c_2 \preceq_0 c_1 \preceq_0 c_5 \preceq_0 c_4 \preceq_0 c_6 \preceq_0 c_3 \preceq_0 c_0$. ◇

Proposition 2 *Let $\mathcal{H}_i$ be an inheritance hierarchy, then the $\preceq_i$ relation defined according to Definition 7 is a total order on $\mathcal{H}_i$. Moreover, this order coincides with $\leq_{ISA}$ when $\leq_{ISA}$ is defined, that is, $\forall c_1, c_2 \in \mathcal{H}_i$ $c_1 \leq_{ISA} c_2 \Rightarrow c_1 \preceq_i c_2$.*

4 Attributes

In this section we consider the structural component of objects. We determine for an object with multiple most specific classes, the state of the object, that is, which are the attributes of the object, and which are the proper domains for these attributes.

Roughly speaking, the state of an object belonging to several most specific classes is the union of the attributes in those classes. However, the sets of attributes in those classes may not be disjoint, that is, *name conflicts* may arise. To handle these situations 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 then the two attributes in the two classes have accidentally the same name, but represent different information, that must be kept in separate ways. Thus, the object may have two different values for the two attributes (a renaming policy is applied).

Example 4 As an example of the first situation, consider the inheritance hierarchy of Figure 1 and suppose that an attribute `spouse` is present in class `person`. This attribute is inherited by all the subclasses of `person`. Consider now an object which is both a noble and a female, then the attribute `spouse` belongs to the intersection of the attribute sets of `noble` and `female`, and it has in both classes the same source (`person`), and thus the object has a single attribute `spouse` (this corresponds to the intuition that the information carried out by the two attributes in the two different classes is the same).

As an example of the second situation, consider the inheritance hierarchy composed by a class `person` and two subclasses `student` and `employee`. Suppose that in class `student` a new attribute `code` is defined, representing the number of the student in the university. Suppose moreover, that in class `employee` a new attribute `code` is defined, representing the number of the employee in the welfare office. Consider now an object which is both an employee and a student, then the attribute `code` belongs to the intersection of the attribute sets of `employee` and `student`, but it has different sources in the two classes: it has source `employee` in class `employee`, while it has source `student` in class `student`. Thus the object have two different attributes, say `student-code` and `employee-code` (this corresponds to the intuition that the information carried out by the two attributes in the two different classes is different). ◇

We now formalize the above notions. We recall that, given a class c, $A(c)$ denotes the set of attributes of that class. Moreover, $A^+(c) = \{a \mid a \in A(c) \wedge \nexists\ c', c <_{ISA} c' \text{ such that } a \in A(c')\}$ denotes the set of proper (that is, not inherited) attributes of class c.

First, we define the notion of *source* of an attribute. Intuitively, the source of an attribute a in a class c is the most general superclass of c in which the attribute a is defined. Thus, it is the class from which c has inherited attribute a.

However, because of multiple inheritance, there may be two different superclasses of c, not related by the ISA relationship, in which attribute a is defined. In such a situation, we impose that in the definition of class c an explicit specification (through a *qualification mechanism*) is given of the class from which the attribute is inherited. Thus, each class inherits each attribute from one of its superclasses, the most general in the path specified by the qualification mechanism. This unique class is called source of the attribute in the class.

Given the set of classes $\mathcal{C}$, a function $inh^a : \mathcal{C} \rightarrow \mathcal{C}$ is defined for each class c, $c \in \mathcal{C}$, such that $a \in A(c) \setminus A^+(c)$. Given a class c, function inh^a returns the direct superclass of c from which c inherits attribute a. The definition of this function is as follows:

- if a unique class c' exists such that $c' \in ISA(c), a \in A(c')$ then $inh^a(c) = c'$
- otherwise (if more than one class c' exists such that $c' \in ISA(c)$ and $a \in A(c')$) the resolution of name conflicts for multiple inheritance in Chimera requires that in c definition an explicit qualification mechanism is exploited to state from which class attribute a is inherited; if c^q is the qualified class, then $inh^a(c) = c^q$.

We remark that, according to its definition, inh^a is a monotonic (strictly increasing) function, in that $\forall c \in \mathcal{C}$ if $inh^a(c)$ is defined $c <_{ISA} inh^a(c)$ holds.

Let $inh^{a^*} \subseteq \mathcal{C} \times \mathcal{C}$ denote the relation corresponding to the transitive closure of the relation associated with function inh^a[14]. If inh^{a^*} contains two pairs (c, c') and (c, c'') then either $c' <_{ISA} c''$ or $c'' <_{ISA} c'$ holds. Thus, inh^{a^*} contains ascending paths of the inheritance hierarchy. Moreover, given a set of classes C, let $mg(C)$ denote the set $\{c \mid c \in C \wedge \nexists\ c' \in C$ such that $c <_{ISA} c'\}$, that is the set of *most general* classes in C.

