A Metaobject Protocol for C++

Shigeru Chiba

Xerox PARC & University of Tokyo
chiba@parc.xerox.com chiba@is.s.u-tokyo.ac.jp

Abstract

This paper presents a metaobject protocol (*MOP*) for C++. This MOP was designed to bring the power of meta-programming to C++ programmers. It avoids penalties on runtime performance by adopting a new meta-architecture in which the metaobjects control the compilation of programs instead of being active during program execution. This allows the MOP to be used to implement libraries of efficient, transparent language extensions.

1 Introduction

A metaobject protocol (MOP) is an object-oriented interface for programmers to customize the behavior and implementation of programming languages and other system software. The usefulness of this kind of customizability has been argued elsewhere[11, 9, 10], and interesting MOPs have been included in languages such as Lisp[20], ABCL[21], and, to a lesser degree, Smalltalk[6]. The goal of our work is to bring the power of meta-programming to the more mainstream language C++, while respecting their performance concerns in that community.

This paper proposes a new MOP for C++, called OpenC++ Version 2. Like previous MOPs, it allows programmers to implement customized language extensions such as persistent or distributed objects, or customized compiler optimizations such as inlining of matrix arithmetic. These can be implemented as libraries[1] and then used repeatedly. Unlike previous MOPs, our proposal incurs *zero* runtime speed or space overhead compared to ordinary C++.

To make this possible, our MOP works by providing control over the *compilation* of programs rather than over the runtime environment in which they execute. Specifically, our MOP provides control over the compilation of the following key aspects of C++: class definition, member access, virtual function invocation, and object creation. This feature means that the design of our MOP is *inherently* efficient, as opposed to MOPs, such as the CLOS MOP, where only sophisticated implementation techniques enable efficient execution.

Our MOP has been developed by a synthesis and re-engineering of a number of ideas in this field: we took our basic protocol structure from the CLOS MOP [11], we took the basic structure of a compile-time MOP from Anibus and Intrigue [18, 13], and we took some ideas for the basic structure of a MOP for C++ from the meta-information protocol (MIP) [2] and our previous work on Open C++ Version 1 [4].

This paper is a status report on the development of OpenC++ Version 2. Our MOP has been prototyped in Scheme and a number of examples are running using the prototype. For simplicity, however, we use C++ notation in this paper. In the rest of the paper, we first discuss what we want our

Appeared in OOPSLA'95 Proceedings pp.285–299

[1] We use the term *library* to mean a collection of reusable code such as functions, data types, and constants, written by the language users within the description capability of the language.

MOP to support, then we present the basic architecture of the MOP, and we illustrate its suitability for real-world programming.

2 What We Want to Enable

The motivation for our C++ MOP is to enable programmers to easily write libraries that provide language extensions transparently and efficiently. This section illustrates what we want to enable with the MOP.

Suppose we want to implement persistent objects as a C++ library. In terms of transparency, the goal is to allow the library users to specify that some of their classes should produce persistent objects simply by adding an annotation to ordinary class definitions.

```
persistent class Node {
public:
  Node*  next;
  double value;
};
```

The program that deals with persistent objects should look like:

```
Node* get_next_of_next(Node* p)
{
  Node* q = p->next;
  return q->next;
}
```

The key point is that adding or removing the simple annotation `persistent` should be all that is required to change this from defining persistent objects or transient ones.

Unfortunately, such a simple annotation is quite difficult to implement in C++. For example, one way to try to do this in C++ is to develop a class library that provides a class `PersistentObject` from which other classes can inherit if they want to be persistent. In such a scheme, the hope is that the definition of the class `Node` would look like:

```
class Node : public PersistentObject {
public:
  Node*  next;
  double value;
};
```

But, the inheritance mechanism does not provide enough access to implement persistent objects by itself; the library user will also have to edit their programs to correctly use that functionality. To implement persistence, references to persistent objects must be detected at runtime. If this is not done in hardware, the software will need to be edited to look something like:

```
Node* get_next_of_next(Node* p)
{
  Node* q = (p->Load(), p->next);
  return (q->Load(), q->next);
}
```

Because the class `PersistentObject` cannot control its subclasses' member accesses, the library user will have to call the member function `Load()` before every access to the object. `Load()` is a member function supplied by the class `PersistentObject` to load objects from disk on demand.

The situation may not appear so bad in this simple example, but tracking down and editing all such accesses can be quite difficult. The library implementor must describe the need for editing in a document, and the user must carefully read and follow those instructions. More importantly, changing code back and forth from persistent to transient is extremely labor-intensive.

Our MOP provides the ability to implement a persistent object library so that persistence can be selected with only a simple edit to the class definition. It enables not only annotations for language extensions but also ones for compiler optimizations.

From the pragmatic viewpoint, the design criteria of such a MOP are high performance and arbitrary customizability. For the former, the MOP should not include any runtime overhead or preclude optimization by the current C++ compiler. For the latter, the MOP should provide the ability to implement common C++ extensions such as persistent C++ or distributed C++.

