New Jersey Institute of Technology

Digital Commons @ NJIT

Dissertations | Electronic Theses and Dissertations

Fall 10-31-1993

A comprehensive part model and graphical schema representation for object-oriented databases

Michael H. Halper

New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation

Halper, Michael H., "A comprehensive part model and graphical schema representation for object-oriented databases" (1993). *Dissertations*. 1176.
https://digitalcommons.njit.edu/dissertations/1176

This Dissertation is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Dissertations by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9409121

A comprehensive part model and graphical schema representation for object-oriented databases

Halper, Michael Howard, Ph.D.

New Jersey Institute of Technology, 1993

Copyright ©1993 by Halper, Michael Howard. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A COMPREHENSIVE PART MODEL AND GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASES

by

Michael H. Halper

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

October 1993

Copyright © 1993 by Michael H. Halper

ALL RIGHTS RESERVED

APPROVAL PAGE

A Comprehensive Part Model and Graphical Schema Representation for Object-Oriented Databases

Michael H. Halper

Dr. Yehoshua Perl, Dissertation Advisor (date)
Professor, Computer and Information Science Department,
New Jersey Institute of Technology

Dr. James Geller, Dissertation Co-Advisor (date)
Assistant Professor, Computer and Information Science Department,
New Jersey Institute of Technology

Dr. Reggie J. Caudill, Committee Member (date)
Professor, Mechanical Engineering Department, and
Executive Director, Center for Manufacturing Systems,
New Jersey Institute of Technology

Dr. Peter A. Ng, Committee Member (date)
Professor and Chairman, Computer and
Information Science Department,
New Jersey Institute of Technology

Dr. Jason T. L. Wang, Committee Member (date)
Assistant Professor, Computer and Information Science Department,
New Jersey Institute of Technology

ABSTRACT

A Comprehensive Part Model and Graphical Schema Representation for Object-Oriented Databases

by

Michael H. Halper

Part-whole modeling plays an important role in the development of database schemata in data-intensive application domains such as manufacturing, design, computer graphics, text document processing, and so on. Object-oriented databases (OODBs) have been targeted for use in such areas. Thus, it is essential that OODBs incorporate a part relationship as one of their modeling primitives. In this dissertation, we present a comprehensive OODB part model which expands the boundaries of OODB part-whole modeling along three fronts. First, it identifies and codifies new semantics for the OODB part relationship. Second, it provides two novel realizations for part relationships and their associated modeling constructs in the context of OODB data models. Third, it provides an extensive graphical notation for the development of OODB schemata.

The heart of the part model is a part relationship that imposes part-whole interaction on the instances of an OODB. The part relationship is divided into four characteristic dimensions: (1) exclusive/shared, (2) cardinality/ordinality, (3) dependency, and (4) value propagation. The latter forms the basis for the definition of derived attributes in a part hierarchy.

To demonstrate the viability of our part model, we present two novel realizations for it in the context of existing OODBs. The first realizes the part relationship as an object class and utilizes only a basic set of OODB constructs. The second realization, an implementation of which is described in this dissertation, uses the unique metaclass mechanism of the VODAK Model Language (VML). This implementation shows that our part model can be incorporated into an existing OODB without having to rewrite a substantial subsystem of the OODB, and it also shows that the VML metaclass facility can indeed support extensions in terms of new semantic relationships.

To facilitate the creation of part-whole schemata, we introduce an extensive graphical notation for the part relationship and its associated constructs. This notation complements our more general OODB graphical schema representation which includes symbols for classes, attributes, methods, and a variety of relationships. OOdini, a graphical schema editor that employs our notation and supports conversion of the graphical schema into textual formats, is also discussed.

BIOGRAPHICAL SKETCH

Author: Michael Howard Halper

Degree: Doctor of Philosophy in Computer Science

Date: October 1993

Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1993
- Master of Science in Computer Science,
  Fairleigh Dickinson University, Teaneck, NJ, 1987
- Bachelor of Science in Computer Science, Summa Cum Laude,
  New Jersey Institute of Technology, Newark, NJ, 1985

Publications:

- M. Halper, J. Geller, and Y. Perl. An OODB "part" relationship model. In Y. Yesha, editor, *Proc. 1st Int'l Conference on Information and Knowledge Management*, pages 602–611, Baltimore, MD, Nov. 1992.
- M. Halper, J. Geller, and Y. Perl. "Part" relations for object-oriented databases. In G. Pernul and A. Tjoa, editors, *Proc. 11th Int'l Conference on the Entity-Relationship Approach*, pages 406–422, Karlsruhe, Germany, Oct. 1992.
- M. Halper, J. Geller, and Y. Perl. Value propagation in OODB part hierarchies. To appear in *Proc. 2nd Int'l Conference on Information and Knowledge Management*, 1993.
- M. Halper, J. Geller, Y. Perl, and E. J. Neuhold. A graphical schema representation for object-oriented databases. In R. Cooper, editor, *Interfaces to Database Systems*, pages 282–307. Springer-Verlag, London, 1993.

Dedicated to

June, Doc, Matt

and

Julie and Ernie

ACKNOWLEDGMENT

First of all, let me extend a warm thanks to my advisors Yehoshua Perl and Jim Geller. They have shown me how to produce a real piece of research and have made this dissertation possible. It is said that a student betrays his teacher if he remains a student forever. Therefore, it is with a profound sense of pleasure that I now call them colleagues and friends. To the two of them, I offer this toast: *L'Chaim.*

Let me next extend my sincere gratitude to Erich J. Neuhold of GMD-IPSI who so graciously served as my outside reader. His intelligence and insightfulness have greatly added to the quality of this work. His enthusiasm is infectious. I'm happy to consider him one of my friends.

Next, let me thank the members of my dissertation committee: Reggie J. Caudill, Peter Ng, and Jason Wang. I single out Peter Ng, chairman of our department, for his unwavering support through all my years of study.

Bill Anderson, chairman of the Math and Computer Science department at FDU, was the one who first encouraged me to pursue this degree. Let me step back now and thank him for his support and encouragement and all he has done on my behalf. I'm glad to count him among my friends, too.

Next, I thank Fritz Lehmann for the enjoyable and, at times, rhapsodic conversations we had at CIKM'92. I'm grateful for his pointers to the classical literature. Leibniz's view aside, Fritz, the integers are one of man's greatest fictions.

Many people at GMD-IPSI are to be thanked as well. They include Peter Fankhauser who delivered a paper for me in Karlsruhe. Gisela Fischer answered all my VML questions with alacrity and precision. Wolfgang Klas hosted my visit to Darmstadt.

Also from GMD-IPSI is Jian Zhao, who I fortuitously met in Glasgow. Since then, Jian has become a close friend, and his companionship and generosity made my stay in Germany a pleasure. By the way, my compliments to the photographer. (The cooking isn't bad, either.) In the port of Amsterdam ...

From the OOdini group, I thank Subrata Chatterjee, Shiv Kuncham, Soniya Shah, Gowthami Nadella, Dipak Shah, and Ram Madapati. Subrata, in particular, brought an overflowing intelligence to the project. His abundant wit was a welcome asset, too. Let me also thank Ken Sayers for his work on the Smalltalk implementation of the part model. See you on the fairways (or in the woods?!?). From the rest of our group, let me say thanks to Ashish Mehta, Aruna Kolla, and Nevil Patel.

I also thank fellow doctoral students Fortune Mhlanga, Tami Zemel, and Rakesh Kushwaha for their moral support throughout the whole process. Guys, I hope to make it out to your necks of the woods one of these days. *Au revoir* and good luck!

Finally, let me thank *mi familia*, to whom this dissertation is dedicated: Morris, June, and Matt Halper, and Julie Halper-Wilson and Ernie Wilson. They all know their contributions, so I'll spare them the verbiage. ("All composed out.")

TABLE OF CONTENTS

Chapter | Page

1 INTRODUCTION 1

1.1 The Part Relationship's Place in the Evolution of Data Models 7

1.2 The Part Relationship in Other Fields 11

1.3 Graphical Schema Representations 18

1.4 Outline of the Dissertation 20

2 A GRAPHICAL SCHEMA REPRESENTATION FOR OODBs 23

2.1 Previous Work 23

2.1.1 A General Approach to Describing OODB Schemata 29

2.2 Classes and Attributes 33

2.3 Generic Relationships 37

2.4 User-defined and Constraint Relationships 40

2.5 Methods 47

2.6 Abridgements 55

2.7 Representing Instances Graphically 57

3 OOdini, AN OODB GRAPHICAL SCHEMA EDITOR 59

3.1 X Windows, the X Toolkit, and OSF/Motif 60

3.2 OOdini's Features and Operation 66

3.3 OOdini Code Conversion 73

Chapter **Page**

4 AN OODB PART RELATIONSHIP 76

4.1 Part Relationships in OODBs 76

4.2 A Generic Part Relationship 81

4.3 The Exclusive/Shared Dimension 88

4.4 The Cardinality/Ordinality Dimension 104

4.5 The Dependency Dimension 118

5 VALUE PROPAGATION AND DERIVED ATTRIBUTES IN OBJECT-ORIENTED DATABASE PART HIERARCHIES 125

5.1 The Value Propagation Dimension 125

5.1.1 Graphical Schema Notation for Value Propagation and Derived Attributes 144

5.1.2 Realization of Value Propagation 150

5.2 Interaction of Part Relationship Dimensions 152

5.3 Generalized Derived Attributes 154

6 IMPLEMENTING THE PART MODEL USING METACLASSES IN VML 167

6.1 The VML Data Model and Metaclasses 168

6.2 The HolonymicMeronymic Instance Type 173

6.2.1 Creating and Querying an HM Class 174

6.2.2 Capturing the Creation Semantics of the Part Relationship using *make* 182

6.2.3 Capturing the Deletion Semantics of the Part Relationship using *destroy* 184

Chapter **Page**

6.3 The HolonymicMeronymic Instance-Instance Type 188

6.3.1 Establishing Part-Whole Connections between Instances . . . 189

6.3.2 Breaking Part-Whole Connections between Instances 192

6.3.3 Querying a Part Hierarchy 194

6.3.4 Performing Value Propagation using NOMETHOD 195

7 CONCLUSIONS AND FUTURE WORK 199

A VML CODE FOR THE HolonymicMeronymic METACLASS 211

B A SAMPLE VML PART SCHEMA 255

REFERENCES 259

LIST OF TABLES

Table **Page**

2.1 The graphical schema constructs 55

5.1 Interaction of part relationship dimensions 154

5.2 The part relationship symbols by dimension . . . . facing 164

LIST OF FIGURES

Figure		Page
2.1	The class `customer` and its attributes	34
2.2	The class `customers`	34
2.3	The attribute *age* with default value 20	34
2.4	A tuple class	36
2.5	A specialization hierarchy.	38
2.6	The `section-student` example	39
2.7	The ER model of Figure 2.6	41
2.8	Alternative form of Figure 2.6	43
2.9	An essential relationship	45
2.10	A range-restricted relationship	45
2.11	The `section-student` example with methods	51
2.12	Some example path methods	53
2.13	An excerpt from a university database schema	facing 55
2.14	An instance of `person` which owns an instance of `car`	57
3.1	The X Windows software architecture	61
3.2	Communication between a C client and X Server using the Xlib	62
3.3	Communication between a C client and its Xt widgets	65
3.4	OOdini's main screen	67
3.5	OOdini in a Level 2 display	facing 73

Figure		Page

3.6 OOdini in a Level 3 display . . . . facing 74

3.7 Conversion from OOdini into an OODB language 74

3.8 Alternative conversion paths from OODAL into OODB language 75

4.1 A part relationship between meronymic class B and holonymic class A . 84

4.2 The generic realization of the part relationship 84

4.3 A muffler owned exclusively by a boat 89

4.4 A document database schema 90

4.5 Class B in n part relationships 90

4.6 Part relationships in a music publication database 92

4.7 An ensemble score and its score expression sequence 93

4.8 Global exclusive part relationships 96

4.9 Revised document schema with class exclusiveness and sharing 97

4.10 Students limited to at most six sections 97

4.11 Realization of exclusiveness 98

4.12 Realization of sharing 100

4.13 Realization of limited sharing 102

4.14 Newsletters in a document database 106

4.15 Inadequate model for the minutes of a meeting 107

4.16 An adequate model for the minutes 108

4.17 Newsletter schema 109

Figure		Page

4.18 A single-valued part relationship 109

4.19 A multi-valued part relationship 110

4.20 A fixed-cardinality part relationship 110

4.21 Article with an essential bibliography 111

4.22 Article with at least one constituent section 111

4.23 Minutes with an ordered list of parts 113

4.24 Books with an indefinite number of chapters 114

4.25 Books with text segments as chapters 115

4.26 Realization of range-restriction 115

4.27 Realization of ordering 116

4.28 Dependency of meronyms on holonym 119

4.29 Table of contents dependent on its book 121

4.30 Bicycle dependent on its part frame 121

4.31 Realization of part-to-whole dependency 122

4.32 A part schema with possible deletion conflicts 123

5.1 Age propagated from airframe to plane 126

5.2 The attribute *font* being propagated from `book` to `chapter` 145

5.3 The attribute *age* propagating upward from `airframe` to `plane` 146

5.4 Attribute *color* propagating upward from `rim` to `piano` 146

5.5 The relationship *attachedTo* propagated from hinge to door 147

Figure		Page

5.6 The class **amplifier** getting *reliability* from **transistor** 147

5.7 Students obtaining their enrollment credits from their sections 148

5.8 Car body getting its color through an upward propagation 148

5.9 The relationship *taughtBy* propagated downward cumulatively 149

5.10 The *dateOfPubl* propagated from the first volume to the complete works 149

5.11 Realization of upward value propagation 151

5.12 Redundant value propagation 155

5.13 Plane getting its colors from its fuselage, wings, nose, and tail 161

5.14 Fuselage getting its own color propagated from its constituent sections . 162

5.15 The weight of a boat as the sum of the weights of its parts 163

5.16 *The New York Times* editorial page 166

6.1 The instance type's effect on a class's instances 169

6.2 The interaction between metaclasses, classes, and instances 170

6.3 The class **car** without part relationships 175

6.4 Car and its parts engine and body 177

CHAPTER 1

INTRODUCTION

The problem of defining a database schema is that of expressing a real-world system or enterprise in a stylized description language. This language, commonly referred to as a data definition language (DDL), ideally comprises an extensive set of core epistemological constructs such as categorization, attribution, association (i.e., relationships between categories or classes), taxonomy (i.e., specialization and generalization), and aggregation. (Cf. [164]; also [103] and [25].) Categorization and taxonomy are ordinarily grouped under a single heading and referred to as classification. A more predominant catch-all phrase for the two is "IS-A" [22]. The last member, variously referred to as part decomposition, mereology [195], or meronymy [210], is typically denoted by the English phrases "is part of" or, conversely, "has part."

This dissertation focuses on the problem of aggregation or part-whole modeling in the context of object-oriented database (OODB) systems. In particular, we present an OODB "part" model comprising a number of different aspects. At the heart of the part model is a part relationship [107] which is used to relate a pair of object classes, making one a "whole" class and the other a "part" class with respect to each other. Relating two classes in this manner has the effect, at the instance level, of making a pair of respective objects of the classes whole and part in relation to each other.

The part model we present in this dissertation expands the boundaries of OODB part-whole modeling along three fronts:

1. It identifies and codifies new semantics for the OODB part relationship.
2. It offers two novel realizations for part relationships and their associated modeling constructs in the context of OODBs.
3. It provides an extensive graphical notation for the development of OODB schemata.

Regarding the first item, an OODB part relationship can be seen as comprising a variety of semantics, which, in our terminology, we separate into different *characteristic dimensions*. ORION, the only OODB system among the commercially-available OODBs[1] that offers such a modeling construct, divides its part relationship along two lines, exclusiveness and dependency, both of which are refined in our treatment. Because, we have found ORION's notion of exclusiveness too restrictive under certain circumstances, we have refined it into two kinds, *global exclusiveness* and *class exclusiveness*. On the other hand, their complementary notion of sharing can be too unrestricted for the development of "logical part hierarchies," which is one of the issues that the ORION part model addresses [107]. For this reason, we have found it necessary to offer a more restricted form of sharing that we refer to as *limited sharing*.

[1]The OODB systems we are referring to here, the so-called Big Seven, are ONTOS [149, 191], ObjectStore [112], ORION, O_2 [47, 48], GemStone [24], Versant [74], and OpenODB (formerly IRIS) [57].

Our notion of dependency supersedes that of ORION by accounting for the possibility of "ontological dependency" [187], where the existence of a whole is dependent on the existence of some defining part. In fact, in our model. dependency can be specified in either direction across the part relationship, from the part to whole or the whole to part. ORION allows parts to be dependent on their wholes, but not vice versa.

Value propagation (variously referred to as attribute propagation [145] or local predication [187]) is the means by which property values at one level of a part hierarchy are assimilated by objects at another level. The mechanism forms the basis for the definition of derived attributes [94] which play an important role in the development of OODB part schemata. Our part model extends the notion of value propagation across a part relationship, as first introduced in SHOOD [145] and subsequently in SORAC [127], by formally identifying three types of propagation that we have found to be prevalent in part hierarchies: invariant. transformational, and cumulative. The invariant propagation serves to formalize SHOOD's attribute propagation, which, as it happens, is limited to the upward direction, from part to whole. All of our types of propagation, in contrast, may occur in either of the two directions across the part relationship. SORAC [127] seems to lay the foundation for a more general kind of propagation with its *derivation* relationship. However, like SHOOD, it informally introduces the notion, and it fails to distinguish properly

among the kinds of value propagation that occur in part hierarchies and the kinds of derived attributes that can be induced by these processes.

Any OODB part model that is worth its salt must be able to express facts such as "the weight of a car is the sum of the weights of all its parts, regardless of their classes." As a further extension to OODB part-whole modeling, we introduce, in our model, the formal notion of a generalized derived attribute which can encompass value propagations from any number of part relationships simultaneously. Not only is such a construct a powerful means of expression, but, as we will discuss, it serves as a natural solution to the "multiple value propagation" problem in the context of part hierarchies (which is analogous to the "multiple inheritance" problem [190] of ordinary OODB IS-A hierarchies).

We offer two realizations for our part model which serve to demonstrate its viability. The first of these, following in the tradition of the Entity-Relationship (ER) Model [36] and other semantic data models [94, 155], realizes the part relationship as an object class in its own right (or, as some would say, as a "fat link" [55]). The realization is noteworthy for its strict adherence to basic OODB modeling primitives. Due to this fact, our model may be incorporated into any number of different OODB settings, and it is hoped that developers of other OODBs will make use of it to quickly and easily add part modeling capabilities to their own systems. While the technique of modeling a "second order" construct as a "first order" construct has been suggested by previous work in the object-oriented community (e.g.,

[8, 49, 108, 127, 170]), this is the first time that it has been applied exclusively to a part model.

In [108], Wolfgang Klas proposed an innovative metaclass mechanism as a means for incorporating new semantic relationships (such as the part relationship) into an underlying OODB data model. He was motivated by the view that no one can predict all the future needs of database designers and that therefore an OODB should employ an open architecture and extensible data model. Recently, his mechanism has become available in the VODAK Model Language (VML), an OODB being developed at GMD-IPSI.[2]

The second realization of our part model is an entirely novel approach using the VML metaclass facility. This realization, which we have actually implemented, has as its foundation a custom VML metaclass called the "HolonymicMeronymic" metaclass that captures the semantics of classes participating in part relationships. (The complete specification for the HolonymicMeronymic metaclass can be found in Appendix A.) From a research standpoint, this implementation addresses two important questions:

1. Can our part model be incorporated into an existing OODB without having to rewrite a meaningful subsystem of the OODB?
2. Can the VML OODB with its open architecture and metaclass facility support the introduction of such an extension?

[2]The Integrated Publication and Information Systems Institute of the *Gesellschaft für Mathematik und Datenverarbeitung*, the German government's computer science research institute.

As demonstrated by the work reported herein, both of these questions can be answered in the affirmative.

An important tool in the development of complex OODB schemata is the graphical schema representation. (We discuss this issue further below.) This is especially true when we introduce a part model and all its additional complexity into the OODB data model. For this reason, we present an extensive schema notation for the part relationship which goes beyond some earlier notational conventions [16, 39, 171]. In fact, we use an enhancement of OMT's [171] aggregation (i.e., part relationship) symbol as the basis for a rich set of symbols that denote the part relationship in all its various guises. Separate graphical symbols are also provided for the different kinds of value propagation as well as for their associated derived attributes. Together, these symbols can graphically define both a derived attribute and its implementation as the propagation of a data value across a single or multiple part relationships.

To provide a framework within which to carry out the development of OODB part schemata, we also introduce a general graphical notation for the representation of OODB schemata. This graphical language comprises a broad range of OODB constructs including classes, attributes, methods, and a variety of different relationship types, thus making it applicable to a wide group of OODB systems. To complement the graphical language, we have built the OOdini graphical schema editor. Not only does OOdini allow for the creation and browsing of OODB schemata described in our notation, it also provides for the conversion of the graphical schema diagram into

a number of textual representations. Included among these is VML source code. As such, OOdini can serve as an effective graphical interface for an OODB.

As we alluded to above, the part relationship at the foundation of our part model is defined in terms of a number of *characteristic dimensions* which specify constraints and functionalities that impose real-world, part-whole semantics on the instances of the participating classes. Among these dimensions are: (1) exclusiveness/sharing, which controls the way parts may be distributed among wholes; (2) cardinality/ordinality, which addresses how parts are combined to form wholes; (3) dependency, which describes the deletion semantics of parts and wholes; and (4) value propagation, which allows for the assimilation of data values by wholes or parts and provides the basis for the definition of derived attributes.

Overall, the work presented in this dissertation can be viewed as the marriage of an improved part model with an existing general OODB data model. We see this marriage as another step in the ongoing effort to make database DDLs more epistemologically complete, or, if you prefer, more "semantic" [41, 46]. For this reason, before getting to some of the background material on the part relationship itself, let us briefly consider the evolution of DDLs and the part relationship's place in it.

1.1 The Part Relationship's Place in the Evolution of Data Models

The ubiquitous relational model [12, 46, 40, 198, 205] has gained tremendous popularity, much of which has resulted from its simplicity. This simplicity derives from

the fact that it incorporates only two of the above mentioned epistemological primitives: categorization in the form of relations (or tables), and attribution as expressed in the fields of these relations. Association (between categories or relations) is not really a core construct, but rather a derivation based on relations and the extrinsic notion of foreign key [89].

The Entity-Relationship (ER) model [36], which was originally introduced as a diagraming methodology for relational schemata, goes a step beyond the relational model by making association fundamental in the design of schemata. In the development of an ER schema, an application domain is mapped onto a group of entity sets along with a set of named relationships (associations) between these. The approach's intuitive appeal has made it very popular and successful. Of course, its success comes in no small part because of its graphical notation, which has become a *de facto* standard in the field of data modeling.

Classification or taxonomic analysis, the ability to recognize and exploit similarities and differences between categories or classes of objects, is one of the most important facets of human knowledge and reasoning [166, 188]. In AI, taxonomy and its accompanying IS-A relationship have spawned a whole body of literature, including research into precise semantics [22, 204] and efficient implementations of "class" reasoners [54, 63, 181]. Early DDLs, such as those just mentioned, provided no facilities for taxonomic descriptions of the application domain. Employing classification in the description of database schemata has the following advantages:

1. It is conceptually natural and promotes better modeling.

2. It provides a more precise view of the application domain.

3. It promotes the reuse of software and specifications, which yields more compact schemata.

While work was done to overlay classification on the relational model (e.g., [41, 189]) and the ER model (e.g., [177]), it was not until the emergence of the semantic data models [94, 132, 155], such as TAXIS [141], SDM [88], IFO [4], and so on, that IS-A became a staple of DDLs. Since then, it can be found in most data models, including the extended relational [122, 168], enhanced ER [50, 51, 52], and, of course, the entire family of OODB models. In fact, the construction of the IS-A hierarchy has been called by H. J. Kim "the main theme of schema design for object-oriented databases" [105].

Aggregation, which can loosely be described as the building up of higher-level objects from lower-level components, is an epistemological construct which is as fundamental as classification and can offer many of the same benefits. The term *aggregation* has been used in two distinct senses within the field of data modeling [51]: (1) to describe the process of combining attributes to form entities (e.g., an employee is said to be an aggregate of *name*, *age*, *telephone-number*, and *address*); and (2) to describe the construction of higher-level entities from lower-level ones. In the former sense, the attributes constitute parts of the *computer* representation (or structure) of the integral object, and not the integral object itself. That is, attributes

are just containers for data values and are not objects in themselves. In the second sense, we usually talk about aggregation as constituting a part relationship between objects of different kinds.

Aggregation in the second sense occupies a niche in data modeling in many advanced application domains. For instance, one of the main activities of manufacturing [95] is combining parts to produce whole products. In Computer Aided Design (CAD) [14, 18, 79, 98, 99, 125], substructures are pieced together to form complete designs. In architectural CAD [125], windows, walls, and doors form rooms, rooms and corridors form floors, floors form buildings, and so on. Similar structuring activities occur in other areas such as graphics [61, 71], multimedia processing [37, 211, 212], text document processing [91, 92, 197], and Computer Aided Software Engineering (CASE) [39, 121].

Many of the areas just mentioned have been heralded as ideal proving grounds for the new generation of OODB systems. However, if OODB systems are to fulfill their expectations, it is imperative that they provide support for aggregation in sense (2). This means formally defining a notion of part relationship along with all its appropriate semantics and functionalities. The work in this dissertation does just that and provides an extensive framework within which to carry out part-whole modeling in an OODB.

1.2 The Part Relationship in Other Fields

In this section, we investigate how aggregation and the part relationship have been employed in other fields of research. As we have said, aggregation is a fundamental notion, and we will see that it has had far-ranging impact on many disparate areas.

The part-whole organizational method is such an intuitive notion that attempts have even been made, mainly in the 20th century, to place a formal notion of part relationship at the foundation of many fundamental mathematical disciplines. The Polish mathematician Leśniewski, the founder of so-called *classical extensional mereology* (CEM) [187, 195], intended his Mereology, along with its underlying theories of Protothetic and Ontology, as nothing less than a new foundation for all of mathematics. Leśniewski was driven in no small part by his distrust of sets and set theory, which was fueled by Russell's famous paradox [137]. He believed that the notion of a part relationship was more fundamental and far more intuitive and would provide a sounder basis for modern mathematics.

Tarski, a student of Leśniewski's, used his own mereology as the basis for an account of solid geometry [202]. Earlier, Whitehead had planned a similar program which was to become the fourth volume of the *Principia Mathematica*, though this was never carried through. It was later discovered that his mereology contained a number of critical flaws. Leonard's and Goodman's Calculus of Individuals [119], the predicate logic version of CEM, was meant as a replacement for set theory [75].

Mereology also makes an appearance in mathematical topology, as in the theories of Tiles [203] and Clarke [38].

Lehmann [115] provides extensive references to the classical literature in the field of mereology. Sowa [195] discusses the history of some of its terminology and concepts. Simons [187] presents a comprehensive survey of mereology while unifying many of its disparate theories and proposing further extensions. While his work is couched in very rigorous mathematics, Simons is primarily concerned with dealing with some of the chief philosophical problems surrounding parts and wholes.

One of the classic part problems in philosophy is that of the ship of Theseus [78, 187]. Legend has it that Theseus, king of Athens, sent a sailing vessel to explore the uncharted seas. At the outset of its voyage, the ship encountered heavy weather which resulted in damage to some of the planks of its hull. These were promptly replaced and discarded on a deserted island. Subsequently, another violent storm caused the ship's mast to snap. The broken mast was also replaced and left as refuse on some deserted island. This continued and, as it turned out, during the entire course of its voyage, every part of the ship was damaged, replaced, and discarded. Now, as it happened, a man in another ship was following Theseus's throughout its journey and was collecting all the abandoned parts. Determining that the parts were in relatively good shape, he repaired them and reassembled them into a ship. The question then arises: Which one of the two ships is the ship of Theseus?

Another form of this problem involves the proverbial axe that George Washington used to chop down the cherry tree. In due time, the axe comes up for auction and is bought by a practical man who does not believe in just mothballing items which can be put to good use. Of course, over time, the head of the axe wears out and has to be replaced. Eventually, the handles cracks and is replaced, too. After all that, is the man still the proud owner of Washington's axe?

As for an answer to these problems, an appeal to a continuity of material would say that the ship reassembled from the original parts is Theseus's. On the other hand, an appeal to the continuity of function would insist that the one containing his crew and carrying out his mission is Theseus's ship.

Interestingly, in the context of an OODB, where we are dealing only with surrogates of reality, there is a third argument, which is the continuity of object identification [5]. In an OODB, we instantiate separate objects for every part as well as for the whole. Thus, it is quite reasonable in such a system to posit the existence of the whole (i.e., instantiate an object for it) without the existence of its parts. Of course, this is somewhat counter-intuitive because in a real-world physical system, the existence of a whole is predicated on the existence of its parts; destroy or discard all the parts, and the whole is reduced to a cipher. This is a form of the notion of the "ontological dependence" [187] of the whole on its parts. As we will see, our part model does support the enforcement of such dependence, if desired. Anyhow, if we decide to create an object representing Theseus's ship, and this object is given

an OID x, then, for us, Theseus's ship will always be the one with identification x. Changing the parts is not a problem due to the immutability of the ship's OID. Even though its composition in terms of other system objects is in flux, the identity of the ship remains the same, following the functional point of view.

A classical statement of the conflicting interpretations of integral objects can be found in a passage of Plato's *Theaetetus* (203E) which is used by Simons as the epigraph of his book [187]:

> Perhaps we ought to have maintained that a syllable is not the letters, but rather a single entity framed out of them, distinct from the letters, and having its own peculiar form.

The philosophical and logical problems of parts and wholes carry over naturally into the area of linguistics and cognitive psychology, where a pervading issue revolves around the part relationship's transitivity [44, 96, 163, 210]: Is it always transitive, and if not, when isn't it? This, of course, has direct bearing on the problem of when syllogisms of the form "A is part of B, and B is part of C, hence A is part of C" constitute valid inferences. For example, consider the strangeness of the following arguments [210].

(1a) Smith's arm is part of Smith.
(1b) Smith is part of the philosophy department.
(1c) Smith's arm is part of the philosophy department.

Or:

(2a) The refrigerator is part of the kitchen.
(2b) The kitchen is part of the house.
(2c) The refrigerator is part of the house.

In their seminal work, Winston, Chaffin, and Herrmann [210] draw an elaborate distinction between different types of part-whole semantic relationships to answer the question of transitivity in the negative. In particular, they distinguish between six types of part relationships which are described as follows: (1) component/integral, (2) member/collection, (3) portion/mass, (4) stuff/object, (5) feature/activity, and (6) place/area. Suggestive examples of each of these are: (1) A wheel is part of a car; (2) a battleship is part of a fleet; (3) a slice is part of a pie; (4) a bicycle is partly composed of steel; (5) paying is part of shopping; and (6) the Everglades is part of Florida.

Their discrimination between the different part relationships is based on the applicability of certain sentence frames and on the so-called common argument criterion which may be stated as: If it is possible to ask two different questions about one and the same object (the common argument) and receive two different answers that both contain the word "part" (or some variation or synonym of it), then the two answers must correspond to two different part relationships. As an example, consider the following two questions about a bicycle: "What are its parts?" and "What is it made of?" It is easy to devise answers to these using "part": "The wheels (among other things) are parts of the bicycle," and "the bicycle is partly steel." Thus, there

must be two distinct part relationships at play here, the component/integral type and the stuff/object type.

After all their analyses, they reach the conclusion that the part relationship is transitive as long as one does not mix its different types in a single argument. Mixing the different types (as we have done in the sample arguments above) is a form of equivocation, and, in such cases, the entire argument breaks down. Similar conclusions are found in [96], where an alternative analysis yields another set of four different part relationships. A dissenting voice is heard in [187] where Simons argues that the part relationship is always transitive. Any denial of this, he says, is a misconception based on notions external to parts. In Artificial Intelligence (AI), the transitivity of part-whole has also been taken for granted by some researchers [54, 154, 162].

Other research on the part relationship has concentrated on its fundamental nature. Concerns include whether or not the part relationship is semantically primitive, and whether or not it can aid in the formation of a valid model of human memory [130]. Several studies by Chaffin and Herrmann [31, 32] have focused on the human capacity to distinguish the part-whole relationship from other semantic relations.

In AI, attention has also been given to the fundamental nature of the part relationship, especially as it applies to reasoning [30, 77, 93, 130, 139, 178, 184]. In the context of semantic networks, the part relationship is used in the analysis of granularity [131]. In [161], Clarke's mereo-topology is extended to capture notions

of space and time. A comprehensive survey of related work in semantic nets appears in [115]. Drawing on the work of [210], Huhns and Stephens [93] propose an algebra for the composition of various semantic relationships including the part relationship. As in [210], the analysis is based on the decomposition of relationships into basic relational elements. Their work is being incorporated into the Cyc knowledge base system [117, 118]. The representation of part-whole configurations in artificial neural nets has also been investigated [90].

Geller [60], following the work of [210], also distinguishes between different kinds of part relationships as an aspect of his general theory of natural language graphics (NLG) [59, 61]. The various part relationships are referred to descriptively as: (1) additive, (2) constituting, (3) replacing, and (4) invoking. All are used for graphical inference and to help supply information needed in the construction of useful diagrams. Specifically, they help to determine what content to place on the display screen of an NLG system and how to organize that content to be optimally useful to a viewer. The general problem of organizing such user presentations in knowledge-based systems has been referred to as *presentation planning* [9].

Our part model draws freely on and has been influenced by much of the work mentioned in this section. Our formal description of the part relationship as a multidimensional mathematical structure encompassing various characteristic dimensions is reminiscent of the description of the part relationship in terms of basic relational elements in [210] and [93]. We also see our part relationship as serving in the role of

an epistemo-linguistic construct in an OODB environment. In other words, we see it as a general modeling vehicle and not one which is limited to the narrow sense of "part" as it is often used in the description of physical assemblies (e.g., cars and boats). As such, it can be used to model the fact that a student is part of the section of a course he is taking, or a battleship is part of a naval fleet.

1.3 Graphical Schema Representations

As we have mentioned above, the graphical representation of the database schema has become a standard tool in database design and user orientation. In this section, we consider this issue further as it relates to OODBs.

When designing an OODB schema, one is faced with a number of challenges. Among these is the need to create and organize and thus comprehend a large number of object classes. The designer must insure that each class contains the attributes and methods necessary to describe its objects. The classes must also be connected with the appropriate relationships, which convey semantic information and allow for the retrieval of relevant, remote data. To accomplish these tasks, the designer needs a solid grasp of the overall structure of the database.

For the user of an OODB, the requirement of understanding the overall structure may be felt even more intensely. Having not built the database's conceptual model, the user must be provided with a means for becoming acquainted. One finds a similar situation in hypertext systems, where the disoriented user is said to be "lost in space" [42]. Natural language descriptions or presentations of the written specification of

the OODB schema are often cumbersome and tend not to serve as apt guides on the road to comprehension.

Consider, also, the need to devise *path methods* [134, 144, 179, 180] (sometimes called *path expressions* [104, 141]), which are used to retrieve information relevant to a given class from other distant, related classes. Usually, a path method is composed of a sequence of user-defined or generic (system-defined) relationships, and sometimes it ends with an attribute. The non-trivial task of creating such a sequence can be further complicated if the designer or user is only vaguely familiar with the "surrounding landscape." In order to construct such methods effectively, a general view of the OODB schema is extremely helpful. A considerable research effort has gone into the problem of providing software support for the construction of such methods. The results of some of this research can be found in [133, 134, 135, 136]. In particular, in [133, 136], a path method generation (PMG) system is described which works interactively with a database user or designer and assists in the formulation of path methods to retrieve desired, distant information based on given source and target data. The system is particularly effective when the user has incomplete or even incorrect knowledge of the database schema. Currently, the system runs in the VML database environment, where it uses a variety of graph traversal algorithms to scan through a VML schema. The graphical schema notation described here was used extensively in that research [133].

The problem of comprehending the structure of a database is not unique to OODBs, but due to the rich modeling capabilities of such systems, this task becomes even more difficult in that context. A medium-sized OODB typically comprises hundreds to thousands of classes. The university database application [19, 34, 120, 209] which our research group has built in VML as a test-bed for ongoing research comprises a couple of hundred classes and far more relationships and other schema connections. Another group project, an electrical circuit design package [206], uses about a hundred classes. Remembering the names of a few dozen of these, let alone a few hundred, as well as the interconnections between them, is almost impossible.

With all this in mind, we have designed a graphical language for the representation of OODB conceptual schemata, a language useful to both schema designers and users. The language includes symbols for classes, attributes, methods, user-defined relationships, constraining relationships, and generic relationships—a wide enough variety to satisfy a diverse group of OODB data models. This general schema language serves as a basis for the development of the part-whole graphical schema notation for the part relationship and derived attributes which form an important point of our part model.

1.4 Outline of the Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we describe our general graphical schema representation for OODBs. First, we review some of

the previous work on graphical schema representations. After that, we go on to discuss the details of the various constructs of the language.

In Chapter 3, we present OOdini, a graphical schema editor that we have built to support and promote the use of the graphical language introduced in Chapter 2. OOdini allows a user to create and manipulate OODB schemata described in that graphical notation. Using its various levels of display, it can also serve as an OODB schema orientation or browsing tool. The system comprises about 30,000 lines of C Language code and runs in the X Windows and OSF/Motif environment. At first, we will cover the details of that windowing environment. After that, we describe the operation and features of OOdini. Also supported by OOdini, making it an effective OODB graphical interface, is the conversion of its pictorial representation into the syntax of the VML OODB and other abstract textual languages [35]. We discuss the details of OOdini's code conversion at the end of the chapter.

The details of our part model are presented in Chapters 4 and 5. In Chapter 4, after a survey of the relevant background literature, we go on to present the mathematical aspects of the part relationship and all its various semantics and functionalities. Included in the formal treatment is a presentation of the graphical schema notation for the part relationship. The various symbols presented there serve to enhance the language of Chapter 2. We also present the realization of the part model using the ordinary constructs of an OODB data model. In Chapter 4, the first three characteristics dimensions of the part relationship are considered: the discussion of

the fourth dimension, that of value propagation, is deferred to Chapter 5. There, we introduce the value propagation mechanism which serves as a powerful basis for the definition of derived attributes in the context of part hierarchies. Once again, to complement the formal aspects of value propagation and derived attributes, we present a graphical schema notation to be used with the general schema language.

In Chapter 6, we describe the implementation of our part model using the metaclass facility of VML. At first, we discuss the theory behind the metaclass mechanism. Afterward, we give the details of our own metaclass, the HolonymicMeronymic metaclass, which captures the various semantics of classes participating in part relationships and part hierarchies.

Finally, in Chapter 7, we conclude with a summary and a discussion of future research directions. Preliminary and shorter versions of the work presented in this dissertation may be found in [83, 84, 85, 86, 87].

CHAPTER 2

A GRAPHICAL SCHEMA REPRESENTATION FOR OODBs

In this chapter, we introduce our graphical schema notation for the representation of OODB schemata. The notation is designed to serve as an intuitive data definition language [46] for OODBs. At first, we survey previous work that has been done on graphical schema representations, not only in the database field but in other related areas as well. After that, we discuss the details of the underlying OODB data model. Then, in subsequent sections, we present the entire schema notation which includes a wide range of symbols including those for classes, attributes, a variety of relationships, and methods. At the end of the chapter, we extend this by a generic notation for objects which can be used when hybrid class/instance diagrams are called for.

2.1 Previous Work

The usefulness of the graphical representation of knowledge-base schemata has long been acknowledged. Early on, the knowledge representation community recognized the importance of graphical aides. Semantic Nets [23, 116, 195] are invariably presented in a graphical form. The idea of presenting knowledge graphically is not a particularly recent development. Lehmann [115] shows what he claims to be a precursor of today's semantic nets: A coat-of-arms from the middle ages with its inherent religious symbolism. In the 19th century, the English lawyer Alfred Bray

Kempe developed a relationship diagraming system for the expression of conceptual knowledge ([101, 102] as cited in [115]). The American philosopher Charles Pierce was so convinced of the power of graphical notations that he developed an alternative formalism to predicate logic which he called *existential graphs* [158, 165, 195]. The Conceptual Graphs of Sowa [192, 193, 194], which were influenced by the work of Pierce, were designed as a graphical formalism, though a (less effective) textual form is provided for use with older generation technology. The representational theory Conceptual Dependency [124, 164, 173, 174] also employs a graphical formalism. Even frames have been given a pictorial form [164].

In the database community, there are a number of data models which present schemata in diagrammatic fashion. Perhaps none of these is more prevalent than the Entity-Relationship (ER) model [36, 51, 205]. In fact, this graphical language is often used as a diagraming device for other data models such as the relational (e.g., Schemadesign [29]). Due to its popularity, some of the ER notation has made its way into our representation.

Another semantic data model with a graphical schema representation is Galileo [6]. Actually, Galileo is a full database programming language with support for ER-like relationships [8]. The schema editor Sidereus [7] has been built as a tool for the construction and realization of Galileo schemata.

The schemata of the formal semantic model IFO [4] are depicted in graphical diagrams. SNAP [26], developed by the originators of IFO, is a graphical schema

editor for IFO which allows for the creation and manipulation of schemata as well as for the querying of any database instantiations. The Functional Model of Shipman [186], which influenced IFO, also employs a pictorial representation of its schemata.

The GOOD [72, 81, 82] database model, also related to the Functional Model, is a model based entirely on graphs. As such, it employs a graphical notation for everything from the schema description through queries against instances of the schema. While the designers of the model make claims to its object-orientation and demonstrate a correspondence between some of the object-oriented concepts promulgated in [13] and their own constructs, most of these constructs are derived or simulated and are not inherent parts of the underlying model. Due to this, their model does not lend itself to the direct representation of all our constructs discussed in Section 2.1.1 below. For example, the notion of a node label in GOOD schemata can be associated with the notion of class in OODB schemata. However, node labels permit multiple and separate descriptions which corresponds to multiple definitions of a single class—an unintuitive situation.

Within the OODB community, some system designers have considered the graphical representation of the class hierarchy. Among these systems are Ode, Iris [57], O_2, and Ontos [149]. Unfortunately, the class hierarchy relates only a limited part of the interrelations between classes. Kim [106] presents a notation he calls a *schema graph* which captures the normal class hierarchy as well as the class-composition

hierarchy. The Object-Oriented Entity-Relationship Model [76], an object-oriented extension of the ER model, uses a diagram derived from the ER model.

The GemStone OODB [24, 27], based on a Smalltalk model originally proposed in [43], now has an accompanying graphical development environment called Geode. Included in this package is a schema design tool which allows for the interactive development of GemStone schemata. However, due to the designers' view [182] that a schema design tool which supports semantic constructs (e.g., the constraint relationships discussed below) is more of a computer-aided software engineering (CASE) tool than a database utility, the system is primarily a visualization mechanism for the structures (e.g., classes) of the underlying Smalltalk object model. Their approach is guided, in part, by their Smalltalk model, where the additional "semantics" of semantic relationships must be hand-coded into methods by the schema designer. Our schema representation, on the other hand, accommodates a number of semantic constructs. We also graphically represent path methods and different generic relationships.

In the area of object-oriented modeling and design, there exists a graphical notation which complements the Object Modeling Technique (OMT) [171]. This notation is geared more toward the general description of software systems built using object-oriented analysis and design rather than the description of OODB schemata per se. In this light, it can be seen as a CASE notation. A software editor, called OMTool [171], is now available to create OMT diagrams. Currently, the system only

allows for the realization of an OMT diagram as a C++ application. Support for the generation of applications which are coupled with bona fide OODBs is under development.

Aside from OMT, there has been a flurry of activity regarding graphical formalisms in the CASE community. This has lead to a plethora of new notations. Examples of these include the OOD notation of Coad and Yourdon [39], that of Booch [20], the Ptech notation [11], etc. Most such notations now have accompanying graphical editors which will automatically generate software systems based on the graphical design. The P-Tech notation is notable for its extensive graphical description of the system dynamics. It also has a close coupling with the ONTOS OODB, and C++ software applications generated with it exploit ONTOS as their persistent store. Another CASE tool of note is ObjectMaker [129], which supports no less than twenty different notational conventions. However, none of these represents an OODB schema. In fact, as this dissertation is being written, we are in the process of negotiating with the company that produces it to have our own schema notation incorporated into ObjectMaker. A shortcoming of some of the CASE notations (e.g., Booch's [20]) is their reliance on textual matter to fill in the details of the system design (e.g., the attributes and methods of a class). Our notation, on the other hand, concentrates almost exclusively on graphical constructs, with graphical representations provided for all the major OODB schema components.

As with OODBs, object-oriented programming languages (OOPLs) can greatly benefit from graphical representations. The designers of the language Eiffel have recognized this and introduced some graphical conventions in [138]. These conventions constitute a portion of a larger graphical formalism which is under development. As was alluded to by Meyer [138], the formalism will focus mainly on aspects unique to OOPLs, such as class preconditions, post-conditions, and invariants.

In [97], Kappel and Schrefl combine aspects of both OODBs and OOPLs by presenting object/behavior diagrams for OODBs. Since they are presenting the object diagram in the context of behavior diagrams, they have chosen to represent class interconnections with symbols inside the class construct rather than with connecting arrows. As with the CASE tools mentioned above, they rely on textual notation to convey much of the schema information, which appears counter-intuitive to the very idea of a graphical tool, and which we therefore decided to avoid in our own graphical notation.

From our review, we observe that many of the graphical OODB representations in use today were influenced by other data models such as the ER model or functional model. Others were guided by the needs of CASE or behavioral aspects. The influences are reflected in the choice of graphical symbols. In our own work, we are seeking a graphical schema representation purely motivated by the needs of OODBs.

2.1.1 A General Approach to Describing OODB Schemata

Fundamental to an OODB system is the notion of a class, which can be regarded as a container for objects that are similar in their structure and semantics in the application [208]. To describe a class, we will avail ourselves of four kinds of properties, defined informally as follows [142, 69]:

1. Attributes–containers (variables) for values of a given data type. They may be required to always have a value, or they may be given a default value [164].

2. User-defined relationships–named references to other classes. Note that we will drop the qualification and refer to these simply as *relationships* when there is no possibility of confusion (cf. *generic relationships* below).

3. Methods–operations which can be applied to instances of a given class.

4. Generic relationships–similar to relationships in that they are references to other classes; however, these are system-defined, while relationships are user-defined.

To give a formal basis to these characteristics, we follow [214] and define the readable properties of a class [namely, its attributes, relationships, and (readable) methods] as functions which map the extension of the class into some given data type. For example, the attribute *height* of class **person** maps persons into the type FLOAT (floating-point numbers). That is, *height*: $E(\texttt{person}) \rightarrow$ FLOAT, where, in general, $E(A)$ denotes the extension (i.e., the set of all instances) of the class A. A

property may be a partial function, meaning that it may be undefined for certain elements of its domain. To guarantee that a property is a total function, a default value (i.e., some constant from the property's underlying domain) can be used: In the case of an object where a value for the property has not been assigned, the property is given this default value.

As defined, relationships and attributes actually reduce to the same underlying theoretical construct. In fact, a relationship can be viewed as a special kind of attribute with type OIDType [56, 108] holding an object identifier (OID) [5, 114] of an object from the target class. However, without straying into an extended discussion of semantic relativism (e.g., see [39, 88, 103]), we regard them as separate and distinct property types because attributes actually store data which is relevant to a class (or, more precisely, to its instances) while relationships provide pathways to remote data. Distinguishing the two tends to produce clearer schemata and promotes a better intuitive understanding of the application domain. This view is reflected strongly in the structure and style of our graphical representation language. By clearly displaying the class interconnections and their various semantics, our representation provides better expressive power and enhanced readability.

The basis for our graphical language is the labeled, directed graph, where both vertices and edges are labeled. The vertex labels allow us to represent the different kinds of classes (see Section 2.2 below). Similarly, the edge labels permit the representation of the various generic and user-defined relationships, and path methods.

In designing this language, we have taken into account the mnemonic value of the graphical icon. Different mnemonic devices are introduced along with the symbols. Various features of a symbol itself are used to convey its semantics and functionality. This is especially true for relationships, where the edge labels are stylized to capture the intended meaning.

The choice of symbols was also influenced by historical precedents. Because certain symbols have been in wide-spread use for a long time, some user intuition now rests on them. It is also the case that certain OODB constructs correspond very closely with those in earlier data models. Therefore, to exploit the acquired intuition and maintain compatibility with earlier models and notations, we have drawn on some previous data models. For example, the similarity between some of our notation and the ER notation is readily apparent. We have also been influenced by the work of Rumbaugh [171], who also used the ER notation as his starting point.

Another major factor in our design was our desire to see the graphical representation used as a pencil-and-paper device. The task of constructing a large database schema is an arduous one. Advances often occur away from any computer workstation. The ability to quickly jot down ideas on paper at such times is a great advantage. Also, some people prefer to do their designing away from the computer. A notation which permits hand-written diagrams is bound to be of greater utility than one which does not. (Witness the great popularity of the ER model.) The simplicity of our symbols readily lends itself to this purpose.

The graphical schema representation presented herein has, to date, been employed successfully in a number of large modeling projects. These include modeling a telecommunication schema at Bellcore [66]. This schema modeled the phone company's business of providing telephone and other communication services to its customers. It also captured many of the aspects of the physical infrastructure of its circuit and switching systems. The schema provided a framework for research into the problem of database integration [64, 66, 67, 68, 70]. In particular, the research spawned a new type of integration called *structural integration.*

A hypothetical company's purchasing department was another modeling task which was accomplished with our graphical notation [68]. This model tracked the activities of such a department, and as with the Bellcore schema, was used as a test-bed for investigating the benefits of structural integration. Both schemata comprised on the order of fifty to one hundred object classes.

An even larger schema, developed in our research group here at NJIT, is the one which models the activities, data processing requirements, and the overall organizational structure of a university [34, 142]. The schema comprises approximately two hundred and fifty object classes and was constructed using our graphical language and the OOdini schema editor. Conversion of the graphical schema into VML syntax was accomplished automatically by OOdini. Particularly useful in this endeavor were the various levels of display (discussed below) which enhanced communication and fostered a better understanding of the domain among the many participants in

this project. As was mentioned above, this schema has been employed successfully in research into the generation of path methods in OODBs.

