Temporal Object-Oriented Data Model for the Schema Modification

Woo Saeng Kim Duk Chul Chang Tae Young Lim Young Ho Shin
Department of Computer Science, Kwangwoon University, Seoul, Korea
woosaeng@dblab.kwangwoon.ac.kr tylim@selab.kwangwoon.ac.kr

Abstract

A temporal database management system is being considered as an important DBMS in new database applications such as engineering databases, VLSI, and CAD. Much researche on temporal databases, temporal query languages and temporal modeling techniques have been proposed. However, thus researche has not considered the schema modification problem in a temporal database yet. In this paper, we propose a new methodology that supports the schema modification in a temporal object-oriented database. The class and instance structure of the existing object-oriented data model are extended to maintain additional schema modification information from which the previous schema can be recovered effectively. We also show how a temporal query is processed efficiently in a temporal object-oriented database where its schema was modified.

1 Introduction

Time has a significant meaning on the real world entity which changes continuously, and an entity is generally understood in terms of time. Such time plays a central role in many new database applications. For example, engineering databases must maintain different versions of complex objects to track the process of the object through the design, analysis, and manufacturing stages. In addition, administrative and legal systems must record changes in policies or laws over time. Existing conventional databases lack the ability to preserve the properties of the time-varying entity. This results in a temporal database that has a mechanism to manage past and present data by adding time information to the existing conventional database.

Initially, researchers extended relational models to temporal models by incorporating temporal dimension and modeling temporal data[6,8,10]. They also introduced temporal query languages which were based on the temporal data model [2,7]. Recently, some researchers have focused on managing supporting and modeling a complex object and non-uniform data [3,9,11]. In design applicatios such as engineering database, VLSI design,and CAD, not only a time notion but also the schema modification is required. Thus, the ability of the schema modification is essential to object-oriented databases which are suitable for design applications. There two different techniques: single-schema modification and schema versioning. Schema modification methods in object-oriented database were discussed in [1,4,5]. However, a schema modification problem in a temporal object-oriented database has not been considered yet.

Therefore, in this paper, we propose a new methodology that supports the schema modification in a temporal object-oriented database. In our model, extra information such as time is added to the existing object-oriented data model in order to maintain historical information about class name, attribute, method, and superclasses/subclasses relationship. A query requering the past information can be processed after the previous schema is recovered by referencing this schema information. Since our method is based on a single-schema modification, it minimizes the storage requirement.

The remainder of this paper is organized as follows. In chapter 2, we review related studies on temporal databases and schema modification methods. A methodology that supports the schema modification in a temporal object-oriented database is presented in chapter 3. In chapter 4, we examine how a temporal query after the schema modification is correctly processed through several examples. Finally, we discuss a conclusion and future research directions in chapter 5.

2 Related Works

In recent years, the technology in storage media has made remarkable progress and has allowed large quan-

Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995
© World Scientific Publishing Co. Pte Ltd

tities of temporal data to be stored and processed efficiently. This motivated many researchers to study temporal databases. In [2], the proposed model allowed temporal elements and historical relations. The historical algebra of [10] allowed only one type of the historical relation and this model used the algebra for interpreting queries expressed in the Time-by-Example language. Among methods using tuple time-stamping, the temporal relational model and its associated algebra were defined primarily to support a time-oriented extension to SQL (TSQL) in [6]. The data model of [7] considered the state- and event-oriented view of the real world and supported only the real world time, named valid time. The expansion and contraction of data between the normalized and non-normalized structures required in the attribute time-stamping techniques cause a serious time overhead in processing temporal data; whereas, in tuple time-stamping, data redundancy is the major shortcoming.

Motivated by the above problems of the current time-stamping techniques and the emergence of object-oriented database technologies, new studies which managed the temporal data by utilizing the advantages of an object-oriented technique were made. In [9], they took an object-oriented approach and knowledge rules to capture special temporal requirements. In [11], they extended the functional object-oriented database and developed a temporal object-oriented model that supports the modeling and manipulation of complex temporal or versioned objects. In [3], they presented the model that supported the temporal complex objects by extending complex object model.

