CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:1337–1362 (DOI: 10.1002/cpe.733)

Frameworks for incorporating semantic relationships into object-oriented database systems

Michael Halper[1,*,†], Li-min Liu[2], James Geller[3] and Yehoshua Perl[3]

[1] *Department of Mathematics and Computer Science, Kean University, Union, NJ 07083, U.S.A.*
[2] *Department of Applied Mathematics, Chung Yuan Christian University, Chung-Li, Taiwan, Republic of China*
[3] *Computer Science Department, New Jersey Institute of Technology, Newark, NJ 07102, U.S.A.*

SUMMARY

A semantic relationship is a data modeling construct that connects a pair of classes or categories and has inherent constraints and other functionalities that precisely reflect the characteristics of the specific relationship in an application domain. Examples of semantic relationships include part–whole, ownership, materialization and role-of. Such relationships are important in the construction of information models for advanced applications, whether one is employing traditional data-modeling techniques, knowledge-representation languages or object-oriented modeling methodologies. This paper focuses on the issue of providing built-in support for such constructs in the context of object-oriented database (OODB) systems. Most of the popular object-oriented modeling approaches include some semantic relationships in their repertoire of data-modeling primitives. However, commercial OODB systems, which are frequently used as implementation vehicles, tend not to do the same. We will present two frameworks by which a semantic relationship can be incorporated into an existing OODB system. The first only requires that the OODB system support manifest type with respect to its instances. The second assumes that the OODB system has a special kind of metaclass facility. The two frameworks are compared and contrasted. In order to ground our work in existing systems, we show the addition of a part–whole semantic relationship both to the ONTOS DB/Explorer OODB system and the VODAK Model Language. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: semantic relationship; object-oriented database; object-oriented modeling; part–whole relationship; metaclass

1. INTRODUCTION

Semantic relationships—binary associations between classes or categories of objects that are more than mere 'named links'—are fundamental to human cognition and reasoning. Examples of semantic

[*] Correspondence to: Michael Halper, Department of Mathematics and Computer Science, Kean University, Union, NJ 07083, U.S.A.
[†] E-mail: mhalper@kean.edu

Published online 22 September 2003

Received 15 February 2002
Revised 28 October 2002
Accepted 2 November 2002

relationships are the part–whole relationship [1–5], ownership [6–9], materialization [10,11] and the role-of relationship [12–15]. Due to their importance in human thought, they also play a pivotal role in the construction of information models for applications. In addition to serving as a link between a pair of classes, a relationship of this kind carries inherent semantics in the form of constraints and other functionalities—such as inheritance, operation propagation or specialized query capabilities—that allow a given enterprise to be modeled more precisely [16,17]. This is the case whether one is using traditional data-modeling techniques like SDM [18] and extended ER [19,20], knowledge-representation languages such as Telos [21] and K-Rep [22] or object-oriented modeling methodologies including OMT [23], Booch [24,25], UML [26–28] and Coad/Yourdon [29].

In this paper, we focus on the issue of bringing the power and expressiveness of semantic relationships to object-oriented database (OODB) systems [30–37]. The reason for concentrating on such systems is the popularity of object-oriented modeling techniques and the use of OODBs as their implementation vehicles for persistent storage. While the popular object-oriented modeling methodologies, like OMT, UML and Coad/Yourdon, include semantic relationships as modeling constructs, 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 the precise modeling of complex applications.

We present two different frameworks for incorporating semantic relationships into the repertoire of built-in modeling primitives provided by OODB systems. These frameworks, in effect, extend the underlying object model to include additional semantics, bringing the OODB system more closely in line with the various object-oriented analysis and design methodologies. An application designer wishing to use a semantic relationship is not burdened with the task of having to hand-code constraints, integrity checks and other behavioral elements associated with the relationship. Instead, the designer simply specifies the semantic relationship in a declarative manner (either textually or pictorially) in the database schema. Thereafter, it is solely the responsibility of the OODB system to ensure that the proper semantics of that relationship are maintained throughout the entire lifetime of the database. For example, a 'part' or 'whole' object will exhibit the appropriate behavior from the moment it is instantiated by one application until the time it is eventually deleted by another.

The first framework can be utilized for most existing commercial OODB systems, e.g. ONTOS DB/Explorer ('ONTOS' for short) [38,39], ObjectStore [39–41] and Versant [42]. It only demands that the host OODB system support *manifest type* [43]. In other words, each object in the database must have the ability to identify its class when queried. The framework primarily exploits the standard OODB subclass inheritance mechanism to define all additional behavior befitting the semantic relationship of interest. For example, objects that might participate in the relationship are given the means for establishing, dissolving and modifying occurrences of the relationship. Built into these capabilities is the guarantee that the proper semantics will be enforced with respect to any transaction. Furthermore, objects are given the ability to respond to queries concerning the relationship, allowing for the retrieval of related objects.

The second framework has the more stringent requirement that the OODB system support the kind of metaclass facility described in [44,45]. This framework utilizes a special metaclass to augment application classes and their instances in such a way as to give them the additional functionality needed for the semantic relationship of interest.

Both frameworks were created to satisfy the two following major conditions. (1) They should not cause any upheaval in the underlying OODB system. That is, there should be no need to rewrite a portion of the system. (2) They should not alter the environment that an application developer is used to working in. For example, it must not introduce exotic syntax into the preferred data manipulation language of the OODB system.

The only other assumption made by our frameworks is in regard to the properties of the semantic relationship. Specifically, we assume that it can be formally described in terms of *characteristic dimensions*, each of which captures one aspect of its nature [2,5,46]. In this way, the semantics of the relationship can be specified in an entirely declarative fashion, allowing for complete inclusion within the OODB schema. Such analyses have previously appeared for part–whole [2], ownership [6,9], materialization [10,11], and can be done for the role-of relationship [12–15].

To ground our work, we will demonstrate the use of the two frameworks in incorporating the part relationship [1–5] into existing OODB systems. We will use the first to introduce the part relationship into ONTOS. The second will allow us to provide the VODAK Model Language (VML) [45,47,48] with the relationship. Both these implementations have been completed and the metaclass for the part relationship has been distributed with VML's standard metaclass library. The two implementations serve to demonstrate the viability of our approach. A semantic relationship can indeed be added to an OODB system without causing an upheaval and without altering the customary development environment. They also show that OODB systems can fulfill the promises of enhanced modeling capabilities.

In related work in this area, a meta-object protocol has been used for the inclusion of semantic relationships in CLOS [10]. An extensive discussion of semantic relationships and their role in data modeling can be found in [16]. The set-membership relationship [49] 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 [50] and was extended in [51] and [52], both of which permit additional constraints on the user-defined relationships. The relationship construct of [51] has been included in the Fibonacci language [53], while that of [52] has been implemented in ADAM [54], a Prolog-based OODB system. The SORAC model [55] utilizes relationships as a means for specifying constraints on designs in a knowledge-based/object framework. In this paper, we present and compare a pair of general frameworks that can be utilized for the incorporation of semantic relationships into a wide range of available OODB management systems. A preliminary presentation of the first framework has previously appeared in [56]. An earlier treatment of the second framework appeared in [57].

The rest of this paper is organized as follows. In Section 2, we discuss the general structure that a semantic relationship must have in order for it to be amenable to our frameworks. We also give a description of the part relationship. The first framework is introduced in Section 3 and its utilization for the incorporation of a part relationship into ONTOS is discussed in Section 4. Section 5 presents the details of our second framework. The application of the second framework for the VML part relationship is described in Section 6. Conclusions appear in Section 7.