Definition 8 *(Source). The source of an attribute $a \in \mathcal{AN}$ in a class $c \in \mathcal{C}$, denoted as $source(a, c)$, is class c if $a \in A^+(c)$. It is a class $c' \in \mathcal{C}$, where $\{c'\} = mg(\{c^* \mid (c, c^*) \in inh^{a^*}\})$, otherwise. The source is undefined if $a \notin A(c)$.* □

Note that for each class $c, c \in \mathcal{C}$, $\{c^* \mid (c, c^*) \in inh^{a^*}\}$ is a totally ordered set with respect to the order $<_{ISA}$. Therefore, if $mg(\{c^* \mid (c, c^*) \in inh^{a^*}\})$ is not empty, it always contains a single element, that is the maximum of the set. As a consequence, the following result holds.

Proposition 3 *Let c be a class, and a be an attribute such that $a \in A(c)$, then $source(a, c)$ is always defined and it is unique.*

We now define which is the set of attributes of an object belonging to two most specific classes.

Definition 9 *Let o be an object such that $o \downarrow 3 = \{c_1, c_2\}$. The set of o attributes is $A(c_1) \uplus A(c_2)$, where the operation $\uplus$ is defined as follows:*

$$\begin{aligned} A(c_1) \uplus A(c_2) = &\{a \mid a \in A(c_1) \cup A(c_2) \wedge a \notin A(c_1) \cap A(c_2)\} \cup \\ &\{a \mid a \in A(c_1) \cap A(c_2) \wedge source(a, c_1) = source(a, c_2)\} \cup \\ &\{c_1\text{-}a \mid a \in A(c_1) \cap A(c_2) \wedge source(a, c_1) \neq source(a, c_2)\} \cup \\ &\{c_2\text{-}a \mid a \in A(c_1) \cap A(c_2) \wedge source(a, c_1) \neq source(a, c_2)\} \end{aligned}$$ □

[14] The relation associated to function inh^a is the set $\{(c, c') \mid c, c' \in \mathcal{C} \wedge inh^a(c) = c'\}$.

Example 5 Consider the inheritance hierarchy composed by class `person` with two subclasses `student` and `employee`. Suppose that

$A(\texttt{person}) = \{\texttt{name}, \texttt{address}, \texttt{spouse}\}$,
$A^+(\texttt{student}) = \{\texttt{code}, \texttt{average_score}, \texttt{year}\}$,
$A^+(\texttt{employee}) = \{\texttt{code}, \texttt{salary}, \texttt{role}\}$

and therefore

$A(\texttt{student}) = \{\texttt{name}, \texttt{address}, \texttt{spouse}, \texttt{code}, \texttt{average_score}, \texttt{year}\}$,
$A(\texttt{employee}) = \{\texttt{name}, \texttt{address}, \texttt{spouse}, \texttt{code}, \texttt{salary}, \texttt{role}\}$.

Then,

$A(\texttt{student}) \uplus A(\texttt{employee}) = \{$ `name, address, spouse, year, role,` `employee` $-$ `code`, `student` $-$ `code`, `average_score`, `salary`$\}$. ◇

Operator $\uplus$ is associative. Thus Definition 9 can be easily generalized to arbitrary sets of classes.

We now consider the domain (that is, the set of legal values) for attributes. For the attributes not belonging to the intersection of the attribute sets of the most specific classes of the object, and for the ones that have been renamed because of different sources, the domain is simply the one specified in the class to which the attribute belongs. By contrast, for the attributes in the intersection of the attribute sets of two classes and with the same source, a new domain must be determined. Intuitively, the legal values for an attribute must be legal values for both the attributes in the two classes, that is, they must belong to both the domain.

Definition 10 *Let* $dom(a, c_1) = T_1$ *and* $dom(a, c_2) = T_2$. *Let* o *be an object such that* $o \downarrow 3 = \{c_1, c_2\}$. *Then, value* v *is a legal value for attribute* a *in* o, *if* $v \in [\![T_1]\!] \cap [\![T_2]\!]$. □

Note that the domain of an attribute for an object belonging to several most specific classes cannot be always characterized in terms of an existing type. attribute In particular, if a greatest lower bound of type T_1 and T_2 exists, then the type of the attribute for the object is $T_1 \sqcap T_2$, but a greatest lower bound of two types in the hierarchy does not always exist. Moreover, in Chimera, $[\![T_1 \sqcap T_2]\!] \subseteq [\![T_1]\!] \cap [\![T_2]\!]$ holds for any two types T_1 and T_2, while the converse $[\![T_1]\!] \cap [\![T_2]\!] \subseteq [\![T_1 \sqcap T_2]\!]$ holds only if $T_1 \leq_T T_2$ (or vice-versa). In this case, indeed, $T_1 \sqcap T_2 = T_1$ and $[\![T_1]\!] \cap [\![T_2]\!] = [\![T_1]\!]$ in that $[\![T_1]\!] \subseteq [\![T_2]\!]$. If $T_1 \not\leq_T T_2$ and $T_2 \not\leq_T T_1$, then $T_1 \sqcap T_2$ can be defined only if T_1 and T_2 have a common subclass (by multiple inheritance). However in Chimera if T is a subclass of both T_1 and T_2 ($T_1, T_2 \in ISA(T)$), an object o may exist such that $o \in [\![T_1]\!] \wedge o \in [\![T_2]\!]$ but $o \notin [\![T]\!]$. Indeed, in such a way o can exhibit a context-dependent behavior, while it would not if $o \in [\![T]\!]$.

