ELSEVIER

Information and Software Technology 42 (2000) 347–356

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

$\mathscr{LOD}^*$: A C++ extension for OODBMSs with orthogonal persistence to class hierarchies ☆

E.-S. Cho[a,*], H.-J. Kim[b]

[a]*ROPAS, Department of Computer Science, KAIST, 373-1 Kusong-dong Yusong-gu, Taejon 305-701, South Korea*
[b]*Department of Computer Engineering, Seoul National University, Shilim-dong 56-1, Gwanak-gu, Seoul 151-742, South Korea*

Received 2 March 1998; received in revised form 30 September 1999; accepted 15 October 1999

Abstract

There exist some preprocessing based language extensions for database management where persistence is orthogonal to the class hierarchy. They allow a class hierarchy to be built from both database classes and non-database classes together. Such a property is important in that classes can be reused in implementing database classes, and vice versa. In this paper, we elaborate on the orthogonality of persistence to class-hierarchies, and find that the existing method to achieve this is not satisfactory because of the side-effects of the heterogeneosity of the links in a class hierarchy; some links represent subset(IsA) relationships between database classes, while the others denote inheritance for code-reuse. Finally, we propose $\mathscr{LOD}^*$, a C++ extension to database access, which separates the different categories of links into independent hierarchies, and supports orthogonal persistence to the class hierarchy, overcoming the limitations in the previous methods.

Keywords: Object-oriented database programming language; C++; Class interface; Class implementation; IsA relationships; Code-reuse

1. Introduction

Most object-oriented database systems (OODBMS) regard programming languages as the main interfaces [19]. Now, it is popular to integrate OODBMSs and general-purpose languages like C++, Smalltalk and Java. In this paper, we call the general-purpose languages the 'base languages'.

There are two widely used approaches to the integration of database management and programming languages. In the first approach, the (class) library for database management is provided for the base-language programmers. For example, the ODMG-93 object model defines such library interfaces with C++, Smalltalk and Java [8] with the persistent root class 'd_Object'. In the second approach [20,21], the syntax and/or the semantics of the base language are extended to support database management facilities. Their programs are usually preprocessed into low-level function calls in the pure base languages. One of the advantages of the latter over the former approach is that a class hierarchy can be constructed with database classes and non-database classes mixed, which enables reusing useful codes of non-database classes in implementing database classes, and vice versa [20]. That is, persistence is orthogonal to class-hierarchies.

In this paper, we elaborate on the orthogonality of persistence to class-hierarchies. We find that the existing method to achieve this is not satisfactory because of the side-effects of the heterogeneosity of the links in a class hierarchy; some links represent subset (IsA) relationships between database classes, while the others denote inheritance for code-reuse between classes. This paper proposes '$\mathscr{LOD}^*$',[1] a C++ extension to database access, supporting orthogonal persistence to the class-hierarchy without the limitations in the existing method, by separating the different categories of links.

The sequence of the paper is as follows. The next section introduces the main ideas and language features of $\mathscr{LOD}^*$ with some examples. Sections 3–5 elaborate on our implementation; Section 3 focuses on the preprocessing phase, and Sections 4 and 5 briefly mention the type system and

☆ This work is supported in part by BK 21 IT Grant, "Research on Object-Oriented Database Programming Language Based on C++ and the Standard Object Model".

* Corresponding author.

E-mail addresses: eschough@ropas.kaist.ac.kr (E.-S. Cho), hjk@oopsla.snu.ac.kr (H.-J. Kim).

[1] ([lo:dsta:r]) is built on top of SOP[2] which is an ODMG-93 based OODBMS developed in Seoul National University in 1993–1996.

0950-5849/00/$ - see front matter
PII: S0950-5849(99)00094-4

others, respectively. Discussions and related works are covered in Section 6. This paper is concluded in Section 7. Throughout this paper, we assume that readers are acquainted with the ODMG C++ binding [8].

2. Overview of $\mathscr{LODS}^*$

2.1. Motivations

As mentioned in the previous section, there already exists the mechanism of constructing a class hierarchy with database classes and non-database classes together. For example, supposing 'D_1' and 'D_2' are database classes, the non-database class N can be one of the subclasses of D_1 and/or one of the super classes of D_2.

This enables two different kinds of links between classes to coexist in a class hierarchy. One is for the subset (IsA) relationships between extents of database classes, which denotes database schema, usually viewed from more than one program sharing the database. The other represents inheritance for code-reuse between classes, which is usually related to the implementation details and likely to be hidden from programs other than the one the class is being implemented in. In the above example, both the relationships D_1–N and N–D_2 are for reuse. On the other hand, when D_0 is a super class of both D_1 and D_2, the relationships D_0–D_1 and D_0–D_2 are subset links.

Unfortunately, this heterogeneosity of the links in a class hierarchy entails the following limitations:

- The programmers have to take much care in defining links of class hierarchies, not to cause side-effects from other kinds of links. In the above example, there would be the link D_1–D_2, which was not originally intended if the implementation of classes was not considered.
- Additional facilities for filtering the IsA hierarchy from the entire hierarchy are needed to encapsulate what depends on implementation. In the above example, we should provide the view of $\{D_0, D_1, D_1\}$ for the users of query languages and schema designers who do not care about the implementation details.
- Changes in non-database classes are discouraged since they may cause database schema evolution.

In the above example, when we decide not to reuse N in the implementation of D_2 any more, this entails the modification of entire class hierarchies.