2. FORMAL STRUCTURE OF A SEMANTIC RELATIONSHIP

In order for a given semantic relationship to be amenable to our frameworks, it must be formally describable in terms of a collection of what we call characteristic dimensions. Each of these dimensions formally denotes the semantics of one aspect of the relationship. For example, in the case of the part relationship (which we will describe further below), the *exclusiveness* dimension handles the exclusiveness constraints associated with parts and wholes; the *inheritance* dimension deals with the relationship's property inheritance aspect.

In general, we assume an occurrence R_1 of a semantic relationship R can be written as follows:

$$R_1 = (d_1, d_2, \ldots, d_n) \tag{1}$$

where $d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n$ and $D_1, D_2, \ldots, D_n$ are the domains of the respective n characteristic dimensions of the generic semantic relationship R. The d_i together denote the entire semantics for the occurrence R_1 of the semantic relationship R. For example, d_1 might denote the exclusiveness constraint that R_1 must exhibit, while, say, d_3 is a specific property to be inherited via R_1.

The part–whole semantic relationship (part relationship, for short) is composed of four characteristic dimensions: (1) *exclusiveness*; (2) *multiplicity*; (3) *dependency*; and (4) *inheritance*. The formal structure of a part relationship between the 'part class' B and the 'whole class' A is as follows [2]:

$$P_{B,A} = (\chi, \kappa, \delta, (\Upsilon, \Delta, \Phi))$$

where χ, κ, δ and (Υ, Δ, Φ) are the values of the four respective dimensions. It should be noted that the value of the final dimension, inheritance, is defined to be a triple itself. Instead of formally describing the interpretations of the various dimensions, we will briefly discuss each of them in an informal fashion in the following. A formal presentation can be found in [2].

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 [2,58], 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. As another example, journal articles may be constrained with a '(3, 10)' section multiplicity, meaning that any article must have at least three sections and at most ten.

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 the bicycle should also be deleted. In the other direction, the deletion of a large CAD/CAM structure in its entirety may imply the deletion of the structure's constituent objects.

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 [59,60], 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

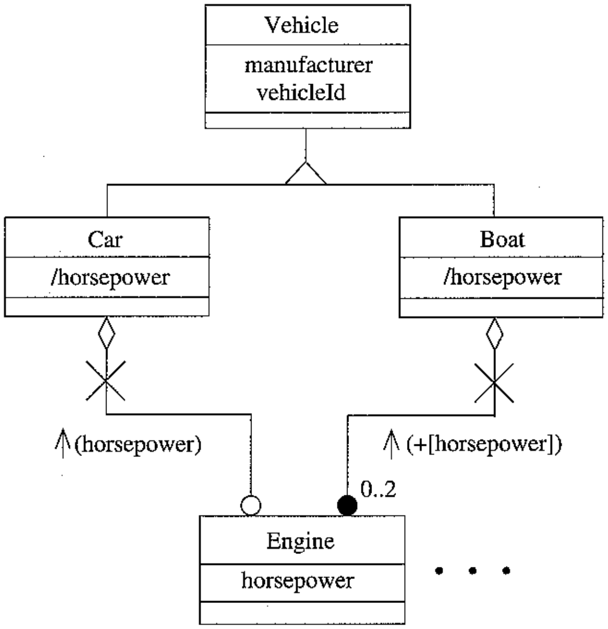

Figure 1. Example OODB schema with part relationships.

engine that happens to be installed in it. Moreover, if the boat is capable of having multiple engines, then its overall horsepower is the sum of the horsepowers of its engines.

An example OODB schema, containing a part schema, is shown in Figure 1. The schema is drawn using a variation of the OMT notation [23,61]. A part relationship is a line connecting the part class with the whole class. The whole class is distinguished by a diamond at the end of its 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 open circle 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 full circle 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, 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 their respective part relationships. The labels '↑(horsepower)' and '↑(+[horsepower])' on the part relationships will be explained in Section 4.1.

3. THE FIRST FRAMEWORK FOR ADDING A SEMANTIC RELATIONSHIP TO AN OODB SYSTEM

Our first framework for incorporating a semantic relationship into an OODB system was influenced by the original Object Database Management Group (ODMG) standard [32,36] as utilized, for example, by ONTOS. In that arrangement, which can be called *persistence via inheritance*, a special 'root class' is introduced (e.g. in the system library) to serve as the root of the persistent hierarchy [62]. It defines all the 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 the database. Any class whose instances are meant to be persistent must be derived, either directly or indirectly, from the persistent root class. Objects, therefore, acquire their persistence (behavior) via inheritance from this root. ODMG calls this root class *Persistent_Object*, while ONTOS calls it *OC_Object*[‡] [38].

In our framework, 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 [1–5], whose implementation we will be focusing on, a class called *PartWholeObject* would define the behavior (methods) necessary for 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 the 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 *RR'Object*, where R' denotes the converse relationship of R. The name is used to convey the fact that objects participating in the R (binary) semantic 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 also has 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 [9], 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.

The class *RR'Object* and its relation to a pair of application classes, A and B, connected via the semantic relationship R is shown in Figure 2 using an OMT-like notation. (The R relationship connecting B to A is drawn as a dashed arrow labeled with 'R'.) The class *RR'Object* defines the set

[‡] We will forgo the prefix and just refer to it as *Object* from here on.

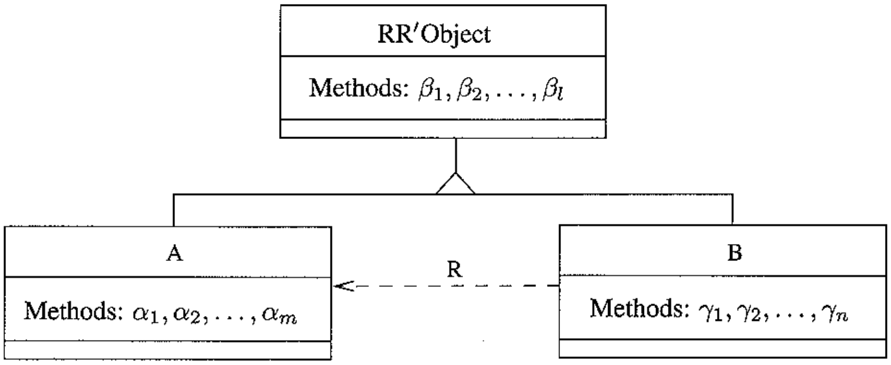
RR′Object
Methods: $\beta_1, \beta_2, \ldots, \beta_l$
A
Methods: $\alpha_1, \alpha_2, \ldots, \alpha_m$
R
B
Methods: $\gamma_1, \gamma_2, \ldots, \gamma_n$

Figure 2. The root class *RR′Object* and its relation to application classes participating in semantic relationship R.

of behaviors (methods) $\beta = \{\beta_1, \beta_2, \ldots, \beta_l\}$ that is required for participation in the relationship R. Therefore, both A and B must be defined as subclasses of *RR′Object* in order for them to inherit the set of methods β and, in turn, endow their respective instances with the appropriate behavior for participating in the R relationship. This behavior augments the intrinsically defined sets of behaviors $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ and $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ of A and B, respectively. We show an example of this scenario in Figure 3, where instances (objects) of A and B are drawn as rounded rectangles. Each instance of B is connected to one or more instances of A via occurrences of the relationship R (drawn as wavy arrows). Such connections are established via a method in β, provided originally by *RR′Object*. In fact, all manipulations of such connections (like creating, deleting, updating and querying) are defined by β.