One last application worth mentioning is an electrical circuit design program built using the VML OODB and the X Windows and OSF/Motif windowing environment. This work focused on migrating an application based on the theory of *graphical deep knowledge* [59] from an Artificial Intelligence (i.e., semantic network) environment to an OODB. As with the university schema, the underlying schema for the electrical component data was constructed using the OOdini schema editor. VML code was generated by OOdini, as well. And here, too, the graphical language's levels of display helped to promote an understanding of the project among successive participants.

The remainder of this chapter describes the graphical schema constructs in detail. Classes and attributes are discussed in Section 2.2. Generic relationships are considered in Section 2.3, while user-defined and constraint relationships are presented in Section 2.4. Path methods are presented in Section 2.5 followed by abridgements to the representation in Section 2.6. The chapter concludes with a brief discussion of how we represent objects (i.e., instances of schema classes) when a mixed instance/schema diagram is necessary.

2.2 Classes and Attributes

We follow the ER practice and represent an object class as a rectangle with its name printed inside. An attribute is an ellipse which is connected to the class rectangle

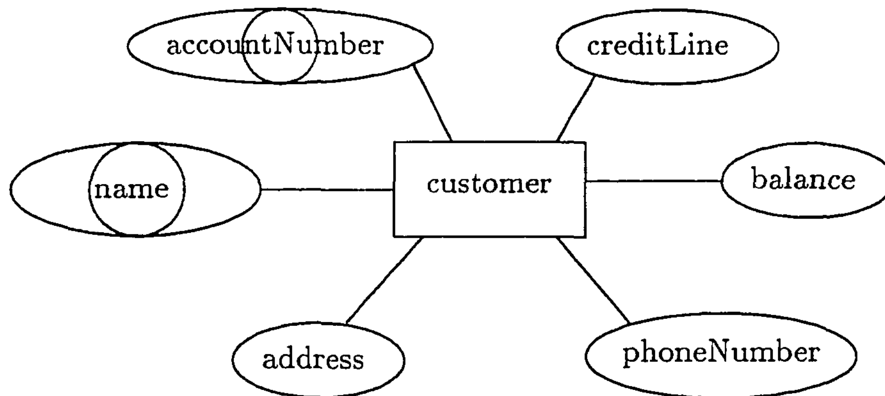


Figure 2.1 The class **customer** and its attributes

via an unlabeled line. The attribute name resides inside the ellipse (Figure 2.1). An attribute can be further classified as *essential*, meaning that its value must be non-nil. In other words, the attribute must be a total function from the extension of the class to the values in its domain. To denote this, an inscribed circle is added to the ellipse. As we will see, the circle will be used consistently throughout our graphical language to represent essentiality (or totality). The attributes *name* and *accountNumber* of the class customer in Figure 2.1 are designated essential.

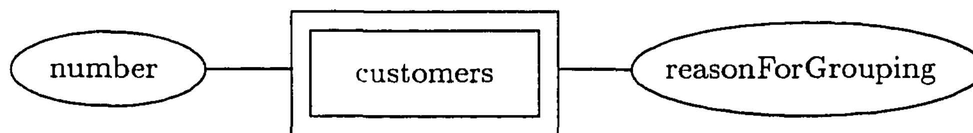


Figure 2.2 The class **customers**

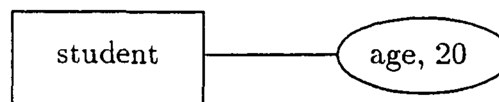


Figure 2.3 The attribute *age* with default value 20

As an alternative to the essentiality condition, an attribute may be given a default value. For example, without knowing a student's age, we may assume that he is twenty years old (and not eligible to drink alcohol). An attribute's default value is written alongside sides its name, inside the ellipse; a comma separates the two. This is illustrated in Figure 2.3.

In addition to a simple class, our system is capable of representing composite classes obtained from other classes by two types of constructors:

1. the set constructor.

2. the tuple constructor.

The set constructor is used to obtain a class whose instances are sets of instances of another class. For example, the class `customers` of Figure 2.2 is obtained by applying the set constructor to the class `customer`. Such a class might have an instance representing the set of all customers who purchased a given product or any product on a given day.

The graphical representation of a set class is a rectangle with a double-framed border. The double-frame is used to convey the inherent multiplicity of sets, their non-atomic nature (Figure 2.2).

The tuple constructor is used for association purposes. i.e., to gather a group of classes together. As a typical example, consider a ternary relation. Sometimes the information expressed in a ternary relation cannot be captured by three binary relations between the pairs of classes [51]. In an OODB, the tuple constructor is

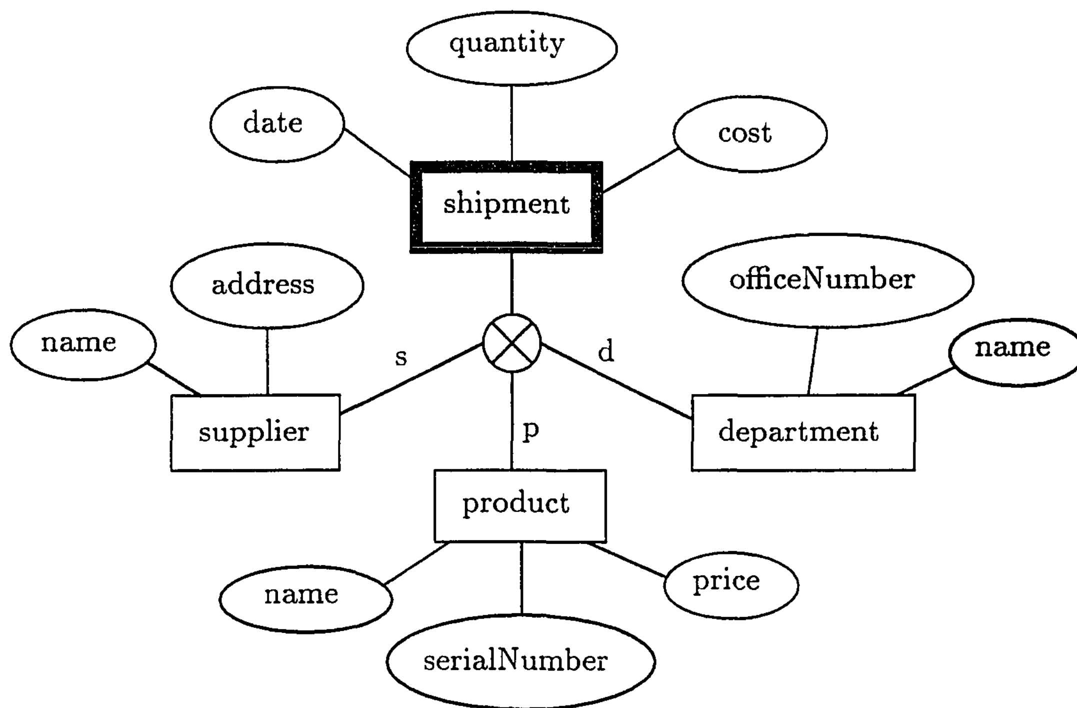


Figure 2.4 A tuple class

used to form a class comprising the three classes of interest. A concrete example of this situation is the class **`shipment`**, which is defined to be a triple composed of **`supplier`**, **`product`**, and **`department`**, as shown in Figure 2.4.

The graphical construct for a tuple class is a rectangle with a heavy borderline representing the class itself. From a *branching point*, a circle enclosing an "X," emanate the connections to the constituent classes. Each of these connections is a line with a label indicating the selector for the particular class. The thick-framed rectangle and its corresponding branching point are connected with an unlabeled line. The branching point has been used for two reasons. First, it distinguishes this intersection of lines from the inevitable incidental crossings of lines in the picture. Second, it conveys the fact that the class is the "Cartesian product" of its component

classes. (More precisely, its domain is such.) Figure 2.4 shows the graphical form of the class `shipment`.

To summarize, the symbol we use for an object class is a rectangle. Composite (i.e., set- and tuple-constructed) classes are represented using rectangles with modified borders: double-framed for sets, and thick for tuples.

2.3 Generic Relationships

As mentioned above, we use the term *generic relationship* to refer to a connection between classes which, due to its generality and importance, is system-defined (or, in other words, is a modeling primitive of the system). The most important generic relationship is *subclass* (IS-A), which enables the expression of specialization and the creation of a class hierarchy. This hierarchy normally forms the skeleton of an application, and its comprehension is essential to an overall intuitive understanding. Thus, in any graphical representation, the hierarchy must be emphasized. For this reason, we have chosen to specify *subclass* as a heavy line directed from the specialized class (subclass) to the more general class (superclass). As we shall see, user-defined relationships are represented using thin arrows; therefore, *subclass* is duly highlighted, and its hierarchy is readily apparent on even the most cursory inspection. To further emphasize the hierarchy, we encourage the placement of a subclass below its superclass.

In the case where the *subclass* specialization is in a different context from that of the superclass, we call the generic relationship *roleof* [144, 179, 180]. The graphical

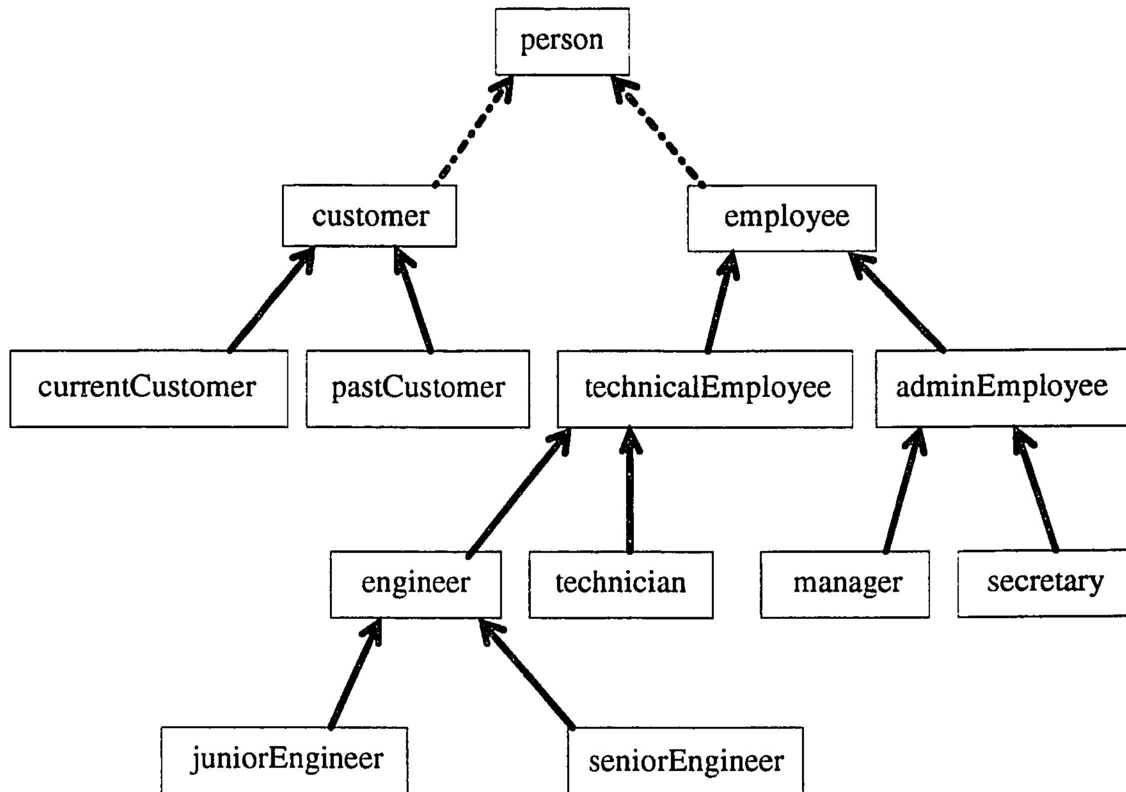


Figure 2.5 A specialization hierarchy.

representation for *roleof* retains the heavy arrow of *subclass*; however, the line is not solid, but a dot-dash pattern. The mnemonic device employed here is borrowed from the world of maps. There, the boundary between any two territorial units, such as states or countries, is defined using a dot-dash pattern. In our case, we denote the *crossing* of the boundary between contexts. Figure 2.5 presents a specialization hierarchy, including *subclass* and *roleof*.

There are two more generic relationships which have to be considered, *setof* and its converse relationship *memberof*. A class A is in a *setof* relationship with class B if the instances of A are sets of instances of B. Conversely, B is in a *memberof*

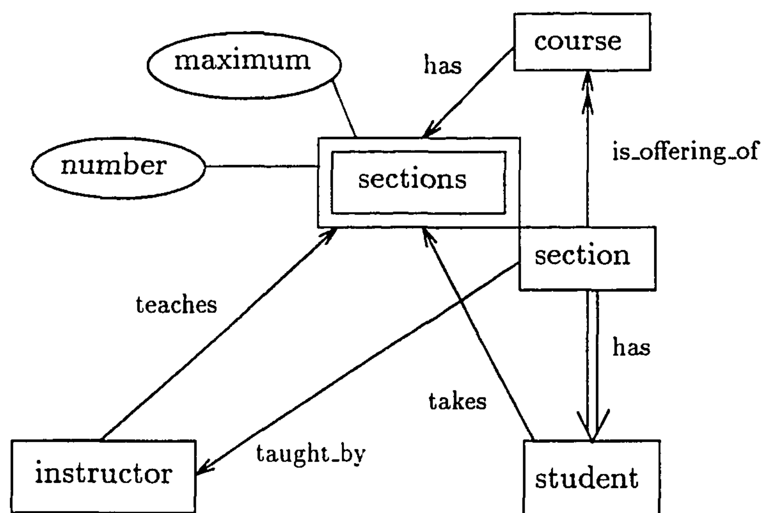


Figure 2.6 The `section-student` example

relationship with A. In such cases, A is obviously a set class and is notated using the double-framed rectangle discussed above. In Figure 2.6, **`section`** and **`sections`** are in a *memberof/setof* configuration. In contrast to the other generic relationships, there is no hierarchy implied by *setof* and *memberof*.

In our graphical representation of these two generic relationships, the two participating classes are drawn so that they touch at one of their corners (Figure 2.6). The set class is, as usual, drawn with a double-framed box. There are a number of reasons why this appears to be a good representation. First, there is the practical issue of conserving space in the picture. Our approach eliminates two bidirectional arrows. In addition, all four sides of each rectangle remain accessible from a graphical standpoint. The adjacency of the two classes along with the presence of the symbol for a set-constructed class clearly conveys the *memberof/setof* relationships between them.

2.4 User-defined and Constraint Relationships

A relationship is a named, user-defined connection directed from one class to another. Since it can be viewed as a pointer, we draw it as a labeled arrow from its class of definition to the target class. The arrow is thin as compared with the heavy arrows of the hierarchical generic relationships.

Often an application requires a relationship from a class `A` to a class `B`, as well as its converse. This situation is handled using a pair of arrows pointing in opposite directions. One should contrast this approach with the ER model, where a relationship is bidirectional and given an "existence" of its own, complete with its own attributes. In OODBs, a relationship is typically defined as a property of one class, acting as a reference to another class.

The ER model supports one-to-many or what we call multivalued relationships. The object-oriented approach supports multivalued relationships in two different ways. The first is a multivalued relationship connection which indicates that an instance of one class can be related to any number of instances of the class to which the relationship is directed. An example of this is the relationship between the classes `section` and `student`, where a given section can have many students (Figure 2.6). We have chosen to represent the multivalued relationship as a dual-lined arrow. This choice emphasizes the multiplicity of the relationship, just as in the case of the set-constructed class (cf. Section 2.2). For comparison, we have included in Figure 2.7 an ER rendition of Figure 2.6.

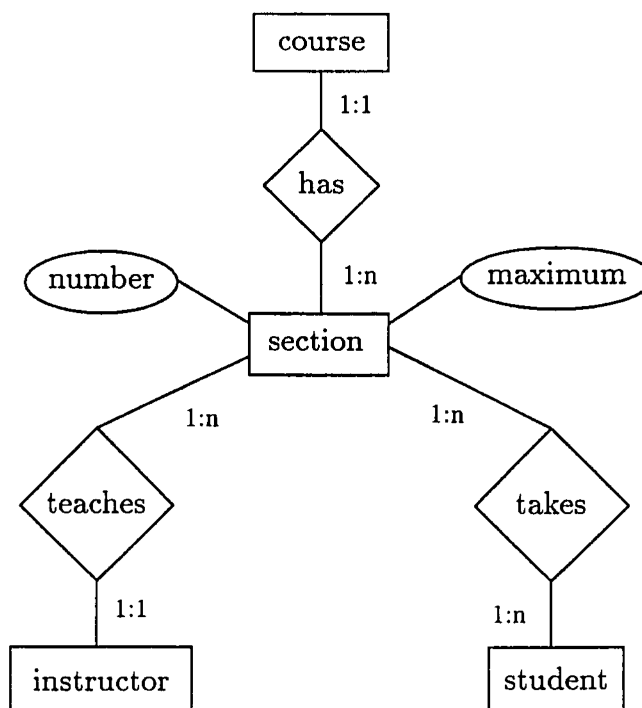


Figure 2.7 The ER model of Figure 2.6

The second alternative is to define a set class. In this case, the multivalued relationship is captured by creating a set class at the "many end" and directing an ordinary single-valued relationship to it. Consider a related example: a student can be in many sections. Using the set alternative, we create a new class `sections`, defined with respect to `section`. We then create a single-valued relationship from `student` to `sections` (Figure 2.6). In this way, we have related a single student with many sections. Here, however, we are required to explicitly group the section objects into a set.

While the two approaches are basically equivalent, the usefulness of the set class alternative becomes apparent when trying to model relationships with cardinality constraints. Assume that we are trying to model the interrelations between courses,

instructors, and students. We first define the classes `course`, `instructor`, and `student`. Because there are a number of sections offered for each course, we also need a class `section`. Now assume the following constraints:

- at most r sections of a given course can be offered in a semester.
- an instructor may teach no more than s sections in a given semester.
- a student may take at most t sections per semester.

We could model this situation by having relationships from each of the three classes `course`, `instructor`, and `student` to a set class `sections`, defined with respect to `section`. `Sections` would be given the attribute *number* to maintain the cardinality of an instance, as well as an attribute *maximum* which would hold the maximum cardinality (Fig. 2.6). This latter attribute would be set at instantiation time to an appropriate value (e.g., to r if the set were to consist of the sections of a particular course). The method to add an instance of `section` to a given instance of `sections` would then check the current cardinality and deny any request which would violate the prescribed maximum.

These cardinality constraints could alternatively be enforced by each of the three classes `course`, `instructor`, and `student` individually. This can be done by placing two additional attributes, *numberofSections* and *maxNumberofSections*, in each of the three classes. These attributes play the same roles that the attributes *number* and *maximum* did in the class `sections` above. Next, multivalued relationships are

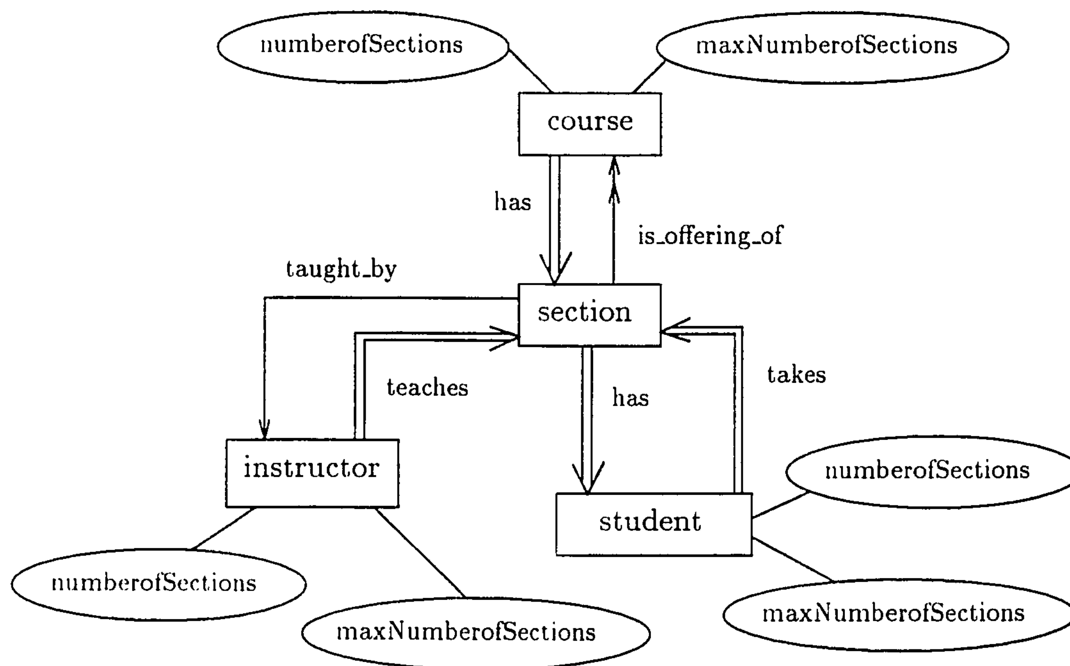


Figure 2.8 Alternative form of Figure 2.6

established between each of the classes and `section` (Figure 2.8). Lastly, each class is equipped with methods to monitor the constraints.

There are a number of reasons why this is not as elegant a solution as the former. First, the multivalued relationships do not convey information about the required cardinality constraints. The set alternative makes the structure of the model more meaningful. Second, the cardinality constraint is really a characteristic of the set of sections associated, for example, with a given course, not of the course itself. Hence, this constraint should be defined as a property of `sections` rather than `course`. Finally, redundant specifications are eliminated by placing the two attributes and the corresponding "watchdog" method in `sections`, instead of repeating them three times.

Constraint relationships are those which impose additional semantic constraints on the participating classes. In general, a constraint relationship requires two aspects of definition: the static or state definition which imposes constraints on the database at any fixed instant of time; and the dynamic or transient definition which expresses the behavior that it implies in the context of change (i.e., the creation, deletion, and update semantics). The dynamic aspect of any constraint relationship is required to maintain the constraints imposed by the static aspect.

We represent three kinds of constraint relationships, *essential*, *range-restricted*, and *dependent*. All of these relationships are ordinarily used to maintain referential integrity [46] among the instances of the database.

An essential relationship is one which must always refer to an existent object (i.e., which may not have a nil value). Its creation semantics is such that the referent class of the relationship must have instances before any instances of the source class can be created. The update semantics is: the relationship cannot be assigned a value of nil. Finally, the deletion of an instance of the referent class is forbidden if there exist instances of the source class which refer to it.

To represent an essential relationship, we place a small circle behind the head of the arrow representing such a relationship. This symbol was chosen to maintain consistency with respect to the rest of the graphical representation, as essential attributes are also denoted by the addition of a circle. Hence, adding a circle to an attribute or relationship consistently expresses essentiality.

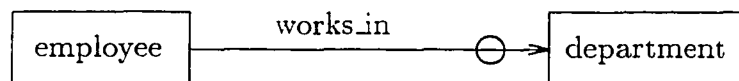


Figure 2.9 An essential relationship

An example will clarify the above points (Figure 2.9). In this database, every employee must have an associated department (and, in fact, exactly one of them). If there are no departments, then no employee can be hired (created). If a department is abolished (deleted), then first all its employees must be moved to other departments or fired. Figure 2.9 could be read: Working in a department is essential to an employee.

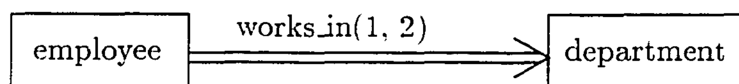


Figure 2.10 A range-restricted relationship

An essential relationship is a special case of the more general *range-restricted* relationship which imposes a range restriction on the cardinality of the set of referent objects of any source object. For example, instead of requiring that an employee work in exactly one department, we may want to allow an employee to work in one or two departments. Thus, the relationship *works_in* should be given a restriction conveying this information. Such a range restriction is denoted by placing a numerical range in parentheses next to the relationship's name. So, in our example, "(1, 2)" is added to the relationship symbol as illustrated in Figure 2.10. Note that the relationship is written with a double line to indicate that, in general, it is multivalued. Omission of

either of the bounds is indicated by writing "–" in its place. If both the upper and lower bounds are the same, then we consolidate them into a single number written in parentheses.

A dependent relationship has the following deletion semantics: Assume that the class A has a dependent relationship to class B; if an instance a of A refers to an instance b of B, and b is deleted, then a is also (automatically) deleted. Thus, the existence of an instance of A is *dependent* on the existence of an instance of B. We represent a dependent relationship as a double-headed arrow (either single-lined or dual-lined). The double head of the arrow emphasizes the "stronger" directed connectivity of this type of relationship. In Figure 2.6 and Figure 2.8, we see the dependent relationship *is_offering_of* from section to course, indicating that if a course is deleted, all its associated sections are deleted, too.

Essentiality and dependency may also occur in the context of multivalued relationships. A *multivalued essential relationship* is one which carries a "one or more" semantics, meaning that each object of the source class is required to have one or more referent objects of the target class. Clearly, such a relationship is equivalent to the special case of a range-restricted relationship with a lower bound of 1 and no upper bound. Nevertheless, for this special case we adopt the convention of adorning the multivalued relationship symbol with a circle. A *multivalued dependent relationship* is similar to an ordinary dependent relationship except here, with possibly many referent objects in general, the deletion is not propagated to the source until the set

of referents becomes empty. Such a relationship is denoted by placing a double arrow on the multivalued relationship symbol.

2.5 Methods

We distinguish between two types of methods in OODBs: path methods and local methods. As their name implies, local methods operate strictly locally to an object; i.e., no remotely accessed data is used in their operation. (We note, however, that calls could be made to other objects provided as parameters.) Local methods can be divided into selectors/mutators (also referred to as readers/writers [100]) and derived attributes [94]. (In later chapters, we will extend the notion of derived attribute so that it may be defined in terms of constructs external to a class. In particular, we introduce derived attributes defined in terms of information propagated across part relationships.) A *selector* method simply reads a given attribute. In contrast, a *mutator* assigns a value to an attribute. Selectors and mutators do not require separate graphical representations. The symbol representing the attribute they operate on is sufficient.

Derived attributes are very similar to the selectors of attributes. These methods derive values from one or more attributes through some computation. An example of a derived attribute is the "available" method of the set class `sections`. This method computes the available "room" in a given set by subtracting the attribute *number* from the attribute *maximum*. Derived attributes require a unique symbol. We have chosen an ellipse with a dashed perimeter enclosing the derived attribute's name.

The ellipse, as usual, is attached to its class via an unlabeled line (Figure 2.11). The reason for our choice is that a derived attribute can be viewed as a hybrid of an attribute and path method, and attributes are represented using ellipses while, as we will see presently, path methods are represented with dashed arrows. (This dashed representation, as shall be seen, is also consistent with our notation for part relationships introduced in succeeding chapters.)

A *path method* [133, 134, 135, 136] is a method which traverses from a source object through a sequence of relationships of a schema of classes to retrieve some related (target) object or the value of an attribute of that target object. In general, it can be concatenated with some mathematical operation. A path method can be viewed equivalently as a sequence of messages passed along a path of relationships, or (using Smalltalk terminology) as a nested message. The concept is similar to that of *path expression* as presented in [24] and, even earlier, in [141]. The more general notion of path expression as defined in [104], which subsumes the preceding definitions, will also be employed in this dissertation when we discuss part relationships.

Before getting to the graphical representation for path methods, let us introduce a special textual notation for them which we will be employing later on. The notation should help to clarify what we mean by the concept of path method.

Syntactically, a path method comprises two distinct parts: (1) a head or signature which includes information regarding its name and formal parameters; and (2) a body describing the path traversed through the schema by the method. The head of

the method, comprising the method's name followed by a pair of parentheses (which enclose the optional formal parameters), is written in front of the method's body and is separated from it by a colon. The body is written as a colon-separated sequence of pairs of the form $r_i \rightarrow A_i$, where r_i is a relationship and A_i is a class for $1 \leq i \leq n$. Each such pair represents the traversal of r_i from the class A_{i-1} (of which r_i is a property) to A_i. A_0 is the method's class of definition, one of whose properties is the relationship r_1; the class A_n is the method's destination. As an example, consider a method "get_object" defined on a class A which retrieves objects of class D related by a sequence of relationships r, s, t (the latter two defined on classes B and C, respectively). This method would be written as:

$$\text{get_object}(\,) : r \rightarrow B : s \rightarrow C : t \rightarrow D.$$

Alternatively, if the method retrieves the value of an attribute, the final pair is of the form $a \rightarrow T$, where a is an attribute of the class of the penultimate pair and T is its data type. For example, if the above method is revised to retrieve the value of attribute d (defined on class D with type τ) of the related object, then it would be written as:

$$\text{get_attribute}(\,) : r \rightarrow B : s \rightarrow C : t \rightarrow D : d \rightarrow \tau.$$

We also note the convention that a path method may replace a relationship in a pair, in which case the second element represents the class at the terminus of that method.

If the relationship in a pair is multivalued, then the transition, in general, will yield a set. This situation is denoted by placing the second item, i.e., the class, in curly brackets. The succeeding pair, if it exists, is also affected in that it now represents an iterative application of the given relationship to the resultant set. The iteration is denoted by placing the "@" sign in front of the relationship. Because the result of this transition is a set as well, its class, too, is written in curly brackets. As an example, assume A has a multivalued relationship u to E, which itself has a relationship v to F. The method to retrieve those F's indirectly related to an A is written as:

$$\text{get_Fs}() : u \rightarrow \{E\} : @v \rightarrow \{F\}.$$

If the multivalued transition precedes a pair which accesses an attribute, then the iteration will actually yield a multiset. This case is distinguished by placing the subscript "m" after the brackets surrounding the name of the data type. The pair following such an iterative attribute access may represent a mathematical operation (e.g., min) intended for the multiset as a whole. In such cases, the "@" designating an iteration is omitted.

The symbol employed for path methods is a dashed, thin-lined arrow pointing from the source class (i.e., the method's class of definition) to the class or attribute

of the target data object. Any trailing mathematical operations of the method need not be represented graphically and are omitted from our discussion. The reason for the choice of this symbol is as follows. The function of a path method is similar to that of a relationship: Each is used to retrieve relevant information from another class. We therefore chose the thin arrow so as to make the symbol for a path method reminiscent of the representation of a relationship. However, there is a difference between relationships and methods. A relationship is a direct connection, while a method is an indirect connection established via a chain of other connections. In this sense, a method can be viewed as a composite construct, and we employ the dashed-line to convey this composition. The pieces of the line signify the pieces of the schema path.

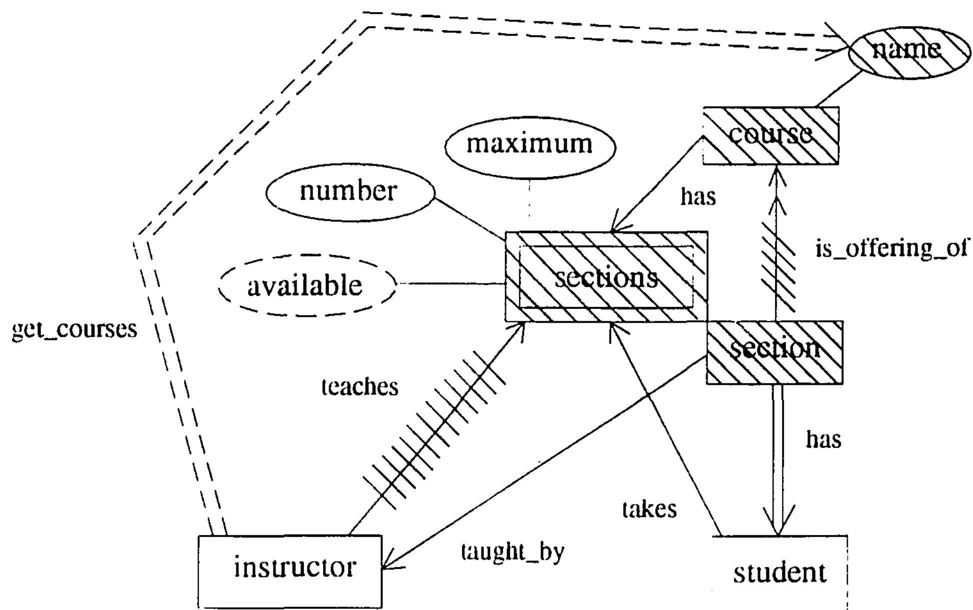


Figure 2.11 The `section-student` example with methods

As an example, consider the method "get_courses" of the class `instructor` (Figure 2.11). This method returns the names of all the courses taught by a particular instructor. To accomplish this, it accesses the attribute *name* of `course` through the user-defined/generic relationship path: *teaches*, *setof*, *is_offering_of*. More specifically, it operates as follows. It starts by applying the relationship *teaches* to `instructor` yielding `sections`. Applying *setof* then gives a set of instances of `section`. Next, applying *is_offering_of* to each instance of the set of sections yields a set of instances of `course`. And, finally, applying the attribute *name* to each instance of this set produces the desired result, a set of course names. In the textual notation described above, the method is written as follows, where "nameType" is the data type of the attribute *name*:

$$\text{get_courses}(): \quad teaches \rightarrow \texttt{sections} : setof \rightarrow \{\texttt{section}\} :$$
$$@is_offering_of \rightarrow \{\texttt{course}\} : @name \rightarrow \{\text{nameType}\}.$$

Since the desired data is stored as the attribute *name* of the class `course`, we represent this method as a dashed line pointing from `instructor` to *name*. The fact that it is a double line indicates that the path method is, in general, multivalued.

We also show the pictorial representations of the example methods "get_object," "get_attribute," and "get_Fs" in Figure 2.12. Again, note that, just as with a mul-

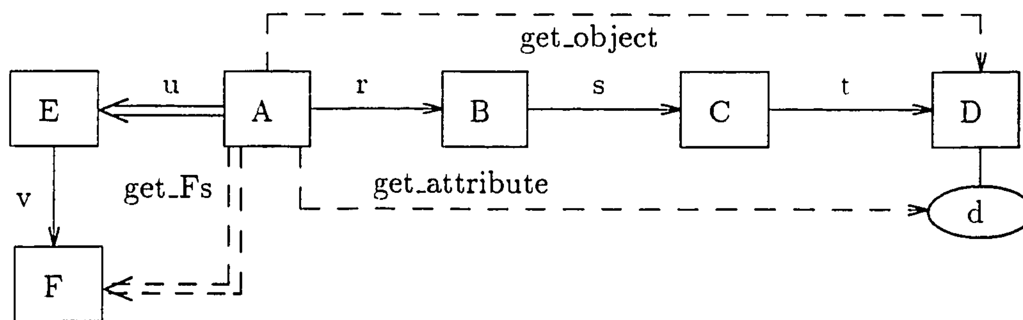


Figure 2.12 Some example path methods

tivalued relationship, a multivalued method like "get_Fs" is drawn with a dual line to convey its multiplicity.

There are actually two aspects of a path method which should be represented graphically. The first one, which we have just presented, displays the connection between the source class and target data item of the method. This aspect reflects the retrieval effect of the path method (i.e., what data it actually returns). As we mentioned above, this aspect functions similarly to a relationship, and hence was given a graphical symbol similar to the relationship icon.

The second aspect of a path method is the chain composed of classes which are connected by generic and user-defined relationships. This aspect reflects the implementation of the retrieval mechanism. Clearly, it is a critical portion of the definition of a path method. Without it, one cannot judge if the method is semantically correct, i.e., whether it correctly retrieves the desired information.

Obviously, we need a graphical representation of this second aspect as well. Since it is a chain composed of elements which are already represented graphically, it

is natural to simply highlight those elements in some manner. We have chosen to "stripe" the elements (see Figure 2.11), though alternate forms of highlighting which do not clash with other parts of the schema representation would be perfectly acceptable. For example, in the context of an editor program (such as the OOdini system), the elements could be emboldened or given some tiling pattern or color.

We note, however, that highlighting the chains of all path methods in a schema will leave much of the schema highlighted and render most chains unrecognizable due to overlaps. Therefore, this highlighting must be used sparingly. In this sense, it is similar to italicization in written natural language. If overused, it becomes confusing and ineffective. In fact, it has been our experience that a designer or user is not interested in the chains of all path methods simultaneously. Typically, one wants to concentrate on the implementation of a single method. In such cases, only the method of interest would be highlighted. If one wants to determine all the available methods, the retrieval aspect represented by the dashed line is sufficient. Therefore, we view the highlighted aspect of the representation of a path method as optional. If employed, it should be restricted to one or a small number of methods.

To conclude our discussion of the graphical language, we provide a summary of all the graphical symbols in Table 2.1. Note that any text in parentheses is descriptive material and is not an aspect of the symbols themselves.

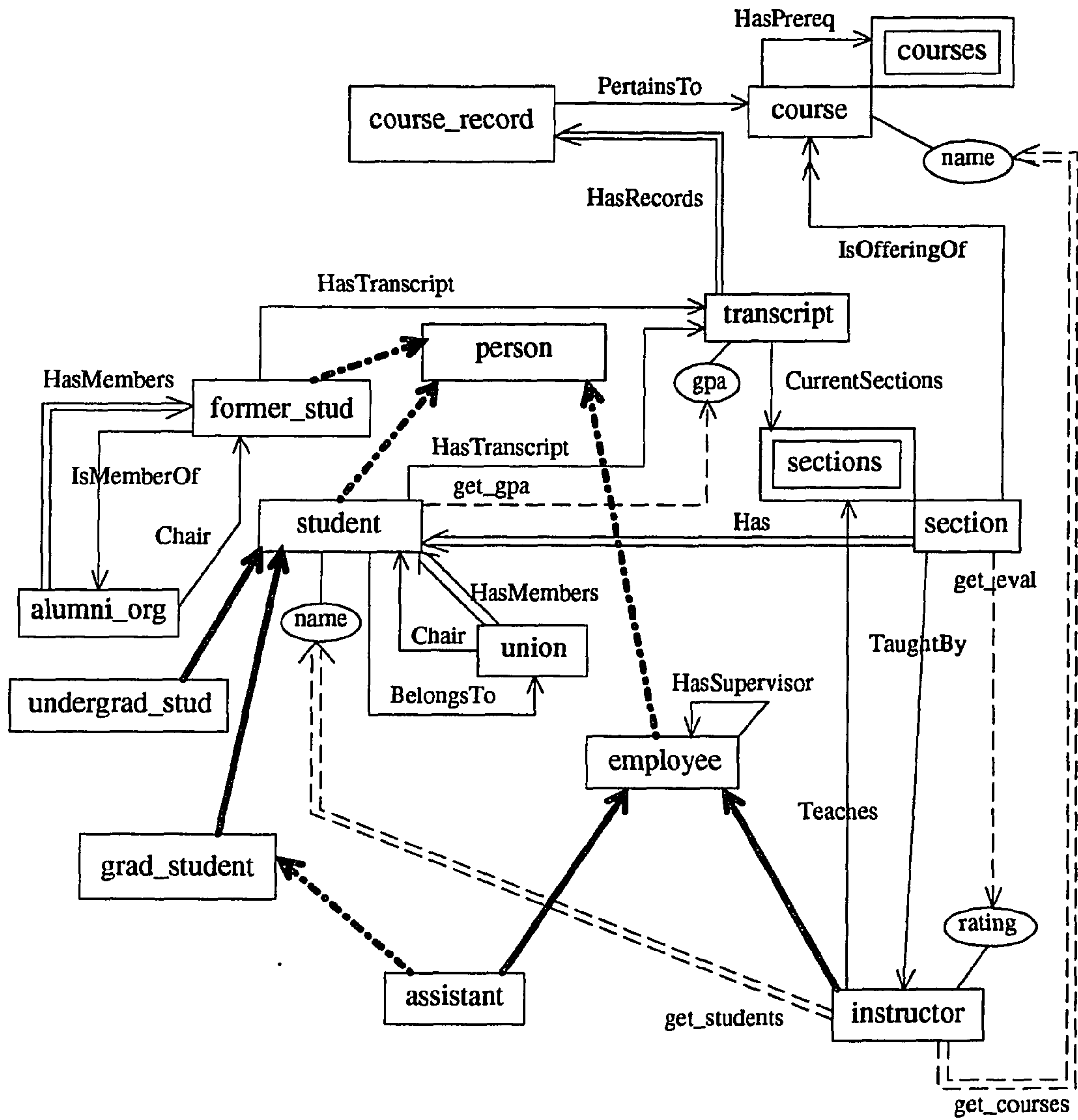
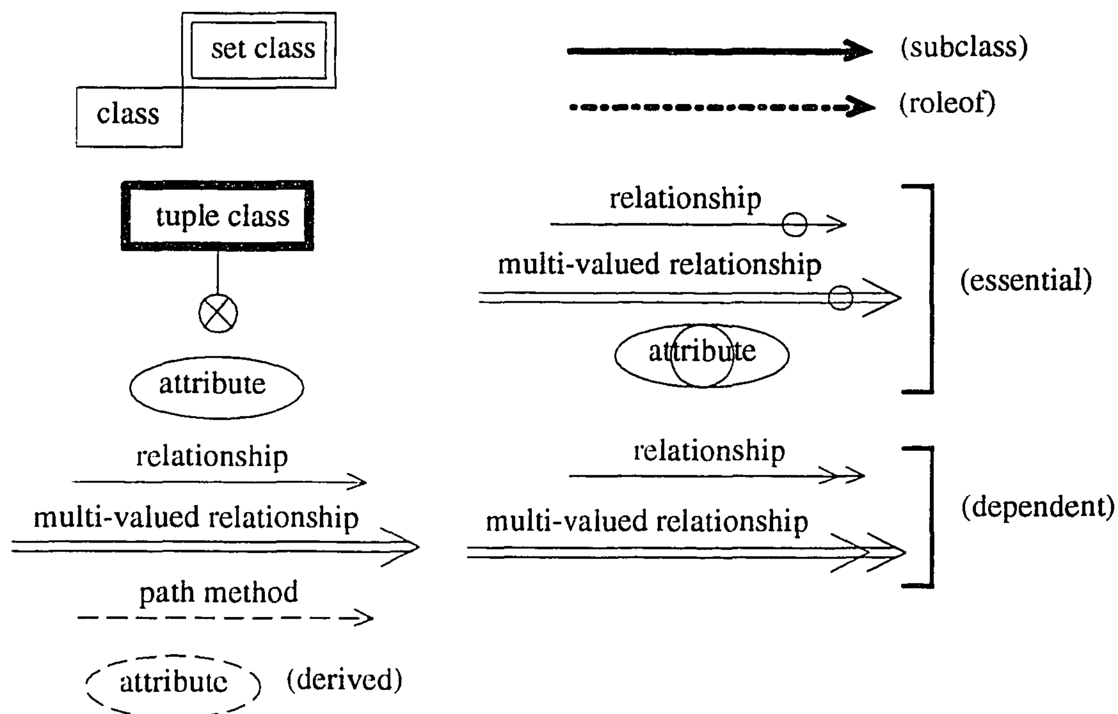


Figure 2.13 An excerpt from a university database schema

Table 2.1 The graphical schema constructs



Finally, in Figure 2.13 we present a larger schema example which has been extracted from the university database schema developed in our research group. Extended versions of this can be found in [19, 34, 120, 209].

2.6 Abridgements

An important issue for any graphical representation language is that of abridgements. It should possible to omit certain features of the language and still have a meaningful picture. The process of omitting extraneous features from a diagram has been referred to as *graphical abstraction* [159, 160]. The omission of features can be important for a couple of reasons. One obvious, and perhaps mundane, reason is

for the preservation of screen real estate. Anyone who has attempted to construct schema diagrams interactively knows how precious additional screen space is. Another more important reason is for conceptual clarity. If a user is trying to become oriented with the schema, then it is probably better to skip the fine details (e.g., the attributes) and just display the higher level constructs like the classes and their associated IS-A hierarchy. This way, the user is not overwhelmed by a flood of too detailed data.

For our graphical language, we have opted for two levels of abridgement or, if you prefer, three "levels of display." These levels of display are characterized by the following:

1. no omissions.

2. omission of attributes and local methods, except for those which participate in path methods.

3. omission of all attributes, relationships, and methods, leaving only the classes, hierarchical generic relationships, and the *setof/memberof* relationships.

Of course, Level 1 is the full-blown representation, where all the details of the schema are included. Level 2 is provided for situations that do not require such fine detail. Browsing the schema for pertinent classes and relationships is an example of such a situation. Level 3 gives an isolated view of the class hierarchy. During the course of our own modeling endeavors, we found this to be particularly useful. Establishing the

IS-A hierarchy and gaining a clear understanding of it is critical when modeling with OODBs. We will see that the same is true when modeling with the part relationship. Its hierarchy is also included in a Level 3 display.

2.7 Representing Instances Graphically

In circumstances where we need to show instances of a database (e.g., when we are discussing constraints imposed on instances by the schema definition), we will employ mixed class/instance diagrams. These diagrams will contain ordinary schema symbols as well as a special notation for instances. For the representation of an individual object, we will follow the convention set forth in [171]. An object will be denoted as a rectangle having rounded edges; the name of its class will be written in parentheses inside of it. This latter convention implicitly captures the "instance of" relationship between an object and its class. Explicit "instance of" arrows tend to clutter the diagram and are avoided. Deviating slightly from the notation of [171], we will typically include an arbitrary OID and perhaps some annotation for an object. The OID carries no specific meaning. It is employed solely to distinguish one instance from another.

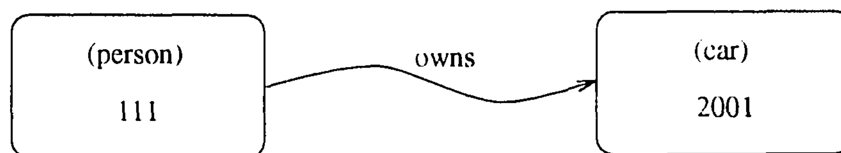


Figure 2.14 An instance of `person` which owns an instance of `car`

An occurrence of a relationship (i.e., an instance-to-instance reference with respect to a schema relationship) will be designated by drawing a curved line connecting the participating instances. The curved line will be labeled accordingly. An illustration of this appears in Figure 2.14, where we show person "111" as the owner of car "2001." The notation for the occurrence of a relationship will be employed extensively when we introduce the part relationship in Chapter 4.

CHAPTER 3

OOdini, AN OODB GRAPHICAL SCHEMA EDITOR

To facilitate and promote the use of the OODB graphical schema notation that we introduced in Chapter 2, we have built a software system called OOdini (Object-Oriented diagrams, New Jersey Institute of Technology). OOdini is a graphical schema editor that allows a user to create and manipulate an OODB schema described in our graphical schema representation. It can also serve in the role of a schema browser as it did frequently in our various research efforts on OODB modeling [34, 120]. Furthermore, OOdini provides the means for converting the graphical schema representation into a number of OODB data definition languages, including the VODAK Model Language (VML) [56, 109] of GMD-IPSI. As such, it is an effective OODB graphical interface.

OOdini comprises about 30,000 lines of C Language code. It runs on a Sun 4/20 workstation and operates in the X Windows [175], X Toolkit (Xt) [10], and OSF/Motif environment [150, 151, 152]. The Motif widget[1] set (see, e.g., [213]) was used exclusively to build its user interface. The work that went into OOdini's construction is reported in a number of Master's projects and theses [35, 111, 128, 183].

In this chapter, we will cover the details of OOdini. First, we give an overview of the X Windows and Motif environment in which OOdini operates. After that, we

[1]A *widget*, in X Windows parlance, is simply a component of a graphical user interface. For example, push buttons, menu bars, sliders, and so on, are all widgets.

describe the operation and features of OOdini. Finally, we consider the conversion of OOdini's graphical schema to textual code representations for various OODBs. In particular, we discuss how OOdini converts the graphical schema into an abstract OODB language, that we have devised, called OODAL. OOdini is also capable of conversion into Dual Model [65, 143] syntax (referred to as DAL) and, as mentioned above, VML.[2]

3.1 X Windows, the X Toolkit, and OSF/Motif

X Windows [175, 176] (or just X) is a device-independent, network-transparent windowing system which was developed to support the use of powerful input/output hardware such as high-resolution bit-mapped graphics displays, mice, track-balls, etc. X employs a client-server software architecture as illustrated in Figure 3.1. At the foundation of a working X implementation is a process called an X Server (or just Server, for short). The Server is in charge of managing all of the I/O resources of a single computer system. On the input side, it monitors the keyboard, mouse, and any other input devices connected to the system, and informs clients when input events occur. On the output side, the Server manages the system's display screen, maintaining its cursor (or sprite [213]) and doing all text and two-dimensional graphics drawing requested by clients. Most importantly, the Server allows clients to partition the display screen by creating *windows*. A window is simply defined to be a rectangular region of the display screen. Windows may overlap and obscure

[2]It does require some human post-editing.

each other. Each window is owned by a single client, which is always the one that created it.

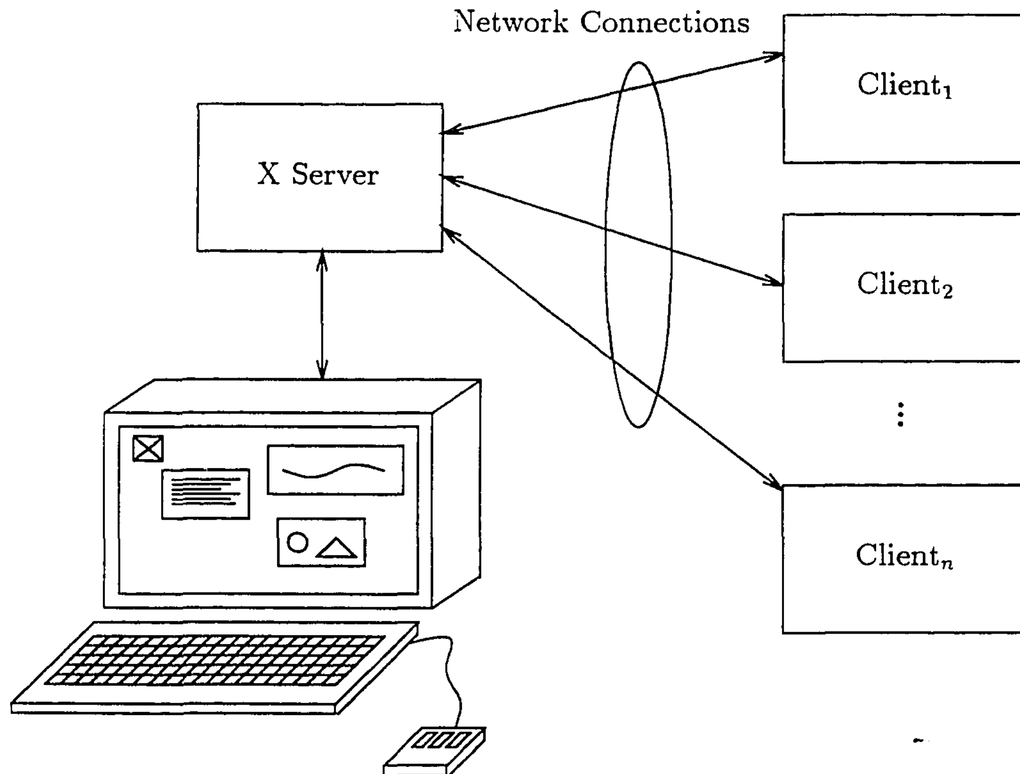


Figure 3.1 The X Windows software architecture

Communication between a client and the Server takes place across a two-way network connection, as shown in Figure 3.1. An important point to remember is that the client and server need not be running on the same machine. In fact, the network may be anything from an intercontinental long-haul link to a local-area network contained within a single building or room. If the Server and the client happen to reside on the same machine, then the network connection is made using

the ordinary interprocess-communication mechanism of the host operating system (e.g., the pipe mechanism of UNIX).

