An algorithm to compare OO-Conceptual Schemas♦

J.Silva, J.A.Carsí, I.Ramos
Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{josilga | pcarsi | iramos}@dsic.upv.es

Abstract

In this paper, an algorithmic study about how to compare object-oriented conceptual schemas in a data migration context is carried out. An algorithm is presented based on the tree-comparison technique that compares conceptual schemas. The algorithm uses some semantic information in order to optimize efficiency. A template of the equivalences and differences between two schemas is generated in an automatic way. The template shows the results in terms of insertions, updates and deletions. This work is the first phase in a tool which performs the automatic migration of databases starting from the differences detected.

Index Terms: Comparison Algorithms, Comparison Strategies, Data Migration, Evolution, Schema Comparison, Tree Processing.

1. Introduction

Many systems and processes can be represented as tree structures, and sometimes it is necessary to compare these structures to determine the interoperation, the differences, or simply the level of similarity between the systems. This technique is used in software engineering field when it is necessary to compare object-oriented conceptual schemas (CS).

The CS comparison is highly related to the database (DB) migration problem. Ideally, software evolution must be performed at the conceptual schema level, when the applications and databases implementing the information systems could be automatically updated using the evolutioned conceptual schema. Sometimes the database of the original system contains information, and this information has to be migrated to the new system; this fact is known as data migration. The main problem concerning data migration is that the databases involved in the migration process are usually very different and are sometimes incompatible.

Precisely in data migration, it is essential to determine the evolution level of the schema and the associated database to be migrated. It is also indispensable to know the changes in the initial system elements as well as the insertions and deletions that have occurred during the evolution process. Moreover it is very important to determine the information to use for the identification of the origin and the target of the data migrated. For example if Table A exists in the original system DB, and there already exists a Table named A in the target system, how could we determine if the information in the initial Table A must be migrated to the final Table A? It may be that Table A in the origin DB has changed its name to B in the target DB, and this information is vital for the migration.

In this context, it is necessary to design an algorithm wich is able to automate the CS comparison process. In addition, it would be useful for the algorithm to use semantic information about the schemas during the comparison in order to improve accuracy. In other words, the algorithm score is not limited to tree comparison; it has to be able to compare conceptual schemas using the specific conceptual schemas information to resolve possible indeterminism. This work represents the first phase of the migration process in the data migration tool developed [1].

Section 2 provides a review of the CS comparison problem and its profound relation to the general tree-comparison problem. In section 3, we present how the comparison problem is currently dealt with. In this section, we demonstrate that current algorithms are not appropriate for solving the conceptual schema comparison problem in data migration contexts. In section 4, we provide an algorithm that solves the conceptual schema comparison problem and automatically generates all the mappings between the schemas. The conclusions of the article are presented in section 5.

2. The conceptual schema comparison problem

We look for an algorithm to automate the CS comparison in order to migrate the data stored in the DB related with the original CS to the DB automatically

♦ Work partially funded by the FEDER project "Object-Oriented Software Systems Automatic Generation" (TIC 1FD97-1102).

Proceedings of the International Conference on Software Maintenance (ICSM'02)
0-7695-1819-2/02 $17.00 © 2002 IEEE

generated from the evolutioned CS. The algorithm has a double utility. First, it can measure the evolution level that exists between both schemas. The algorithm measures different features of the schemas such as the cyclomatic complexity of the services, the number of classes, the mean number of attributes of each class, the inheritance level, etc. Second, when the conceptual schemas that are compared are models of the same system, the algorithm can determine the individual evolution of each element in those schemas. In this way, the algorithm will determine the origin and the target of the data and then use them in a migration tool.

Some comercial tools [2, 3] are able to identify the evolution between schemas without a comparison process. These tools keep the trace of the operations that converts the initial schema to the final schema. The problem with this solution is that it is not flexible, because both schemas must be dependent; that is, one of them has to be an evolution of the previous one. However, using a schema comparison process allows the schemas to be independent or to have a different format. Moreover, the operation trace solution is not appropriate in a data migration context, because, in practice, most migrations imply legacy systems. Systems of this kind don't have an associated conceptual schema. Thus, no trace of evolution services exists between the models. In this situation, our approach models the legacy system using an inverse engineering process, and then the comparison process is performed. A specific algorithm for the conceptual schema comparison is then necessary.

