# QUALIFIED INHERITANCE IN SPATIO-TEMPORAL DATABASES

**Pier DONINI, Sophie MONTIES**
Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland
Computer Science Department, Database Laboratory
Pier.Donini@epfl.ch, Sophie.Monties@epfl.ch

**KEY WORDS:** Data models, Data structures, Database, Object-oriented, Spatio-temporal.

## ABSTRACT

In a growing number of applications, different complementary views of real-world objects are needed. This is particularly true in spatial modelling - where the designer can propose different geometries for a given spatial object - or, in temporal modelling - where different life cycles can be defined on an object depending on the adopted point of view. A solution to provide multiple representations on objects is to offer a model supporting multiple instantiation, i.e. allowing real world entities to be instantiated in several classes.
At the same time, object-oriented languages and object-oriented modelling have become common in computer science (C++ and Java for the former and UML (Rum., 97) for the latter). Hence, oriented-object properties as inheritance, polymorphism and dynamic binding are commonly used and their expressiveness quite naturally exploited. However, classical object-oriented models and languages only allow an implicit form of multiple instantiation among an inheritance hierarchy, i.e. an instance of a class is also a member of all its super-classes.
In this paper we propose a solution to integrate classical object oriented mechanisms into models handling multiple instantiation and illustrate it on the conceptual model MADS. To achieve this, the dynamic binding mechanism was revisited, through the concept of *scope* of an instance (awareness of other instances), in order to manage the ambiguities induced by multiple instantiation. Last, providing an operational solution, a set of operators to modify the *point of view* of an object (determining which of its class instances is considered) and its *scope* is defined.

## 1 INTRODUCTION

Conceptual models that put a strong emphasis on spatiality or temporality often use particular underlying attributes which are used in objects in order to represent their spatiality or the temporality (for instance in the spatial models MODUL-R (MOD) or CONGOO (CON)).
This is the case in the conceptual model MADS (acronym for "Modelling of Application Data with Spatio-temporal features") which describes the shape and location of a spatial object through the predefined geometry attribute. Orthogonally, MADS models the temporal characteristics of an object by a life-cycle attribute. Since MADS is an object (and relationship) model, it must be clearly defined what behaviour is adopted for these attributes in a hierarchy of object types. For an attribute that has already been defined in a class (object type) its redeclaration in a subclass (e.g. redeclaration of the geometry of a Building in a Church) may have different semantics:

- *Overloading*: the newly defined attribute only shares the same name and bears no other relation to the original attribute. As will be discussed later, the overloading is done by hiding the inherited attribute and then by simply declaring the new one.
- *Redefinition*: the domain of the new attribute is redefined and a new value is stored in the subclass. We will see that in order to benefit of a form of dynamic binding on this attribute, the new domain must be a subtype of the corresponding one in the superclass.
- *Refinement*: the same value is shared in the subclass and in the superclass. As above, its domain is constrained.

It is to be noted that even though predefined temporal and spatial attributes reveal the necessity of explicitly specifying the behaviour of their redeclaration, such an explicit declaration is also needed whenever the same attribute identifier is used in a class and reused in one of its subclasses.

In order to distinguish the consultation of an attribute value from its creation or modification, a simple way is to encapsulate it through two *accessor* and *modifier* methods. These methods (whose implementation is system defined) own the same name than the one of the attribute they refer to and differ by their signature (*accessors* only have a return type while *modifiers* only accept an input type). Also, by overloading these methods every time an attribute is overloaded, redefined or refined (instead of accessing directly to the attribute) it is then possible to benefit from the dynamic binding properties of object-oriented languages and thus dynamically provide access to the most specific value of a given attribute (e.g. when scanning through the occurrences of a class Building which have an imprecise geometry, it is useful to directly

access, by dynamic binding, to their more precise geometry when they also are occurrences of the subclass Church).
However, in order to provide complementary views of real-world objects, MADS and any oriented-object model supporting multiple instantiation (i.e. that allow at least subclasses to share objects without forcing the designer to add a new class which is their intersection), no longer have, for each object, an unique most specialised class in which it is instantiated. Since an object can then be instantiated in several classes of the inheritance hierarchy (provided that they form a connex subgraph containing the topmost class[es]), the dynamic binding mechanism that exists in classical object-oriented languages needs to be revisited. Otherwise ambiguities may exist whenever a method that is declared in a class is overloaded in two (or more) of its direct subclasses (e.g. when the method `Display()`, existing for the superclass Building and its two subclasses Church and Historical Monument, is invoked for an instance of Building that also plays a role in the two subclasses, it is undecidable which implementation has to be provided).
Also, in order to offer a maximal flexibility while expressing queries (e.g. for MADS DML), a comprehensive mechanism to access specific properties of a class, thus breaking the dynamic binding mechanism, must be provided (e.g., in the above example, to always access to the Building geometry). Users may also need to be able to dynamically switch from one representation of a real-world object to another available representation. This modification of their *point of view* on the real-world object translates by considering a new instance of this object in another class.