Clients pass messages to the Server to issue orders or obtain information concerning the I/O hardware. For example, a client may request that a window be created; or it may query the Server to determine the background color of an existing window. Such messages must conform to a special communication protocol referred to simply as the X Protocol [146, 176]. The server also responds using this protocol.

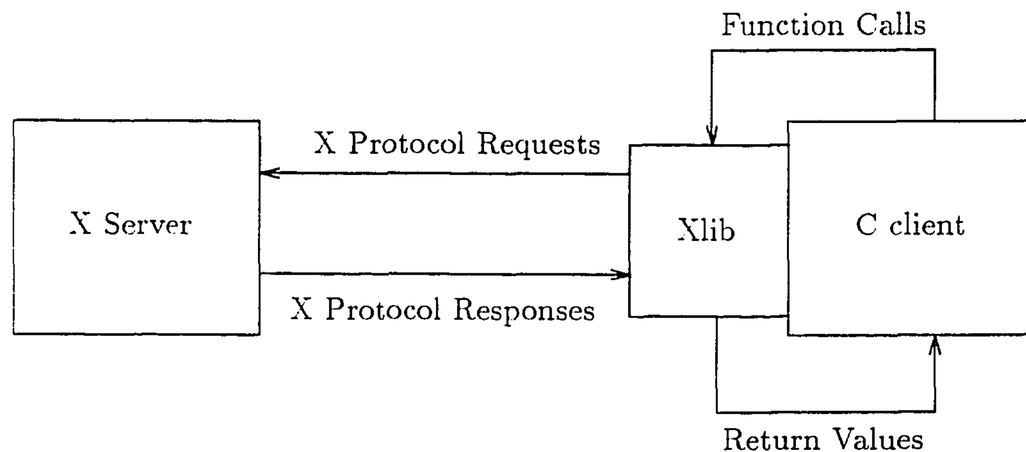


Figure 3.2 Communication between a C client and X Server using the Xlib

A client written in C (such as OOdini) or LISP can use the X Library (Xlib, for short) to communicate with the Server. The Xlib is a collection of functions which translate requests from the client into actual X Protocol messages across the network. We illustrate this in Figure 3.2. In order to issue a request (e.g., to create a window), the client calls a specific Xlib function. (See [147] and [167] for complete descriptions of the available Xlib functions.) The function then sends a corresponding X Protocol request via the network to the Server. If appropriate, the Server replies with an X

Protocol message of its own. This is then translated by the Xlib function into an appropriate return value for use by the client process.

The interaction between a client employing Xlib and the X Server usually follows a set pattern. The client first issues requests to the Server to establish certain I/O resources such as windows. The client then informs the Server of the kinds of I/O events that it is interested in receiving. (I/O events include such things as key presses, mouse button presses, mouse movement, window exposure, and so on.) After that, the client process enters an *event loop* [147] where it waits for any incoming event messages from the Server. When a desired event (e.g., a mouse button press) occurs, the Server sends the client an event message containing all the details of the event (e.g., which mouse button was actually pressed). The client, of course, can then act on the message as it sees fit. Once it finishes processing the current message, the client re-enters the event loop to await additional messages.

The above described interaction between a client and the Server takes place at the level of the I/O hardware, i.e., in terms of input events and window manipulation. As such, it is often described as a low-level interaction [10]. However, for a client (such as OOdini) to effectively employ an advanced graphical user interface (GUI), it must be able to operate at a higher level of abstraction. For example, a client using a push button only cares to know that the button was "pushed" or activated, not that the 1st mouse button was pressed while the screen cursor was in the region of the graphic display where the button is drawn. Using Xlib, a client would be

forced to make such translations itself, which would entail an enormous and complex software construction effort. What is needed is the means for a client to operate exclusively at the higher-level of abstraction. This is provided by the X Toolkit or Xt [10, 148, 153].

Xt, built on top of Xlib, is a software module which provides the framework for the construction of advanced GUIs. At the heart of this framework is an abstraction known as a widget. As we mentioned above, "widget" is just a generic term for any graphical user interface component. Such components include things like command buttons, scrollbars, dialog boxes, menus, and so on. Each kind of widget is endowed with a set of abstract behavioral patterns (analogously to objects in the object-oriented paradigm). For example, the push button, mentioned earlier, can exhibit three types of behavior: *arming*, *disarming*, and *activation* [150, 151]. More complex widgets like scrolled lists and menus exhibit a larger variety of behavioral patterns.

To build a GUI using Xt, a client does the following three things. First, it creates a group of widgets of various types. It then pieces these together to form the complete GUI that is presented to the user. Finally, the client attaches the widgets to various aspects of its own functionality as described presently.

In order to connect widgets to their clients, a list of *callback functions* (or simply *callbacks*) is associated with each type of abstract behavior. A callback is an application function registered with the widget by the client to be invoked any time the widget exhibits (or is forced to exhibit) a specific behavioral pattern. For example,

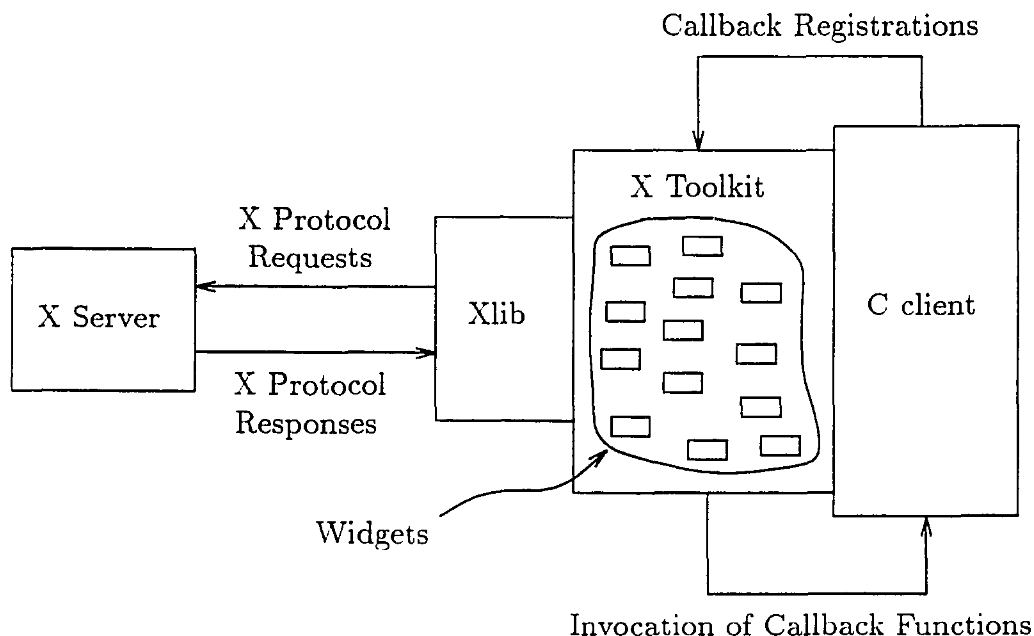


Figure 3.3 Communication between a C client and its Xt widgets

every push button widget has an "activation" callback list. When such a button is pushed (i.e., activated), each of the client's functions appearing on its activation callback list is invoked in succession. The client can then perform actions commensurate with the button activation (such as carrying out a prescribed task). The expression "call back" refers to the fact that, after registering functions with a widget, a client is "called back" at a later time. The callback interaction model between a client and its widgets is illustrated in Figure 3.3. There it can be seen that, instead of working with the low-level Xlib, a client communicates exclusively with its widgets; direct interaction with the Server is left for the widgets themselves.

As it turns out, Xt only provides a core set of widgets. It is up to outside developers to build sets of widgets which exhibit desired behavior or "look and feel"

[213]. Widget sets have been created by a number of organizations, most notably AT&T with its OPEN LOOK widgets [201] and the Open Software Foundation (OSF) with its Motif widget set [150, 151]. Motif, in fact, comprises not only a widget set but also a window manager [147, 150] and a GUI style guide [152]. Of course, the Motif widgets were designed to conform to and promote OSF's own preferred interface style which is laid out in [152]. The OOdini interface is constructed entirely from the Motif widget set. In the next section, we discuss OOdini's functionality and refer to some of the Motif widgets that went into its interface.

3.2 OOdini's Features and Operation

OOdini is a constraint-based graphical schema editor designed specifically for the OODB schema representation presented in Chapter 2. To see what we mean by constraint-based, consider the case of a user-defined relationship which is defined as a labeled arrow directed from some source class to a target class. Specifically, consider the case where such a relationship is emanating from a class but left dangling or unattached at its other end. Clearly, such a construction is meaningless in our schema language and, hence, should not be allowed. Toward this end, OOdini *constrains* a relationship symbol such that it always touches a class at both of its ends. So, during input, OOdini requires that the user fasten each end of a relationship to some class. Moreover, if at a later time one of these classes is moved, the relationship is automatically moved relative to it to maintain the proper connection. In this way, OOdini guarantees that the integrity of the schema diagram is always maintained,

and it relieves the user of excessive, tedious manipulation. We do point out that OOdini is not a general-purpose editor incorporating the features of a graphical-constraint toolkit such as Garnet [140]. Rather, it is a software tool fine-tuned for the manipulation of our own OODB schema representation.

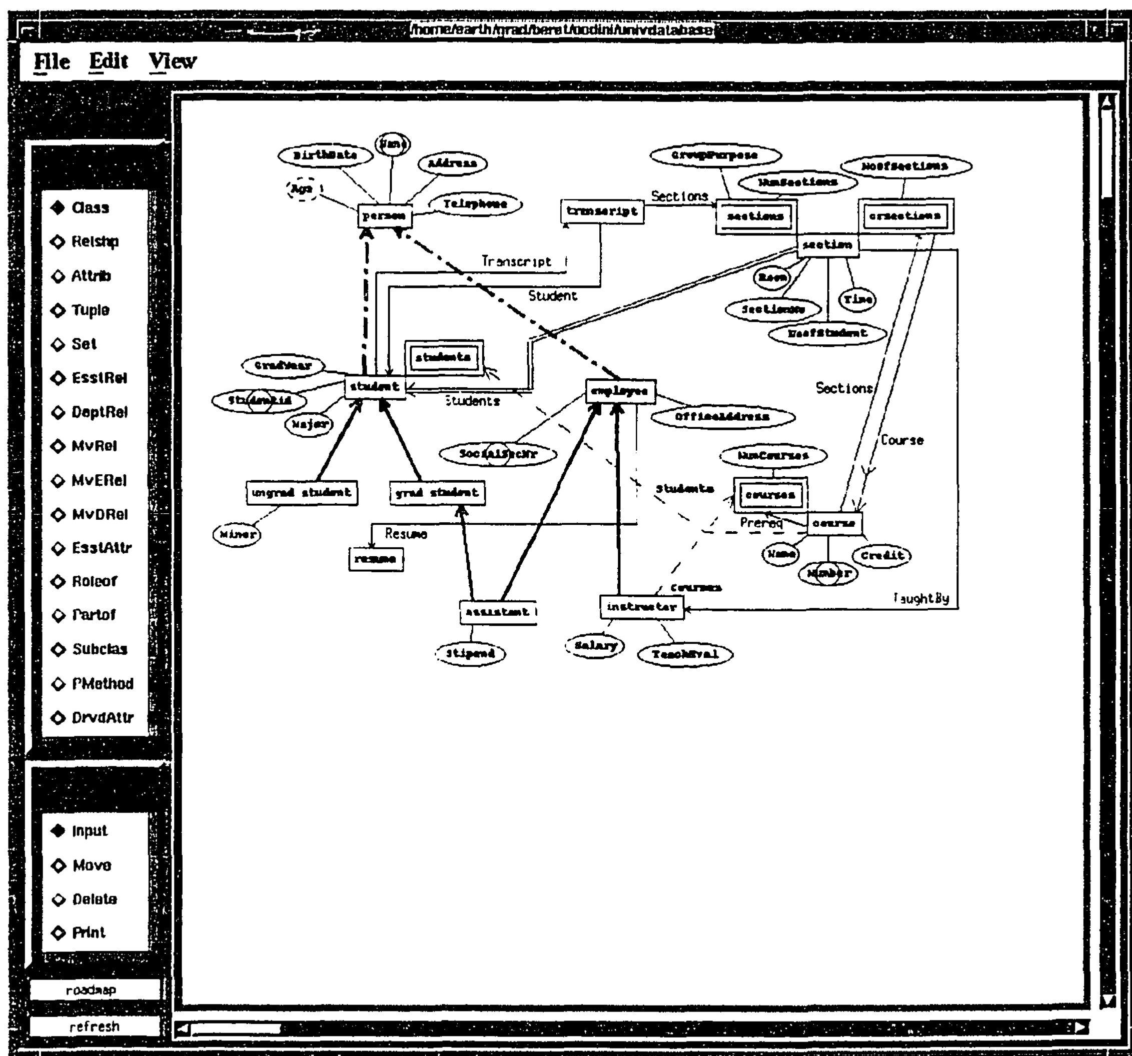


Figure 3.4 OOdini's main screen

To maintain conformance with the Open Software Foundation/Motif Style Guide [152] and allow OOdini to fit smoothly into the Motif working environment, OOdini's primary interface is built using Motif's requisite MainWindow widget [150]. (See [152] for further details.) This widget holds a variety of other widgets which permit access to OOdini's functionality. OOdini's main window can be seen in Figure 3.4. (The schema shown there is an excerpt from our university database schema; it includes examples of most of OOdini's graphical schema symbols.) At the uppermost portion of this main window, we see the menu bar which contains the standard array of entries, "File," "Edit," and "View." Each of these activates a pull-down menu comprising commands relevant to the topic. These will be discussed below. On the left-hand side appears a pair of RadioBox widgets (which are special forms of RowColumn widgets [151, 213]). The upper one, labeled "Object," is used to select the current type of schema object. For example, if we are currently interested in working with (e.g., inputting) classes, then we select the button labeled "class" in this radiobox. We refer to this widget as the *Object Radiobox*. The other one, which has the title "Mode," allows the user to select the current system mode which may be "input," "move," "delete," or "print." It is called the *System Mode Radiobox*. Beneath the radioboxes are command buttons for viewing the *roadmap* (discussed below) and refreshing the screen. The graphical schema itself is constructed in the DrawingArea widget (the *canvas*) on the right-hand side of the interface. As we can

see, the canvas is equipped with horizontal and vertical scrollbars that are used when working with large schemata.

As with most software systems built on top of Xt, OOdini primarily relies on the mouse for interaction with the user. The keyboard is required occasionally in response to a dialog box in order to input textual data, such as the name of a class. Most of the interaction with OOdini occurs in the DrawingArea widget where the graphical schema is constructed. The interaction during schema creation follows a regular pattern: The user first puts OOdini in input mode by selecting "input" in the System Mode Radiobox. Next, the user selects the desired schema construct, such as a class or a relationship, from the Object Radiobox widget and then proceeds to add any number of instances of that symbol to the schema. The techniques for inputting the different symbols are described in [111, 128, 183]. When finished with this "current" symbol, the user may choose another from the Object Radiobox and further expand the schema by inputting instances of this new symbol type. This continues until the schema is complete, at which time the user can request that it be saved to disk or printed. If the user wishes to modify the schema, say, by moving or deleting an instance of a schema object, then he selects one of the alternative system modes "move" or "delete" from the System Mode Radiobox. We emphasize that all modifications to the schema are constraint-based. For example, the movement of a class always entails the movement of all its associated graphical symbols (e.g,

attributes, relationships, set classes, and so on). Likewise, the deletion of a class propagates into the deletion of those associated symbols.

As alluded to above. OOdini manages a large drawing canvas, allowing the database designer to create extremely large schemata. This is a very important characteristic of the system because OODBs typically comprise many hundreds of classes. Our university database [34, 120] includes about two hundred and fifty. A schema editor which provides only a single "sheet" on which to draw a schema becomes totally worthless for such applications. Horizontal and vertical scrollbars are provided to allow the user to reposition the current working window (in the ordinary graphics sense) of the canvas. Using the scrollbars, the user can readily pan left and right, or up and down through the schema.

While it is possible for the user to quickly navigate to and view any portion of the canvas, the current working area presents only a small fraction of the entire schema. It is normally not possible to display a schema of substantial size in its entirety with a reasonable magnification. To give the user the ability to view the schema globally, we provide a mechanism which we call a *roadmap*. The roadmap is a special kind of dialog box which serves the following two purposes:

1. It provides a global schema view, that is, a reduced view of the schema showing all its elements at one time.

2. It provides a means for repositioning the current working area of the canvas. The repositioning is accomplished by moving a "focus rectangle" (i.e., a rectangle representing the current working window) with the mouse.

The second feature of the roadmap is particularly useful when it comes to rapidly moving between distant regions of the schema. Of course, the scrollbars could be used for this same purpose, but they can be tedious; the destination area is not in full view, and it is likely that the user will end up "oscillating" about that desired region during the search. In general, we have found that the scrollbars are used to make fine positional adjustments to the current working region, while the roadmap is employed for large jumps.

The "File" entry in the menu bar drops down a menu giving the user access to a number of disk storage and retrieval commands. Among these are ordinary "New," "Open," "Save," and "Save As" commands. (See [128, 152] for a description of their operation.) Also included is the command "Print Screen" which can be used to print the canvas's entire current working area to disk as a figure in a special X Window format. System utilities then allow for the conversion of the figure into a variety of formats for printing on different kinds of laser printers. Furthermore, through this mechanism, figures may be converted into a special PostScript format for use with the LaTeX macro package Psfig. In this way, the schemata created with OOdini can be easily incorporated into a LaTeX document. This can be an extremely handy feature for researchers and database system annotators. As a complementary

feature, by putting OOdini in "print" mode (using the System Mode Radiobox), one can select a sub-portion of the schema for storage in the X Window format. Thus, one can extract fine details of a schema for inclusion in documents. The File menu also contains command buttons for storing the schema in the various textual formats. We discuss these in the next section.

The "Edit" menu simply provides the user with a command "Clear" for clearing all the components of the current schema. The "View" menu offers two commands: "Level" lets the user alter the schema's current level of display (see Section 2.6 in the previous chapter), and "Search For" seeks out a schema component with a given name and repositions the canvas's current working window around it.

Invoking the command "Level" pops up a dialog box comprising a slider which ranges in value from 1 to 3, corresponding to the three levels of display that we defined in the last chapter. To select a level, the user simply adjusts the slider to the desired value and then presses "OK." The schema is then shown according to the rule governing the chosen level. For example, by setting the slider value to "2" and pressing "OK," the schema is shown in Level 2, meaning that attributes are no longer visible. Choosing "3" leaves only classes and generic relationships on the screen. A choice of "1" restores the schema back to the full-blown representation. By exploiting these different levels of display, OOdini can serve as an effective schema browser or OODB orientation device. To illustrate the effect of the different levels,

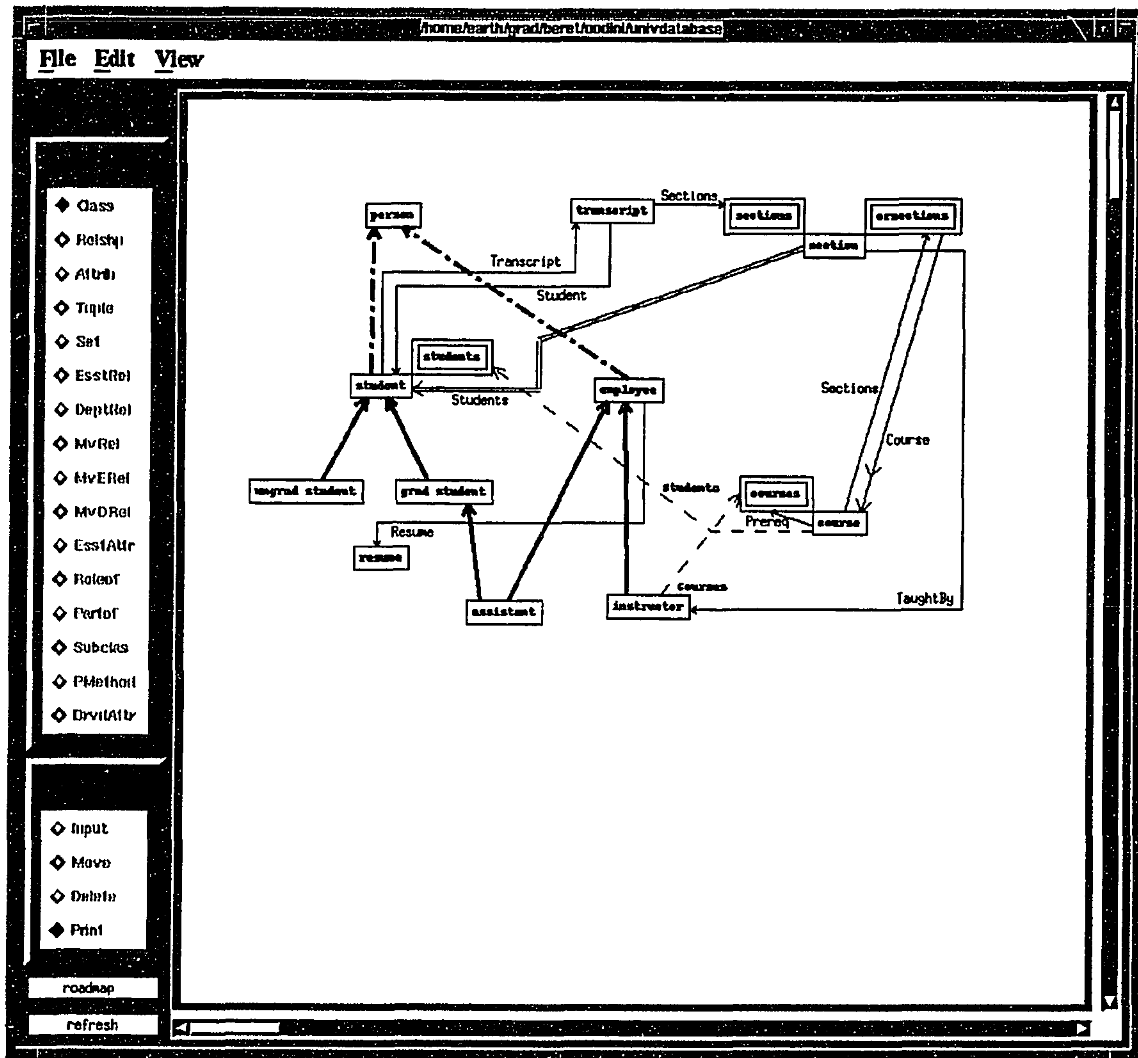


Figure 3.5 OOdini in a Level 2 display

we show the schema of Figure 3.4 (which is in a Level 1 display) in Level 2 in Figure 3.5 and Level 3 in Figure 3.6.

After choosing "Search For," the user is presented with a pop-up dialog box comprising two main components. The first, a so-called *option menu* [152, 151], displays the type of schema object to search for. (By default, the system looks for classes). The other, a *selection box*, presents a scrolled list of the names of all components of the selected type that currently appear in the schema. Again, the list comprises only components of a single type (e.g., classes). If one wanted to see the names of, say, all relationships, then one would choose "Relationship" in the option menu. Selecting one of these names from the list causes the current working window to be repositioned in such a way that the chosen object is at its center. Therefore, as with the roadmap, this command gives the user yet another method of repositioning the current working window. It, too, can aid both browsers and designers in their navigation through the schema.

3.3 OOdini Code Conversion

The "File" pull-down menu contains three command buttons which can be used to save OOdini's graphical schema in various textual formats. The first of these is an abstract textual language which we have devised called OODAL (from OOdini Abstract Language) [35]. The entire BNF specification for OODAL syntax can be found in [35]. The constructs of OODAL mirror those of the graphical schema language and, in fact, there is a one-to-one correspondence between the two. OODAL

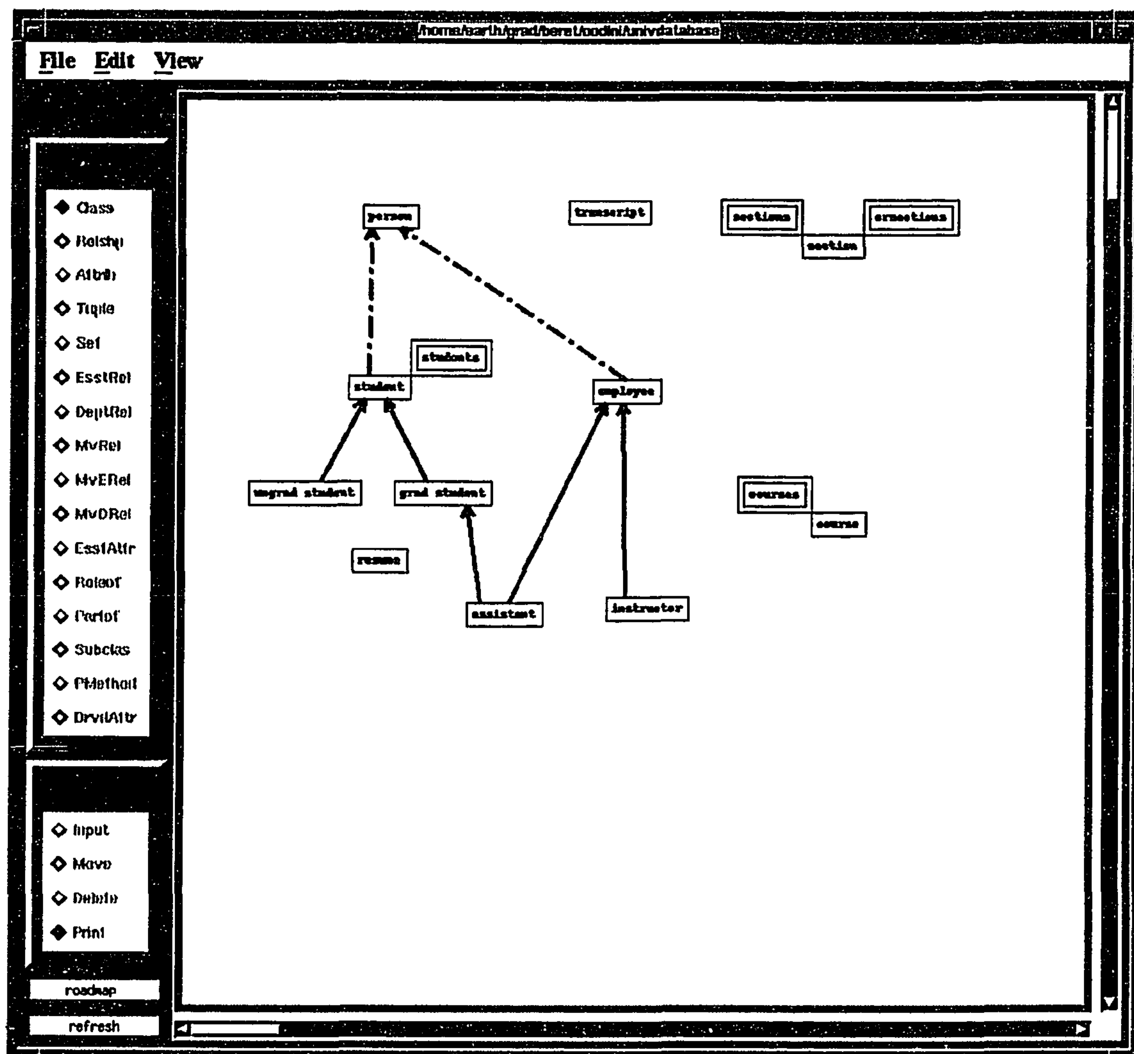


Figure 3.6 OOdini in a Level 3 display

was created primarily as an intermediate stage between the graphical representation and bona fide OODB data definition languages, and its syntax was constructed in such a way as to make this readily attainable. As it happens, we use OODAL in the conversion process from the graphical schema to VML code. The VML code converter is also available as a command in the "File" menu. The conversion into an OODB language through OODAL is illustrated in Figure 3.7.

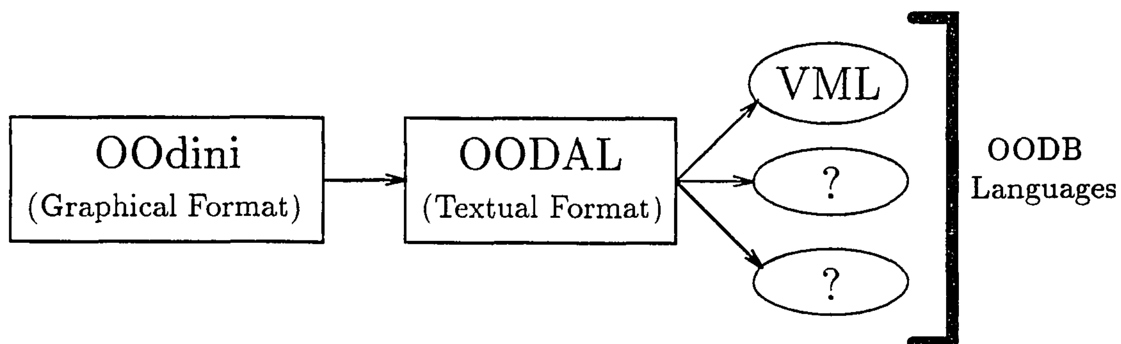


Figure 3.7 Conversion from OOdini into an OODB language

Because the Dual Model [65, 143] has played a central role in much of the modeling work carried out in our research group, we have included the means to convert an OOdini schema into its syntax. A command to carry this out is found under the "File" menu. The Dual Model's distinction between structure and semantics [69] requires the bifurcation of the OOdini description of a class and its properties into separate class and object type representations. (Actually, the same is true for VML which also employs a kind of Dual Model; the details of VML's data model are covered in Chapter 6.) We refer to the Dual Model's target syntax as DAL which

is short for Dual Model Abstract Language. Again, a complete BNF account of its syntax can be found in [35].

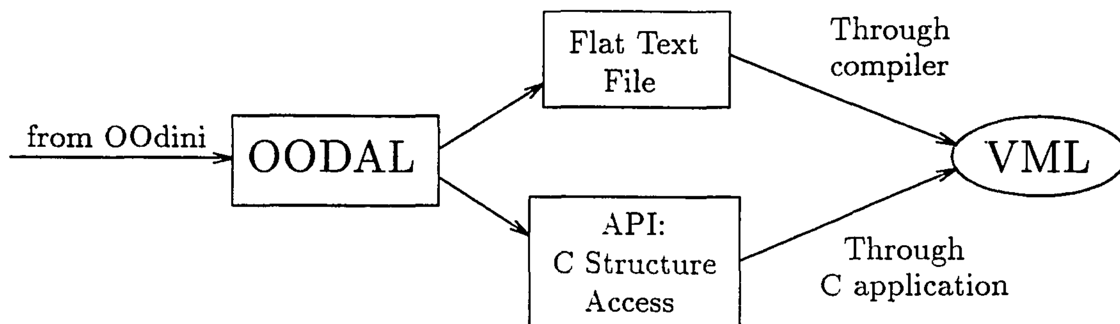


Figure 3.8 Alternative conversion paths from OODAL into OODB language

OODAL code generated by OOdini is normally stored in a "flat" text file. To convert this text file into the syntax of some OODB language, a compiler must be constructed [35]. To circumvent this requirement and facilitate the development of OODAL converters for other OODB languages, we provide an application programming interface (API) to OODAL as a C Language library and header file. This API permits direct access to a C language graph structure containing the OODAL representation of a given schema. This structure can be easily traversed at the programming level tó generate code for various OODB data definition languages. In fact, our VML converter was constructed in this manner. Use of the API is described thoroughly in [35]. The two alternative conversion paths from the graphical schema of OOdini through OODAL are illustrated in Figure 3.8.

CHAPTER 4

AN OODB PART RELATIONSHIP

In this chapter, we introduce the notion of a part relationship in the context of OODBs. At first, we consider previous work on part relationships in the context of OODB systems. Next, we go on to describe what we call a "generic" part relationship for connecting two object classes as holonymic and meronymic classes. After that, we discuss the different characteristic dimensions of the part relationship which capture the various semantics and functionalities of part-whole modeling. In the process of describing the formal aspects of the different dimensions, we also present a graphical schema notation for the part relationship to be used in the construction of OODB part schemata. At the same time, we provide a realization for the part relationship using the basic modeling constructs of the OODB model as described in Chapter 2.

4.1 Part Relationships in OODBs

Among existing OODBs, ORION [15] incorporates a part model [107] which distinguishes between four kinds of part relationships, derived by imposing exclusiveness and/or dependency on *weak* references [or what we call user-defined relationships (Chapter 2)]. In the ORION model, the exclusiveness constraint imposes a "part" reference restriction on the entire database topology. For example, if an engine is part of some car, then an exclusiveness constraint would require that it not be part of another car or any other type of object (e.g., a plane) at the same time. We have

found that such a restriction is too broad in certain circumstances, and we refine it by allowing exclusiveness to be imposed on a single class and relaxed otherwise. This refinement leads to two types of exclusiveness, *global exclusiveness* and *class exclusiveness*, whose formalizations follow very closely that offered in [107]. As a refinement to ORION's sharing, we also allow for the specification of a range restriction on the number of holonyms of a given type which may contain a certain meronym.

ORION also allows a part to be made dependent on the existence of its whole: If the whole is deleted, then the part is deleted automatically. Such a feature is useful in alleviating the burden of having to search out and manually delete parts, particularly in cases where the whole is a very complex object like a CAD drawing. A whole, however, is sometimes barely more substantial than one of its defining parts, as in the case of a bicycle and its frame. (Cf. the notion of "ontological dependency" discussed in Chapter 1.) In such circumstances, it may be desirable to define the whole as dependent on its part. For this reason, our part relationship permits the specification of dependency in either of the two directions, from the part to the whole, or vice versa.

The SHOOD model [145], an object model couched in a knowledge representation framework, incorporates some of the exclusiveness and dependency semantics of the ORION model. In addition, it addresses the issue of value propagation or what the developers call attribute propagation [145], which is the assimilation by holonyms

of data values from meronyms. (In extensional mereology, the same phenomenon is referred to as *local predication* [187].) As an example of this process, the color of a car as a whole may be obtained from the color of the body which is its part. In SHOOD, attribute propagation is limited to the upward direction, from parts to wholes. Moreover, the whole treatment of the issue is informal. In the next chapter, we formalize the notions of both upward and downward value propagation and their attendant derived schema components, called *derived attributes.* We also expand the notion of value propagation to encompass many part relationships simultaneously and more general computations with respect to propagated values. This leads to a powerful mechanism for the definition of derived attributes, permitting the specification in the database schema of expressions such as "the weight of a car is the sum of the weights of its parts, regardless of their classes."

The SORAC data model [127], another object model built within a knowledge-based environment and influenced by the semantic data models [155], includes a generic *is-part-of* as one of its set of core semantic relationships. In a schema, an actual part relationship (such as the "has-part" relationship of the design-support system ArchObjects [125] which is being built on top of SORAC) is constructed using underlying semantic relationships like *is-part-of*, *collection*, and *derivation*. These relationships are, in turn, constructed from a "menu" of semantic options (cf. [103]), which are described as update rules [156] affecting the creation and deletion semantics of the various related objects. In all this, there is an assumption of an underlying

rule manager [196]: we make no such assumptions about rule handling in our base OODB model. While the *is-part-of* relationship allows cardinality constraints on the number of wholes per part, it omits the global exclusiveness semantics of ORION. It does offer a variety of warning, error, and blocking semantics, where an outside agent can be alerted to violations of certain integrity constraints. In the formal description of our part model, we omit such considerations. Decisions on them are deferred to an actual implementation, such as the one using metaclasses which we present in Chapter 6. This issue will be brought up again at the end of this chapter and in the next. SORAC's derivation relationship, which allows for the definition of derived attributes, fails to capture any of the formal aspects of the derived computation. Instead, it addresses the issue of late versus early evaluation [196], which is more an implementation detail than a conceptual modeling concern.

The Object Modeling Technique (OMT) [171] and the conceptual object-oriented model (COOM) of [16] also approach the part relationship as a higher-level structure composed from a set of semantic options, as suggested previously in [103]. Both present a graphical notation for the various guises of their informal part relationships. Actually, OMT is a general object-oriented design or CASE methodology rather than just an OODB schema notation. In fact, a CASE tool called OMTool [171] based on OMT is now available from General Electric, where OMT was developed. This program automatically generates a software system in C++ code from a description in the OMT graphical notation. As for conceptual modeling, OMT permits all

the characteristics of ordinary associations (such as cardinality constraints) to be overlaid on its part relationship. However, it too fails to consider the exclusiveness and dependency semantics presented in [107]. As for the notation itself, we will draw on it as a basis for our own palette of part relationship symbols introduced below. The reasons for this will be discussed there.

Coad and Yourdon [39] also introduce a CASE methodology for object-oriented software development which includes a part-whole relationship and a graphical design notation. Their symbol for the part relationship is a directed line from the meronymic class to the holonymic class. As we will discuss below, we feel this is an inadequate notation for part-whole modeling. As with OMT, the part relationship can be loaded with the same constraints as ordinary relationships (e.g., with cardinality constraints). Interestingly, the technique encourages the use of part-whole in three different senses which are *not* distinguished graphically: (1) Assembly/Parts, (2) Container/Contents, and (3) Collection/Members. While (1) and (3) have corresponding senses among those of Winston, Chaffin, and Herrmann [210] discussed earlier, (2) seems somewhat dubious. Instead of being synonymous with sense (3) [or even sense (1)], as it apparently is, (2) is distinguished from it with an example that describes a pilot as part of the aircraft he is flying because he is inside [39, page 94]!

4.2 A Generic Part Relationship

In this section, we present a formal definition of a "generic" part relationship between a pair of OODB classes. This relationship is described formally as a quintuple comprising a relation between the extensions of the participating classes, and four "characteristic" dimensions: (1) *exclusiveness/sharing*, (2) *cardinality/ordinality*, (3) *dependency*, and (4) *value propagation*. The first of these addresses the issue of how parts may be distributed among wholes. The next is concerned with the way parts of the same kind are collected together to form wholes. The third dimension deals with the dependency semantics, i.e., how the deletion of a holonym or meronym affects its counterpart in a part-whole arrangement. The final dimension addresses the issue of propagating relevant data across the part relationship from the whole to the part, or vice versa, leading to the definition of derived attributes. A discussion of this latter dimension is deferred until Chapter 5.

We use the term "generic" in this context because the formal definition is really a template for the actual part relationships. Portions of the quintuple may be refined into nested structures in order to accommodate other defining elements implied by certain values of the characteristic dimensions. There are also interactions among dimensions, where a given value in one dimension precludes some values in others. These, too, are not captured explicitly in the description of the generic relationship, but are elucidated as we consider each dimension in turn.

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

$$P_{B,A} = \langle \diamond, \chi, \kappa, \delta, \nu \rangle \tag{4.1}$$

where $\diamond$ is a relation from $E(B)$ to $E(A)$. The pair $(b, a) \in \diamond$ indicates that the instance b of class B *is part of* the instance a of class A. We will ordinarily express this fact in an infix expression as $b \diamond a$. At times, the relation $\diamond$ may carry a subscript to distinguish between multiple part relationships. For example, if we have a part relationship $P_{B_1,A}$ between the classes B_1 and A and another part relationship $P_{B_2,A}$ between the classes B_2 and A, then we would write their constituent relations as $\diamond_1$ and $\diamond_2$.

The remainder of the quintuple represents the values of the four characteristic dimensions which have the following domains:

$$\begin{aligned} \chi \in X &= \{\textit{global-exclusive, class-exclusive, limited-shared}\}, \\ \kappa \in C &= \{\textit{range-restricted, ordered-definite, ordered-indefinite}\}, \\ \delta \in D &= \{\textit{part-to-whole, whole-to-part, nil}\}, \\ \nu \in V &= \{\textit{up, down, upTrans, downTrans, up\&down, nil}\}. \end{aligned} \tag{4.2}$$

Note that the values of both δ and ν may be *nil*, indicating that the particular characteristic (either dependency or value propagation) is inapplicable. Complete accounts

of each dimension will be given in subsequent sections, where formal descriptions of each dimension are provided. To accomplish this, we will need the following two definitions. Assume that there exists a part relationship $P_{B,A}$.

Definition 1: $\forall a \in E(A)$, let $M_{\diamond}(a) = \{b \mid b \in E(B) \wedge b \diamond a\}$. $M_{\diamond}(a)$ is called the *meronym set* of a with respect to the part relationship $P_{B,A}$, i.e., the set of instances of B which are parts of a.

Definition 2: $\forall b \in E(B)$, let $H_{\diamond}(b) = \{a \mid a \in E(A) \wedge b \diamond a\}$. $H_{\diamond}(b)$ is called the *holonym set* of b with respect to the part relationship $P_{B,A}$, i.e., the set of instances of A of which b is a part.

In Chapter 2, we presented a graphical schema representation for OODBs that encompasses a wide range of schema constructs. We now begin to enhance that notation with a basic graphical symbol for the part relationship. This symbol serves as a basis for a variety of symbols which denote the part relationship in its various guises. In the course of this chapter and the next, we further augment the symbol to mnemonically capture the part relationship's rich semantics and functionalities.

The graphical symbol for the part relationship is a bold, dashed line connecting the meronymic and holonymic classes. A diamond head at one end of the line indicates the holonymic class. Figure 4.1 shows a part relationship between classes B and A. (As we will see much later, this symbol actually represents a shared, single-valued part relationship with neither dependency nor value propagation.)

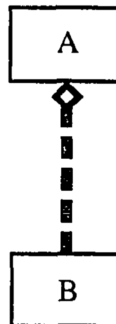


Figure 4.1 A part relationship between meronymic class B and holonymic class A

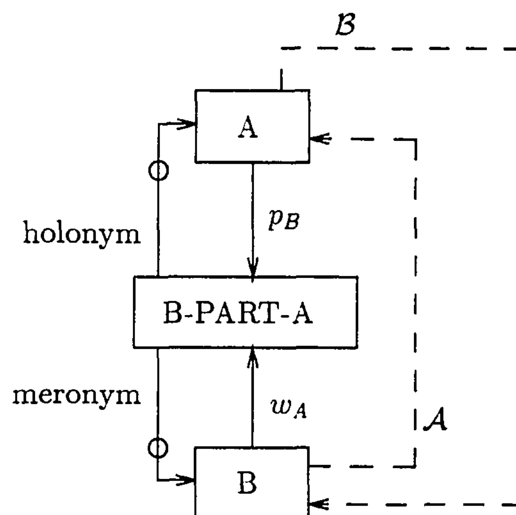


Figure 4.2 The generic realization of the part relationship

A goal in the design of our graphical schema language has been to exploit the mnemonic nature of the graphical icon. The dashes of the part relationship's symbol have been used expressly for this purpose. Here, the mnemonic device is the association between the pieces of the line and the parts of the object. Just as a whole object is decomposed into individual part objects, so too is the part relationship's line broken into constituent segments. The choice of the bold aspect of the line was influenced by the desire to highlight the connection in the overall context of

the schema and contrast part relationships with "ordinary," user-defined relationships, which are represented with thin lines. This contrast is important because it makes the hierarchy produced by part-whole relationships clearly recognizable as a backbone of the graphical schema, just as with the graphical representation of the specialization (IS-A) hierarchy where bolds lines were used for emphasis. As such, it helps to promote an intuitive understanding of the application. The final aspect of the part relationship symbol, namely, its diamond head, maintains consistency with the OMT notation [171], and is employed, instead of the customary arrowhead, to avoid the impression of directedness which would belie the fact that our part relationship constitutes a powerful two-way access and constraint mechanism. Nevertheless, there is a need to distinguish the parts from the wholes, and the location of the diamond head near the holonymic class serves this purpose.

The realization of the part relationship is based on its expansion into an object class in its own right along with a set of connecting relationships and methods. Initially, we present the realization of a part relationship devoid of any of the semantics embodied by the different characteristic dimensions. We will refer to this as the "generic realization," and, as with the graphical schema symbol, its augmentation will allow us to realize all the desired semantics.

The generic realization for the part relationship between meronymic class B and holonymic class A can be seen in Figure 4.2. There, we can see the four aspects of this realization: (1) a class `B-PART-A`, (2) a pair of outgoing, essential relationships,

holonym and *meronym*, from this new class, (3) two incoming relationships, p_B and w_A, to the new class from the holonymic and meronymic classes, respectively, and (4) a pair of path methods, $\mathcal{A}$ and $\mathcal{B}$, defined on the meronymic (holonymic) class and referring to the holonymic (meronymic) class.

The instances of B-PART-A are the occurrences of the part relationship $P_{B,A}$ (i.e., elements of $\diamond$), and its two outgoing relationships, *holonym* and *meronym*, represent the projections of $\diamond$ onto the holonymic and meronymic classes, respectively. In other words, for each pair $(b, a) \in \diamond$, there exists an instance o, whose relationships *meronym* and *holonym* have values b and a, respectively. We will write $o(b, a)$ when referring to this instance. This approach of modeling a "second order" object as a "first order" object follows in the tradition of the ER and other semantic models and has been suggested by previous work in the object-oriented community (e.g., [8, 49, 108, 127, 170]). Semantic relationships realized in this manner have been referred to as "fat links" [55].

The relationships *holonym* and *meronym* are defined as essential (total) as a consequence of our extensional approach of relating actual instances of the participating classes (cf. [103]), as opposed to an intensional or type-based approach. An instance of B-PART-A cannot exist without the existence of those instances of the holonymic and meronymic classes that it serves to relate as whole and part. By making the two properties essential, we ensure that any transaction which creates an instance of B-PART-A (i.e., some "make part of" transaction) also assigns these properties values

that are valid references to *existing* objects of the referent classes. It also implies that the deletion of a meronym or holonym is disallowed until its part connection is broken by the deletion of any relational elements of the class B-PART-A it may participate in. (The dependency characteristic can alter this such that deletion of a part or whole propagates into the automatic deletion of the relational elements. This is discussed below.)

The relationships p_B and w_A are added to the holonymic and meronymic classes, respectively, and are used for two purposes. First, they aid in the imposition of some of the constraints defined by the various characteristic dimensions of the part relationship. Second, each constitutes a portion of the "bridges" between the part and whole. These bridges, the path methods $\mathcal{B}$ and $\mathcal{A}$, provide the means by which a whole can access its part, and a part can access its whole, and in this respect, play a role in our value propagation mechanism. In our "path" notation, the method $\mathcal{B}$ for retrieving a part B from its whole A is defined as:

$$\mathcal{B}() : p_B \rightarrow \texttt{B-PART-A} : \textit{meronym} \rightarrow B.$$

In other words, $\mathcal{B}$ first traverses the relationship p_B to reach the class B-PART-A. From there, it crosses *meronym* and arrives at its destination, the class B. Likewise, the method $\mathcal{A}$ of class B which acts as a selector for a whole within a part is defined as follows:

$$\mathcal{A}() : w_A \rightarrow \texttt{B-PART-A} : \textit{holonym} \rightarrow A.$$

This traversal proceeds through w_A to `B-PART-A`, after which it crosses *holonym* to reach A.

We note that the selector methods $\mathcal{A}$ and $\mathcal{B}$ are defined as multivalued when there are many wholes per part and many parts per whole, respectively. The traversals in such cases are actually iterative, and the method bodies are written slightly differently (as discussed in Chapter 2) to reflect this.

4.3 The Exclusive/Shared Dimension

The exclusive/shared dimension of the part relationship $P_{B,A}$ regulates the way that meronyms may be distributed among different holonyms. In particular, each value from its domain

$$X = \{\textit{global-exclusive},\ \textit{class-exclusive},\ \textit{limited-shared}\}$$

imposes a different set of constraints on the cardinalities of the holonym sets of the instances of B. Before getting to the formal specifications of these constraints, let us first consider the ways in which we would like to distribute parts among wholes and see how this leads to our three-way distinction.

Part relationships in general can be divided along the lines of exclusive and shared [107, 145]. Exclusiveness refers to the constraint that the ownership of a meronym be restricted to a single holonym. In other words, a meronym may be part of only one holonym. Exclusiveness represents perhaps the most intuitive constraint

which may be imposed on objects in 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" [39]. Two boats, naturally, cannot share the same muffler.

