

Class Query Language and its application to ISO13584 Parts Library Standard

Yumiko Mizoguchi-Shimogori, Hiroshi Murayama, Noriko Minamino
Corporate Research & Development Center
TOSHIBA Corporation
1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki
212-8582, JAPAN
E-mail: hiroshi.murayama@toshiba.co.jp

KEYWORDS

CQL, PLIB, ISO13584, Set-theoretic, Product Data Interchange and Standards.

ABSTRACT

Set operations on tables are widely used by RDBMS (Relational Database Management Systems) and play a pivotal role in data representation, processing, and storage. This is because the operations and queries about data in RDBMS are performed by a standardised language known as SQL (Structured Query Language) and the basis of the language is well-founded in mathematic logic, i.e., the set-theoretic Boolean algebra. Thus the users of the RDBMS may retrieve the data and predict the resulting set based on mathematic logic, without knowing the algorithmic detail of the data-interface language. In contrast, the users of OODBMS (Object-Oriented Database Management System) must well understand the data-interface specification of the embedding language, because the data-retrieval is done procedurally and is influenced by its syntax.

The Class Query Language(CQL), presented herein, is a theoretical extension of the SQL toward the classes in mathematical logic, and is capable of performing set-theoretic Boolean operations between those classes that form an acyclic graph structure. This language is especially useful to retrieve, manipulate, and construct data in an ISO13584 Parts Library standard (PLIB) based Library Management System. However the scope of the language is not limited to PLIB but is applicable to wide-ranging object-oriented class libraries.

1. Introduction

Set operations on tables, first introduced by E.F. Codd (Codd 1970), are widely used in Relational Database Management System (RDBMS) and still play a pivotal role in data representation, processing, and storage. This is because the operations and queries about data in a Relational Database (RDB) are performed by a query language known as SQL (Structured Query Language) and the basis of the language is well-founded in mathematic logic, i.e., the set-theoretic Boolean algebra. Thus the users of the RDBMS may select the required data and formally calculate the resulting set based on mathematical set-theory, without knowing the algorithmic detail of the data-interface language; the users don't have to mind the procedural steps to retrieve the information, because the query is declarative and not procedural. In contrast, the users of Object-oriented Database

Management Systems (OODBMS) must well understand the detailed specification of the embedding language, often called "language-binding", because the data-processing is performed procedurally and thus the result is totally order-dependent on the steps of procedures and is largely influenced by the syntactic specification of the data-interface of the embedding language, provided that this may optimize data processing time for certain objects and for certain purpose, if the operations on them are well standardised and well understood by the users as analogous to the operations on real world objects.

The Class Query Language (CQL, hereafter), presented in this paper, is an extension of the SQL language toward the classes of mathematical logic, and is capable of performing set-theoretic Boolean operations between classes that together form an acyclic graph structure (ACGS). In particular, this language is effective to retrieve and manipulate data stored in a Library Management System (LMS) based on ISO13584 Parts Library standard (PLIB), but the scope is not limited to PLIB but also to cover similarly structured class library systems that have a graph-like ontology representation. In such a system, each node is supposed to be represented by a class whose members are characterised by having at minimum the same set of properties, and data records are represented as instances of one of such classes. To be exact, PLIB does not define the database structure itself, but only defines the file format of data exchange between LMS's or product parts databases, but the basic data requirement of the exchange file virtually defines the underlying conceptual database model.

2. Theoretical Background

The notion of class employed in this paper, and henceforth within CQL specification, corresponds to the class notion in mathematical logic, which is roughly a "superset" of mathematical set. This notion of the class whose members are qualified by having the same set of properties can be expressed by a predefined set of logical formulae, known as Zermelo-Fraenkel classes (ZFC), but does not necessarily coincide with the class notion of Object-oriented Programming language (OOP) which merely implies such an entity be a named aggregation of data records of composite data types, that allows inheritance of member attributes to its subclasses, often accompanied by a functionality of selective data encapsulation. A class in OOP does not necessitate that every class be a member of another class, neither the collection of those classes forms an ACGS. In a marked difference to this, every class in the sense of

mathematical logic is a subset of *Universal Class*, or *Universe*, thus having the same ancestor node. This class definition neither corresponds with that of semantic ontology in which the analyses of the meaning of vocabulary plays a significant role in the configuration of class hierarchy. In CQL, a subclass A' of class A can be defined strictly by the formula (1) in the following;

$$A' \subseteq A \leftrightarrow \forall x(x \in A' \rightarrow x \in A). \quad (1)$$

Hence, the subclass A' must have all the properties that class A has and may have some more properties than class A .