2.2. Basic characteristics of $\mathscr{LODS}^*$

In this section, we introduce the main idea of $\mathscr{LODS}^*$.

2.2.1. Separate modules for a database class

There are two different aspects of a database class. The 'interface' of a class is related to the semantics of the database, which makes classes form in the subset (IsA) relationships. The 'implementation' of a class is the whole description for implementing the class, which is responsible for the inheritance for code-reuse.

This motivates two separate modules for a database class—one for the interface and one for the implementation. The idea of such separation is not new in data sharing environments, especially in multi-language environments and/or distributed database environments, where the interface is usually referred to by more than one program, while class implementation is hidden from users other than the one implementing the class. In the previous example, for each of D_0, D_1 and D_2, an interface and implementation are given.

2.2.2. Distinct class hierarchies

A $\mathscr{LODS}^*$ program deals with two distinct class hierarchies for interfaces of database classes and for implementations of database classes and non-database classes. The interface hierarchies are based on subset relationships between extents of related schema classes, which are viewed by all of the users including query language users and schema designers. The hierarchies for the implementations of database classes and non-database classes are based on code-reuse and are independent from the interface hierarchies. In the above example, one hierarchy is $D_0^{\text{interf}} - \{D_1^{\text{interf}}, D_2^{\text{interf}}\}$, while the other is $D_0^{\text{impl}} - D_1^{\text{impl}} - N - D_2^{\text{impl}}$.

Note that implementations of database classes can become super/sub-classes of non-database classes, while interfaces cannot. This is because only the implementation aspect of a database class requires relationships with non-database classes.

Independence between interfaces and implementation frees users from considering one kind of link in defining other kinds of link.

Although the separate interface hierarchy would look similar to a distinct database class hierarchy from the specific root class 'd_Object' in the database library, it is absolutely independent from the implementation of the database-classes, which eliminates the needs for interactions with non-database classes. Thus, although the hierarchies with the implementations of database classes and non-database classes are constructed together, users can still enjoy orthogonality of the persistence to the class-hierarchy.

2.2.3. More than one implementation for a class

In $\mathscr{LODS}^*$, one interface and more than one implementation is allowed for a database class. It is easy to see that multiple implementations for a class is necessary for the independence between two kinds of hierarchies.

In addition, the schema evolution cost can be degenerated in some cases. We will discuss this more in a later section.

2.3. Examples

2.3.1. Definition of interfaces of persistent classes

In most preprocessing based DBPLs [1,20,21], a database

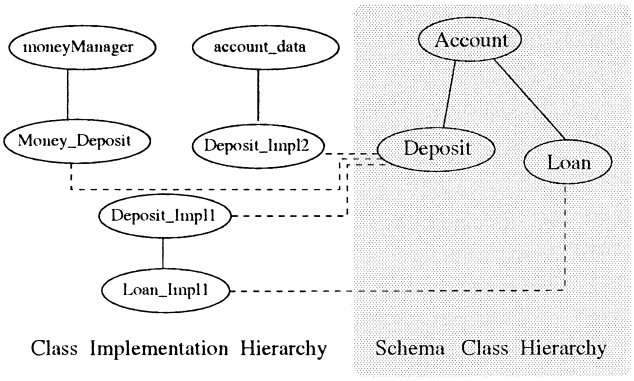

Fig. 1. Separated class hierarchies in the example.

class is usually defined with the new keyword 'persistent', as follows:[2]

```
// Schema definition in most C++ based DBPLs
  persistent class Deposit {
    private:
      money amount; time issue_date;
    public:
      number account_number;
      money show_amount(); time show_date();
      void put_money(money); ….
  };
```

However, in $\mathscr{LOD}^*$, two more object constructors other than the *class* are introduced for class-separation—'*interface*' and '*implementation*'. An '*interface*' is an object constructor consisting of public members and member function signatures. An '*implementation*' is the implementation part of a schema class that includes internal members/member functions and member function implementations. When a schema class is separated into an interface I and an implementation M, it is said that M *implements* I, and objects created by M can also be considered objects of I.

Classes with the keyword 'persistent' represent only interfaces in $\mathscr{LOD}^*$. For example, the persistent class 'Deposit' consists of only its public part, as follows:

```
persistent class Deposit {
  number account_number;
  money show_amount(); time show_date();
  void put_money(); ….
};
```

Otherwise, we can also define 'Deposit' as a subclass of another persistent class, as follows:

```
persistent class Account {
  number account_number;
  money show_amount(); time show_date(); ….
};
```

[2] It is assumed that basic types like number, money and time are already defined somewhere.

```
persistent class Deposit: Account {… void put_money(); …};
persistent class Loan: Account {…(void borrow_money();…};
```

Not only can these schema interfaces be defined in $\mathscr{LOD}^*$, but also in ODLs (Object Definition Language) [7].

2.3.2. Definition of implementations and bindings to interfaces

With both the public and private parts of a class, an implementation has the same declaration syntax as a non-database class, except for its additional keyword 'implements', which binds it to an interface. For example, the implementation 'Deposit_Impl1' in the next example implements the interface 'Deposit'.

```
class Deposit_Impl1 { // implementation
  implements Deposit;
  …. // same as declaration in Deposit(can be omitted)
  money amount; time issue_date;
  …. // method implementation …
};
```

Implementations have to be declared and defined only in programming language interfaces, like $\mathscr{LOD}^*$. In the declaration of an implementation, the data/methods declared in the corresponding interface can be omitted or rephrased. If rephrased, they are checked to see whether the user-defined binding destroys the type safety. If a binding is not destructive, we call it an '*acceptable binding*' [10].

