UC Irvine
ICS Technical Reports

Title
PolyView : an object-oriented data model for supporting multiple user views

Permalink
https://escholarship.org/uc/item/8c00d2tp

Author
Gilbert, Jonathan Paul

Publication Date
1990

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 90-05

UNIVERSITY OF CALIFORNIA

IRVINE

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

PolyView: An Object-Oriented Data Model For Supporting Multiple User Views

Technical Report #90-05

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Jonathan Paul Gilbert

Dissertation Committee:

Professor Lubomir Bic, Chair

Professor Dennis Kibler

Professor Alexandru Nicolau

1990

© 1990

Jonathan Paul Gilbert

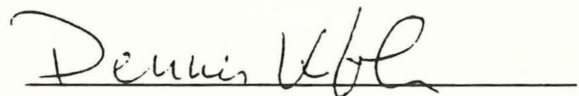
ALL RIGHTS RESERVED

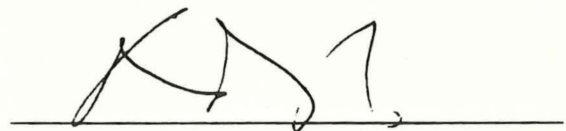
© 1990
Jonathan Paul Gilbert
ALL RIGHTS RESERVED

The dissertation of Jonathan Paul Gilbert is approved,

and is acceptable in quality and form for

publication on microfilm:





Committee Chair

University of California, Irvine

1990

Dedication

This work is dedicated to my wife,

Darlene,

who has always encouraged and supported me,

and to our families,

in both England and the U.S.

Contents

List of Figures . . . . *vi*

Acknowledgements . . . . *ix*

Curriculum Vitae . . . . *x*

Abstract . . . . *xi*

Chapter 1: Preliminary Remarks 1

Motivation 1

Contributions 3

Road Map 4

Chapter 2: The Evolution of Data Modeling 6

Organizing Data 6

Higher Level Models 11

Learning From Other Areas 18

Supporting Relativism 24

Chapter 3: The PolyView Data Model 30

The Basics 31

Derived Data and Groupings 39

The Object Structure 44

Summary — A Unique Framework for Supporting Relativism 52

Chapter 4: Asynchronous Message-Driven Processing 53

PolyView Structures: Objects, Views and Queries 53

Message-Driven Processing 57

Updating PolyView Databases through View Templates 71

Chapter 5: Supporting Semantic Relativism 85

Motivation 85

Basic View Transformations 88

Applying Sequences of Transformations 109
Chapter 6: Concluding Remarks 125
Summary 125
Directions for Future Work 126
References 128
Appendix 1: Object and Message Structures 134
Appendix 2: Generic Methods 141

List of Figures

Figure		Page
1.	An Entity-Relationship Diagram	13
2.	Integration by Alternative Generalization	26
3.	The Basic PolyView Class Hierarchy	33
4.	The Global Symbol Table Structure ...	37
5.	Single Branch "Modes-of-Transportation"	40
6.	An Example of Derived Data	45
7.	Derived Classes: Internal Representations	46
8.	The Global Symbol Table Including Derived Classes	47
9.	A Class Object	48
10.	The Generic Message Processing Strategy	49
11.	Object Structure Including Semi-Public Interfaces (Views)	55
12.	The User Support System Architecture ...	59
13.	The Four Retrieval Message Types	66
14.	External Schemas	68
15.	Internal Object Descriptions	69
16.	A Sample Query (part 1)	70
17.	A Sample Query (part 2)	71
18.	Find Insert Target Set	74
19.	Partially Expanded External Schemas	77
20.	Partially Expanded Internal Object Descriptions	78
21.	A Sample Insert Operation	79
22.	Deleting an Instance from a PolyView Database	82
23.	Applying the Name Method ...(An Internal Representation)	91
24.	Applying the Name Method ... (An External Representation)	92

25. Applying the Color Transformation 93

26. The Clone Method is Applied by the Cars Class ... (Internal ...) . . 94

27. The Clone Method is Applied by the Cars Class ... (External ...) . 95

28. The Clone Method is Applied by the Cars ... (Global Symbol Table) 96

29. The Attach Method Applied by the Cars and Chevys (Internal) . . 97

30. The Attach Method Applied by the Boats and Chevys (External) . 98

31. The Hide Method Applied by the Cars Class (An Internal Snapshot) 100

32. The Hide Method Applied by the Cars Class (An External Snapshot) 101

33. The Effect of Sending a Hide Message ... on the Global Symbol Table 101

34. The A-Name Method (Applied by Employees ... Internal) 102

35. The A-Name Method (Applied by Employees ... External) 103

36. The A-Remove Method (Applied by Employees ... Internal). 103

37. The A-Remove Method (Applied by Employees ... External) 104

38. The A-Restrict Method (Applied by Employees ... Internal). 105

39. The A-Restrict Method (Applied by Employees ... External) 106

40. The A-Insert Method (Applied by Employees and Cars — Internal) 108

41. The A-Insert Method (Applied by Employees and Cars — External) 109

42. The A-Move Method (... Internal). 110

43. The A-Move Method (... External) 111

44. The Global Symbol Table for the Simple Animal Taxonomy. 113

45. The Internal Structure (Before Creation of the New Views) 114

46. The External Structure (the Simple Animal Taxonomy) 115

47. The Three Year Old's Point of View. 115

48. The Zoo Keeper's Point of View. 116

49. The Global Symbol Table (After Creation of the New Views). . . . 117

50. The Internal Structure (After Creation of the New Views). 119

51. The Collapse Abstraction (A Global Symbol Table). 120

52. The Collapse Abstraction (The "After" Internal View) 121
53. The Collapse Abstraction (An External View) 122
54. A More Detailed Look at Tom and Kitty 123

Acknowledgements

Serious work on any complex subject cannot be accomplished without help. I owe a debt of gratitude to many members of the ICS community.

I have benefited from a particularly supportive group of faculty and graduate students. Among the most significant were Dan Easterlin, Doug Fisher, Rogers Hall, Dennis Kibler, Craig Lee, George Lueker, Alex Nicolau, Rami Razouk, and Kären Wiekert. I am particularly grateful to my officemates, Elke Rundensteiner, Wang-chan Wong and Meng-lai Yin, for their encouragement and for listening patiently to long explanations about my research.

The staff at the ICS department has been an invaluable asset. I would particularly like to thank Rose Allen, Mary Day, Pat Harris, Susan Hyatt, Candy Mamer, Fran Paz and Phyllis Siegel.

Finally, I would like to express my appreciation to my advisor, Lubomir Bic, for his initial suggestions and for helping to fine tune my ideas.

Curriculum Vitae
Jonathan Paul Gilbert

March 30, 1954	Born London, England
1979	B.S. in Computer Science, University of California, Irvine
1979-1984	Teaching Assistant, Dept. of Information and Computer Science, University of California, Irvine
1979-1982	Visiting Lecturer (summer sessions), Dept. of Information and Computer Science, University of California, Irvine
1982	M.S. in Computer Science, University of California, Irvine
1984-1988	Research Assistant, Dept. of Information and Computer Science, University of California, Irvine
1987	Consultant, Nadek Computer Systems, Tustin, California
1988-present	Engineer/Scientist, McDonnell Douglas Space Systems Company, Huntington Beach, California
1990	Ph.D. in Computer Science, University of California, Irvine

Publications

Learning from AI: New Trends in Database Technology, *IEEE Computer*, Vol. 19,3, March 1986, (with L. Bic)

Asynchronous Data Retrieval from an Object-Oriented Database, *Proc. European Conference on Object Oriented Programming*, Springer-Verlag, August 1988, (with L. Bic)

Set-Related Restrictions for Semantic Groupings, Technical Report 89-07, Dept. of Information and Computer Science, University of California, Irvine, January 1989, (with E. Rundensteiner, L. Bic and M. Yin)

Abstract of the Dissertation

PolyView: An Object-Oriented Data Model For Supporting Multiple User Views

by

Jonathan Paul Gilbert

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1990

Professor Lubomir Bic, Chair

In a typical database application, there are many different users with a great variety of skills, needs and perceptions. The problem of supporting this plethora of *user views* in a dynamic, data intensive environment is the topic of this dissertation.

In traditional record-based systems, all information is represented by an idealized data structure and a set of operations on that structure. User views are defined by simple variations in this structure, such as permuting field names, selecting a subset of the data, or creating links between records. Semantic database models support more complex, "natural" structures. It is often claimed that relativism is supported because semantic schemas can be correctly *interpreted* (by users) in different ways. The object-oriented paradigm, with its simple and elegant structural semantics, provides both simplicity and richness. Unfortunately, current object-oriented systems only provide a single object interface (or protocol). This dissertation presents *PolyView*; an object-oriented data model capable of simultaneously supporting many points of view. In *PolyView*, objects encapsulate a single structure and *any number* of object interfaces (*view instance descriptions*). *PolyView*, therefore, supports *distributed* mappings from user views to the underlying database structure.

Algorithms are presented for generic methods which retrieve and update information through user views. *PolyView* "colors" queries (messages) by attaching a view identity to them. As messages are propagated through the schema, *each* receiving object uses the color to determine how the message is to be processed. The color is used to select the user's protocol and allows different user's queries to be processed through apparently different database structures. Because objects act independently, *PolyView* is a data-driven system; messages are processed without any centralized control or shared memory.

Finally, *PolyView* provides a set of view transformations which allow view administrators to build object interfaces. Since views are supported by both global and localized mechanisms, there are transformations which operate at each of these levels. There are three major categories of transformations presented in this thesis: those which customize the schema as a whole, transformations for changing the structure of the IS-A hierarchy and transformations for customizing attributes.

CHAPTER 1
Preliminary Remarks

Motivation

The study of modeling and organizing large data intensive applications is a relatively new and rapidly expanding field. During the last two decades, several data models were developed to facilitate the efficient organization of highly structured data on magnetic disk. As computer technology has become more accessible, more information has become available to more people than ever before. This proliferation of computer based information systems has caused an increasing need for user-oriented systems and new priorities have become apparent which are beyond the scope of traditional data models. A great deal of effort is now being concentrated on the development of:

1. Models which incorporate higher level abstractions for capturing the semantics of traditional database applications.
2. Models that are suitable for non-traditional data intensive applications (like office automation and computer aided design and manufacturing).
3. Better "user-oriented" environments which include, for example, menu-driven and graphic interfaces.
4. Systems which support many different, perhaps conflicting, perspectives of the information content and organization of the data.

CHAPTER 2
The Evolution of Data Modeling

Organizing Data

Traditionally, little interaction has existed between researchers in the areas of database, artificial intelligence and programming languages. Formalisms were developed independently in each of these areas as solutions to apparently quite different problems. Recent trends have shown that techniques developed in one of these areas may also be applicable to problems in the other two. The major interest in this work is the evolution of database modeling formalisms and the development of enhanced database management systems. It is from this perspective that the overlap between these three major areas of computer science research will be examined.

Database management systems evolved in response to the need for efficiently maintaining increasingly large amounts of data. The relatively slow speed of secondary storage devices which hold the data is one of the main limitations to database design. Hence, the internal organization and structuring of databases has been the primary focus of research in the past. Another major influence of database research was the need to share information among a variety of users. In such an environment, strict rules governing the manipulation of data had to be imposed to preserve the integrity of the database and to guarantee privacy for each user.

The need to organize data in some well defined, rigorous manner led to the development of a number of *classical* data models [DATE81, ULLMAN82]. These first models, the best known of which are the *relational* data model [CODD70, KIM79,

BRODIE81] the *hierarchical* data model [MCGEE77, TSICH77] and the *network* data model [TAYLOR76, TSICH78], are variations of the *record model* [KENT78]. Data are arranged in fixed *linear* sequences of field values; they are machine oriented (organized for efficiency of storage and retrieval operations) and each model is based on some idealized data structure. The record model, which was easily adapted to the computer environment, is often awkward to the inexperienced user and is frequently semantically inadequate for modeling the application environment.

The Relational Model

In a relational database, information is organized in tables. Each table has a unique name and is a special case of the set-theoretic *relation.* The rows of a relation table are called *tuples*; columns are called *attributes.* Each column within a relation has a unique name. The set of values from which actual values in a column are selected is called the *domain* of the attribute and may be shared among different columns. A relation name and its set of attribute names is called a *relational schema* and a collection of relational schemas is a *relational database.* One of the most important features, and perhaps the biggest drawback of the relational approach, is that associations between tuples are exclusively represented by *attribute values drawn from a common domain.*

The main attraction of the relational model is its mathematical clarity which facilitates the formulation of non-procedural, high level queries and thus separates the user from the internal organization of data. Among the three classical database models, the relational model is, therefore, considered the most user oriented. It is, however, far from able to satisfy the needs and requirements of an increasingly diversified user community.

2 In order to support more user oriented interfaces, the database must provide a mechanism for storing *meta-knowledge.* This meta-knowledge would include information about the database itself and the ways in which it is used.

3. Most database models have only two levels: the database schema, which describes the data types constituting the database, and the actual collection of records, which are instances of the existing types. There are no provisions to extend the two levels into a more general hierarchy of types, meta-types, subtypes, and instances, even though this extension would increase the model's expressive power and provide a mechanism which supports the reuse of common properties.

4. Another problem is the inability to distinguish between a type and a set — records which form the schema usually implicitly represent both. Consider, for example, an employee database. The schema will describe the form of an employee record by listing the names of possible attributes each employee might have. The actual attribute values are kept with each employee record. From this point of view, the schema contains the description of a "typical employee". Alternately, the schema could be interpreted as representing the "set of all employees" constituting the database. Taking the latter point of view, it should be possible to include attributes that apply to the set as a whole but not to each individual element (e.g. the set cardinality or the average salary of an employee). Unfortunately, most conventional database models have no facilities for capturing this information.

5. When modeling an enterprise, elements and concepts representing it may be viewed from different perspectives depending on the application. In particular, the concepts of entity, relationship, and attribute may be interchangeable. Similarly, what is considered a type from one user's point of

view may be seen as an instance in another. The ability to model such phenomena, referred to as *relativism*, is missing in conventional database models.

6. The distinction between data and program in a database system was always clear in the past but is disappearing. New models are needed which capture the behavioral aspects of the database enterprise as well as its structure.

Higher Level Models

In the field of database research there have been two basic approaches to solving (some or all of) these problems. Attempts have been made to extend the classical models by building higher level conceptual models on top of a conventional database. New more powerful *semantic* data models have also been developed to capture database concepts at a more user oriented level.

To better facilitate the design of large database systems, many data models and techniques have been developed to enhance the classical models. These modeling tools include: normalization techniques for relational schemas [DATE81, KENT83], introducing the concept of the database abstraction [SMITH77], categorizing database entities [CHEN76, CODD79], and introducing new data models [HAMMER81, SHIPMAN81]. The systems cited are neither all inclusive nor mutually exclusive. For example, the concepts of generalization and (Cartesian) aggregation can be found in most models that have been developed since they were introduced in [SMITH77] by J. Smith and D. Smith.

The Entity-Relationship Model

One of the first steps taken toward developing a higher level data model was the Entity-Relationship (ER-)model, introduced by Chen [CHEN76]. It was aimed primarily at the first deficiency listed above — an increased orientation toward the user needs and expectations rather than machine efficiency. The ER-model may

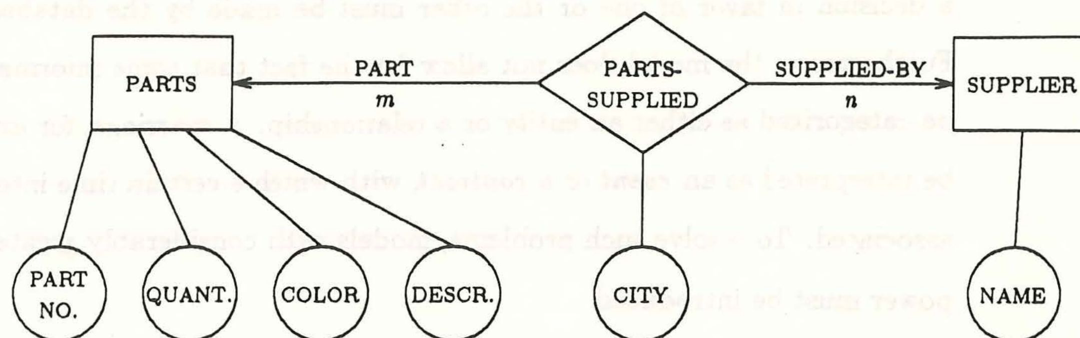
be viewed as a *generalization* of the three classical data models. Chen presented it as "... *a basis for unification of different views of data* ..." — in other words, it emphasizes the similarities and not the differences in the classical models.

The basic components of the ER-model are *entity sets* and *relationship sets*, where each entity set and each relationship set represents a generic classification of *entities* and *relationships*, respectively. The membership of entities in entity sets is determined by a predicate. Hence, each entity may be a member of more than one set. For example, a person could be a student as well as an employee of a given university.

Relationships are associations among entities and are defined as mathematical n-ary relations of the form $R = \{[r_1/e_1, ..., r_n/e_n] \; where \; e_1 \in E_1, ..., e_n \in E_n\}$. The terms r_i represent *roles* played by the corresponding entity e_i in the relationship R. For example, if "marriage" is represented as a relationship between a man and a woman, then the roles "husband" and "wife" may be associated with the two participating entities, "man" and "woman", respectively.

Both entity and relationship sets may have properties called *attributes* associated with them, where each attribute is defined as a mapping between the entity or relationship set and a *value set*. *Multivalued* attributes are permitted in the ER-model.

The ER-model is used primarily as a *database design tool*; the actual database is then implemented with some other data model. The key step in designing a database enterprise is determining the point of view from which the real world is to be modeled. When the ER-model is used, the entity sets and the relationships among them are chosen by the designer. The procedure for this selection cannot be precisely defined — it is a rather subjective process. Once the entity and relationship sets have been selected, the next crucial (and subjective) step is the selection of the relevant attributes. The design process is facilitated by a



PARTS-SUPPLIED is a many-to-many relationship between PARTS and SUPPLIERS. Each SUPPLIER supplies *m* PARTs and each part is SUPPLIED-BY *n* suppliers.

Figure 1

An Entity-Relationship Diagram.

pictorial design tool called *entity-relationship diagrams*, in which rectangular nodes represent entity sets, circular nodes represent attributes and diamond shaped nodes represent relationships. The arcs in an entity-relationship diagram are undirected and may be labeled with appropriate role names. The numbers on these arcs indicate the possible relationship type, that is, 1-to-1, 1-to-many, or many-to-many. Figure 1 is an illustrative example of a simple ER-diagram.

In summary, the ER-model is a significant improvement over the classical database models. In particular, the explicit distinction between entities and relationships and the introduction of distinct types of arcs (i.e. role arcs) has significantly increased its expressive power. Unfortunately, it is not always easy to categorize things as either entities or relationships. This difficulty can be illustrated with the concept of *marriage* which, from one point of view, can be categorized as a relationship between a man and a woman, and from another, as a legal entity, in which the man and the woman participate as attributes. The ER-model does not allow any specific "thing" to be both an entity and a relationship. Hence,

a decision in favor of one or the other must be made by the database designer. Furthermore, the model does not allow for the fact that some information cannot be categorized as either an entity or a relationship. A marriage, for example, may be interpreted as an *event* or a *contract*, with which a certain time interval is to be associated. To resolve such problems, models with considerably greater expressive power must be introduced.

The Hierarchical Semantic Model

Numerous attempts have been made recently to extend the usefulness and the expressive power of the relational model. Most of these have their roots in the work of D. Smith and J. Smith, who introduced two important concepts to database modeling: *aggregation* and *generalization* [SMITH77]. The first of these permits a relationship between data elements to be viewed as a single aggregate object; at the same time, properties of the individual records may be ignored, implying that aggregation is a form of abstraction. For example, an "employee" could be viewed as the aggregation of lower level data, such as "name", "address", "salary", "dependents", et cetera.

The second concept — generalization — is aimed at modeling a hierarchical ordering of information constituting an enterprise. It is an abstraction which permits a class of data to be viewed as a typical (generic) object of that class. It allows attributes with a common value for all members of the class to be recorded with the generic object, rather than being replicated many times at lower levels. For example, the fact that all secretaries have typing skills may be kept with the generic object "secretary" and inherited by each individual belonging to that class. The generalization concept, however, does not distinguish between attributes which are inheritable by individuals and those which apply to the set as a whole. Thus, the model would, for example, permit the recording of the average age of all employees as a value attached to the generic object "employee" even though it

applies only to the set as a whole; i.e., it may not be inherited by any individual in that set.

The aggregation and generalization abstractions form the basis for what became known as Smith and Smith's *Hierarchical Semantic Model* (HS-model). This system comprises a methodology for database modeling. Schemas are built by stepwise decomposition of initial entities into smaller components along the aggregation and generalization hierarchies. A set of semantic and syntactic rules is provided which guarantee that the decomposition process yields a collection of valid relations in Codd's relational model. Thus the HS-model may be viewed as a significant extension of the classical relational model toward a more accurate and more powerful modeling tool for database applications.

The Tasmania Relational Model

Codd [CODD79] describes another extension of the relational data model, named after the conference site at which it was originally presented. Like the HS-model, the objective is to capture more meaning in data to facilitate the process of database design and to permit the system itself to respond in a more intelligent manner. It encompasses many forms of abstraction (including aggregation and generalization); the approach, however, is much more theoretically oriented than in the HS-Model.

As a first step, two kinds of data semantics are identified: *atomic* semantics, representing the basic building blocks of the model, and *molecular* semantics which permit clusters of atoms, constituting meaningful units of information, to be formed. Any n-ary relation is interpreted as an atomic fact. These may be combined according to the following four rules of the molecular semantics:

1. *Cartesian aggregation* is the type of aggregation described by Smith and Smith.
2. *Generalization* (also defined by Smith and Smith).

3. *Cover aggregation* extensionally describes a subset of entities — a convoy of ships is often used as an example of this type of aggregation.
4. *Event precedence* — entities of type event have a start time and an end time. For some applications, ordering events is important. This is facilitated by *alternative* and *unconditional successor* and *precedence relations* which, respectively, define what *may* and *must* follow and precede a given event. For example, suppose we have an inventory database that includes two event entities called ORDERS and SHIPMENTS. It is desirable to ensure that only those goods ORDERed are received in SHIPMENTS. An *unconditional precedence* relation would be used to specify that all shipments must be preceded by an order for the goods delivered.

These enhancements allow the relational model to represent situations that can be represented by any semantic data model. Unfortunately, incorporating the additional semantics, together with the corresponding operators, has made this model *extremely* complicated. For this reason, no claim regarding the ease of use is made for the extended relational model. In fact, it is "*... intended primarily for database designers and sophisticated users ...*" [CODD79].

Although the extended relational model is more data structure than user oriented, it does provide four different types of user interface: *tables* that are used for extensional information, a *set-theoretic* interface which can be used to specify searches without including navigational information, an interface based on *inferential predicate logic* for stringwise expression of intensional information, and a *graph-theoretic* interface which provides a pictorial medium to aid in the design and maintenance of the database.

The Semantic Data Model (SDM)

The Semantic Data Model (SDM) [HAMMER81] was developed as an alternative to the classical data models, which were considered to be "*... too low level and*

machine oriented, requiring the users to think in terms of representation rather than in terms of meaning ...". SDM was the first of many semantic database models (see, for example, [HULL87] for a survey of semantic models) and was designed to be a high level user oriented model for database application environments. Contrary to knowledge representation systems in AI, the objective of SDM is not to model the "real" world; rather, it is a model of a database enterprise which is quite different.

An SDM database is a collection of *classes* which represent "relevant abstractions" in a particular application environment. There are two types of class — *base* and *nonbase* classes. The former are defined independently of other classes while nonbase classes are defined in terms of other classes. Classes are logically linked to other classes via *interclass connections* and are composed of entities called *members*. There are five types of entity: the *concrete object* (e.g. Cars, Students), point and duration *events* (actions and activities), *abstractions* (e.g. the generalizations described previously) and *names* which are identifiers for objects and events. Both classes and entities have *attributes*, which may fall into three categories: *member attributes* (in which each member of a class has this property), *class-determined attributes* (in which all members of a class have the same *value* associated with this property) and *class attributes* (which are properties of the class as a whole).

SDM has a structured user interface that provides facilities for three different classes of users: *naive* nonprogrammers, *routine* users and *experienced* programmers. For naive nonprogrammers, SDM provides an interactive query system that can make suggestions, offer advice, and generally help the user to formulate queries. Routine users are those executing predictable, repetitive tasks. This class of user performs the "busy work" associated with database maintenance, i.e. database editing, which involves simple updates requiring only minor changes to the database contents, but not to its structure, and periodic report generation. Finally, for

the experienced programmer, it is suggested that the SDM formalism *could be* integrated into a conventional general purpose programming language (like COBOL) — a non-trivial, and for the most part unexplored area.

When comparing the higher-level models presented above it is clear that all of them present methodologies for supporting more designer oriented database environments. The ER-model presents a unifying view of data, which permits the designer to organize information in terms of his own perception of the enterprise, rather than in response to artificial constraints imposed by a hardware/data model architecture. While the HS-model and the Tasmania relational model are extensions of the classical relational data model, the ER-model is meant to aid in the design of databases that will be implemented with a classical data model (usually relational). Finally, SDM provides higher level modeling constructs which aid the designer and, with its built-in structured user interface, is more user oriented than either of the other models. SDM is, therefore, both an alternative and an improvement to these models. Although all these approaches fulfill many of the requirements and expectations of information management systems, the desire to include more flexibility (by explicitly supporting mechanisms for retrieving inferred information, derived data, and multiple user views) still persists. The higher-level models described above are representative of research done by database researchers on "semantic data models"; however, recently, database enhancements have come from adapting ideas from other areas as well.

Learning From Other Areas

There has been a shift in database research away from the traditional record-oriented data model towards models which support design oriented semantic constructs. In this context, database researchers have begun to recognize the value of research in both artificial intelligence (particularly in knowledge representation)

and programming languages (particularly in the areas of data abstraction and object-orientation) [BRODIE80, REITER83, KING83, BRODIE84, BIC86, BRODIE87, MYLOP88].

Traditionally, there have been a number of significant differences between the type of information that database researchers are concerned with and the type of information studied in artificial intelligence (AI). In the former, representations tend to be biased toward a large number of instances of a small number of formatted data types. Knowledge representations in AI, on the other hand, are designed to deal with a relatively small number of instances of a much larger variety of types. This implies that knowledge bases tend to be comparatively amorphous while databases are highly structured. In addition, knowledge bases have usually been designed to support a single user while databases have had to provide mechanisms which allow information to be shared. Database management strategies must be able to deal with many users attempting to read and *write* the same piece of information at the same time. Another significant difference is in the amount of implicit information. Knowledge base queries must often use inferential information inherent in the structure of the data to produce a result. In databases, such capabilities either have a very rudimentary form or, more typically, are not present at all. Finally, knowledge bases are usually special-purpose systems, aimed at a particular application, while databases are often constructed to facilitate the needs of a community of users whose requirements may be quite diverse.

One area which is of particular interest in the programming language research from the database point of view is the development of languages which support data abstraction [LISKOV74]. Data abstraction mechanisms are found, to some degree, in SIMULA [DAHL66, DAHL70], CLU [LISKOV77], ADA [LEDGARD81] and later in all object-oriented languages (for example, Smalltalk-80[1] [GOLDBERG83],

[1] Smalltalk-80 is a registered trademark of ParcPlace Systems.

C++ [STROU86] and BETA [KRISTEN87]). Because of its simplicity, object-oriented programming has become widely used in the design and implementation of data intensive systems. Unfortunately, programming environments lack several important capabilities which are essential database applications. Programming languages do not allow users to (easily) share data and they do not provide mechanisms for supporting data (or object) persistence.

Recently, researchers have been looking at the design and implementation of persistent data intensive systems which must deal with information which is less regularly structured than would be found in traditional database applications. These areas include office automation, computer-aided design, computer-aided manufacturing and hypermedia systems. All of these areas require a model which can support complex (perhaps irregular) structures (similar to those which have been prevalent in knowledge representations and object-oriented programming languages) coupled with the capabilities (such as fast secondary storage management and concurrency control) which database management systems provide. For this reason, database researchers have recently begun to apply knowledge representation and programming language techniques to database problems.

The new *conceptual* database models which have emerged are entity rather than record-oriented. The basic building blocks are entities which have fixed properties associated with them and can be created and destroyed for the duration of the application. Information is organized along many dimensions — there may be aggregation, generalization and classification hierarchies, or the information may be partitioned into spaces corresponding to particular user views. Conceptual models are usually built on top of an existing (classical) database management system. Thus, knowledge must be organized, structured and represented so that the translation from conceptual model to data model is easy.

To illustrate how database research has been influenced by other research efforts, conceptual models that have applied one or more techniques developed as knowledge representations (particularly *semantic networks* and/or as aids or alternatives to program development (particularly *object-oriented* and *functional programming*) will be presented. The following discussion is not an exhaustive survey of existing conceptual models; the goal is to demonstrate the approaches which have been influenced this dissertation. These models have been grouped according to the AI/programming technique which was most prominent in its evolution; however, most of the models combine several of these techniques.

Using Semantic Networks

Semantic networks [QUILLIAN68, FINDLER79, BRACH83] were originally developed as a psychological model of the human mind. They have since been used by computer scientists to model knowledge in various intelligent systems. Semantic networks are a knowledge representation formalism that have *labeled nodes* and *labeled arcs.* Nodes usually represent *entities*, *concepts* or *situations* in the domain being modeled while arcs represent *relationships* between nodes.

Mylopoulos et al [MYLOP80] developed a system called TAXIS. TAXIS combines semantic networks with the SIMULA-67 programming language [DAHL66, DAHL70]. It is characterized as "*...a language for the design of interactive information systems...*". A database is designed using the semantic network formalism, then translated into a relational database schema. The latter is extended to include *classes* (of entity) and a *generalization relationship*, which can be used to implement an IS-A *hierarchy.*

Following the principles of data abstraction, TAXIS uses appropriate procedures to integrate the database. In addition to exploiting knowledge representation techniques from AI, it combines ideas from programming language research with the basic principles of the relational data model. This cross-fertilization process

simplifies use of the system. Applications are described at a higher level than the underlying relational database and the application description can itself be manipulated by programming language commands.