Example 6 Consider the inheritance hierarchy of Figure 1. Suppose that an attribute `spouse` $\in A(\texttt{person})$ and that $dom(\texttt{spouse}, \texttt{person}) = \texttt{person}$. Then,

`spouse` $\in A(\texttt{student}), A(\texttt{noble}), A(\texttt{male}), A(\texttt{female})$

and suppose that its domain is refined in the following way:

$dom(\texttt{spouse}, \texttt{student}) = \texttt{person}$
$dom(\texttt{spouse}, \texttt{noble}) = \texttt{noble}$

$dom(\texttt{spouse}, \texttt{male}) = \texttt{female}$
$dom(\texttt{spouse}, \texttt{female}) = \texttt{male}$.

In this case, if an object is a student and a noble, then the domain of the attribute `spouse` for this object can be characterized in terms of an existing type, that is $\texttt{noble} = \texttt{noble} \sqcap \texttt{person} = dom(\texttt{spouse}, \texttt{noble}) \sqcap dom(\texttt{spouse}, \texttt{student})$.

By contrast, if an object is a noble female, since
$dom(\texttt{spouse}, \texttt{noble}) \sqcap dom(\texttt{spouse}, \texttt{female}) = \texttt{noble} \sqcap \texttt{male}$
is not defined, the domain of the attribute `spouse` for this object cannot be characterized in terms of an existing type, but we can only say that the legal values for this attribute are those in $[\![\texttt{noble}]\!] \cap [\![\texttt{male}]\!]$ (that is, to objects belonging to the two most specific classes `noble` and `male`). ◇

Thus, the attributes for which a new domain is determined are the ones which are present in two different classes with a common source. This means that the attribute was defined in a class c (the source) with a domain, say T. Classes c_1 and c_2 have both inherited the attribute, possibly refining its domain into T_1 and T_2, respectively. Due to the refinement condition, both $T_1 \leq_T T$ and $T_2 \leq_T T$ hold. Thus a "compatibility" between types T_1 and T_2 exists, in that they are both subtypes of a common supertype. However, the existence of a common supertype does not ensure neither that $T_1 \sqcap T_2$ is defined nor that a legal value for the attribute exists (that is, $[\![T_1]\!] \cap [\![T_2]\!] \neq \emptyset$). Indeed, referring to the hierarchy of Figure 1, though `male` and `female` have a common supertype (`person`), $\texttt{male} \sqcap \texttt{female}$ is undefined and $[\![\texttt{male}]\!] \cap [\![\texttt{female}]\!] = \emptyset$.

The state of an object (that is, $o \downarrow 2$) is characterized by the set of most specific classes of the object (that is, $o \downarrow 3$) as follows.

Definition 11 *Let o be an object, such that $o \downarrow 2 = (a_1 : v_1, \ldots, a_n : v_n)$. Then:*

- $\biguplus_{c \in o\downarrow 3} A(c) = \{a_1, \ldots, a_n\}$
- $\forall i, 1 \leq i \leq n, v_i \in \bigcap_{c \in o\downarrow 3} [\![dom(a_i, c)]\!]$. □

4.1 Attribute access

We now discuss how an object state can be accessed from outside the object[15]. The attribute renaming, introduced for handling attributes with different sources, is used only for ease of internal representation, but from the outside, both attributes are referred to with the same name. Thus, the problem arises of choosing the correct (that is, the intended) attribute in each expression containing an attribute access.

Referring to Example 5, suppose that we must determine the semantics of the expression `X.code`, where `X` is a variable denoting an object which is both a student and an employee. The choice of accessing an attribute instead of the other one is carried out depending on the *preferred* class of the object, in the context of the specified expression. In our model, to allow different *views* of the object, depending on the context in which it is considered, the preferred class

[15] Note that in Chimera object attributes can be directly accessed, for example in queries.

for the object is determined by the context (static type) of the object reference denoting the object in the considered expression.

Note that the problem of choosing the intended attribute to be accessed arises only for attributes with different sources, because in the case of attributes with common source a single value is stored, and this value is always returned, independently from the class considered for the object.

We now formalize these notions. A function $value : \mathcal{O} \times \mathcal{C} \rightarrow \mathcal{V}$ is defined that, given an object o and a class c, if o is a member of c (that is, if $o \downarrow 1 \in [\![c]\!]$), returns the state of object o seen as an instance of class c. This function only returns the fields of the state value proper of the structural component of c. $value(o, c)$ is determined as follows. Let $o \downarrow 2$ be the record value $(a_1 : v_1, \ldots, a_n : v_n)$. Then, for $i = 1 \ldots n$:

- if $a_i \in A(c)$ then $a_i : v_i$ is a field of $value(o, c)$, and
- if $a_i = c\text{-}a'_i$ and $a'_i \in A(c)$, then $a'_i : v_i$ is a field of $value(o, c)$.

