Macro Processing in Object-Oriented Languages

Shigeru Chiba
Institute of Information Science and Electronics, University of Tsukuba
and PRESTO, Japan Science and Technology Corporation
E-mail: chiba@is.tsukuba.ac.jp

Abstract

There are a number of programmable macro systems such as Lisp's. While they can handle complex program transformation, they still have difficulty in handling some kinds of transformation typical in object-oriented programming. This paper examines this problem and, to address it, presents an advanced macro system based on ideas borrowed from reflection. Unlike other macro systems, our macro system provides metaobjects as the data structure used for the macro processing instead of an abstract syntax tree. This feature makes it easy to implement a range of transformations of object-oriented programs.

1: Introduction

Macros are probably the oldest example of meta programming. They have been used for processing a source program at the meta level, that is, from a point of view other than the computation described in the program. Their *meta* view has been tokens or syntax trees; most macro systems, including the ANSI C preprocessor [1], deal with a source program as a sequence of tokens. Other systems, like Lisp's macro system, deal with it as a parse tree or an abstract syntax tree. These traditional macro systems have been satisfactory since the basic constructs of the target languages are procedures and functions, which are relatively simple.

However, the basic constructs of object-oriented languages are not procedures but objects, classes, and methods, which are semantically and syntactically rich concepts. This fact reveals the limitations of the traditional macro systems. Some kinds of source-level translation that are useful in object-oriented programming are extremely difficult with those macro systems. Such translation requires the ability to deal with a source program at a higher level than the level of tokens and syntax trees. For example, it may need the ability of adding a specific method to a given class if that method is not defined in the given class.

This paper presents our macro system called OpenC++, which addresses the problem above for the C++ language. This macro system was designed on the basis of some ideas borrowed from *reflection* [21, 17] and hence it allows programmers to deal with a source program from a logical viewpoint instead of a syntactic one. OpenC++ exposes the source program as a set of C++ objects representing the logical structure of the program. This logical representation makes it easy to implement macro processing typical in object-oriented programming. This fact is somewhat obvious but the contribution of this paper is that it reports that the logical representation that the reflection community has been studying is also workable for macro processing in object-oriented programming.

Some versions of OpenC++ have been available to the public since 1995 and they have been evolved according to the feedback from dozens of world-wide users (Several papers about research activities using OpenC++ are found in [7]). For example, one group is using OpenC++ to develop a C++ compiler to produce CORBA-compliant fault tolerant objects [12]. OpenC++ has been also used to implement a tool for analyzing a program and producing useful information such as an inheritance graph. The problem discussed in this paper has been articulated during the interaction with those users. This paper focuses on the recent improvement of OpenC++ for addressing that problem and presents details of its new programming interface. This is a new argument not included in the author's previous paper [3], which proposed the conceptual design of OpenC++ in the context of reflection.

This paper has six sections. In Section 2, we show an example of macro processing, which is typical in object-oriented programming but that current macro systems cannot easily handle. Section 3 presents details of OpenC++ and how it handles the example in Section 2. Section 4 shows other examples of the use of OpenC++. We discuss the comparison with related work in Section 5. Section 6 is conclusion.

2: Motivating Example

Programmable macro systems, such as Lisp macros, are powerful tools for meta programming. They enable programmers to implement various language facilities that the target language lacks, such as control statements, data abstractions, and even a simple object system. Macro systems are also useful for executing inline expansion of a simple function to improve runtime performance.

However, the current macro systems are not satisfactory for object-oriented programming. There are several examples that are typical in object-oriented programming but cannot be easily implemented by traditional macros. Since most of the macro systems can only transform function-call expressions, they are not practical if programmers want to transform statements or expressions that do not look like a function-call expression. This section shows one of those examples and mentions that another kind of macro system is needed in object-oriented languages.

"Design patterns" [8] show us a number of programming techniques called *patterns* to solve typical problems we frequently encounter in object-oriented programming. Although those patterns are well designed and tested, strictly following the patterns sometime causes the programmers to repeatedly write similar code. This tedious work should be automated by macros but traditional macro systems cannot handle this. A special preprocessor is needed for the automation.

