AN APPROACH TO INTRODUCE THE REFLECTION TO C++

Chen Ping Cai Xiyao Jin Yimin

Department of Computer Science
Xidian University
Xian, Shaanxi 710071, China

ABSTRACT

The reflective architecture supports the incremental development of object-oriented systems, but mostly for the interpretation-based languages. This paper describes our work to introduce this architecture to a popular compilation-based language, C++, without to modify the compiler. Our reflective architecture makes a disciplined split to the object level and the meta-object level in a class-based form. The unit of causal connections is the class member function. Mechanisms based on the methods diversion are constructed to support a kind of implicitly causal connections. A prototype has been constructed in the Advantage C++ 1.1M4 running on MSDOS.

1. Introduction

It has been shown that the class-instance model of the object-oriented programming languages (OOPL's) leads to more modular and better structured programs. However, it requires a proper categorization of the problem domain, and this is not always available. Incremental development supports, therefore, are needed for the classes design of a system, but may be concerned some new features not initially present in the popular OOPL's, e.g., C++[13].

On the concept of knowledge abstraction[1], it needs to split the domain knowledge and programming knowledge into different parts of a program to design easy understood and extensible systems. The above stated categorization is also required here. Though it done, to keep the separation will still be difficult during practical developments. For example, although the tracing and testing operations do not belong to the category of domain knowledge, they may still be defined in the classes and/or methods about the domain for the usages of developments and extensions.

The computational reflection is a helpful concept to solve the above problem [11]. It has been shown that this concept allows for the definition of new features not initially present in an OOPL in a reflective architecture, a computational system is viewed as incorporating an object part and a reflective (meta-object) part. The task of the former is to solve problems and return information about the problem domain, while the task of the later is to solve problems and return information about the object computation. Some OOPL's have introduced the reflective architecture [11][14]. However, this model defined in 3-KRS [11] does not support the class-instance model.

J. Farber [8] has presented an approach to introduce this model in the class-based OOPL's that use message passing as their basic control structure, but these languages are not compilation-based.

We are carrying on an object management system for a software engineering environment [3]. The computational reflection model is chosen for incremental development of new features of the system, and the C++ language for the efficiency and portability. The first step towards the target is to introduce the computational reflection into C++.

Our system makes a disciplined split to the object level and the meta-object level similar to the S-KRS but in a class-based form. A meta-object can also have its own meta-object(s) just as a general object. The unit of causal connections is the class member function. Mechanisms based on the methods diversion are constructed to support a kind of implicitly causal connections between those two levels. A prototype of the system has been constructed in the Advantage C++ 1.1M4 running on MSDOS. It has presented that our reflective architecture results simpler implementations for constructing new mechanisms, without to modify the C++ compiler.

This paper presents a part of our experience. In section 2, we shall describe the structural features of our reflective architecture. The next 3 sections will present the reflective mechanisms of the architecture. Then, in section 6 a simple example is given. Finally, the conclusions and the further plan are discussed and revealed in section 7.

2. The Reflective Architecture

The proposed reflective architecture is conceptively composed of a structural part and an operational part. The structural part provides a number of built-in classes and constraints to extend the class-instance model of C++ to be reflective. The operational part,
Different from the concept of structural reflection [6] [7][8], our structural extension of class-instance model is to represent the reflective relationship between objects and their meta-objects through definitions of their classes. This extension is based on the following three constraints upon the user-defined class hierarchy.

C1: The root of a class hierarchy must be a class named _top.  
C2: The meta-object class of a class c must be defined as a subclass of c. Both classes should have a constructor taking no arguments. 
C3: If a member function m of a class c (called method) will be designed as a causal connection form, m must be defined as a virtual member function of c.

The class _top is provided by the system and defined as follows:

```cpp
class _top {
public:
    class winf;
    virtual void emplx();
    _top( ) { infx = 0; }
};
```

where the class winf is designed to represent the reflective information about a class and its instances, and the virtual function emplx is just used to form a regular head for all objects. Its usages will be described in section 5.