Note that, according to its definition, function $value$ is well-defined, that is, for each $o \in \mathcal{O}$ and each $c \in \mathcal{C}$ if o is a member of c then $value(o, c)$ is uniquely determined. Indeed, if o is a member of c, o is structurally consistent with respect to the classes to which it belongs, either as direct or indirect member.

Let $t^s(X)$ be the static type of an object reference X. We recall from Section 2 that this type is always unique. Let o be an object to which reference X is instantiated; therefore, $t^s(X)$ is the static type of object o in the current expression. Then, an attribute access $X.a$ is solved by simply returning $value(o, t^s(X)).a$.

Example 7 Consider the inheritance hierarchy of Example 5 and the object

$$\begin{aligned} o = (i_1, (&\texttt{name} : \texttt{'sarah'}, \texttt{address} : \texttt{'genova'}, \texttt{spouse} : i_2, \\ &\texttt{student-code}{:}643, \texttt{average_score} : 28, \texttt{year} : 2, \\ &\texttt{employee-code}{:}14453, \texttt{salary} : 1000, \texttt{role} : \texttt{'secretary'}), \\ &\{\ \texttt{student, employee}\ \}\ ) \end{aligned}$$

where i_1 and i_2 are object identifiers. Then,

$$\begin{aligned} value(o, \texttt{employee}) &= (\texttt{name} : \texttt{'sarah'}, \texttt{address} : \texttt{'genova'}, \texttt{spouse} : i_2, \\ &\qquad \texttt{code} : 14453, \texttt{salary} : 1000, \texttt{role} : \texttt{'secretary'}), \\ value(o, \texttt{student}) &= (\texttt{name} : \texttt{'sarah'}, \texttt{address} : \texttt{'genova'}, \texttt{spouse} : i_2, \\ &\qquad \texttt{code} : 643, \texttt{average_score} : 28, \texttt{year} : 2), \\ value(o, \texttt{person}) &= (\texttt{name} : \texttt{'sarah'}, \texttt{address} : \texttt{'genova'}, \texttt{spouse} : i_2). \end{aligned}$$

Consider now the attribute access expression $X.\texttt{code}$, where X is an object reference which at run-time is instantiated with o:

- if $t^s(X) = \texttt{person}$, an expression containing the attribute access $X.\texttt{code}$ is not well typed, and causes a type error to occur;
- if $t^s(X) = \texttt{employee}$, the attribute access $X.\texttt{code}$ is solved as $value(o, \texttt{employee}).\texttt{code}$ and returns the value 14453 (that is, $o \downarrow 2.\texttt{employee-code}$);
- if $t^s(X) = \texttt{student}$, the attribute access $X.\texttt{code}$ is solved as $value(o, \texttt{student}).\texttt{code}$ and returns the value 643 (that is, $o \downarrow 2.\texttt{student-code}$).

By contrast, consider the attribute access expression o.`spouse`. If $t^s(X)$ is `person`, `employee` or `student`, the value returned by the expression is i_2, since

$value(o, \texttt{person}).\texttt{spouse} = value(o, \texttt{employee}).\texttt{spouse} =$
$= value(o, \texttt{student}).\texttt{spouse} = i_2.$ $\diamond$

Finally, the following result holds, stating that each type correct attribute access $X.a$ is correctly handled by returning $value(o, t^s(X)).a$, in that: *i*) type checking is done with respect to the static type of the object reference X; thus, if $X.a$ is a type correct expression, then for each o member of $t^s(X)$ $value(o, t^s(X)).a$ is defined; *ii*) the value returned by $value(o, t^s(X)).a$ is a legal value for the static type of the expression $X.a$.

Proposition 4 *Let T be the static type of the attribute access $X.a$, and let o be an object to which reference X is instantiated, then $value(o, t^s(X)).a \in [\![T]\!]$.*

Note that the resolution of name conflicts for attributes is fully static, that is, it only makes use of information known at compile time, that is, the database schema and the static types of object references.

5 Methods and dispatching

In this section, we deal with the behavioral component of objects. As an initial remark, note that methods introduce more problems than attributes. An attribute is a stored value for which the only redefinition allowed is a domain refinement but, whatever class the object is seen an instance of, the stored value is the same. By contrast, each class in the hierarchy may define a different implementation for the same method, and 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 the database state.

According to one of the basic principles of object-orientation, if, due to subtype polymorphism [4], 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 we consider, which we call *argument specificity approach*, does not determine the preferred class of an object to solve a method invocation, rather it makes use of the other actual arguments in the method call, thus considering the method as a multi-method [1, 7, 10, 13].