We illustrate this problem with the visitor pattern. This pattern is used for visiting the elements of a given data structure such as a tree structure and performing different operations on every node. Suppose that we implement an abstract syntax tree for a compiler. The nodes of this tree structure are instances of different classes such as `AssignExpr` and `PrintExpr`. The visitor pattern implements operations on this abstract syntax tree, such as a type-checking operation.

A fundamental idea of the visitor pattern is to implement the type-checking operation separately from the implementation of the node classes. The program should include not only the node classes but also an abstract class `Visitor` and its subclass `TypeCheckVisitor`. These classes define the operations performed during the tree traversal. They have methods such as `VisitAssign-Expr()` and `VisitPrintExpr()`, each of which represents the operation executed when a node of the corresponding class is visited.

To execute the type checking, the method `Accept()` is called with an instance of `Type-CheckVisitor` on the root node of an abstract syntax tree. The visitor pattern requires that all the tree-node classes have this method. For example, `Accept()` of the class `AssignExpr` should be:

```
bool AssignExpr::Accept(Visitor* v) {
    return v->VisitAssignExpr(this);
}
```

This calls the method `VisitAssignExpr()` on a given `Visitor` object, which is an instance of `TypeCheckVisitor` in this case. The called method depends on the class; `Accept()` of the class `PrintExpr` calls `VisitPrintExpr()`. In either case, the invoked method recursively calls `Accept()` on the children of that node so that the child nodes are visited and the type checking is executed on this subtree.

A benefit of this design is that the programmer can define another subclass of `Visitor` and easily implement a tree traverse for a different operation. For example, the compiler would also need `CodeGenerationVisitor`. However, this benefit comes with a relatively complex class structure; all the tree-node classes must have a method `Accept()` and an abstract class `Visitor` needs to be defined. This extra work for taking advantage of the visitor pattern is not difficult, but is tedious and error-prone, since it is boring repetition of simple work, which is to add a method `Accept()` to all the tree-node classes and to define a class `Visitor` so that it has methods corresponding to every tree-node class.

A natural and simple idea for avoiding this problem is to use macros for automating this extra work for the visitor pattern. Ideally, only a simple annotation to the tree-node classes should invoke macro expansions so that the method `Accept()` is added and the abstract class `Visitor` is defined. Suppose that `visitorPattern` is an annotation. The following code:

```
visitorPattern class AssignExpr {
public:
    Expr *left, *right;
         :
};
visitorPattern class PrintExpr {
public:
    Expr* printed_value;
         :
};
```

should be processed by the macro system and translated into something like this:

```
class Visitor {          //  abstract class
public:
    bool VisitAssignExpr(AssignExpr*) = 0;
    bool VisitPrintExpr(PrintExpr*) = 0;
};
class AssignExpr {
public:
    bool Accept(Visitor* v) { return v->VisitAssignExpr(this); }
    Expr *left, *right;
         :
};
class PrintExpr {
public:
    bool Accept(Visitor* v) { return v->VisitPrintExpr(this); }
    Expr* printed_value;
         :
};
```

The macro expansion shown above is, unfortunately, difficult to handle with traditional macro systems. Obviously, the standard C++ macro system cannot handle it since its macros are applied only to a symbol or a function-call expression. Even if the whole class declaration is given to a macro as an argument, the C++ macro system cannot expand the declaration to include the method `Accept()`. This is because the macro arguments are not divided into smaller pieces for performing complex substitution. For example, a macro `m` cannot expand `m(ABC)` into `ABxC` (the macro argument `ABC` cannot be divided into `AB` and `C`).