The algorithm must be able to automatically generate the differences between two schemas. The differences are expressed in terms of insertions, deletions and updates. In the data migration context, the final objective is to transfer the stored instances between the DBs; then, we only have to consider the elements of the schemas that have a direct repercussion on data persistence. These elements are classes, attributes, aggregation and specialization relationships [2]. With these elements, we represent a conceptual schema like an acyclic and non-directed graph; in other words, like a tree. See Fig. 1.

Figure 1 shows the representation of each element of the conceptual schema as a node of the tree; the root-node represents the conceptual schema.

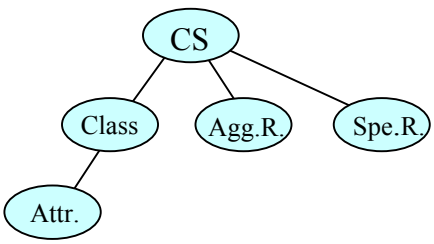


Figure 1: Representation of conceptual schemas as trees

In reality, in a data migration context, the obtention of the minimum transformations between two trees is not the main objective. The main objective is to identify, as well as possible, the individual evolution that undergoes each element of the conceptual schemas. The associated cost of this evolution is not very important. With this consideration in mind, the algorithmic approach to be used is radically different in both cases. First, the algorithm has to identify each one of the elements of both trees, and then it has to establish a bijective function between each pair of elements identified as mapped elements (two elements of two distinct trees are mapped elements, if one element is derived from the other). In this paper, the bijective function is called in the paper 'mapping'. The formal definition of mapping is presented below:

Tree Mapping: It intuitively denotes how an operation set transforms a tree T into another tree T', without considering an order in the application of the operations. Formally, a tree mapping is 3-ple (M, T, T'); where T is the initial tree, T' the final tree, and M a set of pairs of integers (i, j) that satisfy:

1) $1 \le i \le |T|$, $1 \le j \le |T'|$;

2) For each (i_1,j_1) and (i_2,j_2) in M:

a) $i_1 = i_2$ if and only if $j_1 = j_2$

b) $i_1 < i_2$ if and only if $j_1 < j_2$

c) $T[i_1]$ is an ancestor of $T[i_2]$ if and only if $T'[j_1]$ is an ancestor of $T'[j_2]$.

Element mapping: Each pair $(i,j) \in M$ of one tree mapping is called an element mapping, or simply, a mapping.

We can now define the main objectives of our work:

1. To obtain a conceptual schema comparison algorithm that correctly identifies the evolution affecting each one of the elements of the initial conceptual schema.
2. The algorithm must establish a mapping for all the elements of the conceptual schemas.
3. The algorithm should not be limited to tree comparison. It has to be a specific CS comparison algorithm, and must use semantic information for the comparison.
4. The CSs that are compared could be dependent or independent.

Following, we will present an algorithmic analysis that solves the problem taking into account each of the established objectives. First, the minimum path between trees algorithm is presented, and then, the specific CS element identification problem is presented.

3. State of the art

3.1. The 'Tree to Tree' problem

The problem of obtaining the differences between two schemas is currently solved by different comercial tools using the trace of the evolution operations; i.e. VDIFF of Rational Rose [3]. The main problem with these tools is that the CSs have to be dependent. Moreover, the need for an evolution trace requires the schemas to be in the same format, or to pertain to the same application. These restrictions do not exist when using a comparison process.

Many structures and processes can be modeled as labeled trees, and sometimes it is very useful to know how to transform one tree into another tree using the minimum number of changes. A particular case of this problem occurs when the trees are of depth two. This is the case when we compare two strings, where each character is represented by one node with a fixed position in the leaf nodes. This problem was solved by several authors. Sankoff [4], and Wagner and Fisher [5] have presented an algorithm that is capable of computing a sequence of edition operations with minimum cost. This algorithm has an associated computational cost O(n*m), where m and n are the number of leaves of the trees. Later on, Wong and Chandra [6]; and Aho, Hirschberg and Ullman [7] demonstrated that Sankoff's algorithm is optimal for many computation models. Another algorithm that solves the problem with optimum cost was presented in 1977 by Selkow [8].