In almost all the cases, classes and sets are mutually replaceable, however, according to the textbooks of mathematics such as (Jech 1997) or in a classic work of mathematical logic such as (Church 1956), a set is always a class but the reverse is not always true. A well known example of such a class being not equivalent to a set is known as Russell's paradox that states a set S is a collection of elements such that they are not a member of themselves;

$$S = \{ X \mid X \notin X \} \quad (2).$$

A class having the above characteristic is given a special name, "*Proper Class*". Beside such special cases given in (2), all the Boolean operations that apply to sets apply to classes as well;

$$A \cap B = \{ X \mid X \in A \wedge X \in B \}, \quad (3)$$

$$A \cup B = \{ X \mid X \in A \vee X \in B \}, \quad (4)$$

$$A - B = \{ X \mid X \in A \wedge X \notin B \}. \quad (5)$$

In this paper, authors denote the *class* with the above definition, but also connote that an assemble of those classes form an ACGS, and it must be distinguished from a simple table composed of a list of tuples, having the same set of data types. As well known, the Boolean algebra allowed in case of RDB are essentially limited within a table or between such tables that have the same set of columns.

The CQL differs from OQL(Object Query Language) proposed by ODMG (Object Data Management Group) (Cattell and Barry 1999), in that CQL allows set-theoretic Boolean operations to define a new class and to extract information, while the OQL uses methods defined in a class to do so. Thus in the OQL, the actual creation of a class depends on what properties it has, in which order the properties are defined, and how the constructor arguments are specified. In CQL, a class is defined by a set of properties but the order of definition of those properties has no relation with the class creation, nor with the data operation. In fact, each property as well as each class is identified by a unique code, named BSU (Basic Semantic Unit) whose issuing organisation is identified by a code named "Supplier BSU", whose uniqueness and integrity are guaranteed by the ICD (International Code Designator) defined in ISO6523. In addition, the collection of properties and classes is bundled together as an exchangeable meta-data, and given a special name, i.e., "dictionary" or "data dictionary".

3. CQL LANGUAGE---SCOPE AND DEFINITION

In the following, an outline of the features of CQL version 1.4. is presented first, and then a structural comparison among CQL, SQL and OQL is shown. Some of the language constructs in view of the application of the language to PLIB are illustrated in the last section.

3.1 Typical Features of CQL

The CQL has the following five distinguished features.

- 1) Boolean operation between classes
- 2) Ancestral search
- 3) Part-Whole Relationship
- 4) Property classification
- 5) Three level security controls based on a business model

3.1.1 Boolean operation between classes

The Boolean operation between operand classes modifies the search scope of the query in the class hierarchy. The expressions are given as below.

$$\text{select } * \text{ from } A^* \text{ AND } B^*; \quad (6)$$

$$\text{select } * \text{ from } A^* + B^*; \quad (7)$$

$$\text{select } * \text{ from } A^* - B^*; \quad (8)$$

where "*" attached to a class variable signifies all the subclasses of the class are included in the operand.

It is clear from the comparison, that the expression (6), (7), (8) corresponds to the mathematical expression (3), (4), (5) respectively, where X is taken to be a collection of instances belonging to a class or to one of its subclasses.

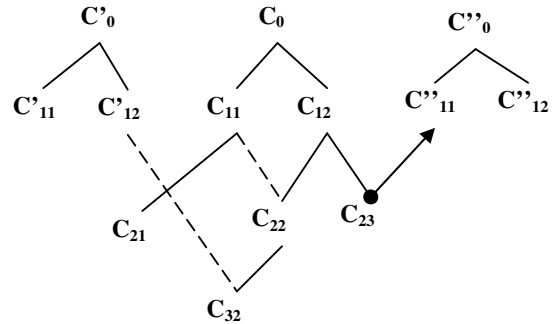


Figure 1: Sample Class Tree

For example, in a class hierarchy like the one depicted in Figure 1, if the *from* clause is written "*from C₁₁*", the search scope is strictly limited to the class C_{11} itself. But if the *from* clause is "*from C₁₁**", the search scope is the class C_{11} and its subclasses. Therefore, *AND* operation of the class C_{11}^* and the class C_{12}^* means that the search scope is the class C_{22}^* . Likewise, "*from C₁₁* and C₁₂**" yields the scope "*from C₂₂**".

