Facilitating Virtual Representation of CAD Data through a Learning Based Approach to Conceptual Database Evolution Employing Direct Instance Sharing

Awais Rashid[1], Peter Sawyer[1]
[1]Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
{marash, sawyer} @comp.lancs.ac.uk

Abstract. This paper presents a framework for a learning based approach to dynamically evolve the conceptual structure of a database in order to facilitate virtual representation of data in a CAD environment. A generic object model is presented which spans applications from a wide range of engineering design domains. The object model is complemented with a schema model. The object model and the schema model are justified through several sample cases depicting the mapping from the object model to the schema model.

1 Introduction

Object databases are considered to be well-suited to computer aided design applications which cannot be easily supported through the built-in data types in relational databases. In CAD applications, progressively more enhanced versions of a design object are created from existing multiple versions of the same. The very nature of CAD applications, however, not only requires the design objects to change but also the design templates. Therefore, not only the objects residing in the database but also the schema of the database needs to evolve when supporting CAD applications. The highly interactive nature of design applications further requires that any such schema changes be dynamic.

In this paper we present a framework for the application of learning techniques to dynamically evolve the conceptual structure of a database in order to facilitate virtual representation of data in a computer aided engineering design environment.

Rule induction and knowledge acquisition from databases [13], [15] are not new concepts. Work has been carried out both for knowledge discovery in databases [9], [16] and to allow the specification and implementation of autonomous reactive behavior in databases [3]. We, however, scope our approach to make use of the knowledge inherent in engineering design data and use this knowledge for database evolution through semi-autonomous learning in a manner that will facilitate virtual representation and manipulation of both the design objects and their templates.

2 The Object Model

This section presents the object model for our application. Data models for CAD applications have mainly been product specific [1]. The drawback for these models is that they are based on relevant information for the particular product and cannot be generalized to fit all or most of the CAD applications. Product models are mostly static in nature and don't incorporate a great deal of flexibility for dynamic evolution. Our object model, GOMCAD - Generic Object Model for CAD - is based on the

observation that objects are *molecular* in nature with each *molecule* consisting of more primitive molecules, the most primitive ones being the *atoms* [4]. Although CAD data is highly diverse in nature, still certain generic graphical objects can be identified that form the basic building blocks for virtual representation of data in most engineering applications. Consider figure 1, for instance, which provides an example from electronic circuit design. Shown in the figure is the virtual representation of a capacitor, which has been built using three cylinders.

