ELSEVIER

Data & Knowledge Engineering 27 (1998) 59–95

DATA & KNOWLEDGE ENGINEERING

An OODB Part-Whole model: Semantics, notation and implementation

Michael Halper[a,*], James Geller[b], Yehoshua Perl[b]

[a]Dept. of Mathematics and Computer Science, Kean University, Union, NJ 07083, USA
[b]CIS Dept., NJIT, Newark, NJ 07102, USA

Received 14 April 1995; revised 14 January 1997; accepted 3 October 1997

Abstract

The notion of a part-whole relationship plays an important role when modeling data in many advanced application domains. It is therefore important that Object-Oriented Database (OODB) systems include support for this modeling primitive. We present a comprehensive part model for OODB systems. The model's foundation is a part-whole relationship that captures a variety of real-world, part-whole semantics, partitioned into four characteristic dimensions: exclusiveness, multiplicity, dependency and inheritance. These impose constraints on any 'part' transactions (like 'add-part') to ensure that the state of the database remains consistent with the prescribed part-whole semantics. They also provide functionality like deletion dependency and several kinds of inheritance, both from the part to the whole and vice versa. The part relationship gives flexibility to an application developer who simply declares the desired semantics and then lets the OODB system automatically enforce it. We also introduce a graphical notation that can be used to express the enhanced semantics in the development of OODB part-whole schemata. Our part model has been integrated into the VODAK Model Language (VML), an OODB system, with the use of its extensible metaclass mechanism. © 1998 Elsevier Science B.V.

Keywords: Part-Whole relationship; Part relationship; Part semantics; Part-Whole inheritance; Derived attribute; Object-Oriented database; Graphical schema representation; Metaclass

1. Introduction

The part-whole relationship (part relationship, for short) plays an important role in many data modeling tasks in diverse application domains such as manufacturing [31], Computer Aided Design (CAD) [5,8], document processing [29,54], and medical informatics [9,13,20,39]. If the modeling is to be carried out correctly and precisely, then it is insufficient to rely on a construct such as an ordinary

* Corresponding author. E-mail: mhalper@turbo.kean.edu

0169-023X/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.
PII: S0169-023X(97)00055-4

relationship (class-to-class connection) that has been given the name 'part-of' or 'has' [61]. What is needed instead is a modeling primitive that captures real-world, part-whole semantics by imposing limitations on the transactions and interactions between the individuals (objects) in a database and by providing those objects with additional functionality befitting parts and wholes. As stated in a recent overview paper in this journal [4], the part relationship's 'specific ontological nature needs to be studied, understood, and integrated within knowledge and data modeling formalisms and methodologies, without leaving the burden to the user.' In this paper, we are addressing precisely that need by introducing a comprehensive part-whole model for object-oriented database (OODB) systems [35,62]. Our part model has as its basis a part relationship that serves as a built-in modeling primitive of an OODB system. This relationship encompasses many of the aspects and 'vertical relationships' outlined in [4]. Specifically, it features the following:

- Constraints that impose appropriate 'part-whole' restrictions on the state of the database and the various part transactions (like 'add-part' and 'remove-part').
- Dependency between part objects and whole objects.
- Inheritance and propagation of properties, both from part to whole and vice versa.

We organize the above into four *characteristic dimensions*: (a) exclusiveness, (b) multiplicity, (c) dependency, and (d) inheritance. (Cf. the description of part-whole in terms of basic relational elements in [30,60]. See also [3,16].) Each dimension can take on different values, giving flexibility to an application or schema designer, who *declares* the desired semantics by choosing the appropriate values. The OODB system then automatically ensures that the chosen semantics is always obeyed. The designer is thus alleviated of the burden of having to 'hand code' the part semantics into the methods of the participating classes. Part-whole semantics, in fact, is one of the areas where an OODB system can offer more than an object-oriented programming language like C++ in capturing the real needs of an application at no additional cost. Instead of programming the part semantics with C++ code for each application, the part model in conjunction with the database system provides a reusable, built-in solution. Our implementation demonstrates the feasibility of such an enhancement for an OODB system without hurting its performance.

Our part model also includes a graphical schema notation for the part relationship. Such graphical representations of database schemata have become standard fixtures of most data models [1,10,12,41,50]. In [28], we presented a graphical notation for the data-definition language of an OODB system; here, we enhance that notation with a variety of part relationship symbols that express the different semantics prescribed by the four characteristic dimensions.

As our implementation vehicle, we have chosen the metaclass mechanism of an OODB system called the VODAK Model Language (VML) [36,37]. We have constructed a metaclass that integrates our part relationship into the underlying VML object model. Via this metaclass, database objects are provided with functionality that makes them 'parts' and 'wholes.' They are given the capability to establish and dissolve, on demand, part-whole connections between each other; the prescribed part-whole semantics is enforced automatically. The objects can then be queried, allowing for the retrieval of related parts and wholes to any depth in the part hierarchy.

In the next section, we discuss previous work on part relationships. In Section 3, we formally define our part relationship and introduce its graphical notation. The definition of generalized derived attributes based on redundant part-whole inheritance is found in Section 4. A description of the implementation of our part model in VML appears in Section 5. Conclusions follow in Section 6.

2. Survey of previous work

The notion of part relationship is so intuitive and widespread in ordinary human discourse that it has been given serious attention in a wide range of areas such as solid geometry, topology, set theory, linguistics, cognitive science, and knowledge representation. In fact, the part relationship's inclusion and the extent of its treatment have been proposed as important criteria in a framework for evaluating and comparing existing ontologies [47]. It is noted in [47] that the part relationship was

> ''represented very differently in the [existing] ontologies and was often not adequately dealt with.... Most of the ontologies do not directly address the issue of part-whole relation and the distinction between subset-of, part-of, member-of, and so on.'' [p. 71]

We believe that the work presented in this paper is a step toward closing this gap.

A number of formal theories, often called *mereologies*, have been proposed for the description of the intuitive concepts of part and whole. (Surveys can be found in [4,22,51].) The use of logic-based formalisms for part-whole analysis is an active area as witnessed by two recent papers in this journal. In [52], a new mereotopology, a formal theory comprising mereology along with elements of topology, is presented. The relationship between 'whole-oriented' topology and 'part-oriented' mereology is examined in a formal light in [57]. Both papers utilize logical part relationships that satisfy basic 'part' axioms, but they do not make it clear how their theories could contribute to implemented systems.

In cognitive science and linguistics, much work has focused on the part relationship's transitivity (or lack thereof) and 'part' reasoning. In one of the most widely cited papers, Winston, Chaffin and Herrmann [60] propose six different kinds of part relationships to account for various usages. (See [4,22] for additional references.) Attempts have been made to incorporate such distinctions into knowledge representation and reasoning systems. For example, in a recent paper in this journal [18], an alternate six-way distinction to that of [60] is proposed as a means for modeling physical assemblies of parts. The prototype presented in [18] employs a diagrammatic CAD interface to objects under consideration. Using a network of concepts along with rules and attached procedures, the prototype can provide a user with varied 'part views' of an object based on its alternate part descriptions. The six part relationships of [60] have also been included in a 'Semantic Relationship Analyzer' that aids a database designer in the creation of a schema using the relational model [55]. In [43], four of the six part relationships are identified as 'core' relationships, and it is shown that one can intermix these and yet maintain the validity of transitive inferences. A strict differentiation between the relationships component-of, part-of, and member-of is advocated in [44], where it is noted that syllogisms combining the latter two tend to be invalid.

In our work, we do not directly incorporate any fixed collection of partitive relationships such as [18,60]. Instead, we allow the schema designer to declaratively construct all sorts of different part relationships from a set of underlying characteristic dimensions (i.e., semantic options), which themselves have been gleaned from ordinary part-whole discourse. Alternative decompositions of wholes can exist side-by-side within our framework.

As pointed out in [9] in an earlier issue of this journal, current knowledge representation

formalisms are not well adapted to part reasoning, especially in complex domains such as medicine. It is also noted there that coding schemes tend to interfere with proper part-whole modeling, and subsumption and partitive links are often confused. Our OODB approach to part modeling is concerned with the semantically correct maintenance of part data. Our extensive research using OODBs for medical terminologies [20,39] leads us to concur with [9] on the importance of medical part modeling. Indeed, our OOHVR (Object-Oriented Healthcare Vocabulary Repository) vocabulary implementation contains 6412 part relationships in a vocabulary of about 46000 terms.

The most widely known work on parts in OODBs is that of ORION [6,34]. In [34], an account of the part relationship's semantics is presented but is limited to the two dimensions that we call exclusiveness and dependency. There appears in [34] an extended treatment of many system issues concerning parts like query capabilities, which we provide with our VML implementation. We do not duplicate ORION's use of parts as units of locking, versioning and authorization. In [43], it is shown how the ORION part relationships can be implemented in the Telos knowledge formalism [45].

The ORION exclusiveness dimension divides part relationships into two kinds: exclusive and shared. An exclusive part can be attached to at most one whole; a shared part, to any number of wholes. Therefore, the exclusiveness constraint imposes a restriction on the entire database. We have found that this is often too constraining, and, in this paper, we refine it by allowing exclusiveness to be imposed on a single class and relaxed otherwise. This leads to two types of exclusiveness, *global exclusiveness* and *class exclusiveness*. Our class exclusiveness does not exist in [34].

The dependency dimension of ORION allows a part to be made dependent on its whole: If the whole is deleted, then the part is deleted automatically. This is useful in alleviating the burden of manually searching out and deleting parts. A whole, however, is sometimes barely more substantial than one of its defining parts, as in the case of a bicycle and its frame. In such cases, it may be desirable to define the whole as dependent on its part. Therefore, our model permits the specification of dependency in both directions. ORION's dependency can only be from the part to the whole.

The Object Modeling Technique (OMT) [48] provides a part (aggregation) relationship which can exhibit all the characteristics of ordinary associations (such as cardinality constraints). In fact, the part relationship is viewed as a special form of association which can have additional semantic connotations in certain circumstances. However, the model does not ascribe additional characteristics to capture the semantics of parts and does not incorporate exclusiveness and dependency. Nor does it deal with the inheritance of properties among classes participating in part relationships. OMT does provide the ability to define 'link attributes' for part relationships, something not covered by our model. The OMT notation includes a symbol for the propagation of operations across associations, and specifically across aggregations between wholes and their parts. Our model incorporates the propagation of data values rather than the propagation of operations. Let us note that our enhancements of ORION's exclusiveness and dependency dimensions have recently been utilized by Motschnig and Kaasboll in a proposed extension to OMT [42].

Property inheritance along part relationships has been dealt with in [7,46] where, e.g., the color of a car can be defined as the color of its body. Both models, though, limit the inheritance to what we call the upward, invariant case. We identify three kinds of part-whole inheritance called *invariant*, *transformational*, and *cumulative*, each of which is permitted upward and downward.

Our inheritance has both schema-level and instance-level ramifications. At the schema-level, one class obtains the definition of a property based on the definition of the same property at another ('part'

or 'whole' related) class, similar to an IS-A hierarchy. At the instance level, instances of parts (wholes) receive their values for the inherited property from related wholes (parts). We refer to properties inherited via part relationships as *derived attributes*. However, this should not be confused with the notion of a generic derived attribute which can be defined in terms of some general computation or query [7,40,48]. We are concerned with identifying the underlying semantics of part-whole inheritance and providing this in a *declarative* fashion to the data modeler. While our part model does not offer generic derived attributes, we rely on any OODB system into which it is integrated to accommodate this capability.

In addition to the inheritance of a property across a part relationship, our model supports the definition of an inherited property in terms of simultaneous inheritances across many part relationships. Such properties are a powerful component of part-whole modeling, permitting the declarative specification of such commonplaces as: 'The weight of a car is the sum of the weights of its parts.'

We realize that some previous systems have offered limited part capabilities. Our view is that there is a need for a comprehensive framework that provides overarching coverage of the various semantic issues involved in part-whole modeling. In this way, the needs of disparate users will be addressed by one model. Such a model should be implemented in an existing OODB system via some process of integration in order to allow current users to exploit its features. It is impractical to build a new separate 'part' OODB system and expect people to adopt it. With this in mind, we chose to integrate the part model into VML which was built to handle this sort of extension.

We have previously presented some elements of the part model [23,25]. A treatment of the semantics of parts and a preliminary version of our graphical schema notation for parts appeared in [23]. The issue of value propagation across part relationships was discussed in [25]. In this paper, we extend and unify those two treatments into one comprehensive, formal framework. The principal extension comes in the recasting of value propagation as a true form of property inheritance. The inheritance dimension of the part relationship is now divided into three aspects: (1) the 'upSet' which defines upwardly inherited properties, (2) the 'downSet' which defines downwardly inherited properties, and (3) a function which determines the kind of value propagation underlying the inheritance of each of the properties in (1) and (2).

In comparison with [25], cumulative inheritance has been refined into two kinds to support sets and bags of values. The dependency dimension of [23] has also been revised in a new formalism so as to exhibit multivalued dependency behavior. In [24], we presented a detailed implementation strategy for part relationships based on their representation as object classes in their own right. That was supplanted by a metaclass implementation which was introduced in [27].

