データベースシステム 104−15
（1995. 7. 18）

オブジェクトデータベースの経路存在制約とその応用

大本 英徹
工学部情報通信工学科
京都産業大学

高松 利行
情報システム部
大和ハウス工業株式会社

田中 克己
工学部情報知能工学科
神戸大学

実世界データの入れ子構造を直接的に表現する複合オブジェクトの概念は，オブジェクト指向データベース (OODBMS) における最も重要な特徴の一つである．この概念を用いてデータベースのオブジェクトの間を「巡航」し，関連する情報を辿ることが出来る．この巡航の経路の概念はOODBの最も重要なものの一つであると言える．他方，一貫性制約はデータベースの重要な基本的概念の一つであるが，オブジェクトの巡航経路に関する一貫性制約の問題は，十分に検討されてきているとは言えない．W. Kimらは OODB における一貫性制約として複合オブジェクトにおける存在従属性制約の概念を提案しているが，それは直観的な提案に留まっており十分な形式化はなされていない．本論文では，複合オブジェクトにおける存在従属性を形式的に議論する枠組みとして，経路存在従属性 (Path Existence Dependency, PED) の概念を導入し，その基本的定義を行う．また与えられたPEDから新たなPEDを導出するための幾つかの推論則を挙げている．また，PEDの具体的応用の可能性を示すため，建築CADにおける幾つかの例を示す．

Path Existence Constraints in Object Databases and its Applications

Eitetsu Oomoto
Dept. of Comp. & Comm. Sci.
Faculty of Engineering
Kyoto Sangyo University

Toshiyuki Takamatsu
Information Systems Office
Daiwa House Industry Co., Ltd.

Katsumi Tanaka
Dept. of Comp. & Syst. Eng.
Faculty of Engineering
Kobe University

The notion of *complex objects* by using object identifiers as attribute values is one of the typical features in object-oriented database management systems (OODBMSs), which handles the complicated nested structure of objects in the real world. Using this notion, we can *navigate* in the database among the related objects one after another. Therefore, the notion of reference path of in complex objects is very essential concept in OODBs. However, the integrity constraints, which is one of the most basic concepts in database systems, for the reference path in complex objects are fully discussed in formal. As one of the integrity constraints in OODB, the notion of existence dependency is one of integrity constraints in object-oriented databases presented by W. Kim et al. However, their notion of existence dependency is only conceptual and they doesn't have made a discussion formally in detail.

In this paper, we propose a notion of *path existence dependencies* (PED) as a method to treat formally the existence dependencies, especially, dependencies between object reference paths. Introducing a notion of selector variables, a suitable object can be specified as a unit of the path existence dependency. Several inference rules for PEDs are shown, and practical application examples are described in order to demonstrate the efficiency of the notion of PEDs.

1 Introduction

In object-oriented database management systems (OODBMSs) [1, 10, 11], the object identifiers (oids)[4] are used to identify each object uniquely without regard to its attribute values. Generally, since oids are used to attribute values of the object, the nested structured objects, *complex objects*, can be represented directly. With this notion, the object sharing, which one object is *shared* by several objects, is accomplished with reference values in attributes. Using the notion of complex objects, we can *navigate* among the related objects one after another with oids, which are used as attribute values. For example, the notation, `person1.father.father`, is generally called a *dot expression* [6] or *path expression* [5], and is denote the "grand father" of `person1`[1].

The notion of reference path of in complex objects is very essential concept in OODBs, consequently, which is very important from both theoretical and practical aspects. However, the integrity constraints, which is one of the most basic concepts in database systems, for the reference path in complex objects are fully discussed in formal.

As one of the integrity constraints in OODB, the notion of *composite objects* [7] by W. Kim et al. is well known. This is consist of the two orthogonal concepts: The *exclusive reference constraint* is that some object is referred by only one other object, and the *existence dependency constraint* is that the existence of some object is dependent to another object. But, both concepts are only intuitive proposal, therefore, formalization in order to discuss characteristics of them is not sufficient.

In this paper, we introduce the notion of *Path Existence Dependencies*[15], (PED), which is an extension of the existence dependency constraint by introducing the notion of *path expressions* and *selector variables* [5]. With this notion of PEDs, the formal treatment for the integrity constraints in terms of the existence dependency between reference paths of objects in OODBs. Additionally, several inference rules for derivation of PEDs are also treated. By those rules, it is enable to remove the redundant PED or to detect a loop of reference path.

In section 2, several motivations are discussed with intuitive examples. The basic definitions needed for our notion of PEDs are described in Section 3. The formal definitions for PEDs are given in Section 4. Several examples to explain the notion of PEDs and inference rules for PEDs are also described. The discussions for the relationships between the PED and other semantic constraint in OODBs are described in Section 5, additionally, a few examples of practical applications for PED are discussed. Section 6 is a concluding remarks.