Using Functions

The fundamental concepts in the functional data model are *entities* and *functions* which represent conceptual objects and their properties. It was first introduced by Sibley and Kershberg [SIBLEY77] for modeling data structures representable in the classical data models. Functions describe both entity types and properties of an entity. They map a given entity into a *set* of entities. From Sibley and Kershberg's foundation Buneman and Frankel [BUNEMAN82] developed a functional notation for data description based on Backus' functional programming (FP) notation [BACKUS78]. Unfortunately, FP notation is not suitable as a user interface language. In the DAPLEX language [SHIPMAN81] the functional data model is expanded; facilities for defining (limited) *user views* using *derived functions* are introduced. A DAPLEX schema forms a semantic network.

An entity may be associated with several types so that the particular function or functions applicable to an entity at an given time may depend on its *role*[2]. For example, an individual might be both a "student" and an "instructor". Both "students" and "instructors" have a function called *courses* associated with them. Courses(Instructor) returns the set of classes *taught by* a particular instructor while Courses(Student) returns the set of classes that the student is *enrolled in.* For the individual for whom both "courses" are defined, deciding which is applicable is determined by looking at the functions' *internal name* which depends on the *role* that the entity is fulfilling at the time.

[2] This term is used by Shipman in much the same way as "type" is used in other data models.

Using An Object-Oriented Approach

The object-oriented paradigm has become increasingly popular in database, programming language and artificial intelligence research. Many object-oriented systems have been proposed and there have been significant differences in the features which have been supported. An excellent description of these variations can be found in [STEFIK86]. Only the basics will be discussed here.

In object-oriented systems, all conceptual entities are *objects.* Objects *encapsulate* a *private memory* (its state) and *methods* (its behavior). An object responds to *messages* (sent by other objects) by executing a method. Methods may retrieve or change information about the object's state and/or cause messages to be spawned which are sent to itself or other objects. Similar objects are grouped together into *classes* — each object is said to be an *instance* of one or more class. Classes are arranged in an IS-A hierarchy which is either a *tree* (like in Smalltalk [GOLDBERG81]) or a *lattice* (found in CommonLoops [BOBROW85]). The basic object-oriented paradigm evolved as a model for program development. Recently, much database research has focused on adding persistence and sharability to object-oriented applications.

The ORION data model [BANERJEE87A] is a prototype database system under development at the Microelectronics and Computer Technology Corporation (MCC). The goal was to provide a persistent back end for non-traditional data-intensive multimedia applications. ORION includes mechanisms for supporting schema evolution [BANERJEE87B] (dynamic changes to class definitions and the hierarchy), composite objects [KIM89] and versions (variations of the same object).

Servio Logic's GemStone[3] [MAIER85] is the only commercially available multi-user object-oriented database system. Much of the original effort was centered around making Smalltalk into a database system [COPE84]. A new language, called

[3] GemStone is a registered trademark of Servio Logic Development Corporation.

OPAL[4], was developed for describing GemStone applications. OPAL was derived from Smalltalk and provides interfaces between the GemStone database and several high-level general purpose programming languages [PURDY87].

Supporting Relativism

The support of *multiple user views* has long been a topic of interest in the database community. Unfortunately, the term, user view, is used to describe many different kinds of database mapping [KLUG78]. User views may be considered to be mappings between various levels of abstraction, like in the DBTG model [KLUG77, TSICH78]. They may be viewed as a protection mechanism [CHAM75, ROWE79] or as a mapping between different data models or languages. Lastly, the term *view* (or database) *intergration* [MOTRO83, DAYAL84, SCHREFL88] is used to describe the problem of merging existing databases into a single *super* database. While these are all valid perspectives none are equivalent to the definition of *relativism*. Relativism is the philosophy that all truth is relative to the individual and the time and place in which he acts. In a database environment this means supporting multiple, perhaps conflicting, perceptions of the *organization* and *information content* of the data.

Relativism in Database Models

In order to support this variety of user expectations, an integrated database must support multiple user views of the data. Most research concerned with user views in the classical data models focused on the relational model, although many of the ideas can be found in (or applied to) the HS-model, the Tasmania relational model, SDM and the functional model. Relational views are usually simple variations of base relations [CHAM75, ROWE79, ABIDA81, WONG83] such as renaming or permuting columns, converting units or representation of a column, selecting a subset which satisfies some predicate, projecting out some columns

[4] OPAL is a registered trademark of Servio Logic Development Corporation.

of a relation, and linking relations together into joins. In particular, virtual relations, representing a type of derived data, are defined in terms of existing relations. A user view of the database is then the collection of base and virtual relations. Virtual relations may be defined using data abstraction techniques [ROWE79], which hide the original underlying base relations. This means that a user cannot, in general, tell which relations are base and which are virtual. Internally, however, just the virtual schema is stored with the database and *not* the relation itself. One of the major problems with defining views in this way is the difficulty of performing update operations via a user view. It is not always possible to translate an update request on derived relations onto an *unambiguous* request on base relations. Therefore, in order to maintain data integrity, it may not be permissible to perform unrestricted modifications on derived relations; Rowe and Shoens [ROWE79] describe several strategies for dealing with this problem. The facility for defining and maintaining user views is often embedded in the language of the user interface. In this case, the interface must automatically handle the translation of a query from virtual to base relations and then translate the result back to the form expected by the user.

In addition to facilitating user views, virtual relations have several other potential uses. They can be employed, for example, to predefine certain information retrieval operations which are complicated to specify yet frequently executed. Virtual relations may also be used to produce *snapshots* [ABIDA81], i.e. materializations of a view at some point in time. Snapshot relations are *read-only* relations and can be useful when applications require access to earlier versions of the database. In a distributed environment, local snapshots may be used to *approximate* remote data, thus avoiding the expensive maintenance of a local replica.

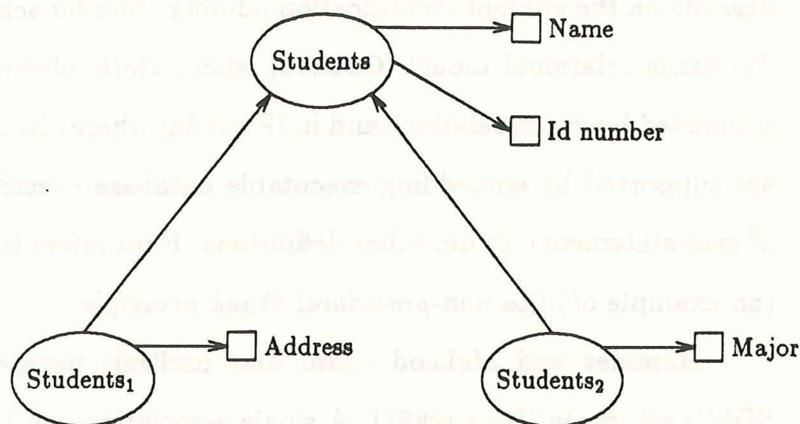


Figure 2

Integration by Alternative Generalization

The functional data model has a richer environment for supporting relativism. In Shipman's DAPLEX [SHIPMAN81], derived data are supported by *derived function definitions* — new properties which are based on the values of other properties. Simple renaming and projection operations are supported and set operations, like union and intersection are used to create new types. Updating is made possible by *explicitly* defined updating procedures.

In a more recent article [DAYAL84], Dayal and Hwang are concerned with the problem of database integration. They use the functional data model as the common model for unification and extend DAPLEX to include a *generalization* or IS-A hierarchy. The major difference between this IS-A hierarchy and most others is that it is possible to create a unified view of a single entity which has different properties in different databases. For example, suppose STUDENTS in DB_1 have a name, student identification number and an address, and in DB_2 STUDENTS have a name, student identification number and a major; figure 2 shows the unified view of STUDENTS. Note that $STUDENTS_1$ and $STUDENTS_2$ represent alternative definitions of the student type. This structure does not imply anything about the structure or organization of the data. It simply states that the STUDENTS record

depends on the student identification number. Similar schemes can be found in the Tasmania relational model [CODD79] where *static* alternative generalizations are supported by graph relations and in [FLINT84] where alternative entity descriptions are supported by embedding executable database description derivations (a type of case statement) within other definitions. Flint refers to this kind of construct as (an example of) the *non-procedural thunk principle.*

Hammer and McLeod claim that multiple perspectives are supported by SDM's schemata [HAMMER81]. A single association can be interpreted by different users in different ways. While this does seem to capture the spirit of relativism, it is up to individual users to interpret a single schema. The schema is sufficiently general to allow for multiple interpretations but it does not directly support different views.

Most database models do not support multiple user views; those that do provide very rudimentary customization facilities. Typically, these are limited to making certain portions of the database invisible to a given user and, in more sophisticated systems, to permit new derived objects and relationships to be included. An ideal system would, in addition, permit the same fragment of information to be represented in different ways depending on a particular user's point of view. Such versatility requires a polymorphic model, which goes beyond the capability of present-day systems. Some advocates of the relational model (and SDM) argue that the versatility of the relation *allows* the user to view an object in any way desired. While this is partially true, the problem of supporting true relativism is not solved. A model using relations (or any single schema) to model a database environment is *too amorphous*; it captures little of the application's semantics and, therefore, offers little guidance for interpreting the data.

In [TANAKA88], a degree of semantic relativism is added to an object-oriented model. Virtual classes are represented by objects which contain predicates on

properties of base classes. They are, therefore, equivalent to simple predefined queries on relational schemas. Virtual schemata are implemented *separately* from the global schema; consequently, the database is *not* represented by a single polymorphic schema.

AI provides a number of techniques for supporting different points of view, as well as contexts and beliefs. Among these are modal logic (where beliefs are treated as static objects), frame based representations and partitioned semantic networks. The latter, because of their similarity to user views, show a strong potential for use in the realm of database modeling. Although these techniques seem promising as enhancements to database modeling, they do not help solve the view update problem [ROWE79]. AI research has traditionally concentrated on designing knowledge representations which, for example, facilitate natural language processing. In this kind of environment, new data may be added to the knowledge base but old data is not removed. Because data and contextual information usually share the same data structures, new information may be added to a local context without having any global side effects.

Partitioned semantic networks, or K-Nets, are a knowledge representation scheme developed by Fikes and Hendrix [FIKES77, HENDRIX79] to enhance the expressive power of conventional semantic networks. K-Nets incorporate the capabilities of first order predicate calculus, facilitate linkage to external procedural knowledge, and, most significantly, provide a mechanism that allows subnetworks to be grouped together and referred to as single objects. This grouping of subnetworks is accomplished by introducing the concepts of *spaces* and *vistas*. All nodes and arcs of a K-Net are *elements* of at least one space (also called a partition). Because each space can be referred to as a *single* unit, it is possible to specify relationships *between* spaces. *Vistas* are lists of spaces; they are intended to give users a manageable perspective of the information. Any access to the knowledge

base is performed through one (or more) of these vistas. The similarity between the concepts of vistas in partitioned semantic networks and the idea of customized user views is obvious. By applying this powerful modeling technique to the database interface, significant gains towards the support of individualized user environments should be realized.

In *PolyView*, a truly polymorphic representation is supported. A *PolyView* application is capable of presenting a different structure for each object (and the application itself) to each user or group of users. This dissertation describes the *PolyView* model. It presents an asynchronous message-driven data retrieval and manipulation strategy and shows how true ***structural polymorphism*** can be supported in this environment.

CHAPTER 3

The PolyView Data Model

The main purpose of this research is to extend the semantics of an object-oriented data model in several ways. Firstly, the message passing paradigm naturally lends itself to asynchronous parallel processing [DENNIS73, ARVIND78, GILBERT88]. A strategy and algorithms will be presented which allow queries to be processed in a concurrent way. A second extension is necessary in order to support *relativism* in database schemas and individual objects. Thirdly, it will be shown that the object-oriented paradigm can be extended in order to incorporate semantic groupings (see, for example, [CODD79, HAMMER81, GILBERT88, RUNDENST89]). Semantic groupings are mechanisms for organizing data in meaningful ways, for example, Cartesian aggregation [SMITH77], is an abstraction which allows the construction of compound (aggregate) entities from other entities in the database environment. Most data modeling systems have evolved, over a period of many years, in an intuitive and random fashion. Models were often proposed, criticized and enhanced and then re-criticized and re-enhanced and so on. No two models have the same features; in other words, there is not a definitive set of features which characterize a "good" data model.

The basic *PolyView* model is not **the** solution to all data modeling problems; it provides a solid yet flexible foundation upon which many database applications can be constructed in a fairly straightforward manner. These applications can be characterized by: static properties (database objects) and dynamic properties (operations on objects). In this chapter the fundamental features of the *PolyView* data model are presented.

The Basics

Philosophy

In *PolyView* we adopt the philosophy found in many sophisticated data models (see, for example, [CODD79, HAMMER81, FLINT84, MAIER85, ABITEBOUL87, BANERJEE87A, PURDY87]): that compound (or molecular) objects are recursively constructed from other simpler database objects (the simplest (atomic) objects are valued-based). *PolyView* supports two basic kinds of association among objects: the IS-A relationship and the ATTRIBUTE relationship. The first is used to construct an *inheritance* lattice (see, for example, [DAYAL84]) while the second is the functional "glue" that binds together molecular structures.

Components

Since there are no standards for the describing concepts and constructs which are included in a semantic database model, it is necessary to identify and define the terms used for that purpose in this dissertation. *PolyView* (classified as a *semantic object-oriented data model*) is a semantic extension of the functional data model [SIBLEY77, BUNEMAN79, SHIPMAN81, DAYAL84]. All conceptual entities in a *PolyView* application domain are represented by *independent* and *persistent objects*. These objects actively participate in query and update processing by sending and receiving *messages* and changing their *local* internal state. Each object has a unique, time invariant, identity which remains with it until it ceases to be part of any application (i.e. until it is deleted).

The Class Lattice

Similar (*instance*) objects are organized into *classes* (which are also objects) which represent both a generic object (or *type* description) and a *set* of similar entities. Classes are arranged in one or more (IS-A) lattice such that class objects near the top of a lattice contain a more general type description than classes

lower in the lattice. This allows common *instance* attributes (like color, size and shape) to be *inherited* by all descendents of a class and *class* attributes (like set cardinality and mean) to be inherited by subclasses. It also guarantees that the IS-A relationship will hold between a class and all of its descendents. Instances are said to *belong to* a class if there is a path containing only IS-A arcs between the instance and the class. Finally, we say that an instance is *owned by* a class if it is directly connected to that class by a single IS-A arc. An instance object is owned by *exactly one* base class.

Derived class objects (also called groupings) contain rules which modify queries so that they can be applied to other (*base*) classes when the derived class is queried. These classes do not have any *direct* instances; they share instance objects with the base classes from which they are derived.

Figure 3 shows part of the *PolyView* class hierarchy. The most basic classes (which are shown as shaded nodes)are present in all *PolyView* hierarchies, a complete hierarchy would depend on a particular application. Two kinds of node and two kinds of arc are shown in figure 3. Boxes represent molecular decomposable objects while ellipses represent atomic or non-decomposable valued-based objects and all arcs represent IS-A relationships.

Notice that *PolyView's* open object-oriented architecture allows new object types to be inserted at *any level* in the IS-A hierarchy making the model highly structured *and* extremely flexible.

The last significant feature of the *PolyView* IS-A hierarchy is its support of *multiple inheritance. Classes* are arranged in a lattice and inherit attributes from all parents. In most systems this can cause problems because there may be conflicts among the attribute definitions of various parents. To remedy this the concept of *identity* [KHOSH86] has been extended to include attributes as well as objects. Attribute identities are system generated, time invariant and unique. This allows

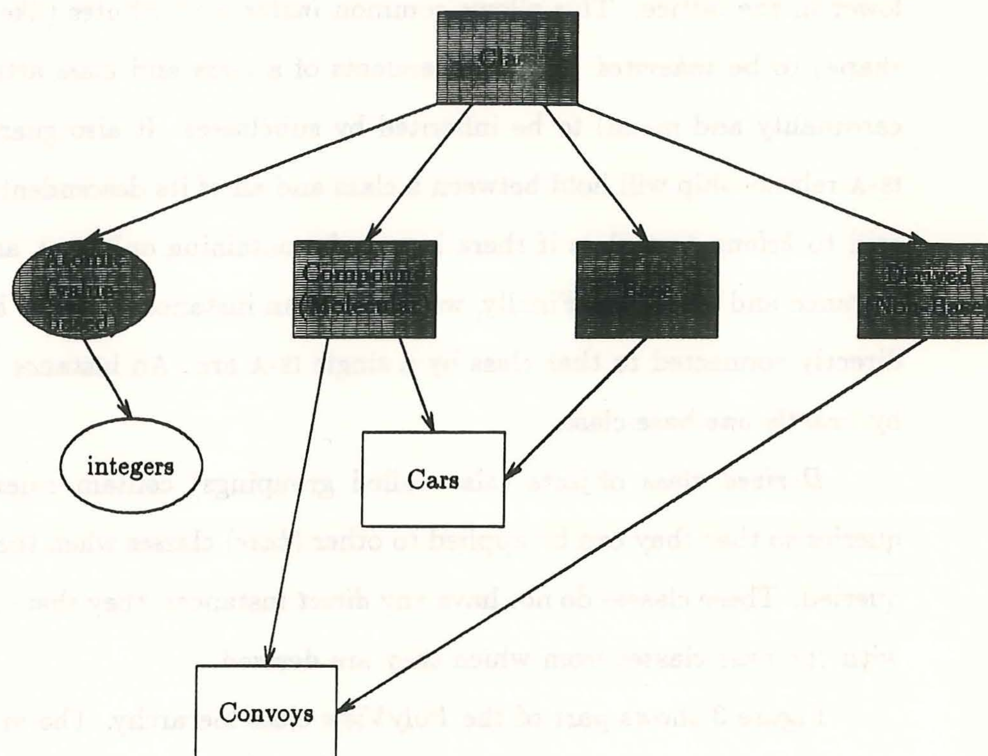


Figure 3

The Basic PolyView Class Hierarchy

name conflicts to be resolved easily and causes no additional problems with query processing because messages always refer to the internal names (identities) which are guaranteed to be unique. There are two other possible conflicts, both of which are only slightly more complex than name conflicts, which are described below.

The first of these conflicts occurs when two or more parents have a common ancestor. When this happens the intersection of the domain of each common attribute becomes the domain of that attribute within the new class. If the intersection is empty there are two subcases which must be considered: (i) the attribute is not mandatory (*key*) in every instance description then it just "disappears" (it is actually kept but its domain is empty) or (ii) the attribute is *key* then class description is invalid (as a *base* class) and cannot be created.

A second kind of attribute mismatch occurs when the new object inherits the "same" attribute *independently* from two or more sources. In this case *PolyView* creates a pair of *functions* (called idf and $\overline{\text{idf}}$) for each source of the inherited attribute. These functions are used to convert the identity of an inherited attribute to its locally defined equivalent and back to the original. An idf is applied to inherited attributes with multiple origins when they are referenced in a message received from a parent. idf functions take the original attribute identity as an argument and return the new attribute identity and the identity of the parent from which the message was received. $\overline{\text{idf}}$s are applied to these same (multiply inherited) attributes' identities before they are returned to a parent. These functions have two arguments, the local attribute indentity and the parent class identity (both of which were returned by the idf) and return the parent's attribute identity. For example, imagine a taxonomy of modes of transportation which includes both land-based and water-based vehicles. A common subclass of both classes is amphibious vehicles. Suppose both land- and water-based vehicles have a color attribute (which is *not* inherited from a common ancestor); there would then be three separate color attributes after the amphibious vehicles class was added to the environment. Let ⟨l-color⟩, ⟨w-color⟩ and ⟨a-color⟩ be the identities of the color attributes of land, water and amphibious vehicles respectively. Similarly, let ⟨land⟩, ⟨water⟩ and ⟨amphibious⟩ represent the class object identities. The identity conversion function (idf) and its inverse ($\overline{\text{idf}}$) are defined in the amphibious vehicles class by the following:

$$\text{idf}(\langle\text{l-color}\rangle) \rightarrow \langle\text{a-color}\rangle, \langle\text{land}\rangle$$
$$\text{idf}(\langle\text{w-color}\rangle) \rightarrow \langle\text{a-color}\rangle, \langle\text{water}\rangle$$
$$\overline{\text{idf}}(\langle\text{a-color}\rangle, \langle\text{land}\rangle) \rightarrow \langle\text{l-color}\rangle$$
$$\overline{\text{idf}}(\langle\text{a-color}\rangle, \langle\text{water}\rangle) \rightarrow \langle\text{w-color}\rangle.$$

Attributes

Attributes capture other important ways in which objects may be interrelated. An attribute may represent the fact that an object has a number of component parts (which may include other objects — for instance, an engine is part of a car). Attributes also represent more symmetrical relationships. For example, the fact that a man and a woman are married can be represented by a bidirectional spouse attribute. Both these relationships are captured by a single concept — the ATTRIBUTE relationship.

In the *PolyView* model there are two distinct types of attribute called *class attributes* which are properties of an entire class of objects (and do not apply to individual instances) and *instance attributes* which are properties of individual instance objects. Both are further subdivided into three subtypes called *compound*, *local* and *method* attributes. Compound attributes are references to complex structures. This means that in order to retrieve (or change) information from compound attributes, it is necessary to send messages to other objects in the database. Local attributes, on the other hand, represent simple locally stored objects which may be accessed directly. Finally, methods (a form of *derived* data) may be localized or distributed but the information which they "retrieve" is *calculated* or *constructed* rather than retrieved from the database. There is one more special type of instance attribute called a *category attribute.* Each category attribute is a special system maintained Boolean value which determines whether an instance belongs to a particular category.[5]

The *six* basic classifications of attribute have other properties associated with them. They may be mandatory (*key*) or not (*non-key*), *single* or *multi-valued*, and *changeable* or *not-changeable.* Note that category attributes are always single-valued and changeable. Finally, all user controlled attributes have inverses. This

[5] A category is a particular kind of derived class which is described later in this chapter.

means that attributes can be thought of as bidirectional arcs which connect objects in a *PolyView* schema.

Organization of the Database

Some useful definitions:

- A *schema* is a collection of classes and the attribute associations among them.
- *Data* is the collection of *instance objects* which are associated with the schema.
- A *view* is a schema and its associated data.
- A *database* is the combination of all of the views.

A database usually includes several views. An important relationship between a database and its views can be characterized by the following observation: the set of classes in any view is a subset of the set of classes contained in the database. Within a given database environment a list of current views (and classes) in kept in a *global symbol table.*

The Global Symbol Table

In order to allow each user to access the database through his own view, *PolyView* maintains a two-tiered global symbol table structure. Appendix 1 includes a description of the global symbol table's data structures. It is composed of two kinds of tables. There is a single *global symbol table* and one or more *view model object tables.* Although there are usually several view model object tables, there is exactly one for each view. Because users may share a view, there may

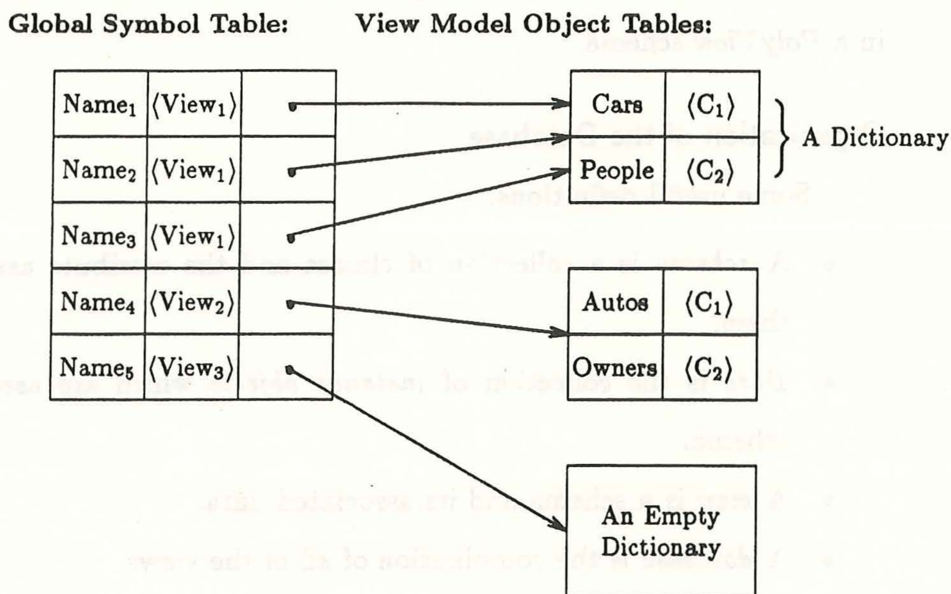


Figure 4

The Global Symbol Table Structure

including View Model Object Tables

be several entries in the global symbol table which point to the same view model object table.

Figure 4 shows a global symbol table and its associated view model object tables. This particular table contains three views. The first column of the global symbol table contains identifiers which are used by users to specify their view of the database. The second and third columns contain (internal) system maintained view identities and pointers to view model object tables respectively. Each view model object is associated with a single view. It's first column contains a view sensitive external name which is associated with the internal name of a class in the database schema. In figure 4, $\langle View_1 \rangle$ and $\langle View_2 \rangle$ are fully instantiated user views. Their global symbol table entries both point to particular view model objects. $\langle View_3 \rangle$, on the other hand, is in the process of being created — its symbol table entry

currently refers to a default view model object table. Notice that this generic model object contains a "place marker" for a structure called a *dictionary.* This concept is not new; it is borrowed from Smalltalk-80 [GOLDBERG83]. A dictionary is a table which allows *associative* (key word) lookup. Further note that the global symbol table is also a dictionary and the external view names are the associative keys to its contents.

Schemas

Associated with each user (or group of users) there is a high level description of the world of the database application. This description is called a *schema.* It presents the entire user's view of the database as a single connected graph. Even though instances are not included in the global schema, it is usually quite large; therefore, the global schema is usually visualized as several graphs rather than a single one.

Many objects and arcs in a schema are not *base* but *derived* (which will be shown as dashed boxes and arrows in all subsequent figures). Base objects have a concrete representation *stored* in the database while derived (or virtual) data (described in more detail later) are calculated by executing a method when they are accessed by a user. In the day to day interactions with the database, the only visible difference between virtual and stored data is that virtual data cannot be directly updated.

A Database

A database is a network of nodes and directed edges. Each node represents an independent database object (for example, people, colors, automobiles, or engines) and arcs represent various associations among them. Figure 5 (which shows a single branch of a "modes-of-transportation" hierarchy) is used to further illustrate

various aspects of the *PolyView* model. Notice that the modes of transportation hierarchy is part of an application specific hierarchy.

The graphic representation shows the two distinct types of nodes: ellipses are non-decomposable *atomic* objects and boxes which represent compound *molecular* objects. The IS-A hierarchy (which is composed of nodes and *unnamed* arcs) facilitates *inheritance* attributes from classes to their descendants. The IS-A arcs' arrows show the direction in which inheritance takes place. A view designer may *choose* how information is displayed. He may decide that those attributes which he perceives as parts of an object should be displayed inside the larger object's node while the more symmetrical relationships are displayed outside of the aggregate object. For example, in figure 5 the attributes *color* and *engine* are "parts of" an automobile while *owner* is a relationship between an automobile and a person. Notice that these choices and many other choices related to the users' perceptions of the data are subjective. Although meaningful to the user, whether an attribute is displayed inside or outside of an object is irrelevant to the semantics of the database itself. Other classifications of attributes are not subjective, for example, there are some *key* attributes which are absolutely essential to the description of an object. All other attributes are *ordinary* (or non-key) *attributes* (which may or may not be instantiated in all instances). For example, in figure 5 the owner attribute (from cars to people) is key because (in this very simple world) all cars must be owned by people. However, the inverse of this relationship is *not* key because some, not all people own cars.

Derived Data and Groupings

Much of the semantic richness of this model comes from its support of a variety of derived data. There are two major categories of derived data: *classes* and *attributes*, which are represented by rules that are included in class descriptions.

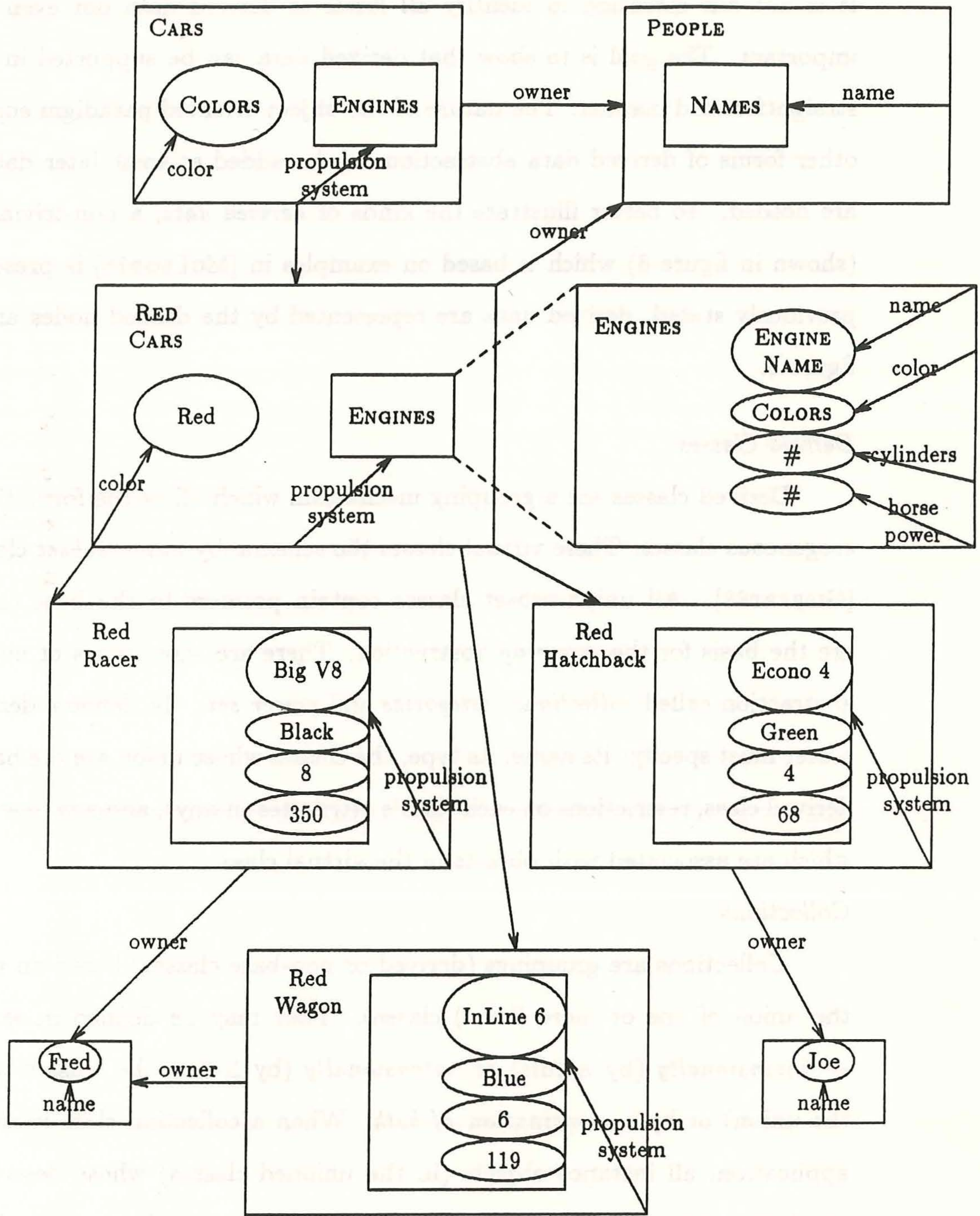


Figure 5

Single Branch "Modes-of-Transportation"

It is *not* our intention to identify *all* kinds of derived data nor even *the* most important. The goal is to show that derived data can be supported in a simple, straightforward manner. The nature of the object-oriented paradigm ensures that other forms of derived data abstraction can be added at some later date, if they are needed. To better illustrate the kinds of *derived data*, a non-trivial example (shown in figure 6) which is based on examples in [McLeod78] is presented. As previously stated, derived data are represented by the dashed nodes and arcs in figure 6.

