A Representation Method of Time-Varying Characteristics of

T. Ushiama and T. Watanabe

Reprint

from

Twentieth Annual International
Computer Software & Applications

Seoul, Korea
August 21-23, 1996

50 YEARS OF SERVICE
IEEE COMPUTER SOCIETY
1946-1996

Washington ◆ Los Alamitos ◆ Brussels ◆ Tokyo

PUBLICATIONS OFFICE, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1314 USA

A Representation Method of Time-Varying Characteristics of Entity on the Basis of Core-Surface Concept

Taketoshi USHIAMA Toyohide WATANABE

Department of Information Engineering,
Graduate School of Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-01, JAPAN
{ushiama, watanabe}@watanabe.nuie.nagoya-u.ac.jp

Abstract

Until today, techniques for modeling and managing the time-varying characteristics of entities in database have been proposed. These techniques are mainly classified into two categories: time-stamp method and version method. These methods limit the representation/manipulation capabilities because the change of class membership for an object is not well specified and the transitional states must be related mutually along the system-defined time axis. In this paper, we propose a method, which is basically derived from the object-oriented paradigm, for modeling time-varying characteristics of entities so as to be free from the above two limitations in the traditional methods. We introduce two representation media: core and surface. The core represents invariant features of entity, while the surface denotes variant features of entity. A snapshot of an entity is represented as an aspect, which is a database instance composed of the core and surfaces.

1 Introduction

The concept of object-orientation is powerful to model various phenomena and activities in the real world effectually and also is applicable to implement constructive functions of information systems successfully[1]. This is partly because the mechanisms such as the encapsulation, inheritance, polymorphism, message passing and so on can support stepwise refinement approaches in design and development processes, and partly because the concept of object makes it possible to represent active/passive entities, which are usually observed in the real world, compositively. The object-oriented database, based on such advanced features of object-orientation, was established clearly in application ranges such as CAD and CASE. This is because the relationships "Is-A" and "Part-Of" make it successful to organize complex entities analytically. Currently, object-oriented databases are expected to be adaptable to more complex applications such as multimedia databases, scientific databases and so on. In these applications, it is important to represent time-varying characteristics of entities with a view to dealing with the information activities of our human beings under the database management.

Time-varing characteristics of entities may be modeled in the following manners: (i) managing state transition of objects, and (ii) changing class memberships of objects. The former is realized by updating attribute values of objects, and the latter is done by object migrations[2–4], such as addition/deletion operations of attributes and methods of objects. Until today, the issues for managing the transitional states of objects in the object-oriented database have been researched[5–13]. The approaches adopted mainly in these researches are classified into two categories: *version method* of objects, and *time-stamp method* of objects.

The version method[6–10] introduces an ordered set of multi-occurrences of object, and these occurrences are called versions. A version is derived from an existing original object or another version of the object. We can look upon the version as the state of an object with respect to the time-dependent representation of object. In order to manage several versions systematically, a few relationships between an original object and a version, and among versions are introduced. For example, the object-oriented database system ORION[8,9] provides a version management facility. ORION supports two relationships with respect to the effective management among versions: *version-of* and *derived-from*. The former is adaptable between the original object and versions, while the latter is useful between a new version and the existing version from which the new version should be derived. Each version has its own identifier, and its original object can be uniquely indicated in terms of version-of. However, this method is too strongly limited in the application range: the concept of version is evaluated toward the engineering design domain such as CAD, but this method may be insufficient for common utilization. In other words, the state of an object (or version) is semantically fixed in the real world though the relationships among entities should be interpretatively specified so as to be able to represent various activities of entities. Additionally, every version of an original object must be statically typed as the class of the original object.

The time-stamp method[11–13] assigns the time

0730-3157/96 $5.00 © 1996 IEEE

data to different instances individually and then serializes the instances by the time-stamps attached to them. Thus, the concept of object identity is supported because the states of an entity correspond to these instances with different time-stamps. However, individual instances must be distinguished through time-stamp data. For example, Su and Chen[11] proposed the following tuple with respect to the representation of time-varying object:
(ex.)
⟨*Start-time, End-time, OID, Value*⟩.
Start-time is the time when the object, attended with the object identifier *OID*, becomes active in database. Similarly, *End-time* is the time when the object becomes inactive. This method is limited in two main respects. First, every state of an entity must be located over the system-defined time axis even if it is not always possible for users to do so. Generally, it is so necessary that users manipulate the situations among instances, which are controlled on the space axis, concerning the interrelations among other objects. Second, all instances having the same *OID* must belong to the same class, because the structure of instances is defined on the class and time-stamps cannot contain information about classes.

