ALGORITHMS FOR STRUCTURAL SCHEMA INTEGRATION

James Geller, Ashish Mehta, Yehoshua Perl
CMS and Inst. for Integrated Sys.,
CIS Dept., NJIT,
Newark, NJ 07102,
geller@vienna.njit.edu

Erich Neuhold
GMD-IPSI,
Darmstadt,
Germany, D-6100

Amit Sheth
Bellcore,
Piscataway
NJ 08854

Abstract

Current view and schema integration methodologies are driven by semantic considerations, and allow integration of objects only if that is valid from semantic and structural viewpoints. We had introduced a new integration technique called structural integration. It permits integration of objects that have structural similarities, even if they differ semantically. This technique uses the object-oriented Dual Model which separates the representation of structure and semantics.

In this paper we introduce algorithms for structural integration. We apply these algorithms to integrate two views of a large university database schema which had significant structural similarities but differed semantically.

1 Introduction

In the process of creating databases for a given application in an enterprise, views are defined, possibly by different designers, that describe subsets of the data. Once the different views have been created, one needs to integrate them into a single schema that describes all the data used by the application. This is called *view integration*. The process of integrating schemas from different (possibly existing) databases is called *schema integration*. The techniques for integrating views and schemas are similar [SL90].

Generalization is a useful technique for integrating views/schemas [DH84]. In the current methodologies integration by generalization can be used for two classes that are similar in structure and semantics, because the description of a class contains both *structural* and *semantic* information. While studying the schemas of a large university database, it was observed that several subschemas had the same or very similar structures, but different semantics. Unfortunately, the generalization technique described above cannot be applied, in spite of the structural similarities. The structural similarities can be exploited for integration if the model supports a clear distinction between structure and semantics, as the object-oriented Dual Model [NPGT91,NGPT90,NPGT89] does. Thus, the Dual Model can support multiple semantics for the same structural specification. It refines the integration-by-generalization technology by using two hierarchies – structural and semantic. In addition, we introduce a unique new method for integration, called *structural integration* [GPN91a, GPCS91]. Structural Integration does not replace integration by generalization but supplements it wherever there is structural similarity between semantically different objects. It works by using a common object type which represents the structural aspects of both classes. Two advantages of structural integration are savings in the specification of properties and especially methods, and easier comprehension of a complicated schema by a user. These advantages are discussed in more detail in Sections 3 and 6.

In this paper we present several algorithms to integrate classes and subschemas with full structural correspondence. We believe that it is important to test these algorithms in complex real world situations. Therefore, the integration is demonstrated using portions of a large university database schema.

This paper is organized as follows. In Section 2 we briefly present the Dual Model for object-oriented databases. In Section 3 we discuss structural integration possible in the Dual Model representation. The representation of subschemas from our university database is discussed in Section 4. Section 5 introduces several algorithms for full structural integration. Section 6 contains the conclusions.

0-8186-2697-6/92 $03.00 © 1992 IEEE

2 The Object-Oriented Dual Model

In this section we briefly summarize necessary features of the Dual Model for understanding structural integration. Additional details can be found in [NGPT91].

A class can be regarded as a container for objects that are similar in their structure and their semantics in the application. A class description consists of four types of properties: attributes, relationships, methods, and generic relations.

1. *Attributes* specify printable values of a given data type.
2. *Relationships* specify pointers to other classes.
3. *Methods* specify operations that can be applied to instances of a given class.
4. *Generic relations* describe system-supported connections between classes.

In many current systems the description of a class contains both, structural aspects and semantic aspects (e.g., [CM84, KNS88, F87, GR83, SR86]). In object-oriented systems (e.g., O_2 [LR88], ORION [KIM90], GemStone [BOS91], ObjectStore [LLOW91]) the subclass hierarchy is used for two purposes: (1) to factorize the common structure and behavior of classes, and (2) to express additional semantic relationships between classes. This leads to a situation in which two classes modeling semantically related objects could only be dealt with if the objects in question are structurally related as well, which results in inadequate conceptual models. Therefore, the Dual Model separates those two aspects of the specification.

An aspect of a specification is considered *structural* if either (1) it is composed of names, types, and logical or arithmetic operations, or (2) it is decidable whether this aspect is consistent with the mathematical representation of the class(es) it connects to.

The structural aspects of a class can be organized as an abstract data type, called the *object type* of that class. Hence, we associate with each class an object type. Different classes may have the same object type. An object type is determined by its properties: attributes, relationships, methods and (structural) generic relations. The same properties are used in an object class. However, in an object type a relationship refers to an object type, while in a class it refers to a class. Similar differences apply to generic relations and methods.

An aspect of a specification is considered *semantic* if either (1) it refers to actual instances of objects in the application or if (2) just based on the mathematical representation of the class(es) an aspect connects to, it is not decidable whether the aspect describes properly the connection between the corresponding real world object(s) and their features. For further discussion of these definitions and their implications see [GPN91c].

Several models have suggested refinements of the IS-A relation [AH87, DH84]. The Dual Model defines two kinds of semantic generic relations between classes, *categoryof* and *roleof*. Both the relations relate the specialized class to the more general class when both classes are in the same context and in the different context, respectively.

The representation of a class may contain semantic *essentiality* and *dependency* constraints. An instance of a class can only exist if the values of its *essential* properties are all different from nil. If the existence of an object depends on the existence of another object, we can model this with a *dependent* relationship.