Derived Classes

Derived classes are a grouping mechanism which allow the formation of heterogeneous classes. These virtual classes the schema, by *union-subset* class objects [Gilbert88]. All union-subset classes contain pointers to the base classes that are the basis for the *grouping* abstraction. There are three types of union-subset abstraction called *collections*, *categories* and *power sets*. To define a derived class, a user must specify: its name, its type, the classes whose union are the basis for the derived class, restrictions on each class's attributes (if any), and any new attributes which are associated with objects in the virtual class.

Collections

Collections are groupings (derived or non-base classes) based on a subset of the union of one or more (base) classes. They may be defined in several ways — *intensionally* (by a rule) or *extensionally* (by listing the classes which form the union) or by a *combination of both*. When a collection class is added to an application, all instance objects (in the unioned classes) whose descriptions are consistent the collection's rule are automatically "inserted" into the collection. In figure 6 Oil Tankers is a collection whose members are all military and merchant ships whose classification is "oil tanker".

Categories

Categories are *user controlled collections.* They are defined in the same way as collections but instance objects are *not* automatically added to categories — users must explicitly insert and delete objects into/from a category. When a category is created, *PolyView* creates a category attribute which is added to all the base classes (and their members) which are part of the category's union-subset class. Initially, the category attributes of all instances are assigned the value "false". When an instance is "inserted" into a category, the corresponding category attribute value is set to "true". Conversely, when an object is removed from a category, the corresponding attribute is reset to "false". The Banned Ships class in figure 6 is an example of a category. Potentially, any ship may be banned but a user must explicitly ban it.

Power Sets

Power sets can be thought of as a generalization of collections and categories. They can be *extensionally-defined*, *intensionally-defined* or *user-controlled.* The major difference between ordinary union-subset classes (collections and categories) and power sets is that a power set is a subset of the *power set of the union* of some base classes instead of their *union.* Each "instance" of a power set is a derived class; that is either a *collection* or a *category.* In figure 6 Convoys are modeled as a power set because each convoy is a set of ships (actually a category) and not a single ship. Notice that the attributes (location and max-speed) are associated with the convoy itself and not the individual ships in that convoy.

A variation on the power set (which we have chosen not to support initially) is called an *aggregate grouping.* Aggregate groupings are based on a "fixed" cover aggregation of the union of predefined classes. This abstraction allows these classes to be treated as instances of the grouping. When this abstraction is introduced, it is also necessary to introduce the "member-of" relationship between classes in the grouping and the set of database instances which they represent. Aggregate

groupings can be extensionally-defined or intensionally-defined. For example, there could be a transformation which changes a base class to a grouping: (ships → ship-types).

Derived Attributes

The method based attributes which were introduced earlier allow users to access derived as opposed to stored data. These *v*irtual *a*ttribute (**VA-**) abstractions are classified by the action taken by the system when it instantiates them. All method based attributes cause a subquery to be substituted for the virtual attribute and spawn a new query which is reprocessed by the object. Some (*VA-arc*) derived attributes cause a virtual arc to be "created" while other (*VA-node*) virtual attributes cause a virtual node to be "created" as well as a virtual arc.

To create a VA method a user provides either a subquery (a similar mechanism is found in POSTGRES [STONEB86]) or a general purpose method. In both cases, the user must specify: the name of the attribute and whether it is evaluated by the *class*) or its *instances.* In addition, for a subquery, the user names the class which will receive the query and a list of attributes and restriction to be applied to them. For a general method, a procedure must be defined.

An example of a VA-node abstraction is *aggregate data.* Aggregate data are defined by aggregate operators which abstract a single object from a class of objects. Examples of aggregate operations include: calculating the maximum speed of a convoy (see figure 6) or determining the average length of an oil tanker (not shown in the figure).

VA-arc abstractions are *inference rules*, so called because the relationship which they make explicit can be inferred from the structure of schema anyway. Information is retrieved by substituting an attribute request subquery for a VA-arc "attribute" which effectively creates a virtual arc. For example, consider the *grandfather* relationship between people. This could be represented explicitly as a

attribute (arc) from an individual to his parents' fathers or it could be represented *implicitly* by including a rule which states: "To find a person's grandfather, first find his parents and then find their fathers" (grandfather(X)=father(parent(X))).

To the user, derived data of all kinds can be used to *retrieve* information in exactly the same way as any base attribute. Figure 7 shows part of the internal structure of the base classes which represent military and merchant ships and the non-base classes which represent convoys, oil tankers and banned ships. The global symbol table containing these same classes is shown in figure 8.

The Object Structure

Several kinds of *PolyView* object have already been identified. They represent both stored (base classes and instances) and derived information (collections, categories and power sets). In this section, their underlying representation will be discussed.

This discussion will focus on class objects because instances have relatively straightforward representations — they are comprised of stored (simple) attribute values and pointers to other object instances. All other information, including generic and specialized message processing methods are inherited from their owner class. In appendix 1 the data structures which represent various *PolyView* objects are shown in detail. Each class object has three major components[6]: the *generic methods* (for retrieving and updating data), the *object description* (which contains a hidden internal representation of the object) and the *view description.* Figure 9 shows a very high level graphic representation of the internals of a class object (more details are given in appendix 1). The overall *structure* of each of these sections remains unchanged over the life time of an object, for example, the various lists of attributes and is-a relationships contained within the object description.

[6] Each instance has a single major component which corresponds to the class' object description

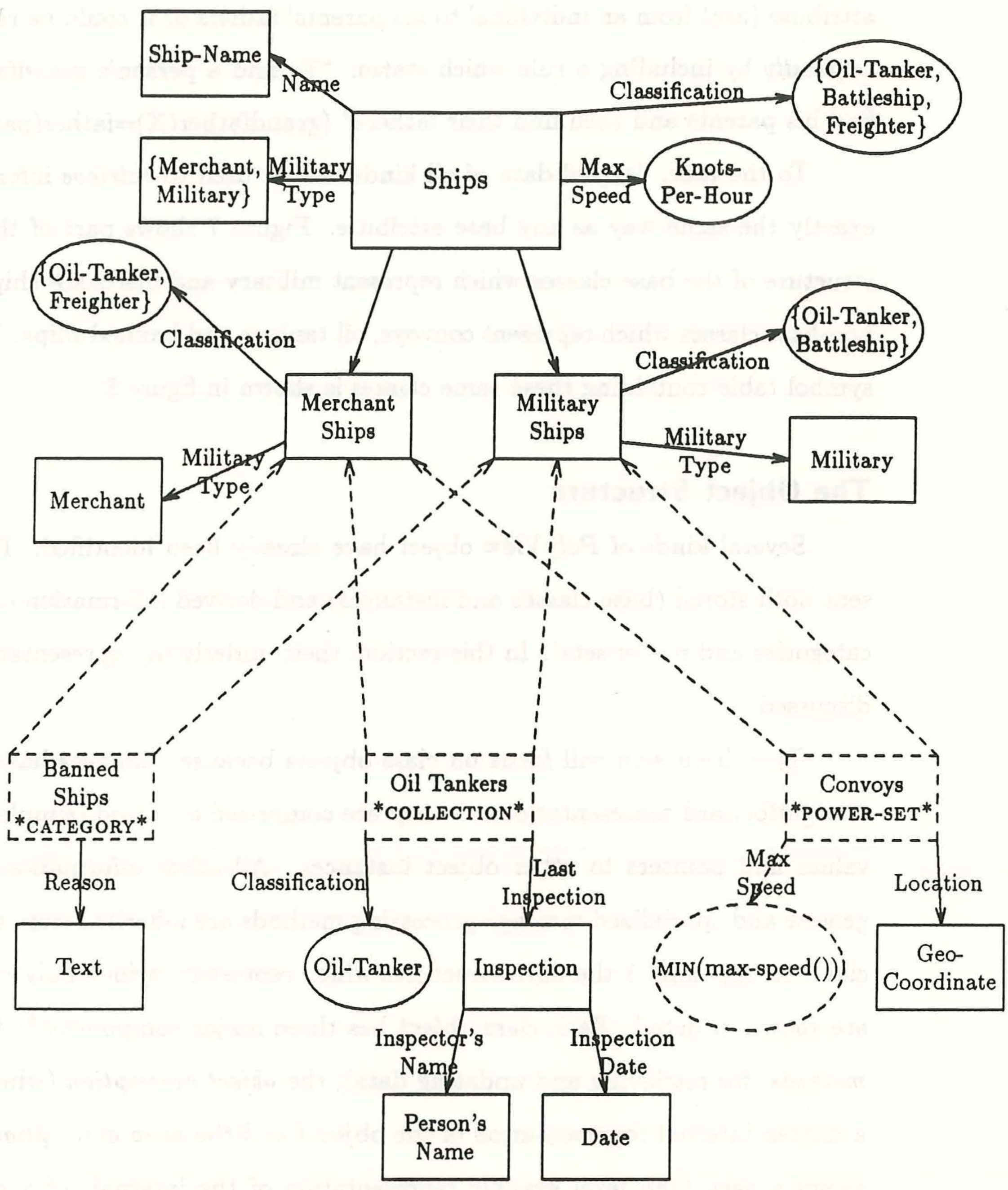


Figure 6

An Example of Derived Data

Attributes and is-a connections may be added and removed from various lists but the lists themselves persist for the lifetime of the object. The view description

```
Object Merchant Ships:  /* for parts of object shown Military Ships class would be identical */
  Object Description:
   Instance Attributes:
     Compound:        < list including < last-inspection > >
       Local:           < list including < reason > >
       Category:         < list including < banned ships > >

Object Banned Ships:
  Object Description:
    Class-type:         Category
    Restrictions:        < banned ships > = 'true'
    Is-A Connections:
      BaseUnion:       < Merchant Ships >, < Military Ships >
    Instance Attributes:
      Local:           < reason >:          < Text >
      Category:         < banned ships >: < Boolean >

Object Tankers:
  Object Description:
    Class-type:         Collection
    Restrictions:       < classification > = 'Tanker'
    Is-A Connections:
      BaseUnion:       < Merchant Ships >, < Military Ships >
    Instance Attributes:
      Compound:        < last-inspection >: < Inspection >

Object Convoys:
  Object Description:
    Is-A Connections:
      BaseUnion:       < Merchant Ships >, < Military Ships >
    Instance Attributes:
      Compound:        < location >:     < Geo-Cord. >
      Method:          < max-speed >:  min(forall x (*in Convoy*) max-speed(x))
```

Figure 7

Derived Classes: Internal Representations

contains a list of *view instance descriptions* (through which various users access the database) which may grow and shrink as views are added and removed from the application environment. However, the view description always contains a global view which acts as an identity mapping to the underlying data structure. New views are always derived from the global view.

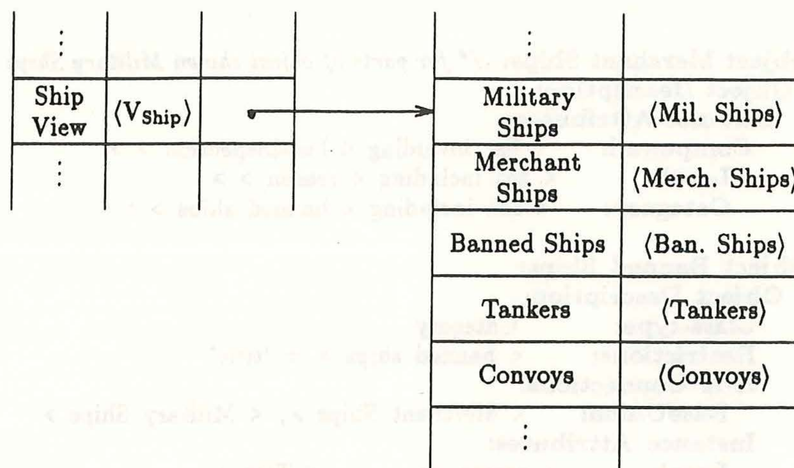


Figure 8

The Global Symbol Table

Including Derived Classes

Generic Methods — Using the Object/View Structure to Answer Queries

The generic methods section of an object contains data manipulation and update procedures. Figure 10 shows *PolyView's* message driven processing strategy. When an object receives a request it must be examined in order to identify the user's point of view. Once the view has been identified the query continues to be processed ***through*** the appropriate ***view description template.*** Local attributes are checked directly while complex (compound and method-based) attributes may cause subqueries to be spawned. Intermediate results are collected and when all the results have been collected the query is either propagated to IS-A related descendents or a result is returned to the source of the query. Note that this is a very high-level sketch of the *PolyView* data retrieval and update mechanism. These operations are discussed in considerable more detail in the next chapter. It is clear, even from such a high-level description, that this basic strategy requires ***no centralized control*** and that messages can be processed in an ***asynchronous message-driven*** manner.

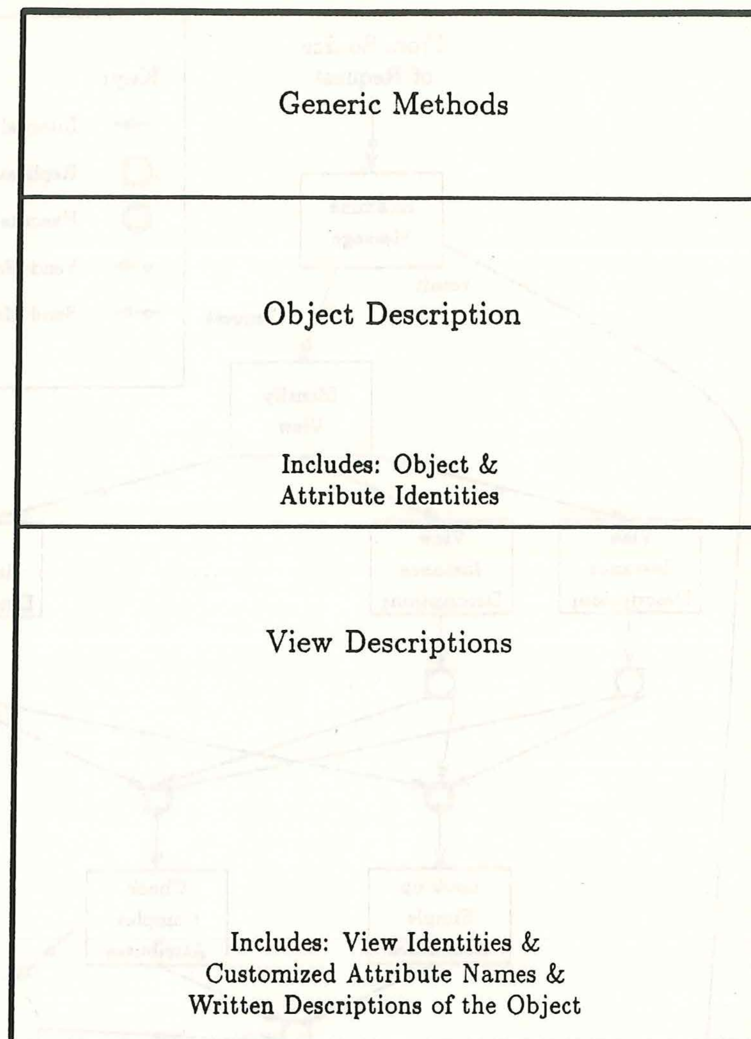


Figure 9

A Class Object

The Object Description

In object-oriented database systems, objects are composed of a private memory, a unique time invariant identity and a public interface. In *PolyView*, these last two features are extended and enhanced. An extended use of identity has already been presented. In *PolyView*, internal names (identities) are assigned to attributes as well as objects.

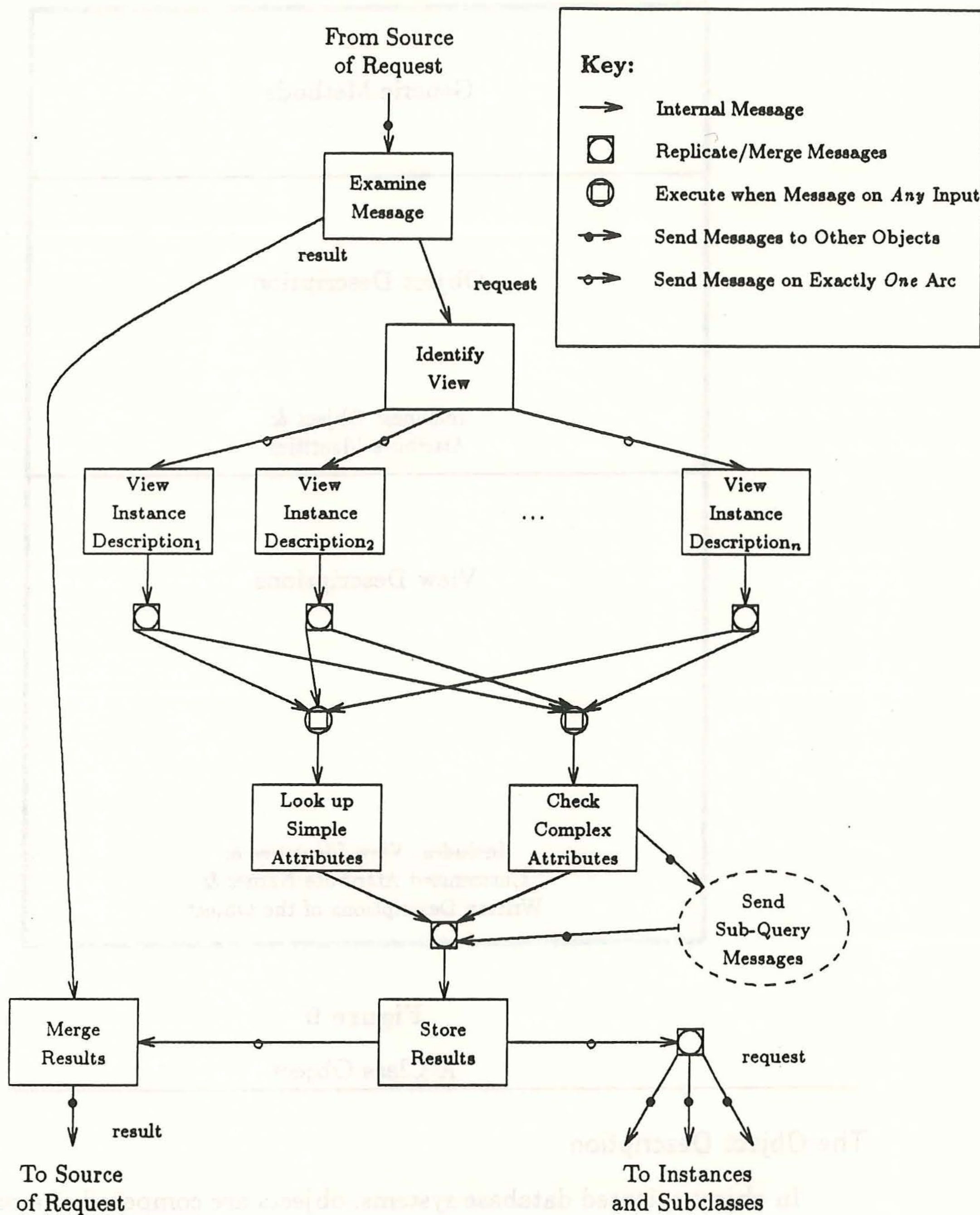


Figure 10

The Generic Message Processing Strategy

Object descriptions are automatically maintained by the system. They are hidden from database users. An object description contains an object's identity and lists of ATTRIBUTE and IS-A relationships (including GROUPINGs).

The object descriptions contains IS-A connection lists which reference all base class parents, descendents and ancestors. Similar object-oriented pointers to non-base classes (collections, categories and power sets) which are related to the class are also found in the class description. In addition to the lists of object-oriented pointers there is also an is-a "list" which contains an index used for accessing instance objects of secondary storage devices.

There are ATTRIBUTE lists for each of the seven attribute subtypes: class (compound, local and method) and instance (compound, local, method and category). Each attribute is an ordered tuple consisting of at least a unique identity and a pointer. The pointer is either an object-oriented pointer (to another independent object) or a reference to the object's local memory. Other information about attributes is also kept with the object description. For example, separate lists are kept for key and non-key attributes plus references to default objects to be associated with the attribute when an new object is created. Finally, each attribute may be declared to be changeable (the default) or not changeable.

The View Description

The view description contains one or more ***view instance descriptions.*** Each view instance description is a "semi-public" user interface into the object structure. The term semi-public is used because each view instance description provides a public interface for a given object for a specific group of users. A view instance description contains several subcomponents including a unique system defined identity for the view (which is shared by all classes which participate in the view) and the class's external name from the current point of view both of which are also found in the view model object table. A view instance description may also

contain a natural language description of the class, its structure and its role in the database enterprise. For example, if an engine object's primary function is to fill the propulsion system role of an automobile then this would be noted in the (written) description.

Each view description also contains lists of attributes which are accessible from the current view. The same attribute may be referred to several times within a single view description. Any attribute which appears in any view instance description must also be part of the object description.

There are two distinct categories of attribute: those which are completely *visible* through the view and those which are *invisible* but to which the system is allowed limited access. Visible attributes each have a unique printable name associated with them within a particular class. Each attribute may have a default (value, object or operation) associated with it. These defaults are used by the system to fill a role when a new object is created and for updating the attribute if the object is changed.

Notice that in order for a view to be *updatable*, all key attributes must be included in the combined list of visible and invisible attributes and *all* the invisible key attributes must have default objects associated with them.

Most views do not encompass all classes in the database. If an object's structure does not include a view instance description which corresponds to a particular user view then that class and its instances simply do not exist in that user's view of the world. There is a special *global* view which is guaranteed to include all classes in the database and all attributes within all of those classes. From this global perspective, *all* attributes are visible *except* the category attributes which are still maintained by the system.

Summary — A Unique Framework for Supporting Relativism

Of *PolyView's* components the two-tiered symbol table, the object structure and the extensions to identity make a unique framework for supporting relativism. At the global level, the global symbol table contains all *classes* (base and derived) which are reachable from a given point of view. *Attribute* reachability is determined locally by the objects themselves. These unique features allow *PolyView* to support many different, perhaps conflicting, points of view and preserve the principle of encapsulation.

CHAPTER 4
Asynchronous Message-Driven Processing

In an object-oriented environment, each object is an *abstract data type* which includes a description of the data it represents and a set of operations (*methods*) for manipulating that data. These methods are triggered by messages received from other methods in this or other objects. The data representation is not visible to the outside world; the user "sees" a "black box" and the protocols (for manipulating the object) in its public interface. In *PolyView*, a similar situation exists except that communication *between* objects is achieved by a small number of *generic* methods.

In this chapter, generic methods for the two basic kinds of query processing will be discussed. Information retrieval operations will be presented first; queries which update the database will be discussed second. All queries access the data through a "semi-public" object interface or user view. Before discussing *PolyView's* concurrent message-passing strategies, the internal structures which support relativism within this paradigm will be presented.

PolyView Structures: Objects, Views and Queries

In the previous chapter the basic object structure was presented. The focus of this chapter is query processing. Queries are sent to objects whose structure (the *object description* and *view description*) is designed to support query processing through views. Before the query processing strategy is discussed in detail, the important features of the query and object structure (in the context of query processing) will be presented.

The Object Description

The object description is part of an object's hidden private storage. All the structures in it are referenced indirectly through the protocol in the user view. This is *PolyView's* way of supporting relativism and guarantees that access to the data is completely controlled by the system.

The View Description

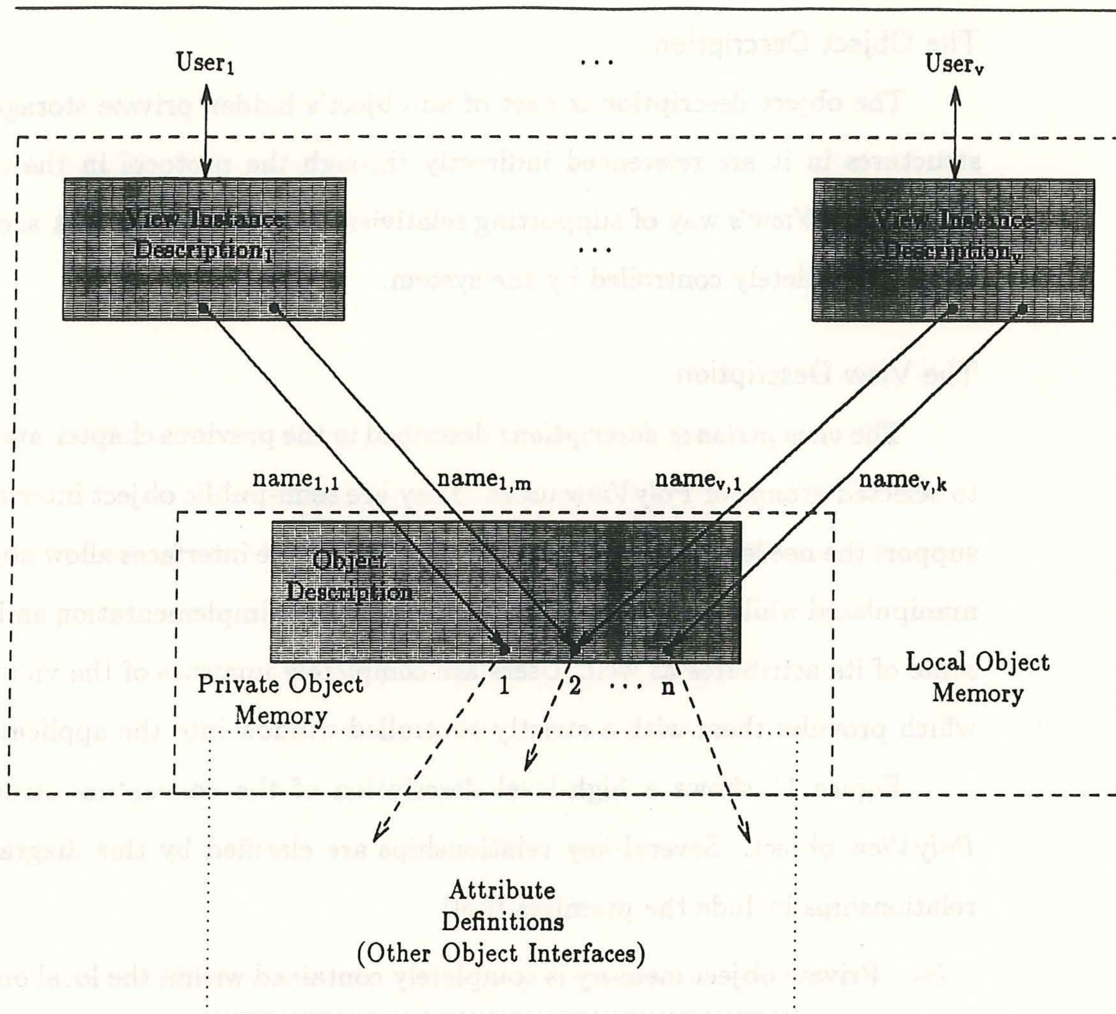
The ***view instance descriptions*** described in the previous chapter ***are*** accessible to selected groups of *PolyView* users. They are semi-public object interfaces which support the needs of a particular group of users. These interfaces allow objects to be manipulated while hiding the internal details of their implementation and, possibly, some of its attributes as well. Users are completely unaware of the view structure which provides them with a strictly controlled window into the application world.

Figure 11 shows a high-level description of the conceptual structure of a *PolyView* object. Several key relationships are clarified by this diagram. These relationships include the premises that:

(i) Private object memory is completely contained within the local object memory.

(ii) Users' access an object through a system controlled interface. (*PolyView* selects the appropriate view instance description for each user.)

(iii) Attributes may be defined locally (within an object's private memory) or outside of the objects local memory (in another object definition). Note that in either case the implementation is hidden from the users.



Key:

1,2, ... n : represent object-oriented pointers

$name_{x,y}$: represents the Yth attribute name in view X

solid single arrows are inside local memory

Figure 11

Object Structure Including

Semi-Public Interfaces (Views)

A Structure for PolyView Query and Update Requests

PolyView requests are messages which are passed between objects in a database application. Objects process messages depending on their content and the point of view of the user who spawned the request. The view is determined by information contained within both the object *and* the message itself.

The query message structure is shown in detail in appendix 1. Each query message is a tuple which contains the following information:

(i) The target (object) of the query.

(ii) The sender of the query (again an object).

(iii) A system generated identity for the query.

(iv) The status of the query.

(v) The user's "signature".

(vi) The type of the request (information retrieval, update, ...).

(v) Information about where to send the reply to the message.

(vi) Restrictions on the query[7] which are a set of paths which describe restrictions on the attributes of objects involved in the query.

