Translation of semantic aspects of OODINI graphical representation to ONTO OODB data definition language

Xiaoyong Wang
New Jersey Institute of Technology

ABSTRACT

TRANSLATION OF SEMANTIC ASPECTS OF OODINI GRAPHICAL REPRESENTATION TO ONTOS OODB DATA DEFINITION LANGUAGE

by
Xiaoyong Wang

In this thesis. we present a system to translate the semantic elements in the graphical schema language of OODINI from API of OODAL to the Type definition of ONTOS DB. To translate semantic constraints of the graphical language, we patch more information to existent class data structure in API of OODAL. After a brief review of OODINI, ONTOS DB and the existent translator without the ability to translate semantic constraints, we describe in detail the methods to translate the essential relationship. dependent relationship, multi-valued essential relationship and multi-valued dependent relationship. We employ an Inverse Reference to a "Set of" Type to achieve the goal. Setof and Tupleof relationship are special cases of the above relationships. For validating the result of the translation, we give examples of translation of a schema containing each of the relationships discussed.

TRANSLATION OF SEMANTIC ASPECTS OF OODINI GRAPHICAL REPRESENTATION TO ONTOS OODB DATA DEFINITION LANGUAGE

by
Xiaoyong Wang

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer and Information Science

Department of Computer and Information Science

October 1994

APPROVAL PAGE

TRANSLATION OF SEMANTIC ASPECTS OF OODINI GRAPHICAL REPRESENTATION TO ONTOS OODB DATA DEFINITION LANGUAGE

Xiaoyong Wang

Dr. Yehoshua Perl, Thesis Advisor — Date
Professor of Computer and Information Science, NJIT

[illegible]

Dr. James Geller, Committee Member — Date
Associate Professor of Computer and Information Science, NJIT

[illegible]

Dr. Jason T. Wang, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Xiaoyong Wang

Degree: Master of Science in Computer and Information Science

Date: October 1994

Undergraduate and Graduate Education:

- Master of Science in Computer and Information Science,
  New Jersey Institute of Technology, Newark, NJ, 1994
- Bachelor of Science in Computer Science,
  Kun Ming Institute of Technology, 1982

Major: Computer and Information Science

This thesis is dedicated to
my father Denglin Wang
and
my mother Xiaohua Zhang

ACKNOWLEDGMENT

I would like to express my most sincere gratitude to my advisor Dr. Yehoshua Perl for his moral support, academic guidance, and precious friendship throughout this work. I would like to extend my warm thanks to Dr. James Geller and Dr. Jason T. Wang for their valuable suggestion and comments on this thesis. I also would like to show special thanks to Dr. Michael Halper for his guidance and technical support.

Let me also express my thanks to Oscar Ou Yang and Aruna Kolla who give me a lot of support and help during the research. I would also like to thank my colleague Venugopal Reddy Cheruku for his cooperation.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION 1

2 GRAPHICAL SCHEMA AND OODAL IN OODINI 4

2.1 Problem Description and Approach 4

2.2 Representation of an Object Class 5

2.2.1 Representation of an Object Class in OODINI 5

2.2.2 Representation of an Object Class in OODAL 6

2.3 Access API (Application Programming Interface) 9

2.3.1 Access OODAL through API 9

2.3.2 How to Access API in C 10

2.4 Patching the Object Classes 11

2.4.1 Internal structure of API 11

2.4.2 Patch Inverse Pointer to a Class 12

2.4.3 Implementation of Patch Inverse Pointer to a class 12

2.5 Glossary 14

3 ONTOS DB AND GENERAL CONVERSION 18

3.1 Problem description and Approach 18

3.2 Introduction of the ONTOS Database System 19

3.2.1 Introduction of ONTOS 19

3.2.2 The Elements of ONTOS 19

3.2.3 Features of ONTOS 20

3.3 Activating the ONTOS DB 21

3.4 General Approach of the Conversion 22

3.4.1 Object Class definition in ONTOS DB 22

3.4.2 The Correspondence Between OODINI and ONTOS DB . . . 22

Chapter Page

3.5 Glossary 24

4 CONVERSION OF BASIC GRAPHICAL ELEMENTS 25

4.1 Problem Description and Approach 25

4.2 The Class and the Attributes 26

4.3 The Subclass and the Roleof 30

4.4 The Ordinary Relationship and the Multi-valued Relationship . . . 33

4.5 Topics Remained 37

5 CONVERSION OF SEMANTIC GRAPHICAL ELEMENTS 38

5.1 Problem Description and Approach 38

5.1.1 Problem Description 38

5.1.2 Approach Outline 40

5.2 The Essential Relationship 44

5.2.1 The Semantics of an Essential Relationship 44

5.2.2 Implementation of an Essential Relationship 45

5.2.3 Translation of an Essential Relationship 46

5.3 The Dependent Relationship 48

5.3.1 The Semantics of a Dependent Relationship 48

5.3.2 Implementation of a Dependent Relationship 48

5.3.3 Translation of a Dependent Relationship 50

5.4 The Multi-valued Essential Relationship 52

5.4.1 The Semantics of a Multi-valued Essential Relationship . . . 52

5.4.2 Implementation of a Multi-valued Essential Relationship . . 52

5.4.3 Translation of a Multi-valued Essential Relationship 54

5.5 The Multi-valued Dependent Relationship 57

5.5.1 The Semantics of a Multi-valued Dependent Relationship . . 57

5.5.2 Implementation of a Multi-valued Dependent Relationship . 58

5.5.3 Translation of a Multi-valued Dependent Relationship 59

Chapter Page

5.6 Setof 62

5.6.1 The Constraints of a Setof 62

5.6.2 Implementation of a Setof 62

5.7 Tupleof 63

5.7.1 The Constraints of a Tupleof 63

5.7.2 Implementation of a Tupleof 63

6 CONCLUSION 64

APPENDIX A EXAMPLE OF AN ESSENTIAL RELATIONSHIP 66

APPENDIX B EXAMPLE OF A DEPENDENT RELATIONSHIP 78

APPENDIX C EXAMPLE OF A MULTI-VALUED ESSENTIAL REL. 90

APPENDIX D EXAMPLE OF A MULTI-VALUED DEPENDENT REL. . . 105

REFERENCES 120

LIST OF TABLES

Table Page

3.1 Correspondence between the concepts of ONTOS and C++ 19

3.2 Correspondence between the OODINI code and the ONTOS DB 23

LIST OF FIGURES

Figure | Page

2.1 Example of the OODINI graphical schema 6

2.2 Graphical symbols used in OODINI 7

2.3 Translation of OODAL's file to ONTOS code. 10

2.4 Outline of the API's internal data structures of OODAL. 11

2.5 Addition of a patch to a class data structure. 13

3.1 Implement the class persistence in ONTOS DB 21

4.1 An example of a class and an attribute 26

4.2 An example of a subclass and a roleof 31

4.3 An example of an ordinary and a multi-valued relationship 34

5.1 An example of an essential relationship 39

5.2 An example of an Inverse Reference 40

5.3 The life time span of a Type 41

5.4 The access scope of ONTOS DB 42

5.5 The translation of essential relationship 45

5.6 The translation of a dependent relationship 49

5.7 The translation of a multi-valued essential relationship 53

5.8 The translation of a multi-valued dependent relationship 58

5.9 The translation of a Setof to a multi-valued dependent relationship 62

5.10 The translation of a Tupleof to several dependent relationships 63

CHAPTER 1

INTRODUCTION

This document discusses in detail the conversion of the OODB graphical representation schema language of the OODINI from OODAL, the OODini Abstract Language, to the ONTOS, a C++ embedded OODB, data definition language.

A graphical editor called OODINI, Object Oriented Diagrams Interface at the New jersey Institute of technology, was designed and developed successfully in the Department of Computer and information Science at New Jersey Institute of Technology[7, 6], as the need for a powerful interactive interface in the field of OODB design is arising. Using OODINI software package, an OODB designer can comfortably design and efficiently manipulate the OODB schema with powerful graphical interface at application level without concerning himself with the details of either related database or related database programming language. Then, OODINI's graphical representation of the resulting schemas can be translated automatically by OODINI software package into two abstract object oriented languages, such as OODAL and DAL, and to research prototype VML of GMD-IPSI.

But unfortunately a big gap does exist between the above mentioned abstract languages and any one of the commercial object-oriented database management systems. Although these languages have more powerful capability to describe the real world than ever, they are mainly used in abstract and theoretical level and can not be used directly by any one of the commercial OODB systems which are available in recent years. So, transferring these abstract languages into a commercial OODB system automatically becomes an urgent task . That is the purpose in this research and thesis works.

We translated the OODAL language into ONTOS DB code. OODAL is OODini Abstract Language. In [1], OODINI was converted to the VML, (VODAK MODELING LANGUAGE), designed by GMD-IPSI. Converting an OODINI graphical schema into VML or DAL code requires the understanding of the OODB Dual model architecture [4]. All object oriented database languages except VML do not adhere to this architecture. To overcome this problem, OODAL, a new and different abstract languages, was proposed[1]. In OODAL, the object class definition remains and object type definition is totally removed as compared with dual model architecture. So, the OODAL language do not depend on the dual model architecture and can be translated into commercial object-oriented database languages.

We converted the OODINI graphical schema directly from the API of OODAL but not OODAL itself. The OODAL code of a graphical schema is a plain text file obtained from the OODINI graphical representation of the schema. Translating it further into OODB code requires first to parse the whole text file. Evidently this is a redundant and tedious task. As an alternative, OODINI provided the API, Application Programming Interface. The API is provided in the form of a C library and we can invoke the library routine directly. The invocation returns a pointer pointing the first object class and then we can traverse all the classes in the schema[1].

ONTOS is one of the commercially available object-oriented database management systems and is installed in the computer system in NJIT. It is a multi-user, distributed OODB system and embedded in C++[12]. It provides all the necessary features needed for an object oriented database system. For instance, it provide type, property, procedure and reference corresponding to class, data member, member function and pointer of a class in C++, and this also corresponds to object class, attribute, method and relationship in the OODAL and the OODINI graphical schema. In addition, it provides the ability of dynamically changing not only the property but also the type and even the procedure of the OODB.

This work describes in detail the conversion of the OODINI's OODB graphical schema from the API of OODAL to the ONTOS data definition language. The purpose of the conversion is translating automatically from API into ONTOS type definition. That allows an OODB user to design schema graphically his application using OODINI, and then get completed ONTOS type definition. All the information in a graphical schema including the semantics of the relationships will be correctly defined in the ONTOS type definition. The user can concentrate then to how to input, manipulate and query the data in the OODB. Therefor he can save tremendous effort to design and implement an application in an OODB system.

This document is organized as following: Chapter 2 discuss the OODINI system and the changing of the API so that it can support the conversion of semantic aspects of a relationship. Chapter 3 discuss the ONTOS system. Chapter 4 describe the conversion of regular objects, class, attribute, regular relationship, roleof and subclass. Chapter 5 entails further the conversion of the more sophisticated semantic elements of the relationships such as. essential, dependent, setof, tupleof, multi-value essential and multi-value dependent.

CHAPTER 2

GRAPHICAL SCHEMA AND OODAL IN OODINI

This chapter proposes the problems we try to solve and the answer to these problems. We first review briefly OODINI and its graphical schema representation. Then we describe how to traverse a graphical schema through the API. Finally we discuss how to create the patch to a class so that the semantic information in a relationship can be maintained in the class. Some examples are shown in this chapter.

2.1 Problem Description and Approach

Converting the OODAL code into ONTOS type definition code will encounter many challenges. How we can traverse through each class of the graphical schema is the first problem we will meet. For regular objects, we can traverse through each class in a graphical schema and translate it directly into the corresponding ONTOS type definition code. For example, class and attribute can be directly converted into ONTOS type definition.

The second problem is that for a more sophisticated object which has specific semantics we can not simplely transfer it into ONTOS code. We need more information in the classes involved than in a regular class in order to implement the semantics between them. The API of OODAL does not have this information explicitly. For instance, suppose a class A has an essential relationship to a class B, so class A has a pointer pointed to B to present this relationship. Class B does not have a pointer to A as per the OODAL. But we need that pointer because if we want delete instance b of B, we must check to see if any a of A pointed to b of B exists. If it does, we can not delete b but give some messages. This is determined by the semantics of the essential relationship.

Traveling through all objects in a graphical schema is not so difficult. OODINI provides API to solve this problem. The API of OODAL allows us to access the internal data structure layout of OODAL. In API, all the objects in graphical schema are organized as various linked list. We can easily traverse through all the objects in a graphical schema by invoking the library routine, oodal(). We will show this in detail later in this chapter.

The second problem demonstrates a more difficulty challenge for us. In general, any class referenced by a special relationship need at least one pointer to refer back to its referee. To solve this problem we need to add more information to all the object classes in the graphical schema. One approach is to change OODINI so it can support such a special semantic application. But this will involve thorough understanding of OODINI internal structures and the changes will effect the whole OODINI system. It is very time consuming and dangerous. Another choice is to create a patch to each object class. All the information needed to be added are put into this patch, and then we attach this patch to the class.

2.2 Representation of an Object Class

2.2.1 Representation of an Object Class in OODINI

In OODINI, all the objects are edited with a graphical interface. The application problem is organized according to an object-oriented database model. The graphical interface is running under the X-window environment. It allows a user input, delete and move the objects in the graphical schema. Many kinds of relations and relationships are defined. That allows the user to choose the appropriate one according to the application. At the end of editing the schema, a user can save the graphical schema into OODAL, DAL and VML. All of them are abstract object-oriented descriptive languages. The output is a flat file or a source code of corresponding language.

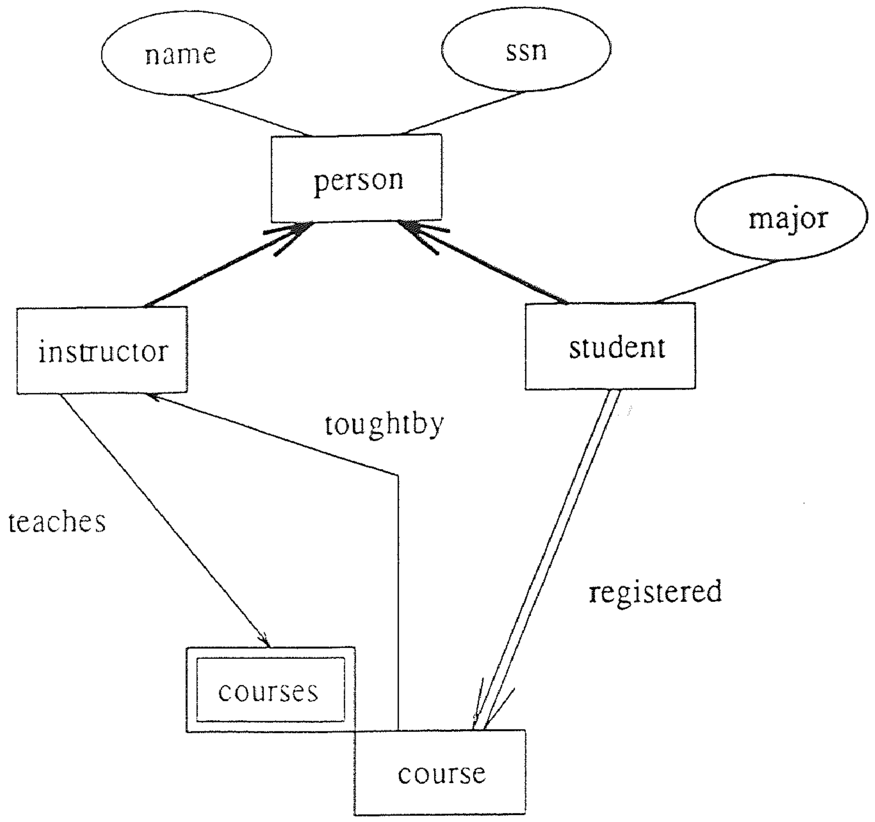

Figure 2.1 Example of the OODINI graphical schema

Figure 2.1 is an example of OODINI graphical schema. Figure 2.2 gives all the symbols used in OODINI.

2.2.2 Representation of an Object Class in OODAL

We choose OODAL as the source to translate to ONTOS code. The graphical schema in OODINI can be translated into several kinds of intermediate object-oriented language, i.e. OODAL, and DAL. DAL and VML are the languages based on the the dual model architecture. DAL is the intermediate language used to translate OODINI into VML. But ONTOS does not support this architecture. Converting DAL into ONTOS will need more unnecessary effort to understand and handle the

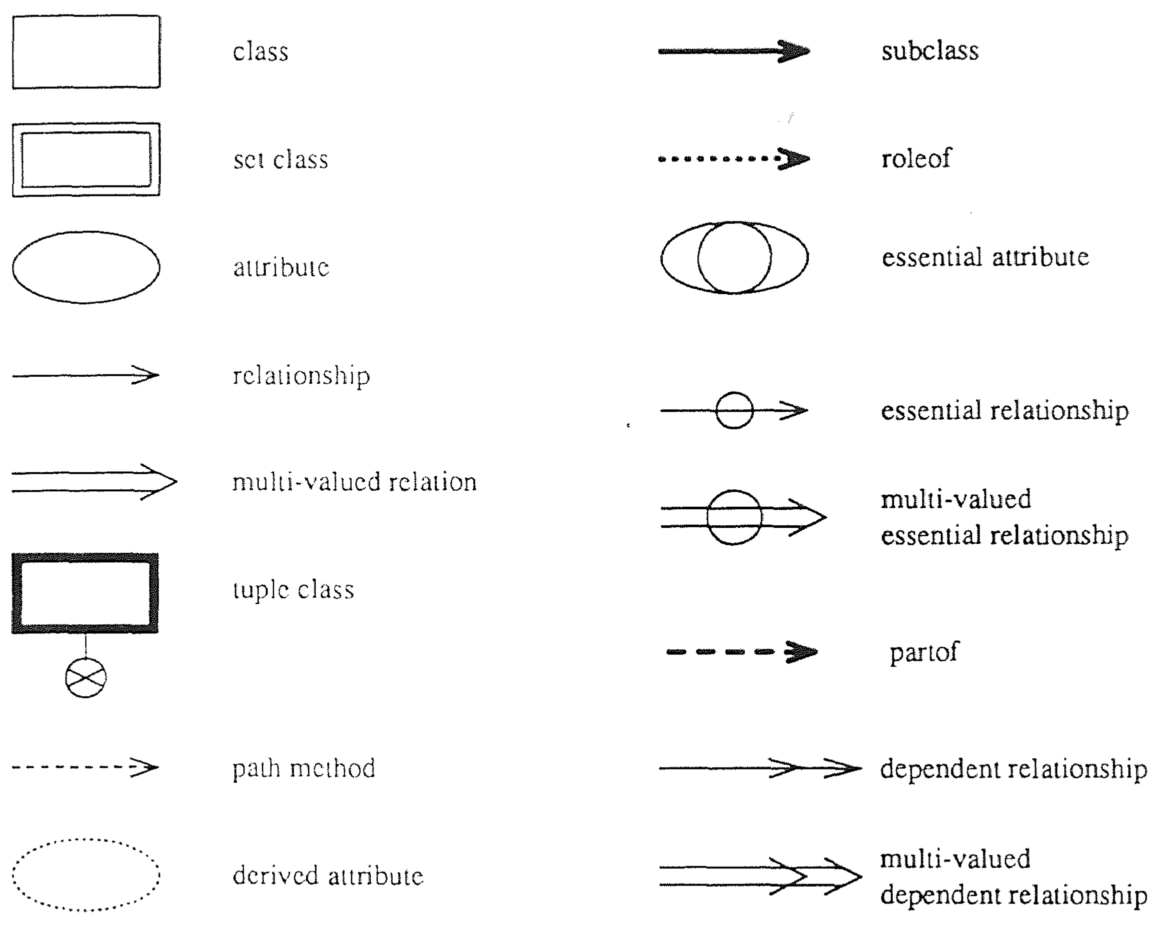

Figure 2.2 Graphical symbols used in OODINI

dual model architecture. So, OODINI supplies a new abstract language OODAL which does not rely on the dual model architecture. We will translate OODAL code into ONTOS code.