2 Motivations

In some cases, the existence of some object depends on that of another object in order to satisfy the semantic integrity constraints in databases. This notion is called the *existence dependency* of objects, which is introduced by W. Kim[7]. Intuitively, the existence dependency is that, when an object o exists, some another object o' also exists.

For example, in the restaurant, the dish should be created *after* the corresponding order have been accomplished. This is a kind of existence dependencies (Figure 1). That is, "When a dish (object) exists, an order (object) corresponding to it should exist. " On the other hand, this constraint is also interpreted as the following: If the order (object) is deleted, then the corresponding dish should be deleted. With this notion of existence dependencies, we can represent some kind of constraints about the existence of objects. In other words, this also can be considered as constraint of the deletion of objects in the database.

In several notable facilities of OODB, the notion of complex objects is very useful one. That is, the object has either a simple atomic value or an object identifier (oid). And, using oids as attribute values, the nested structure of data can be represented directly. Therefore, on the discussion of existence dependencies in OODBs, we consider, the notion of complex object should be taken into account. Consequently, as the unit of existence dependencies, the individual objects do not only considered but also the group of objects such that they have references of attributes each other. In our path existence dependency, the dependency between the reference structures of objects can be treated with the *path expression*.

We consider the problem of spouse under monogamy as another example. Suppose that some person (object) has his spouse (object). Then, his spouse should also have her spouse. In this case, the existence dependency, "If a person has a spouse, then his spouse also should have a spouse," is not sufficient to semantically maintain the integrity constraint. The reason is that, as shown in Figure 3, the undesirable situation such that three persons are mutually spouse of each other can arise.

In order to solve this situation, the constraint should be modified such as "A person X has a spouse Y, then X is also the spouse of Y" using some way (in this case, variable X and Y) to represent explicitly the relationship between objects. But, in the notion of the existence dependency by W. Kim, lack of a constructor to represent the reference among objects and to specify objects on the reference path, the above example can not be represent accurately. On the other hand, in order to represent such situation, we introduce the notion of *path expression* such as `person.spouse`, and *selector variables* to denote the specified object in the references.

3 Basic Definitions

In this section, several terms are defined, which are needed for the following formal discussion. As the basic construct of our data model, O_2 data model [2], which is given some restrictions, is assumed.

Intuitively, an *object* is the tuple type, that is, it is a pair of an tuple value and an object identifier (oid), and only atomic values or oid's are allowed as attribute values.

3.1 Basic Definitions

First, the following sets, mutually disjoint, are assumed.

- The set of all strings, $\mathcal{S}$.
- The set of all numerics, $\mathcal{N}$.
- The set of all boolean, $\mathcal{B}$.
- The set of all object identifiers, $\mathcal{I}$.
- The set of all attributes, $\mathcal{A}$
- The set of all classes, $\mathcal{C}$, where $String, Number$ and $Boolean$ are classes, and are the elements of $\mathcal{C}$.

[1] In this case, `person1` denotes an oid, `father` is the attribute, whose value is an oid of the *Person* type object.

Also, two predefined functions are assumed. *class* is a many-to-one mapping from oids to a class, and *oids* is a mapping from a class to the powerset of oids. These functions are denoted formally as follows[2]:

$$class : \mathcal{I} \rightarrow C$$

$$oids : C \rightarrow 2^{\mathcal{I}}$$

Definition 1 Value

Values are defined as follows:

1. Special symbol *nil* is a value. It corresponds to *null value*.
2. All strings, numerics, and booleans are values. They are called atomic values.
3. An arbitrary object identifier $i \in \mathcal{I}$ is value.
4. n-tuple $[A_1 : v_1, \ldots, A_n : v_n](A_i \in \mathcal{A}, 1 \leq i \leq n)$ is a value, where $v_1, \ldots, v_n$ are *nil*, an atomic value or oid, respectively[3].

Definition 2 Type

Type are defined as follows:

1. a class C is a type.
2. For arbitrary classes, $C_1, \ldots, C_n$ and attributes, $A_1, \ldots, A_n$, n-tuple, $[A_1 : C_1, \ldots, A_n : C_n]$, is a type[4].