The user's unique view identity (an internal signature) is automatically included in any query by the system. This is retrieved from the global symbol table and corresponds to an identity found in some view description instances. By matching a user's signature to a view identity the system determines which object interface it should use. The target of the query must also be identified. This is usually the *class* (either base or derived) which is the starting point for the query. The user identifies the class by name (which must be unique within the view) and *PolyView* uses the global symbol table to find its equivalent class identity. The query restriction includes properties (which are mostly, but not exclusively attributes) and their current status (found, not found ...). Most query restrictions

[7] The format of these query restrictions is comparable to the body of the *is-there?* query in Omega [ATTARDI86]. The processing strategy, however, is not the same.

are *required* restrictions which means that they must be satisfied in order for the query to succeed. However, queries may include other restrictions which determine what may be displayed. Finally, the message contains information which allows the system to determine the type and destination of the reply to the message.

Message-Driven Processing

The object-oriented paradigm with its abstract data types and message passing semantics make *PolyView* suitable for implementation on a highly parallel loosely coupled multiprocessor. An ideal architecture would not need centralized synchronous control or a shared global memory and each (class) object could be mapped onto a different processing element (PE) as long as there were enough physical communication paths for each logical arc. There are many architectures that satisfy this requirement.

To query a *PolyView* application users must specify the type of the query (retrieve, insert, etc.), target of the query (which is a view dependent external name) and the restriction. *PolyView* takes the following actions:

- The system generates a **query-id** and looks up the user's **view-id** (in the global symbol table). Both the view and query identities become part of the internal message structure.
- The class object identity of the **target** is extracted from the model view object table.
- The **sender** is identified as a source outside of the *PolyView* system.
- The **message-type** and **query-retriction** are determined directly from the user query, the **query-status** is initialized to "not found" and the **reply-type** is set to *user*.

Once a *PolyView* message has been constructed, it can be sent to the target object for processing. All responses eventually come back to the system/user interface. This interface transforms the retrieved information into a form which the user can understand and then displays the result for the user. Figure 12 shows a high level graphic representation of this interface. The dashed path (along the top of the figure from User_n to System via View_n) represents the conceptual system/user interface. Conceptually, a user formulates a query against a view which is sent directly to the view object. The receiving object translates the query to the underlying structure, performs the requested action, translates it back to its original form and returns the result. Unfortunately, this is too simplistic because view classes are often "distributed" over several database classes. Requests must, therefore, be processed by traversing the schema and gradually converting them from a user view representation to a system representation. The system/user interface (which is shown as the path from User_n to System via the lower three nodes in the diagram) *uses* part of View_n in *each* visited object to perform the piecemeal translation process from user to system and vice versa.

Information Retrieval

Conceptually, the simplest request for information either points to a set of objects and retrieves the subset of those instances that satisfy some restrictions on their outgoing attributes or retrieves information about some of an object's attributes. Restrictions are recursively decomposed and applied to objects reached via attribute arcs starting at the original object, until the entire restriction is satisfied or fails. In order to describe the query, processing strategy references will be made to the internal structure of *PolyView* queries and objects.

When processing any query, *PolyView* must differentiate between *key* and *non-key* attributes. The reason for this is obvious: If an attribute in a class's object description is key then it definitely exists for *all* instance objects associated

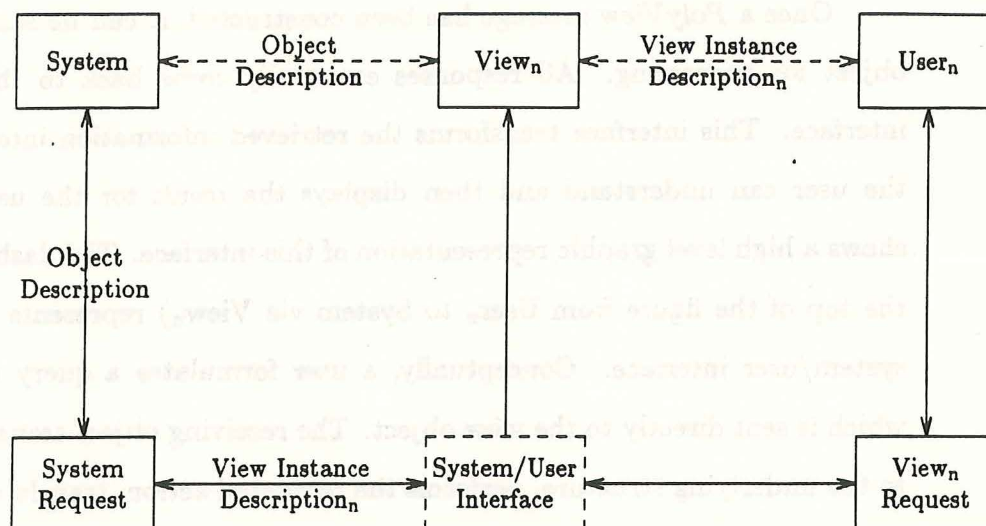


Figure 12

The User Support System Architecture

Showing the User/System Interface

with that class; if it is non-key then it may exist in some of the instances. Notice that this definition of *key* is quite different from a key attribute in many traditional database models since uniqueness is not necessary.

The Attribute-Query-Request, Attribute-Query-Result, Request-From-Outside, Check-Attribute and Test-Type methods are used to process attribute request queries. The Request-From-Outside method is used initially because when a user query is received by any object, it may be necessary to adjust it before *PolyView* can continue to process it (for example, additional restrictions may be added).

A query names an injection point r and lists the attributes (and restrictions on those attributes) which are the focus of the query. When r receives the request, it activates the user view and filters all attribute restrictions (and status values) through that interface. A status value is calculated for all attributes named in the query by sending an attribute request (sub)query message along each of the named arcs. Each attribute object processes it's subquery (through the appropriate interface) independently of all other attribute objects and the strategy is exactly the same as that followed at the injection point. The overall strategy is that the query is *dynamicly* recursively decomposed for parallel processing. Eventually, for each attribute path, a *terminal* node is reached. A terminal node is an object which can determine a status (and a value) for a particular (sub)attribute; it is *not* necessarily a leaf node in the database structure. Once the status is known, it is returned (in an attribute query *result* message) along the arc on which the original request arrived. When a non-terminal node has collected results from all its subqueries, they are used to determine its own status which is then sent back to the sender of the request. Note that because of the distributed structure of the database and the absence of centralized control in this strategy, the subqueries are distributed and the results collected in an asynchronous manner.

There are five possible status values for individual attributes; their most *general* meanings are listed below. Note that all five status values are not necessary for processing the simplest attribute request queries because in this case it is not necessary to differentiate between *key* and *non-key* attributes. All status values are necessary when processing more complex user requests.

1. This attribute was found and (the restrictions on it) satisfied for all possible instances of the set rooted at this node (for key attributes only).

2. This attribute definitely exists for all possible instances, however, the restriction on this attribute may not be satisfied (once again key attributes only).

3. This attribute was found and exists for some instances of the rooted set (for non-key attributes only).

4. This attribute was not found.

5. This attribute was found and is definitely not satisfiable for any instance of the rooted set.

The *maximum* value of the individual attributes' status values is taken as the status of the query for the entire object. The basic meanings of the object status values (used by all query types) are listed below:

1. All restrictions (on attributes) were satisfied.
2. All restricted attributes definitely exist but some *may* not be satisfied.
3. Some restricted attributes may exist for some instances and not others.
4. Some restricted attributes were not found.
5. Some restricted attributes are definitely not satisfiable.

The semantics of subset query request processing is slightly more complicated because subset queries spawn attribute queries. Again appendix 2 includes algorithms (**Request-From-Outside** and **Subset-Query-Result**) which are the top level methods executed by a database object when it receives a subset query message. The processing strategy depends on the propagation of messages from the injection point down through the IS-A hierarchy, possibly all the way to the leaves. At each node visited, subset query requests spawn attribute request subqueries to determine whether individual restrictions have been satisfied. There are four basic invariants which describe what happens to object descriptions as the IS-A hierarchy is traversed towards the leaves: (1) more attribute descriptions may be added, (2) *any* attribute's definition may become more restricted, (3) *non-key* attributes may

become *key* or so restricted that they "disappear" and (4) *virtual* attributes are always treated like *non-key* attributes.

The two recursive procedures which capture the semantics of a subset request query are applied in the following way: The query names a node s as the target set, from which elements are to be retrieved; S represents the set of nodes reachable from s by following IS-A arcs and L is a subset of S containing only leaf nodes (elements). Each element of L is an object which may be retrieved by the query, if it satisfies the specified restrictions.

In each element of S, the status of all attributes named in the query is determined by sending attribute request queries to all attributes listed on the query restriction list. In each object in the set S–L (i.e., non-leaf nodes), a status is determined for each attribute by the attribute request query which is compared with the status obtained by the node's parent. This is necessary because some non-key attributes "disappear"; if the previous status was **3** and the current status is **4** then the current status must be changed to **5**. The object's status is then calculated and if it is *not* **5** then the query (including the status values) is passed to its descendants. Nodes in the set L determine the status in a similar way. This final value determines whether the object satisfies the given query; if it does, the data specified in the query's output field are retrieved and output.

Notice that all non-singleton attribute status values are calculated independently and that an *object* must wait for all of its attributes to report their status before it continues processing a query. The first observation suggests a potentially high degree of parallelism if the system is implemented on a loosely coupled multiprocessor architecture. The second observation seems to imply that any benefit from this parallelism is lost because objects spend much of their time waiting for results from other objects. This conclusion is incorrect for several reasons: First, the fact that objects spend much of their time waiting does *not* imply that PEs

are *busy waiting* or even idle. When a *PE* receives a request message, it creates an activity record for the request and when all the necessary subqueries have been spawned, it stores the activity record until it receives result messages for that request. When a result message is received, the PE determines whether it is the last result for the query; if it is not, the message is stored with the activity record. Otherwise, it is combined with the other results in order to calculate the object's status. This strategy allows for true asynchronous processing of queries and enables a high degree of parallelism without using a database management system query optimizer.

Conditionally Retrieved Information

In some cases, it is desirable to display information which depends on the status of a retrieved object or some attributes of that object. Some examples of this type of query are listed below:

1. When listing all employees, a human resource administrator might want to know the GPAs of those employees who are students.
2. When examining a list of employees, it might be desirable to list the salaries of all employees who make more the $50,000 a year.

These illustrate the two cases of dependent retrieval. In (1), displaying the additional information depends on the status of the object (in this case if it belongs to the class of students). In (2), although all employees have a salary only those salaries which are greater than $50,000 are to be displayed.

In *PolyView*, these *dependent* data are retrieved only when the unconditional query has succeeded. Notice that the Attribute-Query-Result procedure includes different strategies for *independent attribute* queries and ordinary *attribute* queries. This simple strategy allows *PolyView* to process this kind of query in a straightforward manner.

Using Views to Answer Queries

When an object receives a message there are several adjustments which it may make to the restriction list of the query. Because the internal representations of *PolyView* objects are hidden from the users, their view of the database is often restricted or incomplete. Therefore, the system must use the information in the appropriate view instance description to compensate for this by completing or further restricting users' requests.

When an object processes a request, it performs the following tasks:

1. It creates an activity record for that request.
2. The message's **view-id** is used to find the correct view instance description (if no match can be found then the query is rejected).
3. All user specified restrictions on *visible* attributes are checked. If *any* are not satisfied then the query can be rejected without further processing.
4. If, from the current point of view, the class is "invisible" and its **subclasses** and **instances** lists are both empty then again the query can be rejected without any further processing.
5. If 4 is false then the attributes (both visible and invisible) are considered in conjunction with the restrictions found in the query's restriction list[8].

   (a) For local attributes *PolyView* simply uses the intersection of the attribute definitions in the object and the message restriction list. If the result is not empty then it becomes part of the new restriction list.

   (b) Compound attributes cannot be validated locally so any view specific restrictions are appended to the restriction list.

[8] It is assumed that restrictions and attributes are mapped from the same domain. If they are not then it is not necessary to spawn a subquery for that attribute and if the attribute is key, the entire query can be rejected.

Once the query has been preprocessed "through a view", the activity record is stored and the query is allowed to propagate asynchronously through the database. Each message (subquery) which is spawned by a query will eventually invoke a response. These responses are combined and *may* be edited before a result is passed back to the source of the query. This editing process is performed by the "determine status of query" operation found in all result methods. For example, in the subset and attribute result methods, it removes "invisible" attributes before returning the result to the user.

Queries are formulated and processed through a view of the database. A view consists of a collection of *local* object interfaces. In order to process a user request, an object must load the appropriate view and use it as a window into its own internal structure and behavior. This method of processing is unique to *PolyView*.

There are two kinds of information retrieval queries. The user spawns a *subset query* when he wants to retrieve all elements of a set of objects which have particular properties. If information about the objects associated with a particular attribute is required then the user issues an *attribute query*. Subset queries refer to class objects as *sets* while attribute queries refer to them as *types*.

In both cases, the user sends a message to an injection point object which examines the request, *through* the users' view instance description, which either replies to the request directly or propagates the query to other objects. When it has received responses from all the (sub)queries, the injection point combines the (sub)results and returns a result to the sender. This query processing strategy and the two information retrieval query types are implemented by using four types of generic message. These messages are called: (1) the subset query request message, (2) the attribute query request message, (3) the subset query result message and (4) the attribute query result message. The four message types (whose methods are presented in detail in appendix 2) are illustrated in figure 13. Note that the

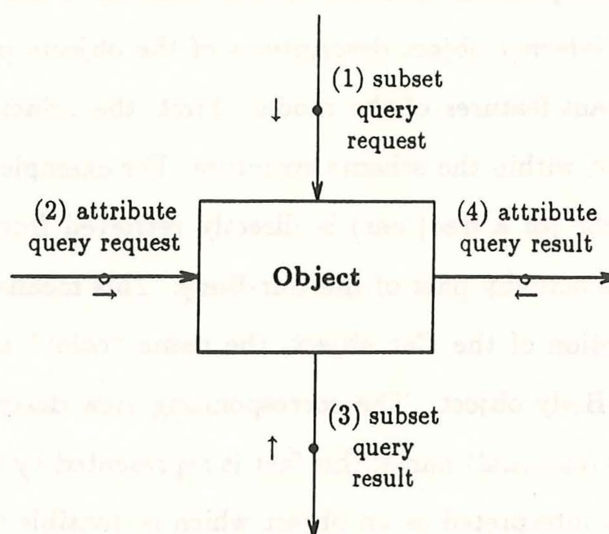


Figure 13

The Four Retrieval Message Types

arcs at the top and bottom of the object represent IS-A relationships and arcs on the sides represent ATTRIBUTE relationships.

By examining the message, an object can determine which action it should take. The data retrieval strategy will now be presented. Descriptions of the procedures used to process user requests will not be presented in the text. They are shown in appendix 2.

Mapping a User Query to a Database Object (An Example)

At the beginning of this section, it was noted that all queries are made on a view and translated to the underlying database schema using the information in the global symbol table and individual class definitions. To better illustrate this process, a simple example will now be presented. For simplicity this example will include: partial object and message definitions only and restrictions on attributes. This does not detract from the example, it simply makes it less tedious to present and easier to follow. Figure 14 shows *external* schemas for a user's view and

the corresponding (partial) system schema. Figure 15 shows the relevant parts of the *internal* object descriptions of the objects presented. It illustrates several important features of the model. First, the principle of encapsulation is strictly followed within the schema structure. For example, from the user's point of view the color (of a [red] car) is directly retrieved from the Car object; in fact, the color is actually part of the Car-Body. This means that within the view instance description of the Car object, the name "color" causes a message to be sent to a Car-Body object. The corresponding view description in the Car-Body object has no (external) name; this fact is represented by the special symbol: *invisible*. This is interpreted as an object which is invisible to the user but which performs a service within the view. In this case, the Car-Body object retrieves the color attribute.

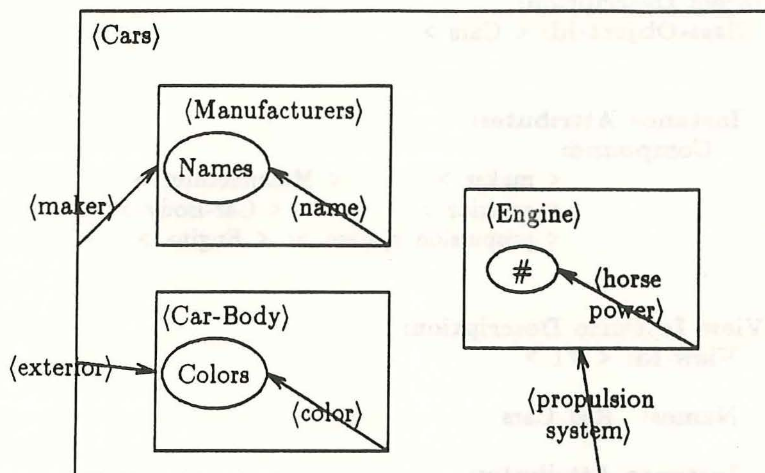
In figures 16 and 17, a simple user query will be presented. The format of the query conforms to the template shown in appendix 1. Although not all query fields are shown, their relative position is significant. Fields are separated by semi-colons and multiple semi-colons are used to indicate that fields have been omitted.

Retrieving Information for Derived Classes

Derived classes use slightly modified versions of the base class retrieval mechanism. Union-subset classes process users queries in the same way as base classes *except* that once the validity of the query has been established the query is passed directly to the base classes whose "union" forms the basis for union-subset. Since this involves a single extra step the algorithm is not shown in the appendix. When a base class receives an information retrieval request from a (union-subset) derived class, the request is treated in exactly the same way as a request from a *user*.

Query processing through power-sets is a little different. Power-Sets treat ordinary union-subset classes as instance objects. Once a subset request query is

Actual Database:



User View:

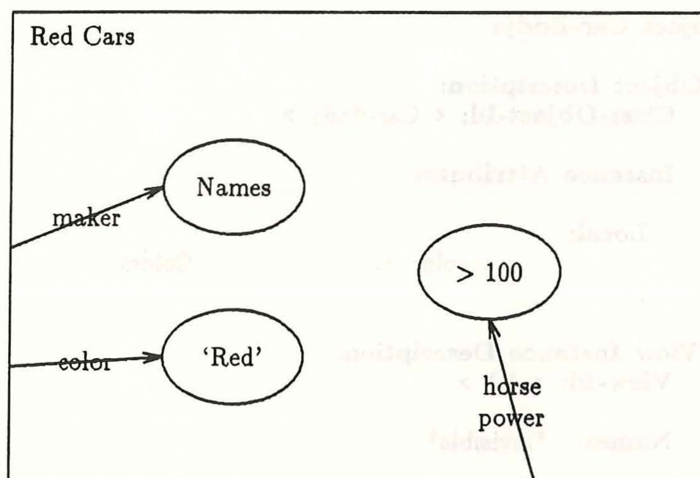


Figure 14

External Schemas

has been found to be processable by a power set it is sent to each of its (union-subset) members for further processing. Each union-subset object retrieves the requested information and returns it to the requesting power set.

Object Cars:

Object Description:
Class-Object-Id: < Cars >
.

Instance Attributes:
Compound:
< maker >: < Manufacturer >
< exterior >: < Car-Body >
< propulsion system >: < Engine >
.

View Instance Description:
View-Id: < V1 >

Names: Red Cars

Instance Attributes:
Compound:
maker: < maker >
color: < exterior >
horse power: < propulsion system>
.

Object Car-Body:

Object Description:
Class-Object-Id: < Car-Body >

Instance Attributes:

Local:
< color >: Colors
.

View Instance Description:
View-Id: < V1 >

Names: *invisible*

Instance Attributes:

Local:
color: < color >, = 'Red'
/* Note that this restricts the choice of colors to Red */

Figure 15

Internal Object Descriptions

The English "equivalent" of the query is shown in *italics*; comments are shown in roman font.

List all Red Cars Made by (a Manufacturer) Named Ford.

In determining the query type, the key word is "list" indicating that the system should return a set of car object descriptions. Further notice that values which are generated by *PolyView* have been enclosed in angle brackets (⟨⟩) and that messages have been enclosed in curly brackets ({}). Abbreviations: Q_1 = query identity for query 1, U_1 = user 1's name, and V_1 = the identity of view 1 ...

Message to *PolyView*:

$Query_1$: {Red Cars; U_1; ⟨Q_1⟩; ⟨5⟩; ⟨V_1⟩; ⟨subset-request⟩; ⟨*user*⟩; maker = "Ford"}

PolyView uses the global symbol table to determine the target of this query. The name "Red Cars" is view dependent so information about the user is used to identify which view model object table should be used. Finally the message is resent to the Cars object.

Message to Cars:

$Query_1$: {⟨Cars⟩; U_1; ⟨Q_1⟩; ⟨5⟩; ⟨V_1⟩; ⟨subset-request⟩; ⟨*user*⟩; maker = "Ford"}

The view description corresponding to ⟨V_1⟩ is used as a guide to spawn subqueries. In this case, queries would be sent to Manufacturers, Engines and Car Bodies — the objects which represent the attributes of Cars. Since the user query specifies that the "maker" is Ford, that restriction is sent on to the Manufacturer class:

Message to Manufacturers:

$Query_{1.1}$: {⟨Manufacturers⟩; ; ⟨$Q_{1.1}$⟩; ; ⟨V_1⟩; ; ⟨attribute-request⟩; maker = "Ford"}

Note that in order for the manufacturer object to process this query, its view instance description (for user 1) would include the following:

maker: ⟨name⟩.

Figure 16

A Sample Query (part 1)

In addition to sending a message to the manufacturer class the car object also sends a message to the car body class. From this user's point of view this is an unrestricted request — from a global perspective it is not.

Message to Car-Body:

$\text{Query}_{1.2}$: {⟨Car-Body⟩; ; ⟨$Q_{1.2}$⟩; ; ...; ⟨empty restriction list⟩}

When the request is received view 1's view description is activated and the restriction on color (⟨color⟩ = "Red") is added to the (empty) restriction list before processing continues.

$\text{Query}_{1.2}$: {⟨Car-Body⟩; ; ⟨$Q_{1.2}$⟩; ; ...; ⟨color⟩ = "Red"}

After the three attribute requests have been completed the Cars object would spawn subset request subqueries to be processed by its subclasses and instances. The restriction list of the new subquery would be altered to reflect the changes made during the processing of the attribute requests. Note that both the user and view specified restrictions on an entire attribute path are now included by the system.

Messages to subsets of Cars:

Query_1: {⟨subsets of Cars⟩; ; ...; ⟨maker⟩.⟨name⟩ = "Ford", ⟨exterior⟩.⟨color⟩ = "Red", ⟨engine⟩.⟨horse-power⟩ 100},

The query restriction comes from both the original query and the user view: The user asked for all red cars made by Ford (which restricts the name attribute of the maker attribute: ⟨maker⟩.⟨name⟩ = "Ford"), in this view red cars are cars whose color is limited to red (the color of the exterior (⟨exterior⟩.⟨color⟩ = "Red") and whose engines have more than 100 horse power (⟨engine⟩.⟨horse-power⟩ 100).

Figure 17

A Sample Query (part 2)

Updating PolyView Databases through View Templates

The following update operations are supported by the *PolyView* system: **insert, delete** and **move** for objects and **insert, delete** and **change** for attributes. A primitive *change object* operation is not supported because it is not necessary.

Changing an object involves changing some of its attributes but not its identity; therefore, this operation can be implemented by executing insert, delete and change *attribute* operations.

Some derived data will not be updatable by users. For example, if a method is used to calculate an aggregate value based on values found in many objects, this value will obviously not be directly updatable. In addition, because some views hide information from users, there is the possibility that a given user may not be able to *consistently* change the contents of some classes in the system. The *PolyView* system uses *invisible* attribute structures to remedy some of these problems.

Each view instance contains several lists of attributes which refer to the internal object description, restrict their domains and, in some cases, provide default object structures to be used by insert operations. The object description contains attribute properties which are invariant across all views. These properties include each attribute's "unrestricted" domain and (more importantly) whether an attribute is **key** or not. In order for a user to have update privileges, the following conditions must be satisfied: (1) the combined list of visible and invisible attributes in the associated view instance description must contain all *key* attributes and (2) all invisible key attributes must have a default object (or value) associated with them. This guarantees that the *PolyView* system can generate consistent objects from a combination of user requests, the view description instance and the object description. Other update operations can be performed if the objects which are to be updated can be identified. The asynchronous message passing strategy used to retrieve information will be used to identify the objects which are to be deleted or changed.

Inserting New Objects and Attributes

Finding the Target Class

All operations are performed "through" a user's view of the application world. A user directs a request at a visible *injection* point which may not be the appropriate owner class (*target*) for the new object. Starting at the injection point, *PolyView* recursively searches for the target. Each descendent of the injection point determines whether it is a potential target using a strategy similar to the one used to retrieve an attribute value. For this reason, when a class object receives an insert request, it first determines whether it may be the target class. The search within the class hierarchy (which is almost identical to the Subset-Query-Request[9] search strategy) continues until a class is found which is *not* a candidate for the target or the bottom of the class lattice is reached. The organization of the *PolyView* hierarchy guarantees that there will be exactly one target. The major actions performed are:

- Spawn attribute query requests.
- If the result indicates that the new object could not be inserted at this class (status = 5) then inform the sender.
- Otherwise, if there are subclasses then send the insert object request to them.
- If *all* results, from subclasses, are 5 then change status to 1 and perform the insert operation. Otherwise (a descendant has already inserted the object), report success to parent.
- If there are *no* subclasses then status values of 2 or 3 are changed to 1 (this is the target) and a status of 4 is changed to 5 (this is not the target).
- If the status is 1 then the insert operation can be performed.

[9] The main difference is that in order to find the target class it is only necessary to search the class hierarchy and *not* the database itself.

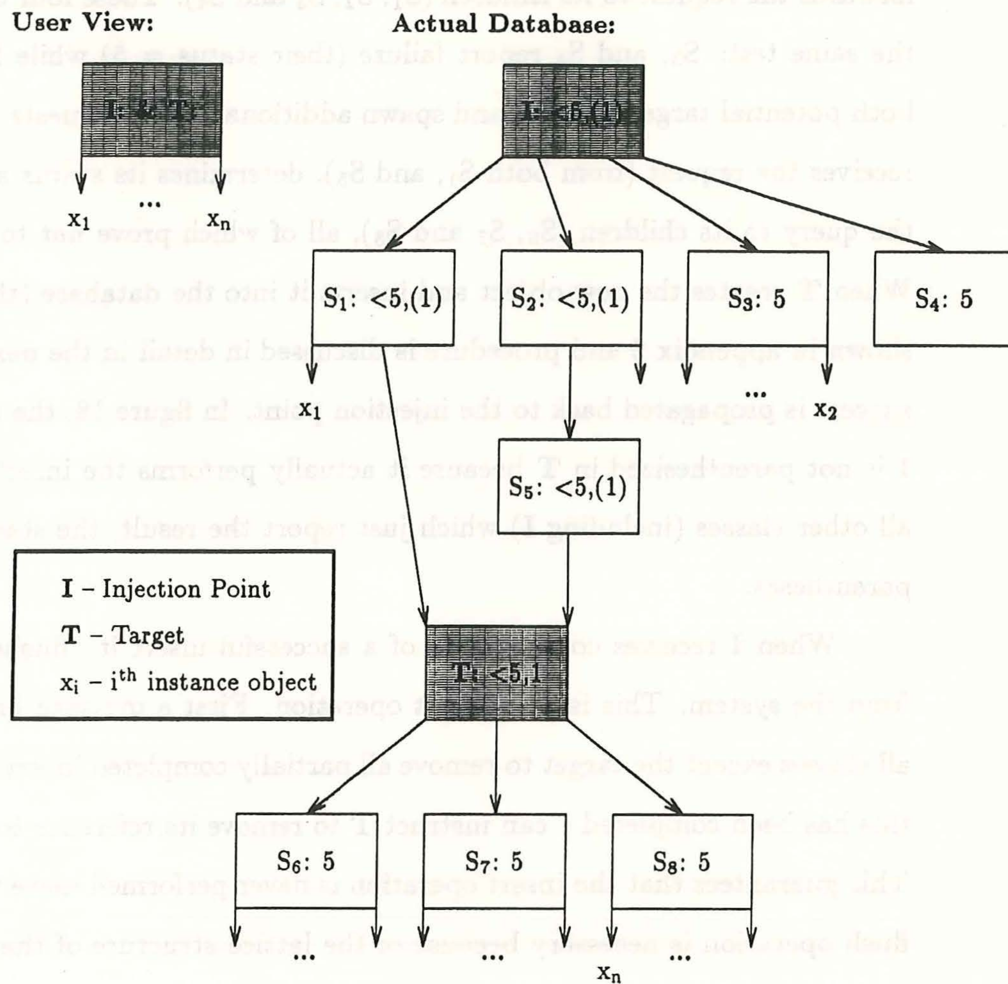


Figure 18

Find Insert Target Set

- If the same request is received from more than one source, ignore the duplicates.

Consider the abstract *PolyView* hierarchy shown in figure 18. From the user's point of view, the target and the injection point are the same object. The injection point class (**I**) is actually an ancestor of the appropriate insert target class (**T**). When class **I** receives the original request for the user, it determines that it is a potential target (shown by the <5 symbol in the class node) and immediately

forwards the request to its children (S_1, S_2, S_3 and S_4). These four classes perform the same test: S_3, and S_4 report failure (their status = 5) while S_1, and S_2 are both potential targets (like **I**) and spawn additional insert requests. Eventually, **T** receives the request (from both S_1, and S_5), determines its status and propagates the query to its children (S_6, S_7 and S_8), all of which prove *not* to be the target. When **T** creates the new object and inserts it into the database (the algorithm is shown in appendix 2 and procedure is discussed in detail in the next section), the success is propagated back to the injection point. In figure 18, the result status of 1 is not parenthesized in **T** because it actually performs the insert operation. In all other classes (including **I**) which just report the result, the status is shown in parentheses.

When **I** receives confirmation of a successful insert it "flushes" the request from the system. This is a two part operation. First a message broadcast causes all classes except the target to remove all partially completed insert requests. Once this has been completed **I** can instruct **T** to remove its reference to the operation. This guarantees that the insert operation is never performed more than once. The flush operation is necessary because of the lattice structure of the *PolyView* IS-A hierarchy through which a target may receive the same request from more than one parent. In the simple sample **T** received insert requests from both S_1 and S_5. Since there is no centralized control, **T** must "remember" that it performed the insert operation until **I** tells it to "forget".