Another direction of research supporting temporal data is the schema modification techniques which are classified into single-schema modification and schema versioning [1,5]. A schema versioning is the versioning of a single logical schema, thus making different views of the database visible to the users under different versions of schema. However, this technique requires a huge storage overhead to maintain different versions. In [5], they defined a model of changes to the schema and the implementation of the schema changes dynamically. A taxonomy of schema changes includes changes to the contents of a class and the superclass/subclass relationship. The model consists of a set of invariants and a set of rules. The sets of invariants provide the basis for the specification of the semantics of schema changes. The schema can be dynamically modified if a schema is changed with preserving the invariants and rules.

3 Temporal object-oriented schema modification

In this chapter, we explain the structure of class and instance that support the schema modification in a temporal object-oriented database. We also explain a technique that can effectively recover the information prior to the change even after the schema is modified.

3.1 Structure of the class and instance for the schema modification

The new structures of class and instance are required because the existing object-oriented data model cannot support the schema modification in a temporal object-oriented database. Thus, the class and instance of this paper are based on the extended structures defined in Figure 1 and 2.

As presented in Figure 1, a class structure contains properties of the class such as class name, attributes, methods, superclass list, and subclass list. In this paper, new information such as FROM, TO, PRE, and NEXT are added to each property of the class in order to keep track of the information about the schema modification over time. Among these, the FROM and TO fields maintain the addition and deletion time of each property. In PRE and the NEXT fields, the before- and after-name are registered whenever the name of a property is changed except the superlist and the sublist.

	C	Attributes	Methods	superlist	sublist
FROM		...	...	...	...
TO					
PRE					
NEXT					

Figure 1 : Extended class structure to the class C

Figure 2 shows the structure of an instance. OID refers to the identifier of an instance. Attr_value_1 and Attr_value_N mean the value of the first and Nth attribute of an instance. BEGIN_TIME refers to the creation time of an instance and END_TIME refers to the deletion or modification time of an instance[3].

(OID Attr_value_1..Attr_value_N BEGIN_TIME END_TIME)

Figure 2 : Extended instance structure

3.2 Changes to the structure of class and instance along the schema modification

This section explains the technique by which information prior to the schema modification can be recovered effectively. Figure 3 depicts a university schema

graph example used in this chapter. Suppose that this schema graph was created on January 1, 1980.

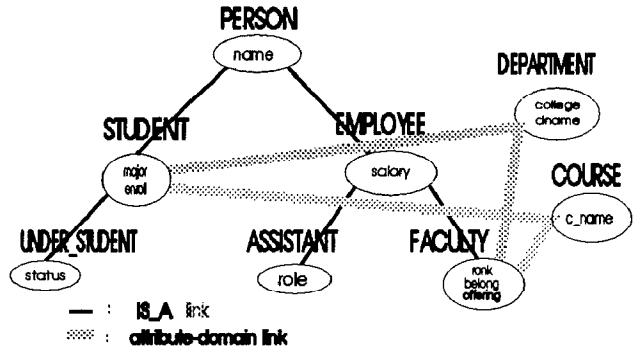


Figure 3 : Schema graph of an university database

3.2.1 Changes to the contents of a class

1) Adding a new attribute(or method) to a class

When a new attribute is added to a class in the class-hierarchy, the attribute must be inherited by all subclasses of the class. In general, suppose that a new attribute Attr_N+1 is added to the class C. If the addition time of the new attribute was 90.01.01, then its name is added to the class and its addition time is recorded in the corresponding FROM field as shown in Figure 4. When information about the class C and its subclasses prior to 90.01.01 is required, Attr_N+1 is excluded from the query result by using time information in the FROM field.