Fig. 1. Virtual representation of a capacitor

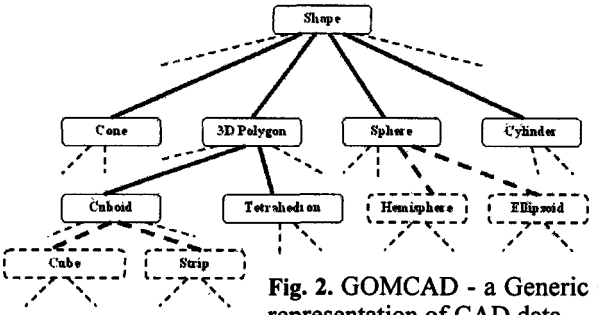

Fig. 2. GOMCAD - a Generic Object Model for virtual representation of CAD data

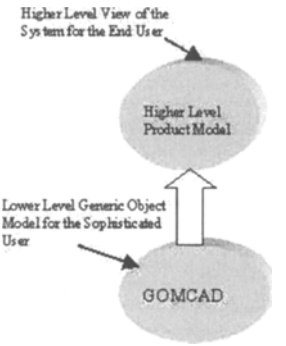

Fig. 3. Two level architecture

Our object model is presented in figure 2. GOMCAD provides inherent flexibility for dynamic evolution. As shown in figure 2 the *shape* objects that act as the basic building blocks are *cone, cuboid, tetrahedron, sphere* and *cylinder.* Note that the classes *shape* and *3D Polygon* are abstract classes. The solid lines between the objects show inheritance relationships while the dotted lines indicate that there can be further sub-classes. The thick dashed lines show derivation relationships. The objects shown dotted are obtained by applying, to the basic objects, one of six operations three of which are categorized as *linear* the other three being *non-linear*. The three linear operations are *scale*, *rotate* and *reflect* while the three non-linear operations are *cut, trim* and *shear*. These six operations can be used to create further atomic objects such as *cube, strip* (through the application of the *scale* operation to a *cuboid*), *hemisphere* (by applying the *cut* operation to a sphere) and *ellipsoid* (obtained by using the *shear* operation on a sphere). The *join* and *merge* operations can then be applied to the basic or derived atomic objects to form molecular objects which are further *joined* or *merged* to form higher level molecular objects.

GOMCAD is well suited for virtual representation of data from several engineering domains such as power plant design, electronic circuit design, mechanical engineering design, architecture, city & regional planning and chemical plant design. As shown in figure 3 above, GOMCAD can act as an underlying object model for defining higher level product models in order to facilitate virtual representation of data for a wide

range of engineering design applications at the same time providing end user isolation from the underlying database evolution process.

3 Conceptual Database Evolution (CDE)

This section describes our approach to conceptual database evolution and how it facilitates CAD applications. Work has been carried out previously to evolve, both statically and dynamically, the conceptual structure of a database. The approaches have been both passive [2], [10], [11], [12] and active [6], [7]. We take the knowledge-based approach suggested by [6], [7] and employ the following three general machine learning techniques for dynamic CDE.

- Learning from Instruction (LFI)
- Learning from Exception (LFE)
- Learning from Observation (LFO)

Historically, the database community has employed three fundamental techniques for modifying the conceptual structure of a database. In the first technique, known as schema evolution [2] [5] [6] [7], the database has one logical schema to which class definition and class hierarchy modifications are applied during the database evolution process. The second one is called schema versioning [11], which allows several versions of one logical schema to be created and manipulated independently. Class versioning [10] [12], the third technique, keeps different versions of each type and binds instances to a specific version of the type. Since our approach is purely in the context of CAD applications, triggering of the appropriate learning rules due to changes in the design modules will require changes to both the class definition and the class hierarchy. At the same time the previous structure of the design modules needs to remain intact should the user choose to revert to an earlier version of the design module. Our approach, therefore, superimposes schema evolution on class versioning and views conceptual database evolution as a composition of the following four components:

1. Class hierarchy evolution
2. Class versioning
3. Object versioning
4. Knowledge-base/rule-base evolution

From this point onwards we use the term *database evolution* to refer to any changes in the conceptual structure of the database. The evolution of the knowledge-base/rule-base must form part of the CDE process since we suggest that the knowledge-base resides within the OODBMS itself in order to guarantee consistency between the rules and the data and to achieve performance improvements [14]. We do not suggest the use of a front-end expert system shell [14] since we want the learning processes to make use of the rich semantics of CAD data. In addition, new knowledge will keep on entering the database due to the highly diverse nature of CAD data. Conceptual evolution of the knowledge-base/rule-base is therefore necessary.

3.1 The Schema Model

This section presents the schema model underlying our CDE framework presented in the following section. The conceptual database schema in our system is a fully connected directed acyclic graph (DAG) depicting the class hierarchy in the system (note that our system allows multiple inheritance). Each node in the DAG is a *class*

version derivation graph [8]. Each node in the class version derivation graph (CVDG) has the following structure:

- Reference(s) to predecessor(s)
- Reference(s) to successor(s)
- Pointer to the versioned class object
- Descriptive information about the class version such as creation time, creator's identification, etc.
- Pointer(s) to super-class(es) version(s)
- Pointer(s) to sub-class(es) version(s)
- A set of reference(s) to object version derivation graph(s)