Inserting an Object (into a Target Class) Using a View Template

Once a target class has been identified, a new object is created and inserted into the database. A *shallow copy* (with its own identity) of the (view sensitive) default object is created and an IS-A connection is made from the class to the new object. This shallow copy is a "shell" from which an object will be built; therefore, it is *not* fully incorporated into the system until the insert object operation has

been successfully completed. When the shell is created, attribute connections are established *to* each (non local) default attribute object but *not* from them because many default objects will be changed by the insert method. Each local attribute is a value and can be changed in a straight forward manner. In order to customize a compound attribute, the attribute connection from the new object to the default attribute object is severed and an Insert-Attribute-Request query is spawned. Once the new object has received Insert-Attribute-Result messages for all the customized compound attributes, the remaining default attributes will be made a permanent part of the new object. Each default object is sent an insert-attribute-arc message which causes it to create an *inverse attribute* connection (an OOP) to the requesting object.

The Insert-Attribute methods are quite simple:

- The system is searched.
- If a *single* object is found which matches the attribute description, it becomes the attribute object and the corresponding attribute/inverse connection (between the two objects) made.
- If several objects or no objects match the attribute description, the Insert-Object-Request[10] method is used to create a unique object.

Adding an Object Using the View Structure (An Example)

The schemas shown in figures 14 and 15 will be used to illustrate how *PolyView* uses information (some of which are *not* visible to the user) in a view description template. In this example, the description of the Car-Body object will be looked at in more detail in order to illustrate the use of "invisible" and default

[10] Notice that Insert-Attribute and Insert-Object are mutually recursive.

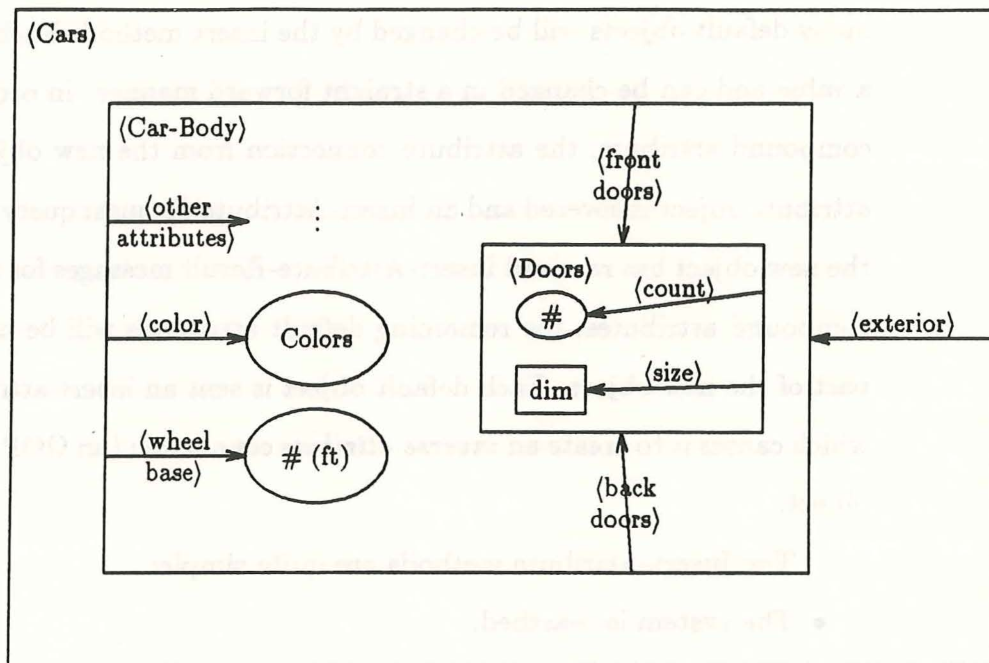


Figure 19

Partially Expanded External Schemas

objects during the insert process. Figure 19 shows the partially expanded database schema. The corresponding internal templates are shown in the next figure.

The insert operation from this perspective (V_1) is very simple: the user simply specifies the (name of the) maker and horse power (of the engine) — *PolyView* does the rest. Figure 21 shows how the request "insert a 150 hp Ford" would be processed by the Cars, Car-Body and Doors (which receives two requests) classes. Exactly the same strategies are applied by the other classes which represent other attributes and will not be illustrated here.

```
Object Car-Body:
  Object Description:
    Class-Object-Id: < Car-Body >
      Instance Attributes:
        Local:
                    < color >:                 Colors
                    < wheel base >:            #
        Compound:
                    < front doors >:           < Doors >
                    < back doors >:            < Doors >
  View Instance Description:
      View-Id: < V1 >
      Names: *invisible*
      Instance Attributes:
        Local:
          Visible:
                    color:           < color >, = 'Red' /* The color is Red */
          Invisible:
                    wheelBase:       < # >, = '18' /* The wheelbase is 18 feet */
        Compound:
          Invisible:
                      < front doors >
                      < back doors >

Object Doors:
  Object Description:
    Class-Object-Id: < Doors >
    Instance Attributes:
      Local:
                    < count >:                 #
                    < size >:                  dim
  View Instance Description:
    View-Id: < V1 >
    Names:  *invisible*
    Instance Attributes:
      Local:
          Invisible:
                    < front doors >.< count >: < count >, = '2'
                    < front doors >.< size >: < size >, = '2x4'
                    < back doors >.< count >: < count >, = '0'
                    < back doors >.< size >: < size >, = 'nil'
```

Figure 20

Partially Expanded Internal Object Descriptions

Deleting Objects and Attributes

It is very important to remember that all requests received from users are (incrementally) translated from a user specific external view to an equivalent

Message to *PolyView*:

Insert$_1$: {Red Cars; U_1; $\langle I_1 \rangle$; $\langle 5 \rangle$; $\langle V_1 \rangle$; ⟨insert-object-request⟩; ; horse power = "150", maker = "Ford"}

PolyView determines that the message should be sent to the Cars object.

Message to Cars:

Insert$_1$: {⟨Cars⟩; ; ... }

For simplicity it is assumed that the Cars is the target of the insert operation. Then view description is used as a template from which a shallow copy of a default object is created (which the system fleshes out as the operation continues). Once the object has been created (complete with default values for local attributes and unique identity), *PolyView* spawns appropriate subqueries. These subquery messages are sent to the owners of Car's compound attributes (Manufacturers, Engines and Car Bodies).

Message to Car-Body:

Insert$_{1.2}$: {⟨Car-Body⟩; ; $\langle I_{1.2} \rangle$; ; $\langle V_1 \rangle$; ; ⟨insert-object-request⟩; color = ""}

The Car Body class behaves like the Car class. It first finds the appropriate view description, then creates a shell and finally spawns two subqueries (both of which are sent to the Doors object).

Message to Doors:

Insert$_{1.2.1}$: {⟨Doors⟩; ; ; ; ; ; ; ⟨front door⟩ = ""} &

Insert$_{1.2.2}$: {⟨Doors⟩; ; ; ; ; ; ; ⟨back door⟩ = ""}

Again the view template would be used to create complete (invisible) default objects which represent the front and back doors of the new car instance respectively. Since doors do not contain references to other complex objects this part of the insert operation is complete. The Doors class spawns reply messages (containing references the newly created instances) which are sent back to the Car Body object which can then respond to the Car object etc.

Figure 21

A Sample Insert Operation

internal representation. This is accomplished by translating printable (external) names to their unique internal equivalences when they are first encountered. The reference to the target class is changed immediately using the global symbol table; attribute names are converted when they are first used by a class object.

When a class receives a delete object request it executes the following strategy:

- An Attribute-Query-Request is spawned to ensure that the class may lead to instances which are to be deleted.
- If the Attribute-Query-Result request succeeds then the delete object request is propagated to all children (subclasses and instances) of the class. Otherwise no objects are deleted and the class can report that result.

When an instance object receives a delete object request:

- It spawns an Attribute-Query-Request.
- It removes itself from the database, if the Attribute-Query-Request succeeds. Otherwise it must only acknowledge the request (send a result message to the requester) but take no further action.

As is the case with all *PolyView* operations, the objects at the lower level report their actions to their parent(s) which combine them with other results and pass the aggregate result back up the hierarchy.

When an instance object recognizes that it is the target of a delete request, it performs a two stage delete operation. First, it spawns a single Delete-All-Attributes-Request which in turn sends Delete-Inverse-Attribute-Requests to all of its compound attribute objects. These subqueries cause the attribute objects to break their connection with the requesting object. When all compound attribute connections have been broken, the second part of the delete operation can be executed. The object requests that its owner delete the IS-A connection to the "deleted" object and remove its own connection to the owner. The resulting

structure is a partially defined unreachable instance object which can be removed without side effects when the system needs its space.

To further illustrate this procedure, consider simple external schema in figure 22. It represents a delete object operation performed on an abstract hierarchy. In the figure, the target instance has two compound attributes, called A_1 and A_2, which are connected to attribute $instance_1$ and $instance_2$, respectively. Before the delete operation is performed, the target is fully connected to its parent and neighbors; after the delete operation has been performed, the target instance is isolated and can be garbage collected at an appropriate time.

The Delete-Attribute-Request method allows the user to delete a single attribute, as opposed to an object. The algorithm is very simple:

- The instance (or instances) from which the attribute is to be deleted are located.
- If the attribute is local then it is deleted. (Note that the algorithm in appendix 2 shows that some local attributes have special delete methods associated with them. Special methods are necessary because, for example, some attributes may not become nil in a consistent database.)
- If the attribute is not local then a request is sent to the corresponding attribute instance object. This request causes the receiver to delete its reference to the target object. Once this has been completed, the target removes its reference to the attribute. Again there are both generic and special methods to accomplish this.

Unlike the delete object operation, this procedure removes the attribute connection to *and from* the target instance but does not necessarily leave any object isolated.

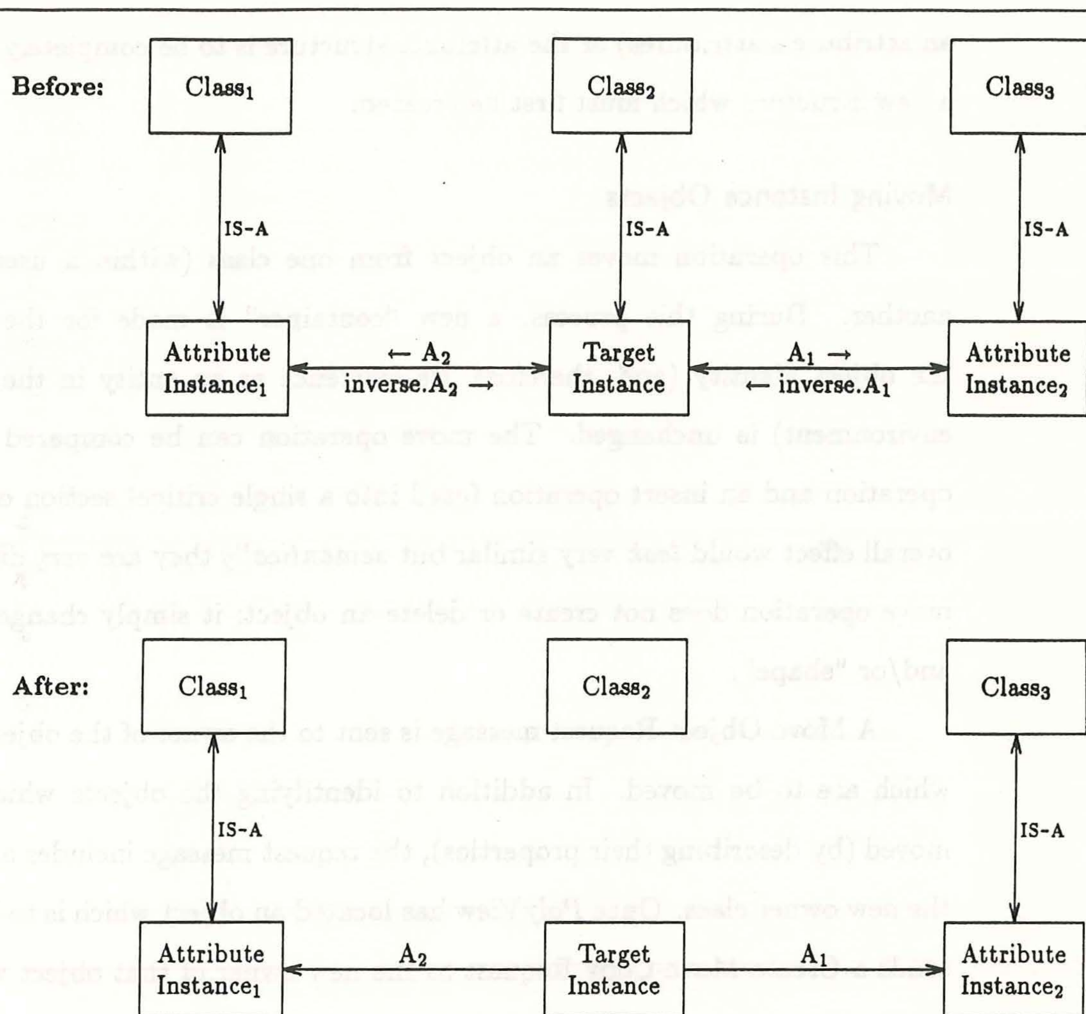


Figure 22

Deleting an Instance from a PolyView Database

Changing Attribute Instances

The Change-Attribute method is very similar to the Delete-Attribute method. Explaining it in detail would be unproductive; therefore, the important differences will be described very briefly. Local attributes are simply assigned a new value. For compound attributes there are two possibilities: either a part of complex object structure is to be changed by recursively applying the change method (to change

an attribute's attributes) or the attribute structure is to be completely replaced by a new structure which must first be created.

Moving Instance Objects

This operation moves an object from one class (within a user's view) to another. During this process, a new "container" is made for the object but the object identity (and, therefore, its existence as an entity in the application environment) is unchanged. The move operation can be compared to a delete operation and an insert operation fused into a single critical section of code. The overall effect would *look* very similar but *semantically* they are *very* different. The move operation does not create or delete an object; it simply changes its "type" and/or "shape".

A Move-Object-Request message is sent to the owner of the object or objects which are to be moved. In addition to identifying the objects which are to be moved (by describing their properties), the request message includes a reference to the new owner class. Once *PolyView* has located an object which is to be moved, it sends a Create-Move-Copy-Request to the new owner of that object which causes it to create a new container for that object. *PolyView* then copies the existing object identity from the old object to its new container and all properties which are associated with its new type. If there are additional properties which are associated with the new object type then the system will insert the appropriate default object references/values. When all this has been completed, the old object is removed and the operation is complete.

Updating Derived Classes

Derived classes contain objects which belong to base classes. The actual creating, deleting and changing of these objects is, therefore, only implemented by base classes. Certain kinds of update can be instigated by derived classes. For

example, a user might want to add or remove something from a category (like Banned Ships). This would be achieved by sending a request to the appropriate base class (Ships) requesting that it change the appropriate category attribute. Nothing would actually be directly changed by the Banned Ships category. Notice that there is no comparable operation for collections because membership in a collection depends on the contents of objects' attributes. Therefore, collections may change as a side-effect of some other operation on the database.

CHAPTER 5
Supporting Semantic Relativism

The concept of *semantic relativism* is very important in any multi-user database environment. The underlying data model must be rich enough to support many, perhaps conflicting, *views* of the data's structure and semantics. In this chapter semantic view transformations are presented which enhance the object-oriented semantic database model introduced in the previous chapters. These transformations enrich the model by allowing many different users' views of the database to be created within a single polymorphic schema.

In chapter 4, the operations on a *PolyView* database were presented. These operations represent one of the forms of *polymorphism* supported by *PolyView.* The same operation (retrieve, for example) is performed by all objects in a database. The semantics of an operation is the same regardless of the object type; however, its implementation may differ significantly between object types. In the last chapter, methods were shown to operate through a view distributed template. In this chapter, a mechanism for constructing these templates will be presented.

Motivation

The typical large database enterprise has many different user groups. Each of these groups may want to use the database in different ways. This means that each class of users is likely to have a different perception of the structure and semantics of the data. The term *user view* will be used to describe the (dynamic) restructuring of the database in order to accommodate the different, and possibly conflicting, needs of the various database users. This situation can be illustrated by

considering the concept of a marriage. Marriage can be thought of as a relationship between a man and a woman, a legal entity (the way the town hall's record office views a marriage), or one spouse may be deemed to be an attribute of the other (i.e. a husband may be listed as a dependent in his wife's employee record). In most database systems, the database designer would have to choose to support a single view of marriage or to maintain several instances of marriage separately. Clearly, if an application requires the support of several different views of marriage, the first choice is unacceptable. Unfortunately, the latter is almost as undesirable because inconsistencies may develop between the independent representations of a single entity. In this chapter, several *generic transformations* will be described which allow *all* the user views to be modeled within a single polymorphic database schema.

PolyView has several unique features for supporting user views. The most significant of these is the concept that a view is a distributed (model) object and that, like any other *PolyView* object, it has a unique *identity* (or *color*). Secondly, each view supports many levels of relativism. Relativism is supported in attributes, instance objects, class objects, IS-A and ATTRIBUTE hierarchies, and the "global" schema. Each view is associated with a group of users and each group of users may access the database through several views. Messages (requests for information and updates) sent from a user have the appropriate view name appended to them. When a message is received by any object, it uses the view information to retrieve the users view instance description. The information in this description is used to adjust its behavior and structure to suit the user's particular needs.

A Framework for Relativism

In chapter 4, it was shown how view descriptions are used as users' windows into an application. In this chapter, the internal structure of (all levels of) the database will be examined more closely. The mechanisms for creating and customizing

user views will be presented in detail. Relativism is achieved in the *PolyView* system by predefined semantic transformations which can be applied to a schema to produce a view graph. A sequence of these applied transformations form a *view definition* which is *distributed* through the class objects within the view schema. The view definition is a "distributed method" which acts as a filter that alters queries as they propagate through the view schema causing the query message to conform to the underlying database and the results message to conform to the view. The view definition is also used to generate an *external view template*. An external template is the user's *guide* to the database; to the user, the external template *is* the database. In order to support all aspects of the *PolyView* data model, each class object is be divided (internally) into three sections: a *class description* section, a *view description* section and a private local *work area section* (which is invisible to *all* users but is necessary and used by the system). The class description and view description sections form the class's *internal template* which contains two kinds of information: The first (which is contained in the class description section) describes the intension and extension of the set of instance objects represented by the class and the second (contained in a series of view instance descriptions) describes the various users' views of that class and its data.

Customizing a User Schema

View transformations can be thought of as graph editing procedures. They include methods for coloring, building and changing graphic structures. Conceptually, editing a *PolyView* view is very simple: A copy of a schema (graph) is made. By using the view transformation methods, nodes and arcs may be added, removed or changed in a view graph. When all the desired changes have been made, the new graph is colored and added (or returned to) the application environment. The coloring procedure starts at the root (the generic application) and follows outgoing IS-A and ATTRIBUTE arcs from each class object visited. Each visited class object

is marked with the view's unique color. The resulting colored graph is the new user schema.

The strategy outlined above is extremely high level. The basic view transformations which are discussed in the remainder of this chapter are necessarily lower level and actually make changes to the *internal template* of one or more class objects. The effects that these methods have on the internal structure of the database, individual objects and the users' perception of the database will be shown.

Basic View Transformations

The basic view forming transformations fall into three general categories: graph tailoring methods, operations on the IS-A hierarchy and methods that define and redefine ATTRIBUTE relationships. Graph tailoring transformation methods create, record or delete an entire view schema. They operate on the view *model*; from this perspective, the view schema is treated as a single object. Other transformations affect individual classes and the IS-A or ATTRIBUTE decomposition (or aggregation) hierarchies *within* a particular view schema. Once a (virtual) view schema has been created (by applying the make-view graph method), other transformation methods are applied to customize the schema to meet the requirements of the user. Descriptions of the transformations will be presented in two parts: first, the operation will be described and then, in most cases, internal and/or external structures will be shown in order to demonstrate or clarify the effect of the operation on a *PolyView* database.

Graph Tailoring Methods

There are four graph tailoring methods: MAKE-VIEW, REMOVE-VIEW, COLOR and QUIT.

Make-view, creates a *virtual* copy of a database (view) schema. It is the first operation performed when a new view is to be created. It creates a unique identity (color) for the view and enters it in the global symbol table which has the effect of returning a reference to the new (temporary) view schema. Each global symbol table entry contains the name of a user associated with the view, the view's unique color (identity), and a reference to a view model object table. Initially, a new view references a generic default object which may, eventually, be included in the permanent database environment. Each view model object table contains the information used by the system to translate external class names (used by users) to the unique internal names used by the system. The structure of the global symbol table was described in chapter 3 and is also shown in appendix 1.

Once a view graph has been created, it can be customized. These changes, made by the view transformations, must eventually be incorporated into the database environment. For example, view dependent names for classes must be associated with the view model (by adding them to the appropriate table) so that queries on view schemas can be forwarded to the correct global subschema.

The *color* method publishes the new view schema. It sends messages which "traverse" the view graph and collect external names for each class in the view while (simultaneously) causing each class object to merge the temporary changes (in its work area) into its view and class description areas. When the entire view graph has been colored, the status of the view model object table in the global symbol table is changed from "generic" to "permanent" and the publishing process is complete. These actions cause the new view to become a permanent part of the database environment. Occasionally, a view designer will want to abort a view editing session; the *quit* method allows him to do just that. Quit is similar to color in that it traverses the view graph, except that instead of saving all the temporary changes (within the class objects), it removes them; when a "quit traversal" is completed,

the (temporary) entry in the global symbol table is also removed. Either color or quit is, therefore, the final operation performed when creating (or changing) a user view.

The ***Remove-view*** method is used to remove an obsolete view from the database environment. Its input, which is provided by the system, is a reference to the view to be removed. Remove-view messages are broadcast to all class objects and those which recognize the view remove it from their internal templates. Finally, all references to the view are removed from the global symbol table.

Customizing the Is-a Hierarchy

CLONE, ATTACH, REMOVE, HIDE and (re)NAME are the five methods for customizing the IS-A hierarchy. They affect class objects and the IS-A relationships between them. These methods are very simple and the changes are "localized" — i.e. they are visible only from the particular user's point of view. It cannot be over emphasized that these transformations to the schema change the user's view but are completely invisible in all other database users. All changes are temporarily associated with the view which is being formed within the class objects' local memory. Only when a color message associated with that view is received are the temporary changes made a permanent part of an object's description.

Name is the simplest of these operations; it associates a list of new external names with a class. For example, if the database contains a class which represents a set of Automobiles and a particular user prefers the name Cars, the *name* method would be used to associate the symbol "Cars" with that class. Name has three parameters: a view identity (color), a reference to the class which is to be renamed and a list of new names. The generic form of name method is: ***Name***($View_{Id}$, Old-name, New-names). The view designer provides old and new external names for the class; the system associates the new name with the view in the class object's local memory. When a color message is received, the new external name(s) will

BEFORE:

Object Automobiles:

Object Description:

•

•

View Descriptions

View Instance Description:

View-Id: < Vjim >

Names: Automobiles

AFTER:

Object Automobiles:

Object Description:

•

•

View Descriptions

View Instance Description:

View-Id: < Vjim >

Names: Cars, Autos

Figure 23

Applying the Name Method to the Automobile Class

(An Internal Representation)

be associated with the unique class identity in the appropriate view model object table. This allows *PolyView* to use the global symbol table in order to translate queries from a user's personalized perspective (with references to private external class names) to an equivalent internal format containing only internal names.

In figures 23 and 24, the *name* transformation is demonstrated using a concrete example. *PolyView's* internal and external representations are presented in order to show the effect of changing the external name of a class from `Automobiles` to `Cars` *or* `Autos`. This is the view of a user called Jim and the color of his view is represented by V_{Jim}. The example includes a class called `Fords` which is a

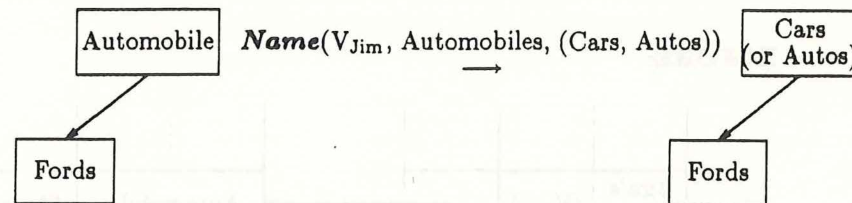


Figure 24

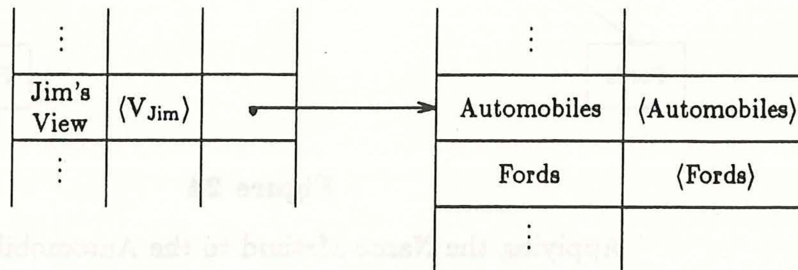
Applying the Name Method to the Automobile Class

(An External Representation)

descendant of **Automobiles**. **Fords** would be unaffected by the renaming of its ancestor. Name, like all other transformation has no apparent effect until a *color* message is received by the **Automobiles** class. This causes the class to permanently record the name in its view description area and to "publish" new external class names in the global symbol table. The full global effect of the *name* transformation followed by *color* operation would include a change to the global symbol table; this is shown in figure 25.

In order to create a new class, a user selects its parent and then asks the parent to *clone* itself. The *clone* method makes a copy of and establishes a subclass IS-A link between them. A *clone* message includes: the message name (clone), a view color, the clone's name and the original class' name. The following representation will be used: ***Clone***($View_{Id}$, Clone-name, Class). The new class object contains copies of most of the original class' object description and its global view description, including its methods and attributes and their external names, but ***not*** its instances and subclasses. The clone's object description differs from the original in two places: the clones *parents* IS-A connection list contains a single reference to the original class which is also added to the *type* list. Once created, a cloned class can be customized by redefining its existing attributes and methods, and by adding new properties and subclasses. This operation adds a new class to

BEFORE:



AFTER:

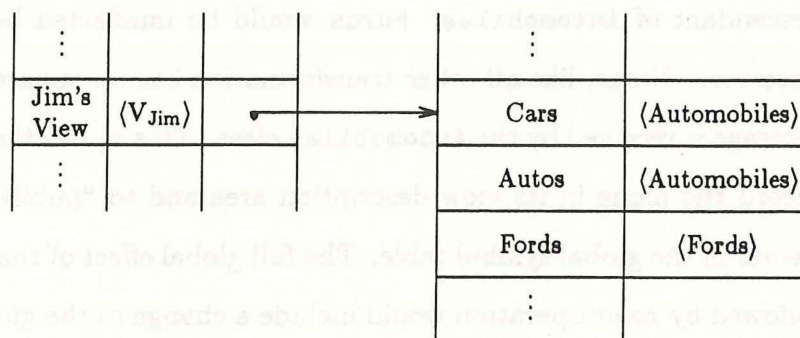


Figure 25

Applying the Color Transformation to the Example in the Previous Figures

the schema but the only pre-existing class which is affected by *clone* is the one which applied it. This class adds a new element to its *subclasses* IS-A list. Note that this, like all local view transformations, causes a temporary change which becomes permanent when the view is colored.

The *clone* operation will now be demonstrated. Its use will be shown in conjunction with the *color* operation. In the previous example, part of a simple IS-A hierarchy which contained exactly two classes (`Cars` and `Fords`) was presented. Figure 26 shows part of the internal representations of `Cars` before and after it has been cloned. The cloned class (`Chevys`) is also shown. Figure 27 shows an external *PolyView* schema before *and* after `Cars` has applied the *clone* method to create

```
BEFORE:

Object Automobiles:
  Object Description:
    Class-Object-Id:  < Automobiles >
    Is-A Connections:
      Types:          < list of types >
      Parents:        < list of parents >
      SubClasses:     < Fords >
      Instances:      < list of instances >

  View Description:
    Global View:      < global view description of automobiles >
    User Views:       < other view instance descriptions >

AFTER:

Object Automobiles:
  Object Description:
    Class-Object-Id:  < Automobiles >
    Is-A Connections:
      Types:          < list of types >
      Parents:        < list of parents >
      SubClasses:     < Fords >, < Chevys >
      Instances:      < list of instances >

  View Description:
    Global View:      < global view description of automobiles >
    User Views:       < other view instance descriptions >

Object Chevys:
  Object Description:
    Class-Object-Id:  < Chevys >
    Is-A Connections:
      Types:          < list of automobile's types > + < Automobiles >
      Parents:        < Automobiles >
      SubClasses:     < nil >
      Instances:      < nil >

  View Description:
    Global View:      < global view description of automobiles >
    User Views:       < nil >
```

Figure 26

The Clone Method is Applied by the Cars Class

Creating the Chevys Class (Internal Structure)

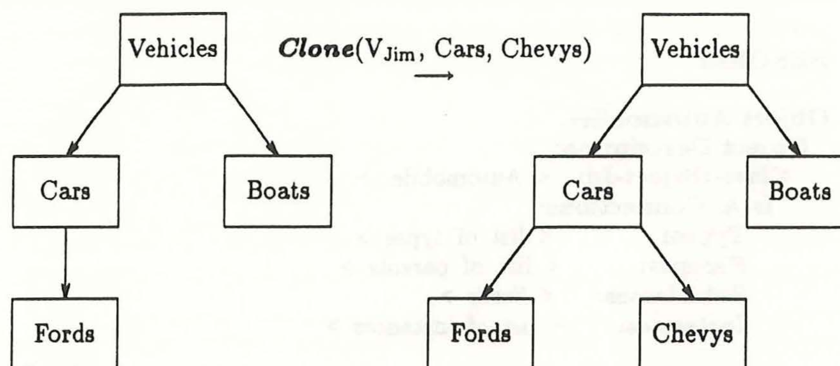


Figure 27

The Clone Method is Applied by the Cars Class

Creating the Chevys Class (External Structure)

`Chevys`. Notice that the view of the database has been expanded to include both the `Vehicles` and `Boats` classes which will be used in future examples. Finally, figure 28 shows the change to the global symbol table after *color* has been applied.

