Integrating a Part Relationship into an Open OODB System using Metaclasses*

Michael Halper
Dept. of Math & Computer Science
Kean College of New Jersey
Union, NJ 07083 USA

James Geller, Yehoshua Perl
CIS Dept. & CMS
NJIT
Newark, NJ 07102 USA

Wolfgang Klas
GMD-IPSI
Dolivostr. 15
D-64293 Darmstadt, Ger.

Abstract

The part-whole semantic relationship (the part relationship, for short) is an important modeling primitive in many advanced application domains such as manufacturing, design, and document processing. In this paper, we examine the problem of integrating such a construct into an OODB system. Specifically, two questions are addressed in this regard. The first is: Can a part relationship be made an intrinsic construct of an existing OODB system without having to rewrite a substantial portion of the system? The second: Can an "open" OODB system which claims to support such an integration really do so, and, more specifically, can the integration be done using a metaclass mechanism which purports to bring extensibility to the VODAK Model Language (VML)? To demonstrate that both questions can be answered "yes," we introduce and discuss the details of a custom VML metaclass—the "HolonymicMeronymic" metaclass—which we have built. This metaclass comprises two items, an "instance" type and an "instance-instance" type. Together, the two endow the classes of a part hierarchy and their instances with structure and behavior consistent with our comprehensive part relationship model and the notions of "part" and "whole." Complete descriptions of each of these two aspects of the metaclass are presented and their effect on schema construction and database usage is discussed.

1 Introduction

The part-whole semantic relationship (the part relationship, for short) is an important modeling primitive in many advanced modeling domains such as manufacturing, design, document processing, and so on. As such, it has been utilized in some object-oriented database (OODB) systems [13, 20, 22] and in knowledge and reasoning systems [2, 3, 12, 24]. In our previous work [6, 7, 8, 9], we have introduced a comprehensive part relationship model for OODBs. In this paper, we wish to examine the possibility of integrating our part relationship model into an existing OODB system. Two questions arise in this regard:

1. Can we make our part relationship an intrinsic construct of an existing OODB system without having to rewrite a substantial portion of the system and without forcing a fundamental change in its underlying data model?
2. Can an "open" OODB system which claims to support this sort of integration really do so? In particular, can such an integration be carried out using a metaclass mechanism [14] which is supposed to provide extensibility for the VODAK Model Language (VML) [17], an OODB system developed at GMD-IPSI?

The VML metaclass mechanism is a powerful extension of other related metaclass schemes including those of Smalltalk [5], Loops [1], and CLOS [21]. Through the description of a custom-built VML metaclass, called the "HolonymicMeronymic" metaclass, we will see that the foregoing questions can be answered in the affirmative. Our metaclass, which was constructed by one of the authors in a short period of time, readily captures the semantics of classes and objects participating in part relationships and part hierarchies. It endows such classes with the means for defining and querying the part relationships between them; it also allows them to create and delete their instances in accordance with the semantics of the part relationship. Furthermore, the metaclass provides the actual instances of the database with functionality consistent with the notions of "part" and "whole." Such instances are given the ability to establish and dissolve part-whole connections between themselves while maintaining the part relationship's characteristics of exclusiveness, multiplicity, dependency, and value propagation. All instances may also be queried regarding their part relationships, allowing for the retrieval of related parts and wholes.

The paper is organized as follows. In Section 2, we review our OODB part relationship model. In Section 3, we give an overview of the VML data model and its notion of metaclass. Then, we go on to discuss the different aspects of the HolonymicMeronymic metaclass; in Section 4, we describe the metaclass's "instance type," and in Section 5, we present its "instance-instance type." Conclusions are found in Section 6.

2 The Part Relationship

In this section, we first present some terminology that is employed throughout the paper. After that, we give an

*This work was partially supported by the Health-Care Information Network and Technology (HINT) project at NJIT.

Appears in: N. Adam, B. Bhargava, and Y. Yesha, editors, *CIKM-94, Proceedings of the Third International Conference on Information and Knowledge Management*, pages 10–17, Gaithersburg, Maryland, Nov. 29–Dec. 2, 1994.

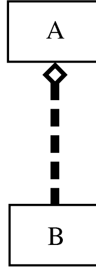

Figure 1: A part relationship between meronymic class B and holonymic class A

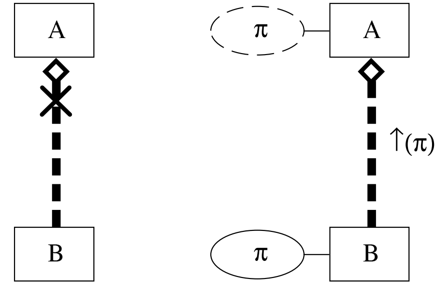

Figure 2: Global exclusive and value propagating part relationships

overview of our part relationship model [6, 7, 8, 9]. Specifically, we present the formal definition of our OODB part relationship and describe some of its characteristics. We also introduce some graphical symbols for the representation of OODB part-whole schemata [7].