The hierarchy of interfaces and that of implementations in a program are independent of each other. The interface hierarchy is based on subset relationships and related to modeling a schema, while the implementation hierarchy is constructed from reuse relationships. For example, the implementation 'Money_Deposit', which implements the interface 'Deposit', is placed in a hierarchy that is independent of the hierarchy of 'Deposit' and 'Account'. Assume that the class 'moneyManager' is an existing class for managing the 'money' type data.

```
class Money_Deposit: moneyManager { implements Deposit; … };
```

Implementations of the interface 'Deposit' can be reused for other interfaces. The following examples show how 'Deposit_Imp1' is used to implement the interface 'Loan'. The resulting class hierarchy is shown in Fig. 1.

```
class Loan_Impl1: Deposit_Impl1{ implements Loan; … };
```

Since interfaces and implementations are not restricted to map on a one-to-one basis, a schema interface can have instances created by different implementations. For example, assuming that 'account_data' is a system-defined

record type for an account, the interface 'Deposit' can have more than one implementation, simultaneously:

```
class Money_Deposit: moneyManager{ implements
Deposit; …//same as above definition }
class Deposit_Impl1{ implements Deposit; …// same
as above definition };
class Deposit_Impl2: account_data { implements
Deposit; … };
```

2.3.3. Usage of objects with handlers

As in conventional language extensions [1,4], a database object is created through an implementation and handled by an 'object handler(a persistent pointer)'. However, note that in $\mathscr{L}\mathscr{O}\mathscr{D}^*$, the type of the object handler is always the pointer to an interface type. For example, if the interface 'Deposit' is implemented by both 'Deposit_Impl1' and 'Deposit_Impl2', the instances can be created through 'Deposit_Impl1' or 'Deposit_Impl2', and handled by an object handler 'x' which is a pointer to the 'Deposit' type.

```
persistent Deposit * x = new (obase) Deposit_Impl1;
…; x = new (obase) Deposit_Impl2;
x → put_money(1000);
```

By the object handler 'x', all members/member functions described in 'Deposit' can be accessed, whether the actual implementation is 'Deposit_Impl1' or 'Deposit_Impl2'. This allows a new style of polymorphism [6]. Such assignment statements are allowed only if 'Deposit_Impl1' and 'Deposit_Impl2' implement the interface 'Deposit'. Here, 'obase' represents an object for the *objectbase* (a database in an OODBMS) [7].

3. Implementation details

In this section, we present our implementation which translates $\mathscr{L}\mathscr{O}\mathscr{D}^*$ codes into the expressions in ODMG C++ binding.

The C++ object structures, deeply reliant on language implementation for efficiency, makes it hard to design an extension of the language [13]. Since this also makes it difficult to support database and class-separation, we have decided to impose some restrictions on C++. First, $\mathscr{L}\mathscr{O}\mathscr{D}^*$ deals with only global interfaces and implementations, i.e. only those defined outside all blocks. Second, protected members/member functions are not considered. Third, no function overloading is allowed. Fourth, we exclude operator overloading. Fifth, we allow neither the array types of user defined types for members, nor the class templates for interfaces/implementations.

3.1. System-defined classes

For each declaration of an interface and implementation, some definitions of C++ classes are generated. As in ODMG C++ binding, where all database classes are defined as subclasses of 'Persistent_Object', implementations are translated to be the subclasses of 'Imp1Object', a system-defined class derived from Persistent_Object. The definition of Imp1Object is as follows:

```
struct Imp1Object: public Persistent_Object{
  Fptrtb1Class * my_class_tb1;
  Imp1Object(_lod_ClassId class_id)
  {my_class_tb1 = &ftb1[class_id];}…
};
```

With a pointer to a member function table named 'my_class_tb1', we determine at run time which function to execute. 'ftb1' is the list of member function tables constructed from the declarations of interfaces and implementations. A 'class_id' is a class identifier assigned to each class during the preprocessing step. ftb1 is searched with the class identifier as a key.

Schema interfaces are defined as subclasses of 'InterObject', another system-defined class derived from 'Ref⟨Imp1Object⟩'. The definition of InterObject is as follows.

```
struct InterObject: public Ref⟨Imp1Object⟩ {
  InterObject (POID t = 0): Ref⟨Imp1Object⟩(t){ }
  InterObject (Imp1Object * t): Ref⟨Imp1Object⟩(t){ }
  InterObject (const RefAny & t): Ref⟨Imp1Object⟩(t){ }
  InterObject (const Ref⟨Imp1Object⟩ & t): Ref⟨Imp1Object⟩(t){ }
…
};
```

3.2. Translation

3.2.1. Interfaces

For each member originally defined in an interface, two member functions prefixed with 'get_' and 'set_' are generated, and every member access is translated into the get_/set_ member function call. For each get_/set_ member function, the preprocessor defines a new global function, and also generates in every get_/set_ function body the code for the global function call. The member functions in the original interfaces remain after the translation. The body of such a member function is converted to the corresponding member function call of the implementation.

