객체지향 데이터베이스의 스키마진화 환경에서 버전에 무관한 인스턴스 접근

진 민†

요 약

객체지향 데이터베이스 시스템의 응용에 있어 스키마진화는 필수적인 기능이다. 스키마진화 환경에서는 여러 스키마 버전과 이들에 의해 생성된 객체가 데이터베이스에 동시에 존재하게 된다. 본 논문은 객체지향 데이터베이스 시스템의 스키마진화 환경에서 각 버전에서 생성된 인스턴스를 버전에 무관하게 접근할 수 있는 한 가지 방법을 제안한다. 허용되는 스키마 변경 연산의 범위와 이를 지원하는데 필요한 비용은 반비례한다. 클래스의 각 버전에서 정의되지 않은 속성에 대해 접근처리루틴이 정의된다. 속성의 의미변화를 지원하기 위해 모든 버전의 쌍에 대해 갱신/역갱신 함수를 명세하는 대신 속성의 의미쌍에 대해 접근처리루틴을 명세함으로써 사용자 부담과 시스템의 복잡성을 줄이면서 속성의 의미변화를 지원한다.

Accessibility to Instances Regardless of Versions in Schema Evolution Environments in Object-Oriented Databases

Min Jin†

ABSTRACT

The facility of schema evolution is necessary in object-oriented database applications. There exist several versions of schema and instances created under different versions of the schema concurrently in the database. This paper proposes a mechanism for supporting accessing instances regardless of versions of the schema in schema evolution environments in object oriented databases. There is trade-off between the flexibility of schema modification operations and the system overhead for supporting such schema modifications. Access handling routines are provided to the undefined attributes in each version of the class. Access handling routines are also defined for each pair of semantics rather than defining backdate/update functions for each pair of versions of a class. Thus, this mechanism supports the change of semantics with reduced user and system overhead.

1. Introduction

There have been tremendous changes in computing technology including major changes in database systems over the past two decades. As database technology continues to develop, new database applications have appeared in various areas. Relational database systems have been developed extensively and widely distributed in organizations using computing [11,29]. However, conventional relational database systems have been demonstrated to be inadequate for emerging database application areas

† 종신회원 : 경남대학교 정보통신공학부 교수
논문접수 : 1998년 9월 28일, 심사완료 : 1999년 1월 29일

such as computer-aided design, manufacturing, software engineering, and office automation [8,12,13,17, 19,20,21,25]. Conventional relational data models have serious drawbacks in representing and manipulating complex and composite data structures since each attribute in relational data models should have a built-in atomic value.

Object-oriented data models appear to be attractive in design and manufacturing environments dealing with large and complex composite products. Core object-oriented technologies are characterized by four important concepts ; objects, encapsulation, inheritance, and polymorphism [5,6,9,28].

Objects are basic entities that have data structures and operations on these structures. Every object has a unique identifier. Classes and/or types describe the structure and behavior of a collection of similar objects. Class descriptions are also used to create individual instance objects. Instance objects within a collection have the same structure and operational behavior.

Objects encapsulate data and operations through classes. Users can not see the inside of objects. Data structures within the objects can be accessed only through public windows that consist of predefined operational functions called "methods". Objects communicate through messages. Methods are defined in a class and can be invoked by the corresponding messages.

Classes can be related through the inheritance feature in IS-A hierarchy within which all of the properties including data and operations of a class are inherited by its subclasses unless they are redefined in the subclasses. IS-A hierarchy supports generalization and specialization relationships.

Polymorphism is closely related to inheritance and encapsulation. The same message to instance objects of different classes may invoke specific operational behavior of the object. It depends on the method which is specially defined in the corresponding class.

Although there is no general agreement on the features of object-oriented database systems, many applications such as CAD, CASE, and office automation systems need schema evolution facility in which databases are required to store different alternatives of schema and multiple versions of the same objects. In single schema evolution, only one version of the schema exists, in which instances should be updated or filtered to the new schema whenever the schema is changed. There are a few drawbacks to the single schema modification [19]. First, it doesn't keep the history of modifications on the classes. Second, any other users are not allowed to access database while a user modifies the schema and all other users have to use the modified schema. Third, alternative schema can not be represented in the common database. Manufacturing environments need alternative schema and concurrent processing by several teams. The scheme of multiple versions of the schema can remove the drawbacks of single schema modification scheme. Just as the versions of the instance objects, the schema is versioned. There is more than one version of the schema active in the database. Histories, revisions, and alternatives of the schema can be represented with multiple versions of the schema. Previous schema versions should be kept with the corresponding instances that were created under the version of schema.