	C	Attr_1	...	Attr_N	Attr_N+1	Methods	superlist	sublist
FROM	80.01.01	80.01.01	...	80.01.01	90.01.01	...	...	...
TO	-	-		-	-			
PRE								
NEXT								

Figure 4 : Adding a new attribute Attr_N+1 to the class C

For those instances that do not include Attr_N+1, the values of BEGIN_TIME are between 80.01.01 and 90.01.01, and those of END_TIME are greater than 80.01.01. Whereas, for those instances that include Attr_N+1, the values of both BEGIN_TIME and END_TIME are greater than 90.01.01.

2) Dropping an existing attribute (or method) from a class

When an attribute is dropped from a class, it is dropped recursively from the subclasses that inherited it. Further, instances of those classes lose their values of that attribute. In a temporal OODB, however, the dropped attribute in a class and its subclasses should not be deleted physically because old information prior to the change should be maintained. In order to maintain old information, we use the technique in which an attribute is dropped logically by recording its dropping time in the corresponding TO field.

Figure 5 depicts the case in which the Nth attribute in class C is dropped on 90.01.01. The Attr_N is not physically but logically dropped by recording its dropping time in the corresponding TO field. Therefore, even after the deletion of Attr_N, it can be recovered by referring the corresponding FROM and TO fields of class C in order to process the query that requests information prior to 90.01.01.

	C	Attr_1	...	Attr_N	Methods	superlist	sublist
FROM	80.01.01	80.01.01	...	80.01.01	...	...	...
TO	-	-		90.01.01			
PRE							
NEXT							

Figure 5 : Dropping an attribute Attr_N from the class C

In terms of instances, for those instances that include Attr_N, the values of BEGIN_TIME and END_TIME are between 80.01.01 and 90.01.01. For those instances that do not include Attr_N, the values of both BEGIN_TIME and END_TIME are greater than 90.01.01.

3) Change to the name of an attribute (or method) of a class

Even if the name of a class, attribute, or method is changed, the previous name should be treated equally as the current name during a query. In this paper, we use PRE and NEXT fields to solve this problem. Suppose that Attr_N is changed to Attr_N+1 on 90.01.01. Then, after the new attribute Attr_N+1 is added to the class C, NEXT field of Attr_N and PRE field of Attr_N+1 reference Attr_N+1 and Attr_N, respectively as shown in Figure 6. In this way, the two attributes are treated equally as the same attribute. In addition, the changed time of 90.01.01 is recorded in the TO field of Attr_N to have a closed effect and the beginning time of 90.01.02 is recorded in FROM field of Attr_N+1, which is the result of adding a minimum unit of time granularity to the changed time. Therefore, in the case that either Attr_N or Attr_N+1 is used in a query, both attributes are treated as the same attributes by using the FROM, TO, PRE, and NEXT fields of the class.

For those instances that use Attr_N, the values of BEGIN_TIME are between 80.01.01 and 90.01.01 and those of END_TIME are greater than 80.01.01. For those instances that use Attr_N+1, the values of both BEGIN_TIME and END_TIME are greater than

90.01.02.

	C	Attr_1	...	Attr_N	Attr_N+1	Methods	superlist	sublist
FROM	80.01.01	80.01.01	...	80.01.01	90.01.02	...	...	...
TO	-	-		90.01.01	-			
PRE					Attr_N			
NEXT				Attr_N+1				

Figure 6 : Change to the name of Attr_N into Attr_N+1 within the class C

3.2.2 Changes to the superclass/subclass relationship

1) Adding (or removing) the existing class A into a superclass of the class B

When the existing class is added or dropped as the superclass of another class, the relationship information between these two classes should be maintained in order to recover the previous schema. In general, when class A is added as a superclass of class B, the name of class B is recorded in the sublist field of class A and its addition time is recorded in the corresponding FROM field. In addition, the name of class A is recorded in the last superlist field of class B and its addition time is recorded in the corresponding FROM field.

For example, suppose that the class STUDENT is added as a superclass of the class ASSISTANT on 90.01.01 in the university schema graph. The necessary process for this case is shown in Figure 7. The name of the class STUDENT is recorded in the last superlist field of the class ASSISTANT and its addition time of 90.01.01 is recorded in the corresponding FROM field. Next, the name of the newly added subclass ASSISTANT is recorded in the sublist field of the class STUDENT and its addition time of 90.01.01 is recorded in the corresponding FROM field. In this way, during a query the class ASSISTANT is treated as the class which has only one superclass EMPLOYEE during the period of 80.01.01 to 89.12.31, and is also treated as the class which has two superclasses after 90.01.01.