Following [26], we refer to a "part" object (e.g., an engine) as a *meronym* (*mero-*, from the Greek *meros*, meaning part). A "whole" object (such as a car) is called a *holonym* (*holo-* meaning whole). A part's class is a meronymic class, whereas the whole's is a holonymic class.

Let us briefly summarize some graphical notational conventions that we will be using to describe general OODB schemata [11]. A class is represented as a box; an attribute of a class is represented by an ellipse attached to its class box via an unlabeled line. A relationship is denoted by a thin, labeled arrow directed from the source class to the target class. When we need to show an instance of a class, we will follow the convention of [23] and denote it as a rectangle with rounded corners.

For a class C, let $E(C)$ denote the extension of C (i.e., the set of all its instances). Also, let $\Pi(C)$ denote the set of all properties of C. The part-whole semantic relationship between a meronymic class B and a holonymic class A (written $P_{B,A}$) is defined as the following quintuple:

$$P_{B,A} = \langle \diamond, \chi, \kappa, \delta, (\{\pi_{B_i}\}, \{\pi_{A_j}\}) \rangle \tag{1}$$

where $\diamond$ is a relation from $E(B)$ to $E(A)$. The pair $(b, a) \in \diamond$ indicates that the instance b of class B *is part of* the instance a of class A. The next four elements are the "characteristic" dimensions, referred to in order as: (a) *exclusiveness*, (b) *multiplicity*, (c) *dependency*, and (d) *value propagation*. The domains of the first three, χ, κ, and δ, are as follows:

$$\begin{aligned} &\chi \in \{\textit{global-exclusive, class-exclusive, shared}\}, \\ &\kappa \in \{(l,u) \mid l \in \{0,1,2,\ldots\}, u \in \{1,2,3,\ldots\} \cup \{\infty\}, l \leq u\}, \\ &\delta \in \{\textit{part-to-whole, whole-to-part, nil}\}. \end{aligned} \tag{2}$$

The situation where $\delta = nil$ indicates the lack of any dependency; $u = \infty$ indicates the lack of an upper bound on the multiplicity. The final dimension, the value propagation dimension, comprises a pair of sets $\{\pi_{B_i}\}$ and $\{\pi_{A_j}\}$, where $\{\pi_{B_i}\} \subset \Pi(B)$, $\{\pi_{A_j}\} \subset \Pi(A)$, and $\{\pi_{B_i}\} \cap \{\pi_{A_j}\} = \emptyset$.

Let us now briefly, and informally, describe the interpretation of each of these characteristic dimensions and their respective values. (For more extensive, formal treatments, see [6, 7, 9].) The *exclusiveness dimension* specifies how parts may be distributed among wholes [7, 13, 22]. A *global-exclusive* part relationship imposes the restriction that a meronym be part of at most one holonym at any given instant of time, as is true for the relationship between engines and cars. (*Class-exclusive* is a related variation of this [7].) A shared part relationship, on the other hand, allows a given meronym to be part of any number of holonyms simultaneously. The *multiplicity dimension* prescribes bounds on the cardinality of the set of meronyms constituting each holonym. The *dependency dimension* deals with the part relationship's deletion semantics. The deletion of the whole may imply the deletion of the part (*part-to-whole*), or vice versa (*whole-to-part*).

The *value propagation dimension* defines *derived attributes* at one class of the part relationship in terms of the same property at the other class. For example, a car's color could be defined as the color of its constituent body. Any number of properties may be defined in this way. In Definition 1, the set $\{\pi_{B_i}\}$ contains all the properties of class A defined in terms of the same properties at class B; it is called the "upSet" because the property values are propagated upward from B to A. The set $\{\pi_{A_j}\}$ holds all the properties of B defined in terms of the properties at A; it is called the "downSet." Recall that, by definition, the sets $\{\pi_{B_i}\}$ and $\{\pi_{A_j}\}$ are disjoint, ensuring that no derived attribute is defined circularly.

The basic graphical schema symbol for the part relationship is a bold, dashed line with a diamond head at one end signifying the holonymic class (Figure 1). Variations of this symbol serve to capture the values of the characteristic dimensions [7]. Those symbols needed in this paper are shown in Figure 2. The first denotes a global exclusive part relationship, indicated by the "X." The second represents a part relationship which propagates the property π upward from B to A. Its *propagation label*, comprising an up-arrow and π in parentheses, indicates that the property π is being propagated upward.[1] The presence of the derived attribute at A is signified by the dashed ellipse enclosing π. Let us note that both relationships in Figure 2 are single-valued (i.e., have minimum cardinality 0 and maximum cardinality 1) as denoted by the single bold line and lack of cardinality labels (see [7]).

3 The VML Data Model and Metaclasses

VML (VODAK Model Language) [16] is an open object-oriented data model which can be tailored to the needs of specific applications [15, 18]. It employs a "dual" model [4] to describe the structure and semantics of the classes and objects of an OODB. The duality arises through the separation of the notions of class and object type. Each class in the schema is associated with exactly one object type, referred to as its *instance type*, which defines the structure and behavior of instances of the class. In Figure 3, we have used a shading pattern to show the effect of the instance

[1] A down-arrow would indicate a downward propagation.

type[2] on an instance. A single object type, on the other hand, may be associated with any number of classes.