Many researchers assume that object-oriented databases are convenient for integration and code reusability due to their generalization capabilities expressed by the subclass hierarchy [BM89, K89]. The Dual Model uses a structural hierarchy and a semantic hierarchy [NGPT89]. Furthermore, the Dual Model permits the assignment of one object type to several classes that are semantically different, but share the same structure. This enables structural integration [GPCS91, GPN91a, GPN91b].

3 Structural Integration

Every property can be viewed as a pair, consisting of the *name of the property* and the *name of the type* (or class) of this property. The function "selector" returns the first element of the pair.

If the pair describes a relationship defined in a class (an objecttype), then the function "class" ("objecttype")returns the second element of the pair. The same conventions are used for the second element of a pair describing a generic relation. We now discuss the conditions that enable structural integration.

Consider two sets of classes of equal cardinality $\mathbf{A} = \{a_1,a_2,...,a_n\}$, $\mathbf{B} = \{b_1,b_2,...,b_n\}$. Structural integration between the sets **A** and **B** is possible if there exists a correspondence between **A** and **B** such that for every two corresponding classes $a \in \mathbf{A}$, and $b \in \mathbf{B}$ one can construct a common object type. There are two cases of correspondence, full structural correspondence and partial structural correspondence. This paper considers only algorithms for full structural correspondence (defined later in this section). The case

of partial structural correspondence is discussed in [GPN91a, GPN91b].

In order to construct a common object type for two corresponding classes a and b there must exist a full structural correspondence between these two classes, i.e., between their sets of properties. Full correspondence for attributes means that their data types must be identical. Full correspondence for relationships means that the referenced classes have to be of the same object type. For attributes and relationships, the selectors may be different. Fully corresponding generic relations must be identical and point to classes of the same object type.

Concerning methods, we note that there are two types of methods, general purpose methods which are program segments and path methods which are used to traverse the database. (Actually a concatenation of these two is possible, where a result found at the end of a path can be used as an argument to a program segment.) In this paper we limit our treatment of integration to path methods which are, informally speaking, compositions of relationships and generic relations. A formal definition of path methods is complicated and long and is given in [NGPT91]. The reason for the restriction to path methods is that checking general purpose methods for equivalence is a subject of other areas of Computer Science, and is also the subject of well known undecidability results. The relevant problem for object-oriented database research is to check the correspondence of path methods.

Formally, let the class a (b) have a set $\{x_i\}$ ($\{y_i\}$) of attributes, a set $\{r_i\}$ ($\{s_i\}$) of relationships, a set $\{m_i\}$ ($\{n_i\}$) of path methods and a set $\{g_i\}$ ($\{h_i\}$) of generic relations to other classes. The classes a and b stand in ***full structural correspondence*** iff:

1. There exists a one-to-one correspondence between the sets of attributes $\{x_i\}$ and $\{y_i\}$ such that if x_i corresponds to y_i, then datatype(x_i)= datatype(y_i).
2. There exists a one-to-one correspondence between the sets of relationships $\{r_i\}$ and $\{s_i\}$ such that either class(r_i) = class(s_i) or they both have the same object type.
3. There exists a one-to-one correspondence between the sets of path methods $\{m_i\}$ and $\{n_i\}$ such that if m_i is a method that defines a path going through the sequence $a_1, a_2, \ldots a_s$ of classes, and n_i defines a similar path $b_1, b_2, \ldots b_t$ of classes then the following conditions hold: (a) $s = t$ and (b) either $a_i = b_i$ or a_i and b_i have the same object type.
4. There is a one-to-one correspondence between the sets of generic relations $\{g_i\}$ and $\{h_i\}$ such that (1) either selector(g_i) = selector(h_i) or one is *roleof* and second is *categoryof* and (2) either class(g_i) = class(h_i) or class(g_i) and class(h_i) have the same object type.

Let us now discuss the advantages of structural integration. One advantage of structural integration is the elimination of some repetition in the specification of classes. This is the case for common attributes and methods of two classes. Rather than specifying a method for each class, it is given with the object type and then parameterized in the classes according to the specific details of this method. This advantage can be compared to the advantage generally achieved in programming by parameterization and comprises a form of code reusability. It applies not only to the path methods that were introduced earlier in this paper for schema traversal, but to any form of general purpose method, i.e., to any program segment. On the other hand, only path methods are relevant for automatic integrations, and therefore only they are treated in our algorithms.

The other advantage of structural integration is a cognitive one. When a user needs to understand a large and complicated schema of many classes, it is easier for him/her to study a comparatively smaller schema of object types and then to apply that understanding to different subschemas of classes that are corresponding to the subschema of object types.

This is similar to defining a macro by extracting text that appears repeatedly in a program. After having studied the macro once, the user understands it at every place where it is invoked and does not have to study it again, just to find out that he has already seen a similar structure before. Just as each invocation of a macro can occur in a different semantic context, each invocation of type-related information occurs in a different semantic context.