This paper summarizes the issues concerning schema evolution and presents an approach to solving accessing to instances regardless of versions in schema evolution in object-oriented databases. The remainder of this paper is organized as follows. In section 2, the issues concerning schema evolution are discussed. The methods for supporting schema evolution are also briefly summarized here. In section 3, related and previous work is summarized. In section 4, access handling routines are introduced to handle access to instances regardless of versions and the mechanism for supporting semantic changes is also discussed. In section 5, the conclusion is offered.

2. Issues in Schema Evolution

There have been three approaches to schema evolution ; versions of classes, versions of schema, and view of schema[1,2,4]. In versions of classes approach, any modification of a class creates a new version of the class and of its subclasses. This approach gives rise to a problem. Users have to choose a particular version for each class defined in the schema in order to represent the state of the schema at a given moment. In views of schema approach, any number of views can be derived on the schema. In versions of schema approach, any changes to the schema leads to derivation of a new version of the complete schema. Objects are associated with the corresponding schema version. A schema version corresponds to the complete state of the database at a given moment of time.

There are some issues to be addressed in the process of developing schema evolution scheme[30].

2.1 Schema Consistency

Schema consistency should be maintained in the process of schema modification. For example, when an attribute is dropped from a class definition, the methods that access the attribute should be changed or dropped to accommodate the change. A class inheritance hierarchy should not be allowed to be cyclic. There are two kinds of consistency ; structural consistency and behavioral consistency[1]. Structural consistency refers to the static part of the database. Informally, the schema is structurally consistent if the class structure is a directed acyclic graph(DAG) and if attribute name and method name definitions, attribute and method scope rules, and attribute domains and method signatures are compatible. Behavioral consistency refers to the dynamic part of the database. Informally, an object-oriented database is behaviorally consistent if each method respects its signature and if its code does not result in run-time error and unexpected results. Schema consistency can be checked by placing consistency checker between users and the system. It can work automatically or via user intervention. Invariants and rules have been suggested for the process of schema changes to keep the integrity of schema[3, 18,22].

2.2 Version Consistency

Each class object might have several versions of itself. In the scheme of schema evolution based on the versions of the whole schema itself, a version of the class that is consistent with each other is contained in a schema version, thereby solving the version consistency problem. In the scheme of schema evolution based on versions of classes, several versions of classes can be derived. In object-oriented database systems, some classes are related to others unlike relational database systems in which relations are independent of *each other*. The consistency among these versions should be controlled in order to keep the integrity of schema. It is necessary to determine which versions of different classes are consistent with each other. This task is called the configuration management. A configuration is a collection of versions of the classes in the database that are mutually consistent.

2.3 Object Consistency and Accessibility to Instances

An object is consistent if the values of the attributes are consistent with the class to which they belong. In addition to this, there is another problem to be solved that originates due to object inheritance among versions of schema. Objects that are created under a schema version are usually inherited by the descendent schema versions. The definitions of classes can be changed during the derivation of new schema versions. If the definition of a class is the same between a schema version and its descendent versions, the objects that were created under the ancestor schema version are obviously accessible to the descendent schema versions. They can be modified in the descendent schema versions, however, they can not be physi-

cally updated in the descendent versions because the ancestor schema version might lose the original values for the objects. Even if the definition of a class is changed during the derivation of a schema version, the objects that were created under the ancestor schema version need to be accessible to the descendent schema versions. This issue is explained with the notion of backward compatibility. A schema change is backward compatible if any query made on the changed schema can access data that were created under the schema before the change. The objects that are created under descendent schema versions also need to be accessible to the ancestor version. This can be explained with the notion of forward compatibility. A schema change is forward compatible if any query made on the schema before the change can access the data that are created under the schema after the change. Schema modification without versioning can achieve sort of backward compatibility by converting existing instance objects to new schema, however, it can not address the problem of forward compatibility. This issue is closely related to the issues of this paper and the details will be discussed in the later section.

