Designing an Extensible Distributed Language with a Meta-Level Architecture

Shigeru Chiba* Takashi Masuda
Department of Information Science, The University of Tokyo
E-mail: {chiba,masuda}@is.s.u-tokyo.ac.jp

In Proceedings of 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, July 1993, LNCS 707, pp.482–501

Abstract

This paper presents a methodology for designing extensible languages for distributed computing. As a sample product of this methodology, which is based on a meta-level (or reflective) technique, this paper describes a variant of C++ called *Open C++*, in which the programmer can alter the implementation of method calls to obtain new language functionalities suitable for the programmer's applications. This paper also presents a framework called *Object Communities*, which is used to help obtain various functionalities for distributed computing on top of Open C++. Because the overhead due to the meta level computation is negligible in distributed computing, this methodology is applicable to practical programming.

1 Introduction

Languages for distributed computing have been designed mostly to provide a general functionality that can be used in a broad range of application domains. Designers of these languages have developed numerous language primitives or functionalities, such as Ada's rendezvous [26], the remote procedure call [2], and Orca's shared data-object [1]. Each of these functionalities has its own most suitable domain of applications, so a language that has a single one of these functionalities will be small and simple but will not be suitable for some applications. It is, on the other hand, possible to design a language that has many or all such functionalities, but such a language would be large and awkward.

*JSPS (Japan Society for the Promotion of Science) Fellow-DC

The goal of this paper is to demonstrate another approach, which is to make a language extensible. By this approach, we have been able to design a language that is, at the same time, simple, elegant, and applicable in a wide range of domains. A programmer can tailor the language to exploit various functionalities. Language extensibility has long been an important issue, and Kiczales et al., for example, have recently discussed the designing of extensible class libraries [11]. A typical approach to supporting various functionalities within a single language is to provide a set of reusable code, called a library program, that implements functionalities that are not supported by the language alone. Although functionalities implemented by this approach may show lower performance than ones implemented by altering the language system such as the compiler, this approach is broadly employed because sufficient performance is usually obtained by this approach in practice. The library-program approach, however, is limited in that it cannot implement a functionality that deals with non-first-class entities of the language.

This paper proposes methodology using an object-oriented meta-level technique in designing of an extensible language for distributed computing. To demonstrate the use of this methodology, we present *Open C++*, which is a C++ [23] variant including a simple *metaobject protocol* (MOP) [10]. In Open C++ the implementation of a method call (or in the object-oriented terminology, message passing) is made open-ended by that MOP. To obtain a new functionality that fits the application, the programmer can easily extend the implementation within Open C++ itself. Performance overheads are one of major issues in meta-level techniques, but they are not critical in domains such as distributed computing, which Open C++ deals with. The seriousness of the overheads depends on the inherent cost of functionalities achieved with the meta-level technique. Since the overhead of Open C++ is negligible in comparison with the implemented functionalities, we believe that our approach is — like the library-program approach, which is useful in spite of its relative slowness — applicable to actual problems.

As with other systems using meta-level techniques, an extension of Open C++ is described in meta code (meta-level program). Although meta code is usually written only by a system specialist because MOP would be often complicated and extension was not frequent, we expect normal programmers (who are not "wizards") to write meta code in Open C++ whenever a new functionality is required for their applications. The Open C++ MOP is therefore designed to provide an abstraction that encapsulates implementation details unnecessary to the extension of a method call. To facilitate extension by normal programmers, this paper also provides a framework, called *Object Communities*, that includes some basic functionalities for extending a method call for distributed computing. With this framework, normal programmers can easily obtain various functionalities for distributed computing on top of Open C++.

2 Open C++: A Simple MOP for C++

In most imperative languages for distributed computing, procedure calls (or in the object-oriented terminology, method calls) are extended to support remote communication across a network. Those extended method calls provide not only a functionality invoking a procedure (or a method) at a remote machine, but also a functionality synchronizing multiple threads of control. In Ada [26] and Concurrent C [5], for example, a statement syntactically similar to a procedure call is used for executing a rendezvous, and a procedure call is extended to block the sender thread until the receiver is ready. In ConcurrentSmalltalk [28], a method call of Smalltalk-80 [6] is extended to be synchronous or asynchronous: an *asynchronous method call* lets a sender thread continue its execution without blocking, whereas a *synchronous method call* blocks the sender thread until the receiver thread finishes a requested task.

By using a meta-level or so-called reflection technique [21], Open C++ offers normal programmers the ability to extend a method call. Normal programmers can modify the implementation of a method call within a user program to obtain various functionalities for remote communication. The implementation of a method call is exposed to programmers as a *metaobject* [15], which is an abstract model of that implementation and conceals implementation details unnecessary to the extension. A metaobject is almost the same as a normal object, but its behavior corresponds to the actual execution of the method call. An object at the *base level* has its metaobject at the *meta level*, and the execution of its methods is controlled by the metaobject. If a method of the object is invoked, the specific method of the metaobject, instead of a default implementation embedded in the compiler, is used to execute the invoked method. Since a metaobject is defined in C++, the programmer can alter the implementation of a method call by defining another metaobject and then substituting it. Our approach does not require rebuilding the compiler but is done within a user program.