The operators proposed by modern languages like C++ or Java to break the dynamic binding mechanism lead to confusing results (e.g. a call to a particular method m of a superclass in C++ via the `::` operator doesn't break the dynamic binding mechanism for the methods invoked within m) and are not adapted to a database model supporting multiple instantiation. Thus, in order to modify the dynamic binding mechanism in a multiple instantiation context and still have the possibility to break it to access a particular method, we introduce the concept of object *scope* (which is the set of classes this object knows to be instance of). We then propose a set of operator primitives to manipulate the *point of view* and the *scope* of an object. These operators allow to, 1) dynamically change the *point of view* for an object from a class to another class, 2) to extend the scope of an object 3) to restrict the *scope* of an object, and, 4) for an object o, to search among a list of classes the first class member of the *scope* of object o. The latter is particularly needed to avoid dynamic binding runtime errors that can arise when a method is defined in a class and overloaded in more than one of its direct subclasses.

In section 2 we briefly present the MADS conceptual model. The refined inheritance mechanism is presented in section 3 and its application in a multiple instantiation context is addressed is section 4.

## 2 THE MADS CONCEPTUAL MODEL

The MADS model (Par., 98), that will hereafter illustrate our inheritance mechanisms, uses an extended object-relationship formalism that allows to orthogonally model classical, spatial and temporal data.

For modelling classical data, MADS offers the following concepts:

- **Global Object**. A global object represents a concrete or abstract entity of the real-world where all its playable roles are considered (e.g. Joe, where Joe is a person, a sportsman, an employee, etc.).
- **Object Type** (or Object Class). An object type represents a set of global objects perceived within a particular context, exhibiting the same structure and behaviour (e.g. Person). An object (or occurrence) is the materialization of a global object into an object type (e.g. Joe as a Person).
- **Inheritance Hierarchy and *Maybe* Links**. In order to allow multiple representations for a real-world object, object-types can be organized into an inheritance hierarchy or can be put in correspondence through *maybe* links. Classically, an oriented inheritance link between a supertype and a subtype can exist only if the population of the former belongs to the population of the latter. An unoriented *maybe* link between two object types expresses that their populations are not necessarily disjoint. This implies that two object types that are not *maybe*-linked and are not part of the same inheritance hierarchy cannot have a common population.
- **Relationship Type** (or Relationship Class). A relationship type represents the association that may exist between two or more object types. A relationship is the instantiation of a relationship type and links one object of each participating object type.
- **Property**. Static (attributes) and dynamic (methods) properties can be associated to object and relationship types. Attributes can be simple (with atomic values), complex (structured, composed of simple or complex attributes) or derived (whose values are computed through a derivation formula from other attributes' values). Attributes must have a minimum and maximum cardinality specifying the number of possible values. Methods are classically defined through their signature and their implementation.

Briefly, the main spatial and temporal characteristics of the MADS model lie in the following concepts:

- **Spatial Types**. A set of spatial abstract types organised in an inheritance hierarchy is provided. At each abstract spa-

tial type are attached some specific manipulation methods. MADS allows to attach a spatiality to object types and to attributes. A spatial attribute is an attribute whose domain is one of the spatial abstract types. The spatiality of an object type is described by a predefined spatial attribute, geometry.

- **Temporal Object Types**. The temporality of a temporal object type is described by a predefined life-cycle attribute. Its values allow to keep track of the evolution of a temporal object within its type.

MADS also offers other concepts (as aggregations, topological and temporal relationship types, space- and time- varying attributes) that are not be described in this paper. See (Par., 98) for further informations.

## 3 REDECLARING PROPERTIES THROUGH INHERITANCE

### 3.1 Subtyping and Inheritance

The concept of inheritance link of oriented-object models allows to refine an object type (or class) into a more precise object type. The semantics of this link expresses that a subset of the real-world objects described by the generic object type also belongs to the specialised object type (i.e. there is an inclusion of population between the global objects of the specialised object type and those of the generic one). Also, it is often interesting to characterise the different abstraction level provided by the specialised object type by defining some particular properties.
During the manipulations, MADS, and most oriented-object models and languages, follows the principle of substitutability issued from the *inclusion polymorphism* defined in (Car., 85): an occurrence of the specialised object type can be used whenever an occurrence of the generic object type is required. This means that, for a global object, all the roles (properties and links) existing in its instantiation as an occurrence of the generic object type must exist in its instantiation as an occurrence of the specialised type. Or, rather that the specialised object type is a *subtype* of the generic one.