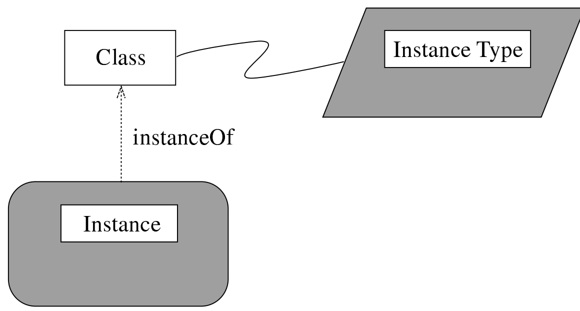

Figure 3: The instance type's effect on a class's instances

To maintain uniformity, classes are also considered objects in VML (cf. Smalltalk [5]). As such, classes themselves are instances of other classes referred to as *metaclasses*. However, as described in [14, 16, 17], the interaction between metaclasses, object types, and (ordinary) classes is different than that between classes, object types, and instances. Just as with an ordinary class, a metaclass has an instance type that describes its instances, which in this case are classes. Furthermore, a second object type may be associated with a metaclass to augment the structure and behavior of the instances of its instances. This second object type is called an *instance-instance type* [14, 16]. Thus, through its two associated object types, a metaclass influences the structure and behavior of both its own instances, which are classes, and the instances of those classes in turn. This arrangement is illustrated in Figure 4 where we have again employed shading patterns to demonstrate the effect of the metaclass's instance type on a class. Figure 4 also shows the effect of the metaclass's instance-instance type and the class's instance type on the class's own instances.

The metaclass mechanism serves as the means for tailoring the VML data model to the needs of a particular application or an entire application domain. Specifically, a semantic relationship, like the part relationship, can be introduced through the definition of a custom metaclass, which endows its instances (which are classes) and their instances, in turn, with structure and behavior befitting the desired semantic relationship. We have availed ourselves of this capability and built the *HolonymicMeronymic* metaclass to capture the semantics of our part relationship. Any class participating in a part hierarchy is defined as an instance of this metaclass. We will refer to such classes as HolonymicMeronymic (HM) classes. Through its instance type and instance-instance type, our metaclass does the following for HM classes and their instances:

- It provides the means for defining a part relationship between a pair of HM classes, making one a holonymic class and the other a meronymic class with respect to each other. It also allows HM classes to be queried in regard to their part relationships.
- It furnishes an HM class with the constructor method *make* and the destructor method *destroy*, which respectively capture the creation and deletion semantics of the part relationship.

[2] We represent the instance type as a parallelogram.

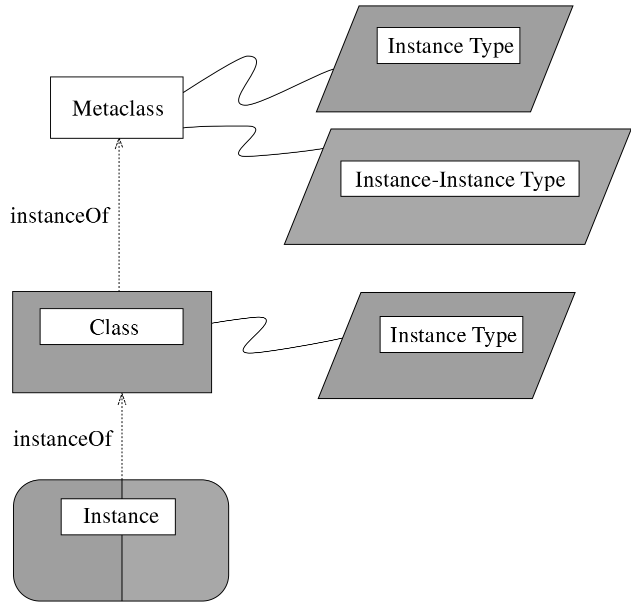

Figure 4: The interaction between metaclasses, classes, and instances

- It provides an HM class's instances with methods for establishing and dissolving part-whole connections and querying with respect to the class's various part relationships. These methods include *addPart*, *removePart*, *changePart*, *getParts*, and *getWholes*.
- It provides the means for performing both upward and downward value propagation across part relationships.

In succeeding sections, we go on to discuss the details of the instance type and the instance-instance type of the HolonymicMeronymic metaclass and describe the different functionality that each contributes.

4 HolonymicMeronymic Instance Type

In this section, we describe the details of the HolonymicMeronymic instance type. We first present its actual public interface which represents its contribution to the behavior of any HM class. After that, we discuss how these methods capture the semantics of the part relationship.