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

5.1 Preferred class

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 assign a preferred class to 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. Consider for example an expression where an object reference has the static type `person`. If at run-time the reference denotes an object belonging both to class `noble` and to class `female`, the context of the object reference does not help in choosing the preferred class.

In those cases, we use the total order on classes introduced in Section 3.1. As seen, this order is 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 these orders, that may be considered too arbitrary and unpredictable by the user, are 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 these orders, 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 and of the total order on classes. The dynamic information consists of the set of most specific classes of the object; such classes are determined at run-time. The following rule formalizes the preferred class dispatching method. In what follows, $t^s(o)$ denotes the static type of object o, while $o \downarrow 3$ denotes the set of its most specific dynamic types. Moreover in what follows let $\mathcal{H}(o)$ denote the inheritance hierarchy to which object o belongs[16] and let $\preceq_{\mathcal{H}(o)}$ be the total order on the hierarchy to which o belongs, as defined in Section 3.1. Finally, given a set of classes S let $min_{\preceq_{\mathcal{H}(o)}} S$ denote a class which is the minimum among the elements in the set S with respect to the order $\preceq_{\mathcal{H}(o)}$. Since $\preceq_{\mathcal{H}(o)}$ is a total order on $o \downarrow 3$ and thus on $\{c \mid c \in o \downarrow 3 \wedge c \leq_{ISA} t^s(o)\} \subseteq o \downarrow 3$, $min_{\preceq_{\mathcal{H}(o)}}\{c \mid c \in o \downarrow 3 \wedge c \leq_{ISA} t^s(o)\}$ exists and is unique.

[16] Note that, since the extents of different hierarchies are disjoint, all the classes in $o \downarrow 3$ belong to the same hierarchy $\mathcal{H}(o)$.

Rule 3: Preferred class dispatching rule

Let $o.m(v_1, \ldots, v_n)$[17] be a method invocation. It is dispatched as follows:

- if $t^s(o) \in o \downarrow 3$, the implementation of m in $t^s(o)$ is selected,
- if $t^s(o) \notin o \downarrow 3$ the implementation of m in $min_{\preceq_{\mathcal{H}(o)}}\{c \mid c \in o \downarrow 3 \wedge c \leq_{ISA} t^s(o)\}$ is selected. □

If we want to privilege the most recently acquired behavior, instead of using the $\preceq_{\mathcal{H}(o)}$ order, we may consider the o acquisition order $\preceq_o$ of the classes in $o \downarrow 3$, where $c \preceq_o c'$ denotes that c has been acquired after c'.

Example 8 Consider the inheritance hierarchy of Figure 1 and let this hierarchy be denoted by $\mathcal{H}_1$. Suppose that for a unary method m, $m \in M(\texttt{noble})$, $m \in M(\texttt{male})$ and $m \in M(\texttt{female})$. Suppose moreover that the classes are related in the ISA-independent order $\sqsubseteq$ as follows

$\texttt{person} \sqsubseteq \texttt{male} \sqsubseteq \texttt{noble} \sqsubseteq \texttt{female}$

and then that the resulting total order $\preceq_1$ is

$\texttt{male} \preceq_1 \texttt{noble} \preceq_1 \texttt{female} \preceq_1 \texttt{person}$.

Consider now the method invocation $o.m(o')$ where $o \downarrow 3 = \{\texttt{noble}, \texttt{female}\}$, so that $\mathcal{H}(o) = \mathcal{H}_1$. Then:

- if $t^s(o) = \texttt{noble}$ the m implementation specified in class `noble` is selected;
- if $t^s(o) = \texttt{female}$ the m implementation specified in class `female` is selected;
- if $t^s(o) = \texttt{person}$ the m implementation specified in class `noble` is selected, since $\texttt{noble} \preceq_{\mathcal{H}(o)} \texttt{female}$. ◇

As illustrated by the previous example, 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 result values and of database 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. To formalize this result the notion of method "dispatchability" must be formalized first. We model dispatching of invocations on method m as a function that, given an object o (and some kind of information used in the dispatching) returns the class in the set of its most specific ones, to which the message $o.m(_)$ is dispatched. Note that the dispatching function is partial, since if $\nexists\ c \in o \downarrow 3$ such that $m \in M(c)$ then the message cannot be dispatched. Note however that the dispatching function is total on type correct messages. Indeed, if $o.m(_)$ is type correct, then a class in $o \downarrow 3$ certainly exists such that $m \in M(c)$; such class is either $t^s(o)$ or a subclass of $t^s(o)$.

Let $d^m_{pc} : \mathcal{O} \times \mathcal{T} \rightarrow \mathcal{C}$ denote the function associated with the preferred class dispatching rule, where the first argument is the method receiver and the second one is its context. The following result holds.

[17] We recall that this is the Chimera syntax for method invocation, where o denotes the privileged receiver of the method. Here we consider the actual method invocation, thus $o, v_1, \ldots, v_n$ are the actual values of the formal parameters $e, e_1, \ldots, e_n$ we have considered in Section 2.