OODAL is OODini Abstract Language. OODAL's source code is a plain text file and has the following syntax template for each object class.

```
class <class-name>
     attributes
          <attribute-name> : unknown-type;
          <essential-attribute-name> :+ unknown-type;
     endattributes;
     setof : <class-name>;
     roleof : <comma separated class-name list>;
     partof : <comma separated class-name list>;
     tupleof : <comma separated <connect : class-name> list>;
     subclass : <comma separated class-name list>;
     relationships
          ordinary-relationship-name : <class-name>;
          essential-relationship-name :+ <class-name>;
          dependent-relationship-name :> <class-name>;
          multivalued-relationship-name :: <class-name>;
          multivalued-essential-relationship ::+ <class-name>;
          multivalued-dependent-relationship ::> <class-name>;
     endrelationships;
     methods
          method-name-1();
          method-name-2();
     endmethods;
end;
```

The following is the OODAL source code of the previous example in Figure 2.1 translated by OODINI.

```
class Courses
     setof : Course;
end;

class Course
     memberof : Courses;
     relationships
```

```
          Toughtby  :  Instructor;
     endrelationships;
end;

class Person
     attributes
          Ssn : unknown_type;
          Name : unknown_type;
     endattributes;
end;

class Instructor
     categoryof : Person;
     relationships
          Teaches  :  Courses;
          endrelationships;
end;

class Student
     categoryof : Person;
     attributes
          Major : unknown_type;
     endattributes;
     relationships
          Registered  ::  Course;
     endrelationships;
end;
```

2.3 Access API (Application Programming Interface)

2.3.1 Access OODAL through API

In OODINI, the OODAL can be thought of as having two parts. One is the output after translation from graphical schema. It is the OODAL source file and is a flat text file. Another is the API, Application Programming Interface, of OODAL. It is the OODAL's internal data structure. Figure 2.3 shows conceptually these relations.

The API of OODAL is a mechanism that allows a user to access the internal data structure of the graphical schema. OODAL created by OODINI is a flat source file. Translating it into ONTOS code needs parsing of the whole text file. This is

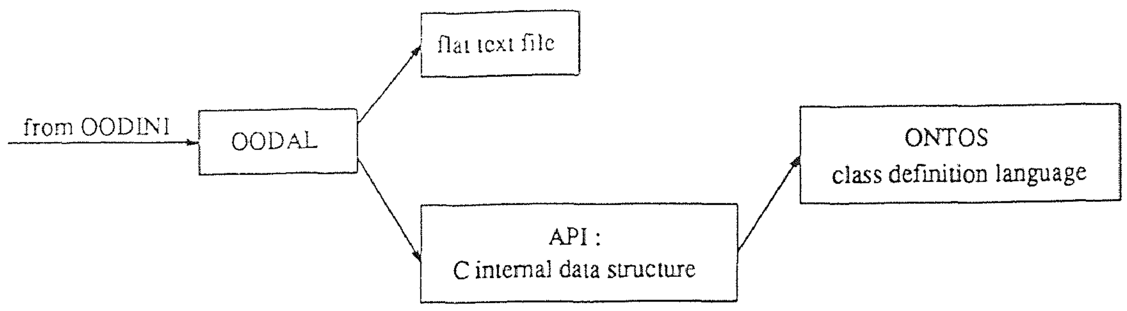

Figure 2.3 Translation of OODAL's file to ONTOS code.

very tedious and evidently unnecessary. The API of OODAL solves this problem properly. API connects all the object classes in the graphical schema into a linked list and will return a pointer to the first class when oodal() library routine is called. This gives us great convenience and efficiency to translate the OODAL code into ONTOS code.

Figure 2.4 describes the outline of API's internal data structures of OODAL for the previous example.

2.3.2 How to Access API in C

We can access API of OODAL and then get the return of the first pointer of the class in schema. Using this pointer, we can traverse through all the classes in the schema and then translate them into ONTOS code accordingly. The actual accessing of the API is as follows,

- Include "oodal.h" file in header of the access program.
- Call oodal() library routine in the program. Give heapfile name of the graphical schema, generally it is .ooheap, by default or given explicitly. The returning of the call will be the pointer pointed to the first object class.
- Link the access program with c library routine.

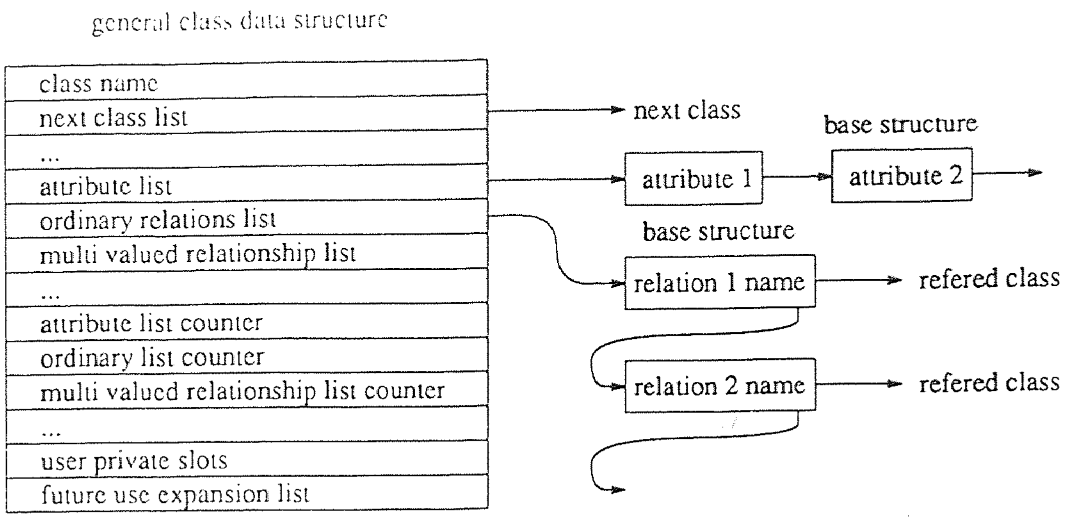

Figure 2.4 Outline of the API's internal data structures of OODAL.

2.4 Patching the Object Classes

2.4.1 Internal structure of API

In API of OODAL, we can see clear the internal data structure of an object class. We have already seen this previously in Figure 2.4. It contains all the necessary information for each class without considering the semantic relations of the class. The standard data structure of the object class is the following,

```
typedef struct oclass {
     struct oclass *next;     /* Next class pointer */
     char          *name;     /* Class name */
     ...                      /* Other definitions for class */
     ...                      /* Relations list pointer */
     ...                      /* Relations list counter */
     char     *foruser[2];    /* User private slots */
     char     *future[8];     /* Future expansion */
}oclass_t;
```

2.4.2 Patch Inverse Pointer to a Class

Without consider the semantics of the relationships between classes, the above structure is sufficient to convert the class code directly into ONTOS code. But when we consider the semantics of a relationship between two classes, we must have twin pointers pointed to each other class involved. For example, we have two classes A and B and have a semantic relation from A to B. The instance a of A should have a pointer to instance b of B. Instance b should also have a pointer to a. So that it can access its related instance when necessary.

For this purpose, we must add more information to all object classes. There are two ways to achieve the goal.

First, we can change the API data structure and add the information to it as needed. But this will involve profound understanding of OODAL translating program and internal representation of graphical schema in OODINI. In addition, any changes in OODINI will effect the whole system and should be very careful. So, this is time consuming and dangerous.

Second, we can patch an extra data structure to each class and add the information to it. In this way, we can add any information as needed and at the same time keep the API compatible with OODINI. So, this way is safer and more efficient. We choose patching to add more information. Figure 2.5 demonstrate this idea conceptually.

2.4.3 Implementation of Patch Inverse Pointer to a class

How can we patch the information to the class in API? In the data structure of the class in API, we can see pointers for future use. We use one pointer to point to a new data structure. In this new structure we define the inverse pointer. For example, class A has an essential relationship to class B. So class A has a pointer to class B. After patching, the class B also has a pointer back to A.

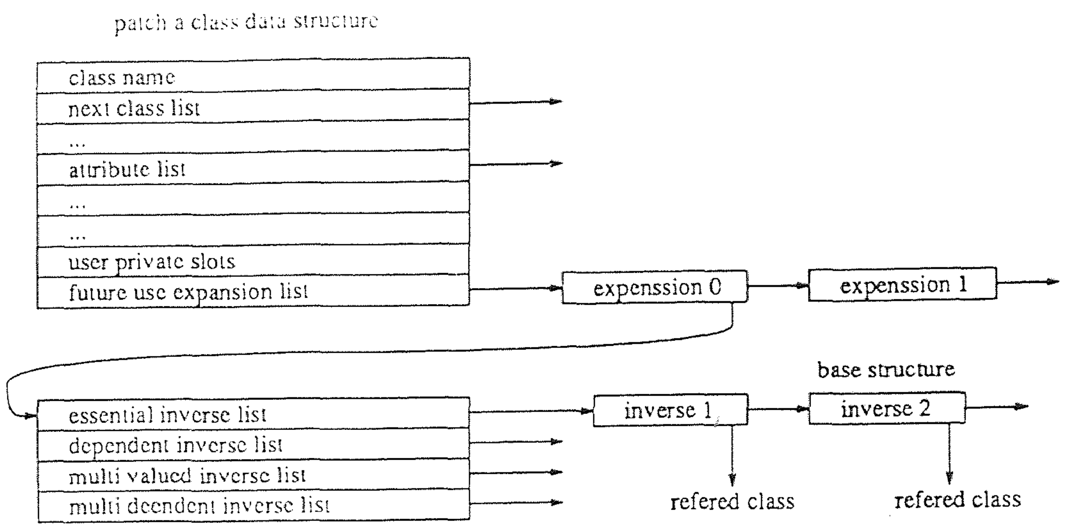

Figure 2.5 Addition of a patch to a class data structure.

We define the new structure with inverse pointers as follows,

```
/* ONTOS inverse class definition */

typedef struct iclass {

     /* essential inverse list and counter */

     basestruct_t   *esninvlist;
     long            esninvcnt;

     /* dependent inverse list and counter */

     basestruct_t   *dpninvlist;
     long            dpninvcnt;

     /* multivalue essential inverse list and counter */

     basestruct_t   *mvesninvlist;
     long            mvesninvcnt;

     /* multivalue dependent inverse list and counter */

     basestruct_t   *mvdpninvlist;
```

```
        long            mvdpninvcnt;

        /* future use list */

        char    *future[10];

    } iclass_t;
```

Then we scan the whole class linked list to attach the patch as follows.

```
    /* for each class patches inverse pointer */

    for (cp = classptr; cp; cp = CNEXT(cp)) {

        if ((ip = (iclass_t *) malloc(sizeof(iclass_t))) ==
            (iclass_t *) NULL) {
            printf("oodalontos : failed to allocate iclass_t.\n");
            exit(1);
        }

        /* patch inverse structure to a class */

        cp->future[0] = (char *) ip;
    }
```

2.5 Glossary

API:

Application Programming Interface.

Attribute :

A structural aspect of a class that is composed of a name and a data type.

BNF:

Backus-Naur Form. BNF is a metalanguage for programming languages. A metalanguage is a language that is used to describe another language. BNF is used to describe the syntax of a programming language. It uses abstractions for syntactic structures.

Category-of relationship:

A semantic relationship between two classes. It relates a specialized class to a more general class where both these classes are viewed within the same application context.

Class:

A container of objects which are similar in their structure and their semantics.

Dependent Relationship:

A relationship where the existence of an object depends on the existence of another object. If the class A has a dependent relationship to class B, then the existence of an instance a of A is dependent on the existence b of B. If b is deleted, then a must also be deleted.

Essential Attribute:

The existence of an object is conditioned on the existence of this attribute. An instance of a class can only exist if the values of its essential attributes are all different from NIL.

Essential Relationship:

A relationship which is not permitted to have a NIL value.

Member-of Relationship:

A relation between two object types. Here an object type is said to be a member of another object type. The latter object type is called a set.

Method:

A program segment with one required parameter of some object type, and any number of optional parameters. A method always returns a value of an object type or data type.

Multi Valued Relationship:

A one to many relationship between two classes. It indicates that an instance of one class can be related to any number of instances of the class to which the relationship is directed. An example of this can be the relationship between the classes "course" and "section", where a given course can have many sections.

Object:

The concept of an object is universal. Literally everything, from items as simple as the integer constant to a file handle system, memory, data structures, etc., are objects. As objects, they are treated uniformly. Objects have local memory, inherent processing ability, the capability for communicating with other objects, and the ability to inherent characteristics from ancestor objects.

Object Type:

In order to express that all instances of a class have a common structure and behavior one can consider them to be of the same abstract data type. This type is called the object type of that class.

OODAL:

OODINI Abstract Language. The graphical image for a database schema is first converted to this abstract language and then to other object oriented database languages.

OODB:

Object Oriented Data Base.

OODINI:

Object Oriented Diagram Interface at New Jersey Institute of Technology. A graphics editor for drawing and manipulating object oriented database schemas.

Part-of Relationship:

A relationship which is used to connect a part of a complex or assembled object to its integral object. An example of this relationship can be class chapter and page with the class book.

Relationship:

A user defined connection between classes that can contain either structural or semantic information in the context of the application.

Role-of Relationship:

A semantic relation between two classes. It relates a specialized class to a more general class, where both these classes are viewed in different application contexts.

Set-of Relationship:

A connection between two object types. Here an object type represents a set of other member object types. In a mathematical sense this is also a relation.

Tuple-of Relationship:

A relation constructor used to gather a group of classes(constituent classes) into a single class(the tuple class) for some purpose. A concrete example of this can be the tuple class shipment which is involved in a ternary relation with its constituent classes supplier, product and department.

VML:

The VODAK Data Modeling Language.

CHAPTER 3

ONTOS DB AND GENERAL CONVERSION

The introduction of ONTOS DB and the general approach of the translation is discussed in this chapter. ONTOS DB is a commercial object-oriented, multi user and distributed database management system, embedded in C++. After the general description of the ONTOS database system, we will see how to manipulate the ONTOS DB. Then we will discuss the general approach of translating OODINI into ONTOS DB.

3.1 Problem description and Approach

Coversion of the graphical elements in OODINI into a commercial object oriented database management system is important . Currently OODINI can transfer a graphical representation of an object oriented graphical schema into one kind of an object oriented language, namely VML . Unfortunlately, VML run on VODAK which is a research prototype but not a commercial OODB. So, an object oriented graphical schema of OODINI can not be translated into a commercial database system directly.

OOTOS DB is a commercial database management system. It is an object-oriented, multi user and distributed database management system and is embedded in C++[13]. Most of the properties and features in graphical representation in OODINI can be mapped into ONTOS DB. We will translate a graphical schema of OODINI automatically into ONTOS DB to solve the above problem.

After a brief review of ONTOS DB, we will propose a general approach of the problem in this chapter. All the graphical elements in OODINI will map into corresponding features of ONTOS DB. The actual conversion of these elements will be discussed in more detail in the coming chapters.

Table 3.1 Correspondence between the concepts of ONTOS and C++

ONTOS DB	C++
type	class
property	data member
procedure	memberfunction
super type	base class
sub type	derived class
direct reference	pointer

3.2 Introduction of the ONTOS Database System

3.2.1 Introduction of ONTOS

ONTOS is an object-oriented database management system. All the data in the database is treated as objects. The most important features of object- oriented database is that, any data or data object in the database has only one copy[3, 5]. Any user who wants to access the data item can use a pointer to access it, or we say refer to it through Reference.

ONTOS database system is embedded in C++. The ONTOS database language is totally compatible with C++ and C[16, 15]. Its data structure definition and programming statment is the same as C++ except that ONTOS has some more statments to support the special database applications. Thus all the features in C++ and C will automatically be inherited by ONTOS database system.

Table 3.1 shows the correspondence between the concepts of ONTOS and C++.

3.2.2 The Elements of ONTOS

Generally, ONTOS consists of three parts as following. Each of them performs a specific task.

- DBATOOL, is the tool which creates and maintenance the ONTOS database physical file in a working environment[14].

- Classify, is the utility to create class schema according to user supplied class definition. The data member or property and member function or procedure are created and then saved in the database.

- Cplus, is the pre-compiler which compiles a user's ONTOS program. Cplus receives a user program and produces an intermedial C++ program and then submit it to C++ to further processing. The output of the cplus is finally an executable file[11].

3.2.3 Features of ONTOS

ONTOS DB supports the ability of class to be persistent. As we know, the life time of a class in C++ will be at most the life time of the process. ONTOS extends the life time of the class, Type in ONTOS, to an unlimited time. This is the persistence of a class. The class will be stored in the database and then be activated or deactivated as needed. Figure 3.1 shows how to implement the class persistence in ONTOS DB.

Another important feature in ONTOS DB is the reference. In C++, a reference can be thought of as a pointer. ONTOS supports two kinds of reference, direct reference and abstract reference. Direct reference is a pointer which is exactly the same pointer in C++. Abstract reference can be thought as a virtual pointer to a class, no matter whether the class is residing in memory or in database. Abstract reference is more powerful, safer and more convenience to use.