2.1 Base-Level Directives

Open C++ provides a very simple MOP (metaobject protocol*) to make a method call extensible. The objects controlled by metaobjects are called *reflective objects*. Because control by a metaobject imposes some performance and memory overhead in Open C++, the programmer can specify whether or not an object is reflective. A nonreflective object is compiled to be a normal C++ object, which has no metaobject, so that it is executed without overhead. To distinguish between reflective and nonreflective objects, a reflective object is identified by a different class name. If the class of an object that may be reflective is `X`, then a reflective object is `refl_X` and a nonreflective object is still `X`. In the current implementation, the class `refl_X` is a subclass of `X`.

*A metaobject protocol is a meta-protocol organized using object-oriented techniques. Here a meta-protocol is a protocol about the behavior and implementation of another protocol, such as interface and functionality.

To create a reflective object, the class of the object and its metaobject must be declared with special directives, which are C++ comments that start with `//MOP`. Note that even if a program includes the directives of Open C++, that program is still a valid C++ program. The declaration of a reflective object takes the form

```
//MOP reflect class X : M;
```

This declaration means that an object of the class `refl_X` is a reflective object controlled by a metaobject of the class `M`. Note that it never means that the classes `X` or `refl_X` are subclasses of `M`. The class `M` is a normal C++ class except that it must inherit from the class `MetaObj`. To extend its implementation of a method call, a metaobject can be a reflective object that is controlled by a meta-metaobject. Open C++ allows such an ascending tower of metaobjects.

The methods of a reflective object are divided into two groups, depending on whether the invocation of the method is controlled by its metaobject. The methods controlled by the metaobject are called `reflect` methods, and although `reflect` methods are invoked in an extended manner, the other methods are invoked according to the plain C++ method call semantics. The following is an example of specifying a `reflect` method.

```
class X {
public:
    X();
//MOP reflect:
    int func(int);
private:
    int p;
};
```

The methods following the directive "`//MOP reflect:`" are specified as `reflect` methods. Here, for example, `func()` is a `reflect` method. Such methods may have a category name to enable their metaobject to recognize a role of the methods. A metaobject may alter the execution of a method call according to the category name. Consider the following example: The method `update()` has a category name "`write`".

```
class Y {
public:
    ...
//MOP reflect(write):
    int update(int);

//MOP reflect(metamethod):
    void Meta_operation();
    ...
```

The category name "`metamethod`" has a special meaning: it is used to call meta-methods of a metaobject from the base level across the boundary of the levels. Calling a `reflect` method in this category is regarded as calling a meta-method that has the same *method* name. The `reflect` methods having the category name "`metamethod`" themselves are never executed.

2.2 Metaobject Protocol

When a `reflect` method is called, its execution is controlled by its metaobject. A metaobject is defined in C++, and its class must inherit from the base class `MetaObj`, which mainly defines the following two methods.

- `void Meta_MethodCall(Id method, Id category, ArgPac& args, ArgPac& reply);`

  This method implements a method call at the base level. It is invoked if a `reflect` method is called.

- `void Meta_HandleMethodCall(Id method, ArgPac& args, ArgPac& reply);`

  This method is used to actually execute a `reflect` method.

To alter the implementation of a method call, the programmer defines a subclass of `MetaObj` in which those methods are redefined so that the metaobject acts in the intended way.

Suppose that a `reflect` method `f()` is called. If the method `f()` is called, then the method `Meta_MethodCall()` is instead invoked at the meta level. The first argument of the method `Meta_MethodCall()` is bound to the integer identifier of the called method `f()` (the type `Id` represents integers), and the second argument represents the category name of the method `f()`. The actual arguments of the method call to `f()` are passed as the third argument, `args`. Note that within a metaobject, the actual argument list of a method call is a first-class entity because the third argument, `args`, is a normal C++ object whose class is `ArgPac`. The argument `args` has the same interface as a stack so that the programmer can access any actual argument stored in `args`. The programmer can also transfer the argument `args` to another metaobject that may reside on a different machine. Converting the actual arguments to an `ArgPac`-class object corresponds to the *reifying* process, which is impossible in C++ alone without support of the Open C++ compiler.

The method `Meta_MethodCall()` carries out certain computation and stores the result into the fourth argument, `reply`. The stored result is returned as a return value to the caller that calls the `reflect` method `f()`. The method `Meta_MethodCall()` usually uses the method `Meta_HandleMethodCall()` to compute the result value. This method takes a method identifier and an actual argument list, and it returns the result value of the specified method. This method allows any `reflect` method to be executed at any time. In the example above, the metaobject can execute another `reflect` method as well as `f()` to compute the result value.

To illustrate the Open C++ MOP, consider a simple example in which this metaobject prints a message before executing a `reflect` method called at the base level:

```
class VerboseMetaObj : public MetaObj {
public:
    void Meta_MethodCall(Id method, Id category,
                         ArgPac& args, ArgPac& reply){
        printf("***reflect method %s() was called.\n",
               Meta_GetMethodName(method));
        Meta_HandleMethodCall(method, args, reply);
    };
};
```

If a metaobject of the class `VerboseMetaObj` is specified, a message is printed on the console every time a `reflect` method is called. The method `Meta_MethodCall()` specifies that this metaobject prints the name of the called method before actually executing that method. Note that if we eliminate the line "`printf(...);`" from this method, the implementation of a method call by this metaobject becomes the same as the implementation in plain C++. Figure 1 shows how a metaobject of the class `VerboseMetaObj` controls a method call. The metaobject controls an object of the class `refl_X` (as previously shown, a reflective object of the class `X`). When a `reflect` method `func()` of that object is called, the metaobject traps that method call and executes the method `func()` according to the method `Meta_MethodCall()`.