Meanwhile, subtraction operation, "*from C₀* - C₂₂**" means that the class C_{22} and subclasses are removed from the search scope of C_0^* .

3.1.2 Ancestral search

The Ancestral search extends the scope of search from a child to parent nodes. The expressions are given as in the following;

select * from A%; (9)

select * from A! ;.. (10)

The expression (9) extends the search scope to all the superclasses of A, while the expression (10) limits the scope to direct ancestors of A. In the example given in Figure 1, if the **from** clause after **select** is “**from C₂₂%**”, the search scope is extended from C₂₂ to all the ancestors, i.e., C₁₁, C₁₂ and C₀. On the other hand, if the clause is “**from C₂₂!**”, the search scope is limited to only its direct ancestors and excepts the class C₁₁ that exports properties to C₂₂. Consequently, the search scope is C₂₂, C₁₂ and C₀. In Figure 1, a dotted line is used to signify a secondary line of inheritance or a partial inheritance that is given a special name “*case_of*” in PLIB standard. In fact, PLIB allows single inheritance alone, but enables importation of designated properties from classes other than its superclass. In this way, it effects an *interface* for those properties, as it is in JAVA™ language. In the same manner, CQL can deal with multiple inheritance, though it arbitrary takes one of its superclasses as the primary superclass for “* **from A!**” type of operations.

3.1.3 Part-Whole Relationship

The Part-Whole relationship enables conditional reference from the whole(body) of a product to constituent parts. The Part-Whole relationship appears in the expression (11),(12),(13) for search, construction and updating;

select P_n from A where B.P' = 'condition'; (11)

insert into A(P_n) values (
B condition qid B.P' = 'condition'
); (12)

update A set P_n =
B condition qid B.P' = 'condition' where P₀= n;
(13)

In PLIB a specific data type named “class instance”, represents this part-whole relationship. But this “class instance” differs from the conventional *part-of* in OOP in that the parts are referred not by an object-ID but by a set of reference conditions on membership.

In Figure 1, an arrow from C₂₃ to C''₁₁ represents *Part – Whole* relationship between C₂₃ (*Whole*) and C''₁₁ (*Part*).

The example in Figure 2 shows the image of this reference relationship from an instance in the *Whole* to a set of instances in a *Part*. Q₁,Q₂,Q₃ are conditions for extracting a set of suitable member instances. In Figure 2, the expression (14) retrieves the set of members from the *Part* class C''₁₁.

select P₂ from C₂₃ where C''₁₁.P'₁ = v₁₁; (14)

To refer to a set of instances in *Part* class from an instance in *Whole* class, it is necessary to set up a condition in the “class

instance” type property. The following expression (15),(16),(17) sets the condition.

insert into C₂₃(P₀, P₁, P₂) values (
v₀₁, v₁₁, C''₁₁ condition Q₁ C''₁₁.P'₁ = v₁₁
); (15)

The expression (16) allows setting the value range as the condition.

insert into C₂₃(P₀, P₁, P₂) values (
v₀₂, v₁₂, C''₁₁ condition Q₂
C''₁₁.P'₁ > v'₂₂ and C''₁₁.P'₁ < v'₂₄
); (16)

The expression (17) sets a condition by using a variable instead of a value.

insert into C₂₃(P₀, P₁, P₂) values (
v₀₃, v₁₃, C''₁₁ condition Q₃ C''₁₁.P'₁ = C₂₃.P₁
); (17)

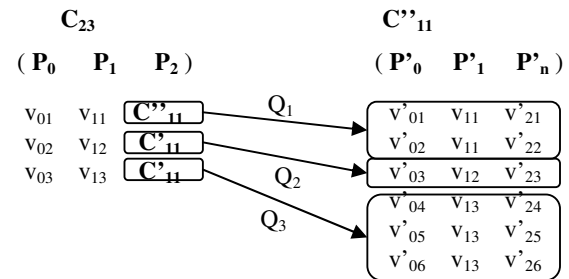


Figure 2: Class Instance

3.1.4 Property classification

The fourth feature, “Property classification”, groups the properties by a label named *classification*, that is currently a number. The expression is (18).

create extension on A (
P₁,
....
P_i constraint cid classification -100,
....
P_n
); (18)

where A is a class and P₁, P_i and P_n are properties.