If a system like Lisp macros is used, macro arguments can be flexibly divided, transformed, and inspected. Lisp's macro system deals with macro arguments as a tree of symbols, which can be manipulated as other regular data structures. Such macro systems have been also developed for languages like C, which has a more complex grammar than Lisp [24, 13, 25, 16]. These macro systems are called programmable syntax macros since their macro arguments are not parse trees but abstract syntax trees (ASTs) so that it is easy to handle the complex grammar. Significant design issues for syntax macros are (1) how to specify places where a macro is expanded in a given AST, and (2) how to divide macro arguments into desirable subtrees. As for (1), A* [13] invokes a macro if a given AST matches a pattern written in BNF. This feature enables macro expansion without a macro name; for example, A* can transform all the `if` statements appearing in a program. As for (2), MS^2 [25] uses a special pattern-match language for parsing macro arguments. In this language, patterns are specified with tokens and the types of ASTs (that is, non-terminal symbols in the grammar).

These syntax macro systems are powerful and general-purpose systems, but they still require their programmers to write complicated patterns for describing a macro for the visitor pattern. Although adding a new method should be a simple operation in object-oriented languages, the syntax macro systems require writing a complex pattern or program to compute where an AST representing the added method is attached to the AST representing a class declaration. Writing a macro for defining an abstract class `Visitor` is also difficult. Although the macro needs to collect the names of all the tree-node classes, no direct supports are provided by the syntax macro systems. Again, the programmer has to write a complex pattern to retrieve the class names. XL [16] provides semantic information obtained by static program analysis in macro functions, but it is a functional language without object orientation. The semantic information provided by XL is limited to the static type of expressions and so on.

3: OpenC++

The first version of our C++ macro system called OpenC++ version 2.0 [3] had the same problem mentioned above. Since we noticed this in response to user requests on OpenC++, we have been examining a solution so that OpenC++ version 2.5 has a complete new interface meeting the users' demands. This section presents the new interface and how it addresses the problem.

3.1: Syntactic Structure vs. Logical Structure

Our observation is that a drawback of the current macro systems is that an abstract syntax tree (AST) is used for macro processing. This is good for languages such as Lisp and C, in which function calls are primary language constructs, but not for object-oriented languages such as C++. Because C++ has a complex grammar, the logical structure of programs is less aligned with the syntactic structure represented by ASTs than it is in procedural programs. C++ macros thus tend to be difficult to write. For example, it is not straightforward to remove a given method from the AST representing a class declaration in C++. To do this, the macro programmer is required to have detailed knowledge about the structure of that AST.

OpenC++ borrows ideas from reflection [21, 17] and thus, instead of ASTs, it provides *metaobjects* for macro processing. The metaobjects are regular C++ objects representing the *meta* view of the processed program in object orientation. As in other reflective systems like the CLOS MOP [11] and ObjVlisp [5], classes and methods (or members in the C++ terminology) appearing in a program are chosen as metaobjects. OpenC++ represents a program by a collection of those metaobjects, which corresponds to a logical structure of that program. Through these metaobjects, macro programs can access the semantic information of the program and even transform the program. For example, the class metaobjects provide methods to get a super class of the class, to get the list of methods of the class, and to add a new method to the class.

While ASTs represent the syntactic structure of a program, metaobjects represent its logical structure. In this logical representation, all the occurrences of language constructs related to each other but spread over a program are collected to form a single aggregate. For example, a class metaobject includes links to all the occurrences of language constructs involved with the class represented by that metaobject. Those language constructs are base classes (i.e. super classes), member functions, data members, expressions for calling a member function on an instance of that class, and so on. Macros can easily access and manipulate those constructs through the links of the metaobject. On the other hand, in the syntactic representation, the occurrences of those language constructs belong to different ASTs if they appear at different locations in the program. For example, an expression for calling a member function on an instance of a class does not belong to the AST representing the declaration of that class. It instead belongs to the AST representing a function body in which that expression appears. Therefore, it is significantly difficult to obtain an aggregate of language constructs involved with each other unless they are adjacently located in a program.

To provide the representation of a logical structure, the OpenC++ compiler first parses a source program into a parse tree and converts it to a collection of metaobjects. Then macros written by programmers, which are called *meta-level programs* in OpenC++, access and modify these metaobjects. The modifications applied to the metaobjects are finally reflected in the source program. The OpenC++ compiler re-converts the metaobjects to a source program according to the modifications and transfers the obtained source program to a regular C++ compiler such as GNU C++. This re-conversion is performed to enable the use of an off-the-shelf compiler for the back end.