Here, it is assumed that a many-to-one function *type* is given. *type* is a mapping from the set of class $\mathcal{C}$ to $\mathcal{T}(\mathcal{C})$, which is the set of all types created with any element of $\mathcal{C}$ [5]:

$$type : \mathcal{C} \rightarrow \mathcal{T}(\mathcal{C})$$

Furthermore, the mapping, *Dom*, from a type to a set of values is defined as follows: Assume that C and $C_1, \ldots, C_n$ are classes respectively,

1. $Dom(String) = \{nil\} \cup \mathcal{S}$
2. $Dom(Number) = \{nil\} \cup \mathcal{N}$
3. $Dom(Boolean) = \{nil\} \cup \mathcal{B}$
4. $Dom(C) = \{nil\} \cup oids(C)$
5. For a type $t = [A_1 : C_1, \ldots, A_n : C_n]$where $A_i \in \mathcal{A}$and $C_i \in \mathcal{C}(1 \leq i \leq n)$,

$$\begin{aligned} Dom(t) &= Dom([A_1 : C_1, \ldots, A_n : C_n]) \\ &= \{[A_1 : v_1, \ldots, A_n : v_n] \mid v_i \in Dom(C_i), \\ &\qquad 1 \leq i \leq n\} \end{aligned}$$

Definition 3 Object

Object is a pair, $o = (i, v)$, defined as:

- i is an oid. That is, $i \in \mathcal{I}$.
- v is a n-tuple value, $[A_1 : v_1, \ldots, A_n : v_n]$[6].

Here, for any object, $o = (i, [A_1 : v_1, \ldots, A_n : v_n])$, it is assumed that the relationships, $type(class(i)) = [A_1 : C_1, \ldots, A_n : C_n]$ and $v_i \in Dom(C_j)(1 \leq j \leq n)$, are satisfied. □

[2] Intuitively, *class* returns the class for a given oid. *oids* returns the set of oids of possible objects which are belong to a given class.

[3] Unlike the original O_2 data model, we do not treat the *nested* value for simplicity.

[4] For simplicity, the *nested type* is not concerned.

[5] Intuitively, for a class, *type* returns the attribute structure of objects which belong to it.

[6] For simplicity, the tuple type object is only considered.

Then, the attribute value of the object, $o = (i, [A_1 : v_1, \ldots, A_n : v_n])$, is denoted by $v_j = i(A_j)$ $(1 \leq j \leq n)$. Furthermore, the set of all objects in the database are called the database instance, DB. And, two arbitrary different objects in DB, $(i_1, v_1), (i_2, v_2)$, the relationship, $i_1 \neq i_2$, is satisfied.

Definition 4 Extension

Extension is the set which is returned for a class, C, by the function, $extent(C)$. It is defined as follows: $extent(C)$ is a function from the set of classes onto the subset of DB which are the set of all the existing objects in the database. The following characteristic should be also satisfied:

$$extent(C) = \{(i, v) \mid class(i) = C \text{and } (i, v) \in DB\}$$

However, for each different classes, C_1, C_2, $extent(C_1) \cap extent(C_2) = \emptyset$ is satisfied.

When the relationship $o \in extent(C)$ is satisfied for an object o, o is called the *instance* of the class C.

3.2 Path and Path Expression

Definition 5 Attribute Sequence

For more than zero attributes in $\mathcal{A}$, A_i $(1 \leq i \leq n)$, the dot notation, $A_1.A_2.\ldots.A_n$, is called the *attribute sequence*. Especially, the attribute sequence of zero length is denoted by Id. □

Intuitively, an attribute sequence corresponds to a navigable route from an object to another object in the database using dot notation of attributes. For example, for the following attributes,

$$\mathcal{A} = \{name, address, body, engine, door, color\},$$

the notation, *body.door.color*, is an attribute sequence.

Definition 6 path

For an object, $(i, v) = o \in DB$, and attributes, $A_i \in \mathcal{A}$ $(1 \leq i \leq n)$, if there exist the following objects in DB,

$$\begin{aligned} o_1 &= (i_1, v_1) \\ o_2 &= (i_2, v_2) \\ &\vdots \\ o_{n-1} &= (i_{n-1}, v_{n-1}) \end{aligned}$$

and the following relationships are satisfied among these objects,

$$\begin{aligned} i(A_1) &= i_1 \\ i_1(A_2) &= i_2 \\ &\vdots \\ i_{n-2}(A_{n-1}) &= i_{n-1} \\ i_{n-1}(A_n) &= v_n \quad \text{where } v_n \text{is not } nil. \end{aligned}$$

then the list of oids, $< i, i_1, \ldots, i_{n-1}, v_n >$, is called *path*, and is denoted by $i.A_1.\ldots.A_n$[7].

However, it is defined that $i(Id) = i$ for an arbitrary oid i. □

Here, for any path, $i.A_1.\ldots.A_n (0 \leq n)$, the function, *tail*(), is defined to return the last value of the given path. That is, when $i.A_1.\ldots.A_n =< i, i_1, \ldots, i_{n-1}, v >$ holds, $tail(i.A_1.\ldots.A_n) = v$.

Furthermore, we extend the function, *Dom* as follows: For an arbitrary class, C, and an attribute sequence, $A_1.\ldots.A_n$ $(n \geq 1)$,

[7] In usual, such dot notations denotes the *tail* value or object of the attribute navigation. But, in this paper, these notations denote the navigable path itself in the database.