4 Representation of two Subschemas of a University Database

A university database was developed and implemented at NJIT [CTWA90]. The development of this database dealt with different complex problems involved in real world modeling. This database schema has more than 300 classes. It contains information about students, professors, courses, employees, schools, colleges, departments, committees, admis-

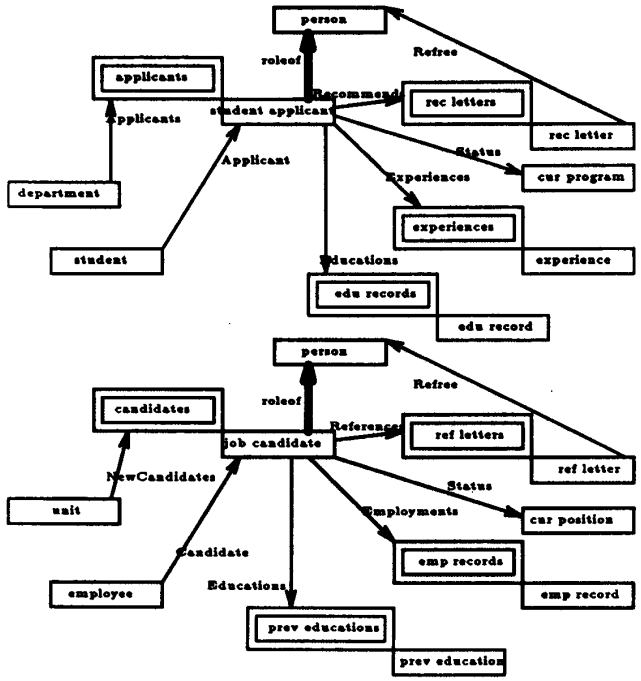

Figure 1: Graphical representation of ADMISSION SUBSCHEMA (top) and HIRING SUBSCHEMA (bottom)

sion, employment, finance, computers, publications, etc.

While working on this database we found two structurally similar subschemas, the ADMISSION SUBSCHEMA and the HIRING SUBSCHEMA, that cannot be integrated because of different semantics. In this section we will represent these two subschemas to demonstrate structural integration. The ADMISSION SUBSCHEMA contains information about all the student applicants who apply to the university. The most important class of this subschema is *student_applicant*. The HIRING SUBSCHEMA contains information about all the job_candidates who have applied for a job. The most important class of this subschema is *job_candidate*. The *student_applicant* class and the *job_candidate* class are given below, using the VML (VML = VODAK Modeling Language) language [FKRST89]. We rely on the intuition of the reader concerning the easy to understand syntax. The other classes for both subschemas are discussed in [GMPNS91] and not shown here due to space limitation.

In Figure 1 the details of the two subschemas are explained using the graphical schema representation language OOdini(=OODINI = Object-Oriented Diagrams at the New jersey Institute) [GHP92]. A rectangle represents a class, and a double line rectangle represents a set class. A thick line arrow represents generic relations such as *roleof* and *categoryof*, and a thin arrow represents a relationship between classes. A set class representation shares one corner with the box that represents its member class.

ADMISSION SUBSCHEMA
class student_applicant
memberof: applicants
roleof: person
attributes:
IdNumber: INTEGER
Citizenship: STRING
StartingDate: DATE
Funds: INTEGER
Program: STRING
essential: IdNumber
relationships:
Status: cur_program
Educations: edu_records
Recommends: rec_letters
Experiences: experiences
methods:
applicantInstitute ():
Status → cur_program
School → STRING

HIRING SUBSCHEMA
class job_candidate
memberof: candidates
roleof: person
attributes:
IdNumber: INTEGER
Citizenship: STRING
ApplicationDate: DATE
Salary: INTEGER
Position: STRING
essential: IdNumber
relationships:
Status: cur_position
Educations: prev_educations
References: ref_letters
Employments: emp_records
methods:
candidateCompany ():
Status → cur_position
Company → STRING

The structural similarity between two subschemas can be observed from the graphical representations shown in Figure 1. These two subschemas, though structurally similar, cannot be integrated by applying the traditional generalization-based integration techniques because of semantic differences. For example, both classes *student_applicant* and *job_candidate* have an attribute with the data type DATE. However, the semantics of these two attributes are different. For *student_applicant* the selector is StartingDate showing the expected enrollment date. For *job_candidate* the selector is ApplicationDate which is considering the date of application rather than the enrollment. The attribute Program is semantically different from Position. Another example is Funds versus Salary. While both consider amounts of money the first is to be received by the university while the second is to be paid by the university. Obviously, two classes that differ in the semantics of some of their attributes are themselves different in their semantics.

Another reason why *student_applicant* and *job_candidate* are semantically different can be seen

from the fact that, if accepted, one will become a student whose function is to study while the second will become an employee whose function is to work. To point out another difference, a *job_candidate* requires an interview while a *student_applicant* does not. Thus, while both classes represent a person which submitted an application in the university environment, the differences in their roles as applicants and their future roles exclude the existence of a common real world class which includes *only these two classes*. There are many more cases where a person submits an application for some purpose in the university environment, e.g., for a library card, a computer account, for a visitor parking permission, etc.

5 Algorithms for Full Structural Correspondence

To help a designer performing structural integration in a large database we supply a sequence of algorithms for different tasks. The *CORRESPONDENCE* procedure checks whether two given classes can have a common object type. In the positive case the output of this procedure is used in the next procedure *STRUCTURAL_INTEGRATION* to create a common object type.

These procedures are repreatedly used by the procedure *SCHEMA_INTEGRATE*. To achieve better results, this procedure should be preceded by the procedure *REORDER* which reorders the classes within the two schemas to be integrated. The procedure *SCHEMA_INTEGRATE* cannot integrate cyclic subschemas. For this case we introduce the more complex procedure *CYCLIC_SCHEMA_INTEGRATE*.

We start with an algorithm to check whether two given classes satisfy full structural correspondence. Before presenting this algorithm some additional notational conventions are introduced.

The function "typeof" returns the object type of a class. The function *typeof*(X) is an abbreviation for typeof(class(X)). The function "class-sequence" returns the sequence of all classes that occur in a path method defined in a class. Similarly, "type-sequence" returns the sequence of all object types that occur in a path method defined in an object type.