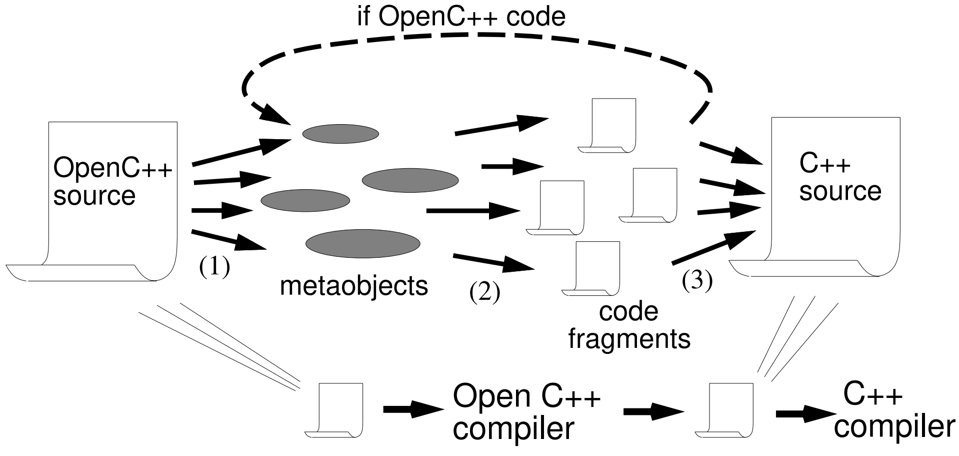

Figure 1: The Protocol Structure

3 The Basic Architecture

The basic architecture of the OpenC++ MOP is similar to that of the CLOS MOP in that metaobjects represent language entities visible to the programmer. There are class metaobjects and function metaobjects. The behavior of the program is controlled by those metaobjects.

3.1 Compile-time and Runtime

A distinguishing feature of the OpenC++ MOP as compared to the CLOS MOP is that the OpenC++ MOP clearly separates the compile-time environment and the runtime environment. Ordinary objects exist only at runtime, and metaobjects exist only at compile-time.[2]

Since the metaobjects exist only at compile-time, the way they alter the behavior of the objects is by controlling the compilation of the program. The metaobjects appropriately translate top-level definitions of the program, and, if necessary, append supplementary runtime functions, types, and data to the translated code.

This means that our MOP inherently implies no penalty in runtime space or speed. On the other hand, in the CLOS MOP, key aspects of the object system are executed through runtime method invocation of the metaobjects. The CLOS MOP hence requires sophisticated implementation tricks to achieve good runtime performance.

3.2 Basic Protocol Structure

The OpenC++ MOP controls source-to-source translation from OpenC++ (extended C++) to C++. First of all, the source code of the OpenC++ program is parsed and divided into top-level definitions for classes and (member-)functions.[3] Then a metaobject is constructed for each such definition. The metaobject then translates the top-level definition into appropriate ordinary C++ (or C) code. The translated code is then collected and assembled into contiguous source code. Figure 1 shows this protocol structure.

To see how this works, we now walk through an example of how the MOP compiles a small program, specifically the two definitions shown below:

```
class Point {
public:
  void MoveTo(int, int);
  int x, y;
};
```

[2] A simple programmer extension that we will show later can allow some of the functionality of runtime metaobjects.

[3] The definition of global variables is also a top-level definition. But in this paper, we ignore it for simplicity.

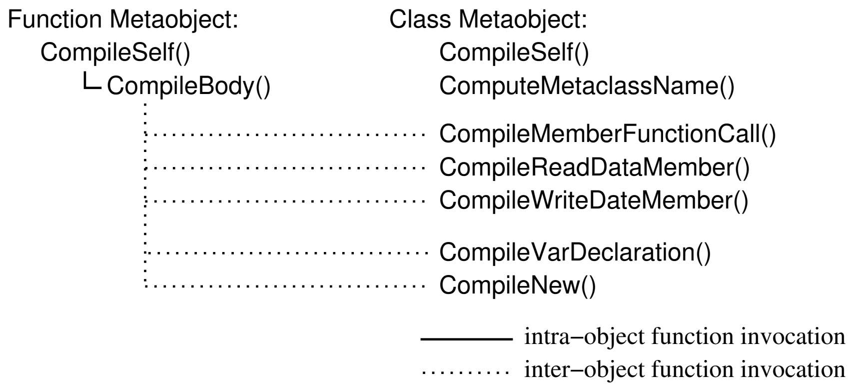

Figure 2: Overview of the Protocol

```
void Point::MoveTo(int new_x,
                   int new_y)
{
  x = new_x; y = new_y;
}
```

In phase 1, after parsing, the MOP constructs two metaobjects, one for the class `Point` and one for the member function `MoveTo()`. By default, the class metaobject is an instance of the class `Class`. It contains information given by the class definition such as its name, base classes, members, etc. The function metaobject is an instance of the class `Function` by default. It contains similar information from the function definition, such as its name, parameters, and the parse tree of the function body. Since this is a member function, it also has a pointer to the class metaobject that supplies this member function.