ONTOS provides some useful pre-packaged classes. The most important one is the Set class in Aggregate classes. The Set class is not supported directly in C++. By using the Set class of ONTOS DB in an application, groups of objects

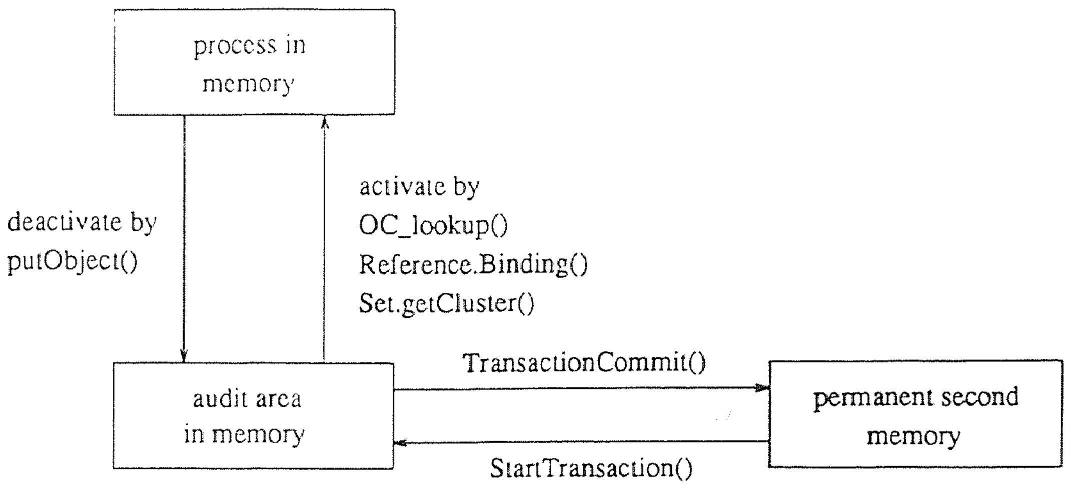

Figure 3.1 Implement the class persistence in ONTOS DB

can be manipulated as one single object. This brings a great convenience to various applications.

Other features related to the issue of multi-user, memory management and crash recovery etc. are also supported in ONTOS DB. The corresponding information can be found in References of this thesis.

3.3 Activating the ONTOS DB

Generally, we follow these steps to create an ONTOS DB and use it according to applications.

- Use DBATool utility to create an ONTOS database environment.
- Use classify procedure to create an object class schema. The classify has the following command format and one example is given also.

```
classify +X +D<dbName> -I<includeDir> +c<controlFile> \
     <headerFile>
classify +X +DontosDB +ccontrolFile -I/usr/cis/ontos/h \
     employee.h \
        department.h
```

- Use classify to create an executable process. The following is the command format and an example.

```
cplus [options] <sourceFile> -CFILE<controlFile>
cplus -g -o main -I/usr/cis/ontos/h \
     main.C \
     employee.C \
     department.C \
     -L/usr/cis/ontos/lib -Bstatic -lONTOS
```

3.4 General Approach of the Conversion

3.4.1 Object Class definition in ONTOS DB

Each object class in OODINI will be translated into ONTOS's Type definitions. In ONTOS DB, a class will be represented by two parts. One is the header definition of the Type which defines the template of that Type, including the properties and procedures. Another one is the code definition of that Type which will describe the procedure of that Type in detail.

A user program then can use them by including Type definition files of all the related Types.

The header definition file will be used by the classify utility to create a class schema. The code definition file will be used by the cplus pre-compiler and its result will produce a executable file.

3.4.2 The Correspondence Between OODINI and ONTOS DB

The OODINI elements are translated into Type definition of ONTOS DB with corresponding elements. The following table shows the relation between them.

Table 3.2 Correspondence between the OODINI code and the ONTOS DB

OODINI Elements	ONTOS Elements
Class	Type
Attribute	Property
Essential attribute	Property with control file
Ordinary relationship	Reference to a Type
Roleof	A Type with inheritance
Subclass	A Type with inheritance
Multi-valued relationship	Reference to a Set of Type
Essential relationship	Reference to a Type and Inverse Reference to a Set of Type
Dependent relationship	Reference to a Type and Inverse Reference to a Set of Type
Multi-valued Essential	Reference to a Set of Type and Inverse Reference to a Set of Type
Multi-valued Dependent	Reference to a Set of Type Inverse Reference to a Set of Type
Set	Reference to a Set of Type and Inverse Reference to a Set of Type
Tuple	Reference Types and Inverse Reference to a Set of Type
Derived attribute	Procedure with a null body
Part-of	Not implement at present
Path Method	Not implement at present

3.5 Glossary

The terminology used to describe ONTOS DB is explained in this section.

Abstract Reference:

A virtual pointer to a Type. A mechanism provided by ONTOS DB to implement Type reference.

Inheritance:

The method of defining a class in term of another class.

Persistence:

Extending the life time of a class beyond that of the process in which it is created.

Procedure:

Member functions of a class which define operations that can be performed on an instance of the class.

Property:

Data members or attributes that define the state of a class.

Reference:

Same as "Relationship". If a A class refer to another class B, it is said class A reference class B.

Sub Type:

The child class or derived class in C++.

Super Type:

The parent class or base class in C++.

CHAPTER 4

CONVERSION OF BASIC GRAPHICAL ELEMENTS

The conversion of basic graphical elements in OODINI from API of OODAL to ONTOS Type definition is discussed in this chapter. The basic graphical elements in OODINI include class, attribute, roleof, subclass, ordinary relationship and multi-valued relationship. The translation of these elements was done previously in our research group by Reddy[2]. As a background for translation of more sophisticated elements in next chapter and for better understanding of next chapter, I include and discuss briefly these translations here.

The way I discuss the translation in this chapter is the following. Given an example element in the graphical schema in OODINI, we will see the corresponding OODAL code. Then give the ONTOS DB result after translation. The result of translation is a Type definition of ONTOS DB which includes two separate files, .h and .C files and both of them are source codes of ONTOS DB. After that, any application program can include these type definitions and use it in ONTOS DB.

4.1 Problem Description and Approach

First, we will translate the basic graphical elements in a schema into ONTOS Type definition. These graphical elements include class, attribute, roleof, subclass, ordinary relationship and multi-valued relationship.

We can find the corresponding feature required directly from the ONTOS DB. The class can be mapped as a Type. The attribute of a class can be mapped as property of that Type. The roleof and subclass can use the feature of inheritance of ONTOS DB. The Ordinary relationship can be mapped as a Reference referred to a

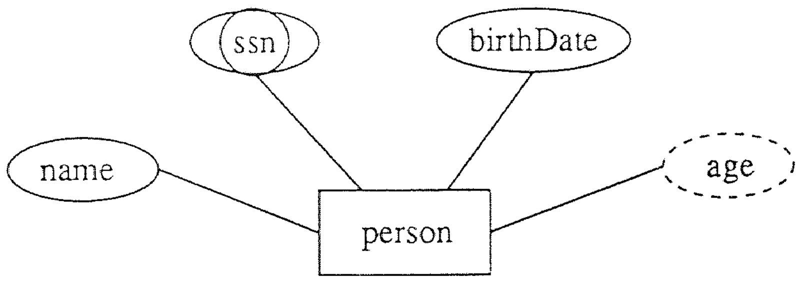

Figure 4.1 An example of a class and an attribute

Type. The Multi-valued relationship can be mapped as a Reference too, but referred to a Set of Type. We will see details in following sections.

4.2 The Class and the Attributes

The most basic graphical elements in OODINI are class and attributes. An example of class and attributes is given in Figure 4.1.

The source code of OODAL generated by OODINI is given below.

```
class   person
     attributes
          birthDate : unknown_type;
          ssn :+ unknown_type;
          name : unknown_type;
     endattributes;
     methods
          age();
     endmethods;
end;
```

The translation result of class person into Type definition of ONTOS DB is shown below.

```
// Type definition : person.h

#ifndef PERSON_H
```

```
#define PERSON_H
#include <Object.h>

class person : public Object {

private :
     unknown_type _birthDate;
     unknown_type _ssn;
     unknown_type _name;

public :
     // Constructor :

     person(unknown_type __birthDate,
          unknown_type __ssn,
          unknown_type __name,
          char*theName=(char*)0);
     person(APL*);
     virtual Type*getDirectType();
     person* person::make(unknown_type __birthDate,
          unknown_type __ssn,
          unknown_type __name,
          char*theName=(char*)0);

     // Attribute Accessors :

     void birthDate(unknown_type __birthDate);
     unknown_type birthDate();

     void ssn(unknown_type __ssn);
     unknown_type ssn();

     void name(unknown_type __name);
     unknown_type name();

     // Derived Attribute Accessories :

     unknown_type  person::age();

     // Distructor ...:

     ~person();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif



// Type definition : person.C

#include "person.h"
#include <Directory.h>
#include <Type.h>

person::person(APL *theAPL) : Object(theAPL) {
}
person::person(unknown_type __birthDate,
     unknown_type __ssn,
     unknown_type __name,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("person") );
     _birthDate = __birthDate;
     _ssn = __ssn;
     _name = __name;
}

person* person::make(unknown_type __birthDate,
     unknown_type __ssn,
     unknown_type __name,
     char* theName) :
     Object(theName)
{
     return new person(__birthDate,
          __ssn,
          __name,
          theName);
}

Type *person::getDirectType() {
     return (Type*)OC_lookup("person");
}

// Attribute Accessors :

void person::birthDate(unknown_type __birthDate) {
     _birthDate = __birthDate;
}
unknown_type person::birthDate() {
     return _birthDate;
}

void person::ssn(unknown_type __ssn) {
     _ssn = __ssn;
}
unknown_type person::ssn() {
     return _ssn;
}

void person::name(unknown_type __name) {
     _name = __name;
}
unknown_type person::name() {
     return _name;
}

// Derived Attribute Accessories :
unknown_type  person::age() {
     // fill in the code by user.
}

// Destructor ...:

person::~person() {
     Destroy(FALSE);
}
void person::Destroy(OC_Boolean aborted) {
     if (aborted) Object::Destroy(aborted);
}
void person::putObject(OC_Boolean deallocate) {
     Object::putObject(deallocate);
}
void person::deleteObject(OC_Boolean deallocate) {
     Object::deleteObject(deallocate);
}
```

As we see, the class and attributes are translated into proper format of ONTOS DB type definition. Some other translations are required by ONTOS DB.

We translate the essential relationship into two parts. First part is the same as the regular attribute. Second part is a translation of its constraint. The constraint is translated into a control file according to ONTOS. The translate result is the following.

```
// Type definition : person.ctrl

person::_ssn is required
```

The derived attribute is translated into a procedure. The procedure is correctly defined but has only a null body. The processing in the procedure will be filled by a user according to the application.

4.3 The Subclass and the Roleof

The translation of subclass and roleof is entirely according to the inheritance feature in ONTOS and the translation of subclass and roleof is similar. In OODINI, Subcalss and roleof are different relationship. A subclass object will inherit object type from its superclass in dual model. A roleof object will inherit object class from its superclass[4]. In ONTOS DB, a Type will inherit the properties and procedures from its superType. So, both subclass and roleof objects can be translated according to the inheritance and the conversion of these two type of relations are exactly the same.

Figure 4.2 gives an example of these relations.

The corresponding source code of OODAL is below.

```
class   gradStudent
end;

class   employee
end;

class   assistant
```

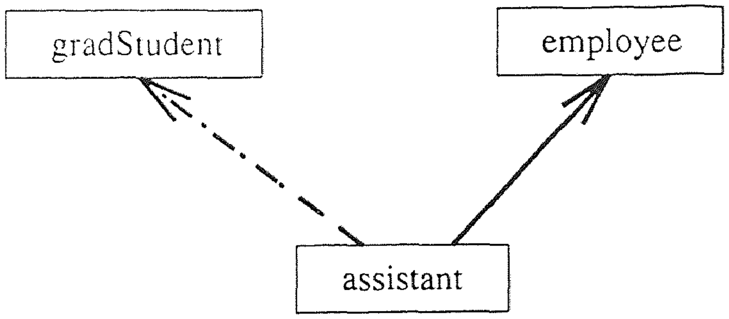

Figure 4.2 An example of a subclass and a roleof

```
        roleof : gradStudent;
        categoryof : employee;
    end;
```

The translation result of these two graphical elements of OODINI into ONTOS Type definition is given here. We can see the inheritance of assistant from gradStudent and employee clearly. As the class gradStudent and employee are totally null classes, it is unnecessary to include the translation results although they exist.

```
    // Type definition : assistant.h

    #ifndef ASSISTANT_H
    #define ASSISTANT_H
    #include "employee.h"
    #include "gradStudent.h"
    #include <Object.h>

    class assistant : public employee,
         public gradStudent{

    private :

    public :
         // Constructor :

         assistant(char* theName=(char*)0);
```

```
     assistant(APL*);
     virtual Type*getDirectType();
     assistant* assistant::make(char* theName=(char*)0);

     // Distructor ...:

     ~assistant();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif



// Type definition : assistant.C

#include "assistant.h"
#include <Directory.h>
#include <Type.h>

assistant::assistant(APL *theAPL) : employee(theAPL),
     gradStudent(theAPL) {
}
assistant::assistant(char* theName) :
     employee(theName) ,
     gradStudent(theName)
{
     initDirectType( (Type*)OC_lookup("assistant") );
}

Type *assistant::getDirectType() {
     return (Type*)OC_lookup("assistant");
}

assistant* assistant::make(char* theName)
{
     return new assistant(theName);
}

// Destructor ...:

assistant::~assistant() {
     Destroy(FALSE);
}
```

```
void assistant::Destroy(OC_Boolean aborted) {
     if (aborted) employee::Destroy(aborted);
     if (aborted) gradStudent::Destroy(aborted);
void assistant::putObject(OC_Boolean deallocate) {
     employee::putObject(deallocate);
     gradStudent::putObject(deallocate);
}
void assistant::deleteObject(OC_Boolean deallocate) {
     employee::deleteObject(deallocate);
     gradStudent::deleteObject(deallocate);
}
```

4.4 The Ordinary Relationship and the Multi-valued Relationship

The translation of ordinary relationship will use an abstract reference in ONTOS. For an ordinary relationship, a class A has a property referring to a class B. Similarly in ONTOS DB, a relationship can be implemented as an abstract reference of Type A referring to Type B.

An abstract reference can be thought of as a virtual pointer from Type A to Type B. An abstract reference will always return a correct pointer to the property whenever the data item resides in memory or in the database.

A multi-valued relationship will be mapped to an abstract reference too. But here, the reference points no longer to a Type. It points to a set of Type.

Figure 4.3 shows an example of an ordinary relationship and a multi-valued relationship.

The source code of the ordinary relationship and multi-valued relationship of OODAL is shown as follows.

```
class   gradStudent
     relationships
          advised  :  faculty;
          takes  ::  course;
     endrelationships;
end;
```

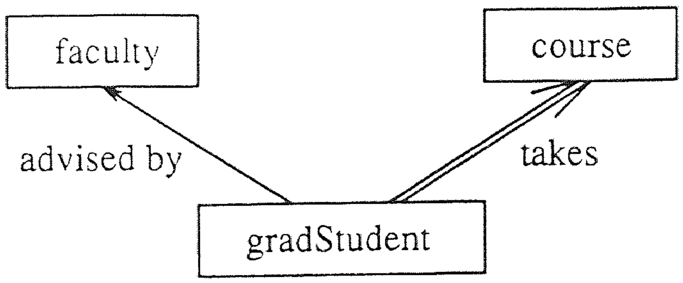

Figure 4.3 An example of an ordinary and a multi-valued relationship

```
class   course
end;

class   faculty
end;
```

The conversion result is given below only for the Type gradStudent.

```
// Type definition : gradStudent.h

#ifndef GRADSTUDENT_H
#define GRADSTUDENT_H
#include "faculty.h"
#include "course.h"
#include <Object.h>
#include <Reference.h>
#include <Set.h>

class faculty;
class course;

class gradStudent : public Object {

private :
     Reference _advised;
     Reference _takes;

public :
     // Constructor :

     gradStudent(char* theName=(char*)0);
     gradStudent(APL*);
     virtual Type*getDirectType();
     gradStudent* gradStudent:make(char* theName=(char*)0);

     // Relationship Accessors :

     void advised(faculty*__faculty);
     faculty*advised();
     void Reset_advised();

     // Multivalued Relationship Accessories :

     Set* takes();
     void Add_to_takes(course* __course);
     void Remove_from_takes(course* __course);
     long unsigned Cardinality_of_takes();

     // Distructor ...:

     ~gradStudent();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif


// Type definition : gradStudent.C

#include "gradStudent.h"
#include <Directory.h>
#include <Type.h>

gradStudent::gradStudent(APL *theAPL) : Object(theAPL) {
}
gradStudent::gradStudent(char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("gradStudent") );
     _advised.initToNull();
     Type* courseType = (Type*)OC_lookup("course");
     _takes.Init(new Set(courseType), this);
}
```

```
gradStudent* gradStudent::make(char* theName)
{
     return new gradStudent(theName);
}

Type *gradStudent::getDirectType() {
     return (Type*)OC_lookup("gradStudent");
}

// Relationship Accessors :

void gradStudent::advised(faculty*__faculty) {
     _advised.Reset(__faculty,this);
}
faculty*gradStudent::advised() {
     return (faculty*)_advised.Binding(this); }
void gradStudent::Reset_advised() {
     _advised.initToNull();
}

// Multivalued Relationship Accessories :

Set* gradStudent::takes() {
     Set* theSet = (Set*)_takes.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void gradStudent::Add_to_takes(course* __course) {
     Set* setof_course = (Set*)_takes.Binding(this);
     setof_course->Insert(__course);
     setof_course->putCluster();
}
void gradStudent::Remove_from_takes(course* __course) {
     Set* setof_course = (Set*)_takes.Binding(this);
     setof_course->Remove(__course);
     setof_course->putCluster();
}
long unsigned gradStudent::Cardinality_of_takes() {
     return ((Set*)_takes.Binding(this))->Cardinality();
}

// Destructor ...:

gradStudent::~gradStudent() {
     Destroy(FALSE);
}
void gradStudent::Destroy(OC_Boolean aborted) {
     Entity* __takes = (Entity*)_takes.Binding(this);
     delete __takes;
     if (aborted) Object::Destroy(aborted);
}
void gradStudent::putObject(OC_Boolean deallocate) {
     ((Set*)_takes.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void gradStudent::deleteObject(OC_Boolean deallocate) {
     ((Set*)_takes.Binding(this))->deleteObject(FALSE);
     Object::deleteObject(deallocate);
}
```

4.5 Topics Remained

There are still several problems in the translation of the basic elements. We will discuss them briefly as follows.

- The constraint of setof is not translated in the previous work. In the previous work, setof relationship was translated as a multi-valued relationship. But some information is lost and the constraint between the class and its set class is no longer maintained.

- The constraint of tupleof is also not translated in the previous work. In previous work, tupleof relationship was translated as ordinary relations. Here too, the constraint information is lost and the constraint between the class and its tupleof class is no longer maintained.