PROC CORRESPONDENCE (IN ***a*** , ***b***: class; OUT ***matcha***, ***matchgr***, ***matchr***, ***matchm*** : array)

[*The correspondence between the matching kinds of properties will be returned in the appropriate arrays-matchafor attributes, matchgr for generic relations, matchr for relationships, matchm for methods.*]

1. IF the number of attributes in a and b is not equal, THEN exit. [*No full correspondence is possible for all exits in this algorithm. Every exit prints a message with the two classes that caused the failure.*]

IF the number of attributes of any given data type in a and b is not equal, THEN exit.

ELSE insert correspondence between attributes into array *matcha*.

[*The correspondence in matcha is set according to data types. If there are several attributes with the same data type, any matching between the two sets of the same data type will do for the full correspondence. However, one can have an option of displaying the sets of attributes to the user for proper matching according to the selectors and other information.*]

2. Consider the *categoryof* and *roleof* generic relations of a and b.

IF their numbers are not equal THEN exit.

IF a and b both have one such relation to classes a_1 and b_1, respectively

THEN [*Single Inheritance*]

IF $a_1 \neq b_1$ and a_1 and b_1 do not have identical object types THEN exit,

3. ELSE [*Multiple Inheritance*]

a has many *categoryof* or *roleof* relations to $a_1, a_2, \ldots, a_m$ and b has many *categoryof* and *roleof* relations to $b_1, b_2, \ldots, b_m$.

[*In this step we look for a one-to-one matching between* $a_1, a_2, \ldots, a_m$ *and* $b_1, b_2, \ldots, b_m$.]

```
FOR k := 1 to m DO
  matchgr[k] := 0
[matchgr is used to record the correspondence.]
FOR i := 1 to m DO {
  flag := false
  [flag is used to indicate whether a_i is matched.]
  k := 1
  WHILE k ≤ m AND NOT flag DO {
    IF matchgr[k] = 0 THEN
    [b_k is free for matching.]
      IF (a_i = b_k) or a_i and b_k have
      a common object type THEN {
        matchgr[k] := i
        [So, b_k cannot match a second a_j.]
        flag := true
      }
      k := k + 1
  }
```

IF NOT flag THEN exit
[*No full correspondence possible*]
}
output the array *matchgr*
[*This is the case of full correspondence.*]

4. Consider the *setof* relations of a and b. (There exists at most one.) The treatment is the same as in step 2. This correspondence is inserted into array *matchgr*.

5. Consider the *memberof* relations of a and b. (There may be more than one.) The treatment is the same as in steps 2 and 3. This correspondence is inserted into array *matchgr*.

6. Consider the relationships of a and b. The treatment is the same as in steps 2 and 3. This correspondence is inserted into array *matchr*.

7. Consider the path methods in a and b.
IF their numbers are not equal THEN exit.

ELSE Let us assume that a has path methods $\{ma_1, ma_2, \ldots, ma_n\}$ and b has path methods $\{mb_1, mb_2, \ldots, mb_n\}$.
FOR k := 1 to n DO
matchm[k] := 0
[*matchm is used to record the correspondence between path methods.*]
FOR i := 1 to n DO {
flag := false
k := 1
WHILE $k \leq n$ AND NOT flag DO {
IF *matchm*[k] = 0 THEN {
[mb_k *is free for matching*]
IF ma_i and mb_k have same number of classes p in sequence THEN {
Let the class sequence of ma_i be $< ma_{i_1}, ma_{i_2}, \ldots, ma_{i_p} >$ and the class sequence of ma_k be $< mb_{k_1}, mb_{k_2}, \ldots, mb_{k_p} >$
num := 0
FOR j := 1 to p DO {
IF $ma_{i_j} = mb_{k_j}$ or ma_{i_j} and mb_{k_j} have the same object type
THEN num := num + 1
}
IF num = p THEN {
matchm[k] := i
flag := true
}}}
$k := k + 1$
}
IF NOT flag THEN exit [*No full correspondence is possible since ma_i was not matched*].
}
output the array *matchm* [*This is the case of full correspondence.*]

If no full correspondence was found the user can now test for partial corresponding [GPN91b]. The complexity of the *CORRESPONDENCE* algorithm is $O(m^2 + n^2L)$ where m is the maximum of {#relationships, #categoryof+#roleof relations, #memberof relations}, and L is the maximum length of the n path-methods. We can use the algorithm *CORRESPONDENCE* to find pairs of corresponding classes from both given databases. If this algorithm completes successfully then we can apply the algorithm *STRUCTURAL_INTEGRATION* which creates a common object type A for the fully corresponding classes a and b.

PROC STRUCTURAL_INTEGRATION (IN ***a***, ***b***: class; ***matcha***, ***matchgr***, ***matchr***, ***matchm***: array; OUT ***A***: object type)

1. For every two corresponding **attributes** x_i, y_i (in array *matcha*)
IF selector(x_i) = selector(y_i) THEN
define in the object type A an attribute (selector(x_i), datatype(x_i))
ELSE
define in A an attribute (z, datatype(x_i)) with z = selector(x_i), or z = selector(y_i), or z is a new selector, freely chosen by the human integrator.
2. For every two corresponding **relationships** r_i, s_i (in array *matchr*)
IF selector(r_i) = selector(s_i) THEN
define in A a relationships (selector(r_i), *typeof*(r_i))
ELSE
define in A a relationship (z, *typeof*(r_i)) with z = selector(r_i), or z = selector(s_i), or z is a freely chosen new selector.
3. For two corresponding **methods**, m_i, n_i (in array *matchm*)
with class-sequence(m_i) = < k_1, k_2, ..., k_s >,
define type-sequence(m) = <typeof(k_1), typeof(k_2), ... , typeof(k_s)>,
IF selector(m_i) = selector(n_i) THEN
define in A a method m, such that selector(m) = selector(m_i).
ELSE
define in A a method m, such that selector(m) = z

where z = selector(m_i),
or z = selector(n_i),
or z is a freely chosen new selector.