3.2: Programming interface of OpenC++

OpenC++ provides two kinds of metaobjects: class metaobjects and member metaobjects (a member means a method or a field). The OpenC++ compiler reads a source program and makes a class metaobject for every class in the source program. The class metaobject contains member metaobjects, each of which represents a member of that class. We below illustrate how these metaobjects are used for macro processing.

3.2.1: Metaobjects

The class metaobjects provide an interface to access the definition of a class from the viewpoint of the logical structure. We below show some of the member functions defined on class metaobjects:

- `Ptree* Name()`
  This returns the name of the class. `Ptree` is the type representing parse trees or symbols.

- `Class* NthBaseClass(int n)`
  This returns the `n`-th base class of the class.
- `bool NthMember(int n, Member& m)`
  This returns `true` if the class has the `n`-th member. The member metaobject for the `n`-th member is returned in `m`.
- `void RemoveBaseClasses()`
  This removes all the base classes from the class.
- `void AppendBaseClass(Class* c, int spec = Public, bool is_virtual = false)`
  This appends a class `c` to the list of the base classes. The arguments `spec` and `is_virtual` specify attributes of the appended base class.
- `void ChangeMember(Member& m)`
  This changes the definition of the member `m`.
- `void RemoveMember(Member& m)`
  This removes the member `m` from the class.
- `void AppendMember(Member& m, int spec = Public)`
  This appends a new member `m` to the class.

The member metaobject returned by `NthMember()` provides an interface to access the definition of a member. The following are some of member functions defined on the member metaobject:

- `Ptree* Name()`
  This returns the name of the member.
- `Ptree* FunctionBody()`
  This returns the function body of the member.
- `void Signature(TypeInfo& t)`
  This returns the signature of the member in `t`. If the member is a data member, the type of that data member is returned.
- `IsPublic()`
  This returns `true` if the member is a public member.
- `void SetName(Ptree* new_name)`
  This changes the name of the member to `new_name`.
- `void SetFunctionBody(Ptree* new_body)`
  This changes the function body of the member to `new_body`. The new function body is given in the form of parse tree.

3.2.2: Macro definition

The class metaobjects are not only passive data structures processed by macros. OpenC++ macros are defined as member functions on class metaobjects. The OpenC++ compiler calls one of these member functions for translating every code fragment, such as a class declaration and an expression for reading a data member. The member function is invoked on a class metaobject representing a class involved with that code fragment. Recall that a class metaobject includes links to related code fragments. Programmers who want macro processing define a new metaclass, that is, a new class for class metaobjects, and override some of these functions.

This framework was originally proposed in OpenC++ version 2.0 [3] and inherited by the new version. However, the class metaobjects in version 2.0 deal with an AST for macro expansion and hence it causes the same problem as other AST-based macro systems. This problem has been addressed in the new version shown here so that the class metaobjects deal with metaobjects.

We below show the member functions for macro processing. All of them are defined on class metaobjects.

Initialization:

The OpenC++ compiler first calls a `static` member function `Initialize()` on every metaclass. It does not translate any code fragment but is used to register a new keyword to the parser so that the compiler accepts extended syntax.

Class declaration:

To expand a macro for a class declaration, `TranslateClass()` is called on every class metaobject. It is responsible for transforming the declaration of the class represented by the class metaobject called on.

Member function:

To expand a macro for the implementation of a member function, `TranslateMemberFunction()` is called on the class metaobject that the member function belongs to. `TranslateMemberFunction()` receives a member metaobject representing the implementation of a member function. It can transform a program through that metaobject.

Expression:

The OpenC++ compiler calls `TranslateMemberCall()` on a class metaobject if it encounters an expression calling a member function on an instance of that class. If the expression is `p->Move(3, 4)` and `p` is an instance of a `Point` class, then the compiler calls `TranslateMemberCall()` on the `Point` class metaobject. The result of this member function is substituted for the original expression `p->Move(3, 4)`. The compiler also calls different member functions if it encounters other kinds of expressions.