These problems will be discussed further in next chapter.

CHAPTER 5

CONVERSION OF SEMANTIC GRAPHICAL ELEMENTS

In this chapter, the translation of more sophisticated graphical elements in OODINI from API of OODAL to the ONTOS Type definition will be discussed. The graphical elements covered here include essential relationship, dependent relationship, multi-valued essential relationship, multi-valued dependent relationship, setof relationship and tupleof relation.

The most important feature of these graphical elements is that all of these relationships have semantic constraints on the classes. ONTOS DB does not support this semantic constraint directly. We must find some new way to implement these semantic constraints using the existing features ONTOS DB supplies. As we can see, All these relationships are converted to their proper features in ONTOS DB.

In each section, the semantics of the relationship is described. Then a corresponding feature in ONTOS DB is chosen and the implementation of the translation is disussed. Finally an example and its translation are given.

5.1 Problem Description and Approach

5.1.1 Problem Description

The most important feature of the graphical elements discussed in this chapter is that all of these relationships have a semantic constraint on their related classes. ONTOS DB does not support this semantic constraint directly. For instance, a class A has an essential relationship to a class B. An instance a of A can not be created if it refers to a null instance of B. We can not find the corresponding notion in ONTOS DB directly. So, we can not translate these relationships directly to ONTOS DB as we did in the previous chapter for the simple elements of OODINI.

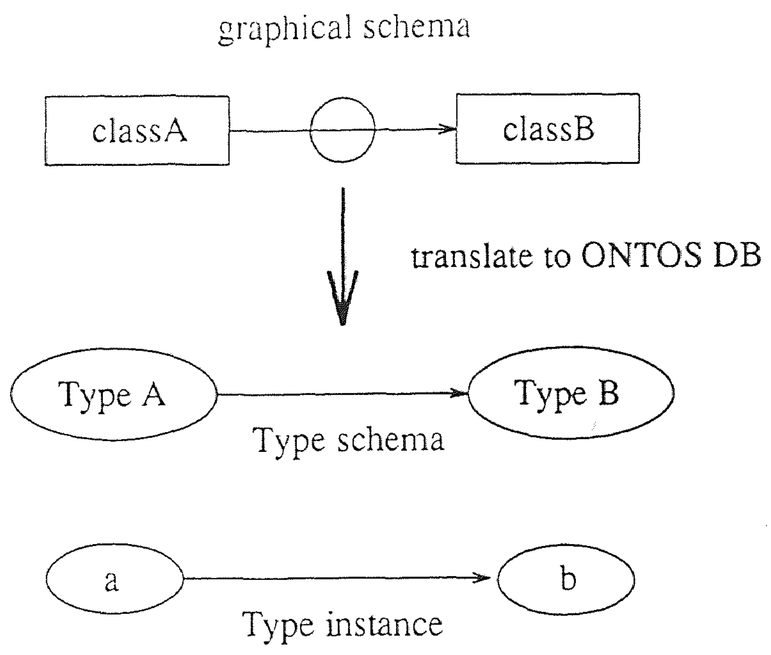

Figure 5.1 An example of an essential relationship

Suppose we have a class A referring to a class B and suppose also the relationship is essential as shown in Figure 5.1. The essential relationship specifies semantic constraints in two aspects.

- Create a: when we create an instance a of A we must check if the instance b of B referred by the relationship is exist. If it does, the creation is successful. Otherwise we can not create this instance of a.
- Delete b: if we want to delete instance b of B, we must check if there is an instance a of A referred to b of B. If there is one, we can not delete this instance b. Otherwise we can do that.

In ONTOS DB, we can check first semantics directly by checking the Reference of Type A. If the Reference of a returns a NULL we can not create a. But when we try to delete b, we don't know which a of A refers to this b. In ONTOS DB, we can

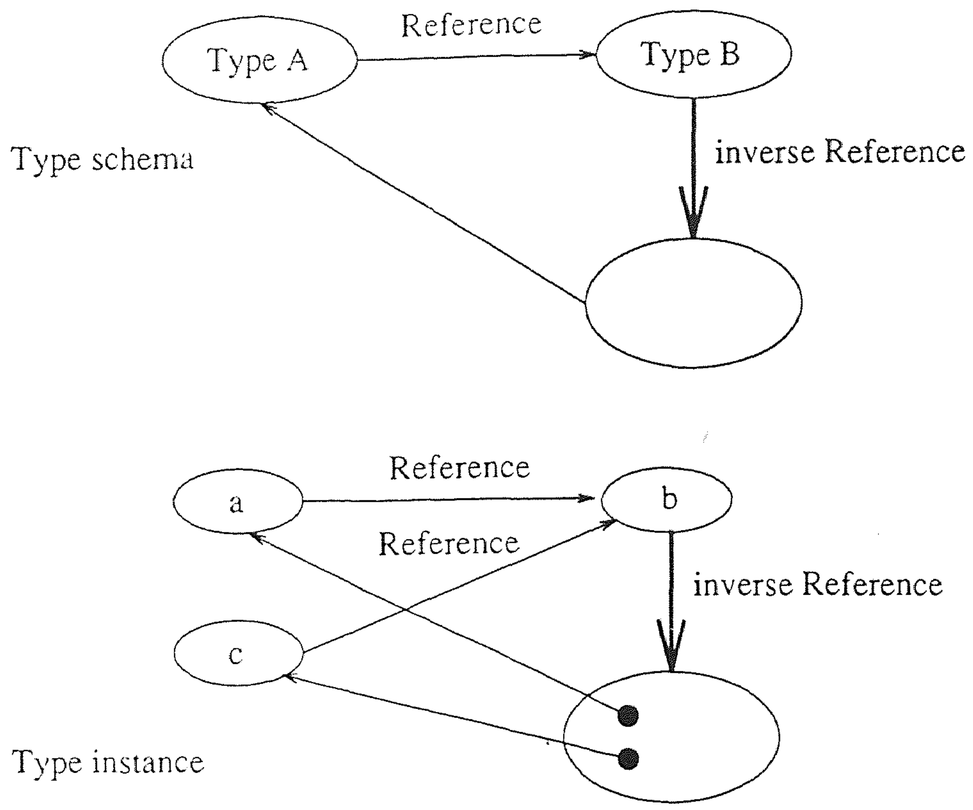

Figure 5.2 An example of an Inverse Reference

not get this information directly. We then must find some way to implement these semantic constraints using features that ONTOS DB has already supplied.

5.1.2 Approach Outline

The basic idea to solve this problem is the inverse Reference. All the semantic constraints can be supported if we create an inverse Reference to a Set of Type, from this set we can refer back to the original Type. We can see this idea in Figure 5.2. In this way, we can check the inverse Reference to see if there are any instances of A referring to b.

As we can see below, the essential relationship and the dependent relationship are mapped into a Reference to Type and an inverse Reference to a Set of Type.

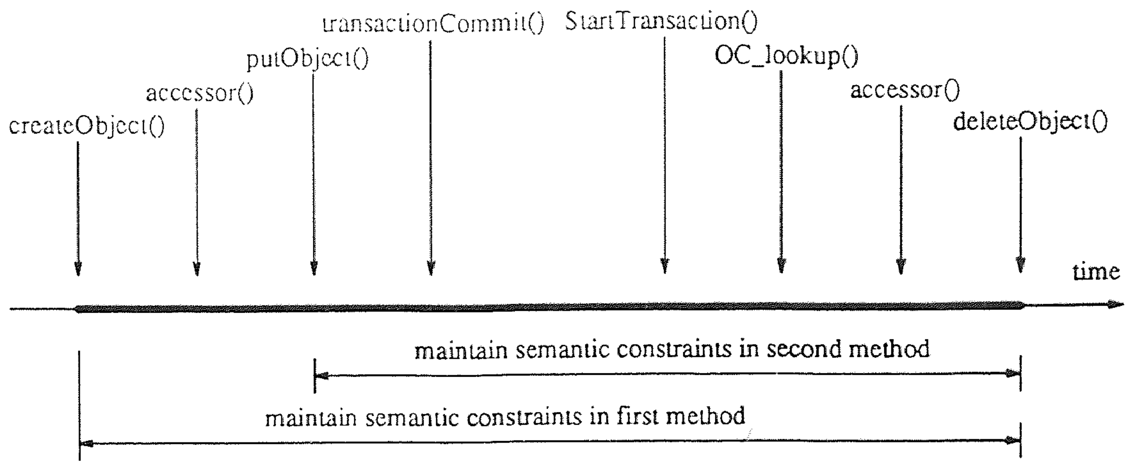

Figure 5.3 The life time span of a Type

The multi-valued essential relationship and the multi-valued dependent relationship are translated as a Reference to a Set of Type and an inverse Reference to another Set of Type. The Setof relationship can be thought of as a multi-valued dependent relationship and then can be translated into a Reference to a Set of Type. Tupleof relationship can be thought of as several dependent relationships and can be mapped into several Reference to several Type.

The important issue in the translation is the scope of the semantic constraints maintained. This issue is demonstrated in Figure 5.3. The next issue we must consider is the life time of the Type that its semantic constraints are maintained. We can see the life time of an instance of a Type clearly from Figure 5.4.

There are mainly two strategies in translation. We describe them as follows.

1. To maintain the semantic constraints at highest level, i.e. beginning at application process memory. This method has the following properties.

   - The semantic constraints are maintained everywhere including application process memory. ONTOS DB audit area and the permanent database

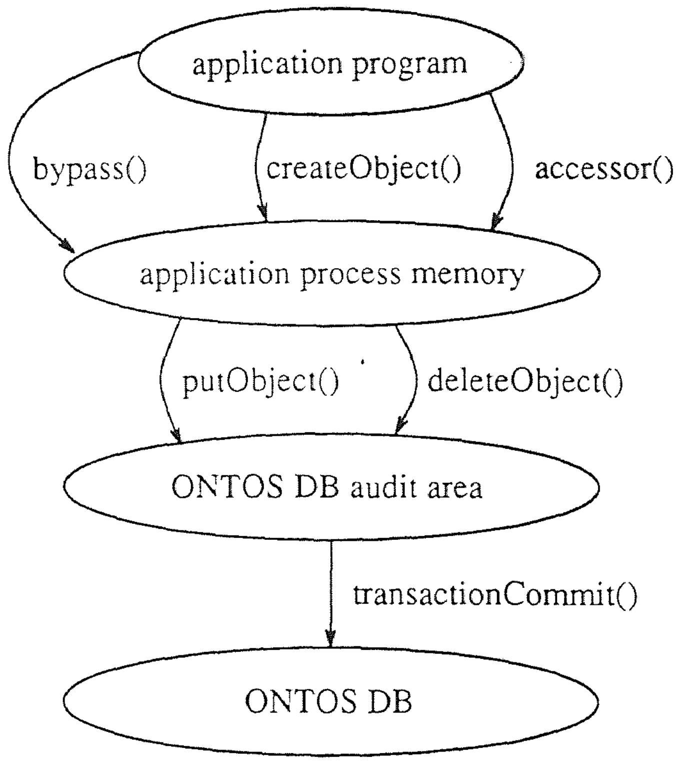

Figure 5.4 The access scope of ONTOS DB

if the application program follows the accessor supplied in the Type. We must use method make() to create an Object instead of using new operator.

- There are three control points. we will control creation of the Object using make(), accessor(), deleteObject() in order to implement the semantic constraints.
- The semantic constraints are effective on the Type during the whole life time after make() operation of the Type.
- This method requires that an application program must follow a specific format when invoking the constructor. The invoking result can be indirectly returned.
- If an application program bypasses the interface supplied, the semantic constraints of the data will be damaged. So, the semantic constraints of this data will no longer be maintained both in memory and database.

2. To maintain the semantic constraints at second level, i.e. beginning at ONTOS DB audit area. This method has the following properties.

   - The semantic constraints are maintained in ONTOS DB audit area and the permanent database. It does not matter if the application program follow the accessor supplied in the Type.
   - There are two control points. we will control putObject(), deleteObject() in order to implement the semantic constraints.
   - The semantic constraints are effective on the Type in the span beginning at putObject().

- This method require that an application program must follow a specific format when invoking the constructor. The invokation's result can be indirectly returned.
- If an application program bypasses the interface supplied, the semantic constraints of the data are still maintained.

We try to maintain the semantic constraints during the whole life time of a Type from creation to deletion of the Type. There are Direct and Indirect solutions for this issue. A Direct solution means that a Type can maintain its semantic constraints from creation until deletion of the Type, no matter whatever an application program does. An Indirect solution can not guarantee this and will ask an application program which follows a specific format to achieve that goal.

Because of the limitations of ONTOS DB and C++, we can not implement the conversion which satisfies a Direct solution. Both of above methods are Indirect solutions and have their own advantages and disadvantages. Finally, we have chosen the first strategy to implement the translation.

5.2 The Essential Relationship

5.2.1 The Semantics of an Essential Relationship

Suppose that we have a class A and class B and class A has an essential relationship to class B.

The semantic constraints in essential relationship are,

- The creation semantic constraint: if we want to create an instance a of a class A, it must refer to an existing instance b of class B.
- The update semantic constraint: we can not assign a NULL value to an instance a of class A.

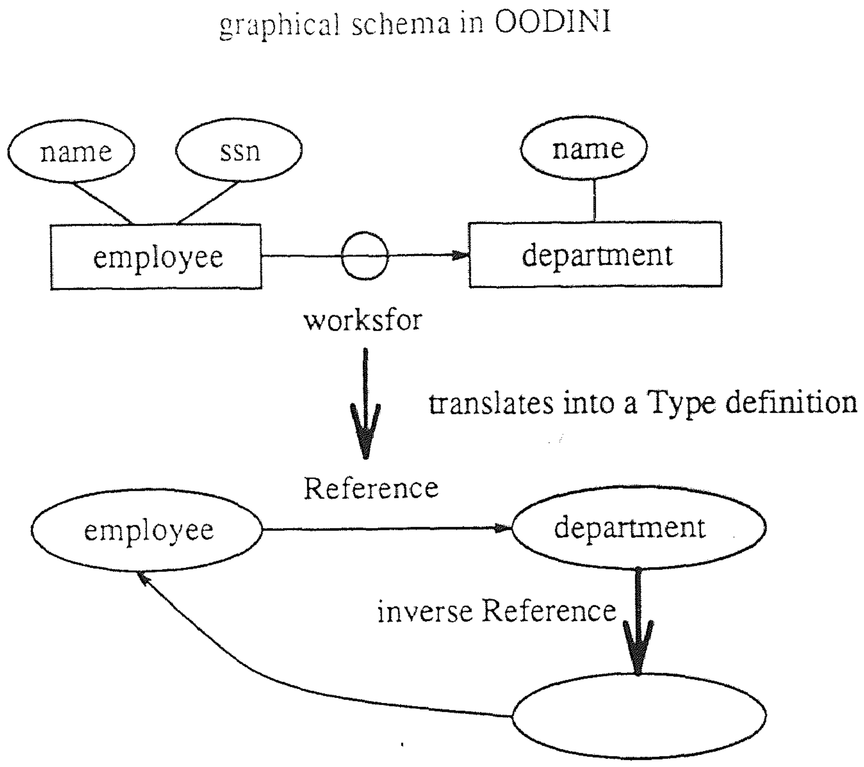

Figure 5.5 The translation of essential relationship

- The deletion semantic constraint: we can not delete an instance b of class B if there is any instance a of A referring to it.

In Figure 5.5, we can see an example of an essential relationship.

5.2.2 Implementation of an Essential Relationship

First we consider the essential relationship as an ordinary relationship. Then we add more content to implement the essential constraints. We translate the essential relationship using a Reference to department Type from employee Type and an inverse Reference to a Set of employee Type from department Type.

For employee Type, we implement the semantic constraints in the following points. In the constructor, we implement the creation of the semantic constraints.

In the relationship accessor, we implement the update of the semantic constraint. We maintain the semantic constraints accordingly in deleteObject() function,.

For the department Type, we implement the semantic constraints in the following points. The deletion semantic constraint is implemented in the deleteObject() function. For the inverse Reference, we implement this as a regular multi-valued relationship.

5.2.3 Translation of an Essential Relationship

Following the same essential relationship example, we give the OODAL code and its translation result of ONTOS DB Type definition.

```
class   department
      attributes
            name : unknown_type;
      endattributes;
end;

class   employee
     attributes
          ssn : unknown_type;
          name : unknown_type;
     endattributes;
     relationships
          worksfor  :+  department;
     endrelationships;
end;
```

In the constructor of employee, we implement the creation constraint.

```
// Creation constraint in constructor of employee :

employee::employee(char* __ssn,
     char* __name,
     department* __department,
     char* theName) :
     Object(theName)
{
        initDirectType( (Type*)OC_lookup("employee") );
        _ssn = __ssn;
        _name = __name;
        _worksfor.Reset(__department,this);
        __department->Add_to_inv_worksfor(this);
    }

    employee* employee::make(char* __ssn,
        char* __name,
        department* __department,
        char* theName)
    {
        if (__department == (department*)NULL)
            return NULL;
        return new employee(__ssn,
            __name,
            __department,
            theName);
    }
```

In relationship accessor of employee, we implement the update semantic constraint.

```
    // Update constraint in relationship Accessors :

    void employee::worksfor(department*__department) {
        if (__department != (department*)NULL) {
            (worksfor())->Remove_from_inv_worksfor(this);
            _worksfor.Reset(__department,this);
            __department->Add_to_inv_worksfor(this);
        }
    }
```

In deleteObject() of employee, we maintain the semantic constraint in inverse Set of Type accordingly.

```
    // Maintain the inverse Set of Type in employee :

    void employee::deleteObject(OC_Boolean deallocate) {
         (worksfor())->Remove_from_inv_worksfor(this);
         Object::deleteObject(deallocate);
    }
```

In deleteObject() of department, we implement the deletion semantic constraint.

```
// Deleting constraint in department :

void department::deleteObject(OC_Boolean deallocate) {

     if (Cardinality_of_inv_worksfor() != 0) {
          return;
     }
     Object::deleteObject(deallocate);
}
```

5.3 The Dependent Relationship

5.3.1 The Semantics of a Dependent Relationship

The semantic constraints of a dependent relationship are basicly the same as for an essential relationship except for the deletion constraint. We describe the semantic constraints of dependent relationship as follows.

- The creation semantic constraint: if we want to create an instance a of a class A, it must refer to an existing instance b of a class B.
- The update semantic constraint: we can not assign a NULL value to an instance a of A.
- The deletion semantic constraint: we delete an instance b of B and all the instance a of A referred to it.

An example of dependent relationship is given in Figure 5.6.