In phase 2, the metaobjects are called upon to generate appropriate code fragments to substitute for the original code fragments of their definitions. To do that, the member function `CompileSelf()` of each metaobject is invoked.

In response to `CompileSelf()`, class metaobjects generate an ordinary C++ definition — in the form of parse tree — for their class. The default is just to emit the original definition. Similarly, function metaobjects generate an ordinary C++ definition — also in the form of a parse tree. Again, the default is to just emit the original definition.

To translate the function body, function metaobjects follow a layered sub-protocol, walking the parse tree of the body step by step, asking appropriate class metaobjects to translate each fragment of the body. This is done by passing the parse tree of the code fragment to the class metaobject, and getting the parse tree of the translated fragment back. This layered protocol is shown in Figure 2. Note that since the function metaobject compiles the function body by making queries of the class metaobjects, the compilation of the program is mainly the responsibility of the class metaobjects. The default action of the class metaobjects is to just return the given parse tree without any translation.

In the example of `MoveTo()`, since `x` is a data member of the class `Point`, it means that when `x=new_x` is encountered during the tree traversal, the function metaobject invokes the member function `CompileWriteDataMember()` of the class metaobject `Point`. That member function translates the parse tree that corresponds to the expression `x=new_x`, and returns the translated parse tree. By default, the translated expression is also `x=new_x`. The original subtree is then replaced with the returned subtree. Similarly, if the function metaobject encounters a variable declaration such as “`Point* p`”, it invokes the member function `CompileVarDeclaration()` of the class metaobject `Point` to translate the subtree of the variable declaration.

In phase 3, the parse trees generated by the individual metaobjects are then collected and converted into C++ source text. This source text is then compiled by the C++ compiler. Since the MOP is implemented as a separate preprocessor of the C++ compiler, the converted C++ text is in the form of character file. So the C++ compiler must parse the contents of the file again. We could avoid this overhead, however, if we integrated the MOP into the C++ compiler.

A conceptually significant point is that the metaobjects are permitted to generate arbitrary code fragments, so the generated fragments may contain OpenC++ code. If that happens, the translation is recursively applied to each generated fragment until it becomes ordinary C++ code. Phases 1 and 2 are repeated on each fragment so that all code fragments become C++ code before starting phase 3.

3.3 A Simple Programmer Customization

Now we show a simple example of what programmers can do with this MOP. Suppose we want to specialize class metaobjects to implement a mechanism for getting class information such as data member names at runtime. In essence, the idea is to implement a subset of the functionality of the Meta-Information Protocol[2]. At runtime, for every class, this mechanism will automatically and transparently make available a record that contains the class information.

Any C++ customization implemented as a MOP-based library is usually divided into two parts: *compile-time code* and *runtime code*. The former is a class library of metaobjects, and the latter is a set of runtime support routines. Given the two, the programmer can then write a *user program* that simply uses the C++ customization (see Figure 3).

First we show the user program the library user wants to be able to write:

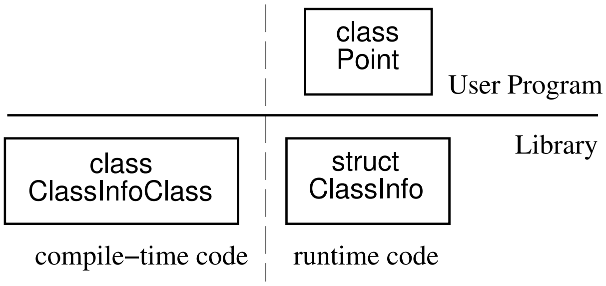

Figure 3: MOP-based library

```
metaclass Point : ClassInfoClass;
class Point { ... };
  ...
int   i = 0;
char* name;
while((name=ClassDataMemberNames(Point)
             [i++]) != NULL)
  printf("Point's data member: %s\n",
         name);
```

This program prints the names of the data members of the class `Point`. The first statement specifies the *metaclass* of the class `Point`. It declares that the class metaobject for `Point` should be an instance of the class `ClassInfoClass`. This annotation directs the MOP to produce runtime class information for the class `Point`. The user can then access it through the runtime functions supplied by the library. In the code above, `ClassDataMemberNames()` is one such runtime access function.

To enable such a user program, the library implementor must write the appropriate compile-time and runtime library, as shown below. First, we show the runtime code. It includes the definition of `ClassInfo`, a record type for class information such as the class name and the data members. It also includes the function `ClassDataMemberNames()` to access `ClassInfo`.[4]

```
struct ClassInfo {
  char*  class_name;
  char** data_member_names;
};
```

[4]That function is implemented as a macro because C++ does not deal with a symbol name as first-class data. `##` is a macro operator for concatenating two symbol names.

```
#define ClassDataMemberNames(name)\
      ((info_##name).data_member_names)
    ...
```

Next, we show the compile-time code. Its objective is to control the compilation so that the runtime support routines work appropriately with the user's program. In this example, compilation of the definition of the class `Point` must translate it into this:

```
class Point { ... };    // not changed
ClassInfo info_Point = {
  "Point",
  {"x", "y", NULL}
};
```

