Copyright Warning & Restrictions

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

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

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

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

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

Order Number 9401728

Algorithms for generation of path-methods in object-oriented databases

Mehta, Ashish Khandubhai, Ph.D.

New Jersey Institute of Technology, 1993

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

ALGORITHMS FOR GENERATION OF PATH–METHODS IN OBJECT–ORIENTED DATABASES

by

Ashish Mehta

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

Department of Computer and Information Science

May 1993

ABSTRACT

Algorithms for Generation of Path–Methods in Object–Oriented Databases

by

Ashish Mehta

A *path–method* is a mechanism in object–oriented databases (OODBs) to retrieve or to update information relevant to one class that is not stored with that class but with some other class. A path–method is a method which traverses from one class through a chain of connections between classes to access information at another class. However, it is a difficult task for a user to write path–methods, because it might require comprehensive knowledge of many classes of the conceptual schema, while a typical user has often incomplete or even inconsistent knowledge of the schema.

This dissertation proposes an approach to the generation of path–methods in an OODB to solve this problem. We have developed the *Path–Method Generator* (PMG) system, which generates path–methods according to a naive user's requests. PMG is based on *access weights* which reflect the relative frequency of the connections and precomputed *access relevance* between every pair of classes of the OODB computed from access weights of the connections. We present specific rules for access weight assignment, efficient algorithms to compute access relevance in a single OODB, and a variety of traversal algorithms based on access weights and precomputed access relevance. Experiments with a university environment OODB and a sample of path–

methods identify some of these algorithms as very successful in generating most of the desired path–methods. Thus, the PMG system is an efficient tool for aiding the user with the difficult task of querying and updating a large OODB.

The path–method generation in an interoperable multi object–oriented database (IM–OODB) is even more difficult than for a single OODB, since a user has to be familiar with several OODBs. We use a hierarchical approach for deriving efficient online algorithms for the computation of access relevance in an IM–OODB, based on precomputed access relevance for each autonomous OODB. In an IM–OODB the access relevance is used as guide in generating path–methods between the classes of different OODBs.

APPROVAL PAGE

Algorithms for Generation of

Path–Methods in Object–Oriented Databases

Ashish Mehta

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

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

Dr. James A. M. McHugh, Committee Member (date)
Professor, Computer and Information Science Department,
New Jersey Institute of Technology

Dr. Bonnie MacKellar, Outside Reader (date)
Assistant Professor, Mathematics and Computer Science Department,
Western Connecticut State University

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

Dr. Aaron Watters, Committee Member (date)
Assistant Professor, Computer and Information Science Department,
New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author: Ashish Khandubhai Mehta

Degree: Doctor of Philosophy in Computer Science

Date: May 1993

Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1993
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1989
- Bachelor of Engineering in Electronics & Communications,
  Gujarat University, Ahmedabad, India, 1987

Publications:

- "Computing Access Relevance to Support Path-Method Generation in Interoperable Multi-OODB", with J. Geller, Y. Perl, and P. Fankhauser, *(Full Paper) In the Proceedings of the RIDE–IMS'93: Third International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems*, Vienna, Austria, April 18–20, 1993, pp. 144–151.
- "Algorithms for Access Relevance to Support Path-Method Generation in OODBs", with J. Geller, Y. Perl, and P. Fankhauser, *In the Proceedings of the Fourth International Hong Kong Computer Society Database Workshop*, Shatin, Hong Kong, Dec. 12–13, 1992, pp. 183–200.
- "Algorithms for Computing Access Relevance in Object–Oriented Databases", with J. Geller, Y. Perl, and P. Fankhauser, *(Extended Abstract) In the Proceedings of the First International Conference on Information and Knowledge Management*, Maryland, USA, Nov. 8–10, 1992, pp. 657.

- "Algorithms for Structural Schema Integration", with J. Geller, Y. Perl, E.J. Neuhold, and A. Sheth, *In the Proceedings of the Second International Conference on Systems Integration*, Morristown, NJ, USA, June 15–18, 1992, pp. 604–614.

- "Cascading LZW algorithm with Huffman Coding: A Variable to Variable Length Compression Algorithm", with Y. Perl, *In the Proceedings of First Great Lakes Computer Science Conference, Western Michigan University, Kalamazoo, Michigan, Published as Computing in 90's, Lecture Notes in Computer Science Series, vol. 507, Springer Verlag*, 1991, pp. 170–178.

This dissertation is dedicated to

my loving father, late Khandubhai P. Mehta

and

my loving mother, Vasantiben K. Mehta

ACKNOWLEDGMENT

I would like to thank my advisor, Professor Yehoshua Perl, for his invaluable contributions in this dissertation. I would also like to thank my co-advisor Dr. James Geller for all good suggestions he made regarding this dissertation. I would also like to thank all the committee members.

I would like to thank all the members of our research group, particularly those who have participated in the development of the university database. Namely, Veena Teli, Hungkway Chao, Christino Wijaya, Munir Ahmedi, Salil Kulkarni, Abhay Bhave, Nimesh Dixit, Manhar Patel, Himanshu Pandit, Bheeman Lingan, and Madhumathi Tulsiram. I would also like to thank Prasanna Venkatesh, Aruna Kolla, and Mary Wang from the graphical knowledge representation project. From the OODINI group, I would like to thank Mike Halper and Subrata Chaterjee. I would also like to thank members of GMD–IPSI, particularly, Wolfgang Klas, Peter Fankhauser, and Gisela Fischer.

I would also like to thank the department chairperson Dr. Peter Ng for his financial support and Dr. Erich Neuhold from GMD–IPSI for his suggestions, financial support and for two trips to GMD–IPSI. I would aiso like to thank Dr. Amit Sheth from Bellcore for his suggestions. I also thank Dr. Erole Gelenbe for partially supporting me for one semester and for a conference trip.

Finally, I would like to thank my family members, and relatives.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Description

One of the most important components of an object–oriented database (OODB) is its query language. Generally, query processing in OODBs requires extraction of information from several classes and possibly combination of the results to form an answer to a query. While processing a query, we apply a message to an object (an instance of a class). If the message can be handled by the object, we retrieve the necessary information from it. If not, we might need to traverse from the object where we applied the message to other related objects, to retrieve an answer, if this answer is available at all in the OODB. To perform this traversal we use a *path–method* mechanism, which accesses information relevant to a class in an OODB, that is not stored with that class but with some other class. Informally, a path–method is defined as a method which traverses from one class through a chain of connections (user–defined or generic relationships) between classes either to retrieve information from another class or to update information at another class.

The current object–oriented databases assume that

1. path–methods to support queries either exist or a user needs to write them, based on his knowledge of the conceptual schema, *or*

2. a user view has already been created and the user can formulate all his queries within this limited scope.

But, we observe that

1. for large OODBs writing such path–methods might require knowledge of many of the classes in the database, even those that will ultimately not be used by a specific application;

2. in writing these path–methods ahead of time it is necessary to predict what kind of user requests will be applied to each class in the database, i.e., there exists a predefined static view;

3. formulating *ad hoc* queries is a frustrating task, as incomplete queries will be rejected without helpful hints by the database.

We introduce as a solution to these problems generation of path–methods for OODBs. The PMG system requires user–interaction to judge, whether the generated path–method is the one s/he desired. One of the major advantages of automatic path–method generation is that it neither requires any prior knowledge of the conceptual schema nor the creation of predefined views. It is also not necessary to browse the schema or write path–methods for all the classes in the system in advance. A (novice) user can formulate his queries as per his (naive) view of the database. For example, to find all the courses of an instructor, the user can specify his request as (instructor, course). In the schema the class instructor does not have a property "courses" but only

"sections". Thus, the generation of a path–method becomes necessary. The system will generate the necessary path–method and display it to the user for verification. After this verification the user query will be processed by executing the path–method. The path–method may then be stored for later reuse.

1.2 Related Work

Most OODBs such as GemStone [BOS91], ONTOS [M91], ObjectStore [LLOW91], O_2 [D91a], IRIS [F87], ITASCA [B91], and VERSANT [GD91] have introduced SQL-based query languages. These query languages are not richer than the relational or the nested relational model for query languages. An improved query model was developed for ORION [BKK88, K89, K90a], which is consistent with object–oriented concepts and is inherently richer than relational or nested relational query models.

It has been observed in [JTTW88] that traversal queries dominate set queries in object–oriented databases and in [G91] that navigational access, is important in object–oriented databases. [KKS92, BNPS92, LR92] discuss query languages which supports traversal using path–expressions. The term *path–expression* was first introduced in [MBW80] and has had many incarnations since. In [KKS92] a query language XSQL is introduced which supports path–expressions to query the OODB schema. In [BNPS92], OODB methods and OODB views are contrasted. It shows importance of methods to query an OODB. In [LR92] a query language WS–OSQL is developed for WS–IRIS prototype which offers a navigational interface. In [KM90]

predefined path–expressions and in [OHMS92] system-known path indexes are maintained for answering queries. Optimization techniques of OODB queries using paths are discussed in [CD92] and [LVZ92].

In our approach we describe path–expressions as path–methods and discuss semi–automatic generation of such path–methods, which is not found in the literature. The automatic generation of join sequences in relational databases is an analogous problem to path–method generation. There are two major approaches to this problem, the universal schema interface [M83, MU83, MRSS87] and the implicit join [L85]. However, in automatic generation of joins, few usable results have been achieved so far. There is a fundamental difference between path–methods discussed in this dissertation and joins in relational databases. Path–methods are generated using connections of the OODB *schema* and can be stored as methods of a class. Joins are used to combine *data* stored in different relations, which may be quite large, and require a large overhead for deriving query–results. On the other hand, once a path–method is generated, its execution requires just the fast traversal of the necessary connections which appear in the definition of the path–method. For example, in a relational database to find all the sections taken by a student, we might need to join three relations, student, transcript, and sections, which contain information about all the students, all the transcripts and all the sections, respectively. Thus the join operation requires processing a large volume of data. In an OODB, once we find a student instance we follow the connections from student to transcript and then

from transcript to sections. This traversal does not require any information about transcripts of other students, or sections taken by other students. Thus, the execution of such a path–method is significantly more efficient than that of a join sequence in a relational database.

The process of path–method generation requires traversal of an OODB schema. In general, OODB schemas contain much richer information compared with relational schemas. For example, some models permit several kinds of specialization relationships (e.g., [NPGT91]). As another example, suppose that a university database includes information about students of different kinds (e.g., graduate_student, undergrad_student, etc.). In a relational database, there would likely be an attribute Studenttype having the various student kinds as its possible values. In an OODB, there would likely be a class **student** having the various student kinds as subclasses. Hence, the OODB schema representation shifts the information about student kinds from the data to the schema [KKS92]. Thus, the user can refer solely to one kind, say graduate_student, without referring to the other kinds at all. The much richer information available within OODB schemas promises more success for automatic generation of path–methods than was achieved by generating join sequences.

In relational and/or semantic data model approaches [BH86, GKG85, KM84, L86a], it has been noticed that an intelligent schema navigation tool can be helpful for better query support. For relational databases, query construction using naive user's tokens is discussed in [M86]. It has been proposed by [L86b] that shared be-

havior in OODB systems can be accomplished by message forwarding. A knowledge–based approach to overcome structural differences in OODB integration is discussed in [SN88a]. It is also observed in [S88, NS88] that when a query processor fails, an interactive knowledge navigator, which accesses various thesauri, can help generating a message forwarding plan. The concept of *dynamic message forwarding plan generation* for incompletely specified global views of integrated databases is discussed in [NS88]. Schema independent query formulation, i.e., finding proper terms defined in the schema from the terms contained in a user–query has been discussed in [KN89]. The PMG system described in this dissertation can be considered as a *powerful underlying traversal tool* for schema independent query formulation [KN89], for dynamic derivation of personalized views [NS88], and in general as a retrieval/update mechanism for OODBs.

There are several techniques to decide semantic relationships between classes [SK92b, SN88a]. A classification of different techniques used for deciding semantic relationships between classes has been discussed in [S91a]. It considers graphical facilities, query languages, and the use of a thesaurus, a dictionary, or meta–data. The semantic relationships between classes are defined in [NPGT91, SN88a, K90b, KNS88, KNS89]. In [K90b] semantic relationships between classes are realized using metaclasses. An approach using semantic resemblance between classes is discussed in [FKN91, FN92]. Another approach which uses the notion of semantic proximity is discussed in [SK92b]. Later on we will show how our approach using the notion of ac-

cess relevance differs from the above two approaches. It has been observed in [SG89, S91a] that, because the system has incomplete, sometimes inconsistent knowledge, semantic relationships cannot always be determined by the system without human input. The PMG system requires a human to check, whether the generated path–method is the desired one or not. Verifying a path–method is certainly easier than generating one.

1.3 Framework of the Dissertation

This dissertation is part of a large collaborative research effort of NJIT and IPSI–GMD[1]. The Path–Method Generator system [MPG92], discussed in this dissertation is implemented as a module of the VODAK OODB system [KNBD92]. This VODAK system is based on an object–oriented database model called the Dual Model [NPGT91].

A human traverses an OODB schema by applying his intuitive understanding of the classes and their connections. The *Path–Method Generator* (PMG) performs a similar traversal. As shown in Figure 1.1, an OODB manipulation request will first be accepted by the translator of the OODBMS. If the query cannot be completed successfully, the Path–Method Generator is called. It will use the object-oriented schema to generate a path–method needed for completing the query process. Gen-

[1]Integrated Information Publications Systems Institute – Gesellschaft fuer Mathematik und Datenverarbeitung

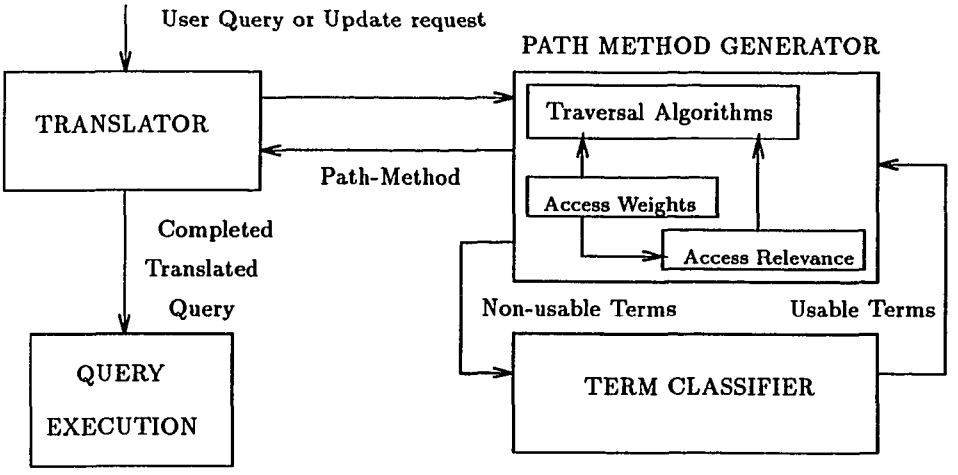

Figure 1.1 Architecture for Query Processing

erated path–methods can be added to the schema or they can be used just for that specific query. Since the user is not assumed to know the details of the schema s/he will often use terms that do not occur in the schema, e.g. "employee of department" instead of "member of department." Here the Term–Classifier which is based on the Knowledge Explorer [K91] will provide for the necessary translation.

In this dissertation we shall concentrate only on the mechanisms of the PMG. We discuss path–method generation using access weights and precomputed access relevance [MPGF92, MPGF93]. The definitions and motivation of using access weights and access relevance for guiding the traversal of the schema and algorithms for the computation of access relevance are discussed. The traversal algorithms of PMG using access weights and precomputed access relevance are compared experimentally to one another and to known uniform traversal algorithms, such as breadth first search and depth first search. In this dissertation, we discuss computation of access relevance in an OODB and in an interoperable multi–OODB to support path–method generation.

1.4 User Interaction with the PMG System

Now we will clarify our assumptions about the interaction between the user and the PMG system introduced in this dissertation. We assume that the user knows the source and the target of the desired path–method. S/he does not know the exact code and not even the corresponding traversal path for the path–methods, but perceives the proper interpretation or "semantics" of the path–method s/he desires and can give a verbal description of it. We further assume that the user supplies to the PMG as input only a pair of (source, target). This is due to our observation that even if the user will supply a verbal description of the path–method it is difficult to utilize the information contained in the verbal description. One idea of utilizing the verbal description is mentioned in Chapter 4.5.

Note that for the same pair of source and target there may exist several possible semantics. For example for the pair (student, course) we list several possible interpretations.

1. The courses already taken by a student.

2. The courses currently taken by a student.

3. The courses a student is registering for, for the next semester.

   The above three interpretations of the given pair are quite straightforward. There might be some interpretations which are less straightforward. E.g.,

4. The courses taught by all current instructors of a student.

5. The courses offered by the department of a student.

Thus, there is no sense in talking about a "right" or "wrong" path–method generated for a given pair. Different users may want different path–methods for the same pair. Therefore, we talk about *desired* path–method which the user can verify while inspecting it. Hence, we take the approach that the PMG system does not try to guarantee that the generated path–method has the desired semantics as this task seems to be beyond the capabilities of an automatic traversal. Rather, the PMG system suggests to the user a path–method for verification. It is much easier for a user to verify that a path–method fits his needs than to traverse the schema and find it. In case the user is not satisfied with the path–method generated s/he can switch into an interactive mode of operation where the feedback provided by the PMG's unsuccessful trial is utilized by the user to set parameters to better direct subsequent applications of the traversal algorithm. In Chapter 4.5 we discuss specific options for the user in the case that the PMG system did not supply the desired path–method in the first trial.

In our approach we have chosen to generate and present to the user only one path–method. In case that the path–method is not the desired one, the user can supply feedback to set parameters for a subsequent traversal until the desired path–method is obtained. Other possible approaches are to generate and provide the user with all or the k–"best"(for some given integer k) path–methods, so the user can scan all of them till s/he finds the desired one. We have not chosen these approaches since we

think they will overload the user with too much information. This is clearly true for the approach of providing all possible path–methods since their number may be large and, as shown in [P87, PG79, PZ81] in the worst case it may be exponential. For the approach of providing the k–"best" path–methods, the problem of overloading the user with information is less critical. However, in view of the high success ratio achieved for our most successful algorithms, we believe that a typical user will prefer only one path–method which is the desired one in most of the cases. He still can provide feedback for a second try if the result was not satisfactory. We believe that this is preferable to forcing the user to understand k path–methods and to verify which of them, if any, is the desired one. Utilizing the parameters provided by the user for the second try will result in higher chances to find the desired path–method than the chances that it is included in the k–"best" path–methods, since the parameters provide additional constraints to the PMG system.

Hence, we have developed a PMG system which does the following:

1. It supplies the desired path–method in most of the cases; and

2. It enables the user to perform subsequent traversals, incorporating feedback, to find the desired path–method in almost all cases.

Such a system will serve as an important tool to support queries in an OODB.

1.5 Outline of the Dissertation

Chapter 2 discusses a general OODB model and describes definitions and syntax for path–methods in this general OODB model. It also describes a graphical representation for this general OODB model.

Chapter 3 introduces the notion of access weights for OODB schema connections. It motivates using access weights in schema traversal for path–method generation. Specific rules for access weight assignment to schema connections are explained. Several access weight traversal algorithms which generate path–methods using access weights are presented. A sample set of path–methods is defined for our experiments with these algorithms. These path–methods are selected from a large university environment object–oriented database schema which was developed as part of this research. Experimental results using access weight traversal algorithms for path–method generation on the sample of path–methods are presented. Some deficiencies of these algorithms are discussed.

Chapter 4 starts by describing human traversal of an OODB schema to find a particular item of information and introduces the notion of access relevance to support similar automatic traversal and overcome the deficiencies mentioned above. Then, it describes computation of access relevance for an OODB for path–method generation. An algorithm for path–method generation using precomputed access relevance is discussed. Experimental results using the algorithm for path–method generation for the above sample of path–methods are presented. Finally, mechanisms of the

Path–Method Generator, for the cases when a generated path–method in the first phase is not the desired one, are presented.

Chapter 5 describes the computation of access relevance in an OODB. Two triangular norms (t–norms) PRODUCT and MINIMUM are considered as tools for this computation. Efficient algorithms for computation of access relevance using PRODUCT and MINIMUM t–norms in a directed schema graph are presented and their correctness is proven. Finally, a more efficient algorithm for bidirected schemas for MINIMUM weighting function is discussed.

Chapter 6 describes the computation of access relevance in an interoperable multi-OODB (IM–OODB) system. We show how to realize an inter–OODB connection between two component OODBs of an IM–OODB. We describe first the computation of access relevance in an IM–OODB containing only two OODBs. Then an algorithm for the computation of access relevance in an IM–OODB containing many OODBs is presented using a hierarchical approach. That is, the algorithm assumes the precomputation of access relevance for each OODB, but not for the IM–OODB. Then the IM–OODB is modeled as a relatively small graph to which we can apply the previously developed algorithms for a single OODB to compute the access relevance values for the IM–OODB.

Chapter 7 describes the university environment OODB, which was developed during this research. A large subschema of this university OODB containing 52 classes has been used as a testbed for our experiments with various schema traversal algo-

rithms discussed.

Chapter 8 is a design for the Path–Method Generator (PMG) as a module of an object–oriented database. It defines all the classes of PMG using our general OODB model and describes how the PMG works.

Chapter 9 describes the implementation of the PMG System as a module for the VODAK/VML OODB system.

Chapter 10 concludes the dissertation with a summary and an outlook on future research issues.

CHAPTER 2

OODB PATH–METHODS

2.1 A General OODB Model

In this section we discuss the most important features of an OODB model that are necessary for understanding path–method generation. We keep this model general enough to reflect a variety of existing OODB models. In our description we use some of the terminology of the Dual Model [NPGT91, NPGT90, NPGT89, GPN91b] of the VML(= VODAK Modeling Language) [KNBD92] system, but we are not referring to its separation of structural and semantic aspects.

A class can be regarded as a container for objects that are similar in their structure and their semantics in the application. A class description consists of the following four kinds of properties: attributes, user–defined relationships, generic relationships and methods. Attributes specify values of a given datatype while user–defined relationships specify pointers to other classes. Generic relationships are system supported connections between classes. Methods specify operations which can be applied to instances of a class.

Our OODB model contains the generic relationships *roleof*, *categoryof*, *setof* and *memberof*. Both *categoryof* and *roleof* are specialization relationships. The first is used for cases where the subclass and the superclass are in the same context, while

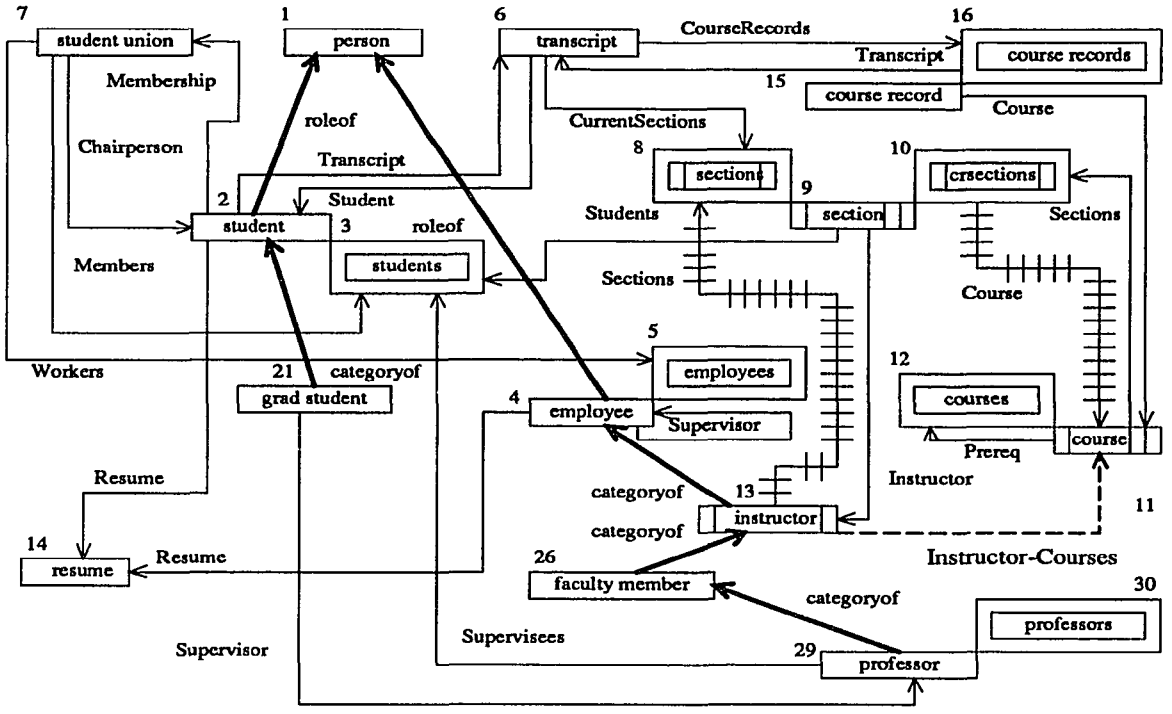

Figure 2.1 A Subschema of a University Database

the second is used when they are in different contexts. Further details on specialization relationships appear in Section 3.1. The generic relationships *memberof* and *setof* are connections between a set class and its member class. In the text of this dissertation the names of classes are printed with lower case bold face letters. The names for attributes, relationships and methods are written in italics with the first letter capitalized. The names of generic relationships are written in lower case italic letters.

To represent an OODB schema graphically, OODINI(= Object-Oriented DIagrams at New jersey Institute) [HGPN92], a graphical schema representation language and system, has been developed. A graphical representation of a subschema

of an OODB from the university domain appears in Figure 2.1. The same schema can be considered as a directed graph G(V, E). The classes are represented as nodes and connections are represented as edges. In OODINI, a rectangle represents a class, and a double line rectangle represents a set class. A set class representation shares one corner with the box that represents its member class, see for example, the class **section** and the class **sections** in Figure 2.1. Note that the subschema of Figure 2.1 contains two different set classes for the class **section**. The class **crsections** represents a set of sections of the same course while the class **sections** represents a set of sections not necessarily of the same course, e.g., the current sections a student is registered for. A thick solid arrow represents specialization generic relationships *categoryof* and *roleof*, and a thin arrow represents a relationship between two classes. Later on we will see that a dotted thick arrow is used for *roleof* with selective inheritance. A path–method is represented by a broken line arrow from the source class to its target class or target attribute. The actual path of the path–method is sometimes highlighted by hatching its classes and connections.

2.2 Path–Methods

In Smalltalk-80 [GR83, PW88] a method is defined as follows. A method is a procedure describing how to perform one of an object's operations; it is made up of a message pattern, a temporary variable declaration, and a sequence of expressions. A method is executed when a message matching its message pattern is sent to an

instance of the class in which the method is defined.

In C++ [S91b, WP88, GOP90] a method is called a "member function". These member functions are similar to methods in Smalltalk-80 with some restrictions, and they are written for a class.

In our OODB model, a method is a program segment with one required parameter of some class, and any number of optional parameters. We will assume that every method returns an instance of a class or a value of a data type. A programming language point of view definition of Path-Methods is given in [NPGT91]. The following definitions are presented based on the discussion in that paper.

- **operation:** If a program segment takes only values of data types as arguments, then we will refer to it as an operation rather than a method, and it will return a value of a data type.

- **computational method:** A *computational method* is a program segment with one required parameter of some class and possibly other optional parameters that makes use of the functionality of the underlying programming language (e.g., C++) but does not modify any stored values outside of its own local memory and returns an instance of a class.

- **primitive method:** A *primitive method* is either a computational method, a relationship, or a generic relationship.

- **method chain:** A *method chain* is either a primitive method or a primitive

method composed with a method chain. Here and later in the dissertation "composed" refers to mathematical composition, i.e, chaining.

- **operation chain:** An *operation chain* is either an operation or an operation composed with an *operation chain.*

- **computational transformer:** A *computational transformer* is program segment that takes as a required argument an instance of a class and returns a value of a data type. Other than that it behaves like a computational method.

- **transformer:** A *transformer* is either a computational transformer or an attribute.

- **transformer chain:** A *transformer chain* is either a transformer or a transformer composed with an operation chain.

  Now a path–method is defined as follows.

- **path–method:** A *path–method* is either a method chain, a transformer chain, or a composition of the two, namely a method chain composed with a transformer chain.

A BNF format will be helpful to better understand the definition of path–method.

<path–method> ::= <method chain>
| <transformer chain>
| <method chain> *o* <transformer chain>

<transformer chain> ::= <transformer>

| <transformer> *o* <operation chain>

<transformer> ::= <computational transformer>
| <attribute>

<operation chain> ::= <operation>
| <operation> *o* <operation chain>

<method chain> ::= <primitive method>
| <primitive method> *o* <method chain>

<primitive method> ::= <computational method>
| <user-defined relationship>
| <generic relationship>

A *traversal path-method* is a path-method which deals only with the part of a path-method that traverses an OODB schema. That is, while a path-method in general may contain, e.g., mathematical operations, a traversal path-method may not, since mathematical operations are localized and do not rely on paths. In this thesis we limit ourselves traversal path-methods. Thus, we add the following definitions:

- **connection:** A connection is either a user-defined relationship or a generic relationship.
- **traversal path-method chain:** A traversal path-method chain is a connection or a connection composed with a traversal path-method chain.
- **traversal path-method:** A traversal path-method is a traversal path-method chain or a traversal path-method chain composed with a transformer.

The following BNF format describes these definitions of traversal path-method.

<connection> ::= <user–defined relationship>
| <generic relationship>

<traversal path–method chain> ::= <connection>
| <connection> *o* <traversal path–method chain>

<traversal path–method> ::= <traversal path–method chain>
| <traversal path–method chain> *o* <transformer>

Now we will discuss the syntax for a traversal path–method in our model. The empty pair of parentheses following the name of a path–method stands for the class in which the path–method is defined. These parentheses may contain additional optional arguments for the path–method. The path–method is described by a list of pairs of the form *property* → *result:*, where *result* is either a class or a data type, meaning that the *property* applied to the *result* at the end of the previous pair yields the *result* of the current pair. When the *result* specifies a set, it is enclosed by {}.

When the *property* is applied to each member of a set it is preceded by the '@' sign otherwise the property is applied to the set as a whole. The colon is used to separate between any two pairs. A path–method *Instructor–Courses* for the class **instructor**, shown graphically in Figure 2.1, is defined as follows.

Instructor–Courses():
Sections ⟶ sections: setof ⟶ {section}:
@memberof ⟶ {crsections}: @Course ⟶ {course}

This path–method finds all the courses being taught by an instructor. First, the path–method finds all the sections taught by the instructor using the relationship *Sections* of the class **instructor** (Figure 2.1). These sections objects are sets of sections. Therefore, the generic relationship *setof* of the class **sections** replaces the

set object by the set of member objects. As a third step, we get a set of course-sections {**crsections**} by applying the generic relationship *memberof* of class **section** to each instance of {**section**}. We can get all the courses for an instructor by applying the relationship *Course* of the class **crsections** to each instance of {**crsections**} yielding a set {**course**} of instances of the class **course**. Note that if an instructor teaches several sections of the same course, this course will appear only once in the result since a set does not allow repetitions.

CHAPTER 3

PATH–METHOD GENERATION USING ACCESS WEIGHTS

In this chapter we will introduce the notion of access weight and discuss traversal algorithms for generating path–methods using access weights.

3.1 Access Weights for an OODB Schema

Following, e.g., VML [KNBD92], GemStone [BOS91], and ORION [K90a, KKS92], we are modeling an OODB schema as a directed graph. Classes are represented as nodes. Directly related classes are connected by a directed edge. Note that a directed graph of a schema may contain cycles. We assign an *access weight* from the range [0, 1] to each connection in the schema. One possible interpretation of such a weight is the frequency of traversal of this connection relative to all the other connections of the class. None of the above OODB models has this enhanced feature of assigning access weights to schema connections.

Let us describe briefly why the Path–Method Generator needs to use access weights. While traversing the schema to generate a path–method, we start with the source class s and consider the different outgoing connections of s. Our observation is that some connections in an OODB schema are more significant than others. In a

series of initial experiments we observed that giving priority to more significant connections, will in many cases produce path–methods more correctly and efficiently than a uniform traversal such as *depth first search* which traverses an arbitrary connection. This was the case even with estimated frequency (significance) values. Although, we have chosen to measure "significance" by frequency of use. We do not exclude other, deeper interpretations. However we have found it much more difficult to map such interpretations into numbers. The access weights associated with the connections emanating from a class should be accumulated during the operation of the OODB for a representative period of time.

In the beginning of the operation of an OODB, access frequency information is not available. Therefore, an application domain expert can suggest initial values to approximate the access weight of each connection. These initial weights will be replaced by experimental weights as they become available. Further research and experiments are needed to determine whether application (view) oriented frequency adjustments will be needed to improve success ratios for automatic path generation. I.e., different applications may use different collections of access weights to service their needs.

The access weight assignments to schema connections are done according to the following rules. For the two generic relationships *setof* and *memberof* and for user–defined relationships the access weights are assigned using Rule 1.

Rule 1: The sum of the weights on the outgoing connections of a class $\sum_{i=1}^{n} W_i =$

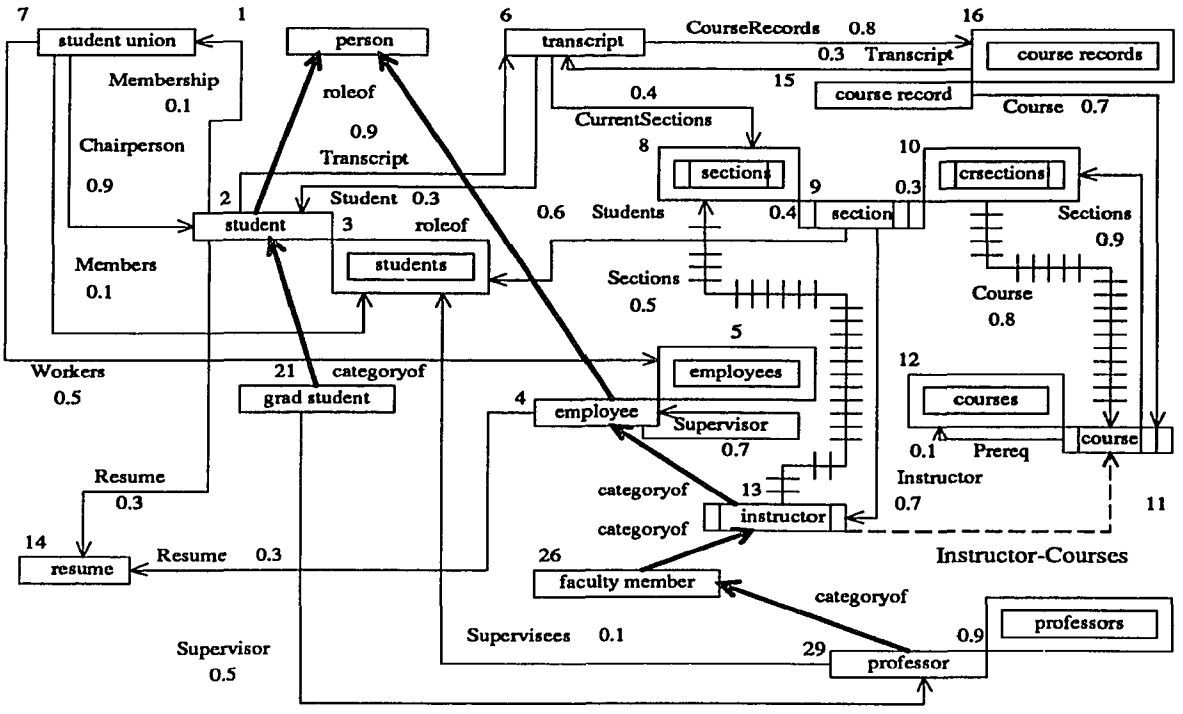

Figure 3.1 A Subschema of a University Database with Access Weights

$0.5*n$, where n is the number of outgoing connections. From this sum, each connection is assigned a weight from [0, 1], reflecting its relative frequency of traversal.

In Figure 3.1, the class **transcript** has three relationships, *CourseRecords* to the class **course_records**, *CurrentSections* to the class **sections** and *Student* to the class **student**, with access weights 0.8, 0.4, and 0.3, respectively, based on their estimated traversal frequencies. Observe that 0.8 + 0.4 + 0.3 = 1.5 = 0.5 * 3, as required by Rule 1. The justification for Rule 1 is as follows. It is not sufficient to assign access weights which add up to 1 to all outgoing connections of one class, although this would appear initially as plausible. This would imply that the connections emanating from a class with few outgoing connections are more significant than the connections out

of a class with many connections. Thus, Rule 1 makes the values of access weights independent of the number of connections of a class. (Actually, this does not matter for access weight algorithms since they decide on the edge to traverse from each node independently. However, such a situation is not acceptable for access relevance algorithms considered in Chapter 4, since the definition of access relevance depends on all access weights of the edges along the path. For uniformity reasons we enforce Rule 1 for the whole dissertation.)

Rule 1 does not work for specialization relationships, and a more elaborate approach is needed for the specialization generic relationships *categoryof* and *roleof*. This is due to the fact that while other relationships are used for traversal of the schema these two relationships are used for inheritance. That is, if a class A is a subclass of a class B then one can traverse from the class A to all the properties of class B in addition to the properties listed explicitly with class A. Hence, while this looks like a traversal of the specialization relationship from A to B followed by the traversal to a property of B, those two traversals are of a different nature. For a specialization relationship the weight should not reflect its frequency of use but the fact that the properties of the superclass are immediately available at the subclass without any change of their weights since this is the desired effect of inheritance. The first idea which comes to mind is to assign to specialization links a weight of 1.