A generalization of the problem is to consider trees of any depth. In this case, the complexity of the problem increases by a quadratic factor. The following algorithm wich was developed by Kuo-Chung Tai [9] solves the problem.

Given two trees T and T', the algorithm by Kuo-Chung Tai, computes their minimal distance with $O(V * V' * L^2 * L'^2)$ cost, where V and V' are the # nodes of T and T', respectively, and L, L' are the maximal depth levels.

3.2. Compound type changes

A model is proposed in [10] for the identification of type changes encountered in schema evolution. In this work, a set of algorithms for the schema comparison and for the compound type comparison are introduced. The comparison algorithm proceeds through three stages. First, in the *name comparison* stage, old and new types that have the same names in both versions are compared. In the second stage, called *use site comparison*, types using types that have been successfully compared are compared. In the final stage, called *exhaustive comparison*, each old type that does not already have a pair is compared to each new type. This algorithm has an associated cost of O(n*m).

The first inconvenience of this work, is that the structure of the algorithm is very rigid; the types are always compared using the same criterium and always in the same order. An error in the first stage of the algorithm, cannot be restored in the following stages, and can lead to errors in the following stages as well.

In [10], a set of compound type changes are presented to be solved by any comparison algorithm. These changes are the following:

-Inline: Replace a type reference by its type definition.

-Encapsulate: Create a new type by encapsulating parts of one or more types.

-Merge: Replace two or more type definitions with a new type that merges the old type definitions.

-Move: Move part of a type definition from one type to another existing type.

-Reverse Link: Reverse the connection between two types.

-Link Addition: Add a link between two existing types.

The following section presents a study to obtain a CS comparison algorithm wich is able to solve the problem by finding all the related compound changes and improving the [10] algorithm.

4. Algorithmic analysis based on semantic information

Selkow's algorithm computes the minimum path between two trees of depth two; and Kuo-Chung Tai's algorithm computes it for trees of any depth. Thus, we have an inferior limit of the cost of a CSs comparison algorithm. Kuo-Chung Tai's algorithm can be used for comparing CS; however, this algorithm wasn't defined to compare CS but rather to compare simple trees, and it wastes a lot of information of the schemas that can be used to improve the comparison result.

A lot of semantic information can in the context of CSs comparison for data migration be used by the comparison algorithms. Similarly, the information that is generated during the comparison of two elements can be reused in future comparisons of the other elements of the conceptual schemas. In this section, we are going to analyze exactly what information should be used and how it can be utilized to design an algorithm wich is specific to the CS comparison.

Proceedings of the International Conference on Software Maintenance (ICSM'02)
0-7695-1819-2/02 $17.00 © 2002 IEEE

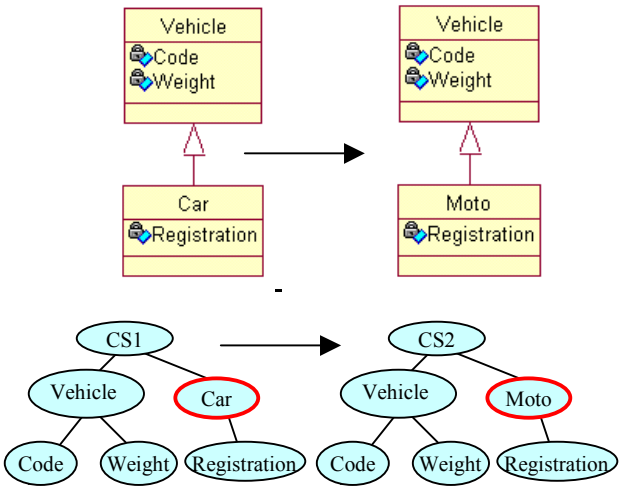


Figure 2: 'Tree to Tree' conceptual schema comparison

The distance concept between two trees can be compared to the measure of similarity between them. In an evolution context, this distance can measure the evolution level that exists between two schemas that are represented as tree structures. Nevertheless, the computation of the minimum path between the trees is not the best solution for identifying the evolution of two conceptual schemas; Figure 2 shows an example of this problem.

