

# Incorporating Semantic Relationships into an Object-Oriented Database System\*

Li-min Liu  
CIS Dept.  
NJIT  
Newark, NJ 07102 USA  
limin@homer.njit.edu

Michael Halper  
Dept. of Mathematics & Computer Science  
Kean University  
Union, NJ 07083 USA  
mhalper@turbo.kean.edu

## Abstract

*Semantic relationships, those class-to-class connections that carry inherent support for constraints and various other functionalities, play an important role when building information models for applications. This is true whether one employs traditional data modeling techniques, knowledge-representation languages, or object-oriented modeling methodologies. An example of such a semantic relationship is the part-whole relationship. In fact, most of the popular object-oriented modeling approaches include such constructs in their repertoire of data modeling primitives. However, commercial object-oriented database (OODB) systems ordinarily do not provide built-in support for them. In this paper, we present a methodology by which a semantic relationship can be incorporated into an existing OODB system. At first, we give an overview of the general methodology for carrying out this integration task. Then, in order to ground our work in a real system, we show the addition of a part-whole semantic relationship to the ONTOS DB/Explorer OODB management system. This implementation is currently up and running, and an example application demonstrating its use is available on the Web.*

## 1 Introduction

Semantic relationships play a central role in building information models for applications. This is true whether one is using traditional modeling techniques like SDM [18] and extended ER [10, 11], knowledge-representation languages such as Telos [27] and K-Rep [25], or object-oriented modeling methodologies including OMT [5, 36] and Coad/Yourdon [8]. By “semantic relationship” we mean more than just a named binary relationship that can be

modeled as a link (pointer) between object classes or categories. Such a relationship carries additional semantics in the form of constraints and various other functionalities—such as inheritance, operation propagation, or specialized query capabilities—that allow it to more precisely model the enterprise of interest [38, 41]. Some examples of semantic relationships are the part-whole relationship [4, 16, 28], ownership [42], materialization [21, 34], and role-of [12]. We will concentrate on the object-oriented methodologies and object-oriented database (OODB) systems because they are commercially most widely available.

While popular object-oriented modeling techniques such as OMT, UML [39], and Coad/Yourdon include semantic relationships, few, if any, commercial OODB systems provide intrinsic support for their use. This engenders the unacceptable situation where in order to gain persistence for an application, some of its sophistication in modeling the enterprise of interest must be given up. This directly contradicts the promise of OODBs for improved modeling of applications.

In this paper, we present a methodology for incorporating semantic relationships into the repertoire of built-in modeling primitives provided by OODB systems. The underlying object model of such a system is extended to include additional semantics that bring it in line with the various modeling techniques. When devising our integration methodology, we imposed two major restrictions: (1) It should not cause a major upheaval in the underlying OODB system. (2) It should not radically alter the environment that an application developer is used to working in. For example, it must not introduce exotic syntax into the data definition language (DDL) and the preferred data manipulation language (DML) of the OODB system.

Our methodology augments the OODB system’s metadata schema (sometimes called the “metaschema” or, more commonly, the “data dictionary”) and an application-level schema with two additional object classes. The first class supplements the data dictionary and provides run-time access to knowledge about occurrences of the semantic re-

\*This research was (partially) done under a cooperative agreement between the National Institute of Standards and Technology Advanced Technology Program (under the HIIT contract #70NANB5H1011) and the Healthcare Open Systems and Trials, Inc. consortium.

relationship of interest appearing in a given OODB schema. The second class, provided as part of an OODB's class library, is used to enhance the structure and behavior of objects (instances) of classes that are connected via the semantic relationship. In this manner, the objects of the database will exhibit functionality befitting participants in the given semantic relationship. In the case of the part relationship, for example, the objects will behave like parts and wholes. For the ownership relationship, the objects will act like owners and possessions.

It is important to note that the incorporation of a semantic relationship into an OODB via our approach relieves application designers of a great deal of tedious work. No burdensome hand-coding of integrity checks falls on the shoulders of a designer who is interested in modeling with, say, parts and wholes. The designer makes a declarative specification of the desired part/whole semantics and can then rest assured that it is properly maintained throughout the entire life-time of the OODB. A "part" or "whole" object will exhibit appropriate behavior from the moment it is instantiated by one application until the time it is eventually deleted by another.

To ground our work in a real system, we will demonstrate the introduction of a part relationship into the ON-TOS DB/Explorer ("ONTOS" for short) [30, 31], a commercially available OODB management system. This implementation has been completed and is currently up and running. A demonstration application employing the ON-TOS part relationship is on the Web [32]. Our implementation demonstrates that a semantic relationship can be added to an OODB system without causing a fundamental upheaval and without drastically altering its familiar development environment.

We have previously presented an alternate methodology for integrating semantic relationships into OODB systems [17]. That technique had the much more stringent requirement that the OODB system include support for "meta-classes" as defined in [19, 20]. Most commercial systems, such as ON-TOS, ObjectStore [22, 29, 37], Versant [40], etc., do not have such a mechanism. A meta-object protocol has been used for the inclusion of semantic relationships in CLOS [21]. An extensive discussion of semantic relationships and their role in data modeling can be found in [38]. The set-membership relationship [26] has been shown to have a significant bearing on the definition of semantic relationships.