The reason why each node of the CVDG must keep pointers to the super-class(es) and sub-class(es) can be observed from figure 4 which presents two nodes of the hierarchy graph along with their associated CVDGs. It should be noted that the dashed arrows inside each node represent the derivation path for the class versions. The solid lines show the conceptual inheritance relationship between the two classes while the dashed lines represent the actual inheritance relationship between various versions of these classes. Note that version *B1* of class *B* has as its super-class version *A3* of class *A* while both versions *B2* and *B3* inherit from the version *A2*. Conceptually, the sub-class can only inherit from the super-class (assuming that the class has only one super-class) but each version of the sub-class might be inheriting from a separate version of the super-class. Also, we can see that version *B2* of class *B* in figure 4 might be the result of changing its super-class to be version *A2* and not *A3*. Pointers to sub-classes need to be maintained to make sure that correct sub-class instances are substituted for the class under consideration. For example, only an instance of the sub-class version *B1* can be substituted where an instance of class version *A3* is required. This does not mean that instances associated with one class version are not compatible with the others. Later in this section we provide an underlying framework for converting instances from one class version to another.

Each node of a CVDG keeps a set of reference(s) to some object version derivation graph(s) (OVDG). Each OVDG node has the following structure [8]:

- Reference(s) to predecessor(s)
- Reference(s) to successor(s)
- Pointer to the versioned instance
- Descriptive information about the instance version such as creation time, creator's identification, etc.

Since an OVDG is generated for each instance associated with a class version, a set of OVDGs results when a class version has more than one instance associated with it. As a result a CVDG node keeps a set of references to all these OVDGs. Figure 5 above shows the structure of a CVDG node with references to two OVDGs.

Note that our schema model allows maintenance of a logical database schema while at the same time keeping track of the historical version derivation paths without storing the information within the versioned types and instances. In the context of CAD applications it will not normally be desirable to convert instances of one class version to another. However, in cases where such a behavior might be expected of the system, missing information must be added or additional information must be removed from

the instance version being manipulated. The resulting instance will then be stored in the database as a new instance version of the class version it was made compatible to. Note that this should not affect the older instance version from which the new instance version was derived.

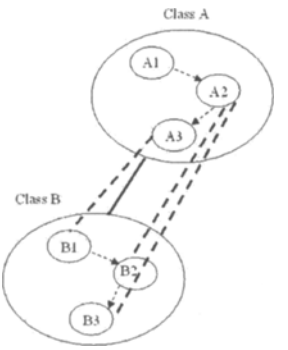

Fig. 4. A simple class hierarchy with super-class sub-class links between various nodes of the CVDGs

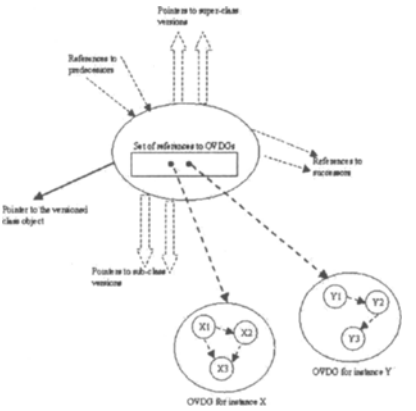

Fig. 5. A CVDG node with references to two OVDGs

It should also be noted that in our approach each instance version is associated with a particular class version and derivation of a new class version does not, by default, require instance versions associated with older class versions to be updated to form instance versions associated with the new class version. Any such action is performed on specific demand. We, therefore, have only one copy of each instance version in the system. As a result, instance versions are directly shared by all the class versions. This is known as *direct instance sharing* [11].

4 Mapping from the Object Model to the Schema Model

In this section we present several sample situations which depict the mapping from our lower level generic object model, GOMCAD, to the schema model through the application of the above mentioned learning techniques. We use the schema change taxonomy presented by [2] as a basis to further classify the four components of conceptual database evolution that we identified above. However, some of the cases, such as manipulating the shared value of an instance variable, in the taxonomy are specific to the ORION data model. We, therefore, support only the general cases identified by [2].