4. The **generic relations** are considered as follows:
For two corresponding *memberof* or *setof* generic relations, g_i, h_i
define in A a generic relation
(selector(g_i), *typeof*(g_i)).
[Note that by the definition of full structural correspondence it must be the case that selector(g_i) = selector(h_i).]
For two corresponding *categoryof* or *roleof* generic relations g_i, h_i
define in A a generic relation
(subtypeof, *typeof*(g_i)).

The complexity of this algorithm is linear in the sum of the number of properties of the classes and the sum of the lengths of the path methods of these classes. We first apply *CORRESPONDENCE* to the two classes *person* (see [GMPNS91]). Since they have only attributes and there is a correspondence of the data types (some of which are composite) the algorithm succeeds and structural integration creates a common object type PERSON. We apply algorithm *CORRESPONDENCE* to two classes *student_applicant* and *job_candidate*. As the numbers of attributes are equal and the numbers of corresponding data types are the same, the first step is successful. As both the classes have one *roleof* generic relation to the class *person*, the second step will also be successful. We will consider the generic relation *memberof* later. Let us consider the relationship *CurrentProgram* to the class *current_program* and the relationship *CurrentPosition* to the class *current_position*. The algorithm will fail in step 2 because the classes *current_program* and *current_position* are not the same and do not share the same object type. On the other hand the algorithm will succeed to integrate the classes *current_program* and *current_position* themselves.

We observe that the order of processing the classes may have an impact on whether it is possible to integrate two classes. Suppose, for example, that class a_1 (b_1) has a generic relation or a relationship to class a_2 (b_2). Suppose further that none of these classes have more connections to other classes and that the attributes of a_1 and b_1 (a_2 and b_2) exhibit full structural correspondence. If we apply *CORRESPONDENCE* to classes a_1 and b_1 first, the matching will fail due to the generic relation or relationship to a_2 (b_2), since $a_2 \neq b_2$ and a_2 and b_2 do not yet have a common object type (see step 2 in procedure *CORRESPONDENCE*). On the other hand *CORRESPONDENCE* for classes a_2 and b_2 will be successful and an object type A_2 will be created for both. If we now apply *CORRESPONDENCE* for classes a_1 and b_1, it will also be successful. This implies that if we consider the two classes *current_program* and *current_position* first then the object type *STATUS* will be created, and the application of the algorithm to *student_applicant* and *job_candidate* would not fail at that point.

If a_1 (a_2) and b_1 (b_2) have cyclic connections, i.e., there is a directed path from a_1 (a_2) to b_1 (b_2) and vice versa, then no order of processing will permit integration with this algorithm. That is, a_1 and a_2 may potentially have the same object type A_1, and b_1 and b_2 may potentially have the same object type A_2, but due to the cyclic nature of the connections of the classes it is impossible to recognize this fact with the *CORRESPONDENCE* procedure applied in any order. This is, for example, the case for *student_applicant* and *student_applicants*. This case will be considered later.

In order to process the classes of each database DB which do not participate in cyclic subschemas and gain the possible results from applying the *CORRESPONDENCE* algorithm we need to reorder the classes in each database, using the array *order*. Let n be the number of classes in DB.

PROC REORDER (IN ***DB*** : SCHEMA ; ***n*** : INTEGER; OUT ***order*** : array)

1. Perform a topological sort for the classes that are involved in the acyclic part of the schema.

2. The remaining classes, involved in the cyclic part of the schema, are arbitrarily ordered.

The code is omitted because this is a well-known technique [AHU87].

If we first apply the algorithm REORDER to reorder all the classes in the database then the classes *current_program* and *current_position* will be integrated before *student_applicant* and *job_candidate*. Now we can present an algorithm *SCHEMA_INTEGRATE* for finding and creating common object types for classes with full structural correspondence of the acyclic parts of two databases DA and DB. Let DA have m classes $a_1, a_2, \ldots, a_m$ and DB have n classes $b_1, b_2, \ldots, b_n$. Let *DO* be the set of the explicitly defined object types O_k for the integrated database.

PROC SCHEMA_INTEGRATE
(IN ***DA***, ***DB*** : SCHEMA, OUT ***DO*** : SCHEMA)

call REORDER(*DA*, *m*, *order*)
call REORDER(*DB*, *n*, *order*)
FOR $i := 1$ to m DO
  FOR $j := 1$ to n DO {
    CORRESPONDENCE (a_i, b_j,
    matcha, *matchgr*, *matchr*, *matchm*)
    IF CORRESPONDENCE is completed
    successfully THEN
      call STRUCTURAL_INTEGRATION (a_i, b_j,
      matcha, *matchgr*, *matchr*, *matchm*, O_j)
  }

For the classes which were not matched, their object types are still defined implicitly in the integrated database as they were in DA and DB prior to the structural integration. The complexity of this algorithm is $O(mn(r^2 + n^2L))$ where r is the maximum number of connections of a given kind and L is the maximum length of all the n path methods.