- Case: $n = 1$

$$Dom(C.A_1) = \{v \mid v = i(A_1), i \in Dom(C)\}$$

- Case: $n \geq 2$

$$Dom(C.A_1.....A_n) = \{v \mid v = i(A_n), i \in Dom(C.A_1.....A_{n-1})\}$$

Intuitively, $Dom(C.A_1.....A_n)$ denotes the set of all the possible end values such that they are reachable along with the attribute sequence, $A_1.....A_n$, from each instance of the class C.

Definition 7 Path Expression

For an arbitrary class, C, and an attribute sequence, $A_1.....A_n$,

$$C.A_1.....A_n$$

is called *path expression.* For the path expression, $C.A_1.....A_n$, the function, $path(C.A_1.....A_n)$, denotes the following set of paths in the database DB.

$$path(C.A_1.....A_n) = \{i.A_1.....A_n \mid o = (i, v) \in DB, class(i) = C\}$$

□

Definition 8 Selector Variable

Assume the path expression, $C.A_1.....A_j.....A_n$. The notation which is added a variable, X, for the class C or any attribute $A_j (1 \leq j \leq n)$ in this expression is also a kind of path expressions. This expression type is called, especially, *path expression with selector variable.* Furthermore, the notation $A_j[X]$ is called *attribute with selector variable.* However, it is defined that $C[X].A_1.... = C.Id[X].A_1....$ holds.

Here, the same selector variable must be bound to the same value. For example, assume that the path expression with selector variable, pe, have two same variable in it, that is,

$$pe = C.A_1.....A_j[X].....A_k[X].....A_n$$

The function, $path(pe)$, denotes the following set of paths such that the attribute values of the attributes A_j and A_k are the same value:

$$path(pe) = \{i.A_1.....A_j.....A_k.....A_n \mid (i, v) \in DB,\ class(i) = C,\ tail(i.A_1.....A_j) = tail(i.A_1.....A_j.....A_k)\}$$

When the path, $i.A_1.....A_n$, is an element of the set of paths, $path(C.A_1.....A_n)$, we call that the path, $i.A_1.....A_n$, *conform* to the path expression, $C.A_1.....A_n$.

□

4 Path Existence Dependency

In this section, we introduce the notion of *path existence dependencies.* and describe the formal definition of it, and several intuitive examples.

4.1 Definition of Path Existence Dependency

Definition 9 Path Existence Dependency

For arbitrary two path expressions, pe_1 and pe_2,

$$pe_1 \Rightarrow pe_2$$

is called *Path Existence Dependency,* in short, PED. When this path existence dependency holds to the database instance, DB, the following conditions should be satisfied:

1. Case 1: pe_1 and pe_2 do not have the common selector variable.

$$\forall p\ (p \in path(pe_1)) \supset \exists q\ (q \in path(pe_2))$$

2. Case 2: For each selector variable, $X_h (1 \leq h \leq n)$, such that they are common to the both path expressions pe_1 and pe_2, the condition $(*)$ holds: For the two path expressions,

$$pe_1 = C_1.A_1.....A_j[X_h].....$$
$$pe_2 = C_2.B_1.....B_k[X_h].....$$
where $A_l\ (1 \leq l \leq j-1)$ and $B_m\ (1 \leq m \leq k-1)$do not have the common selector variable.

the following condition holds:

$$(*)\ \forall p\ \exists q\ (p \in path(pe_1) \supset q \in path(pe_2) \wedge tail(i_1.A_1.....A_j) = tail(i_2.B_1.....B_k))$$
where $p = i_1.A_1.....A_j.....$ and $q = i_2.B_1.....B_k.....$

□

Intuitively, when some PED holds to the database,DB, for each path, which conforms to the left side of the given PED, there exist, at least, one such path in DB that conforms to the right side.

4.2 Examples of Path Existence Dependencies

In this subsection, we explain the above notion of path existence dependencies with several intuitive examples.

Let us suppose a database such that it manages the tuple type objects. In this database, the tuple type object is that it has finite attribute and has, at most, one value for each attribute. This is similar to the tuple in the relational databases. For instance, we assume that we have the family registration management database, and that an object (Person) has two attributes, *spouse* and *child.* As shown in figure 3, the class in the database in only *person,* the object o_1 has his spouse, o_w, and his child, o_3. And the another person o_4 has her spouse, o_5, and her child, o_6.

In such family database, a variety of integrity constraints may be considered for each object, or among objects.

1. When a person have his (her) child, someone must have his (her) spouse.

   Although this is not a practical example, this constraint can be represented by the following PED:

$$Person.child \Rightarrow Person.spouse$$

   Intuitively, whenever any instance, $o = (i, v)$, of $Person$ class have the path $i.child$, in other word, o has a value in *child* attribute, an object, $o' = (i', v')$, as the instance of the class $Person$ must exist such that the path, $i'.spouse$, exists in the database (Figure 4).