Of course, in a database design tool it would not be necessary to explicitly show the subclass relationships connecting the application classes A and B to the root *RR′Object* as in Figure 2. These subclass relationships can be inferred from the R relationship linking B to A.

Because the object behavior associated with a semantic relationship R is specified generically at the level of the root class *RR′Object*, it is necessary that the database schema's entire set of occurrences of the R relationship be run-time accessible. (We will use the term 'R-schema' to denote the entire set of occurrences of the R relationship or, equivalently, that subschema which is the edge-induced subgraph [63,64] of the overall OODB schema with respect to R.) 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? 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 R relationship appearing in the application schema is represented by an object r that is an instance of *R-Relationship* in the data dictionary (hence the class's name). Such an object r is, in effect, the formal description of its

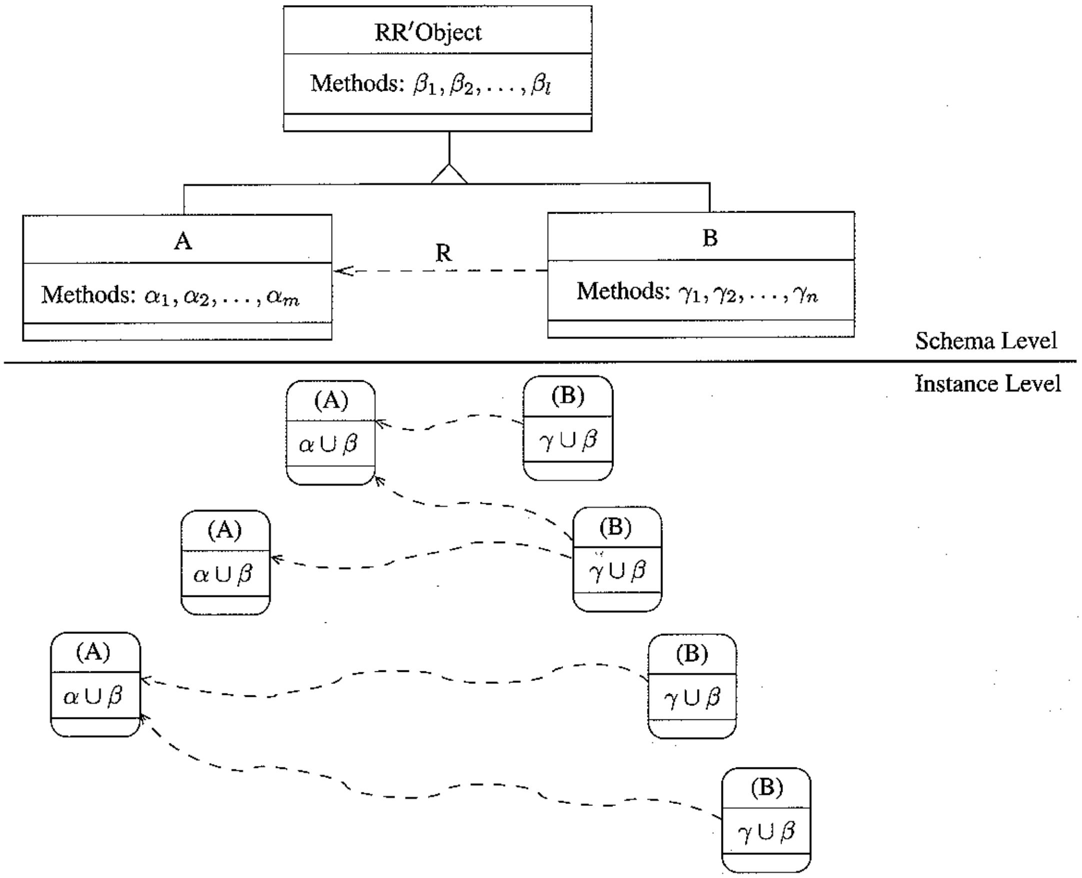
RR′Object
Methods: $\beta_1, \beta_2, \ldots, \beta_l$
A
Methods: $\alpha_1, \alpha_2, \ldots, \alpha_m$
R
B
Methods: $\gamma_1, \gamma_2, \ldots, \gamma_n$
Schema Level
Instance Level
(A)
$\alpha \cup \beta$
(B)
$\gamma \cup \beta$

Figure 3. Instances of application classes *A* and *B* connected via relationship *R*.

corresponding *R* relationship, containing some declarative form of the relationship's semantics[§]. For example, *r* would have the properties *source-class* and *target-class* denoting, respectively, the source class and target class of the relationship. Additionally, *r* would have separate properties—and associated access methods—to hold each characteristic dimension value d_i from Equation (1). As an illustration, we show in Figure 4 the mapping of the *R* relationship connecting class *B* to class *A* (Figure 2) into its data dictionary representation. The dotted arrow denotes 'is an instance of'.

[§] 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. ONTOS, for example, has a data dictionary of this form.

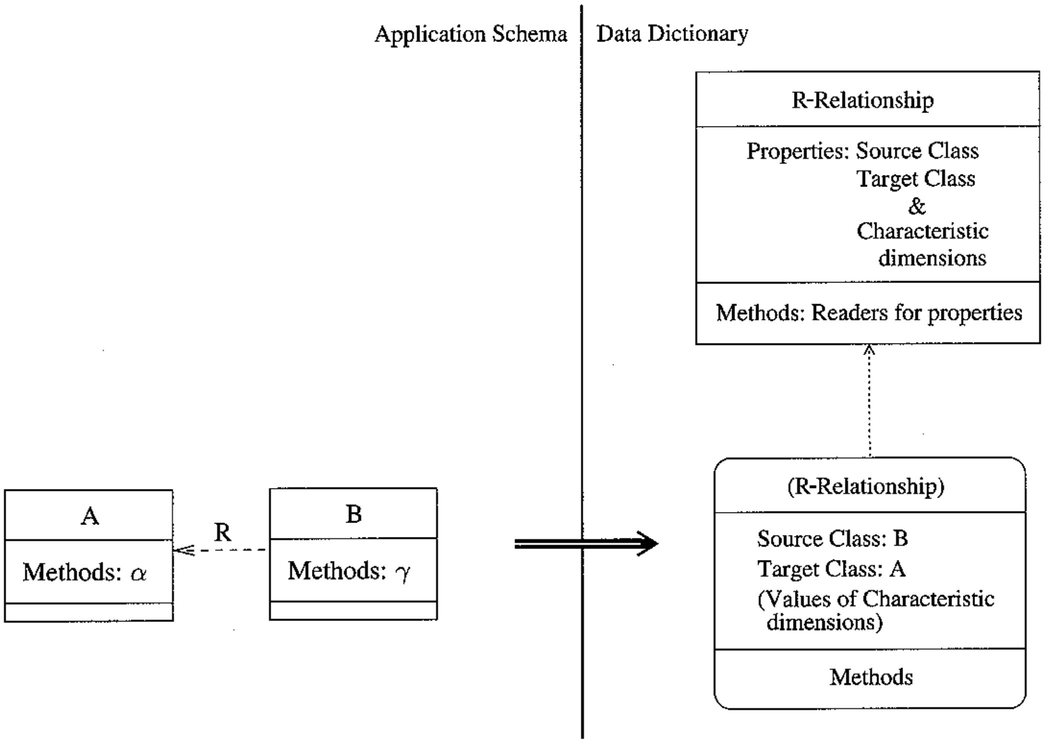

Figure 4. Representation of the R relationship from B to A as an object in the data dictionary.

Overall, the extension of *R-Relationship* is the entire R-schema. 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 example, in the case of the part relationship, the data dictionary class would be *PartRelationship*. Instances of *PartRelationship* would each denote a single occurrence of the 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). 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 [2], ownership [6,9] and materialization [10,11].