If the algorithm searches the minimum path between the trees, it will compute the substitution of 'Car' by 'Moto'. However, the class 'Car' has been deleted and the class 'Moto' has in reality been created so, in this case, a mapping between 'Car' and 'Moto' wouldn't exist. The search of a minimum way is not a good solution for the CSs comparison, because the CSs evolution follows random evolution patterns, that do not have to coincide with minimum paths. Optimization algorithms find the fastest form of converting one tree into another tree. In the following sections, the algorithms attempt to find alternative methods for comparing conceptual schemas; they attempt to identify the real evolution of the class instances. Also, the algorithms presented above do not take advantage of the semantic information of the models.

The 'Tree to Tree' approximation can be used when the conceptual schemas do not have instances to be migrated; but, in a data migration and evolution context, the conceptual schemas have a set of instances associated to them that have to be transferred to another database. In this context, 'Tree to Tree' approximation is not appropriate because it does not take into account the data evolution features.

To solve the problem, it is necessary to find alternative comparison methods; these methods have to take into account the data migration and have to establish the goal of identifying the analyst's proposal with each of the elements of the schemas changes when the evolution is performed. Accordingly, we need to find an algorithm that compares conceptual schemas taking advantage of their tree structure and their CS exclusive properties as well.

One of the initial premises was that the two trees must be labeled trees. The question is what information the labels of the nodes must contain? In a CS there is a lot of information available to compare CSs (identifiers, names, relationships, number of atributes of the classes, etc.) and this information can strongly determine the success of the comparison. Several criteria for CS comparison are analized in detail in [11]. We call the information used to compare a set of elements **comparison criteria**.

Limiting the comparison of two CS to only trees does not take advantage of a lot of semantic information. A CS comparison algorithm must make use of the tree structure, but also has to use the CS structure properties. This idea can provide a lot of rich information during the comparison. The algorithms of the previous section waste this semantic information, and consequently, process all nodes in the same way. Comparing all elements with all? In a CS-Tree, there exist four kinds of branches:

- Classes
- Attributes (subbranch of a class)
- Aggregation relationships
- Specialization relationships

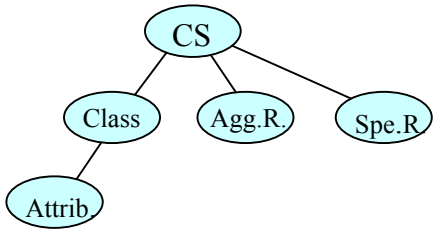
With these four kinds of branches of a CS, the comparisons done by an algorithm are limited to the comparisons between elements of the same type. It does not make sense to compare an attribute to a class. Therefore, the logical comparisons are classes with classes, attributes with attributes, aggregation relationships with aggregation relationships and, finally, specialization relationships with specialization relationships. Because of this, the general tree comparison algorithms can be strongly improved. To do this, we are going to utilize a fragmentation technique with the goal of dividing the trees into comparable subtrees.

4.1 Improving the 'Tree to Tree' algorithms using fragmentation techniques

Given any two labeled trees, where V and V' are its numbers of nodes, and where L and L' are its respective depths; the Kuo-Chung Tai's algorithm computes the minimum way in terms of Additions, Updates and Deletions with a cost:

$$O(V \cdot V' \cdot L^2 \cdot L'^2)$$

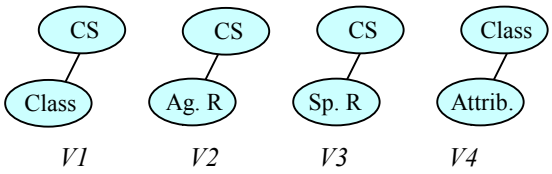
If we represent a conceptual schema as a tree:



The cost of the application of Kuo's algorithm is:

Cost1 = V·V'·4·4 = 16·V·V' ∈ O(V·V')