In summary, this paper presents a more comprehensive and in-depth exploration of part-whole modeling. Overall, the main contributions of this work are:

(1) The extension of existing semantics for the part relationship in the context of data modeling and OODB systems.
(2) The unification of the extended semantics with existing semantics in a formal framework. The formalization divides the part relationship into a number of semantic aspects (called characteristic dimensions). (Cf. [33].)
(3) An extensive graphical notation for the part relationship and its various semantics useful for the construction and dissemination of OODB schemata.
(4) An efficient implementation of the part model in the context of VML.

3. The OODB part relationship

In this section, we formally define the part relationship for OODB systems. We will be using our OOdini graphical notation [28] to represent OODB schemata. A class is drawn as a rectangle, while an attribute is represented by an ellipse attached to a class via a line (Fig. 2). The IS-A relationship is a thick arrow directed from the subclass to the superclass (Fig. 4). Following [48], an instance of a class is drawn as a rectangle with rounded corners and the parenthesized class name inside (Fig. 3).

A class whose instances are the parts in a given part relationship is called the *part class* in that relationship. The other class whose instances are the wholes is called the *whole class*. For example, if classes *Chapter* and *Book* are in a part relationship, then *Chapter* is the part class and *Book* is the whole class with respect to that relationship. Because a class may play the role of both part and whole (in different part relationships), we will often refer to such a class as a PartWhole (PW) class.

3.1. Definition of part relationship

For a class C, let $E(C)$ denote the extension of C (i.e., the set of all its instances). Also, let $\Pi(C)$ denote the set of all properties of C.

The part relationship between a part class B and a whole class A (written $P_{B,A}$) is defined as the following quintuple:

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

where $\Diamond_{(B,A)}$ is a relation from $E(B)$ to $E(A)$. The pair $(b, a) \in \Diamond_{(B,A)}$ indicates that the instance b of class B *is part of* the instance a of class A. The next four elements are the *characteristic dimensions* referred to respectively as: (a) *exclusiveness*, (b) *multiplicity*, (c) *dependency*, and (d) *inheritance*. Their domains will be discussed in detail in the following subsections.

The characteristic dimensions serve to declaratively specify the semantics or characteristics of the part relationship. In a database, users will manipulate parts and wholes through a set of standard transactions like adding a part to a whole, removing a part, or replacing a part. If proper part-whole semantics are to be maintained, then there are circumstances under which certain transactions must be forbidden. The characteristic dimensions help to monitor and enforce the legality of the various transactions. Beside this 'constraint-satisfaction' aspect, they are also used to control the dependency among parts and wholes and to define inheritance behavior.

In the subsequent sections, we will need the following definitions. Assume a part relationship $P_{B,A}$.

Definition 1. $\forall a \in E(A)$, let $M_{\Diamond_{(B,A)}}(a) = \{b | b \in E(B) \wedge (b, a) \in \Diamond_{(B,A)}\}$. $M_{\Diamond_{(B,A)}}(a)$ is called the *part set* of a with respect to $P_{B,A}$. It is the set of instances of B which are parts of a.

Definition 2. $\forall b \in E(B)$, let $H_{\Diamond_{(B,A)}}(b) = \{a | a \in E(A) \wedge (b, a) \in \Diamond_{(B,A)}\}$. $H_{\Diamond_{(B,A)}}(b)$ is called the *whole set* of b with respect to $P_{B,A}$. It is the set of instances of A of which b is a part.

To complement our formal part model, we will introduce a graphical notation that will allow us to express the various semantics of part-whole modeling. The symbol for the part relationship (without characteristic dimensions) is a bold, dashed line. A diamond head at one end of the line marks the

Fig. 1. Part relationship $P_{B,A}$.

whole class.[1] We chose a bold symbol to highlight the part hierarchy within the overall database schema. The dashes were chosen as mnemonic symbols as they are parts of the line (Fig. 1).

We shall be referring to a running example describing the editorial page of a newspaper (Fig. 2) in the remainder of this paper. (The model was gleaned from *The New York Times*. The ellipsis indicates that the other parts of *Newspaper* have been omitted.) This schema demonstrates the hierarchical nature of the part relationship.

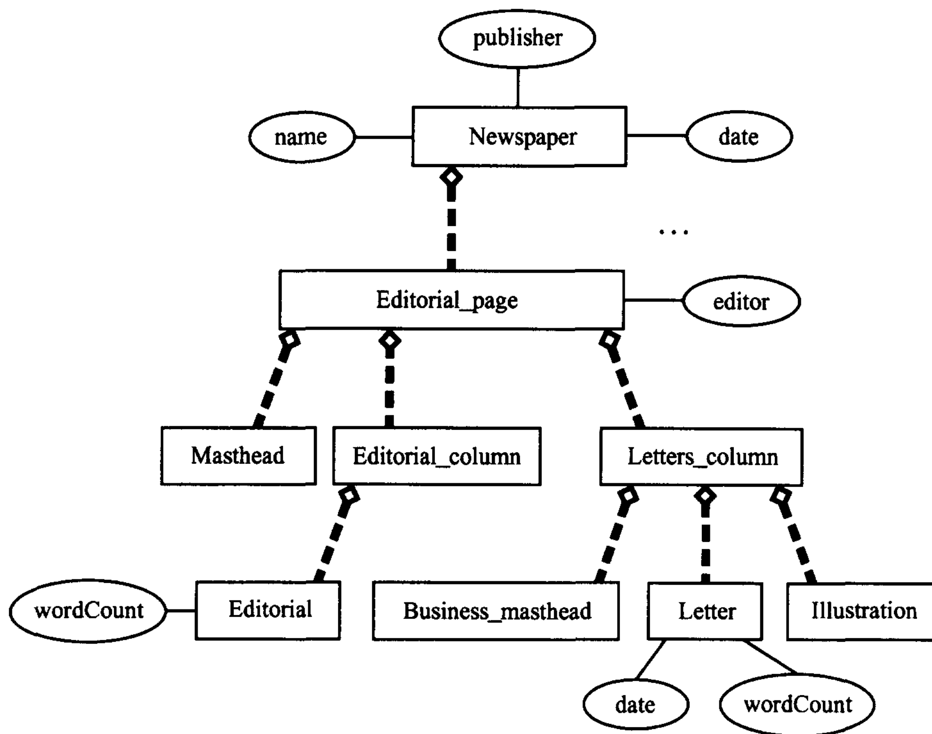


Fig. 2. Part schema for an editorial page from a newspaper.

[1] An occurrence of a part relationship (i.e., an instance-level connection) will be drawn using a curved version of this symbol (see Fig. 3).

3.2. Exclusiveness dimension

The exclusiveness dimension of the part relationship $P_{B,A}$ regulates the way that parts may be distributed among different wholes. In particular, each value from its domain

{*global-exclusive*, *class-exclusive*, *shared*}

imposes a different set of constraints on the cardinalities of the whole sets of the instances of B. Before formally defining these constraints, let us consider the ways we might want to distribute parts among wholes; this will motivate our three-way distinction.

Assume that we have two classes *Engine* and *Boat* and a part relationship between them. If an engine is part of some boat, then it cannot be part of another boat. In other words, two boats, naturally, cannot share one engine. Such a constraint is referred to as *exclusiveness*. It is perhaps the most intuitive constraint which may be imposed on a part-whole relationship as it is a fundamental characteristic of physical assemblies such as boats, bridges, and buildings—things that one can 'go out and kick' [14].

Beyond the fact that two boats cannot share an engine, it is also the case that a boat and a car cannot share one, either. Hence, in a database (Fig. 3), the exclusiveness constraint of the part relationship between *Engine* and *Boat* must also constrain the part relationship between *Engine* and *Car*. In fact, it must constrain any part relationship that *Engine* participates in as the part class. Therefore, to be more precise, our original observation must be refined: If an engine e is part of a boat b, then it cannot be part of any other object o, *regardless of that object's class*.

The exclusiveness just described is often contrasted with *sharing*, which allows a part to be shared by any number of wholes [34,46]. Such sharing is common among 'logical' part relationships [34]. For example, scientific articles may appear in books that are collections of reprinted articles. Such books may freely share articles as parts.

While exclusiveness is valid for physical assemblies, there are situations where it is too rigid. Consider a document database for all the publications of an organization which sponsors technical conferences. A partial schema for such a database is shown in Fig. 4 where there appear six classes and various part relationships between them. We see that a proceedings has articles as parts, and an article, in turn, has an abstract. It is obvious that two articles cannot share the same abstract, so exclusiveness should be imposed there. However, a conference program might contain the abstracts of

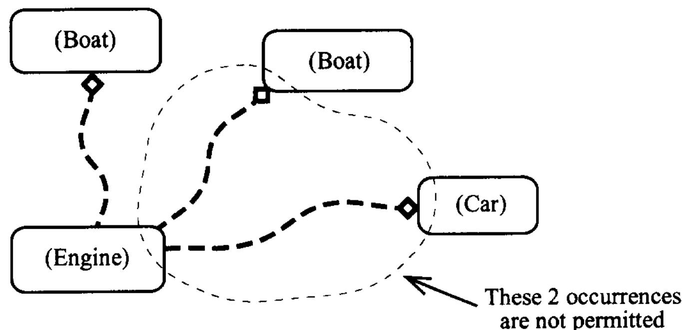


Fig. 3. A global-exclusive engine of a boat.

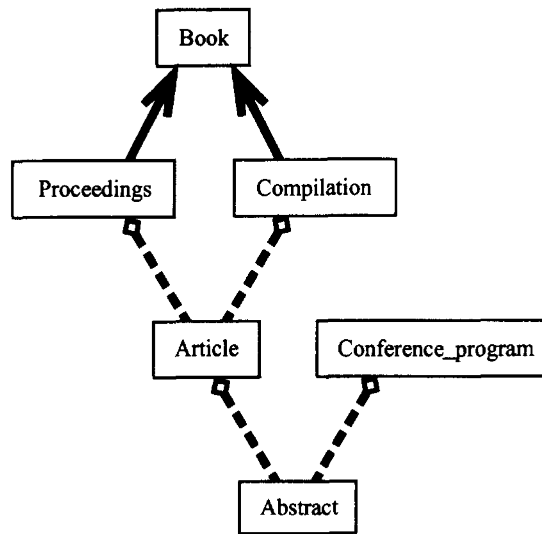


Fig. 4. A document database schema.

the articles appearing in the proceedings. Therefore, while an abstract cannot be shared among articles, it can be part of both an article and a conference program.

A similar situation exists in the case of articles and their relationships to proceedings and compilations. As noted, conference articles may be reprinted in compilations. Even though a given article may only appear in a single proceedings, it can also be part of some compilations. Therefore, the exclusiveness constraint is again inappropriate. What is required is the imposition of exclusiveness on a single part relationship (that between *Article* and *Proceedings*) and its relaxation with respect to others. For this reason, we distinguish between two types of exclusiveness:

- *Global exclusiveness*: Enforces the exclusive reference constraint on the entire database.
- *Class exclusiveness*: Enforces the exclusive reference constraint within a single class, and relaxes it otherwise.

Assume that there exist n part relationships P_{B,A_1}, $P_{B,A_2}, \ldots, P_{B,A_n}$, as in Fig. 5, with constituent relations $\diamond_{(B,A_1)}, \diamond_{(B,A_2)}, \ldots, \diamond_{(B,A_n)}$, respectively.

Definition 3. For a part relationship P_{B,A_i}, $\chi = \textit{global-exclusive}$ implies that $\forall b \in E(B)$, if $\exists a \in E(A_i)$ such that $(b, a) \in \diamond_{(B,A_i)}$, then $|H_{\diamond_{(B,A_i)}}(b)| = 1$ and $\forall j \neq i, |H_{\diamond_{(B,A_j)}}(b)| = 0$. P_{B,A_i} is said to be a global-exclusive part relationship.

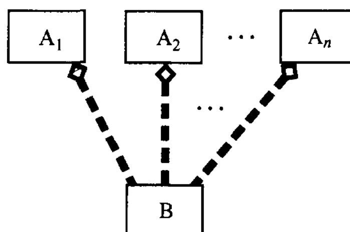


Fig. 5. Class B in n part relationships.

As we see, a global-exclusive part relationship P_{B,A_i} not only places constraints on the whole sets defined with respect to itself, but on all whole sets defined with respect to the class B. The existence of P_{B,A_i} imposes the following limits on 'part' references to instances of B: If an instance $b \in E(B)$ has a whole $a \in E(A_i)$, then a is b's only whole in the entire database.

Definition 4. For a part relationship $P_{B,A}$, $\chi = \textit{class-exclusive}$ implies that $\forall b \in E(B)$, $|H_{\diamond_{(B,A)}}(b)| \leq 1$. $P_{B,A}$ is said to be a class-exclusive part relationship.

In other words, the part relationship between a class B and a class A is class-exclusive if $\diamond_{(B,A)}$ is a partial function from $E(B)$ to $E(A)$. It should be noted that global and class exclusiveness are equivalent if the part class participates in only a single part relationship.