It is important to distinguish the purpose of the two classes *RR′Object* and *R-Relationship*. The root class *RR′Object* defines additional behavior (and implicit structure) for objects that are designed to participate in the semantic relationship of interest; it directly provides this functionality via ordinary subclass inheritance (see Figure 2). On the other hand, the class *R-Relationship* in the data dictionary maintains detailed information about each R relationship in the OODB schema using a separate instance (see Figure 4). Overall, the extension of *R-Relationship* is the R-schema of the entire OODB schema. This arrangement differs from previous proposals (e.g. [50,52]) 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). This avoids the proliferation of objects that are required to represent the numerous object-to-object (i.e. instance-level) occurrences of the various R relationships. An R-connection between a pair of objects is maintained within the scope of those objects by utilizing the behavior and structure provided by *RR′Object*.

In addition to the classes *RR′Object* and *R-Relationship*, the framework also requires a pair of utility programs to complete the integration. The first program, called *LoadRSchema*, is responsible for populating the extension of the class *R-Relationship* before the actual underlying OODB—corresponding to the schema—can itself be populated. That is, it must load all the information concerning R relationships from some source schema specification, which could be a special text file or a diagram. For simplicity, we chose the text-file option in our integration of the part relationship into ONTOS, described in the next section. (Of course, a simple extension to a diagramming tool like ObjectMaker [65] could serve the same purpose.) Additionally, the *LoadRSchema* program is responsible for guaranteeing the integrity of the configuration of the R relationships appearing in the schema. For example, a cycle in the R-schema may not be permitted. It would be the job of *LoadRSchema* to test for this condition and raise an error if it were detected.

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 *RR′Object*. Such aspects may include R's creation semantics and inheritance behavior, particularly when the implementation is in the context of a C++-binding [33] with the OODB system. In order to properly incorporate these, it may be 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 addition 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 is defined to come after the use of the *LoadRSchema* program.

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

1. A 'root' class, called *RR′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.

Additionally, a pair of support programs, *LoadRSchema* and *R-preprocessor*, will also be required. Note that this first framework only assumes that the underlying OODB system supports manifest type [43]. In the next section, we will show how this approach allows for the inclusion of useful part–whole functionality into the OODB system. For example, we will see how a whole object can retrieve all its parts, or how an attribute can be inherited along a part link.

4. ONTOS PART RELATIONSHIP

In this section, we describe the specific components that make up the implementation of the part relationship in the context of the ONTOS OODB management system. We present the data dictionary class *PartRelationship* and the program *LoadPartSchema*. Next, we describe the root class *PartWholeObject* and the Part Preprocessor. This closely resembles the ordering in which an application designer is likely to utilize the components as an application is being built.

ONTOS uses C++ as its primary data definition and manipulation language. For this reason, any class definitions or code fragments that we present will be specified in C++. Every ONTOS database contains a data dictionary comprising a group of system-defined classes, the instances of which maintain run-time accessible information about all application classes. One of the classes in this group is *Type*[¶], 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* is the entire set of classes in the schema. 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 directly queried in order to retrieve its type.

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 of the schema's part relationships (see Figure 4). Information about a specific part relationship between two classes can be obtained at run-time by querying one of these objects.

The public interface for *PartRelationship* is shown below[‖]. 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,

[¶] We omit the prefix 'OC'.

[‖] For the sake of brevity, we have omitted some additional utility methods.

respectively. Exclusiveness provides the value of the exclusiveness dimension of the part relationship. Possible values are GLOBAL_EXCLUSIVE, CLASS_EXCLUSIVE and SHARED [2].

minMultiplicity and maxMultiplicity return the values of the multiplicity dimension. A value of zero for the maximum multiplicity is interpreted as infinity, meaning that there is no upper-bound restriction on the number of parts (from the specific part class) that can relate to 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).

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. The term '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. The term 'downSet' is defined analogously: it is the set of properties that are inherited by the part class from the whole class, upSet and downSet must be disjoint to avoid circular definitions.

According to our theory, each inherited property has 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_name*, *op*), each consisting of a property name *p_name* and an operator *op*. The interpretation of a pair (*p_name*, *op*) in the upSet is as follows. The property *p_name* is inherited by the whole class from the part class and the operator *op* is applied when a propagation of data occurs to a whole from its part(s) with respect to *p_name*. An element of the downSet is interpreted similarly. Presently, we provide a fixed set of potential operators for a schema designer to choose from (e.g. arithmetic sum). Theoretically, though, any symmetrical operator is a viable choice for this role [2,60].

As an example, let us refer back to Figure 1. The classes *Car* and *Boat* both inherit the property *horsepower* through their respective part relationships with *Engine*. However, *Car*'s is an 'invariant' inheritance: the value of the horsepower for a given car is identical to the value of the horsepower for its part engine. This information is conveyed by the label '↑(horsepower)' on the part relationship, with '↑' denoting the direction of the inheritance. In this case, the operator *op* is simply the identity operator for the data type of *horsepower* at *Engine*. In other words, no transformation of the value takes place. This is denoted by the absence of any operator symbol. In contrast, *Boat*'s is an 'additive transformational' inheritance: the horsepower of a boat is the sum of the horsepowers of its constituent engines [60]. In this case, the operator is denoted by the '+' prefixing '[horsepower]' in the label on the part relationship between *Engine* and *Boat*.

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—as data—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 the simple textual specifications of all part relationships.

4.2. The class *PartWholeObject*

After declaring in the part schema file that a class *A* participates in a part relationship, it is then necessary to derive *A* from the class *PartWholeObject*, which will endow all instances of *A* with the behavior required to be parts or 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 a significant percentage of the 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. If a class is a leaf (i.e. it has no parts), its instances will not utilize the functionality appropriate to wholes, such as the retrieval of their 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);
};
```

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.

As mentioned above, we have omitted the 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 shown at the moment. This issue will be discussed further below.

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 we were to define *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 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 by inheritance from *Vehicle*.

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);
};
```

The methods defined here by *PartWholeObject* constitute the minimal set of behaviors required by any part–whole model incorporating our characteristic dimensions as described in Section 2. If these behaviors are not made available, then the semantics of parts and wholes cannot be properly maintained by the OODB system. As defined, these behaviors are invoked with respect to a whole object in order to create, delete, update and query part connections. Part–whole dependency semantics are also captured.

`addPart` creates (if allowable) a part–whole connection between the given part object and the target object, which then becomes the whole object in the relationship. `removePart` dissolves a connection between a whole and a part. `replacePart` is the atomic operation defined as `removePart` followed by `addPart`. Inherent in these behaviors are the integrity checks that guarantee the proper maintenance of the semantics of the prescribed part relationships—making OODB parts behave like real-world parts. As an example, `addPart` will not allow a car to be connected to a boat and it will not allow three engines to be connected to a boat (see the schema in Section 2). Such decisions can be readily made by consulting the instances of *PartRelationship* in the data dictionary. (The details of the algorithm that `addPart` implements can be found in [59].) These integrity services are provided without the need for an application designer to write specific code. This is an example of the power of our framework. Simple declarative specifications in the four characteristic dimensions result in a complex behavior mirroring the expected real-world behavior.

No explicit 'add whole' operation is included to complement `addPart` because, in our view, the construction of objects with respect to part hierarchies is inherently a bottom-up process: wholes are built up from lower-level parts. However, an operation 'add whole' is invoked implicitly by `addPart`. Therefore, to implement `addWhole(a, b)`, simply apply `addPart(b, a)`.