This cost can be improved if we reduce the number of comparisons by dividing the tree into comparable subtrees. Thereby, if we fragment the tree, we can reduce the cost of both variables L and L' to one:



The cost would be:

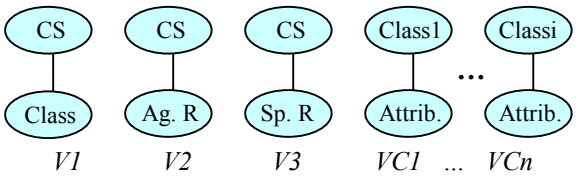
Cost2 = (V1·V'1) + (V2·V'2) + (V3·V'3) + (V4·V'4) ∈ O(V·V')

As $V = V1 + V2 + V3 + V4$ and $V' = V'1 + V'2 + V'3 + V'4$ then:

Cost1 = V1·V'1 + V1·V'2 + V1·V'3 + V1·V'4 + V2·V'1 + V2·V'2 + V2·V'3 + V2·V'4 + V3·V'1 + V3·V'2 + V3·V'3 + V3·V'4 + V4·V'1 + V4·V'2 + V4·V'3 + V4·V'4 ≥ Cost2

As any CS has V1>0 and V4>0, then cost1 is always greater than cost2: cost1 > cost2.

You can decrease this cost if you continue fragmentizing the subtrees (as we will explain later, this is useful only in case the comparison follows a logical order):



The cost would be:

Cost3 = (V1·V'1) + (V2·V'2) + (VC1·VC'1) + … + (VCn·VC'n) ∈ O(V·V')

As $V4 = VC1 + ... + VCn$

then

Cost2 = (V1·V'1) + (V2·V'2) + (V3·V'3) + (VC1·VC'1) +...+ (VC1·VC'n) +...+ (VCn·VC'1) +...+ (VCn·VC'n) ∈ O(V·V') > Coste3

The following always holds: cost1 > cost2 > cost3.

Another kind of semantic information that must be taken into consideration is the comparison order. It is not logical to compare attributes if its classes[1] have not already been compared. The CS comparison must follow a logical order, not a preestablished one but one that depends on the comparison criteria. For example, comparing two schemas using a criterium based on the OID, the logical order would be:

- Analysis of classes (part 1).
- Analysis of attributes.
- Analysis of classes (part 2).
- Analysis of relationships.

The first step (the analysis of classes) is to identify what classes of the initial conceptual schema have been deleted and what classes of the final conceptual schema have been added; the rest of the classes are considered to be invariant invariant[2] classes. The second step is the analysis of attributes wich is done for each of the invariant classes (all attributes of added classes are new; and all attributes of deleted classes have also been deleted). When the analysis of attributes ends, it is possible to determine which classes have been updated, and which are really invariant. Finally, when all classes are identified, the analysis of the relationships between them is done.

When semantic information is applied, the algorithm does not have to compare trees of depth two; it has to compare many trees of depth one. In this case, Selkow's algorithm might solve the problem perfectly by apply it to each pair of trees; but this is false. Selkow's algorithm considers the order of the nodes in the tree when it makes the comparison, and this is irrelevant for the CS comparison. Thus, we can reduce still more the restrictions of the problem even.

4.2 The compound class changes

According to the compound type changes related in [9], the algorithm must incorporate mechanisms for the treatment of compound type changes. The analysis of each compound type change applied to the object oriented CSs follows:

-Inline: Replace a type reference by its type definition.

Inline occurs when all the attributes of two associated classes are included in one class or in both classes. An algorithm that solves inline has to establish mappings

[1] Parent nodes in the tree.
[2] They might have been modified, but are mapping classes.

between attributes of two diferent but related classes. Then, it cannot limit the search of attributes of a class to attributes of its mapping class. In other words, with inline, mappings between attributes of unmapped classes, but related by association, aggregation or specialization in the original CS, can exist.