Attach creates an IS-A connection between two *predefined* classes in a database application. This operation must be supported because *PolyView* supports multiple inheritance. When sending an *attach* message, the user must identify the new child and parent objects while the system provides the view's color. When a (new parent) class object receives an *attach* message, it adds an IS-A arc to its subclass list to reference the new child and it sends an *attach-parent* (which contains its own identity) message to that child. When the (new child) object receives an *attach-parent* message, it adds an IS-A arc to its parent list and (if necessary) to its type IS-A list. Now *PolyView* must make sure that the two class descriptions are consistent with each other. In all *common* properties, the domain of the child's property must be a subset of the domain of the parent's property. If this is not the case then *PolyView* enters an interactive mode so that the view administrator can resolve the anomalies caused by the *attach* transformation or abort it if the differences cannot be resolved. Note that a successful *attach* operation causes *both*

BEFORE: — see AFTER from previous example

AFTER:

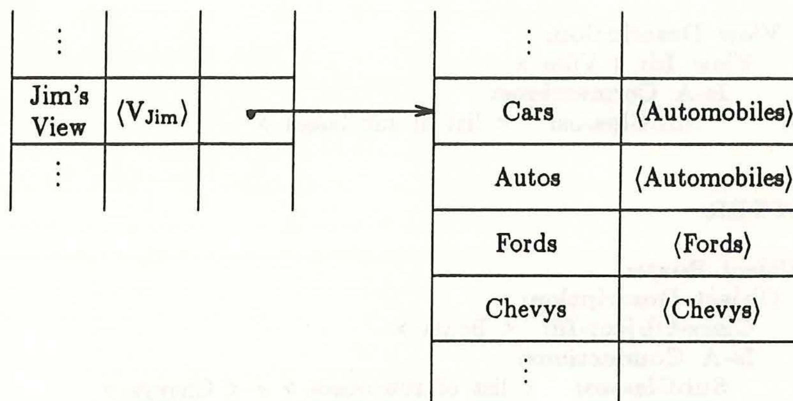


Figure 28

The Clone Method is Applied by the Cars Class

Creating the Chevys Class (Global Symbol Table)

the parent and the child class objects to change their internal templates. The generic form of this operation is ***Attach***(View$_{Id}$, Parent, Child).

Clone and attach are often used together — clone is used to create a subclass of a single parent and attach adds additional parents if they are necessary. Figures 29 and 30 show a continuation of the previous example. The newly created `Chevys` class object is *attach*ed to the `Boats` class. Figure 29 shows an internal snapshot of the "after" of this operation has been performed; the "before" can be seen in the previous example. Figure 30 shows the equivalent external before views. The global symbol table will not be presented in this example because this operation does not change it.

Some (or all) *outgoing* IS-A arcs may be *remove*d by a class object. When applied to a single arc, remove's effect is *almost* exactly the inverse of *attach.* The

```
BEFORE: /* for Chevys see AFTER from previous example */

Object Boats:
  Object Description:
    Class-Object-Id:  < Boats >
     Is-A Connections:
        SubClasses:    < list of subclasses >

  View Description:
     View Id: < Vjim >
        Is-A Connections:
           SubClasses:    < list of subclasses >

AFTER:

Object Boats:
  Object Description:
     Class-Object-Id:  < Boats >
     Is-A Connections:
        SubClasses:    < list of subclasses > + < Chevys >

  View Description:
     View Id: < Vjim >
        Is-A Connections:
           SubClasses:    < list of subclasses > + < Chevys >

Object Chevys:
  Object Description:
     Class-Object-Id:  < Chevys >
     Is-A Connections:
        Types:          < Automobiles >, < Boats >
        Parents:       < Automobiles >, < Boats >

  View Description:
     View Id: < Global >
        Is-A Connections:
           Types:          < Automobiles >, < Boats >
           Parents:       < Automobiles >, < Boats >

     View Id: < Vjim >
        Is-A Connections:
           Types:          < Automobiles >, < Boats >
           Parents:       < Automobiles >, < Boats >
```

Figure 29

The Attach Method

Applied by the Cars and Chevys (Internal)

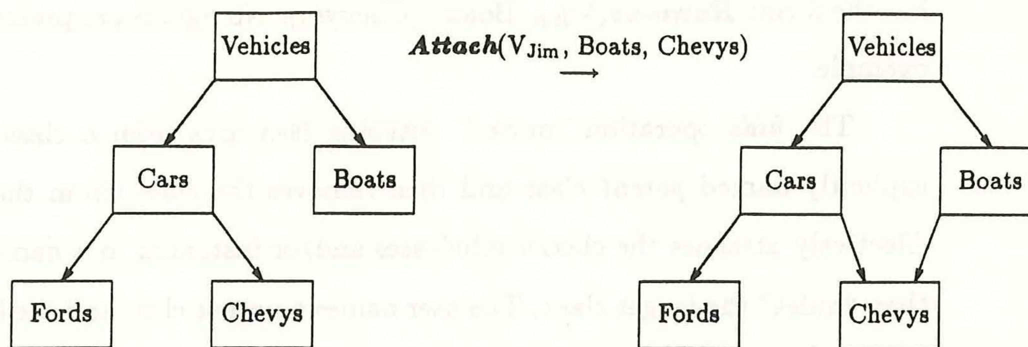


Figure 30

The Attach Method

Applied by the Boats and Chevys (External)

user specifies the target class and a list of children (to be removed). The view's identity is provided by the system. ***Remove***($View_{Id}$, Target, (Children)) is the generic form of the *remove* message. When a class object receives a *remove* message, it sends *remove-parent* messages to each of the named children and removes its IS-A connection with each of them (within the current *view description* only). Because the *instance* objects are directly connected to exactly one class object, *PolyView* does not explicitly forward the *remove-parent* message into the database — all instances connected to a "removed" class are effectively invisible. These *remove-parent* messages are necessary because the *remove* operation only eliminates IS-A connections from a view. Since *PolyView* classes may have more than one parent, removing an IS-A arc makes a class unreachable from one branch of the IS-A lattice but does not necessarily cause it to be completely removed from the user view. Therefore, the remove transformation *sometimes* causes the global symbol table to be changed.

Returning once again to the example, if Chevys are now removed by Boats, the schema will revert to its form before the attach message was sent. The message

has the form: ***Remove***(V_{Jim}, Boats, (Chevys)). No figures are presented with this example.

The *hide* operation "moves" outgoing IS-A arcs from a class object to an explicitly named *parent* class and then removes the class (from the view). This effectively attaches the class's subclasses and/or instances to a named parent and then "hides" the target class. The user names a parent class and the list of children which are to be moved. This latter may contain the special identifier *instances* which indicates that the target class's instances are to be moved to its parent, as well. There will usually be far too many instances in a database to make naming them explicitly feasible; therefore, the view designer can either hide all instances or none of them. When a class receives a hide message, it creates a new view description if necessary, changes its external view name to *invisible* and removes all but the named subclasses from its from its subclass IS-A list. Finally, if instances are NOT requested then the instances IS-A list is replaced with *nil*. Although externally the changes are significant, internally only the target class is affected.

Figures 31 and 32 show the internal and external state of the schema before and after the `Cars` class has performed the *hide* transformation. This particular *hide* operation moves `Cars`' children (`Chevys` and `Fords`) to its parent (`Vehicles`). The hide operation also affects the global symbol table. If the view was colored immediately after this operation had been performed, the `Cars` class would be removed from the global symbol table (see figure 33).

Redefining Attributes in a Decomposition Hierarchy

View customizing methods performed by classes on their attribute relationships are similar to those performed in the class hierarchy. There are methods for renaming, creating, restricting, removing and moving attributes. A major difference between these two categories of transformation is that the attribute customizing transformations do not affect the global symbol table. This is because

BEFORE:
Object Automobiles:
Object Description:
Class-Object-Id: < Automobiles >

View Description:
View Id: < Vjim >
Names: Cars, Autos
Is-A Connections:

SubClasses: < Chevys >, < Fords >
Instances: < list of instances >

AFTER:

Object Automobiles:
Object Description:
Class-Object-Id: < Automobiles >

View Description:
View Id: < Vjim >
Names: ***invisible***
Is-A Connections:

SubClasses: < Chevys >, < Fords >
Instances: < list of instances >

Figure 31

The Hide Method

Applied by the Cars Class (An Internal Snapshot)

changes to classes (like, for example, hiding and renaming) must be "published" in a view model object table while attribute descriptions are encapsulated in object descriptions. The five attribute transformation methods are called A-NAME[11], A-REMOVE, A-RESTRICT, A-INSERT and A-MOVE.

The *a-name* method is used to rename an attribute. Old and new external names for the attribute are supplied by the user and *PolyView* identifies the view (***A-Name***($View_{Id}$, old-name, new-name)). The new external name is associated with the attribute identity, the user's view instance description.

[11] The **a-** prefix denotes a method which customizes an attribute (**a-** methods may be applied to *both* instance and class attributes).

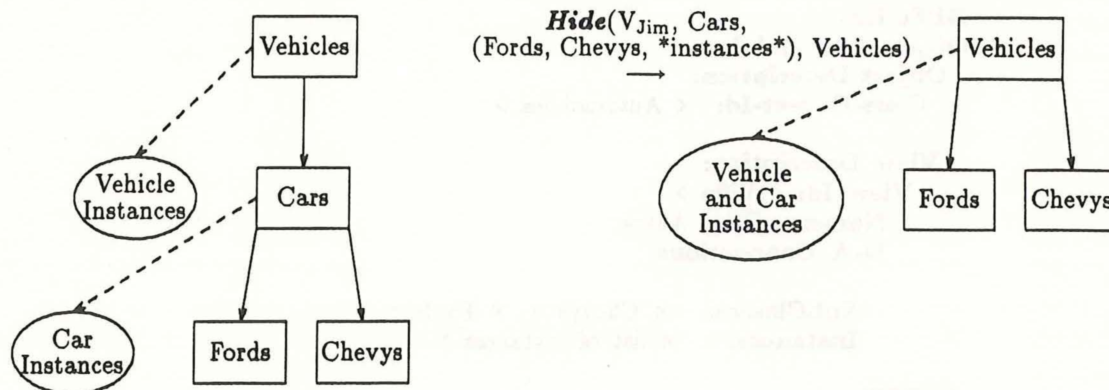


Figure 32

The Hide Method

Applied by the Cars Class (An External Snapshot)

BEFORE: — see AFTER from previous example

AFTER:

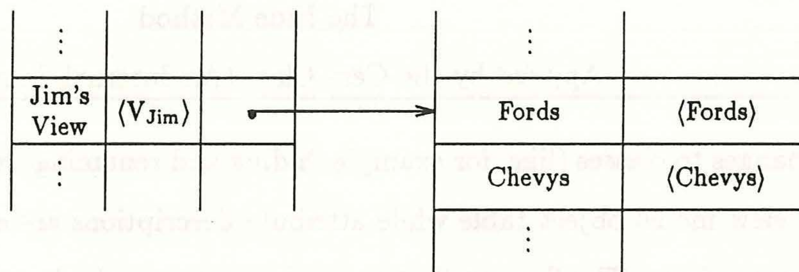


Figure 33

The Effect of Sending a Hide Message to the Cars Class

on the Global Symbol Table

For example, suppose that there is an attribute called `position` which associates an employee with a job description. If the executive vice president believes that `job` would be a more appropriate name for this relationship then the a-name

```
BEFORE:

Object Employees:
  Object Description:
    Class-Object-Id: < Employees >
     Instance Attributes:
        < position >:  < Job Description >
        < salary >:    < $ >
              View Description:
     View Id: < Vjim >
       position:  < position >
       salary:    < salary >
     View Id: < Vvp >
       position:  < position >
       salary:    < salary >
AFTER:

Object Employees:
  Object Description:
    Class-Object-Id:  < Employees >
    Instance Attributes:
        < position >:  < Job Description >
        < salary >:    < $ >
  View Description:
    View Id: < Vjim >
      position:  < position >
      salary:    < salary >
    View Id: < Vvp >
      job:        < position >
      salary:    < salary >
```

Figure 34

The A-Name Method

(Applied by Employees to ⟨position⟩ — Internal)

transformation could be used to create this preference. Figures 34 and 35 show the internal and external schemas before and after the `Employee` class object has executed the a-name method.

A-Remove hides attributes by removing them from the user's personal list of attributes. The method is passed a list of attributes (to be hidden) and the color of the view. The syntax of the a-remove is: ***A-Remove***(View$_{Id}$, (attribute-list)).

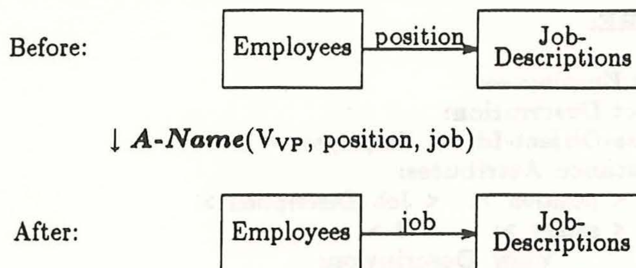


Figure 35

The A-Name Method

(Applied by Employees to ⟨position⟩ — External)

BEFORE: */* see AFTER from previous example */*

AFTER:

Object Employees:
Object Description:
Class-Object-Id: < Employees >

Instance Attributes:

< position >: < Job Description >
< salary >: < $ >

View Description:
View Id: < Vjim >

position: < position >
/* < salary > *has now been removed* */

View Id: < Vvp >

Names: Employees

job-title: < position >
salary: < salary >

Figure 36

The A-Remove Method

(Applied by Employees to ⟨salary⟩ — Internal)

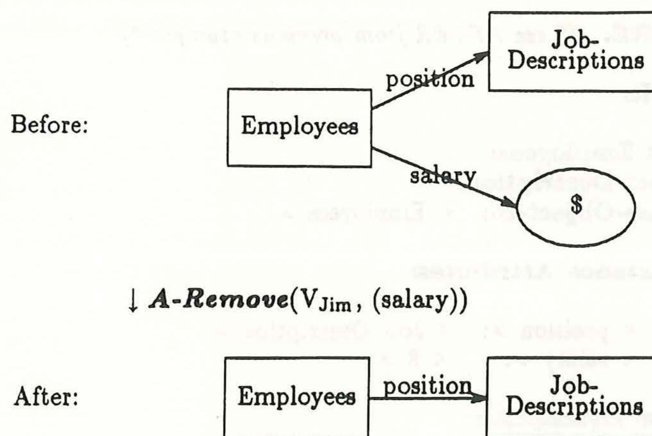


Figure 37

The A-Remove Method

(Applied by Employees to ⟨salary⟩ — External)

For example, if employees' salaries are confidential then it would be appropriate to hide them from Jim. (See figures 36 and 37.)

Sometimes it is necessary to restrict the range of an attribute; to do this the designer uses the *a-restrict* method. When applied to a particular attribute, a-restrict places a restriction on that attribute which will be combined with all subquery messages that are sent along the a-restricted arc. The user supplies the restriction in the form of a binary predicate which explicitly names the attribute: ***A-Restrict***(View$_{Id}$, binary-predicate). When a class receives an a-restrict message, it first checks that the predicate identifies a subset of the original attribute's range. Next, if the named attribute is not already part of the current *view* description, a-restrict adds its internal and external names to the appropriate view instance description. Finally, the binary predicate (constraint) is appended as a suffix to the external/internal name pair in the view description. Now, when a query through the current view, it is restricted by the explicit constraints in the view description. Note that if a query that contains a restriction on a restricted

BEFORE: */* see AFTER from previous example */*

AFTER:

Object Employees:
Object Description:
Class-Object-Id: < Employees >

Instance Attributes:

< position >: < Job Description >
< salary >: < $ >

View Description:
View Id: < Vjim >

position: < position >

View Id: < Vvp >

Names: Overpaid Employees

job-title: < position >
salary: < salary >, > 1000

Figure 38

The A-Restrict Method

(Applied by Employees to ⟨salary⟩ — Internal)

attribute then the logical conjunction of the two restrictions is used to constrain the query.

Suppose that the aforementioned executive was only interested in the salaries of employees who make more that $1,000.00 per week. Figures 38 and 39 demonstrate how a sequence of transformation methods (name and a-restrict) could be applied to form a new (view) schema which contains the overpaid employees[12].

Creating a new attribute relationship between two classes (using the *a-insert* method) is slightly more complex than the previously described operations for two

[12] Even though the name method changes the global symbol table the a-restrict method does not, therefore, the global symbol table is not shown.

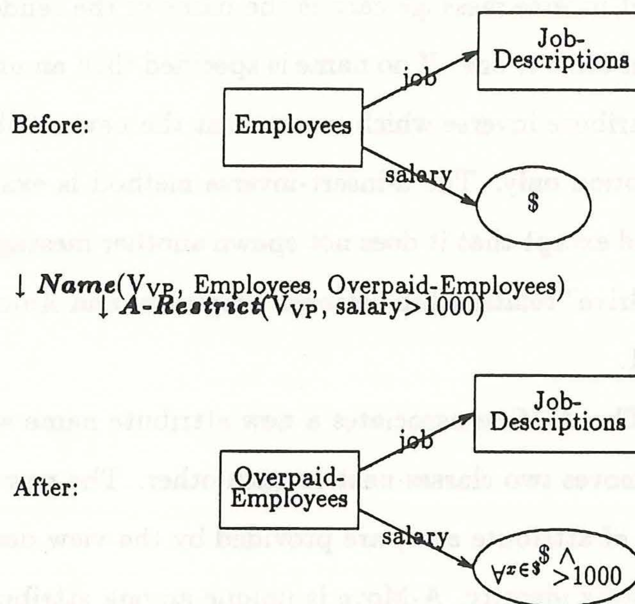


Figure 39
The A-Restrict Method
(Applied by Employees to ⟨salary⟩ — External)

reasons. Firstly, it involves two classes and secondly, it causes changes to both class descriptions as well as their view descriptions.

Both classes must exist somewhere in the schema; if a new relationship is to be established between an existing class a new one then the clone method must be used to create the new class prior to sending the a-insert message. The user is responsible for supplying external names for the new attribute and its inverse (if there is one) and for identifying both classes involved in the new relationship. *PolyView* generates a unique internal name for the attribute and passes the view's color to the a-insert message. The syntax for a-insert is ***A-insert***(View$_{Id}$, Class$_1$, new-attribute, inverse-name, Class$_2$). This transformation does not specify any instances of the new function; it simply declares a new relationship. The message is sent to Class$_1$ which records the new attribute in *both* its object and view descriptions. Simultaneously, an a-insert-inverse message is sent to Class$_2$. An

a-insert-inverse message carries the name of the sender and the (inverse) property name if there is one. If no name is specified then an unnamed arc is associated with the attribute inverse which means that the new attribute is recorded in the object description only. The a-insert-inverse method is exactly the same as the a-insert method except that it does not spawn another message. The effect of a-inserting a "can-drive" relationship between *Employees* and *Automobiles* is shown in figures 40 and 41.

The *A-Move* associates a new attribute name with an existing ***attribute path*** — it moves two classes next to each other. The new attribute name and the path (a list of attribute arcs) are provided by the view designer and *PolyView* provides the view's identity. A-Move is unique among attribute redefining transformations because it is spread (recursively) through the classes on the path between the "a-moved" classes. The a-move method distributes its side effects in order to minimize violating the principle of encapsulation which is salient to the object-oriented paradigm. When a class receives an a-move message, it removes the first attribute name from the path list and finds the local attribute which corresponds to that name. If a corresponding local attribute cannot be found then the a-move transformation must be aborted and recursively undone. Otherwise, the new attribute name is associated with that attribute. The new attribute name is "marked", to indicate that it is visible only in the context of the a-moved attribute. If the new path is not empty, an a-move message is sent along that attribute's arc. A-move has the following syntax: ***A-Move***(View$_{Id}$, new-attribute, path).

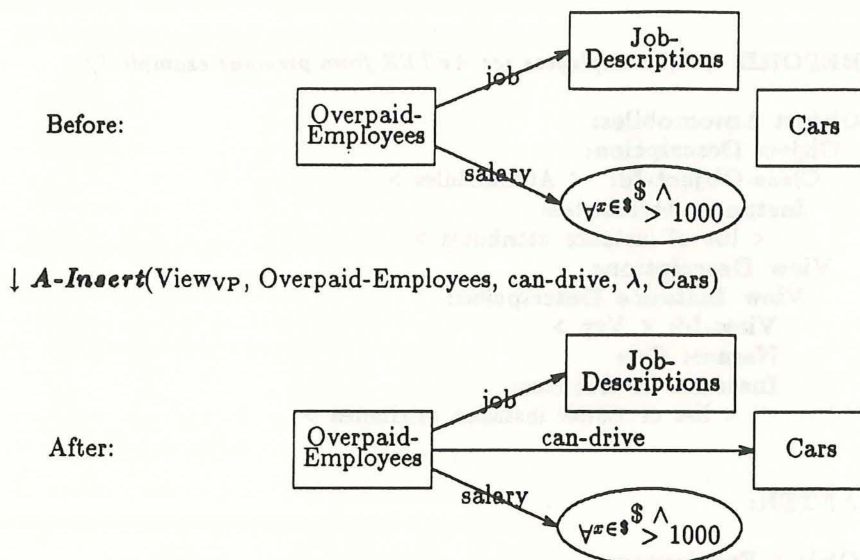
For example, consider the `position` attribute which is associated with a complex object called `Job-Descriptions` in the employee example. A job description has several properties including a job title and a description of the job. If the foreman (Jim) wants the job title to be directly associated each employee then an a-move message would be sent to the Employees class. Figures 42 and 43 show

```
BEFORE: /* for employees see AFTER from previous example */

Object Automobiles:
  Object Description:
    Class-Object-Id:  < Automobiles >
     Instance Attributes:
         < list of instance attributes >
   View Descriptions
     View Instance Description:
       View-Id: < Vvp >
       Names: Cars
       Instance Attributes:
           < list of visible instance attributes >

AFTER:

Object Employees:
  Object Description:
    Class-Object-Id:  < Employees >
    Instance Attributes:
        < position >:   < Job Description >
        < salary >:     < $ >
        < can-drive >:  < Automobiles >
    View Id: < Vjim >

    View Id: < Vvp >
      Names:    Overpaid Employees
      job:          < position >
      salary:      < salary >, > 1000
      can-drive:    < can-drive>

Object Automobiles:
  Object Description:
    Class-Object-Id:  < Automobiles >
    Instance Attributes:
        < list of instance attributes > + inverse of  < can-drive >
  View Descriptions
    View Instance Description:
      View-Id: < Vvp >
      Names: Cars
      Instance Attributes:
          < list of visible instance attributes >  /* note that this is unchanged */
```

Figure 40

The A-Insert Method

(Applied by Employees and Cars — Internal)



Note: λ **represents the empty string.**

Figure 41

The A-Insert Method

(Applied by Employees and Cars — External)

the schema's internal representation and the user's view, respectively. Notice that "marked" attribute names in figure 42 begin with an asterisk.

Applying Sequences of Transformations

The view transformation methods which were presented in the previous sections may be *almost* arbitrarily combined in order to form valid higher level transformations. In other words, given any valid *PolyView* view, apply any *valid* sequence of transformations and the result will be another *PolyView* view.

There are two issues which naturally arise in the context of the discussion of sequences of transformations. The first pertains to what a "valid" view is and what effect an arbitrary transformation has on a valid view. It will be shown that applying any transformation to a valid view always results in another valid view.

BEFORE:

Object Employees:
Object Description:
Class-Object-Id: < Employees >
View Descriptions
View Id: < Vjim >
Names: Employees
Instance Attributes:
position: < position >

Object Job Description:
Object Description:
Class-Object-Id: < Job Description >
< job-title >: < String >
View Descriptions
View-Id: < Vjim >
Instance Attributes:
< list of *visible* instance attributes >

AFTER:

Object Employees:
Object Description:
Class-Object-Id: < Employees >
View Descriptions
View Id: < Vjim >
Names: Employees
Instance Attributes:
position: < position >
job-title: < position >

Object Job Description:
Object Description:
Class-Object-Id: < Job Description >
< job-title >: < String >
View Descriptions
View-Id: < Vjim >
Instance Attributes:
< list of *visible* instance attributes >
*job-title: < job-title >

Figure 42

The A-Move Method

(Applied by Employees and Job Descriptions — Internal)

The second issue deals more directly with sequences of transformations, specifically which sequences cannot be applied.

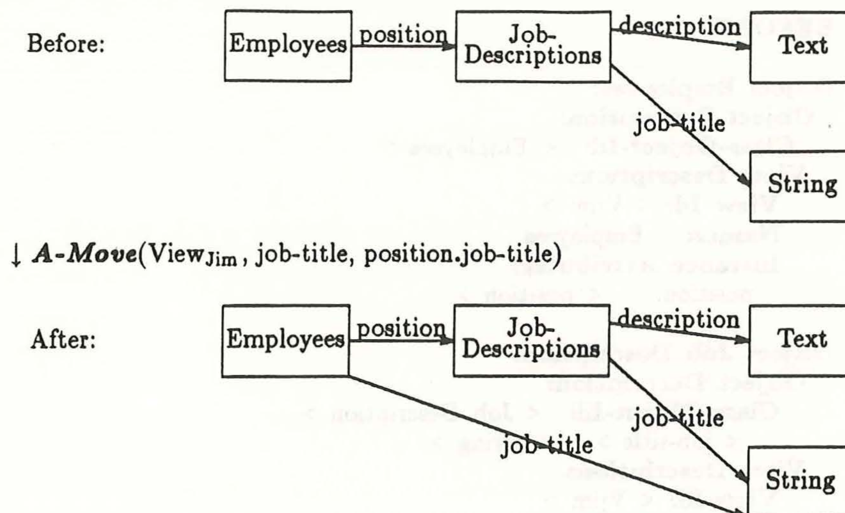


Figure 43

The A-Move Method

(Applied by Employees and Job Descriptions — External)

In *PolyView*, a valid view of an object is represented by a view instance description. Each view instance description (view) is a subset of the global view instance description (global view). To clarify this relationship, recall that the global view contains *pointers to all* of the underlying data structures and methods (attributes) in the class object description while each view contains *pointers to some* of the attributes. Therefore, for any "valid view" the following relationship will hold: {pointers in view} $\subseteq$ {pointers in global} which will be abbreviated to view $\subseteq$ global.

PolyView transformations fall into four major categories: $\mathbf{C}_{\mathbf{hide}}$: those which hide part of a schema (like a-remove and hide), $\mathbf{C}_{\mathbf{name}}$: those which rename objects or attributes (like name and a-name), $\mathbf{C}_{\mathbf{rest}}$: those which restrict the objects which an attribute refers to (like a-restrict), and $\mathbf{C}_{\mathbf{add}}$: those which add an attribute or object to the schema (like attach and a-insert). In the discussion which follows, it

is assumed that the pre-transformed view is valid and that $\mathbf{T}_i$ represents a generic transformation from category $\mathbf{C}_i$.

$\mathbf{C}_{hide}$ These transformations do not have any direct effect on the global view or the object description. Since they remove a reference from a view description, the following relationship holds $T_{hide}(view) \subset view \Rightarrow T_{hide}(view) \subseteq global$.

$\mathbf{C}_{name}$ This kind of transformation has no effect at all on the pointers which the view description contains. These operations replace the symbol which will be used to refer to a pointer; therefore, $view \subseteq global \Rightarrow T_{name}(view) \subseteq global$.

$\mathbf{C}_{rest}$ Like the transformations in $\mathbf{C}_{name}$ these transformations do not change any pointers. They simply restrict the information which can be retrieved along a transformed arc.

$\mathbf{C}_{add}$ These transformations *do* change both the global view and the object description. What will be shown here is that not only does the transformation result in a valid view but, since it also changes the global view, that it does not invalidate any other existing views. $\mathbf{C}_{add}$ transformations add a new attribute or is-a relationship to an existing object description. New pointers are added to the view performing the transformation *and* to the global view. Therefore, after the operation has been performed the following relationships hold: $T_{add}(view) \subseteq T_{add}(global)$ and $global \subset T_{add}(global) \Rightarrow \forall_{view_i \subseteq global}$ $view_i \subseteq T_{add}(global)$. In other words, all views which were valid before the transformation was applied are also valid afterwards.

It has been shown that all transformations produce valid schemas, but some of them hide or change the view so that completely arbitrary sequences of transformations are not possible. The exceptions are obvious and can be checked by a purely syntactic mechanism. Transformations cannot refer to classes or attributes which have been hidden from the view nor can they refer to the old name of a

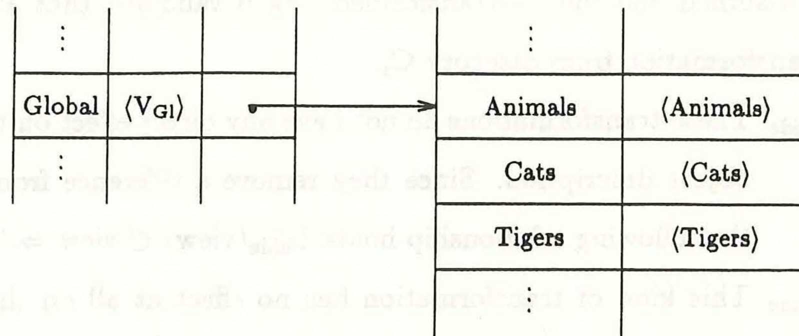


Figure 44

The Global Symbol Table

for the Simple Animal Taxonomy

renamed class or attribute. In each of these cases, an attempt would have been made to refer to some aspect of the view which no longer exists.

Supporting Higher Level Transformations

The transformations methods presented so far directly change a single aspect of the structure of a schema and may indirectly affect some operational aspects of the database. It has been shown that sequences of these methods can be used to define higher level abstractions. Combinations of transformations might, for example, change the shape of the IS-A hierarchy by creating new specializations and generalizations from existing classes. The goal is *not* to present *the* complete set of semantic operations but to show that the methods presented in this thesis can be used to define higher level operations in a straightforward manner.

Class Collapsing Transformations

Each of the methods described in this subsection combines a class with one of its ancestors and/or one of its descendants. They are referred to collectively as *class collapsing* transformations.

The class **Animals** is not shown because it is not significantly changed by the *collapse* transformations.

```
Object Cats:
  Object Description:
    Class-Object-Id:  < Cats >

  View Description:
    View Id:      < Vglobal >
    Is-A Connections:
      SubClasses:    < Tigers >
      Instances:     < list of instances >

Object Tigers:
  Object Description:
    Class-Object-Id:  < Tigers >
      Instance Attributes:
        < name >    :  string
        < weight >  :  fixed     /* in kilos */
        < length >  :  fixed     /* in feet  */
        < dob >     :  date      /* month/day/year */
        < diet >    :  < Diet >

  View Description:
    View Id:      < Vglobal >
      Instance Attributes:      < list of attributes *including* Cat attributes >
           name            : < name >
           weight          : < weight >
           length          : < length >
           date-of-birth   : < dob >
           diet            : < diet >
```

Figure 45

The Internal Structure

(Before Creation of the New Views)

For pedagogic ease, we begin by considering simple concrete examples of class collapsing. Consider figures 44, 45 and 46; they refer to the global symbol table, internal and external representations of part of a very simple taxonomy of animals. Two quite different points of view will be considered.