However, here we need to distinguish between the two specialization relationships. As mentioned above these two relationships differ in their meanings in terms of mod-

eling. Class A is *categoryof* (*roleof*) of class B if class A is in the same (different) context of class B. This implies a difference in the inheritance mechanism for these two relationship. For *categoryof* there is an automatic inheritance of all properties of the superclass to the subclass.

This is not always true for *roleof*. For *roleof* we may want selective inheritance of some of the properties of the superclass to the subclass. Inheritance of selected properties can be implemented through a special path–method which utilizes the *roleof* connection. For example, the *roleof* connection from **assistant** to **grad_student** can be used to inherit only the transcript, since some information on the academic standing of a graduate student is used to determine his/her eligibility for assistantship. On the other hand we are not interested in any further details of the function of a graduate student while dealing with his employment capacity as assistant. For the *roleof* connection from **admin_appt** (which represents administrative appointment) to professor we need not inherit any specific property, just keep the connection to enable later coding of methods for the retrieval of information on the formal appointments of an academic administrator for the case that s/he leaves her/his administrative appointment, since this administrator does not function at the present as a professor.

Alternatively, we may actually need to inherit all properties of the superclass through the *roleof* connection. For example, in the schema of Figure 3.2, we need full inheritance for the following *roleof* connections: **former_student** *roleof* **person**, **employee** *roleof* **person**, **dept_chair_person** *roleof* **professor** and **phd_advisor**

roleof **professor.** For all these *roleof* connections we need to inherit all the properties of the superclass inspite of the change of context between the subclass and the superclass. The reason is that in all these cases in spite of the change of context, the extra properties of the subclass are additive to the properties of the superclass rather than replacing them. For example, a student continues also to function as a person and a chairperson continues to function as a professor.

Thus, for specialization relationships we will have three choices:

1. *categoryof* (always full inheritance)

2. *roleof* with selective inheritance

3. *roleof* with full inheritance

As a matter of fact, this classification of specialization relationships corresponds to the classification found in the Dual Model [NPGT91] and the VODAK/VML prototype [KNBD92, K90b] and is implied by the separation of structural and semantic aspects of the OODB in these models. To distinguish graphically between the two kinds of *roleof* connections we use a solid heavy arrow for the case of full inheritance (as for *categoryof*) and dotted heavy arrow for selective inheritance (see Figure 3.2). We are now in a position to specify access weight assignments for *roleof/categoryof.*

Rule 2a: An access weight of 1.0 is assigned to each *categoryof* connection and to each *roleof* connection with full inheritance. An access weight of 0.0 is assigned to each *roleof* connection with selective inheritance. To enable selective inheritance, in

spite of the 0.0 weight, copy the properties of the superclass, that should be inherited, to the subclass.

An alternative to Rule 2a would be to assign *roleof* with selective inheritance a weight between 0 and 1. However, Rule 2a is better for the following reason. The fraction weight would weaken the chances of inheritance for all properties, while we need to keep full strength inheritance for the selected properties and block totally the inheritance of the rest of the properties. The fraction weight is serving neither of these needs and thus it is not acceptable.

Rule 2a for full inheritance implies that the properties of the superclass are available at the subclass without decreasing the access weight. However, Rule 2a has the following disadvantage. It enables the traversal of a specialization connection as a regular connection rather than an inheritance link. I.e., it enables traversal which stops at the superclass as a target, rather than continuing to use one of its properties. But there is no reason for such traversal since it does not lead to any meaningful information not available at the subclass. In fact it leads to a "dilution" of information. An example of a path which ends with an inheritance link is shown in Chapter 4.4 to yield undesired path–methods. We would like to block traversals which end with specialization connections, while still enabling the inheritance of properties. But an access weight of 1.0 enables such a traversal and furthermore gives it high priority. Thus, we introduce an alternative rule.

Rule 2b: An access weight of 0.0 is assigned to all *categoryof* and *roleof* connections.

For cases of *categoryof* and *roleof* with full inheritance copy all the properties of the superclass to the subclass in the schema's underlying graph to achieve the effect of inheritance. For cases of *roleof* with selective inheritance copy only the selected properties of the superclass to the subclass.

Rule 2b allows the traversal of the schema's underlying graph exactly as discussed before, but it practically disallows unwanted traversal of specialization connections (see demonstration in Chapter 4.4). Thus, Rule 2b is considered a better rule for the traversal of the schema's underlying graph. One disadvantage of Rule 2b is that the schema graph becomes more dense. The schema visible to the user, however, does not change. The changes of a schema graph according to Rule 2a and Rule 2b are demonstrated in Chapter 6.

Rule 2b raises the question what weights to assign to the inherited properties. One solution is to copy the weights of properties from the superclass to the subclasses so the impact will be that of using Rule 2a. This is the approach we used in this dissertation.

However, this solution ignores the fact that the frequencies of the inherited properties may differ between a subclass and the superclass and among several subclasses of the same superclass. Furthermore, by copying the weights from the superclass to a subclass we give up the possibility of balancing the weights of the inherited properties and of the properties defined at the subclass according to the relative frequencies of all these connections.

Thus, one may take the approach of keeping different weights for the inherited properties of each subclass according to their frequency of use. It is a matter of tradeoff between the effort of computing and maintaining the extra information for separate weights of the inherited properties and the better modeling capabilities and their impact on path–method generation. There is a need for further research to determine the impact of the more accurate representation of the frequencies of the inherited properties on the success ratio of generating the desired path–methods.

Another issue which was not considered in our treatment is traversing existing path–method during the effort to generate a new path–method. A schema may already have some path–methods provided by the designers and users of the schema as well as some already generated by the PMG system and added to the schema. Such a path method can be used as a connection in the definition of a new path–method. Using such a path–method rather than the list of its connections will shorten and simplify the definition of the new path–method and is preferred. In order to use connections representing path–methods in our traversal algorithms we need to define weights for these connections. One can accumulate the frequency of use of the existing path–methods in the schema to define the weights of the proper connections similar to the treatment of the other connections in the schema.

Another alternative with regards to the access weight assigned for a connection of a path–method is to use the access relevance value for the corresponding path. For example, for the PRODUCT weighting function it is the product of all the access

weights of the edges of the path. However, we did not choose this option due to some difficulties with it. One is technical the value assigned to a path–method connection will be smaller than the value for the other connections emanating from a class for both weighting function user and specially for the PRODUCT weighting function. This will give low chance of using the path–method connection while we actually want to give it a high chance. The other difficulty is conceptual if the choice between other connections is according to frequency of use than this connection once it is added to the schema should get the same treatment rather than utilizing a weight involved in the creation of the path–method.

A few more comments on the issue of using path–methods as connections in other path–methods. Upto now we considered the accumulation of frequencies to be independent of whether a connection is used as part of a path–method or not. However, when a connection is used as part of a path–method one can accumulate information about the frequency of using a connection depending on the connection used before it. We call such values *dependent frequencies of use.* These dependent frequencies can be utilized when considering the next connection to be picked, since at this point we know the previous connection in the path–method being generated. Further research is needed to determine the impact of such additional information on the success ratio of generation of path–methods.

3.2 Access Weight Traversal Algorithms

An *access weight traversal algorithm* is a traversal algorithm which uses access weights on the connections of an OODB schema for guiding path–method generation. These traversal algorithms generate desired path–methods by selecting a connection of a class with maximum access weight at each step of traversal. The connection of a class with maximum access weight is assumed to be "more significant" and preferred for generating a desired path–method.

The literature reports two major approaches to traversal of graphs without weights on their edges. The aggressive approach, represented by depth first search (DFS), goes forward as quickly as possible without first checking the near vicinity. When necessary it backtracks as little as possible before rushing forward again. The conservative approach, represented by breadth first search (BFS), does not proceed forward before an exhaustive search of the close vicinity. Both these approaches have their advantages and disadvantages explored in the Artificial Intelligence and Algorithms literature. Some traversal algorithms are hybrids between these two approaches (see e.g. [RC78]).

A natural idea to utilize the access weights to guide the traversal is to use them to set priorities in the choice of traversing edges emanating out of a vertex. That is, these edges will be considered in decreasing order of the access weights. When this enhancement is added to DFS we obtain the known best first search [CM81] traversal algorithm. In this algorithm we choose to traverse out of a vertex through the edge of highest access weight, not traversed yet.

The *best breadth first search* algorithm is a similar enhancement for breadth first search. The edges emanating out of a vertex are considered in descending order of their access weights. The advantage of each of these algorithms is that it selects "more significant" connections first and then "less significant" connections.

Experiments comparing the above four traversal algorithms are reported later in this chapter. The enhanced versions, preferring more significant connections, will be shown to perform better than their counterparts which choose to traverse an emanating edge arbitrarily.

We note that breadth first search and best breadth first search can find only path–methods with a shortest path from the source to the target, where length is measured in number of edges in the path. As our experiments will show, these two algorithms provide the best results. Path–methods with a path from the source to the target that is longer than the shortest path can only be generated by DFS and Best First Search. In order to obtain such path–methods we suggest another enhancement which involves a union operation.

The *breadth first search* ∪ *best first search* algorithm calls *breadth first search* and also calls *best first search* and generates two path–methods. When the two path–methods are different, both are provided to the user for inspection. A user selects the desired one of these two path–methods. On the other hand when the two path–methods are equal, only one will be displayed. The advantage of this algorithm is that it combines results of *breadth first search* and *best first search* and is more

likely to generate the path–method that will serve the user best.

The *best breadth first search* ∪ *best first search* algorithm calls the *best breadth first search* algorithm and also the *best first search* and generates two path–methods. The treatment is identical to that of *breadth first search* ∪ *best first search.*

3.3 A Sample Set of Path–Methods for a University Database

We have performed a large number of experiments on a subschema of our university OODB. All the classes of this subschema are discussed in detail in Chapter 7. It is quite difficult to comprehend this subschema, although it is much smaller than the total OODB schema. It comprises only about a third of the university database designed by our group to model part of the university environment [CT90, WA90, K90c, B90, D91b, P91a, P91b]. It is difficult for a user to traverse such a schema to find path–methods by himself, or even worse, find them without the graphical representation of the schema.

Classes related to student, professor, course, employee, resume, and university are shown in Figure 4. We have experimented with several access weight traversal algorithms in an effort to generate a sample of 50 path–methods. The requirements for these path–methods were defined using the schema but independently and before the design of the algorithms. The first eighteen of the 50 path–methods are shown in Figure 3.2 as dashed line arrows. The paths of these eighteen path–methods are not

highlighted in Figure 3.2. For each path–method we list the source and the target and use a verbal description of its purpose. This verbal description identifies the "semantics" of the path–method.

1. The path–method *Student–Courses* finds all the courses already taken by a student.

   source: **student** *target:* {course}

   Student–Courses ():
   Transcript ⟶ transcript: CourseRecords ⟶ course_records:
   setof ⟶ {course_record}: @Course ⟶{course}

2. The path–method *Grad_student–Instructors* finds all the current instructors for a graduate student.

   source: **grad_student** *target:* {instructor}

   Grad_student–Instructors ():
   Transcript ⟶ transcript: CurrentSections ⟶ sections:
   setof ⟶ {section}: @Instructor ⟶ {instructor}

   Note that the relationship *Transcript* is inherited from **student**.

3. The path–method *Instructor–Courses* finds all the courses currently being taught by an instructor.

   source: **instructor** *target:* {course}

   Instructor–Courses():
   Sections ⟶ sections: setof ⟶ {section}:
   @memberof ⟶ {crsections}: @Course ⟶ {course}

4. The path–method *Instructor–Students* finds all the students, in the sections an instructor is teaching.

   source: **instructor** *target:* {student}

Instructor–Students ():
Sections ⟶ sections: setof ⟶ {section}:
@Students ⟶ {students}: @setof ⟶ {{student}}:
union ⟶ {student}

At line 3 the intermediate result is a set of sets, one for each section. Thus if a student is registered for two sections, s/he will appear twice. In line 4 the operation **union** is performed on the set of sets to get one set as an answer to the user request. In this way a student will appear once even if s/he is registered for two sections of this instructor. This is not a traversal path–method because of the use of the **union** operator. Therefore, the last step cannot be generated automatically. The set operations in an OODB are discussed in [AR91, RB92]. The object–oriented knowledge–base model Jasmine [I93] supports set oriented access queries.

5. The path–method *Instructor–TeachingEvaluation* finds the set of teaching evaluations of all the instructors who are teaching a given course. Note that this traversal path–method is terminated by an attribute.

source: **course** *target:* {TeachingEvaluation}

Instructor–TeachingEvaluations ():
Sections ⟶ crsections : setof ⟶ {section}:
@Instructor ⟶ {instructor}: @TeachingEvaluation ⟶ {REAL}

One could define *MaxTeachEval*, a path–method which is not a traversal path–method, by concatenating the above path–method with **max** ⟶ REAL.

6. The path–method *Course–Students* finds all the students, currently registered for a given course.

source: **course** *target:* {student}

Course–Students():
Sections ⟶ crsections : setof ⟶ {section}:
@Students ⟶ {students}: @setof ⟶ {{student}}:
union ⟶ {student}

7. The path–method *Professor–Refereed_conference_papers* finds all the refereed conference papers of a professor.

source: **professor** *target:* {refereed_conference_paper}

Professor–Refereed_conference_papers ():
Resume ⟶ resume: Publications ⟶ publications:
Conferences ⟶ refereed_conference_papers:
setof ⟶ {refereed_conference_paper}

8. The path–method *Research_assistant–Bachelor_degrees* finds the bachelor degrees of a research assistant.

source: **research_assistant** *target:* {bachelor_degree}

Research_assistant–Bachelor_degrees ():
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
BachelorDegrees ⟶ bachelor_degrees: setof ⟶ {bachelor_degree}

9. The path–method *University–Dept_phd_advisors* finds all the Ph.D. Advisors in the university. (I.e., the directors of the Ph.D. programs of their respective departments.)

source: **university** *target:* {dept_phd_advisor}

University–Dept_phd_advisors ():
Colleges ⟶ colleges: setof ⟶ {college}:
@Departments ⟶ {departments}: @setof ⟶ {{department}}:
@DeptPhdAdvisor ⟶ {{dept_phd_advisor}}: **union** ⟶ {dept_phd_advisor}

10. The path–method *University–College_deans* finds all the college deans in the university.

source: **university** *target:* {college_dean}

University–College_deans ():
Colleges ⟶ colleges: setof ⟶ {college}:
@CollegeDean ⟶ {college_dean}

11. The path–method *College–Research_assistants* finds all the research assistants in a college.

source: **college** *target:* {research_assistant}

College–Research_assistants ():
Departments ⟶ departments: setof ⟶ {department}:
@Professors ⟶ {professors}: @setof ⟶ {{professor}}:
@ResearchAssistants ⟶ {{research_assistants}}:
@setof ⟶ {{{research_assistant}}}:
union ⟶ {{research_assistant}}: **union** ⟶ {research_assistant}

12. The path–method *Course–Refereed_conference_papers* finds all the refereed conference papers of each of the instructors who are teaching a given course.

source: **course** *target:* {refereed_conference_paper}

Course–Refereed_conference_papers ():
Sections ⟶ crsections: setof ⟶ {section}:
@Instructor ⟶ {instructor}: @Resume ⟶ {resume}:
@Publications ⟶ {publications}: @Conferences ⟶ {refereed_conference_papers}:
@setof ⟶ {{refereed_conference_paper}}:

13. The path–method *Undergrad_student–Sections* finds all the current sections taken by a given undergraduate student.

source: **undergrad_student** *target:* {section}

Undergrad_student–Sections ():
Transcript ⟶ transcript: CurrentSections ⟶ sections:
setof ⟶ {section}

14. The path–method *Professor–Courses* finds all the courses currently being taught by a professor.

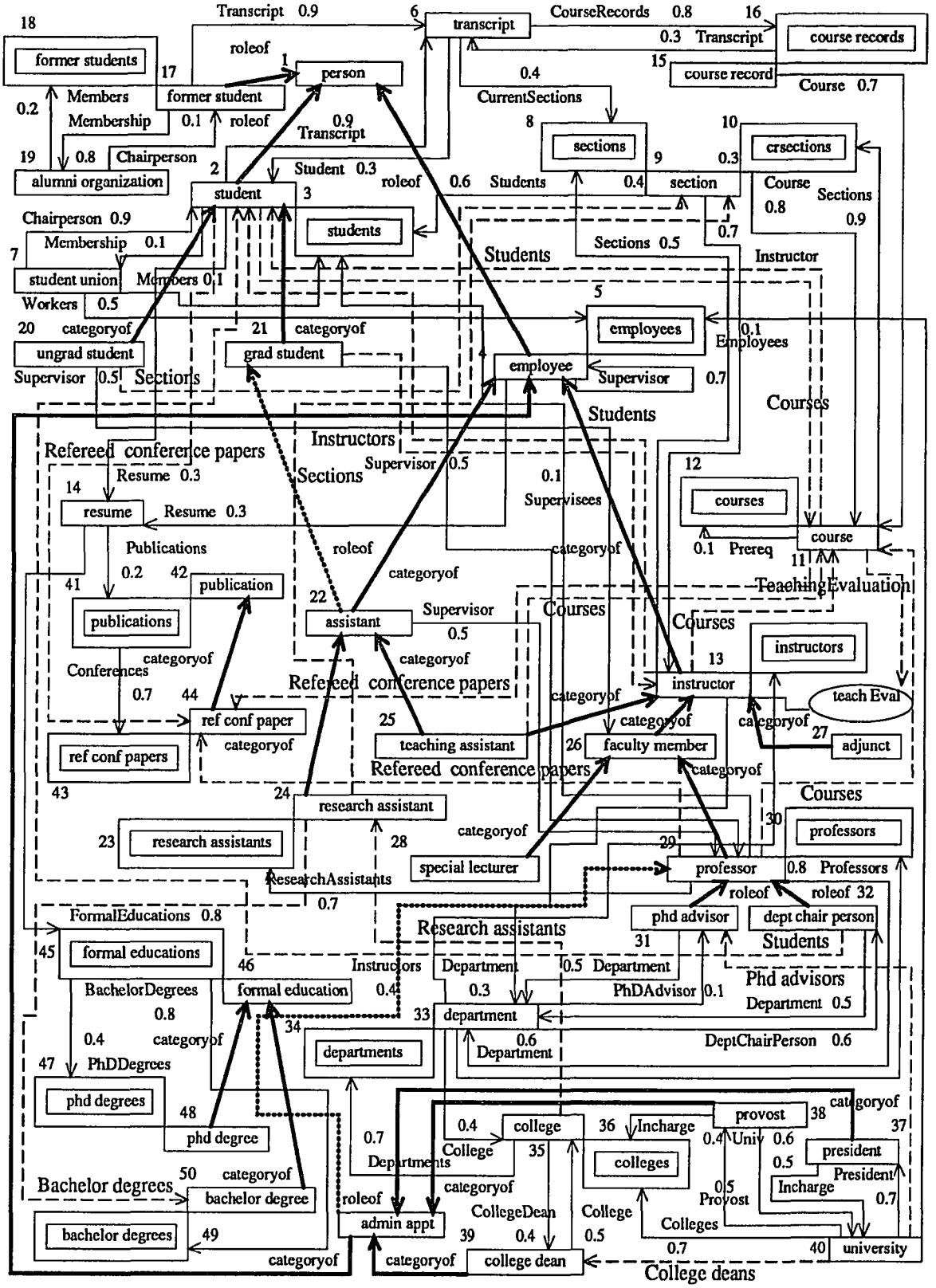

Figure 3.2 A Larger Subschema of a University Database

source: **professor** *target:* {course}

Professor–Courses ():
Sections ⟶ sections: setof ⟶ {section}:
@memberof ⟶ {crsections}: @Course ⟶ {course}

15. The path–method *Teaching_assistant–Courses* finds all the courses taught by a teaching_assistant.

source: **teaching_assistant** *target:* {course}

Teaching_assistant–Courses ():
Sections ⟶ sections: setof ⟶ {section}:
@memberof ⟶ {crsections}: @Course ⟶ {course}

16. The path–method *Dept_chair_person–Students* finds all the students registered in sections currently being taught by a dept_chair_person.

source: **dept_chair_person** *target:* {student}

Dept_chair_person–Students ():
Sections ⟶ sections: setof ⟶ {section}:
@Students ⟶ {students}: @setof ⟶ {{student}}:
union ⟶ {student}

17. The path–method *Research_assistant–Sections* finds all the sections currently taken by a given research_assistant.

source: **research_assistant** *target:* {section}

Research_assistant–Sections ():
Transcript ⟶ transcript: CurrentSections ⟶ sections:
setof ⟶ {section}

18. The path–method *Student–Refereed_conference_papers* finds all the refereed conference papers of a student.

source: **student** *target:* {refereed_conference_paper}

Student–Refereed_conference_papers ():
Resume ⟶ resume: Publications ⟶ publications:
Conferences ⟶ refereed_conference_papers:
setof ⟶ {refereed_conference_paper}

19. The path–method *Grad_student–College_dean* finds the dean of the college where a graduate student is currently enrolled.

source: **grad_student** *target:* college_dean

Grad_student–College_dean ():
Supervisor ⟶ professor: Department ⟶ department:
College ⟶ college: CollegeDean ⟶ college_dean

20. The path–method *Former_student–Courses* finds all the courses taken by a former student.

source: **former_student** *target:* {course}

Former_student–Courses ():
Transcript ⟶ transcript: Course_records ⟶ course_records:
setof ⟶ {course_record}: @Course ⟶ {course}

21. The path–method *College–Instructors* finds all the instructors of a college.

source: **college** *target:* {instructor}

College–Instructors ():
Departments ⟶ departments: setof ⟶ {department}:
@Instructors ⟶ {instructors}: @setof ⟶ {{instructor}}:
union ⟶ {instructor}

22. The path–method *Course–Bachelor_degrees* finds bachelor degrees of all the instructors teaching a given course.

source: **course** *target:* {bachelor_degree}

Course–Bachelor_degrees ():
Sections ⟶ crsections: setof ⟶ {section}:

@Instructor ⟶ {instructor}: @Resume ⟶ {resume}:
@FormalEducations ⟶ {formal_educations}:
@BechelorDegrees ⟶ {bachelor_degrees}:
@setof ⟶ {{bachelor_degree}}: **union** ⟶ {bachelor_degree}

23. The path–method *Research_assistant–College* finds the college of a research_assistant.

source: **research_assistant** *target:* college

Research_assistant–College ():
Supervisor ⟶ professor: Department ⟶ department:
College ⟶ college

24. The path–method *Special_lecturer–Bachelor_degrees* finds bachelor_degrees of a special lecturer.

source: **special_lecturer** *target:* {bachelor_degree}

Special_lecturer–Bachelor_degrees ():
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
BachelorDegrees ⟶ bachelor_degrees: setof ⟶ {bachelor_degree}

25. The path–method *College–Professors* finds all the professors of a college.

source: **college** *target:* {professor}

College–Professors ():
Departments ⟶ departments: setof ⟶ {department}:
@Professors ⟶ {professors}: @setof ⟶ {{professor}}:
union ⟶ {professor}

26. The path–method *Grad_student–Bachelor_degrees* finds all the bachelor degrees for a graduate student.

source: **grad_student** *target:* {bachelor_degree}

Grad_student–Bachelor_degrees ():
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
BachelorDegrees ⟶ bachelor_degrees: setof ⟶ {bachelor_degree}

27. The path–method *Department–Courses* finds all the courses being taught by an academic department.

source: **department** *target:* {course}

Department–Courses ():
Instructors ⟶ instructors: setof ⟶ {instructor}:
@sections ⟶ {sections}: @setof ⟶ {{section}}:
@memberof ⟶ {{crsections}}: @course ⟶ {{course}}:
union ⟶ {course}

28. The path–method *Grad_student–Dept_chair_person* finds the chair person of a graduate student's department.

source: **grad_student** *target:* dept_chair_person

Grad_student–Dept_chair_person ():
Supervisor ⟶ professor: Department ⟶ department:
DeptChairPerson ⟶ dept_chair_person

29. The path–method *Instructor–College_dean* finds the dean of the college of an instructor.

source: **instructor** *target:* college_dean

Instructor–College_dean ():
Department ⟶ department: College ⟶ college:
CollegeDean ⟶ college_dean

30. The path–method *University–Departments* finds all the departments in a university.

source: **university** *target:* {department}

University–Departments ():
Colleges ⟶ colleges: setof ⟶ {college}:
@Departments ⟶ {departments}: @setof ⟶ {{department}}:
union ⟶ {department}

31. The path–method *President–Phd_degrees* finds all the PhD degrees of the president of the university.

source: **president** *target:* {phd_degree}

President–Phd_degrees ():
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
PhdDegrees ⟶ phd_degrees: setof ⟶ {phd_degree}

32. The path–method *Provost–Bachelor_degrees* finds all the bachelor degrees of the provost of a university.

source: **provost** *target:* {bachelor_degree}

Provost–Bachelor_degrees
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
PhdDegrees ⟶ phd_degrees: setof ⟶ {phd_degree}

33. The path–method *Department–Students* finds all the students registered in the courses currently being taught by a department.

source: **department** *target:* {student}

Department–Students ():
Instructors ⟶ instructors: setof ⟶ {instructor}:
@Sections ⟶ {sections}: @setof ⟶ {{section}}:
@Students ⟶ {{{student}}}: **union** ⟶ {{student}}:
union ⟶ {student}

34. The path–method *Dept_chair_person–Publications* finds all the publications of the department chairperson.

source: **dept_chair_person** *target:* {publication}

Dept_chair_person–Publications ():
Resume ⟶ resume: Publications ⟶ publications:
setof ⟶ {publication}

35. The path–method *Ungrad_student–Department* finds the department of an undergraduate student.

source: **Ungrad_student** *target:* {department}

Ungrad_student–Department ():
Supervisor ⟶ faculty_member: department ⟶ department

36. The path–method *College_dean–Formal_educations* finds the degree programs successfully terminated by a college dean.

source: **college_dean** *target:* {formal_education}

College_dean–Formal_educations ():
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
setof ⟶ {formal_education}

37. The path–method *College_dean–Refereed_conference_papers* finds refereed conference papers of a college dean.

source: **college_dean** *target:* {refereed_conference_paper}

College_dean–Refereed_conference_papers ():
Resume ⟶ resume: Publications ⟶ publications:
Conferences ⟶ refereed_conference_papers:
setof ⟶ {refereed_conference_paper}

38. The path–method *Research_assistant–Course_records* finds the course records of courses taken by a research assistant.

source: **research_assistant** *target:* {course_record}

Research_assistant–Course_records ():
Transcript ⟶ transcript: CourseRecords ⟶ course_records:
setof ⟶ {course_record}

39. The path–method *Adjunct–Formal_educations* finds all the formal educations

completed by an adjunct.

source: **adjunct** *target:* {formal_education}

Adjunct–Formal_education ():
Resume ⟶ resume: FormalEducations ⟶ formal_educations:
setof ⟶ {formal_education}

40. The path–method *Teaching_assistant–Sections* finds all the sections taught by a teaching assistant.

source: **teaching_assistant** *target:* {section}

Teaching_assistant–Sections ():
Sections ⟶ sections: setof ⟶ {section}

41. The path–method *Teaching_assistant–Course_records* finds the course records of courses taken by a teaching assistant.

source: **teaching_assistant** *target:* {course_record}

Teaching_assistant–Course_records ():
Transcript ⟶ transcript: Course_records ⟶ course_records:
setof ⟶ {course_record}

42. The path–method *College–Dept_chair_persons* finds all the chairpersons of a college.

source: **college** *target:* {dept_chair_person}

College–Dept_chair_persons ():
Departments ⟶ departments: setof ⟶ {department}:
@DeptChairPerson ⟶ {dept_chair_person}

43. The path–method *University–Professors* finds all the professors in a university.

source: **university** *target:* {professor}

University–Professors ():

Colleges ⟶ colleges: setof ⟶ {college}:
@Departments ⟶ {departments}: @setof ⟶ {{department}}:
@professors ⟶ {{professors}}: @setof ⟶ {{{professor}}}:
union ⟶ {{professor}}: **union** ⟶ {professor}:

44. The path–method *College–Dept_phd_advisors* finds all the phd_advisors of a college.

source: **college** *target:* {dept_phd_advisor}

College–Dept_phd_advisors ():
Departments ⟶ departments: setof ⟶ {department}:
DeptPhdAdvisors ⟶ {dept_phd_advisor}

45. The path–method *Research_assistant–Dept_phd_advisor* finds a research assistant's phd_advisor.

source: **research_assistant** *target:* **dept_phd_advisor**

Research_assistant–Dept_phd_advisor ():
Supervisor ⟶ professor: Department ⟶ department:
DeptPhdAdvisor ⟶ dept_phd_advisor

46. The path–method *Teaching_assistant–College* finds a teaching assistant's college.

source: **teaching_assistant** *target:* college

Teaching_assistant–College ():
Department ⟶ department: College ⟶ college

47. The path–method *Teaching_assistant–Dept_chair_person* finds the chair person of a teaching assistant's department.

source: **teaching_assistant** *target:* {dept_chair_person}

Teaching_assistant–Dept_chair_person ():
Department ⟶ department: DeptChairPerson ⟶ dept_chair_person

48. The path–method *Department–Sections* finds all the sections offered by a depart-

ment.

source: **department** *target:* {section}

Department–Sections ():
Instructors ⟶ instructors: setof ⟶ {instructor}:
@Sections ⟶ {sections}: setof ⟶ {{section}}:
union ⟶ {section}

49. The path–method *Teaching_assistant–Dept_phd_advisors* finds the Ph.D. advisor of a teaching assistant's department.

source: **teaching_assistant** *target:* { dept_phd_advisor }

Teaching_assistant–Dept_phd_advisors ():
Department ⟶ department: DeptPhdAdvisor ⟶ dept_phd_advisor

50. The path–method *Research_assistant–College_dean* finds the college_dean of a research_assistant.

source: **research_assistant** *target:* college_dean

Research_assistant–College_dean ():
Supervisor ⟶ professor: department ⟶ department:
college ⟶ college: CollegeDean ⟶ college_dean

3.4 Experimental Results of Sample Set of Path–Methods

We compared the results of several access weight traversal algorithms with depth first search and breadth first search [AHU83, M89]. The results are shown in Table 3.1. The comparison is based on the path–method generated. We show also the path length (PL) and the number of visited nodes (NVN). A circle around PL and NVN in Table 3.1 indicates a case where the path–method obtained is not the desired one.

No.	Depth First Search		Breadth First Search		Best First Search		Best Breadth First Search		Breadth First Search ∪ Best First Search		Best Breadth First Search ∪ Best First Search	
	PL	NVN	PL	NVN	PL	NVN	PL	NVN	PL	NVN	PL	NVN
1	(5)	(38)	4	22	4	9	4	22	4 *	31 *	4 *	29 *
2	(6)	(22)	(3)	(29)	(7)	(17)	(3)	(25)	(3)	(46)	(3)	(42)
3	(9)	(25)	4	30	(8)	(36)	4	27	4	66	4	63
4	(9)	(24)	(4)	(28)	4	9	4	26	4	37	4 *	35 *
5	4	7	4	7	4	7	4	7	4 *	14 *	4 *	14 *
6	(10)	(27)	4	12	(10)	(47)	4	11	4	59	4	58
7	(13)	(23)	4	36	(6)	(25)	4	36	4	61	4	61
8	(9)	(13)	4	34	(16)	(38)	4	32	4	72	4	70
9	(6)	(40)	5	26	5	37	5	22	5	63	5	48
10	(8)	(11)	3	26	3	38	3	16	3 *	64 *	3 *	54 *
11	(5)	(8)	(5)	(27)	(5)	(16)	(5)	(24)	(5)	(43)	(5)	(51)
12	(12)	(22)	7	38	(12)	(43)	7	37	7	81	7	80
13	(5)	(17)	3	23	(6)	(14)	3	22	3	37	3	36
14	(14)	(32)	4	39	(6)	(39)	4	33	4	78	4	72
15	(7)	(19)	(4)	(32)	(4)	(9)	4	38	(4)	(39)	(4)	(56)
16	(7)	(12)	(2)	(14)	(11)	(38)	(2)	(14)	(2)	(52)	(2)	(52)
17	(8)	(17)	(3)	(30)	(3)	(23)	3	23	(3)	(53)	3	46
18	(7)	(14)	4	25	(16)	(40)	4	24	4	65	4	64
19	(9)	(34)	4	38	(10)	(38)	4	36	4	76	4	74
20	(5)	(7)	4	16	4	9	4	13	4 *	25 *	4 *	22 *
21	(6)	(8)	4	16	(12)	(31)	(3)	(20)	4	47	4 *	34 *
22	(14)	(21)	7	37	(12)	(38)	7	35	7	75	7	73
23	(8)	(28)	3	26	(6)	(41)	3	26	3	67	3	67
24	(13)	(17)	4	33	(12)	(36)	4	31	4	69	4	67
25	(6)	(10)	(4)	(17)	(6)	(36)	(2)	(9)	(4)	(53)	(4)	(34)
Average	8.2	19.84	4.04	26.36	7.68	28.56	4.04	24.2	4.12	55.32	4.04	53.2

Table 3.1 Results of Access Weight Traversal Algorithms

In fact, Table 3.1 shows detailed results for the first 25 path–methods.

Depth first search performed very badly. It found only 1 of the 25 path–methods and this was a case of a unique path in the schema from the source to the target. This is the only case when *depth first search* must generate a desired path–method assuming the necessary information is in the schema at all. Thus, it is obvious that the arbitrary traversal of *depth first search* can not be used for path–method generation.

On the other hand, *breadth first search* performed relatively well, finding 18 out of 25 path–methods. The success in these cases is due to the fact that the corresponding paths are the only shortest paths (i.e., each has a minimum number of edges) from the source to the target. Breadth first search always finds such paths. For 5 cases BFS finds a path shorter than the desired one. In the remaining 2 cases it found a path of equal length to the desired one. Hence the performance of BFS is dictated only by its property of finding shortest paths. It is guaranteed to find the desired path if it is the only shortest path. It is guaranteed to fail, if the desired path is not a shortest path. In addition, it may fail if there exist more than one shortest paths, because it does not provide any mechanism which can generate a particular shortest path. Its success in cases of several shortest paths is a matter of coincidence.

Now let us turn to access weight traversal algorithms. The *best first search* algorithm found 6 out of 25 path–methods. This is an improvement over DFS, due to the choice of high access weight edges over choice of arbitrary edges, confirming our expectation. However, those results are quite disappointing. We can conclude that

a path–method containing the significant connections out of each class on the path often may not be the desired one. It seems that in most cases at least one edge on the desired path has not the highest access weight out of the edges emanating from its class. The greedy best first search algorithm fails most of the times due to the lack of a look–ahead property. An algorithm using a look–ahead property is discussed in the next chapter. The *best breadth first search* algorithm found 20 out of 25 path–methods. It shows just a slight improvement over the corresponding BFS, again due to the processing of the edges emanating out of each vertex in decreasing order of their access weights. However, it found all the path–methods with a shortest path, that is, all the path–methods which such a search can potentially find. Hence, the impact of the enhancement over BFS is slight only since BFS was itself performing that well.

We see that the *best breadth first search* gives the best results out of the four algorithms. The bottom row in Table 3.1 shows the average path length and average number of nodes visited for each algorithm. We see that *best breadth first search* is also the most efficient of the last three algorithms as measured in terms of the number of nodes visited until the desired path–method is generated. (DFS is more efficient, but its very low performance makes this efficiency irrelevant.) However, *best breadth first search* misses some of the desired path–methods due to its deficiency to generate path methods not of shortest path. The following two enhancements were introduced to generate these missing path–methods which may be generated by best first search.

The *breadth first search* ∪ *best first search* found 19 desired path–methods out of 25 path–methods. A '*' indicates cases when the two generated path–methods are equal. It found only one more desired path–methods than the *breadth first search*. This path–method has a shortest path. The *best breadth first search* ∪ *best first search* found 20 out of 25 path–methods. It did not find any more path–method than *best breadth first search* algorithm. The results of the last two enhancements are disappointing. As a matter of fact, *best breadth first search* found one more path–method than *breadth first search* ∪ *best first search*, and it did so much faster. Thus it is not recommended to use the last algorithm. Different techniques are needed to generate path–methods without a shortest path. See the next chapter for such techniques. The *best breadth first search* ∪ *best first search* found the same number as *best breadth first search*. By looking at results of all the algorithms discussed above, the overall conclusion is that the algorithm *best breadth first search* can be considered a good candidate for developing traversal algorithms.

To explore the path–method generation in more detail we experimented with these traversal algorithms on the sample of 50 path–methods. Results are shown in Table 3.2. The *best breadth first search* algorithm found 43 out of 50 path–methods. The *breadth first search* ∪ *best first search* found 42 and *best breadth first search* ∪ *best first search* found 43 out of 50 path–methods. By looking at Table 3.2, we can say that results for the larger sample correspond to the results for the smaller sample earlier. We believe that these results are typical for a wide range of schemas and

path–methods. However, further experiments are needed to verify this conjecture.

No.	Name of the Algorithm	Generated Path-Methods (out of 50)	
		Desired	Undesired
1.	Depth First Search	5	45
2.	Breadth First Search	41	9
3.	Best First Search	13	37
4.	Best Breadth First Search	43	7
5.	Breadth First Search $\cup$ Best First Search	42	8
6.	Best Breadth First Search $\cup$ Best First Search	43	7

Table 3.2 Results for a Sample of 50 Path-Methods

CHAPTER 4

PATH–METHOD GENERATION USING ACCESS RELEVANCE

In the previous chapter we have discussed generation of path–methods using access weights. The best algorithm reported did not find 7 out of 50 desired path–methods. We observe that the traversal using only access weights does not lead to the target in these seven cases, as access weights are assigned based on frequency of use of the connections and not based on the path associated with the corresponding path–method.