	ASSISTANT		superlist	
			EMPLOYEE	STUDENT
FROM	80.01.01	..	80.01.01	90.01.01
TO	-		-	-
PRE				
NEXT				

	STUDENT	-	superlist	sublist	
			PERSON	UNDER_ STUDENT	ASSISTANT
FROM	80.01.01	..	80.01.01	80.01.01	90.01.01
TO	-		-	-	-
PRE					
NEXT					

Figure 7 : Adding a class STUDENT as a superclass of a class ASSISTANT

In terms of instances of the class ASSISTANT, for the instances that only inherit attributes and methods from the class EMPLOYEE, the values of BEIGN_TIME are between 80.01.01 and 90.01.01 and the values of END_TIME are greater than 80.01.01. On the other hand, the values of BEGIN_TIME and END_TIME of instances that inherit attributes and methods from both the class EMPLOYEE and STUDENT are greater than 90.01.01.

2) Adding a new class into the class hierarchy

There are two cases when new class is added: when the superclasses are explicitly specified or not specified. When the multiple superclasses are specified, their names are recorded in the superlist fields of the newly added class and their addition times are recorded in the corresponding FROM fields. Also, the new addition time is recorded in the corresponding FROM field of each attribute and method of the new class. Also, the name of a newly added class is recorded in each sublist field of the specified superclasses and its addition time is recorded in the corresponding FROM fields. If no superclasses are specified for a newly added class, the default root class, CLASS, is recorded as the superclass of the new class.

For example, when the class GRADUATE_STUDENT is added as a subclass of the class STUDENT in the University Schema Graph on 90.01.01, the updated structures of these classes are shown in Figure 8. The name of the class STUDENT is recorded in the superlist field of the class GRADUATE_STUDENT, and its addition time 90.01.01 is recorded in the corresponding FROM field. Next, the name of the class GRADUATE_ STUDENT is recorded in the sublist field of the class STUDENT and its addition time is recorded in the corresponding FROM field. After this process is completed, the class GRADUATE_STUDENT becomes valid in the class hierarchy from 90.01.01. Therefore, even after the class GRADUATE_STUDENT is added, if a query requires information prior to 90.01.01, information about the class GRADUATE_STUDENT is excluded from the results of the query by referring to the FROM field of the class GRADUATE_STUDENT or the class STUDENT.

	STUDENT	..	superlist	sublist	
			PERSON	UNDER_ STUDENT	GRADUATE _ STUDENT
FROM	80.01.01		80.01.01	80.01.01	90.01.01
TO	-		-	-	-
PRE					
NEXT					

	GRADUATE _STUDENT	degree	..	superlist
				STUDENT
FROM	90.01.01	90.01.01		90.01.01
TO	-	-		-
PRE				
NEXT				

Figure8: Adding a new class GRADUATE_STUDENT as a subclass of the class STUDENT

All the values of BEGIN_TIME and END_TIME of the instances of the class GRADUATE_STUDENT are greater than 90.01.01.

3) Dropping an existing class from the class hierarchy

Dropping an existing class can be classified into two categories: dropping a middle class or dropping a leaf class in the class hierarchy. Suppose that class B and C are the subclasses of class A and class S is the superclass of class A. In order to drop class A, the following steps are required. First, the connection with class B and class C is logically removed by recording the dropping time in the corresponding TO fields of all the sublist fields of class A and of the corresponding superlist fields of class B and class C. Second, the connection with superclass S is logically removed by recording the dropping time in the corresponding TO fields of the superlist field of class A and the sublist field of class S which contains 'A'. Third, class A is logically removed from the class hierarchy by recording the dropping time only in the corresponding TO field of the name of class A. Finally, the name of class S is recorded in the superlist fields of both class B and C and its connection time is recorded in the corresponding FROM field. Also, the names of the class B and C are recorded in the sublist fields of class S and their connection times are recorded in the corresponding FROM fields. For the leaf class, of the above steps, only the following two steps are needed: disconnecting the relationship with its superclasses and dropping the class itself.