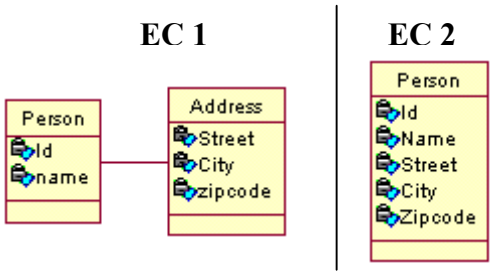


Figure 3: Inline example

-Encapsulate: Create a new type by encapsulating parts of one or more types.

In the object-oriented approach, the encapsulate type change is a generalization of inline. Encapsulate allows the classes to be formed by attributes of one or more classes, and these classes can be related to it or not. If one algorithm solves the Encapsulate type changes, then it also solves Inline type changes. When an algorithm solves Encapsulate, the fragmentation presented in section 4.1 cannot be applied. In this case, attributes do not establish a new subbranch of the CS tree, because the algorithm must search a mapping of one attribute in all attributes of the conceptual schema.

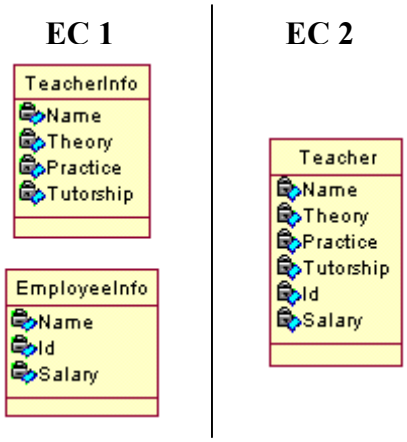


Figure 4: Merge example

-Merge: Replace two or more type definitions with a new type that merges the old type definitions. The main difference between Inline and Merge is that in Merge at least one of the attributes must have the same value in both classes of the CS. When Merge occurs, in reality, two objects exist in the system that are representing two different features of one object of the real model that is being represented. It implies that to determine when two objects of the initial CS have to be transferred to one of the final CS, it is necesary to verify values of the instances's attributes. To do this, the algorithm should use population comparison criteria; that is, criteria that use information from the instances stored in the databases [11].

-Reverse Link: Reverse the connection between two types.

In object-oriented conceptual schemas, this kind of evolution is irrelevant, because the links of the types in the databases's schema are automatically generated from the CS.

-Link Addition: Add a link between two existing types.

This compound type change, like Reverse Link, does not influence the algorithm's design.

-Move: Move part of a type definition from one type to another existing type. This compound change is a subcase of Inline where only a few attributes change class. If Inline is solved, then Move is also solved.

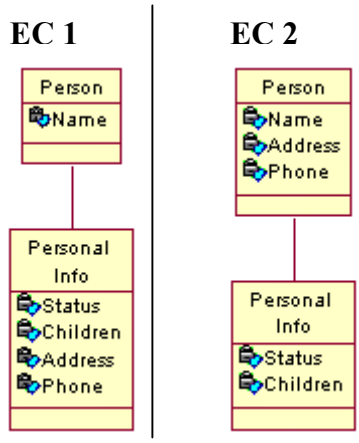


Figure 5: Move example

-Duplicate: Duplicate part of a type definition in another type definition.

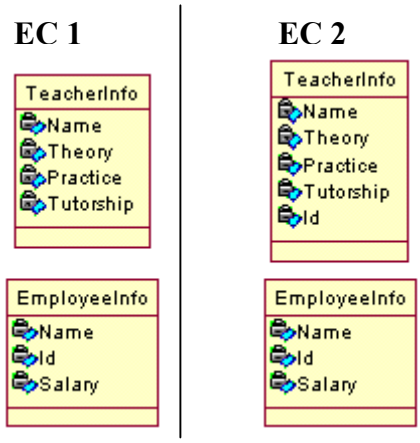


Figure 6: Duplicate example

Like Merge, Duplicate requires population comparison criteria to be solved. In addition, this compound type change implies that several mappings can exist for just one attribute of the original CS. That

is, the information of one object in the original CS, must be transferred to several objects of the final CS.