For example, consider a simple schema interface 'SCollection' and its super class 'SCollectionSuper'.

```
interface SCollectionSuper { void insert(int x);};
interface SCollection: SCollectionSuper {char *
name; };
```

After translation, they are redefined as a subclass of the class InterObject with some new member functions.

```
class SCollectionSuper: virtual public InterObject {
  public:
    void insert(int x){
```

```
    void (*f) (const RefAny&, int);
    f = (void (*) (const RefAny&, int))
    (*operator → () → my_class_tb1)
    [FTN_insert_ID] → ptr;
    f(* this, x);
  }
  public:
    SCollectionSuper(POID t = 0): InterObject(t) {}
    SCollectionSuper(Imp1Object * t): InterObject(t)
    {}
    SCollectionSuper(const Ref ⟨Imp1Object⟩ & t):
    InterObject(t) {}
    SCollectionSuper(const RefAny & t): InterOb-
    ject(t) {}
    SCollectionSuper& operator = (const Ref⟨Im-
    p1Object⟩ & t) {
    SCollectionSuper * nthis = this; *nthis = t;
    return *this;}
    …
  };
  class SCollection: public SCollectionSuper
  {
  public:
    void set_name(char n[]){
    void (*f) (const RefAny&, const char []);
    f = (void (*) (const RefAny&, const char []))
  (*operator → () → my_class_tb1)
  [FTN_set_name_ID] → ptr;
  f(*this,n);
  }
    const char * get_name() {
      char * (*f) (const RefAny&);
      f = (char * (*) (const RefAny&))
      (*operator → () → my_class_tb1)
      [FTN_get_name_ID] → ptr;
      return f(*this);
    }
  public:
    SCollection(POID t = 0): InterObject(t) {}
    SCollection(Imp1Object * t): InterObject(t) {}
    SCollection(const Ref⟨Imp1Object⟩ & t): Inter-
    Object(t) {}
    SCollection(const RefAny & t): InterObject(t) {}
    SCollection&operator = (constRef⟨Imp1Object⟩
    & t) {
    SCollection * nthis = this; (* nthis) = t; return
    * this;}
    …
};
```

As seen above, an interface without super classes is translated into a direct subclass of InterObject. For the body of a member function like 'insert()', the preprocessor generates code for the call of the corresponding global function through the corresponding function pointer in my_class_tb1. For each member, get_/set_ functions are generated, with their bodies filled with the corresponding global function calls.

The constants prefixed with 'FTN_' are the identifiers for member function names, and are used as indices for the member functions in each my_class_table. Such identifiers should be carefully selected, to avoid the conflict of indices in the case of inheritance. For example, if we assign the integer 1 to the insert of SCollectionSuper, it should be ensured that the identifier for insert remains 1 in all subclasses of SCollectionSuper, in order for its call through the my_class_table entry to work properly. It would be more sophisticated in the case of multiple inheritance.

In $\mathscr{LOD}^*$, during the preprocessing step, the entire interface hierarchy is scanned, and a unique 'color' (identifier) assigned to each member function name in the hierarchy. These colors are found by the graph coloring algorithm [12]. We use colors as indices for the member functions in each my_class_table. The member function identifiers for the above example are defined by our preprocessor, as follows:

```
enum_lod_FtnId {FTN__ERROR_ID = −1,
FTN_insert_ID = 0, FTN_set_name_ID, FTN_get_-
name_ID};
```

3.2.2. Implementations

The declaration of an implementation is made containing the copies of the members and the member functions of its corresponding interface, unless they are rephrased in the declaration of the implementation. For example, we could define the implementations 'BSetSuper', 'BSet' and 'BSetSub' as follows:

```
class BSetSuper
  {public: char name[10]; void insert(int x) {…..} };
class BSet: public BSetSuper
  {implements SCollection; int number_of(){…} };
class BSetSub: public BSet
  {public: void insert(int x) {…..} };
…..
```

which would be translated as

```
class BSetSuper: virtual public Imp1Object{
  public:
    char name[10];
    void insert(int x){ … }
    BSetSuper():      Imp1Object(CLASS_BSetSu-
    per_ID){ }
    … };
class BSet: public BSetSuper{
  int number_of () {…}
public:
  BSet(): Imp1Object(CLASS_BSet_ID){ }
… };
class BSetSub: public BSet{
  public:
```

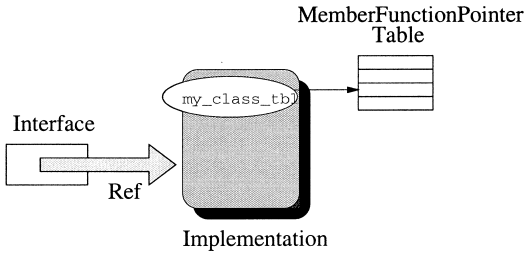

Fig. 2. An object with class-separation.

```
void insert(int x){ …..}
BSetSub(): Imp1Object(CLASS_BSetSub_ID){
}
… };
```

The preprocessor discriminates implementations from non-database classes through syntactical differences, such as the keyword 'implements', since implementations, unlike non-database classes, should be registered into the database. Classes such as BSetSuper and BSetSub are also made subclasses of Imp1Object, as long as they are linked with implementation BSet in the inheritance hierarchy.

The constants prefixed with 'CLASS_' are the identifiers for the implementations. They are used to search ftb1 in the constructor of the Imp1Object, in order to get appropriate my_class_tables. The identifiers for the example implementations defined by the $\mathscr{LOD}^*$ preprocessor are as follows:

```
enum_lod_ClassId { CLASS_ERROR_ID = −1,
  CLASS_BSetSuper_ID = 0,    CLASS_BSet_ID,
  CLASS_BSetSub_ID};
```

3.2.3. Definition and registration of global functions

During the translation, some global functions are generated in order to connect corresponding interfaces and implementations. They contain the call of the corresponding member function of the implementation. For any pair of interface and its implementation, one global function is needed for each member function, and two for each member.