The system may perform different process according to the *classification* of the properties. It depends on the implementation what kind of process is actually assigned to it, but the original intent of this feature is to control the visibility of properties depending upon the user groups.

3.1.5 Layered Security Controls

The last feature, “Layered Security Controls”, exerts security control at the three levels, class, property, and instance.

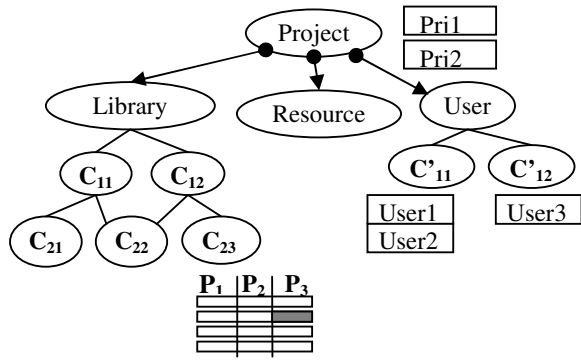


Figure 3: Special reserved classes for administration

In addition, CQL provides the following four special-purpose classes to administer the LMS.

- 1) **Project**
- 2) **Library**
- 3) **Resource**
- 4) **User**

The **Project** is the basic unit of data management. If a user wants to address a product data through CQL, he needs to be enrolled in at least one of the **project instance** as depicted in Figure 3 that has one or several libraries of products. This concept does not derive from PLIB, since PLIB presumes the use of a textual file for data exchange; however, it is necessary for the CQL-based LMS to manage projects that utilise resources, because several people are engaged in several projects to exploit limited resources. The **Project** is the *whole* class for the three *Part* classes, **Library**, **Resource**, and **User**.

The **Library** is the root class of ordinary class hierarchies modelled in a project. From time to time, several libraries may be found in a projects. A library (in plain letter) here means a class hierarchy(ies) with product instance data. The **Resource** is the class which stores a physical file such as image files related to the project. The **User** is the class which stores information about the users of the project.

The **User** class may have subclasses, and each such a class has individual users as its instances.

Now, if a system grants a privilege to a class under **User** to do a specific operation to specific data, the expressions are as follows. To give access right to the classes under **A** to a **User** class,

$$\text{grant select on } A^* \text{ to } \text{User_class}; \quad (19)$$

If the privilege should be revoked,

$$\text{revoke select on } A^* \text{ from } \text{User_class}; \quad (20)$$

CQL grants privileges to a class under **User** instead of an individual user. Thus all the users, belonging to a class under **User**, have same privileges. By appropriately organizing classes under **User**, based on a business model, user access may be controlled for each class of people.

The example in Figure 3 shows a project's structure. The following expression grants "select" operation on class **C₁₂** and its subclasses to the **User** class **C'11**.

$$\text{grant select on } C_{12}^* \text{ to } C'_{11}; \quad (21)$$

Other operations could be also permitted instead of "select". Secondly, CQL can exert access control over a property. The class **C₁₂** is assumed to have properties, **P₁**, **P₂**, **P₃**. In this case the following expression revokes "select" operation on the property **P₁** from the **User** class **C'11**.

$$\text{revoke select on } P_1 \text{ from } C'_{11}; \quad (22)$$

At third, it can exert access control about an instance. The following expression revokes "select" operation on the class **C₁₂** and subclasses with the **where** clause meets the condition specified by "**P₃ := 'painting'**" from the (users in) class **C'11**.

$$\text{revoke select on } C_{12}^* \text{ where } P_3 \text{ := 'painting' from } C'; \quad (23)$$

"Property classification" that has been mentioned before, may also be used as security level classifier for properties.

