データベース・システム 80−12
人 工 知 能 73−12
（1990. 11. 9）

Quixote のオブジェクト識別性

森田 幸伯，羽生田 博美
沖電気工業 (株) 総合システム研究所
横田 一正
(財) 新世代コンピュータ技術開発機構

オブジェクト識別性 (オブジェクト・アイデンティティ) は演繹・オブジェクト指向データベース (DOOD) を含むオブジェクト指向システムで重要な役割を果たしている。本稿では、従来のオブジェクト識別性に関する議論を再検討し、それを表現するための基準を提示し、現在 ICOT を中心に設計している知識表現言語 (かつ DOOD 言語) *Quixote* でのオブジェクト識別性に焦点を当てる。*Quixote* では、オブジェクト識別性は拡張項 (部分項) として表現され、包摂関係に基づいた束を構成する。また、*Quixote* におけるオブジェクト識別性に関連した諸性質についても議論する。

Object Identity in *Quixote*

Yukihiro MORITA, Hiromi HANIUDA
Systems Laboratory, Oki Electric Industry Co., Ltd
4-14-12, Shibaura, Minato-ku, Tokyo 108, JAPAN
e-mail: morita@okilab.oki.co.jp, haniuda@okilab.oki.co.jp
Kazumasa Yokota
Institute for New Generation Computer Technology (ICOT)
21F., Mita-Kokusai Bldg., 1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN
e-mail: kyokota@icot.or.jp

Object identity plays a key role in object-oriented systems, including deductive and object-oriented databases (DOODs). We revisit various discussions about it, propose some criteria to represent object identity, and focus mainly on the characteristics in *Quixote*, which is a knowledge representation language developed by ICOT and can be considered as one of DOOD languages. In *Quixote*, object identifiers are written in the form of extended term and constitute a lattice based on the subsumption relation. We also discuss some features related to object identity of *Quixote*.

1 Introduction

Why do we need new generation databases? Why should we extend to conventional databases such as relational, hierarchical, and network databases? The reason is that conventional ones have not only difficulties to cope with various applications such as engineering databases and knowledge bases but also make users take a burden such as impedance mismatch, and computational and semantic responsibility. In order to overcome such difficulties, many approaches have been proposed: deductive databases, object-oriented databases, and semantic data models. Common features or main trends among them are towards integration of database, programming, and knowledge representation languages.

Among the approaches, we take an approach for deductive and object-oriented databases (DOODs), which seems to be more flexible and comprehensive than others. That is not a fixed concept of databases or a data model but a framework for extensions to conventional ones. [11] classifies extensions to deductive databases as follows:

1) Logical Extensions

2) Data Modeling (Encapsulary) Extensions

   (a) Introduction of Object Identity

   (b) Introduction of Complex Data Structure

   (c) Encapsulation of Data and Procedures

3) Computational (Paradigmatic) Extensions

   (a) Object-Orientation Paradigm

   (b) Constraint Logic Programming Paradigm

We can extend par each item and combine them for a new database. In our sense, DOODs are databases along the above framework.

Object-orientation concepts have been ambiguously used in various contexts, although they are considered to be useful in many applications of databases. In object-oriented programming languages, the active aspects are emphasized from a computational point of view, while in object-oriented databases, the passive aspects are emphasized from a data modeling point of view. Furthermore, other concepts such as object identity, encapsulation, and inheritance are not also used uniformly even in database society. Under the present situation, many efforts are devoted to the formalization, while most of object-oriented systems are developed very practically.

In the above framework, we persist in their formalization for both active and passive aspects of objects as extensions to deductive databases, that is, in logic programming paradigm. In the sense, object-orientation concepts should be made clearer. Among various features of the concepts in DOODs, the concept of object identity is one of the most important over passive and active objects. Besides object identification, from a data modeling point of view, object identity works for construction of complex data structure and property inheritance, while, in a computational point of view, an object works autonomously and communicates by message passing through object identity with other objects. In this paper, we mainly focus on the concept and features of object identity in a knowledge representation language *QUIXOTE*.

We proposed a DOOD language *Juan* in [12], which is integrated with a knowledge representation language *QUINT* and now called *QUIXOTE*. In Section 2, we revisit various discussions on object identity and show several criteria of object identity for DOODs. In Section 3, we represent an object identifier (oid) formally in the form of extended terms, and compared it with other works. In Section 4, we describe various features in the context of *QUIXOTE*.