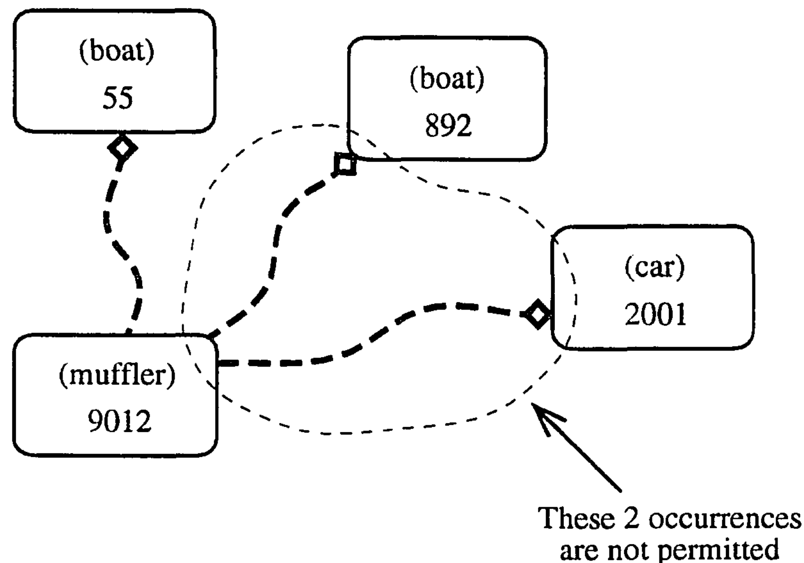


Figure 4.3 A muffler owned exclusively by a boat

Beyond that fact, it is also the case that a boat and car cannot share a muffler either. Hence, in a database (as illustrated in Figure 4.3), the exclusive constraint on the part relationship between the classes `boat` and `muffler` must have implications on the part relationship between `car` and `muffler`, and, in fact, on any part relationship that `muffler` participates in as the meronymic class. Therefore, to be more precise, our original statement of exclusiveness must be revised to read: a meronym may be part of one and only one holonym, *regardless of the holonym's class.*

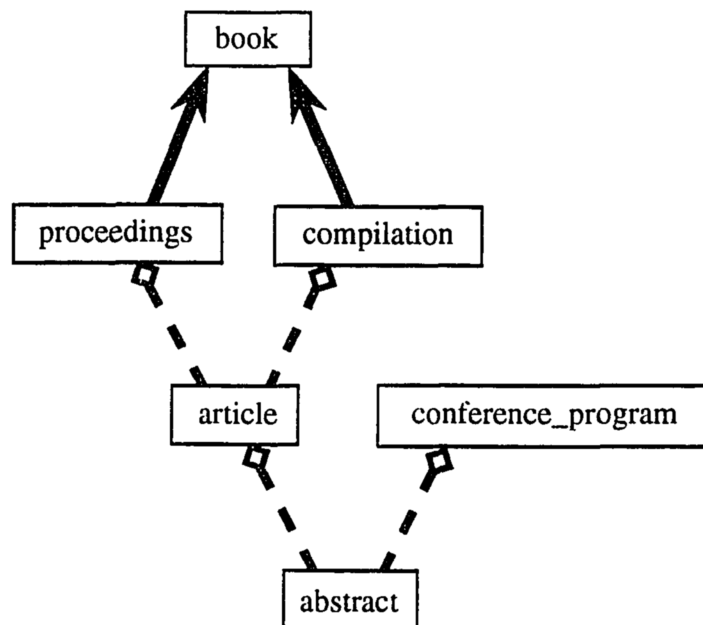


Figure 4.4 A document database schema

While this constraint is valid for physical assemblies, there are other situations where it is too rigid. For instance, consider a document database maintained for all the publications of an organization which sponsors technical conferences and workshops. A partial schema for such a database is shown in Figure 4.4 where there appear six classes, `proceedings`, `article`, `abstract`, `conference_program`, `book`,

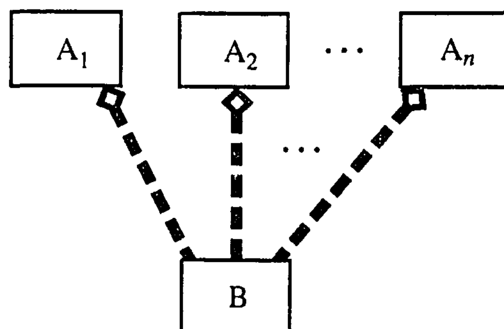


Figure 4.5 Class B in n part relationships

and `compilation`. Part relationships in this schema are indicated by the generic symbol. We see that a proceedings has articles as parts, and an article, in turn, has an abstract as a part. (Note that the multiplicity of the part relationship between `proceedings` and `article` is not made explicit in this diagram. This issue will be addressed later.) Furthermore, it is obvious that two articles cannot share the same abstract, so an exclusive reference constraint should be imposed on the relationship between their respective classes. However, a conference program might contain the abstracts of 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. While an article cannot appear in more than one conference proceedings, it is often the case that selected articles from conferences are reprinted in compilations (i.e., books comprising articles of related interest). Again, even though a given article may only appear in a single proceedings, it can also be part of some compilation.

As another example, consider the case of a musical document publication OODB system. In music publication, the score for an orchestral composition is typically available in two formats, the full (or ensemble) score and individual instrument scores. The "staff object" representing the music to be played, say, by the first violin section in Beethoven's Ninth Symphony can be modeled as part of both an

ensemble score object and an instrument score object. However, it cannot be part of more than one ensemble score because different musical compositions do not have identical music played by the same instrument. For example, the music for the first violin section is not the same for Shostakovich's Ninth as it is for Beethoven's Ninth (Figure 4.6). Thus, we do want an exclusive ownership restriction enforced by the part relationship between the classes `ensemble_score` and `staff`; but we do not want this part relationship to impose this constraint on other part relationships (in particular, the one between `instrument_score` and `staff`). We illustrate this point in Figure 4.6.

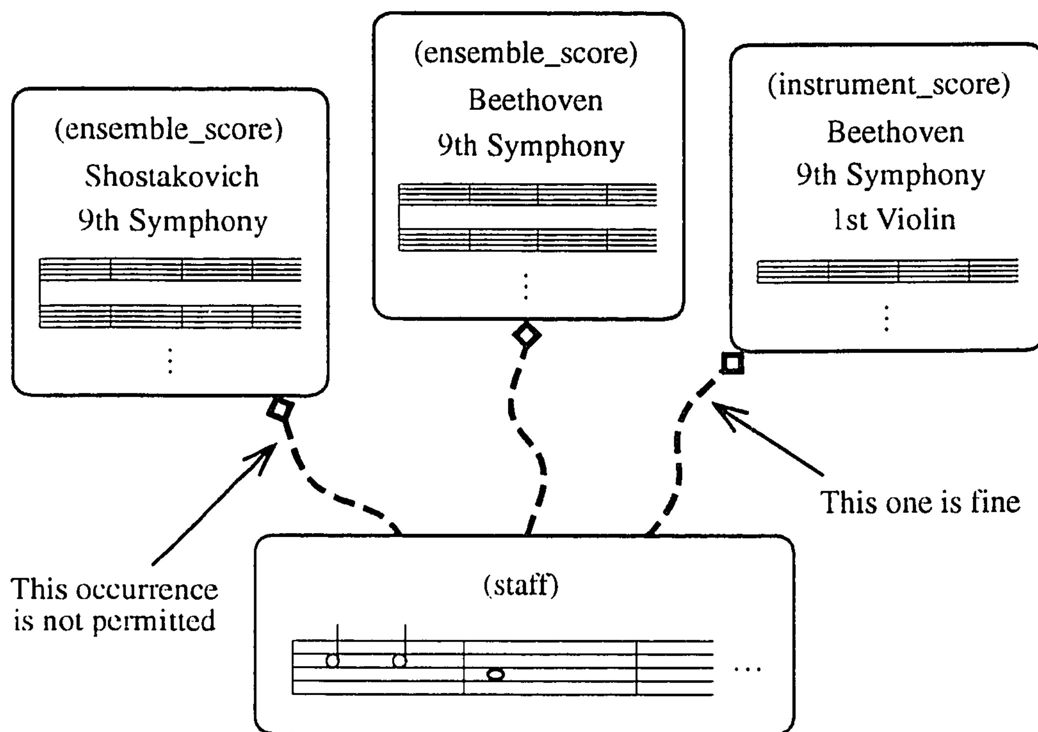


Figure 4.6 Part relationships in a music publication database

Score Expression Seq. *Allegro* *accel.* *rit.*
Violin
Viola
Cello

Figure 4.7 An ensemble score and its score expression sequence

In this modeling environment, we find that the same constraints are required for the part relationship between the classes `ensemble_score` and `score_expression_sequence` (Figure 4.7). A score expression sequence is the line above the staff of a musical score containing annotations such as "Allegro" (e.g., see the musical excerpt in Figure 4.7). As tempo markings and other such performance notation vary from score to score, a score expression sequence object will always be part of only a single ensemble score. In contrast, the same score expression sequence will always constitute a part of all instrument scores associated with a particular ensemble score. Thus, once again we would like to impose the exclusive ownership constraint on the part relationship between `ensemble_score` and `score_expression_sequence` and not have it affect that between the classes `instrument_score` and `score_expression_sequence` (which, itself, happens to be shared). In this situation, and in those discussed above, the exclusive ownership restriction is inappropriate.

In all these scenarios, what is required is the imposition of the exclusivity constraint with a single holonymic class (e.g., on `article` in relationship to `abstract`) and its relaxation with regard to other classes. For this reason, we distinguish between two types of exclusiveness:

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

Those part relationships which are neither global exclusive nor class-exclusive are referred to as *shared*, in which case a meronym may be shared freely among any number of holonyms. Such is the case for the part relationship between `article` and `compilation`, where an article may appear in any number of compilations.

Actually, the example of sharing just mentioned is a special case of the more general *limited-sharing* supported by our part relationship. Such a part relationship enforces a range constraint on the number of holonyms that a part may be owned by. For example, we may wish to restrict the number of compilations that an article can appear in to, say, no more than three in order to satisfy some licensing agreement. Or if we choose to model students as parts of the sections that they are registered for, then we may want to require that a student be registered for at most six sections in any one semester. Likewise, the navy may model its forces in such a way that a battleship is part of between one and two fleets.

To formalize the different notions of exclusiveness and sharing, constraints are placed on the cardinalities of the holonym sets of the instances of the meronymic class. First, global exclusiveness: Assume that there exist n part relationships $P_{B,A_1}, P_{B,A_2}, \ldots, P_{B,A_n}$, as in Figure 4.5, with constituent relations $\diamond_1, \diamond_2, \ldots, \diamond_n$, respectively.

Definition 3: The part relationship P_{B,A_i} is *global exclusive* (i.e., $\chi =$ *global-exclusive*) if $\forall b \in E(B), \exists a \in E(A_i)$ such that $b \diamond_i a \Rightarrow |H_{\diamond_i}(b)| = 1 \wedge \forall j \neq i, |H_{\diamond_j}(b)| = 0$.

As we see, the global exclusive part relationship P_{B,A_i} not only places constraints on the holonym sets defined with respect to itself, but on all holonym sets defined with respect to the class B. Thus, its existence has ramifications on the entire database topology, placing limits on "part" references to instances of B from instances of its holonymic class A_i as well as from instances of all the other holonymic classes of B.

Definition 4: The part relationship $P_{B,A}$ is *class-exclusive* (i.e., $\chi =$ *class-exclusive*) if $\forall b \in E(B), |H_\diamond(b)| \leq 1$.

In other words, the part relationship between a meronymic class B and holonymic class A is class-exclusive if $\diamond$ is a partial function from $E(B)$ to $E(A)$ and the relationship does not impose the additional constraints of global exclusiveness. It should be noted that global and class exclusiveness are equivalent if the meronymic class participates in only a single part relationship.

To formally define the *limited-shared* part relationship, we will need to modify the quintuple into a more deeply nested one in order to include the additional information

(namely, the bounds of the range restriction). In the following definition, **N** denotes the natural numbers $\{0, 1, 2, \ldots\}$, and $\mathbf{Z}^+$ is the set of positive integers.

Definition 5: The part relationship $P_{B,A} = \langle \diamond, (\textit{limited-shared}, p, q), \kappa, \delta, \nu \rangle$, with $\diamond$, κ, δ, and ν defined as in (4.1) and (4.2), and $p \in \mathbf{N}$, $q \in \mathbf{Z}^+ \cup \{\infty\}$, and $p \leq q$, is called a *limited shared* part relationship and satisfies the following: $\forall b \in E(B)$, $p \leq |H_{\diamond}(b)| \leq q$. The condition $q = \infty$ means that the upper bound does not apply.

Theoretically, the symbolic value *limited-shared* appearing in the triple at the second position of the quintuple is extraneous. It is used strictly for the sake of readability to make the interpretation clear. Following the terminology of [107], we will refer to a limited-shared part relationship with a lower bound of 0 and an upper bound of ∞ (i.e., one with no cardinality constraints on the number of wholes per part) as simply *shared*.

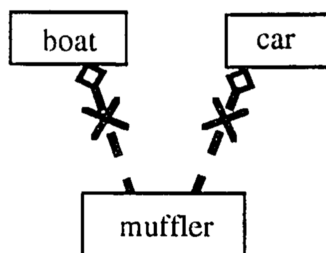


Figure 4.8 Global exclusive part relationships

We add an "X" to the generic graphical symbol to indicate that a part relationship is global eXclusive (Figure 4.8). An "X" inscribed in a rectangle adorns the generic symbol in the case of class eXclusiveness. As sharing implies the lack of any

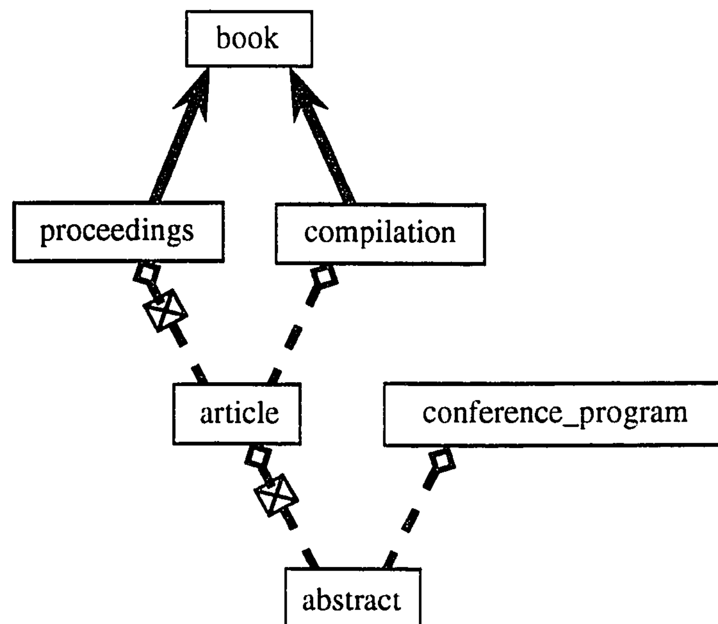


Figure 4.9 Revised document schema with class exclusiveness and sharing

exclusivity constraints, it is indicated by the lack of these two embellishments. Class-exclusive and shared relationships can be seen in the revised publication schema of Figure 4.9.

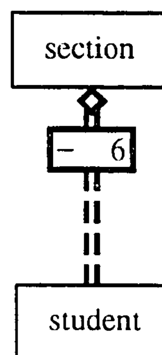


Figure 4.10 Students limited to at most six sections

In the case of a limited-shared part relationship, we modify the class-exclusive symbol slightly to incorporate the desired cardinality constraint. Instead of drawing

an "X" inside the rectangle, we write the lower bound in its left half and the upper bound in its right. Either of the bounds may be omitted by writing "–" in place of a number. This representation was chosen because, as can be gathered from the definitions, class exclusiveness is a special case of limited sharing. To be consistent with the rest of our notation, if it is essential for a part to be in a whole (i.e., if the bounds are both 1), then we inscribe a circle inside the rectangle to indicate the essentiality. In Figure 4.10, we show that a student is limited to enrolling in at most six sections.

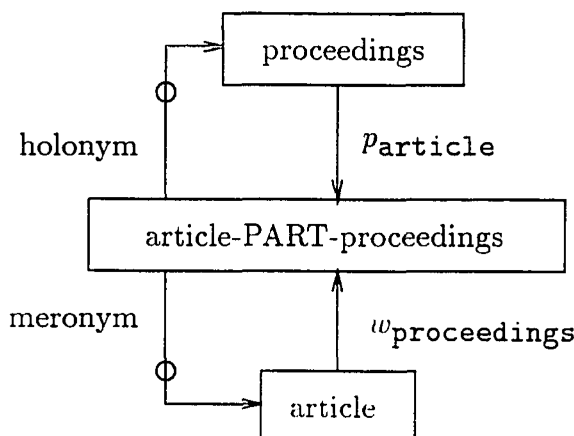


Figure 4.11 Realization of exclusiveness

The realization of the semantics of this characteristic dimension is based on the manipulation of the relationship w_A defined on the meronymic class in the generic realization. This manipulation encompasses two aspects. The first involves modifying the relationship's cardinality which was originally defined as single-valued. It remains that way for the two kinds of exclusiveness but is changed to multivalued for sharing. The second aspect revolves around the augmentation of the relationship's

writer method [100], using it as a *facet* in the sense of frame-based knowledge representation systems [55, 115, 123, 207]. In the language of that area, the facet monitors the value of the "slot" w_A to ensure its integrity. In particular, the facet is used to realize the insertion semantics or "make-component" rules [107] of exclusiveness and sharing.

The class-exclusive part relationship is realized with w_A single-valued as in the generic realization. Consider, for example, the realization of the part relationship (minus the path methods) between `article` and `proceedings`, shown in Figure 4.11. With $w_{\texttt{proceedings}}$ single-valued, an instance of `article` may be related to just one instance of `article-PART-proceedings`. This, in turn, implies that it can only be part of a single `proceedings`, which means that such a proceedings has exclusive ownership of the article with respect to all other proceedings. The configuration, though, in no way precludes an object of another class (e.g., a compilation) from also having the article as its part. Hence, the expansion as given seemingly captures the desired class-exclusive semantics. However, there is a subtlety which is not captured, namely, the constraint placed on this part relationship by any global exclusive part relationships that the meronymic class happens to participate in. By definition, a global exclusive part relationship affects the insertion semantics of class exclusiveness as follows: If a meronym is already part of some holonym with respect to a global exclusive relationship, then the meronym cannot be made part of another

holonym with respect to a class-exclusive relationship. A facet for w_A will enforce this additional constraint, as discussed below.

Global exclusiveness, in isolation, can be realized using the configuration for class exclusiveness because a meronym must still be part of at most one holonym. But, while by its very existence it imposes constraints on all other part relationships (involving the given meronymic class), a global exclusive part relationship is constrained by these in turn. Its definition does not call for it to override any previously established part connection for a given meronym involving another part relationship, be it shared, class-exclusive, or global exclusive. Therefore, before allowing the establishment of a part connection with respect to the global exclusive relationship, it must be verified that there exists no other part connection involving the meronym of interest. This precondition is monitored by the "global exclusive" facet, which will be considered shortly.

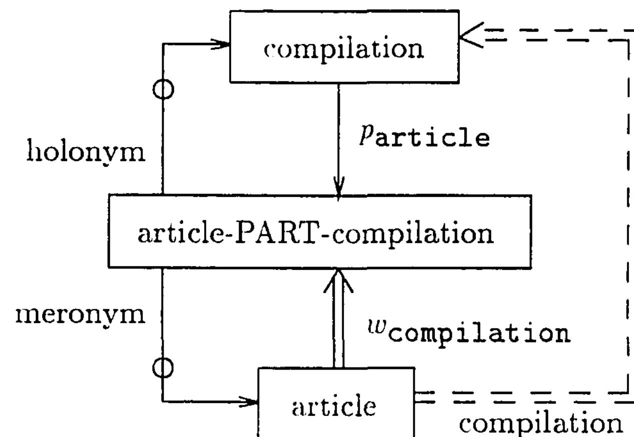


Figure 4.12 Realization of sharing

The configuration for a shared part relationship is obtained by making w_A multivalued, which implies that a meronym may be associated with many "part relationship" objects and, therefore, with many holonyms. The realization of the part relationship between `article` and `compilation` is given in Figure 4.12. (Recall that the dual line of the relationship $w_{\texttt{compilation}}$ indicates its multivaluedness.) As with class exclusiveness, the insertion semantics of a shared part relationship is affected by the existence of any global exclusive relationships. This, too, is captured by a facet for w_A, which is discussed below. Due to the fact that w_A is multivalued, the selector method for the holonym $\mathcal{A}$ is multivalued as well. In our path notation, it is written as:

$$\mathcal{A}() : w_A \rightarrow \{\texttt{B-PART-A}\} \ : \ @\mathit{holonym} \rightarrow \{A\}.$$

The initial traversal across w_A now yields a set of relational objects. After that, the relationship *holonym* is applied iteratively to this set, yielding a set of A, the holonyms of a given meronym. As an example, see Figure 4.12 again, where the selector method "compilation" is written as a dual, dashed line to indicate its multiplicity.

The realization of the limited-shared part relationship is similar to that for the shared, except that the relationship w_A now has a cardinality range restriction imposed on it. This can be seen in Figure 4.13 where students are required to be part of at least three and at most five sections.

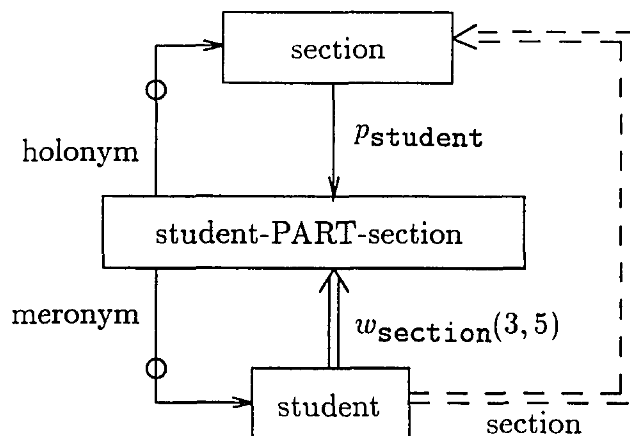


Figure 4.13 Realization of limited sharing

The different facets for w_A that we have been discussing essentially encode a variation of the "make-component" rule of [107] which here is supplemented by the class-exclusive/global exclusive distinction. Depending on the type of part relationship, either global exclusive, class-exclusive, or shared, one of three different facets replaces w_A's ordinary writer method. Each of these three defines a precondition that must be met before the assignment of w_A, and hence the establishment of a part relationship occurrence, can take place.

As its precondition, the global exclusive facet verifies that the target meronym does not belong to a holonym with respect to any other part relationship. To be more specific, assume once again that we have the schema configuration as in Figure 4.5, with meronymic class B in n different part relationships and P_{B,A_i} global exclusive. The precondition can then be stated as: For the target meronym, each w_{A_j} $(j \neq i)$ must either be *nil* when it is single-valued or the empty set $\emptyset$ when it is multivalued. The facet for w_{A_i} is specified in the following. Recall that w_{A_i} is single-valued for

a global exclusive part relationship and is assigned the OID of a single relational object [denoted $o(b, a)$] if the precondition is satisfied. (We use "$\leftarrow$" to denote the assignment operator.) Also note that we use the shorthand notation $\neg w_{A_j}$ for the above said condition, that is, w_{A_j} equal to *nil* or $\emptyset$, depending on its cardinality.

GLOBAL EXCLUSIVE FACET $w_{A_i}[o(b, a)]$

IF $\forall j : 1 \leq j \leq n, \neg w_{A_j}$

THEN $w_{A_i} \leftarrow o(b, a)$

ELSE Signal Failure.

The ELSE clause is used to indicate to the invoking "make-component" transaction that the necessary precondition was not satisfied and the part connection could not be established as requested.

In the case of class exclusiveness, the facet need only check those w_{A_j}'s associated with global exclusive part relationships for *nil*, as its precondition. This is because if the target meronym already participates in one of these, then it obviously may not participate in the desired class-exclusive relationship. However, as the meronym may participate freely in any other class-exclusive or shared part relationships simultaneously, these have no bearing on the assignment and can be ignored. So, assuming that $P_{B.A_k}$ (in Figure 4.5) is class-exclusive, its facet looks like:

CLASS EXCLUSIVE FACET $w_{A_k}[o(b,a)]$

IF $\forall j$ such that P_{B,A_j} is *global-exclusive*, $\neg w_{A_j}$

THEN $w_{A_k} \leftarrow o(b,a)$

ELSE Signal Failure.

The precondition of the facet for a limited-shared (and, likewise, a shared part relationship) is identical to that for class exclusiveness. But since w_A is a multivalued relationship (i.e., a set of OIDs), the assignment now involves the union of the facet's argument (taken as a singleton set) with the set of existing referents (i.e., w_A's current value). The facet for a shared P_{B,A_l} looks like:

SHARED FACET $w_{A_l}[o(b,a)]$

IF $\forall j$ such that P_{B,A_j} is *global-exclusive*, $\neg w_{A_j}$

THEN $w_{A_l} \leftarrow w_{A_l} \cup \{o(b,a)\}$

ELSE Signal Failure.

Of course, for the limited-shared part relationship, the range restriction on w_A must also be enforced. This is handled automatically by the range-restricted relationship in the schema of Figure 4.13, and we do not need to explicitly include it in our precondition.

4.4 The Cardinality/Ordinality Dimension

The next characteristic dimension of the part relationship deals with the ways in which parts from a single meronymic class can be grouped together to form wholes.

In our part model, there are three ways to do this as reflected in the domain of this dimension:

$$C = \{\mathit{range\text{-}restricted}, \mathit{ordered\text{-}definite}, \mathit{ordered\text{-}indefinite}\}.$$

Actually, the latter two are related, and so the model offers two distinct possibilities for the combination of parts:

1. As sets, possibly with cardinality restrictions.

2. As ordered lists where each part functions in some capacity, according to its position in the list.

The first of these is employed when the model does not call for any inherent ordering of the parts; the parts are collectively viewed as a set which may have explicit range restrictions imposed on its cardinality [21, 53, 103]. As an example, consider the model of a newsletter in a document database. A newsletter can, among other things, be viewed as a collection of articles of no particular order, whose layout is determined by an algorithm defined as a method for newsletters. As there is normally a strict page limit on such documents and a newsletter needs to contain articles on a variety of subjects in order to appeal to all its readers, we may want to hold the number of articles within some tight bounds such as five to six, as seen in Figure 4.14. The portion of the database state appearing on the right side of the figure represents

an integrity violation because the newsletter there has too few articles. That on the left side constitutes a valid state of the database.

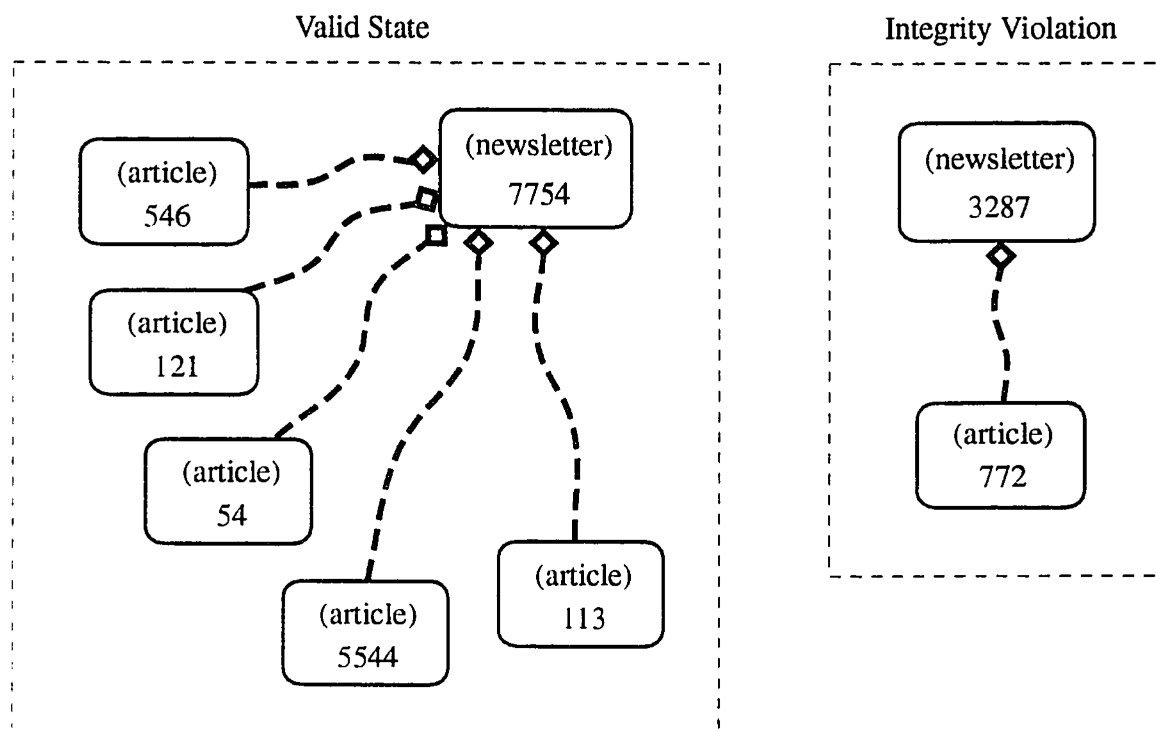


Figure 4.14 Newsletters in a document database

In many modeling situations, the parts from a given meronymic class function in different capacities within the holonym. For instance, the "text segments," which make up the minutes of a meeting, ordinarily assume three different capacities: the topics considered, an account of the general discussion, and the decisions arrived at. The model in Figure 4.15 showing an instance of minutes with a set of three text segments is inadequate because it fails to distinguish between these. The topics segment, discussion segment, and decisions segment cannot be distinguished from one another.

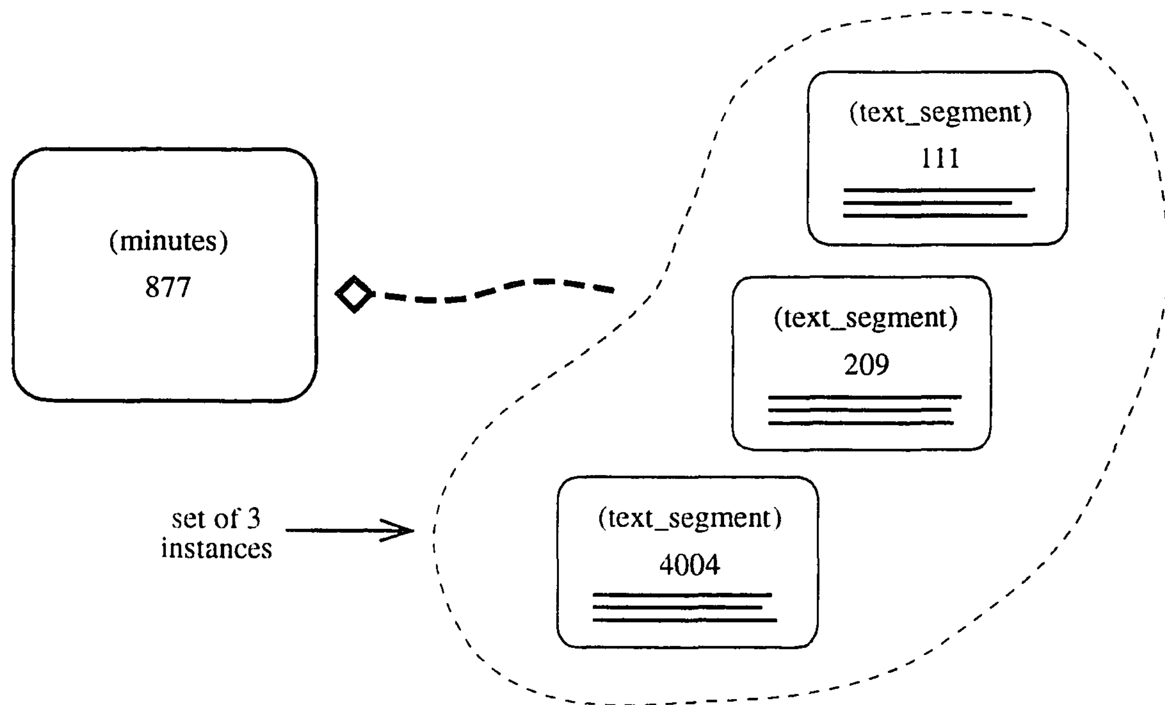


Figure 4.15 Inadequate model for the minutes of a meeting

To properly model the minutes, there must be an ordering or explicit naming of the capacities of its various text segments. This is illustrated in Figure 4.16, where the part link occurrences between instances have been labeled according to the capacity of the part within the whole. Because there are a fixed and definite number of parts of the given type, we refer to this case as an *ordering of definite number* (ODN).

An *ordering of indefinite number* (OIN) is used when the number of parts varies or cannot be determined *a priori*, i.e., at the time of class definition. Modeling the part relationship between an article and its sections is just such an application. An

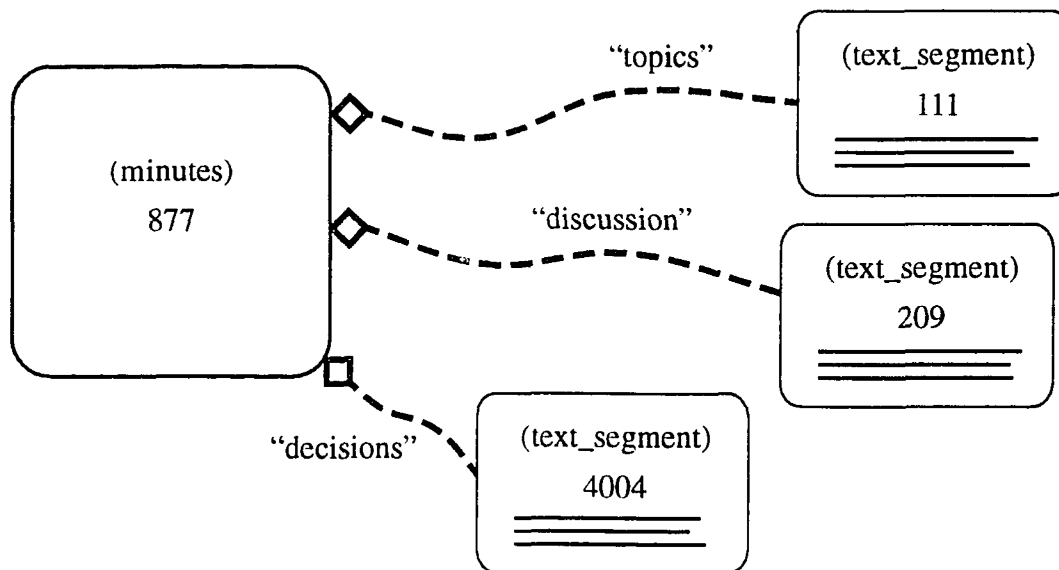


Figure 4.16 An adequate model for the minutes

article comprises an ordered sequence of sections, but the size of the sequence varies from article to article and cannot be fixed in advance.

As in the case of the limited shared part relationship, to formally define $P_{B,A}$ in this context, we will need to modify the quintuple into a more deeply nested structure to include additional information. The range-restricted version of the part relationship requires the addition of a numerical range, while ordering requires revisions to the base relation $\diamond$.

Definition 6: The part relationship $P_{B,A} = \langle \diamond, \setminus, (\textit{range-restricted}, m, n), \delta, \nu \rangle$, with $\diamond$, $\setminus$, δ, and ν defined as in (4.1) and (4.2), and $m \in \mathbf{N}$, $n \in \mathbf{Z}^+ \cup \{\infty\}$, and $m \leq n$, is called a *range-restricted* part relationship and satisfies the following: $\forall a \in E(A)$, $m \leq |M_\diamond(a)| \leq n$. The condition $n = \infty$ means that the upper bound does not apply.

Once again, the value *range-restricted* appearing in the triple at the third position of the quintuple is extraneous. It is retained to make the interpretation clear.

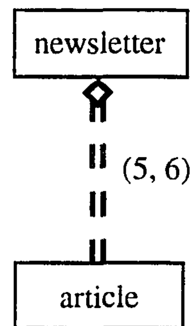


Figure 4.17 Newsletter schema

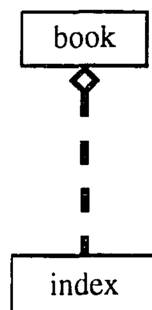


Figure 4.18 A single-valued part relationship

Pictorially, the range-restricted part relationship is a twofold modification of the generic symbol. First, the dashed line is doubled up in order to convey the relationship's multiplicity ("multiple lines represent multiple parts"). Second, its numerical range restriction is placed along side the line in parentheses. The range, as defined above, is interpreted as including both endpoints. The values $m = 0$ and $n = \infty$ signify the omission of the respective bounds and are by convention written as "-". The schema representation for the newsletter example is shown in Figure 4.17.

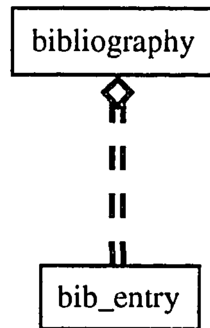


Figure 4.19 A multi-valued part relationship

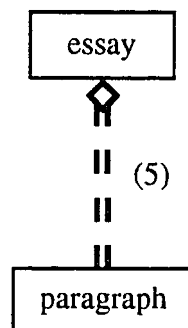


Figure 4.20 A fixed-cardinality part relationship

There are many interesting special cases of range restriction which we denote with a variety of symbols. An obvious case of interest is $m = 0$ and $n = 1$, where the converse of $\diamond$ (which denotes the "has part" direction of the relationship) is required to be a partial function from $E(A)$ to $E(B)$. For this case, referred to as the *single-valued* part relationship, the numerical range is omitted altogether and the generic symbol's single-lined connection is restored to convey the single-valuedness (Figure 4.18). (The mnemonic: "a single line represents a single part.") With $m = 0$ and $n = \infty$ (i.e., no constraints at all on the cardinalities of the meronym sets), we have what we call a *multi-valued* part relationship. Here, the range is also omitted, but the dual-line is retained (Figure 4.19).

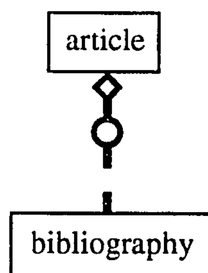


Figure 4.21 Article with an essential bibliography

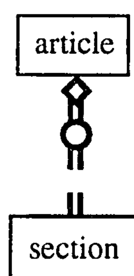


Figure 4.22 Article with at least one constituent section

When $m = n$, $P_{B,A}$ is said to be a *fixed-cardinality* part relationship. Instead of writing two identical numbers side by side, we consolidate them into a single number within parentheses as in Figure 4.20. A separate notation is provided for the special case where $m = n = 1$, that is, the case of *essentiality*. Essentiality refers to the fact that the mapping from $E(A)$ to $E(B)$ is essential or total, with all holonyms having exactly one part of the given type. Stated differently, it is essential that a holonym have a part of that type. To represent this, we forgo the parentheses and adorn the single-valued part relationship symbol with a circle. (This convention is consistent with that used in [211] and the one we used for ordinary essential relationships in Chapter 2.) We see in Figure 4.21 that an article is required to have exactly one

bibliography. *Multi-valued essentiality* denotes a "one or more" semantics and is represented by the multi-valued symbol adorned with a circle (Figure 4.22).

The part relationship which models ODN requires a modification to $\diamond$, as follows:

Definition 7: Let $(s_1, s_2, \ldots, s_n)$ be an n-tuple of selectors with $n > 1$. The *ordered part relationship of definite number* is defined as $P_{B,A} = \langle \diamond^{(s)}, \chi, \textit{ordered-definite}, \delta, \nu \rangle$, where $\diamond^{(s)} = (\diamond^{(s_1)}, \diamond^{(s_2)}, \ldots, \diamond^{(s_n)})$ is an n-tuple of relations from $E(B)$ to $E(A)$ such that for $1 \leq i \leq n$, the converse of $\diamond^{(s_i)}$ is a partial function from $E(A)$ to $E(B)$.

If $(b, a) \in \diamond^{(s_i)}$, then b is said to be part of a in the capacity of s_i. The condition that the converses be partial functions means that for a given holonym each capacity will be filled by at most one meronym.

With $\diamond$ replaced by an n-tuple of relations, the definitions of the meronym and holonym sets, and consequently the constraints of the previous characteristic dimension, become inapplicable. To remedy this, we define $\diamond = \bigcup_{1 \leq i \leq n} \diamond^{(s_i)}$ for an ODN part relationship and note that $(b, a) \in \diamond$ indicates that b is part of a independent of capacity.

The graphical schema representation for the ODN part relationship involves the replacement of the numerical range by the list of selectors $s_1, s_2, \ldots, s_n$ written in square brackets. In Figure 4.23, minutes are defined to have text segment parts with capacities *topics*, *discussion*, and *decisions*.

An OIN part relationship is defined similarly to that above.

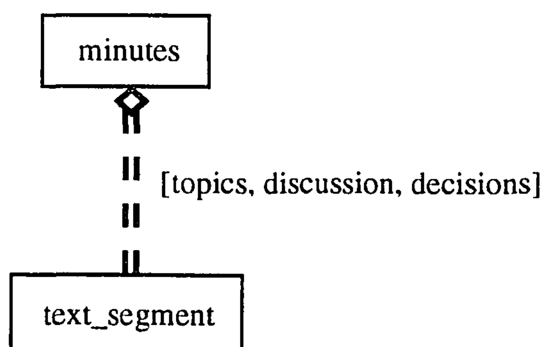


Figure 4.23 Minutes with an ordered list of parts

Definition 8: The *ordered part relationship of indefinite number* is defined as $P_{B,A} = \langle \diamond^{(s)}, \chi, \textit{ordered-indefinite}, \delta, \nu \rangle$, where $\diamond^{(s)} = \{\diamond^{(s_i)}\}$ is a sequence of relations from $E(B)$ to $E(A)$ such that $\forall i \in \mathbf{Z}^+$, the converse of $\diamond^{(s_i)}$ is a partial function from $E(A)$ to $E(B)$.

In this case, if $(b, a) \in \diamond^{(i)}$, we say that b is part of a in the capacity of its ith part with respect to $P_{B,A}$. This part relationship defines, for each holonym, a partial mapping of the positive integers to $E(B)$, and, as above, it comprises a sequence of relations $\diamond^{(s)}$ rather than a single relation $\diamond$. Therefore, we similarly define $\diamond = \bigcup_i \diamond^{(i)}$. It should be noted that Definition 8 does not force any continuity on the sequence of meronyms associated with a holonym. So, while it is being written, an article can have, say, a section three without a section two.

Graphically, the schema symbol for OIN is nearly identical to that for ODN. Instead of the selectors, however, an ellipsis is placed in the square brackets to convey the indefiniteness (Figure 4.24). In some situations, it may be necessary to label the sequence of parts in order to improve readability. For example, the chapters

of a book might be modeled as text segment objects and not as chapters per se. So, a label "chapter" can be placed in front of the brackets to make clear the exact function of the text segments (Figure 4.25).

The realization of range-restriction is accomplished by placing the restriction directly on a multivalued p_B of the generic realization. In this way, holonyms are forced to have some number of associated relational objects and, hence, parts in turn. The realization of the newsletter example is shown in Figure 4.26, where the relationship $p_{\texttt{article}}$ has a range-restriction of 5–6.

Because the relationship p_B is now multivalued and there may exist many parts per holonym, the selector method $\mathcal{B}$ defined on the holonymic class to retrieve the parts is, in general, multivalued, too. It is written in our path notation as follows:

$$\mathcal{B}() : p_B \rightarrow \{\texttt{B-PART-A}\} \ : \ @meronym \rightarrow \{B\}. \tag{4.3}$$

The initial transition across p_B yields a set of relational objects. The next transition, which iteratively applies *meronym* to that result, produces the desired set of parts. In

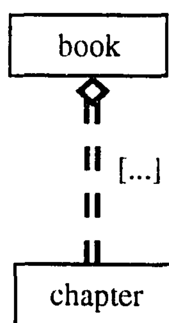


Figure 4.24 Books with an indefinite number of chapters

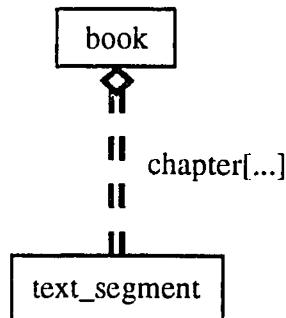


Figure 4.25 Books with text segments as chapters

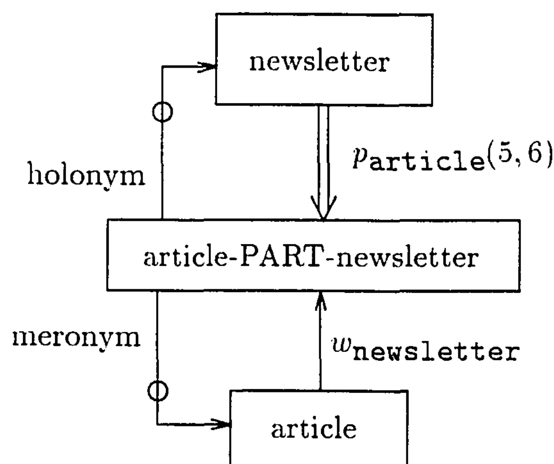


Figure 4.26 Realization of range-restriction

the cases of single-valuedness and essentiality, the iteration in the second transition is degenerative, and the resulting set reduces to a singleton.

The basis of the realization for the two types of ordering is the addition of an attribute *capacity* to the definition of the part relationship class `B-PART-A` (see Figure 4.27; note that p_B is multivalued without a range-restriction). This attribute, as its name implies, serves as a discriminator for the different capacities that a part may function in. (As such, it may be referred to simply as the discriminator). Its domain is either the set of specified selectors (i.e., a finite subset of the type STRING) in the

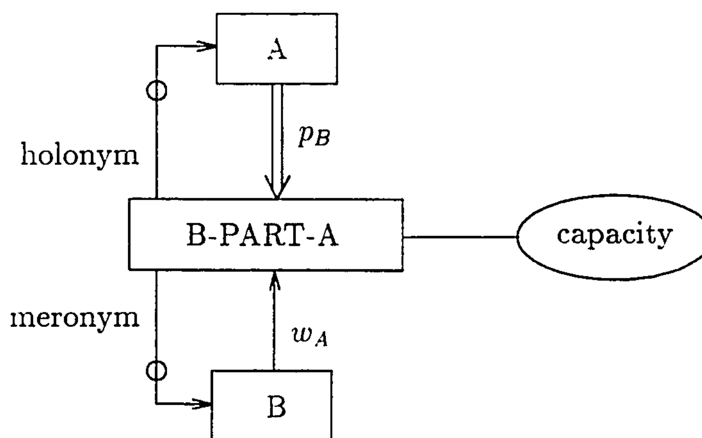


Figure 4.27 Realization of ordering

case of ODN, or the positive integers for OIN. So, for example, if the object $o(b, a)$ is an instance of B-PART-A and $capacity[o(b, a)] = x$, this means that $b \in E(B)$ is part of $a \in E(A)$ in the capacity x.

In conjunction with this new attribute, the selector method $\mathcal{B}$ for the parts must be refined to accommodate the notion of capacity. In particular, it must traverse the schema path occurrence (from A to B) satisfying the condition that the part (i.e., the terminus of the path) serve in the capacity supplied as an argument to the method. Satisfaction of the condition is determined by examining the terminus of the path from A to the discriminator. Since our own path notation is not powerful enough to express this selection query, we turn to the more general concept of *path expression* as defined in [104]. Using a notation based on that of XSQL [104], the method $\mathcal{B}$ is written in the following, where A is the holonymic class, and B the meronymic class.

METHOD $\mathcal{B}(x : \text{capacityType}) \rightarrow B$

SELECT y

FROM A z OID z

WHERE $z.p_{\mathcal{B}}$.B-PART-A.$meronym[y]$

and $z.p_{\mathcal{B}}$.B-PART-A.$capacity[x]$

The variable z is bound to the instance of A for which the method is invoked, as expressed in the "FROM" clause. (As normal, it can be taken as the implicit 0th argument to the method.) As seen on the signature line, the method itself is defined to return the instance b (of class B) which serves as a part of z in the capacity x (the formal parameter of the method). The body of the method is a selection query written in XSQL notation. The first conjunct of the "WHERE" clause states the obvious condition that the result y must be part of z with respect to the given part relationship (independent of capacity), and is analogous to the previous path notation for $\mathcal{B}$. The second says that y must fill the given capacity x in z.

For flexibility, we alternatively allow $\mathcal{B}$'s argument to be omitted, with the consequence that all parts, regardless of their capacities, are returned as the result. In this case, $\mathcal{B}$ reduces to the method (4.3) above.

Part ordering, be it ODN or OIN, calls for a slight modification to the realizations of global and class exclusiveness in order to maintain the proper semantics. Both types of exclusivity dictate that a part have no more than one holonym and enforce this by making the relationship w_A single-valued. Neither, though, says anything

about a part functioning in more than one capacity within the *same* holonym. But with w_A single-valued, this is inherently disallowed, adding a constraint not called for by either. To correct this, the following modifications are made to the realization of a given global or class exclusive part relationship in the presence of ordering. First, its relationship w_A is made multivalued, and the assignment carried out by its facet is changed to a "union" statement like that in the shared facet. Second, the precondition of the facet is augmented with a clause requiring that the following hold: Either the target meronym does not already participate in the given part relationship, or, if it does, it does so with respect to the same holonym in a different capacity.

4.5 The Dependency Dimension

In our part model, a part relationship can be endowed with different forms of dependency as specified by the domain of the third characteristic dimension:

$$\delta \in D = \{\textit{part-to-whole},\ \textit{whole-to-part},\ \textit{nil}\}.$$

The third value indicates that the part relationship lacks any dependency semantics.

Earlier, we discussed the philosophical issue of ontological dependency in the context of parts and wholes. Wholes may be said to lose their identity or go out of existence when all their parts are removed or destroyed. In OODB modeling, dependency is a more mundane notion, used more for the sake of convenience than

as a conceptual tool. Dependency, in this context, describes the deletion semantics of parts and wholes. If, for example, a part b is made dependent on a whole a, and the whole a is deleted, then the part b is deleted automatically.

Such dependency semantics is often desired when modeling with parts [107], especially when the holonyms are large objects comprising numerous meronyms. Such a scenario is illustrated in Figure 4.28 where we show a CAD drawing along with its many parts (in the figure, one part relationship symbol denotes all the occurrences). Having the parts deleted in one clean sweep on the deletion of the drawing as a whole would alleviate the burden of searching these parts out and deleting them explicitly. As is meant to be conveyed by the picture, such a process can be tedious and time-consuming. For this reason, we include *part-to-whole dependency* in our model.