Proposition 5 *Let $o.m(v_1, \ldots, v_n)$ be a method invocation. The function d^m_{pc} is defined for any pair $(o, t^s(o))$ such that $m \in M(t^s(o))$*[18].

Finally, note that if the dispatching strategy, based on the preferred class, is used with a contravariant redefinition rule for method arguments, type correctness is ensured.

5.2 Argument specificity

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 the actual parameters of the invocation (in addition to the type of the receiver object). This approach is close to multiple dispatching or multi-method approaches where the selection of the method to be executed 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 which method is executed and the other arguments only provide values but 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. Indeed, 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 it is contained.

We point out that in Chimera a contravariant redefinition rule is used for method input parameters. As recently discussed by Castagna [5], when using multiple dispatching the input parameter redefinition should be covariant for parameters involved in dispatching, whereas it should be contravariant for other parameters. However, the extension of the Chimera model to allow covariant input parameter refinement is currently under investigation. Note, moreover, that we use multiple dispatching only for choosing an implementation among the ones in sibling classes, and never to choose an implementation among the ones in a path in a given inheritance hierarchy.

In the following we consider multiple dispatching when objects may possibly have multiple most specific classes. Chimera methods, however, are not really multi-methods [10] 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 implementations in classes that are most specific for the receiver are considered as "candidates" for the dispatching. Thus, the dispatching we propose here is not purely multiple in that

[18] If the method invocation is type correct, then $m \in M(t^s(o))$ holds.

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.

Let $mst(v) = \{T \mid v \in [\![T]\!] \wedge \nexists\ T' \in \mathcal{T}$ such that $T' <_T T, v \in [\![T']\!]\}$ be the set of most specific types of a value. Note that if v is the identifier of an object o then $mst(v) = o \downarrow 3$; otherwise $mst(v)$ is a singleton set containing the (unique) most specific value type of v.

Definition 12 *(Method applicability). A method implementation*
$m : T_1^j \times \ldots \times T_n^j \rightarrow T_r^j$
defined in a class c_j *is applicable to a method invocation*
$o.m(v_1, \ldots, v_n)$
if $c_j \in o \downarrow 3$ *and* $\forall i, 1 \leq i \leq n, \exists T_i \in mst(v_i)$ *such that* $T_i \leq_T T_i^j$. □

Example 9 Consider the class hierarchy of Example 8 where $m \in M(\texttt{noble})$, $m \in M(\texttt{female})$, $m \in M(\texttt{person})$ and consider the following signatures for method m
$sign(m, \texttt{noble}) = \texttt{noble} \rightarrow \texttt{person}$
$sign(m, \texttt{female}) = \texttt{male} \rightarrow \texttt{person}$
$sign(m, \texttt{male}) = \texttt{person} \rightarrow \texttt{person}$.
Consider the method invocation $o.m(o')$

1. if $mst(o) = \{\texttt{noble}, \texttt{female}\}, mst(o') = \{\texttt{noble}, \texttt{male}\}$, then both the method in `noble` and the method in `female` are applicable;
2. if, by contrast, $mst(o) = \{\texttt{noble}, \texttt{male}\}$, $mst(o') = \{\texttt{female}\}$, only the method in `male` is applicable. ◇

To define the argument specificity dispatching rule a notion of *method specificity*, that is, an order $\prec$ on methods must be used. This order is exploited in choosing the method to be executed, among the applicable ones. The following rule formalizes the argument specificity dispatching method. For the sake of simplicity, we denote with m^{c_i} (m^{c_j}) method m in class c_i (c_j).

Rule 4: Argument specificity dispatching rule
Let $o.m(v_1, \ldots, v_n)$ be a method invocation, and let $\prec$ be the considered method specificity order. The method invocation is dispatched as follows.
Method m in class c is executed if
$m^c = min_{\prec}\{\ m^{c'} \mid c' \in o \downarrow 3 \wedge m^{c'}$ is applicable $\}$. □

The most specific among the applicable methods is thus executed.

We now discuss how the $\prec$ order can be defined. Consider two method implementations, in class c_i and in class c_j, such that
$sign(m, c_i) = T_1^i \times \ldots \times T_n^i \rightarrow T_r^i$
$sign(m, c_j) = T_1^j \times \ldots \times T_n^j \rightarrow T_r^j$.

A first possibility (which corresponds to the one adopted by Cecil [7] and to the *argument subtype precedence* in [1]) is to define m^{c_i} as more specific than m^{c_j} (denoted as $m^{c_i} \prec_{asp} m^{c_j}$) iff $\forall h, 1 \leq h \leq n, T_h^i \leq_T T_h^j$ and $\exists k, 1 \leq k \leq n, T_k^i <_T T_k^j$. However, if we consider this order on methods there are several ambiguous situations, in which we are not able to dispatch the message.