2 Discussion about Object Identity

Object identity is one of the most important issues of object-oriented programming languages and also object-oriented database languages. For an object itself, identity is the property of the object that distinguishes it from all others in a system. On the other hand, from a user point of view, it is the property with which users can find a specific object from a pool of objects. There are several papers [7, 8, 3, 12] that discuss object identity in object-oriented systems. However some of the papers miss the second point of view for object identity. In this section, we revisit them and discuss how to represent object identity.

2.1 For Whom Is Object Identity

Khoshafian and Copeland [7] discussed object identity in general purpose programming languages and database languages. They classify degrees of support of object identity of languages in a two-dimensional space: the *representation* dimension and the *temporal* dimension. In the temporal dimension, object identity is classified as follows:

1) temporal data
   (a) within a program or transaction (e.g., Smalltalk-80, Pascal, Prolog)
2) persistent data
   (a) between transactions (e.g., RM/T, UNIX shell)
   (b) between structural reorganizations (e.g., OPAL)

In the representation dimension, there are three classes;

1) data value (e.g., identifying employees by social security number)
2) user-supplied name (e.g., Pascal, Prolog, UNIX shell)
3) system built-in (e.g., Smalltalk-80, RM/T, OPAL)

According to [7], an succeeding item in each dimension supports more strongly the notion of object identity than the preceding ones.

As for the implementation of object identity, [7] discusses two concepts of *data independence* and *location independence*, and concludes that the most powerful technique is through *surrogates* as identifier.

However, in [7], object identity is discussed from a viewpoint of behaviors of objects in a system but not from a viewpoint of user's manipulation of objects. Considering how to support both of these properties with an identifier, we classify attributes of an object into two categories, that is, attributes which partially take a role of identity and ones which are independent of identity.

We explain this classification with an example. Suppose there is an object corresponding to a person who is teen-age, or more exactly a type of objects each of which is a teen-aged individual. What kind of an oid should we give to such an object? Suppose that an attribute of the object indicating "teen-age" is $age \rightarrow teens$. Since the attribute plays an essential role for identification, we construct the oid as

$$person[age \rightarrow teens].$$

In many papers, object identity is discussed to be independent of the attributes (or the state) of the object. [7] points out four problems for identifier keys which are attributes for identification. However, these discussions does not take both properties of identity into consideration. Surrogates in [7] could not be used to find a specific object with some attributes of the object. Several papers[2, 3, 6] pointed out that 'pure' object-based languages are inconvenient, because there is a case that an *oid* is not so important for some object. For example consider the following[6]:

eiffel_tower
tuple(name = "Eiffel Tower",
admission_fee = 25 FF,
address = *eiffel_address*)
eiffel_address
tuple(city = *paris*,
street = "champ de mars")
paris
tuple(name = "Paris",
country = "France"
population = 2.6)

where, *eiffel_tower*, *eiffel_address*, and *paris* are oids. However it is convenient to treat *eiffel_address* not as an object but as a pure value, since it is immutable and is not shared by other objects.

2.2 Beyond Conventional Object Identity

Ullman [8] also discussed value-oriented systems and object-oriented systems. He listed advantages and disadvantages of each system and concluded: "Value-oriented systems will win simply because they offer the user arbitrary access to data, with access expressed declaratively". However, he assumed that "declarative programming is hard to integrate with object-oriented systems". He misses requirements of new applications for object-oriented systems and many efforts for formalization of object-oriented systems, although most of the efforts have been done after [8]. In *QUIXOTE*, object-oriented and value-oriented features are integrated into an object term as an identifier.

Beeri [3] took another standpoint: "O-ids are supposed to implement the ideas that each object has an identity, different from that of any other object, that does not change throughout its lifetime. We claim this is unnecessary. O-ids are implementation concepts". [3] denies to deify object identity which has been discussed abstractively, and consider it as an implementation element. Alternatively, he selects names and references in order to refer to objects. That means a schema including such a name space that contains the names of the relations and the attributes in the database world. Although "How do we refer to objects?" was considered in [3], it is still insufficient. In *QUIXOTE* an identifier is regarded as a "name" in the same sense of [3], but not as an implementation element.

Kifer and Lausen [4] proposed F-logic, in which object-orientation concepts are embedded in logic programming. They define *id-terms* for oids and labels, which are based on a first order predicate notation. Kifer et al [5] revised a syntax of labels to take any number of arguments as overloading of the predicate, but not one of an oid. Oids become to be defined explicitly, but there remain some disadvantages of a first order predicate notation: fixed number of arguments, fixed location of arguments, et al. *QUIXOTE* adopts object terms in the form of extended term representation to represent oids and labels uniformly.