In summary, these two methods have the following problems for representing/managing transitional states of an object:

1. These methods assume that all instances (or versions) of an object have the same structure and behavior. Therefore, these methods do not support the change facility of class memberships for objects;
2. The representation methods for state transitions of objects are strongly dependent on the structure of time-stamps and the semantic of version mechanism. This means that these methods limit to represent time-varying characteristics of entities independently from a viewpoint of user.

This paper describes a data model for manipulating various transitional states of object and altering dynamic class memberships for object in database. Our main objective is to establish the alteration of class memberships for object as well as the manipulation of transitional states of object, and specify the transitional properties of objects organically along user-defined time/space axes. Concerning this objective, we regard snapshots of an entity as *aspects* which are derived constructively on the basis of different time axes and roles of entities. Thus, our framework for managing time-varying characteristics of entities is characterized by means of the specification of aspects. From this viewpoint, we introduce three media for the representation of entity: *core*, *surface* and *situation*. The core specifies invariant features of entity, and the surface denotes variant features of entity, the situation represents the circumstance in which a surface is active and individual surfaces can be identified through the corresponding situations. The concept of situation is more generalized than the time-stamp in the time-stamp method and version number/name in the version method. An aspect is represented by the combination of a core and a surface, and the situation is controlled as a mediate object in order to identify individual surfaces constructively.

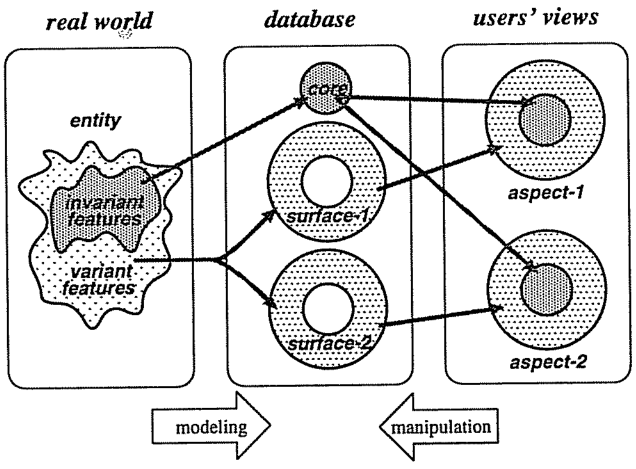

Figure 1: Modeling view for entity

2 Framework

Our objective is to develop a data model which supports the following features:

1. to manage both transitional states of object and class memberships for object;
2. to support arbitrary representation means for identifying individual transitional states of object in class.

2.1 State of Object

The version method and time-stamp method are adaptable for representing transitional states of an object. Their primary technique is only to distinguish instances/versions by time-stamps and identifiers, because this technique is based on the assumption that all instances/versions of an object are categorized into the same class. Therefore, these methods are not suitable to assign different class memberships to objects.

In order to overcome this problem, we propose an approach that can model an entity and its time-varying characteristics compositively in terms of different media. In our data model, time-varying characteristics of an entity are specified by two different objects in database, and are also composed as the combination of these two objects: core and surface. A core is derived from invariant features of an entity; and a surface corresponds to a snapshot of the variant features. An aspect corresponds to a state of an object to be understandable from users' views, and is derived from the combination of a core and a surface. The core and surface which construct an aspect have their own attributes and methods, and are categorized into distinct classes. Therefore, the change of attribute values and change of class memberships are realized by the replacement of surface in the uniform manner. Figure 1 shows such a modeling view.