The member functions for translating expressions receive an AST and return another AST. This is the same protocol as AST-based macro systems; We chose this since the logical structures of expressions are well aligned with their syntactic structures. However, those member functions are called on a class metaobject and translate only expressions involved with a specific class. Other macro systems do not directly support this class-based macro expansion.

Finalization:

After all the expressions are processed, the OpenC++ compiler calls `FinalizeInstance()` on every class metaobject and `FinalizeClass()` on every metaclass. These member functions are used to append a code fragment at the end of a source program. Since the whole source program has been already processed, these member functions can access all the occurrences of related language constructs and append code fragments depending on them. A user-defined metaclass can override these member functions and append, for example, the declaration of a class inheriting from all the classes included in a source program. This kind of class declaration is extremely difficult to produce by traditional macro systems since only adjacent language constructs are accessible from macros.

3.3: The Macro for the Visitor Pattern

In the rest of this section, we illustrate how the macro for the visitor pattern is written in OpenC++. If using a macro written in OpenC++, programmers can write a program following the visitor pattern as below:

```
metaclass VpClass;

visitorPattern class AssignExpr { ... };
visitorPattern class PrintExpr { ... };
```

The `metaclass` declaration in the first lines informs the OpenC++ compiler that a macro set `VpClass` (visitor pattern class) is used. A keyword `visitorPattern` associates the following class with the metaclass `VpClass` and invokes the macro on that class.

The metaclass `VpClass` overrides three member functions inherited from the default metaclass `Class`:

```
class VpClass : public Class {
public:
    static bool Initialize();
    void TranslateClass(Environment*);
    static Ptree* FinalizeClass();
};
```

The first member function `Initialize()` is called by the OpenC++ compiler during initialization. It registers the keyword `visitorPattern`:

```
static bool VpClass::Initialize() {
    RegisterMetaclass("visitorPattern", "VpClass");
    return true;
}
```

The second member function `TranslateClass()` is a macro function for translating the declaration of a class associated with this metaclass. It adds `Accept()` to the associated class.

```
void VpClass::TranslateClass(Environment* e){
    Ptree* mf = Ptree::Make(
        "public: "
        "bool Accept(Visitor* v){"
        "    return v->Visit%p(this); }", Name());
    AppendMember(mf);
}
```

This member function first constructs a parse tree representing the member function `Accept()` and then appends it to the class. `Ptree::Make()` is a function to construct a parse tree from a character string. The occurrence of `%p` is replaced with the resulting value of `Name()`, which is the name of the class. `AppendMember()` appends to the class a member represented by either a member metaobject or a parse tree.

The last member function `FinalizeClass()` is called at the end of compilation. It produces the definition of an abstract class `Visitor`:

```
Ptree* VpClass::FinalizeClass(){
    ClassArray classes;
    int n = InstancesOf("VpClass", classes);
    // mems  is the list of members
    Ptree* mems = nil;
```

```
    for(int i = 0; i < n; ++i){
        Ptree* name = classes[i]->Name();
        Ptree* m = Ptree::Make("bool Visit%p(%p*) = 0;",
                               name, name);
        //   concatenate the constructed member to the rest
        mems = Ptree::Cons(m, mems);
    }
    ofstream* file = new ofstream("visitor.h");
    file << Ptree::Make("class Visitor {"
                        "public: %p };", mems);
    return nil;
}
```

This member function first calls `InstancesOf()` and collects all the classes associated with this metaclass. Then, it constructs a parse tree representing an abstract class `Visitor` and writes the constructed parse tree to a file `visitor.h`. Note that the implementation of `VpClass` assumes that all the tree-node classes are included in a single source file. This means that separate compilation is not possible if `VpClass` is used. To avoid this problem, the implementation of `VpClass` can be improved to record processed tree-node classes in a file and produce `Visitor` by referring to that file at the end of separate compilation. OpenC++ provides mechanisms supporting this improved implementation, but details of that are beyond the scope of this paper.