One major limitation of the previous algorithm is that it does not work for any pair of schemas which have cycles. For an object-oriented database schema it is very common to be cyclic. We now present a procedure which can integrate schemas with cycles.

This procedure accepts human advice to avoid matching all permutations of two sets of properties which would increase its time complexity drastically. It uses the user's intuition in picking two classes for starting the matching process, rather than to run over all possibilities.

Once the procedure finds two corresponding subschemas it halts. This is the case when full structural correspondence between the two sets of classes is achieved and there exist no classes in the subschemas with a relationship or relation to a class outside of its subschema. To continue searching for more corresponding subschemas the procedure needs another pair of unmatched classes to start with.

The algorithm uses two queues, Q_A and Q_B, and two arrays, R_A and R_B, to process the classes of the two desired corresponding subschemas, S_A and S_B. Initially, Q_A, Q_B, R_A, and R_B are all empty.

PROC CYCLIC_SCHEMA_INTEGRATE (IN D_A, D_B: SCHEMA; OUT R_A, R_B: SCHEMA; ***MATCH_A***, ***MATCH_GR***, ***MATCH_R***, ***MATCH_M***: array);
[*The four arrays MATCH_A, MATCH_GR, MATCH_R, and MATCH_M contain as elements arrays. For example, MATCH_A at position i contains an instance of the array matcha. This instance of matcha contains the correspondence of attributes of the classes a_i and b_i. The three other arrays contain arrays describing the correspondences*

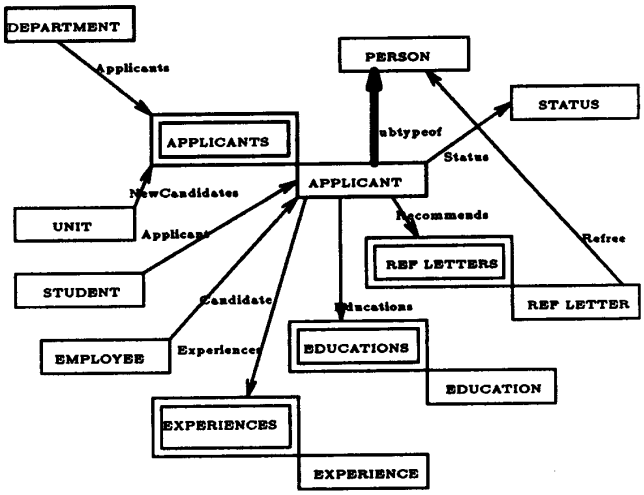

Figure 2: Graphical representation of the integrated subschema for ADMISSION SUBSCHEMA and HIRING SUBSCHEMA

for generic relations, relationships, and methods.]

1. Initialization: The user picks two object classes a_1 and b_1 which seem to be corresponding but are not identical in their properties.

   Insert a_1 into Q_A and b_1 into Q_B.

2. WHILE the queues Q_A and Q_B are not empty DO {

   Let a_i (b_i) be the class at the front of Q_A (Q_B).
   Transfer a_i (b_i) from Q_A (Q_B) to R_A (R_B).
   3. IF the number of attributes in
   a_i and b_i is not equal,
   THEN exit(a_i,b_i).
   [*No full correspondence is possible.*]
   IF the number of attributes of any given data type in a_i and b_i is not equal,
   THEN exit(a_i, b_i).
   [*No full correspondence is possible.*]
   ELSE insert the correspondence into array *matcha*, and insert *matcha* into to MATCH_A.
   4. Consider the *categoryof* and *roleof* relations of a_i and b_i.
   IF their numbers are not equal THEN exit(a_i,b_i).
   IF a_i and b_i both have one such relation to object types a_j and b_j, respectively THEN
   IF a_j and b_j do not have identical

object types or
a_j and b_j do not appear as a pair
in the arrays R_A and R_B, or
they do not appear as a pair in the
queues Q_A and Q_B THEN
insert a_j into Q_A and
insert b_j into Q_B.
[*They are now candidates for comparison. If they appeared in R_A, R_B they were compared already. If they already waiting for processing.*]
Insert the correspondence in array *matchgr*
and *matchgr* is inserted into *MATCH_GR*.

5. [*Multiple inheritance:*]
IF a_i has many *categoryof* and *roleof*
relations to $a_{j_1}, a_{j_2}, \ldots, a_{j_m}$
and b_i has many *categoryof* and *roleof*
relations to $b_{j_1}, b_{j_2}, \ldots, b_{j_m}$ THEN
FOR $i := 1$ to m DO
FOR $k := 1$ to m DO
IF a_{j_i} and b_{j_k} have
a common object type or
are stored as a pair in the
array R_A and R_B or
are stored as a pair in the
queues Q_A and Q_B
THEN delete a_{j_i} and
b_{j_k} from the appropriate list
($a_{j_1}, a_{j_2}, \ldots, a_{j_m}$
or $b_{j_1}, b_{j_2}, \ldots, b_{j_m}$).
6. Set m to the number of remaining
object classes. Consider the two sets
of remaining object classes
$a_{j_1}, a_{j_2}, \ldots, a_{j_m}$
and $b_{j_1}, b_{j_2}, \ldots, b_{j_m}$.
Display these two sets to the user so
that he can suggest a one-to-one
correspondence between these two sets
and can rearrange the order of the
b_{j_k} classes respectively, such
that a_{j_k} corresponds to
b_{j_k}, $k = 1, \ldots, m$.
Insert $a_{j_1}, \ldots, a_{j_m}$
into Q_A and $b_{j_1}, \ldots, b_{j_m}$
into Q_B. Store this correspondence in
array *matchgr* and *matchgr* is inserted
in MATCH_GR.
[*The only alternative for the described user interaction will be to try all permutations of these two sets of classes.*]
7. Consider the *setof* relations of
a_i and b_i (There exists at most one.)
The treatment is the same as for
categoryof and *roleof* in step 4.
Store this correspondence in *matchgr*
and *matchgr* is inserted into MATCH_GR.
8. Consider the *memberof* relations of
a_i and b_i. (There may exist more than one.)
The treatment is the same as for *categoryof*
and *roleof* in steps 4, 5 and 6.
Store this correspondence in *matchgr*
and *matchgr* is inserted into MATCH_GR.
9. Consider the relationships of
a_i and b_i.
The treatment is the same as for *categoryof*
and *roleof* in steps 4, 5 and 6.
Store this correspondence in *matchr*
and *matchr* is inserted into MATCH_R.

