A Reflective Architecture for an Object-Oriented Distributed Operating System

Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro*

Sony Computer Science Laboratory Inc.
Takanawa *Muse* Building,
3-14-13 Higashi-gotanda, Shinagawa-ku, Tokyo, 141
JAPAN

ABSTRACT

This paper proposes the Muse object model which provides a reflective architecture for an object-oriented distributed operating system. The model makes a clear distinction between meta hierarchy and class hierarchy. It provides an inheritance mechanism for programming/compile-time facility and a delegation mechanism for run-time facility. It also helps to manage physical resources which are indispensable in describing operating systems. The paper also describes the structure of the Muse operating system which is being implemented based on the Muse object model.

1 INTRODUCTION

Several distributed operating systems have been designed and implemented including V-system[Cheriton 88], Amoeba[Mullender 85], Clouds[Dasgupta 86], Mach[Young 87], Choices[Campbell 87], Sprite[Ousterhout 87], and DASH[Anderson 88]. These systems have some common features: they are open-ended, composed of collections of processes which can be created at a low cost, and the kernels of these systems are implemented as small as possible. However, these systems have not been considered from the programmer's point of view: the system cannot manage all the resources in it uniformly, and cannot always satisfy the expanding requirements from various kinds of users. There is a large gap between the abstraction provided by the operating system and that seen by users. We claim that an operating system should provide high-level abstraction to users, since it can be viewed as a virtual machine and should be considered as a programming environment.

Smalltalk-80[Goldberg 83] can be considered as an implementation of an operating system. It incorporates the notion of objects and inheritance so that it achieves code reuse. It provides memory management and a scheduler. However, it lacks many features usually demanded of an operating system such as files, multilingual support, multiprocessing, and distributed computing.

Muse is a distributed operating system based on an object-oriented paradigm with a reflective architecture. One of the goals of Muse is to provide a *self-advancing* feature

*also with Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223 JAPAN

so that the operating system can be optimized to fit the needs of applications. An optimized operating system is implemented by *meta-objects* in Muse. Meta-objects can be viewed as *a definition of the computation* of objects, or a *virtual machine*. The notion of *meta-meta-objects* is introduced to define computation of meta-objects. A meta-meta-object can be viewed as a kernel in conventional systems.

This paper focuses on proposing the object model with a reflective architecture called the Muse object model and the structure of the Muse operating system which is being implemented based on the model, while other issues on reliability, availability, security, real-time processing, and so on will be described in future papers. In Section 2, the motivation and goals of the Muse operating system are described. Section 3 proposes the Muse object model which provides a reflective architecture for a distributed operating system. In Section 4, the hierarchy of the Muse operating system based on the Muse object model is described in terms of the meta hierarchy, the class system, communication, scheduling, and storage management. In Section 5, the plan and current status of the Muse project are described. Section 6 concludes this paper.

2 MOTIVATION AND GOALS

This section first describes the motivation of investigating a new operating system called Muse. Then several goals of Muse are discussed. Finally, the Muse approach for our goals is presented.

2.1 Motivation

Several distributed operating systems have been designed and implemented. They provide several novel features to users. However, they cannot manage all their resources uniformly, and cannot always satisfy the expanding requirements from various kinds of users. We claim that an operating system should provide a high-level of abstraction to users, since it can be viewed as a virtual machine and should be considered as a programming environment.

Advancing hardware technology leads us to a new world where an ultra-personal workstation is put on your desk which has a dozen CPUs exceeding 100MIPS speeds, physical memory exceeding Giga-byte capacity, and communication methods exceeding Giga-bps speeds. An operating system has to manage the above hardware efficiently. There are several issues which arise in managing the hardware such as how to utilize a dozen CPUs efficiently, whether the virtual storage based on paging strategy for the large amount of physical memory is adequate or not, and how to reduce the overhead in communication modules for high speed communication media.

2.2 Goals

The following issues have to be considered in creating a new operating system:

1. How can the system be created in the user's point of view?

2. How is computation in ultra large distributed systems (ULDS) accommodated?

Considering issue 1, we establish the following goals to design the model of the system as well as the system itself.

1. The system should present a uniform perspective to users. Existing operating systems provide multiple perspectives to users. In early UNIX[1], for example, there are two abstractions in the system: a file and a process. A file is a static, abstract entity representing all of the resources managed by the system. A process is a dynamic entity by which a file is accessed. A file can be used for communication between processes. Recent UNIX, however, provides several abstractions such as message passing and shared memories in parallel with those two abstractions.

2. The system should be self-advancing. Various applications have wide varieties of requirements. Some applications require a very large virtual address space which uses their own paging strategy. Some applications require real-time properties. To meet these requirements, the system should be self-advancing: several parameters of the system will be dynamically changed in accordance with the execution of applications to improve the performance of the system. Some functionalities of the components will be replaced for better performance of the system.