The second line makes the record `info_Point`, which contains the class information of `Point`. To do such translation, we define a subclass of `Class` for the compile-time library.

```
class ClassInfoClass : public Class {
public:
  Expression CompileSelf();
};

Expression
ClassInfoClass::CompileSelf()
{
  Expression code = MakeParseTree(
    "ClassInfo info_%s = {\"%s\", {",
    name, name);

  for(int i = 0;
      i < n_data_members;
      ++i)
    code->Append(
      MakeParseTree(
        "\"%s\",",
        data_member_names[i]));

  code->Append(
    MakeParseTree("NULL}};"));

  return Class::CompileSelf()
         ->Append(code);
}
```

The member function `CompileSelf()` simply produces a parse tree that corresponds to a C++ statement that makes the record data such as `info_Point`. It first constructs the statement in the form of a character string, then converts it to a parse tree by using a utility function `MakeParseTree()`, which receives a string and replaces `%s` symbols with other sub-strings in a similar way to C language's `printf()` function. Then it converts the string to a parse tree. The result of `CompileSelf()` is the concatenation of the produced parse tree and the result of the base class' `CompileSelf()`. It is substituted for the original definition of the class `Point` in the user's program.

4 Suitability for the Real World

A MOP itself is a mechanism to implement something necessary. This section presents how the OpenC++ MOP is utilized for practical programming.

The OpenC++ MOP can be viewed as a tool for implementing libraries that efficiently and transparently provide useful facilities for the programmer. In other words, the OpenC++ MOP is primarily a mechanism for library implementors. The benefit to library users is not the MOP itself but the transparency and efficiency of the library implemented with the MOP.

4.1 Persistent-Object Library (revisited)

Using the OpenC++ MOP, the persistent objects described in Section 2 can be transparently provided for library users. A new metaclass is used to encapsulate the implementation of the extension from the user-defined classes. Such a new metaclass is developed through the three steps seen before: 1) decide what the user program should look like, 2) figure out what it should be translated to, and 3) write appropriate compile-time and runtime code to perform and support the translation. We show the persistent-object library along those three steps.

First, the program the library user writes should look as follows:

```
metaclass Node : PersistentClass;
class Node {
public:
  Node*  next;
  double value;
};

Node* get_next_of_next(Node* p)
{
  Node* q = p->next;
  return q->next;
}
```

That is, the library user can obtain persistent objects simply by specifying the metaclass `PersistentClass`.[5] There is no need for the user program to explicitly invoke `Load()` as in the C++ code in Section 2.

The next step is to figure out how to translate the user program. The library implementor must decide what should be supplied as the runtime code of the library, what should be generated for each user class and should be directly embedded in the translated code. In this example, the function `Load()` is supplied as a runtime support routine, while invocation of `Load()` is embedded in the translated code by the metaobjects. The user program should be translated to:

```
Node* get_next_of_next(Node* p)
{
  Node* q = (Load(p), p->next);
  return (Load(q), q->next);
}
```

Italic letters indicate code inserted by the metaobjects. The metaobjects translate the user program so that it appropriately invokes runtime support routines. Unlike in Section 2, the inserted code is `Load(p)` instead of `p->Load()`. This syntactical change is because `Load()` is an ordinary function. `Load()` is not supplied as a member function any more since the class `Node` does not inherit from any base class.

Finally, the library implementor writes the compile-time code of the library. For the translation specified at the previous step, the class `PersistentClass` is defined as a subclass of `Class`. It redefines the member function `CompileReadDataMember()` and `CompileWriteDataMember()` so that the member accesses like `p->next` are appropriately translated. The definition of `CompileReadDataMember()` is as follows:

```
Expression
PersistentClass::CompileReadDataMember(
                    Environment env,
                    String member_name,
                    String variable_name)
{
  return MakeParseTree(
           "(Load(%s), %e)",
           member_name,
           Class::CompileReadDataMember(
                    ...));
}
```

The `CompileReadDataMember()` supplied by the base class `Class` returns the parse tree that corresponds to the original code, in this example, `p->next`.

More Realistic Implementation

The implementation shown above is only part of the persistent library. In a typical implementation, when a persistent object is loaded onto memory, pointers that the object contains must be translated to point to correct memory addresses because the actual layout of objects changes every session. This translation, which is often called *pointer swizzling*, is needed even if references to persistent objects are detected by a virtual memory system, which is more sophisticated implementation.

To perform pointer swizzling, the runtime function `Load()` has to know which data fields of persistent objects hold pointer values. So the library must record the type of an object and its type definitions when the object is constructed at runtime. Recording the type definitions can be implemented in a similar way to the Meta-Information Protocol shown in Section 3.3.

To record the type of the object, for example, an expression `new Node()` should be translated into:

[5] Our MOP makes it possible to put even more syntactic sugar on this so the programmer can simply write: `persistent class Node {...};`. This is discussed in Section 5.

```
(Node*)RecordObjectType(new Node(),
                        "Node")
```

The runtime support function `RecordObjectType()` receives a pointer to the constructed object and the type of that object, records the type, and returns the received pointer.