The public interface for the HolonymicMeronymic instance type appears in the following.[3] It shows that the HolonymicMeronymic metaclass provides ten new methods for all HM classes. (Remember that a class in VML is itself an object, and these methods augment the behavior of a class, not the behavior of its instances.) As we mentioned above, the methods *make* and *destroy* are used, respectively, to create and delete instances of an HM class, and encode the creation and deletion semantics of the part relationship. They are further discussed in Section 4.2. The additional eight methods are used to define and obtain information about the part relationships in which an HM class participates. Let us note that one of our design decisions was to store the values of all the characteristic di-

[3] For the sake of brevity, we have omitted some extraneous utility methods which are described in [10]. Note also that, as required by VML, this instance type is defined as a subtype of "Metaclass_InstType"; see [16, 17] for details.

mensions of a part relationship at its holonymic class. Even though such information could have been stored at the meronymic class, we believe that the inherent bottom-up construction associated with part hierarchies—where integral objects are built up from lower-level component objects—makes ours the proper choice. This decision accounts for the asymmetry of the methods *defHolonymicClasses* and *defMeronymicRelshps*, which define the part relationships of a given HM class. It also explains the need for the formal parameter "anHMClass" (representing a given meronymic class) of the last six methods which query an HM class in regard to the characteristic dimensions of its part relationships. These methods are elaborated on in the next section.

```
OBJECTTYPE HolonymicMeronymic_InstType SUBTYPEOF
                                Metaclass_InstType;
INTERFACE
  METHODS
//
// Constructor and Destructor for the instances of
// an HM class.
//
   make(someParts: {OID}): OID;
   destroy(anObject : OID);
//
// Used to define HM class's part relationships.
//
   defMeronymicRelshps(someRelshps:
                      {PartRelationshipType});
   defHolonymicClasses(someClasses: {OID});
//
// Used to obtain values of the characteristic
// dimensions of the specific part relationships
// that an HM class participates in.  The formal
// parameter "anHMClass" of each method
// represents the meronymic class of the part
// relationship of interest.
//
   exclusiveness(anHMClass: OID):
                          ExclusivenessType;
   minCardinality(anHMClass: OID): INT;
   maxCardinality(anHMClass: OID): INT;
   dependency(anHMClass: OID): DependencyType;
   propertyUpPropagated(methodName: STRING,
                     anHMClass: OID): BOOL;
   propertyDownPropagated(methodName: STRING,
                       anHMClass: OID): BOOL;
```

4.1 Creating and Querying an HM Class

Before we begin our discussion of HM classes, let us first consider the task of creating an "ordinary" application class (i.e., one without any part relationships) in VML. This will show the impact of introducing part relationships into class definitions. Assume that we wish to define a class *Car* which has the schema illustrated in Figure 5. *Car* has three attributes, *serialNumber*, *model*, and *year*, and a single relationship *manufacturedBy* to the class *Company*.

Because each class in VML has an instance type, a complete class specification is divided into two portions: (1) an instance type declaration and (2) the class declaration itself which contains a reference to the instance type via its "INSTTYPE" clause. To illustrate this, we show the VML code for *Car* and its instance type *CarType* in the following.

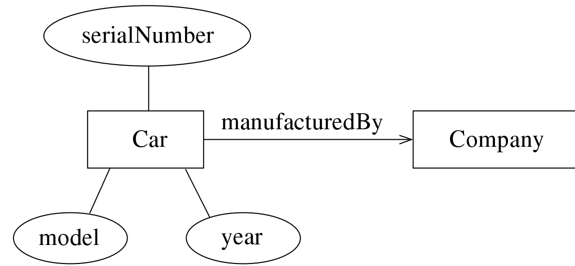

Figure 5: The class *Car* without part relationships

```
CLASS Car
    INSTTYPE CarType[Company]
END;

OBJECTTYPE CarType[Comp: CompanyType];
IMPLEMENTATION
   PROPERTIES
    serialNumber: INT;
    model: modelType;
    year: yearType;
    manufacturedBy: Comp;
END;
```

In the declaration of the object type,[4] shown below the class declaration, we see the definition of the four properties from Figure 5. In VML, there is no syntactical distinction between attributes and relationships, and they are written together in the "PROPERTIES" section. However, the three attributes are followed by appropriate data types, while the relationship *manufacturedBy* is instead followed by a parameterized reference to the class *Company* (whose own definition is not shown here).

Now, to define an HM class, one does two things. First, one declares the class to be an instance of the Holonymic-Meronymic metaclass by including a METACLASS clause on its opening line [17]. Second, one adds appropriate invocations of the two companion methods *defMeronymicRelshps* and *defHolonymicClasses* to the class's "INIT" clause[5] in order to establish its desired part relationships. The method *defHolonymicClasses* informs the new HM class of all other HM classes that are holonymic classes with respect to it. Analogously, the method *defMeronymicRelshps* notifies it of all related meronymic classes. However, *defMeronymicRelshps*, as its name suggests, provides more than just the names of the meronymic classes; it also provides the values of all characteristic dimensions of each of the part relationships in which the new HM class plays the role of the holonymic class. The reason for this, as noted above, is because all such information is stored at and retrievable through the holonymic class.