For example, suppose that the class STUDENT is dropped on 90.01.01. The updated structures of the class STUDENT, UNDER_STUDENT and PERSON are depicted in Figure 9. First, the relationships between the class STUDENT and its superclasses (and subclasses) are logically disconnected by recording its dropping time in the corresponding TO fields of the superlist (and the sublist). Next, the dropping time of 90.01.01 is recorded in the corresponding TO field of the class name 'STUDENT' to remove the class STUDENT logically. Next, the disconnection time of 90.01.01 is recorded in the TO field of the superlist field 'STUDENT' of the class UNDER_STUDENT. In addition, a newly connected superclass name of PERSON is added and its connection time of 90.01.02 is recorded in the corresponding FROM field of the class UNDER_STUDENT as shown in Figure 9-(a) and (b).

(a)

	STUDENT	major	superlist	sublist
			PERSON	UNDER_STUDENT
FROM	80.01.01	80.01.01	80.01.01	80.01.01
TO	90.01.01		90.01.01	90.01.01
PRE				
NEXT				

(b)

	UNDER_STUDENT	status	superlist	
			STUDENT	PERSON
FROM	80.01.01	80.01.01	80.01.01	90.01.02
TO	-	-	90.01.01	-
PRE				
NEXT				

Figure 9-(a),(b) : Dropping a class STUDENT from the class-hierarchy

The class PERSON logically disconnects the relationship with its subclass STUDENT by recording the dropping time of 90.01.01 in the corresponding TO field of the sublist field 'STUDENT'. It also registers the class UNDER_STUDENT in its sublist field as a new subclass and records the connection time of 90.01.02 in the corresponding FROM field. The above process is summarized in Figure 9-(c). After the process, when a query requires information prior to 90.01.01, the dropped class STUDENT can be recovered by referring both the sublist fields of the class PERSON and the superlist fields of the class UNDER_STUDENT.

(c)

	PERSON	name	superlist	sublist		
			CLASS	STUDENT	EMPLOYEE	UNDER_STUDENT
FROM	80.01.01	80.01.01	80.01.01	80.01.01	80.01.01	90.01.02
TO	-	-	-	90.01.01	-	-
PRE						
NEXT						

Figure 9-(c) : Dropping a class STUDENT from the class-hierarchy

From the perspective of instances of the class STUDENT, the values of BEGIN_TIME and END_TIME are between 80.01.01 and 90.01.01.

4 Temporal query processing after the schema modification

In this chapter, we will briefly explain the structure of the temporal object-oriented query language. We then examine how a query is processed in a temporal object-oriented database that supports the schema modification.

4.1 Structure of a temporal query language

The following temporal query language is an extension of the existing query lanugage to process temporal database information [3,8]. As a major difference to the existing query language, this temporal query language contains temporal functions such as AT, DURING, or CORRESPONDING. We can retrieve various kinds of historical information by using these temporal functions. The general structure of a temporal query language is as follows:

```
SELECT  target clause
FROM    range clause
WHERE   qualification clause
[AT|DURING|CORRESPONDING] temporal condition
```

The syntax of the temporal query language consists of four main clauses. The SELECT clause is used to spec-

ify the attributes to be retrieved. The FROM clause is used to describe the classes to work on. The WHERE clause is used to specify a qualification condition as a boolean combination of predicates. The temporal functions such as AT, DURING, or CORRESPONDING may be used to specify the temporal database of interest. In the AT clause, we express a time-point which indicates a point of time like 't1' as a temporal condition. A time-interval can be specified in the DURING clause by a specific time reference. The keyword DURING is followed by an explicit time interval specification of the form [t1, t2]. And there 't1' and 't2' are two time points and 't1' is less than 't2' or equal to 't2'. On the other hand, a time-interval may be specified in the CORRESPONDING clause by a data reference with a combination of predicates instead of specification of an explicit time.