5.3.2 Implementation of a Dependent Relationship

A dependent relationship can be thought of as an ordinary relationship first. Then we add more information to implement the dependent constraints. We translate the

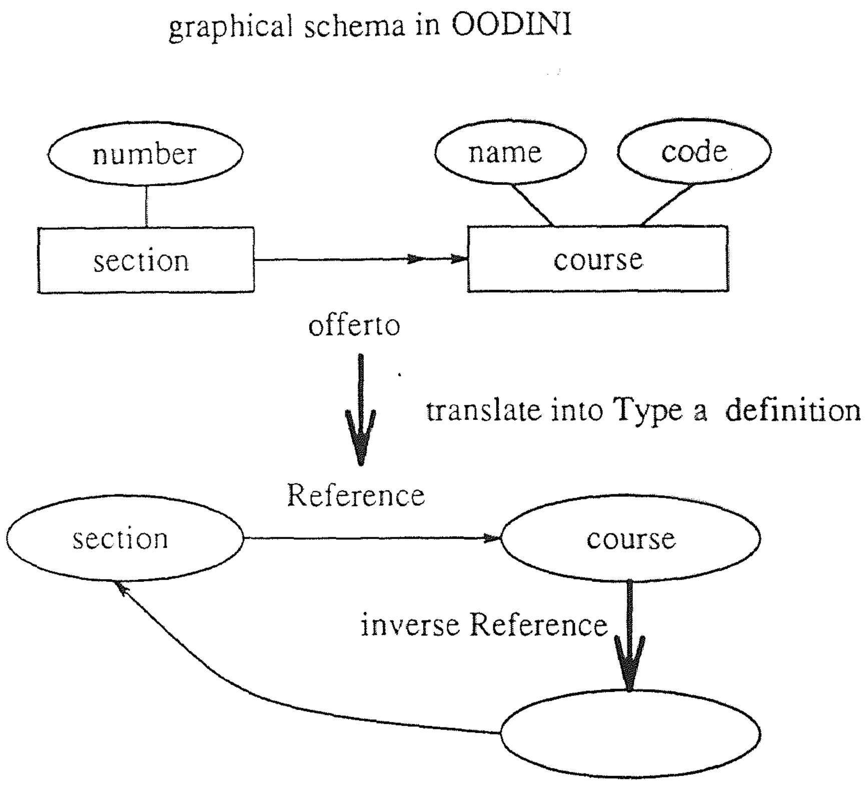

Figure 5.6 The translation of a dependent relationship

dependent relationship using a Reference to course Type from section Type and an inverse Reference to a Set of section Type from the course Type.

In section Type, we control the semantic constraints in these points. In the constructor, we implement the creation semantic constraints. In the relationship accessor, we implement the update semantic constraint. We maintain the semantic constraint accordingly in the deleteObject() function,.

For course Type, we implement the constraint as follows. The deletion semantic constraint is implemented in the deleteObject() function. For inverse Reference, we implement it as a regular multi-valued relationship.

5.3.3 Translation of a Dependent Relationship

As the given dependent example, we give the OODAL code and its translation result of ONTOS DB Type definition.

```
class   course
     attributes
          code : unknown_type;
          name : unknown_type;
     endattributes;
end;

class   section
     attributes
          number : unknown_type;
     endattributes;
     relationships
          offerto  :>  course;
     endrelationships;
end;
```

In the constructor of section, we implement the creation constraint.

```
// Creating constraint in constructor of section :

section::section(char* __number,
     course* __course,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("section") );
     _number = __number;
     _offerto.Reset(__course,this);
     __course->Add_to_inv_offerto(this);
}

section* section::make(char* __number,
     course* __course,
     char* theName)
{
     if (__course == (course*)NULL)
          return NULL;
     return new section(__number,
          __course,
          theName);
}
```

In relationship accessor of section, we implement the update semantic constraint.

```
// Updating constraint in relationship Accessors :

void section::offerto(course*__course) {
     if (__course != (course*)NULL) {
          (offerto())->Remove_from_inv_offerto(this);
          _offerto.Reset(__course,this);
          __course->Add_to_inv_offerto(this);
     }
}
```

In deleteObject() of section, we maintain the semantic constraint in inverse Set of Type accordingly.

```
// Maintain the inverse Set of Type in section :

void section::deleteObject(OC_Boolean deallocate) {
     (offerto())->Remove_from_inv_offerto(this);
     Object::deleteObject(deallocate);
}
```

In deleteObject() of department, we implement the deletion semantic constraint. The deleteCluster() will delete the Set and all the members in the Set.

```
// Deleting constraint in course :

void course::deleteObject(OC_Boolean deallocate) {
     ((Set*)_inv_offerto.Binding(this))->deleteCluster();
     Object::deleteObject(deallocate);
}
```

5.4 The Multi-valued Essential Relationship

5.4.1 The Semantics of a Multi-valued Essential Relationship

The semantic constraints of multi-valued essential relationship are,

- The creation semantic constraint: if we want to create an instance a of a class A, it must refer to existing instances of a class B.
- The update semantic constraint: we can not assign a NULL value to an instance a of A. But we can assign an a many bs.
- The deletion semantic constraint: we can not delete an instance b of B if there is any instance a of A which has only one reference to the instance b of B.

An example of multi-valued essential relationship is given in Figure 5.7. In this example, an employee can work in many departments instead of work only in one department in previous example of essential relationship.

5.4.2 Implementation of a Multi-valued Essential Relationship

The multi-valued essential relationship is basicly a multi-valued relationship. We translate that as a multi-valued relationship and use a Reference to a Set of department Type from a employee Type. Then some more information is added to

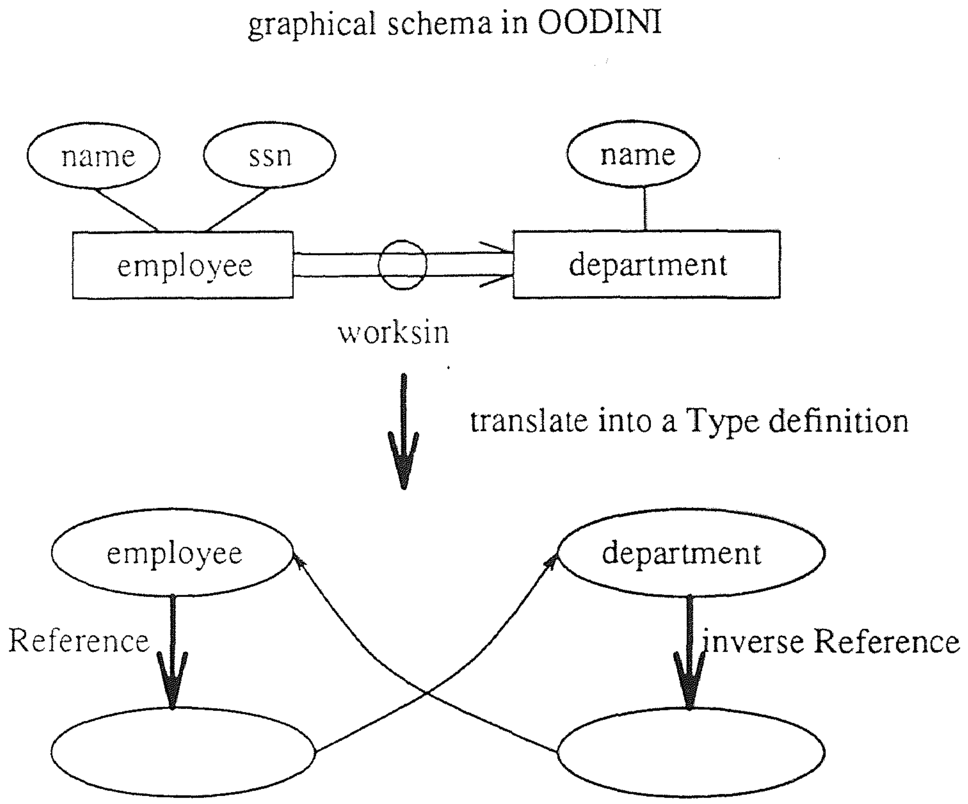

Figure 5.7 The translation of a multi-valued essential relationship

maintain the constraints imposed. Here again, we use an inverse Reference to a Set of the employee Type from department Type.

For the employee Type, we translate the semantic constraints as follows. We implement the creation constraints in the constructor of employee. In relationship accessors, we implement the update constraint. As we can see below, it is more complex than a Reference to a Type. We maintain the constraint at deleteObjeject() of the employee Type accordingly.

For the department Type, we control the semantic constraints at these points. we implement the deletion constraint in deleteObject(). For the inverse Reference, we treat it as a regular multi-valued relationship.

5.4.3 Translation of a Multi-valued Essential Relationship

As per the given example of multi-valued essential relationship, we have the OODAL and the translation result of OOTOSDB Type definition.

```
class   employee
     attributes
          ssn : unknown_type;
          name : unknown_type;
     endattributes;
     relationships
          worksin  ::+  department;
     endrelationships;
end;

class   department
     attributes
          name : unknown_type;
     endattributes;
end;
```

In the constructor of employee, the creation constraint is translated like this.

```
// Creating constraint in constructor of employee :
```

```
employee::employee(char* __ssn,
     char* __name,
     Set* __worksinSet,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("employee") );
     _ssn = __ssn;
     _name = __name;
     Type* departmentType = (Type*)OC_lookup("department");
     _worksin.Init(new Set(departmentType), this);
     Set* departmentSet = (Set*)_worksin.Binding(this);
     department* departmentIn;
     AggregateIterator* worksinIt = __worksinSet->getIterator();
     while (worksinIt -> moreData()) {
          departmentIn = (departments*)
               (Entity*)(worksinIt->operator()());
          if (departmentIn != (department*)NULL) {
               departmentIn->Add_to_inv_worksin(this);
               departmentSet->Insert(departmentIn);
          }
     }
}

employee* employee::make(char* __ssn,
     char* __name,
     Set* __worksinSet,
     char* theName)
{
     department* departmentIn;
     AggregateIterator* worksinIt = __worksinSet->getIterator();
     while (worksinIt -> moreData()) {
          departmentIn = (departments*)
               (Entity*)(worksinIt->operator()());
          if (departmentIn == (department*)NULL)
               __worksinSet->Remove(departmentIn);
     }
     if (__worksinSet->Cardinality() == 0)
          return NULL;

     return new employee(__ssn,
          __name,
          __worksinSet,
          theName);
}
```

In the relationship accessor of employee, we implement the update semantic constraint.

```
// Updating constraint in relationship Accessors :

void employee::Add_to_worksin(department* __department) {
     if (__department == (department*)NULL) return;
     Set* setof_department = (Set*)_worksin.Binding(this);
     setof_department->Insert(__department);
     setof_department->putObject();
     __department->Add_to_inv_worksin(this);
}

void employee::Remove_from_worksin(department* __department) {
     if (__department == (department*)NULL) return;
     Set* setof_department = (Set*)_worksin.Binding(this);
     if (setof_department->Cardinality() <= 1) return;
     if ((setof_department->isMember(__department)) == TRUE) {
          setof_department->Remove(__department);
          setof_department->putObject();
          __departments->Remove_from_inv_worksin(this);
     }
}
```

In the deleteObject() of employee, we maintain the semantic constraint accordingly.

```
// Maintain the inverse Set of Type in employee :

void employee::deleteObject(OC_Boolean deallocate) {
     Set* setof_department = (Set*)_worksin.Binding(this);
     department* departmentIn;
     AggregateIterator* worksinIt =
          setof_department->getIterator();
     while (worksinIt -> moreData()) {
          departmentIn = (department*)(Entity*)
               (worksinIt->operator()());
          departmentIn->Remove_from_inv_worksin(this);
     }
     Object::deleteObject(deallocate);
}
```

In the deleteObject() of department, we implement the deletion semantic constraint. The deleteCluster() will delete the Set and all the members in the Set.

```
    // Deleting constraint in department :

    employees* employeesIn;
    Set* employeesSet = (Set*)_inv_worksin.Binding(this);
    AggregateIterator* inv_worksinIt = employeesSet->getIterator();
    while (inv_worksinIt -> moreData()) {
         employeesIn = (employees*)
              (Entity*)(inv_worksinIt->operator()());
         if (employeesIn->Cardinality_of_worksin() <= 1) {
              return;
         }
    }
    inv_worksinIt = employeesSet->getIterator();
    while (inv_worksinIt -> moreData()) {
         employeesIn = (employees*)
              (Entity*)(inv_worksinIt->operator()());
         employeesIn->Remove_from_worksin(this);
    }

    Object::deleteObject(deallocate);
}
```

5.5 The Multi-valued Dependent Relationship

5.5.1 The Semantics of a Multi-valued Dependent Relationship

Like dependent relationship, the multi-valued dependent relationship is basicly the same as multi-valued essential relationship. The only difference is that an instance a of A does no longer exist if an instance b of B is deleted. The semantic constraints of multi-valued relationship is as follows.

- The creation semantic constraint: if we want to create an instance a of a class A, it must refer to an existing instance b of a class B.

- The update semantic constraint: we can not assign a NULL value to an instance a of A. But we can assign a of A more than one b of B.

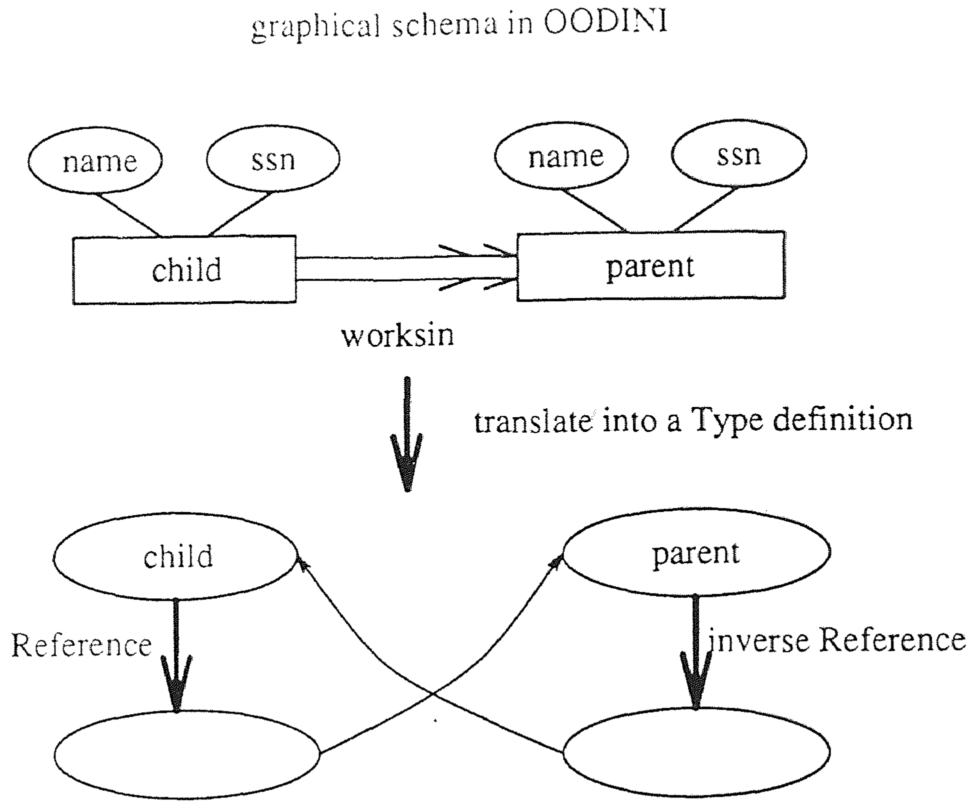

Figure 5.8 The translation of a multi-valued dependent relationship

- The deletion semantic constraint: we delete an instance b of B and all instances of a of A referred to it.

An example of multi-valued dependent relationship is given in Figure 5.8.

5.5.2 Implementation of a Multi-valued Dependent Relationship

The multi-valued dependent relationship is basicly the same as the multi -valued essential relationship. The only difference is the deletion semantic constraints. So, we use a Reference to a Set of parent Type from a child Type and use an inverse Reference to a Set of child Type to maintain the constraints.

In the child Type, we implement the constraints in the constructor, the relationship accessor and deleteObject(). In parent Type, we implement the constraints at deleteObject() and the regular multi-valued relationship.

5.5.3 Translation of a Multi-valued Dependent Relationship

For the example given above, we show the OODAL code and result after translation.

```
class   child
     attributes
          ssn : unknown_type;
          name : unknown_type;
     endattributes;
     relationships
          has  ::>  parent;
     endrelationships;
end;

class   parent
     attributes
          ssn : unknown_type;
          name : unknown_type;
     endattributes;
end;
```

In the constructor of a child type, the creation constraint is implemented as below.

```
// Creation constraint in the constructor of child type:

child::child(char* __ssn,
     char* __name,
     Set* __hasSet,
     char*theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("child") );
     _ssn = __ssn;
     _name = __name;
     Type* parentType = (Type*)OC_lookup("parent");
     _has.Init(new Set(parentType), this);
     Set* parentSet = (Set*)_has.Binding(this);
     parent* parentIn;
     AggregateIterator* hasIt = __hasSet->getIterator();
     while (hasIt -> moreData()) {
          parentIn = (parent*)
               (Entity*)(hasIt->operator()());
          if (parentIn != (parent*)NULL) {
               parentIn->Add_to_inv_has(this);
               parentSet->Insert(parentIn);
          }
     }
}

child* child::make(char* __ssn,
     char* __name,
     Set* __has,
     char* theName)
{
     parent* parentIn;
     AggregateIterator* hasIt = __has->getIterator();
     while (hasIt -> moreData()) {
          parentIn = (parent*)
               (Entity*)(hasIt->operator()());
          if (parentIn == (parent*)NULL)
               __has->Remove(parentIn);
     }
     if (__has->Cardinality() == 0)
          return NULL;
     return new child(__ssn,
          __name,
          __has,
          theName);
}
```

The update constraint is implemented in the relationship accessors as follows.

```
// Update constraint of child type:

void child::Add_to_has(parent* __parent) {
     if (__parent == (parent*)NULL) return;
     Set* setof_parent = (Set*)_has.Binding(this);
     setof_parent->Insert(__parent);
     setof_parent->putCluster();
     __parent->Add_to_inv_has(this);
}
void child::Remove_from_has(parent* __parent) {
     if (__parent == (parent*)NULL) return;
     Set* setof_parent = (Set*)_has.Binding(this);
     if (setof_parent->Cardinality() <= 1) return;
     if ((setof_parent->isMember(__parent)) == TRUE) {
          setof_parent->Remove(__parent);
          setof_parent->putCluster();
          __parent->Remove_from_inv_has(this);
     }
}
```

At the deleteObject(), we keep the constraint consistent.

```
// Maintain the consistency in the child type:

void child::deleteObject(OC_Boolean deallocate) {
     Set* setof_parent = (Set*)_has.Binding(this);
     parent* parentIn;
     AggregateIterator* hasIt =
          setof_parent->getIterator();
     while (hasIt -> moreData()) {
           parentIn = (parent*)(Entity*)
                (hasIt->operator()());
           parentIn->Remove_from_inv_has(this);
     }

     Object::deleteObject(deallocate);
}
```

In the parent Type, we implement the deletion constraint at deleteObject().

```
// Delete constraint in parent type:

void parent::deleteObject(OC_Boolean deallocate) {
     ((Set*)_inv_has.Binding(this))->deleteCluster();
     Object::deleteObject(deallocate);
}
```

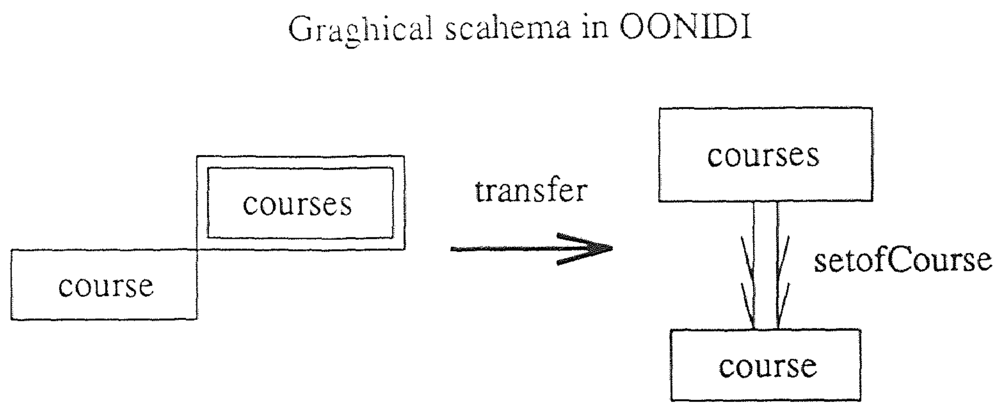

Figure 5.9 The translation of a Setof to a multi-valued dependent relationship

5.6 Setof

5.6.1 The Constraints of a Setof

The Setof relationship can be thought of as a multi-valued dependent relationship. In the previous work, the Setof relationship is translated as a regular multi-valued relationship. But Setof relationship has some constraints. If class A is the Setof class B, any members in instance a of A must be an instance of B. If we try to delete an instance b of B, we will delete the member in all the instances of A. That is exactly the multi-valued dependent relation.