The part connections that a given object is involved in can be queried with the use of `getParts` and `getWholes`. `getParts` returns the entire set of immediate parts of the target object. To obtain a ‘parts explosion’ of an object to a certain depth or an unlimited depth, this method can be applied recursively. `getWholes` is defined analogously.

`deleteObject` is invoked with respect to an object when that object is to be deleted from the database. `deleteObject` encodes the combined deletion semantics of all the target object’s various part relationships, including, for example, the automatic propagation of the deletion operation to all other part and whole objects that are dependent on the target [59].

4.3. 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; and (2) the inclusion of reader methods for the properties inherited via part relationships. We discuss these two issues in the following.

4.3.1. Creating parts and wholes

The semantics of part relationships must be maintained throughout the entire lifetime of any object starting at its ‘birth’. The creation semantics of the part relationship comprise 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 in addition to 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 from the outset; if it detects a potential violation, then it aborts the instantiation.

The following demonstrates the use of `make` for the classes *Engine*, *Car* and *Boat*[**]:

```
(1) Engine *eng1 = Engine::make(300, {});
(2) Engine *eng2 = Engine::make(1250, {});
(3) Engine *eng3 = Engine::make(1250, {});
(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 initially installed in the new engine. (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 1250 horsepower. Line (4) creates a new instance of *Car* (with manufacturer Chevrolet and vehicle ID 7417) and installs ‘eng1’

[**] 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.

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) were changed to

```
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.

If a part relationship has a minimum multiplicity *min* greater than one, then the set-of-parts argument to make must contain at least *min* such parts. make will not allow the new whole object to come into existence with a violation of this constraint.

4.3.2. Part–whole inheritance

A property inherited via a part relationship should be accessible in the same way as an intrinsic property. The application programmer should see no distinction between the two. For example, we would like to obtain the value of the property *horsepower* for the car we have 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 appends an appropriate reader method to the proper header file. 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 should 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 the car's part. On the other hand, line (2) produces 2500 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.

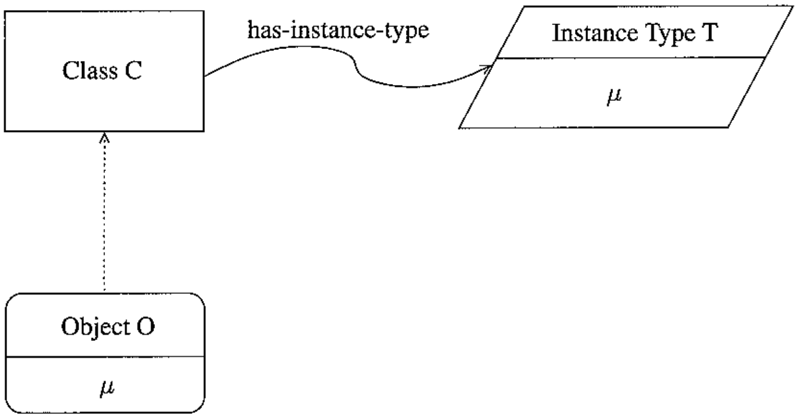

Figure 5. A class, its instance type and a class instance.

5. THE SECOND FRAMEWORK: METACLASS-BASED SYSTEMS

5.1. Metaclasses, classes and objects

The second framework for integrating a semantic relationship into an OODB system is based on the notion of metaclasses as defined in [44,45]. Any target system must be equipped with a metaclass facility of that kind in order for this framework to be applicable. One such system is VML, whose data model and metaclass facility are described in the following.

A class in a VML database schema serves as both an object generator and a container for its extension (i.e. the set of all objects that it has created) [37]. However, a VML class does not directly define the structure and behavior of its instances. This is done by an associated object type, called the *instance type* of the class. This association is depicted in Figure 5, where we use a parallelogram to denote the instance type and a dotted arrow to denote 'is an instance of'. Note that the set of behaviors μ (and structure) is 'passed through' the class from the instance type to the instances.

The notion of a metaclass arises from the fact that classes themselves are considered objects in VML (cf. Smalltalk [66]). Each class is an instance of what is referred to as a metaclass. Like an ordinary class, a metaclass has an instance type that defines the behavior of its instances, which in this case are classes. In addition, a metaclass can have a second associated object type, called an *instance–instance type* [44,47], that affects the instances of the classes which are the immediate metaclass instances. If we stack up the concepts of metaclass, class and (non-class) object into three distinct levels (Figure 6), then we can say that an instance type has an effect one level below where it is utilized, whereas an instance–instance type has an impact two levels below. Under this arrangement, the behavior of an object is defined in two separate places: at the instance type of its class and at the instance–instance type of its class's metaclass. This is illustrated in Figure 6.

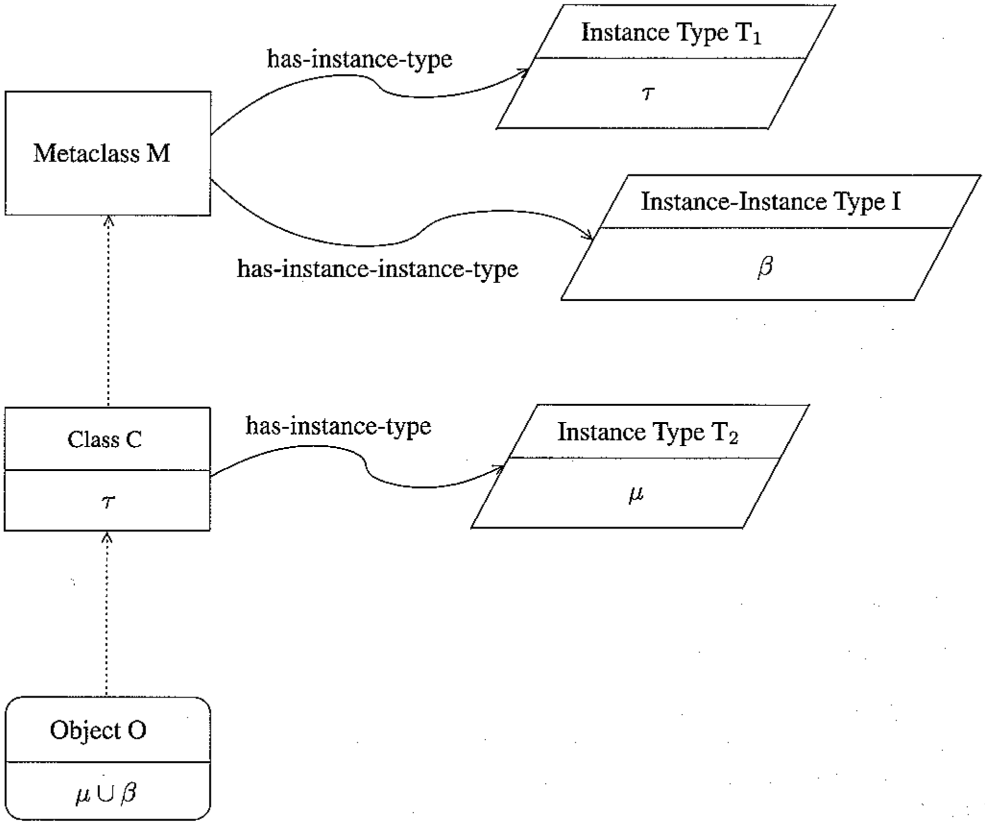

Figure 6. Three levels: metaclass, class and object.