Definition 5. For the part relationship $P_{B,A}$, $\chi = \textit{shared}$ implies that $\forall b \in E(B)$, $P_{B,A}$ does not impose any constraints on $|H_{\diamond_{(B,A)}}(b)|$. $P_{B,A}$ is said to be a shared part relationship.

Graphically, we add an 'X' to the part relationship symbol to indicate that it is global-eXclusive (Fig. 6). An 'X' inscribed in a rectangle adorns the symbol in the case of class eXclusiveness. Sharing is indicated by the lack of these two embellishments (Fig. 7).

3.3. Multiplicity dimension

The next characteristic dimension of the part relationship determines how parts from a part class can be grouped together in the formation of wholes. In particular, it imposes cardinality restrictions on the sets of such parts in accordance with the specified value from its domain:[2]

$$\{(l, u) | l \in \{0, 1, 2, \ldots\}, u \in \{1, 2, 3, \ldots\} \cup \{\infty\}, l \leq u\}$$

As an example, an editorial column (Fig. 2) can be defined as always having between three and four editorials (as *The New York Times* does). Or a letters-column can be required to have at least one letter and, at all times, a single business masthead.

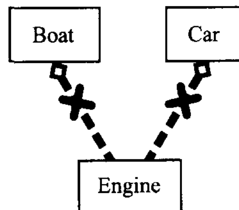


Fig. 6. Global exclusive part relationships.

[2] We have also developed part relationships which permit ordering among the parts. See [22,23] for further details.

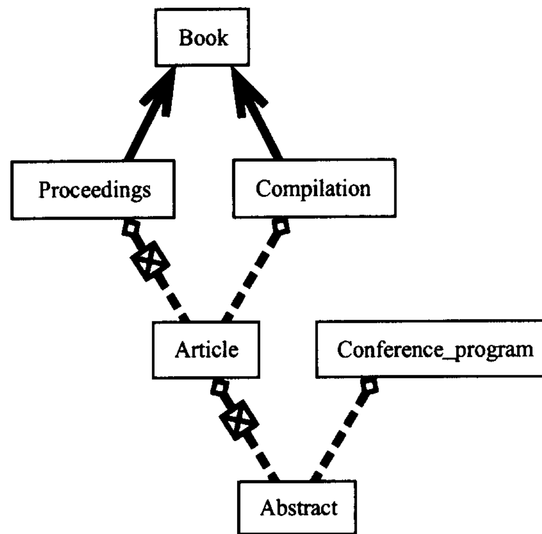


Fig. 7. Revised document schema with class exclusiveness and sharing.

Definition 6. For a part relationship $P_{B,A}$, $\kappa = (l, u)$ implies that $\forall a \in E(A)$, $l \leq |M_{\diamond_{(B,A)}}(a)| \leq u$. The condition $u = \infty$ means that the upper bound does not apply.

To summarize, κ defines a constraint on the number of parts of a type B that any whole of type A can have. This constraint is enforced throughout the entire lifetime of the 'whole' object, from the moment it is created to the moment it is destroyed.

Pictorially, the multiplicity is shown in a number of different ways. In general, if $l > 1$ or $u > 1$ (meaning that wholes can have more than one part of the given type), then the dashed line of the part relationship is doubled up to convey multiplicity. The value of κ is placed alongside (Fig. 8).

We employ graphical symbols instead of the number pairs for certain multiplicities. *Single-valued* part relationship: If $l = 0$ and $u = 1$, κ is omitted and the basic single-lined symbol is used (Fig. 9). With $l = 0$ and $u = \infty$ (*multi-valued* part relationship), κ is again omitted, but the dual-line is retained (Fig. 10). A part relationship for which $l = u = 1$ is called *essential* and is represented by the single-lined part relationship symbol with a circle (Fig. 11). Finally, if $l = 1$ and $u = \infty$ (*multi-valued*

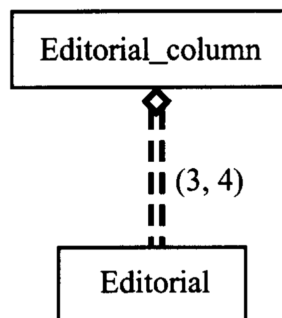


Fig. 8. *Editorial_column* schema.

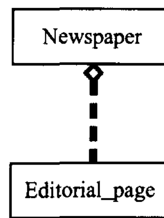


Fig. 9. A single-valued part relationship.

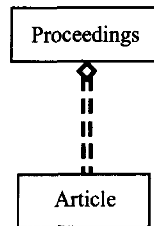


Fig. 10. A multivalued part relationship.

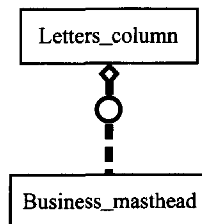


Fig. 11. Letters-column with essential business masthead.

essential part relationship), then $P_{B,A}$ is represented by the dual-line with a circle. In Fig. 12, each letters-column has at least one letter but no maximum.

3.4. Dependency dimension

Dependency semantics is often desired when modeling with parts [34], especially when the wholes comprise numerous parts. For example, having the parts of a large CAD drawing deleted

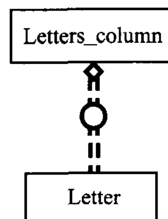


Fig. 12. Letters-column with at least one letter.

automatically when the drawing itself is deleted alleviates tedious manipulation. For this reason, we include *part-to-whole dependency* in our part model.

There are part-whole configurations where a part acts as a defining element, without whose existence the whole becomes insubstantial. Consider, for example, that without its frame, a bicycle may be seen as nothing more than a collection of 'spare' parts. Therefore, it makes sense to propagate the deletion of a frame into the deletion of its bicycle. We call this *whole-to-part dependency*. The domain of the dependency dimension includes three values:

$$\{part\text{-}to\text{-}whole, whole\text{-}to\text{-}part, nil\} .$$

The value *nil* indicates no dependency. The other two values are defined below, where the notation $del(x)$ is used to denote the application of an operator that deletes the instance x from the database [34,37]. Actually, *del* is taken to be a predicate that returns TRUE if the deletion is successful and FALSE otherwise. The deletion of the object takes place as a side-effect.

Definition 7. For the part relationship $P_{B,A}$, $\delta = part\text{-}to\text{-}whole$ implies that

$$\forall a \in E(A), \forall b \in E(B)[[(b, a) \in \Diamond_{(B,A)} \wedge H_{\Diamond_{(B,A)}}(b) = \{a\}] \Rightarrow [del(a) \Rightarrow del(b)]]$$

$P_{B,A}$ is called a *part-to-whole dependent* part relationship.[3] In words: If a is the only whole of b, and a is deleted, then b must be deleted for a part-to-whole dependent relationship.

Definition 8. For the part relationship $P_{B,A}$, $\delta = whole\text{-}to\text{-}part$ implies that

$$\forall a \in E(A), \forall b \in E(B)[[(b, a) \in \Diamond_{(B,A)} \wedge M_{\Diamond_{(B,A)}}(a) = \{b\}] \Rightarrow [del(b) \Rightarrow del(a)]]$$

$P_{B,A}$ is called a *whole-to-part dependent* part relationship.

In both definitions, the condition requiring that the deleted object (for example, a in the case of part-to-whole dependency) be the only existing referent implies a 'multivalued' deletion semantics. In other words, the deletion is not propagated until the set of referents on which a given object depends becomes empty. (Cf. [34].)

To express the dependency graphically, an arrowhead facing in the direction of the dependency (i.e., against the direction of the deletion propagation) is placed behind the diamond. Fig. 13 shows the dependency of the table of contents on its book; when a book is deleted, its table of contents is discarded automatically. Fig. 14 shows the converse dependency of bicycle on its frame.

3.5. Inheritance dimension and derived attributes

Inheritance (of properties) plays a major role in OODB schemata, where it promotes more comprehensible conceptual models and the reuse of specifications. By the term *inheritance* we mean the attainment of (the definition of) a property by one class from another class by virtue of the fact that the two classes are connected via some hierarchical semantic relationship. Let us note that inheritance can be either automatic or selective. With respect to the IS-A relationship, which is the

[3] Strictly speaking, the antecedent must be temporally before the consequent.

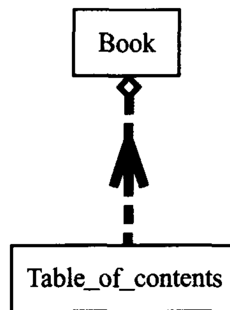


Fig. 13. Table of contents dependent on its book.

primary instrument of property inheritance in an OODB schema, inheritance is always automatic. For example, the class *Graduate_student* exhibits the property *ID#* because the class *Student* defines that property and *Graduate_student* IS-A *Student*. For the part relationship, the inheritance is selective, meaning that the designer needs to choose which properties will be inherited.

The usefulness of IS-A inheritance stems from the fact that it comes directly from ordinary human usage: A subcategory of objects (e.g., dog) has all the properties of its supercategory (e.g., mammal). For the same reason, it is also important to endow the part relationship with certain inheritance capabilities. Consider that the age of an airplane is the age of its airframe [46]. Furthermore, an airplane gets its color from its fuselage, or, alternatively, it may be regarded as multi-colored and get the colors from its fuselage, wings, and tail.

The above are all examples of *upward inheritance* from the part class to the whole class. Our editorial page schema (Fig. 2) contains some examples of *downward inheritance* from the whole class to the part class. The date of an editorial page is just the date of the newspaper where it appears. Furthermore, the date of each editorial is that of its editorial page. It may also be the case that a font family is defined as a property of the newspaper overall and inherited by the various parts (like the editorial page) in order to maintain a consistent look (cf. [38]).

Inheritance for the part relationship differs in a number of ways from IS-A inheritance. First, the inheritance of IS-A is, at bottom, just a template sharing mechanism [19] which transmits the definitions of *all* properties from one class (the superclass) to another (the subclass). The assignment

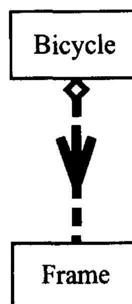


Fig. 14. Bicycle dependent on its part frame.

of values to these properties for instances of the subclass is not a function of this process.[4] In contrast, part-whole inheritance, as defined below, provides the receiving class with appropriate definitions for the inherited properties, and propagates values for those properties to the instances. Definitionally, a car has the color of its body, and, more specifically, some Ford Mustang (an instance) is blue because it has a blue body. Therefore, we refer to a property inherited via a part relationship as a *derived property* or *derived attribute*. Thus, while IS-A inheritance is strictly a schema-level mechanism, part-whole inheritance is both a schema-level mechanism and an instance-level mechanism. The latter promotes more concise representations at the instance level of a database because data is stored in one location (e.g., at a car's body) and propagated on demand to another location (e.g., to the car); property values are not replicated, which alleviates the burden of explicit integrity maintenance associated with redundant data storage [15].

The second distinction is that IS-A implies inheritance of all properties, whereas the part relationship requires selective inheritance. This again has its root in our everyday usage of parts, where the inheritance of all properties is rarely needed and may in fact be semantically incorrect. For example, the drive-train of a car has a serial number which is not the serial number of the car. It is up to the schema designer to determine which properties are appropriate for inheritance.

The last distinction is that inheritance across IS-A, in its ordinary OODB system usage, is downward, but part-whole inheritance can be either upward or downward. A notion of upward inheritance for IS-A has been proposed for integrating pre-existing classes [49]. In knowledge representation, such a notion has been used to derive default values for properties [11].

Let us note that the inheritance mechanism of the part relationship is not meant to be a general computational facility. That is, it will not *directly* permit a designer to define a derived attribute whose value is computed via some arbitrarily complex procedure. We have sought to separate such general computation from the propagation and combination of data values that humans would associate with inheritance. We have identified three such kinds of inheritance—namely, *invariant*, *transformational*, and *cumulative*—which we have made available to the modeler in a *declarative* way. Declarative representations are widely assumed to be superior to procedural representations [59], fostering understandability, reusability and maintainability.

To be sure, it is possible in an OODB system to program methods that carry out the desired inheritance behavior. However, this puts an unnecessary burden on a programmer who would be forced to write code involving potentially complicated path expressions. The correctness and the maintenance of such code would also fall on the programmer's shoulders.

Let us also emphasize that our model does not forbid a schema designer from specifying a derived attribute of arbitrary complexity with respect to part relationships. We just do not support a declarative means of doing so. The schema designer would have to manually program some method to define such a property. Part-whole inheritance will help in this task.

The three important kinds of part inheritance that we have identified are distinguished by the manner in which the values of inherited properties (derived attributes) are computed. As was alluded to in our examples above, the value of a derived attribute is often identical to the value of its source property. Here, only a simple *invariant* propagation of the data value must be performed. It is obvious

[4] In framed-based knowledge representation [58], 'value inheritance' across IS-A links is sometimes used to give instance frames default values.

that the part or whole which provides the value must be unique (i.e., it must be the only part or whole); otherwise, the value for the derived attribute will be ambiguous.