Pure object-based languages have another inconvenient from a database point of view. Most of them employ system built-in oids, where the only way to get an oid is to create an object or to get from creator of the object. Consider a query for retrieval of monuments in Paris which is represented as an object in a system, when we have some information about Paris. How can we get the oid of Paris for the query? Some mechanism is needed to get an oid easily. As mentioned above, in *QUIXOTE* we can construct an oid with a name in the sense of [3] with attributes, which play an essential role in the identification.

2.3 Criteria

Yokota [12] discussed the following criteria for oids from a DOOD point of view:

1) Rules should support a mechanism for dynamically generating the oid of intentionally defined object.

2) Object sharing needs a 'global' oid referred from the related objects, especially in a dis-

tributed environment.

3) A persistent object also needs an oid, which should be possible to be recalled when the object is activated in memory again.

4) An oid should be given even when we have only partial information about some object, because we cannot expect an object has a fixed number of attributes and fixed structure as the identification information.

Now we consider object structure which is essential to identify the object. For example, Figure 1 shows a graph with a cycle as a structural object. As we mentioned above that an oid could include some attributes of the object to identify the object, we could construct the oid for the object, which reflects the structure.

So, we inherit above four criteria and add another one:

5) We should prepare rich constructors for representing oids even with recursive structure

3 Representation of Object Identifiers

Mentioned in the previous section, we construct an oid freely, as the representation of object identity.

3.1 Individual Object Term

First assume a set $\mathcal{O}$ of basic objects, which has partial ordering $\preceq$ and constitutes a lattice ($\sqcup$ for a join operation and $\sqcap$ for a meet operation), and a set $\mathcal{V}$ of variables. An *individual object term* is recursively defined as follows:

1) A basic object o ($\in \mathcal{O}$) is an individual object term.

2) A variable X is an individual object term.

3) If o is a basic object, $o_1, \cdots, o_n$ are individual object terms, and $l_1, \cdots, l_n$ are labels, then $o[l_1\ \theta_1\ o_1, \cdots, l_n\ \theta_n\ o_n]$ is an individual object term, where θ_i for $1 \leq i \leq n$ is $\rightarrow$, $\leftarrow$, or $=$.

4) If o is an individual object term but not a variable, and X is a variable, then $X@o$ is an individual object term.

An individual object term in the form of $X@o$ is called an *annotated variable*, which is used for construction of cyclic structure. For example, consider $X@o[l = X]$. If we unfold it step by step, we get $o[l = o[l = o[l = o[\cdots]]]]$ with infinite structure. Such structure frequently appears in many applications: a *person*'s *parent* is a *person*, whose *parent* is a *person*, whose *parent* is a *person*, whose $\cdots$.

The operators $\rightarrow$, $\leftarrow$, and $=$ correspond to ordering among object terms (see the details in Section 4). That is, we can translate an object term into the constraint form:

$o[l \rightarrow o'] \Leftrightarrow o[l=X]|\{X \sqsubseteq Y, Y \cong o'\} \Leftrightarrow o[l=X]|\{X \sqsubseteq o'\}$
$o[l \leftarrow o'] \Leftrightarrow o[l=X]|\{X \sqsupseteq Y, Y \cong o'\} \Leftrightarrow o[l=X]|\{X \sqsupseteq o'\}$
$o[l = o'] \Leftrightarrow o[l=X]|\{X \cong o'\}$

Now we allow the constraint form in the definition of object terms and extend the representation:

3') If o is an individual object term, $l_1, \cdots, l_n$ are labels, and $X_1, \cdots, X_n$ are variables, then $o[l_1 = X_1, \cdots, l_n = X_n]|\{c_1, \cdots, c_m\}$ is an individual object term, where c_i for $1 \leq i \leq m$ is $X_i \sqsubseteq o_j$, $X_i \sqsupseteq o_j$, or $X_i = o_j$ for any individual object term o_j.

Note that an annotated variable also can be written in the form of the constraint form. For example, $X@o[l \rightarrow X]$ can be written as $X|\{X \cong o[l = Y], Y \sqsubseteq X\}$.

3.2 Object Term in the Context of Object-Orientation

Oids are represented uniformly in such a way and play such a key role in the context of object-orientation.