4.2 Example of the schema modification

In this section, an example of the schema modification based on the university schema graph in Figure 3 is presented. The structures of some classes and instances after this schema modification are also illustrated. The following is the schema modification example:

1) An attribute 'type' was added to the class STUDENT on 90.01.01.

2) The exisiting class STUDENT was added as a superclass of the class ASSISTANT on 91.01.01.

3) The class GRADUATE_STUDENT with an attribute 'degree' was added as the subclass of the class STUDENT on 92.07.01.

4) The class UNDER_STUDENT was dropped on 92.12.30.

5) The attribute 'rank' of the class FACULTY was changed to 'position' on 93.01.01.

The schema graph and the structures of some classes after the change are depicted in Figure 10 and 11.

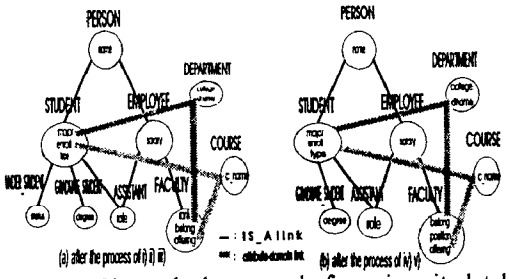


Figure 10: Changed schema graph of an university database

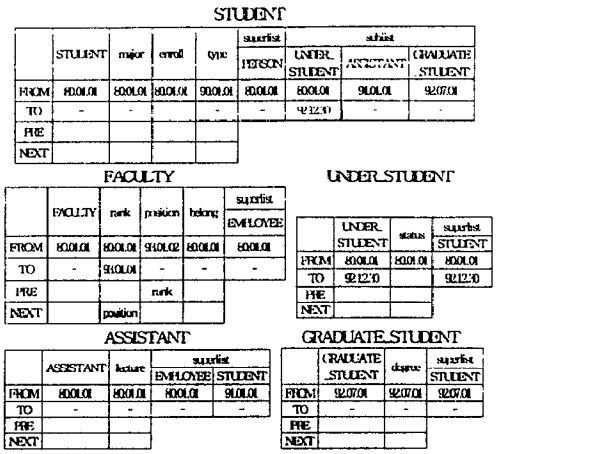


Figure 11 : Class structures after the schema modification

We assume that Figure 12 depicts instances of some classes, which have been created since January 1, 1980. In order to avoid data redundancy, the symbol '#' is introduced to mark unchanged attributes' values during instance evolution [9]. The symbol '-' in the END_TIME field of an instance means that the instance has not been closed yet. An attribute value beginning with symbol '@' refers to the identifier of another instance.

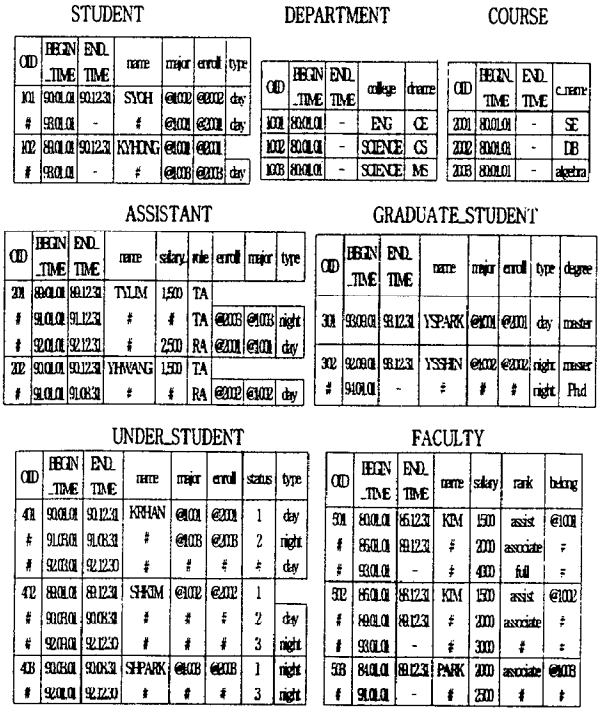


Figure 12 : Instances of the university database

4.3 Example of a temporal query processing

In this section, we show that a temporal query on our temporal object-oriented data model is processed correctly in spite of the schema modification. For the purpose of processing a query more efficiently, we assume that the BEGIN_TIME field of instances are indexed.