The treatment of user-defined relationships as "first-class" constructs in OODBs was expounded in the seminal paper of Rumbaugh [35], and was extended by [3] and [9], both of which permit additional constraints on the user-defined relationships. The relationship construct of [3] has been included in the Fibonacci language [2], while that of [9] has been implemented in ADAM [33], a Prolog-based

OODB system. The SORAC model [24] utilizes relationships as a means for specifying constraints on designs in a knowledge-based/object framework. In contrast to this previous work, we are *not* introducing another model of relationships for OODB systems. Our goal is to present a general methodology that can be utilized for the incorporation of semantic relationships into a wide range of commercially available OODB management systems without drastically altering their customary working environments.

The rest of this paper is organized as follows. In Section 2, we give a brief overview of the part relationship and of ON-TOS. Section 3 presents our methodology for incorporating a semantic relationship into OODB systems. After that, in Section 4, we discuss the specific case of adding the part relationship to ON-TOS. Conclusions appear in Section 5.

## 2 Background

### 2.1 Part-Whole Semantic Relationship

In previous work [13, 14, 16], we have developed a comprehensive part-whole semantic relationship ("part relationship") for OODB systems. This relationship can be used by a designer to correctly capture the proper part-whole semantics between various objects within an application. The designer simply specifies in a declarative fashion which semantics—out of a wide array of choices available for parts and wholes—is desired. The burden of ensuring the maintenance of the semantics is then placed solely on the OODB system.

Our part relationship is organized into four characteristic dimensions, each of which is responsible for one aspect of the semantics of parts and wholes [13, 14, 16]. These dimensions are: (1) exclusiveness, (2) multiplicity, (3) dependency, and (4) inheritance [15]. In the following, we briefly describe each of these in turn.

The *exclusiveness dimension* provides constraints dealing with the distribution of parts among wholes. An example would be a power boat having its engine exclusively at any one time: No other boat would be allowed to have that same engine simultaneously, just as we would expect. Such a constraint is called global-exclusiveness. Our model also supports a variation called class-exclusiveness [14], as well as sharing, where a given part can be a constituent of any number of wholes concurrently.

The *multiplicity dimension* specifies the number of parts, of a certain kind, that can be used in the construction of a whole. For example, a power boat may be defined to contain up to two engines. On the other hand, lower-bound constraints in this dimension can be used to capture *essentiality* among parts and wholes. The table of contents and index may be modeled as essential parts of a book.

The *dependency dimension* deals with the deletion semantics of parts and wholes, i.e., the way the deletion operation is propagated between such objects. The deletion of the whole may imply the deletion of one or more of its parts, or vice versa. As an example, the existence of a bicycle may be predicated on the existence of a constituent frame. If the frame is deleted from the database, then so too should its bicycle.

The *inheritance dimension* specifies which properties are inherited by the whole from the part, or the other way around, and how the inheritance takes place. As we have discussed in [15, 16], there is a great deal of subtlety that distinguishes part-whole inheritance from ordinary subclass (IS-A) inheritance in OODB schemata. Among the distinctions is the fact that part-whole inheritance has both a schema-level (i.e., intensional) aspect and an instance-level (i.e., extensional) aspect. A power boat, in general, has the property “horsepower” by dint of its having engines; a specific boat has the horsepower of the engine that happens to be installed in it. Moreover, if the boat has multiple engines, then its overall horsepower is the sum of the horsepowers of its engines.

## 2.2 Aspects of ONTOS

We have employed ONTOS as the implementation vehicle for the part relationship. ONTOS uses C++ as its primary DDL and DML. For this reason, any class definitions or code fragments that we present will be specified in that language.

Every schema for an ONTOS database contains a group of system-defined classes, the instances of which maintain run-time accessible information about all application classes. The ONTOS standard terminology for this collection of classes is “metaschema.” (Note, however, that this collection does not truly conform to the notion of metaschema in the sense defined, e.g., in [20]. A more apt term is data dictionary.) One of the classes in the group is *Type*,<sup>1</sup> which ONTOS uses to support manifest type. An instance of *Type* represents a single application class in the schema; such an instance contains the name of its corresponding class and other relevant information. Overall, the extension of *Type* captures the entire collection of classes. Each object in an ONTOS database is inherently endowed (at creation-time) with a reference to the instance of *Type* representing its class. As such, any object can be queried directly in order to ascertain its type.

Before an ONTOS database can be populated, a “schema load” step must be carried out. This is simply the process of creating the appropriate instances of *Type* for the database’s schema which originally resides in source files in the form of C++ class declarations.

<sup>1</sup>We omit the prefix “OC.”