3.2 Structural comparison of CQL, SQL, and OQL

In application of a database language to PLIB, we must first consider that PLIB has rich reservoir of data constructs to represent product data. In particular, it has an ability to describe a product that is composed of several components or parts. In addition, PLIB is extensively used in many leading industrial projects such as MERCI (Wilkes and Broking 2000) ,in Japan and in Europe to replace old-fashioned product database systems. Thus the language to describe product library system must reflect the PLIB model that requires the following four layers for representation.

- 1) Meta-schema for housing ontology
- 2) Ontology data as a class hierarchy or dictionary
- 3) Product database Schema
- 4) Database content data (product instance data)

Class library such as PLIB must provide competence to model those layers at least from the second to the fourth, and the first layer might be necessary if an LMS (Library Management System) is in consideration for the incorporation of various product libraries over time. SQL could define the data structure of a table and its attributes, but it doesn't take care of the ontological structure among tables. SQL-99 allows inheritance of data elements, however it only takes care about simple inheritance from another table. Thus, it is difficult to represent product ontology by SQL.

On the other hand, OQL uses methods to define a class and store data. Because of the data encapsulation, indeed, an OQL user has to comprehend the intent of the class-designer to effectively manipulate data, for the views of the world are different among designers of the objects and there is no "objective view" about this. Note that CQL uses Boolean operations between classes and the specification of class properties in where clause to retrieve the requested data. Thus the order of the property specifications is irrelevant to the resulting set.

CQL provides three types of definition language, the first is Class Modeling Language (CML), the second is Data Representation Language (DRL), and the third is Library Manipulation Language (LML). In addition, if CQL processor is available, a record of CQL commands stored in a file addressed the necessity for the first layer. Thus, the CML manages the product ontology and also a meta-schema for ontology, the DRL manages a database schema and the LML manages content data. *Table1* shows the comparison among CQL, SQL, and OQL in each PLIB layer.

Table 1: Comparison of Query Languages

	CQL	SQL	OQL
Product Ontology	CML (create class alter class i.e.)	-----	Method
Database Schema	DRL (create alter i.e.)	DDL (create alter)	Method
Content Data	LML (insert, update i.e.)	DML (insert, update i.e.)	Method

3.3 Application of CQL Language to PLIB

3.3.1 Basic notation of property and class.

The fundamental concept of CQL about object identification is that it identifies an object by a code not by a name of the object. In addition, in application of the CQL to PLIB, some additional functionalities are provided. Firstly, the name space resolving functionality that converts a name of a given object in a known context into a full BSU notation. This does not imply that any automated semantic analysis of terms was brought into CQL like the one in Semantic WEB project in W3C. Unlike the semantic approach, CQL only enables a kind of shortcut notation at the level of user-interface, that translates an input class name into a full fledged BSU notation since a property of a class is noted as,

supplier_BSU. class_BSU. property_BSU . (24)

The name space resolver only adds a supplier_BSU code to the class_BSU, linked by a dot when a supplier_BSU code is defined a priori as the default value.

Secondly, following the description of updating rules defined in Part42 (Pierra 1997) and Part24 (Pierra 2001) of ISO13584, the version and revision controls are added to the BSU notation of CQL. Thirdly, the dictionary is assigned an identifier with a version and a revision; when two systems employ exactly the same set of classes and properties, the two must have the same dictionaries with the same version and the same revision. This will facilitates the parts library exchange between two systems, since the export of data from one system to another is only possible if the target system has an identical or a newer version of the dictionary .

3.3.2 Usage examples of CQL

The example in *Figure 4* shows a simple class tree of vehicle and each class has properties as illustrated in *Figure5*. The class tree and the properties are simplified just for the ease of explanation. For simplification, hereafter, supplier_BSU's are omitted for the notation.

- *Creation of Class and Property, and its container*

The class and properties are defined by CML.

The following expression creates a class named "Electric Vehicle" as in *Figure 4*. The class_BSU code is assumed to be "Electric" for the ease of understanding, although in PLIB this is an unrecommended practice.

```
create class Electric (
  super_class Ecological,
  preferred_name 'Electric Vehicle',
  short_name 'EV',
  definition 'The Vehicle equipped with
    the electric motor as a motor.'
) (25)
```

where "super_class", "preferred_name", "short_name", "definition" .. i.e., are reserved words in PLIB.