In this chapter we discuss path–method generation using precomputed access relevance between classes of an OODB schema. As computation of an access relevance reflects the access weights of all edges along a path in an OODB schema, the path–method generation using precomputed access relevance is predicted to generate more desired path–methods.

4.1 Human Traversal in Object-Oriented Databases

In this section we will illustrate how a human navigates an OODB schema to find a particular item of information. This kind of navigation motivates the design of the following algorithm to generate path–methods. Consider the path–method *courses*,

shown before, to find the courses an instructor is teaching (see Figure 2). It is required to find a path from the class **instructor** to the target information {**course**}(read: set of courses). We assume that the path-method *Courses* is not yet available for the class **instructor** in the OODB schema.

A human will first look at all the properties defined in the class **instructor** and decide which one of them will most likely lead him to the required result. First, the attributes of the class are scanned. If one of them contains the desired information, then the search is completed. Otherwise, s/he chooses a most likely connection out of the class. This connection will lead to another class. There s/he looks for a most likely property again. S/he will continue this process until s/he finds the necessary information. Of course, a human may need to backtrack. Whenever we refer in this section to a *match of two phrases* we refer to a match which is done by a human (using intuition) and not a match done by a machine which requires strict rules. We explore now this technique in more detail.

1. In this example the user compares all the attributes of the class **instructor** with the target information *course*. (Note that all the properties defined for **person** and **employee** are inherited by the class **instructor**.) None of the attributes matches with *course*, therefore s/he considers the relationships of the class **instructor**. From three relationships *Supervisor, Resume* and *Sections* s/he selects the relationship *Sections* to **sections**, because it is the most relevant to the target information *course*.

2. The class **sections** has two attributes *Numsections* and *GroupPurpose*. As both of them do not match with *course*, s/he considers the generic relationship *setof* to the class **section**, which is the only connection defined for the class **sections**. The *setof* generic relationship is one-to-many, thus, we will represent this class enclosed by {}, as {section}.

3. None of the attributes of the class **section** matches the target information *course*, therefore s/he looks for relationships. The relationship *Instructor* to the class **instructor** is not considered because the class **instructor** was already visited. The relationship *Students* to the class **students** will not be considered because it does not match the target information. From the two *memberof* generic relationships s/he selects *memberof* to the class **crsections** because the class **sections** was already visited. This generic relationship is applied for each element of the set {**section**} yielding a set {**crsections**}.

4. None of the attributes of the class **crsections** are selected as they don't match the target information *course*. The generic relationship *setof* is not selected because the class **section** was already visited. The relationship *Course* to **course** is selected because it matches the target information completing the traversal.

The generated path–method is the same as Instructor–Courses shown in Chapter 2.2.

4.2 Definition of Access Relevance for an OODB Schema

The significance of a path is measured by the *access relevance value* (ARV). The *ARV of a path* is obtained by applying a triangular–norm (t–norm) [FKN91, K92, Z65, KF88] to access weights of all the connections of the path. For example, for the commonly used t–norm PRODUCT, the access relevance of a path is the product of the access weights of all its edges. There exist several infinite families of t–norms and corresponding conorms [SS61]. However, in [BD86] it is empirically shown that for most practical purposes two to five different t–norms suffice. In [FKN91, K92] three different t–norms are used. From these we have chosen PRODUCT and the more optimistic MINIMUM t–norms to compute access relevance of a path. The third t–norm was not useful for the path–method generation. We refer to t–norms as weighting functions.

PRODUCT weighting function: The weight of a single path between two classes is the product of all the access weights of all the edges in the path.

MINIMUM weighting function: The weight of a single path between two classes is the minimum of all access weights of all the edges in the path.

The minimum weight edge of a path is called the *bottleneck edge.*

Formally, the *access relevance value for a path P* is defined as the result of applying a weighting function WF (i.e., PRODUCT or MINIMUM) to a single path P(a_s, a_t) = $a_s(= a_{i_1}), a_{i_2}, a_{i_3}, \ldots, a_{i_k}(= a_t)$.

$$ARV(P) = WF_{(1 \le r < k)}\ W(a_{i_r}, a_{i_{r+1}})$$

Note that we are using operator notation for WF, because WF stands for $\prod$ or min which are commonly written in operator notation.

The access relevance between non–adjacent classes a_s and a_t is a measure of the significance or strength of the indirect connection from a_s to a_t or the accessibility from a_s to a_t. If several paths exist between the source and target classes then we use the co–norm MAXIMUM to compute a single value. The *access relevance from* a_s *to* a_t is defined as the maximum of ARV(P) over all paths from a_s to a_t.

$$AR(a_s, a_t) = \max_{j=1}^{m} ARV(P_j) = \max_{j=1}^{m} WF_{((a_{i_r}, a_{i_{r+1}}) \in P_j)} \; W(a_{i_r}, a_{i_{r+1}})$$

Note the difference between two terms *access relevance value* and *access relevance.* The first one is computed for a single path, while the second is computed between two given classes. A path with the maximum access relevance value is called a *most relevant path.* In other words, for the most relevant path the access relevance value is identical to the access relevance. Maximizing the MINIMUM weighting function finds a path with the bottleneck edge of highest value. Maximizing the PRODUCT weighting function finds a path with the highest product of access weights of all its edges. We note that sometimes the user may be interested in the access relevance between a class a_s and an attribute atr_{a_t} of another class a_t. Our definition can be extended to handle this case by representing the connection between a class and its attributes by an edge with a given access weight.

For a weighting function WF we define AR for attributes as follows:

$$AR(a_s, atr_{a_t}) = WF\ (AR(a_s, a_t), W(a_t, atr_{a_t}))$$

Note that if $W(a_t, atr_{a_t}) = 1$ then for both weighting functions $AR(a_s, atr_{a_t}) = AR(a_s, a_t)$.

Let us consider the paths from the class **professor** to the class **course** (Refer to Figure 2.1). The path p_1 consists of the class sequence (professor, sections, section, crsections, course). This class sequence will retrieve all the courses currently being taught by an instructor. The access relevance value of path p_1 using the PRODUCT (MINIMUM) weighting function is (0.5 * 0.5 * 0.3 * 0.8) = 0.06 (0.3). The alternative path p_2 consists of the class sequence (professor, students, student, transcript, sections, section, crsections, course). This class sequence will retrieve all the courses currently taken by all the students supervised by a professor. The access relevance value of path p_2 using PRODUCT (MINIMUM) weighting function is (0.1 * 0.5 * 0.9 * 0.4 * 0.5 * 0.3 * 0.8) = 0.00216 (0.1). Because 0.06 > 0.00216 and 0.3 > 0.1, p_1 is a more relevant path and is actually, the most relevant path. This is not surprising, since the interpretation of p_1 is more starightforward compared to p_2. Later on we will see that the algorithm *PathMethodGenerate* generates a path–method along the path with the access relevance value 0.06 rather than the path with the access relevance value 0.00216. A path does not necessarily maximize both weighting functions. For example consider path p_3 with the class sequence (grad_student, transcript, sections, section). This class sequence will retrieve all the sections currently taken by a graduate student. The access relevance value of path p_3 using the PRODUCT (MINIMUM) weighting function is (0.9 * 0.4 * 0.5) = 0.18 (0.4). The alternative

path p_4 has the class sequence (grad_student, professor, sections, section). This class sequence will retrieve all the sections currently being taught by the supervisor of a student. The access relevance value of path p_4 using the PRODUCT (MINIMUM) weighting function is (0.5 * 0.5 * 0.5) = 0.125 (0.5). As 0.18 > 0.125, p_3 is a most relevant path using the PRODUCT weighting function and as 0.5 > 0.4, p_4 is a most relevant path using the MINIMUM weighting function.

Algorithms for efficient computation of access relevance are presented in Chapter 5.

Later on we will need the following property of a most relevant path.

Property 1: For every pair of nodes there exists a simple (i.e., no cycles) most relevant path.

Proof: Let P = $((s =)a_{i_1}, a_{i_2}, \ldots, a_{i_k}(= t))$ be a most relevant path from s to t. Suppose P contains a cycle. That is P can be written as $((s =)a_{i_1}, a_{i_2}, \ldots, a_{i_j}(= b_{j_1}), b_{j_2}, b_{j_3}, \ldots, b_{j_k}(= a_{i_j}), \ldots, a_{i_k}(= t))$. Now let $P_1 = (a_{i_1}, a_{i_2}, \ldots, a_{i_j})$, $P_2 = (b_{j_1}, b_{j_2}, b_{j_3}, \ldots, b_{j_k})$, $P_3 = (a_{i_j}, \ldots, a_{i_k})$. Let Q be the path obtained by concatenation of P_1 and P_3. Since $ARV(P_2) \leq 1$ for both PRODUCT and MINIMUM weighting functions

$$\begin{aligned} ARV(P) &= WF(ARV(P_1), ARV(P_2), ARV(P_3)) \\ &\geq WF(ARV(P_1), ARV(P_3)) \\ &= ARV(Q) \end{aligned}$$

Hence Q is also a most relevant path between s and t. If Q has no cycle the proof

is completed. Otherwise we continue removing cycles until an acyclic most relevant path is obtained. □

One may try to use, as an alternative approach, the semantics of the different connections in the OODB schema rather than the frequencies of the different connections. However, one needs to specify a combination rule for the semantics of the connections along a path in order to derive the semantics of a connection of two classes which are not directly related. Recent work [FKN91, FN92, SK92a, SK92b, SSR92] addresses this problem for resemblance between classes either from same database or from several databases (either integrated or interoperable) using various semantic measures such as, semantic resemblance, semantic proximity, etc. First we note an essential difference between resemblance and accessibility. In resemblance we try to measure similarity or closeness between the concepts represented by two classes. In accessibility we try to measure the strength of an indirect connection (i.e. path) between two classes not directly related. That is, we are not interested in similarity, but in the possibility of access from one class to another. Furthermore, the ideas for combining resemblance are not applicable to measuring the semantics of the connection between two indirectly related classes, that is, measuring the degree to which the path between two classes in the schema is meaningful. The problem is that while resemblance [FKN91] can supply a semantic interpretation to the combination of different adjacent connections, we do not know of an easy way to generalize this notion for accessibility. Thus, we take another approach in measuring the significance of the

connection between two indirectly related classes by defining access relevance.

4.3 An Algorithm for Path–Method Generation using Precomputed Access Relevance

The simplest greedy traversal algorithm is to choose at each node the outgoing connection of highest frequency. However, our experiments with the best first search traversal show that this algorithm, like many greedy algorithms [HS89], lacks the look–ahead property necessary in many cases to create a desired result, in this case the desired path–method. Thus, we use a measure that incorporates the access weights of all connections that make up the path. Our algorithm will decide on the connection to be traversed from s, based on the access weight of the connection to a neighboring class u and the access relevance from u to the target class t. These choices will be made for each node in the path traversal. This mechanism adds to the greedy traversal approach the necessary look–ahead property which dramatically improves the results as reported later in this chapter.

We will now describe a traversal algorithm, *PathMethodGenerate*, for generating path–methods. Access weights are stored in the matrix W and precomputed access relevance are stored in the matrix ARM. Before presenting this algorithm, some additional conventions are necessary.

If a class n_1 has a connection to a class n_2, then the class n_2 is a neighbor of the class n_1. Note that the class n_1 is not necessarily a neighbor of the class n_2, because

connections are directed. Suppose a class has a set of neighbors N. The traversal algorithms may consider only a subset P of *permissible neighbors* of N, for reasons to be explained shortly.

We also need to use a special notation for properties, called *pair notation.* For instance, a relationship r from a class a to a class b is written in pair notation as (r, b). The two components of a pair can be retrieved with the usual functions *head* and *tail.* For readability it is useful to introduce notational variants of *head* and *tail* that express their functionality at a specific point. Thus, we introduce *selector* $\equiv$ *head*, *datatype* $\equiv$ *tail* for attributes, and *classname* $\equiv$ *tail* for relationships. More generally we introduce *selector* $\equiv$ *head*, and *result* $\equiv$ *tail.* For methods, *result* might be either a data type or a class name. For instance, the relationship *Transcript* to the class **transcript** would be written as (*Transcript*, **transcript**), and *selector*((*Transcript*, **transcript**)) = *Transcript.* The function *connection* takes two classes as argument and returns the property that connects them in pair notation. E.g., with a relationship r from a class a to a class b, *connection*(a, b) = (r, b).

The algorithm uses a stack *stk.* Each element of the stack *stk* is a pair (*selector*, *result*). The algorithm accepts three required parameters, the source s, the target information t, which may be a class or an attribute in the schema, and the name of the method m as a string. It also accepts two optional parameters, a set of forbidden classes F, and an intermediate class c. No class of F may occur in the resulting path–method, while the class c, if given, must occur in the path–method. The algorithm

returns the generated path–method in an array PM. If the Path–Method Generator is unsuccessful, PM remains empty.

The variable U contains at all times the set of all visited nodes. Initially U contains only the source class *s*. In each step of the while–loop (step 2), we first check whether the current node *u* has an attribute a_i the selector of which is identical to *t*. In such a case we set a boolean variable *found* to true. Otherwise, we find a set of neighbors of *u* in step 4. In step 5 a pair (*selector*(*connection*(*u*, *v*)), *v*) is pushed onto *stk*. For the selection of a most relevant neighbor *v* of *u* we apply a t-norm (PRODUCT or MINIMUM) to the access weight W[*u*, *v*] and the access–relevance–matrix entry ARM[*v*, *t*]. We select the neighbor where this value is maximized. If the selected neighbor *v* of *u* is identical to the target message *t*, we set *found* to true. If there is no such neighbor (which can happen if F $\neq \phi$), the algorithm backtracks, pops the current node *u* from *stk*, and tries to find permissible neighbors of the previous node in *stk*. For the successful cases (step 7), the algorithm transfers all the pairs of a path–method from *stk* into the array PM, reversing the order. It also creates the header line of the method, using the name parameter *m*, and makes cosmetic changes as follows. A pair (*a*, *b*) is stored in PM as $a \rightarrow b$:. This is called *arrow notation* of a pair and was used in previous examples.

```
PROCEDURE PathMethodGenerate
                    (IN s: class; t: classNameOrAttributeSelector; m: string;
                    OUT PM: array; OPTIONAL IN F: class_set, c: class)
var
   stk: stack; found: boolean; u, v, r: class; U, P, N: class_set;
begin
1.    [Initialization:]
```

found := false; [found is true when the target information is found.]
U := ϕ; [Make visited set empty.]
r := t; [the target class is stored in r.]
if not c = nil **then begin**
[Case when intermediate class is given.]
    t := c; [The intermediate class is set temporarily as the target.]
end;
push ((dummy, s), stk); [push the source class into the stack.]
2. **while not** empty(stk) **do**
begin
    u := *result*(*top*(stk));
    U := U ∪ {u};
3. **if** u has an attribute a_i the selector of which is identical to t **then**
    begin
        push ((selector(a_i), datatype(a_i)), stk)
        found := true;
    end
4. **else begin**
        N := set of neighbors of u;
        P := ((N - U) - F); [Remove visited and forbidden classes from N]
5.   **if not** empty(P) **then**
        begin [This is the case when there are permissible neighbors of u.]
            choose the element v from P such that
                t-norm (W[u, v], ARM[v, t]) is maximal in P;
6.   **if** v is identical to t **then**
                found := true;
            push ((*selector*(*connection*(u, v)), v), stk);
        end
7.   **else begin** [Case when there is no permissible neighbor of u.]
            pop(*stk*);
        end
    end
8. **if** found **then begin**
        if not t = r **then begin**
        [This is the case where the first half of the path–method
            to intermediate class is found successfully.]
            found := false;
            t := r;
        end
9.   **else begin**
        [This is the case when either there is no intermediate class (c = **nil**)
        or the second part of the path–method is found successfully.]

```
            Add m(): at the first position of the array;
            while not empty(stk) do
                pop(stk) into array PM; [This reverses class order.]
                [Pairs are stored in arrow notation.]
            Delete the last element of PM which contains a dummy.
        end
    end [of if]
end [of while]
end [of PathMethodGenerate]
```

Now we will show how the path–method *Student–Courses* (refer to path–method 1 in the sample set) for the class **student** is generated using the algorithm *Path-MethodGenerate.*

The steps of the *PathMethodGenerate* algorithm are demonstrated in Table 4.1, making use of Figure 3.2. We start with a class **student** (2). It has four permissible neighbors **person** (1), **transcript** (6), **studentunion** (7) and **students** (3). Now, W[2, 1] * ARM[1, 11] = 0.0 * 0.0 = 0.0, W[2, 6] * ARM[6, 11] = 0.9 * 0.28 = 0.252, W[2, 7] * ARM[7, 11] = 0.1 * 0.227 = 0.0227, and W[2, 3] * ARM[3, 11] = 0.5 * 0.252 = 0.126. Because 0.252 > 0.126 > 0.0227 > 0.0, the relationship to the class **transcript** is selected, and the corresponding pair (Transcript, transcript) is pushed on *stk.*

The class **transcript** has permissible neighbors **sections** (8), and **course_records** (16). Now, W[6, 8] * ARM[8, 11] = 0.4 * 0.12 = 0.048, W[6, 16] * ARM[16, 11] = 0.8 * 0.35 = 0.280. Thus, in Step 2 class **course_records** will be selected, and the pair (CourseRecords, course_records) is pushed on *stk.* The class **course_records** has only one permissible neighbor, **course_record.** Thus, in step 3 the pair (setof,

course_record) is pushed on the *stk*. The class **course_record** has only one permissible neighbor and that is the target information. Thus, the pair (Course, course) is added to the *stk*.

Iteration	u	P	new element of stk
Initial	–	–	(Courses student)
1	{student}	{person, transcript, studentunion, students}	(Transcript transcript)
2	{transcript}	{course_records, sections}	(CourseRecords course_records)
3	{course_records}	{course_record}	(setof course_record)
4	{course_record}	{course}	(Course course)

Table 4.1 Steps of the Algorithm PathMethodGenerate

The complexity of the algorithm in the worst case in $O(e)$, where e is the total number of connections and attributes in the schema. For the validity proof for the algorithm for the PRODUCT t–norm we need the following lemma. This Lemma 1 shows that the most relevant path using the PRODUCT t–norm satisfies the principle of optimality which is the basis for dynamic programming [HS89].

Lemma 1: For the PRODUCT t–norm a subpath of a most relevant path is a most relevant path.

Proof: Let $P = ((s =)a_{i_1}, a_{i_2}, \ldots, a_{i_k}(= t))$ be a most relevant path from s to t. Consider a subpath $Q = a_{i_j}, a_{i_{j+1}}, \ldots, a_{i_l}$, $1 \leq j < l \leq k$, of P. We have to show that Q is a most relevant path from a_{i_j} to a_{i_l}. Suppose to the contrary that there exists another path R from a_{i_j} to a_{i_l} $R = ((a_{i_j} = b_{i_1}, b_{i_2}, \ldots, b_{i_m}(= a_{i_l}))$ with higher access relevance than Q, that is $AR(b_{i_1}, b_{i_2}, \ldots, b_{i_m}) > AR(a_{i_j}, a_{i_{j+1}}, \ldots, a_{i_l})$. Consider the path $U = (a_{i_1}, a_{i_2}, \ldots, a_{i_{j-1}}, b_{i_1}, b_{i_2}, \ldots, b_{i_m}, a_{i_{l+1}}, \ldots, a_{i_k})$. For the PRODUCT t–norm the access relevance AR satisfies

$$\begin{aligned}\text{AR}(U) &= \text{PRODUCT } (AR(a_{i_1},\ldots,a_{i_j}), AR(b_{i_1},\ldots,b_{i_m}), AR(a_{i_l},\ldots,a_{i_k})) \\ &> \text{PRODUCT } (AR(a_{i_1},\ldots,a_{i_j}), AR(a_{i_j},\ldots,a_{i_l}), AR(a_{i_l},\ldots,a_{i_k})) \\ &= \text{AR}(a_{i_1},\ldots,a_{i_k}) = \text{AR(P)}\end{aligned}$$

If the path R is node disjoint with the subpaths (a_{i_1}, a_{i_2}, ..., $a_{i_{j-1}}$) and ($a_{i_{l+1}}$, ..., a_{i_k}) then the path U is a simple path from s to t with higher access relevance than the most relevant path P, a contradiction. On the other hand, consider the case where R has joint nodes with $(a_{i_1},\ldots,a_{i_{j-1}})$. Let a_{i_r} be the first such joint node, that is $a_{i_r} = b_{i_x}$, $1 < r \leq j-1$, $1 < x < m$. Consider the path $M = (a_{i_1}, \ldots, a_{i_{r-1}}, b_{i_x}, \ldots, b_{i_m}, a_{i_{l+1}}, \ldots, a_{i_k})$. The weights of the edges deleted from U are in the range [0, 1]. Thus, AR(M) $\geq$ AR(U) $>$ AR(P). If (a_{i_1}, ..., $a_{i_{r-1}}$, b_{i_x}, ..., b_{i_m}) still has more joint nodes with ($a_{i_{l+1}}$, ..., a_{i_k}) the removal of joint nodes is done similarly. Thus, the obtained path M is a simple path from s to t with higher access relevance than the most relevant path P, a contradiction. Thus, there exist no such path R such that AR(R) $>$ AR(Q), and Q is a most relevant path from a_{i_j} to a_{i_l}. □

For the MINIMUM t–norm we need a different lemma.

The following Lemma 2 satisfies the principle of optimality.

Lemma 2: For the MINIMUM t–norm a subpath containing all the bottleneck edges of a most relevant path is a most relevant path.

Proof: Let $P = ((s =)a_{i_1}, a_{i_2}, \ldots, a_{i_k} (= t))$ be a most relevant path from s to t. Consider a subpath $Q = a_{i_j}, a_{i_{j+1}}, \ldots, a_{i_l}$, $1 \leq j < l \leq k$, of P containing all the bottleneck edges. We have to show that Q is a most relevant path from a_{i_j} to

a_{i_l}. Suppose to the contrary that there exists another path R from a_{i_j} to a_{i_l} $R = ((a_{i_j} = b_{i_1}, b_{i_2}, \ldots, b_{i_m} (= a_{i_l}))$ with higher access relevance than Q, that is $AR(b_{i_1}, b_{i_2}, \ldots, b_{i_m}) > AR(a_{i_j}, a_{i_{j+1}}, \ldots, a_{i_l})$. Consider the path $U = (a_{i_1}, a_{i_2}, \ldots, a_{i_{j-1}}, b_{i_1}, b_{i_2}, \ldots, b_{i_m}, a_{i_{l+1}}, \ldots, a_{i_k})$. For the MINIMUM t–norm the access relevance AR satisfies

$$\begin{aligned} \text{AR}(U) &= \text{MINIMUM } (AR(a_{i_1}, \ldots, a_{i_j}), AR(b_{i_1}, \ldots, b_{i_m}), AR(a_{i_l}, \ldots, a_{i_k})) \\ &> \text{MINIMUM } (AR(a_{i_1}, \ldots, a_{i_j}), AR(a_{i_j}, \ldots, a_{i_l}), AR(a_{i_l}, \ldots, a_{i_k})) \\ &= \text{AR}(a_{i_1}, \ldots, a_{i_k}) = \text{AR(P)} \end{aligned}$$

If the path R is node disjoint with the subpaths $(a_{i_1}, a_{i_2}, \ldots, a_{i_{j-1}})$ and $(a_{i_{l+1}}, \ldots, a_{i_k})$ then the path U is a simple path from s to t with higher access relevance than the most relevant path P, a contradiction. On the other hand, consider the case where R has joint nodes with $(a_{i_1}, \ldots, a_{i_{j-1}})$. Let a_{i_r} be the first such joint node, that is $a_{i_r} = b_{i_x}$, $1 < r \leq j-1$, $1 < x < m$. Consider the path $M = (a_{i_1}, \ldots, a_{i_{r-1}}, b_{i_x}, \ldots, b_{i_m}, a_{i_{l+1}}, \ldots, a_{i_k})$. The weights of the edges deleted from U are in the range [0, 1]. Thus, $\text{AR}(M) \geq \text{AR}(U) > \text{AR}(P)$. If $(a_{i_1}, \ldots, a_{i_{r-1}}, b_{i_x}, \ldots, b_{i_m})$ still has more joint nodes with $(a_{i_{l+1}}, \ldots, a_{i_k})$ the removal of joint nodes is done similarly. Thus, the obtained path M is a simple path from s to t with higher access relevance than the most relevant path P, a contradiction. Thus, there exist no such path R such that $\text{AR}(R) > \text{AR}(Q)$, and Q is a most relevant path from a_{i_j} to a_{i_l}. □

Definition 1: For the MINIMUM t–norm a subpath without any bottleneck edge of a most relevant path is a *non-effective subpath.*

Note that all the edges of a non–effective subpath have higher access weights than the access weight of the bottleneck edge (i.e., access relevance of the path). Although the principle of optimality is not satisfied, the algorithm finds a most relevant path. This is because of the nature of the MINIMUM t–norm, which selects the minimum edge access weight of the path and the rest of the edges can have larger access weights than the minimum edge access weight and not necessarily the largest possible. Hence, for the MINIMUM t–norm any non–effective subpath of a most relevant path need not be a most relevant path.

Theorem 1: The algorithm *PathMethodGenerate* generates a path–method corresponding to a most relevant path from the source class s to the target class t. (If t is an attribute then the target class is the class containing t.)

Proof: The proof is by induction on the sequence of nodes of a most relevant path.

Basis of the induction: We need to show that the first connection of the path–method generated corresponds to the first edge of a most relevant path. In the first iteration of step 5, $u = s$ and the algorithm selects a neighbor v of u such that t–norm ($W[u, v]$, ARM$[v, t]$) is maximum. But ARM$[v, t]$ is the access relevance of the most relevant path from v to t and W$[u, v]$ is the access weight from u to v. As $u = s$, then by the definition ARM$[s, t]$ = $\max_v$(t–norm ($W[s, v]$, ARM$[v, t]$)). Hence, the selected neighbor v will be the first node on a most relevant path from s to t.

Induction Step: Suppose the path from s to u which corresponds to a subsequence of the path–method generated by the algorithm is a subpath of a most relevant path

from s to t. We need to show that the edge which corresponds to the next connection of the generated path–method is on a most relevant path from s to t. By Lemma 1 for the PRODUCT t–norm, the subpath from u to t of a most relevant path from s to t, is a most relevant path from u to t. For the MINIMUM t–norm, the subpath from u to t of a most relevant path from s to t, is either a most relevant path (Lemma 2) or a non–effective subpath, from u to t.

The algorithm picks in step 5 a neighbor v of u such that t–norm ($W[u, v]$, ARM[v, t]) is maximized. By the definition of the ARM, ARM(u, t) = $\max_v$(t–norm ($W[u$, $v]$, ARM[v, t])). Hence the selected neighbor v is a first node on a most relevant path from u to t for the first two cases. For the third case of a non–effective subpath it is not required to be on a most relevant path from u to t. Hence, the selected node v and edge (u, v) are on a most relevant path from s to t. □

4.4 Results of Experiments for the Sample Set of Path–Methods

The results of path–method generation using the algorithm *PathMethodGenerate* are shown in Table 4.2. In this section we report applying the algorithm with an empty forbidden set F and no required intermediate class c. These two parameters are utilized in the next section.

From the Table 4.2, we observe the following conclusions. The results for *Path-MethodGenerate* for the PRODUCT t–norm (using Rule 2a as well as Rule 2b) are

No.	PathMethodGenerate								Modified PathMethodGenerate							
	Product Rule 2a		Product Rule 2b		Minimum Rule 2a		Minimum Rule 2b		Product Rule 2a		Product Rule 2b		Minimum Rule 2a		Minimum Rule 2b	
	PL	NVN	PL	NVN	PL	NVN	PL	NVN	PL	NVN	PL	NVN	PL	NVN	PL	NVN
1	4	9	4	9	4	9	(5)	(11)	4	9	4	9	4	13	4	9
2	(3)	(19)	4	13	(3)	(19)	(3)	(22)	(3)	(19)	(3)	(15)	(3)	(22)	(3)	(22)
3	4	9	4	9	(8)	(20)	(8)	(20)	4	9	4	9	(13)	(43)	(13)	(43)
4	4	9	4	9	4	13	(7)	(45)	4	19	4	9	(8)	(32)	(8)	(32)
5	4	7	4	7	4	7	4	7	4	7	4	8	4	7	4	8
6	4	7	4	7	(11)	(40)	(10)	(56)	4	7	4	7	(11)	(38)	4	7
7	4	11	4	13	(6)	(29)	(11)	(43)	4	13	4	13	(11)	(36)	(11)	(36)
8	4	10	4	11	4	13	4	15	4	11	4	11	4	14	4	14
9	5	12	5	12	(6)	(33)	(6)	(34)	5	12	5	12	(6)	(15)	(6)	(15)
10	3	7	3	7	(6)	(29)	(6)	(32)	3	7	3	7	(6)	(29)	(6)	(29)
11	(3)	(18)	(5)	(16)	6	16	6	16	(6)	(23)	(5)	(16)	6	18	6	18
12	7	14	7	16	7	16	(12)	(34)	7	16	7	16	(14)	(40)	(12)	(38)
13	3	13	3	10	(3)	(13)	(5)	(17)	(3)	(12)	(3)	(12)	(8)	(27)	(8)	(27)
14	4	21	4	21	(6)	(29)	(10)	(31)	(6)	(17)	(6)	(17)	(15)	(45)	(15)	(45)
15	(4)	(11)	(4)	(9)	(5)	(12)	(4)	(9)	(6)	(18)	(4)	(16)	(6)	(22)	(6)	(22)
16	4	17	4	13	(6)	(29)	(8)	(25)	(2)	(9)	(2)	(9)	(9)	(33)	(9)	(33)
17	3	16	3	14	(5)	(31)	(5)	(28)	(3)	(14)	(3)	(14)	(8)	(40)	(8)	(40)
18	4	10	4	10	(13)	(41)	(13)	(41)	4	10	4	10	(7)	(15)	4	10
19	4	23	4	23	4	23	4	23	4	23	4	23	4	29	4	29
20	4	9	4	9	4	11	4	9	4	9	4	9	4	36	4	36
21	(6)	(26)	4	12	(6)	(26)	(7)	(23)	(6)	(26)	(5)	(16)	(6)	(28)	(6)	(31)
22	7	17	7	17	(13)	(39)	(12)	(35)	(7)	(17)	(7)	(17)	(7)	(22)	(7)	(22)
23	3	19	3	19	3	27	3	27	3	19	3	19	(5)	(25)	3	15
24	4	11	4	11	(11)	(33)	(11)	(31)	4	11	4	11	4	14	4	14
25	(4)	(15)	4	9	(4)	(15)	4	9	(4)	(15)	4	9	(4)	(17)	4	11
Average	4.12	13.6	4.16	12.24	6.08	23.32	6.88	25.72	4.32	14.0	4.16	12.48	7.2	26.4	6.64	25.48

Table 4.2 Results of PathMethodGenerate and Modified PathMethodGenerate

very good. For the PRODUCT t–norm using Rule 2b, only two of the 25 path–methods generated for the sample were not the desired ones. Furthermore, the number of nodes visited during this algorithm is in general lower than for any of the other algorithms, showing the efficiency of the algorithm. The cases where the generated path–methods are not the desired ones will be analyzed in the next section. The results for the MINIMUM t–norm (using Rule 2a as well as Rule 2b) are disappointing. The conclusion is that the MINIMUM t–norm appears not fit for the calculation of access relevance for guiding path–method generation. We conjecture that the reason is that the MINIMUM t–norm does not reflect all the weights of the edges of the path, as does the PRODUCT t–norm.

Now, we compare these results with results obtained by a new algorithm, *Modified PathMethodGenerate*. The *Modified PathMethodGenerate* algorithm selects a neighbor whose access relevance to the target is maximal. Unlike the *PathMethodGenerate* algorithm, the *Modified PathMethodGenerate* ignores the access weight from the current node to a neighbor. The results for the *Modified PathMethodGenerate* are disappointing. Only 16 of 25 path–methods generated are desired ones for the PRODUCT t–norm and 11 of 25 are desired ones for MINIMUM t–norm. The conclusion is that the access weight from a node to a neighbor has a major impact on the generated path–method. This phenomenon is not so obvious, intuitively, since one may just want to choose a neighbour most relevant to the target without looking at the properties of the edge to that neighbour. However this phenomenon can be

better understood when one realizes that the *Modified PathMethodGenerate* algorithm is not guaranteed to generate path–methods with most relevant path as does the *PathMethodGenerate* algorithm as proven in Theorem 1 above. Note that for the PRODUCT t–norm using Rule 2a we get undesired results for path–methods 2, 21, and 25 in Table 4.2, in addition to three undesired results common to both rules.

Now, we will demonstrate why Rule 2a gives an undesired result for the path–method 2. The generated path–method is shown below.

Instructors ():
Supervisor $\longrightarrow$ professor: categoryof $\longrightarrow$ faculty_member:
categoryof $\longrightarrow$ instructor

Note that the class **grad_student** has a relationship *Supervisor* to the class **professor**. Being an indirect subclass of the class **instructor** (i.e., the target information), the class **professor** inherits all the properties of the class **instructor**. Thus, once we reach to the class **professor**, the most relevant path (using Rule 2a) to the class **instructor** requires traversal of two *categoryof* generic relationships through **faculty_member** without utilizing inheritance, which does not make sense. For Rule 2b the same path is not a most relevant path, as we assign an access weight 0.0 to the generic relationship *categoryof*.

We have compared results of different traversal algorithms in Chapter 3. There, the best access weight traversal algorithm *best breadth first search* found 20 of 25 path–methods. Here, the algorithm *PathMethodGenerate* using Rule 2b and PRODUCT t–norm found 23 of 25 path–methods. Obviously, basing the traversal on access

relevance yields much better results, since it adds the feature of lookahead which was missing in the previous algorithms. In particular this algorithm generated four path–methods without a shortest path not obtainable by *best breadth first search*. By looking at results of all the algorithms discussed above, the overall conclusion is that the algorithm *PathMethodGenerate* using Rule 2b and PRODUCT t–norm is the best of all traversal algorithms.

No.	Name of the Algorithm	Generated Path-Methods (out of 50)	
		Desired	Undesired
1.	Depth First Search	5	45
2.	Breadth First Search	41	9
3.	Best First Search	13	37
4.	Best Breadth First Search	43	7
5.	Breadth First Search $\cup$ Best First Search	42	8
6.	Best Breadth First Search $\cup$ Best First Search	43	7
7.	PathMethodGenerate (PRODUCT (Rule 2b))	46	4
8.	PathMethodGenerate (PRODUCT (Rule 2a))	37	13
9.	PathMethodGenerate (MINIMUM (Rule 2b))	16	34
10	ModifiedPathMethodGenerate (PRODUCT (Rule 2b))	31	19
11	ModifiedPathMethodGenerate (MINIMUM (Rule 2a))	21	29

Table 4.3 Results for a Larger Sample

To investigate the problem of path–method generation more thoroughly, we applied these algorithms to a sample set of 50 path–methods to be generated for the

classes of the subschema of Figure 3.2 (including the 25 path–methods discussed previously). The results are shown in Table 4.3.

In general, the results for the larger sample correspond to the results for the smaller sample recorded earlier. Our results show that the algorithm *PathMethodGenerate (PRODUCT (Rule 2b))* is the best of all. It found 46 desired path–methods out of 50 given path–methods. Thus, whenever the access relevance is already computed we should apply this algorithm. There are four undesired path–methods: two from the small sample set and the other two from the 25 path–methods of the larger sample. These four cases are discussed in the next section.

4.5 Parameterized Path–Method Generation

In this section we explore techniques to improve the results of the traversal algorithm in the cases where it failed to provide the desired results. While the techniques are general enough to be applied to different traversal algorithms, we illustrate them here only with the most successful algorithm of the PMG system, namely *PathMethod-Generate (PRODUCT (Rule 2b))*.

In the previous section we reported two cases where the desired path–method was not found by the *PathMethodGenerate* algorithm for the PRODUCT t–norm.

In the first case (path–method 11) the path–method generated by the algorithm is

Research_assistants ():

Departments ⟶ departments: setof ⟶ {department}:
@DeptChairPerson ⟶ {dept_chair_person}:
@ResearchAssistants ⟶ {{research_assistants}}:
@setof ⟶ {{{research_assistant}}}: **union** ⟶ {{research_assistant}}:
union ⟶ {research_assistant}

This path–method finds all the research assistants of the department chair_persons of a college rather than all the research assistants of all the professors of the departments of a college, the desired path–method. This path–method was obtained since the class subpath (department, dept_chair_person, research_assistants [last relationship inherited from professor]) has higher access relevance than (department, professors, professor, research_assistants) mainly due to the use of inheritance. The path–method generated does not make sense since its inclusion of the **dept_chair_person** class limits unnecessarily the range of the query.

In the second case our algorithm generates the following path–method:

Teaching_Assistant–Courses ():
Transcript ⟶ transcript: CourseRecords ⟶ course_records:
setof ⟶ {course_record}: @Course ⟶{course}

This path–method finds the classes a teaching_assistant is currently registered for, rather than those s/he is teaching. In this case both path–methods make sense, but the desired one is the straightforward interpretation of the source **teaching_assistant** and the target {**course**}, since a teaching_assisting exists in a capacity of teaching rather than studying. Other interpretation is actually inherited from **grad_student**.