The first example explains how the query on the instances created after addition or deletion of attributes is processed.
Example 1. "What are the names of students who was in the daytime school and took an 'algebra' course during the period of 85.01.01 to 94.06.30 ?" The query language for this example is as follows:

```
SELECT STUDENT.name
FROM STUDENT*[1]
WHERE (major.c_name = "algebra"
        and type = "day" )
DURING [85.01.01, 94.06.30]
```

The system first examines the class STUDENT and recognizes that the attribute 'type' has been added only after 90.01.01. The sytem then processes not only instances which include the attribute 'type' but instances which do not include the attribute 'type' by referencing temporal infomation stored in the class STUDENT, UNDER_STUDENT, GRADUATE_STUDENT, and ASSISTANT. Through the recorvered information, the system only searches those instances whose BEGIN_TIME is greater than 90.01.01 and less than 94.06.30 which is the latter time condition specified in DURING clause. Among those students, 'KYHONG' and 'KRHAN' were daytime students and took an 'algebra' course.

The following example shows how a temporal query is processed on the instances whose class is added as an subclass of the existing class.
Example 2. "How many assistants attended the 'CS' Department when the assistant 'TYLIM' worked as a TA ?". We write the query as the following:

```
SELECT COUNT(UNIQUE(ASSISTANT.name))
FROM ASSISTANT
WHERE (major.dname = "CS" )
CORRESPONDING
        (ASSISTANT.name = "TYLIM" and
        ASSISTANT.role = "TA" )
```

The system recognizes that the class ASSISTANT was added as a subclass of the class STUDENT after 91.01.01 by examining the class structure ASSISTANT. This informs the system that only those instances whose BEGIN_TIME is greater than 91.01.01 include the attribute 'major'. Next, a valid time-interval is evaluated by using the CORRESPONDING clause which specifies a temporal condition by a data reference. The system finds that the assistant 'TYLIM' worked as a 'TA' during the period of [89.01.01, 89.12.31] and [91.01.01, 91.12.31] by examining the instances of the class ASSISTANT. Thus, the system searches only those instances of the class ASSISTANT, whose BEGIN_TIME is between 91.01.01 and 91.12.31. Among those students, only 'YHWANG' attended the 'CS' Department. Thus, one is returned as the query result.

The next example shows how a temporal query language is processed when a new class is added to the class hierarchy.
Example 3. "What are the names of the master students who took a 'SE' course during the period of 85.07.01 to now." A query language for the example is as follows:

```
SELECT GRADUATE_STUDENT.name
FROM GRADUATE_STUDENT
WHERE ( major.c_name = "SE"
        and degree = "master" )
DURING [ 85.07.01, - ]
```

The system discovers that the class GRADUATE_STUDENT was added in the class hierarchy after 92.07.01. Thus, all the instances of the class GRADUATE_STUDENT are searched, because they satisfy the time condition specified in the DURING clause. Only 'YSPARK' student took a 'SE' course when he was a master student.