2.4 Shareability

During the derivation of versions of schema, usually a small part of the schema is changed so that most part of the ancestor version of the schema remains unchanged in the new version. These unchanged parts of the schema can be shared among versions. The delta technique has been used in versioning files, in which the different parts between successive versions are maintained and the unchanged part is shared by the successive versions. The unchanged part that is common to successive versions can be selectively inherited by the descendent versions.

3. Related Work

There have been several approaches to solving the accessibility to instance objects among versions [7,10,18,22,23,24,26,27]. Skarra and Zdonik [26,27] proposed a type evolution scheme in a prototype object-oriented database system, ENCORE. In this scheme, types can be versioned. All versions of a type definition are called a version set of the type. The version set interface is defined for representing the most general interface to a type. This is constructed as a collection of all distinct properties, operations, and constraints of all versions of the type. When a new version of a type is created, new attributes that were not defined in the existing versions are added to the version set interface of the type. The version set interface is a single interface to all versions of the type. A handler is defined for every attribute in each version of the type in which the attribute is not defined, but it is defined in the version set interface. Handlers are used to help access instances that were created under different versions by a program expecting instances of its version. Since each instance is not allowed to have additional storage for undefined attributes, evolution operations are limited. Semantic change and name change can not be accommodated in this scheme. The mismatches between the instances and accessing programs are handled by the run-time system.

Monk and Sommerville [23,24] proposed a model for versioning class definitions. This model has been implemented in a prototype object environment called CLOSQL [23]. Forward and backward compatibility issues are addressed. Forward compatibility describes the ability of a program that was written for accessing instances of previous versions to access instances of new versions. Backward compatibility describes the ability of a program that is written using new versions of a class to access instances of previous versions of the class. This model for schema evolution is driven by the principle that it is insufficient to change the structure of the classes without considering the underlying semantics of the data. Semantic information is considered in the form

of special functions that convert instances of a version of a class to those of another version of the class. Update and backdate functions are provided on the attributes of the previous and current version respectively to help access instances that are created under other versions. When a new version of a class is derived by adding an attribute or changing semantics of an attribute, the backdate function is defined for that class version to access instances that were created under the previous versions. At the same time, the update function is defined for the current version to access instances that will be created under the new version. The query processor will convert all instances of the class to the format of the version of the class in the query. It is necessary to apply corresponding update or backdate methods consecutively along the path unless these methods are provided between every pair of versions. Thus, name change and semantic change are supported in this model. This model has some drawbacks. First, users are required to write their own update/backdate functions. Second, update/backdate methods should be provided between each pair of versions in order to avoid consecutive application of update/backdate methods between successive versions.

Kim and Chou [18] proposed a model for schema evolution based on versions of schema. This model is based on versions of objects that was implemented on ORION. They developed a model of versions of schema by extending the version capabilities and version derivation hierarchies of versions of objects to versions of schema, whereas the whole schema rather than each class is versioned in this model. The access scope of a schema version SV-i is defined as a set of objects that are accessible to SV-i. This set includes the objects that are created under SV-i and those that are inherited from the ancestor schema versions. The set of objects that are created under a schema version SV-i is called a direct access scope of the schema version. They introduced several rules dealing with the access scope issue such as whether the access scope should be inherited by the descendent schema versions and whether the access scope can be updatable in the descendent versions.

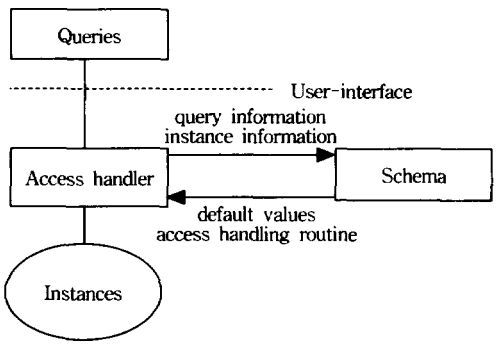
AVANCE [7] is an object management system that aims primarily at the applications in the field of office information systems. It uses version control on both the system and application levels. On the application level, there are object versions and object type versions. They use exception handling similar to the method used in [26,27] to cope with type version mismatches between the type version of the object expected by the query and the type version of the object accessed. Mismatch exception handlers are provided to service such queries. The handler may convert the instances to conform with the version expected by the invoker or may simply emulate old specifications, or both.