Let us consider an example of the aspect construction. Suppose that a person is modeled as the core and that his/her name and birthday are defined as attributes of the core. When he/she is a student, his/her

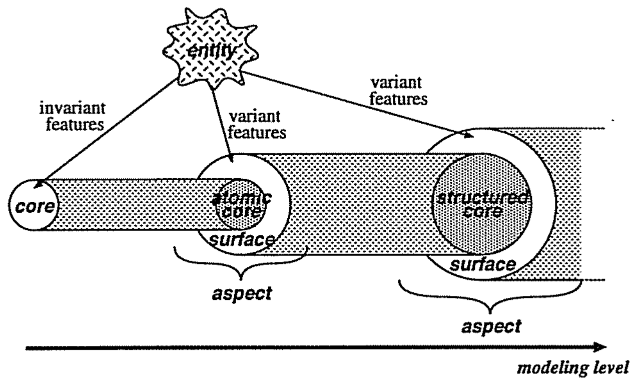

Figure 2: Aspect structure

course is defined as an attribute of a surface and the aspect has three attributes: name, birthday and course. If he/she becomes an employee, his/her salary is defined as an attribute of another surface. In this case, the aspect has three attributes: name, birthday and salary.

The notion about aspect construction can be expanded on the basis of the principle that the core object may be composed of core and surface objects recursively, if we can look upon the aspect as a *structured core*. Figure 2 shows this structure graphically. Namely, Figure 2 indicates that the aspect for an entity is a set of the atomic core and all surfaces, linked reversely from the most external surface.

2.2 Identity of Surface

Aspects represent time-varying characteristics of an entity in our data model. Each of aspects which share the same core is characterized by its surface. Thus, the representation and identification issues about surfaces are very important.

In the object-oriented paradigm, the concept of object identity is very important to make sure of the uniqueness of objects, modeled derivatively from the corresponding entity in the real world[14]. As for the management of transitional states, the object identity corresponds to the existence identification for the essential property of an entity. In order to model the transitional states effectively we consider the following three requirements:

1. the preservation of object identity: each surface must preserve the identity of object(core), because without this preservation each aspect loses the correspondence for the entity in the real world;
2. the state identification: each surface is distinguished and identified uniquely from others;
3. the possession of active circumstance: each surface must include the tag that represents the information about circumstance in which the surface is active.

These three requirements are partly satisfied with the instances in the time-stamp method and versions in the version method, in their own manners. For example, to make active circumstances clear, instances in the time-stamp method are labeled by the time-stamp tags; and versions in the version method are labeled by the version numbers and other versions which are linked by the relationship "derived-from". However, these labels must be predefined by system.

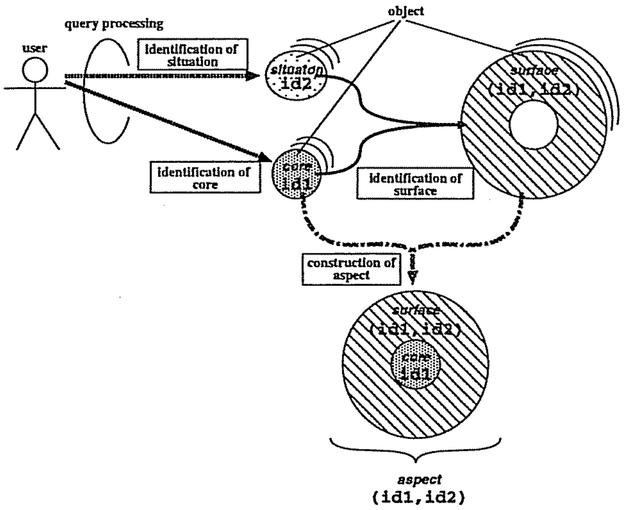

Figure 3: Construction of aspect

As mentioned before, it is one of our objective to support arbitrary representation means for identifying individual transitional states of object in class. As for this objective, we define the surface as a binary relationship between core and situation. The situation is an object that indicates the circumstance in which the surface is active. In our data model, the structure of situation can be defined by user arbitrarily.

Remarkably, to fulfill three requirements, the surface in our data model has the following two identities in addition to the object identity:

- **Core identity:** each surface attends with only one core and can indicate the core;
- **Situation identity:** each surface is assigned to only one situation and can indicate the situation.

For the implementation of surface with the above identities, we introduce the *structured identifier*, which is defined as an ordered pair of core identifier and situation identifier. These notations are illustrated in Figure 3. In Figure 3, there is a surface which has the structured identifier $(id1, id2)$. The first element $(id1)$ of this structured identifier denotes the core of this surface, and the last element $(id2)$ indicates the situation of this surface.