### 3.2 Dynamic binding

Oriented-object languages allow the subtype to declare some new version of the methods existing in the supertype, thus overriding the inherited methods, and resolve the method invocation with the mechanism of dynamic binding. When an object type T defines a method m, and a subtype T' of T redeclares it, it is not known statically which definition of m will be invoked by the occurrences of T. At run-time, if an occurrence of T is also member of T' (i.e. if it refers at a global object that is also instantiated in T'), the definition of m in T' will be invoked. Otherwise, the definition of m in T is used.
This form of *polymorphism*, referred by (Pla., 98) as *inheritance polymorphism*, a subdivision of the *ad-hoc polymorphism* defined in (Car., 85), allows new object types to be added to an application without affecting existing designs (if the newly defined subtype needs a specific version of an existing method, it is only defined within this subtype) and allows a reduction in programming complexity by replacing switch statements with simple calls. For instance, in graphical toolkits, the method `Draw()` defined on the object type `Window` is overridden its subtypes `Button` or `ListBox`.
Some oriented-object languages (e.g. Eiffel) allow the signature of the redeclared method to be different from the one in the supertype. However, to be fully compatible with the dynamic binding mechanism the signature proposed in the subtype must always be coherent with the one proposed in the supertype. The redeclared signature must then be covariant for its result type (i.e. subtype of the result type of supertype's method) and contravariant for its input types (i.e. supertypes of the input types of supertype's method). Since contravariance for the input types does not result in an expressively gain, some languages (e.g. Eiffel or O2) seek for covariance for both input and result types. However, this is potentially dangerous since it can lead to run-time errors (e.g. when the redefined method receives a parameter whose type is a different subtype than the one expected). It is said that using covariance for input types makes the signature of the method in the supertype *lie* about its truly accepted parameters in the subtype(s). MADS thus prefers to use covariance for the result type and *invariance* (same type) for the input types.

### 3.3 Attributes

Describing the temporality or the spatiality of objects through a predefined attribute (e.g. life-cycle and geometry for MADS) requires to clearly specify what are the available semantics when such an attribute defined in a class c is redeclared in a subclass of c (e.g. the geometry of a Building can be declared as a point and be redeclared as an area in a Church subclass). This redeclaration of spatial or temporal predefined attributes is a particular case of the generic case where a subclass defines an attribute with the same name than one defined in its superclasses.

**3.3.1 Overloading, refinement and redefinition.** Classically (e.g. C++), for non predefined attributes, the newly defined attribute in the subclass only shares the same name but is otherwise completely distinct from the one in the superclass. As long no dynamic binding mechanism on attributes is defined in the model, there are no potential ambigui-

ties while accessing to an attribute value. Attributes are accessible through the classes where they are defined. For an instance of a class c' where an attribute x declared in a superclass c is redeclared, the value of x in c' is directly accessible, and the value of x in c, overloaded in c', is referred by using its fully qualified class name (e.g. `o.c::x` in C++).
However, this overloading behaviour is not suited in all modelling contexts, and especially not for predefined temporal or spatial attributes. The designer might wish to express the fact that the superclass and the subclass attributes correspond to the same real world property seen at different abstraction levels, where the subclass attribute provides a more precise representation than the one defined in the superclass (as above for the geometries of the Building and the Church). Thus, orthogonally to the redefinition of methods in subclasses, it is interesting to define a form of dynamic binding on attributes, i.e., to automatically provide access, at run time, to the most precise definition (and hence its value) of a referred object attribute.

So, in addition to the overload of attributes, MADS accepts two kind of redeclaration behaviour linking the newly declared attribute in the subclass to the original one in the superclass: the *refinement* and *redefinition* of attributes. In *refinement* the same attribute values are shared for global objects instantiated in the subclass and in the superclass, while in *redefinition*, distinct values - one for each attribute - are stored.

**3.3.2 Accessors and modifiers**. In order to benefit from the existing literature on dynamic binding mechanism on methods and to cleanly distinguish the consultation of an attribute value from its modification, the access to attributes is only provided through their encapsulation by *accessors* and *modifiers* methods. These methods own the same name than the attribute they refer at, and are automatically provided by the system every time a new attribute is declared. *Accessors* only have a return type while *modifiers* only accept an input type.

Let the definition of an attribute a of type t (be it simple or complex) and of cardinalities [min, max] (i.e., a value of the attribute a consists in a set of values, whose cardinality is in [min, max], of type t). The system then automatically defines, in pseudo-code:

- *a*'s *accessor*: signature, `a() : [min, max] t`, body: `return a.copy()`,
- *a*'s *modifier*: signature, `a(value [min, max] t)`, body: `a = value`.