Converting the actual arguments to an `ArgPac`-class object is similar to the marshaling/unmarshaling process in remote procedure calls. In the current implementation, the class `refl_X` (which the Open C++ compiler generates) redefines a `reflect` method so that the method carries out such conversion and then invokes the method `Meta_MethodCall()` of its metaobject. The current Open C++ compiler converts some atomic types (integers, pointers, etc.) implicitly but does not class types (i.e., objects). The class types that can be an argument of a `reflect` method must have some specific methods for the conversion. A similar limitation also appears in the marshaling/unmarshaling process because the efficiency of converting complex data, such as an object, often depends on the program semantics. Such conversion should be under programmer's control [8]. Open C++, however, provides a convenient library to implement the methods for the conversion, and it also provides some predefined classes that facilitate to use a character string etc. as an argument of a `reflect` method. Thereby, the limitation on argument types of `reflect` methods is not awkward.

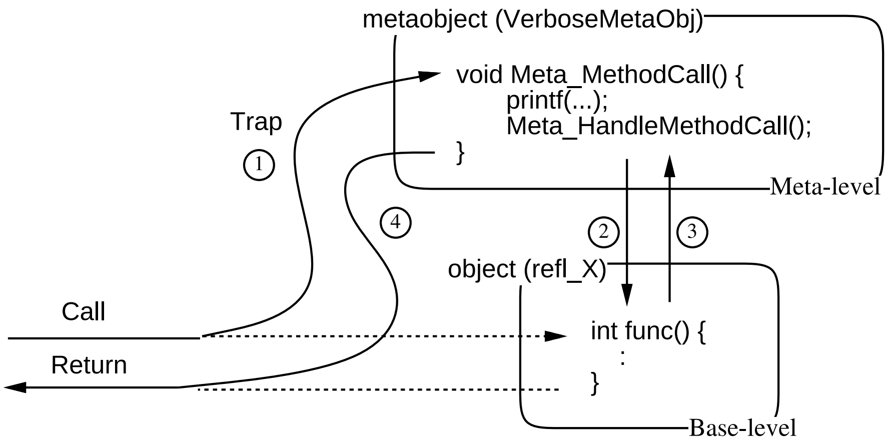

Figure 1: Metaobject protocol of Open C++

2.3 Why Meta? Pros and Cons

Open C++ does not expose the implementation of a method call directly, but through an abstract interface. Although the original implementation of a method call, which is embedded in the compiler, is described in assembly code, the programmer who attempts to extend the method call describes a new implementation of C++ methods such as `Meta_MethodCall()` instead of assembly code. Because of the description through the abstract interface, the programmer need not consider such details of the implementation as a stack image and the number of arguments. The programmer can thus concentrate on matters strongly relevant to the extension.

This feature of Open C++ is due to the meta-level technique that Open C++ uses. When a method of an object is invoked, the computation of the method call is *reified* to be entities available in a C++ program, and operations on these entities are *reflected* in the actual computation. This is a difference from "pseudo-open" systems, which directly expose their internal structure to be extensible. Smalltalk-80, for example, provides the whole source code of its runtime system. Thus in a sense, it is an open-ended system because user programmers can freely modify classes of kernel objects to extend the system behavior. This feature of Smalltalk-80 may be a kind of reflection†. Such modification of kernel objects, however, can easily lead the system into collapse because the programmer deals with the complicated kernel code directly, without an abstract interface.

On the other hand, the reifying process implies that the performance of Open C++ degenerates. The

†Peter Deutsch pointed this out at the BOF session in the '92 workshop on reflection and meta-level architecture.

cost of reifying and reflecting is not negligible compared with the original implementation fully described in assembly code. This is because the reifying process bridges the wide gap between the assembly level and the C++ level. The higher the abstract interface Open C++ provides for extension, the bigger the performance degeneration of the reifying process will be. This degeneration is negligible, however, when Open C++ is used for distributed computing. The method call extended for distributed computing is so slow that the performance degeneration becomes relatively insignificant. This issue is discussed in detail in Section 5.

Another benefit of Open C++ is that meta code defines the extension independently of each object so that meta code has high reusability. The same meta code can be used to extend method calls to different objects. Because meta code is organized according to the metaobject protocol, furthermore, part of it is also reusable by class inheritance.

Open C++ improves the expressive power of a class library, which is also a technique for supporting various functionalities within a single language. If a functionality like remote method calls is implemented solely by means of class libraries, the translation of an argument list into a network message becomes responsibility to the programmer. This is because the class library alone cannot deal with any entities except these available at the base level, and an argument list is available not at the base level but at the meta level. On the other hand, Open C++ enables a class library to deal with an entity available at the meta level through a metaobject. For example, it can use a metaobject for transferring an argument list to a different machine and can execute a remote method.

3 Object Communities — An Additional MOP for Distributed Computing

Because a method call is a good basis of functionalities for distributed computing, various functionalities can be implemented on top of Open C++. Most imperative languages include a method call statement, and it has been used to implement a lot of existing functionalities for distributed computing. A method call can be extended to support not only a remote method call but also asynchronous message passing and message broadcasting. It can also be extended to be a synchronization mechanism such as a rendezvous or a distributed semaphore.