One advantage of this approach is the fact that certain desired features can be given to all instances of a group of classes, without forcing those classes to be ancestors or descendants of each other in the subclass hierarchy. If the features are those required for participation in a semantic relationship, then the metaclass effectively introduces that relationship into the schema. Using that same metaclass in other schemas then adds the semantic relationship to the object model of the system. This is the basic idea underlying our second framework. More details are provided in the next section.

5.2. Semantic relationship metaclass

Suppose that we wish to incorporate the semantic relationship R into the OODB system, where R has the multidimensional structure described above. A new metaclass is introduced into the system with a name of the form *RObjectR′Object*. As with all metaclasses, there will be an associated instance type and instance–instance type.

The main purpose of the instance type of the semantic relationship metaclass is to endow classes with the ability to directly store information about the semantic relationships R in which they participate. In other words, a class participating in a semantic relationship will hold the values of the characteristic dimensions of that relationship. That information is made available at run-time via a query to the class itself.

This aspect of the instance type serves the same purpose as the class *R-Relationship* from the first framework. Where *R-Relationship* augmented the data dictionary in order to store the information about the entire R-schema, the metaclass's instance type alters the structure of the application classes themselves. In the first framework, the R-schema is in the data dictionary; in the second, it is stored in a distributed fashion among the application classes that participate in it. This arrangement effectively eliminates the need for a *LoadRSchema* program, because all the details of the R relationships can be specified directly in the class definitions. However, the dimensional data still must be 'loaded' into the classes (as they are objects) after their creation. This is done with additional methods defined at the metaclass's instance type and invoked by the system at the class-creation time.

The instance type of the metaclass also accomplishes a task allocated to the R-preprocessor in the first framework; namely, that of installing the object-generation method `make` in the classes. This method functions in exactly the same way as in the first framework: it creates objects and ensures that the R relationship's creation semantics are adhered to.

The role of the instance–instance type is analogous to the root class *RR′Object*. It supplies the objects (i.e. instances of classes) with the capability of actually participating in R relationships. Behavior is defined for connecting, disconnecting and reconnecting the objects via the R relationships, with the integrity of the semantics of the prescribed relationship dimensions inherently guaranteed. Retrieval of related objects is also supported. The major difference here is that a class does not need to be defined explicitly as derived from some root. It must simply be denoted as an 'instance' of the particular metaclass. No multiple inheritance configuration is required in the application schema.

The instance–instance type also serves to carry out another function of the R-preprocessor (from the first framework) and thus eliminates its need from this second framework entirely—handling the inheritance that might take place along the R relationship. In our first framework, the R-preprocessor augmented class definitions to include methods for accessing inherited properties. The preprocessing was necessary to avoid any compile-time errors. In VML, however, missing method definitions are never considered compile-time offenses. Instead, they are trapped at run-time.

To see what we mean by this, assume that we have an object O that is an instance of the class C. Assume further that the method `get_prop` is not defined for C. Now, consider the following statement:

```
O->get_prop();
```

Ordinarily, we would expect this statement to cause a compile-time error. In VML, a run-time exception is raised. This exception is caught in such a way as to generically carry out all the different inheritance behaviors that may be associated with relationship R. Any invalid invocations of methods against an object participating in an R relationship are trapped. It is determined whether the requested property (being accessed via a reader method) is inherited. If so, the prescribed inheritance is manifested via a computation and the return of a legitimate value. Otherwise, the run-time exception is passed up.

6. VML PART RELATIONSHIP

In this section, we present the two components, namely, the PartWhole instance type and the PartWhole instance–instance type, that define the VML PartWhole metaclass and integrate the part relationship into the VML object model.

6.1. PartWhole instance type

The PartWhole instance type defines the additional behavior for the classes that participate in part relationships. Again, let us recall that, in our second framework, classes are assumed to be objects themselves. So, the behavior pertains to the classes and not their instances. The behavior (methods) is divided into three groups, as denoted by the comments in the following specification of the PartWhole instance type's public interface[††]:

```
OBJECTTYPE PartWhole_InstType SUBTYPEOF Metaclass_InstType;
INTERFACE
  METHODS
//
// Group 1
//
   make(someParts: {OID}): OID;
   deleteObject(anObject : OID);
//
// Group 2
//
   defPartRelshps(someRelshps: {PartRelationshipType});
   defWholeClasses(someClasses: {OID});
//
// Group 3
//
   exclusiveness(aClass: OID): ExclusivenessType;
   minMultiplicity(aClass: OID): INT;
   maxMultiplicity(aClass: OID): INT;
   dependency(aClass: OID): DependencyType;
   propertyUpwardInherited(propertyName: STRING, aClass: OID): BOOL;
   propertyDownwardInherited(propertyName: STRING, aClass: OID): BOOL;
   phi(propertyName: STRING, aClass: OID): STRING;
END;
```

The two methods in Group 1, `make` and `deleteObject`, are the constructor and destructor methods provided to a class participating in a part relationship. They operate in the exact same manner as their namesake methods from the first framework, encoding the creation and deletion semantics, respectively, of the part relationship. Here, they are defined directly in the instance type of the metaclass. This arrangement is neater than in the first framework where the definitions appear separately: `make` is installed by the part preprocessor, while `deleteObject` is defined in the root.

[††] For the sake of brevity, we have omitted some extraneous details from the public interface. Also, we forgo any discussion of the VML syntax which is relatively straightforward [48].

The methods denoted by Group 2 are invoked at the class-creation time. They serve to inform the new class of the part relationships that it is involved in. One will note the asymmetry of the names of the methods and their parameters. The first method is named `defPartRelshps` and takes a set of 'part relationship structures' as an argument, while the second is called `defWholeClasses` and takes a set of classes as an argument. The first method informs the class of any part relationships in which it plays the role of the whole class. The related 'part' classes and the respective dimensional data of the part relationships are delivered as a set of simple vectors (i.e. `PartRelationshipType` structures). Thus, all dimensional data of a given part relationship are stored with the whole class rather than with the part class. This choice is consistent with our view that a part hierarchy is fundamentally a bottom-up construction. The method `defWholeClasses`, in contrast, informs the class of all the part relationships in which it plays the role of the part class. The 'whole' classes are provided as arguments and no dimensional data is needed because those data values are provided to the respective whole classes (using `defPartRelshps`). To summarize, `defPartRelshps` and `defWholeClasses` are used to establish a class's part relationships and store the dimensional data associated with each. In this way, they function equivalently to the *LoadPartSchema* program of the ONTOS part relationship.

The Group 3 methods are used to query a class in order to obtain values of the characteristic dimensions of part relationships that the class participates in. The parameter 'aClass' of each represents the object identifier (OID) of the part class in the relationship of interest. Overall, these methods operate equivalently to the public interface of the class *PartRelationship* from the first framework[‡‡]. There, the dimensional data were stored as separate part relationship objects, whereas here that same data are stored directly with the whole class of the part relationship.