In part-whole modeling, defining the value of a derived attribute as a combination of values from multiple source objects is also viable. For example, the word-count of an editorial column is the sum of the word-counts of its editorials. Also, the color of a car's body can be modeled as the union of the colors of its panels. In the first example, the multiple values of word-count are combined by adding them. In the second, the colors are gathered into a set (cf. union inheritance [58]).

In our part model, the three different kinds of derived attributes are:

(1) *Invariant*: The inherited value is obtained without modification from a unique source.
(2) *Transformational*: The inherited value is computed by combining (potentially) many source values into a single value of the source property's data type via a single symmetric[5] operator.
(3) *Cumulative*: The inherited value is computed by accumulating (potentially) many source values into a set or bag, with the derived attribute becoming multivalued with respect to the source property's data type.

When a schema designer defines a transformational derived attribute, he or she must specify a symmetric operator that takes a variable number of arguments of some data type and returns a single value of that same data type. We will call such an operator *generic*. The need to accommodate a variable number of arguments is dictated by the fact that the number of part instances for a given whole instance (or vice versa), in general, is not known at schema definition time and can vary during the lifetime of the database. In the example of the property word-count being inherited by the class *Editorial-column* from the class *Editorial*, the generic operator is Σ_Z, integer summation, where summation over a single integer is defined as the identity (i.e., $\Sigma_Z(x) = x$).

The reason for symmetric operators is that there is no ordering among the parts (Section 3.3). This requirement is not overly restrictive because the most useful generic operators for inheritance are iterative applications of a common binary operation like addition, minimum, etc.

Again, let us note that an extra layer of computation can be easily done on top of the symmetric inheritance operators. This would permit the definition of derived attributes of any degree of complexity. However, such properties would not be declaratively specified, but would require the writing of method code.

Formally, the inheritance dimension of the part relationship is a triple:

$$(\Upsilon, \Delta, \Phi) .$$

The first component $\Upsilon \subseteq \Pi(B)$ is called the 'upSet' of the part relationship. It contains those properties of B that are inherited by A. The second component $\Delta \subseteq \Pi(A)$ is the 'downSet.' It contains those properties of A inherited by B. It is required that $\Upsilon \cap \Delta = \emptyset$ in order to ensure that no inherited property is defined circularly.[6] The last component, Φ, is a function which maps the elements of Υ and Δ (i.e., all the inherited properties) into operators that determine the kinds of derived attributes they induce. For an invariant derived attribute, the operator will be the identity operator. In the case of a transformational derived attribute, it will be any generic operator that handles values of the

[5] 'Symmetric' is analogous to 'commutative' [17]. The term is usually applied to operators with three or more parameters.
[6] For the same reason, it is forbidden to have a directed cycle where the same attribute is inherited along all part relationships of the cycle. It is the responsibility of the schema designer to avoid this situation. It is a simple matter to write a procedure to detect such occurrences.

appropriate type. For a cumulative derived attribute, it will be either the set-accumulation operator or the bag-accumulation operator (defined below) of the source's type.

To define (Υ, Δ, Φ), we follow [62] by viewing attributes, relationships, and (readable) methods as functions which map the extension of a class into some given data type. For example, the attribute *height* of class *Person* maps persons into (a subset of) $\boldsymbol{R}$, the real numbers. That is, *height*: $E(Person) \to \boldsymbol{R}$. A property may be undefined for certain elements of its domain.

For data types, we will be using the following notations. $\mathbb{P}(\tau)$ denotes the power set of the data type τ, i.e., that data type which comprises sets of values of τ. $\mathbb{B}(\tau)$ is the data type comprising bags of values of τ. The n-way Cartesian product over τ is written as τ^n. I_τ is the identity operator for τ. To define cumulative derived attributes, we also need the following:

Definition 9. (Accumulation operators) For the data type τ, U_τ denotes the *set-accumulation operator* which is defined as follows:

$$U_\tau(x_1, x_2, \ldots, x_n) = \bigcup_{i=1}^{n} \{x_i\}$$

where $x_1, x_2, \ldots, x_n \in \tau$. As we see, U_τ is the composition of n canonical injections [2] and ordinary set union. G_τ denotes the *bag-accumulation operator* defined as:

$$G_\tau(x_1, x_2, \ldots, x_n) = \langle x_1, x_2, \ldots, x_n \rangle$$

where $x_1, x_2, \ldots, x_n\rangle$ is the bag containing the n values $x_1, x_2, \ldots, x_n \in \tau$.

We can now formally define the three aspects of the inheritance dimension of the part relationship.

Definition 10. (Function Φ) For a part relationship $P_{B,A}$, Φ is a function which maps each element of Υ and Δ into an operator that is used in the computation of the value of its corresponding derived attribute. The value of Φ for a property $\pi \in (\Upsilon \cup \Delta)$ with data type τ_π is as follows:

$$\Phi(\pi) = \begin{cases} I_{\tau_\pi}, & \textbf{Case 1: } \text{Invariant}, \\ T_{\tau_\pi}, & \textbf{Case 2: } \text{Transformational}, \\ U_{\tau_\pi}, & \textbf{Case 3a: } \text{Cumulative (set)}, \\ G_{\tau_\pi}, & \textbf{Case 3b: } \text{Cumulative (bag)}, \end{cases}$$

where T_{τ_π} denotes a generic operator (as introduced above) whose arguments and return value are of data type τ_π.

Note that the value of the function Φ for any inherited property is specified solely by the schema designer depending on the desired inheritance behavior. If an invariant derived attribute is desired, then the operator chosen is the identity for the data type of the source property. For a cumulative derived attribute (Cases 3a and 3b), the operator is either the set-accumulation operator or the bag-accumulation operator of the appropriate data type. With a transformational derived attribute, the schema designer is required to choose some generic operator to transform the many source values into a single value of the same type. In the editorial column example where we defined *wordCount* to be

transformational, we would have $\Phi(wordCount) = \Sigma_Z$, that is, integer summation. Let us note that it is a straightforward matter to incorporate standard type-casting operations used in programming languages into the different cases; however, we have omitted this in order to simplify the presentation. Indeed, this option was discussed in [22]. Its inclusion, for example, would permit the result of an integer summation to be a real number. It would also make it possible for the propagated value to be the average of some integer values, assuming that a 'count' function were available.

Definition 11. (UpSet Υ) For a part relationship $P_{B,A}$, $\pi_B \in \Upsilon$ implies the existence of a new property $\pi_B \in \Pi(A)$ (called a derived attribute) whose value for an instance $a \in E(A)$ is computed in terms of the partial function $\mathbb{D}_{\pi_B}$ which is defined with respect to $\Phi(\pi_B)$ as follows. Let τ be the data type of π_B, and assume $M_{\diamond_{(B,A)}}(a) = \{b_1, b_2, \ldots, b_q\}$.

- **Case 1.** (Invariant) $\Phi(\pi_B) = I_\tau$. $\mathbb{D}_{\pi_B}: E(A) \to \tau$.

$$\mathbb{D}_{\pi_B}(a) = \begin{cases} I_\tau(\pi_B(b_1)) = \pi_B(b_1), & q = 1 \wedge \pi_B(b_1) \text{ is defined}, \\ \text{undefined}, & q = 0, \end{cases}$$

- **Case 2.** (Transformational) $\Phi(\pi_B) = T_\tau$. $\mathbb{D}_{\pi_B}: E(A) \to \tau$.

$$\mathbb{D}_{\pi_B}(a) = \begin{cases} T_\tau(\pi_B(b_1), \ldots, \pi_B(b_q)), & q \neq 0 \wedge \forall i : 1 \leq i \leq q, \pi_B(b_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

- **Case 3a.** [Cumulative (set)] $\Phi(\pi_B) = U_\tau$. $\mathbb{D}_{\pi_B}: E(A) \to \mathbb{P}(\tau)$.

$$\mathbb{D}_{\pi_B}(a) = \begin{cases} U_\tau(\pi_B(b_1), \ldots, \pi_B(b_q)), & q \neq 0 \wedge \forall i : 1 \leq i \leq q, \pi_B(b_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

- **Case 3b.** [Cumulative (bag)] $\Phi(\pi_B) = G_\tau$. $\mathbb{D}_{\pi_B}: E(A) \to \mathbb{B}(\tau)$.

$$\mathbb{D}_{\pi_B}(a) = \begin{cases} G_\tau(\pi_B(b_1), \ldots, \pi_B(b_q)), & q \neq 0 \wedge \forall i : 1 \leq i \leq q, \pi_B(b_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}. \end{cases}$$

Definition 11 states that a new property is defined at class A for each property in Υ. This new property (called a derived attribute) is said to be inherited by A upwardly from B via $P_{B,A}$. It is called invariant,[7] transformational or cumulative (set or bag) depending, respectively, on whether the schema designer has chosen Case (1), (2), (3a) or (3b).

Downward inheritance and the corresponding downSet are defined analogously to the above:

Definition 12. (DownSet Δ) For a part relationship $P_{B,A}$, $\pi_A \in \Delta$ implies the existence of a new property $\pi_A \in \Pi(B)$ (called a derived attribute) whose value for an instance $b \in E(B)$ is computed in

[7] It should be noted that Case (1) does not handle the situation where some $a \in E(A)$ has two or more part instances from class B; only the identity operator is applicable. Our part model deems any schema where such an occurrence is possible as ill-defined and, in order to guard against it, imposes the following 'cross-dimensional' constraint on the part relationship: If there is an upwardly inherited, invariant derived attribute, then the upper bound of the part relationship's multiplicity dimension must be 1 (i.e., the source part must be unique, if it exists).

terms of the partial function $\mathbb{D}_{\pi_A}$ which is defined with respect to $\Phi(\pi_A)$ as follows. Let τ be the data type of τ_A, and assume $H_{\diamond_{(B,A)}}(b) = \{a_1, a_2, \ldots, a_m\}$.

- **Case 1.** (Invariant) $\Phi(\pi_A) = I_\tau$. $\mathbb{D}_{\pi_A}: E(B) \rightarrow \tau$.

$$\mathbb{D}_{\pi_A}(b) = \begin{cases} I_\tau(\pi_A(a_1)) = \pi_A(a_1), & m = 1 \wedge \pi_A(a_1) \text{ is defined}, \\ \text{undefined}, & m = 0, \end{cases}$$

- **Case 2.** (Transformational) $\Phi(\pi_A) = T_\tau$. $\mathbb{D}_{\pi_A}: E(B) \rightarrow \tau$.

$$\mathbb{D}_{\pi_A}(b) = \begin{cases} T_\tau(\pi_A(a_1), \ldots, \pi_A(a_q)), & m \neq 0 \wedge \forall i : 1 \leq i \leq m, \pi_A(a_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

- **Case 3a.** [Cumulative (set)] $\Phi(\pi_A) = U_\tau$. $\mathbb{D}_{\pi_A}: E(B) \rightarrow \mathbb{P}(\tau)$.

$$\mathbb{D}_{\pi_A}(b) = \begin{cases} U_\tau(\pi_A(a_1), \ldots, \pi_A(a_m)), & m \neq 0 \wedge \forall i : 1 \leq i \leq m, \pi_A(a_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

- **Case 3b.** [Cumulative (bag)] $\Phi(\pi_A) = G_\tau$. $\mathbb{D}_{\pi_A}: E(B) \rightarrow \mathbb{B}(\tau)$.

$$\mathbb{D}_{\pi_A}(b) = \begin{cases} G_\tau(\pi_A(a_1), \ldots, \pi_A(a_m)), & m \neq 0 \wedge \forall i : 1 \leq i \leq m, \pi_A(a_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}. \end{cases}$$

Definition 12 states that a new property is defined at B for each property included in the downSet Δ. This new property (called a derived attribute) is said to be inherited by B downwardly from A via $P_{B,A}$. As before, the derived attribute is invariant,[8] transformational or cumulative.

Let us look at the formal definitions of some of the examples cited above. The class *Editorial_page* has the invariant derived attribute *date* which it inherits downwardly from *Newspaper*. Therefore, $\Phi(date) = I_{dateType}$, where *dateType* is *date's* data type at *Newspaper*, and *date* at *Editorial_page* is defined in terms of $\mathbb{D}_{date}$ which is defined as follows:

$$\mathbb{D}_{date}(g) = \begin{cases} date(p), & \text{if } \exists p \in H_{\diamond_{(E,N)}}(g), \\ \text{undefined}, & |H_{\diamond_{(E,N)}}(g)| = 0. \end{cases}$$

where g is an editorial page, and p, if it exists, is its only newspaper. In the above, E is short for *Editorial_page*, and N denotes *Newspaper*. The derived attribute *wordCount*, inherited upwardly by class *Editorial_column* from class *Editorial*, is defined according to the following:

$$\mathbb{D}_{wordCount}(c) = \begin{cases} \sum_{i=1}^{n} wordCount(e_i), & n \neq 0 \wedge \forall i : 1 \leq i \leq n, wordCount(e_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

where $e_1, \ldots, e_n$ are the editorials contained in the editorial column c.