Clamen [10] proposed a schema evolution scheme that allows existing databases to be adjusted to different formats over time. It supports schema evolution and instance adaptation. It allows multiple representations of instances to persist simultaneously and provides programmers with the specification of how to adapt existing instances. When a version of a class is created, a new version of every instance of the class is created. These instance versions are called facets. The storage for the attributes that are used commonly in several facets can be shared among them. In deriving a new version, the attributes can be divided into several groups such as shared, independent, derived, and dependent attributes, depending on their relationships between versions. Users specify the adaptation strategy according to the classification.

OTGen [22] is a tool that is designed to support schema changes and database reorganization. It generates a transformer that applies the mappings between the old and new schema to update the data. The transformer consists of programs and tables that transform the existing data into the new schema.

4. Access Handling Routines

As discussed in the previous section, the accessibility to instances among versions has been one of the important issues in schema evolution. The notion of compatibility was introduced to explain accessibility [23,24]. There are two kinds of compatibility concerning the accessibility ; forward compatibility and backward compatibility.



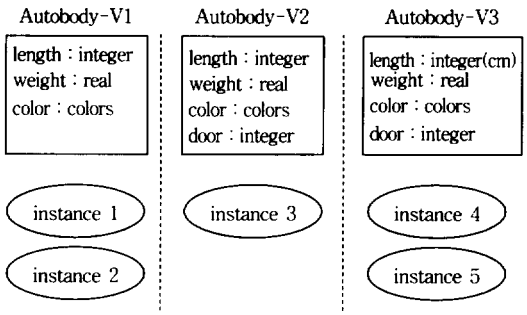
(Fig. 1) Access Handler

Forward compatibility implies that the programs that were written to access instances of old versions of a class could still access the instances that are created under new versions of the class. This leads to the program usability. Forward compatibility guarantees that the program that was written assuming the old version of the class can be used to access instances of new versions of the class. Backward compatibility implies that the programs that are written assuming the new version of a class could still access the instances that were created under the previous versions of the class. Several schemes have been proposed to cope with the problem of the accessibility to the instances among versions in schema evolution [10,18,23,27], which were discussed in the previous section.

The scheme, as seen in Figure 1, for coping with the accessibility problem is similar to that of exception handlers that was proposed in [26,27]. However, the scopes differ from each other. The objective of their work [26,27] was to make changes of a class transparent with respect to programs that use the class. Programs can access instances that are created under other versions of a class. This is achieved by using a version control mechanism and a set of exception handlers associated with the versions of a class. Exception handlers are added to versions of a class in order to do the behavior not defined in these versions but defined in other versions of the class. The handlers in these versions are carried out when programs expect them to do the undefined behavior. Thus, the handlers provide programs with class change transparency. Class change transparency leads to program usability. The scope of this work is different. It focuses on the accessibility to instances regardless of versions. The instances of a class might be accessed as a whole regardless of versions under which they are created at any moment of the process of schema evolution. The instances of a certain version of the class also might be accessed. It is necessary to access some or all of the product instances regardless of versions under which they were created. Instances can be accessed from any version without difficulty via the attributes that are commonly defined in all versions with the same semantics.

There are two cases in which accessibility problem arises. Name change is not considered since it is not allowed here. The first case is when the instances are accessed through the undefined attributes ; the invoking program assumes a version in which the attribute is defined, but the instances to be accessed don't have the attribute. The other case is when the instances are accessed through the attributes with different semantics ; the attribute is defined in the version that the invoking program assumes and it is also defined in the version under which the instances were created, however, the semantics of the attribute in both versions are different.

An access handling routine is provided for the version with an undefined attribute. In Figure 2,



(Fig. 2) Versions and Instances

Class	Attribute	Version	Access handling routine
Autobody	Door	1	door : = 4;

(Fig. 3) Access Handling Routine

when the attribute "door" is added to Autobody-V2, an access handling routine is provided for the undefined attribute "door" in Autobody-V1 as shown in Figure 3. This access handling routine usually gets predefined default-value. It is also necessary to allow semantic changes in the description of attributes. For example, "inch" in representing length can be changed to "cm". In this case, the domain of the attribute such as integer is not changed, but the semantics of the attribute is changed. In [27], where exception handlers are used for accessing instances across the versions, semantic changes are not allowed. In [23,24], backdate/update methods are used for accessing instances among versions and semantic changes are allowed and supported. However, backdate/update functions on each attribute should be provided for every pair of versions unless the backdate/update functions are consecutively applied along the version derivation hierarchy. Access handling routines are provided for every pair of semantics, not for every pair of versions. In Figure 2, the semantic of the attribute "length" in Autobody-V3 is changed to "cm" from "inch" that is valid in Autobody-V1 and Autobody-V2. Changes of the semantics of attributes are represented as shown in Figure 4.