Now, we can present the algorithm that solves the comparison of conceptual schemas:

We start with the following information:

- **A, B:** Are lists with the elements of the tree T and T', respectively.
- **Criterium**(x, y)**:** Returns TRUE if x and y are the same in accordance with any preestablished comparison criterium.
- **Pair**(x, y)**:** Introduce x and y in the pair list, and take them out of the respective node lists.
- **Deletion**(x): Insert the element x in the deleted elements list.
- **Addition**(x): Insert the element x in the added elements list.
- **Sort**(Ai): As Ai is always a two depth subtree, all elements are at the same depth level. This function sorts the A list with a metric according with the comparison criterium used.

ALGORITHM:

Procedure *CompareTrees(A, B);*

(We assume that A and B have subtrees $A_1,...,A_n$; $B_1,...,B_m$ where the first represents the classes subtree, the second represents the aggregations subtree, the third represents the specializations subtree and the following represents all the attributes subtrees)

Procedure *CompareSubtrees(i, j)*

Begin

Sort(A_i); P_a = ***first***(A_i);

Sort(B_j); P_b =***first***(B_j);

/* P_a and P_b are pointers to the first element of A_i and B_j respectively */

While ((a<Length(A_i)) ***and*** (b<Length(B_j)))

If Criterium(P_a, P_b) = *TRUE* ***then***

Pair(P_a, P_b); P_a = ***next***(A_i); P_b = ***next***(B_j);

Else If $P_a < P_b$ ***then***

Deletion(P_a); P_a = ***next***(A_i);

Else If $P_a > P_b$ ***then***

Addition(P_b); P_b = ***next***(B_j);

End if;

End While; */* Complete the next part of the list with insertions or deletions */*

End */* CompareSubtrees */*

Begin

For *i = 1* ***to 3 do***

CompareSubtrees(i, i);

End For*;*

For *each mapping of classes established in the first iteration where c_i, c_j are the parent nodes of the subtrees A_i and B_j respectively* ***do***

CompareSubtrees(i, j);

End For*;*

/ Look for compound type changes */*

For *each attribute b of B in a mapping class c_b* ***do***

/ Look for Inline and move */*

For *each attribute a of A in a class R related with the c_b mapping class where maximum cardinality of the R relationship is 1* ***do***

If Criterium*(a, b) = TRUE* ***then***

Pair*(a, b);*

End if;

End For*;*

/ Look for encapsulate */*

For *each attribute a of A in a class non-related with the c_b mapping class* ***do***

If Criterium*(a, b) = TRUE* ***then***

Pair*(a, b);*

End if;

End For*;*

End For*;*

End */* CompareTrees */.*

When the algorithm ends, we have three lists. The first one contains all elements that have been deleted from the initial CS; the second list contains all elements that have been added to the final CS; and the third list contains all elements that have been changed during the evolution process. This algorithm is totally dependent on the selected comparison criterium; this is right because the selected criterium determines when two elements are considered the same; this fact is very important because, all criteria are not equivalent, and no criterium exist that is the best in all cases.

Assuming a constant cost for the function 'criterium', this algorithm has an associated cost $O(n*\log(n))$[3] for the first part, because it sorts the lists before its computation. For the search of the compound type changes, the cost is O(s*s), where s is the number of attributes in the CS. The comparison criterium can present a high complexity if it uses the information that the algorithm obtains during the comparison process.

The proposed algorithm is the core of a tool developed at DSIC in the Valencia University of

[3] Cost would be n*log(n) + m*log(m) + n + m : the first two components are derived from the sort of the lists (using for example 'Mergesort' or 'Quiksort'); the third and fourth components of the cost represent the treatment of the lists.

Technology. This tool provides a CASE environment supporting evolution of schemas and the posterior transfer of data between their associated databases. This algorithm automatically determines the individual evolution of each element of the schemas and proposes a default migration plan for its instances.

5. Summary and concluding remarks

The CS comparison problem is a richer process than the simple tree-comparison problem. The optimized algorithms for tree-comparison can be used for comparing conceptual schemas, but they can also be improved using semantic information:

- In a conceptual schema there exist four kinds of comparable elements. Only elements of the same group must be compared.
- Each element of a CS is unique, and is perfectly identified: The first successful comparison of the algorithm can end the search for this element.
- Unlike the simple tree-comparison algorithm, CS comparison algorithms can make use of semantic information that is generated during the comparison process.
- The comparison algorithm must be driven using a logical order of comparison.
- The comparison criteria will strongly determine the final result of the comparison.