We have no way to prevent the algorithm to produce such path–methods due to the access weights in the schema reflecting *roleof* inheritance (full or selective) which

is required for other purposes. Thus, our approach is that it is the responsibility of the user to screen the path–method obtained and judge if this is the desired one. Our experiments indicate that the desired path–method will be generated with high probability.

We will now show how the user can apply the algorithm again in case of an unsatisfactory result, using the information contained in the unwanted path–method, to obtain his desired path–method. The user can constrain the generation of the desired path–method by specifying the optional parameters for the algorithm. For example the user can pick classes in the generated path–method to be in the set F of forbidden classes which are in a different context than the desired path–method. As a result the repeated application must generate a different path–method which does not contain these classes. For the first undesired path–method we can add to F the class **dept_chair_person**, and for the second one we can add to F the class **transcript**. Our experiments show that with this modification the algorithm produces the other possible path–method successfully.

Another option is to identify a class c which does not appear in the path–method generated but its participation in the path–method appears to improve the chances for successful generation. Such a class has to be chosen as an intermediate class parameter. The subsequent application of the algorithm with the intermediate parameter set to this class will force the newly generated path–method to contain the desired class. For the first path–method we can pick $c =$ **professors**. For the second

one we can pick c = **instructor** (i.e., using the inheritance from **instructor**). This mechanism is based on the Lemma 1 from Section 4.3. Each of these options will increase the chances of generating the desired path–method in the second trial. The user may even set both parameters simultaneously for the second trial, increasing further the chances for generating the desired path–method.

We examine these two mechanisms with the two undesired path–methods which occur in the large sample. For the first case (path–method 27) the undesired path–method generated is shown below. This undesired path–method is generated because of traversal of relationship *DeptChairPerson* from the class **department** similar to the case of the first undesired path–method above. Note that the relationship *Sections* is inherited from class **instructor** through class **professor**.

Department–Courses ():
DeptChairPerson $\longrightarrow$ dept_chair_person: Sections $\longrightarrow$ sections:
setof $\longrightarrow$ {section}: @memberof $\longrightarrow$ {crsections}:
@Course $\longrightarrow$ {course}

Our experiments show that if we select the class **dept_chair_person** as forbidden node for the first parameter or pick the intermediate class c = **instructors** for the second parameter, then the desired path–method is obtained.

For the second case (path–method 33) the undesired path–method generated is shown below. This undesired path–method is generated because of traversal of relationship *DeptChairPerson* from the class **Department**, again similar to the case of the first undesired path–method above.

Department–Students ():

DeptChairPerson $\longrightarrow$ dept_chair_person: Sections $\longrightarrow$ sections:
setof $\longrightarrow$ {section}: @Students $\longrightarrow$ {students}:
@setof $\longrightarrow$ {{student}}: **union** $\longrightarrow$ {student}

Our experiments show that if we select the class **dept_chair_person** as forbidden node for the first parameter and or pick the intermediate class c = **instructors** for the second parameter, then the desired path–method is obtained.

As we see, these two mechanisms for dealing with cases of generated undesired path–methods work for the four such cases in our larger sample. More experiments are necessary to judge the success ratio of these two mechanisms.

CHAPTER 5

ALGORITHMS FOR COMPUTING ACCESS RELEVANCE IN AN OODB

In Chapter 3 we have discussed motivations for using access weights and rules for access weight assignment. In Chapter 4 we discussed the definition of access relevance. In this chapter we describe efficient algorithms for computing access relevance.

5.1 Access Relevance Computation for the PRODUCT Weighting Function

We propose an algorithm *PRODUCT_AR* for the PRODUCT weighting function which computes access relevance from a source class to all the classes in the schema. This algorithm is a variation of the well-known nearest neighbor greedy algorithm of Dijkstra (e.g., [AHU83]). The algorithm of Dijkstra solves the single source shortest path problem to all the targets in the graph. The shortest path is defined as the path with the minimum sum of weights. In order to find the access relevance for all pairs of classes in a schema of n classes we need to apply the algorithm, *PRODUCT_AR*, n times, once for each class as a source class.

The *PRODUCT_AR* algorithm finds the access relevance AR[v] from a source class represented by node s to every other class v. The algorithm is described in terms

of the graph representation of the schema. It assumes, without loss of generality, that vertices are labeled by consecutive natural numbers, V = {1, 2, ..., n}. The algorithm works by maintaining a set S of nodes whose maximum access relevance from the source is already computed. Initially, S contains only the source node {s}. At each step, we add to S a node $u \in$ V–S of maximum access relevance. A path from s to a node v is called *special* if all its nodes (except possibly v itself) belong to S. At each step of the algorithm, we use an array AR to record the maximum access relevance value of a special path to each node. W is a two-dimensional array, where W[i, j] is the access weight of the edge (i, j). If there is no edge (i, j), then we assume W[i, j] = 0. In each step, after u is chosen to be inserted into S, a new special path to v, $v \in$ V – S – {u}, containing u, may result, that has a larger ARV than until now. Hence, we update AR[v] for each node $v \in$ V–S as follows. AR[v] is the maximum of two values: (1) The old AR[v] containing the access relevance of a special path not containing u; and (2) AR[u] * W[u, v] representing the access relevance of a special path containing u as the last node before v. Once S includes all nodes, all paths are "special," so AR[v] will hold the maximum access relevance from the source to each node $v \in$ V.

```
     Procedure PRODUCT_AR (IN s: node, OUT AR: array[1..n] of REAL)
     begin
(1)      S := {s};
(2)      for each node v other than s do
(3)           AR[v] := W[s, v];
(4)      for i := 1 to n–1 do begin
(5)           choose a node u in V – S such that
                   AR[u] is a maximum;
(6)           S := S ∪ {u};
```

```
(7)              for each node v in V - S do
(8)                  AR[v] := max (AR[v], AR[u] * W[u, v])
         end
    end;
```

The described algorithm differs from Dijkstra's algorithm by using the weighting function "*" in line (8) instead of the usual "+" operator. We will argue below why this change results in a correct algorithm that preserves the essential features of Dijkstra's algorithm. This algorithm will work for an undirected graph as well.

We will consider the schema shown in Figure 5.1, which is not directly a sub-schema of a university OODB but a variation of it. We have chosen this schema such that the same schema is used as one of two schemas to illustrate computation of access relevance in an interoperable multi-OODB. The corresponding directed graph representation is shown in Figure 5.2.

Let us apply *PRODUCT_AR* to the directed graph of Figure 5.2 assuming Rule 2b for specialization relationships. The source is 12 (**professor**). In steps (2)–(3), S = {12}, AR[3] = 0.3, AR[1] = 0.0, AR[7] = 0.7, and for the rest of the entries of the array, AR = 0. In the first iteration of the for-loop of lines (4)–(8), u = 7 is selected as the node with the maximum ARV. Then we set AR[8] = max(0, 0.7 * 0.5) = 0.35. Other values of the array AR do not change. The sequence of the AR values after each iteration of the for-loop is shown in Table 5.1.

The results of this application of *PRODUCT_AR* appear in row 5 in the matrix ARM (Access Relevance Matrix) of Table 5.2, showing the access relevance for each pair of nodes. Note that each diagonal value ARM[i, i] is set to 1.0. We will now

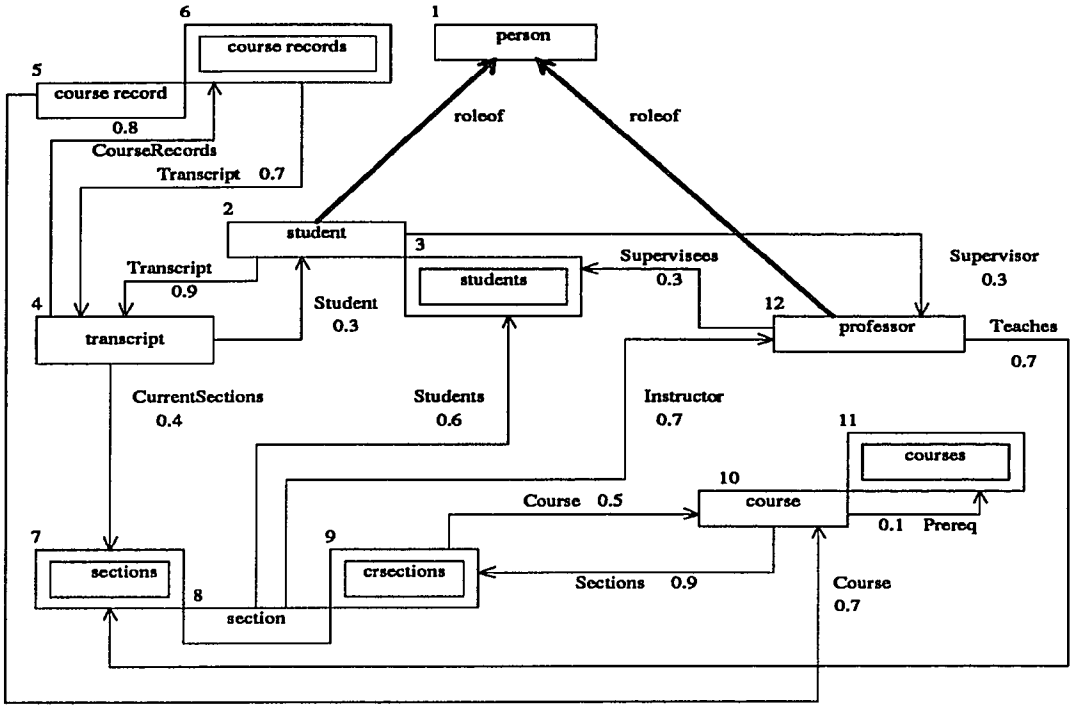

Figure 5.1 A Subschema of a University Database

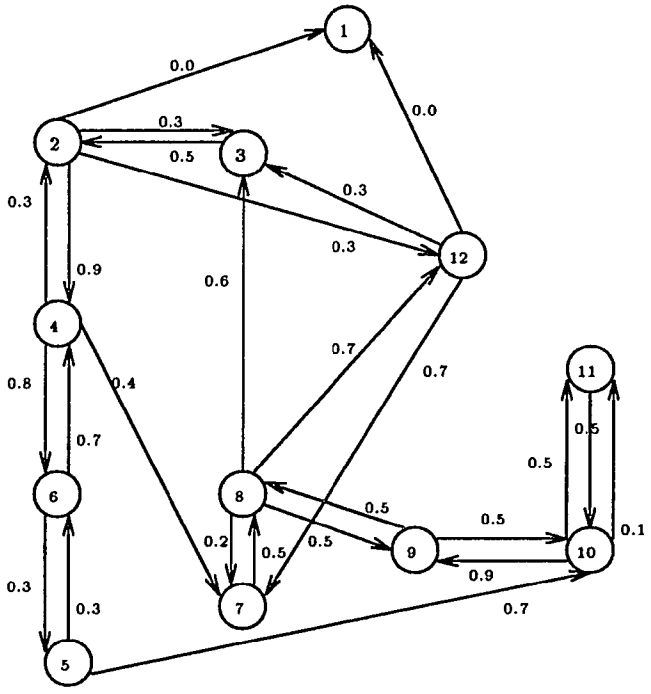

Figure 5.2 The Subschema as a Directed Graph

prove the validity of the *PRODUCT_AR* algorithm.

Iteration	S	u	new value of AR
Initial	{12}	–	AR[3] = 0.3, AR[1] = 0.0, AR[7] = 0.7
1	{12, 7}	7	AR[8] = 0.35
2	{12, 7, 8}	8	AR[9] = 0.175
3	{12, 7, 8, 3}	3	AR[2] = 0.150
4	{12, 7, 8, 3, 9}	9	AR[10] = 0.088
5	{12, 7, 8, 3, 9, 2}	2	AR[4] = 0.135
6	{12, 7, 8, 3, 9, 2, 4}	4	AR[6] = 0.108
7	{12, 7, 8, 3, 9, 2, 4, 6}	6	AR[5] = 0.032
8	{12, 7, 8, 3, 9, 2, 4, 6, 10}	10	AR[11] = 0.044
9	{12, 7, 8, 3, 9, 2, 4, 6, 10, 11}	11	–
10	{12, 7, 8, 3, 9, 2, 4, 6, 10, 11, 5 }	5	–
11	{12, 7, 8, 3, 9, 2, 4, 6, 10, 11, 5, 1}	1	–

Table 5.1 Computation of PRODUCT_AR on Graph of Figure 5.2

Theorem 2: In the *PRODUCT_AR* algorithm, AR[v] contains at all times the highest access relevance of a special path from node s to node v, for every node $v \in$ V.

Proof: The proof is by induction on the iterations of the algorithm. Initially, the theorem is true following lines (2) and (3) of the algorithm, since S = $\{s\}$ and the only existing special path contains just s and v. Suppose the theorem is true before a node u is added to S, and prove it is true after u is added to S. By the induction, the theorem is true for the nodes of S for special paths not containing u. We now show that a special path containing u can not increase AR[v], for $v \in$ S. By the order of selecting nodes for S, AR[v] $\geq$ AR[u], since whenever AR[u] is increased by update (step (8)) through a node x selected for S after v, AR[x] $\leq$ AR[v] and AR[u] $\leq$ AR[x] (since W[x, u] ≤ 1). The theorem is true for u itself by its choice. Then, it is left to prove the theorem for all nodes of V – S.

To support the argument observe that when we add a new node u to S at line (6), lines (7) and (8) adjust AR to take account of the possibility that there is now a special path to v going through u. If that path goes through the old S to u and then immediately to v, its access relevance, AR[u] * W[u, v], will be compared with AR[v] at line (8), and AR[v] will be increased if the new special path has higher access relevance.

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.3	0.9	0.216	0.72	0.36	0.18	0.136	0.151	0.076	0.3
3	0.0	0.5	1.0	0.45	0.108	0.36	0.18	0.09	0.068	0.076	0.038	0.15
4	0.0	0.3	0.12	1.0	0.24	0.8	0.4	0.2	0.151	0.168	0.084	0.14
5	0.0	0.095	0.189	0.21	1.0	0.3	0.154	0.315	0.63	0.7	0.35	0.221
6	0.0	0.21	0.084	0.7	0.3	1.0	0.28	0.14	0.189	0.21	0.105	0.098
7	0.0	0.15	0.3	0.135	0.032	0.108	1.0	0.5	0.25	0.125	0.063	0.35
8	0.0	0.3	0.6	0.27	0.065	0.216	0.49	1.0	0.5	0.25	0.125	0.7
9	0.0	0.15	0.3	0.135	0.032	0.108	0.245	0.5	1.0	0.5	0.25	0.35
10	0.0	0.135	0.27	0.122	0.029	0.097	0.221	0.45	0.9	1.0	0.5	0.315
11	0.0	0.068	0.135	0.061	0.015	0.049	0.11	0.225	0.45	0.5	1.0	0.158
12	0.0	0.15	0.3	0.135	0.032	0.108	0.7	0.35	0.175	0.088	0.044	1.0

Table 5.2 Access Relevance Matrix ARM for Graph of Figure 5.2

The only possibility for a special path with higher access relevance is shown in Figure 5.3, where the path travels to u, and then back into the old S, to node x of the old S, then possibly through other nodes of S to v. But as we now show, such a path cannot exist. Since x was placed in S before u, AR[x] $\geq$ AR[u] (first paragraph of proof). Thus, there exists a special path with the highest access relevance from

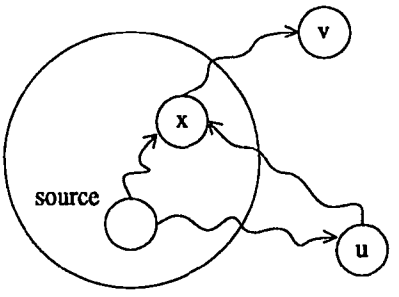

Figure 5.3 Impossible Special Path

the source to x which runs only through nodes of the old S. Therefore, the path to x through u as shown in Figure 5.3 is of no higher access relevance than the path directly to x through S, since the PRODUCT weighting function cannot increase access relevance along the path. Thus, AR[v] cannot be increased by a path through u and x as in Figure 5.3, and we need not consider the corresponding update of the access relevance of such paths. □

When the operation of the algorithm is complete S = V, i.e., all paths are special paths. Hence, Theorem 2 implies that AR[v] is the highest access relevance of a general path to v when the algorithm is completed.

The running time of *PRODUCT_AR* algorithm is O(n^2). If $e = |E|$ is much less than n^2, we might do better by using an adjacency list representation of the directed graph and using a priority queue implemented as a heap [AHU83] to organize the nodes in V–S. Choosing and deleting a maximum access relevance node from S in lines (5) and (6) takes O(lg n) time. This operation is repeated n times yielding O(n lg n) time. The loop of lines (7) and (8) can then be implemented by going down

the adjacency list for u and updating the access relevance in the priority queue. At most a total of e updates will be made, each at a cost of O(lg n), so the total time is now O(e lg n), rather than O(n^2). Thus, running time of *PRODUCT_AR* algorithm is O(e lg n). This running time is considerably better than O(n^2) if $e << n^2$, as it is for a typical OODB schema whose graph representation is a sparse graph.

5.2 An Algorithm for the MINIMUM Weighting Function

We now present an algorithm *MINIMUM_AR* for the MINIMUM weighting function which computes access relevance from a source s to all other classes in the schema. This algorithm is similar to the previous algorithm *PRODUCT_AR*.

The algorithm begins with a set S initialized to source $\{s\}$. At each step the algorithm chooses a node $u \in$ V – S maximizing AR[u]. The main difference from the previous algorithm is in the mechanism for updating AR[v] for all $v \in$ V – S. For each neighbor v of u, after u is added to S, we compare the value of the access relevance of u with the access weight of the edge (u, v). The minimum of these two values is compared to the current access relevance of v. If this minimum value is higher than the current AR[v], then AR[v] is set to this value. As for the *PRODUCT_AR* algorithm if there is no edge (i, j) $\in$ E then we define W[i, j] = 0.

```
     Procedure MINIMUM_AR (IN s: node, OUT AR: array[1..n] of REAL)
     begin
(1)      S := {s};
(2)      for each node v other than s do
(3)           AR[v] := W[s, v];
(4)      for i := 1 to n–1 do begin
```

```
(5)             choose a node u in V - S such that
                    AR[u] is a maximum;
(6)             S := S ∪ u;
(7)             for each node v in V - S do
(8)                 AR[v] := max (AR[v], min(AR[u], W[u, v]))
        end
    end;
```

Let us apply *MINIMUM_AR* algorithm to the graph shown in the Figure 5.2. The results for this algorithm, for source node 12, are shown in Table 5.3. Note that in Step 2 the access relevance value of 3, AR[3] is updated from 0.3 to 0.6. By applying this algorithm to each node in the schema, we compute all–pair access relevance (similar to Table 5.2 for PRODUCT_AR).

Iteration	S	u	new value of AR
Initial	{12}	–	AR[3] = 0.3, AR[1] = 0.0, AR[7] = 0.7
1	{12, 7}	7	AR[8] = 0.5
2	{12, 7, 8}	8	AR[9] = 0.5
3	{12, 7, 8, 3}	3	AR[2] = 0.5
4	{12, 7, 8, 3, 9}	9	AR[10] = 0.5
5	{12, 7, 8, 3, 9, 2}	2	AR[4] = 0.5
6	{12, 7, 8, 3, 9, 2, 10}	10	AR[11] = 0.5
7	{12, 7, 8, 3, 9, 2, 10, 4}	4	AR[6] = 0.5
8	{12, 7, 8, 3, 9, 2, 10, 4, 11}	11	–
9	{12, 7, 8, 3, 9, 2, 10, 4, 11, 6}	6	AR[5] = 0.3
10	{12, 7, 8, 3, 9, 2, 10, 4, 11, 6, 5}	5	–
11	{12, 7, 8, 3, 9, 2, 10, 4, 11, 6, 5, 1}	1	–

Table 5.3 Computation of MINIMUM_AR on Graph of Figure 5.2

For validity proof of *MINIMUM_AR* Theorem 3 is given below.

Theorem 3: In the *MINIMUM_AR* algorithm, AR[v] contains at all times the highest access relevance of a special path from node s to node v, for every node $v \in$ V.

Proof: The proof is by induction on the iterations of the algorithm. Initially, the theorem is true following lines (2) and (3) of the algorithm, since S = $\{s\}$ and the

only existing special path contains just s and v. Suppose the theorem is true before a node u is added to S, and prove it is true after u is added to S. By the induction, the theorem is true for the nodes of S for special paths not containing u. We now show that a special path containing u can not increase AR[v], for $v \in$ S. By the order of selecting nodes for S, AR[v] $\geq$ AR[u], since whenever AR[u] is increased by update (step (8)) through a node x selected for S after v, AR[x] $\leq$ AR[v] and AR[u] $\leq$ AR[x] (since W[x, u] $\leq$ 1). The theorem is true for u itself by its choice. Then, it is left to prove the theorem for all nodes of V – S.

To support the argument observe that when we add a new node u to S at line (6), lines (7) and (8) adjust AR to take account of the possibility that there is now a special path to v going through u. If that path goes through the old S to u and then immediately to v, its access relevance, min (AR[u], W[u, v]), will be compared with AR[v] at line (8), and AR[v] will be increased if the new special path has higher access relevance.

The only possibility for a special path with higher access relevance is shown in Figure 5.3, where the path travels to u, and then back into the old S, to node x of the old S, then possibly through other nodes of S to v. But as we now show, such a path cannot exist. Since x was placed in S before u, AR[x] $\geq$ AR[u] (first paragraph of proof). Thus, there exists a special path with the highest access relevance from the source to x which runs only through nodes of the old S. Therefore, the path to x through u as shown in Figure 5.3 is of no higher access relevance than the path

directly to x through S, since the MINIMUM weighting function cannot increase access relevance along the path. Thus, AR[v] cannot be increased by a path through u and x as in Figure 5.3, and we need not consider the corresponding update of the access relevance of such paths. □

When the operation of the algorithm is complete S = V, i.e., all paths are special paths. Hence, Theorem 3 implies that AR[v] is the highest access relevance of a general path to v when the algorithm is completed.

The running time of *MINIMUM_AR* algorithm is similar to the running time of *PRODUCT_AR*.

5.3 An Improved Algorithm for Bidirected Schemas

The graph representation of an OODINI OODB schema is a general directed graph. Examples of other OODB systems with directed relationships are VML [KNBD92], GemStone [BOS91], and ORION [K90]. By "directed relationships" we mean that a connection does not guarantee an inverse connection. In addition, if a relationship has an inverse relationship it may have a different access weight (see, e.g., Figure 5.1). The reason is that the weight of a relationship is determined by its relative traversal frequency for the class for which it is defined. Thus, two directed opposite relationships may have different access weights due to the different relative frequencies of traversal of the connections for each of the classes. However, several object-oriented database systems, e.g., ObjectStore [OHMS92] and ONTOS [M91], model each con-

nection as bidirectional. Such bidirectional schemas can be represented as undirected graphs.

The *PRODUCT_AR* and the *MINIMUM_AR* algorithms are applicable to bidirectional schemas. By applying these algorithms to all nodes as source nodes, we can compute the access relevance for all pairs of nodes in $\min(O(n^3), O(ne \log n))$ time. However, we shall present a more efficient algorithm for the MINIMUM weighting function for bidirectional schemas requiring only $O(n^2)$ time.

A *spanning tree* of a graph is a subgraph which is a tree that connects all the nodes of the graph [AHU83]. A maximum–weight spanning tree (MWST) is a spanning tree maximizing the sum of the weights of the edges in the tree compared to that sum of all other possible spanning trees. Our algorithm for bidirectional schemas is based on the following theorem.

Theorem 4: Let T be an MWST of an undirected graph G = (V, E). The unique path P in T between a node s and a node t is a most relevant path between s and t in G.

Proof: By contradiction. Let P be the path in T between s and t with access relevance AR(P). There is an edge $(u', v') \in$ P such that $W(u', v') =$ AR(P). Assume that there exists another path Q $\in$ G between s and t, which is not necessarily in T, with access relevance AR(Q), such that AR(Q) > AR(P). Now, deleting (u', v') from T will result in two disconnected subtrees T_1 and T_2. There exists an edge $(u, v) \in$ Q which connects two nodes $u \in T_1$ and $v \in T_2$. It is also true that $W(u, v) > W(u', v')$

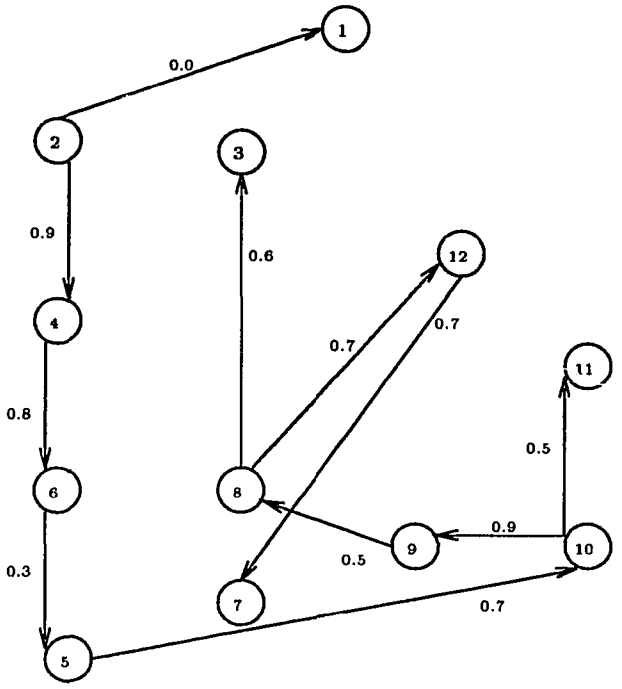

Figure 5.4 The Rooted MWST (Rooted at 2)

because each edge $\in$ Q has access weight higher than AR(P) = W(u', v') (MINIMUM weighting function). Now let T' = (T – {(u', v')}) $\cup$ {(u, v)}. Obviously, the weight of T' is larger than the weight of T. This contradicts our assumption that T is an MWST. Hence P is a most relevant path between s and t. □

Our algorithm is based on first finding an MWST of the bidirectional schema. There are famous algorithms of Prim and of Kruskal [AHU83] for this purpose. The Prim algorithm, which requires O(n^2) time can be obtained from our *MINIMUM_AR* algorithm by replacing min(AR[u], W[u, v]) by W[u, v] in line (8).

Theorem 4 shows that a MWST yields the maximum access relevance paths for a bidirectional schema. However, we shall show that for a directional schema an MWST, rooted at s, does not necessarily yield the maximum access relevance values. Hence,

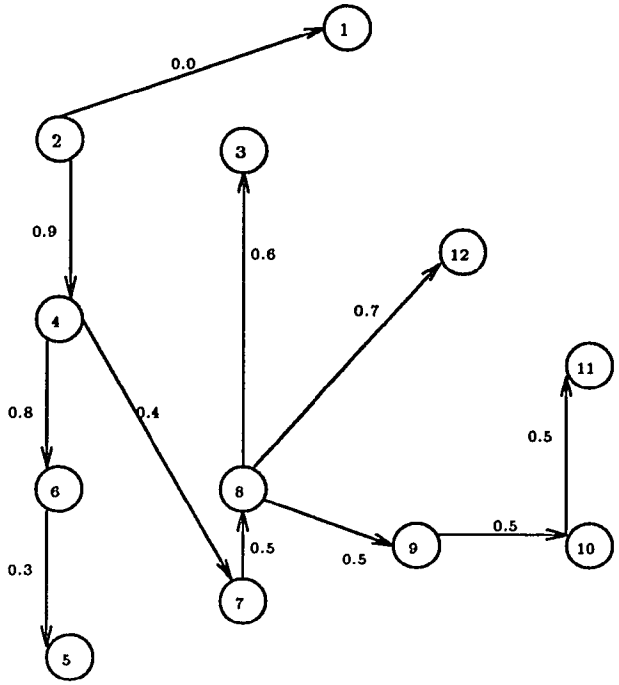

Figure 5.5 A Rooted AR Spanning Tree (Rooted at 2)

this approach is not applicable for directed schemas. See Figure 5.4 and Figure 5.5 showing a rooted MWST and a *rooted AR spanning tree* maximizing the ARVs from 2 to all nodes, respectively. The sum of access weights for the MWST rooted at 2 is 6.6, while for the AR spanning tree rooted at 2 it is 5.7. The ARV for the nodes 7, 8, 9, 10, 11, 12, and 3 are all equal to 0.4 for the AR spanning tree. The ARVs (i.e., the MINIMUM values) for these nodes in the MWST are 0.3. Thus, the MINIMUM_AR algorithm of the previous section cannot be applied to a rooted MWST of a directed graph.

By Theorem 4, the weights of the n–1 edges of the MWST enable us to compute the access relevance for each pair of nodes as the minimum weight along the unique path connecting the pair. The following algorithm computes the access relevance for

all pairs of nodes for a bidirected graph. It stores the access relevance in a matrix ARM[i, j] for each pair (i, j), $i < j$, requiring only $O(n^2)$ time for calculating these $n(n-1)/2$ values. Since Prim's algorithm requires $O(n^2)$ too, this is the complexity of finding all–pair access relevance for a bidirectional schema.

The algorithm *COMPUTE_ARM* first initializes all elements of the matrix ARM to 1. This is different compared to the previous algorithms because in line (8) of *COMPUTE_ARM* we select a minimum value while in *PRODUCT_AR* and *MINIMUM_AR* we needed a maximum. Therefore, we initialize with the largest possible weight, which is 1. It then begins with a set U of nodes initialized to {1}. In each iteration it calculates ARM between all nodes $x \in$ U and a new node $v \in$ V–U, adjacent to a node $u \in$ U, using ARM[x, u]. For this, the algorithm chooses an edge (u, v) such that $u \in$ U and $v \in$ V–U. Then it computes the access relevance from each node $x \in$ U to v, by choosing the minimum of W[u, v] and ARM[x, v]. This is because the bottleneck edge on the path between x and v in T is either the new edge (u, v) or the bottleneck edge of the path between x and u in T. Then the algorithm adds the node v to U, finishing the iteration. It terminates when U = V. Without loss of generality, we assume that the nodes are renumbered in the order of their traversal. Thus, we compute ARM[i, j] only for $i < j$.

```
     Procedure COMPUTE_ARM (IN T: tree; OUT ARM: matrix)
     var
          U: set of nodes;
          u, v, x: node;
     begin
(1)       for i := 1 to n do
(2)            for j := i + 1 to n do
```

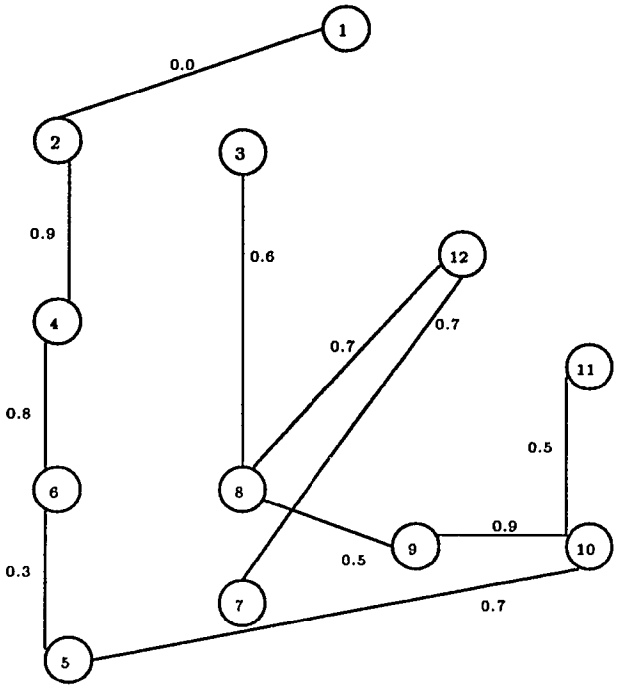

Figure 5.6 The Undirected Rooted MWST

```
(3)                 ARM[i, j] := 1;
(4)         U := {1};
(5)         while U ≠ V do begin
(6)               let (u, v) be an edge in T such that
                        u is in U and v is in V–U;
(7)               for each node x ∈ U do
(8)                     ARM[v, x] := min(W[v, u], ARM[u, x])
(9)               U := U ∪ {v};
            end
      end;
```

To demonstrate the operation of the algorithm *COMPUTE_ARM* we shall use the bidirected MWST of Figure 5.6. This MWST is actually the bidirected version of the MWST of Figure 5.4. We shall demonstrate the iteration when nodes 11, 10, 9, and 8 are in U and the next edge to be added is (8, 3). The triangular matrix of Figure 5.7 shows access relevance calculated by the algorithm *COMPUTE_ARM*.

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	0	0	0	0	0	0	0	0
2		1	0.3	0.9	0.8	0.3	0.3	0.3	0.3	0.3	0.3	0.3
3			1	0.3	0.5	0.3	0.6	(0.6)	(0.5)	(0.5)	(0.5)	0.6
4				1	0.3	0.8	0.3	0.3	0.3	0.3	0.3	0.3
5					1	0.3	0.5	0.5	0.7	0.7	0.5	0.5
6						1	0.3	0.3	0.3	0.3	0.3	0.3
7							1	0.7	0.5	0.5	0.5	0.7
8								1	0.5	0.5	0.5	0.7
9									1	0.9	0.5	0.5
10										1	0.5	0.5
11											1	0.5
12												1

Figure 5.7 The Triangular ARM Matrix

It is clear from this triangular matrix that we need to store only $(n^2/2)$ values for bidirectional schemas. All the encircled values of Figure 5.7 show the ARVs calculated in this iteration from all nodes in U to node 3. Note that the access relevance of paths between the pairs (9, 3), (10, 3), and (11, 3) is 0.5, due to the values of ARM[9, 3], ARM[10, 3], and ARM[11, 3] which are 0.5. The access relevance of the path between the pair (8, 3) is 0.6 due to the access weight of the edge (8, 3).

For the validity proof of *COMPUTE_ARM* algorithm Theorem 5 is given below.

Theorem 5: In the *COMPUTE_ARM* algorithm, ARM[x, v] contains the access relevance from $x \in$ U to $v \in$ U.

Proof: The proof is by induction of the algorithm. Initially, the theorem is true following line (4) of the algorithm, since U = {1}, there is only one node in U.

Suppose the theorem is true before a node v is added to U, and prove it is true after v is added to S. When v is added to U, in line (8) of the algorithm ARM[v, x], $x \in$ U, is decreased using min (W[v, u], ARM[u, x]). Since *COMPUTE_ARM* algorithm is applied to a tree T, there exists only one path between any two nodes in T. Thus, every path from v to $x \in$ U, contains edge (v, u). As ARM[u, x] contains the access relevance from u to x, the only way ARM[v, x] can be lower than ARM[u, x] is if W[v, u] $<$ ARM[u, x], as considered in the algorithm.

In T there are $n - 1$ edges, where n is the number of nodes in T. The algorithm considers all the edges, i.e., all the nodes in T. When the algorithm is complete U = V, ARM contains access relevance all–pair access relevance. □

CHAPTER 6

COMPUTING ACCESS RELEVANCE IN AN INTEROPERABLE MULTI–OODB

For large scale interoperable databases the path–method mechanism for supporting schema independent query formulation is even more important, as it is unrealistic to maintain a completely integrated schema which equally serves all users' needs. Rather, only a loosely coupled form of interoperable multi–database [SL90] can be achieved by specifying simple cross-database relationships. In such interconnected schemas it is particularly difficult for individual users to combine information from multiple resources, i.e., to choose the most adequate cross-database relationship for navigating between the different schemas and further navigate in a different schema. We will discuss efficient algorithms to compute access relevance in an IM–OODB.

We assume an IM–OODB system, i.e., each autonomous database is an OODB. However, the different OODBs may use different object–oriented database models. For communication between different OODBs few classes of each component OODB need connections to classes in other component OODBs. We will describe an approach how to realize such connections based on an object–oriented approach for partial–integration of database systems discussed in [CT91a, CT91b]. Other approaches using object–oriented data models for integration of heterogeneous databases are, e.g., [B89,

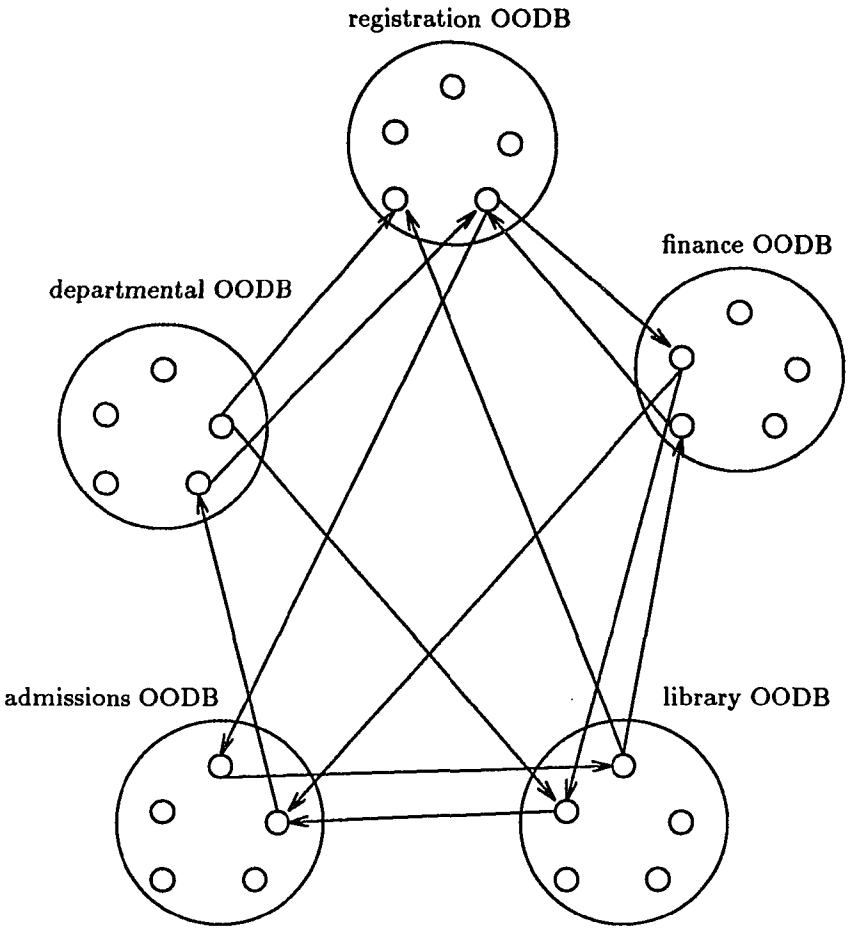

Figure 6.1 An Interoperable Multi-OODBs System

KDN90]. In [GMPN92, GPN91a, GPNS92, GPCS92] a new integration technique "structural integration" has been developed using an object–oriented approach.