The learning processes are triggered when a design object is *checked in* to the database. Dependent upon the degree of modification of this design object, the LFO (learning from observation) process employs one of the following three rules in order to resolve the default type of evolution which the conceptual structure of the database undergoes:

- Any non-linear changes in the structure of any atomic or molecular object, such as those caused through the application of the *cut*, *trim* or *shear* operations, lead, by default, to the creation of a new version of the class with the object being bound to the newly created class version (addition of a CVDG node: Class Versioning)

- Any linear changes in the structure of any atomic or molecular object, such as those caused through the application of the *scale*, *rotate* or *reflect* operations, by default lead to the creation of a new version of the object (addition of an OVDG node: Object Versioning)
- Creation of higher level molecular objects, such as through *joining* and *merging* lower level molecular or atomic objects or both, or transition to lower level molecular objects because of the removal of molecular or atomic objects, by default, leads to changes in conceptual class hierarchy (changes to the DAG: Class Hierarchy Evolution)

Referring to figure 2 the above rules suggest that *cube*s and *strip*s (obtained through the application of the linear operation *scale* to a *cuboid*) will be treated as instances of *cuboid* while both *hemisphere* and *ellipsoid* (obtained by applying the non-linear operations *cut* and *shear* to a *sphere* respectively) will be treated as versions of the class *sphere*. However, the semantics of *cube* & *strip* and *hemisphere* & *ellipsoid* can be conceptually well understood if these are sub-classes of *cuboid* and *sphere* respectively. This implies that there can be certain situations where the default behavior described in the above three general rules might be undesirable in order to maintain the conceptual integrity of various classes in the system. Human guidance has, therefore, been incorporated into the system since there exists a trade-off between autonomy and versatility [9].

4.1 Example CDE Cases

For our example CDE cases we choose electronic circuit design as our application domain. The four evolution cases are presented below:

Case # 1: Creation of a new atomic type

Type of Change and Object: Linear change to an atomic object

Sample Situation: Suppose that the user takes an instance of the generic class *cylinder* and scales it with an intention to create a *wire*. When the modified object is checked in to the database the LFO process will, by default, consider that it be checked in as a new instance or instance version of the class *cylinder* depending on whether the object was newly created or checked out of the database. However, the user would like *wires* to be instances of a separate class. The user, therefore, instructs the system to create a new class *wire* and check the object in as an instance of that class. The existing knowledge in the knowledge-base informs the system that the new class to be created has the same properties as the class *cylinder*. With the user's approval, it therefore, checks the object in as a new instance of the newly created sub-class *wire* of the class *cylinder*. The whole evolution process is depicted in figure 6 below.

Learning Techniques Applied: LFO, LFI

Type and Semantics of Evolution: Class Hierarchy Evolution i.e. addition of a new DAG node. Also, by default, the addition of a new class creates a new CVDG node and the addition of a new instance creates a new OVDG node.

Case # 2: Creation of a new molecular type

Type of Change and Object: Creation of a molecular object from atomic objects subject to linear change

Sample Situation: Suppose that the user wants to create a capacitor. For this purpose the user will create a *cylinder* instance and *merge* with it two new instances of the

class *wire* in order to form a capacitor. When the newly created object is checked in to the database the LFO process observes a new molecular object and uses the knowledge in the knowledge-base to check for existing classes with possible similarities to the new object being checked in. Failing to find any such molecular classes the LFO process suggests creating a new sub-class - an aggregation of *cylinder* and *wire* - of the class *shape* and checking the new object in as an instance of the newly created class. Under instructions from the user the system creates a new sub-class *capacitor* of the class *shape* and checks the new object in as an instance of the newly created *capacitor* class. Note that the cylinder instance that forms part of the capacitor will automatically be stored as an instance of the class *cylinder* while the two associated *wire* instances will be stored as new instances of the class *wire* since these are new instances and have undergone only linear changes. The evolution process is shown in figure 7 below.