Class	Attribute	Set of versions	Semantics
Autobody	length	{ 3 }	cm

(Fig. 4) Representation of Changes of Semantics

The semantics of the attribute "length" in V1 and V2 should be represented once changes to the semantics of the attribute occur as shown in Figure 5 although there is originally no explicit declaration on the semantics of the attribute. Thus, the semantics of "length" is represented in both V1 and V2 as "cm". An access handling routine is provided for each pair of semantics as shown in Figure 6. Hence, the number of functions to be provided can be reduced to a small number compared to that in the backdate/update function approach. A corresponding routine is provided for a tuple (*class, attribute, instance-semantics, accessing-semantics*).

Class	Attribute	Default semantics
Autobody	length	inch

(Fig. 5) Representation of Default Semantics

Class	Attribute	From semantics (instances)	To semantics (accessing)	Routine
Autobody	length	inch	cm	length := 2.54 * length
Autobody	length	cm	inch	length := length / 2.54

(Fig. 6) Access Handling Routines for Semantic Changes

The algorithm for accessing instances is described in Figure 7.

```
procedure AccessAllInstances(AQuery, Results) ;
/* This procudure receives a query and returns the result. */
begin
   if the query is against a set of versions,
      then  /* The query is against a group of versions. */
         begin
            for each version of the set do
               AccessAVersion(AQuery, Results);
         end;
      else  /* The query is against a specific version. */
            AccessAVersion(AQuery, Results);
   endif
end.
```

```
procedure AccessAVersion(AQuery, Results) ;
/* This procudure receives a query against a version and re-
   turns the result. */
begin
   if the attribute is defined in the target version
      then /* The attribute is defined in the target version. */
         begin
            if the semantic of the attribute differs
               then /* The semantic of the attribute is dif-*/
                    /* ferent from that of the attribute */
                    /* in the target version. */
                    invoke the corresponding access han-
                    dling routine for the semantic change;
            endif
         end;
      else /* The attribute is not defined in the target */
           /* version. */
           invoke the corresponding access handling routine for
           the undefined attribute;
   endif
end.
```

(Fig. 7) Algorithm for Accessing Instances of Versions

5. Conclusion

This paper proposes a mechanism for accessing instances regardless of versions of classes in schema evolution in object-oriented databases. It also summarizes the issues concerning schema evolution and approaches to accessing instances regardless of versions in schema evolution in object-oriented databases. There is trade-off between the flexibility of schema modification operations and the overhead needed for supporting such operations. Semantic change is not allowed in [26,27]. The scheme proposed in this paper supports semantic change as an operation of schema modifications. Access handling routines are provided to each pair of semantic changes. Thus, it could reduce the overhead drastically compared to the previous work in which backdate/update functions are provided to each pair of versions of the class unless the backdate/update functions are applied consecutively along the version derivation hierarchy.

There is further work. Implementation issues including user interfaces that support the need in the process of schema evolution are being investigated. This work can be exploited in the development of major applications which need schema evolution facility such as product data management system[13,15,16]. We are working on these issues.

References

[1] S. Abiteboul, P. C. Kanellakis, and E. Waller, "Method schemas," In Building an Object-Oriented Database System, The O2 story, Morgan Kaufmann, 1992.

[2] F. Bancilhon, C. Delobel, and P. Kanellakis, 'Building an Object-Oriented Database System, The Theory of O2,' Morgan Kaufmann, San Mateo, CA, 1992.

[3] J. Banerjee, W. Kim, and H. F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," In Proceedings of the ACM SIGMOD International Conference on Management of Data. pp.311-322, May 1987.

[4] D. Beech and B. Mahbod, "Generalized Version Control in an Object-Oriented Database," In Proceedings of IEEE 4th International Conference on Data Engineering, pp.14-22, Feb. 1988.

[5] E. Bertino and L. Martino, "Object-Oriented Database Management Systems : Concepts and Issues," IEEE Computer pp.33-47, April 1991.