Let us consider, for example, a university environment which typically contains several academic units. Each unit has its own autonomous OODB which contains necessary information for its day–to–day operations. In addition, these OODBs are interoperable, because it is necessary for one unit to access information from other units. An IM–OODB for a university environment, which consists of five OODBs, is shown in Figure 6.1. The *admissions OODB* contains information regarding stu-

dent_applicants, admission requirements, degree programs, etc. The *registration OODB* contains information regarding students, courses, transcripts, etc. This is the subschema discussed in the previous chapter. The *departmental OODB* contains information for an academic department regarding professors, chair_person, etc. The *finance OODB* contains information regarding student–fees, employee–salaries, tuition–remission, budgets, etc. The *library OODB* contains information regarding books, journals, proceedings, periodicals, lending, etc.

In an IM–OODB one component OODB can retrieve information from another component OODB. For example, the *registration OODB* retrieves information from the *departmental OODB* about every professors's teaching sections. It retrieves information from the *admissions OODB* about the admission status for new students, from the *library OODB* about overdue books before a student may graduate, and from the *finance OODB* about overdue payments of fees before a student may register each semester.

We have shown only a few classes of each component OODB in Figure 6.1, but in reality the number of classes in each OODB is large. Let us assume that each component has n classes. All–pair computation of access relevance for each component OODB requires computation of n^2 values. As we have k component OODBs it is necessary to compute kn^2 values. If these component OODBs are interoperable, there are totally kn classes which require computation of $(kn)^2$ values, a number much larger than kn^2. Even if we subtract the kn^2 values that were already computed, the

combination into a Multi–OODB still requires to compute and store $k(k-1)n^2$ values, a number that is large compared to the number of values stored for all individual databases. Therefore, it is not practical to simply extend our approach from Chapter 5 to IM–OODBs. Thus, we apply a hierarchical approach. While the internal values for each component OODB are precomputed, the values between pairs of classes from two different OODB components will be computed on the fly, based on the precomputed access relevance of each component OODB and the access weights of the relatively few connections between pairs of classes from different OODBs. For this purpose we model in the next sections the whole IM–OODB as a relatively small graph for which we apply the algorithms of Chapter 5. Using this model we present efficient algorithms for the required online computations in the next sections.

6.1 An IM–OODB Containing Only Two OODBs

In this section we discuss the computation of access relevance in an IM–OODB containing only two databases. For this special case we present an algorithm which is more efficient than in the general case. Furthermore, this case will serve as an introduction to the more complex general case in Section 6.2.

To explain how to realize a connection between two component OODBs, we follow ideas from [CT91a]. The problem is that a class in an autonomous OODB cannot have pointers to a class in another OODB. The solution of [CT91a] selects two classes, one in each OODB, that represent the same real world objects. In their example the

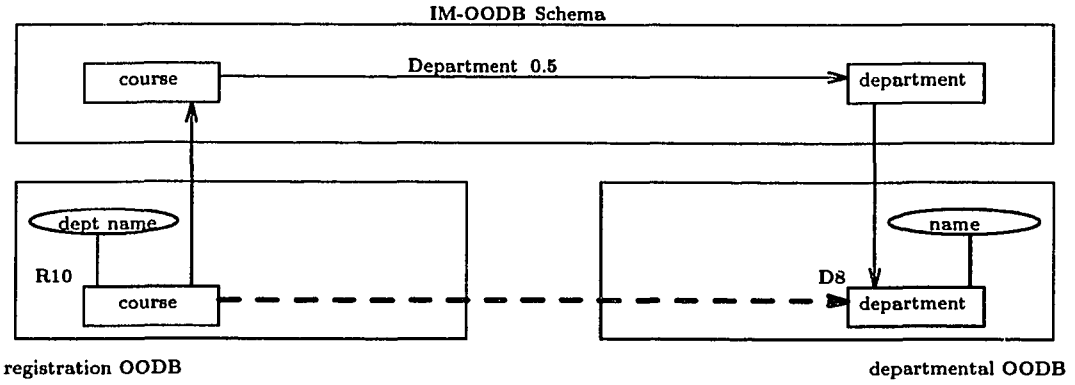

Figure 6.2 Realization of Connections between Two Component OODB

two classes represent social security numbers, one with dashes and one as integers. For both of these classes they define a class in the IM–OODB schema. Then a correspondence is defined between the two IM–OODB schema classes and implemented as a mathematical transformation capable of transforming an instance of one class to an instance of the other class. This way, we avoid need for pointers for all instances, which cannot exist between IM–OODB schema classes. The connection between the two classes of the two OODBs is realized by a path–method (in our terminology) from one class to its IM–OODB schema class and on through the transformation to the IM–OODB schema class of the other class and then to the other class.

In [CT91a] every item of information is represented as a class. However, in our abstract model as well as in many other models (e.g., VML [KNBD92], ONTOS [M91], ObjectStore [OHMS92]), most items of information which are stored with a class are represented as attributes. For example, the social security number of a person will be an attribute of the class person. Thus, we have to modify the solution of [CT91a] for our model as follows. We pick two classes, one in each OODB, representing the

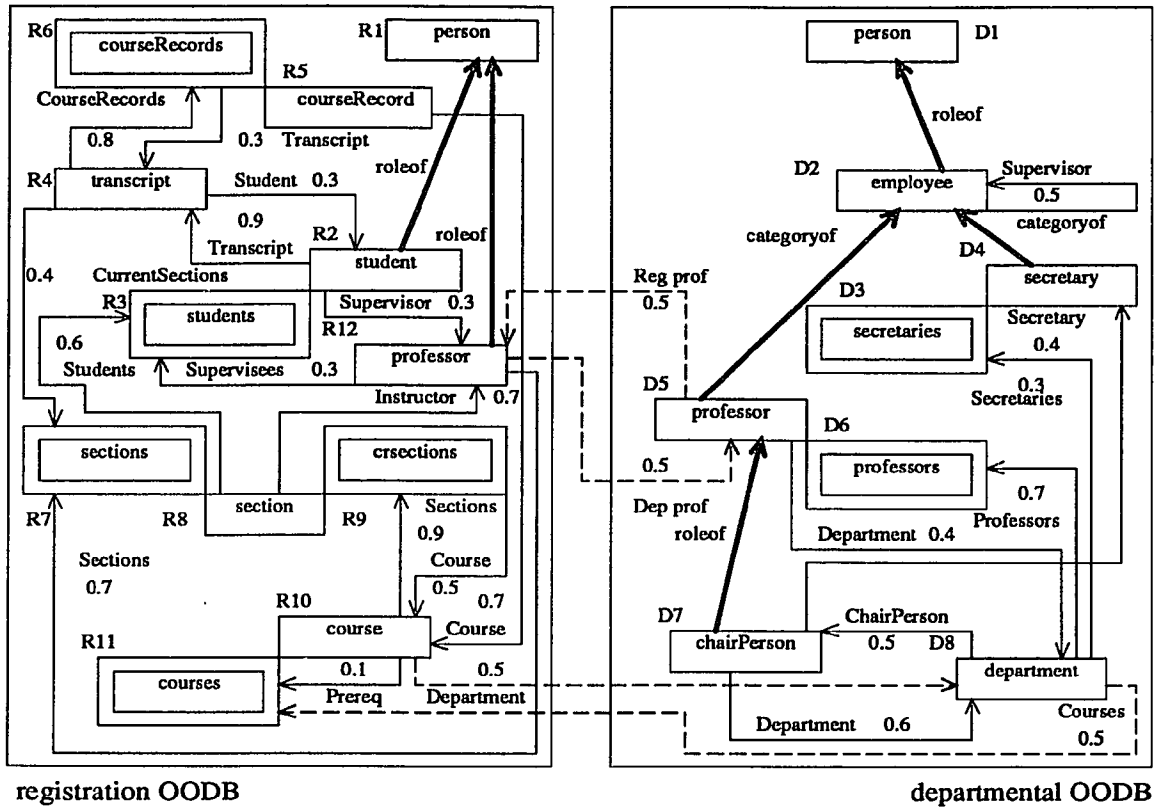

Figure 6.3 An IM–OODB Containing the Registration and the Departmental OODB

same real world object, e.g., in our upcoming example dep.professor and reg.professor to be represented in the IM–OODB schema. The dot notation is used to distinguish classes of different OODBs. The mathematical transformation between these two classes in the IM–OODB schema is based on the correspondence of their appropriate attributes, e.g., an attribute representing the name or the social security number of the professor.

However, in our model we can have a connection between two classes, one in each OODB, *even if they do not represent the same real world object.* If both classes have corresponding attributes representing the same real world information, the correspon-

dence can be realized based on the mathematical transformation of the attributes of the two classes, eventhough the classes do not represent the same real world object. This enables more flexibility in establishing connections between different OODBs, as seen in the following example.

In Figure 6.2 the class **course** of the registration OODB has an attribute *dept_name* which represents the department that offers this course. The class **department** in the departmental OODB has an attribute *name*. Presumably the department names used in these two databases are identical. Thus, we can have a path–method with the class sequence (course, im–course, im–department, department), where classes **im–course** and **im–department** are defined in the IM–OODB schema. An attribute–pair (*dept_name*, *name*) can be used for implementing the transformation in the IM–OODB schema.

Two small subschemas of the *registration OODB* (Figure 5.1 of Chapter 5) and the *departmental OODB* are shown in Figure 6.3 using OODINI. The corresponding graph representations appear in Figure 6.4. Both the OODBs have a class **professor**. Two path–methods between these two classes, *Dep_prof* and *Reg_prof*, consist of the class sequences (reg.professor, im–reg.professor, im–dep.professor, dep.professor) and (dep.professor, im–dep.professor, im–reg.professor, reg.professor), respectively. An attribute–pair (*name*, *name*) can be used for establishing the correspondence between instances. The path–method *Department* for the class **course** is described in Figure 6.2. There is a path–method *Courses* defined from the class **department**

to the class **courses** (Figure 6.3). The attribute *dept_name*, is also defined for the class **courses** and has a non-nil value only for instances of a set of courses of the same department. Thus, the same attribute-pair (*dept_name*, *name*) can be used for correspondence between such instances of the class **courses** and instances of the class **department**. There may be instances of the class **courses** representing a set of courses which have several department names. Those instances are created for other purposes, for example, prerequisites of a course can be a set of courses of different departments. Such instances are not considered for correspondence with the class **department**.

Let us consider another example for an IM-OODB for the case where Rule 2a (discussed in Chapter 3) causes unwanted traversal in an IM-OODB. Consider the access relevance from the class **course** (R10) to **dep.professor** (D5). One path is p_1, which represents the class sequence (course (R10), crsections (R9), section (R8), reg.professor (R12), dep.professor (D5)). This class sequence can be interpreted to find all the professors which teach the sections of a given course. Another path, p_2, represents the class sequence: (course (R10), department (D8), chair_person (D7), dep.professor (D5)). This path can be interpreted to find the instance of the **chair_person**, of the department of the given course, as a professor. That is, it finds the internal ID of the chair-person in the class **professor**. This path, which is enabled by traversing the *roleof* connection from **chair_person** to **professor** as its last connection, is inacceptable since its last connection provides no additional

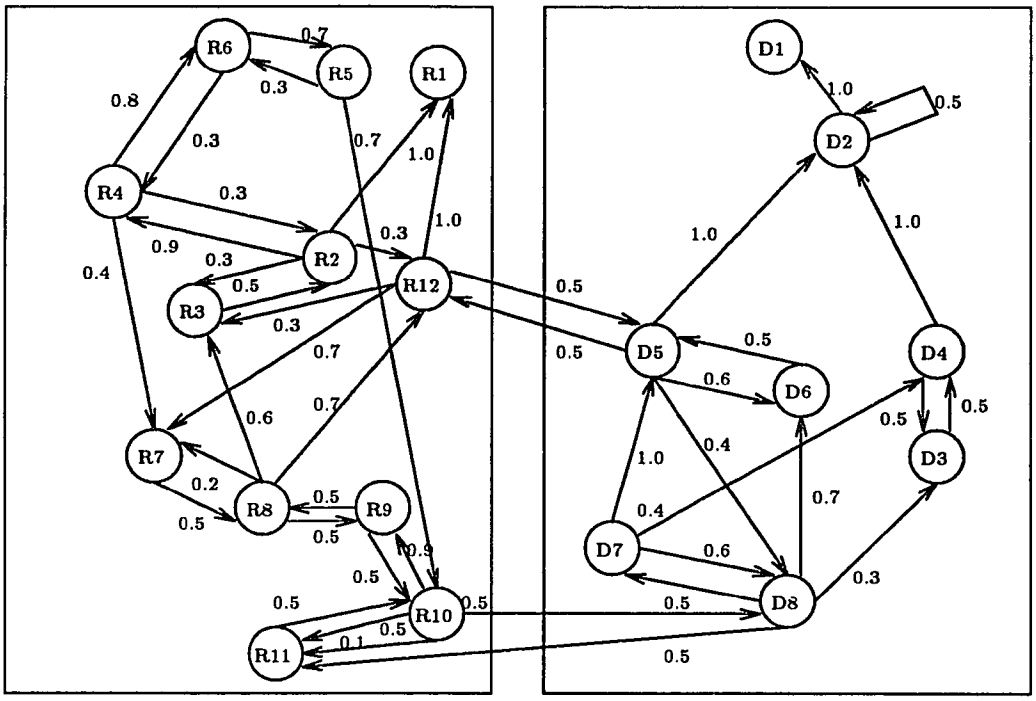

Figure 6.4 Registration and Departmental Schemas as a Directed Graph (Rule 2a)

information to the user. There is no meaning to traversing the *roleof* connection unless it is utilized to inherit a property of a professor to the chair_person such as the sections s/he teaches, in which case the traversal does not stop at the superclass. By the PRODUCT weighting function p_1 has an ARV = 0.158 and p_2 has an ARV = 0.25. However, p_2 is possible only due to the traversal of the *roleof* connection. Thus, we would like to block traversals through specialization connections while still having the inheritance properties. But an access weight of 1.0 enables such a traversal and furthermore gives it high priority.

To overcome such unwanted traversals we introduced Rule 2b (in Chapter 3) which avoids such traversal. The graph representation of the schema of Figure 6.3

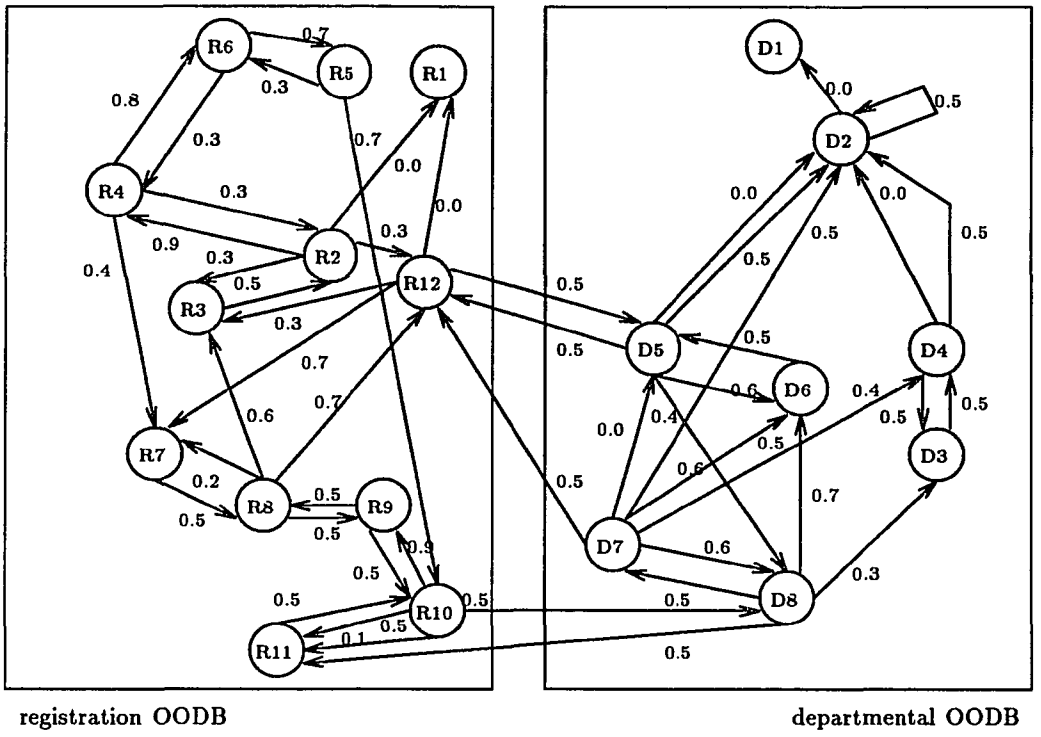

Figure 6.5 The Graph Representation of the Schema of Figure 8 using Rule 2b

according to Rule 2b appears in Figure 6.5. Note the extra edges of D4, D5, and D7 when compared to Figure 6.4. In the worst case the increase in edges could lead to $O(n^2)$ for a complete graph. In practice, the number of additionally introduced edges will be much smaller.

We will now define the problem of computing access relevance in an IM–OODB. In a typical IM–OODB, a component OODB is developed first, and later on it is added to the IM–OODB. We assume that all–pairs access relevance for each OODB are precomputed with the algorithms discussed in Chapter 5. In the following discussion we will define several terms needed to compute access relevance in an IM–OODB. Denote by a_p a class of $OODB_i$ and by b_q a class of $OODB_j$, where $i \neq j$.

Definition 2: A connection from a class $a_p \in \text{OODB}_i$ to another class $a_q \in \text{OODB}_i$ is called an *intra–OODB connection.*

Definition 3: A connection from a class $a_p \in \text{OODB}_i$ to another class $b_q \in \text{OODB}_j$, $i \neq j$, is called an *inter–OODB connection.*

As discussed earlier, such inter–OODB connections are realized as path–methods. Their weights are determined by Rule 3 since they should not interfere with weights of intra–OODB connections.

Rule 3: The sum of the weights on the outgoing inter–OODB connections of a class $\sum_{i=1}^{n} W_i = 0.5 * n$, where, n is the number of outgoing inter–OODB connections from this class. From this sum, each connection is assigned a weight from [0, 1], reflecting its relative frequency of traversal.

Definition 4: For each component OODB_i, a class $a_p \in \text{OODB}_i$, which has an inter–OODB connection *or* is referred to by an inter–OODB connection is called a *contact class.*

We will assume that there are relatively few inter–OODB connections and contact classes in IM–OODBs. Considering the difficulties in defining such classes [CT91a] this is a realistic assumption. In Figure 6.3, the relationship *Transcript* of the class **student** to the class **transcript** is an intra–OODB connection. The path–method *Department* of the class **course** to the class **department** is an inter–OODB connection. The classes **course** and **department** are contact classes.

Let a_s and a_t be classes in OODB_i. A path $P(a_s, a_t) = a_s(= a_{i_1}), a_{i_2}, \ldots, a_{i_k}(= a_t)$

using only intermediate classes of $OODB_i$ is called an *intra–OODB path.* Theoretically, there may exist a most relevant path between two classes of the same OODB going through classes of another OODB. But we will not consider such paths since it contradicts the autonomy assumption of the OODBs. This limitation will be relaxed in Section 6.2. Let a_p and b_q be classes of $OODB_i$ and $OODB_j$, $i \neq j$, respectively, then a path $P(a_p, b_q)$, is called an *inter–OODB* path. In Figure 6.3, the path of the class sequence: (student, transcript, sections) is an intra–OODB path, while the path of the class sequence: (section, reg.professor, dep.professor, department) is an inter–OODB path.

Definition 5: Let $P(a_p, b_q)$ be an inter–OODB path from class $a_p \in OODB_i$ to class $b_q \in OODB_j$, $i \neq j$. In general, such a path may contain several inter–OODB connections. An inter–OODB path containing only one inter–OODB connection is called a *direct inter–OODB path.*

Note that in an IM–OODB containing only two OODBs we shall assume first that an inter–OODB path is a direct inter–OODB path since other kinds of paths traversing back and forth between the two OODBs are very unlikely to have a reasonable interpretation. However, such paths will also be considered in Section 6.2.

A direct inter–OODB path P has the form $a_p(= a_{i_1}), a_{i_2}, \ldots, a_{i_k}, b_{j_1}, b_{j_2}, \ldots, b_{j_l}(= b_q)$ where a_{i_m}, $1 \leq m \leq k$ are classes of $OODB_i$ and b_{j_n}, $1 \leq n \leq l$ are classes of $OODB_j$. Hence, $(a_{i_r}, a_{i_{r+1}})$, $1 \leq r < k$ and $(b_{j_r}, b_{j_{r+1}})$, $1 \leq r < l$ are intra–OODB connections and (a_{i_k}, b_{j_1}) is the only inter–OODB connection in $P(a_p, b_q)$. The access

relevance value (ARV) of P for a weighting function WF is defined as

$$ARV(P) = WF\ \ (AR(a_p, a_{i_k}), W(a_{i_k}, b_{j_1}), AR(b_{j_1}, b_q))$$

The access relevance from a_p to b_q is defined by maximizing the access relevance ARV(P) over all paths P(a_p, b_q), that is, over all the paths P(a_p, a_{i_k}) and all the paths P(b_{j_1}, b_q), for all inter–OODB connections (a_{i_k}, b_{j_1}) between all possible contact classes $a_{i_k} \in$ OODB$_i$ and all possible contact classes $b_{j_1} \in$ OODB$_j$.

$$\begin{aligned} AR(a_p, b_q) &= \max_P ARV(P) \\ &= \max_{(all\ inter-OODB\ connections\ (a_{i_k}, b_{j_1}))} WF(AR(a_p, a_{i_k}), \\ &\qquad W(a_{i_k}, b_{j_1}), AR(b_{j_1}, b_q)) \end{aligned}$$

We assume that using the efficient algorithms of Chapter 5, access relevances for each component OODB are already computed and stored. All–pair access relevances for OODB$_i$ (OODB$_j$) are stored in a matrix ARM$_i$ (ARM$_j$). Thus we have a simple algorithm to compute AR(a_p, b_q) as follows (see Figure 6.6):

procedure Compute_AR_IM_OODB (**IN** a_p, b_q : class) ;
begin
(1) AR[a_p, b_q] := 0;
(2) **for** each inter–OODB connection (a_{i_k}, b_{j_1}) such that
(3) $a_{i_k} \in$ OODB$_i$ and $b_{j_1} \in$ OODB$_j$ **do**

$$AR[a_p, b_q] := \max(AR[a_p, b_q], WF\ (ARM_i[a_p, a_{i_k}], W[a_{i_k}, b_{j_1}], ARM_j[b_{j_1}, b_q]))$$

end

Suppose we want to find the access relevance between **student** and **department**. In step 1, AR[a_p, b_q] is set to zero. In the for–loop of step 2, for each inter–OODB connection we try to find the maximum access relevance value. Two

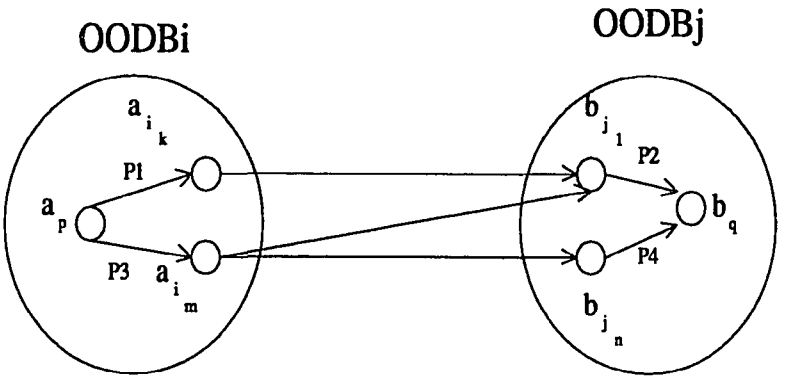

Figure 6.6 Computation of Access Relevance

access relevance matrices for registration OODB and departmental OODB are shown in Table 5.2 (Chapter 5) and Table 6.1, respectively. The steps of algorithm *Compute_AR_IM_OODB* for computing AR (for WF = PRODUCT) from **student** to **department** using two alternative inter-OODB connections (R12, D5), and (R10, D8) are shown below.

1. AR[R2, D8] := 0
2. AR[R2, D8] := max (AR[R2, D8], WF (AR[R2, R12], W[R12, D5], AR[D5, D8]))
   := max (0.0, WF (0.3, 0.5, 0.4)) := max (0.0, 0.06) := 0.06
3. AR[R2, D8] := max (AR[R2, D8], WF (AR[R2, R10], W[R10, D8], AR[D8, D8])
   := max (0.06, WF (0.151, 0.5, 1.0)) := max (0.06, 0.0755) := 0.0755

The complexity of this algorithm is $O(c)$ where c is the number of inter-OODB connections from $OODB_i$ to $OODB_j$. Typically in IM-OODBs, c is a constant or a sublinear function of n, such as $\lg n$ or $\sqrt{n}$. Hence, this is a very efficient algorithm, which is appropriate for online computation.

	1	2	3	4	5	6	7	8
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.25	1.0	0.5	0.0	0.0	0.0	0.0
4	0.0	0.5	0.5	1.0	0.0	0.0	0.0	0.0
5	0.0	0.5	0.12	0.06	1.0	0.6	0.0	0.4
6	0.0	0.25	0.06	0.03	0.5	1.0	0.0	0.2
7	0.0	0.5	0.2	0.4	0.25	0.5	1.0	0.6
8	0.0	0.18	0.3	0.15	0.35	0.7	0.0	1.0

Table 6.1 ARM for Departmental OODB

6.2 An IM–OODB Containing Many OODBs

Consider, for example, an IM–OODB with 5 OODBs: OODB_A, OODB_B, OODB_C, OODB_D, OODB_E. Denote a class of OODB_A (OODB_B, OODB_C, OODB_D, OODB_E) by a_i (b_m, c_l, d_k, e_j), respectively. Consider an inter–OODB path–method from a_p to b_q which involves classes of all 5 OODBs. This is an indirect inter–OODB path–method. The corresponding path P in G can have, for example, the form

$(a_p(= a_{i_1}), a_{i_2}, \ldots, a_{i_v}, e_{j_1}, e_{j_2}, \ldots, e_{j_w}, d_{k_1}, d_{k_2}, \ldots, d_{k_x}, c_{l_1}, c_{l_2}, \ldots, c_{l_y}, b_{m_1}, b_{m_2}, \ldots, b_{m_z}(= b_q))$.

This path involves 4 inter–OODB connections: (a_{i_v}, e_{j_1}), (e_{j_w}, d_{k_1}), (d_{k_x}, c_{l_1}), and (c_{l_y}, b_{m_1}). Others are intra–OODB connections. The access relevance value (ARV) of P for a weighting function WF is

$$ARV(P) = WF(AR(a_{i_1}, a_{i_v}), W(a_{i_v}, e_{j_1}), AR(e_{j_1}, e_{j_w}), W(e_{j_w}, d_{k_1}), AR(d_{k_1}, d_{k_x}),$$
$$W(d_{k_x}, c_{l_1}), AR(c_{l_1}, c_{l_y}), W(c_{l_y}, b_{m_1}), AR(b_{m_1}, b_{m_z}))$$

The access relevance from a_p to b_q is defined by maximizing access relevance values

ARV(P) over all paths P(a_p, b_q). Suppose for the moment that all those paths actually traverse these 5 OODBs in the same order of the above mentioned P, i.e., (A, E, D, C, B), Then we obtain

$$\begin{aligned} AR(a_p, b_q) &= \max_P ARV(P) \\ &= \max \; WF\,(AR(a_p, a_{i_v}), W(a_{i_v}, e_{j_1}), AR(e_{j_1}, e_{j_w}), W(e_{j_w}, d_{k_1}), \\ &\qquad AR(d_{k_1}, d_{k_x}), W(d_{k_x}, c_{l_1}), AR(c_{l_i}, c_{l_y}), W(c_{l_y}, b_{m_1}), AR(b_{m_1}, b_q)) \end{aligned}$$

where *max* is taken over all possible inter–OODB connections of the form (a_{i_v}, e_{j_1}), (e_{j_w}, d_{k_1}), (d_{k_x}, c_{l_1}), (c_{l_y}, b_{m_1}). The pair (a_{i_v}, e_{j_1}) stands for any pair of contact classes, such that a_{i_v} is in OODB_A, e_{j_1} is in OODB_B and there exists an edge between a_{i_v} and e_{j_1}. There might be several such edges, and maximization is done by selecting the one edge that leads to the largest overall access relevance value. The same applies to the other pairs of OODBs. The ARVs are precomputed for each OODB. The access weights of the inter–OODB connections are known to the IM–OODB system only.

If the number of such connections between every two OODBs is c then our optimization has to select from c^4 possibilities. Furthermore there are many more possible patterns of paths from OODB_A to OODB_B using different subsets (whose number is an exponential function of the number of OODBs) and different orders (whose number is a factorial function of the number of OODBs). Thus, it is not practical to extend the computation of the previous section to the case of many OODBs. Furthermore, paths may return to an OODB several times, using each time different classes. Although most of such paths would not have reasonable interpretations, a few of them

may be desired by a user and should be taken into account.

Thus, we shall look for a solution which involves modeling the graph representation G(V, E) of the IM–OODB as another small graph H(U, D). For each OODB, U will contain a node for each contact node of the OODB and an extra center node representing a variable node of the OODB. This variable node will at each time represent another node of the OODB. However, the graph H contains only one such node for each OODB, keeping the number of nodes of H relatively small. Each OODB is represented by a clique of its contact classes and a star with its variable class as center and its contact classes as end points. That is, for each OODB, D will contain "clique" edges connecting each pair of contact nodes and "star" edges connecting the variable node to each one of the contact nodes. In addition, D contains an edge for each inter–OODB connection of the IM–OODB. See Figure 6.7, for an example. The inter–OODB edges have given access weights. For clique edges we define the access weight as the access relevance in G between the two nodes of the edge, which are precomputed for each OODB, e.g., $W_H(e_{j_1}, e_{j_w}) = AR(e_{j_1}, e_{j_w})$. While H is described in Figure 6.7 as a bidirectional graph, since in general we can have a path from every node to every other node, the access weights are usually different for both directions unless the original OODBs' schemas are bidirectional. The access weight of a star edge varies. Whenever the center node a_i represents a specific node e.g., a_{i_1}, then the access weight of a star edge (a_i, a_{i_v}) is defined as the access relevance $AR(a_i, a_{i_v})$ in G, i.e., $W_H(a_i, a_{i_v}) = AR\ (a_i, a_{i_v})$.

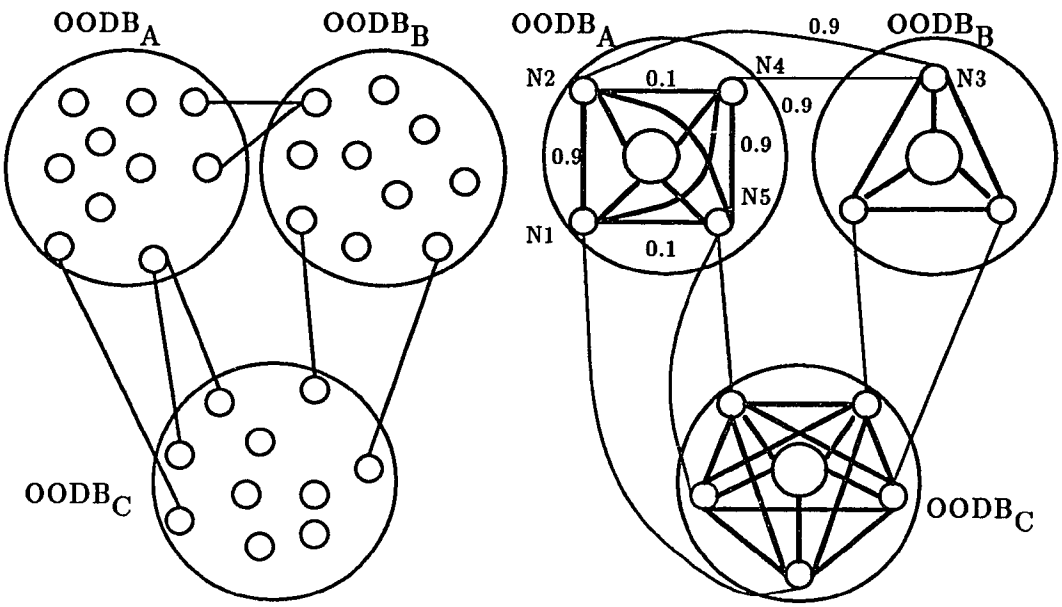

Figure 6.7 Two Graphs G(V, E) and H(U, D) for IM–OODB with Many OODBs

Definition 6: A path P between two center nodes in H is called *proper* if (1) all the rest of the nodes are contact nodes, (2) the sequence of edges except for the two end edges (which are star edges) consists of inter–OODB edges and clique edges such that no two clique edges are consecutive.

In general, the path will have inter–OODB edges and clique edges in alternating order, but a proper path may sometimes use only one contact node of an OODB rather than two, in the case that this is the only node of the path which belongs to this OODB.

Lemma 3: For every path Q between two center nodes in H there exists a proper path R such that AR(R) $\geq$ AR(Q).

Proof: Let Q be a non–proper path between the center nodes a_i and b_m in H. If Q

contains a center node, say c_s of OODB_C, then, by the definition of H, the two nodes adjacent to c_s in Q are contact nodes c_{l_i} and c_{l_j}. But c_s can be omitted from Q since H has a clique edge (c_{l_i}, c_{l_j}) and by the definition of access relevance in G, $\text{AR}(c_{l_i}, c_{l_j}) \geq WF\ (AR(c_{l_i}, c_s), AR(c_s, c_{l_j}))$. Thus, $\text{W}_H(c_{l_i}, c_{l_j}) \geq WF\ (W_H(c_{l_i}, c_s), W_H(c_s, c_{l_j}))$.

If Q contains two consecutive clique edges say (c_{l_i}, c_{l_j}) and (c_{l_j}, c_{l_k}) then we can replace them by the clique edge (c_{l_i}, c_{l_k}) since by the definition of the access relevance in G, $\text{AR}(c_{l_i}, c_{l_k}) \geq WF\ (AR(c_{l_i}, c_{l_j}), AR(c_{l_j}, c_{l_k}))$. Thus $\text{W}_H(c_{l_i}, c_{l_k}) \geq WF\ (W_H(c_{l_i}, c_{l_j}), W_H(c_{l_j}, c_{l_k}))$. The proper path R obtained from Q by performing these two kinds of transformations satisfies AR(R) $\geq$ AR(Q). □

Lemma 4: For every most relevant path P from a node a_p to a node b_q in G there exists a corresponding proper path Q in H from the center node a_i to the center node b_m, such that AR(P) = AR(Q).

Proof: Consider, for example, the most relevant path P from a_p to b_q

$P = (a_p(= a_{i_1}), a_{i_2}, \ldots, a_{i_v}, e_{j_1}, e_{j_2}, \ldots, e_{j_w}, d_{k_1}, d_{k_2}, \ldots, d_{k_x}, c_{l_1}, c_{l_2}, \ldots, c_{l_y}, b_{m_1}, b_{m_2}, \ldots, b_{m_z}(= b_q))$.

This path is represented in the graph H by a path

$Q = (a_p(= a_{i_1}), a_{i_v}, e_{j_1}, e_{j_w}, d_{k_1}, d_{k_x}, c_{l_1}, c_{l_y}, b_{m_1}, b_{m_z}(= b_q))$.

Obviously Q is a proper path in H.

$$\begin{aligned} AR_H(Q) \ &= WF(W_H(a_{i_1}, a_{i_v}), W_H(a_{i_v}, e_{j_1}), W_H(e_{j_1}, e_{j_w}), W_H(e_{j_w}, d_{k_1}), \\ &\quad W_H(d_{k_1}, d_{k_x}), W_H(d_{k_x}, c_{l_1}), W_H(c_{l_1}, c_{l_y}), W_H(c_{l_y}, b_{m_1}), W_H(b_{m_1}, b_{m_z})) \end{aligned}$$

Since P is a most relevant path, $W_H(a_{i_1}, a_{i_v}) = AR_G(a_{i_1}, a_{i_v})$ and similarly for all clique edges. For the inter–OODB edges $W_H = W$.

$$AR_H(Q) = WF(AR_G(a_{i_1}, a_{i_v}), W(a_{i_v}, e_{j_1}), AR_G(e_{j_1}, e_{j_w}), W(e_{j_w}, d_{k_1}), AR_G(d_{k_1}, d_{k_x}), W(d_{k_x}, c_{l_1}), AR_G(c_{l_1}, c_{l_y}), W(c_{l_y}, b_{m_1}), AR_G(b_{m_1}, b_{m_z}))$$
$$= AR_G(P).\square$$

Lemma 5: For every proper path Q from center node a_i representing a_p to center node b_m representing b_q in H there exists a pair of nodes $a_p \in \text{OODB}_A$ and $b_q \in \text{OODB}_B$ and a corresponding path P from a_p to b_q in G such that AR(Q) = AR(P).