In our data model, users retrieve an aspect in the following manner. First, users identify core and situation for identification of a surface, and then construct the aspect by means of this surface and its core. This process is illustrated in Figure 3.

3 Data Model

Our data model is formulated fundamentally on the basis of the framework of a data model O_2[15], which defines explicitly the concepts of type and class. First of all, we define a simple object-oriented data model as the means for the constructive modeling.

Some primary notations are as follows:

- **integer**, **string** and **boolean** are names of *atomic value type*;
- A set of symbols $\boldsymbol{A}$ is called *attribute*;
- A set of symbols $\boldsymbol{ID_{atom}}$ is called *atomic identifier*. We use the number following the symbol "#" in order to represent a member of $\boldsymbol{ID_{atom}}$;
- **atomic** is a name of *atomic identifier type*;
- A set of symbols $\boldsymbol{CN}$ is called *class name.*

The atomic identifiers correspond to the object identifiers which are necessary for object identity in the traditional object-oriented data model. Using two kinds of identities which were already discussed in the section 2.2, we define *structured identifier*. Then, we define the object identifier as a disjoint union of atomic and structured identifiers.

Definition 1: *object identifier*
1. Each atomic identifier in $\boldsymbol{ID_{atom}}$ is an object identifier.
2. If id_c and id_s are distinct object identifiers, then the pair (id_c, id_s) is an object identifier, and is called *structured identifier.* Here, the object identifiers id_c and id_s are called *core identifier* and *situation identifier*, respectively. □

For example, #35, (#35, #55) and ((#35, #55), #122) are object identifiers. The first is an atomic identifier, and the rest is structured identifiers. For the structured identifier (#35, #55), the first argument #35 is the core identifier and the second argument #55 is a situation identifier. Similarly, for the structured identifier ((#35, #55), #122), (#35, #55) and #122 are the core identifier and situation identifier, respectively.

Definition 2: *value*
1. Every element in the domain is a value.
2. If $id_1, \ldots, id_n$ are object identifiers and $a_1, \ldots, a_n$ are distinct attribute names, then a n-tuple $[a_1 : id_1, \ldots, a_n : id_n]$ is a (tuple) value.
3. If $id_1, \ldots, id_n$ are distinct object identifiers, then a set $\{id_1, \ldots, id_n\}$ is a (set) value. □

Definition 3: *object*
An object is a pair (id, val), where id is an object identifier and val is a value. If the object has a structured identifier, it is called *surface.* □

The following descriptions are some examples of objects:
(ex.)
(#2,"John Smith"),
(#1, [name: #2,birthday: #3]),
((#1, #4), [weight: #6]).

The first object is a printable object which has a value "John Smith". The second object corresponds to a person whose name is "John Smith". The last object is a surface of the second object, and this surface has a snapshot of invariant feature of "John Smith". Note that it is not an aspect because the attributes and their values have not been yet inherited from its core.

Definition 4: *identifier type*
1. **atomic** is an identifier type.
2. If cn_c and cn_s are distinct class names, then (cn_c, cn_s) is an identifier type, and is called *structured identifier type.* Here, the class which has a class name cn_c is called *core class*, and similarly the class which has a class name cn_s is called *situation class.* □

For example, (SUBSTANCE,STATE) is an identifier type. The class SUBSTANCE is a core class, and STATE is a situation class. The identifier type (SUBSTANCE,STATE) represents the type of the surface which is composed of SUBSTANCE and STATE. The surfaces which have structured identifiers of this type denote the states of substances.

Definition 5: *value type*
1. Each atomic value type (e.g. **integer, string** and **boolean**) is a value type.
2. If $cn_1, \ldots, cn_n$ are class names and $a_1, \ldots, a_n$ are distinct attribute names, then a n-tuple $[a_1 : cn_1, \ldots, a_n : cn_n]$ is a value type.
3. If cn is a class name, then $\{cn\}$ is a value type.□

Definition 6: *class*
The class is a triple (cn, it, vt), where
1. cn is a class name in $\boldsymbol{CN}$.
2. it is an identifier type.
3. vt is a value type.