So we can convert a Setof relationship into multi-valued dependent relationship as shown in Figure 5.9.

5.6.2 Implementation of a Setof

The translation of the Setof relationship is the same as the multi-valued dependent relationship. We can use the same translation result above.

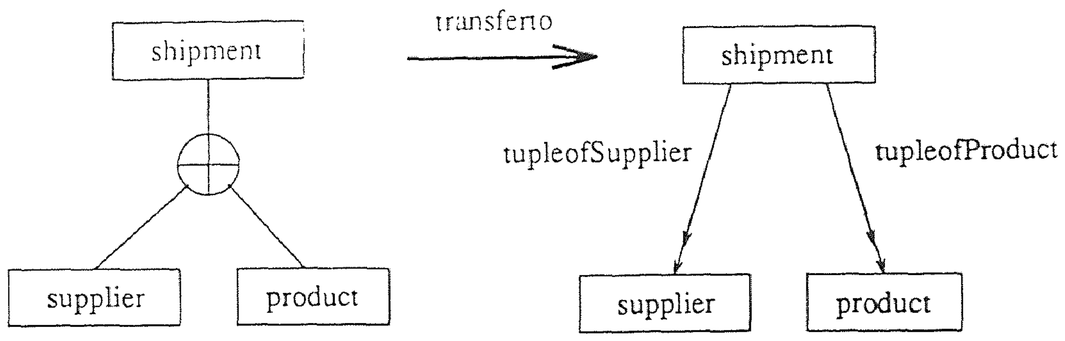

Figure 5.10 The translation of a Tupleof to several dependent relationships

5.7 Tupleof

5.7.1 The Constraints of a Tupleof

Tupleof relationship can be transferred to several dependent relationships. In the previous work, tupleof is mapped to several ordinary relationships. But tupleof also have some constraints. Suppose class A has a tupleof relationship with class B and C. The instance a of A can only be exist if it refers to an existent instances b of B and c of C. If we want to delete b of B, or c of C, the corresponding instance a in A will be deleted also. This relationship has exactly the same property as dependent relationships.

We can see this property further in the example shown in Figure 5.10.

5.7.2 Implementation of a Tupleof

According to the above mapping, the translation of tupleof relationship can be transferred to several dependent relation. We can see the translation result from previous example.

CHAPTER 6

CONCLUSION

In this thesis we enhanced the previous OODINI to ONTOS DB translator by adding the ability to translate the sophisticated graphical elements in the graphical representation schema of OODINI from API of OODAL to the Type definition of ONTOS DB. These graphical elements include essential relationship, dependent relationship, multi-valued essential relationship and multi-valued dependent relationship. Both the structure and the semantic constraints of these graphical elements are transferred correctly to the Type definition of ONTOS.

In order to implement the translation, we employ several techniques to accommodate the conversion of the semantic constraints. We patch an extra data structure to a class in API of OODAL. We employ the inverse Reference to maintain the semantic constraints in a Type. We implement the semantic constraints of the above graphical elements in the procedures of the Type involved in the translation. We have chosen the best indirect solution from several choices to implement the translation.

To validate the translation result, we give examples for each of the above graphical elements in the Appendix. Both the Type definition and the trial main program are included.

Two relationships, Setof and Tupleof, also have constraints. They are the special case of the above relationships. Setof is a multi-valued dependent relationship. Tupleof can be thought of as several dependent relationships.

We implement the translation on a Sun work station under UNIX and Motif window manager environment using C and C++.

There is still some more research to be done in the future.

- We need to translate more sophisticated graphical elements which have complex semantic constraints, such as Partof and Ownership relationships which are still under development[8, 9].

- We need to try to maintain the semantic constraints during the whole life time of a Type, and so will try to search for a Direct solution but we need to wait for the availability of the proper software changes in the ONTOS system as it is still not supported by the ONTOS system.

APPENDIX A

EXAMPLE OF AN ESSENTIAL RELATIONSHIP

```
////////////////////////// employee.h //////////////////////////

#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include "department.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>

class department;

class employee : public Object {

private :
     char* _ssn;
     char* _name;
     Reference _worksfor;

public :
     // Constructor :

     employee(char* __ssn,
          char* __name,
          department* __department,
          char* theName=(char*)0);
     employee(APL*);
     Type* getDirectType();
     static employee* make(char* __ssn,
          char* __name,
          department* __department,
          char* theName=(char*)0);

     // Attribute Accessors :

     void ssn(char* __ssn);
     char* ssn();

     void name(char* __name);
     char* name();

     // Relationship Accessors :

     void worksfor(department* __department);
     department* worksfor();
     // Distructor ...:

     ~employee();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

///////////////////////// employee.C //////////////////////////

#include "employee.h"
#include <Directory.h>
#include <Type.h>

employee::employee(APL *theAPL) : Object(theAPL) {
}
employee::employee(char* __ssn,
     char* __name,
     department* __department,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("employee") );
     _ssn = __ssn;
     _name = __name;
     _worksfor.Reset(__department,this);
     __department->Add_to_inv_worksfor(this);
}

Type *employee::getDirectType() {
     return (Type*)OC_lookup("employee");
}

employee* employee::make(char* __ssn,
     char* __name,
     department* __department,
     char* theName)
{
     if (__department == (department*)NULL)
          return NULL;
     return new employee(__ssn,
          __name,
          __department,
          theName);
}

// Attribute Accessors :

void employee::ssn(char* __ssn) {
     _ssn = __ssn;
}
char* employee::ssn() {
     return _ssn;
}

void employee::name(char* __name) {
     _name = __name;
}
char* employee::name() {
     return _name;
}

// Relationship Accessors :

void employee::worksfor(department*__department) {
     if (__department != (department*)NULL) {
          (worksfor())->Remove_from_inv_worksfor(this);
          _worksfor.Reset(__department,this);
          __department->Add_to_inv_worksfor(this);
     }
}
department*employee::worksfor() {
     return (department*)_worksfor.Binding(this);
}

// Destructor ...:

employee::~employee() {
     Destroy(FALSE);
}
void employee::Destroy(OC_Boolean aborted) {
     if (aborted) Object::Destroy(aborted);
}
void employee::putObject(OC_Boolean deallocate) {
     (worksfor())->putObject();
     Object::putObject(deallocate);
```

```
}
void employee::deleteObject(OC_Boolean deallocate) {
     (worksfor())->Remove_from_inv_worksfor(this);
     Object::deleteObject(deallocate);
}

//////////////////////// department.h ////////////////////////

#ifndef DEPARTMENT_H
#define DEPARTMENT_H
#include "employee.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>
#include <Set.h>

class employee;

class department : public Object {

     friend class employee;

private :
     char* _name;
     Reference _inv_worksfor;

     Set* inv_worksfor();
     void Add_to_inv_worksfor(employee* __employee);
     void Remove_from_inv_worksfor(employee* __employee);
     long unsigned Cardinality_of_inv_worksfor();

public :
     // Constructor :

     department(char* __name,
          char* theName=(char*)0);
     department(APL*);
     Type* getDirectType();
     static department* make(char* __name,
          char* theName=(char*)0);

     // Attribute Accessors :

     void name(char* __name);
     char* name();

     // Distructor ...:

     ~department();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

///////////////////////// department.C /////////////////////////

#include "department.h"
#include <Directory.h>
#include <Type.h>

department::department(APL *theAPL) : Object(theAPL) {
}
department::department(char* __name,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("department") );
     _name = __name;
     Type* employeeType = (Type*)OC_lookup("employee");
     _inv_worksfor.Init(new Set(employeeType), this);
}

Type *department::getDirectType() {
     return (Type*)OC_lookup("department");
}

department* department::make(char* __name,
     char* theName)
{
     return new department(__name,
          theName);
}

// Attribute Accessors :

void department::name(char* __name) {
     _name = __name;
```

```
}
char* department::name() {
     return _name;
}

// Multivalued Relationship Accessories :

Set* department::inv_worksfor() {
     Set* theSet = (Set*)_inv_worksfor.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void department::Add_to_inv_worksfor(employee* __employee) {
     Set* setof_employee = (Set*)_inv_worksfor.Binding(this);
     setof_employee->Insert(__employee);
     setof_employee->putCluster();
}
void department::Remove_from_inv_worksfor(employee* __employee) {
     Set* setof_employee = (Set*)_inv_worksfor.Binding(this);
     setof_employee->Remove(__employee);
     setof_employee->putCluster();
}
long unsigned department::Cardinality_of_inv_worksfor() {
     return ((Set*)_inv_worksfor.Binding(this))->Cardinality();
}

// Destructor ...:

department::~department() {
     Destroy(FALSE);
}
void department::Destroy(OC_Boolean aborted) {
     if (aborted) Object::Destroy(aborted);
}
void department::putObject(OC_Boolean deallocate) {
     ((Set*)_inv_worksfor.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void department::deleteObject(OC_Boolean deallocate) {
     if (Cardinality_of_inv_worksfor() != 0) {
          return;
     }
     Object::deleteObject(deallocate);
}
```

```
//////////////////////////// main.C ////////////////////////////

#include <string.h>
#include <stream.h>
#include <Database.h>
#include <Type.h>
#include <Directory.h>
#include <Exception.h>
#include "department.h"
#include "employee.h"

void createDepartment();
void deleteDepartment();
void printDepartment();
void createEmployee();
void deleteEmployee();
void printEmployee();
void changeEmployee();

main()
{
     OC_open("ontosDB1");

     char choice;
     while (1) {

     cout << "\nTesting Semantic Means of Essential Relationship\n\n";

     cout << "        1.  Create a Department\n";
     cout << "        2.  Delete a Department\n";
     cout << "        3.  Print Departments\n\n";
     cout << "        4.  Create an Employee\n";
     cout << "        5.  Delete an Employee\n";
     cout << "        6.  Print Employees\n\n";
     cout << "        7.  Change Employee's Department\n\n";

     cout << "        q.  Exit.\n\n";

     cout << "Please enter a choice : ";
     cin >> choice;

     switch (choice) {
          case '1':
                OC_transactionStart();
                createDepartment();
                OC_transactionCommit();
                break;
           case '2':
                OC_transactionStart();
                deleteDepartment();
                OC_transactionCommit();
                break;
           case '3':
                OC_transactionStart();
                printDepartment();
                OC_transactionCommit();
                break;
           case '4':
                OC_transactionStart();
                createEmployee();
                OC_transactionCommit();
                break;
           case '5':
                OC_transactionStart();
                deleteEmployee();
                OC_transactionCommit();
                break;
           case '6':
                OC_transactionStart();
                printEmployee();
                OC_transactionCommit();
                break;
           case '7':
                OC_transactionStart();
                changeEmployee();
                OC_transactionCommit();
                break;
      }
      if (choice == 'q') break;
      else cout << "\n\n\n";

      }

      OC_close();
}

void createDepartment()
```

```
{
     char name[40];

     cout << "Enter name of Department : ";
     cin >> name;

     department* d = (department*) OC_lookup(name);
     if (d == (department*)NULL) {
          d = department::make(name, name);
          if (d == NULL)
               cout << "\nCreation failed, name : " << name <<"\n";
          else
               d->putObject();
     } else {
          cout << "Department " << d->name() << " already exists.\n";
     }
}

void deleteDepartment()
{
     char name[40];

     cout << "Enter department to be deleted : ";
     cin >> name;
     department* d = (department*) OC_lookup(name);
     if (d == (department*)NULL) {
          cout << "No department " << name << " exists.\n";
     } else {
          d->deleteObject();
     }
}

void printDepartment()
{
     InstanceIterator dIt((Type*)OC_lookup("department"));
     department* d;
     int i = 1;

     while (dIt.moreData()) {
          d = (department*)(Entity*) dIt();
          cout << "Department " << i << " : " << d->name() << "\n";
          i++;
     }
     if (i == 1) {
          cout << "No department exists !\n";
     }
}

void createEmployee()
{

     char name[40];
     cout << "Enter name of employee : ";
     cin >> name;

     char ssn[40];
     cout << "Enter ssn of employee : ";
     cin >> ssn;

     char dpmt[40];
     cout << "Enter department of employee : ";
     cin >> dpmt;

     employee* e = (employee*) OC_lookup(name);
     if (e == (employee*)NULL) {
          department* d = (department*) OC_lookup(dpmt);
               employee* e = employee::make(ssn, name, d, name);
          if ( e == NULL )
               cout << "\nCreation failed, name : " << name <<"\n";
          else
               e->putObject();
     } else {
          cout << "Employee " << e->name() << " already exists.\n";
     }
}

void deleteEmployee()
{
     char name[40];

     cout << "Enter employee to be deleted : ";
     cin >> name;
     employee* e = (employee*) OC_lookup(name);
     if (e == (employee*)NULL) {
          cout << "No employee " << name << " exists.\n";
     } else {
          e->deleteObject();
     }
```

```
}

void printEmployee()
{
      InstanceIterator eIt((Type*)OC_lookup("employee"));
      employee* e;
      int i = 1;

      while (eIt.moreData()) {
            e = (employee*)(Entity*) eIt();
            cout << "Employee " << i << " name : " << e->name() << "\n";
            cout << "Employee " << i << " ssn  : " << e->ssn() << "\n";
            cout << "Employee " << i << " dpmt : "
                  << (e->worksfor())->name() << "\n";
            i++;
      }
      if (i == 1) {
            cout << "No employee exists !\n";
      }
}

void changeEmployee()
{

      char name[40];
      cout << "Enter name of employee : ";
      cin >> name;

      char dpmt[40];
      cout << "Change department of employee : ";
      cin >> dpmt;

      employee* e = (employee*) OC_lookup(name);
      department* d = (department*) OC_lookup(dpmt);
      if (e == (employee*)NULL) {
            cout << "Employee " << name << " does not exists.\n";
      } else {
            if (d == (department*)NULL) {
                  cout << "Department " << dpmt << " does not exists.\n";
            }
            e->worksfor(d);
      }
}
```