To obtain a functionality suitable for the application, normal programmers should themselves describe meta code to extend a method call. Although previous systems usually expected meta code to be written only by a specialist, the simple MOP of Open C++ makes meta programming possible for programmers with little knowledge as well as for specialists. The MOP of Open C++, however, does not in itself support distributed computing; it only provides a platform on top of which a functionality for distributed computing is implemented. This section proposes a framework, called *Object Communities*, that facilitates to implement such a functionality on top of Open C++. This framework is a class library of metaobjects and includes facilities that are commonly used to extend a method call. *Object Communities* add a layered protocol onto the MOP of Open C++. It provides the classes of metaobjects that implement some typical functionalities for distributed computing so that programmers can obtain functionalities tailored to their applications by redefining some methods of those classes.

3.1 Background Problem

Object Communities are designed to be a framework for implementing various application-specific functionalities for distributed computing, such as distributed shared data, distributed transactions, remote procedure calls. Such a framework must provide a facility managing computation distributed to multiple processes on different machines. A simple client-server framework based on remote procedure calls is not sufficient as such a framework because although it can request computation to another process, it cannot synchronize computation between processes.

The simple client-server framework, for example, cannot in an easily understandable way implement the functionality required by groupware[4] (or multiuser applications), which supports collaborative work by multiple users. The essential feature of groupware is that an application program consists of multiple *autonomous* processes that are responsible for interaction with each user. Those processes interfere with each other because the users manipulate shared entities, such as shared documents and pictures, and their actions are therefore restricted by the actions of other users. The processes may also notify each other when shared entities are updated and they can request computation, such as redrawing the displays, in order to keep consistent images of the entities on the displays. To do these things, the application needs a functionality that makes it possible to block the execution of other processes as well as to request computation to other processes.

3.2 Overview of Object Communities

The fundamental functionality of *Object Communities* is the management of a group of objects distributed in different machines. Such a group is called *an object community* (Figure 2). We assume that each object belongs to a single process that has its own address space separated from others and communicates with other processes across a network. A process is invoked explicitly by the user, and it performs cooperatively with other processes in the same application.

Each object of an object community acts in a manner that depends on behavior of other objects of that object community. The method calls to the objects are executed cooperatively by the metaobjects so that the objects provide a certain functionality for distributed computing. Note that although a group of objects as a whole provides some functionality, the definition of the objects does not include any distributed concepts:

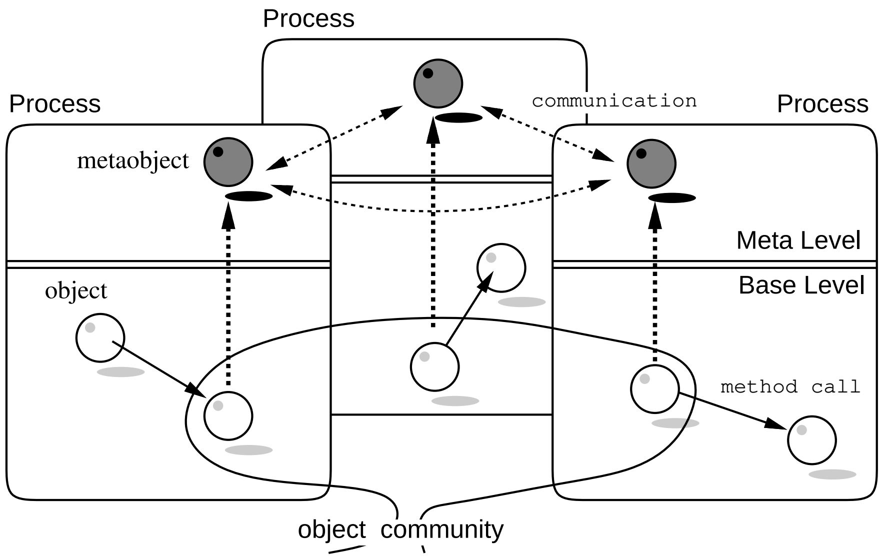

Figure 2: An object community

these appear only in the definition of the metaobjects. The functionality provided is implemented at the meta level, and the base-level programmer has only to know how a method call is extended at the base level. *Object Communities* provide a clear separation between distributed computation and the substantial computation executed in the application.

In *Object Communities*, a metaobject has the following additional abilities.

- *Concurrency Control*. A metaobject controls the internal concurrency of its object. It can ignore and delay execution of a called method of the object until some condition is satisfied. A metaobject can also execute multiple methods of its object concurrently. And a metaobject can execute a method of its object when other metaobjects request that the method be executed.

- *Communication*. A metaobject has two means of communicating with other metaobjects of the same object community: a remote method call and message broadcasting. A metaobject can call a remote method of other metaobjects. This is done in a manner similar to that of a local method call. The caller metaobject is blocked until a reply is returned from the called object. A metaobject can also send a message to all metaobjects of the same object community. Because broadcast messages are serialized by the underlying system, all metaobjects receive the messages in the same order. A broadcast message is also delivered to the metaobject that sent the message.

Although a metaobject controls the internal concurrency of its object, there is with few exceptions no internal concurrency of the metaobject by default. The methods of a metaobject are executed sequentially, so the behavior of a metaobject is easily understandable. If internal concurrency of a metaobject is necessary, it must be controlled by an explicitly specified meta-metaobject.

3.3 MOP of Object Communities

To append the *Concurrency Control* and *Communication* abilities, *Object Communities* provide the class `OcCoreMetaObj`, which is a subclass of `MetaObj`, and the other classes of metaobjects that implement functionalities based on *Object Communities* must inherit from this subclass.