END [*of WHILE (2)*]

The treatment of methods, which is an extension of the corresponding part in the first algorithm, is deleted from this algorithm due to space limitation.

Let m_i = maximum(#relationships, #categoryof+#roleof relations, #memberof relations) then the complexity of the algorithm (assuming human interaction) is $\sum_i m_i^2$ with i covering all the classes. Without human interaction the complexity will rise to $\sum_i m_i! m_i$ over all classes due to considering all permutations.

The subschema S_A consists of the classes in R_A and the subschema S_B consists of the classes in R_B, where the correspondence is given by the order in R_A and R_B. For our example initially we enter *student_applicant* into Q_A and *job_candidate* (see Section 4) into the queue Q_B. These two classes will immediately be transferred to R_A and R_B, respectively, and the classes they refer to will be processed. The classes *student_applicants, person, current_program, education_records, recommend_letters, and experiences* will be inserted into Q_A and *job_candidates, person, current_position, previous_educations, reference_letters, and employ_records* will be inserted into Q_B. Then each pair of classes will be checked for full_structural correspondence while inserting in the process the rest of the classes into the queues. Only the classes *student, employee, department,* and *unit* are not processed since they are not reachable from *student_applicant* and *job_candidate.* As a matter of fact these pairs of classes are not in full correspondence and are not structurally integrated at all. (Their details are omitted). The algorithm will successfully find that these two subschemas show full_structural correspondence. Then we apply the algorithm *STRUCTURAL_INTEGRATION* to create

shared object types for each pair of classes. The object type APPLICANT, shown below, will be shared by the two classes *student_applicant* and *job_candidate*.

objecttype APPLICANT
  memberof: APPLICANTS
  subtypeof: PERSON
 attributes:
  IdNumber: INTEGER
  Citizenship: STRING
  Date: DATE
  Funds: INTEGER
  AppliedFor: STRING
 relationships:
  Status: STATUS
  Educations: EDUCATIONS
  Recommends: REFERENCES
  Experiences: EXPERIENCES
 methods:
  ApplicantInstitute ():
  Status → STATUS
  Institute → STRING

In the Dual Model representation this object type is followed by a specification of the classes *student_applicant* and *job_candidate* containing the semantic aspects such as *roleof* and *essential*, references to actual classes wherever an object type is common to more than one class, and the properties with different selectors than in the object type. For lack of space we omit the code of the classes as it can be deduced from their format before integration (Section 4) and the object type specification above.

The following Table 1 shows the correspondence between the object types and the pairs of classes that are structurally integrated by these object types. Figure 2 graphically shows the schema of object types.

Admission Subschema	Integrated Subschema	Hiring Subschema
student_applicant	APPLICANT	job_candidate
applicants	APPLICANTS	candidates
edu_record	EDUCATION	prev_education
edu_records	EDUCATIONS	prev_educations
rec_letter	REF_LETTER	ref_letter
rec_letters	REF_LETTERS	ref_letters
emp_record	EXPERIENCE	experience
emp_records	EXPERIENCES	experiences
cur_program	STATUS	cur_position
person	PERSON	person

Table 1: object types(middle) and corresponding classes(left and right)

6 Conclusions

We presented algorithms as tools for assisting the designer of an OODB in performing structural integration. This technique is possible using the Dual Model. This paper is limited to the case of *full structural correspondence*. However, the results can be extended for partial structural correspondence.

The recommended use of this given sequence of algorithms is to integrate the acyclic parts of two schemas, using the fully automatic *SCHEMA_INTEGRATE* algorithm, which calls the algorithms for integration of pairs of classes. Once the acyclic parts of the schemas were structurally integrated, the more complex *CYCLIC_SCHEMA_INTEGRATE* procedure, which accepts human advice, is to be applied to structurally integrate cyclic parts of the schema.

Throughout the paper the applicability of each algorithm for structural integration has been demonstrated by an example based on a university database developed at NJIT. This example demonstrates the applicability of structural integration in the field of object-oriented database integration.

The advantages of structural integration can be demonstrated with our example as follows. The attribute IdNumber does not have to be repeated in the two integrated classes. Because both classes are using the same selector it is sufficient to specify this selector in the object type. The method ApplicantInstitute will not have to be repeated in the two class specifications after it is included in the object type. The classes will contain only a signature-like header that contains the correct replacement classes for the object types in the method.