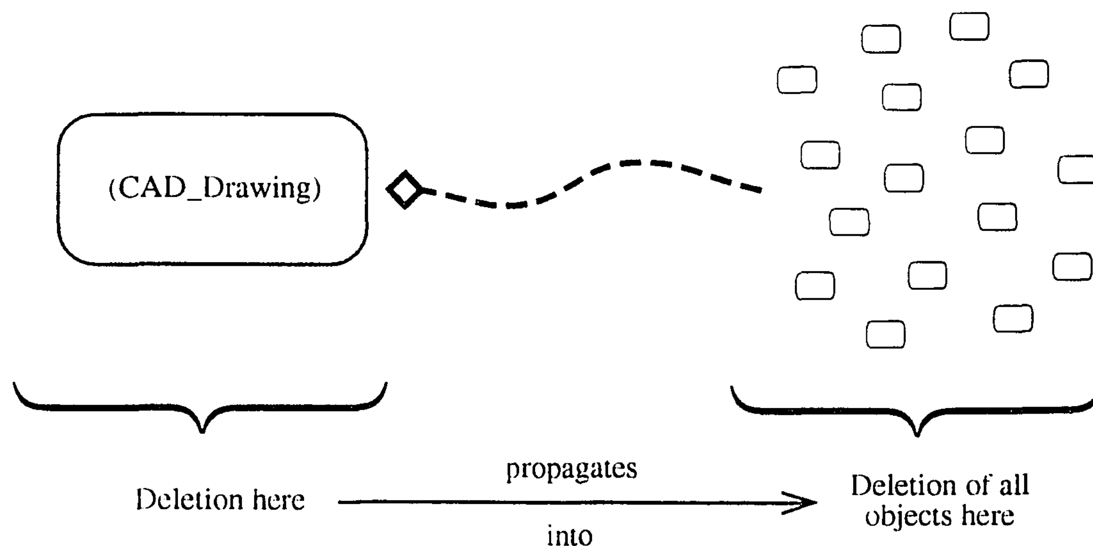


Figure 4.28 Dependency of meronyms on holonym

Of course, guided by philosophical insights, one may view a specific type of part in a part-whole configuration as a defining element for the whole, without whose existence the whole becomes rather insubstantial. Consider, for example, that without its frame, a bicycle may be seen as nothing more than a collection of "spare" parts. In this case, it makes sense to propagate the deletion of a frame into the deletion of its containing bicycle. We refer to the situation where the deletion of the part propagates into the deletion of the whole as *whole-to-part dependency* and include it as an alternative in our part model.

To be more precise about the two types of dependency, we define part relationships which exhibit these characteristics in the following. There, we use the notation $del(x)$ to denote the application of a method to delete the instance x [107]. (Cf. the methods *delInstance* and *destroy* in Chapter 6.)

Definition 9: The part relationship $P_{B,A}$ is *part-to-whole dependent* (i.e., $\delta =$ *part-to-whole*) if $\forall a \in E(A)\ del(a) \Rightarrow \forall b \in E(B)$ such that $b \diamond a \wedge H_{\diamond}(b) = \{a\}$, $del(b)$.

Definition 10: The part relationship $P_{B,A}$ is *whole-to-part dependent* (i.e., $\delta =$ *whole-to-part*) if $\forall b \in E(B)\ del(b) \Rightarrow \forall a \in E(A)$ such that $b \diamond a \wedge M_{\diamond}(a) = \{b\}$, $del(a)$.

If the value of δ is *nil*, then neither of the above deletion semantics is applicable. It will be noted that in both cases, the condition requiring that the independent deleted item (e.g., a in the case of part-to-whole dependency) be the only existing referent

implies a "multivalued" deletion semantics in that the deletion is not propagated until the set of referents on which a given object depends becomes empty (cf. [107]).

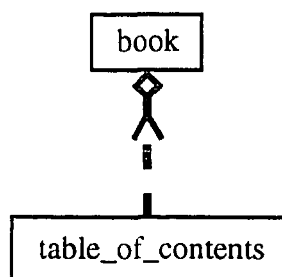


Figure 4.29 Table of contents dependent on its book

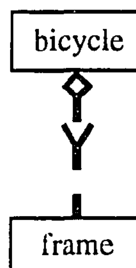


Figure 4.30 Bicycle dependent on its part frame

To express the dependency in our graphical schema representation, an arrowhead facing in the direction of the dependency (i.e., against the direction of the deletion propagation) is placed immediately behind the diamond head. This is systematic with our notation for dependency in Chapter 2. Figure 4.29 shows the dependency of the table of contents on its book, while Figure 4.30 shows the converse dependency of bicycle on its constituent frame.

The realization of the dependency semantics is accomplished by making the connections along the path between the two classes dependent in the direction of the dependency. This means that for part-to-whole dependency both w_A and *holonym*

are made dependent, as in Figure 4.31 where we show the realization of the schema from Figure 4.29. Likewise, for the whole-to-part case, the relationships p_B and *meronym* are made dependent.

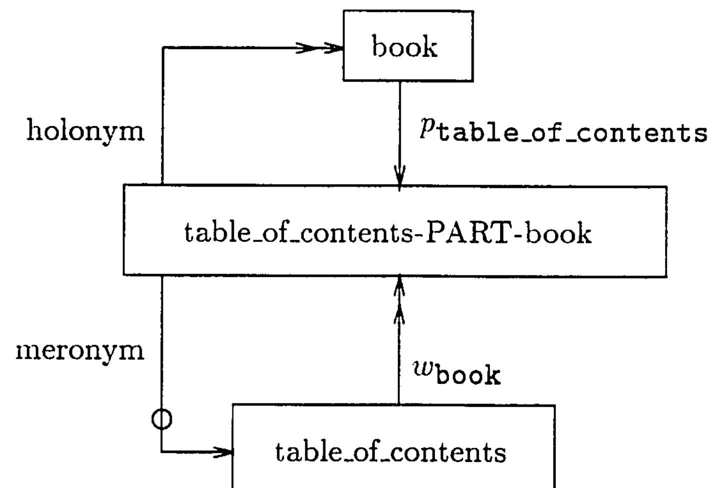


Figure 4.31 Realization of part-to-whole dependency

While the dependency dimension constitutes a set of imperatives that explicitly affects the deletion semantics of parts and wholes, there is also deletion semantics implied by the cardinality constraints of the previous characteristic dimension. For example, if we say that an article must have one and exactly one abstract (i.e., it is essential for an article to have an abstract), then we are committed to disallowing the deletion of any abstract which is currently part of some article and being used to satisfy the foregoing constraint. As we will now see, the combination of these constraints and imperatives across different part relationships in an OODB schema can lead to certain conflicts when it comes time for enforcement.

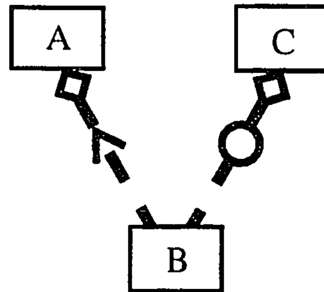


Figure 4.32 A part schema with possible deletion conflicts

Consider the schema shown in Figure 4.32 which contains three classes A, B, and C. There, we see two part relationships: A shared, part-to-whole dependent relationship between B and A and a shared, essential relationship between B and C. The presence of these two part relationships implies the following:

1. An instance of B may be a part of an instance of A and an instance of C at the same time.

2. Deleting an instance of A requires deleting its part which is an instance of B.

3. An instance of C must at all times have one and only one part B.

Now, assume that there exists an instance b of B such that b is part of an instance a of A and is, at the same time, part of an instance c of C. [This is permitted by (1).] Furthermore, assume that a is b's only holonym from the class A. What happens, at this point, if an attempt is made to delete a? According to (2), b must be deleted. However, (3) requires that c always have exactly one part which is an instance B. Deleting b would deprive c of such an essential part. Therefore, we have

two conflicting imperatives which are products of the two part relationships: Thou shalt delete b, and thou shalt not delete it.

While such conflicts are not an inherent part of the above schema (i.e., the conflict does not necessarily arise for all instances of B), some measures must be decided upon in the context of a realization to make certain that such conflicts are resolved when they do come up. One possible measure, for example, is to defer a decision on the matter to some external agent, such as a database user (the warning semantics option) [127]. Another choice would be to raise a system exception, and invoke an exception handler to break the deadlock (the error semantics option). Yet another option would be to prioritize the part relationships and to defer to one over the other. This latter approach is the one employed in our VML metaclass realization discussed in Chapter 6, where we adhere to the *conservation of cardinality constraints*, giving strict existence constraints priority over dependency. So, in the above scenario, the object b would not be deleted because the essentiality constraint would override the dependency.

Other issues such as this, affecting the deletion semantics of parts and wholes, are implied by certain part relationship configurations within a schema. These will be addressed when we take up the matter of the "destroy" method in the context of our metaclass realization in Chapter 6. There, we present rules which, along with the constraints and imperatives of the various characteristic dimensions, govern the deletion semantics within a part hierarchy.

CHAPTER 5

VALUE PROPAGATION AND DERIVED ATTRIBUTES IN OBJECT-ORIENTED DATABASE PART HIERARCHIES

In this chapter, we describe the notion of value propagation and its concomitant derived attributes. In the first section, we introduce the last characteristic dimension of the part relationship, the value propagation dimension. In this context, we present the various kinds of propagation that can take place across a single part relationship. As we will see, this dimension provides a powerful means for defining derived attributes of classes with respect to the part relationship. After that, we give a summary of the interaction of the various characteristic dimensions of the part relationship. Finally, we present the notion of generalized derived attribute which may be defined in terms of value propagations across many part relationships.

5.1 The Value Propagation Dimension

There are times, when modeling with parts and wholes, that a certain feature of a part is naturally assimilated as a feature of its whole, or vice versa. For example, the age of a plane may be modeled as the age of its airframe rather than as an intrinsic property of the plane as a whole [145]. Likewise, the color of the plane can be taken to be the color of its fuselage, or alternatively the plane may be regarded as multi-colored with the set of colors being obtained, for example, from its fuselage, wings, and tail. In the former case, the value of *age*, rather than being duplicated as an attribute of plane, should be stored solely with the airframe and propagated

upward on demand (Figure 5.1). Age, in this sense, is a *derived* property of plane. Such derived schema components have been a mainstay of traditional semantic data modeling [94]. Their use tends to make a database's conceptual schema a more accurate reflection of the application domain it is designed to model. They also promote more concise representations and alleviate the burden of explicit integrity maintenance and the anomalies associated with redundant data storage [46]. In this chapter, we introduce formal notions of *value propagation* and *derived attribute* to address these issues. Informally, value propagation can be described as the flow of data values across the part relationship in either direction, allowing for the access of additional, derived properties at the receiving class.

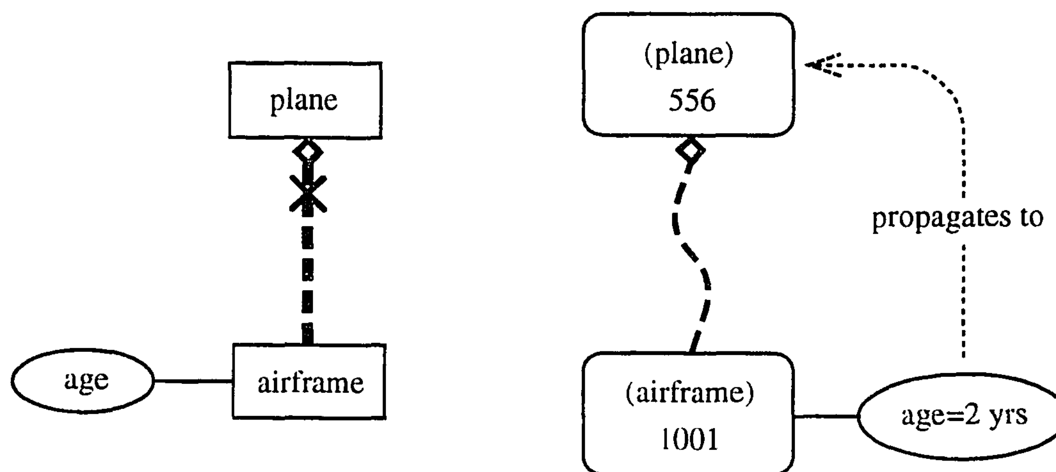


Figure 5.1 Age propagated from airframe to plane

In our part model, value propagation is defined in terms of two aspects: (1) An underlying formal connection between instances (either meronyms or holonyms) and the data values being propagated to them, and (2) a derived attribute of the receiving

class complete with its own method and message in the class's public interface. The second item gives value propagation a resemblance to IS-A inheritance. In fact, it may even be referred to as part-whole inheritance, but we try to avoid this because it leads to confusion and tends to blur the distinctions between the two. Let us consider some of these distinctions.

The inheritance of IS-A is, at bottom, a template sharing mechanism [73] which transmits the definitions of *all* properties from one class (the superclass) to another (the subclass). The assignment of values for these properties for instances of the subclass, however, is in no way a function of the IS-A link (except, in the case of defaults). In contrast, value propagation along the part relationship always takes place at the instance level, directly between meronyms and holonyms, with values for a selected property being passed from one object to another. The resulting, newly defined property of the receiving class, which we shall refer to as a *derived attribute*, is inherently given a value through this mechanism. We note that there are other proposals which exploit such instance-to-instance transfers. For example, in [33], a "value inheritance" between objects is used as a means for supporting versioning. There, too, it is noted that such value transmissions can, in general, be defined across any system-defined relationship connecting instances.

Objects can theoretically be represented in a "delta" fashion across IS-A links [41, 51, 110], with basic instantiation occurring at all levels of the hierarchy. With this division of an object's representation, property inheritance indeed becomes an

instance-level affair. Creating an instance of, say, **grad_student** would cause the creation of an instance of its superclass **student** and an instance of that class's superclass **person**, etc. An instance of **grad_student** might then be forced to obtain its student identification number from its associated instance of **student**.

Some attempts to overload IS-A with additional semantics regarding "roles" and "contexts" [65, 110] also call for the division of an object's representation and, in effect, set up a system of value inheritance. Additionally, in framed-based knowledge representation [207], value inheritance is used across IS-A links to aid in the definition of "class" frames and to give instance frames default values.

Even if it is granted that the division of an object's representation among superclasses is conceptually significant, there is no chance that an object of the subclass will ever have more than one such associated object in a single superclass. This is key to our second and most important distinction, explained presently.

Value propagation across a *single* part relationship has the potential for being ambiguous while inheritance across a single IS-A link does not. In any OODB model, a given property (i.e., one with a given identifier) cannot be defined more than once for the same class. Therefore, only one definition for it will ever cross a single IS-A link. On the other hand, because value propagation is an instance to instance phenomenon and the part relationship allows a single holonym to have many meronyms, or vice versa, there is a chance that the source of the propagation may

not be well defined. As it turns out, this potential ambiguity gives the mechanism its power and adds richness to the definition of derived attributes.

In our part model, we offer two ways of dealing with the potential ambiguity. The first is to require the uniqueness of the source and pass along the value as it appears there. This we call *invariant value propagation.* The second approach is to permit the ambiguity of the source and allow for the specification of a family of symmetric operators [58] that transform the multiple values into a single value of the data type of the property. This latter approach is called *transformational value propagation.* We also define a special case of this, called *cumulative value propagation*, similar to union inheritance in frame-based knowledge representation systems [207], where the values are collected together into a set of the given type. We have already alluded to an example of this where the color of a plane is taken to be the set of colors of its wings (among other things). Another example would be the body of a car obtaining its color as the union of the colors of its constituent panels.

The foregoing discussion should not be confused with the ambiguity problems created by multiple inheritance involving many *different* part or IS-A relationships [190]. That issue will be addressed in Section 5.3, where we will further extend the notion of derived attribute so that it may be defined in terms of ambiguous value propagations across a variety of part relationships. Canonical examples of this are "weight" and "cost" (of, e.g., a car), defined as sums of values of properties from all constituent parts, regardless of class. Or, as another example, the material make-up

of a golf club is the set of materials from its shaft, head, and grip. We will see that this mechanism serves as a natural third resolution strategy for the "multiple value propagation" (or the "multiple inheritance") problem within part hierarchies. An analogue to this solution is not applicable in ordinary OODB IS-A hierarchies.

A third distinction between value propagation across the part relationship and IS-A inheritance is that the former may be defined in either direction across the link, whereas the latter ordinarily proceeds strictly downward from superclass to subclass. We do note, however, that in [179], an upward inheritance mechanism along IS-A is proposed for defining generalization classes to integrate other classes. In AI, where IS-A is sometimes erroneously employed both at the instance and class level [22], upward inheritance has been used to derive default values [28, 62]. We have already seen examples of upward propagation from part to whole; an example of downward propagation is the case where the type-font of a book is passed along to its constituent chapters.

In our part model, the value propagation dimension may take on six different values as expressed by its domain:

$$V = \{up,\ down,\ upTrans,\ downTrans,\ up\&down,\ nil\}.$$

A value of *up* signifies upward invariant propagation; a value of *down*, downward invariant propagation. The two succeeding values stand for upward and downward transformational propagation, respectively. A value of *up&down* indicates that there

is both upward and downward propagation, and *nil* indicates the absence of any propagation.

Recall that, following [214], we define the (readable) properties of a class as functions which take instances into values of an associated type. For example, the attribute *age* is a function which maps instances of class **person** into values of type REAL (or some restricted range thereof). First, let us consider the invariant propagations.

Definition 11: Let $\pi_A : E(A) \rightarrow \tau$ be a property of class A. The *invariant downward propagating part relationship which propagates the value of* π_A is defined as $P_{B.A} = \langle \diamond, \chi', \kappa, \delta, (down, \mathcal{D}_{\pi_A}) \rangle$. Here, $\chi' \in X \setminus \{\textit{limited-shared}\}$, τ is any data type, and the function $\mathcal{D}_{\pi_A} : E(B) \rightarrow \tau$, called a "derived attribute," is defined as follows (where $C \in \tau$):

$$\mathcal{D}_{\pi_A}(b) = \begin{cases} \pi_A(a), & \text{if } \exists a \in H_\diamond(b) \wedge \pi_A(a) \text{ is defined} \\ C, & \text{otherwise.} \end{cases}$$

We note that the derived attribute, the new property of B, is a function defined simply as the value of the property π_A for the holonym a when such a holonym exists for the given part b. One will note the stipulation in the definition that states that invariant downward propagating part relationships cannot be limited-shared. This restriction ensures that $\diamond$ is a partial function and, consequently, that the derived attribute $\mathcal{D}_{\pi_A}$ is well defined. In other words, it ensures that there is no ambiguity

regarding the instance a which is the source of the propagation, because there can be at most one a. In the case where the meronym b does not have a holonym $a \in E(A)$ or the holonym's property π_A is undefined, then the derived attribute takes on the default value C [164], some constant value of the data type τ. This default value may be omitted with the consequence that the derived attribute may be undefined for some elements of its domain.

Derived attributes, as with any derived schema components, should fit seamlessly into their respective classes and be accessible in the same manner as every other property. Toward this end, we augment the public interface of the class B with a message to retrieve the value of the derived attribute. The introduction of this message serves to make the value propagation mechanism transparent in that the value of the derived attribute is obtained like any other locally defined property. Following the conventions discussed in [138], this message is chosen to be the same as that for accessing the property π_A at class A. So, for example, the interface of the class **chapter** would be augmented with the message "font" which, for any of its instances, would retrieve the value of the derived attribute $\mathcal{D}_{font}$ defined as:

$$\mathcal{D}_{font}(c) = \begin{cases} font(k), & \text{if } \exists k \in H_{\diamond}(c) \wedge font(k) \text{ is defined} \\ \text{times-roman}, & \text{otherwise.} \end{cases}$$

times-roman. otherwise.

In this example, we see that if a chapter is not currently part of a book or if the book has not yet received a font assignment, then the chapter is displayed or printed using a times-roman font.

Invariant propagation in the direction from the part to the whole is defined analogously to that above:

Definition 12: Let $\pi_B : E(B) \rightarrow \tau$ be a property of B. The *invariant upward propagating part relationship which propagates the value of* π_B is defined as $P_{B,A} = \langle \diamond, \chi, (\textit{range-restricted}, m', n'), \delta, (\textit{up}, \mathcal{D}_{\pi_B}) \rangle$. Here, $0 \leq m' \leq n' \leq 1$, $n' \neq 0$, τ is any data type, and the function $\mathcal{D}_{\pi_B} : E(A) \rightarrow \tau$, called a derived attribute, is defined as follows:

$$\mathcal{D}_{\pi_B}(a) = \begin{cases} \pi_B(b), & \text{if } \exists b \in M_\diamond(a) \wedge \pi_B(b) \text{ is defined} \\ C, & \text{otherwise.} \end{cases}$$

Once again, we see that the derived attribute is a function defined simply as the value of the property π_B for the part b when such a part exists for the given whole a. This kind of part relationship is required to be either single-valued or essential in order to ensure that $\diamond^{-1}$ is a partial function and that, therefore, the derived attribute is well defined (meaning that the propagation source, an instance b, is unique if it exists). As above, a default value C, a constant of the data type τ, may be given for cases when the given whole a does not have a part $b \in E(B)$ or the part's property π_B is undefined.

For the example of *age* propagating from airframe to plane, the derived attribute $\mathcal{D}_{age}$ is defined as:

$$\mathcal{D}_{age}(l) = \begin{cases} \text{age}(r), & \text{if } \exists r \in M_{\diamond}(l) \wedge age(r) \text{ is defined} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Here, we say nothing about the age of a plane when its airframe is missing. As with downward propagation, the receiving class of the upward propagation (i.e., the holonymic class) is equipped with a message in its public interface to handle requests for the value of the derived attribute. In the example, the public interface of the holonymic class `plane` is given the additional message "age" that is used to retrieve a given plane's age.

As an example with a default, a piano as a whole could be defined to obtain its color from its rim, and since we know thàt most pianos are black, we may wish to make that the default value as in:

$$\mathcal{D}_{color}(n) = \begin{cases} \text{color}(r), & \text{if } \exists r \in M_{\diamond}(n) \wedge color(r) \text{ is defined} \\ \text{BLACK}, & \text{otherwise.} \end{cases}$$

where n is a piano and r is its rim.

According to our definition, the property being propagated from the part to the whole is not restricted to a simple attribute but may be any readable property of the class, including relationships or even methods. For example, in a CAD environment

[127], we may want to propagate the value of an *attachedTo* relationship: If a door's hinge is attached to some wall, then the door itself is attached that wall. So, the derived attribute $\mathcal{D}_{attachedTo}$ is defined as:

$$\mathcal{D}_{attachedTo}(d) = \begin{cases} attachedTo(h), & \text{if } \exists h \in M_{\diamond}(d) \wedge attachedTo(h) \text{ is defined} \\ \text{NULL}, & \text{otherwise}, \end{cases}$$

where h is a hinge, and *attachedTo*: $E(\texttt{hinge}) \rightarrow$ OIDType. If a door does not have a hinge, then it is not attached to any wall and the value for the derived attribute is the default value NULL, the null object identifier [56].

Now, let us extend the above ideas and move on to transformational propagation. Transformational upward value propagation relaxes the restriction on the part relationship's cardinality, allowing any number of source meronyms to be present. As its name implies, it is designed to transform the multiple property values from these meronyms into a single value of the property's data type using algebraic tools. In particular, the transformation is carried out with the use of a specified family of symmetric operators [58].

Definition 13: Let $\pi_B : E(B) \rightarrow \tau$ be a property of B. The *transformational upward propagating part relationship which propagates* π_B is defined as $P_{B,A} = \langle \diamond, \setminus, \kappa, \delta, (upTrans, \mathcal{D}_{\pi_B}, \{T^{(n)}\}) \rangle$. Here, τ is any data type, $\{T^{(n)}\}$ is a family of symmetric operators $T^{(n)} : \tau^n \rightarrow \tau$, with $n > 0$, and the function $\mathcal{D}_{\pi_B} : E(A) \rightarrow \tau$,

called a derived attribute, is defined in terms of $\{T^{(n)}\}$ as follows. (Note that the meronym set of an instance a of A is taken to be $M_{\diamond}(a) = \{b_1, b_2, \ldots, b_m\}$, $m \geq 0$.)

$$\mathcal{D}_{\pi_B}(a) = \begin{cases} T^{(m)}[\pi_B(b_1), \pi_B(b_2), \ldots, \pi_B(b_m)], & m \neq 0 \wedge \pi_B(b_i) \text{ is} \\ & \text{defined for } 1 \leq i \leq m \\ C, & \text{otherwise.} \end{cases}$$

As an example of this type of part relationship, we turn to electronics where the reliability of an amplifier could be defined as the minimum reliability of the transistors which constitute its various stages. The derived attribute $\mathcal{D}_{\text{reliability}}$ of the class `amplifier` would then be written in terms of the attribute *reliability*: E(`transistor`) $\rightarrow$ PERCENT of the class `transistor`, as follows:

$$\mathcal{D}_{\text{reliability}}(a) = \begin{cases} \min[\text{reliability}(t_1), \ldots, \text{reliability}(t_n)], & n \neq 0 \wedge \textit{reliability}(t_i) \text{ is} \\ & \text{defined for } 1 \leq i \leq n \\ 0.99, & \text{otherwise,} \end{cases}$$

where $t_1, \ldots, t_n$ are the transistors of an amplifier a. In this application, the data type PERCENT is assumed to be some normalized quantity falling in the range of 0 to 1. In this example, we have specified a default reliability of 0.99 for cases where a value cannot be acquired from the transistors. While this value was chosen arbitrarily here, in a real application it might be decided upon through a statistical analysis of previous reliability measures.

As with invariant propagation, the property being propagated does not necessarily have to be an attribute but may be a relationship or method. As an example, assume that there is a method *wordCount* defined on the class `section` which for a given section of text computes the number of words it contains. Using this method, we can define a propagating part relationship which computes the number of words in an entire article. The derived attribute $\mathcal{D}_{wordCount}$ defined on the class `article` is given as follows:

$$\mathcal{D}_{wordCount}(a) = \begin{cases} \sum_{i=1}^{n} wordCount(s_i), & n \neq 0 \\ 0, & \text{otherwise,} \end{cases}$$

where $s_1, s_2, \ldots, s_n$ are the sections of the given article a. An article is assumed to be of length 0 if it has no sections. Here, we assume that *wordCount* is a total function (i.e., defined for all existing sections). This is reasonably because the property *wordCount* is a method, and no explicit assignment for it need be done.

A special case of transformational propagation, referred to as *cumulative value propagation*, is defined in terms of a set-valued derived attribute $\mathcal{D}_{\pi_B} : E(A) \rightarrow \{\tau\}$ (where $\{\tau\}$ denotes a type comprising sets of values of τ), specified as follows:

$$\mathcal{D}_{\pi_B}(a) = \begin{cases} \bigcup_{i=1}^{n} \{\pi_B(b_i)\}, & n \neq 0 \\ C, & \text{otherwise.} \end{cases}$$

This transformation amounts to the union over the sets created through canonical injection of every property value into a singleton set. It thus corresponds to a form of union inheritance [207]. As usual, the default value is C which may be the empty set. The color of the body of a car modeled as the unique colors of its constituent panels is an application of cumulative propagation, where the set-valued, derived attribute $\mathcal{D}_{color}$ has the specification:

$$\mathcal{D}_{color}(a) = \begin{cases} \bigcup_{i=1}^{n} \{color(p_i)\}, & n \neq 0 \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

where a is a car body and the p_i's are its panels. Here, we leave the color of the body undefined until it is obtained from the panels.

Transformational value propagation in the downward direction is defined analogously to that in the upward direction.

Definition 14: Let $\pi_A : E(A) \rightarrow \tau$ be a property of A. The *transformational downward propagating part relationship which propagates* π_A is defined as $P_{B,A} = \langle \diamond, \chi, \kappa, \delta, (downTrans, \mathcal{D}_{\pi_B}, \{T^{(n)}\}) \rangle$. Here, τ is any data type. $\{T^{(n)}\}$ is a family of symmetric operators $T^{(n)} : \tau^n \rightarrow \tau$, with $n > 0$, and the function $\mathcal{D}_{\pi_B} : E(A) \rightarrow \tau$, called a derived attribute, is defined in terms of $\{T^{(n)}\}$ as follows. (The holonym set of an instance b of B is taken to be $H_\diamond(b) = \{a_1, a_2, \ldots, a_m\}$, $m \geq 0$.)

$$\mathcal{D}_{\pi_A}(b) = \begin{cases} T^{(m)}[\pi_A(a_1), \pi_A(a_2), \ldots, \pi_A(a_m)], & m \neq 0 \wedge \pi_A(a_i) \text{ is} \\ & \text{defined for } 1 \leq i \leq m \\ C, & \text{otherwise.} \end{cases}$$

As an example of this kind of propagation, we return to the university environment. Earlier, we modeled students as parts of the sections which they are taking. Of course, students may be part of many sections. (Above, we placed an upper bound of six on the number of such sections.) Ordinarily, the class `section` has a property *credits* which for any section gives the number of academic credits it is worth. This property may be an attribute or even a path method defined with respect to a class, say, `course`. In any event, the number of credits that a student is currently enrolled for is just the total number of credits of all the sections that he is part of. Thus, the property *credits* of class `student` is a derived attribute defined with respect to a downward transformational value propagation across the part relationship between `student` and `section`. Formally, the derived attribute $\mathcal{D}_{credits}$ (whose message in the public interface of `student` is "credits") is defined as follows:

$$\mathcal{D}_{credits}(s) = \begin{cases} \sum_{i=1}^{n} credits(c_i), & n \neq 0 \\ 0, & \text{otherwise.} \end{cases}$$

where $c_1, c_2, \ldots, c_n$ are the sections of the given student s. We assume that the property *credits* of the class `section` is a total function. A student that is not part of any section is deemed to be enrolled for 0 credits.

The derived attribute for a cumulative downward propagation of a property π_A is also set-valued. In general, it has the following definition.

$$\mathcal{D}_{\pi_A}(b) = \begin{cases} \bigcup_{i=1}^{n} \{\pi_A(a_i)\}, & n \neq 0 \\ \emptyset, & \text{otherwise.} \end{cases}$$

As a specific example of this phenomenon, we again turn to the part relationship between `student` and `section`. The class `section` is assumed to be related to the class `instructor` via a relationship *taughtBy*, which for any section refers to the instructor teaching that section. At the class `student`, the derived attribute *taughtBy* denoting the set of teachers that a given student is currently being taught by is defined in terms of a cumulative downward propagation of the relationship *taughtBy* of class `section`. Once again, this example demonstrates that properties other than attributes may serve as the source of propagation. The derived attribute $\mathcal{D}_{taughtBy}$ of `student` is given as follows:

$$\mathcal{D}_{taughtBy}(s) = \begin{cases} \bigcup_{i=1}^{n} \{taughtBy(c_i)\}, & n \neq 0 \\ \emptyset, & \text{otherwise,} \end{cases}$$

Here, as above, $c_1, c_2, \ldots, c_n$ are the sections of the given student s. A student not enrolled in any sections is not taught by anyone; hence, the empty set as a default value.

There is also the possibility that one property may be propagated downward and another upward across the same part relationship. This situation is described in the following.

Definition 15: Let $\pi_A \colon E(A) \to \tau$ be a property of class A, and let $\pi_B \colon E(B) \to \vartheta$ be a property of the class B. The *invariant propagating part relationship which propagates the value of* π_A *downward and the value of* π_B *upward* is defined as $P_{B,A} = \langle \diamond, \chi', (\textit{range-restricted}, m', n'), \delta, (\textit{up\&down}, \mathcal{D}_{\pi_B}, \mathcal{D}_{\pi_A}) \rangle$. Here, $\chi' \in X \setminus \{\textit{limited-shared}\}$, $0 \leq m' \leq n' \leq 1$, $n' \neq 0$, τ and ϑ are any data types. The function $\mathcal{D}_{\pi_A} : E(B) \to \tau$, the derived attribute of the class B, and the function $\mathcal{D}_{\pi_B} : E(A) \to \vartheta$, the derived attribute of the class A, are defined as in Definition 11 and Definition 12, respectively. The properties π_A and π_B must not be equal in the sense that they not have the same identifiers (or messages in the public interfaces of the respective classes).

We note that the last restriction is necessary to avoid a circular definition. It will also be noted that we have limited the definition of the *up&down* propagating part relationship to invariant propagations in both definitions. Clearly, this is an artificial limitation which can easily be relaxed. There is no reason why we cannot have a transformational upward propagation and an invariant downward propagation, or

a pair of transformational propagations, or any other combination across a single part relationship. We have limited ourselves, in this respect, solely to simplify the exposition. In fact, there are some other simplifying assumptions which we have employed so far in our discussion of this dimension. Their relaxation leads to other straightforward extensions, which we discuss presently.

In our descriptions of the various types of value propagation, we have limited the discussion to a single propagated property. This was done for two reasons. First of all, as alluded to, this helped to simplify the discussion. Second, it emphasized the fact that "part-whole inheritance" is done selectively, with the choice of propagated properties left solely to the database designer. This is, of course, in contrast to IS-A where there is wholesale property inheritance, with all properties of the superclass appearing as properties of the subclass. Such wholesale propagation is rarely, if ever, necessary in the context of parts and wholes.

Aside from these issues, there are no theoretical reasons why the single property restriction is necessary. In fact, in general, it is undesirable. For example, if we store the font family as an attribute with a class book and propagate it downward as we did above, then it is likely that we would also want to store a nominal font size there and propagate it, too. (LaTeX [113] employs this sort of arrangement.) Therefore, as an extension to the above definitions, we allow a set of properties to be specified for propagation in place of the single property. For transformational propagation, each such property requires its own family of operators, so properties

and operator families must be specified in pairs. For the case where upward and downward propagation coexist, we require that the set of properties going upward be disjoint from that going downward. This is analogous to the earlier restriction in Definition 15 and avoids the possibility of any circular definitions. We will see that in our metaclass realization in VML, which we discuss in Chapter 6, we allow any number of properties to propagate in either direction across the part relationship.

The use of a family of symmetric operators in the definition of upward transformational propagation (Definition 13) ignores any ordering which may exist among the parts. However, in certain modeling situations, we may wish to exploit the additional information provided by the capacities in the computation of a derived attribute. For example, consider the complete works of an author or composer appearing as a single edition in many volumes.[1] The date of publication for the collection as a whole is normally considered to be the date on which the first of its volumes appears, even though successive volumes may appear later. Thus, this value should be propagated from the single part that is in the capacity of the first volume; all other such values should be ignored. For this reason, we drop the symmetry requirement in Definition 13 when the part relationship is ODN or OIN so that the capacities of the different parts may be exploited by the operators. We still insist, however, that the same property from each or some of the parts be used in the computation of the derived attribute at the holonymic class.

[1] In English, there is no single technical term for such a collection. It is also variously referred to as a "complete edition" [157] or just simply an "edition" [45]. In German, the term is *Gesamtausgabe*.

5.1.1 Graphical Schema Notation for Value Propagation and Derived Attributes

The invariant propagating part relationships are represented graphically in a schema by two aspects. First, a *propagation label* is written alongside the ordinary part relationship symbol. This propagation label comprises the name of the propagated property written in parentheses with an arrow placed out in front to indicate the direction of the propagation: an up-arrow for part to whole, and a down-arrow for whole to part. Following the name of the property inside the parentheses and separated from it by a comma is C, the default value of the derived attribute. Second, the class receiving the propagation is given a derived attribute whose symbol is a dashed ellipse. (Recall that an ordinary attribute is a solid ellipse, and that the derived attribute which is a form of local method also uses a dashed ellipse.) As we have discussed, the name of the derived attribute (i.e., its message in the public interface) is exactly the same as that of the propagated property. Thus, the same name which appears in the propagation label also appears as the name inside the dashed ellipse. We have employed the dashed ellipse in order to signify that the attribute is derived by means of a part relationship propagating its value. In fact, the overall representation can be seen as symbolically defining both a derived attribute and its implementation in terms of the propagation of a data value across a single part relationship. Remember that owing to the fact that the invariant propagations impose constraints on other characteristic dimensions of the part relationship, the

part symbol must also convey global or class exclusiveness in the upward case, and single-valuedness or essentiality in the other.

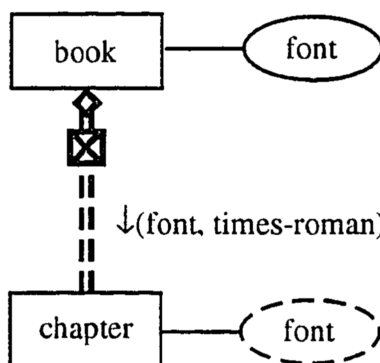


Figure 5.2 The attribute *font* being propagated from `book` to `chapter`

To illustrate the invariant propagation, we show some of the above mentioned examples in pictorial form. In Figure 5.2, we see the propagation of *font* from `book` to `chapter`. Note the default value of "times-roman" in the propagation label. The example of a plane obtaining its age from its airframe can be seen in Figure 5.3. In that example, no default was specified, so only the property's name appears in the propagation label. In contrast, the propagation label denoting the upward propagation of the attribute *color* from `rim` to `piano` contains a default value of BLACK, as shown in Figure 5.4. The propagation of the relationship *attachedTo* from `hinge` to `door` is shown in Figure 5.5.

The graphical schema notation for transformational value propagation is similar to that for invariant propagation. The difference is that here we must further provide the family of symmetric operators which define the computation for the derived attribute. To do this, we modify the propagation label slightly. A symbol representing

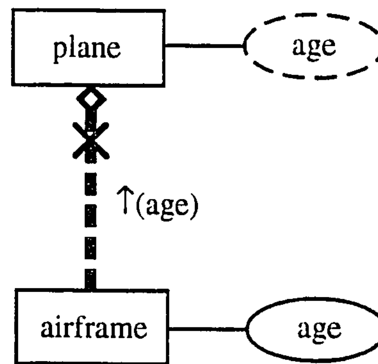


Figure 5.3 The attribute *age* propagating upward from `airframe` to `plane`

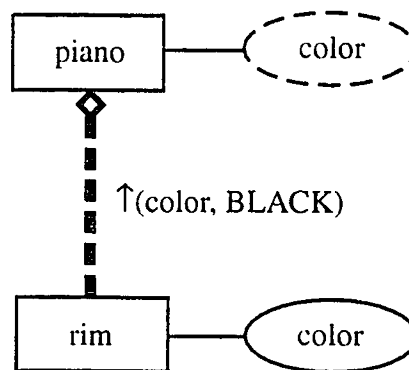


Figure 5.4 Attribute *color* propagating upward from `rim` to `piano`

the entire family is written inside the parentheses in front of the name of the propagated property, which itself now appears in square brackets. Of course, in general, there will be no single graphical symbol for the entire family. In such cases, we will use some generic label (such as "T") and place an annotation for it in a legend for the overall schema. For some common families, such as summation, multiplication, min, max, and so on, whose operators can be decomposed into a closed, commutative, associative, binary operation, we use the ordinary operation symbol in the propagation label: E.g., upward propagating summation is written as $\uparrow(+[\pi])$, and

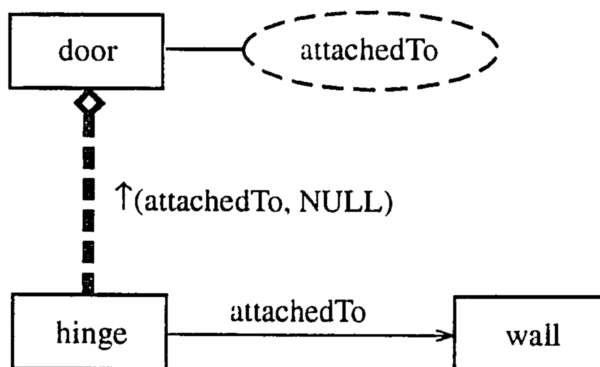


Figure 5.5 The relationship *attachedTo* propagated from hinge to door

multiplication is written as $\uparrow(*[\pi])$, where π is the propagated property. The same holds for min and max. As an illustration, we show that an amplifier derives its reliability measure as the minimum value of that of its transistors in Figure 5.6. The default value is 0.99. For a downward propagating example, we present in Figure 5.7 the schema of a student getting his total number of credits from the sections that he is a part of. The default here is 0.

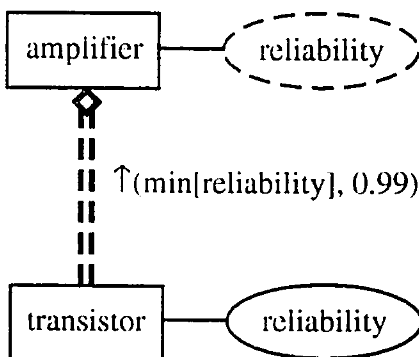


Figure 5.6 The class `amplifier` getting *reliability* from `transistor`

We adopt a special convention for the propagation label for cumulative value propagation. Instead of the parentheses, operator family symbol, and square brack-

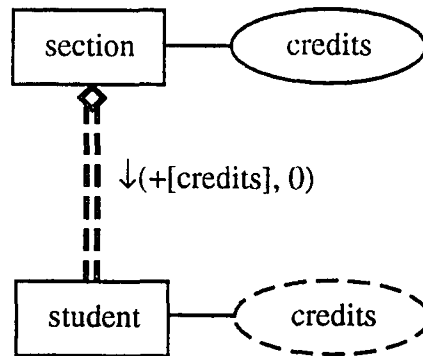


Figure 5.7 Students obtaining their enrollment credits from their sections

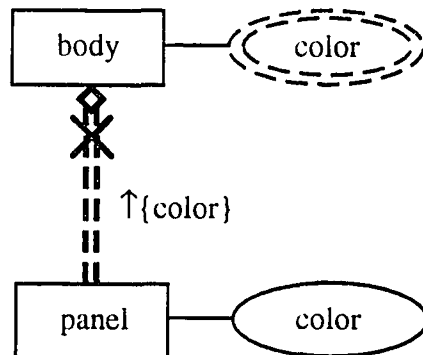


Figure 5.8 Car body getting its color through an upward propagation

ets, the label comprises a pair of curly brackets enclosing the name of the desired property. The graphical representation of the derived attribute is changed to a double ellipse to reflect the fact that it is set-valued. In Figure 5.8, we show the schema for the car body and its panels. The color of the body is not given a default value. For the example of a cumulative propagation of a relationship, we show the schema of the relationship *taughtBy* being propagated from `section` to `student` in Figure 5.9.

For an upward transformational propagation in the presence of ordering where the operators are not symmetric, we alter the propagation label as follows: Instead of

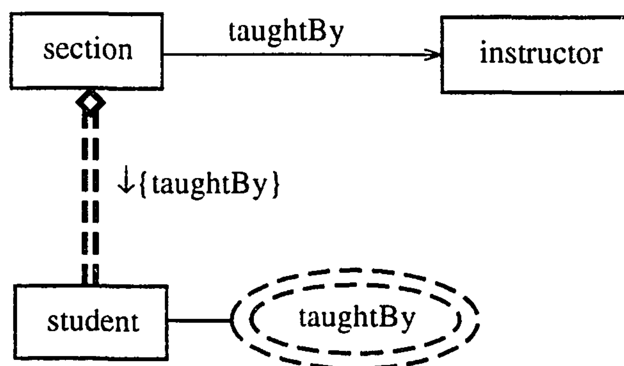


Figure 5.9 The relationship *taughtBy* propagated downward cumulatively

placing a symbol for the operators in front of the property's name, we write the property first, followed, in parentheses, by an expression representing the computation of the derived attribute in terms of the capacities which themselves are written in angled brackets. For the example of the propagation of the date of publication (i.e., the attribute *dateOfPubl*) from the first volume to the collection of complete works as a whole, we use the propagation label ↑(*dateOfPubl*(⟨1⟩)), as seen in Figure 5.10. Here, ⟨1⟩ denotes the volume in the capacity of the first book in the collection.

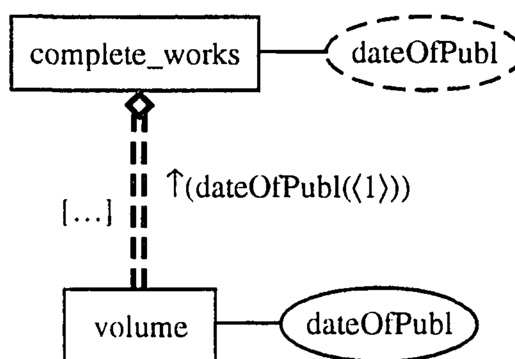


Figure 5.10 The *dateOfPubl* propagated from the first volume to the complete works

5.1.2 Realization of Value Propagation

The realization of value propagation is accomplished by introducing a path method for the derived attribute in the receiving class. This method, called the *propagation method*, is invoked by the message added to the public interface for the derived attribute and is given an identical name. This redundant naming (which is used intentionally to emphasize the correspondence) may cause some confusion in our descriptions below, blurring the distinction between the propagated property and the propagation method used to retrieve it. We will, therefore, alter the propagation method's name slightly by capitalizing its first letter to avoid any confusion (e.g., "Age" instead of "age" below). In what follows, we limit ourselves to the realization of upward value propagation. The entire discussion can easily be recast in terms of downward propagation without further comment.

Assume that we have an invariant upward value propagation of a property π_B (of type τ) from meronymic class B to holonymic class A. The propagation method Π_B which realizes this propagation is added to A and is defined in terms of the selector method $\mathcal{B}$ for the meronym (introduced earlier in Section 4.2) as follows:

$$\Pi_B() : \mathcal{B} \rightarrow B : \pi_B \rightarrow \tau.$$

If we expand $\mathcal{B}$ fully, the method looks like:

$$\Pi_B() : p_B \rightarrow \texttt{B-PART-A} : \textit{meronym} \rightarrow B : \pi_B \rightarrow \tau.$$

In our example of the airframe propagating age to plane, the propagating method "Age" is:

$$\text{Age}(): \quad \mathcal{P}_{\texttt{airframe}} \rightarrow \texttt{airframe-PART-plane} : \textit{meronym} \rightarrow \texttt{airframe} : \text{age} \rightarrow \text{INTEGER}.$$

The method "Age" is illustrated in the schema diagram of Figure 5.11.

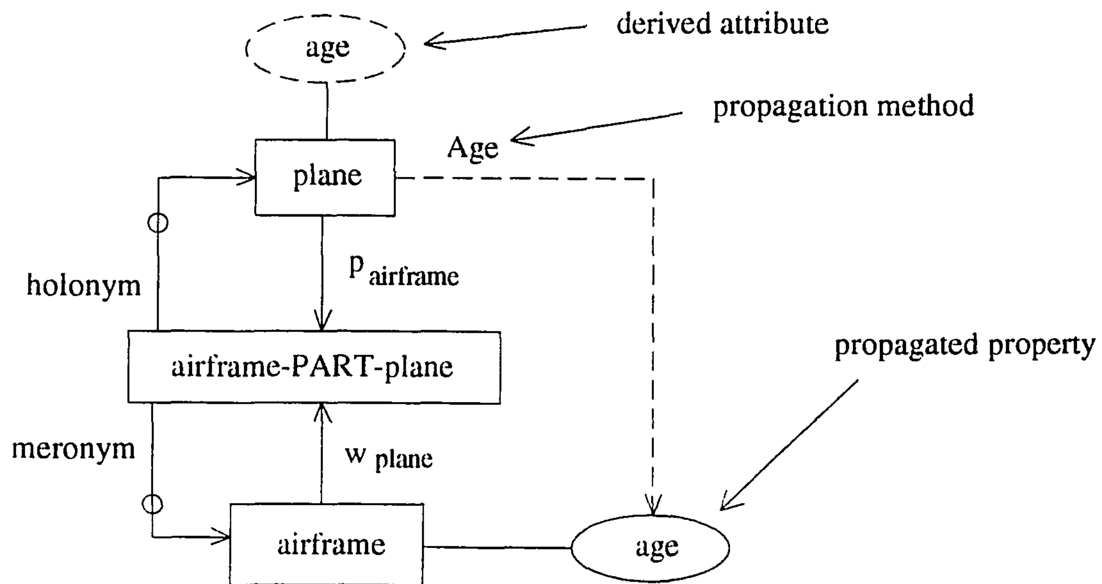


Figure 5.11 Realization of upward value propagation

Transformational upward propagation is realized by a method defined as a composition of the aboved described path method and the specified family of operators. The propagation method "Reliability" for the reliability of an amplifier is given in

the following, where "PERCENT" is the data type of the attribute *reliability*.

$$\text{Reliability}(): \quad p_{\texttt{transistor}} \rightarrow \{\texttt{transistor-PART-amplifier}\} :$$
$$@\mathit{meronym} \rightarrow \{\texttt{transistor}\} : @\text{reliability} \rightarrow \{\text{PERCENT}\}_m :$$
$$\min \rightarrow \text{PERCENT}.$$

Because the relationship $p_{\texttt{transistor}}$ is multivalued, the first transition yields a set of the relational objects. To each of these, the relationship *meronym* is applied, producing a set of transistors. An iterative application of the attribute *reliability* to this set, in the penultimate transition, yields a multiset of percentages. The min operator in the final pair is then taken over this multiset, resulting in a single percentage.

5.2 Interaction of Part Relationship Dimensions

Before moving on to the notion of generalized derived attribute, we give a synopsis of the interactions between the various characteristic dimensions of the part relationship. In particular, we examine the constraints that certain values in one dimension impose on those in another. All these were already mentioned explicitly in the discussions of the various dimensions themselves, but are reiterated here for clarity.

To begin with, we note that the first two characteristic dimensions, the exclusive/shared (E/S) dimension and the cardinality/ordinality (C/O) dimension, are

entirely orthogonal and have no impact on each other. The E/S dimension does not affect the value of the dependency dimension, either. It does, however, have an interaction with the value propagation dimension, which we will review momentarily.

The values *ordered-definite* and *ordered-indefinite* in the C/O dimension cause the base relation $\diamond$ to be refined into finite and infinite sequences of relations, respectively. The C/O dimension is independent of the dependency dimension but does have an interaction with value propagation, also discussed momentarily.

Dependency, as mentioned, is orthogonal to both the E/S and C/O dimensions. In fact, unlike them, it has no interaction with the value propagation dimension and the base relation, and is thus an orthogonal concept to all other dimensions.

Invariant, upward value propagation requires that, for any holonym, the source meronym be unique, if it exists. This is the same as saying that any holonym may have at most one meronym, which is a constraint on the C/O dimension. Therefore, for this type of value propagation, the part relationship must be either single-valued (with $m = 0$ and $n = 1$ in Definition 6) or essential (with $m = n = 1$).

A similar constraint exists for invariant, downward propagation. In this case, though, the unique source requirement falls on the holonyms, meaning that for any meronym, there may be at most one holonym. This translates into the restriction on the ES dimension that it be either global exclusive or class exclusive; sharing in this context is prohibited.