Learning Techniques Applied: LFO, LFI

Type and Semantics of Evolution: Class Hierarchy Evolution i.e. addition of a new DAG node. Also, by default, the addition of a new class creates a new CVDG node and the addition of a new instance creates a new OVDG node. Also the storage of new atomic instances results in changes to the CVDG nodes these instances are associated with.

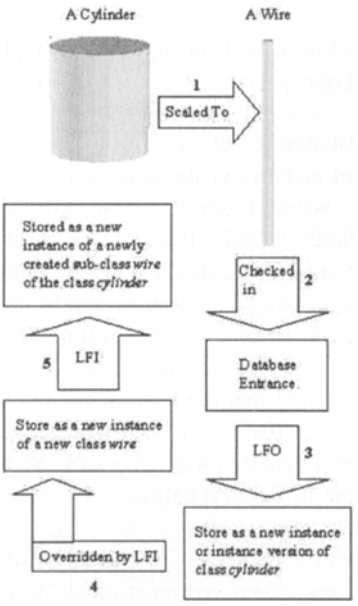

Fig. 6. The database evolution process for evolution case # 1

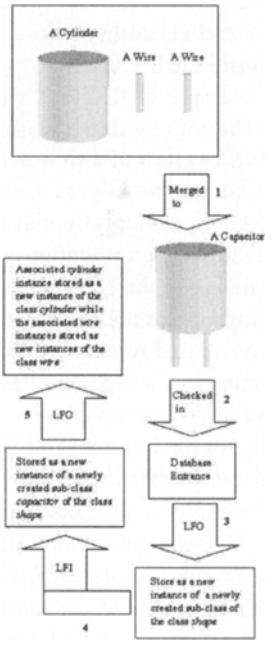

Fig. 7. The database evolution process for evolution case # 2

Case # 3: Creation of a new molecular type similar in structure to an existing one

Type of Change and Object: Creation of a molecular object from atomic objects subject to both linear and non-linear changes

Sample Situation: Suppose that the user wants to create a resistor. For this purpose the user will create a *cylinder* instance and *scale* and *rotate* it so that it lies horizontally. Then the user will create two instances of class *wire* and *scale* and *shear* them in order to form terminals for the resistor. The above three instances will then be *merged* together to form a resistor. When the newly created object is checked in to the database the LFO process observes a new molecular object and uses the knowledge residing in the knowledge-base to check for any existing classes with possible similarities. Since a resistor contains the same components as a capacitor but with some linear and non-linear changes the LFO process suggests creating a new version of the class *capacitor* and storing the new object as an instance of this class version. However, a resistor is conceptually different from a capacitor so the LFI process guides the system to create a new class *resistor* and store the object as an instance of this class. The system, finding no other similarities to the *resistor* class within the existing classes, with the user's approval creates a new sub-class *resistor* of the class *shape* and stores the new object as an instance of this newly created class. Note that the cylinder instance that forms part of the resistor will automatically be stored as an instance of the class *cylinder* since it has undergone a linear change only (*rotation*). On the other hand the two associated *wire* instances have undergone non-linear changes. However they are mere reflections of each other. The LFO process will, therefore, create a new class version of the class *wire* and associate both the instances with this new version of the class.

Learning Techniques Used: LFO, LFI

Type and Semantics of Evolution: Class Hierarchy Evolution i.e. addition of a new DAG node. Also, class versioning takes place since a new CVDG node has been added to the atomic class *wire*. Also, by default, the addition of a new class creates a new CVDG node and the addition of a new instance creates a new OVDG node. Figure 8 below presents the conceptual database schema after the evolution. The database schema before the evolution can be seen in figure 10.