Constrained by C1 and C2, a user-defined reflective class hierarchy will have a form of conceptual structure as shown in Fig. 1.

![Diagram of class hierarchy](image)

Fig. 1. The conceptual structure of a reflective class hierarchy.

In Fig. 1, classes c_1, c_2 and c_11 are those about a certain domain, i.e., their instances belong to the object level, called C. The classes M0_c_1, M0_c_2 and M0_c_11 are defined as the meta-object classes of c_1, c_2 and c_11, respectively. The instances of these classes will belong to meta-object level, M0. The class M1_c_1 is the meta-object class of M0_c_1. Its instances will belong to another meta-object level, M1.

3. Methods Diversion

On the structural features stated above, the reflective mechanisms can be constructed in C++ itself. The methods diversion is the basic one among these mechanisms. It is designed to dynamically exchange or reexchange the elements of a pair of vtbls of a class and its meta-object class, where vtbl is the abbreviation of the jump table constructed by the C++ compiler for virtual functions. Two C++ features are used to ensure the correctness of the diversion:

F1: Any invocation to a virtual function of an object without using the scope resolution operator :: will be an indirect function call.

F2: The interface of a redefined virtual function must not be changed.

The vtbl of a class is composed of the entries of the virtual functions, which are defined or redefined in this class, and an end mark 0. For example, two vtbls concerned with a class c_1 and its subclass M0_c_1 may have following forms represented in standard C++:

```cpp
int(* c_1_vtbl[x]) = {
    (int(*) ) c_1_method1,
    (int(*) ) c_1_method2, 0;
}

int(* M0_c_1_vtbl[x]) = {
    (int(*) ) M0_c_1_method1,
    (int(*) ) M0_c_1_method2,
    (int(*) ) M0_c_1_method3, 0;
}
```

Here suppose the virtual function method2 is redefined in class M0_c_1 but the method1 not. In the vtbl of M0_c_1, the first element is copied from the vtbl of c_1, the second is the entry of method2 redefined in M0_c_1, and the third corresponds to a new defined method3 in M0_c_1.

The elements of a vtbl can be changed using another vtbl dynamically if the feature F2 is kept. Since it may need to recover them later, our strategy is to exchange other than to modify them. After the diversion, the vtbls in the above example will be changed into:

```cpp
int(* c_1_vtbl[x]) = {
    (int(*) ) c_1_method1,
    (int(*) ) c_1_method2, 0;
}

int(* M0_c_1_vtbl[x]) = {
    (int(*) ) M0_c_1_method1,
    (int(*) ) c_1_method2,
    (int(*) ) M0_c_1_method3, 0;
}
```

It can be seen that any invoking to the method2 of c_1 will be implicitly diverted to the method2 of M0_c_1, on the feature F1. The second element of the vtbl of M0_c_1, which stores the original value of the vtbl of c_1, is just for later recovery.

Four macros concerned with the methods diversion are provided:

- `DIVERSION(c,mp)`: to construct the class information objects about class c and its meta-object class
AC, and then exchange the elements of their vtbls.

- RECOVER(c): to reexchange the elements of vtbls of class c and its meta-object class, if they have been diverted.

- DIVERI(c): to divert vtbls of class c and its meta-object class, if they have been recovered, and

- RECOVERALL: to recover all of pairs of the diverted vtbls.

A problem in the implementation is how to refer to a certain vtbl, because it is named by the compiler. Our solutions are as follows:

- The DIVERSION should be used in a separately compiled file, and in that file the vtbl names will be declared as external variables by the macro.

- A vtbl name is formed by the corresponding class name using the same naming rule as the compiler.

- The compiler switches KO and KE1 should be used to ensure each class has only one version of its vtbl and allows for external references, like [5].

4. Meta-methods Sharing

There are no inheritance relationships between the meta-object classes whose instances belong to the same meta-object level (we call that these classes are in the same level). For example, come back to consider Fig.1. There is no inheritance between classes MD_c_1 and MD_c_11 which are in the same meta-object level MD. Since they correspond to the class c_1 and its subclass c_11, it is naturally required to make MD_c_11 inherit the redefined methods of MD_c_1 when needed.