Transformational value propagations in either direction do not impose any restrictions on the meronym or holonym sets of the instances of the participating classes. Thus, they do not constrain the values of the other dimensions.

The interactions of the different dimensions are summarized in Table 5.1. An entry in the table denotes the logical condition(s) characterizing the interaction between the respective dimensions. An "X" denotes no interaction. Note that the table is symmetric; so entries have been omitted from the upper half. One should consult the corresponding entries in the lower half.

Table 5.1 Interaction of part relationship dimensions

dimens. \ *dimens.*	$\Diamond$	Exclusive/Shared (E/S)	Cardinality/Ordinality (C/O)	Dep.	Value Prop.
$\Diamond$			**(Table is symmetric. See lower half for corresponding entry.)**		
E/S	X				
C/O	ODN $\Leftrightarrow \Diamond^{(s)}$ OIN $\Leftrightarrow \{\Diamond^{(i)}\}$	X			
Dep.	X	X	X		
Value Prop.	X	*down* $\Rightarrow$ NOT *limited-shared*	*up* $\Rightarrow$ *single-valued* OR *essential*	X	

5.3 Generalized Derived Attributes

Earlier in this chapter, we introduced different ways to define derived attributes in terms of a single part relationship propagating the value of some property from the meronymic class to the holonymic class, or vice versa. In this section, we extend the notion of derived attribute so that it may be defined with respect to many value

propagations stemming from different part relationships simultaneously. Throughout this section, to simplify the presentation, we will limit the discussion to upward value propagation without any loss of generality.

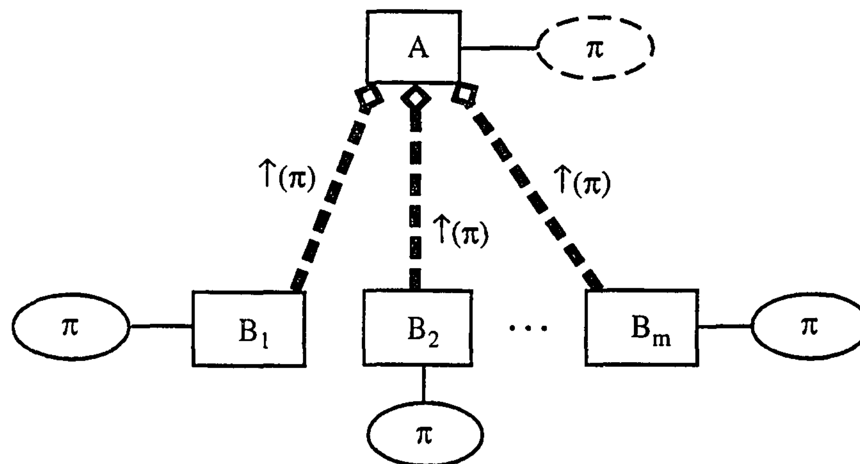


Figure 5.12 Redundant value propagation

An obvious problem that can arise when specifying derived attributes in part hierarchies is similar to the so-called "multiple inheritance" problem in IS-A hierarchies. In Figure 5.12, we illustrate the problem. There, we see multiple part relationships propagating the value of the same property π to a single class A. In this schema, it is not at all obvious what the value of π at A should be; in fact, the schema is not well defined.

To resolve this problem, we could employ one of the two strategies ordinarily used in IS-A hierarchies: Either we could disallow such ambiguous value propagation altogether and consider the schema invalid, or we could employ a precedence list [100, 190]. However, in part hierarchies, there exists a third, more natural solution,

which is not available for IS-A hierarchies. (We note that, at times, the solution can be used in *roleof* hierarchies.) Often, it is sensible to model a property of the whole in terms of the values of the same property at its parts, regardless of their classes. Many examples readily come to mind: the color of an airplane is the combined colors of its fuselage, wings, nose, and tail; the weight of a car is the sum of the weights of its engine, drive train, frame, fenders, and so on; the reliability of a computer is the minimum of that of its monitor, CPU, disk drive, and keyboard; the materials of a golf club are those of its shaft, head, and grip; etc. We therefore view the inherent ambiguity of this "multiple value propagation" as a desired generalization of transformational value propagation across a single part relationship. We resolve it in an analogous manner by combining the multiple values with the use of a specified symmetric transformation. The new derived attribute induced by this process is called a *generalized derived attribute* because it is defined across many part relationships and supersedes those derived attributes which are induced by each part relationship individually. In a pattern mirroring the structure of the part hierarchy itself, the value propagation from each part relationship contributes to the value of the generalized derived attribute at the holonymic class. As before, the holonymic class's public interface is augmented with a message (having the same name as the propagated property) to retrieve the value of this new property for a given whole. Let us now formally define what we mean by a generalized derived attribute and consider the conditions under which it is applicable.

Definition 16: Let π be a property (of data type τ) of the classes $B_1, B_2, \ldots, B_m$ (i.e., for all $1 \leq i \leq m$, π: $E(B_i) \rightarrow \tau$). Assume that we have part relationships $P_{B_1,A}, P_{B_2,A}, \ldots, P_{B_m,A}$ such that each propagates the value of π up to A as in Figure 5.12. Assume also that there does not exist a part relationship $P_{A,Q}$ which propagates a value for π down to A. Furthermore, for all $1 \leq i \leq m$, let $\mathcal{D}_\pi^{(i)}$: $E(B_i) \rightarrow \tau$ be the function defined for $P_{B_i,A}$ which ordinarily serves as the definition for the derived attribute induced by that part relationship. The function $\mathcal{D}_\pi$: $E(A) \rightarrow \tau$, called a *generalized derived attribute*, is defined in terms of the symmetric operator ψ: $\tau^m \rightarrow \tau$ as follows:

$$\mathcal{D}_\pi(a) = \begin{cases} \psi[\mathcal{D}_\pi^{(1)}(a), \mathcal{D}_\pi^{(2)}(a), \ldots, \mathcal{D}_\pi^{(m)}(a)], & \text{if for all } 1 \leq i \leq m,\ \mathcal{D}_\pi^{(i)}(a) \text{ is defined} \\ C, & \text{otherwise.} \end{cases}$$

The generalized derived attribute resolves the redundant value propagations by combining the values of each individual propagation (i.e., the values of the functions $\mathcal{D}_\pi^{(i)}$, $1 \leq i \leq m$) into a single value through the symmetric transformation ψ. In this way, the value of the derived attribute $\mathcal{D}_\pi$ for a given holonym $a \in E(A)$ is now determined by all a's parts participating in a relationship propagating π, regardless of their classes. It should be noted that the definition makes no stipulation regarding the kind of value propagation that an individual part relationship may perform; it may be either invariant or transformational, with the function $\mathcal{D}_\pi^{(i)}$ defined accordingly (as in Definition 12 or Definition 13 above). Therefore, the generalized derived

attribute may obtain a contribution for its value at some whole a invariantly from some lone part of a of a given type, or collectively from multiple parts through a transformation. The only requirement in this respect is that the resultant values from all the value propagations be of the same data type τ. We point out that if the provision excluding a downward propagation of π to A is violated, then the part hierarchy *in toto* is deemed invalid. As with the derived attributes from the previous sections, the generalized derived attribute may be given an optional default value C. If the default is omitted, then the generalized derived attribute may be undefined for certain elements of its domain.

As it stands, Definition 16 contains some restrictions which we adopted to simplify the presentation. These restrictions include the following:

1. The data type of the generalized derived attribute must be identical to the types of the propagated properties.

2. Cumulative propagation is not supported.

3. The data type of the property π at each meronymic class must be identical.

We will now consider how the relaxation of these restrictions generalizes our representation further.

As defined, the function $\mathcal{D}_\pi$ yields a value of data type τ, the type of the propagated properties. In a manner analogous to cumulative value propagation across a single part relationship, we relax the first restriction by allowing the alternate form

$\mathcal{D}_\pi$: $E(A) \rightarrow \{\tau\}$, where the generalized derived attribute is now defined to accumulate each part relationship's propagation value (taken as a singleton set) into a set of values of τ.

Regarding the second restriction, Definition 16 also assumes that all part relationships propagate a value of type τ (i.e., for all $1 \leq i \leq m$, the range of $\mathcal{D}_\pi^{(i)}$ is τ), the type of the property π at each meronymic class. As such, cumulative value propagation is not permitted in this context. To remove this restriction, we allow an alternative form of the operator ψ such that it takes arguments which are sets of values of type τ and, in turn, yields a single set of such values via some symmetric, set-theoretic operation. In this new form, ψ: $\{\tau\}^m \rightarrow \{\tau\}$. And, as in the case of cumulative propagation, the generalized derived attribute becomes set-valued: $\mathcal{D}_\pi$: $E(A) \rightarrow \{\tau\}$. We still do require that the type of value propagated by each part relationship, whether it is atomic or set-valued, be uniform across all relationships.

Although the property π at each meronymic class is taken to be semantically analogous to the same property at all other meronymic classes, the third restriction requiring that each have the same data type is often too limiting. For example, there may be discrepancies in the data types which really should not inhibit our ability to define a generalized derived attribute in terms of these properties. Consider the case where the weight of an airplane's engine is described in kilograms, while its fuselage's weight is given in pounds. Or consider the case where one of the weights is represented as an integer, and the other as a floating-point number. (One can find

similar problems arising in the field of database integration [17, 185].) Because such discrepancies should not impede the definition of the generalized derived attribute, we adopt the following convention: As long as the data types of each π are compatible with each other, either in the sense that they have a common supertype in some type lattice [1] or that they can be cast into one another, then the definition of the generalized derived attribute is deemed valid. However, we still insist that the types of the values propagated by each part relationship be identical, with any required type conversion incorporated into the family of operators $\{T^{(n)}\}$ defined by the schema designer for the individual part relationships.

To carry this point further, it may even be the case that the property π is set-valued at one meronymic class and single-valued in another, as when the fuselage of a plane is multi-colored, while its tail is of a single color. As above, if the values can be type-cast into uniform arguments for ψ (e.g., through a canonical injection of the atomic values into singleton sets), then the third restriction may be relaxed and the definition of the generalized derived attribute is admissible. Again, if one part relationship propagates a set, then all other relationships must do the same.

The graphical representation for a generalized derived attribute is based on the notation used for the derived attributes already presented. The only additional item is the operator ψ, which is handled in a similar manner to the family of operators in the propagation label of transformational value propagation. A symbol representing ψ is placed in front of the derived attribute's name, which is now bracketed, inside

the dashed ellipse. Following our above stated convention, if the family of operators comprises related iterative binary operations, then the symbol for that binary operation (e.g., "+") is employed in the schema representation; otherwise, some generic symbol and an annotation are required. The individual participating part relationships, as we have discussed, may be either invariant or transformational and retain their ordinary propagation labels to indicate their contributions to the computation of the generalized derived attribute's value. If the generalized derived attribute has a default value defined for it, then this is placed after its name inside the ellipse.

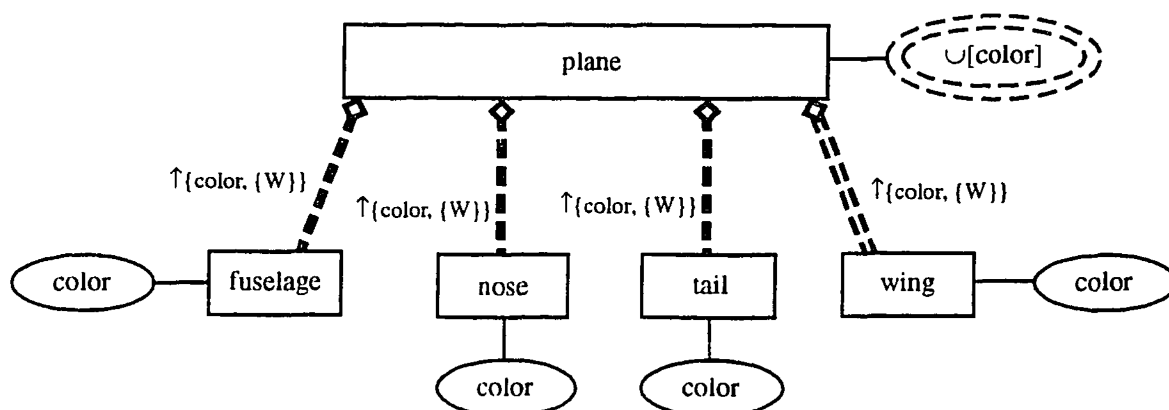


Figure 5.13 Plane getting its colors from its fuselage, wings, nose, and tail

In Figure 5.13, we show how the color of a plane is defined as the union of the colors of its fuselage, wings, nose, and tail. The propagation from the class `wing` is a cumulative propagation because there may be several wing instances per plane. Because of this, the propagations from `fuselage`, `nose`, and `tail` require the transformation of the single color value into a singleton set. The derived attribute *color* of `plane`, being multivalued in general, is depicted by a double ellipse. The

"∪" in front of its bracketed name indicates that the operator family is that of the set-union operators. Therefore, the value of *color* (for a given instance of `plane`) is the union of the sets of colors propagated to it through the four respective part relationships. Note that, by default, each part relationship sends a singleton set containing the color white (represented by "W"), so there is no need to give the generalized derived attribute *color* at the class `plane` a default of its own.

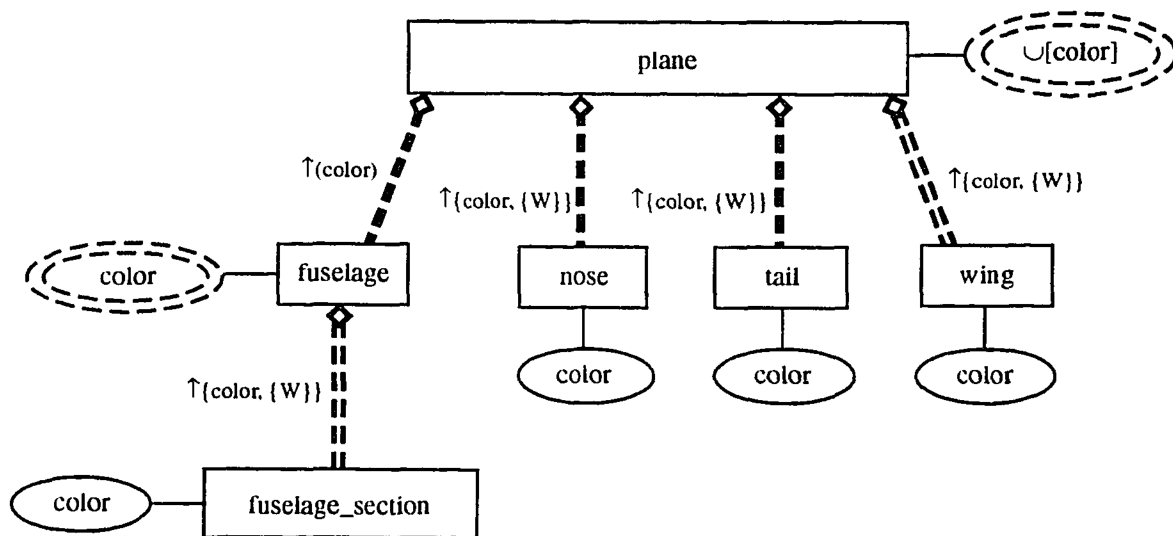


Figure 5.14 Fuselage getting its own color propagated from its constituent sections

To demonstrate that derived attributes may be manipulated in the same way as other class properties, let us consider a revised version of the above airplane schema where the description of fuselage is further refined into a set of constituent sections (Figure 5.14). Now, the property *color* of the class `fuselage` is itself a set-valued, derived attribute defined with respect to a cumulative value propagation from the class `fuselage_section`. Because of this, the propagation of *color* from `fuselage` to

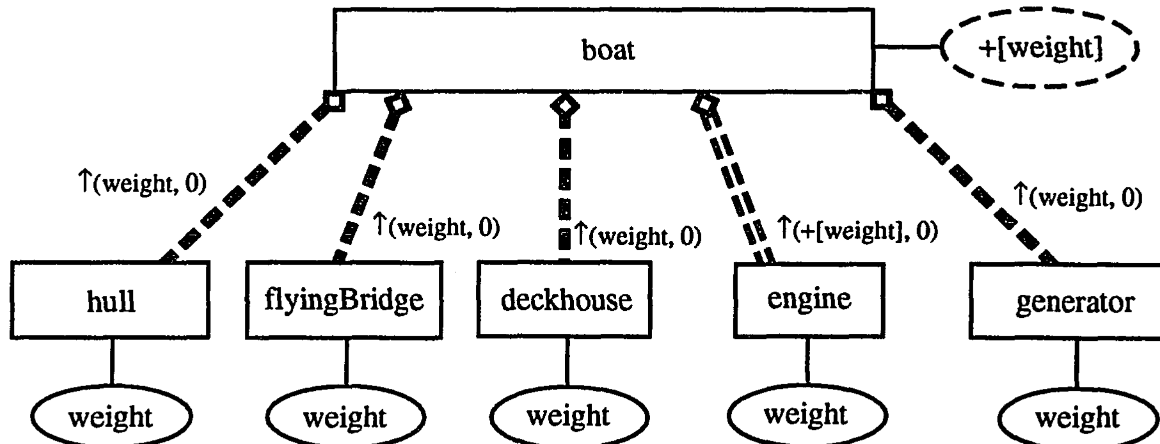
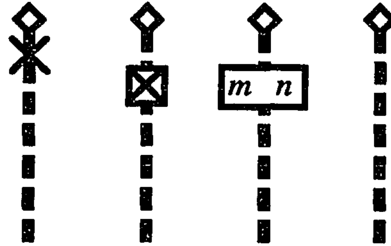


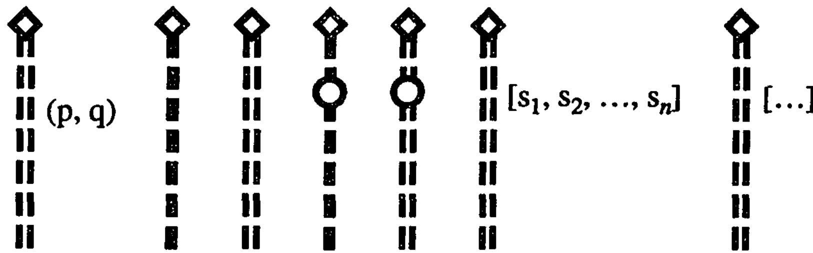
Figure 5.15 The weight of a boat as the sum of the weights of its parts

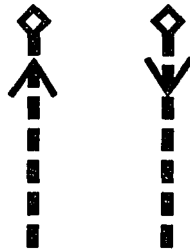
`plane` is no longer transformational but rather invariant as indicated by the ordinary parentheses enclosing "color" in the propagation label. The value propagation from `wing` remains cumulative, and, as before, the propagations from `nose` and `tail` must be cumulative in order to bring their data types in line with the other two.

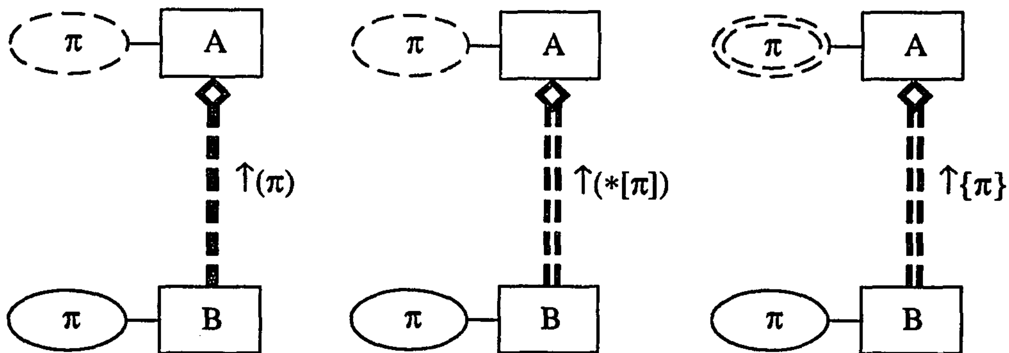
In our final example (Figure 5.15), we show how a boat's weight may be written as the sum of the weights of its parts. Let us observe a few subtleties of this schema. First, the propagation label $\uparrow(+[\text{weight}], 0)$ of the part relationship between `engine`

Table 5.2 The part relationship symbols by dimension









and `boat` indicates a transformational value propagation, whose contribution is a single weight value derived as the sum of the weights of all engines (of a given boat). In contrast, the similar expression +[weight] appearing in the symbol for the generalized derived attribute at class `boat` means that the weight of a specific boat is the sum of the weights propagated to it from the classes `hull`, `engine`, `deckhouse`, and so forth. Note that the default value of each of the value propagations is 0. So, a boat without any parts has no weight.

The realization for a generalized derived attribute is a method which is a straightforward combination of those which otherwise would have been defined for each of the part relationships separately (see Section 5.1). For example, the realization for the derived attribute *weight* of class `boat` is defined as the sum of the results of the path methods to retrieve the weights from `hull`, `flyingBridge`, and so on.

At this point, we would like to summarize all the graphical notation that we have introduced for the part relationship. Refer to Table 5.2 where we have categorized the symbols according to their characteristic dimensions. At the top is the exclusive/shared dimension where we show, from left to right, the symbols for: global exclusiveness, class exclusiveness, limited sharing, and unrestricted sharing. Next is the cardinality/ordinality dimension, where again from left to right we have part relationships exhibiting range-restriction, single-valuedness, multivaluedness, single-valued essentiality, multivalued essentiality, ordering of definite number, and ordering of indefinite number. The dependency dimension has two symbols: part-to-whole

dependency is depicted with an arrowhead pointing to the holonymic class (which is allows located at the end of the connection where the diamond head is situated); whole-to-part dependency is denoted by an arrowhead pointing in the opposite direction. The bottom of the table shows the symbols associated with value propagation, namely, the propagation labels and derived attributes. On the left is the propagation label for invariant propagation. which here indicates the upward propagation of the property π from class B to class A. The derived attribute at A is drawn with a dashed ellipse. In the middle, we show the upward transformational propagation of π involving the generic operator $*$. Finally, on the right side, we see an example of cumulative value propagation. Note that the derived attribute at A is drawn with a double, dashed ellipse to indicate that it is set-valued.

Before moving on to the realization of the part relationship using metaclasses, we present a part schema in Figure 5.16 which models the editorial page of *The New York Times*. Some things to note about the schema: First, the *masthead* is the item on the editorial page which contains information such as the newspaper's official name, its publisher, and the editorial staff. As these items change infrequently, the masthead may remain the same for a number of years and be shared by many papers. The *business masthead*, found among the letters, displays information regarding the newspaper's ownership. It, too, changes very rarely and is shared. One will note that some parts obtain their date through a value propagation, while others inherently contain this property. For example, editorials receive their date

through a sequence of propagations from the newspaper as a whole. (In fact, the class **newspaper** is included solely for this reason; as indicated by the ellipsis, its description is incomplete.) On the other hand, letters are dated independently of the paper and, therefore, have their own attribute **date**.

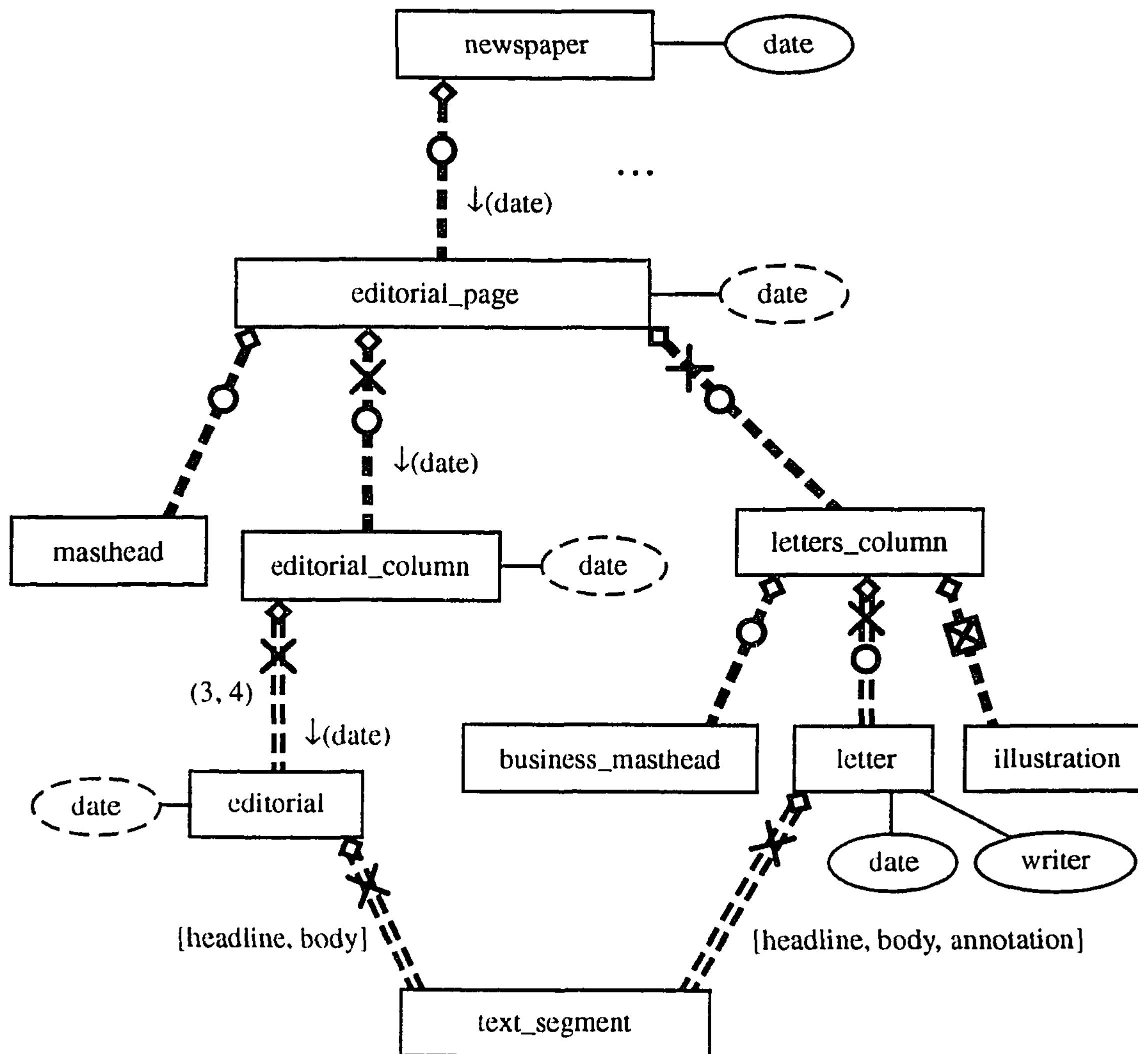


Figure 5.16 *The New York Times* editorial page

CHAPTER 6

IMPLEMENTING THE PART MODEL USING METACLASSES IN VML

In this chapter, we address the issue of incorporating a realization of our part model into an existing OODB without having to rewrite a meaningful subsystem of the OODB and without causing a fundamental upheaval in its underlying data model. Specifically, we show how this can be done in the context of an OODB with an open architecture designed to anticipate such additions. The VODAK Model Language (VML) [56, 109] was built with a metaclass subsystem [108] to facilitate extensions and allow for the customization of its data model in terms of new semantic relationships [108]. We have availed ourselves of this metaclass mechanism and built a custom metaclass called the "HolonymicMeronymic" metaclass that captures the semantics of classes which participate in part relationships and part hierarchies. The entire VML code specification for our metaclass can be found in Appendix A.

The work reported in this chapter also addresses another important issue, and that is whether or not the VML OODB, with its open architecture and metaclass mechanism, can support the introduction of a part model extension. The implementation of the HolonymicMeronymic metaclass described in this chapter and the VML code which appears in Appendix A demonstrate that such an extension is indeed possible.

For a realization of our part model in an OODB without metaclasses, we refer the reader back to Chapters 4 and 5, where we have presented one that only requires

the use of basic OODB constructs. Presently, that realization is being carried out in a Smalltalk environment as part of a student's Master's thesis [172].

The chapter is organized as follows. First, we give an overview of the VML data model. The focus will be on the use of metaclasses to implement various semantic, generic relationships. After that, we go on to discuss the different aspects of the HolonymicMeronymic metaclass. Specifically, we first describe the metaclass's "instance type" which affects the behavior of its instances which themselves are classes. This instance type captures the creation and deletion semantics of parts and wholes through the definition of two methods, *make* and *destroy*. Finally, we present the metaclass's "instance-instance type" which affects the structure and behavior of instances of the metaclass's instances which, as noted, are classes in a part hierarchy. In this manner, such instances of the metaclass's instances are given the "look and feel" of parts and wholes with respect to each other. In particular, they are given methods (defined in the instance-instance type) that allow them to be updated and queried as parts and wholes.

6.1 The VML Data Model and Metaclasses

VML employs a variant of the Dual Model [65, 143] to describe the structure and semantics of the classes and objects of an OODB. The duality arises through the separation of the notions of class and object type [69]. Each class in the schema is associated with exactly one object type, referred to as the *instance type*, which defines the structure and behavior of the instances of the class. This is illustrated in

Figure 6.1 where we have used a shading pattern to show the effect of the instance type on an instance of the class. A single object type, on the other hand, may be associated with any number of classes. The benefits of such an arrangement have been discussed in detail in [143, 142, 65].

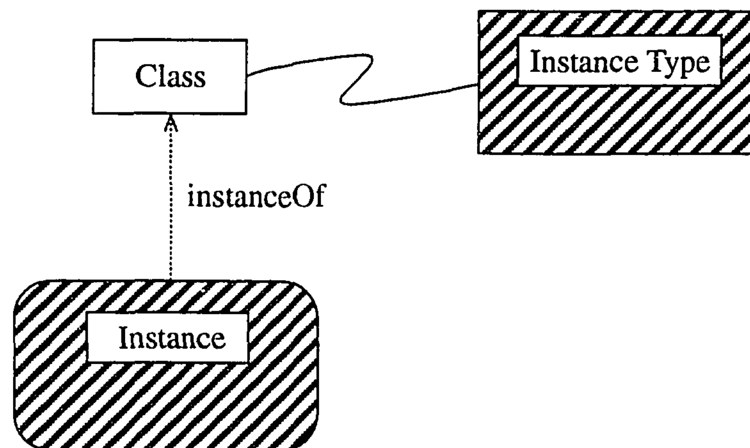


Figure 6.1 The instance type's effect on a class's instances

To maintain uniformity in the data model, all classes are considered objects in VML (cf. Smalltalk [73]). As such, classes themselves are instances of other classes, which are referred to as *metaclasses*. However, as described in [108, 109], the interaction between types, classes, and metaclasses has a different character than that between types, instances, and classes. Just as with an ordinary class, a metaclass has an associated object type that describes its instances, which in this case are classes. This object type is, as above, referred to as the *instance type* of the metaclass. Furthermore, as an interesting and powerful extension, one may associate a second object type with a metaclass to augment the structure and

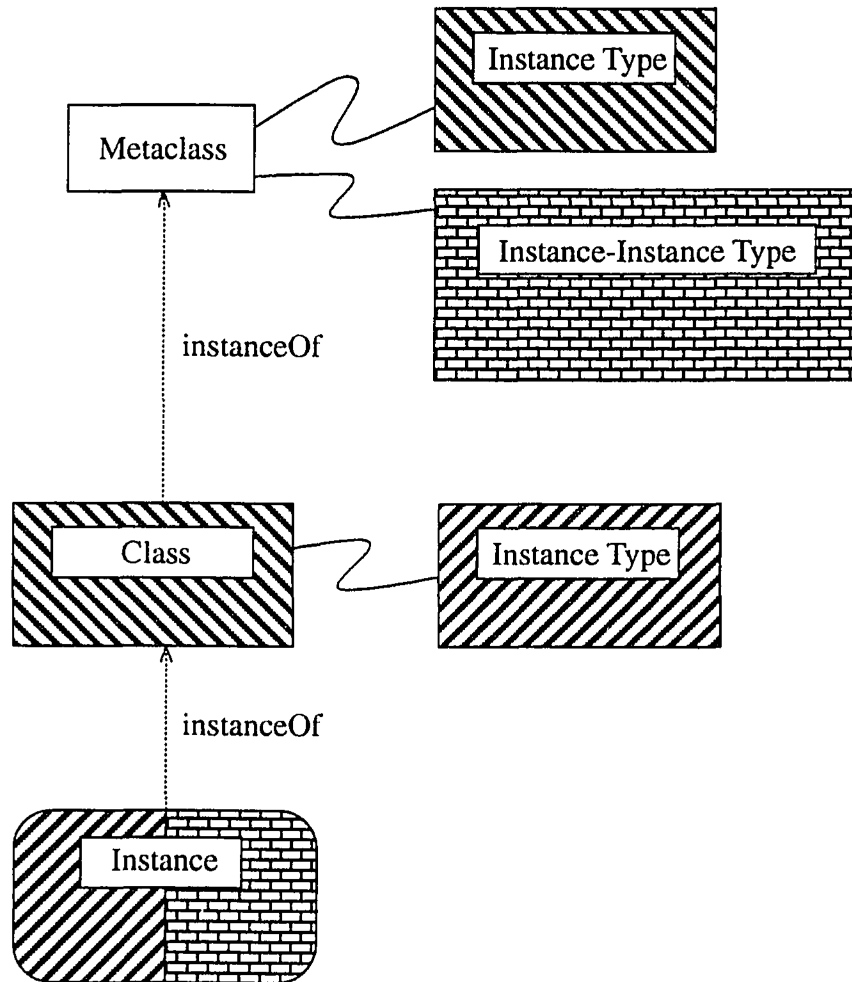


Figure 6.2 The interaction between metaclasses, classes, and instances

behavior of the instances of the classes which are instances of the metaclass. This object type has been given the colorful name *instance-instance type* [108]. Thus, through its two associated object types, a metaclass influences the structure and behavior of both its own instances, which are classes, and the instances of these classes. This arrangement is illustrated in Figure 6.2 where we have again employed shading patterns to demonstrate (a) the effect of the metaclass's instance type on

a class and (b) the effect of the metaclass's instance-instance type and the class's instance type on the class's own instances.

In VML, one does not actually define a separate entity to represent a generic, semantic relationship, such as the part relationship, as we have done in previous chapters. Instead, one creates a new, custom metaclass which endows its instances (which are classes) and their instances, in turn, with structure and behavior befitting the semantic relationship of interest. In this manner, the VML data model is an open model which can be tailored to the needs of specific applications or whole application domains in general.

Custom metaclasses introduced into the VML model are always defined as subclasses of METACLASS, the root of the VML metaclass hierarchy, or one of its subclasses. We will refer to METACLASS simply as M. As the root of the metaclass hierarchy M defines the basic behavior for all system classes and objects. For example, it provides all classes with the methods *new* and *delInstance* that allow classes, respectively, to create and destroy their instances. Through M's instance-instance type, each instance gets the method *class*, which may be used to determine its class. Actually, the custom metaclass is not directly made a subclass of M. Rather, one defines the new instance type and instance-instance type as subtypes of the corresponding types of M. This scheme effectively captures the subclass relationship between the metaclasses and permits the requisite property inheritance.

We have expanded the base VML object model to include a part-whole semantic relationship by defining the HolonymicMeronymic metaclass. Any class participating in a part hierarchy is defined as an instance of this metaclass. We will refer to such classes as HolonymicMeronymic (HM) classes. Through its instance type and instance-instance type, the metaclass does the following for HM classes and their instances:

- It provides the means for establishing a part relationship between a pair of HM classes, making one a holonymic class and the other a meronymic class.

- It furnishes an HM class with methods *make* and *destroy* which replace the standard methods *new* and *delInstance*. (Note: The current version of VML does not allow overriding of methods or method combination, as would be appropriate in this situation. To deal with this, we have chosen this renaming scheme to avoid any identifier conflicts and have defined the methods *make* and *destroy* as "front-ends" to the standard methods.)

- It provides an HM class's instances with a standard palette of methods for updating and querying with respect to the various part relationships that the class participates in. Such methods include *addPart*, *removePart*, and *getParts*.

- It provides the means for performing upward and downward value propagation across part relationships.

In succeeding sections, we go on to discuss the details of the instance type and the instance-instance type of the HolonymicMeronymic metaclass and describe what each of these contributes. Included are discussions of the various methods that each defines.

6.2 The HolonymicMeronymic Instance Type

In this section, we describe the details of the HolonymicMeronymic instance type which augments the structure and behavior of classes participating in part hierarchies (i.e., HM classes). We first present the actual public interface of the instance type which represents the instance type's contribution to the public interface of any HM class. In subsequent sections, we discuss the ways in which these methods capture some of the semantics of the part relationship.

The public interface for the HolonymicMeronymic instance type is as follows. (We include the object type's signature on the first line for clarity. Note that it is defined as a subtype of "Metaclass_InstType," which is the instance type of METACLASS.)

```
OBJECTTYPE HolonymicMeronymic_InstType SUBTYPEOF Metaclass_InstType;
INTERFACE
   METHODS
    make(someParts: {OID}): OID READONLY; // replaces method "new"
    destroy(anObject : OID) READONLY;     // replaces "delInstance"
    defMeronymicRelshps(someRelshps: {PartRelationshipType})
                                                       READONLY;
    defHolonymicClasses(someClasses: {OID}) READONLY;
    isMeronymicClassOf(aClass: OID): BOOL READONLY;
    isHolonymicClassOf(aClass: OID): BOOL READONLY;
    getMeronymicClassesOf(): {OID} READONLY;
```

```
getHolonymicClassesOf(): {OID} READONLY;
exsh(aClass: OID): ExShType READONLY;
minCard(aClass: OID): INT READONLY;
maxCard(aClass: OID): INT READONLY;
dependencyStatus(aClass: OID): DependType READONLY;
propertyUpPropagated(meth: STRING, aClass: OID): BOOL READONLY;
propertyDownPropagated(meth: STRING, aClass: OID): BOOL READONLY;
```

This public interface shows that the HolonymicMeronymic instance type provides fourteen new methods for all HM classes. Remember, a class in VML is itself an object, and these methods augment the behavior of a class, not the behavior of the class's instances. As we have discussed above, the methods *make* and *destroy* are used, respectively, to create and delete instances of an HM class, and encode the creation and deletion semantics of the part relationship. Each one, in turn, is discussed in detail in Sections 6.2.2 and 6.2.3. The remainder of the methods are used to establish and obtain information about the actual part relationships that an HM class participates in. All are described further in the next section.

6.2.1 Creating and Querying an HM Class

Before we begin our discussion of HM classes, let us first consider the task of creating an "ordinary" application class [109] in VML, i.e., one without any part relationships. This will help to demonstrate the exact impact of introducing part relationships into class definitions. Assume that we are working in an automotive manufacturing environment and wish to define a class car which has the schema illustrated in Figure 6.3. There, we see that car has three attributes *serialNumber*, *model*, and *year*, and a single relationship *manufacturedBy* to the class company.

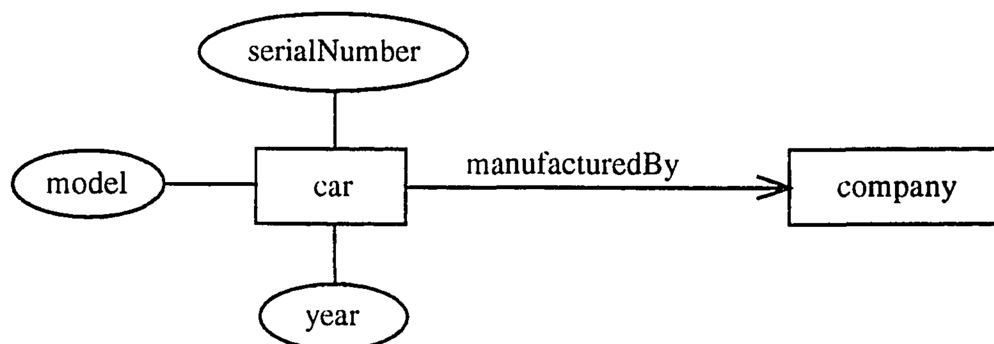


Figure 6.3 The class **car** without part relationships

Recall that in VML, an object type is associated with each class as its instance type in order to define the structure and behavior of the class's instances. Therefore, the complete specification of a class is divided into two portions, an object type declaration and the class declaration itself which contains a reference, via its "INSTTYPE" clause, to the object type. To illustrate this, we show the VML code for the class **car** in the following.

```
CLASS Car
    INSTTYPE carType
END;

OBJECTTYPE carType;
IMPLEMENTATION
   PROPERTIES
    serialNumber: INT;
    model: modelType;
    year: yearType;
    manufacturedBy: Company;
END;
```

At the top, we see the class declaration which is denoted by the keyword "CLASS." Its initial line is used to convey two pieces of information. The first is the class's

name, written immediately after the keyword. By convention, the names of classes in VML are capitalized, so we have written "`Car`" instead of "`car`." The second item, which happens to be optional, is the name of the class's metaclass. In the specification, this is preceded by the reserved word "METACLASS," and the two together appear after the class's name. If the metaclass is omitted, as is the case here, then the class is taken to be an instance of the default KERNEL-APPLICATION-METACLASS [109] which is used for ordinary application classes (i.e., those without semantic relationships).

The instance type of `Car` is the object type **carType**, as indicated by the "INST-TYPE" clause on the second line. In the declaration of the object type,[1] shown below the class declaration, we see the definition of the four properties from Figure 6.3. In VML, there is no syntactical distinction between attributes and relationships, and they are written together in the "PROPERTIES" section. However, the three attributes are followed by appropriate data types, while the relationship *manufacturedBy* is instead followed by the name of its referent class `Company` (not shown here).

To define a class as a member of a part hierarchy (i.e., as an HM class), one does two things. First, one defines the class to be an instance of the Holonymic-Meronymic metaclass (using the METACLASS keyword on the opening line). Second, one invokes, through the class's "INIT" clause, the two companion methods *defMeronymicRelshps* and *defHolonymicClasses* to establish the desired part rela-

[1] We have omitted some of the details of the object type declaration that are not relevant here. See [109] for a description of the complete syntax.

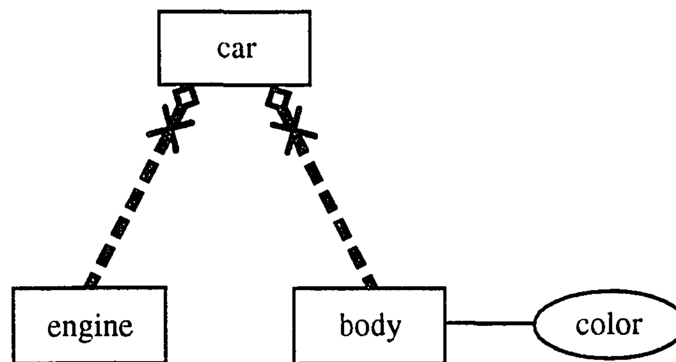


Figure 6.4 Car and its parts engine and body

tionships between the new class and other HM classes. Specifically, the method *defHolonymicClasses* informs the class of all classes which are holonymic classes with respect to it in part relationships. The method *defMeronymicRelshps*, on the other hand, informs the class of all its related meronymic classes. The naming discrepancy between the two (i.e., "Classes" versus "Relshps") denotes the fact that, besides the names of the meronymic classes, *defMeronymicRelshps* carries additional information pertaining to the characteristic dimensions of the respective part relationships. In particular, it provides the values of all characteristic dimensions to the new class, which, of course, plays the role of the holonymic class in such part relationships. This arrangement is necessitated by the fact that, in our implementation, we have decided to store all such information about a part relationship with the class participating in it as the holonymic class. While we could have done otherwise (e.g., we could have relegated all information to the meronymic classes), we believe that the inherent bottom-up construction associated with part hierarchies—where inte-

gral objects are built-up from lower-level component objects—makes this a natural choice.

To illustrate the above points, let us expand our earlier example and define **car** as the holonymic class in two part relationships, one with the class **engine** and the other with the class **body**. These classes and the part relationships are shown graphically in the schema of Figure 6.4. There, we have omitted all the properties of **car** presented earlier and have included only one attribute, *color* of the class **body**, which we will later propagate upward. The VML syntax corresponding to this schema is as follows. Note that we have not included the declarations of the instance types **carType**, **engineType**, and **bodyType** because they are not relevant to the discussion.

```
CLASS Car METACLASS HolonymicMeronymicClass
    INSTTYPE carType
    INIT Car->defMeronymicRelshps(
                                  {
                                   [theMeronymicClass:Engine,
                                    es:GLOBAL_EXCL,
                                    cardinality:[min: 0, max: 1],
                                    dependency:NONE,
                                    upSet:{},
                                    downSet:{}],
                                   [theMeronymicClass:Body,
                                    es:GLOBAL_EXCL,
                                    cardinality:[min: 0, max: 1],
                                    dependency:NONE,
                                    upSet:{},
                                    downSet:{}]
                                  }
                                 )
END;
```

```
CLASS Engine METACLASS HolonymicMeronymicClass
   INSTTYPE engineType
   INIT Engine->defHolonymicClasses( {Car} )
END;

CLASS Body METACLASS HolonymicMeronymicClass
   INSTTYPE bodyType
   INIT Body->defHolonymicClasses( {Car} )
END;
```

As we see, the first line of the declaration of `Car`—as well as the first lines of `Engine` and `Body`—now contains an explicit reference to the HolonymicMeronymic metaclass, which, in its actual code specification, has been given the name "HolonymicMeronymicClass" in order to follow VML conventions. (See the appendix for all the details of the code; in our discussions, we will continue to use the shorter "HolonymicMeronymic.") Thus, `Car`, `Engine`, and `Body` are all instances of the HolonymicMeronymic metaclass (i.e., they are all HM classes).

The INIT clause is a characteristic of all classes in VML. It is used as a means for performing certain initialization procedures at the time the class is created (i.e., instantiated). In our case, as was mentioned, we use it to invoke the methods *defHolonymicClasses* and *defMeronymicRelshps* in order to establish the part relationships between HM classes. It will be noted that for a class that has no associated holonymic classes (i.e., a class which is the root of a part hierarchy), such as `Car` in the example, the method *defHolonymicClasses* is not needed in the INIT clause. For those classes with no associated meronymic classes (i.e., leaves of a part hierarchy),

the method *defMeronymicRelshps* is not included. Such is the case for the classes `Engine` and `Body`.

The argument to *defHolonymicClasses*, as can be seen in the declarations of `Engine` and `Body`, is the set of holonymic classes of the given class. In the example, we see that both `Engine` and `Body` have the single holonymic class `Car`, which is passed as a singleton set to *defHolonymicClasses* in their INIT clauses.

The argument to *defMeronymicRelshps* is more complicated because the values of the characteristic dimensions of the respective part relationships must accompany each meronymic class. Therefore, rather than just being a set of classes, the argument is a set of structures of type "PartRelationshipType" having the following definition:

```
DATATYPE PartRelationshipType = [theMeronymicClass: OID,
                                  es: ExShType,
                                  cardinality: CardType,
                                  dependency: DependType,
                                  upSet: {STRING},
                                  downSet: {STRING}];
```

The first member of the structure is the name of the meronymic class which, in VML, is just an alias for the class's OID—hence, the data type "OID" for this member. (Remember, classes in VML are objects, too.) The second, third, and fourth members hold the values for the first three characteristic dimensions of the part relationship. In the declaration of `car` above, we see that its part relationship with `engine` is global exclusive (represented by the symbolic constant GLOBAL_EXCL)

and single-valued (represented by a minimum cardinality of 0 and a maximum cardinality of 1). It also lacks any dependency semantics as specified by NONE, which is used instead of *nil*.

The last two members of the structure represent, respectively, the set of properties being propagated upward across the part relationship and the set of properties being propagated downward. As we have mentioned in the previous chapter, these sets must be disjoint. One will note that the properties are specified as VML strings. We will see how this is exploited when we discuss the realization of value propagation using the NOMETHOD clause below. In the example, we notice that both sets are empty, meaning that no propagation takes place with respect to this specific part relationship.

Because all the information about a part relationship is stored with its respective holonymic class, it is necessary to directly query that class in order to obtain such information. This can be done with certain methods that are provided by the HolonymicMeronymic instance type. These are described in the following. The methods *isMeronymicClassOf* and *isHolonymicClassOf* are predicates which determine, respectively, whether or not the target class is a meronymic class or a holonymic class of the class given as an argument. The related methods *getMeronymicClassesOf* and *getHolonymicClassesOf* return the meronymic classes and the holonymic classes of the target class, respectively. To determine the values of the various characteristic dimensions of the part relationship between a meronymic class B and holonymic

class A, the last six methods of the HolonymicMeronymic instance type can be invoked for A with B as their argument. The method *exsh* returns the value of the part relationship's exclusive/shared dimension, which may be GLOBAL_EXCL, CLASS_EXCL, or SHARED. To obtain cardinality information, we use the methods *minCard* and *maxCard*, each of which returns an integer. The maximum cardinality may have the symbolic value INFINITY indicating the absence of an upper bound. Dependency information is gathered through *dependencyStatus* whose possible values are PART_TO_WHOLE, WHOLE_TO_PART, and NONE. As mentioned above, NONE denotes the lack of any dependency semantics. The final two methods, *propertyUpPropagated* and *propertyDownPropagated*, are predicates used by the NOMETHOD clause to perform value propagation. We defer a discussion of them to below.

6.2.2 Capturing the Creation Semantics of the Part Relationship using *make*

The method *make* defined by the HolonymicMeronymic instance type as a replacement for the method *new* is the means by which instances of an HM class are created. Encoded in this method are the creation semantics dictated by the various characteristic dimensions of the part relationships which we will consider shortly. To create an instance of an HM class, one simply invokes *make* for that class. In VML, a method invocation is denoted using an arrow notation: An alias for the target object's OID

(such as a class name or variable) is followed by "->" and the name of the desired method. So, to create a new instance of **car**, we write:

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

Since the method returns the OID of the newly created object (or the constant NULL on failure), we ordinarily find such an expression in the midst of a VML assignment statement.

From its signature, we see that *make* takes as its argument a set of OIDs. This set comprises parts which are to be initially attached to the new object. The set is heterogeneous in that it may contain the OIDs of objects of any type. Obviously, if any of the objects is from a class other than the prescribed meronymic classes of the target class, then the creation of the new instance is aborted. In the above invocation, the argument is just the empty set, so no engine is installed in the new car initially.