APPENDIX B

EXAMPLE OF A DEPENDENT RELATIONSHIP

```
///////////////////////// section.h /////////////////////////

#ifndef SECTION_H
#define SECTION_H
#include "course.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>

class course;

class section : public Object {

private :
     char* _number;
     Reference _offerto;

public :
     // Constructor :

     section(char* __number,
          course* __course,
          char* theName=(char*)0);
     section(APL*);
     Type* getDirectType();
     static section* make(char* __number,
          course* __course,
          char* theName=(char*)0);

     // Attribute Accessors :

     void number(char* __number);
     char* number();

     // Relationship Accessors :

     void offerto(course* __course);
     course* offerto();

     // Distructor ...:

     ~section();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

//////////////////////// section.C ////////////////////////

#include "section.h"
#include <Directory.h>
#include <Type.h>

section::section(APL *theAPL) : Object(theAPL) {
}
section::section(char* __number,
     course* __course,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("section") );
     _number = __number;
     _offerto.Reset(__course,this);
     __course->Add_to_inv_offerto(this);
}

Type *section::getDirectType() {
     return (Type*)OC_lookup("section");
}

section* section::make(char* __number,
     course* __course,
     char* theName)
{
     if (__course == (course*)NULL)
          return NULL;
     return new section(__number,
          __course,
          theName);
}

// Attribute Accessors :

void section::number(char* __number) {
     _number = __number;
}
char* section::number() {
     return _number;
}

// Relationship Accessors :

void section::offerto(course*__course) {
     if (__course != (course*)NULL) {
          (offerto())->Remove_from_inv_offerto(this);
          _offerto.Reset(__course,this);
          __course->Add_to_inv_offerto(this);
     }
}
course*section::offerto() {
     return (course*)_offerto.Binding(this);
}

// Destructor ...:

section::~section() {
     Destroy(FALSE);
}
void section::Destroy(OC_Boolean aborted) {
     if (aborted) Object::Destroy(aborted);
}
void section::putObject(OC_Boolean deallocate) {
     (offerto())->putObject();
     Object::putObject(deallocate);
}
void section::deleteObject(OC_Boolean deallocate) {
     (offerto())->Remove_from_inv_offerto(this);
     Object::deleteObject(deallocate);
}

//////////////////////// course.h /////////////////////////

#ifndef COURSE_H
#define COURSE_H
#include "section.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>
#include <Set.h>

class section;
class course : public Object {

     friend class section;

private :
     char* _code;
     char* _name;
     Reference _inv_offerto;

     Set* inv_offerto();
     void Add_to_inv_offerto(section* __section);
     void Remove_from_inv_offerto(section* __section);
     long unsigned Cardinality_of_inv_offerto();

public :
     // Constructor :

     course(char* __code,
          char* __name,
          char* theName=(char*)0);
     course(APL*);
     Type* getDirectType();
     static course* make(char* __code,
          char* __name,
          char* theName=(char*)0);

     // Attribute Accessors :

     void code(char* __code);
     char* code();

     void name(char* __name);
     char* name();

     // Distructor ...:

     ~course();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif
```

```
//////////////////////// course.C /////////////////////////

#include "course.h"
#include <Directory.h>
#include <Type.h>

course::course(APL *theAPL) : Object(theAPL) {
}
course::course(char* __code,
               char* __name,
               char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("course") );
     _code = __code;
     _name = __name;
     Type* sectionType = (Type*)OC_lookup("section");
     _inv_offerto.Init(new Set(sectionType), this);
}

Type *course::getDirectType() {
     return (Type*)OC_lookup("course");
}

course* course::make(char* __code,
     char* __name,
     char* theName)
{
     return new course(__code,
          __name,
          theName);
}

// Attribute Accessors :

void course::code(char* __code) {
     _code = __code;
}
char* course::code() {
     return _code;
}

void course::name(char* __name) {
     _name = __name;
}
char* course::name() {
     return _name;
}

// Multivalued Relationship Accessories :

Set* course::inv_offerto() {
     Set* theSet = (Set*)_inv_offerto.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void course::Add_to_inv_offerto(section* __section) {
     Set* setof_section = (Set*)_inv_offerto.Binding(this);
     setof_section->Insert(__section);
     setof_section->putCluster();
}
void course::Remove_from_inv_offerto(section* __section) {
     Set* setof_section = (Set*)_inv_offerto.Binding(this);
     setof_section->Remove(__section);
     setof_section->putCluster();
}
long unsigned course::Cardinality_of_inv_offerto() {
     return ((Set*)_inv_offerto.Binding(this))->Cardinality();
}

// Destructor ...:

course::~course() {
     Destroy(FALSE);
}
void course::Destroy(OC_Boolean aborted) {
     if (aborted) Object::Destroy(aborted);
}
void course::putObject(OC_Boolean deallocate) {
     ((Set*)_inv_offerto.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void course::deleteObject(OC_Boolean deallocate) {
     ((Set*)_inv_offerto.Binding(this))->deleteCluster();
     Object::deleteObject(deallocate);
}

///////////////////////// main.C ///////////////////////////
```

```
#include <string.h>
#include <stream.h>
#include <Database.h>
#include <Type.h>
#include <Directory.h>
#include "course.h"
#include "section.h"

void createCourse();
void deleteCourse();
void printCourse();
void createSection();
void deleteSection();
void printSection();
void changeSection();

main()
{
     OC_open("ontosDB2");

     char choice;
     while (1) {

     cout << "\nTesting Semantic Means of Dependent Relationship\n\n";
     cout << "        1.  Create a Course\n";
     cout << "        2.  Delete a Course\n";
     cout << "        3.  Print Courses\n\n";
     cout << "        4.  Create a Section\n";
     cout << "        5.  Delete a Section\n";
     cout << "        6.  Print Sections\n\n";
     cout << "        7.  Change Section\n\n";
     cout << "        q.  Exit.\n\n";

     cout << "Please enter a choice : ";
     cin >> choice;
     switch (choice) {
          case '1':
               OC_transactionStart();
               createCourse();
               OC_transactionCommit();
               break;
          case '2':
               OC_transactionStart();
```

```
                deleteCourse();
                OC_transactionCommit();
                break;
            case '3':
                OC_transactionStart();
                printCourse();
                OC_transactionCommit();
                break;
            case '4':
                OC_transactionStart();
                createSection();
                OC_transactionCommit();
                break;
            case '5':
                OC_transactionStart();
                deleteSection();
                OC_transactionCommit();
                break;
            case '6':
                OC_transactionStart();
                printSection();
                OC_transactionCommit();
                break;
            case '7':
                OC_transactionStart();
                changeSection();
                OC_transactionCommit();
                break;
        }
        if (choice == 'q') break;
        else cout << "\n\n\n";

        }
        OC_close();
}

void createCourse()
{
        char name[40];
        char code[40];

        cout << "Enter name of Course : ";
        cin >> name;
```

```
    cout << "Enter code of Course : ";
    cin >> code;

    course* c = (course*) OC_lookup(name);
    if (c == (course*)NULL) {
        c = course::make(name, code, name);
        if (c == NULL)
            cout << "\nCreation failed, name : " << name <<"\n";
        else
            c->putObject();
    } else {
        cout << "Course " << c->name() << " already exists.\n";
    }
}

void deleteCourse()
{
    char name[40];

    cout << "Enter course to be deleted : ";
    cin >> name;

    course* c = (course*) OC_lookup(name);
    if (c == (course*)NULL) {
        cout << "No course " << name << " exists.\n";
    } else {
        cout << "Course " << c->name() << " deleted.\n";
        c->deleteObject();
    }
}

void printCourse()
{
    InstanceIterator cIt((Type*)OC_lookup("course"));
    course* c;
    int i = 1;

    while (cIt.moreData()) {
        c = (course*)(Entity*) cIt();
        cout << "Course " << i << " : " << c->name() << "\n";
        i++;
    }
    if (i == 1) {
        cout << "No course exists !\n";
    }
}

void createSection()
{
    char name[40];
    cout << "Enter number of section : ";
    cin >> name;

    char cous[40];
    cout << "Enter course of section : ";
    cin >> cous;

    section* s = (section*) OC_lookup(name);
    if (s == (section*)NULL) {
        course* c = (course*) OC_lookup(cous);
        s = section::make(name, c, name);
        if (s == NULL)
            cout << "\nCreation failed, name : " << name <<"\n";
        else
            s->putObject();
    } else {
        cout << "Section " << s->number() << " already exists.\n";
    }
}

void deleteSection()
{
    char name[40];

    cout << "Enter section to be deleted : ";
    cin >> name;
    section* s = (section*) OC_lookup(name);
    if (s == (section*)NULL) {
        cout << "No section " << name << " exists.\n";
    } else {
        cout << "Section " << s->number() << " deleted.\n";
        s->deleteObject();
    }
}

void printSection()
{
    InstanceIterator sIt((Type*)OC_lookup("section"));
```

```
     section* s;
     int i = 1;

     while (sIt.moreData()) {
          s = (section*)(Entity*) sIt();
          cout << "Section " << i << " name : " << s->number() << "\n";
          cout << "Section " << i << " course : "
               << (s->offerto())->name() << "\n";
          i++;
     }
     if (i == 1) {
          cout << "No section exists !\n";
     }
}

void changeSection()
{

     char name[40];
     cout << "Enter number of section : ";
     cin >> name;

     char cous[40];
     cout << "Change course of section : ";
     cin >> cous;

     section* s = (section*) OC_lookup(name);
     course* c = (course*) OC_lookup(cous);
     if (s == (section*)NULL) {
          cout << "Section " << name << " does not exists.\n";
     } else {
          if (c == (course*)NULL) {
               cout << "Course " << cous << " does not exists.\n";
          }
          s->offerto(c);
     }
}
```

APPENDIX C

EXAMPLE OF A MULTI-VALUED ESSENTIAL REL.

```
///////////////////////// employee.h //////////////////////////

#ifndef EMPLOYEES_H
#define EMPLOYEES_H
#include "departments.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>
#include <Set.h>

class departments;

class employees : public Object {

private :
     char* _ssn;
     char* _name;
     Reference _worksin;

public :
     // Constructor :

     employees(char* __ssn,
          char* __name,
          Set* __worksin,
          char* theName=(char*)0);
     employees(APL*);
     Type* getDirectType();
     static employees* make(char* __ssn,
          char* __name,
          Set* __worksin,
          char* theName=(char*)0);

     // Attribute Accessors :

     void ssn(char* __ssn);
     char* ssn();

     void name(char* __name);
     char* name();

     // Multivalued Relationship Accessories :

     Set* worksin();
     void Add_to_worksin(departments* __departments);
     void Remove_from_worksin(departments* __departments);
     long unsigned Cardinality_of_worksin();

     // Distructor ...:

     ~employees();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

///////////////////////// employee.C //////////////////////////

#include "employees.h"
#include <Directory.h>
#include <Type.h>

employees::employees(APL *theAPL) : Object(theAPL) {
}
employees::employees(char* __ssn,
     char* __name,
     Set* __worksin,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("employees") );
     _ssn = __ssn;
     _name = __name;
     Type* departmentsType = (Type*)OC_lookup("departments");
     _worksin.Init(new Set(departmentsType), this);
     Set* departmentsSet  = (Set*)_worksin.Binding(this);
     departments* departmentsIn;
     AggregateIterator* worksinIt = __worksin->getIterator();
     while (worksinIt -> moreData()) {
          departmentsIn = (departments*)
               (Entity*)(worksinIt->operator()());
          if (departmentsIn != (departments*)NULL) {
               departmentsIn->Add_to_inv_worksin(this);
               departmentsSet->Insert(departmentsIn);
          }
     }
}
```

```
Type *employees::getDirectType() {
     return (Type*)OC_lookup("employees");
}

employees* employees::make(char* __ssn,
     char* __name,
     Set* __worksin,
     char* theName)
{
     departments* departmentsIn;
     AggregateIterator* worksinIt = __worksin->getIterator();
     while (worksinIt -> moreData()) {
          departmentsIn = (departments*)
               (Entity*)(worksinIt->operator()());
          if (departmentsIn == (departments*)NULL)
               __worksin->Remove(departmentsIn);
     }
     if (__worksin->Cardinality() == 0)
          return NULL;
     return new employees(__ssn,
          __name,
          __worksin,
          theName);
}

// Attribute Accessors :

void employees::ssn(char* __ssn) {
     _ssn = __ssn;
}
char* employees::ssn() {
     return _ssn;
}

void employees::name(char* __name) {
     _name = __name;
}
char* employees::name() {
     return _name;
}

// Multivalued Relationship Accessories :
```

```
Set* employees::worksin() {
     Set* theSet = (Set*)_worksin.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void employees::Add_to_worksin(departments* __departments) {
     if (__departments == (departments*)NULL) return;
     Set* setof_departments = (Set*)_worksin.Binding(this);
     setof_departments->Insert(__departments);
     setof_departments->putCluster();
     __departments->Add_to_inv_worksin(this);
}
void employees::Remove_from_worksin(departments* __departments) {
     if (__departments == (departments*)NULL) return;
     Set* setof_departments = (Set*)_worksin.Binding(this);
     if (setof_departments->Cardinality() <= 1) return;
     if ((setof_departments->isMember(__departments)) == TRUE) {
          setof_departments->Remove(__departments);
          setof_departments->putCluster();
          __departments->Remove_from_inv_worksin(this);
     }
}
long unsigned employees::Cardinality_of_worksin() {
     return ((Set*)_worksin.Binding(this))->Cardinality();
}

// Destructor ...:

employees::~employees() {
     Destroy(FALSE);
}
void employees::Destroy(OC_Boolean aborted) {
     Entity* __worksin = (Entity*)_worksin.Binding(this);
     delete __worksin;
     if (aborted) Object::Destroy(aborted);
}
void employees::putObject(OC_Boolean deallocate) {
     ((Set*)_worksin.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void employees::deleteObject(OC_Boolean deallocate) {
     Set* setof_departments = (Set*)_worksin.Binding(this);
     departments* departmentsIn;
     AggregateIterator* worksinIt =
          setof_departments->getIterator();
     while (worksinIt -> moreData()) {
          departmentsIn = (departments*)(Entity*)
               (worksinIt->operator()());
          departmentsIn->Remove_from_inv_worksin(this);
     }
     Object::deleteObject(deallocate);
}

//////////////////////// department.h ////////////////////////

#ifndef DEPARTMENTS_H
#define DEPARTMENTS_H
#include "employees.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>
#include <Set.h>

class employees;

class departments : public Object {

     friend class employees;

private :
     char* _name;
     Reference _inv_worksin;

     Set* inv_worksin();
     void Add_to_inv_worksin(employees* __employees);
     void Remove_from_inv_worksin(employees* __employees);
     long unsigned Cardinality_of_inv_worksin();

public :
     // Constructor :

     departments(char* __name,
          char* theName=(char*)0);
     departments(APL*);
     Type* getDirectType();
     static departments* make(char* __name,
          char* theName=(char*)0);
     // Attribute Accessors :

     void name(char* __name);
     char* name();

     // Distructor ...:

     ~departments();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

//////////////////////// department.C /////////////////////////

#include "departments.h"
#include <Directory.h>
#include <Type.h>

departments::departments(APL *theAPL) : Object(theAPL) {
}
departments::departments(char* __name,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("departments") );
     _name = __name;
     Type* employeesType = (Type*)OC_lookup("employees");
     _inv_worksin.Init(new Set(employeesType), this);
}

Type *departments::getDirectType() {
     return (Type*)OC_lookup("departments");
}

departments* departments::make(char* __name,
     char* theName)
{
     return new departments(__name,
          theName);
}

// Attribute Accessors :
```

```
void departments::name(char* __name) {
     _name = __name;
}
char* departments::name() {
     return _name;
}

// Multivalued Relationship Accessories :

Set* departments::inv_worksin() {
     Set* theSet = (Set*)_inv_worksin.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void departments::Add_to_inv_worksin(employees* __employees) {
     if (__employees == (employees*)NULL) return;
     Set* setof_employees = (Set*)_inv_worksin.Binding(this);
     setof_employees->Insert(__employees);
     setof_employees->putCluster();
}
void departments::Remove_from_inv_worksin(employees* __employees) {
     if (__employees == (employees*)NULL) return;
     Set* setof_employees = (Set*)_inv_worksin.Binding(this);
     setof_employees->Remove(__employees);
     setof_employees->putCluster();
}
long unsigned departments::Cardinality_of_inv_worksin() {
     return ((Set*)_inv_worksin.Binding(this))->Cardinality();
}

// Destructor ...:

departments::~departments() {
     Destroy(FALSE);
}
void departments::Destroy(OC_Boolean aborted) {
     if (aborted) Object::Destroy(aborted);
}
void departments::putObject(OC_Boolean deallocate) {
     ((Set*)_inv_worksin.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void departments::deleteObject(OC_Boolean deallocate) {
    employees* employeesIn;
    Set* employeesSet = (Set*)_inv_worksin.Binding(this);
    AggregateIterator* inv_worksinIt = employeesSet->getIterator();
    while (inv_worksinIt -> moreData()) {
        employeesIn = (employees*)
            (Entity*)(inv_worksinIt->operator()());
        if (employeesIn->Cardinality_of_worksin() <= 1) {
            return;
        }
    }
    inv_worksinIt = employeesSet->getIterator();
    while (inv_worksinIt -> moreData()) {
        employeesIn = (employees*)
            (Entity*)(inv_worksinIt->operator()());
        employeesIn->Remove_from_worksin(this);
    }

    Object::deleteObject(deallocate);
}

/////////////////////////// main.C /////////////////////////////

#include <string.h>
#include <stream.h>
#include <Database.h>
#include <Type.h>
#include <Directory.h>
#include "departments.h"
#include "employees.h"

void createDepartment();
void deleteDepartment();
void printDepartment();
void createEmployee();
void deleteEmployee();
void printEmployee();
void addDepartment();
void removeDepartment();

main()
{
    OC_open("ontosDB3");

    char choice;
```

```
while (1) {

cout << "\nTesting Semantic Means of Multivalue Essential Relationship\n\r
cout << "       1.  Create a Department\n";
cout << "       2.  Delete a Department\n";
cout << "       3.  Print Departments\n\n";
cout << "       4.  Create an Employee\n";
cout << "       5.  Delete an Employee\n";
cout << "       6.  Print Employees\n\n";
cout << "       7.  Add a Departments to an Employee\n";
cout << "       8.  Remove a Departments from an Employee\n\n";
cout << "       q.  Exit.\n\n";

cout << "Please enter a choice : ";
cin >> choice;
switch (choice) {
     case '1':
          OC_transactionStart();
          createDepartment();
          OC_transactionCommit();
          break;
     case '2':
          OC_transactionStart();
          deleteDepartment();
          OC_transactionCommit();
          break;
     case '3':
          OC_transactionStart();
          printDepartment();
          OC_transactionCommit();
          break;
     case '4':
          OC_transactionStart();
          createEmployee();
          OC_transactionCommit();
          break;
     case '5':
          OC_transactionStart();
          deleteEmployee();
          OC_transactionCommit();
          break;
     case '6':
          OC_transactionStart();
          printEmployee();
```

```
                OC_transactionCommit();
                break;
            case '7':
                OC_transactionStart();
                addDepartment();
                OC_transactionCommit();
                break;
            case '8':
                OC_transactionStart();
                removeDepartment();
                OC_transactionCommit();
                break;
        }
        if (choice == 'q') break;
        else cout << "\n\n\n";

        }

        OC_close();
}

void createDepartment()
{
        char name[40];

        cout << "Enter name of Department : ";
        cin >> name;
        departments* d = (departments*) OC_lookup(name);
        if (d == (departments*)NULL) {
            d = departments::make(name, name);
            if (d == NULL)
                cout << "\nCreation failed, name : " << name <<"\n";
            else
                d->putObject();
        } else {
            cout << "Department " << d->name() << " already exists.\n";
        }
}

void deleteDepartment()
{
        char name[40];

        cout << "Enter department to be deleted : ";
```

```
	cin >> name;
	departments* d = (departments*) OC_lookup(name);
	if (d == (departments*)NULL) {
		cout << "No department " << name << " exists.\n";
	} else {
		d->deleteObject();
	}
}

void printDepartment()
{
	InstanceIterator dIt((Type*)OC_lookup("departments"));
	departments* d;
	int i = 1;

	while (dIt.moreData()) {
		d = (departments*)(Entity*) dIt();
		cout << "Department " << i << " : " << d->name() << "\n";
		i++;
	}
	if (i == 1) {
		cout << "No department exists !\n";
	}
}

void createEmployee()
{
	char name[40];
	cout << "Enter name of employee : ";
	cin >> name;

	char ssn[40];
	cout << "Enter ssn of employee : ";
	cin >> ssn;

	employees* e = (employees*) OC_lookup(name);
	if (e == (employees*)NULL) {

	char dpmt[40];
	char dpmtSet[40];
	int i;
	for (i = 0; name[i]; dpmtSet[i] = name[i], i++);
	dpmtSet[i] = 'S';
	dpmtSet[i+1] = '\0';
```

```
        Set* s = new Set((Type*)OC_lookup("departments"), dpmtSet);
        while (1) {
            cout << "Enter department of employee (exit): ";
            cin >> dpmt;
            if (strcmp(dpmt, "exit") == 0) {
                break;
            }
            departments* d = (departments*) OC_lookup(dpmt);
            s->Insert(d);
        }
        e = employees::make(ssn, name, s, name);
            if ( e == NULL )
                cout << "\nCreation failed, name : " << name <<"\n";
            else
                e->putObject();
        } else {
            cout << "Employee " << e->name() << " already exists.\n";
        }
}

void deleteEmployee()
{
        char name[40];

        cout << "Enter employee to be deleted : ";
        cin >> name;
        employees* e = (employees*) OC_lookup(name);
        if (e == (employees*)NULL) {
            cout << "No employee " << name << " exists.\n";
        } else {
            e->deleteObject();
        }
}

void printEmployee()
{
        InstanceIterator eIt((Type*)OC_lookup("employees"));
        employees* e;
        int i = 1;
        AggregateIterator* worksinIt;
        departments* departmentsIn;

        for (i = 1; eIt.moreData(); i++) {
            e = (employees*)(Entity*) eIt();
```

```
          cout << "Employee " << i << " name : " << e->name() << "\n";
          cout << "Employee " << i << " ssn  : " << e->ssn() << "\n";
          worksinIt = ((Set*)e->worksin())->getIterator();
          while (worksinIt -> moreData()) {
               departmentsIn = (departments*)(Entity*)
                    (worksinIt->operator()());
               cout << "Employee " << i << " dpmt : "
                    << departmentsIn->name() << "\n";
          }
     }
     if (i == 1) {
          cout << "No employee exists !\n";
     }
}

void addDepartment()
{
     char name[40];
     char dpmt[40];

     cout << "Enter name of employee : ";
     cin >> name;

     cout << "Enter department adding : ";
     cin >> dpmt;

     employees* e = (employees*) OC_lookup(name);
     departments* d = (departments*) OC_lookup(dpmt);

     if (e == (employees*)NULL) {
          cout << "No employee " << name << " exists.\n";
     } else {
          e->Add_to_worksin(d);
     }
}

void removeDepartment()
{
     char name[40];
     char dpmt[40];

     cout << "Enter name of employee : ";
     cin >> name;
```

```
    cout << "Enter department removing : ";
    cin >> dpmt;

    employees* e = (employees*) OC_lookup(name);
    departments* d = (departments*) OC_lookup(dpmt);

    if (e == (employees*)NULL) {
         cout << "No employee " << name << " exists.\n";
    } else {
         e->Remove_from_worksin(d);
    }
}
```