## 3 Adding a Semantic Relationship to an OODB System

Our methodology for incorporating a semantic relationship into an OODB system was influenced by the original ODMG standard [6, 23] as utilized, e.g., by ONTOS. In that methodology, which can be called *persistence via inheritance*, a special class is introduced (in the system library) to serve as the root of the persistent hierarchy. It defines all behavior needed to satisfy the notion of persistence. For example, the class would typically define a “put object” method that causes the target object to be written to disk. Any persistent class must be directly or indirectly derived from the root persistent class. ODMG calls this root class *Persistent\_Object*, while ONTOS calls it *OC\_Object*.<sup>2</sup>

In our methodology, a special root class provides all the functionality necessary for objects to participate in the semantic relationship of interest. For example, in the case of the part relationship [13, 14, 16], which we will be focusing on in this paper, a class called *PartWholeObject* would define methods for allowing objects (more specifically, “parts” and “wholes”) to establish, dissolve, and modify part-whole connections among themselves. It would also furnish query methods that would allow flexible retrieval of related parts and wholes. These methods would be entirely responsible for ensuring that any activity carried out with respect to the part relationship is done in accordance with the desired part semantics. As such, no burdensome hand-coding of integrity checks would fall on the shoulders of an application designer. The designer declares the intended part semantics with the assurance that the semantics will be maintained by the OODB system.

In general, for a semantic relationship  $R$ , a root class defining its associated generic behavior is added to the OODB system’s class library. Its name will have the form  $R/R'$ -*Object* to convey the fact that objects participating in the  $R$  (binary) relationship can potentially do so in the role on either side of the relationship. For the part relationship, the root class is denoted *PartWholeObject*. Objects participating in part-whole connections must be instances of classes that are derived from *PartWholeObject*. Such objects can be parts or wholes or both depending on the specific relationships. An engine, for example, is part of a car, but an engine can also have its own parts. Hence, an engine is both a part and a whole object, or, as the root’s name conveys, an engine is a “PartWholeObject.” Of course, some objects may only be parts, and others may only be wholes. In the case of the ownership relationship [42], the root class would be *OwnerOwnedObject*: Some objects play the role of owner; some play the role of owned object; and some may play both roles.

<sup>2</sup>We will forgo the prefix and just refer to it as *Object* from here on.

Because object behavior associated with a semantic relationship  $R$  is specified generically at the level of the root class, it is necessary that the database schema's  $R$  relationships (i.e., the " $R$ -schema") be run-time accessible. This allows for the proper maintenance of the semantics with respect to specific  $R$  relationships. For example, is it acceptable to attach a given engine to a given car? Will such a connection violate a declared exclusiveness constraint? Or a program might attempt to attach a door to an engine! A consultation of the  $R$ -schema will reveal that no such attachments are permitted.

The database's  $R$ -schema is made available at run-time by augmenting the OODB system's data dictionary with an additional class called  $R$ -Relationship. Each object  $r$  that is an instance of  $R$ -Relationship represents exactly one occurrence of the  $R$  relationship appearing in the schema (hence, the class's name: "R-Relationship"). Overall, the extension of  $R$ -Relationship is the entire  $R$ -schema. Each object  $r$  is, in effect, the formal description of its corresponding  $R$  relationship, containing some declarative form of its semantics. (Note that this is consistent with the common practice of OODB systems where data dictionaries often comprise classes whose instances represent the various components of the OODB schema like its classes, attributes, methods, etc.) It is not necessary that  $R$ -Relationship be an actual constituent of the system's data dictionary. In fact, it could simply be another application class that functions in the capacity of a meta-level class. We make no assumption about any special access features for it.

For the part relationship, the data dictionary class would be  $PartRelationship$ . Instances of  $PartRelationship$  would each denote a single part relationship in the schema. Such an object would contain the values of all its part relationship's dimensional data: Exclusiveness, multiplicity, dependency, and inheritance (see Section 2.1 above). We stress that in order to use our methodology for a semantic relationship  $R$ , a formal dimensional decomposition of  $R$ 's semantics is needed. Such analyses have appeared for parts [13, 14, 16], ownership [42], and materialization [21].

It is important to distinguish the purposes of the two classes,  $R/R'$ -Object and  $R$ -Relationship. The root class  $R/R'$ -Object defines functionality for objects that are designed to participate in the semantic relationship of interest; it directly provides this functionality via ordinary subclass inheritance. On the other hand, the class  $R$ -Relationship in the data dictionary maintains detailed information about each  $R$  relationship that appears in the schema. Each such relationship is represented as its own object. This differs from previous proposals [9, 35] in that actual  $R$ -connections between objects are *not* maintained as objects themselves. Only the schema-level  $R$  relationships connecting pairs of classes appear in the database in the form of objects (within the data dictionary).

In addition to the two classes, our methodology also comprises a pair of utility programs. The first program is responsible for populating the extension of the class  $R$ -Relationship. That is, it must load all information concerning  $R$  relationships from some source specification, which could be a special text file, some diagram, or the OODB system's ordinary class definition mechanism (e.g., C++ header files) augmented to include the  $R$ -schema. We rejected the latter because it implies a modification of the established syntax of class definitions, which could lead to a disruption of system-defined utilities that operate on these. As noted above, we prefer *not* to alter the customary operating environment of the system. We chose the option of using a separate text file to hold the entire  $R$ -schema. In general, the program is designated  $Load$ - $R$ -Schema. For the part relationship, it is called  $LoadPartSchema$ .