To perform the macro expansion by `VpClass`, the OpenC++ compiler first compiles the meta-level program describing the metaclass `VpClass` and then a source program including `AssignExpr` and `PrintExpr`. If it encounters the `metaclass` declaration in the source program, it dynamically loads the compiled `VpClass` and performs the macro expansion according to the definition of `VpClass`. As we showed in Section 2, this macro processing produces the definition of an abstract class `Visitor` and appends a member function `Accept()` to all the tree-node classes.

4: Other examples

The power of OpenC++ for macro processing enables a number of useful class libraries and rapid implementation of extended C++ languages for distribution, transactions, persistence, and others. However, those realistic examples are complex and long. In this section, we instead present examples that are simple but useful to illustrate the functionality of OpenC++.

4.1: Wrapper

Making *wrappers* is a programming technique frequently used for adding extended functionality such as distribution and persistence to objects. It is also similar to the decorator pattern [8]. We below show how OpenC++ helps programmers make wrappers.

Although there are several variations of this technique, the basic idea is illustrated by the following source-code translation. Suppose that we have a class `Rectangle`:

```
class Rectangle {
public:
    void Stretch(int, int);
private:
    int ulx, uly, lrx, lry;
};
void Rectangle::Stretch(int dx, int dy){
    lrx += dx; lry += dy;
}
```

Making a wrapper means to rename `Stretch()` to `_Stretch()` and add another version of `Stretch()` to the class `Rectangle`:

```
class Rectangle {
public:
    void Stretch(int dx, int dy) { ++counter; _Stretch(dx, dy); }
    void _Stretch(int, int);
private:
    int ulx, uly, lrx, lry;
};
void Rectangle::_Stretch(int dx, int dy){
    lrx += dx; lry += dy;
}
```

The added member function `Stretch()` is a wrapper. It performs computation necessary for extended functionality (in the case above, to increment `counter` for counting the number of invocations of this function) and then calls the original function, which has been renamed to `_Stretch()`.

This translation, while conceptually simple, is complicated in an AST-based (abstract syntax tree based) macro system. The difficulties are due to the complexity of the grammar of C++. The first difficulty is to retrieve the name of a member function. Since the type system of C++ allows a member function to have a complex signature such as:

```
int (*Stretch(int, int))()
```

(This means that `Stretch()` returns a pointer to a function), it is not straightforward to write a general pattern that can retrieve the function name in all cases. Another difficulty is to retrieve the argument names, `dx` and `dy`. Since an argument name is optional in a declaration, it must be implicitly filled if it is not specified by the programmer. In the example above, since the original declaration of `Stretch()` does not include argument names, appropriate names must be chosen for constructing the declaration of new `Stretch()`.

The metaobjects of OpenC++ simplify these tedious tasks. The metaclass `WrapperClass` for performing the translation shown above is written as follows:

```
class WrapperClass : public Class {
public:
    void TranslateClass(Environment*);
    void TranslateMemberFunction(Environment*, Member&);
private:
    Ptree* Rename(Ptree* name);
    void MakeWrapper(Member& member, Ptree* org_name);
};
```

First, `TranslateClass()` performs renaming of member functions and adding wrapper functions:

```
void WrapperClass::TranslateClass(Environment* env)
{
    Member member;
    int i = 0;
    while(NthMember(i++, member))
        if(member.IsPublic() && member.IsFunction()){
            //  execute the following code for every member function
            Member wrapper = member;
            //  rename the original member
            Ptree* new_name = Rename(member.Name());
```

```
            member.SetName(new_name);
            ChangeMember(member);
            //   add a wrapper function
            MakeWrapper(wrapper, new_name);
            AppendMember(wrapper, Class::Public);
        }
}
```

The function `Rename()` in the above code is also defined in `WrapperClass`. It returns a parse tree representing a new name such as `_Stretch()`. Note that the original member name is obtained by simply calling `member.Name()`.

The function `MakeWrapper()` makes a member metaobject for the wrapper from the metaobject for the original member function:

```
void WrapperClass::MakeWrapper(Member& member, Ptree* org_name)
    Ptree* body = Ptree::Make("{ ++counter; %p(%p); }",
                              org_name, member.Arguments());
    member.SetFunctionBody(body);
}
```

Note that the actual arguments for calling the original function is computed by `member.Arguments()`. This function inspects a member declaration and fills missing argument names if they are not specified. The programmer does not need to be concerned that formal argument names are optional in C++.