The preprocessor also generates the code for registering those global functions into ftb1.

Fig. 2 depicts a $\mathscr{LOD}^*$ object structure of our implementation. Note that, after translation, an actual database object becomes the implementation object, while its interface has the same layout as that of the ODMG Ref handler [7], but with the more difference in its functionality.

3.2.4. Expressions

As mentioned earlier, a database object is created by the 'new' expression and handled by a persistent pointer. Let us consider the following example which involves creating and handling persistent objects:

```
SCollection * sp1 = new (obase) BSet;
sp1 → insert(3);
sp1 → name = "MySet1";
cout ⟪ sp1 → get_name() ⟪ end1;
SCollectionSuper *sp2 = new (obase) BSetSub;
sp2 → insert(5);
```

The preprocessor translates this code segment into;

```
SCollection sp1 = new (obase) BSet;
sp1.insert (3);
sp1.set_name ("MySet1");
cout ⟪ sp1.get_name() ⟪ end1;
SCollectionSuper sp2 = new (obase) BSetSub;
sp2.insert(5);
```

Note that, to avoid multiple indirections, an interface pointer is not translated into a smart pointer of an interface, but instead into an interface object itself, since an interface object is already a smart pointer.

3.2.5. Cost analysis

The memory required for member function tables is $\sum_{i=1}^{I} M_i \times (number\_of\_member\_functions\_of\_interface_i + 2 \times number\_of\_members\_of\_interface_i)$ words, where I is the total number of interfaces, and M_i is the number of implementations implementing the $interface_i$, since a member function table entry is one word per table entry. An interface requires no more words than a normal smart pointer of Ref. An implementation object needs one more word than an object of Persistent_Object, for the member function table. Thus, the total space overhead for class-separation comes to:

$$Space\_Overhead = \sum_{i=1}^{I} M_i \times (number\_of\_member\_functions\_of\_interface_i + 2 \times number\_of\_members\_of\_interface_i) + number\_of\_implementation\_objects \text{ words.}$$

The overhead for calling a member function through an interface object is roughly the same as the time for one member function call and one global function call. The overhead for accessing a member through an interface object is roughly the same as the time for two member function calls (for the get_/set_ function) and one global function call. Note that no extra time is required for object creation.

The time needed for building up the member function tables and for registering system-defined global functions depends on the total number of members and member functions of interfaces, that is, $\sum_{i=1}^{I} M_i \times (number\_of\_member\_functions\_of\_interface_i + 2 \times number\_of\_members\_of\_interface_i)$.

The overhead for assigning colors is in proportion to $\sum_{i=1}^{I} M_i \times (number\_of\_member\_functions\_of\_interface_i)$,

and the space for the color table in preprocessing depends on $\sum_{i=1}^{I} M_i \times (number\_of\_member\_functions\_of\_interface_i)$. However, they do not affect the execution time, but increases only the preprocessing time.

4. Type checking

In pure C++ programs, an object of 'Set⟨Bicycle⟩' cannot be assigned to the variable of 'Set⟨Person⟩'. However, in ODMG C++ binding [7], objects of Ref can appear at any position expecting RefAny, even when the type parameters of the template class Ref are different. For example, an object of 'Ref⟨Bicycle⟩' can be used in contexts expecting 'Ref⟨Person⟩'. Accordingly, it is preprocessor's responsibility to type check to make the smart pointer type Ref behave like a real C++ pointer type. At the same time, it means that the preprocessor is allowed to have its own type system for persistent pointer types.

Type checking for class-separation support is categorized into four cases, as follows:

1. *Subtyping relationship checking between two interfaces*: to see if objects of one interface can be used in contexts expecting the other interface.
2. *Subtyping relationship checking between two implementations*: to see if objects of one implementation can be used in contexts expecting the other implementation.
3. *Acceptability checking*: to see if the user-defined binding of an interface and an implementation affects type safety.
4. '*Implementing*' *relationship checking between an interface and an implementation*: to see if objects of the implementation can be used in contexts expecting the interface.

Note that an *implementing* relationship does not simply mean an acceptable binding, because an implementation not explicitly bound to an interface may implement the interface [10]. The acceptability checking is made at the implementation declaration; the subtype relationship and implementation relationship are checked at each expression.[3]

While type checking, the preprocessor builds and refers to a table which contains the description of the classes being preprocessed and the information from the schema manager. Details on the type checking in $\mathscr{LODA}*$ are found in Ref. [10].

5. Others, besides preprocessing

The schema manager maintains a flag for each class to determine if the class is an interface or an implementation. The schema class generated by an on-line importing tool is always considered as an interface. As for an implementation, it also keeps the information on the interface it is bound to.

An extent for a schema class is collected by traversing all its implementations, as well as its subclasses in the interface hierarchy. In application language interfaces such as $\mathscr{LOD}^*$ and ODMG C++ binding, however, it is hard to clearly define extents because of their transient instances and uncommitted object creation. Thus, containers like 'Set's and 'Bag's usually replace extents and collect the instances explicitly.

The preprocessing steps are summarized in Fig. 3.

6. Related works and discussions

6.1. The ODMG 2.0 model

The ODMG object model [8] supports the separate implementation for a class, in a different way from ours. First of all, unlike $\mathscr{LOD}^*$, the ODMG C++ binding allows only one implementation for a database class in a program. That is, to a C++ programmer, the implementation looks identical to its class, while another language is used for the schema design in order to define the separate interface of a class. Thus in a C++ program, all the database classes are required to be subclasses of the root class 'd_Object', and persistence is not orthogonal to the class-hierarchy.