The first is from the perspective of a three year old child. He has a general idea about what an animal is and he "knows" what tigers are, but he has never seen a cat. When the three year old sees `Tom` and `Kitty` he realizes that they

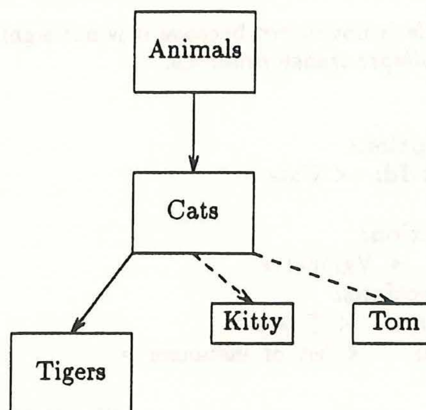


Figure 46

The External Structure

(the Simple Animal Taxonomy)

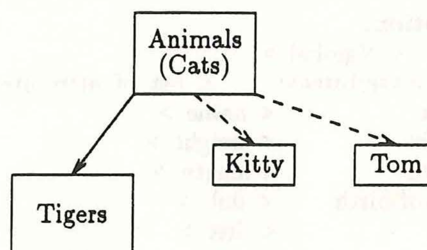


Figure 47

The Three Year Old's Point of View.

are animals and not tigers. An external representation of the three year old's point of view (after his confrontation with `Tom` and `Kitty`) is shown in figure 47. In this case, all database instances and subclasses IS-A connected to `Cats` are "moved" to `Animals` and all attributes of `Cats` which were *not* inherited from `Animals` "disappear". This operation (applied by `Cats`) is a combination of the HIDE method and the A-REMOVE method. The HIDE method is applied with all subclasses and *instances* in it's argument list. The A-REMOVE method is applied to all attributes which were *added* by `Cats`.

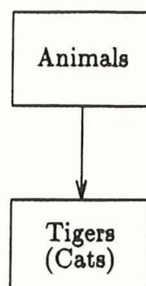


Figure 48

The Zoo Keeper's Point of View.

A second point of view is that of the tiger keeper. She is very knowledgeable about cats in general (i.e. she knows that they eat meat) and tigers in particular (i.e. that they might eat people). From her (professional) point of view, information about cats is only useful when it is combined with information about tigers. In this view of the world, information about (non-tiger) cats, the class itself and individuals like `Tom` and `Kitty`, is hidden. Figure 48 shows external schema after it has been transformed to the zoo keeper's point of view. Notice that this is again the `HIDE` method applied by Cats with a single subclass (Tigers) as its argument.

It is significant that although these two view schemas are derived from the same database using similar derivation methods, their attribute structures are *very* different. Figure 49 shows the global symbol table after the two views have been formed. As expected, the "global" picture (i.e. the classes) of both the three year old and zoo keeper views are identical. However, when the internal structures of the objects are considered (see figure 50), it is obvious that the views have significant differences. From the three year old's point of view, the `Cats` class and all the properties associated with cats just do not exist (they are "invisible" in his view) but the individual instances of that class are undeniably real. From the zoo keeper's perspective, individual cats are irrelevant but all the properties of the `Cats` class are necessary and must be retained. The first case (the rule by which

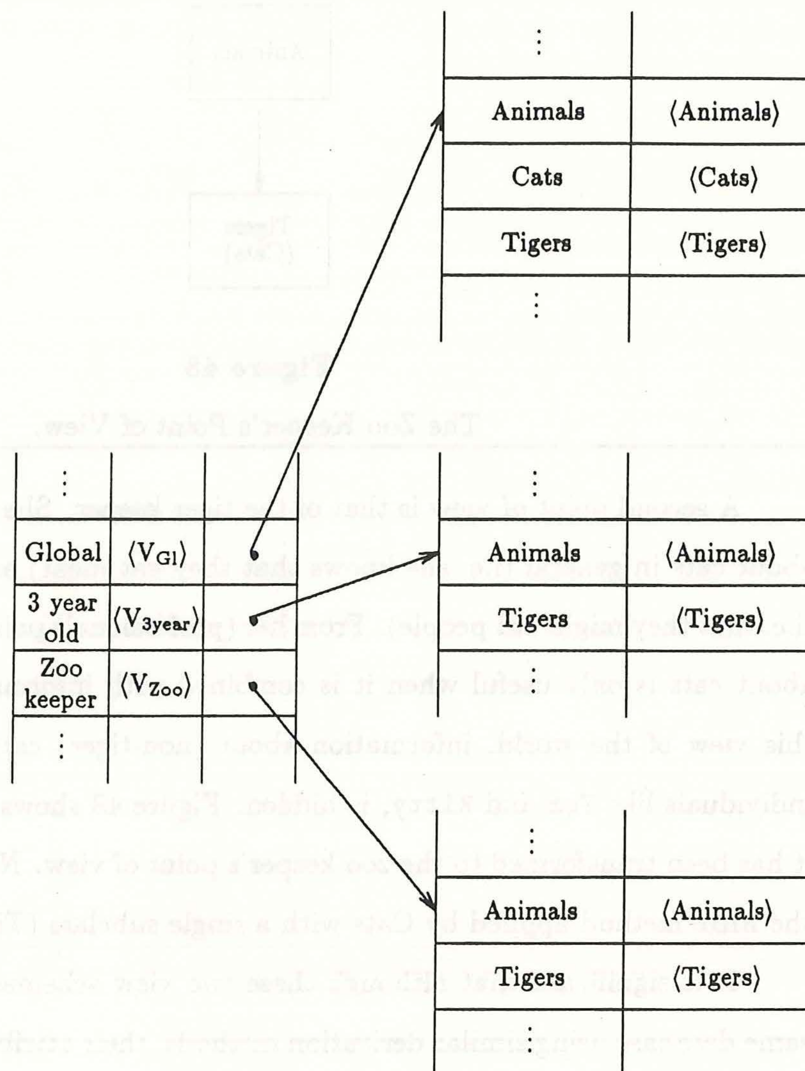


Figure 49

The Global Symbol Table

(After Creation of the New Views)

the three year old's point of view was formed) is called collapsing a class *up* into a parent and the latter case will be referred to as collapsing a class *down* into a child.

The general forms of these two abstractions are: ***Collapse-Up***(View$_{Id}$, Target, Parent) and ***Collapse-Down***(View$_{Id}$, Target, Parent, Child). Notice that

the "collapse down" operation requires a reference to a parent as well as a child. This is necessary because *PolyView* supports a multiple inheritance structure. This simple example is sufficient to illustrate why class collapsing abstractions are desirable. The fact that *PolyView* can support both the child's and the keeper's points of view, *without* introducing any unnecessary redundancy into the *data*, makes it superior to any other view support system.

The collapsing abstractions can easily be generalized; for example, a method which performs both collapsing transformations on an entire *path* can easily be defined. This COLLAPSE method would iteratively collapse the two ends of a path into a target class. The user would specify the *target* and *paths* to *ancestor* and *descendant* classes. *PolyView* would iteratively collapse the path *down* from the ancestor and then *up* from the descendent. The iterative collapse operation is shown in figures 51, 52 and 53. Notice that figure 52 is slightly different from other figures which show an internal representation. Since it represents many objects, it attempts to show in a general way how the structure of these objects would have changed.

Using Independent Views — An Example

In this section, we return again to the two views of tigers. For the purposes of this discuss it will be assumed that Tom and Kitty are tigers and not domestic cats. The external view descriptions of the individual tigers will be presented in an expanded form so that query and update processing can be demonstrated using these two independent views of a single database representation. Information will be retrieved from both views using a similar query. Several update requests which affect one or both of the views will be presented. In figure 54, external views of Tom and Kitty are presented.

After each sample operation has been performed, the result of the query "Display all tigers" will be presented from both *users* (the zoo keeper and the

```
Object Cats:
  Object Description:
    Class-Object-Id:  < Cats >
  View Description:
    View Id:     < Vglobal >
      Is-A Connections:
        SubClasses:     < Tigers >
        Instances:      < list of instances >
    View Id:     < V3year >
      Is-A Connections:
        SubClasses:     < Tigers >
        Instances:      < list of instances >
      Instance Attributes: /* Attributes defined here for first time have been removed */
    View Id:     < Vzoo >
      Is-A Connections:
        SubClasses:     < Tigers >
        Instances:      *nil*
      Instance Attributes: /* unchanged */

Object Tigers:
  Object Description:
    Class-Object-Id:  < Tigers >
  View Description:
    View Id:     < Vglobal > /* unchanged */
    View Id:     < V3year >
      Instance Attributes:      < list of attributes *excluding* Cat attributes >
      /* this is an abbreviated notation - view definitions contain */
      /* labeled pointers to methods and not the methods themselves */
        name   : < name >
        size   : if (< length > greater than 3) then big else small
        weight : case (< weight > less than 50)   : light
                      (< weight > greater than 100) : very heavy
                      (default)                     : heavy
    View Id:     < Vzoo >
      Instance Attributes:      < list of attributes *including* Cat attributes >
           name     : < name >
           weight   : < weight > * 2.2
           age      : < date-today > - < dob >
           length   : < length >
           diet     : < diet >
```

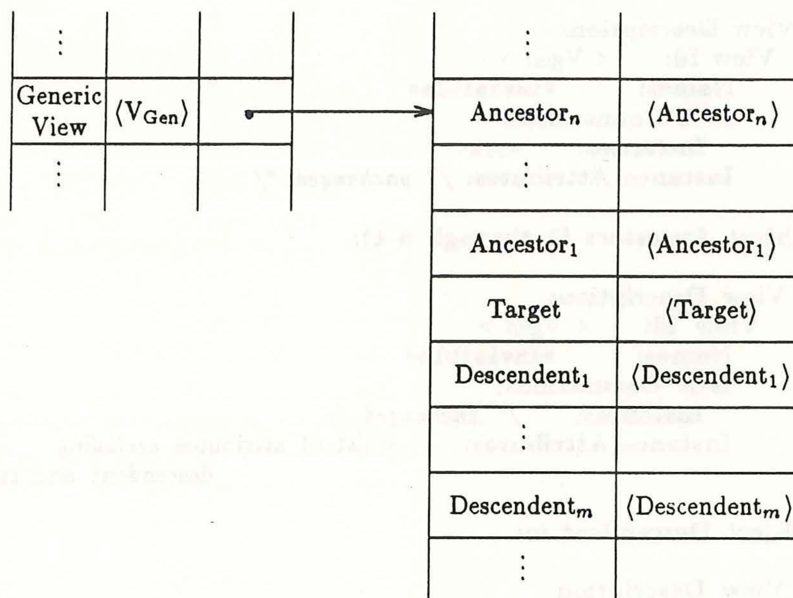
Figure 50

The Internal Structure

(After Creation of the New Views)

three year old) points of view. For ease of presentation the results of these queries will displayed in tabular form. The result of the query on the initial database is shown below:

BEFORE:



AFTER:

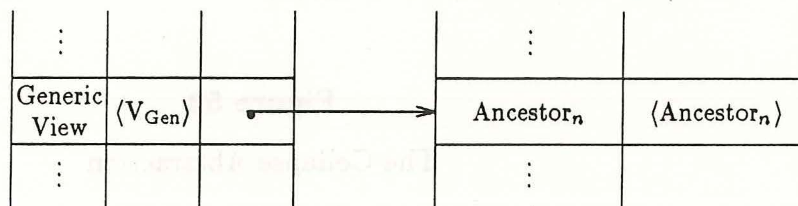


Figure 51

The Collapse Abstraction

(A Global Symbol Table)

From the Zoo Keeper's Point of View:

Name	*Length in feet*	*Age*	*Weight in pounds*	*Meat to Cereal*	*Quantity in pounds*
Kitty	6	10	330	7:3	20
Tom	3	1	88	1:1	7

Object Target & Descendents (1 through m-1):

View Description:
View Id: < Vgen >
Names: ***invisible***
Is-A Connections:
Instances: ***nil***
Instance Attributes: /* *unchanged* */

Object Ancestors (1 through n-1):

View Description:
View Id: < Vgen >
Names: ***invisible***
Is-A Connections:
Instances: /* *unchanged* */
Instance Attributes: < list of attributes *excluding* descendent and target attributes >

Object Descendent m:

View Description:
View Id: < Vgen >
Names: ***invisible***
Is-A Connections:
Instances: /* *unchanged* */
Instance Attributes: /* *unchanged* */

Figure 52

The Collapse Abstraction

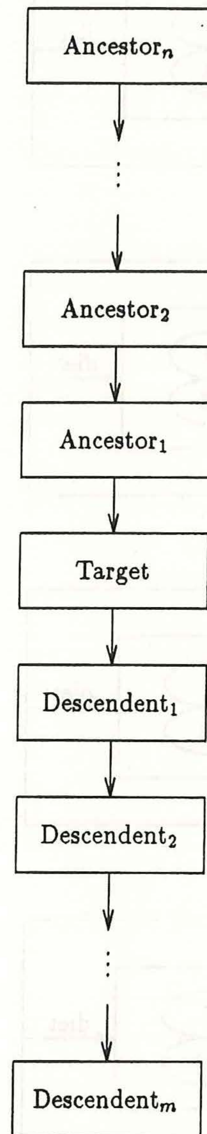
(The "After" Internal View)

From the Three Year Old's Point of View:

Name	*Size*	*Weight*
Kitty	big	very heavy
Tom	small	light

Because of its greater level of detail many changes can be made to the zoo keeper view without affecting the other view. For example, Tom's weight can be changed to 45 kilos and the quantity of food consumed by Kitty may be increased from 20lbs to 22lbs. After these changes have been made the three year old's view is clearly unchanged because 45 kilos is still less than 50 kilos (so Tom is still small)

BEFORE: AFTER:



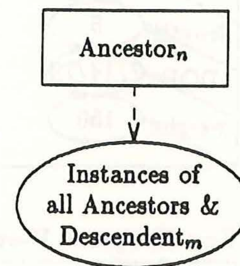


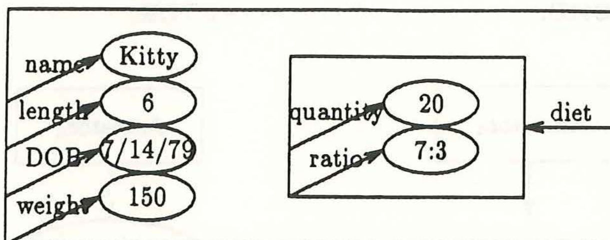
Figure 53

The Collapse Abstraction

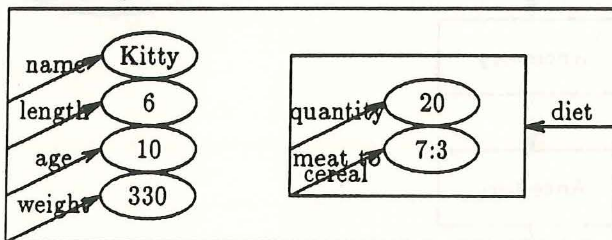
(An External View)

and the tigers' diet is not part of that view at all. The results from the sample query, after these changes have been made, is shown below:

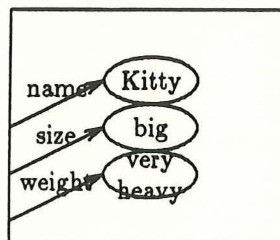
Global View



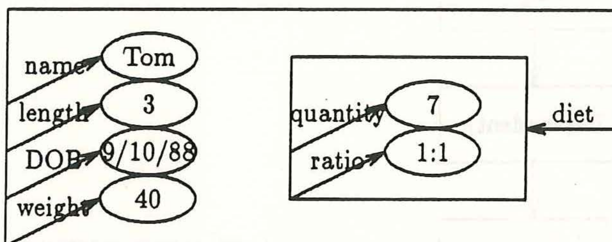
Zoo Keeper's View



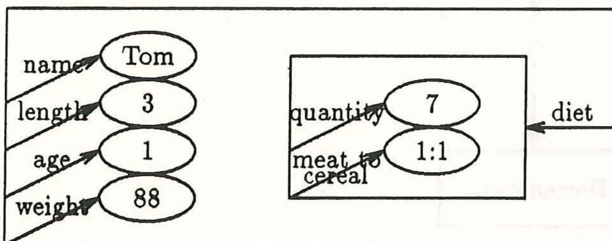
3 Year Old's View



Global View



Zoo Keeper's View



3 Year Old's View

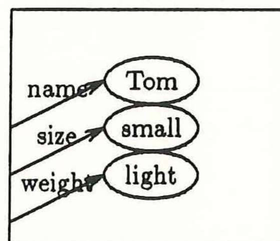


Figure 54

A More Detailed Look at Tom and Kitty

Zoo Keeper (changes shown in *italic* font):

Name	*Length in feet*	*Age*	*Weight in pounds*	*Meat to Cereal*	*Quantity in pounds*
Kitty	6	10	330	7:3	*22*
Tom	3	1	*99*	1:1	7

Three Year Old (unchanged):

Name	*Size*	*Weight*
Kitty	big	very heavy
Tom	small	light

Some changes will be reflected in both views. For example, if Tom's weight increases from 45 kilos to 55 kilos then, from the three year old's point of view the tiger is no longer "light". Adding a new tiger or removing an existing one would also change both views so, for example, if Kitty was replaced by a younger, larger, hungrier animal called Hobbs both users would immediately notice the change. If a new database which reflected the above changes was sent the "Display all tigers" query, the result would look like this:

Zoo Keeper (changes shown in *italic* font):

Name	*Length in feet*	*Age*	*Weight in pounds*	*Meat to Cereal*	*Quantity in pounds*
Tom	3	1	*121*	1:1	7
Hobbs	*6.5*	*7*	*440*	*3:1*	*28*

Three Year Old (changes shown in *italic* font):

Name	*Size*	*Weight*
Tom	small	*heavy*
Hobbs	*big*	*very heavy*

CHAPTER 6
Concluding Remarks

Summary

This dissertation has presented a data model capable of supporting many different user perspectives. As the computer user population continues to grow and becomes more diversified, we believe that all information systems will have to provide mechanisms which support different users' needs and preferences. *PolyView* is the first step towards this ultimate goal.

In *PolyView*, user views are supported through a number of unique and innovative features:

- Objects have a unique internal structure which allows a single data structure (object description) to have many independent user interfaces (view instance descriptions).
- The concept of object identity has been extended to include both attributes and views. By associating a time invariant identity with each attribute, it is possible to avoid some problems associated with multiple inheritance in an IS-A lattice and to support different external (printable) names for each attribute for each user. By associating an identity with a view, it is potentially possible to allow users to change from one environment to another by asking the system to allow them to use a different view.
- Generic procedures permit queries to be processed and changes to be made to the data in a purely message-driven manner. Since a database is represented as a network of nodes and arcs in which each node is capable of

communicating with other nodes by exchanging messages, no centralized control or shared memory is necessary.

- Finally, transformation rules which facilitate the controlled customization of *PolyView* schemas have been presented. If these rules are used to create or change user views then the new view is guaranteed to be a consistent view.

Directions for Future Work

There are several possible directions for future research; in this section some of the more significant ones will be suggested.

In order to better understand how well the various algorithms will perform, it will be necessary to implement the system described in this thesis. Since *PolyView* uses a purely message-driven paradigm, it will also be constructive to investigate multiprocessor architectures and implement (or simulate) the *PolyView* algorithms on the most promising of those.

There are a number of issues dealing with the management of information stored on secondary storage devices which warrant research. For example, how data should be clustered in order to minimize the frequency of access to secondary storage. This can be further improved by query optimization techniques which effectively predict what should be retrieved without traversing a path of objects in secondary storage. It will also be necessary to extend the query processing capabilities of *PolyView* in order to support transactions and concurrency control.

Finally, there is the issue of user/system interfaces[13]. This is what is actually presented to the user, not the underlying data structures which have been the topic of this dissertation. Without support for different kinds of "user friendly" interface, it ultimately will not matter how good the underlying data model is. A system

[13] These interfaces are sometimes also called user views.

which is easy to learn and pleasant to use will be fully utilized by users of many different backgrounds and skill levels. The object-oriented paradigm provides an excellent foundation on which multimedia interfaces can be built. Therefore, the development of attractive user environments is a natural extension of the current model.

References

[ABIDA81] ABIDA, M. Derived Relations: A Unified Mechanism for Views, Snapshots and Distributed Data. In *Proc. Seventh Int'l. Conf. on Very Large Data Bases*, ACM, 1981, pp. 293–305.

[ABITEBOUL87] ABITEBOUL, S. AND HULL, R. IFO: A Formal Semantic Database Model. *ACM Trans. on Database Systems 12*, 4 (Dec., 1987), 525-565.

[ARVIND78] ARVIND, GOSTELOW, K.P. AND PLOUFFE, W. An Asynchronous Programming Language and Computing Machine. In *Advances in Computing Science and Technology*, Yeh, R., Ed., 1978.

[ATTARDI86] ATTARDI, G. AND SIMI M. A Description-Oriented Logic for Building Knowledge Bases. *Proc. of the IEEE 74*, 10 (Oct., 1986), 1335–1344.

[BACKUS78] BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.. *Comm. ACM 21*, 8 (Aug., 1978), 613-641.

[BANERJEE87A] BANERJEE, J., CHOU, H., GARZA, J., KIM, W., WOELK, D., BALLOU, N. AND KIM, H. Data Model Issues for Object-Oriented Applications. *ACM Trans. on Office Information Systems 5*, 1 (Jan., 1987), 3-26.

[BANERJEE87B] BANERJEE, J., KOM, W., KIM, H-J. AND KORTH, H. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *Proceedings SIGMOD'87* (1987), 311–322, ACM.

[BIC86] BIC L. AND GILBERT J.P. Learning from AI: New Trends in Database Technology. *Computer 19*, 3 (Mar., 1986), 44–54.

[BOBROW85] BOBROW, D.G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M. AND ZDYBEL, F. CommonLoops: Merging Common Lisp and Object-Oriented Programming. Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC, Palo Alto, CA.

[BRACH83] BRACHMAN, R.J. What IS-A Is and Isn't An Analysis of Taxonomic links in Semantic Networks. *Computer 16*, 10 (Oct., 1983), 30–36.

[Brodie80] *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, Brodie, M.L. and Zilles, S.N, Ed., Sponsored by the Nat'l. Bureau of Standards, ACM SIGART, SIGMOD and SIGPLAN, Pingree Park, Colordo, 1980.

[Brodie81] Brodie M.L. and Schmidt J.W. *Final Rep. of ANSI/X3/SPARC DBS-SG Relational Database Task Group*, SPARC-81-690, 1981.

[Brodie84] *On Conceptual Modelling*, Brodie, M.L., Mylopoulos, J. and Schmidt, J.W., Ed., Springer-Verlag, 1984.

[Brodie87] *On Knowledge Base Management Systems*, Brodie, M.L. and Mylopoulos, J., Ed., Springer-Verlag, 1987.

[Buneman82] Buneman, P., Frankel, R.E., and Nikhil, R. An Implementation Technique for Database Query Languages. *ACM Trans. on Database Systems 7*, 2 (June, 1982), 164–186.

[Buneman79] Buneman, P. and Frankel, R.E. FQL — A Functional Query Language. In *Proc. 1979 ACM SIGMOD Int'l. Conf. on the Management of Data, Boston Mass*, ACM, 1979.

[Cham75] Chamberlin, D.D., Gray, J.N., and Traiger, I.L. View Authorization and Locking in a Relational Database System. In *Proc. Nat'l. Computer Conf., Vol. 44*, AFIPS Press, 1975, pp. 425–430.

[Chen76] Chen, P.P. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Trans. on Database Systems 1*, 1 (Mar., 1976), 9–36.

[Codd70] Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM 13*, 6 (June, 1970), 377–387.

[Codd79] Codd, E.F. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems 4*, 4 (Dec., 1979), 397–434.

[Cope84] Copeland, G. and Maier, D. Making Smalltalk a Database System. In *ACM SIGMOD'84*, Yormark, B., Ed., ACM, 1984, pp. 316–325.

[Dahl66] Dahl, O.J. and Nygaard, K. SIMULA — An ALGOL-Based Simulation Language. *Comm. ACM 9*, 9 (1966), 671–678.

[DAHL70] DAHL, O.J., MYHRHANG, B. AND NYGAARD, K. The SIMULA 67 Common Base Language. Rep. No. S-22. Norwegian Computing Center, Forskningsveien 1B, Oslo 3.

[DATE81] DATE, C.J. *An Introduction to Database Systems — Third Edition*, Addison Wesley, 1981.

[DAYAL84] DAYAL, U. AND HWANG, H.-Y. View Definition and Generalization for Database Integration in a Multibase System. *IEEE Trans. on Software Engineering SE-10*, 6 (Nov., 1984), 628–645.

[DENNIS73] DENNIS, J.B. First Version of a Dataflow Procedure Language. MAC Tech. Memo. 61. MIT, Cambridge, MASS.

[FIKES77] FIKES, R.E. AND HENDRIX, G.G. A Network-Based Knowledge Representation and its Natural Deduction System. In *Proc. Fifth Int'l. Joint Conf. on AI*, 1977, pp. 235–246.

[FINDLER79] *ASSOCIATIVE NETWORKS Representation and Use of Knowledge by Computers*, Findler, N., Ed., Academic Press, 1979.

[FLINT84] FLINT, R.A. An Approach to Modeling Database Activity. Tech. Rep. No. 239. Univ. of CA., Irvine, Dept. of Info. and Comp. Sci..

[GILBERT88] GILBERT, J.P. AND BIC, L. Asynchronous Data Retrieval from an Object-Oriented Database. In *Proc. European Conf. on Obj. Oriented Programming, Oslo, Norway*, Springer Verlag, 1988.

[GOLDBERG81] GOLDBERG A. Introducing the Smalltalk-80 system. *BYTE 8*, 4 (Aug., 1981), 14-26.

[GOLDBERG83] GOLDBERG, A. AND ROBSON, D. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading Mass., 1983.

[HAMMER81] HAMMER M. AND MCLEOD D.J. Database Description with SDM: A Semantic Data Model. *ACM Trans. on Database Systems 6*, 3 (Sept., 1981), 351–386.

[HENDRIX79] HENDRIX, G.G Encoding Knowledge in Partitioned Networks. In *Associative Networks Representation and Use of Knowledge by Computers*, N. Findler, Ed., Academic Press, 1979, pp. 51–92.

[HULL87] HULL, R. AND KING, R. Semantic Database Modeling: Survey, Applications and Research Issues. *ACM Computing Surveys 19*, 3 (Sept., 1987), 201–260.

[KENT78] KENT, W. *Data and Reality — Basic Assumptions in Data Processing Reconsidered*, North-Holland, 1978.

[KENT83] KENT, W. A Simple Guide to Five Forms in Relational Database Theory. *Comm. ACM 26*, 2 (Feb., 1983).

[KHOSH86] KHOSHAFIAN, S.N., AND COPELAND, G.P. Object Identity. In *Proc. OOPSLA'86*, ACM, Portland, OR, 1986, pp. 406–416.

[KIM79] KIM, W. Relational Database Systems. *ACM Computing Surveys 11*, 3 (Sept., 1979), 185–211.

[KIM89] KIM, W., BERTINO, E. AND GARZA, J.F. Composite Objects Revisited. In *Proc. of the 1989 SIGMOD, Int'l. Conf. on the Management of Data*, ACM, Portland, OR, 1989, pp. 337–347.

[KING83] KING, J. (ED.) Special Issue on AI and Database Research. *ACM SIGART Newsletter*, 86 (Oct., 1983), 32–72.

[KLUG77] KLUG, A. AND TSICHRITZIS, D. Multiple View Support within the ANSI/SPARC Framework. In *Proc. Third Int'l. Conf. on Very large Data Bases*, ACM, 1977, pp. 477–488.

[KLUG78] KLUG, A.C. Theory of Database Mappings. Rep. No. CSRG-98. Computer Systems Research Group, Univ. of Toronto.

[KRISTEN87] KRISTENSEN, B.B., MADSEN, O.L., MØLLER-PEDERSEN, B. AND NYGAARD, K. The BETA Programming Language. In *Research Directions in Object-Oriented Programming*, Shriver, B. and Wegner P., Ed., MIT Press, 1987, pp. 7–48.

[LEDGARD81] LEDGARD, H. *ADA — An Introduction / ADA Reference Manual (July 1980)*, Springer Verlag, 1981.

[LISKOV74] LISKOV, B.H. AND ZILLES, S.N. Programming with Abstract Data Types. *ACM SIGPLAN Notices 9*, 4 (Apr., 1974), 50–59.

[LISKOV77] LISKOV, B.H. ET AL Abstraction Mechanisms in CLU. *Comm. of the ACM 20*, 8 (Aug., 1977), 564–576.

[MAIER85] MAIER, D., OTIS, A. AND PURDY, A. Object-Oriented database development at Servio Logic. *Database Engineering 5*, 1 (Dec., 1985), 58–65.

[McGee77] McGee, W.C. The Information Management System IMS/VS. *IBM Systems Journal 16*, 2 (1977), 82–168.

[McLeod78] McLeod, D. A Semantic Data Base Model and its Associated User Interface. Rep. No. MIT/LCS/TR-214. Lab. for Computer Sci., MIT, Cambridge.

[Motro83] Motro, A. Interrogating Superviews. In *Second Int'l. Conf. on Databases (ICOD-2)*, Cambridge Univ., Cambridge, England, 1983, pp. 107–126.

[Mylop80] Mylopoulos, J., Bernstein, P.A., and Wong H.K.T. A Language Facility for Designing Database-Intensive Applications. *ACM Trans. on Database Systems 5*, 2 (June, 1980), 185–207.

[Mylop88] *Artificial Intelligence and Databases*, Mylopoulos, J. and Brodie, M.L., Ed., Morgan Kaufmann, 1988.

[Purdy87] Purdy, A., Schuchardt, B. and Maier, D. Integrating an Object Server with Other Worlds. *ACM Trans. on Office Information Systems 5*, 1 (Jan., 1987), 27–47.