The second program is called the  $R$ -preprocessor. There are typically aspects of the semantic relationship  $R$  that must be maintained directly in the definition of classes that participate in the  $R$  relationship, but cannot be inherited from  $R/R'$ -Object. Such aspects may include  $R$ 's creation semantics and inheritance behavior, particularly when the implementation is in the context of a C++-binding [7] with the OODB system. In order to properly incorporate these, it is necessary to alter the class definitions themselves. This is done using the  $R$ -preprocessor. We do not permit these alterations to constitute any special syntactic or operational extensions. Fortunately, they are often just the additions of some new methods, which can be done in a straightforward manner. Our experience is that the preprocessor is not difficult to produce. The  $R$ -preprocessor typically needs to consult the extension of class  $R$ -Relationship. Therefore, this preprocessor step must come after the use of the  $Load$ - $R$ -Schema program.

To summarize, our methodology for the incorporation of some semantic relationship  $R$  into an existing OODB system consists of the following four components.

1. A "root" class, called  $R/R'$ -Object, from which all classes whose objects participate in the semantic relationship must be derived.
2. A data dictionary class, called  $R$ -Relationship, that is used to provide run-time accessibility to the occurrences of the semantic relationship that appear in the OODB schema.
3. A program  $Load$ - $R$ -Schema to load the semantic relationship schema information into the data dictionary (with respect to the class  $R$ -Relationship).
4. A program  $R$ -preprocessor that augments the definitions of classes which utilize the semantic relationship.

Let us note that our methodology only assumes two things about the underlying OODB system: (1) It supports *man-*

ifest type [1]. In other words, each object in the database must have the ability to identify its class when queried. (2) It permits multiple inheritance. That is, a given class can be a subclass of more than one class.

## 4 The ONTOS Part Relationship

In this section, we describe the four components that make up the implementation of the part relationship in the context of the ONTOS OODB management system. We first discuss the data dictionary class *PartRelationship* and then the program *LoadPartSchema*. After that, we describe the root class *PartWholeObject* and the Part Preprocessor. This ordering closely resembles that in which an application designer is likely to encounter the components.

### 4.1 The Class *PartRelationship*

The class *PartRelationship* enhances the ONTOS data dictionary with information about all the part relationships appearing in the given OODB schema. Each of its instances denotes exactly one part relationship from the schema. Information about a specific part relationship can be obtained at run-time by querying one of these objects.

The public interface for *PartRelationship* is shown in the following.<sup>3</sup> There we see eight methods that permit access to the values of the dimensional data for a specific part relationship. Also note that *PartRelationship* is a subclass of *Object*, making the part relationship information persistent.

```
class PartRelationship: public Object
{
public:
    char* partClassName(void);
    char* wholeClassName(void);
    exclusiveness_t exclusiveness(void);
    int minMultiplicity(void);
    int maxMultiplicity(void);
    dependency_t dependency(void);
    set_of_inherited_properties upSet(void);
    set_of_inherited_properties downSet(void);
};
```

The first two methods *partClassName* and *wholeClassName* return the names of the two classes related by the part relationship, namely, the part relationship's part class and whole class, respectively. *Exclusiveness* provides the value of the exclusiveness dimension of the part relationship. Possible values are GLOBAL\_EXCLUSIVE, CLASS\_EXCLUSIVE, and SHARED.

The methods *minMultiplicity* and *maxMultiplicity* together provide the value of the multiplicity dimension. A value of zero for the maximum multiplicity is

<sup>3</sup>For the sake of brevity, we have omitted some additional utility methods.

interpreted as infinity, meaning that there is no upper-bound restriction on the number of parts (from the specific part class) that can go into the construction of a whole (from the related whole class).

The value of the part relationship's dependency dimension is retrieved via *dependency*. Potential values are PART\_ON\_WHOLE (if the whole is deleted, the part is deleted, too), WHOLE\_ON\_PART (if the part is deleted, its whole goes as well), and NIL (indicating a lack of dependency semantics for the part relationship).

The last two methods, *upSet* and *downSet*, deal with the inheritance dimension of the part relationship. We use the term "upSet" to denote the set of properties that are inherited by a whole class from its associated part class in a particular relationship. "Up" denotes the movement from the part to the whole. The upSet of a part relationship can be empty, indicating that there is no upward inheritance with respect to that part relationship. "DownSet" is defined analogously: It is the set of properties that are inherited by the part class from the whole class. The upSet and downSet must be disjoint to avoid circular definitions.

According to our theory, each inherited property may have an associated operator that is applied when an instance-to-instance transfer of data values occurs. In order to capture this, we declare the elements of upSet and downSet to be pairs  $(p, o)$ , each consisting of a property name and an operator. The interpretation of a pair  $(p, o)$  in the upSet is as follows: Property  $p$  is inherited by the whole class from the part class, and the operator  $o$  is applied when a propagation of data occurs between a whole and its part(s) with respect to  $p$ . An element of the downSet is interpreted similarly. Presently, we provide a fixed set of potential operators for a schema designer to choose from. Theoretically, though, any symmetrical operator is a viable choice for this role [13, 15].

### 4.2 The Program *LoadPartSchema*

In ONTOS, before a database can be populated, all the information about the database's schema must be loaded into the data dictionary consisting of the class *Type*, among others. ONTOS provides a program called *classify* that performs this task. It takes as its input C++ header files containing class definitions and creates instances of *Type* (and other classes), effectively loading the entire schema and making it available at run-time.