[6] E. Bertino and L. Martino, 'Object-Oriented Database Systems, Concepts and Architectures,' Addison-Wesley, 1993.

[7] A. Bjrnerstedt and S. Britts, "AVANCE : An Object Management System," In Proceedings of OOPSLA '88, pp.206-221, 1988.

[8] R. G. G. Cattell, 'Object Data Management : Object-Oriented and Extended relational Database Systems,' Revised Edition, Addison-Wesley, Reading MA, 1994

[9] R. G. G. Cattell, 'The Object Database Standard : ODMG 2.0,' Morgan Kaufmann, San Francisco, CA, 1997.

[10] S. M. Clamen, "Schema Evolution and Integration," Distributed and Parallel Databases, Vol. 2, No.1, pp.101-126, 1994.

[11] R. Elmasri and S. B. Navathe, 'Fundamentals of Database Systems,' 2nd Ed., The Benjamin/Cummings, 1994.

[12] R. Gupta and E. Horowitz, 'Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD,' Prentice Hall, NJ, 1991.

[13] M. Jin, "An Object-Oriented Database Approach for Supporting Product Evolution in Agile Manufacturing," University of Connecticut, Ph.D., 1997.

[14] 진 민, 김봉진, "스키마 진화에서 버전에 무관한 개체 접근", 한국정보과학회 춘계학술논문발표집, pp.190-192, 1998.

[15] M. Jin & T. C. Ting, "An Object-Oriented Database Framework for Supporting Product Evolution," Proceedings of the ISCA 13th International Conference on Computers and Their Applications, pp.169-172, March 1998.

[16] R. H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," ACM Computing Surveys, Vol.22, No.4, Dec. pp.375-407, 1990.

[17] A. Kemper and G. Moerkott, 'Object-Oriented Database Management, Applications in Engineering and Computer Science,' Prentice Hall, NJ, 1994.

[18] W. Kim and H. T. Chou, "Versions of Schema for Object-Oriented Databases," In Proceedings of the 14th VLDB Conference, pp.148-159, 1988.

[19] W. Kim, 'Introduction to Object-Oriented Databases,' The MIT-Press, Cambridge MA, 1990.

[20] W. Kim, 'Object-Oriented Databases : Definition and Research Directions,' IEEE Transaction on Knowledge and Data Engineering, Vol.2, No.3, pp.327-341, September 1990.

[21] H. F. Korth, W. Kim and F, Bancilhon, "On Long-Duration CAD Transactions," In Readings in Object-Oriented Database Systems, Morgan Kaufmann, 1990.

[22] B. S. Lerner and A. N. Habermann, "Beyond Schema Evolution to Database Reorganization," In Proceedings of ECOOP/OOPSLA '90, pp.67-76, 1990.

[23] S. R. Monk and I. Sommerville, "A Model for Versioning of Classes in Object-Oriented Databases," Proceedings of the 10th British National Conference on Databases, pp.42-58, July 1992.

[24] S. Monk and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning," SIGMOD Record, Vol.22, No.3, September 1993.

[25] D. R. Rao, 'Object-Oriented Databases : Technology, Applications, and Products,' McGraw-Hill, 1994.

[26] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," In Proceedings of OOPSLA, pp.483-495, 1986.

[27] A. H. Skarra and S. B. Zdonik, "Type Evolution in an Object-Oriented Database," In Research Directions in Object-Oriented Systems, pp.393-413, 1987.

[28] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," In Proceedings of the ACM OOPSLA Conference, pp.38-45, Sept. 1986.

[29] M. Stonebraker, 'Readings in Database Systems,' 2nd Ed., Morgan Kaufmann, San Mateo, CA, 1994.

[30] E. Waller, "Schema Updates and Consistency," In Proceedings of DOOD, pp.167-188, 1991.

진 민

e-mail : mjin@zeus.kyungnam.ac.kr

1982년 서울대학교 계산통계학과 졸업(이학사)

1984년 한국과학기술원 전산학과 졸업(공학석사)

1997년 코네티컷 주립대학교 컴퓨터공학과 졸업(공학박사)

1985년~현재 경남대학교 정보통신공학부 부교수

관심분야 : 객체지향 데이터베이스, 멀티미디어 데이터베이스, 데이터 모델링, 분산처리