The main objective of a comparison is not the search of minimum paths; the main objective is the identification of the individual evolution of each element. Finally, the CS comparison problem is not an optimization problem.

References

[1] **Carsí J.A.**, **Ramos I.**, and **Molina J.C.**, "*Automatic generation of data migration plans from OO conceptual schemas"*, IDEAS2001,IV Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes Software, San José - Costa Rica, ISBN: 9968-32-000, April 2001. (in Spanish)

[2] **Carsí J.A.,** *"OASIS as conceptual framework for software evolution"*, PhD Thesis, Faculty of Computer Science, Valencia University of Technology, October 1999. (in Spanish)

[3] **Rational**, Rational Rose Software Development Company Home Page: http://www.rational.com/.

[4] **Sankoff D.**, "*Matching sequences under deletion/insertion constraints"*, Proc. Nat. Acad. Sci., USA 69, pp. 4—6, 1972.

[5] **Wagner R.A.** and **Fisher M.J.**, *"The string to string correction problem"*, *Journal of the ACM*, 21(1):168-173, January 1974.

[6] **Wong C.K.** and **Chandra A.K.**, *"Bounds for the string editing problem"*, Journal of the ACM, 23(1):13-16, January 1976.

[7] **Aho A.V., Hirschberg D.S.** and **Ullman J.D.**, *"Bounds on the complexity of the longest common subsequence problem"*, Journal of the ACM, 23(1):1-12, January 1976.

[8] **Selkow S.,** *"The tree-to-tree editing problem"*, Department of Computer Science, University of Tennessee, Knoxville, *Information Processing Letters*, 6(6):184-186, December 1977.

[9] **Kuo-Chung Tai,** *"The tree-to-tree correction problem"*, Department of Computer Science, North Carolina State University, Raleigh, Journal of the ACM, vol. 26 no 3, pp. 422-433, 1979.

[10] **Staudt B.,** *"A model for compound type changes encountered in schema evolution"*, University of Massachusetts, Amherst, *ACM Transactions on Database Systems, Vol. 25, No. 1, March 2000, Pages 83–127.*

[11] **Silva J.**, **Carsí J.A.** and **Ramos I.**, *"Classification of comparison criteria for object-oriented conceptual schemas"*, Technical Reports DSIC-II/04/02, Valencia University of Technology, February 2002. (in Spanish)

[12] **Assenova P., Johannesson P.**, *"Improving Quality in Conceptual Modeling by the Use of Schema Transformations"*, in: B. Thalheim (Ed.): Conceptual Modeling: proceedings ER '96, Springer, Berlin et al., S. 277-291, 1996.

[13] **Grau A.,** *"Computer-Aided validation of formal conceptual models"*, PhD. Institute For Software, Information Systems Group, Technical University of Braunschweig, 1998.

[14] **Herden O.**, *"Measuring Quality of Database Schemas by Reviewing - Concept, Criteria and Tool"*, Oldenburg Research and Development Institute for Computer Science Tools and Systems, Escherweg 2, 26121 Oldenburg, Germany.

[15] **Letelier P., Ramos I., Sánchez P.** and **Pastor O.,** *"OASIS 3.0: A formal approach for object-oriented conceptual modelling"*,SPUPV-98.4011, Department of Information Systems and Computation, Valencia University of Technology, 1998.

[16] **Pérez J., Anaya V., Silva J., Carsí J.A** and **Ramos I.,** *"Automatic generation of data migration plans between object-oriented databases",* I Workshop on Distributed Objects Languages Methods & Environments, DOLMEN2001, Sevilla. (in Spanish)

[17] **Pastor O., Insfrán E., Pelechano V., Romero J.,** and **Merseguer J.**, "*OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods",* Conference on Advanced Information Systems Engineering (CAiSE'97). LNCS (1250), pag. 145-159. Springer-Verlag 1997. ISBN: 3-540-63107-0. Barcelona, Juny 1997.