The class `OcCoreMetaObj` defines some methods for manipulating an object community, for network communication, for controlling concurrency, and so on. The following methods are to manipulate an object community.

- `Meta_CreateOc(...)` creates an object community.
- `Meta_DestroyOc(...)` destroys an existing object community.
- `Meta_Join(...)` lets an object join a specified object community.
- `Meta_Leave(...)` lets an object leave a specified object community.

An object community is treated if it were a communication channel. An object can join or leave an object community at any time, but the object community remains even if no object belongs to it. It exists until it is destroyed explicitly. To give initial information to a metaobject that joins an object community, the underlying system holds an initializing message for each object community. This message, which can be dynamically updated by a metaobject, is passed to a metaobject when its object joins to an object community.

The class `OcCoreMetaObj` defines three methods for communication with other metaobjects.

- `Meta_EventNotify(...)` broadcasts a message to the other metaobjects of the same object community.
- `Meta_Query(...)` calls a method of other metaobjects in a manner like that of the remote procedure call. The metaobject is blocked until a reply message arrives.
- `Meta_WaitForEvent(...)` blocks a metaobject until it is ready to receive a broadcast message. A metaobject can use this method to wait for a message broadcast by itself.

A message sent with the first two methods must be a pair consisting of an method identifier (`Id`) and an actual argument list (`ArgPac`). By sending a message, a metaobject requests other metaobjects to execute a method of their object so that the methods are executed cooperatively.

The behavior of a metaobject receiving a message is defined by the following methods. The class `OcCoreMetaObj` only declares these methods; their bodies are defined in its subclasses to alter the behavior of each metaobject.

- `Meta_EventCallbackBody(...)` is executed when a broadcast message is received.
- `Meta_SelfEventCallbackBody(...)` is executed when a broadcast message that the metaobject itself sent is received.
- `Meta_ReplyQueryBody(...)` is a method exported to other metaobjects. This method can be called by other metaobjects with the method `Meta_Query()`.

Although basically there is no internal concurrency of a metaobject, these three methods may be executed concurrently when the metaobject is blocked by either the method `Meta_Query()` or `Meta_WaitForEvent()`. This exception is necessary to prevent a deadlock.

The current implementation of *Object Communities* does not provide a preemptive scheduler. The programmer must therefore voluntarily cause a context switch at short intervals. The class `OcCoreMetaObj` defines methods like `WakeupTaskSv()` and `RecvMessage()` to cause a context switch. Note that implementing a preemptive scheduler is possible, and that a preemptive scheduler can, in fact, be obtained if a timer-signal handler is available. The reason that a nonpreemptive scheduler is selected is to prevent the internal concurrency of a metaobject that has no meta-metaobject. The methods of a metaobject are executed atomically; they are not preempted.

4 Examples of Method-Call Extension

Many functionalities for distributed computing can be implemented as a group of objects on different machines. Since *Object Communities* provide the ability to manage a group of objects, such a functionality is implemented on top of Open C++ by defining a subclass of the class `OcCoreMetaObj`. In fact, *Object Communities* already include some subclasses of `OcCoreMetaObj`, which implement various functionalities for distributed computing. Figure 3 illustrates the class hierarchy of metaobjects provided by *Object Communities* in default.

The class `NullMetaObj` is irrelevant to *Object Communities*: it implements a method call that is done in the original manner of C++ method calls. The other subclasses correspond to various functionalities. They implement distributed shared data, transactions, and remote procedure calls. They also implement remote object pointers with which an object can transparently call a method of an object on a different machine. The implementation of remote object pointers exploits other programming techniques such as

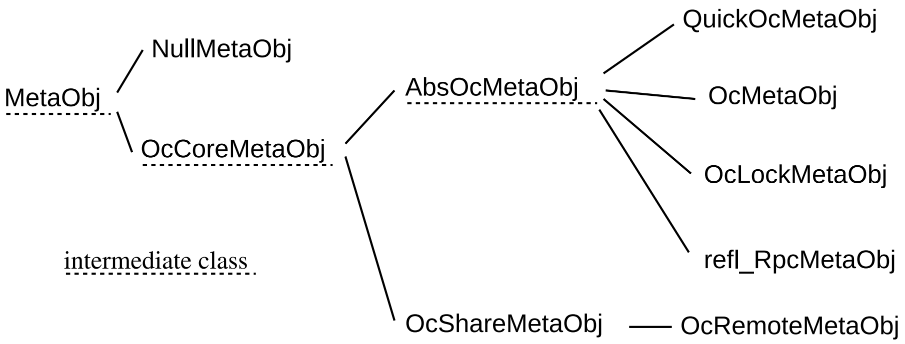

Figure 3: Class hierarchy of metaobjects

"smart pointers" [23] so that remote object pointers are naturally available in C++. Furthermore, another subclass implements persistent objects by using the ability of Open C++ to deal with instance variables of an object by the metaobject. Because of space limitation, the details of this ability are not given here; we will present them in another paper.