The definition of the member function `CompileNew()` that performs that translation is as follows:

```
Expression
PersistentClass::CompileNew(
                  Environment env,
                  Expression arguments,
                  Expression keywords)
{
  char* pat =
    "(%s*)RecordObjectType(%e, \"%s\")";
  return MakeParseTree(
           pat,
           name,
           Class::CompileNew(env,
                             arguments,
                             keywords),
           name);
}
```

4.2 Matrix Library

The next example is a matrix library that can be implemented more efficiently if our MOP is exploited. A new metaclass is used to optimize the implementation of a specific class, which is also provided by the library.

This example is based on C++'s mechanism of overloading operators that allows us to implement a matrix library with which matrices are available in an arithmetic expression. For example, the programmer can write:

```
Matrix a, b, c, d;
  ...
a = b + c + d;
```

Although matrix arithmetic is transparently provided, the performance of this library is quite bad because each `+` operation is done through a function call and thus the intermediate value is passed as a `Matrix` object between the function calls. The following inlining is clearly better.

```
Matrix a, b, c, d;
  ...
for(i = 0; i < number_of_rows; ++i)
  for(j = 0;
      j < number_of_columns;
      ++j)
  {
    a.element[i][j] = b.element[i][j]
                    + c.element[i][j]
                    + d.element[i][j];
  }
```

Directing such an inlining scheme to the compiler is not possible within the confines of C++. Expecting the C++ compiler to automatically detect all such kinds of optimization is not realistic [1].

Our MOP enables the programmer to write a library of such customized optimizations. It allows the library implementor to provide the class `Matrix` with a metaclass for the inlining by using knowledge of implementation details of the class `Matrix`. The metaclass `MatrixClass` will have this member function:

```
Expression
MatrixClass::CompileMemberFunctionCall(
               Environment env,
               String member_name,
               String variable_name,
               Expression arguments)
{
  if(member_name == "="){
    Expression assigned_expr
               = arguments->GetFirst();
    if(inlining is applicable)
      return MakeParseTree(
               "for(i = 0; ...", ...);
  }

  // Otherwise, do the default translation.
  return
    Class::CompileMemberFunctionCall(
             env, member_name, ...);
}
```

Because C++ regards an overloaded operator as a member function call, the member function `CompileMemberFunctionCall()` shown above is invoked to translate the expression `a = b + c + d` into the inlined form. On this invocation, the operator `=` is interpreted as a member function call on the object `a`. The sub-expression `b + c + d` is an argument of the `=` operator, so the parse tree of that sub-expression is passed to `CompileMemberFunctionCall()` as the parameter `arguments`. The invoked member function checks the passed parse tree, and then, if the inlining is applicable, it actually applies the inlining to the expression; it generates a parse tree that corresponds to the inlined expression. Otherwise it would have delegated the translation to the base class.

To determine whether the inlining is applicable, the compile-time code must traverse the passed parse tree and see whether the shape of the tree matches a particular pattern. For example, if the parse tree represents repetition of an identifier and the symbol `+`, then the inlining is applicable.

4.3 Customizing Implementation of Objects

Some C++ compilers, such as GNU C++ 2.5.8 for SPARC and Sun C++ 2.0, do not allocate objects in registers. For example, if we compile the following program:

```
class Vector {
public:
  double x, y;
};

double xpos[1000], ypos[1000];
Vector v;
  ...
for(int i = 0; i < 1000; ++i){
  xpos[i] += v.x;
  ypos[i] += v.y;
}
```

In the `for` loop, the compiled code loads the values of `v.x` and `v.y` into registers for every iteration. This is obviously redundant since the values should be loaded only once before the iteration starts.

An experienced programmer can remove those redundant load instructions by editing the `for` statement as follows:

```
double s = v.x, t = v.y;
for(int i = 0; i < 1000; ++i){
  xpos[i] += s;
  ypos[i] += t;
}
```

Since `double` variables `s` and `t` are allocated on registers, the compiled code does not include redundant load instructions and runs faster.

Although this kind of technique is popular in C and C++ programming, the performance of the compiled code depends on implementation of compilers. The technique shown above works for particular compilers, but it may not for other compilers or even other versions of those compilers.

Our MOP provides the ability to do such optimization in a more sophisticated way. Programmers can separate description for optimization from the rest of the program. The program would look like:

```
metaclass Vector LightWeightClass;
class Vector {
public:
  double x, y;
};

double xpos[1000], ypos[1000];
Vector v;
  ...
for(int i = 0; i < 1000; ++i){
  xpos[i] += v.x;
  ypos[i] += v.y;
}
```

The metaclass `LightWeightClass` specializes implementation of instances of the class `Vector`.[6] Since the implementation scheme is separately described from the program above, programmers can independently customize it to fit their compilers.