[8] Here, as with an upwardly inherited, invariant derived attribute, the uniqueness of the source property must be ensured if the schema is to be well defined. This is done by imposing the following additional constraint on the part relationship: If there is a downwardly inherited, invariant derived attribute, then the part relationship must be either global-exclusive or class-exclusive (i.e., the source whole must be unique).

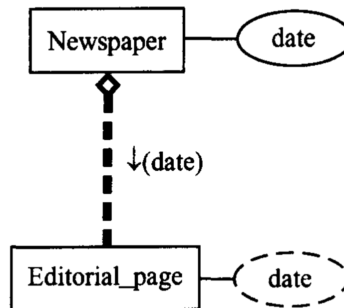


Fig. 15. Inheritance of *date*.

For the example of class *Body* inheriting the cumulative derived attribute *color* upwardly from class *Panel*, we have $\Phi(color) = U_{colorType}$, where *colorType* is the data type of *color*, and:

$$\mathbb{D}_{color}(y) = \begin{cases} \bigcup_{i=1}^{n} \{color(p_i)\}, & n \neq 0 \wedge \forall i : 1 \leq i \leq n,\ color(p_i) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

where y is a car body and the p_i's are its panels. Note that the data type of *color* at class *Body* is $\mathbb{P}(colorType)$, i.e., the one comprising sets of colors.

Inheritance via a part relationship is represented graphically by two aspects. First, the derived attribute is drawn as a dashed ellipse at the class that inherits it. The dashed appearance reminds one of the part relationship line itself. Second, a *propagation label* is written alongside the part relationship line. It begins with an arrow indicating the direction of inheritance: upward (i.e., part to whole) is denoted by an up-arrow; downward, by a down-arrow. Depending on the kind of derived attribute, one of the following comes after the arrow.

- *Invariant*: The name of the property in parentheses (Fig. 15).
- *Transformational*: A pair of parentheses containing a symbol denoting the value of Φ for the given property and the property's name in square brackets. For a common operator like summation '+' would be used for Φ's value (Fig. 16). If no such symbol exists, then a generic one (like '$\odot$') is employed and an annotation is placed in a schema legend.

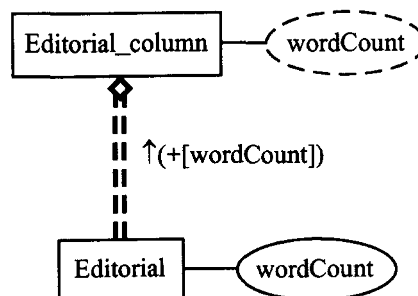


Fig. 16. Inheritance of *wordCount*.

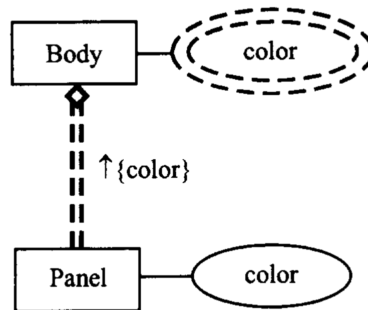


Fig. 17. Inheritance of *color*.

- Cumulative: Set: The name of the property in curly brackets (Fig. 17). Bag: The name in angled brackets. Note also that because the derived attribute is multivalued in this case, it is drawn with a double-rimmed ellipse (Fig. 17).

4. Generalized derived attributes

In this section, we turn our attention to the 'redundant inheritance of a given property' problem, an example of which is depicted in Fig. 18 where we see the class A inheriting the property π upwardly from each of the classes $B_1, B_2, \ldots, B_m$. According to our discussion so far, this schema must be viewed as ill-defined because the definition of π is ambiguous. Which of the π's at the m part classes should be used to determine the value of π (for some instance) at A?

In an IS-A hierarchy that permits multiple inheritance (i.e., a DAG structure), there is the possibility of inheriting competing definitions for the same property. This competition or redundancy can be resolved by applying a precedence mechanism [32,53]. In a part hierarchy, redundant inheritance of a given property does not necessarily signify competing definitions. Since it is natural for a whole to consist of several parts of different kinds, there is a conceptually meaningful alternative and that is to combine the data values provided via the *different part relationships* into one value using a specified

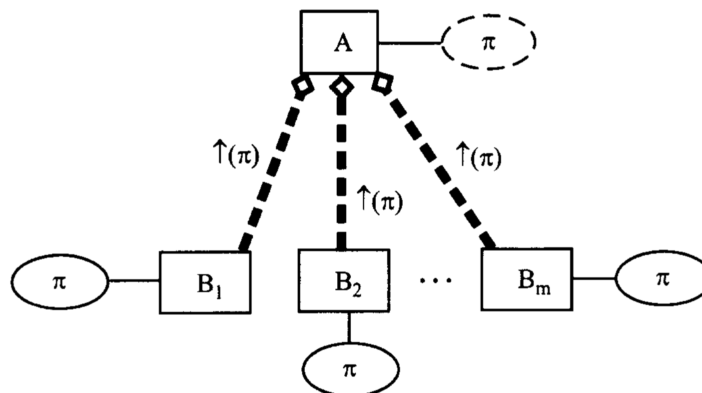


Fig. 18. Redundant upward inheritance.

transformation or accumulation. In fact, in many modeling situations it is sensible to define a property of a whole class in terms of the same property at a number of its part classes.[9] Examples abound: a plane gets its color from its fuselage, wings, and tail; the weight of a car is the sum of the weights of its parts; etc.

As we see, the seeming redundant inheritance problem is, in part hierarchies, actually a desired modeling scheme that generalizes ordinary part-whole inheritance. We call a property defined in this manner a *generalized derived attribute* because its definition supersedes those of the derived attributes that would have been induced by the individual inheritances in isolation.

Let us now formally define the notion of a generalized derived attribute.

Definition 13. The existence of part relationships $P_{B_1,A}, P_{B_2,A}, \ldots, P_{B_m,A}$ such that π is in the upSet Υ of each[10] (Fig. 18) implies the existence of a new property $\pi \in \Pi(A)$ (called a generalized derived attribute). The value of π for an instance $a \in E(A)$ is computed in terms of the partial function $\mathbb{D}_\pi$ which is defined with respect to the partial functions $\mathbb{D}_\pi^{(1)}, \mathbb{D}_\pi^{(2)}, \ldots, \mathbb{D}_\pi^{(m)}$ (associated, respectively, with $P_{B_1,A}$ $P_{B_2,A}, \ldots, P_{B_m,A}$ by Definition 11) and the operator ψ as follows.

$$\mathbb{D}_\pi(a) = \begin{cases} \psi[\mathbb{D}_\pi^{(1)}(a), \mathbb{D}_\pi^{(2)}(a), \ldots, \mathbb{D}_\pi^{(m)}(a)], & \forall i : 1 \leq i \leq m,\ \mathbb{D}_\pi^{(i)}(a) \text{ is defined}, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

under the assumption that the following three restrictions are met:

(1) The property π at each of the part classes is either a single-valued or multivalued property of the same underlying data type τ. That is, each π has data type τ, $\mathbb{P}(\tau)$, or $\mathbb{B}(\tau)$.

(2) The functions $\mathbb{D}_\pi^{(1)}, \mathbb{D}_\pi^{(2)}, \ldots, \mathbb{D}_\pi^{(m)}$ all have the same range which is either τ, $\mathbb{P}(\tau)$, or $\mathbb{B}(\tau)$. Note that according to Definition 11 it is not possible for the range of a $\mathbb{D}_\pi^{(i)}$ to be τ if its domain is $\mathbb{P}(\tau)$ or $\mathbb{B}(\tau)$.

(3) The operator ψ has a range of either τ, $\mathbb{P}(\tau)$, or $\mathbb{B}(\tau)$. (The same is true of $\mathbb{D}_\pi$.) If any of the arguments to ψ is a set (bag), then its return value must be a set (bag) as well.

Note that in (1)–(3), τ itself may be a set type of another basic data type. In such a case, $\mathbb{P}(\tau)$ ($\mathbb{B}(\tau)$) would comprise sets of sets (bags of bags). Depending on the height of the hierarchy, more deeply nested structures might be created. Restrictions (1) and (2) ensure compatibility between the data types of the π's (the source properties) and that of the generalized derived attribute. Restriction (1) says that the source properties must all be of the same base type. The schema designer is responsible for ensuring 'semantic compatibility.' As in the previous section, we note that type-casting among compatible data types, as it is used in languages like C++, can easily be introduced in order to transparently relax the restrictions and give the designer more flexibility. In this way, for example, the property π at one class could be of type integer, while at another it could be of type real. Such a situation does not impede the definition of the generalized derived attribute.

Restriction (2) states that the values delivered to the operator ψ via the $\mathbb{D}_\pi^{(i)}$'s must all be of the

[9] In this section, we will limit our discussion to upward inheritance for the sake of brevity.

[10] In such a situation, we further assume that there does not exist any part relationship $P_{A,Q}$ via which class A inherits π downwardly. Allowing such a scenario could lead to a contradiction in the definition. Besides, from a modeling standpoint, it is arguably meaningless for a class to inherit the same property upwardly and downwardly simultaneously.

same data type, namely, τ, $\mathbb{P}(\tau)$, or $\mathbb{B}(\tau)$. If there are any discrepancies among the data types of the π's (e.g., some are τ while others are $\mathbb{P}(\tau)$), then these must be resolved by the $\mathbb{D}_{\pi}^{(i)}$'s before the values are combined by ψ to produce the value of the generalized derived attribute. Type-casting, as mentioned above, could be employed in the $\mathbb{D}_{\pi}^{(i)}$'s.

Restriction (3) states that the data type of the generalized derived attribute must be single-valued or multivalued with respect to the same type as its source properties. If any of the π's are multivalued (i.e., have the type $\mathbb{P}(\tau)$ or $\mathbb{B}(\tau)$), then ψ: $[\mathbb{P}(\tau)]^m \rightarrow \mathbb{P}(\tau)$ or ψ: $[\mathbb{B}(\tau)]^m \rightarrow \mathbb{B}(\tau)$. In other words, ψ in that case must be an operation such as set or bag union.

What (2) and (3) seek to avoid is the deep nesting of sets within sets between two adjacent levels of the hierarchy, something for which we have not found any use. To reiterate, though, such deeply nested sets are available for multi-level hierarchies.

As an example, the definition of the property *wordCount* at the class *Editorial_page* is as follows:

$$\mathbb{D}_{WordCount}(p) = \begin{cases} wordCount(c) + wordCount(l)\,, & wordCount(c),\ wordCount(l) \text{ defined}\,, \\ \text{undefined}\,, & \text{otherwise}\,, \end{cases}$$

where p is an editorial page, and c and l are its editorial column and letters-column, respectively. Here, the operator ψ is summation. As shown in the previous section, the property *wordCount* at the class *Editorial_column* is itself a transformational derived attribute defined via an upward inheritance. (The same is true of *wordCount* at *Letters_column*.) Thus, we see that derived attributes can be used in the same manner as other 'ordinary' properties of a class.

The graphical symbol for a generalized derived attribute is the same as that for a derived attribute with the addition of the operator ψ which is placed in front of the derived attribute's name, now appearing in brackets. In Fig. 19, we show the property *wordCount* being inherited by class *Editorial_page*, and also show that *wordCount* is a transformational derived attribute defined in terms of summation at both *Editorial_column* and *Letters_column*. Furthermore, *wordCount* is an inherent property of *Editorial* and *Letter*, both of which are two levels below *Editorial_page*.

As another example, Fig. 20 shows how a car's weight can be written as the sum of the weights of

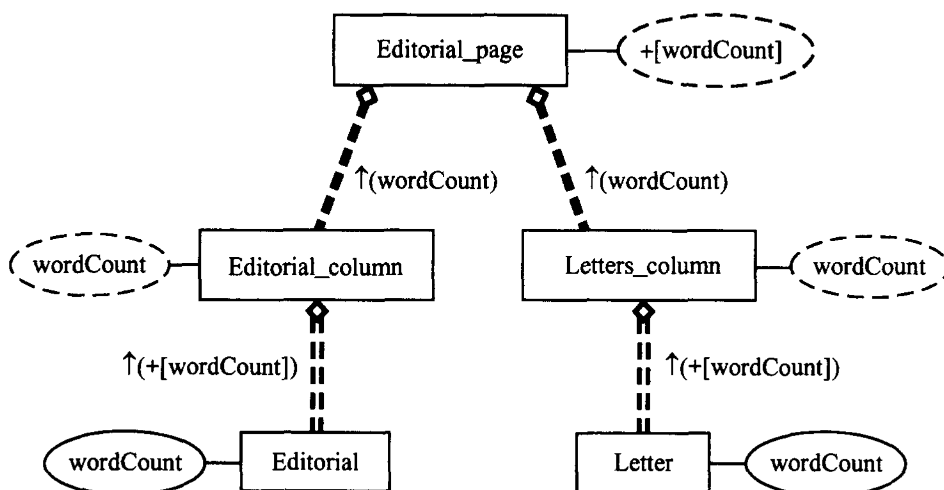


Fig. 19. Inheritance of *wordCount*.