APPENDIX D

EXAMPLE OF A MULTI-VALUED DEPENDENT REL.

```
////////////////////////// child.h //////////////////////////

#ifndef CHILD_H
#define CHILD_H
#include "parent.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>
#include <Set.h>

class parent;

class child : public Object {

private :
     char* _ssn;
     char* _name;
     Reference _has;

public :
     // Constructor :

     child(char* __ssn,
          char* __name,
          Set* __has,
          char* theName=(char*)0);
     child(APL*);
     Type* getDirectType();
     static child* make(char* __ssn,
          char* __name,
          Set* __has,
          char* theName=(char*)0);

     // Attribute Accessors :

     void ssn(char* __ssn);
     char* ssn();

     void name(char* __name);
     char* name();

     // Multivalued Relationship Accessories :
```

```
     Set* has();
     void Add_to_has(parent* __parent);
     void Remove_from_has(parent* __parent);
     long unsigned Cardinality_of_has();

     // Distructor ...:

     ~child();
     virtual void Destroy(OC_Boolean aborted = FALSE);
     virtual void putObject(OC_Boolean deallocate = FALSE);
     virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

////////////////////////// child.C ///////////////////////////

#include "child.h"
#include <Directory.h>
#include <Type.h>

child::child(APL *theAPL) : Object(theAPL) {
}
child::child(char* __ssn,
     char* __name,
     Set* __has,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("child") );
     _ssn = __ssn;
     _name = __name;
     Type* parentType = (Type*)OC_lookup("parent");
     _has.Init(new Set(parentType), this);
     Set* parentSet  = (Set*)_has.Binding(this);
     parent* parentIn;
     AggregateIterator* hasIt = __has->getIterator();
     while (hasIt -> moreData()) {
          parentIn = (parent*)
               (Entity*)(hasIt->operator()());
          if (parentIn != (parent*)NULL) {
               parentIn->Add_to_inv_has(this);
               parentSet->Insert(parentIn);
          }
     }
```

```
}

Type *child::getDirectType() {
     return (Type*)OC_lookup("child");
}

child* child::make(char* __ssn,
     char* __name,
     Set* __has,
     char* theName)
{
     parent* parentIn;
     AggregateIterator* hasIt = __has->getIterator();
     while (hasIt -> moreData()) {
          parentIn = (parent*)
               (Entity*)(hasIt->operator()());
          if (parentIn == (parent*)NULL)
               __has->Remove(parentIn);
     }
     if (__has->Cardinality() ==.0)
          return NULL;
     return new child(__ssn,
          __name,
          __has,
          theName);
}

// Attribute Accessors :

void child::ssn(char* __ssn) {
     _ssn = __ssn;
}
char* child::ssn() {
     return _ssn;
}

void child::name(char* __name) {
     _name = __name;
}
char* child::name() {
     return _name;
}

// Multivalued Relationship Accessories :
```

```
Set* child::has() {
     Set* theSet = (Set*)_has.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void child::Add_to_has(parent* __parent) {
     if (__parent == (parent*)NULL) return;
     Set* setof_parent = (Set*)_has.Binding(this);
     setof_parent->Insert(__parent);
     setof_parent->putCluster();
     __parent->Add_to_inv_has(this);
}
void child::Remove_from_has(parent* __parent) {
     if (__parent == (parent*)NULL) return;
     Set* setof_parent = (Set*)_has.Binding(this);
     if (setof_parent->Cardinality() <= 1) return;
     if ((setof_parent->isMember(__parent)) == TRUE) {
          setof_parent->Remove(__parent);
          setof_parent->putCluster();
          __parent->Remove_from_inv_has(this);
     }
}
long unsigned child::Cardinality_of_has() {
     return ((Set*)_has.Binding(this))->Cardinality();
}

// Destructor ...:

child::~child() {
     Destroy(FALSE);
}
void child::Destroy(OC_Boolean aborted) {
     Entity* __has = (Entity*)_has.Binding(this);
     delete __has;
     if (aborted) Object::Destroy(aborted);
}
void child::putObject(OC_Boolean deallocate) {
     ((Set*)_has.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void child::deleteObject(OC_Boolean deallocate) {
     Set* setof_parent = (Set*)_has.Binding(this);
     parent* parentIn;
     AggregateIterator* hasIt =
          setof_parent->getIterator();
     while (hasIt -> moreData()) {
          parentIn = (parent*)(Entity*)
               (hasIt->operator()());
          parentIn->Remove_from_inv_has(this);
     }
     Object::deleteObject(deallocate);
}

///////////////////////// parent.h /////////////////////////

#ifndef PARENT_H
#define PARENT_H
#include "child.h"
#include <Object.h>
#include <Reference.h>
#include <stream.h>
#include <Set.h>

class child;

class parent : public Object {

     friend class child;

private :
     char* _ssn;
     char* _name;
     Reference _inv_has;

     Set* inv_has();
     void Add_to_inv_has(child* __child);
     void Remove_from_inv_has(child* __child);
     long unsigned Cardinality_of_inv_has();

public :
     // Constructor :

     parent(char* __ssn,
          char* __name,
          char* theName=(char*)0);
     parent(APL*);
     Type* getDirectType();
```

```
      static parent* make(char* __ssn,
           char* __name,
           char* theName=(char*)0);

      // Attribute Accessors :

      void ssn(char* __ssn);
      char* ssn();

      void name(char* __name);
      char* name();

      // Distructor ...:

      ~parent();
      virtual void Destroy(OC_Boolean aborted = FALSE);
      virtual void putObject(OC_Boolean deallocate = FALSE);
      virtual void deleteObject(OC_Boolean deallocate = TRUE);
};
#endif

///////////////////////// parent.C /////////////////////////

#include "parent.h"
#include <Directory.h>
#include <Type.h>

parent::parent(APL *theAPL) : Object(theAPL) {
}
parent::parent(char* __ssn,
     char* __name,
     char* theName) :
     Object(theName)
{
     initDirectType( (Type*)OC_lookup("parent") );
     _ssn = __ssn;
     _name = __name;
     Type* childType = (Type*)OC_lookup("child");
     _inv_has.Init(new Set(childType), this);
}

Type *parent::getDirectType() {
     return (Type*)OC_lookup("parent");
}
```

```
parent* parent::make(char* __ssn,
     char* __name,
     char* theName)
{
     return new parent(__ssn,
          __name,
          theName);
}

// Attribute Accessors :

void parent::ssn(char* __ssn) {
     _ssn = __ssn;
}
char* parent::ssn() {
     return _ssn;
}

void parent::name(char* __name) {
     _name = __name;
}
char* parent::name() {
     return _name;
}

// Multivalued Relationship Accessories :

Set* parent::inv_has() {
     Set* theSet = (Set*)_inv_has.Binding(this);
     Set* returnSet = new Set(*theSet);
     return(returnSet);
}
void parent::Add_to_inv_has(child* __child) {
     if (__child == (child*)NULL) return;
     Set* setof_child = (Set*)_inv_has.Binding(this);
     setof_child->Insert(__child);
     setof_child->putCluster();
}
void parent::Remove_from_inv_has(child* __child) {
     if (__child == (child*)NULL) return;
     Set* setof_child = (Set*)_inv_has.Binding(this);
     setof_child->Remove(__child);
     setof_child->putCluster();
}
long unsigned parent::Cardinality_of_inv_has() {
     return ((Set*)_inv_has.Binding(this))->Cardinality();
}

// Destructor ...:

parent::~parent() {
     Destroy(FALSE);
}
void parent::Destroy(OC_Boolean aborted) {
     Entity* __inv_has = (Entity*)_inv_has.Binding(this);
     delete __inv_has;
     if (aborted) Object::Destroy(aborted);
}
void parent::putObject(OC_Boolean deallocate) {
     ((Set*)_inv_has.Binding(this))->putObject(FALSE);
     Object::putObject(deallocate);
}
void parent::deleteObject(OC_Boolean deallocate) {
     ((Set*)_inv_has.Binding(this))->deleteCluster();
     Object::deleteObject(deallocate);
}

////////////////////////// main.C //////////////////////////

#include <string.h>
#include <stream.h>
#include <Database.h>
#include <Type.h>
#include <Directory.h>
#include "parent.h"
#include "child.h"

void createParent();
void deleteParent();
void printParent();
void createChild();
void deleteChild();
void printChild();
void addParent();
void removeParent();

main()
```

```
{
     OC_open("ontosDB4");

     char choice;
     while (1) {

     cout << "\nTesting Semantic Means of Multivalue Dependent Relationship\n\n
     cout << "        1.  Create a Parent\n";
     cout << "        2.  Delete a Parent\n";
     cout << "        3.  Print Parents\n\n";
     cout << "        4.  Create a Child\n";
     cout << "        5.  Delete a Child\n";
     cout << "        6.  Print Children\n\n";
        cout << "        7.  Add a Parent to a Child\n";
        cout << "        8.  Remove a Parent from a Child\n\n";
     cout << "        q.  Exit.\n\n";

     cout << "Please enter a choice : ";
     cin >> choice;
     switch (choice) {
          case '1':
               OC_transactionStart();
               createParent();
               OC_transactionCommit();
               break;
          case '2':
               OC_transactionStart();
               deleteParent();
               OC_transactionCommit();
               break;
          case '3':
               OC_transactionStart();
               printParent();
               OC_transactionCommit();
               break;
          case '4':
               OC_transactionStart();
               createChild();
               OC_transactionCommit();
               break;
          case '5':
               OC_transactionStart();
               deleteChild();
               OC_transactionCommit();
```

```
                break;
            case '6':
                OC_transactionStart();
                printChild();
                OC_transactionCommit();
                break;
            case '7':
                OC_transactionStart();
                addParent();
                OC_transactionCommit();
                break;
            case '8':
                OC_transactionStart();
                removeParent();
                OC_transactionCommit();
                break;
        }
        if (choice == 'q') break;
        else cout << "\n\n\n";

        }
        OC_close();
}

void createParent()
{
        char name[40];
        cout << "Enter name of Parent : ";
        cin >> name;

        char ssn[40];
        cout << "Enter ssn of Parent : ";
        cin >> ssn;

        parent* p = (parent*) OC_lookup(name);
        if (p == (parent*)NULL) {
            p = parent::make(ssn, name, name);
            if (p == NULL)
                cout << "\nCreation failed, name : " << name <<"\n";
            else
                p->putObject();
        } else {
            cout << "Parent " << p->name() << " already exists.\n";
        }
```

```
}

void deleteParent()
{
	char name[40];
	cout << "Enter parent to be deleted : ";
	cin >> name;

	parent* p = (parent*) OC_lookup(name);
	if (p == (parent*)NULL) {
		cout << "No parent " << name << " exists.\n";
	} else {
		p->deleteObject();
	}
}

void printParent()
{
	InstanceIterator pIt((Type*)OC_lookup("parent"));

	parent* p;
	int i = 1;
	while (pIt.moreData()) {
		p = (parent*)(Entity*) pIt();
		cout << "Parent " << i << " name : " << p->name() << "\n";
		cout << "Parent " << i << " ssn  : " << p->ssn()  << "\n";
		i++;
	}
	if (i == 1) {
		cout << "No parent exists !\n";
	}
}

void createChild()
{
	char name[40];
	cout << "Enter name of child : ";
	cin >> name;

	char ssn[40];
	cout << "Enter ssn of child : ";
	cin >> ssn;

	child* c = (child*) OC_lookup(name);
```

```
    if (c == (child*)NULL) {

    char par[40];
    char parSet[40];
    int i;
    for (i = 0; name[i]; parSet[i] = name[i], i++);
    parSet[i] = 'S';
    parSet[i+1] = '\0';
    Set* s = new Set((Type*)OC_lookup("parent"), parSet);
    while (1) {
         cout << "Enter parent of child (exit): ";
         cin >> par;
         if (strcmp(par, "exit") == 0) {
              break;
         }
         parent* p = (parent*) OC_lookup(par);
         s->Insert(p);
    }
    c = child::make(ssn, name, s, name);
    if (c == NULL)
         cout << "\nCreation failed, name : " << name <<"\n";
    else
         c->putObject();

    } else {
         cout << "Child " << c->name() << " already exists.\n";
    }
}

void deleteChild()
{
    char name[40];
    cout << "Enter child to be deleted : ";
    cin >> name;

    child* c = (child*) OC_lookup(name);
    if (c == (child*)NULL) {
         cout << "No child " << name << " exists.\n";
    } else {
         cout << "Child " << c->name() << " deleted.\n";
         c->deleteObject();
    }
}
```

```
void printChild()
{
     InstanceIterator cIt((Type*)OC_lookup("child"));
     child* c;
     AggregateIterator* hasIt;
     parent* parentIn;
     int i;

     for (i = 1; cIt.moreData(); i++) {
          c = (child*)(Entity*) cIt();
          cout << "Child " << i << " name : " << c->name() << "\n";
          cout << "Child " << i << " ssn  : " << c->ssn() << "\n";
          hasIt = ((Set*)c->has())->getIterator();
          while (hasIt -> moreData()) {
               parentIn = (parent*)(Entity*)
                    (hasIt->operator()());
               cout << "Child " << i << " parent : "
                    << parentIn->name() << "\n";
          }
     }
     if (i == 1) {
          cout << "No child exists !\n";
     }
}

void addParent()
{
     char name[40];
     char par[40];

     cout << "Enter name of child : ";
     cin >> name;

     cout << "Enter paret adding : ";
     cin >> par;

     child* c = (child*) OC_lookup(name);
     parent* p = (parent*) OC_lookup(par);

     if (c == (child*)NULL) {
          cout << "No child " << name << " exists.\n";
     } else {
          c->Add_to_has(p);
     }
}

void removeParent()
{
     char name[40];
     char par[40];

     cout << "Enter name of child : ";
     cin >> name;

     cout << "Enter parent removing : ";
     cin >> par;

     child* c = (child*) OC_lookup(name);
     parent* p = (parent*) OC_lookup(par);

     if (c == (child*)NULL) {
          cout << "No child " << name << " exists.\n";
     } else {
          c->Remove_from_has(p);
     }
}
```

REFERENCES

1. S. Chatterjee, "Graphical image persistence and code generation for object oriented database," Master's thesis, New Jersey Institute of Technology, Newark, NJ., May 1992.

2. V. R. Cheruku, "Graphical image persistence and code generation for object oriented database," Master's thesis, New Jersey Institute of Technology, Newark, NJ., May 1994.

3. C. J. Date. *An Introduction to Database Systems*, Addison-Wesley Publishing Co., Inc., Reading MA., 1986.

4. J. G. E. Neuhold. Y. Perl and V. Turau, "The dual model for object oriented databases." Tech. Rep. 30, New Jersey Institute of Technology, Newark, NJ., 1991.

5. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA., 1989.

6. Y. P. M. Halper, J. Geller and E. Neuhold, "A graphical schema representation for object oriented databases," Tech. Rep. 17, New Jersey Institute of Technology, Newark, NJ., 1992.

7. Y. P. M. Halper, J. Geller and E. Neuhold, "An oodb graphical schema representation." Tech. Rep. 1, New Jersey Institute of Technology, Newark, NJ., 1992.

8. J. G. O. O. Yang, M. Halper and Y. Perl, "The oodb ownership relationship," Proceedings, OOIS'94, London, UK, Dec 1994.

9. J. G. M. Halper, J. Geller and Y. Perl, "Part Relations for Object-Oriented Databases," Tech. Rep. 18, New Jersey Institute of Technology, Newark, NJ., 1994.

10. J. G. O. O. Yang, M. Halper and Y. Perl, "The oodb ownership relationship," Tech. Rep. 18, New Jersey Institute of Technology, Newark, NJ., 1994.

11. I. ONTOS, *ONTOS DB 2.2 Developers Guide*, ONTOS, Inc., Three Burlington Woods, Burlington, MA., 1993.

12. I. ONTOS, *ONTOS DB 2.2 First Time Users Guide*, ONTOS, Inc., Three Burlington Woods, Burlington, MA., 1993.

13. I. ONTOS, *ONTOS DB 2.2 Reference Manual, Volume 1*, ONTOS, Inc., Three Burlington Woods, Burlington, MA., 1993.

14. I. ONTOS, *ONTOS DB 2.2 Tools and Utilities Guide*, ONTOS, Inc., Three Burlington Woods, Burlington, MA., 1993.

15. I. Pohl, *TURBO C++*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA., 1991.

16. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Co., Inc., Reading MA., 2nd ed., 1991.