To illustrate the above points, let us expand our earlier example and define *Car* as the holonymic class in two global-exclusive, single-valued part relationships, one with *Engine* and the other with *Body* (Figure 6). In the figure, we have omitted all the properties of *Car* presented earlier and have included only *Body*'s attribute *color*, which, as we will see later, will be propagated upward. The VML syntax corresponding to this schema is as follows. (We have omitted the

[4]We have omitted some of the details of the object type declaration that are not relevant here. See [16, 17] for a complete description of the syntax.

[5]The INIT clause is a characteristic of all classes in VML [17]. It is used as a means for performing initialization procedures at the time the class is created.

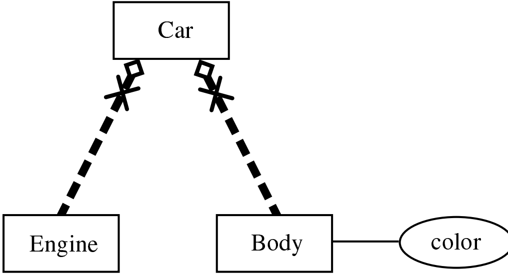

Figure 6: Car and its parts Engine and Body

declarations of the instance types *CarType* and *EngineType* because they are not relevant to the discussion. *BodyType* is shown because it defines the property *color*.)

```
CLASS Car METACLASS HolonymicMeronymic
    INSTTYPE CarType[Company]
    INIT Car->defMeronymicRelshps(
                {
                 [theMeronymicClass:Engine,
                  exclusiveness:GLOBAL_EXCLUSIVE,
                  multiplicity:[min: 0, max: 1],
                  dependency:NIL,
                  upSet:{},
                  downSet:{}],
                 [theMeronymicClass:Body,
                  exclusiveness:GLOBAL_EXCLUSIVE,
                  multiplicity:[min: 0, max: 1],
                  dependency:NIL,
                  upSet:{},
                  downSet:{}]
                })
END;

CLASS Engine METACLASS HolonymicMeronymic
   INSTTYPE EngineType
   INIT Engine->defHolonymicClasses( {Car} )
END;

CLASS Body METACLASS HolonymicMeronymic
   INSTTYPE BodyType
   INIT Body->defHolonymicClasses( {Car} )
END;

OBJECTTYPE BodyType;
IMPLEMENTATION
   PROPERTIES
    color: colorType;
END;
```

The opening line of each class declaration now contains an explicit reference to the HolonymicMeronymic metaclass. In the INIT clauses of *Engine* and *Body*, the invocation of *defHolonymicClasses* with the singleton set containing *Car* informs each class that *Car* is the one holonymic class with respect to it. The invocation of *defMeronymicRelshps* in the INIT clause of *Car* notifies that class of its related meronymic classes (i.e., *Engine* and *Body*) as well as the values of the respective characteristic dimensions. The argument to *defMeronymicRelshps* is a set of structures of type "PartRelationshipType" having the following definition:

```
DATATYPE PartRelationshipType =
               [theMeronymicClass: OID,
                exclusiveness: ExclusivenessType,
                multiplicity: MultiplicityType,
                dependency: DependencyType,
                upSet: {STRING},
                downSet: {STRING}];
```

The first member of the structure is the name of the meronymic class which, in VML, is just an alias for the class's OID—hence, the data type "OID" for this member. The remainder represent the values of the characteristic dimensions as described above. The "upSet" and "downSet" comprise VML strings corresponding to the names of the propagated properties.

In the declaration of *Car* above, we see that each part relationship is global-exclusive (as represented by the symbolic constant GLOBAL_EXCLUSIVE) and single-valued (represented by a minimum cardinality of 0 and a maximum cardinality of 1). Each also lacks any dependency semantics as specified by NIL. The two upSets and downSets are empty, meaning that no propagation takes place with respect to either part relationship.

It will be noted that for a class having no associated holonymic classes (i.e., a class which is the root of a part hierarchy), such as *Car* in our example, the method *defHolonymicClasses* is not needed in the INIT clause. For those classes with no associated meronymic classes (i.e., leaves of a part hierarchy), the method *defMeronymicRelshps* is not included in the INIT clause. Such is the case for the classes *Engine* and *Body*.

As we have discussed, the values of a part relationship's characteristic dimensions are stored with its holonymic class. To obtain these values, it is necessary to query that class directly using the last six methods provided by the HolonymicMeronymic instance type. Each method takes as an argument the related meronymic class in the desired part relationship and returns the appropriate dimensional value. For example, to retrieve the value of the exclusiveness dimension of the part relationship between *Car* and *Engine*, we query *Car* as follows:

```
x := Car->exclusiveness(Engine);
```

Here, the variable x would get assigned the value GLOBAL_EXCLUSIVE, as specified in the declaration of *Car* above. Note that the two methods, *propertyUpPropagated* and *propertyDownPropagated*, do not return the entire upSets and downSets, but instead serve as their characteristic functions. They are used by the NOMETHOD clause (discussed below) to perform value propagation.