If the class has a structured identifier type, it is called *surface class.* □

The following descriptions are some examples of classes:
(ex.)
(PERSON,**atomic**, [name:PNAME,birthday:DATE]),
(PERSON-IN-AN-AGE, (PERSON,AGE), [weight:WEIGHT]).
The first class is a class which has class name PERSON, identifier type **atomic** and value type [name:PNAME,birthday:DATE]. The second class PERSON-IN-AN-AGE is the surface class whose core class is the class PERSON.

For the definition of schema, we use a notation:
- ref denotes by a function from $\boldsymbol{CN}$ in 2^{CN}, which associates to all the class names defined in its identifier type and its value type.

Definition 7: *schema*
A set of classes $\boldsymbol{S}$ is *schema* if it satisfies the following conditions:
1. $\boldsymbol{S}$ is a finite set.
2. The class name of each class in $\boldsymbol{S}$ is distinct.
3. For each class C in $\boldsymbol{S}$, $ref(C) \subseteq \boldsymbol{S}$.
4. For each surface class C_s in $\boldsymbol{S}$, both C_s and the core class of it must have tuple value types. □

The fourth condition in Definition 7 guarantees that each aspect can inherit the properties from its core.

Definition 8: *instance*
Let $\boldsymbol{O}$ be a set of objects and $\boldsymbol{S}$ be a schema. If a class C is (cn, it, vt) in $\boldsymbol{S}$, then a set of instances $I(cn)$ in the class C is $I_{id}(cn) \cap I_{val}(cn)$.

1. $I_{id}(cn)$, which is a set of instances in C based on its identifier type, is defined as follows:
   (a) If it is **atomic**, then
   $I_{id}(cn) = \{id \mid id \text{ is an atomic identifier }\}$.
   (b) If it is (cn_c, cn_s), then
   $I_{id}(cn) = \{(id_c, id_s) \mid (id_c, id_s) \in O, id_c \in I(cn_c), id_s \in I(cn_s)\}$.
2. $I_{val}(cn)$, which is a set of instances in C based on its value type, is defined as follows:
   (a) If vt is an atomic value type, then
   $I_{vt}(cn) = \{id \mid (id, val) \in O, val \in \text{corresponding natural domain }\}$.
   (b) If vt is $[a_1 : cn_1, \ldots, a_n : cn_n]$, then
   $I_{vt}(cn) = \{id \mid (id, [a_1 : id_1, \ldots, a_n : id_n]) \in O, id_i \in I(cn_i), i = 1, \ldots n\}$.
   (c) If vt is $\{cn'\}$, then
   $I_{vt}(cn) = \{id \mid (id, \{id_1, \ldots, id_n\}) \in O, \{id_1, \ldots, id_n\} \subseteq I(cn')\}$. □

Definition 9: *method*
We attach a set of *methods* to every class in $\boldsymbol{S}$. Each method has a *signature*:a mapping $m : c \times t_{v1} \times \ldots \times t_{vn} \rightarrow t$, where m is the name of method and $c, t_{v1}, \ldots, t_{vn}, t$ are class names or value types. The first type c of a method signature is a class which the method is attached to: it is called *receiver class* of the method. □

In order to manipulate the core, surface and situation, our model has some methods. Now, we suppose that the surface class whose name is *surface* has an identifier type $(core, situation)$.

1. **create_surface**: $core \times situation \rightarrow surface$
2. **get_aspect**: $core \times situation \rightarrow surface$
3. **get_core**: $surface \rightarrow core$
4. **get_situation**: $surface \rightarrow situation$

The method **create_surface** creates a new surface by means of given core and situation. The method **get_aspect** returns an aspect which is suitable to given core and situation. As mentioned in the section 3, each state of entities is organized by means of an aspect which is a combination of a core and a surface. Because surfaces contain only variant features of entities and cores contain only invariant features, the properties of a core and a surface must be merged in order to get the properties of a state. When users access a surface, its core is selected automatically and tuple values corresponding to the core and surface are joined. For example, if a core has a tuple value $v_1 = [\texttt{name}: id_1, \texttt{birthday} : id_2]$ and its surface has a tuple value $v_2 = [\texttt{weight}: id_3]$, then v_1 and v_2 are joined to $v_3 = [\texttt{name}: id_1, \texttt{birthday} : id_2, \texttt{weight}: id_3]$. When users access this surface, it does not have v_2 but v_3 as its value. A pair of a structured identifier of situation and its joined value is called *aspect*. Aspects do not exist in database but are created when surfaces are accessed, so users can always manipulate each surface as an aspect. The methods **get_core** and **get_situation** return the core and situation for given surface (aspect) respectively.