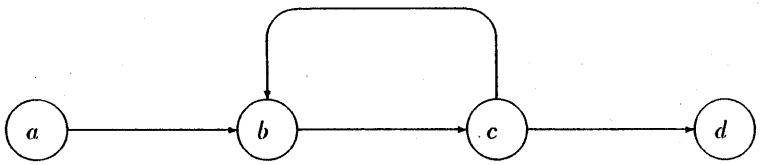

Figure 1: A Graph with a Circle

As shown in the construction of object terms, each object term is treated not as object-based, but as value-based. For example,

$$john[have = pen[maker = a], like = pen]$$

is a different oid from

$$john[have = pen[maker = a], like = pen[maker = a]],$$

although there is a subsumption ordering among them (see Section 4).

We attach properties and methods to such an oid. Each property is represented in the triple of a label, an operator, and an object, and can be considered as a kind of methods, because a label and an object term in an attribute can be considered to correspond to a message identifier with messages and the return value, respectively. An oid and methods are written in a head part of a rule and the corresponding implementations of methods can be written in the body of the same rule:

$$oid/[method_1, \cdots, method_n] \Leftarrow implementation_1, \cdots, implementation_m$$

A label represented by extended term can be regarded as a kind of method. For example, we can represent a method for a class *human* that return set of common friend with other person as following:

$$X/[common\_friend[with = Y] \rightarrow \{Z\}] \Leftarrow X \sqsubseteq person/[friend \rightarrow \{Z\}], Y \sqsubseteq person/[friend \rightarrow \{Z\}].$$

A set of rules with the same oid corresponds to an object, and a method could have multiple implementations in rules of an object.

Furthermore, the oid also corresponds to a type, a class, or an instance. In an object-oriented system, a *type* summarizes the common features of a set of types [1]. By introducing some ordering between oids, such common features can be attached to the upper oid. The notion of a *class* contains twoand an object warehouse [1]. As for the former, our oid can create instances by binding variables in the oid, which corresponds to the operation *new*. Instantiated objects are located under the original object by natural subsumption ordering among oids. In the sense, oids also contains the concepts of an object warehouse.

3.3 Semantics and the Extensions

An oid takes a set of labeled graphs as the semantics and the correspondence is guaranteed by Solution Lemma of ZFC$^-$/AFA set theory [9]. Inversely, the Solution Lemma allows more extensions of oids. Here we list some of them under consideration:

1) Introduction of Set Constructor:
   if $o_1, \cdots, o_n$ are individual objects, then $\{o_1, \cdots, o_n\}$ is a set object.

2) Combination of Set and Tuple Constructors:
   $o[l \rightarrow \{o_1, \cdots, \leq_n\}, l' \rightarrow o']$

3) Introduction of Element-of Relation:
   $o[l \in \{o_1, \cdots, o_n\}]$.

Thus, an *object term* is represented in various ways. For example, a graph in Figure 1 is written by using a set constructor in two ways as follows:

$$p[fr = a, to = X@p[fr = b, to = p[fr = c, to = \{X, p[fr = d, to = nil]\}]]].$$

$$p[fr=a,\ to=p[fr=b,\ to=Y@p[fr=c,\ to=p[fr=b,\ to=\{Y,p[fr=d,\ to=nil]\}]]]].$$

The expressions have the same semantics as shown in [10].

4 Extended Term in $\mathcal{QUIXOTE}$

In this section, we discuss several characteristics of extended terms as an oid in a DOOD language $\mathcal{QUIXOTE}$.

4.1 Ordering among Object Terms

These is a partial ordering among basic objects ($\mathcal{O}$). For example, $human \preceq animal$, which means that *human* is an *animal*.

We define a partial ordering $\sqsubseteq$ extended terms. In this paper, we give an informal definition by example. See [9] about the formal definition.

- $a \sqsubseteq b$ if a and b are basic object such that $a \preceq b$
- $o[l = a] \sqsubseteq o[l = b]$ if $a \sqsubseteq b$
- $o[l_1 = a, l_2 = b] \sqsubseteq o[l_1 = a]$
- $o[l_1 = X, l_1 = X] \sqsubseteq o[l_1 = X, l_2 = Y]$

where, X, Y are variables.