Here we explain two of the subclasses of *Object Communities*: distributed shared data and transactions. Distributed shared data are implemented by the class `OcMetaObj`. The shared data are replicated and held by the objects that belong to the same object community. The metaobjects control those objects to hold consistent values of the shared data. Suppose that the shared data is an integer and is represented as a variable `p` of the class `SharedData` at the base level. To update the variable `p`, the class `SharedData` has a method `Update()`. If the variable `p` is updated, this method redraws a graphical display according to the new value of `p`:

```
class SharedData {
    ...
public:
//MOP reflect:
    void Update(int new_p) { p = new_p; RedrawDisplay(); }
private:
    void RedrawDisplay();
    int  p;       // inaccessible from the outside of the object
};

//MOP reflect class SharedData : OcMetaObj;
```

An object of the class `SharedData` can be a reflective object, and the class of the metaobject is `OcMetaObj`. The method `Update()` is a `reflect` method. If an object of the class `refl_SharedData` is created, the metaobject makes the object join the specified object community.

Then the variable p of the object is maintained by the metaobject to hold the same value as the values of p of the other objects of the same object community. If the method Update() is called, the metaobject requests the other metaobjects to use the same argument new_p, and execute the same method of their objects. Thus if the method Update() of an object of the object community is called, then the methods of all the objects are executed and the values of p are updated keeping consistency. Note that the definition of the class SharedData does not include any code concerning distributed computation; such code is in the definition of the metaobject. Methods of the metaobject are defined as follows:

```
void OcMetaObj::Meta_MethodCall(Id method_id, Id category,
                                ArgPac& args, ArgPac& reply){
    // notifying others of a method call
    Id event = Meta_EventNotify(method_id, args);
    // waiting until that notification is serialized
    Meta_WaitForEvent(event, args);
    // executing the called method actually
    Meta_HandleMethodCall(method_id, args, reply);
}

void OcMetaObj::Meta_EventCallbackBody(Id method_id,
                                ArgPac& args, ArgPac& reply){
    // if other metaobjects report a method call,
    // the metaobject executes the called method.
    Meta_HandleMethodCall(method_id, args, reply);
}
```

The consistency between the values of p is guaranteed even if two metaobjects attempt to execute the method Update(). This is because the notifications by those metaobjects are serialized so that every metaobject receives the notifications in the same order.

Since there is no restriction in terms of the definition of the method Update(), the programmer can define any action that is executed whenever the shared data are updated. This kind of processing cannot be adequately treated by other mechanisms for distributed shared data, such as distributed shared memory [13], because they do not support a functionality that invokes user-defined actions on each machine that shares the data.

Although in the example above other metaobjects are notified of a method call immediately, some mechanisms for distributed shared data improve performance by using an algorithm in which the notification is delayed [24]. Such an algorithm is also available in Open C++ if the programmer defines a subclass of OcMetaObj to implement it. When implementing such an algorithm, it is necessary to distinguish methods that modify the shared data from methods that simply read the data. Category names of reflect methods are useful for this. For example,

```
class SharedData2 {
    ...
```

```
public:
//MOP reflect(write):
    void Update(int new_p) { p = new_p; RedrawDisplay(); }
//MOP reflect(read):
    int Get() { return p; }
    ...
};
```

The category names let the metaobject identify the method `Update()` as a "write" method, and the method `Get()` as a "read" one.

Next we show another subclass of *Object Communities*. The class `OcLockMetaObj` of metaobjects implements atomic transactions. Although the concept of atomic transactions includes recoverability (a transaction causes no side-effect if it fails), the class `OcLockMetaObj` does not support recoverability. It only guarantees atomicity; the sequence of the operations in a transaction is executed continuously. The method `Meta_MethodCall()` of the class `OcLockMetaObj` is as follows.

```
void OcLockMetaObj::Meta_MethodCall(Id method_id, Id category,
                                    ArgPac& args, ArgPac& reply){
    while(locked)
        Meta_WaitForEvent();      // block until a lock is released.

    // the following is the same as the method of OcMetaObj
    Id event = Meta_EventNotify(method_id, args);
    Meta_WaitForEvent(event, args);
    Meta_HandleMethodCall(method_id, args, reply);
}
```

The metaobject of the class `OcLockMetaObj` delays the method execution while the execution is locked. To begin a transaction, the programmer calls a method of the metaobject, which locks method execution with a broadcast message. Receiving the message, the other metaobjects of the same object community stop method execution until that metaobject releases the lock. The variable `lock` indicates whether execution is locked or unlocked. It is maintained by messages between metaobjects.

5 Overheads due to having a Meta Level

Efficient implementation of meta-level techniques is a major research topic. Because execution of a reflective object in Open C++ is partly interpreted by a metaobject, its execution is slower than that of a nonreflective object. This section briefly shows the result of measurements in terms of the execution speed.

The current Open C++ compiler is a preprocessor of the C++ compiler. Because no modification is added to the C++ compiler, an Open C++ program is translated into a plain C++ code. Calling a `reflect`

Table 1: Average Latency (μsec.) of a null method call

number of arguments	0	1	5	$5 \times$ `double`
C++ function	0.3	0.6	1.3	2.1
C++ `virtual` method	0.8	1.0	1.8	2.2
`reflect` method	1.8	6.3	13.8	21.7
`reflect`/`virtual` ratio	2.3	6.3	7.7	9.9

SPARC 40 MHz (28.5 MIPS) and Sun C++ 2.1

method thus imposes some overhead that by some standards is not small.[‡] We show the result of performance measurements of method calls.