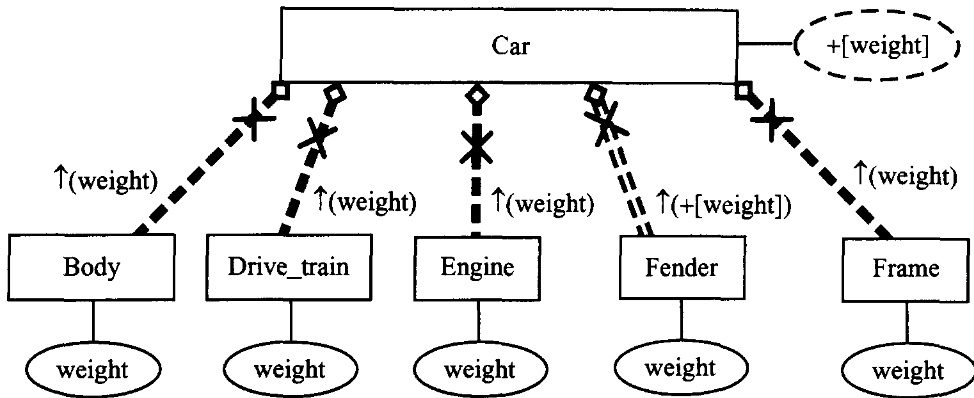


Fig. 20. Inheritance of *weight*.

its parts. In an expanded version of this schema, the property *weight* at the different part classes might itself be inherited upwardly.

To conclude the discussion of the theoretical aspects of the part relationship, we show the revised version of the editorial page schema in Fig. 21. It now includes specifications for the various

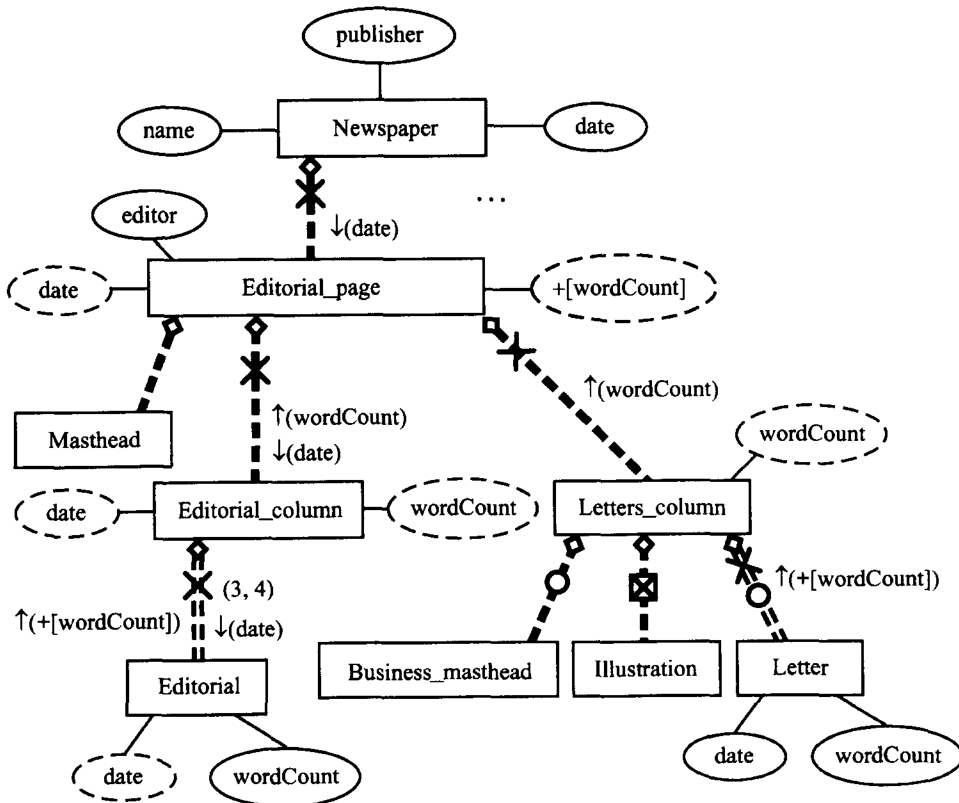


Fig. 21. Revised editorial page schema.

characteristic dimensions and derived attributes. Note that *Illustration* has a class-exclusive part relationship with *Letters_column* since the same illustration should not be part of more than one letters-column, but can be used as part of an article or in a contents announcement elsewhere in the newspaper. The part relationships between *Masthead* and *Editorial_page* and between *Business_masthead* and *Letters_column* are *shared* because mastheads, which contain general information about a newspaper's editorial staff, change infrequently. Therefore, the same masthead may appear in a series of newspapers spanning months or years.

5. Integration of the part relationship into an OODB system

In this section, we describe an actual implementation of the part relationship in the context of an OODB system. The implementation was carried out using the metaclass facility of VML (VODAK Model Language) [37]. We will first give an overview of the VML data model and its notion of metaclass. Then, we go on to discuss the different aspects of our PartWhole metaclass.

5.1. The VML data model and metaclasses

VML [37] is an open object-oriented data model which can be tailored to the needs of specific applications [36]. It employs a 'dual' model, i.e., it separates the notions of class and object type. Each class in a VML schema is associated with exactly one object type, called its *instance type*, which defines the structure and behavior of the class's instances. In Fig. 22, we have used a shading pattern to show the effect of the instance type[11] on an instance. A single object type, on the other hand, may be associated with any number of classes.

To maintain uniformity, classes themselves are objects in VML (cf. Smalltalk [19]). As such, they are instances of other classes referred to as *metaclasses*. However, the interaction between metaclasses, object types and (ordinary) classes is different from that between classes, object types and instances [36,37]. As with an ordinary class, a metaclass has an instance type that describes its instances, which are classes. Furthermore, a second object type may be associated with a metaclass to augment the structure and behavior of the *instances of its instances*. This additional object type is

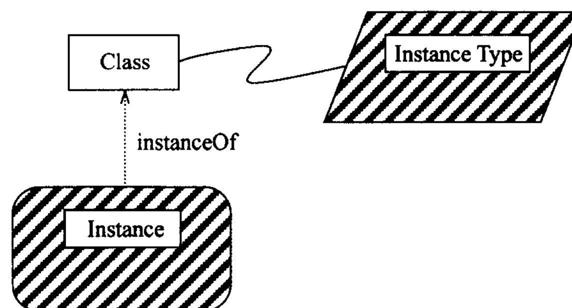


Fig. 22. The instance type's effect on a class's instances.

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

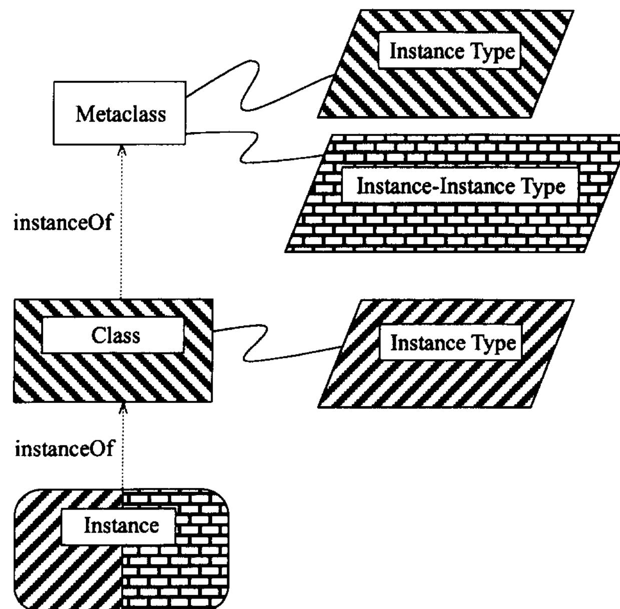


Fig. 23. The interaction between metaclasses, classes and instances.

called an *instance-instance type* [36]. Thus, through its two associated object types, a metaclass influences both its own instances, which are classes, and the instances of those classes. This is illustrated in Fig. 23 where we have again employed shading to demonstrate the effect of the metaclass's instance type on a class. Fig. 23 also shows the effect of the metaclass's instance-instance type and the class's instance type on the class's own instances.

The introduction of new metaclasses serves as the means for introducing semantic relationships, like the part relationship, into the VML data model. We have built the *PartWhole* metaclass that endows classes participating in a part hierarchy [i.e., PartWhole (PW) classes] and their instances with structure and behavior befitting the part relationship. Specifically, through its instance type and instance-instance type, our metaclass does the following:

- It provides the means for defining a part relationship between a pair of PW classes, making one a whole class and the other a part class with respect to each other. It also allows PW classes to be queried regarding the characteristics of their part relationships.
- It furnishes a PW class with the constructor method *make* and the destructor method *destroy* which, respectively, capture the creation and deletion semantics of the part relationship.
- It provides a PW class's instances with methods that carry out the part transactions of establishing, dissolving, changing, and querying part-whole connections between themselves and other objects. These include *addPart*, *removePart*, *changePart*, *getParts*, and *getWholes*.
- It provides the means by which the value propagation associated with part-whole inheritance may be carried out.

In succeeding sections, we will discuss the details of the instance type and the instance-instance type of the PartWhole metaclass and describe the different functionality that each contributes.

5.2. PartWhole instance type

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

The public interface for the PartWhole instance type appears in the following, where OID denotes the built-in VML data type representing object identifiers.[12] It shows that the PartWhole metaclass provides twelve new methods for all PW classes. (Remember that a class is itself an object, and these methods augment the behavior of a class, not the behavior of its instances.) *Make* and *destroy* are used to create and delete instances of a PW class. As such, they are used to encode the creation and deletion semantics of the part relationship (Section 5.2.2). The other methods are used to define and query the part relationships in which a PW class participates. Note that we store all information about the characteristic dimensions with the whole class.

```
OBJECTTYPE PartWhole_InstType SUBTYPEOF Metaclass_InstType;
INTERFACE
  METHODS
//
//Constructor and Destructor for the instances of a PW class.
     make(someParts: {OID}): OID;
     destroy(anObject : OID);
//
//Used to define PW class's part relationships.
     defPartRelshps(someRelshps: {PartRelationshipType});
     defWholeClasses(someClasses: {OID});
     defGenDerivedAttrs(someDrvdAttrs: {AttributeSpecType});
//
//Used to obtain values of the characteristic dimensions of the specific part
//relationships that a PW class participates in. The formal parameter
//'aPWClass' of each method represents (the OID of) the part class of the
//part relationship of interest.
     exclusiveness(aPWClass: OID): ExclusivenessType;
     minCardinality(aPWClass: OID): INT;
     maxCardinality(aPWClass: OID): INT;
     dependency(aPWClass: OID): DependencyType;
     propertyUpwardInherited(propertyName: STRING, aPWClass: OID): BOOL;
     propertyDownwardInherited(propertyName: STRING, aPWClass: OID): BOOL;
     phi(propertyName: STRING, aPWClass: OID): STRING;
END;
```

[12] For the sake of brevity, we have omitted some extraneous utility methods which are described in [26]. Note also that, as required by VML, this instance type is defined as a subtype of 'Metaclass_InstType'; see [37] for details.

5.2.1. Creating and querying a PW class

To demonstrate the impact of introducing part relationships into classes, let us first consider the the task of creating an 'ordinary' class (i.e., one without any part relationships) in VML. In the following we show the declarations for the two classes *Newspaper* and *Editorial_page* (Fig. 21) without their part relationships. Because each class in VML has an associated instance type, a complete declaration is divided into two portions: (1) an instance type declaration and (2) the class declaration itself which contains a reference to the instance type via its 'INSTTYPE' clause.

```
CLASS Newspaper                 CLASS Editorial_page
  INSTTYPE NewspaperType          INSTTYPE EPType
END;                            END;
OBJECTTYPE NewspaperType;       OBJECTTYPE EPType;
IMPLEMENTATION                  IMPLEMENTATION
  PROPERTIES                      PROPERTIES
    name: String;                   editor: String;
    publisher: String;          END;
    date: dateType;
END;
```

To add the part relationships to the schema, the declarations must be modified in two ways. First, each class must be made an instance of the PartWhole metaclass by including a METACLASS clause in its declaration [37]. If no such clause is included (as above), then the class is assumed to be an instance of an intrinsic default metaclass.