4.2 Creating and Deleting Parts and Wholes

The method *make* is used to create instances of an HM class, and it encodes the creation semantics dictated by the part relationship's characteristic dimensions. Before we consider these, let us see how it is used. To create an instance of an HM class, one simply invokes *make* for that class. To create an instance of *Car*, we write[6]

```
Car->make({});
```

From its signature, we see that *make* takes as its argument a set of OIDs, which represents the objects that are to be

[6] VML's method invocation is syntactically like that of C++ [25].

initially attached as parts to the newly created object. The set is heterogeneous in that it may contain the OIDs of objects of any type. If one of the given objects is from a class other than the prescribed meronymic classes of the target class, then the creation of the new instance is aborted. In the above invocation, the argument is just the empty set, so the new car initially does not have a body or an engine.

The creation semantics of the part relationship comprises two kinds of constraints which *make* is responsible for monitoring. First, *make* must ensure that the bounds imposed by the multiplicity dimensions of any of the target class's part relationships are satisfied at the outset of the instance's lifetime. (As we will see below, the methods *addPart* and *removePart* of the instance itself carry this same responsibility for the remainder of its lifetime.) If *make* detects a potential violation of this constraint (e.g., there are too few or too many parts of some type), then it aborts the creation and returns NULL, a special VML constant.

The second constraint, related to the first, involves the semantics of exclusiveness. Even if *make* receives valid numbers of parts, it may still be the case that one (or more) of these parts cannot legally be installed because it is owned exclusively by another holonym. Such a situation is also handled by an abort.

Instances of an HM class are deleted using *destroy* as follows:

```
Car->destroy(c);
```

The argument c is the OID of the car of interest. Analogously to *make*, *destroy* encodes the part relationship's deletion semantics, which are dictated by its characteristic dimensions. The algorithm which captures the deletion semantics can be found in [6, 10]. We omit it here for the sake of brevity.

5 HolonymicMeronymic Instance-Instance Type

In this section, we describe the details of the Holonymic-Meronymic instance-instance type which endows meronyms and holonyms with behavior consistent with the semantics of the part relationship. Remember that this object type determines the influence of the metaclass on the instances (Figure 4, brick pattern). Specifically, it gives such instances the ability to establish and break, as well as change and query, part-whole connections with other instances. Furthermore, through its NOMETHOD clause, it provides the means by which value propagation is accomplished.

The public interface for the instance-instance type is as follows.

```
OBJECTTYPE HolonymicMeronymic_InstInstType
                SUBTYPEOF Metaclass_InstInstType;
INTERFACE
  METHODS
   addPart(aPart: OID): BOOL;
   removePart(aPart: OID): BOOL;
   changePart(oldPart: OID, newPart: OID): BOOL;
   getParts(): {OID};
   getWholes(): {OID};
```

In the following subsections, we discuss the details of these methods.

5.1 Establishing Part-Whole Connections between Instances

Assume that there exists a part relationship $P_{B,A}$. To establish a part connection between the instance b of B and the instance a of A, we invoke *addPart* for a as follows.[7]

```
a->addPart(b);
```

A Boolean value is returned by *addPart* to indicate the success or failure of the part installation. Failures can occur in two different scenarios. In the first, the argument b is not an instance of an appropriate meronymic class.[8] In the second, b's attachment to a would be a violation of either an exclusiveness or a multiplicity (i.e., maximum cardinality) constraint. The details of the algorithm for detecting such violations can be found in [6, 10].

5.2 Dissolving Part-Whole Connections between Instances

To dissolve an occurrence of a part relationship between a pair of instances, we use *removePart*. For example, to remove the meronym b from the holonym a, we write:

```
a->removePart(b);
```

Like *addPart*, *removePart* returns a Boolean value to indicate its success or failure.

It should be noted that in the context of a part relationship with identical upper and lower multiplicity bounds, *removePart* is guaranteed to fail because the removal of the part is certain to violate the lower bound. Thus, in such circumstances, it is not possible, using the methods described so far, to exchange one part for another. To rectify this, we provide the method *changePart* which in a single transaction removes one part and replaces it with another of the same type as follows:

```
a->changePart(b1, b2);
```

As with the other methods, *changePart* returns a Boolean value to indicate success or failure. The exchange fails if b_1 and b_2 are not of the same type, or if the attachment of b_2 to a violates an exclusiveness constraint.

5.3 Querying a Part Hierarchy

Once part-whole connections have been established between instances, we can query these using the methods *getParts* and *getWholes*.

The method *getParts* returns *all* the parts of the target instance. For example, to get the parts of an instance a, we do the following:

```
theParts := a->getParts();
```

[7] To be consistent with our bottom-up construction of HM classes (see Section 4) and to simplify the coding of *addPart*, we have adopted, without loss of generality, a protocol that requires the establishment and dissolution of part connections at the holonym only. *AddPart* is also responsible for informing the involved meronym about the transaction.

[8] Because *addPart* is defined generically in the instance-instance type for all possible application schemata, it is not possible to check for such an error statically.

If a has no parts, then the resultant set ("theParts") is empty. Note that this method only returns the direct parts of an instance; it is not recursive and does not return the parts of the parts.