Table 1 lists latency time for three kinds of null method calls. These values were measured on a SPARC station 2 (SunOS 4.1.1), and the compiler was Sun C++ 2.1. The latency was measured for different numbers of arguments. The type of arguments was `int` except for the data of the rightmost column, for which the type was `double`. Although the 0-argument method does not return anything, the other methods return an `int` value. A method that takes 5 `double` arguments returns a `double` value. The three kinds of null method calls are a C++ function, a `virtual` method, and a `reflect` method. The first two are supported by both C++ and Open C++, whereas the last is available only in Open C++. A C++ function call is to call a method of an object pointed to by a variable. This takes a form like `ptr->func()`. A `virtual` method call is to call a method of an object whose class is unknown at compile time; a method name is dynamically bound to a method body. A `reflect` method call is one controlled by a metaobject of the class `NullMetaObj`, which implements a method call so that its behavior is the same as that of a C++ function call.

The last line of the table shows the ratio of the latency of a `reflect` method call to that of a `virtual` method call. This ratio increases with the number of arguments because the overhead of a `reflect` method call is mainly due to the reifying process of the argument list of the method call. Arguments are copied to an `ArgPac` class object separately when the `reflect` method is called. The overhead for this copying increases in proportion to the number of arguments. Since the 0-argument method takes no argument, its overhead is smaller than that of the other methods.

[‡]The initial version of the Open C++ compiler showed that a `reflect` method call was 100 times slower than a `virtual` method call of C++.

The result of these measurements shows that a `reflect` method call is 6 to 8 times slower than a `virtual` method call. Although this overhead seems important, it is actually negligible if Open C++ is used for distributed computing, since the network latency time is between several hundred microseconds and several milliseconds. The overhead is also reduced by a proper designing of the applications. In carefully designed applications, distributed computation is localized in a small number of objects, which would be reflective, and the other objects are executed without overhead since Open C++ allows to specify whether or not an object is reflective. We believe that meta-level techniques are already applicable to practical programming if the programmer selects a domain in which the overhead is negligible in comparison with the overhead for performance of a functionality implemented with the meta-level technique.

Furthermore, from the viewpoint of distributed computing, the overhead of Open C++ is due to the cost of the marshaling/unmarshaling process, in which transferred data are converted into a network message. Because this process commonly appears in distributed computing, the overhead of Open C++ is almost equivalent to that of other approaches such as Sun's RPC [25]. When Sun's RPC library is used, each conversion of an `int` argument takes a few microseconds because that library is a general one, and a few nested function calls are needed whenever a converting routine (an XDR routine) is called.

If the increased overhead of a meta-level technique is limited to within a factor of 10, then the advantage of that meta-level technique is worthwhile. In the concurrent language ABCL/R2 [17], for example, the execution that involves a meta-level operation is 6 or 7 times slower than a normal execution [16]. As in Open C++, the programmer can select whether or not an object is controlled by a metaobject. As a result, ABCL/R2 improves the execution speed of a program by a meta-level technique.

6 Related Work

C++ provides some meta-level operations. The macro set of handling a variable argument list can be considered to provide a few restricted meta-level operations. It allows the programmer to traverse an argument list whose length and element types are variable, as if the argument list were a first-class entity. Operator overloading is also a meta-level operation because it enables the replacement of predefined operators, such as `+` and `->`, with user-defined procedures. No meta-level information is available in a overloading procedure, however, because operator overloading is not implemented by using the concept of reflection.

The stub generator [2] of remote procedure calls, such as Sun's `rpcgen` [25], has a functionality similar to that of the Open C++ compiler. It reads the description file of a remote procedure and then generates a stub routine, which is a utility routine for calling the remote procedure. Unlike the Open C++ compiler, however, the stub generator does not expose the inside of a stub routine, so the programmer cannot alter the implementation of a stub routine in a well-organized manner. The FOG compiler [7] provides the ability of extending a generated code. It allows to use in C++ a fragmented object (FO), which is a distributed object. In the FOG compiler, the programmer can specify a communication protocol of a remote procedure call.

Meta-level (or reflection) techniques have been applied in various domains and they are still an active area of research. CLOS MOP [10] is the first try to apply the meta-level techniques to a practical language. It provides an extensible implementation of CLOS [22]: all specifications of CLOS are modifiable. The mechanism for method lookup, for example, is extensible by a metaobject. There are several reflective language systems other than CLOS MOP. ABCL/R2 applies a meta-level technique to parallel computation, and RbCl [9] tries to minimize the run-time kernel that is not extensible. AL1/D [18] provides multiple abstract models for each aspect of the implementation, and this is effective when many aspects of the implementation are exposed. The programmer can alter each aspect independently, without considering other aspects.

Meta-level techniques are also beginning to be used for commercial systems. The Meta-Information-Protocol (MIP) [3] used in some commercial systems, is a mechanism for accessing the type information of a C++ object at run time. It represents type information by a metaobject so that typesafe downcast is available in C++. Because a metaobject in the MIP exposes internal information but a change of the metaobject does not influence behavior of an object, the overheads of the MIP is obviously small with respect to execution speed compared with Open C++. Meta-level techniques are also used for developing systems other than languages, such as an operating system and a window system. Apertos [27] is an operating system completely based on a meta-level technique, and Silica [19] is a window system with which the programmer can alter how the system draws an image on a window, how the relationship of windows is maintained, and so on. The Choices operating system uses a meta-level technique to implement its kernel and subsystems [14]. Using macros and programmer conventions, Choices exploits a meta-level technique within the confines of plain C++.