The next example shows how the query on the class which is already dropped from the class hierarchy can be processed.
Example 4. "What are the names and types of the undergraduate students who took a 'DB' course when Professor 'KIM' worked for the CS Department?" The query language for the example is as follows:

```
SELECT UNDER_STUDENT.name,UNDER_STUDENT.type
FROM UNDER_STUDENT, FACULTY
WHERE UNDER_STUDENT.major.c_name = "DB"
CORRESPONDING
        (FACULTY.name = "KIM" and
        FACULT.belong.dname = "CS")
```

The UNDER_STUDENT class was deleted on 92.12.30. Thus, the system should recover the class UNDER_STUDENT in order to process the query

[1] If the name of the user specified class is followed by the operator '*', the query applies to all classes in the hierarchy rooted at the class.

correctly. After that, the system can recognize that the class UNDER_STUDENT had been existed from 80.01.01 to 92.12.30. Through this information and the instances of the class FACULTY, the system can obtain the time intervals of [86.01.01,89.12.31] and [93.01.01, -] as the period when professor 'KIM' worked for the CS Department. The instance 'SHKIM' of the class UNDER_STUDENT took the 'DB' course during the period of 86.01.01 to 89.12.31. But only his name except the type is actually returned as the query result because the attribute type was not exsisted at that time.

We also illustrate how the temporal query could be processed when some of the attribute names of a class is changed.
Example 5. "What are the names of the associate professors whose salary were more than 2500 dollars during the period of 80.01.01 to now ?" The query is as follows:

```
SELECT FACULTY.name
FROM FACULTY
WHERE ( position = "associate"
and salary >= "2500" )
DURING [ 80.01.01, - ]
```

By referencing the class FACULTY, the system realized that the attribute 'rank' was changed to the attribute 'position' on 93.01.01. Thus, these two attributes are treated as the same attributes. Next, the system searched the instances of the class FACULTY, whose BEGIN_TIME is greater than 80.01.01 to satisfy the time condition specified in the DURING clause. Among those instances, KIM and PARK are the associate professors who earn more than 2500 dollars. Thus, these two professors are returned as the query result.

5 Conclusions

Historical data is an important part of many database applications. This motivates many researches in temporal databases toward the development of some prototype and a commercial temporal database system. These researche, however, do not consider the schema modification problem in a temporal database as yet. Therefore, we propose a new methodology that supports the schema modification in a temporal object-oriented database. The class and instance structures of the existing object-oriented data model are extended to maintain historical information about the schema modification. Thus, a query requiring past information can be processed correctly after the previous schema is recovered by using this additional information. A desirable extension of the current work would be to formalize and analyze our extended object-oriented schema modification method.

References

[1] Banerjee, J., W. Kim, H.J.Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. in Proc. ACM SIGMOD Conference on Management of Data, San Francisco, CA., May 1987.

[2] S.K.Gadia and C.S.Yeung. A generalized model for a relational temporal database. In Proceedings of ACM SIGMOD International Conference on Management of Data, pp 251-259, Chicago, I1, June 1988.

[3] Wolfgang Kafer. and Harald Schoning. Realizing a Temporal Complex-Object Data Model, ACM SIGMOD Int. Conf. on Management of Data, 1992, pp.6-275.

[4] Hyoung-joo Kim. and Henry F. Korth. Schema Versions and Views in Object-Oriented Databases. Proc., of INFO JAPAN. 1990.

[5] Won Kim. Introduction to the Object-oriented database. MIT pres, 1990

[6] Navathe, S. N. and R. Ahmed. A Temporal Relational Model and a Query Language. Information Sciences, 49(1989), pp. 147-175.

[7] Sarda, N. Extensions to SQL for Historical Databases, IEEE Transactions on Knowledge and Data Engineering, 2, No. 2, June 1990, pp. 220-230.

[8] R. Snodgrass, The Temporal Query Language TQuel, ACM TODS, Vol. 12, No. 2, Jun. 1987.

[9] Su, S.Y.W., H.H.M Chen. A Temporal Knowledge Representation Model OSAM*/T and Its Query Language OQL/T. Proc., Seventeenth VLDB Conf., 1991. pp. 432-441.

[10] Tansel, A. U., Arkun, M. E., and Ozsoyoglu, G., Time-by-example query language for historical databases. IEEE Trans. Software Engineering 15, April, 1989, pp 464-478

[11] G. Wuu, U. Dayal. A Uniform Model for temporal Object-Oriented databases. Proc., Eighth IEEE Data Engineering Conf., February 1992. pp 584-593.