The VML code of the instance type definitions and class definitions for the schema of Figure 1 is

```
OBJECTTYPE VehicleType;
  PROPERTIES
  manufacturer: STRING;
  vehicleID: INT;
END;

CLASS Vehicle
  INSTTYPE VehicleType
END;

OBJECTTYPE CarType SUBTYPEOF VehicleType;
END;

CLASS Car METACLASS PartWhole
  INSTTYPE CarType
  INIT Car->defPartRelshps({[thePartClass:  Engine,
                             exclusiveness: GLOBAL_EXCLUSIVE,
                             multiplicity:  [min: 0, max: 1],
                             dependency:    NIL,
```

[‡‡] There is a slight difference here regarding the way the inheritance information is retrieved. A pair of predicates and a 'mapping' function are defined [2] to aid the operation of the so-called NOMETHOD clause. Note, however, that the obtainable dimensional information is identical.

```
                             upSet:         {'horsepower'},
                             downSet:       {},
                             phi:           {['horsepower','identity']}]
                           })
END;

OBJECTTYPE BoatType SUBTYPEOF VehicleType;
END;

CLASS Boat
  INSTTYPE BoatType
  INIT Boat->defPartRelshps({[thePartClass:  Engine,
                              exclusiveness: GLOBAL_EXCLUSIVE,
                              multiplicity:  [min: 0, max: 2],
                              dependency:    NIL,
                              upSet:         {'horsepower'},
                              downSet:       {},
                              phi:           {['horsepower','add']}]
                            })
END;

OBJECTTYPE EngineType;
  PROPERTIES
  horsepower: INT;
END;

CLASS Engine
  INSTTYPE EngineType
  INIT Engine->defWholeClasses({Car, Boat})
END;
```

Above each class definition, we see a corresponding object type specification, which is the instance type of the respective class. The properties of the classes are defined there. In the definitions of the classes *Car* and *Boat*, we find invocations of the method `defPartRelshps` (in their 'INIT' clauses). These invocations, as noted above, occur at the class-creation time. For example, `defPartRelshps` informs the class *Car*, at its creation time, about its participation as the whole class in a part relationship whose part class is *Engine*. Additionally, the relationship's dimensions are as follows. It is globally exclusive, has a minimum multiplicity of zero and a maximum of one and does not exhibit any dependency. Since the upSet contains *horsepower*, that property is inherited by *Car* from *Engine*. The inheritance is invariant [2] as indicated by the mapping of *horsepower* to the identity operator in phi. Thus, the value of *horsepower* will be passed up unchanged. There is no downward inheritance because the downSet is empty.

The part relationship between *Boat* and *Engine* differs slightly from that between *Car* and *Engine*. As we see in the call to `defPartRelshps` in *Boat*'s definition, the maximum multiplicity is two and the inheritance of *horsepower* is additive transformational, due to the mapping of *horsepower* to 'add' in phi. Therefore, the upward-inherited horsepower value of the whole (boat) will be derived from a summation of the values at the related parts (engines).

The invocation of `defWholeClasses` in the class *Engine*, establishing the fact that *Engine* is the part class in two part relationships, completes the definition of the part schema. Since *Engine* is a leaf in the part hierarchy in our simple schema, there is no need to invoke defPartRelshps with respect to it; it is not serving in the role of the whole class in any relationship. Likewise, there is no need to invoke defWholeClasses for *Car* and *Boat* since each is at the top of the part hierarchy.

6.2. PartWhole instance–instance type

Our VML specification of the schema above was still lacking the instance–instance type definition. The PartWhole instance–instance type implicitly supplements the definitions of the classes of the part hierarchy by giving their instances additional methods pertaining to the part relationship. Specifically, those methods allow the instances to connect, disconnect and reconnect themselves with respect to part relationships. The instances can also be queried regarding their related parts and wholes. In this manner, the PartWhole instance–instance type holds the same place in the second framework as the root class *PartWholeObject* does in the first. Moreover, the instance–instance type implements the part–whole inheritance.

The public interface of the PartWhole instance–instance type is as follows:

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

All the methods here function identically to the corresponding methods of *PartWholeObject* (see Section 4.2 for details).

7. CONCLUSION

We have presented two frameworks for incorporating semantic relationships into an existing OODB system that was not originally built to support them. The first framework is valid for most target OODB systems; it assumes only that the system supports manifest type. The second framework relies on the existence of a metaclass mechanism. Each framework does require an existing formal dimensional characterization of the semantic relationship of interest.

The primary benefit of either framework is the fact that they allow application developers to easily exploit the expressiveness of various semantic relationships. The developers are freed from the tedious task of hand-coding the proper behavior associated with the relationship each time they build a system. Instead, they simply produce a declarative specification of the desired relationships and let the OODB system enforce the proper semantics automatically. Their applications can then readily connect, disconnect, reconnect and query various objects with respect to the semantic relationship of interest, while knowing that the integrity of the relationship is being maintained.

In order to demonstrate the viability of our frameworks, we presented the details of integrating a part–whole semantic relationship into the ONTOS OODB system. We have also presented the integration of the part relationship into VML. The PartWhole metaclass has been distributed as a portion of the VML metaclass library.

As for future work, several other semantic relationships need to be implemented, including the ownership relationship. An issue that would benefit from further investigation is schema evolution. Currently, all semantic relationship meta-data is assumed to be immutable. Any tampering with that data by an application program could lead to a breakdown in the maintenance of the proper semantics for the entire database. Allowing the semantic relationships to evolve, whether in terms of the addition of new relationships or the modification of the dimensional values of existing ones, may be desirable.

REFERENCES

1. Artale A, Franconi E, Guarino N, Pazzi L. Part–whole relations in object-centered systems: An overview. *Data & Knowledge Engineering* 1996; **20**(3):347–383.
2. Halper M, Geller J, Perl Y. An OODB part-whole model: Semantics, notation, and implementation. *Data & Knowledge Engineering* 1998; **27**(1):59–95.
3. Kim W, Bertino E, Garza JF. Composite objects revisited. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, OR, June 1989; 337–347.
4. Noy NF, Hafner CD. The state of the art in ontology design: A survey and comparative review. *AI Magazine* 1997; **18**(3):53–74.
5. Winston ME, Chaffin R, Herrmann DJ. A taxonomy of part-whole relations. *Cognitive Science* 1987; **11**(4):417–444.
6. Halper M, Perl Y, Yang O, Geller J. Modeling business applications with the OODB ownership relationship. Freedman RS (ed.). *Proceedings of the 3rd International Conference on Artificial Intelligence Applications on Wall Street*, New York, June 1995; 2–10.
7. McCarty LT. Ownership: A case study in the representation of legal concepts (preliminary draft). *Conference in Celebration of the 25th Anniversary of the Instituto Documentazione Giuridica*, Florence, Italy, 1993; 1–23.
8. McCarty LT. An implementation of Eisner v. Macomber. *Proceedings of the 5th International Conference on Artificial Intelligence and Law*, College Park, MD, May 1995; 276–286.
9. Yang O, Halper M, Geller J, Perl Y. The OODB ownership relationship. *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'94)*, London, December 1994; 389–403.
10. Kolp M. A metaobject protocol for reifying semantic relationships into reflective systems. *Proceedings of the 4th Doctoral Consortium of the 9th International Conference on Advanced Information Systems Engineering (CAiSE'97)*, Barcelona, Spain, June 1997; 89–100.
11. Pirotte A, Zimányi E, Massert D, Yakusheva T. Materialization: A powerful and ubiquitous abstraction pattern. *Proceedings of VLDB'94*, Santiago, Chile, 1994; 630–641.
12. Geller J, Perl Y, Neuhold E. Structure and semantics in OODB class specifications. *SIGMOD Record* 1991; **20**(4):40–43.
13. Neuhold EJ, Schrefl M. Dynamic derivation of personalized views. *Proceedings of the 14th Conference on Very Large Databases*, Long Beach, CA, 1988.
14. Schrefl M, Neuhold EJ. A knowledge-based approach to overcome structural differences in object-oriented database integration. *Proceedings of the IFIP Working Conference on the Role of AI in Database and Information Systems*, Guangzhou, China. North-Holland: Amsterdam, 1988.
15. Schrefl M, Neuhold EJ. Object class definition by generalization using upward inheritance. *Proceedings of the 4th International Conference on Data Engineering*, Los Angeles, CA, February 1988; 4–13.
16. Storey VC. Understanding semantic relationships. *VLDB Journal* 1993; **2**(4):455–488.
17. Woods WA. What's in a link: Foundations for semantic networks. *Readings in Knowledge Representation*, Brachman RJ, Levesque HJ (eds.). Morgan Kaufmann: San Mateo, CA, 1985; 218–241.
18. Hammer M, McLeod D. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems* 1981; **6**(3):351–386.
19. Elmasri R, Navathe SB. *Fundamentals of Database Systems*. Benjamin/Cummings: New York, 1989.
20. Elmasri R, Weeldreyer J, Hevner A. The category concept: An extension to the entity-relationship model. *International Journal of Data and Knowledge Engineering* 1985; **1**(1):75–116.
21. Mylopoulos J, Borgida A, Jarke M, Koubarakis M. Telos: Representing knowledge about information systems. *TOIS* 1990; **8**(4):325–362.
22. Mays E, Apte C, Griesmer J, Kastner J. Experience with K-Rep: An object-centered knowledge representation language. *Proceedings of the IEEE AI Application Conference*, San Diego, CA, March 1988.

23. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object-Oriented Modeling and Design.* Prentice-Hall: Englewood Cliffs, NJ, 1991.
24. Booch G. *Object-Oriented Design.* Benjamin/Cummings: Redwood City, CA, 1991.
25. Booch G. *Object-Oriented Analysis and Design with Applications* (2nd edn). Benjamin/Cummings: Redwood City, CA, 1994.
26. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide.* Addison-Wesley: Reading, MA, 1999.
27. Fowler M, Scott K. *UML Distilled* (2nd edn). Addison-Wesley: Reading, MA, 2000.
28. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual.* Addison-Wesley: Reading, MA, 1999.
29. Coad P, Yourdon E. *Object-Oriented Analysis* (2nd edn) (*Yourdon Press Computing Series*). Prentice-Hall: Englewood Cliffs, NJ, 1991.
30. Bancilhon F, Delobel C, Kanellakis P (eds.). *Building an Object-Oriented Database System: The Story of O_2.* Morgan Kaufmann: San Mateo, CA, 1992.
31. Bertino E, Martino L. *Object-Oriented Database Systems: Concepts and Architectures.* Addison-Wesley: New York, 1993.
32. Cattell RGG (ed.). *The Object Database Standard: ODMG-93.* Morgan Kaufmann: San Francisco, CA, 1994.
33. Cattell RGG, Barry DK (eds.). *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann: San Francisco, CA, 1997.
34. Gray PMD, Kulkarni KG, Paton NW. *Object-Oriented Databases: A Semantic Data Model Approach.* Prentice-Hall: New York, 1992.
35. Kim W, Lochovsky FH (eds.). *Object-Oriented Concepts, Databases, and Applications.* ACM Press: New York, 1989.
36. Loomis MES. *Object Databases: The Essentials.* Addison-Wesley: Reading, MA, 1995.
37. Zdonik SB, Maier D (eds.). Fundamentals of object-oriented databases. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann: San Mateo, CA, 1990; 1–32.
38. ONTOS, Inc., Lowell, MA. *ONTOS DB/Explorer 4.0 Reference Manual,* 1996.
39. Soloviev V. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, and O_2. *SIGMOD Record* 1992; **21**(1):93–104.
40. Lamb C, Landis G, Orenstein J, Weinreb D. The ObjectStore database system. *Communications of the ACM* 1991; **34**(10):50–63.
41. eXcelon Corporation. ObjectStore. http://www.exln.com [2002].
42. Versant. http://www.versant.com [2002].
43. Abelson H, Sussman GJ. *Structure and Interpretation of Computer Programs.* MIT Press: Cambridge, MA, 1985.
44. Klas W. A metaclass system for open object-oriented data models. *PhD Thesis,* Technical University of Vienna, January 1990.
45. Klas W. Tailoring an object-oriented database system to integrate external multimedia devices. *Workshop on Heterogeneous Databases and Semantic Interoperability,* Boulder, CO, 1992.
46. Huhns MN, Stephens LM. Plausible inferencing using extended composition. *Proceedings of IJCAI-89,* Detroit, MI, 1989; 1420–1425.
47. Klas W, Aberer K, Neuhold EJ. Object-oriented modeling for hypermedia systems using the VODAK Model Language (VML). *Object-Oriented Database Management Systems (NATO ASI Series),* Özsu MT, Biliris A (eds.). Springer: Berlin, 1994.
48. Klas W *et al.* VODAK design specification document, VML 3.1. *Technical Report,* GMD, Sankt Augustin, Germany, July 1993.
49. Motschnig-Pitrik R, Storey VC. Modelling set membership: The notion and the issues. *Data and Knowledge Engineering* 1995; **16**:145–185.
50. Rumbaugh J. Relations as semantic constructs in an object-oriented language. *Proceedings of OOPSLA-87,* October 1987; 466–481.
51. Albano A, Ghelli G, Orsini R. A relationship mechanism for a strongly typed object-oriented database programming language. *Proceedings of VLDB '91,* 1991; 565–575.
52. Diaz O, Gray PMD. Semantic-rich user-defined relationships as a main constructor in object-oriented databases. *Proceedings of the IFIP TC2 Conference on Database Semantics.* North-Holland: Amsterdam, 1990.
53. Albano A, Ghelli G, Orsini R. Fibonacci: A programming language for object databases. *VLDB Journal* 1995; **4**(3): 403–444.
54. Paton NW. ADAM: An object-oriented database system implemented in Prolog. *Proceedings of the 7th British National Conference on Data bases,* 1989.
55. MacKellar B, Peckham J. Representing design objects in SORAC: A data model with semantic objects, relationships and constraints. *AI in Design '92,* Pittsburgh, PA, 1992.
56. Liu L, Halper M. Incorporating semantic relationships into an object-oriented database system. *Proceedings 32nd Hawaii International Conference on System Sciences (HICSS-32),* Maui, HI, January 1999. IEEE, 1999 (CD-ROM).

57. Halper M, Geller J, Perl Y, Klas W. Integrating a part relationship into an open OODB system using metaclasses. *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM-94)*, Adam N, Bhargava B, Yesha Y (eds.), Gaithersburg, MD, 1994; 10–17.
58. Halper M, Geller J, Perl Y. An OODB 'part' relationship model. *Proceedings of the ISMM 1st International Conference on Information and Knowledge Management*, Yesha Y (ed.), Baltimore, MD, November 1992; 602–611.
59. Halper M. A comprehensive part model and graphical schema representation for object-oriented databases. *PhD Thesis*, New Jersey Institute of Technology, October 1993.
60. Halper M, Geller J, Perl Y. Value propagation in OODB part hierarchies. *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM-93)*, Bhargava B, Finin T, Yesha Y (eds.), Washington, DC, November 1993; 606–614.
61. Blaha M, Premerlani W. *Object-Oriented Modeling and Design for Database Applications*. Prentice-Hall: Upper Saddle River, NJ, 1998.
62. Khoshafian S, Dasananda S, Minassian N. *The Jasmine Object Database: Multimedia Applications for the Web*. Morgan Kaufmann: San Francisco, CA, 1999.
63. Even S. *Graph Algorithms*. Computer Science Press: Potomac, MD, 1979.
64. Harary F. *Graph Theory*. Addison-Wesley: Reading, MA, 1969.
65. Mark V Systems, Ltd., Encino, CA. ObjectMaker Documentation, 1993.
66. Goldberg A, Robson D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley: Reading, MA, 1983.