The second modification is to add specifications for the part relationships, including their characteristic dimensions and any generalized derived attributes. This is done in the class's 'INIT' clause[13] using the two companion methods *defWholeClasses* and *defPartRelshps* as well as the method *defGenDerivedAttrs*. The method *defWholeClasses* informs a newly created PW class (at the time it is instantiated) of all other PW classes that are whole classes with respect to it. Likewise, *defPartRelshps* notifies it of all related part classes. However, this latter method also provides all the information pertaining to the characteristic dimensions of the part relationships in which the new PW class is the whole class because, as we noted, that data is stored there. The method *defGenDerivedAttrs* informs the class of any generalized derived attributes it may have and their transformations. To illustrate the above, we expand our code to include the part relationships that the two classes are involved in. (The object types have been omitted.)

```
CLASS Newspaper METACLASS PartWhole
  INSTTYPE NewspaperType
  INIT Newspaper->defPartRelshps(
    {
```

[13] The INIT clause is a characteristic of all classes in VML [37]. It is used as a means for performing initialization procedures at the time the class is created.

```
      [thePartClass:Editorial_page,
      exclusiveness:GLOBAL-EXCLUSIVE,
      multiplicity:[min: 0, max: 1],
      dependency:NIL,
      upSet:{},downSet:{'date'},phi:{['date','dateIdentity']}]
    })
END;
CLASS Editorial_page METACLASS PartWhole
  INSTTYPE EPType
  INIT Editorial_page -> defWholeClasses({Newspaper})
    Editorial_page -> defPartRelshps(
      {
        [thePartClass:Masthead,
        exclusiveness:SHARED,
        multiplicity:[min: 0, max: 1],
        dependency:NIL,
        upSet:{},downSet:{},phi:{}],
        [thePartClass:Editorial_column,
        exclusiveness:GLOBAL_EXCLUSIVE,
        multiplicity:[min: 0, max: 1],
        dependency:NIL,
        upSet:{'wordCount'},downSet:{'date'},
        phi:{['wordCount','intIdentity'],['date','dateIdentity']}],
        [thePartClass:Letters_column,
        exclusiveness:GLOBAL_EXCLUSIVE,
        multiplicity:[min: 0, max: 1],
        dependency:NIL,
        upSet:{'wordCount'},downSet:{},phi:{['wordCount','intIdentity']}]
      })
    Editorial_page -> defGenDerivedAttrs({['wordCount','add']})
END;
```

In the INIT clause of *Editorial_page*, the invocation of *defWholeClasses* establishes the fact that *Newspaper* is *Editorial_page*'s sole whole class. The method *defWholeClasses* does not appear in the declaration of *Newspaper* because that class is the root of the part hierarchy and therefore does not have any whole classes.

The invocation of *defPartRelshps* in *Newspaper*'s INIT clause notifies it that *Editorial_page* is its only related part class.[14] It is also given the following information: The part relationship is global-exclusive (as denoted by the symbolic constant GLOBAL_EXCLUSIVE); it is single-valued (i.e., the minimum cardinality is 0; the maximum is 1); it lacks any dependency (as specified by NIL); there is no upward inheritance (the upSet is empty); the property *date* is inherited invariantly by

[14] As noted above, we have omitted the other parts of *Newspaper* in our example schema.

Editorial_page from *Newspaper* (the downSet contains 'date' and phi associates it with the dateType identity operator 'dateIdentity').

The class *Editorial_page* is informed of its three related part classes *Masthead*, *Editorial_column*, and *Letters_column* as well as the respective characteristic dimensions by *defPartRelshps* in its INIT clause. It is notified by *defGenDerivedAttrs* of the fact that it has a generalized derived attribute *wordCount* which is computed via summation ('add'). The property *wordCount* at each of the classes *Editorial_column* and *Letters_column* serves as the basis for the computation because it appears in the upSets of the two respective part relationships.

The part relationships established in the class declarations can be queried at run-time via the last seven methods provided by the PartWhole instance type. These methods are invoked against the whole class (where characteristic dimensions are stored) with the part class of the desired part relationship as an argument.[15] E.g., to retrieve the value of the exclusiveness dimension of the part relationship between *Editorial_page* and *Newspaper*, we query *Newspaper* as follows:[16]

```
x := Newspaper -> exclusiveness(Editorial_page);
```

Here, the variable *x* would get assigned the value GLOBAL_EXCLUSIVE, as specified in the declaration of *Newspaper*.

5.2.2. Creating and deleting Parts and Wholes

The method *make* is used to create instances of a PW class. In doing so, it enforces the creation semantics dictated by the part relationship's characteristic dimensions. To create an instance of *Editorial_page*, for example, we write:

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

From its signature, we see that *make* takes as its argument a set of objects that are to be initially installed as parts of the new object. If one of the given objects is from a class other than a prescribed part class, then the creation of the new instance is aborted. In the above invocation, the argument is the empty set, so the new editorial page initially has no parts.

The creation semantics encoded by *make* comprises two kinds of constraints. First, *make* must ensure that the bounds imposed by the multiplicity dimensions of any of the target class's part relationships are satisfied at the outset of the new object's lifetime. (The methods *addPart* and *removePart*, discussed below, carry this same responsibility for the remainder of an object's lifetime.) If *make* detects a violation of any such constraint (for example, there are too few or too many parts of some type), then it aborts the object creation.

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

Instances of a PW class are deleted using *destroy* as follows:

```
Editorial_page -> destroy(p);
```

[15] In VML, the name of a class serves as its run-time identifier.

[16] VML's method invocation is syntactically like that of C++ [56].

The argument p is the OID of the editorial page of interest. Whereas *make* encodes the part relationship's creation semantics, *destroy* encodes its deletion semantics. For example, it may be forbidden to delete some object because another object needs it as an essential part. The complete algorithm describing *destroy*'s operation can be found in [22,26].

5.3. PartWhole instance-instance type

In this section, we describe the details of the PartWhole instance-instance type which endows parts and wholes with correct behavior. Remember that this object type determines the influence of the metaclass on the instances (Fig. 23, brick pattern). Specifically, it gives such instances the ability to establish and break, as well as change and query, part-whole connections with other instances. Furthermore, through its NOMETHOD clause, it provides the means by which part-whole inheritance is accomplished. The public interface for the instance-instance type is as follows.

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

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

5.3.1. Establishing Part-Whole connections between instances

Assume that there exists a part relationship $P_{B,A}$. To establish a part-whole connection between the instance b of B and the instance a of A, we invoke *addPart* as follows.[17]

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

A Boolean value is returned by *addPart* to indicate the success or failure of the part installation. Failures can occur in two scenarios: (1) the argument b is not an instance of an appropriate part class,[18] and (2) b's attachment to a would be a violation of either an exclusiveness or a multiplicity constraint. The details of the algorithm for detecting such violations can be found in [22,26].

5.3.2. Dissolving Part-Whole connections between instances

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

[17] We have adopted, without loss of generality, a protocol that requires the establishment and dissolution of part connections at the whole only. *AddPart* is also responsible for informing the involved part about the transaction.

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

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

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

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

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

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

5.3.3. Querying a Part hierarchy

Once part-whole connections have been established between instances, we can query these using the methods *getParts* and *getWholes*. The method *getParts* returns *all* the parts (i.e., the union of the part sets) of the target instance. For example, to get the parts of an instance *a*, we do the following:

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

If *a* has no parts, then the resultant set ('theParts') is empty. Note that this method only returns the direct parts of an instance, not the parts of the parts. To obtain all the wholes of an instance *a* (i.e., the union of its whole sets), we use the method *getWholes* as in:

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

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

We note that both methods return a set of undifferentiated instances. That is, *getParts* returns all of an instance's parts, irrespective of their classes, and in the same fashion *getWholes* returns all its wholes. It is a straightforward matter to write a VODAK Query Language [37] selection query against the result of *getParts* to obtain the part set with respect to a particular part relationship.

5.3.4. Part-Whole inheritance using the NOMETHOD clause

VML uses a dynamic-binding strategy with respect to method invocations. As such, a method invocation on an object for which the method is not defined is not flagged at compile-time. For example, assume that we have an object *a* which is an instance of the class *A*. Assume also that *A* does not define the method *m*. Consider the following VML statement:

```
a -> m();
```

In most languages, that statement would produce a compile-time error. In VML, it is considered a run-time violation. By itself, the usefulness of this is not so apparent. However, VML provides an error-handling mechanism to deal with errant invocations. The run-time resolution coupled with this error-handling serves as the basis for part-whole inheritance.

The VML error-handler comes in the form of the NOMETHOD clause (NOMETHOD, for short) [37] of a metaclass's instance-instance type. NOMETHOD is simply a VML code segment that is

added to objects whose classes are instances of the particular metaclass. It can be thought of as an additional method, but it cannot be invoked explicitly within a VML program. As its name implies, NOMETHOD is executed automatically any time a spurious method invocation is attempted against an object.

In the context of the PartWhole metaclass, we have designed NOMETHOD such that it will trap an invalid invocation of a reader method for a property and then attempt to resolve it by determining if the requested property is inherited (either upwardly or downwardly). If it is, the prescribed value propagation is carried out. Otherwise, NOMETHOD signals a run-time error. (For a detailed description of our NOMETHOD algorithm, see [26].)

As an illustration of part-whole inheritance, consider the following VML program.

```
(1) aNewspaper := Newspaper->make({});
(2) aNewspaper->setDate([6, 22, 95]);
(3) date1 := aNewspaper->date();
(4) anEditorialPage := Editorial_page->make({});
(5) aNewspaper->addPart(anEditorialPage);
(6) date2 := anEditorialPage->date();
```

In line (1) a new instance of *Newspaper* is created, and in line (2) its date is set to 6/22/95. Line (3) is simply the assignment of that date to the variable 'date1' via an invocation of the reader method 'date.' Line (4) creates a new instance of *Editorial_page*, and in Line (5) it is made part of the newspaper. Finally, in line (6), the variable 'date2' gets assigned the value 6/22/95, which is the date of the newspaper 'aNewspaper' of which 'anEditorialPage' is a part. In other words, 'date1' and 'date2' have been assigned the exact same value!

It should be noted that the method 'date' is not directly defined for *Editorial_page*. However, because the downSet of the part relationship between *Editorial_page* and *Newspaper* contains 'date,' NOMETHOD makes it available (i.e., executable) in the context of editorial pages. This provides the inheritance effect. Its value for any editorial page is identically the value of the method 'date' for the editorial page's newspaper. This provides the value propagation associated with the inheritance.

6. Conclusions

We have presented a comprehensive part model for object-oriented databases. At its foundation is a part relationship which captures the richness of the interaction between real-world parts and wholes. The part relationship is placed in a formal framework by defining four characteristic dimensions: (1) exclusiveness, (2) multiplicity, (3) dependency, and (4) inheritance. The first three of these impose constraints and functionalities that allow applications to exhibit proper part-whole semantics. Dimension (4) provides the basis for the inheritance of properties (either upward or downward) across the part relationship. Such inherited properties contribute to more precise and compact database schemata. The part model also supports the combination of identical properties inherited from multiple sources, thus allowing for the declarative specification of expressions like 'the weight of the car is the sum of the weights of its parts' in the schema.

The implementation of our part model was carried out using VML's powerful metaclass mechanism. Specifically, we built the PartWhole metaclass which serves to integrate our part relationship and its associated derived attributes into the VML data model. As we have shown, the metaclass allows a database designer to exploit all the expressiveness of the part model by declarative means; one simply adds a few lines to an ordinary class declaration. After that, all the proper behavior is enforced implicitly by the database system. Among the features provided by the PartWhole metaclass is the ability for parts and wholes to establish, dissolve, and modify part-whole connections between each other. Query capabilities with respect to part hierarchies are also included.

A version of the PartWhole metaclass is currently implemented as part of the metaclass library provided with VML (which is available as a prototype from GMD-IPSI[19]). It has already been used for the development of a VML class library for a document versioning model at GMD-IPSI.

To facilitate the process of building part-whole database schemata, we have introduced a graphical notation for our part model. Its constructs include symbols for the various guises of the part relationship and for derived attributes. The notation enhances a general OODB graphical schema language called OOdini that we have previously developed and around which we have built an X Windows graphical editor. We are using ObjectMaker (of Mark V Systems) to construct a new graphical schema editor OOdini II which incorporates the part-whole notation.

Acknowledgments

This work was partially supported by: NIST Healthcare Information Infrastructure Technology (HIIT) ATP through HOST (Healthcare Open Systems and Trials); CMS; the HINT (Healthcare Information Networking and Technology) Project at NJIT through a grant of the state of New Jersey; and GMD-IPSI. The first author was supported in part by a Garden State Graduate Fellowship from the state of New Jersey. Thanks go out to Matthew Halper for his helpful discussions and to Fritz Lehmann for his references to the classical literature and enlightening conversations. We would also like to thank the following people from GMD-IPSI: Erich Neuhold for suggesting the integration of the part model with VML; Gisela Fischer for all her help with VML; and Wolfgang Klas, our gracious host, for his collaboration.

References

[1] S. Abiteboul, R. Hull, IFO: A formal semantic database model, *ACM Trans. Database Syst.* 12(4) (1987) 525–565.

[2] H. Aït-Kaci, R. Boyer, P. Lincoln, R. Nasr, Efficient implementation of lattice operations, *ACM Trans. Prog. Lang. Syst.* 11(1) (Jan. 1989) 115–146.

[3] A. Albano, G. Ghelli, R. Orsini, A relationship mechanism for a strongly typed object-oriented database programming language, In *Proc. VLDB '91* (1991) 565–575.

[4] A. Artale, E. Franconi, N. Guarino, L. Pazzi, Part-whole relations in object-centered systems: An overview, *Data and Knowledge Engineering* 20(3) (Nov. 1996) 347–383.