3. The system should provide building blocks for future operating system development. One well-used feature of UNIX has been availability for experiments on operating systems. New ideas are examined by incorporating modules into the sources of UNIX. Under UNIX, however, one cannot examine the new idea in real usage: the system must be rebooted to load a new module which implements the new idea.

4. The system should be multilingual. The property of self-advancement and a uniform perspective of the system can easily be implemented by using a single programming language as in Cedar[Swinehart 86]. But, to meet various kinds of requirements from programmers, the operating system should support multiple programming languages.

While a detailed discussion of issue 2 (computation in ULDS) is beyond the scope of this paper and will be discussed in the future papers, a brief description follows:

- The system should be available on a very large number of nodes. A network of computers will be spread widely on the planet earth, and they must be connected organically. Some lap-top machines move readily. Thus the system should provide transparent services in the large by which users can use network resources transparently even though the resource is topographically far away and by which programmers can readily construct distributed programs.
- The system should be available in a heterogeneous environment. An ULDS will be composed of heterogeneous hardware which the operating system must accommodate.
- The system should provide cooperability and autonomy. Cooperability is orthogonal notion with the autonomy. Cooperability can be easily accomplished in small distributed systems. In an ULDS, however, some cooperability will be restricted due to the hardware heterogeneity and the wide range of communication delays,

[1] UNIX is a trademark of AT&T Bell Laboratories.

so that autonomy should be provided along with cooperability. Autonomy can also increase the availability of the system.

- The system should be reliable. In an ULDS, there are serious problems with reliability. If a database server, for example, has obsolete data due to hardware crash and simultaneous access to data, users cannot obtain correct data.

- The system should be available in spite of the hardware crash and network partitioning. In an ULDS, we must address the availability issues. Some users may get into trouble such that they cannot continue their jobs due to the server crashing or the network partitioning. The probability of this situation increases with the scale of the system.

- The system should be secure. There are several administrative domains in an ULDS. Some domains may have trivial security issues: for example, a small domain. Other domains may require strict security: for example, military information. To protect the system against unpredictable malicious access of other systems, the system should provide secure accessing methods to objects.

- The system should provide real-time communication. In order to realize the distributed scheduling of objects and in order to present moderate response time to users for the ideal user interface, applications must obtain the desired information in specified time periods.

- The kernel of a system should be small and easily portable to various hardware. Since hardware technologies have been rapidly advancing, new computers with more powerful processing ability are appearing in dramatically short intervals.

2.3 The Muse Approach

To achieve the above claims, we have designed the Muse object model and the Muse operating system to have the following features:

1. uniform perspective. Muse presents a uniform perspective of the system, an object, to users. An object is a fundamental entity in Muse. All resources in Muse are abstracted as objects. There are no distinctions between processes and files, nor between active objects and passive objects. Interaction between objects is accomplished by message passing.

2. self-advancement. Basically, self-advancement is similar to open-endedness in the sense that users can modify the system dynamically to meet their requirements. The notion of self-advancement, however, is more advanced and powerful than the notion of open-endedness. The system can improve its behavior dynamically by itself. The self-advancement of the system in Muse is realized by a reflective architecture[Maes 87] which has two kinds of objects, objects and meta-objects, which are causally connected. In the field of programming languages, several reflective object-oriented programming languages have been investigated such as 3-KRS[Maes 87], ObjVlisp[Cointe 87], and ABCL/R[Watanabe 88]. We found some problems in these languages when we tried to apply their concepts to building operating systems, which include:

- How is the local storage of an object managed? Who provides the storage for an object while the representation is provided by the meta-object? The above languages are implemented in Lisp and an object is represented in a list structure of the meta-object. The meta-object is also implemented in Lisp. Thus, there are no definitions of the storage which holds the list structure.
- How is a communication scheme such as message passing, remote procedure calls, and streams described?

These issues are not crucial in programming languages. The programming languages are implemented on the operating system which is the final meta-object of the system. The operating system supports the virtual environment of the system. Therefore, the finite physical environment needs not to be considered when implementing the languages. We must, however, resolve these issues when designing an operating system. The Muse object model presented in the next section answers these questions.

3. building blocks. Meta-objects define computation of an object. New ideas can be examined by adding new meta-objects without affecting the real usage. Later, the new features can be dynamically integrated into the existing system. The inheritance and delegation mechanisms provided by the class system of Muse encourage experimental use of systems which include new ideas.

4. multilingual. All kinds of programs are constructed in the Muse class system. For example, a C program which corresponds to a load module of UNIX is constructed in the class system. A C program is encapsulated into object-oriented flavor. Programs which are described by languages with their own class system can be integrated into the class system of Muse. Compatibility with classes which are described in different languages is maintained by the Muse class system. The detailed discussion of the class system will be presented in another paper.