Next, the following expression creates a property named "motor_max_power" within the class "Electric Vehicle".

```
create property motor_max_power (
  preferred_name 'motor maximum power',
  name_scope Electric,
  data_type real_measure_type,
  unit 'kW',
  definition 'The maximum power of motor which
    can be exploited within the limit of
    the number of rotation.'
); (26)
```

Likewise "case of" class, is constructed as follows;

```
create class Hybrid (
  super_class Electric,
  preferred_name 'Hybrid Vehicle',
  short_name 'Hybrid car'
  definition ',
  is_case_of (Sedan)
  imported_properties (displacement, engine_type)
); (27)
```

After the creation of classes and properties, a container for storing instances is required. It is defined by DRL. The following gives the container of the class "Electric Vehicle".

```
create extension on Electric(
  pid, name, maker, weight, max_speed,
  motor_max_power, motor_type
); (28)
```

Similarly the container for the "Hybrid Vehicle" can be built.

Now, it is ready to insert an instance into each container. LML manipulates an instance into a container. In the following manner, instances are inserted into respective containers:

```
insert into Electric(pid, name, maker, weight, max_speed,
motor_max_power, motor_type)
values ('2', 'EV1', 'GM', ,129, 102, 'Transverse-
mounted, front-wheel drive' ); (29)
```

```
insert into Hybrid(pid, name, maker, weight, max_speed,
motor_max_power, motor_type, displacement,
engine_type)
values ('4', 'prius', 'toyota', 1220, , 33, '2CM', 1496,
DOHC); (30)
```

```
insert into Sedan(pid, name, maker, weight, max_speed,
displacement, engine_type)
values('5', 'E CLASS', 'BENZ', 1690, , 4265, 'SOHC')
(31)
```

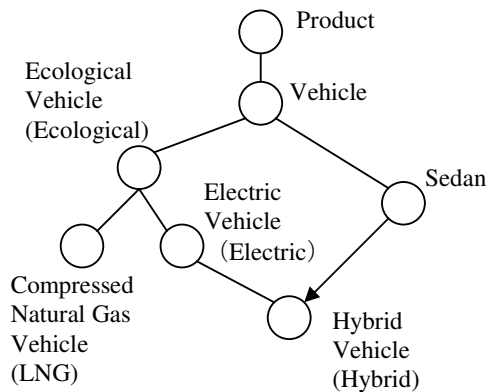


Figure 4: Sample tree for vehicle

```
Product(pid, name, maker, price)
Vehicle(weight, max_speed)
Sedan(displacement, engine_type)
Electric Vehicle (motor_max_power, motor_type)
```

Figure 5 : Sample: Class (Properties)

- Querying the data

The formal syntax of CQL for *select* operation is as follows:

```
SELECT * | [supplier_BSU.class_BSU.] property_BSU
{ , [supplier_BSU.class_BSU.] property_BSU }
FROM [supplier_BSU.] class_BSU[*] { [ AND | - | ,
[supplier_BSU.] class_BSU[*] }
[WHERE [supplier_BSU.class_BSU.] property_BSU
OPERATOR VALUE
{ [AND | OR]
[supplier_BSU.class_BSU.] property_BSU
OPERATOR VALUE }
[ INSTANCENUM
COMPARISON_OPERATOR NUMBER ]
]
[ORDER BY [supplier_BSU.class_BSU.] property_BSU
```

```
[ASC] | [DESC]
{ , [supplier_BSU.class_BSU.] property_BSU
[ASC] | [DESC] } ; (32)
```

OPERATOR may be either <, >, =, <=, >=, <>, LIKE . *VALUE* may be a literal value or a sequence of *supplier_BSU.class_BSU. property_BSU* . *COMPARISON_OPERATOR* may be either <, <= . *NUMBER* must be a number value.

In the example given in Figure 4, the subclasses inherit their superclass's properties. The class "Electric Vehicle" is a superclass of the "Hybrid Vehicle" class, and the "Sedan" class exports its own properties to the "castoff" class "Hybrid Vehicle".