2. When a person has his child, he must have his spouse.

   This example is concerning with the integrity constraint among specified objects. In order to represent the relationships among specified objects in terms of the given constraint, the *selector variables* are introduced in path existence dependencies. In the following example, the selector variable, X, appeared in the both side, denotes the same object.

$$Person[X].child \Rightarrow Person[X].spouse$$

Intuitively, whenever any object, $o = (i, v)$, in the instances of the class $Person$ have the path $i.child$ (o has the value in the *child* attribute), o itself must have the value in the *spouse* attribute (This is shown in Figure 5).

3. When a person has a child, his (her) spouse must have the same child.

   This constraint is represented by the following PED:

   $$Person[Y].child[X] \Rightarrow Person[Y].spouse.child[X]$$

   This PED means the followings: If the object, $o = (i, v)$, in the instances of the class $Person$ has the path $i.child$ o must have another path, $i.spouse.child$, and the tail objects of $i.child$ and $i.spouse.child$ must be the same, that is,

   $$tail(i.spouse.child) = tail(i.child)$$

   (See Figure 6).

   From another viewpoint, this PED restricts the combination of the values of *spouse* and *child* attributes, that is, in terms of the pair of the attribute value, (*spouse*, *child*), of a $Person$ object, the combinations ($value1, value2$) or ($value, nil$) are allowed (Figure 7), but ($nil, value$) is not allowed (Figure 8). In other words, the admissible null pattern of the attribute value in an object is also represented.

4. When "some person" has a spouse, the spouse of spouse of "some person" must be "somebody" himself.

   This is represented by

   $$Person[X].spouse[Y] \Rightarrow Person[Y].spouse[X]$$

   By this PED, the following situation is represented: Whenever the instance, $o = (i, v)$, of the class $Person$ has a path, $i.spouse$, at least, one person object, $o' = (i', v')$, must exist such that the path, $i'.spouse$, exists and $tail(i'.spouse) = i$ holds (Figure 9).

4.3 Inference Rules for Path Existence Dependencies

In this subsection, we show the inference rules for path existence dependencies.

Rule 1 For an arbitrary class, C, and an attribute sequence, p,

$$C.p \Rightarrow C.p$$

always holds. □

Rule 2 For an arbitrary class, C, and attribute sequences, p_1 and p_2, an attribute, A, and a selector variable, X,

$$C.p_1.A[X].p_2 \Rightarrow C.p_1.A.p_2$$

always holds. □

Rule 3 For an arbitrary class, C, and attribute sequences, p_1 and p_2, an attribute, A, and a selector variable, X, such that it appear in neither p_1 nor p_2,

$$C.p_1.A.p_2 \Rightarrow C.p_1.A[X].p_2$$

always holds. □

Rule 4 Suppose that C is an arbitrary class, p is an attribute path which has more than zero length, and A is an attribute or attribute with selector variable. Then,

$$C.p.A \Rightarrow C.p$$

always holds. □

Rule 5 For the two PEDs, $C_1.p_1 \Rightarrow C_2.p_2$ and $C_2.p_2 \Rightarrow C_3.p_3$, the transitive PED, $C_1.p_1 \Rightarrow C_3.p_3$, holds. Here, C_1, C_2 and C_3 are arbitrary classes, and p_1, p_2 and p_3 are arbitrary attribute sequences with more than zero length. □

Intuitively, when the transitive relationships between several PEDs, a new PED such that it satisfies these relationships is also derived.

Rule 6 If the PED,

$$C_1.p_1.A[X] \Rightarrow C_2.p_2.B[X]$$

holds, then

$$C_1.p_1.A[X].A_1.....A_n \Rightarrow C_2.p_2.B[X].A_1.....A_n \ (1 \leq n)$$

also holds. Here, C_1 and C_2 are arbitrary classes, p_1 and p_2 are attribute sequences with more than zero length, and A, B and $A_i (1 \leq i \leq n)$ are arbitrary attributes respectively. □

Intuitively, if the selector variable appeared in the left side of the given PED denotes the object such that it is the last element of the path to conform to the right side, then the attribute sequence of any length following the attribute with its selector variable can be appended to the right side.

Rule 7 For a path expression, $C_1.p_1.p_2$, if the relationship, $Dom(C_1.p_1) = Dom(C_2)$, holds, then the following PED is also holds:

$$C_1.p_1.p_2 \Rightarrow C_2.p_2 \quad \square$$

The formal discussion for these rules in detail such that proof, characteristics, completeness, etc. is beyond the scope of this paper because of lack of space.