To do the optimization shown first, the metaclass `LightWeightClass` redefines the member functions to translate the program into:

```
double xpos[1000], ypos[1000];
double x_of_v, y_of_v;
  ...
```

[6] Our MOP enables more smart annotation. As discussed later, programmers can switch implementation by simply adding or removing an annotation `lightweight` to variable declaration. For example, if they write:

```
lightweight Vector v;
```

the variable `v` is implemented in the lightweight way. Otherwise, it is done in the ordinary way.

```
for(int i = 0; i < 1000; ++i){
  xpos[i] += x_of_v;
  ypos[i] += y_of_v;
}
```

Now the `Vector` object `v` is implemented with two distinct `double` variables.

The definition of the member function `CompileVarDeclaration()` is as below. It translates variable declarations.

```
Expression
LightWeightClass::CompileVarDeclaration(
                     Environment env,
                     String variable_name)
{
  return MakeParseTree(
           "double x_of_%s, y_of_%s",
           variable_name,
           variable_name);
}
```

Also, the member function `CompileDotReadDataMember()` is redefined to translate data-member accesses.

```
Expression
LightWeightClass
::CompileDotReadDataMember(
    Environment env,
    String member_name,
    String variable_name)
{
  return MakeParseTree("%s_of_%s",
                        member_name,
                        variable_name);
}
```

4.4 Selecting a Concrete Class at Compile Time

The last example is a mechanism to select the most appropriate concrete class for a given abstract class at compile time. With this mechanism, programmers do not have to directly instantiate a specific concrete class. Instead, they can instantiate an abstract class with an annotation about the requirement for the implementation of the instance. The effective concrete class is selected by the compiler.

A class like `Set` can be implemented with different data structures and algorithms. Since appropriate implementation depends on user programs, a typical implementation scheme is to define several subclasses each of which corresponds to a different implementation scheme, such as `LinkedListSet`, `ArraySet`, and `SortedSet`. But in the typical implementation scheme, selecting the appropriate concrete class happens at runtime, thereby incurring a performance overhead.

The mechanism we show below is similar to that in [14], but it automatically selects the most appropriate subclass at compile time. The user just annotates requirements for each instantiation of a class, then the compiler selects a subclass to mostly satisfy the requirements. For example, the user program of the `Set` library will look like:

```
Set* s = new Set("size<=1000;sorted");
```

The requirement is specified as the first initialization parameter to the class `Set`.[7] The compiler selects a subclass of `Set` that matches the specified requirement, and translates the code above into like:

```
Set* s = new SortedArraySet();
```

`SortedArraySet` is a subclass of `Set`, which the compiler selects for that particular user program.

To enable that translation, the library implementor defines a metaclass `SetClass` of the class `Set`. It redefines the member function `CompileNew()` as follows:

```
Expression
SetClass::CompileNew(Environment env,
                     Expression arguments,
                     Expression keywords)
{
  Expression requirement
              = arguments->GetFirst();
  String class_name = SelectSubclass(
                        requirement);
  return MakeParseTree("new %s()",
                       class_name);
}
```

[7] If we use our MOP's capability to extend the syntax, we can separate the requirement from the initialization parameter. The program could be:

```
new require("size<=1000;sorted") Set().
```

```
String SetClass::SelectSubclass(
                    Expression requirement)
{
  ...
}
```

The member function `SelectSubclass()` interpretes the requirement, which is passed as a character string, and returns the name of the selected subclass for that requirement.

Although the code shown above selects a subclass at compile time, it is also possible to postpone the final decision to runtime. For example, the user may want to use runtime information for the requirement. In this case, the metaobject translates a `new` expression into an expression that calls an appropriate runtime support function, which selects a subclass and returns its instance. This implementation is also available without the MOP, but if we use the MOP, interpretation of the requirement can be done at compile-time. For example, the metaobject may translate the requirement from a string to an appropriate data structure for the runtime support function to efficiently handle it.

5 Other Issues

This section briefly surveys several other issues on the OpenC++ MOP.

Inheritance

An interesting issue is whether a subclass should inherit the metaclass of its base class. This is important because selecting the metaclass is the main mechanism for directing what MOP-based customization programmers use. The OpenC++ MOP provides customizability on this issue as well.

Our MOP selects the metaclass of a given class `X` with the following algorithm:

1. If the class `X` has a base class, then call `ComputeMetaclassName()` on the metaobject for that base class, and select its resulting value for the metaclass of `X`. By default, this member function simply returns the same metaclass as that of the base class.
2. If the metaclass of `X` is explicitly specified by the programmer with the `metaclass` declaration, then select that metaclass.
3. Otherwise, select the default metaclass `Class`.

By the first rule, subclasses inherit the metaclass from their base class. This inheritance policy can be customized by programmers, however. They can redefine the member function `ComputeMetaclassName()` to customize the policy.

In the case where a class has more than one base class and the class metaobjects for them give different metaclasses, we simply raise a compilation error. Other researchers have proposed automatic derivation of a mixed-in metaclass in this case [5], but applying that idea to the OpenC++ MOP is not straightforward because combining the same member function of two metaclasses is not always possible.