The ODMG object model also supports distinct hierarchies for 'interfaces' and 'classes', but unlike the interfaces and implementations in $\mathscr{LOD}^*$, its separation is aimed at the database modeling power, an 'interface' describes abstract behaviors without its own extent, and a 'class' specifies the abstract behaviors and states of its objects, which are collectively called 'types',[4] and viewed from multiple programs sharing a database. Neither the 'interfaces' nor 'classes' is concerned in implementing database classes.

Fig. 4 depicts how the ODMG 2.0 object model supports separate implementation from classes, compared with how $\mathscr{LOD}^*$ does so. The dotted lines represent the binding relationships between interfaces and implementations. In the ODMG model, only one-to-one mapping is allowed, without independent hierarchies for implementations.

6.2. Schema evolution

Due to the independence of interface hierarchies and implementation hierarchies, the schema evolution cost can be degenerated, as mentioned earlier. For example, let us assume that there is an interface of a persistent class 'Deposit' which has three ways of implementation named 'Deposit_Impl1', 'Deposit_Impl2' and 'Deposit_Impl3'.

[3] Actually, checking subtyping between interfaces can be put in the charge of the C++ compiler instead of the preprocessor, since the subtyping is based on C++ inheritance [10].

[4] There is one more constructor called 'structure' [8] which describes abstract states, but we omit it here since it is not so related to this paper.

Without interface/implementation separation, such multiple implementations have to be made subclasses of 'Deposit' in a class hierarchy. This can be represented in a C++-like syntax as follows:

```
// a schema class
persistent class Deposit { … };
// various ways of implementing Deposit
persistent class Deposit_Impl1: virtual Deposit { … };
persistent class Deposit_Impl2: virtual Deposit { … };
persistent class Deposit_Impl3: virtual Deposit { … };
```

Now, if a new class 'SpecialDeposit' is created as a subclass of 'Deposit' in the schema, it should be inherited from all three classes:

```
// a subclass of Deposit
persistent class SpecialDeposit:
Deposit_Impl1, Deposit_Impl2, Deposit_Impl3 { … };
```

Otherwise, the way of implementation of the SpecialDeposit is required to be decided in the schema designing phase.

```
// we decide to make it a subclass of Deposit_Impl1
persistent class SpecialDeposit: Deposit_Impl1 { …
};
```

In the former case, ambiguities [13] may arise if any pair of those three subclasses happen to have common names of attributes/methods, which is not rare, and users have to override them in the class 'SpecialDeposit'. In the latter case, schema designers have to consider the implementation of the class, and in addition, it is hard to add other implementations of the SpecialDeposit to the class hierarchy later.

In our approach, such a schema class can be added in a simpler and more elegant way. Since a hierarchy for implementations can be built regardless of the interface hierarchy, 'DepositImpl1', 'DepositImpl2' and 'DepositImpl3 in the above example do not have to be subclasses of 'Deposit' any more. Instead, they are bound to 'Deposit':

```
class DepositImpl1{implements Deposit;…};
class DepositImpl2{implements Deposit;…};
class DepositImpl3{implements Deposit;…};
```

Since the changes in the implementation hierarchy do not affect the interface hierarchy and vice versa, the new interface 'SpecialDeposit' can inherit from 'Deposit' directly, without consideration of the implementations of Deposit:

```
persistent class SpecialDeposit: Deposit{…};
```

Also, when a user wants to add/delete/modify the private attributes/methods in the implementation of the class 'Deposit' or 'SpecialDeposit', she/he does not have to change the whole class declaration or application programs concerned. Instead, he/she is supposed to change only the very implementation, or add a new implementation of the interface and uses it from then on. Hence, our approach of degeneration of schema evolution cost has similar advantage to that of the schema versioning [22].

6.3. Implementation

6.3.1. Java

Like $\mathscr{L}\mathscr{O}\mathscr{D}^*$, Java supports the independent hierarchies of interfaces from those of implementations. However, ODMG Java-binding [8] does not take advantage of the semantics in database access.

In Java [24], an object is represented by a pointer to a 'handle pool entry' which has two pointers again: a pointer to instance data and a pointer to class data.

Although this is similar to the object layout of $\mathscr{L}\mathscr{O}\mathscr{D}^*$, Java uses naive pointers to the object. On the other hands, pointers to database objects in $\mathscr{L}\mathscr{O}\mathscr{D}*$ are translated into 'InterObject' objects, a special kind of 'd_Ref' objects in ODMG C++ binding. This enables $\mathscr{L}\mathscr{O}\mathscr{D}^*$ programs to share the object management facilities of the ODMG C++-binding, for the portability of standardization. Each object handlers in $\mathscr{L}\mathscr{O}\mathscr{D}^*$ contains the information of the interface that the pointed object belongs to. This is another difference from a Java, where the information of the interface can be obtained by indirect retrieval via the information of a class that is linked from the object.

6.3.2. ODMG

Currently, we have developed $\mathscr{L}\mathscr{O}\mathscr{D}^*$ on top of the previous version 'SOP[2]' that supports the ODMG model release 1.2. However, even if it will be built on the system of the ODMG 2.0 release, the way of the implementation of $\mathscr{L}\mathscr{O}\mathscr{D}^*$ will not have remarkable changes, since we cannot make use of any new features of the new release; the separate implementation support in the new ODMG release are not helpful for $\mathscr{L}\mathscr{O}\mathscr{D}^*$ implementation, because it mismatches the model of $\mathscr{L}\mathscr{O}\mathscr{D}^*$.