Example 10 Consider the methods of Example 9, then $m^{\texttt{noble}} \prec_{asp} m^{\texttt{male}}$ and $m^{\texttt{female}} \prec_{asp} m^{\texttt{male}}$ but $m^{\texttt{noble}} \not\prec_{asp} m^{\texttt{female}}$ and $m^{\texttt{female}} \not\prec_{asp} m^{\texttt{noble}}$. Thus, the method invocation in case 1. cannot be dispatched. $\diamond$

Example 11 Let A, B, C be three classes, such that $ISA(A) = C$ and $ISA(B) = C$. Suppose that $m \in M(A)$, $m \in M(B)$ where $sign(m, A) = B \times C \to C$ and $sign(m, B) = C \times A \to C$. The invocation $o.m(o_1, o_2)$ with $o \downarrow 3 = \{A, B\}$, $o_1 \downarrow 3 = \{B\}$, $o_2 \downarrow 3 = \{A\}$ cannot be dispatched, since $m^A \not\prec_{asp} m^B$ and $m^B \not\prec_{asp} m^A$. $\diamond$

The proposed order can then be refined by taking into account an ordering on arguments (for example, a left-to-right ordering). The obtained notion of method specificity, which corresponds to the *argument order precedence* in [1], is defined as follows: m^{c_i} is more specific than m^{c_j} (denoted as $m^{c_i} \prec_{aop} m^{c_j}$) iff $\exists k, 1 \leq k \leq n, T_k^i <_T T_k^j$ and $\forall h, 1 \leq h < k, T_h^i = T_h^j$. However, also considering this order on methods there are still ambiguous situations, in which we are not able to dispatch the message. Referring to Example 9, $m^{\texttt{noble}} \not\prec_{aop} m^{\texttt{female}}$ and $m^{\texttt{female}} \not\prec_{aop} m^{\texttt{noble}}$. Thus, in case 1. of Example 10, the message cannot be dispatched.

Example 12 Consider classes A, B and C such that $ISA(C) = \{A, B\}$ and two classes C_1 and C_2 such that $m \in M(C_1)$, $m \in M(C_2)$ with $sign(m, C_1) = A \to C_1$ and $sign(m, C_2) = B \to C_2$, invocation $o.m(o_1)$, where $o \downarrow 3 = \{C_1, C_2\}$, $o_1 \downarrow 3 = \{C\}$, cannot be dispatched since $m^{C_1} \not\prec_{aop} m^{C_2}$ and $m^{C_2} \not\prec_{aop} m^{C_1}$. $\diamond$

Then, we further refine the proposed order by taking into account an ordering on arguments (for example, a left-to-right ordering) and the total order $\prec_{\mathcal{H}(o)}$ on the classes in the hierarchy to which the method receiver belongs, in addition to the subtype relationship. The resulting notion of method specificity is formalized as follows.

Definition 13 *(Method specificity). Let m^{c_i} denote method m in class c_i and m^{c_j} denote method m in class c_j, with $sign(m, c_i) = T_1^i \times \ldots \times T_n^i \to T_r^i$ and $sign(m, c_j) = T_1^j \times \ldots \times T_n^j \to T_r^j$. m^{c_i} is more specific than m^{c_j} (denoted as $m^{c_i} \prec_t m^{c_j}$) iff $\exists k, 1 \leq k \leq n, T_k^i <_T T_k^j$ or $T_k^i \prec_{\mathcal{H}(o)} T_k^j$ and $\forall h, 1 \leq h < k$, $T_h^i = T_h^j$.* $\square$

Given the method specificity order, the only situations, in which messages cannot be dispatched are when the signatures of the methods in the two classes are the same.

Example 13 Consider the hierarchy of Example 9, with the total order on classes of Example 8. Since $\texttt{male} \prec_{\mathcal{H}(o)} \texttt{noble} \prec_{\mathcal{H}(o)} \texttt{female} \prec_{\mathcal{H}(o)} \texttt{person}$, $m^{\texttt{female}} \prec_t m^{\texttt{noble}}$ holds. Thus, the message in case 1. is dispatched, and the implementation for m defined in class `female` is executed. $\diamond$

We remark that neither by using an *inheritance order precedence* nor a *global type precedence* as in [1] we would be able to dispatch the method invocation of Example 9 in case 1. Moreover, under the argument specificity approach, not all

the type correct method invocations can be dispatched. Let $d^m_{as} : \mathcal{O} \times 2^{\mathcal{T}} \times \ldots \times 2^{\mathcal{T}} \rightarrow \mathcal{C}$ be the function associated with the argument specificity dispatching rule, where the first argument is the method receiver and the other ones are the most specific types of the invocation actual arguments. Then, the following result holds.