Syntax extension

The OpenC++ MOP provides limited ability to extend the language syntax. The programmer can register new keywords, which can appear only in certain limited places: the modifiers of type names, class names, and the `new` operator. For example, the following code is available.

```
distributed class Point { ... };
lightweight Vector v;
p = require("sorted") new Set;
```

`distributed`, `lightweight` and `require` are registered keywords. These keywords are passed to a metaobject when a code fragment is translated. The metaobject can use those keywords to decide how to translate that code fragment. The keywords may be followed by some *meta* arguments. For example, `"sorted"` is an argument of the keyword `require`. The arguments are simply passed to the metaobject together.

Otherwise, the keyword registered by the programmer must appear as the member name in a member access expression. In this case, the parser recognizes the whole member access expression as a user-defined statement. For example,

```
Matrix big_matrix = ...;
big_matrix->foreach(row == 1) {
  element = 1.0;
}
```

`foreach` is a keyword. It must be followed by an arbitrary expression `(row == 1)`, and a statement `{ element = 1.0 }`. The expression may be a list of formal arguments, or it may take the `for` statement's style, which is `(`<*expression*>`;` <*expression*>`;` <*expression*>`)`. The OpenC++ MOP itself does not specify any meaning to the interpretation of that `foreach` statement. The interpretation is responsibility of the class metaobject for `Matrix`, which may translate that statement to an appropriate statement of C++ as macros do in Lisp. The translated code may look like:

```
for(int row = 0;
    row < number_of_rows;
    ++row)
  for(int col = 0;
      col < number_of_columns;
      ++col)
    if(row == 1){
      big_matrix->element[row][col]
        = 1.0;
    }
```

Although this code is not efficient, the library implementor can write compile-time code to translate the statement into more efficient code if the condition such as `(row == 1)` matches a particular pattern. For example, the compile-time code may check whether the parse tree of the given condition represents a sequence of the identifier `row`, the symbol `==`, and an integer. If so, the compile-time code can remove the `for` loop on `row`.

Protocol Overheads

Since programs translated by the OpenC++ MOP invoke runtime support functions, the MOP may initially seem to involve runtime penalties. But this kind of penalty is not due to the MOP itself. It is due to the implementation scheme of the library. For example, in the persistent-object library, the translated program invokes a function `Load()` every member access. But the performance penalty

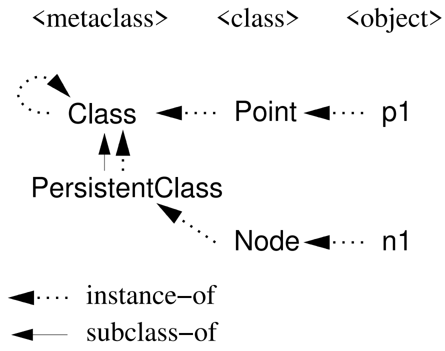

Figure 4: Metaclass, Class, and Object

with this function invocation is inherent in the implementation scheme we chose for the library. In fact, this function invocation is needed even if we do not use the MOP, as we saw in Section 2.

Although the OpenC++ MOP does not involve runtime penalties, it involves compile-time penalties since it moves meta-level computation from runtime to compile-time. However, at least regarding to the default metaobjects, the compile-time penalties can be reduced by elaborate implementation. The phase 2 and 3 of the protocol structure, which we showed in Section 3.2, are fused into very simple computation since the member function `CompileSelf()` on the default class metaobjects is an identity function that generates the same source code as the input.

Meta Circularity

The design of the OpenC++ MOP is, like other MOPs, conceptually meta-circular. There are no substantial differences between metaclasses and classes. A metaclass is simply a class that instantiates other classes (i.e. metaobjects for the classes). The relationship between a class and a metaclass is equivalent to the class-instance relationship (Figure 4). Thus when a program includes definitions of metaclasses, the MOP also constructs class metaobjects for those metaclasses. The constructed metaobjects control compilation of the metaclasses.

The OpenC++ MOP, however, avoids the apparent infinite regress of this meta circularity in a way similar to that of other meta-circular MOPs. To compile a class, its metaclass must be first compiled, before compiling that metaclass, its metaclass must be compiled, and so on. But such a chain of compilation is not infinite because the metaclass `Class` is the root of any class-metaclass chain and it is the metaclass of itself. The only question we have to answer is how the MOP compiles `Class` for bootstrapping. The answer is simple. `Class` is directly compiled by the C++ compiler because the compilation specialized by `Class` is equivalent to the compilation done by the C++ compiler.

6 Related Work

In previous work, we have proposed another MOP for C++, called OpenC++ Version 1 [3]. That MOP had the ability to transparently implement language extensions for distributed computing as libraries on top of ordinary C++. But, because that MOP was based on a meta-architecture in which the metaobjects exist at runtime, it implied runtime overheads. Also, that MOP provided only limited ability to control program behavior, covering only member access and object creation. The degree of transparency of libraries written with that MOP was not enough for real-world programming.