The cognitive advantage can be demonstrated as follows. Compare the classes described in Figure 1, and the object types described in Figure 2. Clearly, Figure 2 is simpler, and the user who has to understand Figure 1 will find it easier to first study Figure 2 and then apply his understanding to parts of Figure 1 that reuse the structures in Figure 2. This is especially the case if Figure 1 appears as part of a much larger schema, and if the corresponding structures of two parts of the schema are not exposed by drawing them with the same layout. Figure 2 expresses a wide range of phenomena contained in Figure 1, and in addition it uncovers the correspondence between the subschemas contained in Figure 1.

REFERENCES

[AH87] Abiteboul, S., and Hull, R., "IFO: A Formal Semantic Database Model", *ACM Trans. on Database Systems*, 12, 1987, 525-565.

[AHU87] Aho, A., Hopcroft J., and Ullman J., "Data Structures and Algorithms", Addition Wesley, 1987.

[BM89] Brodie, M. L., and Manola, F., *Database Management: A Survey.* Readings in Artificial Intelligence and Databases, J. Mylopoulos and M. L. Brodie (Eds.), Morgan Kaufmann Publishers, San Mateo, California, 1989.

[BOS91] Butterworth, P., Otis, A., and Stein, J., "The GemStone Object Database Management System", CACM, Oct. 1991, pp. 64-77.

[CM84] Copeland G. and Maier, D., "Making Smalltalk a Database System", *Proc. of the ACM SIGMOD Conf. on Management of Data*, June 1984.

[CTWA90] Chao, H., Teli, V., Wijaya, C., Ahmedi, M., "Development of a University Database using the OODB Dual Model ", two Master's theses, NJIT, Newark, NJ, 1990.

[DH84] Dayal, U. and H. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Trans. on Soft. Eng.*, SE-10, 6, Nov. 1984, pp. 628-644.

[EL89] Elmasri, R., S. Navathe,*Fundamentals of Database Systems*,Benjamin/Cummings,CA,1989.

[F87] Fishman, D. et al.,"IRIS: An Object-Oriented DBMS", *ACM Trans.on Office Info.Syst.*,4(2),1987.

[FKRST89] Fischer D., Klas, W., Rostek, L., Schiel, U., Turau, V., "VML - The VODAK Data Modelling Language", GMD-IPSI, Technical Report, Dec. 1989.

[GHP92] Geller, J., Halper, M., and Perl, Y., "An OODB Graphical Schema Representation", *NJIT TR*, CIS-92-1, submitted for publication.

[GMPNS91] Geller, J., Mehta, A., Perl, Y., Neuhold, E. J., Sheth, A., "Algorithms for Structural Schema Integration", *NJIT TR*, CIS-91-31.

[GPCS91] Geller, J., Perl, Y., Cannata, P. and Sheth, A., "Structural Integration using the Dual Model", Tech.Memorandum TM-STS017628/1,Bellcore,1991.

[GPN91a] Geller, J., Perl, Y., and Neuhold, E. J., "Structural Schema Integration in Heterogeneous Multi-Database Systems using the Dual Model", Proc. First International Workshop on Interoperability in Multidatabase systems, Kyoto,Japan,1991,200-203.

[GPN91b] Geller, J., Perl, Y., and Neuhold, E. J., "Structural Schema Integration with Full and Partial Correspondence using the Dual Model", NJIT, T.R., CIS-91-11, submitted for publication.

[GPN91c] Geller, J., Perl, Y., and Neuhold, E. J., "Structure and Semantics in OODB class Specifications", *Special Issue, Semantic Issues in Multidatabase Systems, SIGMOD record*,1991,40-43.

[GR83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading Massachusetts, 1983.

[K89] Kim, W., "Research Direction for Integrating Heterogeneous Databases," *1989 Workshop on Heterogeneous Databases*, Chicago, IL, December 1989.

[KIM90] Kim, W. "Introduction to Object-oriented Databases", *The MIT Press*, Readings, 1990.

[K90] Klas, W., "A Metaclass System for Open Object-Oriented Data Models," *PhD dissertation*, Technical University of Vienna, Austria, 1990.

[KNS88] Klas W., Neuhold, E. J., Schrefl, M., "On an Object-oriented Data Model for a Knowledge Base," In *Research into Networks and Distributed Applications - EUTECO 1988*, R. Speth Ed., North-Holland.

[LLOW91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D., "The Objectstore Database System", CACM, Oct. 1991, pp. 50-63.

[LR88] Lecluse, C. and Richard, P., "Modeling Inheritance and Genericity in Object-Oriented Databases", *LNCS #326*, ICOT, Japan, 1988. pp. 223-237.

[NGPT91] Neuhold, E. J., Geller, J., Perl, Y., Turau, V.. "The Dual Model for Object-Oriented Databases", NJIT, *T.R. CIS-91-30.*

[NGPT90] Neuhold, E. J., Geller, J., Perl, Y., Turau, V.. "A Theoretical Underlying Dual Model for Knowledge-Based Systems", *Proc. of the first Intl. Conf. on Systems Integration*,NJ,1990,96-103.

[NPGT89] Neuhold, E. J., Perl, Y., Geller, J., Turau, V., "Separating Structural and Semantic Elements in Object-Oriented Knowledge Bases", *Proc. Advanced Database System Symposium*,Japan,1989,67-74.

[SL90] Sheth, A. and J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, September 1990, pp. 183-236.

[SR86] Stonebraker M. and L.Rowe, "The Design of POSTGRES", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, D.C., May 1986.