If the user would like to search instances whose maker is 'toyota', without caring about detailed classification, the following query expression is helpful.

```
select * from Vehicle* where maker = 'toyota'; . (33)
```

The result gives instances found not only in the class "Vehicle" but also in the subclasses, "Sedan", "LNG", and "Hybrid Vehicle". Figure 6 gives the summary of this.

lpid	name	lmaker	lweight	lmax_speed
12	EV ₁	GM	129	
14	prius	toyota	1220	
15	E class	BENZ	1690	

Figure 6: The result example

If the user would like to search the ecological vehicles with the characteristics of Sedan, the Boolean operation like the following expression is useful.

```
select * from Ecological* AND Sedan*; (34)
```

The result of this query might be the instances in "Hybrid Vehicle".

lpid	name	lmaker	lweight	lmax_speed
14	prius	toyota	1220	

Figure 7: The result example for the Boolean operation

Like an Electric Vehicle, according to the development of new technology, it might be necessary to create a new subtype of vehicle class. For this, the class classification needs to evolve. In this situation, the instances built with conventional technologies would remain in some ancestral classes, while the new instances would be stored in a derived class. The following expression provides the ancestral search for those conventional instances.

```
select * from Hybrid% where motor_max_power > 30 (35)
```

This query searches not only "Hybrid Vehicle" but also the ancestor classes having the property "motor_max_power".

Thus the search scope extends to “Hybrid Vehicle” and “Electric Vehicle”. Consequently ,and the result of this query is summarised as in the following.

lpid	name	maker	weight	max_speed	motor_max_power
12	EVI	GM	1	129	1102
14	prius	toyota	1220	1	133

Figure 8: The result example for the ancestor search

If the search scope is required to except the class with “case of” relationship, “!” is used instead of “%”.

3.3.3 Class instance and the conditional reference

Tires are considered to be parts of a product, named car. Since class instance make reference from products to parts, we add the tire property whose data_type is class_instance, to the Vehicle class. This property refers to the tire class.

```
create property tire(
    preferred_name 'tire information',
    name_scope Vehicle,
    data_type class_instance tire
);
```

(36)

And then, we create the tire_size property and the container for contents.

```
create property tire_size(
    preferred_name 'tire size',
    name_scope Vehicle,
    data_type string_type
);
```

(37)

```
create extension on Sedan(
    pid constraint spk1 key,
    name constraint spk2 key,
    maker constraint spk3 key,
    weight,
    max_speed,
    displacement,
    engine_type
    tire_size,
    tire
);
```

(38)

The reference condition is set by the following expression. It means this product refers to tires whose size is specified as ‘175/70R13 82S’.

```
insert into Sedan(pid, name, maker, tire_size, tire)
values('7', 'familia', 'mazda', '175/70R13 82S',
    tire condition q1 tire.size = '175/70R13 82S');
```

(39)

In conditions, variables are also available. The following expression uses Sedan’s property “tire_size” as a variable instead of the string value.

```
insert into Sedan(pid, name, maker, tire_size, tire)
values('7', 'familia', 'mazda', '175/70R13 82S',
```

```
tire condition q1 tire.size = Sedan.tire_size );
```

(40)

maker	name	size	outline_size
BRIDGESTONE	GR-7000	175/70R13 82H	578
BRIDGESTONE	GRID-II	175/70R13 82H	576
BRIDGESTONE	B-Road SF-270	175/70R13 82S	576
FALKEN	SINCERA	175/70R13 82S	576
TOYO TIRE	TRANPAT	175/70R13 82S	576

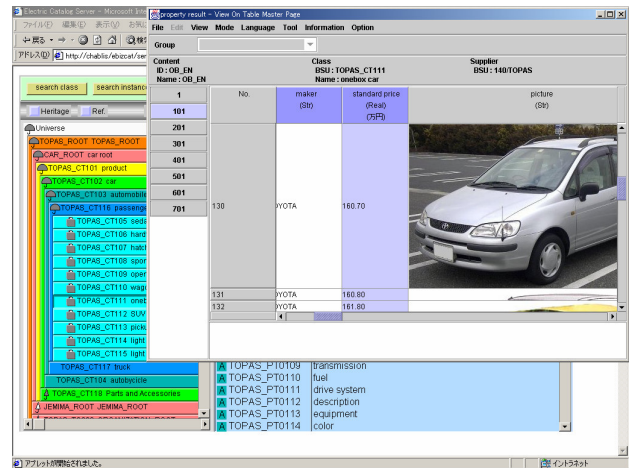
Figure 9: The sample of tire instances

If the tire class is assumed to have instances in Figure 9. The applicable tires for this car would be the instances given in Figure10.

maker	name	tire_size	outline_size
BRIDGESTONE	B-Road SF-270	175/70R13 82S	576
FALKEN	SINCERA	175/70R13 82S	576
TOYO TIRE	TRANPAT	175/70R13 82S	576

Figure 10: The applicable tires

In this way, the car makers and tire makers can concentrate on building their own catalogues without incorporating the data of the other into the body of their catalogue description. Also, this helps making the process of the engineering more concurrent.