The method *make* is responsible for monitoring two constraints. First, it must ensure that any cardinality constraints imposed by the part relationships of the target class are satisfied. In other words, if we wish to create an instance of class A and the part relationship $P_{B,A}$ requires that such an instance have between, say, m and n parts from B, then it is the responsibility of *make* to ensure that this constraint holds at the outset of the instance's lifetime. (As we will see below, the methods *addPart* and *removePart* of the instance itself carry this same responsibility for the remainder of the instance's lifetime.) If, among the given set of initial parts, *make* detects a

violation of the constraint (e.g., there are too few or too many parts of type B), then it aborts the creation and returns NULL, a special VML constant. Since using *make* is the only way to create an instance of an HM class, this arrangement guarantees that no holonym comes into existence with any of its cardinality constraints in violation.

The second constraint, related to the first, involves the semantics of exclusiveness and sharing. Even if *make* receives a valid number of parts from a meronymic class, it may still be the case that some of these may not be legally installed as parts of the new instance. This can happen if a given meronym is already part of another holonym having exclusive ownership of it and thus precluding the desired installation. This possibility is also handled by *make* which, if it discovers such a problem, aborts the instantiation.

6.2.3 Capturing the Deletion Semantics of the Part Relationship using *destroy*

Instances of an HM class are deleted using the method *destroy* instead of VML's customary *delInstance*. To delete an instance c of the class car, we invoke *destroy* as follows:

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

The argument c will ordinarily be a variable of VML data type OID holding a reference to the object of interest.

Analogously to *make*, *destroy* encodes the part relationship's deletion semantics, which are primarily dictated by the first three characteristic dimensions. Other

external factors also exert influence. As was discussed in Chapter 4, we must take into account the possible conflicts that can arise between the dependency specification of one part relationship and the cardinality constraints of another. Also, in order to conform to our alternative realization using classes, we require that all part-whole connections of a given instance be dissolved before that instance is allowed to be deleted. This latter behavior is overridden by dependency.

Because *destroy* directly encodes the deletion semantics of the part relationship, it is important to understand its operation. Assuming that it is given the instance O of some HM class to delete, *destroy* operates in accordance with the following five cases:

1. O does not participate in any part relationships.

   In this case, it is immediately deleted.

2. O participates either as a meronym or as a holonym in a part relationship where there is no dependency.

   In this case, the deletion of O is disallowed. Any such part connections must be broken explicitly using the method *removePart* or *changePart* (discussed below) before the deletion is allowed to take place.

   As an exception to this case, O is allowed to be deleted if it is strictly a holonym (i.e., it is not a meronym with respect to any existing instance) and the part relationships it is participating in all have minimum cardinality constraints

greater than zero. In such a scenario, there is absolutely no way to break all of O's part-whole connections prior to invoking *destroy*. (At some point, the method *removePart* would forbid a part removal because it would leave O in violation of a cardinality constraint.) Thus, we would have the unacceptable situation that the instance O could never be deleted from the database.

3. O participates in a part relationship where it is dependent on the associated object, be it a holonym or meronym.

   As in Case (2), the deletion of O is disallowed; it can only take place after all such connections have been undone explicitly using *removePart* or *changePart*.

4. O is a meronym with respect to a part relationship that has a minimum cardinality greater than zero.

   In such a case, the deletion of O may cause the cardinality constraint to be violated. Therefore, we defer to that constraint and do not allow the deletion of O. (Cf. Case [5, (b)] below.)

5. O participates only in part-whole relationships where the associated objects, be they holonyms or meronyms, are dependent on it.

   In this case, O is deleted. Furthermore, for each object Q that is dependent on O, we have the following subcases. [In the following, we assume that Q is a part; analogous conditions hold for wholes. Case (b), in fact, is identical for both parts and wholes.]

(a) Q is part of another object O' in O's class.

Here, Q is not deleted. This case captures the multivalued dependency semantics which we ascribed to the part relationship (Section 4.5).

(b) Q is part of an object W which has a minimum cardinality restriction greater than zero with respect to the part relationship between them.

In this case, propagating the deletion to Q may leave W without a required part, as was the case in the scenario discussed in Section 4.5. To resolve such potential conflicts between dependency and cardinality constraints, we defer to the cardinality constraint and do not delete Q. We refer to this as *the conservation of cardinality constraints.*

(c) Q is not part of any such instances as described in (a) and (b).

In this circumstance, the deletion is propagated to Q with the following actions taken in order:

i. All of Q's part-whole connections with objects which are not dependent on it are explicitly broken.

ii. The method *destroy* is then invoked for the deletion of Q. Note that by performing the disconnections in Step (i), we ensure that either Case (1) or Case (5) is applicable on the iterative invocation of *destroy*. Thus, Q is certain to be deleted. All objects dependent on it (if any exist) will be handled by an iterative application of the present case.

Again, one will note that this method is in charge of enforcing the semantics of dependency as laid out in Case (5). If an object in a part-whole relationship is dependent on O and certain other conditions are satisfied, then it too is deleted. It should also be noted that while the method *destroy* is invoked from the body of *destroy*, this is *not* a recursive call. Rather, it is iterative because the reference is to the *destroy* method of another HM class.

6.3 The HolonymicMeronymic Instance-Instance Type

In this section, we cover the details of the instance-instance type of the HolonymicMeronymic metaclass. In a centralized and uniform manner, this object type endows instances in a part hierarchy (i.e., meronyms and holonyms) with structure and behavior consistent with the semantics of the part relationship. Specifically, it gives such instances the ability to establish and break part-whole connections with other instances. It also allows those connections to be changed or queried. And furthermore, through its NOMETHOD clause, it provides the means by which value propagation is accomplished.

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

```
OBJECTTYPE HolonymicMeronymic_InstInstType
                       SUBTYPEOF Metaclass_InstInstType;
INTERFACE
   METHODS
    addPart(aPart: OID): BOOL READONLY;
    addWhole(aWhole: OID) READONLY;
    removePart(aPart: OID): BOOL READONLY;
```

```
removeWhole(aWhole: OID) READONLY;
changePart(oldPart: OID, newPart: OID): BOOL READONLY;
getParts(): {OID} READONLY;
getWholes(): {OID} READONLY;
```

The above is actually not the entire public interface, which can be found in the appendix. We have omitted certain utility methods that are not of interest here and have shown only those methods which are used to manipulate instances of HM classes as parts and wholes. In the following sections, we turn our attention to these methods.

6.3.1 Establishing Part-Whole Connections between Instances

Assume that we have a part relationship $P_{B,A}$ between the meronymic class B and the holonymic class A. To establish a part connection between the instance b of B and the instance a of A, we invoke the *addPart* method for a as follows.

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

Here, we take a and b to be VML variables of type OID holding the OIDs of the respective objects.

According to its signature, *addPart* returns a Boolean value which is used to indicate whether or not the desired part installation was carried out successfully. TRUE indicates that all went well, while FALSE signals a failure. Failures can occur in two different scenarios. First of all, if the given object b is not an instance of an appropriate meronymic class (i.e., an HM class serving as a meronymic class in a

part relationship with respect to *a*'s class), then it certainly cannot be attached to *a*. Because *addPart* is defined generically in the metaclass's instance-instance type with a formal parameter of data type OID instead of one that is more specific, it is not possible to statically check for such an error. Thus, the method itself must do this checking and abort the attachment if an incompatibility is found.

A failure can also occur if the method detects a potential violation of a prescribed constraint of a characteristic dimension. Specifically, *addPart*, like *make*, is responsible for upholding the constraints of the first two part relationship dimensions: the exclusive/shared dimension and the cardinality dimension. In the first place, it must ensure that the addition of the new meronym does not violate the maximum cardinality of the part relationship in question. If it does, then the connection is disallowed. Of course, there is no possibility of a minimum cardinality violation as with *make*, so this is not an issue. Secondly, *addPart* must make certain that attaching the given part does not violate an exclusive ownership held on it. The process of confirming that this second constraint is not violated is more involved because it requires that the method query the new part and then inquire with its existing wholes and their part relationships about ownership rights. The details of this activity can be gleaned from the code in the appendix. Obviously, if the part has no holonyms, then the attachment can be made straight away.

To simplify the coding of *addPart* and other various methods, we have adopted, without loss of generality, a protocol that requires the establishment of part con-

nections at the holonym only. That is, parts are attached to wholes, but not vice versa. Thus, only the method *addPart* should ever be used by an application to relate two instances. The use of *addWhole* is strictly reserved for trusted clients like the method *addPart* itself. In fact, *addPart* takes responsibility for informing the holonym that it has a new meronym and invokes *addWhole* to accomplish this. So, at the end of its execution, both objects are aware that they are part and whole with respect to each other, and each can be queried in this regard.

Actually, *addWhole* should not even appear in the public interface of the instances of HM classes. However, due to a limitation of the current version of VML, we are forced to expose it in this manner. What we would have preferred to do was make it a "friend function" (C++ terminology [199, 200]) for use only by methods in the related holonymic classes. However, this option is currently not available in VML. Besides, because such holonymic classes could not have been made available to *addWhole* when it was defined (remember, its definition appears in the HolonymicMeronymic instance-instance type and is independent of any specific HM class). What is needed is a kind of "parameterized friend function" where the friendly classes can be specified parametrically at the time an HM class is instantiated. We see such a mechanism as a useful and even necessary extension to the concept of friend function in the context of metaclasses.

6.3.2 Breaking Part-Whole Connections between Instances

Our adopted protocol also requires that part-whole connections be broken at the whole and not at the part. This is done using the method *removePart*. For example, to remove the meronym b from the holonym a, *removePart* is invoked as follows:

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

Both a and b are once again taken to be VML variables holding the respectives OIDs.

The method *removePart*, operating analogously to *addPart*, guarantees that the involved meronym is properly informed of the disconnection. The corresponding method *removeWhole* need not be invoked explicitly and, in fact, should never be invoked by an application program. As with *addWhole*, it appears in the public interface of the instance-instance type only because of a limitation of the current version of VML. It, too, would be better defined as a friend function of the respective holonymic classes.

A return value of TRUE for *removePart* indicates that it was able to remove the given meronym; FALSE signals a failure, which may have occurred for one of two reasons. First of all, the instance passed as its argument may not actually have been a part of the target whole, in which case there was nothing to be done. Second, it may have detected a potential violation of the minimum cardinality constraint of the part relationship under consideration and thus refused to perform the requested action.

It should be pointed out that in the case of a fixed-cardinality part relationship (i.e., one with identical upper and lower cardinality bounds), *removePart* is guaranteed to fail because the removal of any such part is certain to violate the lower bound. Thus, it is not possible, using the methods described so far, to remove one part and exchange it with another in such circumstances. To rectify this, we provide an additional method *changePart* which in a single transaction removes a part of a given type and replaces it with another of the same type. For example, to replace part b_1 with b_2 in the whole a, we do the following:

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

As with the other methods of the instance-instance type discussed thus far, *changePart* returns a Boolean value to indicate success or failure. If b_1 and b_2 are not of the same type, then the exchange is aborted. Clearly, there is no chance of a cardinality violation because the cardinality of the respective meronym set will be the same both before and after the transaction. However, there is still the potential for a violation of an exclusive ownership constraint with respect to the replacement part. The detection of this leads to a failure.

Because of our protocol, there is no need for the definition of a corresponding method *changeWhole* in the HolonymicMeronymic instance-instance type. All the necessary notifications are performed by *changePart* using the methods already described.

6.3.3 Querying a Part Hierarchy

Once part-whole connections have been established between instances of various HM classes, we would like to be able to query the instances with respect to their part relationships. This can be done using the methods *getParts* and *getWholes* provided by the instance-instance type.

The method *getParts* returns *all* the parts of the target instance. As is customary in VML, the result of the method is the set of OIDs of the particular meronyms. This result can be assigned to a VML variable of type "set of OID" for further processing. For example, to get the parts of an instance a, we do the following:

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

Here, the set of parts returned by the method is assigned to the VML variable "theParts." If the target instance has no parts, then the resultant set is empty.

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

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

Once again, the result is a set of OIDs which in this case is assigned to "theWholes," a VML OID-set variable. If the instance is not part of any wholes, then the result is empty.

We note that each of the two methods does not take any arguments. It is also important to note that each returns 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, for example, to write a VODAK Query Language [3, 2] selection query in terms of the result of *getParts* to obtain the meronym set with respect to a particular part relationship or meronymic class.

6.3.4 Performing Value Propagation using NOMETHOD

In VML, the inheritance behavior specified by a semantic relationship is captured using the NOMETHOD clause (or simply NOMETHOD) [109] of the instance-instance type of a custom metaclass. As its name implies, NOMETHOD is the mechanism by which an instance in a VML database deals with a message that it is not equipped to handle (i.e., for which it has *no method*). NOMETHOD is actually a VML code segment that is invoked automatically when an unknown message is encountered by an object. It looks very much like an ordinary method, with the following exceptions. First, NOMETHOD cannot be given any formal parameters. Second, in its scope, there are two special, predefined identifiers: (1) *currentMeth*, which is bound to the offending message (or method name) that was sent to the object and caused NOMETHOD to be invoked; and (2) *arguments*, which is bound to the list of arguments of that message.

The NOMETHOD clause may be defined simply to pass *currentMeth* and *arguments* along "as is" to some related object of the target through another method invocation. This is what happens in the case of category specialization [108]. How-

ever, by passing *currentMeth* through a filter, NOMETHOD can implement different forms of *selective inheritance* [108, 109]. As it happens, the identifier *currentMeth* is of data type VML string, and our filters for it take the form of characteristic functions of sets of constant VML strings. Such a filter was used previously in the implementation of a filter-based role specialization semantic relationship [109] in VML. We have adapted the technique to perform value propagation in the context of the part relationship.

For each part relationship, we employ two filters described by the following sets of property names (represented as constant VML strings): *upSet* which holds the names of properties being propagated upward, and *downSet* which holds the names of properties being propagated downward. Each of these is specified declaratively as a portion of the "part relationship structure" passed to *defMeronymicRelshps* at the time an HM class is instantiated. For example, assume that we have two classes, car and body. If we wish to define a part relationship between them such that the property *color* is propagated upward from body to car, then we write:

```
CLASS Car METACLASS HolonymicMeronymicClass
    INSTTYPE carType
    INIT Car->defMeronymicRelshps({[theMeronymicClass:Body,
                                   es:GLOBAL_EXCL,
                                   cardinality:[min: 0, max: 1],
                                   dependency:NONE,
                                   upSet:{'color'},
                                   downSet:{}]})
END;
```

```
CLASS Body METACLASS HolonymicMeronymicClass
   INSTTYPE bodyType
   INIT Body->defHolonymicClasses( {Car} )
END;
```

As we see, the *upSet* contains a single element, the constant VML string 'color' (written in single quotes). So, only that property is propagated upward. The *downSet* is empty, indicating that no properties are propagated downward. As we have mentioned, the two sets should always be disjoint. Moreover, no two part relationships should pass the same property to a single class.

As with all other information concerning a specific part relationship, both the *upSet* and the *downSet* are stored with the class participating as holonymic class in the part relationship. Access to these sets is limited to the predicates (filters) *propertyUpPropagated* and *propertyDownPropagated*, respectively. Given a VML string m, representing the name of a property, and the OID of a meronymic class B, *propertyUpPropagated* determines whether or not m is propagated upward from class B to the target holonymic class. Likewise, *propertyDownPropagated* indicates whether or not a property is propagated downward from the target class to the given meronymic class.

The NOMETHOD clause uses these filters to perform value propagation as follows. Given an offending message bound to the predefined identifier *currentMeth*, NOMETHOD first successively scans the holonyms of the target object and uses the method *propertyDownPropagated* of their respective classes to determine if any of them propagates the property *currentMeth* downward to the target. If one that

does is found, then the message *currentMeth* is delegated to it through a method invocation in order to obtain the desired property value. If not, then NOMETHOD next iteratively scans the parts of the target and employs *propertyUpPropagated* to determine if any propagates *currentMeth* upward to the target. If one does, the message *currentMeth* is delegated to it to get the requested property value. The fine details of this filtering process can be found in the code for NOMETHOD in Appendix A. If the message cannot be handled by any of the target object's wholes or parts, then NOMETHOD fails and a run-time error occurs [109].

Of course, the NOMETHOD clause may be invoked iteratively in situations where a property is propagated across many levels of a part hierarchy. Such is the case in our model of the editorial page of *The New York Times* (see Section 5.3), where the property *date* is propagated downward four levels from `newspaper` to `editorial`. We show some of the VML code for that schema in Appendix B. In the specification of the part relationships as arguments to *defMeronymicRelshps*, a derived attribute, which itself is being propagated, is treated in the same manner as any other propagated properties. An example of this can be seen in the definition of `editorial_column` which receives *date* from `editorial_page` and propagates it downward to `editorial`.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have presented a comprehensive OODB part model which greatly enhances the usefulness and effectiveness of part-whole modeling in the context of OODBs. The contributions of our work stretch in three directions: (1) New semantics for the part relationship has been identified and codified. (2) Two novel realizations for the part relationship and its accompanying modeling constructs have been introduced. (3) An enhanced graphical schema notation for the development of OODB part schemata has been presented.

At the heart of our part model is a mathematical part relationship comprising a variety of semantics and functionalities. In particular, the part relationship is divided into four characteristic dimensions: (1) exclusive/shared, (2) cardinality/ordinality, (3) dependency, and (4) value propagation.

The exclusive/shared dimension refines the semantics of the ORION part model [107] and distinguishes three ways that parts can be distributed among wholes. A whole can be given exclusive ownership of a part across the entire database topology, forbidding any other objects from claiming that part as their own; this is the exclusiveness of ORION which we refer to as global exclusiveness. Because such a constraint can be too restrictive under certain circumstances, we have also defined another kind of exclusiveness, called class exclusiveness, where the exclusive reference restriction is confined to the extension of the holonymic class. Unrestricted

sharing of parts, which is the alternative in ORION, has been found to be too loose or unrestrictive for the construction of logical part hierarchies. For this reason, our part model offers the more refined notion of limited sharing. Ordinary, unrestricted sharing appears as a special case of this.

The cardinality/ordinality dimension of the part relationship describes how many and in what ways parts of a specific type are combined in the formation of wholes. Parts of a single type can be grouped together as a set which has constraints on its cardinality. Alternatively, the parts can be organized in an ordered list, with each part functioning in a certain capacity denoted by its position in the list. The length of the list can be fixed for all the holonyms of a class at the time the class is created; this yields an ordering of a definite number of parts. It is also possible to have the length of the list vary from holonym to holonym, in which case we have an ordering of an indefinite number of parts.

The dependency dimension deals with the deletion semantics of parts and wholes. As we discussed, there are times when it is desirable to have the deletion of a whole imply the deletion of some or all of its parts. Such an arrangement can be particularly useful when the holonym is an extensive object comprising a large number of meronyms. The ORION part relationship allows for this kind of dependency. On the other hand, in our part model, we also avail ourselves of ontological dependency, where the existence of a whole can be made dependent on the existence of some one

or more defining parts. Thus, in our model, dependency may be defined in either direction, from the part to whole, or vice versa.

Part hierarchies are a natural place in which to employ derived attributes. Often, characteristics of parts are assimilated by their wholes, or vice versa. The fourth dimension of the part relationship, the value propagation dimension, forms the basis for the formal definition of such constructs in our part model. Specifically, it is used to define derived attributes with respect to propagations of property values across the part relationship, in the direction from the holonymic class to the meronymic class, or the other way around. In our model, a value propagation can take on one of the following three forms which extend and formalize previous notions of propagation in OODB part hierarchies [127, 145]: invariant, transformational, or cumulative. The first of these limits the source of the propagation to a single object (which is either a whole or a part, depending on the propagation's direction), and data values are passed along "as is" without any intervening computation. Transformational propagation drops the uniqueness requirement for the source object and allows for the specification of a computation in the process of propagation. In this way, it can be used to transform property values obtained from (possibly) many source objects into a single value of a given data type. Cumulative value propagation, like transformational propagation, does not require a unique source object; however, instead of producing a single data value, it collects the multiple property values into a set which is propagated to the target object. All three types of value propagation

together provide a powerful means for the definition of derived attributes in terms of a single part relationship.

In a part hierarchy, derived attributes are often best described in terms of identical properties of many source objects, regardless of their classes. A canonical example of this is the fact that the weight of a car is the sum of the weights of all its parts, not just those of a single type. Or, for example, the material make-up of a golf club is the set of materials from its shaft, head, and grip. To accommodate these situations, our part model allows identical properties from different classes to be propagated simultaneously across many part relationships to a single source class. In a pattern mirroring the part structure itself, the values of these propagations are combined to form what we call a generalized derived attribute. As was mentioned, this mechanism constitutes a third resolution strategy for the "multiple inheritance" problem (or more precisely, the multiple value propagation problem) in the context of OODB part hierarchies.

To facilitate the construction of OODB part-whole schemata and provide a sound means for communicating about parts and wholes in an OODB environment, we have introduced an extensive graphical notation for the part relationship with all its various semantics and functionalities. Our notation extends some previous graphical conventions and, in fact, uses an enhancement of the OMT [171] part relationship symbol as its basis. Variations of a small set of symbols, including the one for the part relationship derived from OMT, mnemonically express all the different semantics

prescribed by the part relationship's characteristic dimensions. There are graphical symbols for the three different types of value propagation and accompanying symbols for the derived attributes induced by that process. Using these, one can symbolically define both a derived attribute and its implementation in terms of the propagation of a property value across a part relationship. Symbols are also provided for the representation of generalized derived attributes and their computation.

In order to provide a framework within which to do OODB part-whole modeling, we have developed a general graphical notation for the representation of OODB schemata. This language captures a full range of OODB constructs including classes, attributes, methods, and a variety of different relationship types. As such, it is applicable to a wide group of different OODB data models. In designing this graphical language, we have taken into account the mnemonic value of the graphical icon and have chosen our symbols accordingly. Features of the symbols themselves are used to convey aspects of the semantics of the constructs they stand for. This graphical notation has been employed successfully in a number of large data modeling projects undertaken by our research group. It is also currently under consideration for inclusion in a commercially-available CASE tool called ObjectMaker [129].

To promote and support the use of our graphical schema representation, we have built the OOdini software system. OOdini is a constraint-based graphical editor designed specifically for the creation and manipulation of OODB schemata described using our representation. The fact that OOdini is fine-tuned to our notation means

that it can ensure the integrity of such diagrams and greatly facilitate their creation. Various features of OOdini also make it an excellent OODB schema orientation or browsing tool. Besides this, OOdini permits conversion of a graphical schema into an OODB abstract textual language called OODAL. OOdini also supports conversion into Dual Model syntax (referred to as DAL) and the VODAK Model Language. In this manner, OOdini is an effective OODB graphical interface. The construction of converters for other OODB languages is simplified by the use of an application programming interface (API), provided with OOdini, for access to an internal C Language representation of OODAL code.

We have demonstrated the viability of our part model by presenting a pair of alternate realizations for it in the context of OODB data models. Following in the tradition of the ER model and other semantic data models, the first of these realizes the part relationship as an object class in its own right whose instances stand for part relationship occurrences. As with the graphical symbols, variations of a "generic" realization are used for all the different types of part relationships. A strength of this realization is its strict reliance on a basic set of existing OODB constructs, avoiding the need for any new modeling primitives. It is our hope that the designers of different OODBs will exploit our part model to quickly and easily add part-whole modeling capabilities to their own systems. We see such prospective implementations as invaluable sources of feedback on our work. In fact, at present, an implementation

of our part model is being carried out in Smalltalk as part of a student's Master's thesis [172].

The second realization, an implementation of which has been described in detail in Chapter 6, exploits the metaclass mechanism of VML. The VML OODB data model is based on an open architecture that can be tailored through introduction of new semantic relationships. The tailoring is done with the use of an extensible metaclass system. To implement our part model, we have constructed a custom metaclass, called the HolonymicMeronymic metaclass, which endows the classes of a part hierarchy (what we call HM classes) and, in turn, their instances with structure and behavior appropriate to the part relationship. (We note that while it is certainly possible to define all classes of a schema to be HM classes, this adds unnecessary overhead to classes which do not actually participate in part relationships.) Through the HolonymicMeronymic instance type, an HM class is given methods for the specification and maintenance of its part relationships with other HM classes. Because the part relationship imposes its own specific creation and deletion semantics, an HM class is also provided with special methods *make* and *destroy* for creating and deleting instances, respectively. Both are designed to enforce the various constraints dictated by the characteristic dimensions. The instance-instance type of the HolonymicMeronymic metaclass augments the instances of HM classes with new methods to give them the "look and feel" of meronyms and holonyms with respect to each other. Specifically, such instances are given the ability to establish and break part-

whole connections with other instances. These connections can also be changed or queried, with the parts or wholes of an instance accessible through a single method call. With the aid of the VML NOMETHOD clause in the HolonymicMeronymic instance-instance type, value propagation in either direction across a part relationship is carried out.

The implementation of our part model in terms of a VML metaclass has demonstrated two important points: (1) Our part model can be seamlessly incorporated into an existing OODB system. In other words, the introduction of our part model does not require the rewriting of a substantial subsystem of the OODB. (2) The VML metaclass facility can indeed support extensions in terms of new semantic relationships such as our part relationship.

During our implementation work in the context of VML, we uncovered the need for a new type of friend function [199] which we have named "parameterized friend function." The need for such a construct is a consequence of VML's unique metaclass facility. The instance-instance type of a metaclass is defined generically, or, in other words, it is defined before any of the classes (and their respective instances) that it affects. Thus, methods in the instance-instance type cannot be given "friends" at the time they are defined—as is usually the case—because these are not known. Instead, the friends must be specified as parameters at the time an instance (which, of course, is a class) of the metaclass is created. Furthermore, the friends of a given

method will vary among the different classes that are instances of the metaclass in question.

Another issue for future research is the explicit incorporation of the different linguistic usages (e.g., the six usages of [210]) of the part relationship into our own OODB part model. For example, it might make sense to allow the database designer to tag each part relationship with its intended linguistic sense, such as component/integral or member/collection. A potential use for such tags is in the context of performing part retrieval, where one may wish to recursively retrieve all the parts of some whole along a portion of the part hierarchy restricted to one sense of the part relationship. For example, one may request the retrieval of all parts that are components of the whole (or its parts) in the component/integral sense.

As we have discussed, the distinctions of [210] have been employed by Huhns and Stephens in an algebra of semantic relationships [93]. This algebra is designed to provide a means for performing valid inferencing in the context of a knowledge-based system. Currently, there is an effort to introduce the algebra into the CYC knowledge-based system [117].

In a system like CYC, it makes sense to limit valid "part" inferences to those involving part relationships of a single type. However, in a database environment, we would not want to impose such a severe restriction on recursive part retrieval. Consider, for example, the case of a factory which produces a daily allotment of cars. Now, assume that we wish to know all the parts used in the production of

cars on a specific day. If, in our database, we have modeled each daily allotment as a collection of cars with respect to a member/collection part relationship, and each car is decomposed using component/integral part relationships, then this query can be formulated simply as a recursive part retrieval involving *two* senses of the part relationship. Therefore, strictly adhering to the linguistic distinctions and ruling out such a retrieval *a priori* is totally inappropriate in this context.

Another area of investigation which should prove fruitful is the introduction of rules into the part model. Of course, the use of rules would require the introduction of some sort of rule manager into the underlying OODB data model (see, e.g., [169, 196]). We have not made any assumption about the existence of such a rule manager in the course of this dissertation. Rules could be used to augment aspects of the constraints imposed by the various characteristic dimensions. In the context of the exclusive/shared dimension, for example, rules defined in terms of derived attributes could serve to enhance the constraints on the number of wholes a part can appear in. So, instead of stating that a student may be part of at most some maximum number of sections, we may want to have a rule stating that a student may be part of any number of sections as long as the value of his derived attribute *credits* (obtained from class `section` across the part relationship and indicating his total number of credits enrolled for) remains below some fixed amount.

We also see rules playing a role in the specification of parameterized constraints in OODB part hierarchies. For example, instead of working with constant values

to define constraints (such as the range-restriction of the cardinality/ordinality dimension), one may want to employ the properties of an HM class for this purpose. In this manner, the constraints could be specified at object instantiation-time and vary from instance to instance. Thus, we could impose, say, different limits on the number of articles that appear in different newsletters.

Rules may also prove to be of interest in the definition of derived attributes. A rule, for instance, could replace a default value and designate an alternate source of value propagation when an appropriate one is not available. Or rules could be used instead of algebraic operators to specify conversions in a transformational value propagation.

Currently, OOdini is not fully equipped to handle all the symbolic machinery we have introduced for our part model. It only provides an unqualified part relationship that conveys a general "is part of" interpretation. As future work, the full range of graphical symbols could be integrated into OOdini. This would allow it to convert graphical part schemata directly into a VML syntax enhanced by our metaclass implementation. In this way, database designers would be further alleviated of the burden of doing hand-coding.

As an important line of future work, we see the application of the methodology used in this dissertation to other semantic relationships besides the part relationship. Its use in the formalization of the part relationship can serve as a prototype for the characterization of others. One such relationship that could readily benefit from a

dimensional decomposition of its semantics is *roleof*. Informally, *roleof* is a semantic relationship used to connect disparate database objects representing the same real-world entity in its various "roles." For example, a person can be represented as an employee in one context and a student in another. A major problem with this relationship has been the lack of agreement on precise semantics. In fact, four different versions of it (or its converse *has-role*) can be found in [108, 126, 143, 179]. Thus, *roleof* could very much benefit from the unified treatment of its different semantics and varied inheritance behavior [108, 143]. Other semantic relationships that also come to mind are versioning, ownership, and containment. In conclusion, we see ahead the application of our methodology as an important tool in the ongoing effort to expand, in a controlled way, the set of semantic relationships that are actually useful in enterprise modeling within OODBs.