[Quillian68] Quillian, M.R. Semantic Memory. In *Semantic Information Processing*, M. Minsky, Ed., The MIT Press, Cambridge, MASS, 1968, pp. 227–270.

[Reiter83] Reiter, R., Gallaire, H., King, J., Mylopoulos, J. and Webber, B. A Panel on AI and Databases. In *Proc. Eighth IJCAI, Karlsruhe, West Germany*, IJCAI, 1983, pp. 1199–1206.

[Rowe79] Rowe, L.A. and Shoens, K.A. Data Abstraction, Views and Updates in RIGEL. In *Proc. of SIGMOD 1979 Conf.*, ACM, 1979, pp. 71–81.

[Rundenst89] Rundensteiner, E., Bic, L., Gilbert, J. and Yin, M.-L. Set-Related Restrictions for Semantic Groupings. Tech. Rep. 89-07. Dept. of Info. and Comp. Sci., University of CA, Irvine.

[Schrefl88] Schrefl, M. and Neuhold, E.J. Object Class Definition by Generalization Using Upward Inheritance. In *Proceedings of the Fourth International Conference on Data Engineering*, IEEE, Los Angeles, CA, 1988, pp. 4–13.

[SHIPMAN81] SHIPMAN, D.W. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. on Database Systems 6*, 1 (Mar., 1981), 140–173.

[SIBLEY77] SIBLEY, E.H. AND KERSHBERG Data architecture and data model considerations.. In *Proc. AFIPS Nat'l. Computer Conf.*, AFIPS, 1977, pp. 85–96.

[SMITH77] SMITH, J.M. AND SMITH D.C.P. Database Abstractions: Aggregation and Generalization. *ACM Trans. on Database Systems 2*, 2 (June, 1977), 105–133.

[STEFIK86] STEFIK, M. AND BOBROW D.G. Object-Oriented Programming: Themes and Variations. *The AI Magazine 6*, 4 (Jan., 1986), 40–62.

[STONEB86] STONEBRAKER, M. AND ROWE, L. The Design of POSTGRES. In *Proc. of SIGMOD 1986 Conf.*, ACM, 1986, pp. 340–355.

[STROU86] STROUSTRUP, B. *The C++ Programming Language*, Addison-Wesley, Reading Mass., 1986.

[TANAKA88] TANAKA, K., YOSHIKAWA, M. AND ISHIHARA, K. Schema Virtualization in Object-Oriented Databases. In *Proceedings of the Fourth International Conference on Data Engineering*, IEEE, Los Angeles, CA, 1988, pp. 23–30.

[TAYLOR76] TAYLOR, R..W. AND FRANK, R.L. CODASYL Data-base Management Systems. *ACM Computing Surveys 8*, 1 (Mar., 1976), 67–103.

[TSICH77] TSICHRITZIS, D.C. AND LOCHOVSKY, F.H. *Data Base Management Systems*, Academic Press, 1977.

[TSICH78] TSICHRITZIS, D. AND KLUG, A. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems. *Info. Systems 3* (1978), 173–191.

[ULLMAN82] ULLMAN, J.D. *Principles of Database Systems — Second Edition*, Computer Science Press, 1982.

[WONG83] WONG, E. Semantic Enhancement through Extended Relational Views. In *Second Int'l. Conf. on Databases (ICOD-2)*, Cambridge Univ., Cambridge, England, 1983, pp. 169–178.

Appendix 1
Object and Message Structures

/ PolyView's 'Global Symbol Table' consists of a single */*

*/** **Global Symbol Table** *and one* **View Model Object Table** **/*

/ for each active user view */*

Global Symbol Table: */* contains any number symbol table listings */*

Global Symbol Table Listing:

View Name: */* a unique external name for the view */*

View Identity: */* system generated identity of view */*

View Model: */* a pointer to the view's model object table */*

/ Note: each view has a single identity and model */*

/ and at least one name */*

View Model Object Table: */* contains any number model table listings */*

Model Table Listing:

Class Name: */* a unique external name for class in view */*

Class Identity: */* system generated identity of view */*

/ Note: each class in the view has* **at least** *one name */*

Base Class Object:

/ contains Generic Methods, an Object Description and a View Description */*

Generic Methods: */* used for data manipulation and updates */*

Object Description:

Class-Object-Id: */* unique internal name */*

Is-A Connections: */* internal names of objects */*

MetaClasses: list of Meta-Classes

Types: list of Ancestors

Parents: list of Parents

/ nonempty sublist-of Types unless Types is empty */*

Collections: list of Collections (& associated rules)

Categories: list of Categories (& associated rules)

PowerSets: list of Powersets

SubClasses: list of Subclasses

Instances: list of Instance Objects */* on secondary storage device */*

Class Attributes: */* some are inherited from meta-classes */*

Compound: list of Compound Attribute Instances

Local: list of Atomic Attribute Instances

Method: list of Derived Attribute Instances

Instance Attributes: */* inherited by all subclasses and instances */*

Compound: list of Compound Attribute Structures

Local: list of Atomic Attribute Structures

Category: list of Category Attribute Structures

/ category attributes are system defined and */*

/ maintained - they determine which objects */*

/ are members of a category */*

Method: list of Derived Attribute Definitions

View Description:

Global View: View Instance Description

User Views: list of users' View Instance Descriptions

End */* Base Class Object */*

View Instance Description:

View-Id: View-Id /* *unique view internal identity* */

Description: /* *description of object from this point of view* */

Names: list of Conditions & External Name Lists /* *for class* */

Is-A Connections:

Types: sublist of Object-Description.Is-A.Types

Parents: sublist of Object-Description.Is-A.Types

Collections: sublist of Object-Description.Is-A.Collections

Categories: sublist of Object-Description.Is-A.Categories

PowerSets: sublist of Object-Description.Is-A.Powersets

SubClasses: sublist of Object-Description.Is-A.SubClasses [restriction]

Instances: *nil* or Object-Description.Is-A.Instances [restriction]

Visible Attributes: /* *Class and Instance - ALL attributes for Global View* */

Compound: list of Compound Attribute View Structures and [restrictions]

Local: list of Local Attribute View Structures and [restrictions]

Category: list of Category Attribute Ids

Method: list of Derived Attribute View Structures

Invisible Attributes: /* *optional* */

Compound: list of Compound Attribute View Structures and Update-Methods

Local: list of Local Attribute View Structures and Default

Category: list of Category Attribute Ids

/* *Note that if the view is updatable then the following relationships must* */

/* *hold for all KEY attributes (compound, local and category):* */

/* *Visible-Attributes + Invisible-Attributes = Global-View* */

End /* *View Instance Description* */

Base Instance Object:

Object Description:

Instance-Object-Id /* *unique internal name* */

Is-A Connections:

Parent: Parent Class Id

Reference-Count: Integer

/* *number of attribute references to instance defined in Parent Class* */

Attributes: /* *inherited from owner class* */

Compound: list of Compound Attribute Instances

Local: list of Atomic Attribute Instances

Method: list of Derived Attribute Instances

Derived: list of References to attributes inherited from Derived Classes

Category: list of Booleans

/* *not visible to users — used by system to* */

/* *determine membership in some derived classes* */

End /* *Base Instance Object* */

Union-Subset Class Object: */* contains Object and View Descriptions */*

/ Both Category and Collection Classes have this format */*

Object Description:

Class-Object-Id: */* unique internal name */*

Class-Type: Category or Collection

Restrictions: list of Restrictions

Is-A Connections: */* internal names of objects */*

MetaClasses: list of Meta-Classes

PowerSets: list of Power Sets

BaseUnion: list of Base Classes

/ whose union forms the basis for the derived class */*

Class Attributes: */* some are inherited from meta-classes & power sets */*

Compound: list of Compound Attribute Instances

Local: list of Atomic Attribute Instances

Method: list of Derived Attribute Instances

Instance Attributes: */* inherited by all instances */*

Category: system generated Boolean */* only Categories have this */*

Compound: list of Compound Attribute Structures

Local: list of Atomic Attribute Structures

Method: list of Derived Attribute Definitions

/ these lists differ from their base class counter parts in */*

/ that they include a class ID in addition to a attribute */*

/ ID and restriction - this is because some attributes belong */*

/ to the union/subset and others belong to base classes - */*

/ restrictions on the latter are appended by the system to */*

/ queries before it is forwarded to appropriate base classes */*

View Description:

Global View: View Instance Description

User Views: list of users' View Instance Descriptions

End */* Union-Subset Class Object */*

Power-Set Class Object:

/ member may classes vary – based on the power set of the union of */*

/ base classes – each member of a power set is a union-subset class */*

Object Description:

Class-Object-Id: */* unique internal name */*

Restrictions: list of Restrictions

Is-A Connections: */* internal names of objects */*

MetaClasses: list of Meta-Classes

BaseUnion: list of Base Classes

UnionSubsets: list of Union-Subset Classes

Class Attributes:

Compound: list of Compound Attribute Instances

Local: list of Atomic Attribute Instances

Method: list of Derived Attribute Instances

Instance Attributes: */* may be empty */*

Compound: list of Compound Attribute Structures

Local: list of Atomic Attribute Structures

Method: list of Derived Attribute Definitions

/ these lists similar to the union-subset lists */*

View Description:

Global View: View Instance Description

User Views: list of users' View Instance Descriptions

End */* Power-Set Class Object */*

Message:

Target: Object Id

Sender: Object Id

Query-Id /* *system generated identity of query* */

Query-Status: Status

View-Id /* *provided by system which has info about users* */

Message-Type: /* *object uses this info to determine its behavior* */

Reply-Type: Message Type OR *user* /* *sender of message* */

Previous-Request: Message Type OR *user* /* *sender of message* */

Query-Restriction:

tree of Required/Independent/Dependent Property Restrictions

/* *restrictions on attributes includes status for* */

/* *each all dependent attribute paths are to be* */

/* *output each conditional attribute path 'may' be* */

/* *marked for output* */

Check-Status: True or False

End /* *Message* */

Appendix 2

Generic Methods

Notes:

1. The message format is shown on the last page of appendix 1.
2. Each time a Request message is sent, the message.reply-type and previous.request are automatically set. This is necessary because many methods generate reply messages for several different message types.
3. Status values are associated with all restrictions and messages.
4. Processing is purely message driven. This is conceptually elegant but can be hard to follow.
5. Loop iterations can be processed independently unless later iterations use objects created by earlier ones.
6. With the exception of the first method, all methods are triggered by messages which bare their name.

Information Retrieval Methods

Request-From-Outside is triggered by a Subset-Query-Request message or an Independent-Attribute-Query-Request message; both of these originate outside the object receiving the message. Each restriction in the message is is examined — if it is not known to satisfy the query then the **Check-Attribute** method is invoked in order to determine the restriction's status.

```
Procedure Request-From-Outside
  create activity record for pending query
  for each independent subtree I in query-restriction
    if Status.head(I)=1
      then( /* it satisfies the query */
        new.RHS.head(I) := intersection(RHS.head(I),
                           union(visible.local-attribute-list(message.view-id),
                                 visible.compound-attribute-list(message.view-id),
                                 visible.method-attribute-list(message.view-id)))
        send an Attribute-Query-Result message containing head(I) to self)
      else(Check-Attribute(I,message.view-id))
  endfor
endProcedure
```

Subset-Query-Result collects the results from subset queries which were spawned by the object. For a particular query, the appropriate reply is sent after the last result has been received. There are three possible reply messages because subset queries may have been spawned by operations which insert attributes and move objects as well as by other subset query operations.

```
Procedure Subset-Query-Result
  store result
  if result.last(activity)
    then(
      determine status of query
      switch /* determine type of original query and send appropriate result */
        case (reply-type.message is Subset-Query)
          send Subset-Query-Result message to sender
        case (reply-type.message is Insert-Attribute)
            /* if a single object satisfies this request then a reference
               to that object is returned otherwise the request fails    */
          send Insert-Attribute-Result message to sender
        otherwise /* reply-type.message is Move-Object */
          send Move-Object-Result message to sender
      endswitch
      destroy activity record)
endProcedure
```

Check-Attribute has two arguments — a restriction and the view identity. It determines the type of the attribute which is to be restricted and initiates a process which determines the status of the restriction. There are five possibilities depending on what is restricted ...

1. The type of the object — the **Type-Test** function is used to determine its status (see below).
2. A local attribute — its status can be determined directly.
3. A compound attribute — an attribute request message is sent.
4. A method in which case the method itself will determine the status.
5. An unrecognized attribute — if the attribute was previously found then the restriction will now fail; otherwise, its status will remain unfound.

```
Procedure Check-Attribute(R,view-id)
 switch /* determine whether head(R) is Type-test, Local, Compound or Virtual */
   case (head(R) is a type-test)
     new.head(R) := Type-Test(head(R),view-id)
     send an Attribute-Query-Result message to self
   case (head(R) is on visible.local-attribute-list(view-id))
     values := lookup(local-attribute-list(view-id))
     /* in general a list of ranges of values */
     result := intersection(RHS.R, values)
     status := if empty(result) then(5 /* restriction not satisfied */)
                 else(if non-key(R) then(3 /* non-key attribute */)
                   else(if value in restriction then(1 /* key & satisfiable */)
                     else(if restriction in value then(2 /* key and found */))))
     send an Attribute-Query-Result message containing status,result to self
   case (head(R) is on visible.compound-attribute-list(view-id))
     send an Attribute-Query-Request message
       containing tail(R) along all attribute-arcs = head(R)
   case (head(R) is on visible.method-attribute-list(view-id))
     if node is not a leaf and head(R) is a class attribute
       OR if node is a leaf and head(R) is a instance attribute
         then spawn appropriate subquery
   otherwise /* there is no attribute which corresponds to head(R) */
     if status.head(R) = 3 then new.status.head(R) = 5
     /* otherwise the head(R) is unchanged - head(R)
     must not have been found before */
     send an Attribute-Query-Result message containing R to self
 endswitch
endProcedure
```

Type-Test determines whether the current object is is-a related to the class referred to by the restriction (R) in the specified view (view-id). Like **Check-Attribute** there are five cases.

1. If the class is an ancestor or power set of the current class then the type test is satisfied.
2. If the class is part of the base union (of a union subset class) then it cannot be determined whether *every* instance object satisfies the type test.
3. If the class is a category based on the current class then the result is identical to 2.
4. If the class is a collection based on the current class then the status is determined by examining the rule associated with the collection.
5. Otherwise, the class was not found on any of the relevant is-a-connection lists. If the object is a leaf then the status is set to fail if it is not, it remains unknown.

```
Function Type-Test(R,view-id)
  /* a type test restriction has the following form: isa(type-name),status
     all references to an object should be prefixed by:  is-a-connections. */
 switch
   case (R.type-name in types(view-id) OR R.type-name in powerset(view-id))
        new.status.R := 1
   case (R.type-name in baseunion(view-id))
        new.status.R := 3
   case (R.type-name in category(view-id))
        new.status.R := 3
        R.restriction := (R.type-name = "true")
   case (R.type-name in collection(view-id))
        new.status.R := status.R.type-name.collection(view-id)
        R.restriction := rule.R.type-name.collection(view-id)
   otherwise
        if leaf
          then(new.status.R := 5)
          else(new.status.R := 4)
 endswitch
 Type-Test := R
endFunction
```

Attribute-Query-Request is very similar to **Request-From-Outside**. The difference is that this method checks to see whether or not it is a terminal node before attempting to process the query.

```
Procedure Attribute-Query-Request
  create activity record for pending query
  if message.query-restriction is empty /* node is terminal */
    then(send an Attribute-Query-Result message to self)
    else(
      for each subtree S in message.query-restriction
        if Status.head(S)=1
          then( /* it satisfies the query */
            new.RHS.head(S) := intersection(RHS.head(S),
                              union(visible.local-attribute-list(message.view-id),
                                    visible.compound-attribute-list(message.view-id),
                                    visible.method-attribute-list(message.view-id)))
            send an Attribute-Query-Result message containing head(S) to self)
          else(Check-Attribute(S,message.view-id))
      endfor)
endProcedure
```

Attribute-Query-Result is similar to **Subset-Query-Result** in several ways — it may collect several results before sending a single message. It must be able to respond in several different ways because **Attribute-Query-Request** may be invoked by several different methods. If initiated by either an attribute request or a delete object request then the corresponding result is spawned in a straightforward manner. If the attribute query was independent and it was satisfied then any associated dependent attribute subqueries must be processed prior to returning the result. Finally, the attribute query message may have been sent by a **Subset-Query-Request**. Although it looks more complicated, the procedure followed in this case is very similar to the previous one. It can be summarized in the following way — once all required restrictions have been satisfied, messages for the optional restrictions (if there are any) can be sent.

```
Procedure Attribute-Query-Result
 store result
 if result.last(activity)
   determine status of query /* combines status of required  subtrees /*
   switch
   case (reply-type.message is Attribute-Query-Request)
     send an Attribute-Query-Result message to sender
     destroy activity record
   case (reply-type.message is Independent-Attribute-Query-Request)
     if query.status < 4 & previous.request is independent-subtree.subquery
          & dependent attribute tree A in activity.query-restriction is not empty
       then(for each dependent subtree D in activity.query-restriction
              if status of corresponding independent subtree < 4
                then(Check-Attribute(D),message.view-id)
                else(send an Attribute-Query-Result message to self /* leave status = 4 */)
            endfor)
       else(send an Attribute-Query-Result message to sender)
   case (reply-type.message is Delete-Object-Request)
     send a Delete-Object-Result message to sender
     destroy activity record
   case (reply-type.message is Subset-Query-Request)
     if class(object) & status.query not = 5
       then(if previous.request is independent-subtree.subquery
         then(for each dependent subtree D in activity.query-restriction
                if status of corresponding independent subtree < 4
                  then(Check-Attribute(D),message.view-id)
                  else(send an Attribute-Query-Result message to self)
              endfor)
         else(for each non-leaf child send a Subset-Query-Request message to child
           if status < 4
             then(for each leaf child send a Subset-Query-Request message to child)
           adjust original activity record to reflect change in query)
       else( /* the node is a leaf OR query has failed */
         send a Subset-Query-Result message to sender
         destroy activity record)
   endcase
endProcedure
```

Insert Object and Insert Attribute Methods

Insert-Object-Request first finds the target class for the insert operation. Once the class is found, **Insert-Object-Request** causes it to create a new object. The new object's local attributes can be inserted directly while compound attributes must be inserted by sending **Insert-Attribute-Request** messages.

```
Procedure Insert-Object-Request
  create activity record for pending query
    /* Before an object can be inserted the class hierarchy must be
      searched for the class which will perform the create operation
      — the search algorithm is not included here because it is
      very similar to the Subset-Query-Request method except that it
      searches only the non-leaf (class) objects
      Locate the appropriate Owner Class & then perform the following: */
  create a new instance object (N)
    /* Create uses information in Object-Description.Instance-Attributes,
        View-Instance-Description.Visible-Attributes and
          View-Instance-Description.Invisible-Attributes.
      This new object is a shallow copy of the "typical" object for the current
      view — it contains a unique object id, defaults for attributes (but with pointers
      from the new object only) and IS-A (parent) arc the latter points to its owner class */
  insert IS-A (child) arc to N
  for each local subtree L in query-restriction
    change value in N
    remove L from query-restriction
  endfor
  if query-restriction does not contain compound attributes
    then(send an Insert-Object-Result message to self)
    else(for each (compound) attribute,subtree pair (A,S) in query-restriction
          delete attribute arc corresponding to A in self
          send an Insert-Attribute-Request(A,S) message to self
        endfor)
endProcedure
```

Insert-Object-Result must wait for all outstanding requests to be answered before reponding. The two cases are very straightforward — the only complication occurs if the original query is an **Insert-Object-Request**. In this case, the system must ensure that any default attributes are properly connected to the new object.

```
Procedure Insert-Object-Result
  store result
  if result.last(activity)
    then(
      determine status of query
      /* determine type of original query and send appropriate result */
      if reply-type.message is Insert-Object
        then(for all remaining non local unchanged default attributes D
                send insert-attribute-arc(inverse(D),self) message along D
            send an Insert-Object-Result message to self)
        else(/* reply-type.message is Insert-Attribute */
            send an Insert-Attribute-Result message to sender)
      destroy activity record)
endProcedure
```

Insert-Attribute-Request requires a special explanation. It initiates a search for an object which can fulfill the role of the attribute before the actual insert operation is performed. The actual insert operation is performed by **Insert-Attribute-Result** after the result of the search has been determined.

```
Procedure Insert-Attribute-Request(A,S)
  create activity record for pending query
  send a Subset-Query-Request message (containing (A,S)) to owner class of attribute object
endProcedure
```

There are two cases for the **Insert-Attribute-Result** method. The **Subset-Query-Request** message which is spawned by the insert attribute request is guaranteed to return a reference to exactly one object or fail. The first, which has three subcases, occurs when an object is found. An attribute relationship is established between the newly created object and found object by creating arcs in both directions. Once the arcs have been created, one of three result messages (again this is dependent on the original message) is sent. Otherwise, a new object must be created to represent the attribute.

```
Procedure Insert-Attribute-Result(A,S)
 if (A,S) was found /* this is guaranteed to be exactly one object */
   then(send insert-attribute-arc(A,S) message to self
        send insert-attribute-arc(inverse(A),self) message to S
        switch
          case (reply-type.message is Insert-Object)
            send an Insert-Object-Result message to sender
          case(reply-type.message is Insert-Attribute)
            send an Insert-Attribute-Result message to sender)
          otherwise /* reply-type.message is Create-Move-Copy-Result */
            send a Create-Move-Copy-Result message to sender
        endcase
        destroy activity record)
   else(send an Insert-Object-Request message (containing S) to attribute's owner class)
endProcedure
```

Delete Object and Delete Attribute Methods

Delete-Object-Request causes the receiving object to determine whether it satisfies the specified restrictions by spawning an attribute query request. In addition, if the object is a class, it sets the query's check status value to true — this information is used by the **Delete-Object-Result** method.

```
Procedure Delete-Object-Request
 create activity record for pending query
 if object is not a leaf
   then(new.check.status := 'true')
 send an Attribute-Query-Request to self
 /* make sure that object is or may lead to a candidate for the delete operation */
endProcedure
```

Once the last result has been received; **Delete-Object-Result** determines the status of the operation. If the object does not satisfy the restrictions then a result message is sent immediately. Otherwise, the delete operation continues. If the object is a leaf then it sends itself a delete all attributes message; otherwise, it uses the check.status value to determine whether it is reporting a result or propagating a request to its descendents.

```
Procedure Delete-Object-Result
  store result
  if result.last(activity)
    then(/* determine status of operation, take appropriate action and report result */
        if status.operation = success
          then(
            switch
              case (leaf(object))
                send a Delete-All-Attributes message to self
              case (class(object) & check.status = 'true')
                new.check.status := 'false'
                for each child send a Delete-Object-Request message to child
              otherwise /* class(object) & check.status = 'false' */
                for each 'deleted' leaf object remove IS-A arc
                send a Delete-Object-Result message to sender
            endswitch)
          else(/* inform sender that object has not been deleted */
            send a Delete-Object-Result message to sender)
        destroy activity record)
endProcedure
```

Notes about deleting attributes:

1. If any attribute delete operation removes a key attribute then it automatically sends out a delete object request message to the local object.
2. There are separate methods for deleting an attribute and its inverse. Requesting an object to delete an attribute causes it to send a **Delete-Attribute-Inverse-Request** message before deleting the attribute.

If the attribute is local then **Delete-Attribute-Request** deletes it immediately, otherwise, it spawns a **Delete-Attribute-Inverse-Request**.

```
Procedure Delete-Attribute-Request(A)
  /* This method shows the response of an instance object — it
     is assumed that the asynchronous search strategy was used
     to identify the target instance object.              */
  create activity record for pending query
  if local(A)
    then(if A has a special delete method associated with it
            then(execute special delete method)
            else(if A is key
              then(send a Delete-Object-Request message (referencing self) to owner class)
              else(new.A := nil))
        send a Delete-Attribute-Result message to sender)
    else(send a Delete-Attribute-Inverse-Request message along A)
endProcedure
```

Delete-Attribute-Result deletes the attribute arc (if necessary) — its inverse has already been removed. It also sends the appropriate result message to the sender.

```
Procedure Delete-Attribute-Result(A)
  switch
    case (previous.request is Delete-Object-Request)
      send Delete-Object-Result to sender
    otherwise /* original request was Delete-Attribute-Request */
      if A has a special delete method associated with it
        then(execute special delete method)
        else(if A is key
          then(send a Delete-Object-Request message (referencing self) to owner class)
          else(new.A := nil))
      send a Delete-Attribute-Result to sender
  endswitch
  destroy activity record
endProcedure
```

Delete-Attribute-Inverse-Request deletes the specified attribute and sends back a message of confirmation.

```
Procedure Delete-Attribute-Inverse-Request(A)
  create activity record for pending query
    if A.inverse has a special delete method associated with it
      then(execute special delete method)
      else(replace A.inverse with nil
          if A.inverse is key
            then(send a Delete-Object-Request message (referencing self) to owner class))
  send a Delete-Attribute-Inverse-Result to sender
endProcedure
```

Delete-Attribute-Inverse-Result is straightforward — it simply sends the appropriate result message to the sender.

```
Procedure Delete-Attribute-Inverse-Result(A)
  switch
    case (previous.request is Delete-Attribute-Request)
      send a Delete-Attribute-Result to self
    case (previous.request is Delete-All-Attributes-Request)
      send a Delete-All-Attributes-Result to self
  endswitch
  destroy activity record
endProcedure
```

Delete-All-Attributes-Request is spawned by the **Delete-Object-Result** method at the leaf level only. It does not have to check for key attributes (see **Delete-All-Attributes-Result** below).

```
Procedure Delete-All-Attributes-Request
  create activity record for pending query
  for each non-nil attribute arc(A)
    send a Delete-Attribute-Inverse-Request(A) along A
endProcedure
```

Delete-All-Attributes-Result removes the object's connection to the class lattice — it effectively deletes the object from the database.

```
Procedure Delete-All-Attributes-Result
  if result.last(activity)
    then(remove IS-A (to parent)
        send a Delete-Object-Result message to sender (of Delete-Object-Request)
        destroy activity record)
endProcedure
```

Change Attribute Methods

Change-Attribute-Request has two arguments: the attribute to be changed (A) and a description of the changed object (C). If A exists then there are two possibilities: A is local and can be changed directly or A is compound and the change must be propagated along A's arc.

```
Procedure Change-Attribute-Request(A,C)
  /* This method shows the response of an instance object — it
     is assumed that the asynchronous search strategy was used
     to identify the target instance object.                  */
  create activity record for pending query
  /* Make sure that attribute A exists & if it does not
     report that fact to the source of the request */
  if not found(A)
    then(new.query.status := 5
        send Change-Attribute-Result message to sender)
    else(if local(A)
          then(new.query.status := 1
               if A has a special change method associated with it
                 then(execute special change method using C)
                 else(new.A := C)
               send a Change-Attribute-Result message to sender)
    else(recursively propagate the change request along A)
endProcedure
```

Change-Attribute-Result is self explanatory.

```
Procedure Change-Attribute-Result
  send a Change-Attribute-Result message to sender
  destroy activity record
endProcedure
```

Move Object Methods — note this is semantically different from applying the Delete Method followed by the Insert Method because moved objects' identities are unchanged by this operation.

The main purpose of **Move-Object-Request** is to find the objects which are to be moved. It sends a subset query request message to do this.

```
Procedure Move-Object-Request
  /* Triggered by an Move-Object-Request message sent to the current owner class */
  create activity record for pending query
  new.found-status := 'false'
  send a Subset-Query-Request to self
endProcedure
```

If the search is successful, the **Move-Object-Result** spawns **Create-Move-Copy-Request** messages which actually perform the move object operation.

```
Procedure Move-Object-Result
  if found-status = 'false' & search.status = 'success'
    then(new.found-status := 'true'
        for each leaf found send a Create-Moved-Copy to new owner class
    else(send a Move-Object-Result to sender
        destroy activity record)
endProcedure
```

Create-Move-Copy-Request is almost identical to **Insert-Object-Request**. It differs in one *major* aspect — instead of creating a new object with its own identity, it creates a new version of an existing object without changing its identity.

```
Procedure Create-Move-Copy-Request
  create activity record for pending query
    /* Before an object can be moved the new class hierarchy must be
       searched for the class which will perform the create operation
       — this is exactly the same search algorithm which is used by
       Insert-Object-Request method
       Locate the appropriate Owner Class & then perform the following: */
  create-version instance object (N)
    /* Create-version uses information in Object-Description.Instance-Attributes,
         View-Instance-Description.Visible-Attributes and
           View-Instance-Description.Invisible-Attributes.
       This new object is a shallow copy of the "typical" object for the current view
       — it shares the moved object's id, it includes defaults for attributes (but with pointers
       from the new object only) and IS-A (parent) arc the latter points to its owner class */
  insert IS-A (child) arc to N
  for each local subtree L in query-restriction
    change value in N
    remove L from query-restriction
  endfor
  for each attribute,subtree pair (A,S) in query-restriction
    delete attribute arc corresponding to A in self
    send an Insert-Attribute-Request(A,S) message to self
  endfor
  for all non local unchanged default attributes D
    send insert-attribute-arc(inverse(D),self) message along D
  send an Create-Move-Copy-Result message to self
endProcedure
```

Create-Move-Copy-Result is very similar to the second case of **Insert-Object-Result** but, in addition to making sure that default attributes are properly connected to the new object, it also spawns a message which causes the old version of the object to be deleted.

```
Procedure Create-Move-Copy-Result
  store result
  if result.last(activity)
    then(
      determine status of query
      /* determine type of original query and send appropriate result */
      for all non local unchanged default attributes D
         send insert-attribute-arc(inverse(D),self) message along D
      send a Move-Object-Result message to self
      send a Delete-Object-Request to owner of copied instance object
      destroy activity record)
endProcedure
```

Create-Move-Copy-Result is very similar to the second case of Lookup-Object-Result but in addition to making sure that default attributes are properly connected to the new object, it also sends a message which causes the old version of the object to be deleted.

```
Procedure Create-Move-Copy-Result
  Begin
  if (query is activated)
    Then
    determine status of query
    /* determine type of original query and send appropriate result */
    for all non local unchanged default attributes D
        send Insert-attribute-and-return(D,self) message along D
    send a Move-Object-Result message to self
    send a Delete-Object message to owner of copied instance object
    deactivate activity record)
endProcedure
```