[5] T. Atwood, An object-oriented DBMS for design support applications, In *Proc. IEEE COMPINT 85*, Montreal, Canada, (1985), pp. 299–307.

[19] Integrated Publication & Information Systems Institute of the German National Research Center for Comp. Sci.

[6] J. Banerjee et al., Data model issues for object-oriented applications, *ACM Trans. Office Inf. Syst.* 5(1) (1987) 3–26.
[7] P.J. Barclay, J.B. Kennedy, Regaining the conceptual level in object-oriented data modelling, In *Proc. British Natl. Conf. on Databases*, Wolverhampton, UK, June 1991.
[8] D. Batory, W.Kim, Modeling concepts for VLSI CAD objects, *ACM Trans. Database Syst.* 10(3) (Sept. 1985) 322–346.
[9] J. Bernauer, Analysis of part-whole relation and subsumption in the medical domain, *Data and Knowledge Engineering* 20(3) (Nov. 1996) 405–415.
[10] D. Bryce, R. Hull, SNAP: A graphics-based schema manager, In *Proc. Intl. Conf. on Data Engineering*, 1986.
[11] J.G. Carbonell, Default reasoning and inheritance mechanisms on type hierarchies, In M.L. Brodie, S.N. Zilles (eds.), *Proc. Workshop on Data Abstraction, Databases and Conceptual Modelling* (Pingree Park, CO, June 1980) pp. 107–109.
[12] P.P.-S. Chen, The Entity-Relationship Model: Toward a unified view of data, *ACM Trans. Database Syst.* 1(1) (1976) 9–36.
[13] J.J. Cimino, P.D. Clayton, G. Hripcsak, S.B. Johnson, Knowledge-based approaches to the maintenance of a large controlled medical terminology, JAMIA 1(1) (1994) 35–50.
[14] P. Coad, E. Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, 2nd ed. (Prentice Hall, Englewood Cliffs, NJ, 1991).
[15] C.J. Date, *An Introduction to Database Systems*, Vol. 1, 4th edn. (Addison-Wesley Publishing Co., Reading, MA, 1986).
[16] O. Diaz, 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).
[17] J.B. Fraleigh, *A First Course in Abstract Algebra*, 3rd edn. (Addison-Wesley Publishing Co., Reading, MA, 1982).
[18] P. Gerstl, S. Pribbenow, A conceptual theory of part-whole relations and Its application, *Data and Knowledge Engineering* 20(3) (Nov. 1996) 305–322.
[19] A. Goldberg, D. Robson, *Smalltalk-80: The Language and Its Implementation* (Addison-Wesley Publishing Co., Reading, MA, 1983).
[20] H. Gu, J. Cimino, M. Halper, J. Geller, Y. Perl, Utilizing OODB schema modeling for vocabulary management, In J. Cimino (ed.), *Proc. 1996 AMIA Annual Fall Symposium*, Washington, DC, Oct. 1996, pp. 274–278.
[21] R. Gupta, E. Horowitz (eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD* (Prentice Hall, Englewood Cliffs, NJ, 1991).
[22] M. Halper, A Comprehensive Part Model and Graphical Schema Representation for Object-Oriented Databases, *Ph.D. thesis* (NJIT, Oct. 1993).
[23] M. Halper, J. Geller, Y. Perl, An OODB 'part' relationship model, in Y. Yesha (ed.), *Proc. ISMM 1st Intl. Conf. on Information and Knowledge Management*, Baltimore, MD, Nov. 1992, pp. 602–611.
[24] M. Halper, J. Geller, Y. Perl, 'Part' relations for object-oriented databases, In G. Pernul, A. Tjoa (eds.) *Proc. 11th Conf. on the Entity-Relationship Approach*, Karlsruhe, Germany, Oct. 1992, pp. 406–422.
[25] M. Halper, J. Geller, Y. Perl, Value propagation in OODB part hierarchies, In B. Bhargava, T. Finn, Y. Yesha (eds.), *CIKM-93, Proc. 2nd Intl. Conf. on Information and Knowledge Management*, Washington, DC, Nov. 1993, pp. 606–614.
[26] M. Halper, J. Geller, Y. Perl, Report on the implementation of part relationships using VML metaclasses, Study 224, GMD, Sankt Augustin, Jan. 1994.
[27] M. Halper, J. Geller, Y. Perl, W. Klas, Integrating a part relationship into an open OODB system using metaclasses, In N. Adam, B. Bhargava, Y. Yesha (eds.), *CIKM-94, Proc. 3rd Intl. Conf. on Information and Knowledge Management*, Gaithersburg, MD, 1994, pp. 10–17.
[28] M. Halper, J. Geller, Y. Perl, E.J. Neuhold, A graphical schema representation for object-oriented databases, In R. Cooper (ed.), *Interfaces to Database Systems* (Springer-Verlag, London, 1993) pp. 282–307.
[29] W. Horak, G. Krönert, An object-oriented office document architecture model for processing and interchange of documents, In *Second ACM-SIGOA Conf. on Office Information Systems*, Toronto, Canada, June 1984, pp. 152–160.
[30] M.N. Huhns, L.M. Stephens, Plausible inferencing using extended composition, In *Proc. IJCAI-89*, Detroit, MI, 1989, pp. 1420–1425.
[31] IEEE Computer Society, *Proc. Second Intl. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, Gaithersburg, MD, Oct. 1989.
[32] S.E. Keene, *Object-Oriented Programming in Common Lisp* (Addison-Wesley Publishing Co., Reading, MA, 1989).
[33] W. Kent, *Data and Reality* (North-Holland, Amsterdam, 1978).

[34] W. Kim, E. Bertino, J.F. Garza, Composite objects revisited, In *Proc. 1989 ACM SIGMOD Intl. Conf. on the Management of Data*, Portland, OR (June 1989) pp. 337–347.

[35] W. Kim, F.H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications* (ACM Press, New York, NY, 1989).

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

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

[38] L. Lamport, *LaTeX: A Document Preparation System* (Addison-Wesley Publishing Co., Reading, MA, 1986).

[39] L. Liu, M. Halper, H. Gu, J. Geller, Y. Perl, Modeling a vocabulary in an object-oriented database, In K. Barker, M.T. Özsu (eds.), *CIKM-96, Proc. 5th Intl. Conf. on Information and Knowledge Management*, Rockville, MD, Nov. 1996, pp. 179–188.

[40] B. MacKellar, J. Peckham, Representing design objects in SORAC: A data model with semantic objects, relationships and constraints, In *AI in Design '92*, Pittsburgh, PA, 1992.

[41] D. McLeod, A perspective on object-oriented and semantic database models and systems, R. Gupta, E. Horowitz (eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD* (Prentice-Hall, Englewood Cliffs, NJ, 1991).

[42] R. Motschnig, J. Kaasboll, Part-whole relationship categories and their application in object-oriented analysis, In *Proc. 5th Intl. Conf. on Information System Development (ISD 96)*, Gdansk, Poland, Sept. 1996.

[43] R. Motschnig-Pitrik, The semantics of parts versus aggregates in data/knowledge modelling. In *Proc. CAiSE'93, Lecture Notes in Computer Science*, No. 685 (Springer, 1993) pp. 352–373.

[44] R. Motschnig-Pitrik, Analyzing the notions of attribute, aggregate, part and member in data/knowledge modeling, *Journal of Systems and Software* 33(2) (May 1996) 113–122.

[45] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis, Telos: Representing knowledge about information systems, *TOIS* 8(4) (1990) 325–362.

[46] G.T. Nguyen, D. Rieu, Representing design objects, In J. Gero (ed.), *AI in Design '91* (Butterworth-Heinemann, 1991).

[47] N.F. Noy, C.D. Hafner, The state of the art in ontology design: A survey and comparative review, *AI Magazine* 18(3) (Fall 1997) 53–74.

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

[49] M. Schrefl, E.J. Neuhold, A knowledge-based approach to overcome structural differences in object-oriented database integration, In *Proc. IFIP Working Conf. on the Role of AI in Database and Information Systems*, Guangzhou, China (North Holland, 1988).

[50] D.W. Shipman, The Functional Data Model and the data language DAPLEX, *ACM Trans. Database Syst.* 6(1) (1981) 140–173.

[51] P. Simons, *Parts, A Study in Ontology* (Clarendon Press, Oxford, 1987).

[52] B. Smith, Mereotopology: A theory of parts and boundaries, *Data and Knowledge Engineering* 20(3) (Nov. 1996) 287–303.

[53] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, In *Proc. OOPSLA-86*, 1986, pp. 38–45.

[54] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, A Guttman, Document processing in a relational database, *ACM Trans. Office Inf. Syst.* 1(2) (Apr. 1983) 143–158.

[55] V.C. Storey, Understanding semantic relationships, *VLDB Journal* 2(4) (1993) 455–488.

[56] B. Stroustrup, *The C++ Programming Language*, 2nd edn. (Addison-Wesley Publishing Co., Reading, MA, 1991).

[57] A.C. Varzi, Parts, wholes, and part-whole relations: The prospects of meretopolgoy, *Data and Knowledge Engineering* 20(3) (Nov. 1996) 259–286.

[58] J. Walters, N. Nielsen, *Crafting Knowledge-Based Systems* (J. Wiley & Sons, New York, NY, 1988).

[59] M. White, J. Goldsmith (eds.), *Standards and Review Manual for Certification in Knowledge Engineering* (Systemsware Corp., Rockville, MD, 1990).

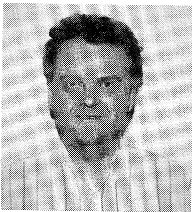
[60] M.E. Winston, R. Chaffin, D.J. Herrmann, A taxonomy of part-whole relations, *Cognitive Science* 11(4) (1987) 417–444.

[61] W.A. Woods, What's in a link: Foundations for semantic networks, In R.J. Brachman, H.J. Levesque (eds.), *Readings in Knowledge Representation* (Morgan Kaufmann Publishers, San Mateo, CA, 1985) pp. 218–241.

[62] S.B. Zdonik, D. Maier, Fundamentals of object-oriented databases, In S.B. Zdonik, D. Maier (eds.) *Readings in Object-Oriented Database Systems* (Morgan Kaufmann Publishers, San Mateo, CA, 1990) pp. 1–32.

Michael Halper received the B.S. Degree (with honors) in Computer Science from New Jersey Institute of Technology (NJIT) in 1985; the M.S. degree in Computer Science from Fairleigh Dickinson University in 1987; and the Ph.D. degree in Computer Science from NJIT in 1993. During his graduate studies, he was the recipient of a Garden State Graduate Fellowship from the State of New Jersey. Dr. Halper is an Assistant Professor of Computer Science at Kean University, and a visiting researcher at the AI & OODB Laboratory of NJIT. His research interests include conceptual and object-oriented data modeling, extensible data models, and object-oriented database systems, as well as computer music. At present, he is involved in a project to model controlled vocabularies using object-oriented database technology. Dr. Halper's research has appeared in a number of international conferences and workshops.

Yehoshua Perl received his Ph.D. degree in Computer Science in 1975 from the Weizmann Institute of Science, Israel. He was appointed Lecturer and Senior Lecturer in Bar-Ilan University, Israel in 1975 and 1979, respectively. He spent a sabbatical at the University of Illinois in 1977–78. From 1982 to 1985, he was visiting Associate Professor at Rutgers University. Since 1985, he has been in the Computer and Information Science Department at the New Jersey Institute of Technology (NJIT) where he was appointed Professor in 1987. Dr. Perl spent short research visits at: University of Capetown, University of Paris VI, University of Toronto, University of British Columbia, University of Rome, and GMD-IPSI. Dr. Perl is the author of more than 80 papers in international journals and conferences. His publications are in the following areas: object-oriented databases, design and analysis of algorithms, data structures, design of networks, sorting networks, graph theory, and data compression. Currently, Dr. Perl is mainly involved in the OOHVR (Object-Oriented Healthcare Vocabulary Repository) project supported by the National Institute for Standards and Technology, Advanced Technology Program. Highlights of his research include among others: the shifting algorithm technique for tree partitioning, analysis of interpolation search, the design of periodic sorting networks, and enhancing the semantics of object-oriented databases. He received the Harlan Perlis Research Award of NJIT in 1996.



James Geller received an Electrical Engineering Diploma from the Technical University Vienna, Austria, in 1979. His M.S. degree (1984) and his Ph.D. degree (1988) in Computer Science were received from the State University of New York at Buffalo. He spent the year before his doctoral defense at the Information Sciences Institute (ISI) of USC in Los Angeles, working with their Intelligent Interfaces group. James Geller received tenure in 1993 and is currently Associated Professor in the Computer and Information Science Department of the New Jersey Institute of Technology, where he is also Director of the AI & OODB Laboratory. Dr. Geller has published numerous journal and conference papers in a number of areas, including knowledge representation, parallel artificial intelligence, and object-oriented databases. His current research interests concentrate on object-oriented modelling of medical vocabularies, and on massively parallel knowledge representation and reasoning. James Geller was elected SIGART Treasure in 1995. His Data Structures and Algorithms class is broadcast on New Jersey cable TV.