APPENDIX A

VML CODE FOR THE HolonymicMeronymic METACLASS

In this appendix, we provide the entire VML code specification for our Holonymic-Meronymic metaclass. Included are the definition of the metaclass itself, various supplementary data types, and the metaclass's instance type and instance-instance type.

```
SCHEMA   HMMetaClass

//////////////////////////////////////////////////////////////////////
//
// The exclusive/shared dimension type.
//
//////////////////////////////////////////////////////////////////////

   DEFINE CLASS_EXCL  1
   DEFINE GLOBAL_EXCL 2
   DEFINE SHARED      3
   DATATYPE ExShType = INT;

//////////////////////////////////////////////////////////////////////
//
//  The cardinality dimension type.
//
//////////////////////////////////////////////////////////////////////

   DEFINE INFINITY 32767
   DATATYPE CardType = [min: INT, max: INT];

//////////////////////////////////////////////////////////////////////
//
//   The dependency dimension type.
//
//////////////////////////////////////////////////////////////////////
```

```
   DEFINE PART_TO_WHOLE 1      // parts depend on wholes.
   DEFINE WHOLE_TO_PART 2      // wholes depend on parts.
   DEFINE NONE          3      // no dependency.
   DATATYPE DependType = INT;

////////////////////////////////////////////////////////////////////
//
//   The following type is used to store information about
//   a part relationship that a given class might participate
//   in.  For any class in the hierarchy, we will keep a set of
//   such structures to record all the "part" classes related
//   to the given class.
//
//   Structure Members:
//
//    theMeronymicClass: OID of the meronymic class.
//    es: value of the exclusive/shared dimension.
//    cardinality: the required number of parts for a
//                 holonym; given as a numerical range.
//    dependency:  the type of dependency associated
//                 with the relationship; either PART_TO_WHOLE,
//                 WHOLE_TO_PART, or NONE.
//    upSet:       The set of properties being propagated
//                 from the meronymic class to the holonymic
//                 class.
//    downSet:     The set of properties being propagated
//                 from the holonymic to meronymic class.
//
////////////////////////////////////////////////////////////////////

   DATATYPE PartRelationshipType = [
                   theMeronymicClass: OID,
                   es: ExShType,
                   cardinality: CardType,
                   dependency: DependType,
                   upSet: {STRING},
                   downSet: {STRING}];
```

```
//////////////////////////////////////////////////////////////////////
//
//   The following constant determines the maximum number
//   of types of parts.
//
//////////////////////////////////////////////////////////////////////

   DEFINE   MAX_PART_CLASSES   10

//////////////////////////////////////////////////////////////////////
//
//   The declaration of the HolonymicMeronymicClass metaclass
//
//////////////////////////////////////////////////////////////////////

  CLASS HolonymicMeronymicClass METACLASS Metaclass
    INSTTYPE HolonymicMeronymic_InstType
    INSTINSTTYPE HolonymicMeronymic_InstInstType
  END;

//////////////////////////////////////////////////////////////////////
//
//   The HolonymicMeronymic metaclass's InstType.
//
//////////////////////////////////////////////////////////////////////

OBJECTTYPE HolonymicMeronymic_InstType
                       SUBTYPEOF Metaclass_InstType;

INTERFACE
  METHODS
    make(someParts: {OID}): OID READONLY; // replaces method "new"
    destroy(anObject : OID) READONLY;     // replaces "delInstance"
    defMeronymicRelshps(someRelshps : {PartRelationshipType})
                                                  READONLY;
    defHolonymicClasses(someClasses : {OID}) READONLY;
    isMeronymicClassOf(aClass : OID): BOOL READONLY;
    isHolonymicClassOf(aClass : OID) : BOOL READONLY;
    getMeronymicClassesOf() : {OID} READONLY;
    getHolonymicClassesOf() : {OID} READONLY;
    exsh(aClass : OID): ExShType READONLY;
    minCard(aClass : OID): INT READONLY;
```

```
    maxCard(aClass : OID): INT READONLY;
    dependencyStatus(aClass : OID): DependType READONLY;
    propertyUpPropagated(meth: STRING, aClass: OID): BOOL READONLY;
    propertyDownPropagated(meth: STRING, aClass: OID): BOOL READONLY;

IMPLEMENTATION

   EXTERN prints(s:STRING);
   EXTERN endline();

   PROPERTIES
      thePartRelshps:
        ARRAY [SUBRANGE 1 .. MAX_PART_CLASSES] OF
                                   PartRelationshipType;
      numberOfPartRelshps: INT;
      theHolonymicClasses: {OID};

METHODS

//////////////////////////////////////////////////////////////////
//
//   The method "make" replaces the method "new" for
//   any class which participates in a part hierarchy.
//   That is, an object of any such class is created
//   by invoking make rather than new.
//
//////////////////////////////////////////////////////////////////

   make(someParts: {OID}): OID READONLY;
   {

       VAR aNewObject: OID;
       VAR aPart: OID;
       VAR thePartsClass: OID;
       VAR i: INT;
       VAR found: BOOL;

   //
   //  The following array represents a mapping from a
   //  meronymic class to the number of its instances
   //  sent in the argument of this method.  If any
   //  of these numbers violates the cardinality constraints
```

```
//  defined by the respective part relationship, then the
//  creation of the new object is abandoned and the method
//  returns NULL.  The correspondence between an entry
//  in this array and the class is determined using the
//  structure "thePartRelshps" defined as a property of
//  the class.
//

    VAR partCount: ARRAY [SUBRANGE 1 .. MAX_PART_CLASSES] OF INT;

//
// Determine how many part from each meronymic class
// are represented in the given set.
//

     FOR(i := 1; numberOfPartRelshps; 1)

          partCount[i] := 0;


     FORALL( aPart IN someParts )
     {

          thePartsClass := aPart->(OID)class();

//
//   First of all, check that the current part is an
//   instance of an acceptable class, that is,
//   check that the part's class is among the meronymic
//   classes of the target class.
//

       IF( NOT thePartsClass->(BOOL)isMeronymicClassOf(SELF) )
       {
          prints('Error in method "make":'); endline();
          prints('   A given object is not an instance of an');
             endline();
          prints('   appropriate meronymic class.');
             endline(); endline();
          RETURN NULL;
       }
```

```
        ELSE
        {

//
//   determine the appropriate index into the "count" array
//   and increment that entry.
//

              found := FALSE;
              i := 1;

           WHILE( (NOT found) & (i <= numberOfPartRelshps) )
           {

              IF(thePartsClass ==
                     thePartRelshps[i].theMeronymicClass)

                   found := TRUE;

              ELSE

                   i := i + 1;

           }

           partCount[i] := partCount[i] + 1;

        }

      }

//
//  Now, check to see that the number of parts
//  from each class at least satisfies the minimum
//  part constraint for the particular relshp.  If
//  any does not, then return NULL.
//

  FOR(i := 1; numberOfPartRelshps; 1)
  {
```

```
//
// Is the min cardinality constraint violated?
//

    IF(partCount[i] < thePartRelshps[i].cardinality.min)
    {
      prints('Error in method "make":'); endline();
      prints('   The min cardinality restriction of one of the');
         endline();
      prints('   part relationships was violated.  The creation');
         endline();
      prints('   of the new object was aborted.');
         endline(); endline();
      RETURN NULL;
     }

  }

//
// At this point, we can go ahead with the creation of the
// new object, and we can try to attach all the given parts.
// Note that we may fail due to a violation of a maximum
// cardinality or an exclusive/shared constraint.
//

         aNewObject :=  SELF->new();

//
//  Note: the following two methods must only be
//  invoked at this point and nowhere else!!!
//

         aNewObject->initMeronymSet();
         aNewObject->initHolonymSet();

//
//  Try to attach the parts.
//

 FORALL(aPart IN someParts)
 {
```

```
   IF(NOT aNewObject->(BOOL)addPart(aPart))
   {
    prints('Error in method "make":'); endline();
    prints('   Could not attach one of the given parts');
       endline();
    prints('   to the newly created object.  This may have');
       endline();
    prints('   been due to a violation of a maximum cardinality');
       endline();
    prints('   constraint or an exclusive ownership constraint.');
       endline();
    prints('   The creation of the new object was aborted.');
       endline(); endline();

//
//  Remove the reference to the new object from
//  all the parts.
//

     FORALL(p IN someParts)

            p->removeWhole(aNewObject);

     //
     //  discard the new object.
     //

            SELF->delInstance(aNewObject);

     //
     //  return NULL to indicate failure.
     //

            RETURN NULL;

   }

  }

 RETURN aNewObject;

};
```

```
/////////////////////////////////////////////////////////////////
//
//   This method replaces the method "delInstance" for
//   any class in a part hierarchy.
//
/////////////////////////////////////////////////////////////////

    destroy(anObject : OID) READONLY;
    {

       VAR objHolonymSet: {OID}; VAR objMeronymSet: {OID};
       VAR objClass: OID; VAR p: OID; VAR w: OID;
       VAR anotherWhole: OID; VAR aWholesClass: OID;
       VAR aPartsClass: OID; VAR psHolonymSet: {OID};
       VAR psMeronymSet: {OID}; VAR wsHolonymSet: {OID};
       VAR wsMeronymSet: {OID}; VAR psClass: OID;
       VAR wsClass: OID; VAR currentWholeClass: OID;
       VAR somePart: OID; VAR someWhole: OID;
       VAR someWholeClass: OID; VAR theDependStatus: DependType;
       VAR theMinCard: INT; VAR hasNonDependentRequiredParts: BOOL;
       VAR found: BOOL;

           objHolonymSet := anObject->({OID})getWholes();
           objMeronymSet := anObject->({OID})getParts();
           objClass := anObject->(OID)class();

   //
   //  Case 1:
   //  If the object is not participating in a part relshp.,
   //  then delete it.
   //

     IF( (objHolonymSet == {}) & (objMeronymSet == {}) )

             SELF->delInstance(anObject);

   //
   //  Cases 2, 3, and 4:
   //  If there is a part or whole of the object which is
   //  not dependent on it, then refuse the deletion.
   //  Or, if deleting the object might cause a cardinality
```

```
//  violation, then don't delete it either.  We also
//  must check for the exception to case 2 where an
//  object has only required parts and these cannot
//  possibly be removed.
//

  ELSE
  {

     //
     //   First, check the wholes.
     //

     FORALL( w IN objHolonymSet )
     {

             aWholesClass := w->(OID)class();

             theDependStatus := aWholesClass->(DependType)
                                          dependencyStatus(SELF);

             theMinCard := aWholesClass->(INT)minCard(SELF);

        IF( (theDependStatus != WHOLE_TO_PART) |
                                          (theMinCard > 0) )
                  RETURN;

     }

     //
     //   Now the parts.
     //

         hasNonDependentRequiredParts := FALSE;

     FORALL( p IN objMeronymSet )
     {

             aPartsClass := p->(OID)class();

             theDependStatus := SELF->(DependType)
                       dependencyStatus(aPartsClass);
```

```
            theMinCard := SELF->(INT)minCard(aPartsClass);

        IF(theDependStatus != PART_TO_WHOLE)
        {
           IF(theMinCard == 0)
                        RETURN;

           ELSE
             hasNonDependentRequiredParts := TRUE;
        }

    }

//
//  Now, if we arrived here, then we know that the object of
//  interest should be deleted.  However, if it has
//  nondependent required parts, (i.e.,
//  hasNonDependentRequiredParts == TRUE), then these
//  must be removed at this point.  After that,
//  Case 5 will hold (or its special case: Case 1), and
//  we can proceed straight to the processing of that case.
//

    IF(hasNonDependentRequiredParts)
    {

        FORALL( p IN objMeronymSet )
        {
             aPartsClass := p->(OID)class();

             theDependStatus := SELF->(DependType)
                          dependencyStatus(aPartsClass);

        //
        //  If the part is not dependent on anObject,
        //  then it must be detached right now.  Otherwise
        //  it would be subject to deletion (which it
        //  shouldn't be.
        //

           IF(theDependStatus != PART_TO_WHOLE)
```

```
            {
                anObject->removePartPrivate(p);
                p->removeWhole(anObject);
            }

        }

    }

//
//  If we have made it this far, then we are dealing
//  with case 5.  That is, the given object O has parts
//  and/or wholes, all of which are dependent on it, and
//  there is no possibility that the deletion of O will
//  violate an integrity (cardinality) constraint.  Therefore,
//  O may be deleted; however, we still need to examine each
//  object which is dependent on O to see if the deletion
//  should be propagated to it.  If such an object (say Q)
//  is in a part-whole relationship with another object from
//  O's class, or if Q is a meronym in a part relationship
//  with some minimum cardinality > 0, then we ignore it
//  (i.e., do not propagate the deletion to it).  Otherwise,
//  we act as discussed in case (iii) above.
//
//  First, start with the parts.  Of course, we must
//  refresh "objMeronymSet" because certain parts
//  may have been removed.
//

        objMeronymSet := anObject->({OID})getParts();

    FORALL( p IN objMeronymSet )
    {

        //
        //  First, disconnect the part from the
        //  object which is being deleted.  Note that
        //  we do not use "removePart" here because we
        //  do not care about any existing cardinality
        //  constraints; the target whole object is being
        //  deleted from the system at this point, no matter
        //  what.  Instead, we use the special method
```

```
//  "removePartPrivate" which does not test for
//  integrity violations.
//

    anObject->removePartPrivate(p);
    p->removeWhole(anObject);

//
//  now see if there are other wholes of p in
//  anObject's class.  Also, see if there
//  are any wholes which may require p in order
//  to satisfy a minimum cardinality constraint.
//

    found := FALSE;
    psHolonymSet := p->({OID})getWholes();
    psClass := p->(OID)class();

FORALL( anotherWhole IN psHolonymSet )
{
   currentWholeClass := anotherWhole->(OID)class();

   IF( (currentWholeClass == objClass) |
       (currentWholeClass->(INT)minCard(psClass) > 0) )

             found :=  TRUE;
}

IF(NOT found)
{

   //
   //  OK, so p should be deleted, too.
   //  Break all the part-whole connections
   //  of p, except for those which are dependent
   //  on p.  Start with its parts.
   //

       psMeronymSet := p->({OID})getParts();

  FORALL( somePart IN psMeronymSet )
  {
```

```
        IF(psClass->(DependType)
    dependencyStatus(somePart->(OID)class()) !=
                                                PART_TO_WHOLE)
        {
              p->removePartPrivate(somePart);
              somePart->removeWhole(p);
        }

      }

       //
       //  Now the wholes.
       //

       FORALL( someWhole IN psHolonymSet )
       {
         IF(someWhole->(OID)class()->(DependType)
             dependencyStatus(psClass) != WHOLE_TO_PART )

         //
         //  "removePart" cannot fail here because
         //  we know that the minCard of the relationship
         //  is 0.
         //

             someWhole->removePart(p);
       }

    //
    //  Now destroy p.
    //

         psClass->destroy(p);

   }

} // FORALL parts of the given object to be deleted.
```

```
//
//   Now, take care of the given object's (i.e., anObject's)
//   wholes.
//

     FORALL( w IN objHolonymSet )
     {

        //
        //  Remove the given object from the whole.  Again,
        //  removePart cannot fail because we know that all
        //  the part relshps that anObject participates in
        //  as a part have a minCard of 0.
        //

             w->removePart(anObject);

        //
        //  now see if there are other parts of w in
        //  anObject's class.  Also, see if there
        //  are any wholes which may require w in order
        //  to satisfy a minimum cardinality constraint.
        //

             found := FALSE;
             wsHolonymSet := w->({OID})getWholes();
             wsMeronymSet := w->({OID})getParts();
             wsClass := w->(OID)class();

        FORALL( anotherPart IN wsMeronymSet )
        {
           IF( anotherPart->(OID)class() == objClass )
                       found := TRUE;
        }

        IF(NOT found)
        {
            FORALL( someWhole IN wsHolonymSet )
            {
                 someWholeClass := someWhole->(OID)class();

              IF( someWholeClass->(INT)minCard(wsClass) > 0 )
```

```
                found := TRUE;
     }
}

IF(NOT found)
{
   //
   //   OK, so w should be deleted.
   //   Break all the part-whole connections
   //   of w except for those which are dependent
   //   on it.  Start with w's part.
   //

  FORALL( somePart IN wsMeronymSet )
  {

    IF(wsClass->(DependType)
        dependencyStatus(somePart->(OID)class()) !=
                                          PART_TO_WHOLE)
    {
             w->removePartPrivate(somePart);
             somePart->removeWhole(w);
    }
  }

   //
   //  Now the wholes.
   //

   FORALL( someWhole IN wsHolonymSet )
   {
      IF(someWhole->(OID)class()->(DependType)
           dependencyStatus(wsClass) != WHOLE_TO_PART)

     //
     //  "removePart" cannot fail here because
     //  we know that the minCard of the relationship
     //  is 0.
     //
             someWhole->removePart(w);

   }
```

```
          //
          //  Now destroy w.
          //

               wsClass->destroy(w);

          }

       }  // FORALL wholes of the given object to be deleted.

       //
       //  Finally, discard the given object.
       //

         SELF->delInstance(anObject);

    }

   };

///////////////////////////////////////////////////////////////////
//
//   This method establishes the part relationships
//   for a given holonymic class.  It should be invoked
//   only once in the INIT clause for the class.
//
///////////////////////////////////////////////////////////////////

   defMeronymicRelshps(someRelshps: {PartRelationshipType})
                                                   READONLY;
   {
            VAR index: INT;
            VAR r: PartRelationshipType;

            index := 1;

   //
   //   Place the given part relationships in the list of
   //   of this class.
   //
```

```
            FORALL(r IN someRelshps)
            {
                thePartRelshps[index] := r;
                index := index + 1;
            }

   //
   //  record the number of part relationships that the
   //  given class participates in.
   //

          numberOfPartRelshps := index - 1;

   };

/////////////////////////////////////////////////////////////////
//
//   The following method records the holonymic
//   classes for the given meronymic class.  If
//   the class is the root of a part hierarchy,
//   this set will be empty.
//
/////////////////////////////////////////////////////////////////

   defHolonymicClasses(someClasses : {OID}) READONLY;
   {
         theHolonymicClasses := someClasses;
   };

/////////////////////////////////////////////////////////////////
//
//   This method determines if the target class is a
//   meronymic class in a part relationship with
//   the given (holonymic) class.
//
/////////////////////////////////////////////////////////////////

   isMeronymicClassOf(aClass : OID) : BOOL READONLY;
   {
        RETURN aClass IN theHolonymicClasses;
    };
```

```
//////////////////////////////////////////////////////////////////////
//
//   This method determines if the target class is a
//   holonymic class in a part relationship with
//   the given (meronymic) class.
//
//////////////////////////////////////////////////////////////////////

   isHolonymicClassOf(aClass : OID) : BOOL READONLY;
   {

           VAR i: INT;
           VAR found: BOOL;

   i := 1; found := FALSE;

   //
   //   scan the part relationships of this class to determine
   //   if the given class is among its meronymic classes.
   //

      WHILE( (NOT found) & (i <= numberOfPartRelshps) )
      {
         IF( aClass == thePartRelshps[i].theMeronymicClass )
                             found := TRUE;

         ELSE
               i := i + 1;
       }

       RETURN found;

   };

//////////////////////////////////////////////////////////////////////
//
//   This method is the selector for the meronymic classes
//   of the target class (i.e., it returns all the meronymic
//   classes as a set).
//
//////////////////////////////////////////////////////////////////////
```

```
   getMeronymicClassesOf() : {OID} READONLY;
   {
        VAR i: INT;
        VAR theClasses: {OID};

        theClasses := {};

   //
   //   Collect the meronymic classes into a set
   //   and return this set.
   //

    FOR(i := 1; numberOfPartRelshps; 1)

      INSERT thePartRelshps[i].theMeronymicClass INTO theClasses;


    RETURN theClasses;

   };

//////////////////////////////////////////////////////////////////
//
//   This method is the selector for the holonymic classes
//   of the target class (i.e., it returns all the holonymic
//   classes as a set).
//
//////////////////////////////////////////////////////////////////

   getHolonymicClassesOf() : {OID} READONLY;
   {
       RETURN theHolonymicClasses;
   };

//////////////////////////////////////////////////////////////////
//
//   This method gets the value of the exclusive/shared
//   dimension of the part relationship of which the
//   argument is the meronymic class and the target class
//   is the holonymic class.
//
//////////////////////////////////////////////////////////////////
```

```
exsh(aClass : OID): ExShType READONLY;
{

     VAR theDimensionValue: ExShType;
     VAR i: INT;
     VAR found: BOOL;

     i := 1;
     found := FALSE;

//
//   Scan thru the list of part relshps until the
//   correct one is found.
//

   WHILE( (NOT found) & (i <= numberOfPartRelshps) )
   {
      IF( aClass == thePartRelshps[i].theMeronymicClass )
      {
               found := TRUE;
               theDimensionValue := thePartRelshps[i].es;
      }
      ELSE
              i := i + 1;
    }

    IF(NOT found)
    {
        prints('Error in method "exsh":'); endline();
        prints('   Given class is not a meronymic class of
                                                  the target');
        prints(' class.');
          endline();
        prints('   Method returned a value of SHARED, but
                                                 this should');
           endline();
        prints('   not be considered meaningful.');
           endline(); endline();

        theDimensionValue := SHARED;
     }
```

```
        RETURN theDimensionValue;

    };

////////////////////////////////////////////////////////////////////
//
//   This method gets the value of the minimum number of
//   parts that a holonym is required to have with respect
//   to given relationship (determined by a given meronymic
//   class).
//
////////////////////////////////////////////////////////////////////

    minCard(aClass : OID): INT READONLY;
    {

         VAR theValue: INT;
         VAR i: INT;
         VAR found: BOOL;

         i := 1;
         found := FALSE;

    //
    //   Scan thru the list of part relshps until the
    //   correct one is found.
    //

       WHILE( (NOT found) & (i <= numberOfPartRelshps) )
       {
           IF( aClass == thePartRelshps[i].theMeronymicClass )
           {
                  found := TRUE;
                  theValue := thePartRelshps[i].cardinality.min;
           }
           ELSE
                 i := i + 1;
        }

        IF(NOT found)
        {
```

```
            prints('Error in method "minCard":'); endline();
            prints('   Given class is not a meronymic class of the');
            prints(' target class.');
               endline();
            prints('   Method returned a value of 0, but this should');
               endline();
            prints('   not be considered meaningful.');
               endline(); endline();
            theValue := 0;
          }

      RETURN theValue;

    };

/////////////////////////////////////////////////////////////////////
//
//   This method gets the value of the maximum number of
//   parts that a holonym is required to have with respect
//   to given relationship (determined by a given meronymic
//   class).
//
/////////////////////////////////////////////////////////////////////

    maxCard(aClass : OID): INT READONLY;
    {

         VAR theValue: INT;
         VAR i: INT;
         VAR found: BOOL;

         i := 1;
         found := FALSE;

    //
    //   Scan thru the list of part relshps until the
    //   correct one is found.
    //

       WHILE( (NOT found) & (i <= numberOfPartRelshps) )
       {
          IF( aClass == thePartRelshps[i].theMeronymicClass )
```

```
        {
                found := TRUE;
                theValue := thePartRelshps[i].cardinality.max;
        }
        ELSE
               i := i + 1;
     }

     IF(NOT found)
     {
         prints('Error in method "maxCard":'); endline();
         prints('   Given class is not a meronymic class');
         prints(' of the target class.');
            endline();
         prints('   Method returned a value of 0, but this should');
            endline();
         prints('   not be considered meaningful.');
            endline(); endline();
         theValue := 0;
     }

   RETURN theValue;

  };

///////////////////////////////////////////////////////////////////
//
//   This method gets the value of the dependency
//   dimension of the part relationship of which the
//   argument is the meronymic class and the target
//   is the holonymic class.
//
///////////////////////////////////////////////////////////////////

  dependencyStatus(aClass : OID): DependType READONLY;
  {

       VAR theDependValue: DependType;
       VAR i: INT;
       VAR found: BOOL;

       i := 1;
```

```
      found := FALSE;

 //
 //   Scan thru the list of part relshps until the
 //   correct one is found.
 //

    WHILE( (NOT found) & (i <= numberOfPartRelshps) )
    {
        IF( aClass == thePartRelshps[i].theMeronymicClass )
        {
               found := TRUE;
               theDependValue := thePartRelshps[i].dependency;
        }
ELSE
              i := i + 1;
    }

    IF(NOT found)
    {
       prints('Error in method "dependencyStatus":'); endline();
       prints('   Given class is not a meronymic class of the');
       prints(' target class.'); endline();
       prints('   Method returned a value of NONE, but this
                                                    should');
          endline();
       prints('   not be considered meaningful.');
          endline(); endline();
       theDependValue := NONE;
    }

  RETURN theDependValue;

 };
```

```
//////////////////////////////////////////////////////////////////////
//
//  This method determines whether or not the given
//  property "meth" (passed as a STRING) is propagated
//  upward from the given class "aClass" to the target
//  class.
//
//////////////////////////////////////////////////////////////////////

   propertyUpPropagated(meth: STRING, aClass: OID): BOOL READONLY;
   {

        VAR i: INT;
        VAR found: BOOL;

        i := 1;
        found := FALSE;

   //
   //   Scan thru the list of part relshps until the
   //   correct one is found.
   //

      WHILE( (NOT found) & (i <= numberOfPartRelshps) )
      {
           IF( aClass == thePartRelshps[i].theMeronymicClass )
                                 found := TRUE;
           ELSE
                   i := i + 1;
      }

      IF(NOT found)
      {
          prints('Error in method "propertyUpPropagated":');
             endline();
          prints('   Given class is not a meronymic class of the');
          prints(' target class.'); endline();
          prints('   Method returned a value of FALSE, but this
                                                          should');
             endline();
          prints('   not be considered meaningful.');
             endline(); endline();
```

```
        RETURN FALSE;
      }

    RETURN meth IN thePartRelshps[i].upSet;

  };

/////////////////////////////////////////////////////////////////
//
//  This method determines whether or not the given
//  property "meth" (passed as a STRING) is propagated
//  downward to the given class "aClass" from the target
//  class.
//
/////////////////////////////////////////////////////////////////

  propertyDownPropagated(meth: STRING, aClass: OID): BOOL
                                                  READONLY;
  {

       VAR i: INT;
       VAR found: BOOL;

       i := 1;
       found := FALSE;

  //
  //   Scan thru the list of part relshps until the
  //   correct one is found.
  //

     WHILE( (NOT found) & (i <= numberOfPartRelshps) )
     {
           IF( aClass == thePartRelshps[i].theMeronymicClass )
                                found := TRUE;
           ELSE
                    i := i + 1;
     }

     IF(NOT found)
     {
        prints('Error in method "propertyDownPropagated":');
```

```
            endline();
         prints('   Given class is not a meronymic class of the');
         prints(' target class.'); endline();
         prints('   Method returned a value of FALSE, but this
                                                        should');
            endline();
         prints('   not be considered meaningful.');
            endline(); endline();
         RETURN FALSE;
      }

    RETURN meth IN thePartRelshps[i].downSet;

   };

END;

///////////////////////////////////////////////////////
//
//   The HolonymicMeronymic metaclass's instInstType.
//
///////////////////////////////////////////////////////

OBJECTTYPE HolonymicMeronymic_InstInstType
                         SUBTYPEOF Metaclass_InstInstType;

INTERFACE
   METHODS
     initMeronymSet() READONLY;
     initHolonymSet() READONLY;
     getParts() : {OID} READONLY;
     getWholes() : {OID} READONLY;
     addPart(aPart : OID) : BOOL READONLY;
     addWhole(aWhole : OID) READONLY;
     removePart(aPart : OID) : BOOL READONLY;
     removeWhole(aWhole : OID) READONLY;
     changePart(oldPart: OID, newPart: OID) : BOOL READONLY;
     removePartPrivate(aPart: OID) READONLY;

   IMPLEMENTATION
      EXTERN prints(s:STRING);
      EXTERN endline();
```

```
PROPERTIES
  theParts : {OID};  // the part's of an object.
                     // This attribute will be referred
                     // to as the meronym set, even though
                     // it contains all the object's parts
                     // regardless of their classes.  This
                     // nomenclature differs slightly
                     // from that in our papers in that there
                     // we always speak of a meronym set with
                     // respect to a given part relationship.

  theWholes : {OID}; // its wholes.
                     // Once again, deviating slightly from
                     // our conventions, we will call this
                     // attribute the holonym set.

METHODS

///////////////////////////////////////////////////////////////////
//
//   The following two methods are used to give the
//   meronym and holonym sets their initial values.
//   In the case of the holonym set, its initial value
//   is always the empty set.
//
///////////////////////////////////////////////////////////////////

    initMeronymSet() READONLY;
    {
       theParts := {};
    };

    initHolonymSet() READONLY;
    {
       theWholes := {};
    };
```

```
//////////////////////////////////////////////////////////////////
//
//  This method returns the meronym set of the object.
//
//////////////////////////////////////////////////////////////////

   getParts() : {OID} READONLY;
   {
       RETURN theParts;
   };

//////////////////////////////////////////////////////////////////
//
//  This method returns the holonym set of the object.
//
//////////////////////////////////////////////////////////////////

   getWholes() : {OID} READONLY;
   {
       RETURN theWholes;
   };

//////////////////////////////////////////////////////////////////
//
//  The following method adds a given part to the target
//  whole.
//
//////////////////////////////////////////////////////////////////

   addPart(aPart : OID) : BOOL READONLY;
   {

        VAR aPartsClass: OID;         //  the argument's class.
        VAR aPartsHolonymSet: {OID}; //  its holonym set.
        VAR myClass: OID;             //  target's class.
        VAR theExShValue: ExShType;  //  "current" part relshp's
                                      //  exclusive/shared value.
        VAR p: OID; VAR count: INT; VAR aWhole: OID;
        VAR aWholesClass: OID;

                aPartsClass := aPart->(OID)class();
                myClass := SELF->(OID)class();
```

```
//
//   Check that the part's class is among the meronym
//   classes of the target object's class.
//

    IF( NOT aPartsClass->(BOOL)isMeronymicClassOf(myClass) )
    {
       prints('Error in method "addPart":'); endline();
       prints('   The given object is not an instance of an');
          endline();
       prints('   appropriate class.');
          endline(); endline();
       RETURN FALSE;
    }

//
//   Count the number of parts that this target object already
//   has from the given part's class.  If the addition of the
//   new part would violate the prescribed maximum, then
//   disallow the connection.
//

         count := 0;

     FORALL( p IN theParts )
     {
        IF( aPartsClass == p->(OID)class() )
                             count := count + 1;
     }

//
//  Check this part count against the max allowed for
//  this relshp.
//

     IF( count == myClass->(INT)maxCard(aPartsClass) )
     {
       prints('Error in method "addPart":'); endline();
       prints('   The addition of the given part to the whole');
          endline();
```

```
          prints('   would violate a prescribed cardinality
                                                   constraint.');
            endline();
          prints('Therefore, the part connection was not
                                             established.');
            endline();
          RETURN FALSE;
        }

//
//   Now check that the addition of the part to the target
//   object doesn't violate any of the exclusive/shared
//   constraints.
//

//
//   If the object is not a part of any holonym presently,
//   then the attachment may proceed regardless of the
//   type of part relationship.  (Of course, this condition
//   is also the one which must be satisfied if the current
//   part relationship is GLOBAL_EXCL.)
//

            aPartsHolonymSet := aPart->({OID})getWholes();

          IF( aPartsHolonymSet == {} )
          {
              //
              //  Go ahead and attach the part.
              //  Also, make the part aware that it is
              //  now attached to this whole.
              //

                INSERT aPart INTO theParts;
                aPart->addWhole(SELF);
                RETURN TRUE;
           }

//
//   Now, if the current part relshp is GLOBAL_EXCL, then
//   we must refuse to allow the connection because a
//   globally exclusively owned part, by definition, may
```

```
//   not belong to any other holonym in the database.
//

       theExShValue := myClass->(ExShType)exsh(aPartsClass);

     IF( theExShValue == GLOBAL_EXCL )
     {
        prints('Error in method "addPart":');endline();
        prints('   Given object is already a part of another
                                                    object.');
           endline();
        prints('   It cannot be made a global exclusive part
                                                    of the ');
           endline();
        prints('   desired holonym.');
           endline(); endline();
        RETURN FALSE;
     }

//
//   At this point we know that the present part relationship
//   (i.e., the one between the target object's class and the
//   argument's class) is either CLASS_EXCL or SHARED, and also
//   that the argument is already a part of something else.  So
//   we need to scan the argument's wholes to see if any has
//   a global exclusive hold on it.  If none does, then we
//   have the following two cases to consider:
//
//   1. If the current relationship is SHARED, then we
//      immediately attach the part.
//
//   2. If the current relationship is CLASS_EXCL, then
//      we have to make certain that no other object
//      from the target object's class (i.e., none
//      of the targets "siblings") has a hold on the
//      part already.
//

         FORALL( aWhole IN aPartsHolonymSet )
         {
                aWholesClass := aWhole->(OID)class();
```

```
    //
    //  Does the current whole have a global exclusive hold on
    //  the given part.
    //

              IF( aWholesClass->(ExShType)exsh(aPartsClass) ==
                                                      GLOBAL_EXCL )
              {
                 //
                 //   Yes; deny the current attachment request.
                 //

                prints('Error in method "addPart":');endline();
                prints('   Given object is already a part of
                                                  another object');
                   endline();
                prints('   which has a global exclusive hold
                                                         on it.');
                   endline(); endline();
        RETURN FALSE;
              }

  //
  //  If the current whole is a sibling of the target object, and
  //  the relationship is CLASS_EXCL, then we must deny the request.
  //

            ELSE IF( (theExShValue == CLASS_EXCL) &
                                (aWholesClass == myClass) )
            {
               prints('Error in method "addPart":'); endline();
               prints('   Given object is already a part of
                                                  another object');
                  endline();
               prints('   in the same class.'); endline();
                  endline();
               RETURN FALSE;
            }
          }
```

```
  //
  //   If we made it here, the other part-whole relationships
  //   that the argument participates in are compatible with
  //   the current one.  So, make the attachment.
  //

          INSERT aPart INTO theParts;
          aPart->addWhole(SELF);

          RETURN TRUE;
   };

//////////////////////////////////////////////////////////////////
//
//  addWhole is used to add a reference to a whole object of
//  which the target object is now a part.
//
//////////////////////////////////////////////////////////////////

   addWhole(aWhole : OID) READONLY;
   {
       INSERT aWhole INTO theWholes;
   };

/////////////////////////////////////////////////////////////////
//
//   Remove the given part from the whole.
//
/////////////////////////////////////////////////////////////////

   removePart(aPart : OID) : BOOL READONLY;
   {

      VAR aPartsClass: OID;
      VAR myClass: OID;
      VAR p: OID; VAR count: INT;

          aPartsClass := aPart->(OID)class();
          myClass := SELF->(OID)class();
```

```
//
//   If this part isn't one of my parts, then it cannot
//   be removed.
//

        IF( NOT (aPart IN theParts) )
        {
          prints('Error in method "removePart":'); endline();
          prints('   Given object is not a meronym of the
                                                   target.');
             endline();
          RETURN FALSE;
        }

//
//  check for a violation of the min cardinality allowed
//  for parts of this class.  First, count the number of
//  parts from the given part's class.
//

          count := 0;

     FORALL( p IN theParts )
     {
         IF( aPartsClass == p->(OID)class() )
                                 count := count + 1;
     }

//
//  Check this part count against the min allowed for
//  this relshp.
//

     IF( count == myClass->(INT)minCard(aPartsClass) )
     {
       prints('Error in method "removePart":'); endline();
       prints('   The removal of the given part from the whole');
          endline();
       prints('   would violate a prescribed cardinality
                                                constraint.');
          endline();
       prints('Therefore, the part connection was not broken.');
```

```
            endline();
          RETURN FALSE;
        }

//
//   Remove the given part from the list of parts.  Also, inform
//   the part that it should discard the reference to the whole.
//

                REMOVE aPart FROM theParts;
                aPart->removeWhole(SELF);

            RETURN TRUE;
   };

//////////////////////////////////////////////////////////////////
//
//  The following removes a part of the target object
//  w/o checking for any cardinality violations.  It is
//  included for use by the "destroy" function of the class.
//  It should *not* be used anywhere else.
//
//////////////////////////////////////////////////////////////////

   removePartPrivate(aPart: OID) READONLY;
   {
        REMOVE aPart FROM theParts;
   };

//////////////////////////////////////////////////////////////////
//
//   Remove the given whole from the part.
//
//////////////////////////////////////////////////////////////////

   removeWhole(aWhole : OID) READONLY;
   {
      IF(aWhole IN theWholes)
          REMOVE aWhole FROM theWholes;
   };
```

```
/////////////////////////////////////////////////////////////////
//
//   The following method exchanges one part for another, that
//   is, it removes the part which is its first argument and
//   replaces it with the part that is its second argument.
//
/////////////////////////////////////////////////////////////////

   changePart(oldPart: OID, newPart: OID) : BOOL READONLY;
   {

       VAR newPartsClass: OID;       // the 2nd argument's class;
                                     // must be the same as that
                                     // of the first.
       VAR newPartsHolonymSet: {OID}; // 2nd arg's holonym set.
       VAR myClass: OID;              // target's class.
       VAR theExShValue: ExShType;    // "current" part relshp's
                                      //  exclusive/shared value.
       VAR aWhole: OID;
       VAR aWholesClass: OID;

                newPartsClass := newPart->(OID)class();
                myClass := SELF->(OID)class();

   //
   //  First of all, if the two given parts are one and the same,
   //  then just return because there is nothing to do.
   //

             IF(oldPart == newPart)
                             RETURN FALSE;

   //
   //  If the proposed new part is already a part of the
   //  given whole, then no replacement would occur; the
   //  operation would effectively be a removal of the
   //  old part only.  This request must be denied.
   //

          IF( newPart IN theParts )
          {
             prints('Error in method "changePart":'); endline();
```

```
          prints('   The object given as the replacement
                                              part is');
             endline();
          prints('   already part of the given whole.');
             endline();
          prints('   Therefore, the replacement was not
                                           carried out.');
             endline(); endline();
          RETURN FALSE;
       }

//
// If the new part is not the same type as the old part,
// then issue an error and return.
//

     IF(oldPart->(OID)class() != newPart->(OID)class())
     {
       prints('Error in method "changePart":'); endline();
       prints('   The object given as the replacement
                                           part is not');
          endline();
       prints('   the same type as the one it is supposed to');
       prints(' replace.'); endline();
       prints('   Therefore, the replacement was not
                                            carried out.');
          endline(); endline();
       RETURN FALSE;
     }

//
// Now, we need to determine if the object given as the
// replacement part can legitimately be made a part of
// the target whole.  That is, we must determine whether
// or not such an arrangement violates any of the prescribed
// exclusive/shared constraints.  If there are no violations,
// then it is added, and the old part is removed.
// The following code is reminiscent of that in "addPart"
// above.
//
//  If the new object is not a part of any holonym presently,
//  then the attachment may proceed regardless of the
```

```
//   type of part relationship.
//

           newPartsHolonymSet := newPart->({OID})getWholes();

         IF( newPartsHolonymSet == {} )
         {
               //
               //  Go ahead and remove the old part
               //  and attach the new one.
               //  Also, make both parts aware of the
               //  new arrangement.
               //

                     REMOVE oldPart FROM theParts;
                     oldPart->removeWhole(SELF);

                     INSERT newPart INTO theParts;
                     newPart->addWhole(SELF);

                     RETURN TRUE;
         }

//
//   Now, if the current part relshp is GLOBAL_EXCL, then
//   we must refuse to allow the connection because a
//   globally exclusively owned part, by definition, may
//   not belong to any other holonym in the database.  And,
//   in particular, it made not be made a part of the target
//   object.
//

          theExShValue := myClass->(ExShType)
                                    exsh(newPartsClass);

      IF( theExShValue == GLOBAL_EXCL )
      {
         prints('Error in method "changePart":');endline();
         prints('   The given replacement object is
                                      already a part of ');
         prints('another object.'); endline();
         prints('   It cannot be made a global exclusive
```

```
                                                    part of the');
            prints(' desired holonym.'); endline();
            prints('   Therefore, the replacement was not
                                                  carried out.');
              endline(); endline();
            RETURN FALSE;
         }

//
//   At this point we know that the present part relationship
//   (i.e., the one between the target object's class and the
//   desired new part's class) is either CLASS_EXCL or SHARED,
//   and also that the new part already belongs to another
//   whole.  So we need to scan the new part's wholes to see
//   if any has a global exclusive hold on it.  If none does,
//   then we have the following two cases to consider:
//
//    1. If the current relationship is SHARED, then we
//         can immediately remove the old part and attach
//       the new one.
//
//      2. If the current relationship is CLASS_EXCL, then
//         we have to make certain that no other object
//         from the target object's class (i.e., none
//         of the target's "siblings") has a hold on the
//         part already.
//

     FORALL( aWhole IN newPartsHolonymSet )
     {

              aWholesClass := aWhole->(OID)class();

//
//  Does the current whole have a global exclusive hold on
//  the desired new part.
//

       IF(aWholesClass->(ExShType)exsh(newPartsClass) ==
                                                   GLOBAL_EXCL)
       {
```

```
          //
          //   Yes; deny the current attachment request.
          //

         prints('Error in method "changePart":'); endline();
         prints('   The desired new part is already a part of');
         prints(' another object');
            endline();
         prints('   which has a global exclusive hold on it.');
            endline();
         prints('   Therefore, the replacement was not
                                              carried out.');
            endline(); endline();
         RETURN FALSE;
       }


//
//  If the current whole is a sibling of the target object, and
//  the relationship is CLASS_EXCL, then we must deny the
//  request.
//


        ELSE IF( (theExShValue == CLASS_EXCL) &
                              (aWholesClass == myClass) )
        {
          prints('Error in method "changePart":'); endline();
          prints('   The desired new part is already a
                                                 part of');
             endline();
          prints(' another object in the same class.');
             endline();
          prints('   Therefore, the replacement was not
                                              carried out.');
             endline(); endline();
          RETURN FALSE;
        }

      } // FORALL
```

```
  //
  //  If we made it this far, then the other part-whole
  //  relationships that the new part participates in are
  //  compatible with the current one.  So, remove the old
  //  part and attach the new one.
  //

          REMOVE oldPart FROM theParts;
          oldPart->removeWhole(SELF);

          INSERT newPart INTO theParts;
          newPart->addWhole(SELF);

          RETURN TRUE;
   };

/////////////////////////////////////////////////////////////////
//
//   The following is the NOMETHOD clause for classes
//   which are in a part hierarchy.  It is invoked
//   when a particular meronym or holonym does not
//   has a method (i.e., an answer) for some message
//   which was passed to it.  In such cases, this
//   NOMETHOD "routine" first determines if the message
//   can be answered (handled) by one of the objects
//   wholes.  If so, the message is delegated there.
//   If not, the routine looks to see if the message
//   can be handled by one of the parts.  Once again,
//   if so, the message is passed onward.  If it cannot
//   be, the routine gives up and simply returns.
//
/////////////////////////////////////////////////////////////////

   NOMETHOD
   {

       VAR myClass: OID;
       VAR w: OID; VAR wsClass: OID;
       VAR p: OID; VAR psClass: OID;

            myClass := SELF->(OID)class();
```

```
   //
   //  First, check the wholes.
   //

         FORALL( w IN theWholes )
         {

                 wsClass := w->(OID)class();

             IF(wsClass->(BOOL)
                  propertyDownPropagated(currentMeth, myClass))

                       RETURN w->currentMeth(arguments);

         }

   //
   //  If not, try the parts.
   //

         FORALL( p IN theParts )
         {

                 psClass := p->(OID)class();

             IF(myClass->(BOOL)
                  propertyUpPropagated(currentMeth, psClass))

                       RETURN p->currentMeth(arguments);

         }

      }
END;

END_SCHEMA;
```

APPENDIX B

A SAMPLE VML PART SCHEMA

In this appendix, we show a sample VML schema based on the schema diagram of Figure 5.16. Note that the code only contains class declarations; object type declarations have been omitted.

```
SCHEMA editorialschema

   IMPORT HMMetaClass FROM HMMetaClass;

CLASS Newspaper METACLASS HolonymicMeronymicClass
   INSTTYPE newspaperType
 INIT Newspaper->defMeronymicRelshps
       ({[theMeronymicClass:EditorialPage,
                                  es:GLOBAL_EXCL,
                                  cardinality:[min: 1, max: 1],
                                  dependency:NONE,
                                  upSet:{},
                                  downSet:{'date'}]})
END;

CLASS EditorialPage METACLASS HolonymicMeronymicClass
   INSTTYPE editorialPageType
 INIT EditorialPage->defMeronymicRelshps(
                               {[theMeronymicClass:Masthead,
                                 es:SHARED,
                                 cardinality:[min: 1, max: 1],
                                 dependency:NONE,
                                 upSet:{},
                                 downSet:{}]
                                [theMeronymicClass:EditorialColumn,
                                 es:GLOBAL_EXCL,
                                 cardinality:[min: 1, max: 1],
                                 dependency:NONE,
                                 upSet:{},
                                 downSet:{'date'}],
```

```
                                          [theMeronymicClass:LettersColumn,
                                           es:GLOBAL_EXCL,
                                           cardinality:[min: 1, max: 1],
                                           dependency:NONE,
                                           upSet:{},
                                           downSet:{}]})

     EditorialPage->defHolonymicClasses( {Newspaper} )

END;

CLASS Masthead METACLASS HolonymicMeronymicClass
   INSTTYPE mastheadType
 INIT Masthead->defHolonymicClasses( {EditorialPage} )

END;

CLASS EditorialColumn METACLASS HolonymicMeronymicClass
   INSTTYPE editorialColumnType
 INIT EditorialColumn->defMeronymicRelshps(
                                      {[theMeronymicClass:Editorial,
                                        es:GLOBAL_EXCL,
                                        cardinality:[min: 3, max: 4],
                                        dependency:NONE,
                                        upSet:{},
                                        downSet:{'date'}]})

     EditorialColumn->defHolonymicClasses( {EditorialPage} )

END;

CLASS LettersColumn METACLASS HolonymicMeronymicClass
   INSTTYPE lettersColumnType
 INIT LettersColumn->defMeronymicRelshps(
                                         {[theMeronymicClass:BusinessMasthead,
                                           es:SHARED,
                                           cardinality:[min: 1, max: 1],
                                           dependency:NONE,
                                           upSet:{},
                                           downSet:{}],
                                          [theMeronymicClass:Letter,
                                           es:GLOBAL_EXCL,
```

```
                                    cardinality:[min: 1, max: INFINITY],
                                    dependency:NONE,
                                    upSet:{},
                                    downSet:{}]
                                   [theMeronymicClass:Illustration,
                                    es:CLASS_EXCL,
                                    cardinality:[min: 0, max: 1],
                                    dependency:NONE,
                                    upSet:{},
                                    downSet:{}]})

     LettersColumn->defHolonymicClasses( {EditorialPage} )

END;

CLASS Editorial METACLASS HolonymicMeronymicClass
   INSTTYPE editorialType
 INIT Editorial->defMeronymicRelshps(
                                 {[theMeronymicClass:TextSegment,
                                   es:GLOBAL_EXCL,
                                   cardinality:[min: 0, max: 1],
                                   dependency:NONE,
                                   upSet:{},
                                   downSet:{}]})

     Editorial->defHolonymicClasses( {EditorialColumn} )

END;

CLASS BusinessMasthead METACLASS HolonymicMeronymicClass
   INSTTYPE businessMastheadType
 INIT BusinessMasthead->defHolonymicClasses( {LettersColumn} )

END;


CLASS Letter METACLASS HolonymicMeronymicClass
   INSTTYPE letterType
 INIT Letter->defMeronymicRelshps(
                                 {[theMeronymicClass:TextSegment,
                                   es:GLOBAL_EXCL,
                                   cardinality:[min: 0, max: 1],
```

```
                                  dependency:NONE,
                                  upSet:{},
                                  downSet:{}]})

      Letter->defHolonymicClasses( {LettersColumn} )

END;

CLASS Illustration METACLASS HolonymicMeronymicClass
   INSTTYPE illustrationType
 INIT Illustration->defHolonymicClasses( {LettersColumn} )

END;

CLASS TextSegment METACLASS HolonymicMeronymicClass
   INSTTYPE textSegmentType
 INIT TextSegment->defHolonymicClasses( {Editorial, Letter} )

END;

END_SCHEMA;
```

Missing Page

Missing Page

[29] R. G. G. Cattell and T. R. Rogers. Entity-Relationship database user interfaces. In M. Stonebraker, editor, *Readings in Database Systems*, pages 359–368. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[30] N. Cercone. The ECO family. In [116], pages 95–131.

[31] R. Chaffin and D. J. Herrmann. Effects of relation similarity on part-whole decisions. *The Journal of General Psychology*, 115(2):131–139, 1988.

[32] R. Chaffin and D. J. Herrmann. Retrieval and comparison processes in part-whole decisions. *The Journal of General Psychology*, 116(4):393–406, 1989.

[33] E. E. Chang and R. H. Katz. Inheritance in computer-aided design databases: semantics and implementation issues. *Computer-Aided Design*, 22(8):489–499, Oct. 1990.

[34] H. Chao and V. P. Teli. Development of a university database using the Dual Model of object-oriented knowledge bases. Master's thesis, NJIT, Newark, NJ, 1990.

[35] S. Chatterjee. Graphical image persistence and code generation for OOdini. Master's thesis, NJIT, Newark, NJ, 1992.

[36] P. P.-S. Chen. The Entity-Relationship Model: Toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[37] S. Christodoulakis, J. Vanderbrook, J. Li, S. Wan, Y. Wang, M. Papa, and E. Bertino. Development of a multimedia information system for an office environment. In *Proc. VLDB '84*, pages 261–271, 1984.

[38] B. L. Clarke. A calculus of individuals based on 'connection'. *Notre Dame Journal of Formal Logic*, 22(3):204–218, 1981.

[39] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, NJ, second edition, 1991.

[40] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[41] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[42] J. Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, Sept. 1987.

[43] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proc. 1984 ACM SIGMOD Int'l Conference on the Management of Data*, pages 316–325, Boston, MA, June 1984.

[44] D. A. Cruse. On the transitivity of the part-whole relation. *Journal of Linguistics*, 15(1):29–38, 1979.

[45] J. A. Cuddon. *A Dictionary of Literary Terms.* Doubleday & Company, Inc., Garden City, NY, 1977.

[46] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Co., Inc., Reading, MA, fourth edition, 1986.

[47] O. Deux et al. The story of O_2. *IEEE Trans. Knowledge and Data Eng.*, 2(1):91–108, 1990.

[48] O. Deux et al. The O_2 system. *Commun. ACM*, 34(10):34–48, Oct. 1991.

[49] O. Diaz and P. M. Gray. Semantic-rich user-defined relationships as a main constructor in object-oriented databases. In *Proc. IFIP TC2 Conf. on Database Semantics.* North Holland, 1990.

[50] R. Elmasri and S. Navathe. Object integration in logical database design. In *Proc. Int'l Conference on Data Engineering*, pages 426–433, Los Angeles, CA, Apr. 1984.

[51] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Co., Inc., New York, NY, 1989.

[52] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to the entity-relationship model. *Int'l J. Data and Knowledge Eng.*, 1(1), May 1985.

[53] R. Elmasri and G. Wiederhold. Properties of relationships and their representation. In *Proc. Nat'l Comp. Conf.*, volume 49, pages 319–326. AFIPS, May 1980.

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

[55] D. Fischer. Consistency rules and triggers for multilingual terminology. To appear in *Proc. TKE93, 3rd Int'l Congr. Terminology and Knowledge Eng.*, 1993.

[56] D. Fischer et al. VML - The Vodak Data Modeling Language. Technical report, GMD-IPSI, Dec. 1989.

[57] D. H. Fishman et al. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. ACM Press, New York, NY, 1989.

[58] J. B. Fraleigh. *A First Course in Abstract Algebra.* Addison-Wesley Publishing Co., Inc., Reading, MA, third edition, 1982.

[59] J. Geller. *A Knowledge Representation Theory for Natural Language Graphics.* PhD thesis. SUNY Buffalo CS Department, 1988. Tech. Report 88-15.

[60] J. Geller. A graphics-based analysis of part-whole relations. Research Report CIS-91-27, CIS Department, NJIT, Nov. 1991.

[61] J. Geller. Propositional representation for graphical knowledge. *Int. J. Man-Machine Studies*, 34(1):97–131, 1991.

[62] J. Geller. Upward-inductive inheritance and constant time downward inheritance in massively parallel knowledge representation. In *IJCAI Workshop on Parallel Processing in AI*, pages 63–68, Sydney, Australia, 1991.

[63] J. Geller and Y. Du. Parallel implementation of a class reasoner. *Journal of Experimental and Theoretical Artificial Intelligence*, 3:109–127, 1991.

[64] J. Geller, A. Mehta, Y. Perl, E. Neuhold, and A. Sheth. Algorithms for structural schema integration. In *Proc. Second Int'l Conf. on Systems Integration*, pages 604–614, Morristown, NJ, June 1992.

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

[66] J. Geller, Y. Perl, P. Cannata, A. Sheth, and E. Neuhold. A case study of structural integration. In Y. Yesha, editor, *Proc. 1st Int'l Conference on Information and Knowledge Management*, pages 102–111, Baltimore, MD, Nov. 1992.

[67] J. Geller, Y. Perl, P. Cannata, A. Sheth, and E. Neuhold. Structural integration: Concepts and case study. To appear in *Journal of Systems Integration*, 1993.

[68] J. Geller, Y. Perl, and E. Neuhold. Structural schema integration in heterogeneous multi-database systems using the Dual Model. In *Proc. First Int'l Workshop on Interoperability in Multidatabase Systems*, pages 200–203, Los Alamitos, CA, 1991. IEEE Computer Society Press.

[69] J. Geller, Y. Perl, and E. Neuhold. Structure and semantics in OODB class specifications. *SIGMOD Record*, 20(4):40–43, Dec. 1991.

[70] J. Geller, Y. Perl, E. Neuhold, and A. Sheth. Structural schema integration with full and partial correspondence using the Dual Model. *Information Systems*, 17(6):443–464, Dec. 1992.

[71] J. Geller and S. Shapiro. Graphical deep knowledge for intelligent machine drafting. In *Tenth Int'l Joint Conference on Artificial Intelligence*, San Mateo, CA, 1987. Morgan Kaufmann Publishers, Inc.

[72] M. Gemis, J. Paredaens, and I. Thyssens. A visual database management interface based on GOOD. In R. Cooper, editor, *Interfaces to Database Systems*, pages 155–175. Springer-Verlag, London, 1993.

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

[74] J. Gomsi and M. DeSanti. Versant overview. In *Proc. Exec. Briefing on Object-Oriented Database Management*, pages 107–109, San Francisco, CA, 1991.

[75] N. Goodman. *The Structure of Appearance.* Reidel, Dordrecht, third edition, 1977.

[76] K. Gorman and J. Choobineh. The Object-Oriented Entity-Relationship Model (OOERM). *Journal of Management Information Systems*, 7(3):41–65, 1991.

[77] A. C. Graesser, S. E. Gordon, and L. E. Brainerd. QUEST: A model of question answering. In [116], pages 733–745.

[78] N. Griffin. *Relative Identity.* Clarendon Press, Oxford, 1977.

[79] R. Gupta et al. The development of a framework for VLSI CAD. In [80], pages 237–260.

[80] R. Gupta and E. Horowitz, editors. *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[81] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proc. Ninth ACM Symposium on Principles of Database Systems*, pages 24–33, Nashville, TN, Apr. 1990.

[82] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object model for database end-user interfaces. In H. Garcia-Molina and H. V. Jagadesh, editors, *Proc. 1990 ACM SIGMOD Int'l Conference on Management of Data*, pages 24–33, Atlantic City, NJ, May 1990. ACM.

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

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

[85] M. Halper, J. Geller, and Y. Perl. On mereological modeling in object-oriented databases. Submitted for journal publication, 1993.

[86] M. Halper, J. Geller, and Y. Perl. Value propagation in OODB part hierarchies. To appear in *Proc. 2nd Int'l Conference on Information and Knowledge Management*, 1993.

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

[88] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Trans. Database Syst.*, 6(3):351–386, 1981.

[89] D. J. Hartzband and F. Maryanski. Enhancing knowledge representation in engineering databases. *Computer*, 18(9):39–48, Sept. 1985.

[90] G. E. Hinton. Representing part-whole hierarchies in connectionist networks. In *Proc. 10th Cog. Sci. Soc. Conference*, pages 48–54, 1988.

[91] W. Horak. Office document architecture and office document interchange formats: Current status of international standardization. *Computer*, 18(10):50–60, Oct. 1985.

[92] W. Horak and G. Krönert. An object-oriented office document architecture model for processing and interchange of documents. In *Second ACM-SIGOA Conf. on Office Information Systems*, pages 152–160, Toronto, Canada, June 1984.

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

[94] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, Sept. 1987.

[95] IEEE Computer Society. *Proc. Second Int'l Conference on Data and Knowledge Systems for Manufacturing and Eng.*, Gaithersburg, MD, Oct. 1989.

[96] M. A. Iris, B. E. Litowitz, and M. W. Evens. Problems of the part-whole relation. In M. W. Evens, editor, *Memory and Learning–The Ebbinghaus Centennial Conference*. Cambridge Univ. Press, New York, NY, 1988.

[97] G. Kappel and M. Schrefl. Object/Behavior diagrams. In *Proc. 7th Int'l Conference on Data Eng.*, pages 530–539, Kobe, Japan, Apr. 1991.

[98] R. Katz, E. Chang, and R. Bhateja. Version modeling concepts for computer-aided design databases. In *Proc. 1986 ACM SIGMOD Conference on Management of Data*, Washington, D.C., May 1986.

[99] R. H. Katz and E. Chang. Managing change in a computer-aided design database. In *Proc. VLDB '87*, pages 455–462, 1987.

[100] S. E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1989.

[101] A. B. Kempe. A memoir on the theory of mathematical form. *Phil. Trans. Royal Society London*, 177:1–70, 1886.

[102] A. B. Kempe. A correction and explanation. *The Monist*, 7:453–458, 1897.

[103] W. Kent. *Data and Reality.* North-Holland, Amsterdam, 1978.

[104] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. 1992 ACM SIGMOD Conference on Management of Data*, San Diego, CA, June 1992.

[105] H.-J. Kim. Algorithmic and computational aspects of object-oriented database schema design. In [80], pages 26–61.

[106] W. Kim. A model of queries for object-oriented databases. In *Proc. 15th VLDB*, pages 423–432, 1989.

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

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

[109] W. Klas et al. Vodak design specification document. Technical report, GMD-IPSI, Nov. 1992.

[110] W. Klas, E. J. Neuhold, and M. Schrefl. On an object-oriented data model for a knowledge base. In R. Speth, editor, *Research into Networks and Distributed Applications. EUTECO 88.* North-Holland, 1988.

[111] S. Kuncham. Graphical representation of object-oriented database. Master's Project, NJIT, Newark, NJ, 1991.

[112] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.

[113] L. Lamport. $\LaTeX$*: A Document Preparation System.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1986.

[114] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 227–236. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[115] F. Lehmann. Semantic networks. In [116], pages 1–50.

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

[117] D. B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1990.

[118] D. B. Lenat, M. Prakash, and M. Shepherd. Cyc: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *The AI Magazine*, 6(4):65–85, 1986.

[119] H. S. Leonard and N. Goodman. The Calculus of Individuals and its uses. *J. Symbolic Logic*, 5:45–55, 1940.

[120] B. Lingan and M. Tulasiram. Implementation of an object-oriented university database using VODAK/VML. Master's Project, NJIT, Newark, NJ, 1993.

[121] L.-C. Liu and E. Horowitz. Object database support for CASE. In [80], pages 261–282.

[122] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Commun. ACM*, 34(10):94–109, Oct. 1991.

[123] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems.* The Benjamin/Cummings Publishing Co., Inc., New York, NY, 1989.

[124] S. L. Lytinen. Conceptual Dependency and its descendants. In [116], pages 51–73.

[125] B. MacKellar and F. Ozel. ArchObjects: Design codes as constraints in an object-oriented KBMS. In J. Gero, editor, *AI in Design '91*. Butterworth-Heinemann Ltd., 1991.

[126] B. MacKellar and J. Peckham. Data modeling support for design databases. Manuscript in preparation. 1991.

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

[128] R. R. Madapati. User interface for schema operations for OOdini graphical schema editor. Master's Project, NJIT, Newark, NJ, 1992.

[129] Mark V Systems, Ltd., Encino, CA. ObjectMaker documentation, 1993.

[130] J. A. Markowitz, J. T. Nutter, and M. W. Evens. Beyond IS-A and part-whole: More semantic network links. In [116], pages 400-407.

[131] G. McCalla, J. Greer, B. Barrie, and P. Pospisil. Granularity hierarchies. In [116], pages 363–375.

[132] D. McLeod and J. M. Smith. Abstraction in databases. In M. L. Brodie and S. N. Zilles, editors, *Proc. Workshop on Data Abstraction, Databases and Conceptual Modelling*, pages 19–25, Pingree Park, CO, June 1980.

[133] A. Mehta. *Algorithms for Generation of Path-Methods in Object-Oriented Databases.* PhD thesis, NJIT, May 1993.

[134] A. Mehta, J. Geller, Y. Perl, and P. Fankhauser. Algorithms for access relevance to support path-method generation in OODBs. In *Proc. Fourth Int'l Hong Kong Comp. Soc. Database Workshop*, pages 183–200, Shatin, Hong Kong, Dec. 1992.

[135] A. Mehta, J. Geller, Y. Perl, and P. Fankhauser. Computing access relevance for path-method generation and IM-OODB. Submitted for journal publication, 1993.

[136] A. Mehta, J. Geller, Y. Perl, and E. Neuhold. The OODB path-method generator (PMG) using access weights and precomputed access relevance. Submitted for journal publication, 1993.

[137] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, third edition, 1987.

[138] B. Meyer. Tools for the new culture: Lessons from the design of the Eiffel libraries. *Commun. ACM*, 33(9):68–88, Sept. 1990.

[139] H. Mili and R. Rada. A model of hierarchies based on graph homomorphisms. In [116], pages 343–361.

[140] B. A. Myers et al. Garnet. comprehensize support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, Nov. 1990.

[141] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A language facility for designing database-intensive applications. *ACM Trans. Database Syst.*, 5(2):185–207, June 1980.

[142] E. Neuhold, Y. Perl, J. Geller, and V. Turau. Separating structural and semantic elements in object-oriented knowledge bases. In *Proc. of the Advanced Database System Symposium*, pages 67–74, Kyoto, Japan, 1989.

[143] E. Neuhold, Y. Perl, J. Geller, and V. Turau. The Dual Model for object-oriented databases. Research Report CIS-91-30, NJIT, 1991.

[144] E. J. Neuhold and M. Schrefl. Dynamic derivation of personalized views. In *Proc. 14th Int'l Conference on Very Large Databases*, Long Beach, CA, 1988.

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

[146] A. Nye. *X Protocol Reference Manual.* O'Reilly & Associates, Inc., Sebastopol, CA, 1989.

[147] A. Nye. *Xlib Programming Manual.* O'Reilly & Associates, Inc., Sebastopol, CA, 1989.

[148] A. Nye and T. O'Reilly. *X Toolkit Intrinsics Programming Manual.* O'Reilly & Associates, Inc., Sebastopol, CA, 1989.

[149] Ontologic, Inc., Burlington, MA. ONTOS 2.01 documentation, 1991.

[150] Open Software Foundation. *OSF/Motif Programmer's Guide.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[151] Open Software Foundation. *OSF/Motif Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[152] Open Software Foundation. *OSF/Motif Style Guide.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[153] T. O'Reilly, editor. *X Toolkit Intrinsics Reference Manual.* O'Reilly & Associates, Inc., Sebastopol, CA, 1989.

[154] M. A. Papalaskaris and L. K. Schubert. Parts inference: Closed and semi-closed partitioning graphs. In *Proc. IJCAI-81*, 1981.

[155] J. Peckham and F. Maryanski. Semantic data models. *ACM Comput. Surv.*, 20(3):153–189, Sept. 1988.

[156] J. Peckham, F. Maryanski, G. Beshers, H. Chapman, and S. Demurjian. Constraint based analysis of database update propagation. In *Proc. Tenth Int'l. Conf. on Information Systems*, pages 9–18, 1989.

[157] J. Peters, editor. *The Bookman's Glossary.* R. R. Bowker Company, New York, NY, sixth edition, 1983.

[158] C. S. Pierce. *The New Elements of Mathematics.* Mouton/Humanities Press, Atlantic Highlands, NJ, 1976. In four volumes.

[159] K. Radermacher. Abstraction techniques in semantic modelling. In H. Jaakkola et al., editors, *Information Modelling and Knowledge Bases IV.* IOS Press, Amsterdam, 1993.

[160] K. Radermacher. An extensible graphical programming environment for semantic modelling. In R. Cooper, editor, *Interfaces to Database Systems*, pages 353–373. Springer-Verlag, London, 1993.

[161] D. A. Randell and A. G. Cohn. Exploiting lattices in a theory of space and time. In [116], pages 459–476.

[162] B. Raphael. Sir: Semantic information retrieval. In M. Minsky, editor, *Semantic Information Processing.* MIT Press, Cambridge, MA, 1968.

[163] N. Rescher. Axioms for the part relation. *Philosophical Studies*, 6:8–11, 1955.

[164] E. Rich and K. Knight. *Artificial Intelligence.* McGraw-Hill, Inc., New York, NY, second edition, 1991.

[165] D. D. Roberts. The existential graphs. In [116], pages 639–663.

[166] E. Rosch. Principles of categorization. In E. Rosch and B. B. Lloyd, editors, *Cognition and Categorization*, pages 27–48. Lawrence Erlbaum Associates, 1978.

[167] R. J. Rost. *X and Motif Quick Reference Guide.* Digital Press, 1990.

[168] L. A. Rowe. A shared object hierarchy. In *Proc. Int'l Workshop on Object-Oriented Database Systems*, pages 160–170, Asilomar, CA, 1986.

[169] L. A. Rowe and M. Stonebraker. The POSTGRES data model. In *Proc. 13th Int'l Conference on Very Large Databases*, pages 83–95, Brighton, England, 1988.

[170] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proc. OOPSLA-87*, pages 466–481, Oct. 1987.

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

[172] K. Sayers. A Smalltalk implementation of an OODB part model. Master's Thesis, NJIT, Newark, NJ, 1993. In preparation.

[173] R. Schank. Conceptual Dependency: A theory of natural language understanding. *Cognitive Psychology*, 3:552–631, 1972.

[174] R. Schank. *Conceptual Information Processing.* North-Holland, Amsterdam, 1975.

[175] R. W. Scheifler and J. Gettys. The X window system. *ACM Trans. Gr.*, 5(2):79–109, Apr. 1986.

[176] R. W. Scheifler and J. Gettys. *X Window System: The Complete Reference.* Digital Press, second edition, 1990.

[177] P. Scheuermann, G. Scheffner, and H. Weber. Abstraction capabilities and invariant properties modelling within the entity-relationship approach. In P. Chen, editor, *Entity-Relationship Approach to Systems Analysis and Design*, pages 121–140. North-Holland, Amsterdam, 1980.

[178] U. Schiel. Abstractions in semantic networks: Axiom schemata for generalization, aggregation, and grouping. *SIGART Newsletter*, (107):25–26, Jan. 1989.

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

[180] M. Schrefl and E. J. Neuhold. Object class definition by generalization using upward inheritance. In *Proc. 4th Int'l Conference on Data Engineering*, pages 4–13, Los Angeles, CA, Feb. 1988.

[181] L. Schubert, M. Papalaskaris, and J. Taugher. Accelerating deductive inference: Special methods for taxonomies, colors and times. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 187–220. Springer Verlag, New York, NY, 1987.

[182] Servio Corp. Personal communication, 1993. Object Expo '93.

[183] S. Shah and G. Nadella. Graphical representation of object-oriented database. Master's Project, NJIT, Newark, NJ, 1991.

[184] S. C. Shapiro and W. J. Rapaport. The SNePS family. In [116], pages 243–275.

[185] A. P. Sheth and S. K. Gala. Attribute relationships: An impediment in automating schema integration. In *Workshop on Heterogeneous Database Systems*, Chicago, IL, 1989.

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

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

[188] E. E. Smith and D. L. Medin. *Categories and Concepts.* Harvard University Press, Cambridge, MA, 1981.

[189] J. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst.*, 2(2):105–133, 1977.

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

[191] V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, and O_2. *SIGMOD Record*, 21(1):93–104, Mar. 1992.

[192] J. F. Sowa. Conceptual graphs as a universal knowledge representation. In [116], pages 75–93.

[193] J. F. Sowa. Toward the expressive power of natural language. In [195], pages 157–189.

[194] J. F. Sowa. *Conceptual Structures, Information Processing in Mind and Machine.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1984.

[195] J. F. Sowa. *Principles of Semantic Networks, Explorations in the Representation of Knowledge.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

[196] M. Stonebraker, M. Hearst, and S. Potamianos. A commentary on the POSTGRES rules system. *SIGMOD Record*, 18(3):5–10, Sept. 1989.

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

[198] M. Stonebraker, E. Wong, P. Kreps, and G. Held. Design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[199] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1986.

[200] B. Stroustrup. An overview of C++. *SIGPLAN Notices*, 21(10):7–18, Oct. 1986.

[201] I. Sun Microsystems. OPEN LOOK, *Graphical User Interface Functional Specification.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1989.

[202] A. Tarski. *Logic, Semantics, Metamathematics.* Clarendon Press, Oxford, 1956. trans. J. H. Woodger.

[203] J. E. Tiles. *Things That Happen.* Aberdeen University Press, Aberdeen, Scotland, 1981.

[204] D. S. Touretzky. Implicit ordering of defaults in inheritance systems. In *Proc. IJCAI-84*, pages 322–325, Austin, TX, 1984.

[205] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, second edition, 1982.

[206] P. Venkatesh. Representation of graphical deep knowledge in an object-oriented database. Master's thesis, NJIT, Newark, NJ, 1991.

[207] J. Walters and N. R. Nielsen. *Crafting Knowledge-Based Systems.* John Wiley & Sons, New York, NY, 1988.

[208] P. Wegner. An object-oriented classification paradigm. In Schiver and Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[209] C. Wijaya and M. Ahmedi. Development of a university database (registration and admission) using the Dual Model for object-oriented knowledge bases. Master's thesis, NJIT, Newark, NJ, 1990.

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

[211] D. Woelk, W. Kim, and W. Luther. An object-oriented approach to multimedia databases. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 311–325, Washington, D.C., May 1986.

[212] N. Yankelovich, N. Meyrowitz, and A. van Dam. Reading and writing the electronic book. *Computer*, 18(10):15–30, Oct. 1985.

[213] D. A. Young. *The X Window System, Programming and Applications with Xt.* Prentice Hall, Englewood Cliffs, NJ, OSF/Motif edition, 1990.

[214] S. B. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 1–32. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.