Figures 11 OmniPhase Screen Image

4. Conclusion and Future Prospect

In this paper, a new database language short named “CQL” is presented. This language has an advantage over SQL in that it can model class hierarchy. It also has an advantage over OQL in that it employs Boolean operation for data operation. This gives ease to end-users and eliminates the necessity to know about the detail of the class definition or the specification of embedding language. Lastly some examples of CQL operation to build a simple class library for automobiles are presented.

With the implementation of CQL in our laboratory, construction of an extended PLIB-LMS code named “OmniPhase” is under way. Figures11 gives a glimpse of

the image of the software. In the system, the CQL is exploited for two objectives; one is to facilitate the communication between database server and its client programs. This was actually programmed in JAVA™ language, and the server employs servlet technology and most of the clients are realised with applets including swing libraries. In one client program, the Java Scripts and DHTML are used to give ease to the integration with external application programs. The other objective is to give user interface to external users or programs. This interface is currently programmed in C++ language and it renders service to a CGI program that is also programmed in C++ by another company. This interface allows a connection to a client program with an `execClassQuery` statement, like the one available in JDBC (Java Database connection). More detailed specification is given in one of our PLIB sites specified by the following URL:

<http://www.toplib.com/en/cql.html>

5. Acknowledgement

Authors express special thanks to Professor Guy Pierra of Ecole Nationale Supérieure de Mécanique et d'Aérotechnique (ENSMA) in France who kindly reviewed an early version of CQL specification and gave us useful comments. Also, we'd like to extend our appreciation to Mr. Akihiro Tomita and his team in Toshiba Information Systems Japan who helped design the CQL language and its integration into our actual PLIB-LMS.

REFERENCES

- Tomas Jech. 1997. "Set Theory." Springer –Verlag
Alonzo Church. 1956. "Introduction to Mathematical Logic. " Princeton University Press.
R.G.G. Cattell and Douglas K.Barry. 1999. "The Object Data Standard: ODMG 3.0. " Morgan Kaufmann Publishers.

- Guy Pierra.1997. "Description methodology: Methodology for structuring parts families" ISO (reference is available from www.plib.ensma.fr)
Guy Pierra. 2001. "Logical resource: Logical model of supplier library. " ISO (reference is available from www.plib.ensma.fr)
Wolfgang Wilkes and Jens Broking, University of Hagen. 2000. "MERC: Integration of EXPRESS based and XML based component information representation. " In *PDT Days 2000* , 127-134 (reference is available from <http://www.merci-project.com/Merci-Portal/MERCISystem/index.htm>)
Hiroshi Murayama. 2000. "InterLIB; an integrated database framework for effective collaboration of Parts Library and EPISTLE Class Library. " In *PDT Days 2000* ,141-146.

AUTHOR BIBLIOGRAPHY

YUMIKO MIZOGUCHI-SHIMOGORI was born in Tokyo. After graduating from Chiba University she joined once Toshiba Corporation, and then decided to take a leave to stay in US for about a year and half. Now she rejoined Toshiba and is working as a research engineer at the Corporate Research and Development Center. Her major interests lie in the field of advanced database design.

HIROSHI MURAYAMA is currently a senior research scientist at the Corporate Research and Development Center of Toshiba Corporation. After several years' work at Nuclear Engineering Laboratory after graduating from Nagoya university, he went to Ecole Nationale Supérieure des Mines de St. Etienne to convert himself to be an informaticien. Now he leads the PLIB research team of Toshiba.

NORIKO MINAMINO joined the Toshiba Corporation having finished master course of mathematics at Osaka University. Now she works at the Corporate Research and Development Center. Currently she enjoys drafting PLIB standards as well as cool draft beer.