5 Discussion about Path Existence Dependencies

In this section, the relationships between PED and other integrity constraints are described. And, the usefulness of practical applications for the notion of PEDs are shown.

5.1 Relationship between PED and Other Constraints

5.1.1 Exclusive Reference Constraints

Here, we consider the relationship between the exclusive reference constraint[7] and our path existence dependency. For example, the following constraint is assumed: A *Car* object has its engine in *engine* attribute. And, an *Engine* object must be exclusively held by only one car (Figure 10).

With the current definition of the PED, this exclusive reference constraint can not be represented. However, several extension for the definition of PED may enable the representation of the exclusive reference constraint. That is, the admission of the combination of several path expressions and the predicate expression for the selector variables. For instance, the following extended PED may be considered:

$$\{Car[X].engine[Y],\ Car[Z]\} \text{ where } X \neq Z \Rightarrow Car[Z].engine[W] \text{ where } W \neq Y$$

The intuitive semantics is that; When there exist two path, $p_1 = o_1.engine$ and $p_2 = o_2.Id$, in the database such that $o_1 \neq o_2$, the relationship, $tail(o_1.engine) \neq tail(o_2.engine)$, holds. Of course, such extensions are need for careful discussion, and, it will be treated in future research.

5.1.2 Existence Dependency

W. Kim introduced the another constraint, the Existence Dependency, in [7]. This is represented by our path existence dependency. For instance, "For each *Engine* object, there must exist a *Car* object that has it for the attribute value of the *engine* attribute," which constraint is represented by the following PED:

$$Engine[X] \Rightarrow Car.engine[X]$$

5.1.3 Functional Dependency

The functional dependency [13] considering the navigating operations in OODB, *path functional dependency* (PED) discussed by Weddell [9, 8]. The possibility for PED to represent Weddell's notion of PFDs. Here, for simplicity, the following simple example; If two *Car* objects have the same value in *model* attribute. Then, the capacities of engines of these car are also same. This is represented by the following PFD:

$$Car(\{model\} \Rightarrow engine.capacity)$$

In order to represent this PFD, the definition of PED should be extended as the following notation can be used:

$$\{Car[X].type[Y], Car[Z].type[Y]\} \Rightarrow \{Car[X].engine.capacity[Z], Car[Y].engine.capacity[Z]\}$$

But, in this case, several extensions of definitions and modifications of semantics for PEDs are needed. Therefore, further careful discussion in detail will be needed.

5.1.4 Object Existence Dependency

In the DBTG network data model[3], the membership operations, *FIXED, MANDATORY* and *OPTIONAL*, among records are already defined. These operations mean some kind of dependencies among records in the database, however, these dose not corresponded to the notion of PEDs directly.

They do not represent the static status of the membership among objects, but the dynamic aspects to maintain the membership on the creation and deletion of objects. This means the following; Under several PEDs are given, what attribute reference should be maintained on the creation and deletion of objects. That is, this is the problem of interpretation for PEDs on execution. Therefore, this will lead to the issue of concurrency controls or active databases.

5.2 Application to Practical Systems

In order to show usefulness for the practical application of the notion of PEDs, For instance, we suppose the computer aided design (CAD) system and manufacturing management system in the construction area.

5.2.1 Integrity Constraints in CAD

Assume that `Window`, `Wall` and `Floor` objects exist in a CAD system for construction. The *support* attribute in each object denotes the another object which should be supported (Figure 11). Here, the following constraints are obvious; A window must be supported by a wall. That is, there must exist a `Wall` object to support for each `Window` object. This is represented by the PED:

$$Window[X] \Rightarrow Wall.support[X]$$

In the same way, the constraint: a `Wall` must be supported by a `Floor` is represented by:

$$Wall[X] \Rightarrow Floor.support[X]$$

From the above PEDs and inference rules for PEDs,

$$Window[X] \Rightarrow Floor.support.support[X]$$

is derived[8]. This intuitively means that "a window must be indirectly supported by way of a wall." The constraints in this example are regarded as a kind of the existence dependency by W. Kim.

5.2.2 Production Management System

Here, we suppose a manufacturing management system for construction parts (See Figure 12). And suppose that a `Wall` is assembled with an `OuterPanel` and `Pillar` object, and an `OuterPanel` is constructed by a `Panel` and a `Window` in general. Furthermore, the information for factories, which these objects are produced or assembled, is denoted by the *factory* or *assemble* attribute in each part object. Let us consider that the following constraint is imposed on a special wall, `SWall` class;

> In the parts constructing the `SWall`, the `Panel` and `Window` must be manufactured in the same factory which assembles `SWall` itself.