3 THE MUSE OBJECT MODEL

This section proposes the Muse object model which provides a reflective architecture for an object-oriented distributed operating system. First, we define an object with the meta hierarchy. Then, a class which can be viewed as a static template of an object is introduced. Finally, a reflective architecture in our model is presented.

3.1 What Is an Object?

An object can be viewed as a small computer which is dynamically created and destroyed, and which has local storage for computation. In this respect, who can create and destroy an object? Who can define communication between objects? Who can manage the local storage of an object? That is, who can define computation of an object? We introduce meta-objects to answer these questions.

As depicted in Figure 1, objects are located on three levels: *object level*, *meta-object level*, and *meta-meta-object level*. Application programs are collections of objects on the object level. Computation of these objects is defined by another object which is on the meta-object level, called a *meta-object*. A meta-object can define computation of two or more objects when they are compatible with each other. A meta-object can be viewed as a virtual machine which defines computation of an object.

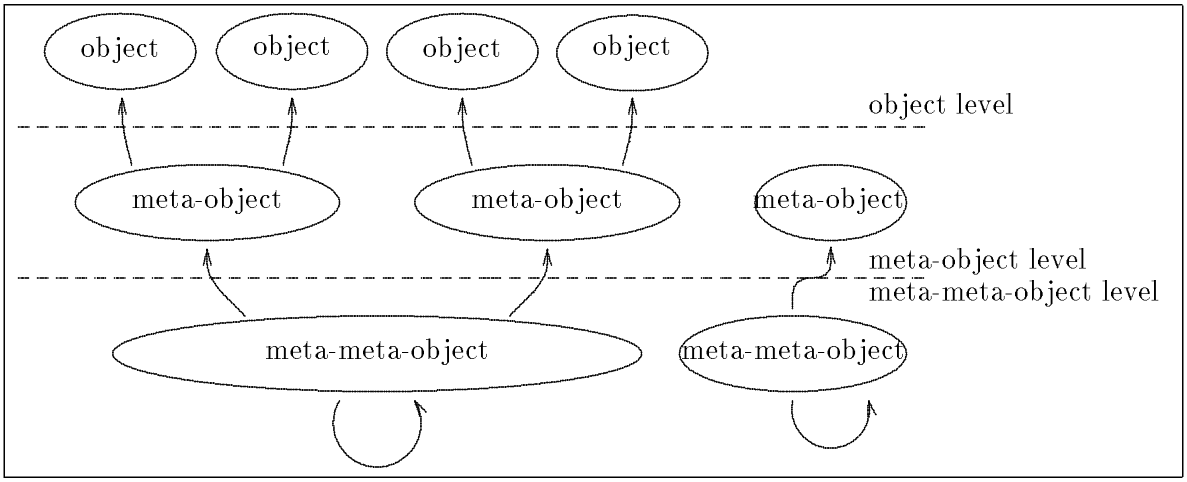

Figure 1: The Meta Hierarchy

A meta-object is also an object, so that it is defined by another object, called a *meta-meta-object*. Computation of a meta-object is simulated by the meta-meta-object which is on the meta-meta-object level. In fact, meta-meta-objects can obscure the hardware heterogeneity and give the common platform to meta-objects.

The relationship between an object and its meta-object from the object level point of view is the same as the relationship between a meta-object and its meta-meta-object from the meta-object level point of view. This tower of levels is conceptually continued infinitely. Since infinite towers are not practical in the design of an operating system, however, the Muse object model collapses the infinite tower at the meta-meta-object level.

In the Muse object model, an object is created by a meta-object when a request is sent to the class. The following expression is an example of creating a new object:

$$anObject \leftarrow AClass\ new.$$

A *new* request is sent to class *AClass* to create a new object. *AClass* sends request *with: self* to the meta-object which can be determined by using the *meta:* field of its class (described later). A new object is assigned to variable *anObject*. A meta-object can be created in the same way that an object is created: a class sends request *with:* to the meta-meta-object. A meta-object has the following properties:

- Execution of a method in the class of the object is conceptually defined by the meta-object.
- The semantics of communication between objects is defined by the meta-object.
- The meta-object is responsible for providing an object with local storage having desired functionality, for example, stable storage property.

In the distributed environment, one of the important issues is to distribute meta-objects. We assume that a transparent name space is provided: an object has a name and can be accessed independently of the location where the object resides. For example, when an object migrates to another host, the meta-object associated with the object has either to migrate or to clone itself on the other host. Otherwise, the migration of the object is prohibited by the meta-object. The decision of what to do when an object migrates is the responsibility of the meta-object. Another example is to provide storage which has the atomic property: modification to the storage is not visible from outside of the object. The meta-object has functions to manage the storage atomically.