Proposition 6 *Let $o.m(v_1, \ldots, v_n)$ be a method invocation. Function d^m_{as} is defined on a $n+1$-tuple $(o, mst(v_1), \ldots, mst(v_n))$ if both the following conditions hold:*

- *$\exists c \in o \downarrow 3$ such that $m \in M(c)$ and m^c is applicable to $o.m(v_1, \ldots, v_n)$*[19]*;*
- *$\forall c_i, c_j$, $i \neq j$, such that both m^{c_i} and m^{c_j} are applicable to $o.m(v_1, \ldots, v_n)$ one of the following conditions holds:*
  - *if $sign(m, c_i) = T^i_1 \times \ldots \times T^i_n \rightarrow T^i_r$ and $sign(m, c_j) = T^j_1 \times \ldots \times T^j_n \rightarrow T^j_r$, $\exists k, 1 \leq k \leq n$, such that $T^i_k \neq T^j_k$;*
  - *$\exists c_k$ such that m^{c_k} is applicable to $o.m(v_1, \ldots, v_n)$ and $m^{c_k} \prec_t m^{c_j}$.*

Finally note that the argument specificity approach ensures the identity of behavior of a method invocation. Indeed, according to Rule 4, a given message $o.m(v_1, \ldots, v_n)$ is dispatched to a class which does not depend on the context $t^s(o)$ of the message receiver in the expression containing the invocation. The argument specificity approach, however, does not ensure type correctness.

6 Conclusions

In this paper we have investigated issues arising from multiple direct membership of objects to classes in the framework of the Chimera object-oriented data model. Both structural and behavioral aspects of objects have been considered. First, the state of an object has been characterized in terms of the set of its most specific classes. A (static) notion of context has been defined and it has been exploited for handling attribute accesses. Then, two different dispatching approaches have been proposed. The two approaches are compared in Table 1. The table summarizes the differences of the two approaches (discussed in detail in Section 5). In the table *applicability* denotes whether the approach is always able to dispatch a message.

	Information exploited	Applicability	Context-dependence	Behavior identity	Static typing
Preferred class	Static and Dynamic	Total	YES	NO	YES
Argument specificity	Dynamic	Partial	NO	YES	NO

Table 1. Comparison between the two proposed dispatching approaches.

Multiple inheritance and multiple class direct membership could seem overlapping design concepts. However, we think that multiple inheritance should

[19] If the method invocation is type correct, then this condition is verified.

be used whenever the obtained class is a semantically meaningful class, with a specific state and behavior (that is, in which new features are added or some features are redefined). By contrast, multiple class direct membership should be used to avoid defining artificial intersection classes and whenever we would like to allow a context dependent behavior.

Currently, only the preferred class approach has been implemented in the Chimera prototype developed at Politecnico di Milano. Indeed, efficient techniques for the argument specificity dispatching approach need to be further investigated. Some efficient techniques for multi-method handling (based on look-up automata) have been proposed [8], that could be extended to our case.

Another possible topic of future work is how to ensure type correctness with the argument specificity dispatching approach. The approach to static type checking for multi-method proposed in [1] could be extended to our case. However, in our approach the confusable methods are much more than in [1] in that an object can have an arbitrary set of types, and the relationship between the static type and the dynamic ones are only those stated by Rule 1.

Finally, we would like to point out that when different implementations for a method are defined in the most specific classes of an object, there are cases in which the most intuitive approach is that of executing both. Consider as an example an *init* method that initializes the class attributes. Problems due to different assignments to the same attribute may however arise. In Chimera, the only way to specify such a behavior is to define a subclass-by-multiple inheritance of the two classes and to redefine the method in the subclass as desired. However, we think that this issue need to be further investigated.

Acknowledgments The ideas presented in this paper have been greatly influenced by discussions with Stefano Ceri and René Bal. We also would like to thank the referees for their comments and suggestions.

References

1. R. Agrawal, L. DeMichiel, and B. 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.
2. 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.
3. E. Bertino and L.D. Martino. *Object-Oriented Database Systems - Concepts and Architecture*. Addison-Wesley, 1993.
4. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polimorphism. *Computing Surveys*, 17:471–522, 1985.
5. G. Castagna. Covariance and Contravariance: Conflict without a Cause. Technical Report LIENS-94-18, Département de Mathématiques et d'Informatique - Ecole Normale Supérieure, Paris, October 1994.
6. S. Ceri and R. Manthey. Chimera: A Model and Language for active DOOD Systems. In *Extending Information System Techology - Second International East-West Database Workshop*, Lecture Notes in Computer Science, pages 9–21, 1994.

7. C. Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proc. Sixth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 615, pages 33–56, 1992.
8. W. Chen, V. Turau, and W. Klas. Efficient Dynamic Look-Up Strategy for Multi-Methods. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 821, pages 408–431, 1994.
9. 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.
10. R. Gabriel, J. White, and D. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):28–38, September 1991.
11. G. Gottlob, M. Schrefl, and B. Rock. Extending Object-Oriented Systems with Roles. To appear in *ACM Transactions on Information Systems*, 1994.
12. G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. Technical Report IDEA.DE.2P.011.01, ESPRIT Project 6333, May 1994. Submitted for publication.
13. W. Mugridge, J. Hamer, and J. Hosking. Multi-Methods in a Statically Typed Programming Language. In P. America, editor, *Proc. Fifth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 512, pages 307–324, 1991.
14. J. Richardson and P. Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 298–307, 1991.
15. E. Sciore. Object Specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.
16. 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.