For the end-user, manipulating accessors and modifiers instead of directly attributes consists in just a syntactic difference. However, their use also removes the need to directly explore the dynamic binding mechanism on attributes and lessens the learning curve by staying within a well-known context of methods redefinition.

Introducing *accessors* and *modifiers* in the model also requires a set of constraints on *refinement* and *redefinition* of attributes. As above, let a class c define an attribute a of type *t* of cardinalities [min, max], and let a subclass c' of c redefine or refine it in type t' with [min', max'] cardinalities. Two accessors are then automatically defined for each attribute, `a(): [min, max] t` in c, and `a(): [min', max'] t'` in c'.
Since, in MADS, methods are covariant for their return types, this implies that to be compatible with the dynamic binding mechanism on accessors, t' must be a subtype of t (or be t itself).
Thus, in order to use the concepts of *redefinition* and *refinement* for spatial attributes (and in particular for the geometry predefined attribute), it is necessary to provide, like in MADS, a hierarchy of spatial abstract types (e.g. a spatial attribute of type line can be refined in oriented line but not in area). Cardinalities follow a similar constraint: since the accessor `a()` in c is expected to return at least min and up to max values of type t, the accessor of `a()` in c', that can be invoked through dynamic binding in the place of the one in c, cannot return less than min or more than max values of type t'; i.e., $0 \leq min \leq min' \leq max' \leq max \leq n$.

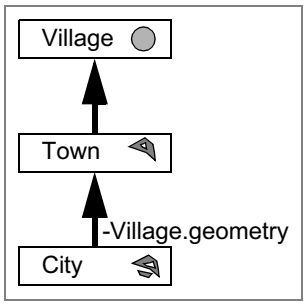
Note: MADS, chooses to cancel the dynamic binding mechanism on modifier methods in order to have a globally comprehensive setup where an attribute has its values always set within the class where it has been declared.

**3.3.3 Method hiding.** The usage of accessors also requires to block the dynamic binding mechanism on accessors when attributes are overloaded. If it wasn't, and a class C defines an attribute a of type t and a subclass of c, c', overloads it by declaring a new attribute a of type t', subtype of t, for instances of c that also are instances of c', a call to the accessor `a()` of c would invoke the one in c'. Since this confusing behaviour is not the one expected in overloading and that it would only happen in the cases where the type of the overloading attribute is subtype of the type of the original one, MADS allows overloading by requiring to hide the accessor of the inherited attribute before defining a new attribute with the same name (and hence its accessor). The dynamic binding mechanism defined in MADS (see section 4 for its algorithm) does not attempt to find a refined version of an invoked method in a subtype if it has not been inherited in it (i.e., if the supertype's method has been hidden within the subtype), even if this subtype defines a new version of the method.

Using hiding extends the definition of subtyping since a property (an accessor) that is available in the supertype is not accessible within the subtype. Although, even if it is not possible to access to the hidden property from the point of view of the subtype, it is always possible to access it from the point of view of the supertype (see section 4 for changing point of views on a manipulated object).

Example: A Village has a simple geo spatiality, that is refined as an area for a Town. City hides the inherited geometry and defines a new one as a complex area in order to also display the districts. Let v ∈ Village, t ∈ Town, c ∈ City.

- `v.geometry()` invokes Village's geometry accessor and returns a simple geo,
- `v = t, v.geometry()` invokes Town's geometry accessor (by dynamic binding) and returns a simple area.
- `v = c, v.geometry()` also invokes Town's geometry accessor (most specific, non hidden, redefinition of Village's one).
- `c.geometry()` invokes City's geometry accessor (or, by dynamic binding, a redefinition of it in City's subclasses) and returns a complex area.




### 3.4 Multiple Inheritance

In multiple inheritance different cases have to be considered when an attribute existing in the superclasses is inherited in the subclass.

- Classically, if the attribute is not redeclared within the subclass, the many inherited definitions coexist within the subclass. Since these can be of the same type and cardinalities (specially in a spatial context), the signature of their accessors is generally not sufficient to unambiguously refer to a given inherited attribute. Thus, in MADS it is necessary to change the *point of view* of the manipulated object to the specified superclass (see section 4 for the manipulation operators) and invoke the chosen accessor from there.
- Obviously when the attribute exists only in one of the superclasses, the redeclaration constraints are exactly the same as for single inheritance (i.e. subtyping and cardinalities inclusion for refinement and redefinition). The same happens when all but one inherited attributes are hidden within the subclass and thus the redeclaration of the attribute has to comply with its only one earlier definition in a superclass.
- Last, many attribute's definitions can exist in superclasses and be visible within the subclass where the attribute is redeclared. In this case, all the redeclaration constraints that exist for each attribute definition have to simultaneously exist. Thus, for refinement and redefinition this implies that the type of the redeclared attribute has to be subtype of all the corresponding inherited attribute types and that its cardinalities have to be included in all the corresponding inherited attribute cardinalities.