Finally, `WrapperClass` overrides the member function `TranslateMemberFunction()` to rename a member name appearing in the implementation of a member function:

```
void WrapperClass::TranslateMemberFunction(Environment* e,
                                            Member& mf)
{
    if(mf.IsPublic())
        mf.SetName(Rename(member.Name()));
}
```

This is called on the implementation of every member function. `mf` is a member metaobject representing that implementation. `SetName()` gives a new name to the member function specified by `mf` and the new name is reflected in the source program.

4.2: Simple Protocol Checker

OpenC++ can be used to produce a warning message specialized in a class library. Although class libraries often have a protocol that the programs using the class library must follow, regular C++ does not provide the ability to produce warning messages if the programs are not following the protocol. OpenC++ enables such warning messages. In fact, OpenC++ itself uses this ability for examining whether a meta-level program is correctly written.

Suppose that a class library includes a class `DialogBox` and its subclasses must not override a `virtual` function `move()` defined in `DialogBox`. To examine whether this protocol is followed by user's subclasses, the class library can include a metaclass `ProtocolCheckClass` associated with `DialogBox`. The metaclass `ProtocolCheckClass` overrides a member function `TranslateClass()` to include the following code:

```
Member m;
if(!Name()->Eq("DialogBox") && LookupMember("move", m)
                                        && m.IsVirtual())
```

```
    WarningMessage("move() is not overridable.");
```

This code produces an warning message if a class except `DialogBox` defines a `virtual` function `move()`. Since subclasses of `DialogBox` inherit the metaclass, the code above is executed on all the subclasses and it examines whether they follow the protocol.

Also, suppose that the class `DialogBox` has provided a `virtual` function `quit()` but this member function is obsolete in a new version of the library. The metaclass `ProtocolCheck-Class` can be extended to produce a warning message if a subclass of `DialogBox` falsely overrides `quit()` of `DialogBox`, which does not exist any more. Regular C++ compilers cannot produce such a warning message because they recognize that `quit()` is not overridden but newly defined in the subclass.

5: Related Work

Although OpenC++ is a reflective system, its reflectiveness is for meta programming at compile time. Most of other reflective systems are for meta programming at runtime [23, 20, 19, 18, 22, 15, 6], and thus OpenC++ should be called a system based on compile-time reflection or, in other words, a macro system the programming interface of which is borrowed from reflection. As for runtime reflection, the usefulness of metaclasses for customizing a class structure has been studied by Classtalk [2, 14]. Although Classtalk is a variant of Smalltalk with runtime reflection but not a macro system, they also claimed that metaclasses were a good framework for implementing customization of a class, for example, adding a method, changing a super class, and so on. They also mentioned that the ability of such customization improved the design of class libraries.

OpenC++ version 1.0 and 2.0 were presented in different articles [4, 3]. Version 1.0 provided the ability of not compile-time reflection but runtime reflection. The difference between version 2.0 and version 2.5 presented in this paper is that the data structure used for macro processing in 2.0 is not a metaobject but an AST. Version 2.0 hence involves the same problem as other macro systems.

This problem is also found in other systems based on compile-time reflection [10, 9]. For example, MPC++ [10] executes macro processing with ASTs. Although MPC++ uses metaobjects, their metaobjects are the nodes of an AST but they do not enable logical accesses to a program. Hence MPC++ still involves a problem due to using ASTs although it provides programming supports for dealing with a complex AST.

6: Conclusion

This paper presented OpenC++, which is a C++ macro system based on a meta-level architecture borrowed from reflection. Unlike traditional macro systems, OpenC++ provides metaobjects as the data structure processed by macros. The OpenC++ macros, which are called meta-level programs, deal with those metaobjects to translate a source program. Since the metaobjects enable accesses to the logical structure of the program, they make it easier to implement typical macro processing in C++ than with other AST-based (abstract syntax tree based) macro systems. This is significant in object-oriented programming, in which the logical structure of programs are not directly reflected in the syntactic structure.