Our program *LoadPartSchema* performs the analogous task of loading the part schema into the database. It creates one instance of *PartRelationship* for each part relationship that appears in the schema. *LoadPartSchema* is also responsible for doing consistency checks that ensure a viable part schema. The input to *LoadPartSchema* is a file, called the "part schema file," containing simple textual specifications of all part relationships. These textual specifications could

be directly extracted from a pictorial representation of the OODB schema.

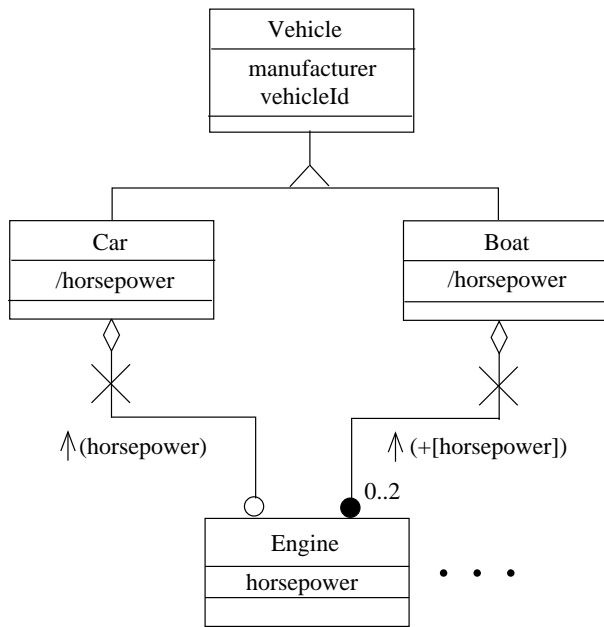


Figure 1. An example OODB schema

An example OODB schema, containing a part schema, is shown in Figure 1. The schema is drawn using a variation of the OMT notation [5, 36]. A class is a rectangle with its name inside. The class’s attributes are written beneath the name. A part relationship is a line connecting the part class with the whole class. The latter is distinguished by a diamond at its end of the line. The schema contains four classes: *Vehicle*, *Car*, *Boat*, and *Engine*. Both *Car* and *Boat* are subclasses of *Vehicle* and, therefore, inherit its two properties *manufacturer* and *vehicleId*. There are two part relationships: One between *Engine* and *Car*, and the other between *Engine* and *Boat*. Both exhibit global exclusiveness as denoted by the large X’s adorning the lines. *Car*’s relationship with *Engine* is single-valued (indicated by the hollow ball near *Engine*), meaning that a car can have no more than one engine. On the other hand, a boat can have up to two engines which is denoted by the solid ball near *Engine* and the “0..2” range value.

For conciseness, we have omitted any other classes that might have part relationships to *Car* and *Boat* (as indicated by the ellipsis). We have also kept the number of intrinsic properties low. *Vehicle* has two of them, as noted above. *Car* and *Boat* do not have any. The class *Engine* has one intrinsic property: the attribute *horsepower*. Both *Car* and *Boat* inherit *Vehicle*’s two properties via ordinary subclass inheritance. Furthermore, they both inherit *horsepower* (as denoted by the “/” preceding it) via the part relationship.

However, *Car*’s is an “invariant” inheritance: The value of the horsepower for a given car is identically the value of the horsepower for its part engine. *Boat*’s is an “additive transformational” inheritance: The horsepower of a boat is the sum of the horsepowers of its constituent engines [15]. This information is conveyed by the “horsepower” labels on the part relationship links in the figure.

### 4.3 The Class *PartWholeObject*

After declaring that a class participates in a part relationship in the part schema file, it is then necessary to derive that class from the class *PartWholeObject*, which will endow all instances of the class with the behavior required to be parts and wholes. This is true for classes that serve only as part classes (i.e., leaves of the part hierarchy), only as whole classes (i.e., roots of the part hierarchy), or as both (i.e., interior nodes of the hierarchy). In practice, we expect that most classes will appear as both part and whole classes with respect to different part relationships, with their instances playing the simultaneous roles of parts and wholes. Hence the name “PartWholeObject.” If a class is strictly a leaf (in other words, it has no further part decomposition of its own), its instances will not utilize the functionality appropriate to wholes, such as the retrieval of related parts.

The definitions of the classes from Figure 1 are given below, showing derivations from *PartWholeObject*.

```

class Vehicle: public Object
{
public:
    string manufacturer(void);
    void set_manufacturer(string aManufact);
    int vehicleId(void);
    void set_vehicleId(int aVehicleId);
};

class Car: public Vehicle, PartWholeObject
{
};

class Boat: public Vehicle, PartWholeObject
{
};

class Engine: public PartWholeObject
{
public:
    int horsepower(void);
    void set_horsepower(int aHorsepower);
};
  
```

As mentioned above, we have omitted intrinsic properties from the definitions of *Car* and *Boat*. *Vehicle* has the intrinsic properties *manufacturer* and *vehicleId*. (Note that the class definitions only display the public interfaces, namely,

the reader and writer methods for the properties.) *Engine* has the property *horsepower*. It should be noted that the inheritance of *horsepower* by *Car* and *Boat* is not reflected in the respective public interfaces of those classes at the moment. This issue will be discussed further below.