On the structural constraints C2 and C3 as well as the C++ feature F2, the above requirement can be satisfied in our system. Consider the example again. We can affirm that any redefined method in MD_c_1, say m, must be one inherited from c_1. The m of c_1 must also be inherited by c_11 and MD_c_11, respectively. So we can affirm that any method redefined in MD_c_1 must have a corresponding one in MD_c_11 with the same interface. Such methods redefined in meta-object classes are referred to as meta-methods.

The meta-methods sharing is a mechanism designed to support this kind of inheritance. There are two ways to construct it. One is through the multiple inheritance, the other is by the delegation [12][2]. The later has been chosen in our system, partly for that the Advantage C++ 1.1.4 does not support the multiple inheritance, partly for that the delegation could have more usages in the system besides the meta-methods sharing.

The delegational rule of the meta-methods sharing in MD is: for a class c and its virtual function m, where c is in OL and m is redefined in neither c nor the meta-object class of c, say MC. m will be diverted to a corresponding meta-method in the meta-object class of a class c, say MC, if c is the nearest superclass of c so that m is also a member of c and is redefined in either c or MC. Fig.2 gives an example.

To restrict the rule in the MD is on the consideration of implementation. It can be shown that, recall the rule, the script of m in the vtbl of MC must be equal to that of m in the vtbl of MC, if it is not greater than the size of vtbl of c . This property can be used to determine the shared meta-method simply on the relationships between these scripts, but only in MD.

The paths of meta-methods sharing are fixed through a depth-first search algorithm during the diversion. These paths, then, can be directly used to share meta-methods until recovered. The details of the implementation of the mechanism will not be dealt with here due to the limitation of space.

5. The Handling of this

In C++ programs: a pointer to the object for which a member function is invoked constitutes a hidden argument to the function. The implicit argument can be explicitly referred to as this [19]. It is normally handled by the operations designed in the source program. However, after the methods diversion the situations will be different. At that time the invoking a method of an object has been diverted to a meta-method, but the this still points to the object other than to the meta-object. Semantic errors, even some run-time errors, would arise if no additional handling of this were made up.

In our system we have built a mechanism, the handling of this, to ensure that in meta-methods the object and meta-object can be properly pointed by current this at
It consists of a pair of macros as shown as follows:  

```c
#define RBEGIN(type)  
{  
  _top = new type();  cinf = find_cinf(this);  
  if (cinf == 0) cinf = find_cinf(this);  
  this = (type *) cinf->get_ptr(p);  
  if (this == 0) {  
    _top = new type();  cinf->set_ptr(p);  
    this = (type *) cinf;  }  
  else {  
    this = (type *) cinf->get_delegate(t);  
    if (this == 0) {  
      _top = new type();  cinf->set_delegate(p);  
      this = (type *) cinf;  }  
  }
}

#define REND(type) this = (type *) _topthis;  }
```

These two macros must appear in pairs within a meta-method to bracket the reflective operations. The RBEGIN is used to save current this and set it with the address of a proper meta-object. The REND recovers current this with one saved by RBEGIN.

The cinf is critical for the mechanism. Consider an object c which belongs to a class c and is being processed by RBEGIN. Since c.cinf has been initialized to be empty in the constructor of _top, it may not be set until current processing. If so, the function find_cinf is invoked to find an object, c, which stores the reflective information about c. As mentioned in section 2, the definition of _top can ensure that each object has a regular structure of head. So there is a standard way to obtain the address of the vtbl of c by means of the this of c and then using it as a key to find c. Two factors related to the operations of the macro will then be analyzed. If the class name, given by the meta-method through the argument, is as the same as the name of the meta-object class of c, the operations to find the normal meta-object will be executed. If not so, the delegate will be searched. If, in both cases, the corresponding meta-object does not exist, it will be dynamically created with a regular form which is guaranteed by the constraint C2. This strategy is similar to 3-KRS [1]: meta-objects are only constructed when they are actually needed.