In order to invoke a specific superclass method classical object oriented languages use the full qualified name of the targeted method (e.g. in C++, Class-name::Method-name). However, the dynamic binding mechanism is still applied for methods invoked within the body of the targeted method. We believe that this targeting mechanism's behaviour is confusing and/or not flexible enough (e.g. it isn't used in Java). For MADS we thus prefer to require a clear specification of the method invocation context by changing the type of the manipulated object to the supertype (changing *point of view)* and possibly by restricting its visibility on the other classes of the inheritance hierarchy (object *scope* modification).

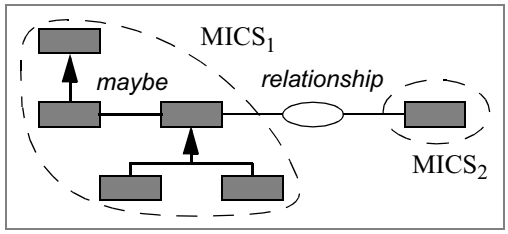
## 4 MULTIPLE INSTANTIATION

The framework described in the previous section would be sufficient if the model didn't provide multiple representations of real-world objects. Unlike classical object oriented models, MADS allows a real world entity (hereafter referred as a *global object* within the database context) to be instantiated in several classes not necessarily belonging to the same inheritance hierarchy, and even though, not necessarily in an unique most specialised class of an inheritance hierarchy.

### 4.1 Instantiation consistency rules

The multiple instantiation consistency rules used in MADS require some preliminary definitions:

- Let C be the set of classes of a given MADS schema.
- **Unoriented instantiation path**
  It exists an unoriented instantiation path between two classes $c_1$ and $c_2$ of C, noted as $c_1 \sim c_2$, iff: $c_1 = c_2$, or $c_1$ is directly maybe-linked with $c_2$, or $c_1$ is a direct superclass or subclass of $c_2$, or $\exists\ c \in C$ such that $c_1 \sim c$ and $c \sim c_2$.
- **Multiple Instantiation Class Sets** (MICS)
  The set of the classes (C) of a given MADS schema can be partitioned in different (n) MIC-sets
  $MICS(C) = \{ MICS_1, MICS_2 \ldots MICS_n \}$,
  where $MICS_i = \{ c_{i1}, c_{i2} \ldots c_{ik} \}$, $c_{ij} \in C$.
  The $MICS_i$ are built as follows:



$\forall$ i, $\forall$ $c_1$, $c_2 \in$ $MICS_i$, $\Rightarrow$ $c_1 \sim c_2$. $\forall$ i, j, i $\neq$ j, $\forall$ $c_1 \in$ $MICS_i$, $\forall$ $c_2 \in$ $MICS_j$, $\Rightarrow \neg(c_1 \sim c_2)$.

$\forall$ c $\in$ C, $\exists$ i such that c $\in$ $MICS_i$. $\forall$ i, $MICS_i \neq \emptyset$.

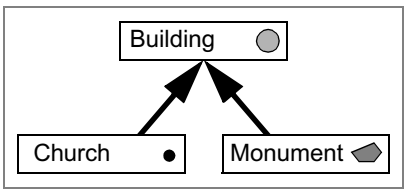
Definition: The (unique) MIC-set containing a given class is also called the *scope* of this class.
Formally, $\forall$ c $\in$ C, $\exists$ i such that c $\in$ $MICS_i$, then scope(c) = $MICS_i$.

Multiple instantiation consistency rules
In MADS, a *global object*, can only be instantiated in the classes of one, and only one, MIC-set. Moreover, if a *global object* is instantiated in a subclass c, it also has to be instantiated in all the superclasses of c.
This partitioning of the classes of a schema in MIC-sets corresponds to the fact that a *global object* o instantiated in a class c could be instantiated in the other classes of the MIC-set containing c (i.e. in scope(c)'s classes) but not elsewhere. For instance, a *global object* corresponding to the real-world entity Joe can be materialized in the maybe-linked classes Employee and Sportsman and in a Manager subclass of Employee but not in an unrelated class Car.
Note: In models not supporting multiple instantiation, all the classes of a MIC-set would have to be merged in one class.