The omitted proof is based on the definition of H and works in reverse order to the previous proof.

Theorem 6: A most relevant path Q connecting a pair of center nodes a_i and b_m in H which represent nodes a_p and b_q in different OODBs in G has a corresponding *most relevant* path P in G from a_p to b_q such that AR(P) = AR(Q).

Proof: By Lemma 3 there exists in H a proper path R connecting a_i and b_m which corresponds to the given path Q in H such that AR(R) $\geq$ AR(Q). By Lemma 5, there exists a path P in G from node a_p to node b_q such that AR(P) = AR(R). Now suppose P is not a most relevant path in G. Thus, there exists a most relevant path P$'$ from a_p to b_q in G such that AR(P$'$) > AR(P). By Lemma 4 there exists a proper path Q$'$ from a_i to b_m in H such that AR(Q$'$) = AR(P$'$) > AR(P) = AR(R) $\geq$ AR(Q) a contradiction to the fact that Q is a most relevant path in H. Hence, P is a most relevant path in G. $\square$

The theorem implies that for calculating the access relevance from node a_p in OODB_A to node b_q in OODB_B we take the nodes a_i and b_m in H to represent a_p and b_q, respectively, and calculate the access relevance in H from a_i to b_m. Note that we

use the center node a_i to represent a_p even if a_p is a contact node. In this way, for each pair of classes of different OODBs, e.g., $a_p \in \text{OODB}_A$ and $b_q \in \text{OODB}_B$, we can find the access relevance by applying for H=(U, D) the single source algorithm from Chapter 5 (i.e. *PRODUCT_AR* or *MINIMUM_AR*) of complexity $\min(O(|U|^2, O(|D|\log|D|))$ which uses the precomputed access relevance of the given OODBs. Clearly $|U| << |V|$ holds here.

This computation of access relevance makes no assumptions at all about the order of traversing component OODBs and about the number of times each component OODB is traversed. Intuitively, we have replaced the graph G by the much smaller graph H and can apply to H exactly the same techniques that we used within a single OODB (Chapter 5). Theorem 5 guarantees that an access relevance computed in H will be identical to the corresponding AR in G.

Specifically, the above calculation takes into account even the possibility of traversing through an OODB more than once. One such case is when there exists a most relevant path in H between two contact nodes of one OODB using at least one node from another OODB. To see this, refer back to Graph H in Figure 6.7. Assume that we are looking for a most relevant path from N_1 to N_5. It is possible that such a path consists of (N_1, N_2, N_3, N_4, N_5). When the access relevance between N_2 and N_4 was computed in OODB_A alone, the "shortcut" through N_3 was not available (see Figure 6.7). However such a path is considered when modeling with H. Another such case is when a most relevant path in H contains two clique edges of the same OODB,

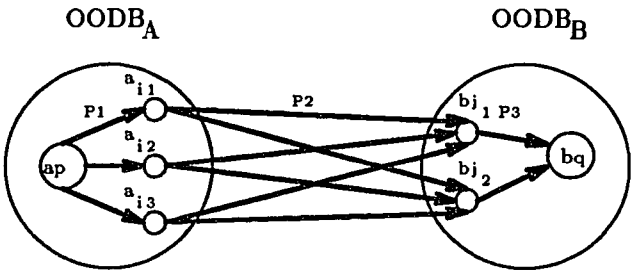

Figure 6.8 An Efficient Computation of Access Relevance

which are not consecutive in the path. Note that in such a case it may happen that the two intra–OODB paths corresponding to two clique edges of the same OODB share a non–contact node. But then the most relevant path contains a cycle which can be removed without decreasing the access relevance, as explained in the proof of property 1 in Chapter 4.

We can further improve the efficiency as follows. We realize that a most relevant path from a_p to b_q starts and ends with a center node in H but contains no other center nodes. Furthermore, the access weights of the star edges in an OODB change with the choice of the source and target classes in the OODBs, but the rest of the access weights of D are independent of this choice. Thus, we define a subgraph I = (U_1, D_1) of H where the star subgraphs are omitted, i.e., each OODB is represented in I only by the clique of its contact classes. Now, we can precompute the access relevance for all pairs of nodes in I by applying the single source algorithm of Chapter 5 $|U_1|$ times, resulting in a complexity of min(O($|U_1|^3$, O($|U_1||D_1|\log|D_1|$)). The result is stored in an access relevance matrix AR_I for I. Now a most relevant path between arbitrary classes $a_p \in OODB_A$ and $b_q \in OODB_B$ is represented as a concatenation of three paths $P_1(a_p, a_i)$, $P_2(a_i, b_m)$, and $P_3(b_m, b_q)$ where a_i and b_m are contact classes

of OODB_A and OODB_B, respectively. The path $P_1(P_3)$ is a most relevant path of OODB_A (OODB_B) and P_2 is a most relevant path of I (Figure 6.8). Thus,

$$AR(a_p, b_q) = \max_{(contacts\ a_i \in OODB_A, b_m \in OODB_B)} WF(AR(a_p, a_i), AR_I(a_i, b_m), AR(b_m, b_q))$$

Now all these access relevances are precomputed and $\text{AR}(a_p, b_q)$ is found with complexity $\text{O}(c_A c_B)$, where c_A and c_B are the numbers of contact classes of OODB_A and OODB_B, respectively. As conjectured previously, these numbers are typically constants or sublinear functions of the number of classes in the OODBs. Thus, we achieve a very fast online algorithm for computing access relevance between classes of different OODBs.

CHAPTER 7

A UNIVERSITY ENVIRONMENT OODB

A university environment object–oriented database schema was developed at NJIT during the last several years under my guidance and supervision. This was a multi–phase project which involved 9 masters students doing their theses and projects [CT90, WA90, K90c, B90, D91b, P91a, P91b]. The major purpose of this development was to gain experience with the complex problems involved in real–world modeling especially modeling with our Dual Model and to serve as a realistic testbed for path–method generation. This database schema contains information about all the aspects of a university.

The development of the database was divided into two phases. In the first part the academic organizational aspects were modeled, including classes related to students, professors, courses, employees, schools, colleges, departments, committees, resumes, etc. The second part dealt with student oriented information including classes related to students, courses, admissions, registrations, and financial aid. Of course there are some common classes. The third part containing the administrative aspects of a university environment is in planning. The first phase of this database which contains around 180 classes was implemented twice using VODAK/VML prototypes 1 & 2 based on Smalltalk and C++ respectively, including necessary interfaces. In this way the project was also the first major application of VML in a site outside of GMD.

7.1 Classes of a Subschema of the University Database

Throughout the dissertation we have used a large subschema of this university database containing 52 classes. We have described 50 sample path–methods used for path–method generation. In this chapter we will show the code of the class definitions of these 52 classes and show a graphical schema representation of the parts of the university database. For ease of discussion we divided these classes into several groups. The schema with these groups of classes shown as overlaid boxes can be found in Figure 7.1. Each group is labeled with the subsection in which classes of that group are discussed. These class definitions are in our general OODB model, and not in the Dual Model, as in our actual university OODB.

7.1.1 Student–related Classes

We will start with student–related classes. First we define the class **person**, which has four attributes. In the following class definitions, we will use different datatypes to make class definitions more readable. For example PerDataType, AddressType, etc. These datatypes are complex datatypes such as tuple, nested tuple, etc. We list the datatypes used at the end of this chapter.

```
class person
    attributes:
        PerData : PerDataType;
        Address : AddressType;
        Telephones : TelephonesType;
        VisaStatus : String;
```

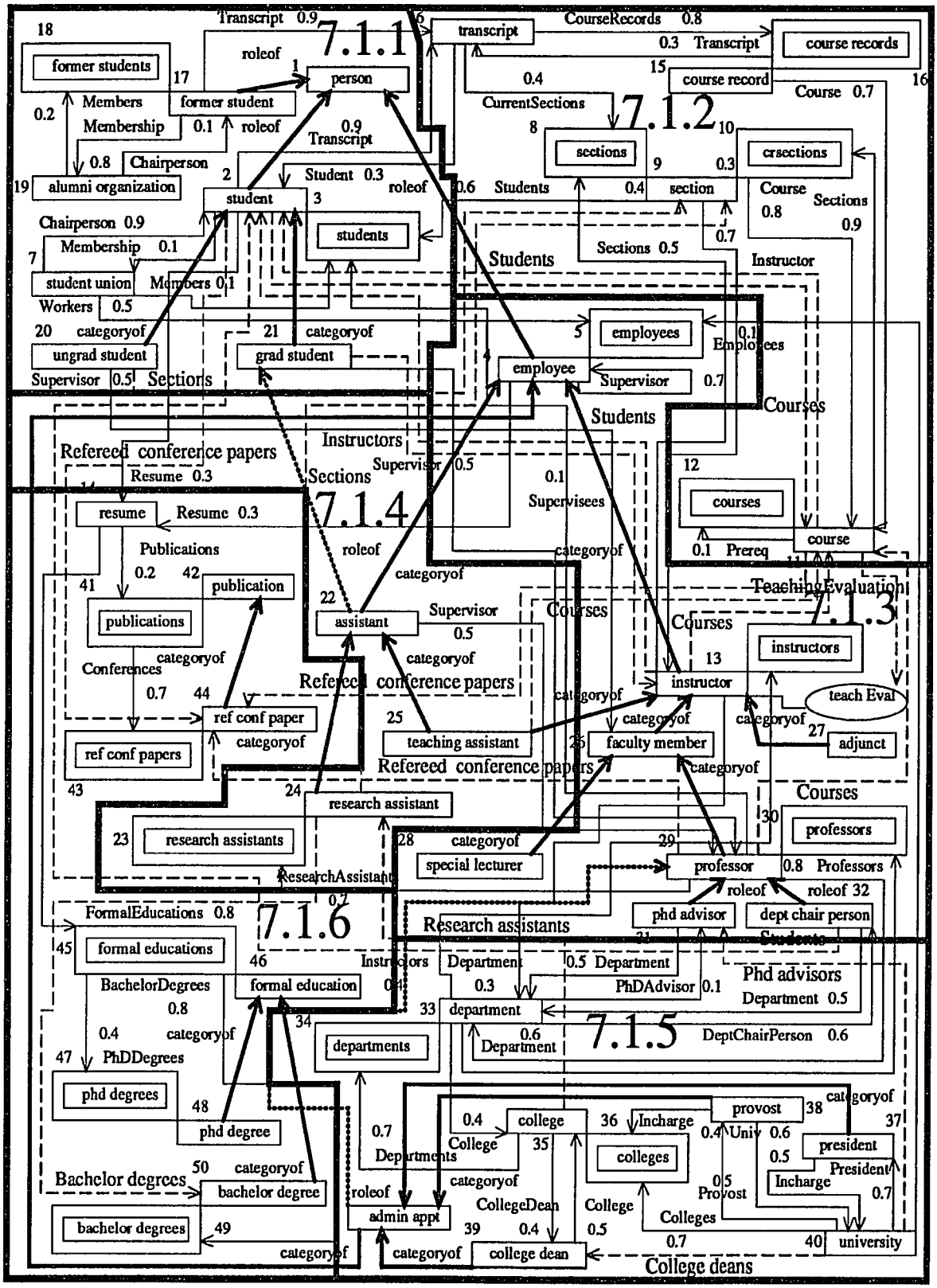

Figure 7.1 The Larger Subschema of a University Database

class student
    roleof: person
    memberof: students
    attributes:
        StudentId : Integer;
        LocalAddress : AddressType;
        GradYear : YearType;
        LastEducation : String;
    relationships:
        Transcript : transcript;
        Membership : student_union;
        Resume : resume;

class former_student
    rolof: person
    memberof: former_students;
    attributes:
        StudentId : Integer;
        Address : AddressType;
        GradYear : YearType;
        Degree : String;
    relationships:
        Transcript : transcript;
        Membership : alumni_organization;

class students
    setof: student
    attributes:
        NumStudents : Integer;
        Purpose : String;

class former_students
    setof: former_student

attributes:
NumFormerStudents : Integer;
Purpose : String;

class student_union
attributes:
NumMembers : Integer;
Address : CompanyAddressType;
relationships:
ChairPerson : student;
Members : students;
Workers : employees;

class alumni_organization
attributes:
NumMembers : Integer;
Address : CompanyAddressType;
relationships:
ChairPerson : former_student;
Members : former_students;

class grad_student
categoryof: student
attributes:
StartDate : DateType;
Degree : DegreeType;
PreviousSchool : String;
Project : String;
relationships:
Supervisor : professor;

class ungrad_student
categoryof: student
attributes:

Year : YearType;
Major : MajMinType;
Minor : MajMinType;
Project : String;
relationships:
Supervisor : faculty_member;

We define two similar classes, **student**, which represents students currently registered in the university, and **former_student**, which represent students graduated from the university. Both the classes have relationship to the class **transcript**. On the other hand the class **student** has relationship to the class **student_union**, the class **former_student** has relationship to the class **alumni_organization**. This is because members of a student_union are only students, and members of an alumni_organization are only former_students. In [NPGT91] we show that these two classes are structurally similar (except for the relationship *Resume*) in terms of the Dual Model. Two classes **students** and **former_students** are set classes for the classes **student** and **former_student**, respectively. We define two classes **grad_student** and **ungrad_student** as *categoryof* of the class **student**. Note that the class **grad_student** has a relationship *Supervisor* to the class **professor** because only a professor can be an advisor for a graduate student. On the other hand the class **ungrad_student** has a relationship *Supervisor* to the class **faculty_member**, as any faculty member can be a supervisor for an undergraduate student.

7.1.2 Course–related Classes

Now we define classes related to courses and sections. The class transcript has a relationship to the class **course_records**, which contains information about all the courses which are already completed by a student. For example, grade point average of a student. It has a relationship to the class **sections** where information about current sections of a student is found. For example, registered credit hours of a student for the current semester are stored there. The class **section** is *memberof* the class **sections** and describes a singular section. Two classes, **sections** and **crsections**, have the generic relationship *setof* to the class **section**. There is a basic difference between them. The class **crsections** describes the set of all sections offered for a given course. On the other hand the class **sections** describes a set of sections, not necessarily of the same course. We need such a set to describe the set of sections a student is registering for or that an instructor is teaching. It is necessary to define the class **courses** to describe a set of courses e.g., the set of courses a student already took. Notice that class **course** has two connections to class **courses**, the generic relationship *memberof* and a relationship *Prereq*, giving prerequisite courses required for a given course.

class transcript
    attributes:
        Date : DateType;
    relationships:
        Student : student;
        CurrentSections : sections;
        CourseRecords : course_records;

class course_record
    memberof: course_records
    attributes:
        SemesterTaken : String;
        Grade : Real;
    relationships:
        Course : course

class course_records
    setof: course_record
    attributes:
        Purpose : String;
        NumCourseRecords: Integer;
    relationships:
        Transcript : transcript;

class section
    memberof: crsections
    memberof: sections
    attributes:
        SectionNo : Integer;
        NumStudents : Integer;
        Room : RoomType;
    relationships:
        Instructor : instructor;
        Students : students;

class sections
    setof: section
    attributes:
        NumSections : Integer;
        GroupPurpose : String;

class crsections

setof: section
attributes:
NumSections : Integer;
relationships
Course : course;

class course
memberof: courses
attributes:
Name : String;
DepartmentName : String;
CreditHour : Integer;
Number : Natural;
relationships:
Prereq : courses;
Sections : crsections;

class courses
setof: course
attributes:
NumCourses : Integer;
Purpose : String;

7.1.3 Instructor–related Classes

Now we will discuss classes related to instructors. The class **employee** is a *roleof* person. The corresponding set class is **employees**. We define the class **instructor** as *categoryof* employee. It has relationship to the class **sections**, which can be used to find information about all the sections s/he teaches. Then we define three classes

faculty_member, **adjunct**, and **special_lecturer** as *categoryof* the class **instructor**. We also define the class **professor** as *categoryof* of the class **faculty_member**. The class **professor** has a relationships to classes **students**, **research_assistants**, and **department**. The relationship *Sections* to the class **sections**, inherited from the class **instructor** can be useful to find all the sections currently being taught by him. Finally, we define three classes **phd_advisor**, **dept_chair_person**, and **admin_appt** (for which code is listed later) as *roleof* of the class **professor**. A dept_chair_person and a phd_advisor are in charge of a department and its phd_program, respectively.

```
class employee
    roleof: person
    memberof: employees
    attributes:
        SocialSecurityNr: Integer;
        EmployeeId: Integer;
        OfficeAddress: CompanyAddressType;
        OfficePhone: TelephonesType;
        Position : String;
        SalaryRate : Real;
    relationships:
        Supervisor : employee;
        Resume : resume;

class employees
    setof: employee
    attributes:
        Purpose: String;
        NumOfEmployees : Integer;

class instructor
```

categoryof: employee;
memberof: instructors;
attributes:
TeachingEvaluation : Real;
relationships:
Sections : sections;
Department : department;

class instructors
setof: instructor
attributes:
Purpose : String;
NumOfInstructors : Integer;

class faculty_member
categoryof: instructor
attributes:
InsPolNum : String;
OfficeHour : String;

class adjunct
categoryof: instructor
attributes:
CompanyAddress : CompanyAddressType;
CompanyPhone : TelephonesType;
CompanyName : String;

class special_lecturer
categoryof: instructor
attributes:
StudiesStatus : String;
ContractPeriod : YearType;

class professor

categoryof: faculty_member
memberof: professors
attributes:
Rank : RankType;
TenureStatus : String;
SpecialArea : String;
relationships:
Supervisees : students;
ResearchAssistants: research_assistants;
Department : department;

class professors
setof: professor
attributes:
Purpose : String;
NumOfProfessors: Integer;

class phd_advisor
roleof: professor
attributes:
NumPhDStudents: Integer;
ReleaseTime: PercentType;
relationships:
Department : department;

class dept_chair_person
roleof: professor
attributes:
YearsAsgndChair : Integer;
relationships:
Incharge : department;

7.1.4 Assistant–related Classes

In the university environment a graduate student can also be an assistant. The class **assistant** is *roleof* the class **grad_student** since it is specialized in the context of employment rather than the studying context. The classes **research_assistant** and **teaching_assistant** are *categoryof* the class *assistant*, since it is specialized in the same context. We do not want the class **assistant** to inherit all the properties of **student** and **grad_student**, since they are not relevant to the function of an individual as an assistant, in spite of the condition that every assistant is a graduate student. We just want to inherit the transcript relationship of student since some information stored there is needed for determining the eligibility for an assistantship. Note that the supervisors for a graduate student that is an assistant can be two different professors or can be two different professors or can be the same professor.

class assistant
  roleof: grad_student
  categoryof: employee
  attributes:
    Stipend : Real;
  relationships:
    Supervisor : professor;

class research_assistant
  categoryof: assistant;
  memberof: research_assistants;
  attributes:
    PositionPercent : Real;
    Research : String;

class research_assistants
    setof: research_assistant
    attributes:
        Purpose : String;
        NumResearchAssistants : Integer

class teaching_assistant
    categoryof: assistant
    categoryof: instructor
    attributes:
        NumOfCourses : Integer;
        PositionPercent : Real;

7.1.5 University–related Classes

Now we will define classes related to university administration. The class **university** has relationships to the classes **president**, **provost**, and **colleges**. A president is in charge of the university, and a provost is in charge of all the colleges. The class **college** has a relationship to the class **college_dean**. A college dean is in charge of a college. The class **college** has a relationship to the class **departments**, because a college will include several academic departments. We have defined a class **admin_appt**, which represents administrative appointments, as *categoryof* of the class **employee**. As president, provost, and college dean are administrators in a university we define these classes as *categoryof* of the class **admin_appt**. As explained in Chapter 3, and mentioned in Section 7.1.3 the class **admin_appt** is *roleof* of the class **professor**.

class university
    attributes:
        UniversityName : String;
        Office : CompanyAddressType;
        Telephones : TelephonesType;
    relationships:
        President : president;
        Colleges : colleges;
        Provost : provost;
        Employees : employees;

class admin_appt
    categoryof employee
    roleof professor
    attributes:
        Responsibilities: String;
        Degree: DegreeType;
        Office: CompanyAddressType;

class president
    categoryof: admin_appt
    attributes:
        YearsInCharge : Integer;
    relationships:
        InCharge : university;

class provost
    categoryof: admin_appt
    attributes:
        YearsInCharge : Integer;
    relationships:
        University : university;
        InCharge : colleges;

class college_dean
    categoryof: admin_appt
    attributes:
        AdditionalResp : String;
    relationships:
        College : college;

class college
    memberof: colleges
    attributes:
        Telephones : TelephonesType;
        Office : CompanyAddressType;
    relationships:
        CollegeDean : college_dean;
        Departments : departments;

class colleges
    setof: college
    attributes:
        NumOfColleges : Integer;
        Purpose : String;

class department
    memberof: departments
    attributes:
        DeptName : String;
    relationships:
        DeptChairPerson : dept_chair_person;
        PhDAdvisor : phd_advisor;
        Instructors : instructors;
        Professors : professors;

class departments
    setof: department

attributes:
NumOfDepartments : Integer;
Purpose : String;

7.1.6 Resume–related Classes

Now we define classes related to resume. We describe here only a sample of few elements regarding the resume while a full description appears in the university OODB. We define class **resume**, which refers to two classes, **publications**, and **formal_educations**. The class **publications** refers to the class **ref_conf_papers**. We define the class **ref_conf_paper** as *categoryof* of the class **publication**. The class **formal_educations** refers to two classes, **bachelor_degrees** and **phd_degrees**. We define classes **bachelor_degree** and **phd_degree** as *categoryof* of the class **formal_education**.

class resume
attributes:
JobTitle : String;
relationships:
Publications : publications;
FormalEducations : formal_educations;

class publication
memberof: publications
attributes:
Year : Integer;
Title : String;
Authors : {String};

class publications
    setof: publication
    attributes:
        NumPublications : Integer;
        Purpose : String;
    relationships:
        Conferences : ref_conf_papers;

class ref_conf_paper
    memberof: ref_conf_papers
    attributes:
        PageNum : Integer;
        Location : CityAddrType;
        TypeOfReview : String;
        Volume : Integer;
        Conference : String;

class ref_conf_papers
    setof: ref_conf_paper
    attributes:
        NumRefConfPapers : Integer;
        Purpose : String;

class formal_education
    memberof: formal_educations
    attributes:
        Degree : DegreeType;
        UniversityName : String;
        YearGranted : YearType;
        Area : MajorType;

class formal_educations
    setof: formal_education

attributes:
NumFormalEducations : Integer;
Purpose : String;
relationships:
BachelorDegrees : bachelor_degrees;
PhDDegrees : phd_degrees;

class phd_degree
memberof: phd_degrees
attributes:
DissertationTitle : String;
Purpose : String;

class phd_degrees
setof: phd_degree
attributes:
NumPhDDegrees : Integer;
Purpose : String;

class bachelor_degree
memberof: bachelor_degrees
attributes:
ProjectTitle : String;

class bachelor_degrees
setof: bachelor_degree
attributes:
NumBachelorDegree : Integer;
Purpose : String;

Finally, we will define all the datatypes used in the above class definitions.

DATATYPE StreetAddressType = [number: Integer, street: String, unit: String];

```
DATATYPE CityAddressType = [city: String, state: String, zip: String];
DATATYPE AddressType = [streetaddress: StreetAddressType,
                        cityaddress: CityAddressType];
DATATYPE NameType = [first: String, middle: String,
                     last: String, extra: String];
DATATYPE SexType = (Male, Female);
DATATYPE StatusType = (Single, Married, Divorced, Widow);
DATATYPE DayType = SUBRANGE 1..31;
DATATYPE MonthType = (Jan, Feb, Mar, Apr, May, Jun, Jul,
                      Aug, Sep, Oct, Nov, Dec);
DATATYPE YearType = SUBRANGE 1800..2100;
DATATYPE DateType = [day: DayType, month: MonthType, year: YearType];
DATATYPE PerDataType = [name: NameType, sex: SexType,
                        maritalstatus: StatusType, birthday: DateType];
DATATYPE TelephoneType = [area: Integer, number: Integer];
DATATYPE TelephonesType = TelephoneType;
DATATYPE DepartmentType = [dept: String, companyname: String];
DATATYPE CompanyAddressType = [department: DepartmentType,
                               streetaddress: StreetAddressType,
                               cityaddress: CityAddressType];
DATATYPE DegreeType = (BS, MS, PhD, NonMat);
DATATYPE MajMinType = String;
DATATYPE RoomType = [buildname: String, roomnumber: Integer];
DATATYPE MajorType = String;
DATATYPE RankType = (Assistant, Associate, Full, Distinguish, Visiting);
DATATYPE PercentType = A Real Number between 1 ... 100;
```

CHAPTER 8
DESIGN OF AN OODB PATH–METHOD GENERATOR

In this chapter we will discuss the design of the Path–Method Generator module for an OODB. The database subsystems including the Path-Method Generator are shown in Figure 8.1. We will define all the necessary classes using the general OODB model discussed in this dissertation. Later on, we will explain how the PMG works and how the PMG module can be incorporated in an OODBMS.

Initially, the user request is accepted by the query translator. If the user request is not written in the host query language, then the query translator fails to process it. This unprocessed user request is forwarded to the Path-Method Generator. The Path-Method Generator contains two major components: (1) Path-Method Editor, (2) Path-Method Navigator. The Path-Method Navigator contains a collection of algorithms for Path–Method Generation. The PMG has two modes (1) navigation mode, and (2) editing mode. The PMG generates a path–method from the source class to the target information, if possible. As discussed earlier, navigation of PMG is done by traversal algorithms. These traversal algorithms uses access weights and access relevance for traversal of an OODB schema. The generated path–method will be returned in the path–method editor to the user for verification. The user can either accept or modify this resultant path–method as per his requirements. The user can also set more parameters and request a second traversal. After verification,

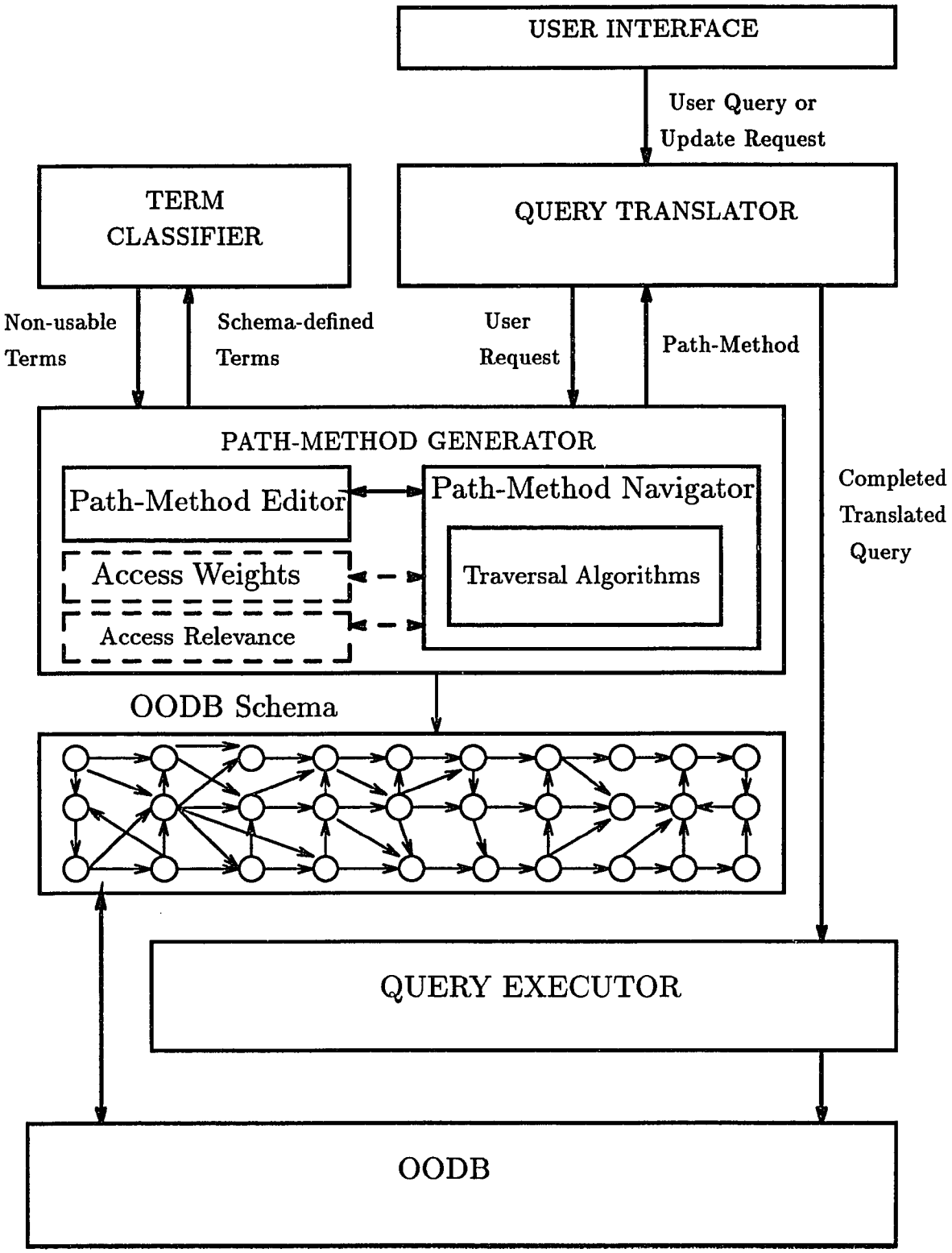

Figure 8.1 The OODB Subsystems Including a Path-Method Generator

the generated path–method is sent to the Query Executor. The Query Executor now applies this completed, translated query to the OODB to retrieve the target information.

A user generally has very little knowledge about the classes defined in the OODB schema. S/he will use his naive view of the database to formulate his/her query and may use terms which are not defined in the schema. The Term Classifier, which is based on the Knowledge Explorer [K91], finds schema–defined terms from the terms defined by the users.

In the next several sections we will define internal classes for the Path–Method Generator module. In the class definitions we have tried to pick self–explanatory names for attributes, relationships, methods. The formal parameter names of methods and datatype names of attributes are also self–explanatory. In addition, we will explain each method defined for a class. We have used several data types in the class definitions. Before discussing class definitions we will show definitions of these data types.

DATATYPE PairType = [property: String, result: String];

DATATYPE AllPairType = ARRAY [1 .. NoNodes] OF PairType;

DATATYPE WeightMatrixType = ARRAY[1..NoNodes, 1..NoNodes] OF Real;

DATATYPE RelevanceMatrixType = ARRAY[1..NoNodes, 1..NoNodes] OF Real;

DATATYPE PathMethodType = [pathlength: Integer, pmpairs: AllPairType];

DATATYPE UserRequestType = [source: String, targetinformation: String];

DATATYPE AttributeType = [property: String, result: String];

DATATYPE StringPairType = [parametername : String, parametertype: String];

DATATYPE ConnectionType = [property: String, result: String, weight: Real];

DATATYPE NodeType = [node_num : Integer, class_name : String,

attributes:{AttributeType}, connections: {ConnectionType}];

Three datatypes StackType, QueueType, and HeapType can be implemented as abstract datatypes based on discussions in algorithms and data structures text, e.g., [AHU83].

8.1 Interface Classes of the Path–Method Generator

We start by defining the class **path_method_generator**. There are three relationships from class **path_method_generator** to classes, **path_method_editor**, **path_method_navigator**, and **query translator**. The relationship to the class **query_translator** is used to return the generated path–method.

class path_method_generator

attributes:

NumEditors : Integer;

NumTravAlgorithms: Integer;

relationships:

PathMethodEditor: path_method_editor;

PathMethodNavigator: path_method_navigator;

QueryTranslator: query_translator;

methods:

InitializePMG ();

InvokePathMethodEditor ();
InvokePathMethodNavigator ();
GeneratePathMethod (user_request: UserRequestType);
ReturnPathMethod (path_method: PathMethodType);

Now we will explain each method of the class **path_method_generator.**

1. *InitializePMG ()* ... This method initializes the PMG. It checks all the necessary components such as Path–Method Editor and Path–Method Navigator.

2. *InvokePathMethodEditor ()* ... This method invokes a Path–Method Editor, which is available with the Path–Method Generator.

3. *InvokePathMethodNavigator ()* ... This method invokes a Path–Method Navigator, which is available with the Path–Method Generator.

4. *GeneratePathMethod (user_request: UserRequestType)* ... This is the main method of the class **path_method_generator** which calls other methods for path–method generation.

5. *ReturnPathMethod (path_method: PathMethodType)* ... This method returns the generated path–method to the Query Translator using the relationship *QueryTranslator.*

When the path–method navigator generates a path–method, it will be displayed in the path–method editor, where it can be changed by a user. It also allows a user

to formulate queries if s/he wants, without using the path–method navigator. The class **path_method_editor** is defined below. The attributes *NumVisitedNode* and *PathLength* contain the number of visited nodes and the length of the generated path–method, respectively. The relationship *OmlCompiler* is used to call **oml_compiler** to include a generated path–method as a method of a source class. Here the Oml–Compiler is a general Object–Manipulation Language compiler that must be supported by the OODBMS.

```
class path_method_editor
    attributes:
        SourceClass: String;
        TargetInformation: String;
        NumVisitedNodes: Integer;
        PathLength: Integer;
    relationships:
        PathMethodGenerator: path_method_generator;
        PathMethodNavigator: path_method_navigator;
        OmlCompiler: oml_compiler;
        TermClassifier: term_classifier
    methods:
        DisplayOneMethod (path_meth: PathMethodType);
        DisplayErrorMessage (message: MessageType);
        ReadUserRequest (source: String; target: String);
        GeneratePathMethod (source: String; target: String):
                PathMethodType;
        CheckUserRequest (source: String; target: String):
                UserRequestType;
        CheckClassName (classname: String) : Boolean;
        AcceptPathMethod (pathmethod: PathMethodType);
        UpdatePathMethod (pathmethod: PathMethodType):
                PathMethodType;
```

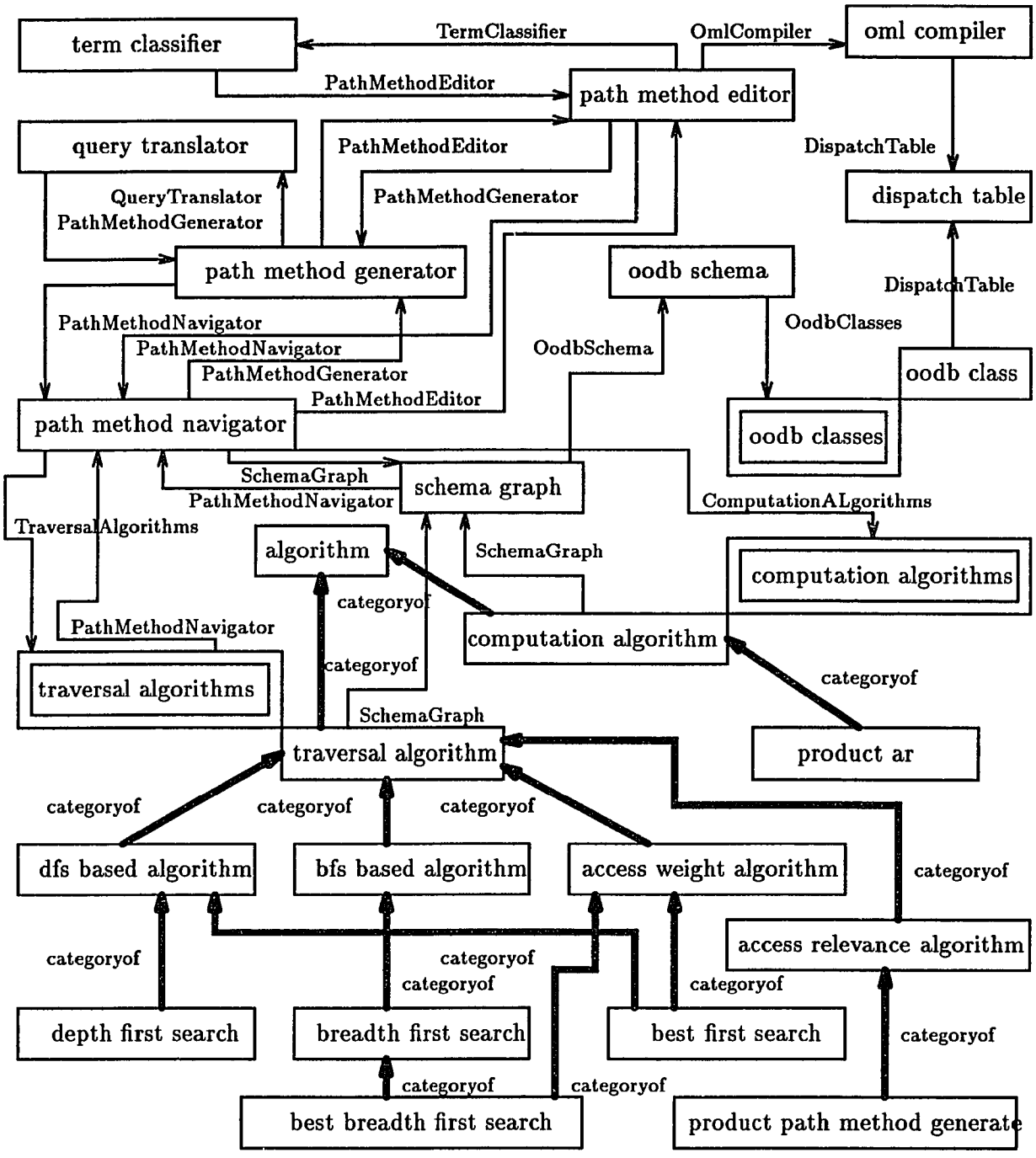

Figure 8.2 Classes for Path–Method Generator

Now we will explain the methods for the class **path_method_editor.**

1. *DisplayOneMethod (path_meth: PathMethodType)* ... Displays one path–method on the path–method editor.