For example, it is possible to translate $\mathscr{L}\mathscr{O}\mathscr{D}^*$ interfaces into the separate 'interfaces' in the ODMG 2.0 model, to realize the independent interface hierarchy in an easier way. However, because of the absence of extents for the 'interfaces' in the ODMG 2.0 model, it entails complications in the management of extents for the $\mathscr{L}\mathscr{O}\mathscr{D}^*$ interfaces. Thus, as in the current translation of $\mathscr{L}\mathscr{O}\mathscr{D}^*$, we had better translate $\mathscr{L}\mathscr{O}\mathscr{D}^*$ interfaces into the 'classes' of the ODMG model, for automatic and efficient management of extents. However, there will be some changes of the class names 'Object' and 'Ref' of the old release into 'd_Object' and 'd_Ref', respectively.

6.3.3. CORBA

CORBA [11] supports the separation of the interface and implementation of classes, and there have been developed two main approaches to binding an implementation of a class to its separate interface.

In the CORBA TIE approach [15], which uses C++ macros for binding the implementation and its interface

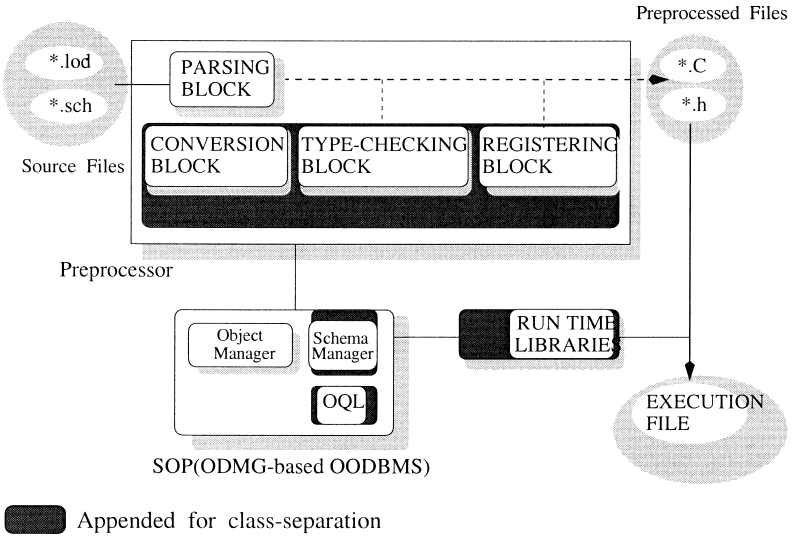

Fig. 3. $\mathscr{LOD}^*$ preprocessing steps.

together, a hidden TIE object is automatically created for each object [15], which holds a reference to its target object and delegates all function invocations to it. However, a simple integration of this mechanism and the ODMG C++ binding will cause multiple indirections with both the d_Ref object and the TIE object involved in sequence. Such indirections are eliminated in our mechanism by merging the interface pointer with the smart pointer Ref, and transferring the functionalities of TIE to those of the implementation objects themselves.

In the CORBA BOAImpl approach [15], an implementation is translated to a C++ derived class of the interface of its class. Although this mechanism is simple and does not cause any indirections, the interfaces which are related by inheritance require that their implementations reflect the same inheritance hierarchy, and the implementation classes have to use the inheritance [15]. This side-effect increases the complexity of the class hierarchies [9].

Some CORBA-compliant systems support OODBMS-persistence of CORBA objects [16,17,23] by extending CORBA loaders [15] for persistent storage. However, they focus neither on supporting persistence orthogonal to the class hierarchy nor on integrating database classes and non-database classes in a proper way, since the purpose of those systems is only to improve CORBA loaders.

6.3.4. Interface-separation in GNU C++

The facility for separate interface is currently incorporated in the GNU C++ compiler [5]. In this mechanism, an interface pointer is made to be the object that contains two pointers again—a pointer to the implementation object and a pointer to the 'member function table object'.

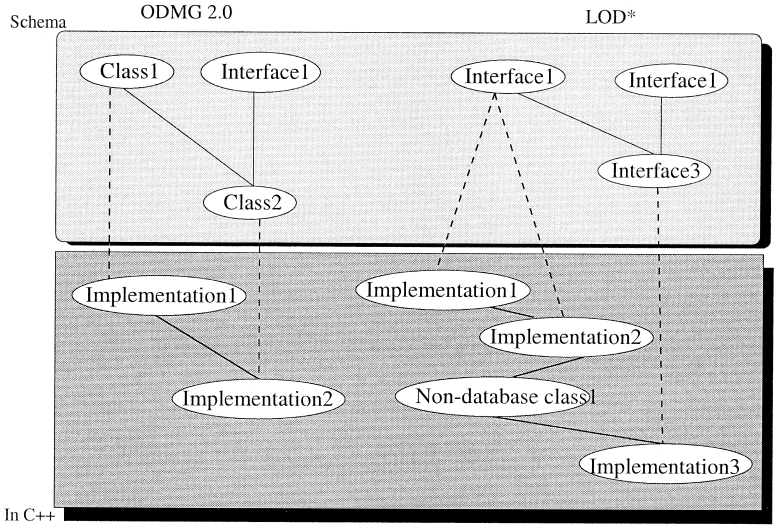

Fig. 4. the ODMG 2.0 object model and $\mathscr{LOD}^*$ model.