### 4.2 Multiple specialisations

The rules defined in the previous paragraph ensure the static consistency between *global objects* and their instances in a MADS database, but are not sufficient to unambiguously determine the dynamic behaviour of methods in multiple specialisations contexts.
Since in MADS a *global object* can be instantiated in many classes of an inheritance hierarchy, it may arise that a *global object* o is instantiated in a given class c and is also instantiated in the different direct subclasses of c (i.e. providing multiple specialisation). Also, a method m might be declared in the superclass and being overloaded in these subclasses. In this case, when the method m is invoked on this *global object* o within the superclass, it is impossible to determine through classical dynamic binding mechanism which implementation of the method m has to be provided.

Example: the consultation of a Building's geometry through the invocation of its accessor is ambiguous for an object which is at the same time a Church and a Monument since three geometries can be returned: the original Building's geosimple geometry, and the point or area geometries redefined (or refined) in the subclasses.




We believe that the solution to abort the dynamic binding mechanism where an ambiguity of definitions occurs is often too restrictive. Therefore, the dynamic binding mechanism has been refined by allowing the manipulation of *inheritance paths*, MICS connex subgraphs, along which it takes effect. For instance, above, it must be possible to statically decide that a Monument's geometry will be returned in priority to the Church's one for objects belonging to the both classes.

### 4.3 Method invocation

In order to modify the dynamic binding mechanism to support multiple instantiation, it is necessary to introduce the concepts of *global object* and instance (or object) *scope*.

- The *scope* of a *global object* is the set of classes of the database schema in which it is instantiated. Since, by definition as seen earlier, the instances of a *global object* can only occur within a define MIC-set, the *scope* of a *global object* is a connex subgraph of a MIC-set containing its topmost classes.
- The *scope* of an instance is a set of classes representing the local (and modifiable) awareness that has a given instance of the other classes in which the *global object* it refers at is also instantiated. Obviously, the *scope* of an instance is always a connex subgraph of its *global object scope* containing not only its topmost classes but also the instance's class. Also, prior to any modifications, the *scope* of an instance is equal to the *scope* of its *global object*.
  Thus, a MADS instance i is the materialization of a *global object* into a specific class (the *global object point of view*) with a definite visibility on other classes, i.e. i = (global-object, class, instance-scope).

Then, the dynamic invocation of a method m on an instance i = (g, c, s) consists into a three steps algorithm:

1. finding the unique most specialized definition of the method m among the superclasses of c.
2. searching the unique most specialized overloading definition of the method m through the subclasses of c where the *global object* g is also instantiated and that belong to the scope of the instance i.
3. applying the most specialized method retrieved from points 1 and 2 (or returning an error in case of ambiguity).

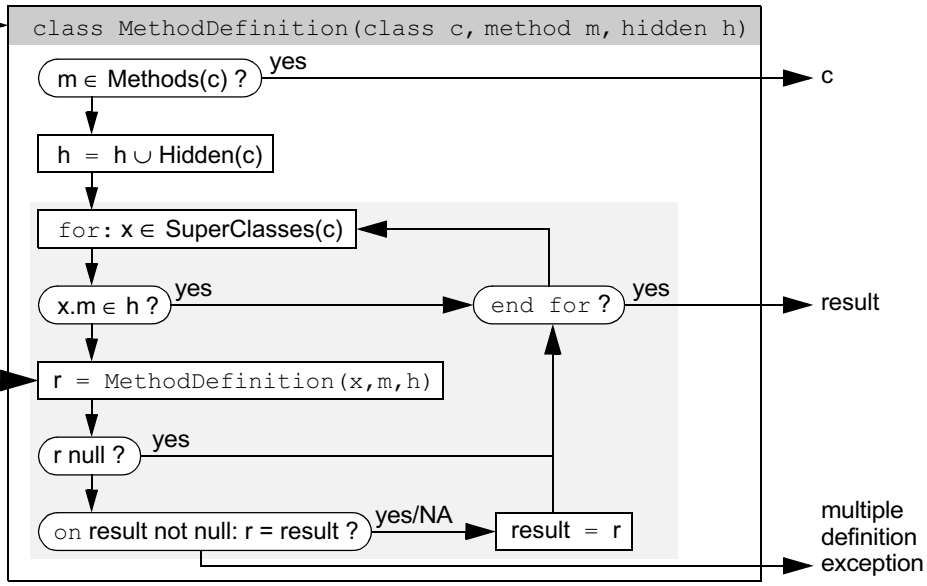
In the example of section 4.2, restricting the scope of the instances of a Building to the Building and Monument classes allows to benefit from this dynamic binding mechanism along one branch of the inheritance hierarchy on the geometries' accessors. Then the invocation of the geometry accessor from the Building point of view returns a geosimple geometry for non-Monuments (only Buildings or Churches) or an area geometry for Monuments.