A number of ideas for the OpenC++ MOP came from previous work. The idea of a compile-time MOP is due to Anibus and Intrigue [18, 13]. These are compile-time MOPs for controlling a Scheme compiler. In those MOPs, the metaobjects are not just language entities, but also represent global information such as the results of flow analysis. Our basic protocol architecture is due to the CLOS MOP[11]. The major difference is that the CLOS MOP's metaobjects are runtime ones and thus the CLOS MOP requires relatively large runtime environment if it is directly applied to C++. The idea of a meta-interface of the early-stage of compilation was also proposed in MPC++[8].

Like the CLOS MOP and OpenC++ Version 1, a number of systems [15, 7, 17, 22] adopt runtime metaobjects, which represent underlying mechanisms such as the language interpreter and the OS kernel, and are responsible for runtime behavior of the system. Since the runtime metaobjects allow users to change various decision policies of the system, such as scheduling and migration, the users can tune the system performance to fit their needs. A drawback of the runtime metaobjects is runtime overhead. A few ideas have been proposed on this problem [12, 16, 19]. For example, inlining and partial evaluation are effective techniques to reduce the overhead. It is difficult to recover the whole overhead of a runtime meta architecture, however.

7 Current Status

We are in the process of developing OpenC++ Version 2. Our methodology is to first develop a simplified version of the target system (i.e. C++), then design and test a MOP for that simplified system, and finally port the developed MOP back to the target system. We have thus developed a C++-like object system, called S++, on top of Scheme, and designed the MOP presented in this paper for S++. A number of examples similar to these presented here have been implemented to test the S++ MOP, and we have repeatedly re-designed the MOP based on the results of the tests. We are currently porting our MOP back to C++.

8 Conclusion

This paper describes a metaobject protocol for C++. It was developed to bring the power of metaprogramming to a more mainstream language. This MOP differs from most existing MOPs in that the metaobjects exist exclusively at compile-time — they control the compilation of programs to alter the behavior of the basic language constructs of C++. This feature means that this MOP involves no runtime speed or space overheads.

This paper also illustrates how the customizability provided by our C++ MOP can be used to implement language extensions efficiently and transparently as libraries. Currently, many language extensions such as persistence or distribution end up being re-implemented for each application because the existing language mechanisms are insufficient for customizing existing code to fit each application. Our C++ MOP enables us to implement those extensions as libraries, making such extensions easier to develop and maintain and thus more reusable.

Acknowledgments

The basic idea of the OpenC++ MOP was produced through discussions with Gregor Kiczales. The author thanks John Lamping, Ellen Siegel, and Chris Maeda for their comments on early drafts of this paper. The author's work was partially supported by Japan Society for the Promotion of Science.

References

[1] Angus, I. G., "Applications Demand Class-Specific Optimizations: The C++ Compiler Can Do More," *Scientific Programming*, vol. 2, no. 4, pp. 123–131, 1993.

[2] Buschmann, F., K. Kiefer, F. Paulisch, and M. Stal, "The Meta-Information-Protocol: Run-Time Type Information for C++," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 82–87, 1992.

[3] Chiba, S., "Open C++ Programmer's Guide," Technical Report 93-3, Dept. of Information Science, Univ. of Tokyo, Tokyo, Japan, 1993.

[4] Chiba, S. 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, pp. 482–501, Springer-Verlag, 1993.

[5] Danforth, S. and I. R. Forman, "Reflections on Metaclass Programming in SOM," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 440–452, 1994.

[6] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[7] Honda, Y. and M. Tokoro, "Soft Real-Time Programming through Reflection," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 12–23, 1992.

[8] Ishikawa, Y., "Meta-Level Architecture for Extendable C++," Technical Report 94024, Real World Computing Partnership, Japan, 1994.

[9] Kiczales, G., "Towards a New Model of Abstraction in Software Engineering," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 1–11, 1992.

[10] G. Kiczales, ed., *Workshop on Open Implementation'94*, internet publication (`http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94`), Oct. 1994.

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

[12] Kiczales, G. J. and L. H. Rodriguez Jr., "Efficient Method Dispatch in PCL," in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 99–105, 1990.

[13] Lamping, J., G. Kiczales, L. Rodriguez, and E. Ruf, "An Architecture for an Open Compiler," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 95–106, 1992.

[14] Lortz, V. B. and K. G. Shin, "Combining Contracts and Exemplar-Based Programming for

Class Hiding and Customization," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 453–467, 1994.

[15] Maes, P., "Concepts and Experiments in Computational Reflection," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, 1987.

[16] Masuhara, H., 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*, pp. 127–144, 1992.

[17] Okamura, H., Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 36–47, 1992.

[18] Rodriguez Jr., L. H., "Coarse-Grained Parallelism Using Metaobject Protocols," Techincal Report SSL-91-61, XEROX PARC, Palo Alto, CA, 1991.

[19] Ruf, E., "Partial Evaluation in Reflective System Implementation," in *Proc. of OOPSLA'93 Workshop on Reflection and Metalevel Architectures*, 1993.

[20] Steele, G., *Common Lisp: The Language*. Digital Press, 2nd ed., 1990.

[21] Watanabe, T. and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 306–315, 1988.

[22] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 414–434, 1992.