Then we can define an equivalence relation $\cong$ between extended terms. [1]

$$o_1 \cong o_2 \ \equiv\ o_1 \sqsubseteq o_2 \wedge o_1 \sqsupseteq o_2$$

An equivalent extended term represents same object. That is, object term o_1, o_2 describe the same object if $o_1 \cong o_2$. For example, $eating\_event[agt = john, obj = apple]$ means same object as $eating\_event[obj = apple, agt = john]$.

[1] The user-defined extended term ordering is not considered here.

4.2 Inheritance and Exception

As mentioned in the previous section, an object term (oid) can have properties, an object term with which is called an attribute term. For example, if a human named "john" is 24 age, we can represent it as:

$$human[name = \text{“}John\text{”}]/[age = 24]$$

Here, a term on the righthand of '/' represents properties of the oid that represented by a term on the lefthand of '/'. In $\mathcal{QUIXOTE}$, label and label value are also represented in the form of extended term.

$\mathcal{QUIXOTE}$ inherits a property inheritance mechanism from *Juan*. Properties are inherited upward and downward along the ordering of object terms. If there are several properties under a same label, these values are joined or merged according to the operator. That is, if $obj \sqsubseteq class$:

$$class/[l \rightarrow p] \ \Rightarrow\ obj/[l \rightarrow p]$$
$$obj/[l \leftarrow p] \ \Rightarrow\ class/[l \leftarrow p]$$

By such a mechanism, multiple inheritance is also possible. For example,

if $john \sqsubseteq human$ and $human/[name = string]$
then $john/[name \rightarrow name]$.

That is, if *john* is *human* and *human*'s *name* is *string* then *john*'s *name* is (an instance of) *string*. Attributes specified in an object term are also considered as properties of the object, and inherited. However, such attributes cannot be changed because this is a essential for the object. In this way, we can represent exception. Consider the following example:

$bird/[flying \rightarrow yes]$
$penguin \sqsubseteq bird[flying \rightarrow no]$
$super\_penguin \sqsubseteq penguin[flying \rightarrow yes]$

Note that

$bird/[flying \rightarrow no] \sqsubseteq bird$, and
$penguin[flying \rightarrow yes] \sqsubseteq penguin$,

according to natural ordering among extended terms.

According to the inheritance mechanism, $bird[flying \rightarrow no]$ inherits $flying \rightarrow yes$ from *bird*. However, it cannot change the property $flying \rightarrow no$, because the property is essential for the object. Then *penguin* inherits only the property $flying \rightarrow no$ and *super_penguin* inherits $flying \rightarrow yes$.

4.3 Module

QUIXOTE has a concept of *modules*, which makes rule inheritance possible. The module can be regarded as a world in *Juan*, a situation in *QUINT*, or a time. Same object term over different modules can be a same object. But these objects can have different states or properties. For example, it is possible that an object 'john' has a value 24 for his age in a module and has 25 in other module.

Each module has a module identifier, the syntax of which is same as an object identifier. The partial ordering among module identifiers is also defined as following two ways:

1) Natural ordering among module identities

2) User-defined ordering

Let $m :: r$ mean a module m has a rule r. Under the ordering, rules are inherited as follows:

$$\text{if } m_1 \sqsubseteq m_2 \text{ then}$$
$$\forall r\, \exists r'(m_1 :: r \Rightarrow r \sqsubseteq r' \wedge m_2 :: r')$$

where m_1, m_2 are module identifiers and r_1, r_2 are rules. In this case, the existence itself of a object, all properties of o in m_2, and rules about o in m_2 are merged with ones in model m_1.

For example, consider following:

$$\text{if } m_1 :: john/[have = pen],$$
$$m_2 :: john/[want = pen], \text{ and}$$
$$m_1 \sqsubseteq m_2,$$

then *john* has a *pen* in module m_2, too.

Figure 2 shows an example of module that describe an 'update' scheme [2]. $u\_mod[target = M]$ defines the parametric module. That is, the rule in this module defined a update rule for a module M. In other words, the target for the rule in the module depends on the parameter M. Consider an update of a module $mod[id = t_1]$ (say m_1), by update the rule in a module $u\_mod[id = add\_age[t = taro], target = m_1]$ (say u_1). Then the result of the update is a module that is represented by $mod[id = update[target = m_1, up\_mode = u_1]]$ (say m_2). Since the first rule of the example represent a ordering of module identifiers, module m_2 contains rules and facts that is in module m_1 but not in $del[target = m_1, up\_mod = u_1]$. In u_1, there appear other module *math* that has (perhaps) rules about arithmetic operation on integer. We can use rules of this module in this way, or using partial ordering among modules. We can modularize knowledge in this way. Furthermore, *QUIXOTE* has also an exception mechanism in rule inheritance among modules (see [10]).