2. *DisplayErrorMessages(message: MessageType)* ... Displays an error message or a warning.

3. *ReadUserRequest(source: String, target: String)* ... This method reads a user request in two strings, a source and a target, either from the Query Translator or from a user.

4. *GeneratePathMethod (source: String, target: String): PathMethodType* ... This method accepts two strings, a source and a target and returns a path–method. It calls Path–Method Navigator to perform traversal.

5. *CheckUserRequest (source: String, target: String): UserRequestType* ... This method accepts two strings, a source and a target and then calls the Term–Classifier to check whether these terms are defined in the schema or not. The Term–Classifier returns two schema–defined terms.

6. *CheckClassName(classname: String): Bool* ... This method checks whether a class is defined in the schema or not. This is done by using the Term–Classifier.

7. *AcceptPathMethod (pathmethod: PathMethodType)* ... This method sends the generated path–method to the dispatch table of the source class, using the

relationship to the class **oml_compiler**. The dispatch table contains all the methods defined for a class.

8. *UpdatePathMethod(pathmethod: PathMethodType): PathMethodType* ... This method allows a user to update the generated path–method if s/he wants to. Then the modified path–method will be the resultant method.

The path–method navigator performs traversal of the schema. It refers to a collection of traversal algorithms. Computation of access relevance is done by computational algorithms. The definition for the class **path_method_navigator** is given below. The relationship *SchemaGraph* is used to access any information about the OODB schema.

```
class path_method_navigator
    attributes:
        Source: String;
        Target: String;
        NumAlgorithms: Integer;
        NumVisitedNodes: Integer;
        PathLength: Integer;
    relationships:
        PathMethodGenerator: path_method_generator;
        PathMethodEditor: path_method_editor;
        SchemaGraph: schema_graph;
        ComputationalAlgorithms: computational_algorithms;
        TraversalAlgorithms: traversal_algorithms;
    methods:
        SelectTraversalAlgorithm (): Integer;
        CallTraversalAlgorithm (choice: Integer);
        ReportUnsuccessfulSearch (message: MessageType);
```

```
GeneratePathMethod (source: String; target: String):
        PathMethodType;
RunPathMethodNavigator ();
ComputeAccessRelevance (relevance_mat: RelavanceMatrixType);
```

Now we will explain all the methods defined for the class **path_method_navigator.** A description of each method is given below.

1. *SelectTraversalAlgorithm (): Integer* ... Display various traversal algorithms to user so s/he can choose one of it.

2. *CallTraversalAlgorithm (choice: Integer)* ... Run the selected traversal algorithm.

3. *ReportUnsuccessfulSearch (message: MessageType)* ... Return an error message or a warning to the Path–Method Editor.

4. *GeneratePathMethod (source: String; target: String): PathMethodType* ... This method accepts two strings, a source and a target, and returns a path–method. It calls Path–Method Navigator to perform traversal.

5. *RunPathMethodNavigator ()* ... This method calls different methods based on the behavior of the path–method generation.

6. *ComputeAccessRelevance (relevance_mat: RelavanceMatrixType)* ... This method calls a computational method to compute access relevance.

The class **schema_graph** is a directed graph representation of an OODB schema. The Path–Method Generator requires that an OODB schema should be converted into a schema graph. Once the schema–graph is initialized, traversal algorithms can access necessary information from it to traverse it. The relationship *Oodb_schema* is used to create a Schema–Graph from an OODB schema.

The attribute *AccessWeightMatrix* is a matrix, where each pair (=(row, column)) contains the corresponding access weight of the edge in the schema graph from the row class to the column class. The attribute *AccessRelevanceMatrix* is a matrix computed by a computation algorithm, where each pair (=(row, column)) contains corresponding access relevance of the most relevant path in the schema graph from row class to the column class. The attribute *AdjacencyList* is the adjacency list representation of the schema graph, which is used for efficient computation of all–pair access relevance in the schema graph.

The definition of the class **schema_graph** is shown below.

```
class schema_graph
    attributes:
        Nodes: {NodeType};
        Connections: {ConnectionType};
        NoNodes: Integer;
        AccessWeightMatrix: WeightMatrixType;
        AccessRelevanceMatrix: RelevanceMatrixType;
        AdjacencyList: AdjacencyListType;
    relationships:
        PathMethodNavigator: path_method_navigator;
        OodbSchema: oodb_schema;
    methods:
```

```
CreateSchemaGraph ();
NodeNumber (classname: String) : Integer;
AccessWeight (class_one: String; class_two: String) : Real;
ConnectionAccessWeight (connection: ConnectionType) : Real;
AccessRelevance (class_one: String; class_two: String) : Real;
Neighbors (classname: String) : {ConnectionType};
NeighborClassNames (classname: String) : {String};
PermissibleNeighbors (classname: String; targetinfo: String;
                              visited: {String}): {ConnectionType};
PermissibleNeighborClassNames (classname: String;
                    targetinfo: String; visited: {String}): {String};
Attributes (classname: String) : {AttributeType};
```

Now we will explain each method of the class **schema_graph**.

1. *CreateSchemaGraph ()* ... creates a Schema–Graph from an OODB schema. If the OODB schema does not have any information regarding access weights, then access weights should be added to the OODB schema.

2. *NodeNumber (classname: String): Integer* ... returns a node number in a Schema–Graph of a class.

3. *AccessWeight (class_one : String, class_two: String): Real* ... returns access weight between two classes if there exists a direct connection between the two classes. Otherwise it returns zero(0).

4. *ConnectionAccessWeight (connection: ConnectionType): Real* ... returns access weight of a connection, which is available in the AccessWeightMatrix.

5. *AccessRelevance (class_one : String, class_two: String): Real* ... returns access relevance between two given classes, which is available in the AccessRelevance-Matrix.

6. *Neighbors (classname: String):* {*String*} ... returns a set of connections of a class in the Schema Graph.

7. *NeighborClassNames (classname: String):* {*ConnectionType*} ... returns a set containing class names of all the adjacent nodes of a class in the Schema Graph.

8. *PermissibleNeighbors (classname: String; targetinfo: String; visited:* {*String*}*):* { *ConnectionType* } ... finds a set of connections of a class in the Schema Graph. Then, it returns only the ones which do not lead to classes which appear in the set *visited.*

9. *PermissibleNeighborClassNames (classname: String; targetinfo: String; visited:* {*String*}*):* {*String*} ... finds a set containing class names of all the adjacent nodes of a class in the Schema Graph. Then it returns only the ones which do not which appear in the set *visited.*

10. *Attributes (classname: String) :* {*AttributeType*} ... returns a set of attributes of a given class.

8.2 Traversal Algorithms of Path–Method Generator

In this section we will discuss different classes that are defined for traversal algorithms. All the classes for PMG are shown in Figure 8.2. We assume that classes **oodb_schema**, **query_translator**, **term_classifier**, **oodb_class**, **oodb_classes**, and **oml_compiler** are already defined for an OODB. Although they are shown in the Figure 8.2, we will not discuss their definitions here as they are not classes of the Path–Method Generator module. We have discussed the algorithm *PathMethodGenerate* in Chapter 4. If one wants to implement other methods for example, *BestBreadth-FirstSearch*, then *PathMethodGenerate* can be modified rather than writing such a method from scratch.

We first define the class **algorithm**. Then we define class **traversal_algorithm** as *categoryof* of the class **algorithm**. Then we define classes **dfs_based_algorithm**, **bfs_based_algorithm**, **access_weight_algorithm**, **access_relevance_algorithm** as *categoryof* of the class **traversal_algorithm**. The class **dfs_based_algorithm** represents depth–based algorithms, while the class **bfs_based_algorithm** represents breadth–based algorithms. The class **access_weight_algorithm** represents algorithms which use access weights and the class **access_relevance_algorithm** represents algorithms which use access relevance for path–method generation.

We define class **depth_first_search** as *categoryof* the class **dfs_based_algorithm**, and class **breadth_first_search** as *categoryof* of the class **bfs_based_algorithm**. We also define class **best_first_search** as *categoryof* of the classes **depth_first_search**

and **access_weight_algorithm.** The class **best_breadth_first_search** as *categoryof* of two classes **breadth_first_search** and **access_weight_algorithm.**

Finally, we define the class **product_path_method_generate** as *categoryof* of the class **access_relevance_algorithm.** Note that in the development of PMG we model different algorithms as classes. This is a novel approach. So far we have found only one approach [S92] which discusses modeling of methods in their knowledge–based system.

class algorithm
attributes:
AlgorithmName: String;
Parameters: {StringPairType};
Purpose: String;
methods:
DisplayAlgorithmName ();
DisplayAlgorithmCode ();
DisplayAlgorithmSignature ();

A description of each method is given below.

1. *DisplayAlgorithmName ()* ... displays the name of the algorithm.

2. *DisplayAlgorithmCode ()* ... displays code of the algorithm.

3. *DisplayAlgorithmSignature ()* ... displays signature (header) of the method, i.e., name of the method, all the parameters enclosed in (), followed by a return type.

class traversal_algorithm
categoryof: algorithm
memberof: traversal_algorithms

attributes:

SourceNode: String;
TargetNode: String;
PathLength: Integer;
NumVisitedNodes: Integer;
Successful: Boolean;

relationships:

SchemaGraph: schema_graph;

methods:

CheckAttributes (source_class: String; target_class: String): Boolean;
GetAttributes (classname: String): {StringPairType};
GetRelationships (classname: String): {StringPairType};
GetUserDefinedRelationships (classname: String):
{StringPairType};
GetGenericRelationships (classname: String): {StringPairType};
GetAttributeSelectors (classname: String) : {String};
GetRelationshipSelectors (classname: String): {String};
GetUserDefinedRelationshipSelectors (classname: String): {String};
GetUserDefinedRelationshipResults (classname: String): {String};
GetGenericRelationshipSelectors (classname: String): {String};
AddToPathMethod (pair: {StringPairType};
pathmethod: PathMethodType): PathMethodType;

A description of each method is given below.

1. *CheckAttributes (source_class: String, target_class: String): Boolean* ... This method checks all the attributes of the source class against the target information.

2. *GetAttributes (classname: String) : {StringPairType}* ... This method returns all the attributes of a given class.

3. *GetRelationships (classname: String):* {*StringPairType*} ... This method returns all the relationships of a given class.

4. *GetUserDefinedRelationships (classname: String):* {*StringPairType*} ... This method returns all the user–defined relationships of a given class.

5. *GetGenericRelationships (classname: String):* {*StringPairType*} ... This method returns all the generic relationships between classes.

6. *GetAttributeSelectors (classname: String):* {*String*} ... This method returns selectors of all the attributes of a class.

7. *GetRelationshipSelectors (classname: String):* {*String*} ... This method returns selectors of all the relationships of a class.

8. *GetUserDefinedRelationshipSelectors (classname: String):* {*String*} ... This method returns selectors of all the user–defined relationships of a class.

9. *GetUserDefinedRelationshipResults (classname: String):* {*String*} ... This method returns results of all the user–defined relationships of a class.

10. *GetGenericRelationshipSelectors (classname: String):* {*String*} ... This method returns selectors of all the generic relationships of a class.

11. *AddToPathMethod(pair:* {*StringPairType*}, *pathmethod: PathMethodType): PathMethodType* ... This method adds a property–type pair to the given path–method and returns the path–method containing this new pair.

class traversal_algorithms
setof: traversal_algorithm
attributes:
NumAgorithms: String;
relationships:
PathMethodNavigator: path_method_navigator;
methods:
DisplayAlgortithmSignatures ();
DisplayAlgorithmChoices ();

A description of each method is given below.

1. *DisplayAlgorithmSignatures ()* ... displays headers of all the traversal algorithms.

2. *DisplayAlgorithmChoices ()* ... displays names and characteristics of each algorithm to the user. Then the user can select one based on his/her needs. These characteristics are parameters, breadth restriction, depth restriction, etc,.

class dfs_based_algorithm
categoryof: traversal_algorithm
attributes:
GroupName: String;
PredeterminedDepth: Integer;
DfsStack: StackType;
methods:
DisplayCharacteristics ();
DisplayStack ();

The attribute *GroupName* is a group name. For example: "Algorithms which are variations of depth first search". The attribute *PredeterminedDepth* is used when

a user wants to specify a depth restriction for traversal. Note that depth–based algorithms use a stack.

A description of each method is given below.

1. *DisplayCharacteristics ()* ... This method displays the characteristics of a depth based algorithm.

2. *DisplayStack ()* ... This method displays the current content of the stack.

class bfs_based_algorithm
    categoryof: traversal_algorithm
    attributes:
        GroupName: String;
        PredeterminedBreadth: Integer;
        BfsQueue: QueueType;
    methods:
        DisplayCharacteristic ();
        DisplayQueue ();

The attribute *PredeterminedBreadth* is used when a user wants to specify a breadth restriction for traversal. Note that breadth–based algorithms use a queue.

A description of each method is given below.

1. *DisplayCharacteristics ()* ... This method displays characteristics of a breadth based algorithm.

2. *DisplayQueue ()* ... This method displays the current content of the queue.

class access_weight_algorithm
    categoryof: traversal_algorithm

The attribute *WeightingFunctionName* could be either PRODUCT or MINIMUM.

A description of methods are given below.

1. *DisplayRelevanceMatrix ()* ... This method displays the access relevance matrix.

2. *GetAccessRelevance (row: Integer, column: Integer): Real* ... This method reads an access relevance from the access relevance matrix of the schema graph.

3. *SetAccessRelevance (row: Integer, column: Integer)* ... This method sets an access relevance.

class depth_first_search
categoryof: dfs_based_algorithm
attributes:
Characteristic: String;
methods:
DepthFirstSearch(source: String; target: String): PathMethodType;
FindNextNeighborForDfs(classname: String) : String;
FindUnvisitedNeighborForDfs(classname: String,
visitedNodes: {String}): String;

A description of each method is given below.

1. *DepthFirstSearch(source: String; target: String): PathMethodType* ... This method accepts two strings, a source and a target, and generates a path–method using the depth first search algorithm.

2. *FindNextNeighborForDfs(classname: String) : String* ... This method finds the next neighbor for a current node for the depth first search algorithm.

3. *FindUnvisitedNeighborForDfs(classname: String, visitedNodes: {String}) : String* ... This method finds the next unvisited neighbor for a current node for the depth first search algorithm.

class breadth_first_search
categoryof: bfs_based_algorithm
attributes:
Characteristic: String;
methods:
BreadthFirstSearch(source: String; target: String): PathMethodType;
FindNextNeighborForBfs(classname: String): String;
FindUnvisitedNeighborForBfs(classname: String,
visitedNodes: {String}): String;

A description of each method is given below.

1. *BreadthFirstSearch(source: String; target: String): PathMethodType* ... This method accepts two strings, a source and a target, and generates a path–method using the breadth first search algorithm.

2. *FindNextNeighborForBfs(classname: String) : String* ... This method finds the next neighbor for a current node for the breadth first algorithm.

3. *FindUnvisitedNeighborForBfs(classname: String, visitedNodes: {String}) : String* ... This method finds the next unvisited neighbor for a current node for the breadth first algorithm.

class best_first_search
categoryof: access_weight_algorithm
categoryof: depth_first_search

attributes:
Characteristic: String;
methods:
BestFirstSearch(source: String; target: String): PathMethodType;
FindNextNeighborForBestFirst (classname: String): String;
FindUnvisitedNeighborForBestFirst (classname: String,
visitedNodes: {String}): String;

A description of each method is given below.

1. *BestFirstSearch(source: String; target: String): PathMethodType* ... This method accepts two strings, a source and a target and generates a path–method using best first search algorithm.

2. *FindNextNeighborForBestFirst(classname: String) : String* ... This method finds the next neighbor for a current node for the best first algorithm.

3. *FindUnvisitedNeighborForBestFirst(classname: String, visitedNodes: {String}) : String* ... This method finds the next unvisited neighbor for a current node for the best first algorithm.

class best_breadth_first_search
categoryof: breadth_first_search
categoryof: access_weight_algorithm
attributes:
Characteristic: String;
methods:
BestBreadthFirstSearch(source: String; target: String):
PathMethodType;
FindNextNeighborForBestBreadth (classname: String): String;

FindUnvisitedNeighborForBestBreadth (classname: String,
visitedNodes: {String}): String;

A description of each method is given below.

1. *BestBreadthFirstSearch(source: String; target: String): PathMethodType* ... This method accepts two strings a source and a target and generates a path–method using the best breadth first search algorithm.

2. *FindNextNeighborForBestBreadth(classname: String) : String* ... This method finds the next neighbor for a current node for the best breadth first algorithm.

3. *FindUnvisitedNeighborForBestBreadth(classname: String, visitedNodes: {String}) : String* ... This method finds the next unvisited neighbor for a current node for the best breadth first algorithm.

class product_path_method_generate
categoryof: access_relevance_algorithm
attributes:
Characteristic: String;
methods:
ProductPathMethodGenerate (source: String; target: String):
PathMethodType ;
FindNextNeighborForProductPmg (classname: String): String;
FindNextNeighborForProductPmgVisited (classname: String,
targetname: String,
visitedNodes: {String}): String;

A description of each method is given below.

1. *ProductPathMethodGenerate(source: String; target: String): PathMethodType* ... generates a path–method using product_path_method_generate algorithm from a source to the target.

2. *FindNextNeighborForProductPmg(classname: String) : String* ... This method finds the next neighbor for a current node for the product_path_method_generate algorithm.

3. *FindUnvisitedNeighborForProductPmg(classname: String, visitedNodes: {String}) : String* This method finds the next unvisited neighbor for a current node for the product_method_generate algorithm.

8.3 Computation Algorithms of Path–Method Generator

We have discussed computational algorithms such as *PRODUCT_AR, MINIMUM_AR, and COMPUTE_ARM* in Chapter 5.

class computational_algorithm
categoryof: algorithm
memberof: computational_algorithms
relationships:
SchemaGraph: schema_graph;
methods:
CheckStatus (matrix: MatrixType): Boolean;

A description of each method is given below.

1. *CheckStatus (matrix: WeightMatrixType): Boolean* ... This method checks whether a weight matrix is updated or not. It is good to check whether we need re–computation or not.

class computational_algorithms
setof: computational_algorithm
attributes:
NumAlgorithms: String;
Purpose: String;
relationships:
PathMethodNavigator: path_method_navigator;
methods:
DisplaySelectors ();
DisplayChoices ();

A description of each method is given below.

1. *DisplaySelectors ()* ... displays signatures of all the computation algorithms.
2. *DisplayChoices ()* ... displays names and characteristics of each algorithm to the user. Then the user can select one, based on his/her needs.

class product_ar
categoryof: computation_algorithm
attributes:
Characteristic: String;
EfficiencyDescription: String;
ProductHeap: HeapType;
AccessRelevanceMatrix: RelevanceMatrixType;
AccessWeightMatrix: WeightMatrixType;
methods:
ComputeProductAccessRelevance(): RelevanceMatrixType;

A description of each method is given below.

1. *ComputeProductAccessRelevance(): RelevanceMatrixType* ... This method accesses a Schema Graph and computes an access relevance matrix using the PRODUCT weighting function.

8.4 How does Path–Method Generator Work?

The path_method_generator$_i$ (PMG) is an instance of **path_method_generator**. Similarly, query_translator$_i$ is an instance of class query_translator$_i$. It is similar for other components of the Path–Method Generator such as Path–Method Navigator, Traversal Algorithms, etc.

Figure 8.3 shows instances for the Path–Method Generator. First, the user query or update request is given to the path_method_generator$_i$. It calls path_method_editor$_i$. The path_method_editor$_i$ calls term_classifier$_i$ to check the terms used in the user–request. Then the schema–defined terms, as a pair (source, target), are passed to path_method_navigator$_i$. The path_method_navigator$_i$ calls traversal_algorithms$_i$, which is a set of algorithms with a variety of characteristics. One of these algorithms is selected for execution, and returns a generated path–method. This generated path–method is returned to path_method_navigator$_i$. Then path_method_navigator$_i$ returns this path–method to path_method_editor$_i$. The path_method_editor$_i$ allows the user to modify the path–method if s/he wants to. Then, the path–method is returned to the query_translator$_i$.

The Figure 8.3 traversal_algorithms$_i$ contains four different Traversal Algorithms,

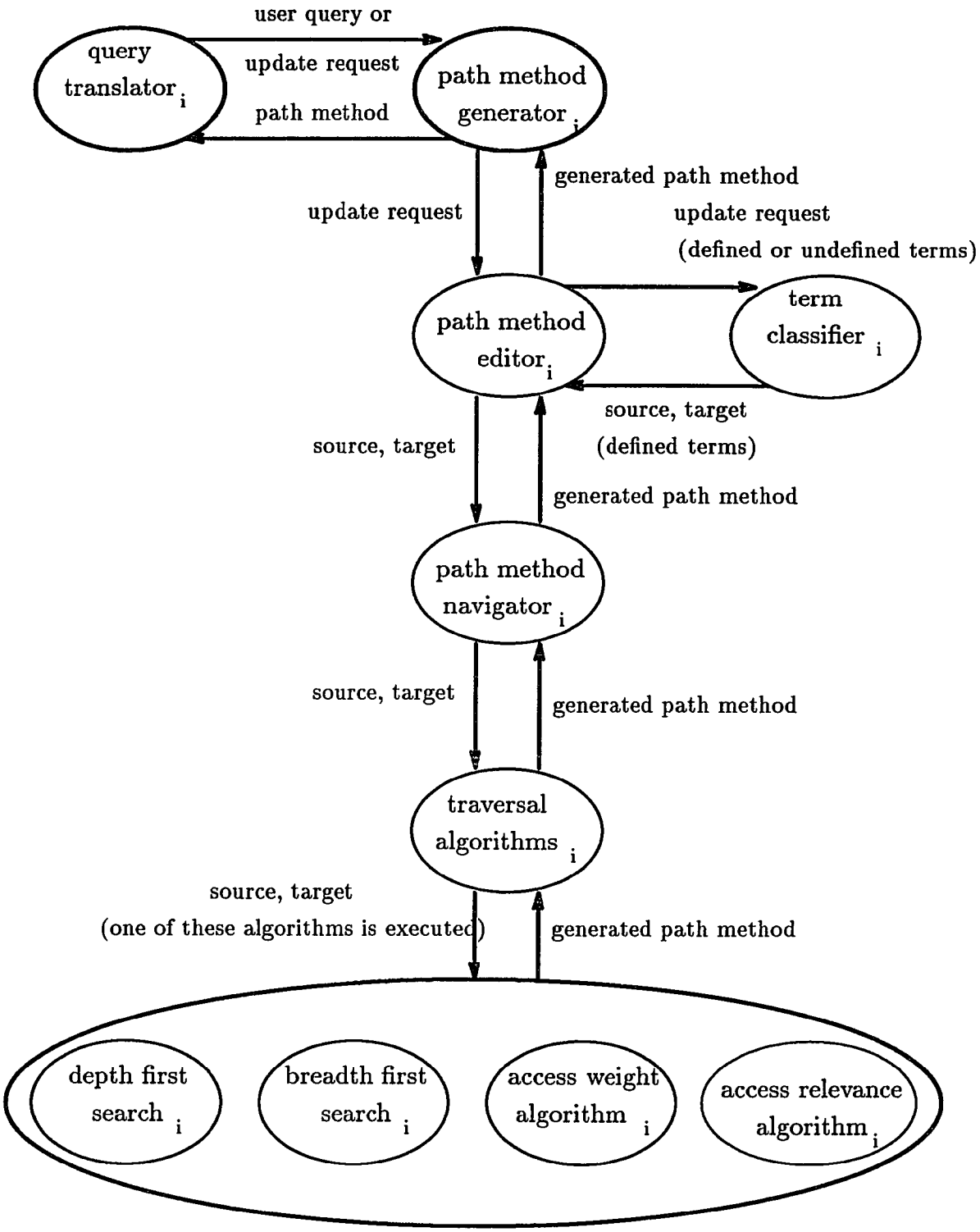

Figure 8.3 Objects for Path-Method Generator

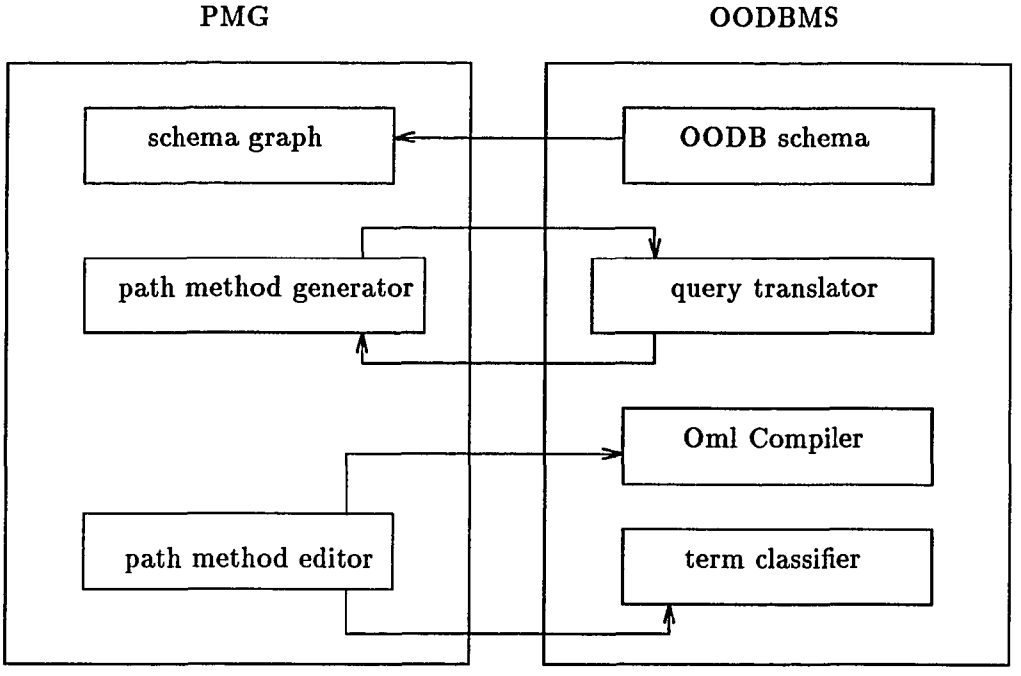

Figure 8.4 Connections Between the PMG and an OODBMS

depth_based_algorithm$_i$, breadth_based_algorithm$_i$, access_weight_algorithm$_i$, and access_relevance_algorithm$_i$.

8.5 Integrating PMG with an OODBMS

Now we will explain how the PMG can be incorporated into an OODBMS. As is shown in Figure 8.4, only four classes of an OODBMS that are involved in this integration process. In the next chapter we will discuss the specification of the Path–Method Generator module for VODAK/VML OODB.

Actually, the Path–Method Generator module can be incorporated into any OODB with little effort. Initially, the OODB schema should be converted into a schema graph. If the OODBMS supports a Term Classifier than Path–Method Editor can

use it to correct terms specified in the user queries. Traversal algorithm will traverse the schema graph without interacting with any of the classes of the OODBMS. The Path–Method Generator will generate a path–method using the syntax discussed in Chapter 2. The Query Translator of the OODBMS should be modified in such a way that it sends all the incomplete queries to the Path–Method Generator, rather than reporting error messages. If we want to store generated path–methods then they must be converted into the Object Modeling Language used by the OODB. Later on it can be added to the dispatch table of the source class of the path–method by the Oml–Compiler.

As shown earlier, the class **schema_graph** has a method *CreateSchemaGraph ()*, which can be modified based on the OODB model. The class **path_method_editor** has a method *CheckUserRequest()*. This method can be modified based on the Term–Classifier of the OODBMS. The class **path_method_generator** has a method *ReturnPathMethod()*. It has to be modified in such a way that it can return the generated path–method in the syntax of the Object Modeling Language supported by the OODB. These changes are necessary in PMG, not in the existing OODBMS. The only change in the existing OODBMS needed is that the Query–Translator should direct all incompleted queries to the Path–Method Generator.

CHAPTER 9

IMPLEMENTING PMG FOR VODAK/VML OODB

In the previous chapter we have discussed the design of the PMG for our general OODB model. For practical research we use the VODAK/VML OODB prototype. In this chapter we will discuss the implementation of PMG using the VODAK/VML prototype. A more detailed discussion on the specification of the PMG system for VODAK/VML OODB prototype can be found in [MPG92]. A detailed implementation of the PMG system is discussed in 'PMG User and Reference Manual' [MPG93].

9.1 VODAK/VML OODB Prototype

The VODAK/VML system is an OODB prototype developed at GMD–IPSI, Darmstadt, Germany. 'VODAK' is an object–oriented data model and 'VML' is an acronym for Vodak Modeling Language. The implementation language of the system is in C++. There are two distinguishing features of the VODAK/VML prototype, compared to most existing OODB models.

1. Use of the Dual Model

2. A sophisticated system of Metaclasses

The VODAK/VML prototype is based on the Dual Model [NPGT91, GPN91, NPGT90, NPGT89] for OODBs. The Dual Model separates structural and semantic

aspects of a class definitions. The structural aspects are specified by an object type and the semantic aspects are specified as a class. Thus, each class has an object type attached to it, which describes its structure. Generally, all the object types are defined first and then all the classes are defined. The advantage is that there may be several classes which share the same object type. A more detailed discussion and examples of classes which share the same object type are given in [NPGT91]. Note that in this dissertation model we have not considered such a separation in the class definition. A more detailed discussion of the VODAK/VML system is given in [KNBD92].

To implement the PMG system using VODAK/VML, one can use the approach discussed in the previous chapter, where only a few classes need to be changed. But as this research is done in close co–operation with GMD–IPSI, where the VODAK/VML prototype has been developed, we will now show all the classes of the path–method generator using the VML formalism. This is done, because the PMG system will become a module of the VODAK/VML OODB system.

We will explain only a class definitions and rest of the class definitions here are shown in the Section 9.3. In a VML object type definition both attributes, and user defined relationships are considered as properties. As *setof*, and *memberof* are not supported by VML, we have considered them as user–defined relationships. We define the object type *PATH_METHOD_GENERATOR* as follows.

```
OBJECTTYPE PATH_METHOD_GENERATOR
                    [path_meth_ed: PATH_METHOD_EDITOR;
```

```
                    path_meth_nav: PATH_METHOD_NAVIGATOR;
                    query_trans: VML_QUERY_TRANSLATOR]

    PROPERTIES
        NoOfEditors: INT;
        NoOfTravAlgorithms: INT;
        PathMethodEditor: path_meth_ed;
        PathMethodNavigator: path_meth_nav;
        QueryTranslator: query_trans;

    INTERFACE
    METHODS
        Initialize_PMG () READONLY;
        Invoke_path_meth_editor () READONLY;
        Invoke_path_method_navigator () READONLY;
        Generate_path_method () READONLY;
        Return_path_method (path_method: Path_method_type) READONLY;
END;

CLASS path_method_generator
        INSTTYPE PATH_METHOD_GENERATOR
                                                [path_method_editor
                                                path_method_navigator
                                                vml_query_translator]
END;
```

It has two attributes *NoOfEditors* and *NoOfTravAlgorithms* and three relationships *PathMethodEditor*, *PathMethodNavigator*, and *QueryTranslator*. For each user–defined relationship we need to pass a formal class parameter. The formal class parameter for the relationship *PathMethodEditor* is *path_meth_ed*. The object type of *path_meth_ed* is *PATH_METHOD_EDITOR*. Similarly, the formal class parameters

for relationships *PathMethodNavigator* and *QueryTranslator* are *path_meth_nav* and *query_trans*, respectively. The object types for the relationships *PathMethodNavigator* and *QueryTranslator* are *PATH_METHOD_NAVIGATOR* and *QUERY_TRANSLATOR*, respectively.

The purpose of class parameters is to handle cases when more than one class shares the same object type. For example, the relationship *PathMethodEditor* refers to a class, which has an object type *PATH_METHOD_EDITOR*. There can be more than one class that can have the same object type *PATH_METHOD_EDITOR*. Actually, for PMG classes, there are no such cases. Each class has its own object type.

After specifying properties, methods are defined. We need a keyword 'INTERFACE' for the public methods. VML requires 'READONLY' in the method header to indicate that the method is side–effect free. From VML, C++ functions can also be called. Here, we will not discuss the complete implementation of each method. For more detailed discussion see [MPG92, MPG93].

The above is a definition of the class **path_method_generator**. VML used the keyword 'INSTTYPE' followed by *PATH_METHOD_GENERATOR* specifies that the class **path_method_generator** has the object type *PATH_METHOD_GENERATOR*. Three following parameters in [...], are classes which have the corresponding object types specified in the object type definition. As an another example, the following is a definition for the object type *PATH_METHOD_NAVIGATOR* and the class *path_method_navigator*.

```
OBJECTTYPE PATH_METHOD_NAVIGATOR
                    [path_meth_gen: PATH_METHOD_GENERATOR;
                    path_meth_ed: PATH_METHOD_EDITOR;
                    schema_graph: SCHEMA_GRAPH;
                    compute_algs: COMPUTE_ALGORITHMS;
                    traversal_algs: TRAVERSAL_ALGORITHMS];

    PROPERTIES
        Source: STRING;
        Target: STRING;
        NoOfAlgorithms: INT;
        NoOfVisitedNodes: INT;
        PathLength: INT;
        PathMethodGenerator: path_meth_gen;
        PathMethodEditor: path_meth_ed;
        SchemaGraph: schema_graph;
        ComputationalAlgorithms: compute_algos;
        TraversalAlgorithms: traversal_algos;

    INTERFACE
    METHODS
        Select_traversal_algorithm (): INT READONLY;
        Call_traversal_algorithm (choice: INT) READONLY;
        Report_unsuccessful_search(message: Message_type) READONLY;
        Generate_path_method(source: STRING; target: STRING):
                        Path_method_type READONLY;
        Run_path_method_navigator () READONLY;
        Compute_access_relevance (relevance_mat: RelevanceMatrixType)
                            READONLY;
END;

CLASS path_method_navigator
        INSTTYPE PATH_METHOD_NAVIGATOR
                            [path_method_generator,
```

```
                    path_method_editor,
                    schema_graph,
                    compute_algorithms,
                    traversal_algorithms]
END;
```

The second important feature of the VODAK/VML system is the extensive use of metaclasses. A more detailed discussion of metaclasses is given in [K90b, KNBD92]. Each objecttype is a *SUBTYPEOF* of *Metaclass_InstType*, a system defined object type. Each class is an instance of the metaclass *VML_CLASS*, a system defined metaclass.

In addition, two semantic generic relationships *categoryof* and *roleof* are realized using metaclasses, i.e., metaclasses are defined to implement the behavior of such semantic relationships. The following is the definition for the class **algorithm**. We define the class **traversal_algorithm** as a *categoryof* the class **algorithm**. The line *METACLASS CATEGORY_SPECIALIZATION_CLASS* in the following class **traversal_algorithm** specifies that a class is an instance of the metaclass **CATEGORY_SPECIALIZATION_CLASS**. The class **CATEGORY_SPECIALIZATION_CLASS** contains necessary attributes and methods to support the behavior of the relationship *categoryof*.