4 Example

This section shows an example of schema which is indicated using our model. Figure 4 is a graphically illustrated example of schema. In this illustration, the class is represented by a rectangle associated with its own class name. If the class takes a structured identifier type, then this rectangle is doubly enclosed by a large rectangle and is attended with a thick solid arrow for its core class or with a thick dashed arrow for its situation class. The atomic value type is pointed out through an oval labeled by its name. The value type for a class is pointed out by a thin solid arrow: if the class has a value type $[a_1 : cn_1, \ldots, a_n : cn_n]$, then there is a labeled arrow a_i which indicates cn_i; if the class has an atomic value type, then there is an unlabeled arrow which indicates its own atomic value type.

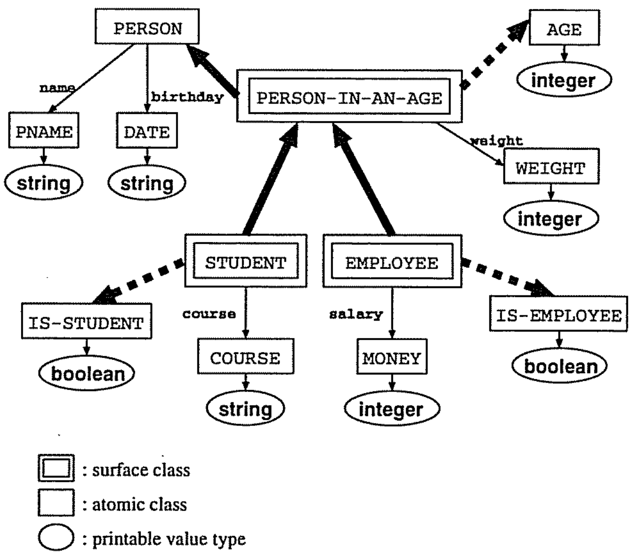

Figure 4: Example of schema

The schema illustrated in Figure 4 includes three surface classes: PERSON-IN-AN-AGE, STUDENT and EMPLOYEE. This means that each object in the class PERSON can have three kinds of surfaces: three different persons for an age, a student and an employee.

Figure 5 illustrates an instance of schema in Figure 4. In this representation, an object is indicated by its own object identifier. If the object takes a structured identifier, then the identifier has a thick solid arrow for its core and a thick dashed arrow for its situation. The atomic value is represented directly by itself. The value attended to an object is represented by labeled thin solid arrows: if the object has a tuple value $[a_1 : id_1, \ldots, a_n : id_n]$, then there is a labeled arrow a_i for id_i; if the object has an atomic value, then there is an unlabeled arrow which indicates its own atomic value.

In the instance shown in Figure 5, we can observe five surfaces about a person "John Smith" (which is represented by a node "#1"):

1. The node (#1, #4) is a surface for "John Smith of twenty-one years old",
2. The node (#1, #5) is a surface for "John Smith of twenty-four years old ",
3. The node ((#1, #4), #8) is a surface for "John Smith of twenty-one years old as a student",

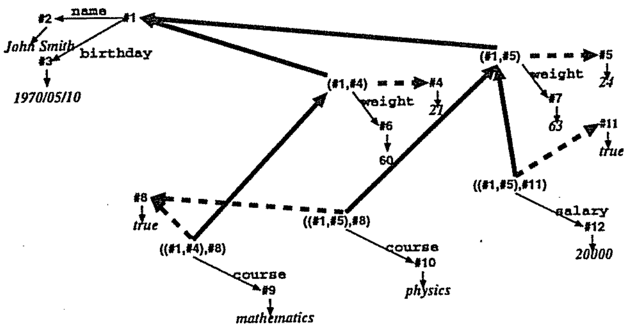

Figure 5: Example of instances

4. The node $((\#1, \#5), \#8)$ is a surface for "John Smith of twenty-four years old as a student",
5. The node $((\#1, \#5), \#11)$ is a surface for "John Smith of twenty-four years old as an employee".