OpenC++ does not provide a general-purpose solution for complex macro processing. In some cases, other macro systems such as A* [13] and MS^2 [25] would be more appropriate. However, we believe that OpenC++ is effective for macro processing typical in object-oriented programming because the metaobjects enable logical accesses to a source program. As we showed in examples, this ability is what typical macro processing requires in object-oriented programming but A* or MS^2 do not provide since they use ASTs and a pattern-match language.

OpenC++ presented in this paper is freely available from the web site:

`http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html`

The distribution package includes source files, documentation, and sample programs.

Acknowledgment

OpenC++ was initially developed when the author was staying at Xerox Palo Alto Research Center. I would like to thank Gregor Kiczales and John Lamping for their helpful comments. I would also thank all the OpenC++ users, who gave me code, bug reports, and valuable feedback.

References

[1] American National Standards Institute, Inc. *ANSI Standard on C*, 1990. ANSI X3.159-1989.

[2] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in smalltalk-80. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 419–431. ACM, October 1989.

[3] S. Chiba. A metaobject protocol for c++. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, number 10 in SIGPLAN Notices vol. 30, pages 285–299. ACM, 1995.

[4] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pages 482–501. Springer-Verlag, 1993.

[5] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 156–167, 1987.

[6] J. C. Fabre and T. Pérennou. A metaobject architecture for fault tolerant distributed systems: The friends approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[7] Jean-Charles Fabre and Shigeru Chiba, editors. *Proc. of Workshop on Reflective Programming in C++ and Java*, UTCCP Report 98-4. Center for Computational Physics, University of Tsukuba, Japan, 1998. (held at ACM OOPSLA'98).

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[9] Yuuji Ichisugi and Yves Roudier. Extensible java preprocessor kit and tiny data-parallel java. In *Proc. of ISCOPE '97*, number 1343 in LNCS, 1997.

[10] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in c++ — MPC++ approach —. In *Proc. of Reflection 96*, pages 153–166, Apr. 1996.

[11] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[12] M. O. Killijian, J. C. Fabre, J. C. Ruiz-Garcia, and S. Chiba. A metaobject protocol for fault-tolerant corba applications. In *Proc. of the 17th IEEE Symp. on Reliable Distributed Systems (SRDS '98)*, pages 127–134, 1998. (Also available as Research Report 98139, LAAS, France).

[13] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Trans. on Software Engineering*, 21(11):894–901, 1995.

[14] Thomas Ledoux and Pierre Cointe. Explicit metaclass as a tool for improving the design of class libraries. In *Proc. of the 2nd Int'l Symp. on Object Technologies for Advanced Software (ISOTAS)*, LNCS 1049, pages 38–55. Springer, Mar. 1996.

[15] C. P. Lunau. A reflective architecture for process control applications. In *ECOOP'97 — Object-Oriented Programming*, volume 1241, pages 170–189. Spriner, 1997.

[16] William Maddox. Semantically-sensitive macroprocessing. Master's thesis (ucb/csd 89/545), University of California, Berkeley, 1989.

[17] P. Maes. Concepts and experiments in computational reflection. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155, 1987.

[18] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 127–144, 1992.

[19] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proc. of European Conf. on Object-Oriented Programming '91*, number 512 in LNCS, pages 231–250. Springer-Verlag, 1991.

[20] A. Paepcke. PCLOS: Stress testing CLOS Experiencing the metaobject protocol. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 194–211, 1990.

[21] B. C. Smith. Reflection and semantics in lisp. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 23–35, 1984.

[22] R. J. Stroud and Z. Wu. Using metaobject protocols to implement atomic data types. In *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, pages 168–189. Springer-Verlag, 1995.

[23] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 306–315, 1988.

[24] B. Wegbreit. *Studies in Extensible Programming Languages*. Outstanding dissertations in the computer sciences. Garland Publishing, Inc., 1980. (based on the author's thesis, Harvard, 1970).

[25] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. of Conf. on Programming Language Design and Implementation*, volume 28, no. 6, pages 156–165. ACM SIGPLAN Notices, 1993.