Note that the object type *TRAVERSAL_ALGORITHM* is a subtype of the object type *ALGORITHM*. This is necessitated by the Dual Model [NPGT91]. There, it is explained that when class a is a *categoryof* of class b, then the corresponding object type A of a, is a *SUBTYPEOF* of the object type B of b.

```
OBJECTTYPE ALGORITHM

    PROPERTIES
        Algorithm_name: STRING;
        Parameters: || STRING → STRING ||;
        Purpose: STRING;

    INTERFACE
    METHODS
        Display_name () READONLY;
        Display_code () READONLY;
        Display_signature () READONLY;
END;

CLASS algorithm
        INSTTYPE ALGORITHM
END;
```

In the following discussion || <key> → <data> || specifies a dictionary datatype. The object types *VML_PROPDESL*, *VML_CLASSDECL*, are defined in VML. A more detailed discussion of all the datatypes supported by VML is given in [KNBD92].

```
OBJECTTYPE TRAVERSAL_ALGORITHM
                    [trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH];
            SUBTYPEOF ALGORITHM;

    PROPERTIES
        SourceNode: STRING;
        TargetNode: STRING;
        PathLength: INT;
        NumVisitedNodes: INT;
        Successful: BOOL;
```

```
        SchemaGraph: schema_gr;
        memberof: trav_algs;

    INTERFACE
    METHODS
        Check_attributes (source_class: STRING; target_class: STRING):
                    BOOL READONLY;
        Get_attributes (classname: STRING):
                || STRING → VML_PROPDESC || READONLY;
        Get_relationships (classname: STRING):
                || STRING → VML_CLASSDECL || READONLY;
        Get_user_defined_rels (classname: STRING):
                || STRING → VML_CLASSDECL || READONLY;
        Get_generic_relationships (classname: STRING):
                || STRING → VML_CLASSDECL || READONLY;
        Get_attribute_selectors (classname: STRING) : {STRING};
        Get_relationship_selectors (classname: STRING): {STRING};
        Get_user_defined_rel_selectors (classname: STRING): {STRING};
        Get_generic_relationship_selectors (classname: STRING): {STRING};
        Add_to_path_method(pair: || STRING → VML_CLASSDECL ||;
            pathmethod: Path_method_type): Path_method_type READONLY;
END;

CLASS traversal_algorithm METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE TRAVERSAL_ALGORITHM
                        [schema_gr : SCHEMA_GRAPH,
                        traversal_algos : TRAVERSAL_ALGORITHMS]
END;
```

9.2 Other PMG Classes for VODAK/VML OODB

In this section we will give definitions for rest of the classes, of the PMG implementation of VODAK/VML.

```
OBJECTTYPE PATH_METHOD_EDITOR
                    [path_meth_gen: PATH_METHOD_GENERATOR;
                    path_meth_nav: PATH_METHOD_NAVIGATOR;
                    term_classifier: VML_TERM_CLASSIFIER;
                    vml_schema: VML_SCHEMA];

    PROPERTIES
        Source: STRING;
        Target: STRING;
        NoOfVisitedNodes: INT;
        PathLength: INT;
        PathMethodGenerator: path_meth_gen;
        PathMethodNavigator: path_meth_nav;
        TermClassifier: vml_term_classifier
        VmlSchema: vml_schema;

    INTERFACE
    METHODS
        Display_one_method(path_meth: Path_method_type) READONLY;
        Display_error_message(message: Message_type) READONLY;
        Read_user_request(source: STRING; target: STRING) READONLY;
        Generate_path_method(source: STRING; target: STRING):
                        Path_method_type READONLY;
        Check_user_request(source: STRING; target: STRING):
                        User_request_type READONLY;
        Check_class_name(classname: STRING) : BOOL READONLY;
        Check_object_type(objecttype: STRING) : BOOL READONLY;
        Accept_path_method(pathmethod: Path_method_type) READONLY;
        Update_path_method(pathmethod: Path_method_type) READONLY;
END;

CLASS path_method_editor
        INSTTYPE PATH_METHOD_EDITOR
                        [path_method_generator,
```

```
                    path_method_navigator,
                    term_classifier,
                    vml_schema]
END;

OBJECTTYPE SCHEMA_GRAPH
                [path_meth_nav: PATH_METHOD_NAVIGATOR;
                vml_schema: VML_SCHEMA;]

    PROPERTIES
        Nodes: {Node_type};
        Connections: {Connection_type};
        NoOfNodes: INT;
        AccessWeightMatrix: Weight_matrix_type;
        AccessRelevanceMatrix: Relevance_matrix_type;
        AdjacencyList: Adjacency_list_type;
        PathMethodNavigator: path_meth_nav;
        VmlSchema: vml_schema;

    INTERFACE
    METHODS
        Create_schema_graph () READONLY;
        Node_number (classname: STRING) : INT READONLY;
        Access_weight(class_one: STRING; class_two: STRING) :
                        REAL READONLY;
        Connection_access_weight(connection: Connection_type) :
                REAL READONLY;
        Access_relevance(class_one: STRING; class_two: STRING) :
                REAL READONLY;
        NeighborClassNames(classname: STRING) : {STRING} READONLY;
        Neighbors(classname: STRING) : {ConnectionType} READONLY;
        Permissible_neighbor_class_names(classname: STRING; visited: {STRING}):
                {STRING};
        Permissible_neighbors(classname: STRING; visited: {STRING}):
```

```
                    {ConnectionType}; END;

CLASS schema_graph
        INSTTYPE SCHEMA_GRAPH
                                [path_method_navigator,
                                vml_schema]
END;

OBJECTTYPE TRAVERSAL_ALGORITHMS
                    [trav_alg: TRAVERSAL_ALGORITHM;
                    path_meth_nav: PATH_METHOD_NAVIGATOR]

    PROPERTIES
        NoOfAgorithms: STRING;
        Purpose: STRING;
        setof: {trav_alg};
        PathMethodNavigator: path_meth_nav;

    INTERFACE
    METHODS
        Display_signatures () READONLY;
        Display_choices () READONLY;
END;

CLASS traversal_algorithms
        INSTTYPE TRAVERSAL_ALGORITHMS
                                [traversal_algorithms,
                                path_method_navigator]
END;

OBJECTTYPE DFS_BASED_ALGORITHM
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
              SUBTYPEOF TRAVERSAL_ALGORITHM
                    [ trav_algs
```

```
                    schema_gr];

    PROPERTIES
        GroupName: STRING;
        PredeterminedDepth: INT;
        DfsStack: Stack_type;

    INTERFACE
    METHODS
        Display_characteristics () READONLY;
        Display_stack () READONLY;
END;

CLASS dfs_based_algorithm
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE DFS_BASED_ALGORITHM
                            [ traversal_algorithms,
                            schema_graph]
END;


OBJECTTYPE BFS_BASED_ALGORITHM
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
            SUBTYPEOF TRAVERSAL_ALGORITHM
                    [ trav_algs
                    schema_gr];

    PROPERTIES
        GroupName: STRING;
        PredeterminedWidth: INT;
        BfsQueue: Queue_type;

    INTERFACE
```

```
    METHODS
        Display_characteristic () READONLY;
        Display_queue () READONLY;
END;

CLASS bfs_based_algorithm
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE BFS_BASED_ALGORITHM
                                [ traversal_algorithms,
                                schema_graph]
END;


OBJECTTYPE ACCESS_WEIGHT_ALGORITHM
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
              SUBTYPEOF TRAVERSAL_ALGORITHM
                    [ trav_algs
                    schema_gr];

    PROPERTIES
        GroupName: STRING;
        AccessWeightMatrix: Weight_matrix_type;
        DfsStack: Stack_type;
        BfsQueue: Queue_type;

    INTERFACE
    METHODS
        Display_weight_matrix () READONLY;
        Get_access_weight (row: INT, column: INT): REAL READONLY;
        Set_access_weight (row: INT, column: INT) READONLY;
END;

CLASS access_weight_algorithm
```

```
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE ACCESS_WEIGHT_ALGORITHM
                                [ traversal_algorithms,
                                schema_graph]
END;


OBJECTTYPE ACCESS_RELEVANCE_ALGORITHM
                        [ trav_algs: TRAVERSAL_ALGORITHM;
                        schema_gr: SCHEMA_GRAPH]
                SUBTYPEOF TRAVERSAL_ALGORITHM
                        [ trav_algs
                        schema_gr];

    PROPERTIES
        GroupName: STRING;
        AccessRelevanceMatrix: Relevance_matrix_type;
        DfsStack: Stack_type;
        BfsStack: Queue_type;
        WeightingFunctionName: STRING;

    INTERFACE
    METHODS
        Display_relevance_matrix () READONLY;
        Get_access_weight (row: INT, column: INT): REAL READONLY;
        Set_access_weight (row: INT, column: INT) READONLY;
END;

CLASS access_relevance_algorithm
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE ACCESS_RELEVANCE_ALGORITHM
                                [ traversal_algorithms,
                                schema_graph]
END;
```

```
OBJECTTYPE DEPTH_FIRST_SEARCH
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
            SUBTYPEOF DFS_BASED_ALGORITHM;
                    [ trav_algs
                    schema_gr];

    PROPERTIES
        Characteristic: STRING;

    INTERFACE
    METHODS
        Depth_first_search(source: STRING; target: STRING):
                    Path_method_type READONLY;
        Find_next_neighbor_for_dfs(classname: STRING) :
            STRING READONLY;
        Find_next_neighbor_for_dfs(classname: STRING,
            visitedNodes: {STRING}):
                    STRING READONLY;
END;

CLASS depth_first_search
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE DEPTH_FIRST_SEARCH
                            [ traversal_algorithms,
                            schema_graph]
END;

OBJECTTYPE BREADTH_FIRST_SEARCH
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
            SUBTYPEOF BFS_BASED_ALGORITHM;
```

```
                    [ trav_algs
                    schema_gr];

    PROPERTIES
        Characteristic: STRING;

    INTERFACE
    METHODS
        Breadth_first_search(source: STRING; target: STRING):
                    Path_method_type READONLY;
        Find_next_neighbor_for_bfs(classname: STRING):
                        STRING READONLY;
        Find_next_neighbor_for_bfs(classname: STRING,
            visitedNodes: {STRING}): STRING READONLY;
END;

CLASS breadth_first_search
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE BREADTH_FIRST_SEARCH
                            [ traversal_algorithms,
                            schema_graph]
END;


OBJECTTYPE BEST_FIRST_SEARCH
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
                SUBTYPEOF ACCESS_WEIGHT_ALGORITHM,
                    DEPTH_FIRST_SEARCH;
                    [ trav_algs
                    schema_gr];

    PROPERTIES
        Characteristic: STRING;
```

```
    INTERFACE
    METHODS
        Best_first_search(source: STRING; target: STRING):
            Path_method_type READONLY;
        Find_next_neighbor_for_best_first(classname: STRING):
            STRING READONLY;
        Find_next_neighbor_for_best_first(classname: STRING,
                visitedNodes: {STRING}): STRING READONLY;
END;

CLASS best_first_search
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE BEST_FIRST_SEARCH
                                [ traversal_algorithms,
                                schema_graph]
END;


OBJECTTYPE BEST_BREADTH_FIRST_SEARCH
                    [ trav_algs: TRAVERSAL_ALGORITHM;
                    schema_gr: SCHEMA_GRAPH]
                SUBTYPEOF BREADTH_FIRST_SEARCH,
                    ACCESS_WEIGHT_ALGORITHM;
                    [ trav_algs
                    schema_gr];

    PROPERTIES
        Characteristic: STRING;

    INTERFACE
    METHODS
        Best_breadth_first_search(source: STRING; target: STRING):
                    Path_method_type READONLY;
```

```
        Find_next_neighbor_for_best_breadth(classname: STRING):
            STRING READONLY;
        Find_next_neighbor_for_best_breadth(classname: STRING,
            visitedNodes: {STRING}): STRING READONLY;
END;

CLASS best_breadth_first_search
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE BEST_BREADTH_FIRST_SEARCH
                                [ traversal_algorithms,
                                schema_graph]
END;


OBJECTTYPE PRODUCT_PATH_METHOD_GENERATE
                        [ trav_algs: TRAVERSAL_ALGORITHM;
                        schema_gr: SCHEMA_GRAPH]
                SUBTYPEOF ACCESS_RELEVANCE_ALGORITHM;
                        [ trav_algs
                        schema_gr];

    PROPERTIES
        Characteristic: STRING;

    INTERFACE
    METHODS
        Product_path_method_generate(source: STRING; target: STRING):
                        Path_method_type READONLY;
        Find_next_neighbor_for_product_pmg(classname: STRING):
                        STRING READONLY;
        Find_next_neighbor_for_product_pmg(classname: STRING,
                        visitedNodes: {STRING}): STRING READONLY;
END;
```

```
CLASS path_method_generate
        METACLASS CATEGORY_SPECIALIZATION_CLASS
    INSTTYPE PRODUCT_PATH_METHOD_GENERATE
                    [ traversal_algorithms,
                    schema_graph]
END;

OBJECTTYPE COMPUTATIONAL_ALGORITHMS
                [compute_alg: COMPUTATIONAL_ALGORITHM;
                path_meth_nav: PATH_METHOD_NAVIGATOR]

    PROPERTIES
        NumAlgorithms: STRING;
        Purpose: STRING;
        setof: {comp_alg};
        PathMethodNavigator: path_meth_nav;

    INTERFACE
    METHODS
        Display_selectors () READONLY;
        Display_choices () READONLY;
END;

CLASS computational_algorithms
        INSTTYPE COMPUTATIONAL_ALGORITHMS
END;


OBJECTTYPE COMPUTATIONAL_ALGORITHM
                [comp_algs: COMPUTATIONAL_ALGS;
                schema_graph: SCHEMA_GRAPH;
                vml_schema: VML_SCHEMA]
            SUBTYPEOF ALGORITHM;
```

```
    PROPERTIES
        SchemaGraph: schema_graph;
        VmlSchema: vml_schema;
        memberof: comp_algs;

    INTERFACE
    METHODS
        Check_status (matrix: Matrix_type): BOOL READONLY;
END;

CLASS computation_algorithm
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE COMPUTATIONAL_ALGORITHM
END;


OBJECTTYPE PRODUCT_AR
                    [comp_algs: COMPUTATIONAL_ALGS;
                    schema_graph: SCHEMA_GRAPH;
                    vml_schema: VML_SCHEMA]
            SUBTYPEOF COMPUTATIONAL_ALGORITHM
                    [comp_algs,
                    schema_graph,
                    vml_schema];
    PROPERTIES
        Characteristic: STRING;
        Efficiency_description: STRING;
        Product_heap: Heap_type;
        Access_relevance_matrix: Relevance_matrix_type;
        Access_weight_matrix: Weight_matrix_type;

    INTERFACE
    METHODS
```

```
        Compute_product_access_relevance(schema: schema_graph):
                        Relevance_matrix_type READONLY;
END;

CLASS product_ar
            METACLASS CATEGORY_SPECIALIZATION_CLASS
        INSTTYPE PRODUCT_AR
                            [ computation_algorithm,
                            schema_graph,
                            vml_schema];
END;
```

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

In this dissertation we have investigated the problem of automatic generation of path–methods in object–oriented databases. A path–method, defined as a method which traverses from a class through a chain of connections between classes, is a mechanism to retrieve or to update information relevant to a source class that is not stored with that class but with some other class.

The state–of–the–art in the object–oriented database technology does not support automatic generation of path–methods. It is assumed that path–methods to support queries are already written. However, it is a difficult task for a user to write such path–methods. It may require knowledge of many of the classes in the schema, while a typical user has incomplete, inconsistent, or even incorrect knowledge of the schema.

In writing path–methods ahead of time it is necessary to predict what kind of user requests will be applied to each class in the schema. To write *ad hoc* queries is a frustrating task, as incorrect queries will be rejected without proper guidance by the database. We have developed the Path–Method Generator (PMG), a System for semi–automatic generation of path–methods in an object–oriented database, based on a naive user's requests. The Path–Method Generator allows a user to formulate his request according to his understanding of the conceptual schema. It does not require any predefined views or prior knowledge of the conceptual schema. The path–methods

will be generated dynamically rather than written in advance for all the classes in the schema.

To support the generation of path–methods in OODBs we introduced the notion of access weights. We enhance an OODB by assigning access weights to all the connections of an OODB schema according to the frequency of their use during the operation of the OODB. Assigning weights to the connections of the schema is a novel apporach, which is not supported by any existing OODB models. We discussed several access weight traversal algorithms. From the large university environment OODB that we designed to be used as a testbed, we have selected a subschema containing 52 classes. We defined 50 sample problems for the Path–Method Generator, using this schema. This was done independently of the development of the traversal algorithms. We generated 50 path–methods for these problems, using various access–weight–based traversal algorithms. Our experiments show that the best of these algorithms, *best breadth first search*, performed relatively well, but not well enough. This algorithm generated 86% of the desired path–methods. Surprisingly, the *breadth first search* found 82% of the desired path–methods.

To find a particular item of information, a human traverses an OODB schema by using his intuitive understanding of the schema and the target information. To perform a similar traversal we have introduced the notion of access relevance of a path, which can be computed from access weights of all the connections of a path–method. This is a better measure than access weight as it incorporates access weights of all the

connections that make up the path. The access relevance from class A to another class B is defined as the maximum access relevance over all paths from the class A to the class B. We have designed efficient algorithms for computing access relevance for all pairs of classes of a schema for one OODB. For directed schemas we have developed two algorithms for the two t–norms used each of complexity $O(ne \lg n)$ or $O(n^3)$ (depending on the implementation) to compute $O(n^2)$ values. For bidirected schema, to compute access relevance using MINIMUM weighting function, we designed a very efficient algorithm of complexity $O(n^2)$ to compute $O(n^2)$ values.

We described an algorithm *PathMethodGenerate* which uses precomputed access relevance to guide path–method generation. At each step of traversal this algorithm considers the access weight from the current class to a neighbor class, and the access relevance from the neighbor class to the target information. This technique improved the results considerably. We have performed experiments with the same schema and sample set of problems. The algorithm *PathMethodGenerate (PRODUCT (Rule 2b))* found 92% of the desired path–methods. The algorithm *PathMethodGenerate (MINIMUM (Rule 2b))* found 32% of the desired path–methods. These results show that access relevance should be computed using the PRODUCT t–norm which reflects all the access weights along the path rather than the MINIMUM t–norm. The algorithm *PathMethodGenerate (PRODUCT (Rule 2a))* found 74% of the desired path–methods. This shows that the Rule 2b is better than the Rule 2a. The results for the PRODUCT t–norm are better than those of **best_breadth_first_search**. Note

that even for this algorithm, there are a few cases when the generated path–methods are not the desired ones. For such cases we introduced two mechanisms which were helpful for finding the desired path–method at the second try. These mechanisms utilize parameters of forbidden classes and intermediate classes, respectively, which are supplied by the user based on the feedback obtained from the undesired path–method. These mechanisms were very helpful in generating desired path–methods for such cases.

Path–method generation in an interoperable multi–OODB is more difficult than for a single OODB. We have introduced a novel hierarchical approach to model an IM–OODB schema by a smaller schema. Then, we discussed efficient algorithms for computation of access relevance for an IM–OODB schema. This algorithm is of complexity $O(c_A * c_B)$, where c_A is a number of contact classes of OODB_A and c_B is a number of contact classes of OODB_B, respectively. We have also shown an enhanced technique for realization of inter–OODB connections, using path–methods through the IM–OODB schema classes.

We discussed techniques incorporating the PMG in an existing OODBMS. We also implemented such a PMG into the VML system.

The following issues are topics of future research.

1. So far in this dissertation we have discussed path–method generation of path–methods from the source class to the target class. However, many queries require a more complex structure. Our approach can be extended to generate

branching methods, which have a tree structure rather than a path structure. One can also extend this approach to generate more complex methods of an acyclic graph structure.

2. The automatic generation of views in an OODB is also an important problem. An interesting approach is to define a view from one or more queries utilizing path–methods or methods with more complex structure as discussed in (1).

3. In this dissertation we have discussed computation of access relevance in an interoperable multi–OODB. One can develop traversal algorithms to generate path–methods to retrieve information in an interoperable multi–OODB.

4. In this dissertation, while discussing realization of inter–OODB connections, we have used similar attributes of different classes to correspond two classes in different databases. One can extend the approach to use attributes which are not defined in the classes needed to correspond, but in other classes in the respective databases. Such a realization can be achieved by using path–methods.

5. In Chapter 3 we have discussed the case that all the properties of the superclass are inherited to the subclasses. We also inherit the frequencies of the connections inherited from the superclass. One can extend this approach by only using each connection, inherited from the superclass, as they are traversed from the subclass rather than inheriting frequencies from the superclass. Then one can study the impact of such an refinement on the performance of PMG.

6. Suppose there already exist path–methods for a given class. How can the PMG use them as connections? What access weights should we associate with path–methods?

7. We have mentioned the notion of semantic resemblance between classes. If a system which assigns semantic resemblance to all the pairs of classes of a schema will be created, one can check the impact of utilizing semantic resemblance rather than access relevance on the performance of the traversal algorithms of the PMG.

8. Different applications might require independent sets of weights. An extension of the model that accomodates such weight vectors is possible.

REFERENCES

[AHU83] Aho, A., Hopcroft, J., Ullman, J.D., "Data Structures and Algorithms", *Addison–Wesley Publishing Company*, Reading, MA, 1983.

[AR91] Andersen, J., Reenskaug, T., "Operations on Sets in OODB", *OOPS Messenger, vol. 2, no. 4*, Oct. 1991, pp. 26–39.

[B89] Bertino, E., et al., "Integration of Heterogeneous Database Applications through an Object–Oriented Interface", *Information Systems, vol. 14*, 1989.

[B90] A. Bhave, "Implementation of University Database using the VML – The Object–Oriented Database System (Release 1)", *Master's Thesis, CIS–Dept*, NJIT, Newark, NJ, 1990.

[B91] Barry, D., "ITASCA Overview", *In the Proceedings of the Executive Briefing on Object–Oriented Database Management*, San Francisco, Sep. 1991.

[BD86] Bonissone, P.P., Decker, K.S., "Selecting Uncertainty Calculi and Granularity: An Experiment in Trading–off Precision and Complexity", *Machine Intelligence Pattern Recognition, vol. 4*, 1986, pp. 217–247.

[BH86] Bruce, H., Hull, R., "SNAP : A Graphics–based Schema Manager", *In the Proceedings of IEEE Computer and Data Engineering Conference*, 1986, pp. 151–165.

[BKK88] Banerjee, J., Kim, W., Kim, K., "Queries in Object–Oriented databases", *In the Proceedings of the Fourth International Conference on Data Engineering*, Los Angelos, CA, 1988, pp. 31–38.

[BNPS92] Bertino, E., Negri, M., Pelagatti, G., Sbattella, L., "Object–Oriented Query Languages: The Notion and the Issues", *IEEE Transactions on Knowledge and Data Engineering, vol. 4*, June 1992, pp. 223–237.

[BOS91] Butterworth, P., Otis, A., and Stein, J., "The GemStone Object Database Management System", *Communications of the ACM, vol. 20*, Oct. 1991, pp. 64–77.

[CD92] Cluet, S., Delobel, C.,"The General Framework for the Optimization of Object-Oriented Queries", *In the Proceedings of 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2–5, 1992, pp. 383–392.

[CM81] Clocksin, W.F., Mellish, C.S., "Programming in Prolog" Springer Verlag, 1981.

[CT90] Chao, H., Teli, V., "Development of a University Database using the Dual Model of Object–Oriented Knowledge Bases", *Master's Thesis*, CIS Department, NJIT, Newark, NJ, 1990.

[CT91a] Czejdo, B., Taylor, M., "Integration of Database Systems using an Object–Oriented Approach", in *Proc. First International Workshop on Interoperability in Multi–Database Systems*, Kyoto, Japan, 1991, pp. 30–37.

[CT91b] B. Czejdo, M. Taylor, "Integration of Database Systems and Smalltalk", in *In the Proceedings of the Symposium on Applied Computing*, Kansas City, April 1991.

[D91a] O. Deux et. al., "The O_2 System", *Communications of the ACM, vol. 34, no. 10*, October 1991, pp. 34–48.

[D91b] Dixit, N., "Implementation of University Database using the VML – The Object-oriented Database System (Release 2)", *Master's Project*, CIS–Dept, NJIT, Newark, NJ, 1991.

[F87] Fishman, D., et al., "IRIS: an object–oriented database management system", *ACM Transaction on Office Information Systems, vol. 5, no., 1,* Jan. 1987, pp. 48–69.

[FKN91] Fankhauser P., Kracker, M., Neuhold, E.J., "Semantic vs. Structural Resemblance of Classes", *SIGMOD Record, Special Issue on 'Semantic Issues in Multidatabase Systems', vol. 34*, Dec. 1991, pp. 59–63.

[FN92] Fankhauser, P., Neuhold, E. J., "Knowledged–Based Integration of Heterogeneous Databases", *In the Proceedings of the IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems, DS–5*, Lorne, Victoria, Australia, November 1992.

[G91] ————-, "Gemstone Product Overview", Servio Corporation, 1991.

[GD91] Gomsi, J., DeSanti, M., "VERSANT Overview", *In the Proceedings of the Executive Briefing on Object–Oriented Database Management*, San Fransisco, Sep. 1991.

[GKG85] Goldman, K. J., Kannellakis, P. C., Goldman, S. A., Zdonik, S. B., "ISIS : Interface for a Semantic Information System", *In the Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, 1985, pp. 328–342.

[GMPN92] Geller, J., Mehta, A., Perl, Y., Neuhold, E.J., Sheth, A., "Algorithms for Structural Schema Integration", *In the Proceedings of the Second International Conference on Systems Integration*, Morristown, NJ, June 1992, pp. 604–614.

[GOP90] K. Gorlen, S. Orlow, P. Plexico, "Data Abstraction and Object–Oriented Programming C++", John Wiley & Sons, 1990.

[GPCS92] Geller, J., Perl, Y., Cannata, P., Sheth, A., Neuhold, E.J., "Structural Integration: A Case Study", *In the Proceedings of the First International Conference on Information and Knowledge Management, CIKM–92*, Maryland, Nov. 1992, pp. 102–111.

[GPN91a] Geller, J., Perl, Y., Neuhold, E.J., "Structural Schema Integration in Heterogeneous Multi–Database Systems using the Dual Model", *In the Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991, pp. 200–203.

[GPN91b] Geller, J., Perl, Y., and Neuhold, E.J., "Structure and Semantics in OODB class Specifications", *SIGMOD Record, Special Issue on 'Semantic Issues in Multidatabase Systems', vol. 34*, Dec. 1991, pp. 40–43.

[GPNS92] Geller, J., Perl, Y., Neuhold, E. J., Sheth, A., "Structural Schema Integration with Full and Partial Correspondence using the Dual Model", *Information Systems, vol. 17, no. 6*, 1992.

[GR83] A. Goldberg, D. Robson, "Smalltalk-80, The Language and Its Implementation", Addison Wesley, 1983.

[HGPN92] Halper, M., Geller J., Perl, Y., Neuhold, E. J., "A Graphical Schema Representation for Object–Oriented Databases", *In the Proceedings of IDS92, International Workshop on Interfaces to Database Systems,* Glasgow, July 1992.

[HS89] Horowitz, E., Sahni, S., "Fundamentals of Computer Algorithms", *Computer Science Press*, Reading, MA, 1989.

[I93] Ishikawa, H., et. al., "The Model, Language, and Implementation of an Object–oriented Multimedia Knowledge Base Management System", *ACM Transactions on Database Systems, vol. 18, no. 1*, March 1993, pp. 1–50.

[JTTW88] Joseph, J., Thatte, S., Thompson, C., Wells, D., "Report on the Object–Oriented Database Workshop Held in Conjunction with the OOPSLA'88", San Diego, California, U.S.A., 1988.

[K89] Kim, W., "A Model of Queries for Object–Oriented Databases", *Proceedings of the Fifteenth International Conference on Very Large Databases*, 1989, pp. 423–432.

[K90a] Kim, W., "Introduction to Object–Oriented Databases", *The MIT Press*, Reading, MA, 1990.

[K90b] Klas, W., "A Metaclass System for Open Object–Oriented Databases", *Ph.D. Dissertation*, Technical University, Vienna, 1990.

[K90c] S. Kulkarni, "Implementation of University Database using the VML – The Object–Oriented Database System",*Master's Project*, CIS–Dept, NJIT, Newark, NJ, 1990.

[K91] Kracker, M., "Fuzzy Associative Concept Knowledge for Supporting the Formulation of Database Queries", *Ph.D–dissertation*, Technical University of Vienna, 1991.

[K92] Kracker, M., "A Fuzzy Concept Network Model and Its Applications", *Proceedings of FUZZ-IEEE '92*, San Diego, 1992, pp. 761-768

[KDN90] Kaul, M., Drosten, K., Neuhold, E.J., "ViewSystem: Integrating Heterogeneous Information Bases by Object–Oriented Views", *In the*

Proceedings of the IEEE International Conference on Data Engineering, 1990.

[KF88] Klir, G.L., Folger, T.A., "Fuzzy Sets, Uncertainty and Information", *Prentice Hall*, 1988.

[KKS92] Kifer, M., Kim, W., Sagiv, Y., "Querying Object-Oriented Databases", *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2–5, 1992, pp. 393-402.

[KM84] King, R., Melville, S., "Ski: A Semantic Knowledge Interface", *In the Proceedings of the Second International Conference on Entity–Relationship Approach*, North–Holand, 1983.

[KM90] Kemper, A., Moerkotte, G., "Advanced Query Processing in Object Bases using Access Support Relations", *In the Proceedings of the 16th International Conference on Very Large Databases, VLDB '90*, 1990, pp. 290–301.

[KN89] Kracker, M., Neuhold, E. J., "Schema Independent Query Formulation", In F. H. Lochovsky (Ed.), editor, *In the Proceedings of the 8th International Conference on Entity–Relationships Approach*, Toronto, Canada, 1989, pp. 233–247.

[KNBD92] Klas W., Neuhold E. J., Bahlke, R., Drosten K., Fankhauser P., Kaul M., Muth P., Oheimer M., Rakow T., Turau V., "VML Design Specification Document", *Tech Report, GMD–IPSI*, Germany, 1992.

[KNS88] Klas, W., Neuhold, E.J., Schrefl, M., "On an Object–Oriented Data Model for a Knowledge Base", *In the Proceedings of Research into Networks and Distributed Applications*, EUTECO 88, North–Holland, 1988.

[KNS89] Klas, W., Neuhold, E.J., Schrefl, M., "Tailoring Object–Oriented Data Models through Metaclasses", *In the Proceedings of Advanced Database System Symposium '89*, Kyoto, 1989, pp. 169–178.

[L85] Litwin, W., "Implicit Joins in the Multidatabase System MRDSM", *IEEE–COMPSAC*, 1985, pp. 495–504.

[L86a] Larson, J. A., "A Visual Approach to Browsing in a Database Environment", *IEEE Computer, vol. 19, no. 6*, 1986, pp. 62–71.

[L86b] Lieberman H., "Using Prototypical Objects to Implement Shared Behavior in Object–Oriented Systems", *In the Proceedings of OOPSLA '86, SIGPLAN Notices, vol. 21, no. 9*, 1986, pp. 214–223.

[LLOW91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D., "The Objectstore Database System", *Communications of the ACM, vol. 20*, Oct. 1991, pp. 50–63.

[LR92] Litwin, W., Risch, T., "Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates", *IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 6*, December 1992, pp. 517–528.

[LVZ92] Lanzelotte, R.S.G., Valduriez, P., Zait, M., "Optimization of Object–Oriented Recursive Queries using Cost–Controlled Strategies", *In the Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2–5, 1992, pp. 256–265.

[M83] Maier, D., "The Theory of Relational Databases", *Computer Science Press*, Reading, MA, 1983.

[M86] Motro, A., "Constructing Queries from Tokens", *In the Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986, pp.120–131.

[M89] McHugh, J.A., "Algorithmic Graph Theory", *Prentice Hall*, 1989.

[M91] Martin, R., "ONTOS Overview", *In the Proceedings of Executive Briefing on Object–Oriented Database Management*, San Fransisco, 1991.

[MBW80] Mylopoulos, J., Bernstein, P.A., Wong, H.K.T., "A Language Facility for Designing Database–Intensive Applications", *ACM Transactions on Database Systems, vol. 5, no. 2*, 1980, 185–207.

[MGPF92] Mehta, A., Geller, J., Perl, Y., Fankhauser, P., "Algorithms for Access Relevance to Support Path–Method Generation in OODBs", *In the Proceedings of the Fourth International Hong Kong Computer Society Database Workshop*, Shatin, Hong Kong, Dec. 12–13, 1992, pp. 183–200. Extended Abstract *In the Proceedings of the First International Conference on Information and Knowledge Management, CIKM–92*, Maryland, USA, Nov. 8–10, 1992, pp. 657.

[MGPF93] Mehta, A., Geller, J., Perl, Y., Fankhauser, P., "Computing Access Relevance to Support Path-Method Generation in Interoperable Multi-OODB", *(Full Paper) In the Proceedings of the RIDE–IMS'93: Third International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems*, Vienna, Austria, April 18–20, 1993, pp. 144–151.

[MPG92] Mehta, A., Geller, J., Perl, Y., "Path–Method Generator (PMG), Design Specification Document", Internal Document, CIS Department, NJIT, 1992.

[MPG93] Mehta, A., Geller, J., Perl, Y., "PMG User and Reference Manual", Internal Document, CIS Department, NJIT, *In Preparation.*

[MRSS87] Maier, D., Rozenshtein, D., Salvater, S., Stein, J., Warren, D., "PIQUE: A Relational Query Language without Relations," *Information Systems, vol. 12, no. 3*, 1987, pp. 317-335.

[MU83] Maier, D., Ullman, J. D., "Maximal Objects and the Semantic of Universal Relation Databases,"*ACM Transactions on Database Systems, vol. 8., no. 1*, 1983, pp. 1–14

[NPGT89] Neuhold, E.J., Perl, Y., Geller, J., Turau, V., "Separating Structural and Semantic Elements in Object–Oriented Knowledge Bases", *Advanced Database System Symposium*, Kyoto, Japan, 1989, pp. 67–74.

[NPGT90] Neuhold, E.J., Perl, Y., Geller, J., Turau, V., "A Theoretical Underlying Dual Model for Knowledge Based Systems", *In the Proceedings of the First International Conference on Systems Integration*, Morristown, NJ, 1990, pp. 96–103.

[NPGT91] Neuhold, E.J., Perl, Y., Geller, J., Turau, V., "The Dual Model for Object–Oriented Databases", *Research Report 91–30*, CIS Department, NJIT, 1991, Submitted for Publication.

[NS88] Neuhold, E.J., Schrefl, M.,"Dynamic Derivation of Personalized Views", *In the Proceedings of the 14th International Conference on Very Large Databases – VLDB '88*, 1988, pp. 183–194.

[OHMS92] Orenstein, J., Haradhwala, S., Margulies, B., Sakahara, D., "Query Processing in the ObjectStore Database System", *In the Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2–5, 1992, pp. 403–412.

[P87] Perl, Y., "Diagraphs with Maximum Number of Paths and Cycles", *Discrete Applied Mathematics, vol. 25*, 1987, pp. 257–271.

[P91a] Pandit, H.,"Implementation of University Database using the C++ Programming Language", *Master's Project*, CIS–Dept, NJIT, Newark, NJ, 1991.

[P91b] Patel, M., "Implementation of University Database using the VML – The Object–Oriented Database System (Release–3)", *Master's Project*, CIS–Dept, NJIT, Newark, NJ, 1991.

[PG79] Perl, Y., Golumbic, M.C., "Generalized Fibonacci Maximum Path Graphs", *Discrete Mathematics, vol. 28*, 1979, pp. 237–245.

[PW88] Pinson, L., Wiener, R., "An Introduction to Object–Oriented Programming and Smalltalk", Addison Wesley, 1988.

[PZ81] Perl, Y., Zaks, S., "Deficient generalized Fibonacci Maximum Path Graphs", *Discrete Mathematics, vol. 34*, 1981, pp. 153–164.

[RB92] Rundernsteiner, E.A., Lubomir, B., "Set Operations in Object–based Data Models", *IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 3*, June 1992, pp. 382–398.

[RC78] Reghbati, E., Corneil, D.G., "Parallel Computations in Graph Theory", *SIAM Journal Computing, vol. 7*, 1978, pp. 230–237.

[S88] Schrefl, M., "Object–oriented Database Integration", *Ph.D. Dissertation*, Technical University Vienna, 1988.

[S91a] Sheth, A., "Semantic Issues in Multidatabase Systems", *SIGMOD Record, Special Issue on 'Semantic Issues in Multidatabase Systems', vol. 34*, Dec. 1991, pp. 5–9.

[S91b] B. Stroustrup, "The C++ Programming Language", *Second Edition, Addison Wesley Publishing Company*, 1991.

[S92] Su, Stanley, "Modeling Methods in a Knowledge–Based System", *Presented at the Workshop on Current Issues in Databases and Applications*, Rut. Univ., Newark, NJ, October 22–23, 1992.

[SG89] Sheth, A., Gala, S., "Attribute Relationships : An Impediment to Automating Schema Integration", *In the Proceedings of the Workshop on Heterogeneous Database Systems*, Chicago, Dec. 1989.

[SK92a] Sheth, A., Kalinichenko, L., " Information Modeling in Multidatabase Systems: Beyond Data Modeling", *In the Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD, USA, Nov. 1992, pp. 8–16.

[SK92b] Sheth, A., Kashyap, V., "So Far (Schematically) Yet So Near (Semantically)", *In the Proceedings of the IFIP TC2/WG 2.6 Conference on Semantics of Interoperable Database Systems, DS–5*, Lorne, Victoria, Australia.

[SL90] Sheth, A.P., Larson, J.A., "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases", *ACM Computing Surveys, vol. 22, no. 3*, Sep. 1990, pp. 183–236.

[SN88a] Schrefl, M., Neuhold, E.J., "Object Class Definition by Generalization using Upward Inheritance", *In the Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, CA, Feb. 1–5, 1988, pp. 4–12.

[SN88b] Schrefl, M., Neuhold, E. J., "A Knowledge–Based Approach to Overcome Structural Differences in Object-Oriented Database Integration",

In the Proceedings of the Role of Artificial Intelligence in Database and Information Systems, IFIP Working Conference, Canton, China, July 1988.

[SS61] Schweizler, B., Sklar, A., "Associative Functions and Statistical Triangle Inequalities", *Publicationes Mathematicae Debercen, vol. 8*, 1961, pp. 169–186.

[SSR92] Sciore, E., Siegel, M., Rosenthal, A., "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems", *In Submission to Transaction on Database Systems.*

[WA90] Wijaya, C., Ahmedi, M., "Development of a University Database (Registration and Admission) using the Dual Model of Object–Oriented Knowledge Bases", *Master's Thesis*, CIS Department, NJIT, 1990.

[WP88] Wiener, R., Pinson, L., "An Introduction to Object–Oriented Programming and C++", *Addison Wesley Publishing Company*, 1988.

[Z65] Zadeh, L.A.,"Fuzzy Sets", *Information and Control, vol. 8*, 1965, pp. 228–353.