5 Concluding Remarks

We discussed a notion of object identity and oids in *QUIXOTE*. Object identity is one of the most important issues of object-oriented programming languages and also object-oriented database languages. There are two viewpoints of the object identity: viewpoint of an object itself and viewpoint of a user. On the first point of view, identity is the property of the object and distinguishes it from all others in a system. On the second one, identity is the property, with which users can find a specific object from a pool of objects. However, many works miss the second point of view of the object identity, and we believe that the point is important for databasse. *QUIXOTE* uses extended term representation (an object term) as oids, each of which can include some attributes of the object. So user can obtain a necessary oid easily.

An object term itself has a characteristic of value-oriented representation, while an attribute

[2] This example does not necessarily mean semantics of update in *QUIXOTE*.

$$mod[id = update[target = Time, up\_mod = U]] \sqsupseteq$$
$$T@mod[id = Time]/del[target = T, up\_mod = U@up\_mod[id = X, target = T]].$$
$$del[target = M, up\_mod = U\_mod] ::$$
$$Fact \Leftarrow u\_mod[target = M] : changes/[should\_retract \leftarrow Fact].$$
$$mod[id = update[target = Time, up\_mod = u\_mod]] ::$$
$$Fact \Leftarrow u\_mod : changes/[should\_assert \leftarrow Fact].$$
$$u\_mod[id = add\_age[t = taro], target = M] ::$$
$$changes/[should\_assert \leftarrow taro[age = X]] \Leftarrow M : taro/[age = Y],$$
$$math : Y \sqsubseteq integer/[add[to = 1] = X].$$
$$u\_mod[id = add\_age[t = taro], target = M] ::$$
$$changes/[should\_retract \leftarrow taro[age = Y]] \Leftarrow M : taro/[age = Y].$$

Figure 2: Example of Module Rules

term has a characteristic of object-oriented one based on the oid. That is, in the sense, the oid in *QUIXOTE* integrates value-oriented and object-oriented concepts.

We also describe how extended term representation as as oid plays an important role in several features: representation of partial information, inheritance with exception, and massage passing. *QUIXOTE* uses uniformly extended term representation as an identifier not only for an oid but also for a label, a value, and a module identifier. Extended term representation makes it possible to represent parametric ones, that is, a kind of abstract data type.

In [7], update is one of the most important notions to discuss object identity. However, this paper does not discuss update, since update semantics of *QUIXOTE* is one of the future works.

Acknowledgments

The authors would like to thank members of *QUIXOTE* meeting and members of ETR-SWG for valuable comments and suggestions.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik: The Object-Oriented Database System Manifesto, *Deductive and Object-Oriented Databases*, Kyoto, 1990.

[2] S. Abiteboul and P.C. Kanellakis: Object Identity as a Query Language Primitive, *SIGMOD*, 1989.

[3] C. Beeri: Formal Models for Object Oriented Databases, *Deductive and Object-Oriented Databases*, Kyoto, 1989.

[4] M. Kifer and G. Lausen: F-Logic: A Higher-Order Languages for Reasoning about Objects, Inheritance, and Scheme, *SIGMOD*, 1989.

[5] M. Kifer, G. Lausen, and J. Wu: Logical Foundations for Object-Oriented and Frame-Based Languages", *Technical Report 90/14 (revised)*, June, 1990.

[6] C. Lecluse and P. Richard, "The O_2 database programming Language", In *VLDB'89*, August, 1989.

[7] S. N. Khoshafian and G. P. Copeland: Object Identity, *OOPSLA'86*, September, 1986.

[8] J.D. Ullman: Database Theory: Past and Future, *PODS*, 1987.

[9] H. Yasukawa and K. Yokota: Labeled Graphs as a Semantics of Objects, *SIGDBS & SIGAI of IPSJ*, Nov., 1990.

[10] H. Yasukawa and K. Yokota: An Overview of of a Knowledge Representation Language *QUIXOTE*draft, 1990.

[11] K. Yokota and S. Nishio: Towards Integration of Deductive Databases and Object-Oriented Databases: A Limited Survey, *Advanced Database System Symposium*, Kyoto, 1990.

[12] K. Yokota: The Outline of a Deductive and Object Oriented Database Language *Juan*, *SIGDBS of IPSJ*, July, 1990.