To obtain all the holonyms of an instance a, we use the method *getWholes* as in:

```
theWholes := a->getWholes();
```

Once again, if a is not part of any objects, then the resultant set ("theWholes") is empty.

5.4 Performing Value Propagation using the NOMETHOD Clause

In VML, the inheritance behavior specified by a semantic relationship is captured using the NOMETHOD clause (or simply NOMETHOD) [16, 17] of the custom metaclass's instance-instance type. As its name implies, NOMETHOD is the mechanism by which an instance in a VML database deals with a method invocation that it is not equipped to handle (i.e., for which it has *no method*). NOMETHOD is actually a VML code segment that is executed automatically when an unknown method is invoked on an object. It looks very much like an ordinary method, with the following exceptions. First, NOMETHOD cannot be given any formal parameters. Second, in its scope, there are two predefined identifiers: (1) *currentMeth*, which is bound to the offending method's name; and (2) *arguments*, which is bound to the list of arguments passed to the method.

NOMETHOD has previously been used to implement the inheritance behavior in hierarchies of semantic relationships, e.g., in category specialization hierarchies [14]. There, it was simply defined to pass *currentMeth* and *arguments* along "as is" to a related category generalization object of the target through another method invocation. In the present context, we use NOMETHOD to determine if there exists a related object (either a holonym or a meronym) which can perform a prescribed value propagation. (The determination is made using the characteristic functions—implemented by the methods *propertyUpPropagated* and *propertyDownPropagated* (Section 4.1)—of the various part relationships' upSets and downSets.) If NOMETHOD can find such an object, then it invokes *currentMeth* on that object in order to obtain the desired property value. For a complete description of our NOMETHOD algorithm, see [10].

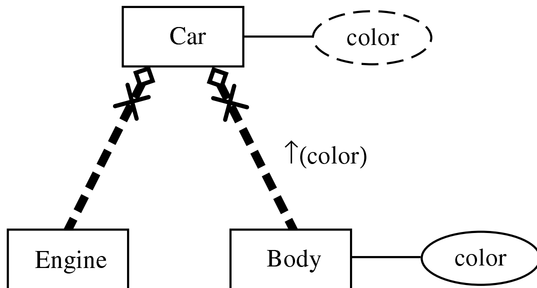

Figure 7: *Body* propagating *color* upward to *Car*

Referring back to our example, assume that we wish to propagate the property *color* upward from *Body* to *Car*, as illustrated in Figure 7. We would then write *Car*'s declaration (minus the part relationship to *Engine*) as:

```
CLASS Car METACLASS HolonymicMeronymic
    INSTTYPE CarType[Company]
    INIT Car->defMeronymicRelshps(
                {[theMeronymicClass:Body,
                  exclusiveness:GLOBAL_EXCLUSIVE,
                  multiplicity:[min: 0, max: 1],
                  dependency:NIL,
                  upSet:{'color'},
                  downSet:{}]})
END;
```

As we see, the upSet for the part relationship between *Body* and *Car* contains only the single element 'color.' The downSet is empty, indicating that no properties are propagated downward along the relationship.

To see the effect of this declaration, consider the following segment of code.

```
(1)   aBody := Body->make({});
(2)   aBody->setColor(RED);

(3)   aCar := Car->make({aBody});
(4)   c := aCar->color();
```

In line (1) a new instance of *Body* is created, and in line (2) its color is set to RED. Line (3) creates a new instance of *Car* and makes "aBody" its part. Now in line (4), c gets assigned the value of the color of the body of the car "aCar," i.e., the color RED. Note that the method 'color' is not directly defined for cars but for bodies. As the upSet for the relationship between *Body* and *Car* contains the element 'color,' this method is made available (i.e., is made executable) in the context of cars.

6 Conclusion

In this paper, we have demonstrated how to seamlessly integrate a powerful part relationship model into an existing OODB system. The integration was performed using the metaclass paradigm of the VML OODB System of GMD-IPSI. The product of this integration effort was a custom metaclass, called the HolonymicMeronymic metaclass. The implementation work was carried out at NJIT in cooperation with GMD-IPSI in Darmstadt. The metaclass itself comprises about seven hundred lines of VML code, written by one of the authors, who had no prior experience with metaclasses, in the full-time equivalent of one month. The ease with which we were able to install an independently developed model of parts (formulated over two years) into VML attests to the power of its metaclass concept and implementation, and lends credence to its "openness."

The HolonymicMeronymic metaclass will be available as part of the metaclass library provided with the next VML release. (VML is available as a prototype from GMD-IPSI.) Using this library, the standard model provided with VML can be adapted to the specific modeling needs of different application domains, particularly those which require modeling support for part relationships. The HolonymicMeronymic metaclass provides classes that participate in part hierarchies and their instances with structure and behavior befitting the part relationship. Such HM classes are given the ability to define part relationships and respond to queries against these. They are also provided with constructor and destructor methods that encode the creation and deletion semantics of the part relationship. Instances of HM classes can establish, dissolve, and modify part-whole connections

while maintaining the part relationship's semantics of exclusiveness, multiplicity, dependency, and value propagation. Such instances can also handle queries with regard to their part relationships, returning related parts or wholes on demand.