Some researchers try to reduce the cost associated with having the meta level. CLOS MOP, for example, has no costs beyond these of plain CLOS. This is achieved by careful protocol design and by implementation devices in which, for example, calls to the meta-level functions are partially evaluated. Because the execution mechanism of CLOS has inherent complexity and costs, the cost due to the meta level can be recovered by those techniques. On the other hand, C++ is designed so that the program is directly translated into efficient assembly code. The C++ method call, for example, is compiled into a few machine instructions. The techniques used for CLOS MOP are therefore insufficient to implement Open C++ MOP without overhead.

Anibus [20] and Intrigue [12] support "compile-time" MOPs to reduce the cost due to the meta level. They are Lisp compilers that are extensible according to MOP. The "compile-time" MOPs modify the compilers to compile a program in a different scheme. Because a meta code replaces an internal code of the compilers instead of a compiled code, this approach, like that of CLOS MOP, does not generate overheads. In this approach, however, meta code must describe not how an object behaves, but how the compiler generates compiled code that makes an object behave according to the programmer's intention. Although this approach has no overhead, its meta code is less straightforward than those in CLOS MOP and Open C++.

7 Conclusion

This paper described Open C++ in order to demonstrate a methodology for designing extensible languages for distributed computing. Open C++ is designed on the basis of an object-oriented meta-level (or reflection) technique so that the implementation of a method call is made open-ended. The programmer can alter the implementation of a method call according to a simple metaobject protocol (MOP), and obtain on top of Open C++ a new language functionality for distributed computing. Open C++ MOP is made so simple and easily understandable that programmers who are not familiar with the meta system can implement a new functionality effortlessly on top of Open C++. The MOP exposes the implementation of a method call with some abstraction. Open C++ also provides *Object Communities*, which is a framework that facilitates meta-level programming for implementing a functionality for distributed computing.

Open C++ clearly separates distributed computation from the other computation that is more substantial to the programmer. Computation concerning communication and synchronization notions appears only at the meta level, and need not be considered by the programmer writing a program at the base level. This feature of Open C++ makes a program more understandable and easier to describe.

The overhead associated with Open C++ MOP is negligible when Open C++ is used for distributed computing, since even though it is not small, it is negligible in comparison with network latency time. How much performance the system using the MOP must achieve depends on the operations controlled by the MOP. Although meta-level techniques are still difficult to implement efficiently, they are already applicable to practical programming if the domain is selected properly.

Unlike CLOS MOP, Open C++ introduces a meta-level technique into a compiler-based language. Because Open C++ must bridge an abstraction gap between C++ and an assembly language, its design considered implementation issues that the design of CLOS MOP did not. It restricts the extensible part of the language specifications in order to reduce the cost associated with the meta level. The entities that the MOP reifies are only those necessary for distributed computing. To apply Open C++ in application domains such as parallel computing as well as distributed computing, however, the overhead due to extensibility needs to be further reduced.

Acknowledgments

We thank Satoshi Matsuoka for his suggestions on clarifying and organizing this work. We also thank Gregor Kiczales, Hidehiko Masuhara, and Frank Buschmann for their helpful comments on earlier drafts of this paper.

References

[1] Bal, H. E., M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language For Parallel Programming of Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 190–205, 1992.

[2] Birrell, A. D. and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[3] 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.

[4] Ellis, C., S. Gibbs, and G. Rein, "Groupware –Some Issues and Experiences," *Commun. of the ACM*, vol. 34, no. 1, pp. 38–58, 1991.

[5] Gehani, N. and W. Roome, "Concurrent C," *Software–Practice and Experience*, vol. 16, no. 9, pp. 821–844, 1986.

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

[7] Gourhant, Y. and M. Shapiro, "FOG/C++: a Fragmented-Object Generator," in *Proc. of USENIX C++ Conference*, pp. 63–74, 1990.

[8] Herlihy, M. and B. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Trans. Prog. Lang. Syst.*, vol. 4, no. 4, pp. 527–551, 1982.

[9] Ichisugi, Y., S. Matsuoka, and A. Yonezawa, "RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 24–35, 1992.

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

[11] Kiczales, G. and J. Lamping, "Issues in the Design and Specification of Class Libraries," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 435–451, 1992.

[12] 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.

[13] Li, K., *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale Univ., 1986.

[14] Madany, P., P. Kougiouris, N. Islam, and R. H. Campbell, "Practical Examples of Reification and Reflection in C++," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 76–81, 1992.

[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] Matsuoka, S., 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*, no. 512 in LNCS, pp. 231–250, Springer-Verlag, 1991.

[18] 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.

[19] Rao, R., "Implementational Reflection in Silica," in *Proc. of European Conf. on Object-Oriented Programming '91*, no. 512 in LNCS, pp. 251–267, Springer-Verlag, 1991.

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

[21] Smith, B. C., "Reflection and Semantics in Lisp," in *Proc. of ACM Symp. on Principles of Programming Languages*, pp. 23–35, 1984.

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

[23] Stroustrup, B., *The C++ Programming Language*. Addison-Wesley, 2nd ed., 1991.

[24] Stumm, M. and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, vol. 23, no. 5, pp. 54–64, 1990.

[25] Sun Microsystems, *Network Programming Guide*. Sun Microsystems, Inc., 1990.

[26] U.S. Dept. of Defense, *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, 1983.

[27] 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.

[28] Yokote, Y. and M. Tokoro, "The Design and Implementation of ConcurrentSmalltalk," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 331–340, 1986.