The class *Vehicle* is defined as a subclass of *Object*, meaning that its instances can be persistent. The classes *Car*, *Boat*, and *Engine* are all defined as subclasses of *PartWholeObject*. This implies that instances of those three classes can participate in part relationships and be made parts and wholes with respect to each other—in accordance with the schema. In other words, cars, boats, and engines exhibit the behavior of parts and wholes. *Car* and *Boat* are additionally subclasses of *Vehicle* and inherit its properties.

Due to the subclass relationships between *Car* and *Vehicle* and between *Boat* and *Vehicle*, we could alternately declare *Vehicle* itself to be a subclass of *PartWholeObject*. In that case, the direct derivations of *Car* and *Boat* from *PartWholeObject* would be unnecessary as the two classes would inherit their part/whole capabilities via *Vehicle*. We chose the above specification to demonstrate what the multiple inheritance from *PartWholeObject* and another class would look like. Moreover, if one defines *Vehicle* as a subclass of *PartWholeObject*, then all vehicles would have the potential of being decomposed into parts within the OODB. While that is fine for the schema as shown, it may not be desired if the schema were expanded to include other kinds of vehicles. For example, one may wish to include motorcycles in the database but not maintain their explicit part decompositions. In that situation, there is no reason to endow motorcycles with part/whole functionality.

The public interface for *PartWholeObject*, which contains six methods, is as follows:

```
class PartWholeObject: public Object
{
public:
    Bool addPart(PartWholeObject *aPart);
    Bool removePart(PartWholeObject *aPart);
    Bool replacePart(PartWholeObject *oldPart,
                    PartWholeObject *newPart);
    set_of_PartWholeObject getParts(void);
    set_of_PartWholeObject getWholes(void);
    void deleteObject(Bool deallocate);
};
```

Note that we define *PartWholeObject* as a subclass of *Object*. As such, all objects that are parts and/or wholes are persistent. In the following, we describe the details of each of these methods.

The method `addPart` connects the given part “aPart” to the target object. Note that the parameter is declared to be a reference to a general part/whole object, i.e., an object that has the capability of functioning in such a role. It is the responsibility of `addPart` to ensure that the type of the

given object is consistent with the part schema. For example, it should not allow a car to be connected to a boat. The method makes its decision about compatibility by querying the involved whole and (potential) part (here is where the manifest type is utilized) as well as the instances of the class *PartRelationship*. In particular, it checks for the existence of a part relationship whose part and whole classes match those of the objects involved. If it cannot find such a part relationship, then it aborts the transaction and signals a failure to the caller. Even if the two objects are compatible, it is still possible that the attachment of the part would lead to a violation of either an exclusiveness or a maximum multiplicity constraint. These, too, are checked for by consulting the appropriate instances of *PartRelationship*. The details of the algorithm that `addPart` implements can be found in [13].

The “add part” operation could be reasonably attached to the part object or the whole object, or maintained globally. Our decision to place it exclusively with the whole object—without loss of generality—comes from our view that the construction of objects with respect to part hierarchies is inherently a bottom-up process: Integral wholes are built up from lower-level component parts. The converse operation “add whole” is invoked implicitly by `addPart`.

`RemovePart` complements `addPart` as the means for dissolving a connection between a whole and a part. In the case that the given part is not actually attached to the target whole, the request is ignored, and the caller is made aware of the failure. The only true failure in this context occurs when an attempt is made to violate the minimum multiplicity specification of the respective part relationship. Again, this condition is detected by querying the part relationship object in the data dictionary.

If a part relationship has identical minimum and maximum multiplicities, then an attempted removal of such a part is certain to fail. The minimum multiplicity will be breached. Therefore, in that circumstance, it is impossible to carry out the often desirable task of replacing one part of the type with another (with respect to some whole). To overcome this problem, it is necessary to employ the method `replacePart`, which removes the old part and replaces it with the new part (of the same type) in a single transaction. Of course, the part replacement could fail if the new part is already exclusively owned by another whole.

The part connections that a given object is involved in can be queried with the use of the methods `getParts` and `getWholes`. Each is defined to return a set of part/whole objects. The method `getParts` returns all the immediate parts of the target object. If it has no parts, `getParts` returns the empty set. To obtain a “parts explosion” of an object to a certain depth, this method can be applied recursively. `GetWholes` functions analogously.

The last method `deleteObject` is defined in the class

*Object* and is overridden here. It is invoked with respect to an object when the object is to be deleted from the database. `DeleteObject` encodes the combined deletion semantics of all the various part relationships that the object can participate in. More specifically, it must: (1) ensure that the deletion of the object does not violate a minimum multiplicity constraint, leaving some whole without a required part; and (2) propagate the deletion of the target object into the deletion of all other objects that are dependent on it, as specified by the dependency dimensions of any relevant part relationships [13].

#### 4.4 The Part Preprocessor

The final component of the part relationship software for ONTOS is a preprocessor that operates on the class definitions (i.e., C++ header files) of an application. The preprocessor has two primary chores that involve the augmentation of the definitions of classes participating in the part hierarchy: (1) The inclusion of code that ensures the legitimate creation of parts and wholes; (2) The inclusion of reader methods for the properties inherited via part relationships. We discuss these two issues in the following.