Here, let us consider an example of query in this schema and its instances. For the query description, we use a SQL-like query language. In this language the "select" clause defines the result, the "from" clause specifies where to get the necessary information and the "where" clause gives the condition. For example, the query "*How much was John Smith's weight when he was twenty-four years old ?*" can be described as follows.

(ex.)

```
select   weight(get_aspect(person,age))
from     person in PERSON, age in AGE
where    name(person)="John Smith" and age=24
```

In this example, the methods "`weight`" and "`name`" return the values of attributes "`weight`" and "`name`" respectively. As mentioned in the section 4, the method "`get_aspect`" returns the aspect which is suitable to the given core and situation.

5 Conclusion

In this paper, we presented an object-oriented data model, which has expressive facilities to model various time-varying characteristics of entities. Our data model can manage the change of both attribute values and class memberships for objects, and also supports arbitrary representing means for identifying individual snapshots of entities.

For representation time-varying characteristics of entities, our data model provides three media: *core*, *surface* and *situation*. The core represents invariant features of an entity, the surface represents variant features, and the situation represents the circumstance where a surface is active. A snapshot of entity is represented by the *aspect*, which is constructed compositively by means of the core and surfaces. The distinct surfaces may have different attribute values and class memberships, so the change of attribute values and class membership of aspects can be managed uniformly by altering surfaces.

Acknowledgments

We are very grateful to Prof. N. SUGIE of Meijyo University, Prof. T. FUKUMURA of Chukyo University, and Prof. Y. INAGAKI and Prof. J. TORIWAKI of Nagoya University for their perspective remarks, and also wish to thank Prof. Y. SAGAWA, Prof. K. ASAKURA and our research members for their many discussions and cooperations.

References

[1] M. Atkinson, et al.: "The Object-Oriented Database System Manifesto", *Proc. of DOOD'89*, pp. 40–57 (1989).

[2] Q. Li and G. Dong: "A Framework for Object Migration in Object-Oriented Databases", *Data & Knowledge Engineering*, Vol. 13, No. 3, pp. 221–242 (1994).

[3] J. Su: "Dynamic Constraints and Object Migration", *Proc. of VLDB'91*, pp. 233–242 (1991).

[4] A. Mendelzon, et al.: "Object Migration", *Proc. of ACM SIGMOD/PODS'94*, pp. 232–242 (1994).

[5] G. Özsiyoğlu and R. Snograss: "Temporal and Real-Time Databases: A Servey", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, pp. 513–532 (1995).

[6] R.H. Katz, E. Chang and R. Bhateja: "Version Modeling Concepts for Computer-Aided Design Databases", *Proc. of ACM SIGMOD'86*, pp. 379–386 (1986).

[7] R.H. Katz and E. Chang: "Managing Change in Computer-Aided Design Database", *Proc. of VLDB'87*, pp. 455–462 (1987).

[8] W. Kim: "*Introduction to Object-Oriented Databases*", chapter 12, pp. 145–171, MIT Press (1990).

[9] W. Kim, et al.: "Features of the ORION Object-Oriented Database System", *in Object-Oriented Concepts, Database, and Applications*, pp. 251–282, Addison-Wesley (1989).

[10] D. H. Fishman, et al.: "Iris: An Object-Oriented Database Management System", *ACM TOIS*, Vol. 5, No. 4, pp. 48–69 (1987).

[11] A. Y. W. Su and H. M. Chen: "A Temporal Knowledge Representation Model OSAM*/T and Its Query Language OQL/T", *Proc. of VLDB'91*, pp. 431–441 (1991).

[12] W. Käfer and H. Scöning: "Realizing a Temporal Complex-Object Data Model", *Proc. of ACM SIGMOD'92*, pp. 266–275 (1992).

[13] W. Chu, et al.: "A Temporal Evolutionary Object-Oriented Data Model and Its Query Language for Medical Image Management", *Proc. of VLDB'92*, pp. 53–64 (1991).

[14] S.N. Khoshafian and G.P. Copeland: "Object Identity", *Proc. of OOPSLA'86*, pp. 406–416 (1986).

[15] C. Lecluse, P. Richard and F. Velez: "O_2: An Object-Oriented Data Model", *Proc. of ACM SIGMOD'88*, pp. 424–433 (1988).