Presently at NJIT, plans call for the HolonymicMeronymic metaclass to be used in a medical informatics modeling project. We also intend to exploit it in the process of converting schemata represented in our graphical notation into the data definition language of VML. The metaclass implementation described in this paper has also served as the basis for the development of a VML class library for a document versioning model at GMD-IPSI.

Acknowledgments

We would like to thank Erich Neuhold of GMD-IPSI for suggesting the idea of incorporating the part relationship into the VML system. We also thank Gisela Fischer of GMD-IPSI for her help with VML.

References

[1] D. G. Bobrow and M. Stefik. The Loops manual. Technical Report KB-VLSI-81-13, Xerox PARC, 1981.

[2] N. Cercone. The ECO family. In [19], pages 95–131.

[3] S. E. Fahlmann. *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press, Cambridge, MA, 1979.

[4] J. Geller, E. Neuhold, Y. Perl, and V. Turau. A theoretical underlying Dual Model for knowledge-based systems. In *Proc. First Int'l Conf. on Systems Integration*, pages 96–103, Morristown, NJ, 1990.

[5] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1983.

[6] M. Halper. *A Comprehensive Part Model and Graphical Schema Representation for Object-Oriented Databases.* PhD thesis, NJIT, Oct. 1993.

[7] M. Halper, J. Geller, and Y. Perl. An OODB "part" relationship model. In Y. Yesha, editor, *Proc. ISMM 1st Int'l Conference on Information and Knowledge Management*, pages 602–611, Baltimore, MD, Nov. 1992.

[8] M. Halper, J. Geller, and Y. Perl. "Part" relations for object-oriented databases. In G. Pernul and A. Tjoa, editors, *Proc. 11th Int'l Conference on the Entity-Relationship Approach*, pages 406–422, Karlsruhe, Germany, Oct. 1992.

[9] M. Halper, J. Geller, and Y. Perl. Value propagation in OODB part hierarchies. In B. Bhargava, T. Finin, and Y. Yesha, editors, *Proc. ISMM/ACM 2nd Int'l Conference on Information and Knowledge Management*, pages 606–614, Washington, DC, Nov. 1993.

[10] M. Halper, J. Geller, and Y. Perl. Report on the implementation of part relationships using VML metaclasses. Study 224, GMD, Sankt Augustin, Jan. 1994.

[11] M. Halper, J. Geller, Y. Perl, and E. J. Neuhold. A graphical schema representation for object-oriented databases. In R. Cooper, editor, *Interfaces to Database Systems*, pages 282–307. Springer-Verlag, London, 1993.

[12] M. N. Huhns and L. M. Stephens. Plausible inferencing using extended composition. In *Proc. IJCAI-89*, pages 1420–1425, Detroit, MI, 1989.

[13] W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. In *Proc. 1989 ACM SIGMOD Int'l Conference on the Management of Data*, pages 337–347, Portland, OR, June 1989.

[14] W. Klas. *A Metaclass System for Open Object-Oriented Data Models.* PhD thesis, Technical University of Vienna, January 1990.

[15] W. Klas. Tailoring an object-oriented database system to integrate external multimedia devices. In *Workshop on Heterogeneous Databases & Semantic Interoperability*, Boulder, CO, 1992.

[16] W. Klas, K. Aberer, and E. J. Neuhold. Object-oriented modeling for hypermedia systems using the VODAK Model Language (VML). In T. Oszu and A. Biliris, editors, *Object-Oriented Database Management Systems*, NATO ASI. Springer-Verlag, Berlin, Feb. 1994.

[17] W. Klas et al. VODAK design specification document, VML 3.1. Technical report, GMD, Sankt Augustin, July 1993.

[18] W. Klas, E. J. Neuhold, and M. Schrefl. Using an object-oriented approach to model multimedia data. *Computer Communications*, 13(4):204–216, 1990. Special Issue on Multimedia Systems.

[19] F. Lehmann, editor. *Semantic Networks in Artificial Intelligence.* Pergamon Press, Tarrytown, NY, 1992.

[20] B. K. MacKellar and J. Peckham. Representing design objects in SORAC: A data model with semantic objects, relationships and constraints. In *Second International Conference on Artificial Intelligence in Design*, Pittsburgh, PA, June 1992.

[21] D. A. Moon. The Common Lisp object-oriented programming language standard. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 49–78. ACM Press, New York, NY, 1989.

[22] G. T. Nguyen and D. Rieu. Representing design objects. In J. Gero, editor, *AI in Design '91.* Butterworth-Heinemann Ltd., 1991.

[23] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[24] S. C. Shapiro and W. J. Rapaport. The SNePS family. In [19], pages 243–275.

[25] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Co., Inc., Reading, MA, second edition, 1991.

[26] M. E. Winston, R. Chaffin, and D. J. Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11(4):417–444, 1987.