Here is a simple example which gives a paradigm of the meta-method design:

```c
int M0_c_1::method2(char *, int *)  
{  
  RBEGIN(M0_c_1)  
  method2();  
  \  
  REND(M0_c_1)  
  return c_1::method2(s, i);  
}
```

In this example, method2 can be designed to do the reflective computations about the object being reflected. The scope resolution operator appeared in the last statement is necessary. If not so, the compiler will treat the statement as a recursive function call. The invoking to method2, on the feature F1, does not use the operator, because this method may also be diverted to a meta-method in the meta-object class of M0_c_1. This example only describes an unconditional reflection. In fact it can also be conditional. One can simply define local variables to control the invoking to c_1::method2, according to the results of reflective computations.

6. An Example

An example of the reflective computation is shown as follows:

```c
#include "manager.hxx"

// The head file of the reflection manager

extern void methods_diversion();  

class employee: public top  
{  
  char * name;  
  short department;  
  public:  
    virtual void change_dept(short d);  
    employee();  
    employee(char * n, short d);  
};

class manager: public employee  
{  
  short level;  
  public:  
    virtual void change_level(short l);  
    manager();  
    manager(char * n, short d, short l);  
};

class Employee: public employee  
{  
  int cd_called;  
  public:  
    void change_dept(short d);  
    Employee();  
    Employee(int cdCalled = 0);  
};

class Manager: public manager  
{  
  int cl_called;  
  public:  
    void change_level(short l);  
    Manager();  
    Manager(int clCalled = 0);  
};

void Employee::change_dept(short d)  
{  
  RBEGIN(Employee)  
  printf("\d th calling change_dept(\d)\n",  
          d, cdCalled);  
  REND(Employee)  
  employee::change_dept(d);  
}

void Manager::change_level(short l)  
{  
  RBEGIN(Manager)  
  printf("\d th calling change_level(\d)\n",  
          l, clCalled);  
  REND(Manager)  
  manager::change_level();  
}

main()  
{  
  employee x("Smith", 1, 1);  
  x.change_dept(2);  
  x.change_level(3);  
}
```
in another file, one should define:

```cpp
void methods_diversion() {
    APPENDINF(employer, manager);
    DIVERSION(employer, Memployee);
    DIVERSION(manager, Memanager);
}
```

where \texttt{APPENDINF} is a macro designed to get and store the hierarchic information about the classes in the CL.

In this example, two classes, \texttt{employer} and \texttt{manager}, and their meta-object classes are defined. Here the reflective computation is a kind of simple tracing operation which counts the number of calling and displays it as well as the name and argument value of the invoked method. At the beginning of the program execution, the methods diversion operations are performed. Then the method invocation with the normal form will be implicitly diverted to the corresponding meta-method. Although the method \texttt{changeDept} is not redefined in \texttt{Memanager}, the practical reflection will be delegated to the meta-method in \texttt{MMemployee} through the meta-methods sharing.

7. Conclusions

Although to introduce the computational reflection into C++ will have more difficulties than into the interpretation-based languages, our work has shown that it is possible to build a reflective architecture within such a language. Our contributions are chiefly in three aspects: 1) introducing a reflective architecture to a popular compilation-based language, 2) using the virtual functions to perform the causal connections without affecting the normal design style of C++ programs, and 3) constructing a delegation mechanism for sharing meta-methods whose classes are not designed as having inheritance.

Recently, we have extended the prototype to support the object identity which is also an object and reflective. And we have made our system reflect itself, i.e., the reflective mechanisms themselves can also be reflectively developed and extended, and support the composite objects and the objects set. In another paper [4] we shall describe this extension.

Our work is still primitive. The further plan includes: 1) to represent and manage much information about objects and classes to increase the power of casual connections, 2) to construct a mechanism to solve the "self-problem" [10] in the meta-methods sharing, 3) to study the theory of reflection and to construct a formal system, and 4) to extend the C++ syntax and semantics when necessary.

REFERENCES

[9] A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983