In the big company spread on world wide, since the production management to minimize for the cost of parts transportation is a matter of course, the above constraint is fully considerable. This is represented by the following PED;

$$SWall[X].assemble[Y] \Rightarrow SWall[X].outerPanel.panel.factory[Y]$$
$$SWall[X].assemble[Y] \Rightarrow SWall[X].outerPanel.window.factory[Y]$$

In the object-oriented databases, the notion of *complex objects*, are very useful in order to represent the semantic relationships among objects. In this case, the information about *factories* are denoted by `Factory` objects in *assemble* or *factory* attributes. By introducing the notion of PEDs, we can give the significant integrity constraints to the reference (navigable) path among related objects.

6 Concluding Remarks

In this paper, we introduced the notion of *PEDs* to represent the integrity constraints between object reference paths in OODBs. The *PED* (Path Existence Dependencies) denotes the following; The path expression on the left side represents the arbitrary reference paths in the database, and the right side represents the reference paths, which allowed to exist under the existence of, at least, one path such that it conforms to the left side. With this notation, the existence dependencies among *reference paths* in OODBs can be treated. Furthermore, by introduction of *selector variables*, the suitable object on the reference paths can be specified. We also showed several inference rules for deriving the

[8] Although the process of derivation is omitted because of lack of pages, it is easily probed.

new PED from the given PEDS. With this notion, the formal basis for the constraints about existence dependency in OODBs will be formed. Also, we discuss about the several example to show the effectiveness of the notion of PEDs for the practical applications.

Further research will be needed about, especially, the theoretical properties of PEDs. The following issues should be focused:

- Soundness and completeness of the inference rules for PEDs.
- Extensions of the expressive power of PEDs.

Since a PED restricts the allowed combination of the null values, it is a very interesting issue whether the admissible combinations of attribute values in a object can be derived from the given PEDs or not. In other words, the admissible null pattern of the attribute value in an object. Furthermore, it is also interesting that the introduction of the regular expression[12, 14] in PEDs.

References

[1] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonic, S., *The Object-Oriented Database System Manifesto,* Proc. of The first International Conference On Deductive and Object-Oriented Databases (DOOD'89), pp.40-57, Dec. 1989.

[2] Deux, O. et al., *The Story of O_2,* Trans. on Knowledge and Data Engineering, Vol. 2, No. 1, pp. 91-108 (March 1990).

[3] Date, C. J., An Introduction to Database Systems 3rd Edition, Addison-Wesley, 1975.

[4] Khoshafian, S. N. and Copeland G.P., *Object Identity,* Proc. of OOPSLA'86, pp.406-416, 1986.

[5] Kifer, M., Kim, W., Sagiv, Y., *Querying Object-Oriented Databases,* Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pp393-402, June 1992

[6] Tsukamoto, M., Nishio, S. and Fujio, M., *Dot: A Term Representation Using Dot Algebra for Knowledge-Bases,* Proc. of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91), Lecture Notes in Computer Science, Vol. 566, pp.391-410, December 1991

[7] Kim, W., Bertino, E., and Garza, J. F., *Composite Objects Revisited,* Proc. of ACM SIGMOD, pp.337-347, June 1989.

[8] Weddell, G. E., *Reasoning about Functional Dependencies Generalized for Semantic Data Models,* ACM Trans. on Database Systems, Vol. 17, No. 1, pp.32-64, March 1992.

[9] Coburn, N. and Weddell, G. E., *Path Constraints for Graph-Based Data Models: Towards a Unified Theory of Typing Constraints, Equations, and Functional Dependencies,* Proc. of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91), Lecture Notes in Computer Science 566, Springer-Verlag, pp.312-331, Dec. 1991.

[10] Cattell, R. G. G., *Object Data Management,* Addison-Wesley, 1991.

[11] Cattell, R. G. G. Ed., *The Object Database Standard: ODMG-93,* Morgan Kaufman Pub., 1994.

[12] Yoshikawa, M., *Circulation and Reuse of Database Constructs through Common Dictionaries,* Future Databases '92 (Proc. of the 2nd Far-East Workshop on Future Database Systems), Advanced Database Research and Development Series - Vol. 3, World Scientific, pp. 67-70, April 1992.

[13] Ullman, J. D., *Principles of Database Systems,* Computer Science Press, 1980.

[14] Yoshikawa, M., Tanaka, K., Jozen, T., Tanaka, Y., Hirui, J. and Hotta, K., *ObaseLang: An Object Database Language with Flexible Syntax and Extended Path Expressions,* Trans. of Information Processing Society of Japan, Vol. 36, No. 4, pp.981-993, April 1995.

[15] Kamitani, M., Oomoto, E. and Tanaka, K., *Path Existence Dependency and Attribute Key Constraints in Object-Oriented Databases,* Proceedings of Advanced Database System Symposium '93, Vol. 93, No. 9, pp.73-82, Tokyo Japan (Dec. 1993).