To be complete, the model also provides a mechanism to define a priority order between the different specialisations of a given class. In order to achieve this, MADS allows the dynamic modification of the point of view on the manipulated global object among an ordered list of classes where it is instantiated.

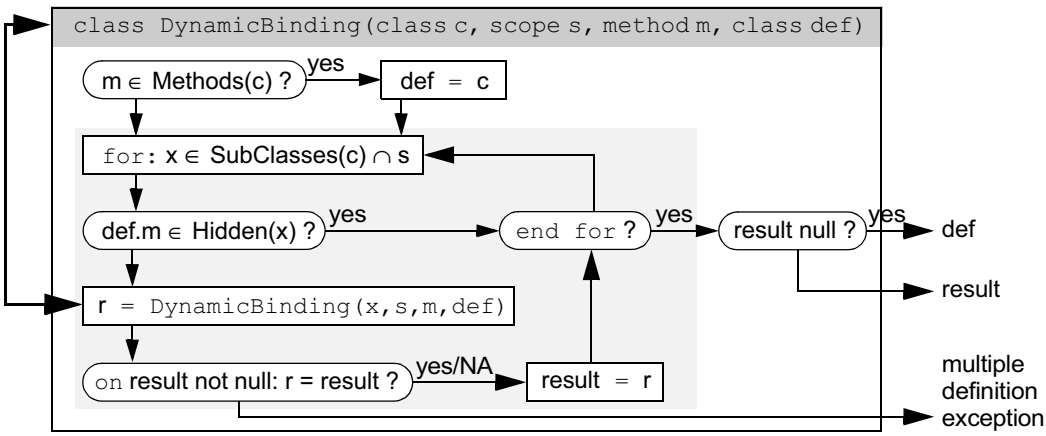
Below are described the details of the MADS dynamic binding mechanism. The following primitives are used:

- Methods(class c) -> { method } returns the set of methods defined in class c.
- SuperClasses(class c) -> { class } returns the set of the direct superclasses of class c.
- SubClasses(class c) -> { class } returns the list of the direct subclasses of class c.
- Hidden(class c) -> { class.method } returns the list of the methods hidden within class c.

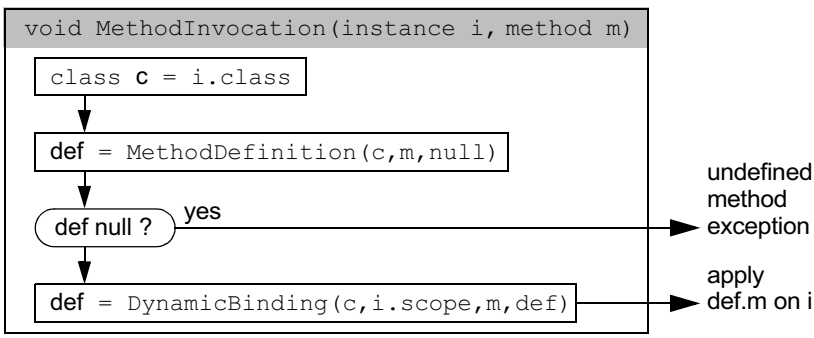
**Method Definition**:
Returns the unique most specialized definition of the method m among the superclasses of class c. An exception is raised if no definition is found, or if different inheritance paths lead to different definitions of m.
Note: the 'm ∈ Methods(c)' first test of the algorithm must check for compatible signatures (covariance for output types).




**Dynamic Binding**:
Returns the unique most specialized overloading definition of the method m through the subclasses of class c belonging to the scope s. An exception is raised if different inheritance paths eventually lead to different final definitions of m.




**Method Invocation:**
Using the above algorithms, invokes a method m on an instance i by applying its most specialized definition accessible within i's scope.
Note: the `DynamicBinding` algorithm always returns a non null class where m is defined (if no subclasses of i's class overload m, the last known class defining m is always returned).



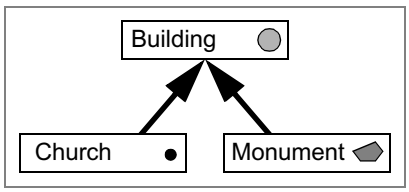

### 4.4 Operator primitives

The following operators will be used as the underlying primitives of the MADS algebra to manipulate the *point of view* and the *scope* of objects. Their purpose is to be able to access any property in the classes where a given *global object* is instantiated, possibly restricting the default dynamic binding mechanism.

In definitions below, let C be the set of classes, O be the set of global objects, I the set of classes' instances, and S the set of instances' scopes. Unless otherwise specified, let i = (o, c, s) where $i \in I$, $o \in O$, $c \in C$, and $s \in S$. Let c+ the set of classes containing a class c and all its superclasses, and let c- the set of classes containing a class c and all its subclasses.