Case # 4: Generalization of an existing and a newly created molecular type into a super-type

Type of Change and Object: Creation of a molecular object from atomic objects subject to linear change

Sample Situation: Suppose that the user wants to create another form of capacitor but one that is flat rather than cylindrical. For this purpose the user creates an instance of the generic class *cuboid* and scales it to form a strip. The user then creates two new instances of the class *wire* and *merges* them together with the strip in order to form the flat capacitor. When the newly created object is checked in to the database the LFO process detects a new molecular object and finding no existing class to which the newly created object can belong to, suggests creating a new sub-class of the class *shape* and storing the newly created object as an instance of this class. With the user's approval the LFO process goes ahead to create a new class *capacitor*. However, the class *capacitor* already exists in the system. Here the LFE (learning from exception) process suggests combining the common features of both the new and old type of *capacitor* into a super-class with the specialized features incorporated into two sub-classes of this new super-class. With the user's guidance the system then renames the

existing *capacitor* class to *cylindrical capacitor* and adds a newly created class *capacitor* as its super-class. Note that the new *capacitor* class will be an abstract class. The system also creates another sub-class of *capacitor* named *flat capacitor*. The system then stores the newly created object as an instance of the class *flat capacitor*. Also note that this will result in the common features of both *cylindrical capacitor* and *flat capacitor* being combined into the new *capacitor* class and any instances associated with the older *capacitor* class will now be associated with the class *cylindrical capacitor*. Note that the cuboid instance that forms part of the newly created flat capacitor will automatically be stored as an instance of the class *cuboid* while the two associated *wire* instances will be stored as new instances of the class *wire* since these are new instances and have undergone only linear changes.

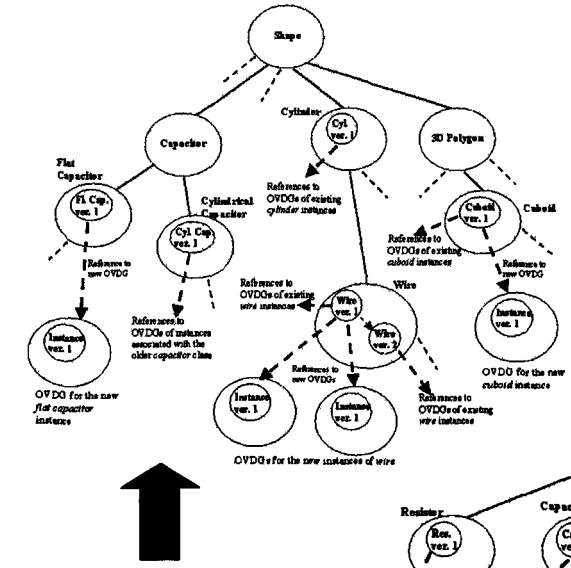

Fig. 8. Case #3: After Evolution

Fig. 9. Case #4: After Evolution

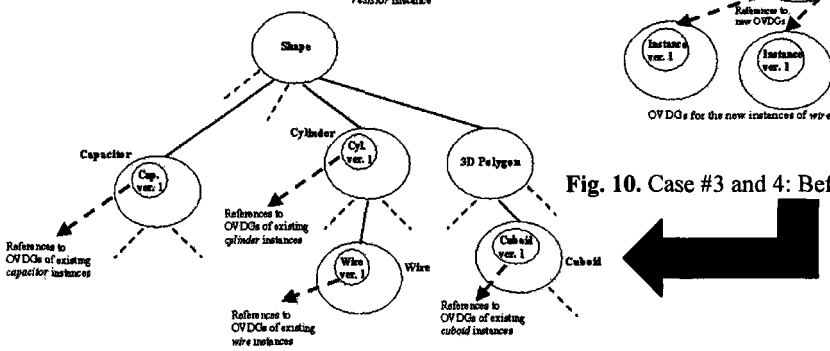

Fig. 10. Case #3 and 4: Before Evolution

Learning Techniques Used: LFO, LFI, LFE
Type and Semantics of Evolution: Class Hierarchy Evolution i.e. name of an existing DAG node has been changed while both a new leaf and non-leaf node have been added. Also, by default, the addition of a new class creates a new CVDG node and the addition of a new instance creates a new OVDG node. Figure 9 above shows the conceptual database schema after the evolution. The database schema before the evolution can be seen in figure 10.