##### 4.4.1 Creating Parts and Wholes

The semantics of part relationships must be maintained throughout the entire life-time of any object starting at its “birth.” The creation semantics of the part relationship comprises two major constraints. The first concerns the multiplicity dimension: No whole should initially have too few or too many parts of a given type. The second constraint involves exclusiveness: No whole should initially have a part that is already exclusively held by another whole.

In the C++/ONTOS environment, it is difficult, if not impossible, to extend the ordinary object-creation facility in order to enforce correct part/whole-creation semantics. Instead, the preprocessor directly installs an “object generation” (static) method `make` in each class of the part hierarchy. This method’s parameters are defined to be those of the class’s constructor plus one for the initial set of parts for the new object. When invoked, `make` creates an object that is guaranteed to satisfy the part semantics right at the outset; if it detects a potential violation, then it aborts the instantiation and returns `NULL` to indicate failure.

The following demonstrates the use of `make` for the classes *Engine*, *Car*, and *Boat*.<sup>4</sup>

```
(1) Engine *eng1 = Engine::make(300, {});
(2) Engine *eng2 = Engine::make(1250, {});
(3) Engine *eng3 = Engine::make(1250, {});
```

<sup>4</sup>The sets of parts included as arguments to `make` would technically need to be instantiated. We omit this and just use the customary set notation.

```
(4) Car *car = Car::make("Chevrolet", 7417,
                        {eng1});
(5) Boat *boat = Boat::make("Hatteras",
                            98568, {eng2, eng3});
```

At line (1), an instance of *Engine* is created having a horsepower of 300. The empty braces denote the empty set, meaning that no parts are to be installed in the new engine initially. (Our sample schema indicates no part decomposition for *Engine*. Any parts passed to `make` in this context would lead to a failure.) Lines (2) and (3) each show the creation of a new engine having 1,250 horsepower. Line (4) creates a new instance of *Car* (with manufacturer Chevrolet and vehicle ID 7417), and installs “eng1” as its part. At line (5), a boat is created having the two engines “eng2” and “eng3” as parts (and Hatteras as its manufacturer and 98568 as its ID).

If line (5) was written as:

```
Boat *boat = Boat::make("Hatteras",
                        98568, {eng1, eng3});
```

then the creation of the new boat would have failed due to the fact that “eng1” is currently installed in the car. `make` would signal this by returning `NULL`.

##### 4.4.2 Part-Whole Inheritance

A property inherited via a part relationship should be accessible in the same way that an intrinsic property is. The application programmer should see no distinction between the two. For example, we would like to obtain the value of property *horsepower* for the car we created above as follows.

```
car->horsepower();
```

The problem is that no reader method for *horsepower* appears in the public interface for *Car*, even though the inheritance was declared in the part schema file. Therefore, the compiler will flag this statement as an error.

To avoid this problem, the preprocessor augments the definitions of any classes that have been declared to receive properties via part inheritance. After the part schema load step, the preprocessor is called upon to scan the entire extension of *PartRelationship* looking for inheritance situations. When it finds one, it goes to the proper header file and adds an appropriate reader method. In general, the form of such a method for an inherited property looks like:

```
<property type> <property name>(void)
{
    // Computation and return of value here.
};
```

The method's name which is identical to the name of the inherited property is obtained directly from the part schema. The same is true for the required computation, which we show above in a comment. The return type of this method (i.e., the inherited property's type) is obtained by examining the source property. Note that access to inherited property values is done "lazily" (i.e., on demand).

In the following, we demonstrate the way the inherited property *horsepower* is accessed for a car and a boat. It will be noted that it is exactly the same as accessing ordinary properties.

```
(1) cout << car->horsepower();  
(2) cout << boat->horsepower();
```

At line (1), the value printed is 300, the value of the horsepower of "eng1" which is currently installed as car's part. On the other hand, line (2) produces 2,500 because the boat has two engines, "eng2" and "eng3," and the value is defined to be the sum of the horsepowers of the constituent engines.

## 5 Conclusion

We have presented a methodology for incorporating semantic relationships into an existing OODB system that was not originally built to support them. The methodology is valid for most target OODB systems; it assumes only that the system supports multiple inheritance and manifest type. One other assumption is an existing formal dimensional characterization of the semantic relationship of interest.

In order to demonstrate the viability of our approach, we presented the details of integrating a part-whole semantic relationship into the ONTOS OODB system. At present, that implementation is up and running, and a sample application is available on the Web [32].

One area that requires further investigation is schema evolution. Currently, for example, all part relationship meta-data is immutable. Any tampering with that data by an application program could lead to a breakdown in the maintenance of part semantics for the entire database. Allowing the part schema to evolve, whether in terms of the addition of new part relationships or the modification of the dimensional values of existing ones, may be desirable. Another enhancement would be the automatic generation of various components of the framework from some declarative specification of the desired semantic relationship. For example, it may be possible for a program to automatically create the entire "root" class from such a specification.

## Acknowledgments

We'd like to thank James Geller and Yehoshua Perl for suggesting the project and reading drafts of this paper.

Thanks also to Craig Harris and Rula Asfour for their help with ONTOS.