- **GetInstance**. Constructs a *global object* class' instance whose *scope* is maximal (i.e. the *global object*'s *scope*).
  GetInstance: $O \times C \rightarrow I$; $(o, c) \rightarrow i = (o, c, scope(o))$.
- **SetViewPoint**. Changes the *point of view* on a global object from a class c as instance i, to a class c* as instance i*, provided that the class c* is in instance i's *scope*. Instance i*'s *scope* is then the same than instance i's.
  SetViewPoint: $I \times C \rightarrow I$; $(i, c^*) \rightarrow i^*$ [ if $c^* \in s$, $i^* = (o, c^*, s)$, otherwise $i^* = null$ ].
- **ScopeExtension**. Extends the *scope* of an instance by inserting a class (and its superclasses), provided that this class exists in the *scope* of the instance's *global object*.
  ScopeExtension: $I \times C \rightarrow I$; $(i, c^*) \rightarrow i^*$ [ if $c^* \in scope(o)$, $i^* = (o, c, s \cup c^{*+})$, otherwise $i^* = null$ ].
- **ScopeRestriction**. Removes a class (and its subclasses) from an instance's scope, provided that this set of classes does not contain the instance's class nor any of its superclasses (to always ensure a bottom-up full visibility).
  ScopeRestriction: $I \times C \rightarrow I$; $(i, c^*) \rightarrow i^*$ [ if $c^{*-} \cap c+ \neq \emptyset$, $i^* = (o, c, s - c^{*-})$, otherwise $i^* = null$ ].
- **ClassSelection**. Recursively searches among a list of classes the first one that is member of an instance's scope.
  Let LC the set of list of classes, $l = <c_1, c_2 \ldots c_n>$ where $l \in LC$ and $c_1, c_2 \ldots c_n \in C$.
  ClassSelection: $I \times LC \rightarrow C$; $(i, l) \rightarrow c^*$ [ if $l = \emptyset$, $c^* = null$, if $c_1 \in s$, $c^* = c_1$, otherwise, $c^* = ClassSelection(i, <c_2 \ldots c_n>)$ ].

Example: Let o be a *global object* that is at least instantiated in the Building class. a = GetInstance(o, Building). Calling a.geometry() might lead to dynamic binding errors (if o is also instantiated in Church and Monument). Two possibilities:




- Get one of the most specific geometries;
  b = SetViewPoint(a, ClassSelection(a, <Monument, Church, Building>)),
- Or, always retrieve the Building's geometry by restricting the dynamic binding mechanism; b = ScopeRestriction(ScopeRestriction(i, Church), Monument),

And then safely invoke b.geometry().

## 5 CONCLUSION

In this paper we have proposed a solution to embed the classical dynamic binding mechanism of object oriented languages into database models supporting multiple instantiation. In order to achieve this, a set of operator primitives allowing to manipulate instance's *point of view* and *scope* has been provided. Through accessor and modifier methods, this extended mechanism supports the concepts of *refinement*, *redefinition* and *overloading* of attributes, and is thus suited for temporal and spatial models that use a predefined attribute to describe the temporality or the spatiality of objects.
We plan to implement these concepts in the future MADS' DML engine currently developed in our lab.

## REFERENCES

Cardelli L., Wegner P., 1985. On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys, Vol. 17, n° 4.

CON. Pantazis D., Donnay J-P., 1996. La conception de SIG - Méthode et formalisme, Hermes.

C++. Stroustrup B.. The C++ Programming Language (3rd edition), ISBN 0-201-88954-4.

Eiffel. Meyer B., 1992. Eiffel: The Language, Prentice Hall, Englewood Cliffs, NJ.

Java. Sun Microsystems. http://java.sun.com/ (1995-2000).

MOD. Bédard Y., Caron C., Maamar Z., Moulin B., Vallière D., 1996. Adapting data models for the design of spatio-temporal databases, Computing, environment and urban systems, Vol. 20, n° 1, p. 19-41.

O2 Technology Reference Manual, Ardent Software.

Parent C., Spaccapietra S., Zimanyi E., Donini P., Plazanet P., Vangenot C., July 1998. Modeling Spatial Data in the MADS Conceptual Model, SDH'98, Vancouver, Canada.

Plaindoux D., Bodeveix J-P., Percebois C., 1998. Types versus classes, Revue L´Objet, Vol. 4, n° 1.

Rumbaugh J., Booch G., Jacobson I., 1996. Unified Modeling Language Reference Manual, Addison-Wesley Object Technology Series.