However, there are two big drawbacks. First, this table is initialized with the actual function pointers of the implementation, which requires casting the member functions of the implementation to those of the table object, but this is prohibited in C++ [13]. They modify the front end of the GNU C++ compiler [5], with the loss of portability. Second, a 'member function pointer table' for an interface pointer is dynamically generated at every assignment of the assigned implementation object to the interface pointer. Note that, in our implementation, placing member function tables in implementation objects instead of interface pointers, management of the table is not required during the assignment.

In most other cases, except for CORBA [11], member access is not supported directly [3,5]. In $\mathscr{LOD}^*$, members are accessed through get/set functions like in CORBA [11]. Studies on reducing the overhead of member access through get_/set_ function calls are in Refs. [14,18].

7. Conclusions

This paper proposes $\mathscr{LOD}^*$, a C++ extension for database access. $\mathscr{LOD}^*$ allows implementing database classes by reusing non-database classes and vice versa, with the persistence support orthogonal to the class hierarchy. The main idea is to have distinct hierarchies for database classes for subset-based hierarchies and code-reuse hierarchies, based on separate modules of interface and implementation for a database class. Only interfaces are viewed to the programs accessing the shared database, with implementation details of classes hidden from users other than those related to implementing the classes. In the case that the implementation-dependent part of a class needs to be changed, users are supposed to simply add a new implementation for the class without care of the existing instances, which decreases schema evolution cost.

$\mathscr{LOD}^*$ uses the C++ pointer interface, from which the preprocessor generates ODMG C++ binding code. After translation, an actual database object is represented as an implementation object, and an interface object behaves like a smart pointer. Unlike in general purpose languages, the implementation introduced in this paper meets the various requirements for the database applications. Since there is not much reference material on the related implementation, we hope this paper will be helpful to future implementers.

Currently, we are expanding the $\mathscr{LOD}^*$ preprocessor to eliminate the restrictions described in this paper. We consider it another interesting issue to apply our idea to the language extensions like the Java interface for databases which already support the independent interfaces and implementations for non-database classes.

References

[1] R. Agrawal, N.H. Gehani, Rationale for the design of persistency and query processing facilities in the database programming language O++, Second International Workshop on Database Programming Languages, Portland OR, June 1989.
[2] J.H. Ahn, H.J. Kim, Seof: An adaptable object prefetch policy for object-oriented database systems, Proc. of the Conf. on Data Engineering, 1997.
[3] K. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, Reading, MA, 1996.
[4] T. Atwood, Two approaches to adding persistence to C++, The Fourth International Workshop on Persistent Object Systems, 1991, pp. 369–383.
[5] G. Baumgartner, V.F. Russo. Signatures: A C++ extension for type abstraction and subtype polymorphism, Technical Report CSD-TR-93-059, Purdue University, September 1993.
[6] A.P. Black, N. Hutchinson, Typechecking polymorphism in Emerald, Technical Report, DEC and UCB, 1991.
[7] R.G.G. Cattell, Object Database Standard: ODMG-93 release 1.2. OMG Group, 1996.
[8] R.G.G. Cattell, D.K. Barry, Object Database Standard: ODMG-93 release 2.0, Morgan Kaufmann, Las Altos, CA, 1997.
[9] E.S. Cho, S.Y. Han, H.-J. Kim, A semantics of the separation of interface and implementation in C++, Proc. of International Conference on COMPSAC, 1996.
[10] E.S. Cho, H.-J. Kim, Class-separation mechanism for integrating OODBMSs and general-purpose OOPLs, submitted for publication.
[11] DEC, HP, HyperDesk, NCR, Object Design, and SunSoft, The Common Object Request Broker Architecture and Specification, OMG group, February 1997.
[12] R. Dixon, T. McKee, P. Schweizer, M. Vaughan, A fast method dispatcher for compiled languages with multiple inheritance, Proc. of the ACM OOPSLA Conf., 1989, pp. 211–214.
[13] M.A. Ellis, B. Stroustrup (Eds.), The Annotated C++ Addison-Wesley, Reading, MA, 1990.
[14] A.L. Hosking, J.E.B. Moss, Compiler support for persistent programming, Technical Report, University of Massachusetts, March 1991.
[15] IONA Technologies Ltd. Orbix: distributed object technology—Programmer's Guide (Release 1.3), July 1995.
[16] IONA Technologies Ltd. Orbix + ObjectStore Adapter, April 1996.
[17] J. Kleindienst, F. Plasil, P. Tuma, Lessons learned from implementing the CORBA persistent object service, Proc. of the ACM OOPSLA Conf., 1996.
[18] B. Liskov, A. Adya, M. Castro, Q. Zondervan, Type-safe heterogeneous sharing can be fast, Technical Report, MIT, 1996.
[19] M.E.S. Loomis, Object Databases, Addison Wesley, Reading, MA, 1995.
[20] Object Design, Inc. ObjectStore Release 4.0 Online Documentation, March 1995.
[21] Objectivity Inc. Objectivity/DB: Getting Started with C++, March 1994.
[22] R. Ramakrishnan, D.J. Ram, Modeling design versions, Proc. of the Conf. on VLDB, 1996.
[23] F. Reverbel, Persistent in distributed object systems: ORB/ODBMS integration, PhD thesis, University of New Mexico, 1996.
[24] B. Venners, Inside of Java Virtual Machine, Java Masters Series, McGraw Hill, New York, 1997.