5 Summary and Conclusions

We have presented a semi-autonomous learning approach to dynamically evolve the conceptual structure of an object oriented database. However, unlike earlier work on database evolution, our approach is specifically in the context of CAD applications. We observe that despite the highly diverse and complex nature of CAD data, several generic graphical objects can be identified that act as the basic building blocks in various engineering design applications. We have complemented our generic object model with a schema model and have provided several example evolution cases depicting how our learning approach makes use of the knowledge inherent in CAD data in order to facilitate dynamic database evolution. The work, at present, is in the early stages of implementation with the overall system architecture being designed. We are building our database evolution system on top of the OODBMS *O2* from *O2 Technology*.

References

1. Anumba, C. J.: Data Structures and DBMS for CAD Systems. Advances in Engineering Software, Vol. 25, No. 2-3, 1996, pp. 123-129
2. Banerjee, J. *et al.*: Data Model Issues for Object-Oriented Applications. ACM Transactions on Office Information Systems, Vol. 5, No. 1, Jan. 1987, pp. 3-26
3. Dittrich, K. R. *et al.*: The Active Database Management System Manifesto: A Rulebase of ADBMS Features. Proceedings of the 2nd Workshop on Rules in Databases, Sept. 1995, Lecture Notes in Computer Science, T. Sellis (ed.), Vol. 985, pp. 3-20
4. Harder, T. *et al.*: PRIMA - a DBMS Prototype Supporting Engineering Applications. Proceedings of the 13th International Conference on Very Large Databases, Sept. 1987, pp. 433-442
5. Kim, W. *et al.*: Architecture of the ORION Next-Generation Database System. IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990, pp. 109-124
6. Li, Q. & McLeod, D.: Conceptual Database Evolution through Learning. Object Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Edited by: Gupta, R. & Horowitz, E., c1991 Prentice Hall Inc., pp. 62-74
7. Li, Q. & McLeod, D.: Conceptual Database Evolution through Learning in Object Databases. IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No.2, April 1994, pp. 205-224
8. Loomis, M. E. S.: Object Versioning. Journal of Object Oriented Programming, Jan. 1992, pp. 40-43
9. Matheus, C. J. *et al.*: Systems for Knowledge Discovery in Databases. IEEE Transactions on Knowledge and Data Engineering, Vol.5, No.6, Dec. 1993, pp.903-913
10. Monk, S. & Sommerville, I.: Schema Evolution in OODBs Using Class Versioning. SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22
11. Ra., Y.-G. & Rundensteiner, E. A.: A Transparent Schema-Evolution System Based on Object-Oriented View Technology. IEEE Transactions on Knowledge and Data Engineering, Vol.9, No.4, July/Aug.1997, pp.600-624
12. Skarra, A. H. & Zdonik, S. B.: The Management of Changing Types in an Object-Oriented Database. Proceedings of the 1st OOPSLA Conference, Sept. 1986, pp.483-495
13. Smyth, P. & Goodman, R. M.: An Information Theoretic Approach to Rule Induction from Databases. IEEE Transactions on Knowledge and Data Engineering, Vol.4, No.4, Aug. 1992, pp.301-316
14. Stonebraker, M.: The Integration of Rule Systems and Database Systems. IEEE Transactions on Knowledge and Data Engineering, Vol.4, No.5, Oct. 1992, pp.415-423
15. Yasdi, R.: Learning Classification Rules from Database in the Context of Knowledge Acquisition and Representation. IEEE Transactions on Knowledge and Data Engineering, Vol.3, No.3, Sept. 1991, pp.293-306
16. Yoon, J. P. & Kerschberg, L.: A Framework for Knowledge Discovery and Evolution in Databases. IEEE Transactions on Knowledge and Data Engineering, Vol.5, No.6, Dec. 1993, pp.973-979