## References

- [1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, 1985.
- [2] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. *VLDB Journal*, 4(3), 1995.
- [3] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proc. VLDB '91*, pages 565–575, 1991.
- [4] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-whole relations in object-centered systems: An overview. *Data & Knowledge Engineering*, 20(3):347–383, Nov. 1996.
- [5] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, Upper Saddle River, NJ, 1998.
- [6] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Francisco, CA, 1994.
- [7] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.
- [8] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, NJ, second edition, 1991.
- [9] O. Diaz and P. M. Gray. Semantic-rich user-defined relationships as a main constructor in object-oriented databases. In *Proc. IFIP TC2 Conf. on DB Semantics*. North Holland, 1990.
- [10] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Co., Inc., New York, NY, 1989.
- [11] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to the entity-relationship model. *Int'l J. Data and Knowledge Eng.*, 1(1), May 1985.
- [12] J. Geller, Y. Perl, and E. Neuhold. Structure and semantics in OODB class specifications. *SIGMOD Record*, 20(4):40–43, Dec. 1991.
- [13] M. Halper. *A Comprehensive Part Model and Graphical Schema Representation for Object-Oriented Databases*. PhD thesis, NJIT, Oct. 1993.
- [14] 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.
- [15] M. Halper, J. Geller, and Y. Perl. Value propagation in OODB part hierarchies. In B. Bhargava, T. Finin, and Y. Yesha, editors, *CIKM-93, Proc. 2nd Int'l Conference on Information and Knowledge Management*, pages 606–614, Washington, DC, Nov. 1993.
- [16] M. Halper, J. Geller, and Y. Perl. An OODB part-whole model: Semantics, notation, and implementation. *Data & Knowledge Engineering*, 27(1):59–95, May 1998.

- [17] M. Halper, J. Geller, Y. Perl, and W. Klas. Integrating a part relationship into an open OODB system using metaclasses. In N. Adam, B. Bhargava, and Y. Yesha, editors, *CIKM-94, Proc. 3rd Int'l Conference on Information and Knowledge Management*, pages 10–17, Gaithersburg, MD, 1994.
- [18] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Trans. Database Syst.*, 6(3):351–386, 1981.
- [19] W. Klas. *A Metaclass System for Open Object-Oriented Data Models*. PhD thesis, Technical University of Vienna, January 1990.
- [20] W. Klas. Tailoring an object-oriented database system to integrate external multimedia devices. In *Workshop on Heterogeneous DBs & Semantic Interoperability*, Boulder, CO, 1992.
- [21] M. Kolp. A metaobject protocol for reifying semantic relationships into reflective systems. In *Proc. 4th Doctoral Consortium of the 9th Int'l Conf. on Advanced Information Systems Engineering (CAiSE'97)*, pages 89–100, Barcelona, Spain, June 1997.
- [22] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.
- [23] M. E. S. Loomis. *Object Databases: The Essentials*. Addison-Wesley Publishing Co., Reading, MA, 1995.
- [24] B. MacKellar and J. Peckham. Representing design objects in SORAC: A data model with semantic objects, relationships and constraints. In *AI in Design '92*, Pittsburgh, PA, 1992.
- [25] E. Mays, C. Apte, J. Griesmer, and J. Kastner. Experience with K-Rep: An object-centered knowledge representation language. In *Proc. IEEE AI Application Conference*, San Diego, CA, Mar. 1988.
- [26] R. Motschnig-Pitrik and V. C. Storey. Modelling set membership: The notion and the issues. *Data & Knowledge Engineering*, 16:145–185, 1995.
- [27] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *TOIS*, 8(4):325–362, 1990.
- [28] N. F. Noy and C. D. Hafner. The state of the art in ontology design: A survey and comparative review. *AI Magazine*, 18(3):53–74, Fall 1997.
- [29] Welcome to ODI. URL: <http://www.odi.com>.
- [30] ONTOS Home Page. URL: <http://www.ontos.com>.
- [31] ONTOS, Inc. Lowell, MA. *ONTOS DB/Explorer 4.0 Reference Manual*, 1996.
- [32] The world of parts. URL: <http://object.njit.edu:2000/~part>.
- [33] N. W. Paton. ADAM: An object-oriented database system implemented in Prolog. In *Proc. 7th BNCOD*, 1989.
- [34] A. Pirotte, E. Zimányi, D. Massert, and T. Yakusheva. Materialization: A powerful and ubiquitous abstraction pattern. In *Proc. VLDB'94*, pages 630–641, Santiago, Chile, 1994.
- [35] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proc. OOPSLA-87*, pages 466–481, Oct. 1987.
- [36] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [37] V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, Object-Store, and O<sub>2</sub>. *SIGMOD Record*, 21(1):93–104, Mar. 1992.
- [38] V. C. Storey. Understanding semantic relationships. *VLDB Journal*, 2(4):455–488, 1993.
- [39] UML document set. URL: <http://www.rational.com/uml/-references>.
- [40] Versant. URL: <http://www.versant.com>.
- [41] W. A. Woods. What's in a link: Foundations for semantic networks. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 218–241. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1985.
- [42] O. Yang, M. Halper, J. Geller, and Y. Perl. The OODB ownership relationship. In *Proc. Int'l Conf. on Object-Oriented Information Systems (OOIS'94)*, pages 389–403, London, UK, Dec. 1994.