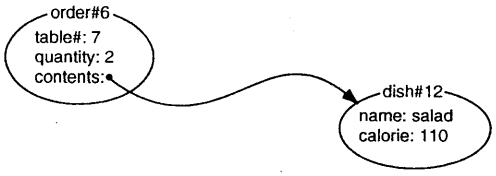

Figure 1: The Relationship between a Dish and an Order

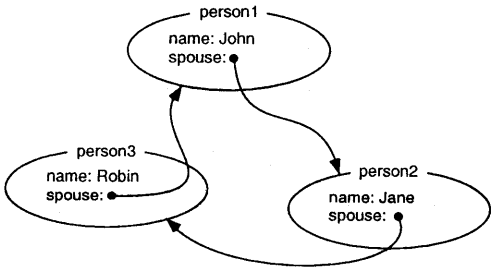

Figure 2: An Invalid Example on Monogamy

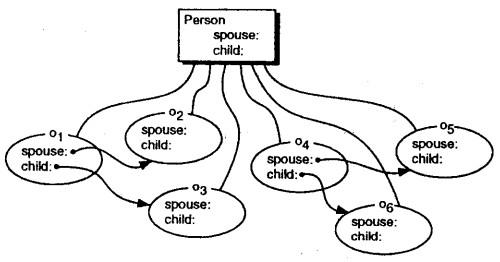

Figure 3: Example Database

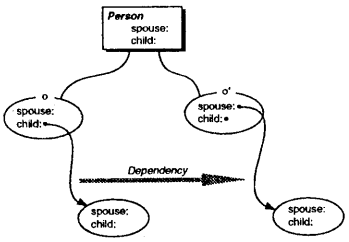

Figure 4: Example of PED

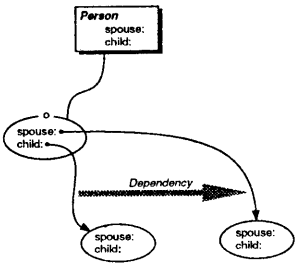

Figure 5: Dependency between A Person's Child and Spouse

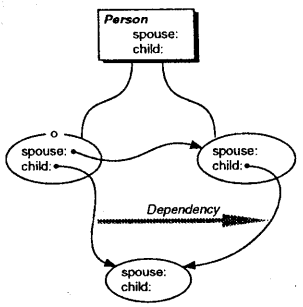

Figure 6: Sharing a Common Child of a Couple

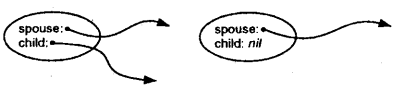

Figure 7: Admissible Combinations of Attribute Values

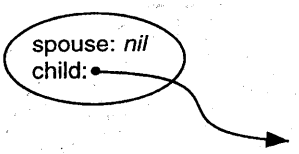

Figure 8: Inadmissible Combination of Attribute Values

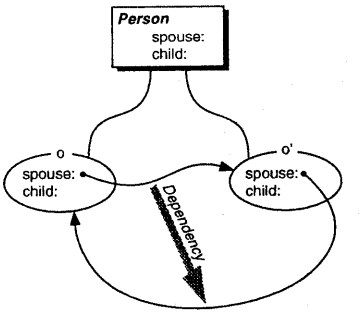

Figure 9: Mutual Dependencies

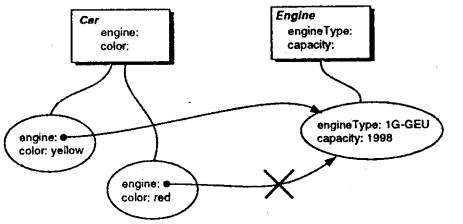

Figure 10: Exclusive Reference Constraint

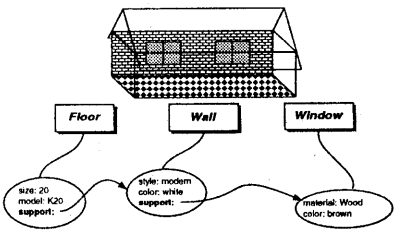

Figure 11: Dependencies among Window, Wall, and Floor

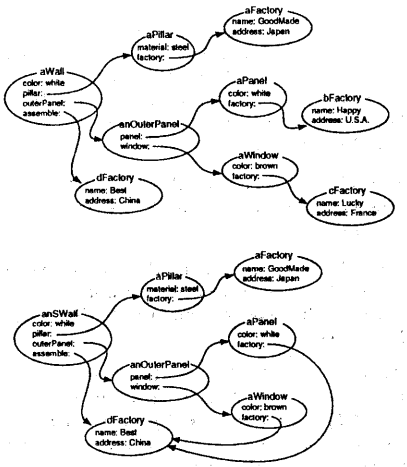

Figure 12: An Example of Production Management