3.2 What Is a Class?

This subsection describes a relationship between meta-objects and classes. In our model, the notion of a class is introduced for the following reasons:

- The code of methods defined in the class can be shared between objects (i.e. instances) of the class.
- A class can be viewed as a template which is used as a hint for creating a new object and converting the representation of an object to be suitable for the target host upon migration.
- A class is defined as a static and immutable entity, so that it is easy to manage duplication of a class in the distributed environment.
- Differential programming is encouraged by using superclass/subclass relationship.

The separation of meta-objects and classes leads us to define a meta-object as a virtual machine and to define a meta-object for an object without regards to the class of an object. A class can be defined as a subclass of other classes, either by single or multiple inheritance. The class hierarchy is independent from the meta hierarchy which is the relationship between the object level, the meta-object level, and the meta-meta-object level. Since a class can be viewed as a static template, it is created on the object level and is independent of the level where an object is created. That is, the class hierarchy is available for when a program is described as a class.

A class can inherit a property from its superclasses. The inheritance is static and is a programming and compile-time facility. We introduce a delegation mechanism [Lieberman 86] to obtain properties of other objects dynamically. Delegation is a run-time facility to utilize the function of other objects. To keep the modularity of an object, the delegation mechanism is somewhat restricted: accessing local storage by a method is restricted to the local storage of the object which the message is delegated to.

As depicted in Figure 2, a class is also an object which is created by the meta-object *Metaclass*. Meta-object *Metaclass* defines computation of all classes: a class is created by meta-object *Metaclass* when a request is sent to class *ClassTemplate*. The class of each class is a class which is named *ClassTemplate*, so that class *ClassTemplate* is the root of the *instance-of* link. The following expression is an example of creating a new class:

Point←ClassTemplate name: #Point subclassOf: Object in: anEnv.

After reception of request *name:subclassOf:in:*, *ClassTemplate* sends request *with:* to *Metaclass* to create a new class which is defined as a subclass of *Object*, and name it to *Point*. To speak in easy terms, classes correspond to cookie-cutters, *ClassTemplate* is a cookie-cutter-die, and *Metaclass* is a cookie-cutter-maker.

Some objects may be restricted to be available on the particular meta-object. The objects have to be imposed to be created by the specific meta-object. The Muse class system maintains information on the dependency of objects to meta-objects, and helps users to be oblivious to this kind of dependency.

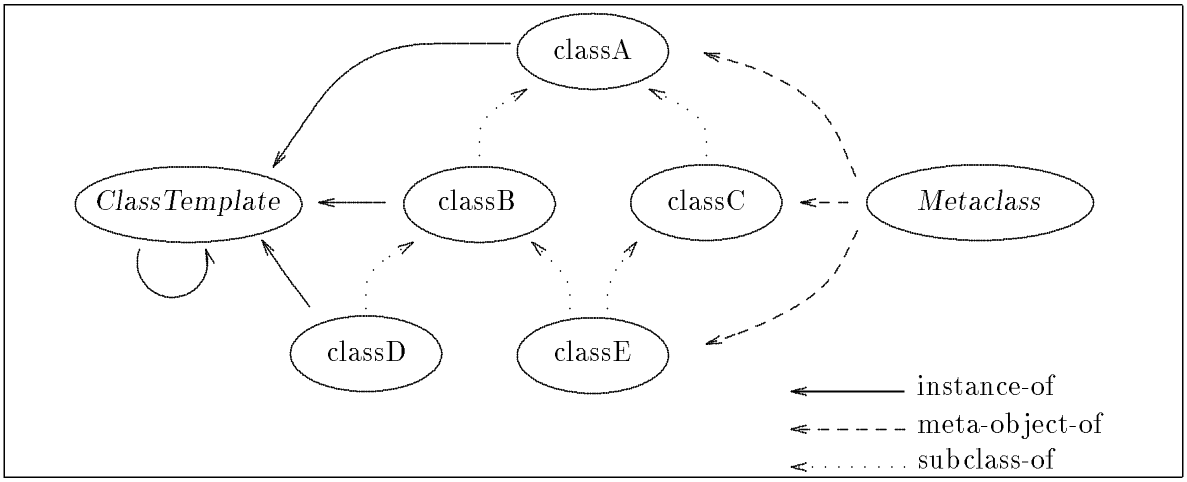

Figure 2: The Class System

3.3 A Reflective Architecture in the Muse Object Model

The existence of objects and meta-objects stems from the macroscopic view of the system. An object contains a meta-object part viewed microscopically. This kind of relationship between objects and meta-objects can be accomplished by using a reflective architecture. In a reflective architecture, an object is interpreted by the meta-object and computation of the meta-object can be influenced by the object. To achieve this, an object has to know about its meta-object and a meta-object has to know about its objects. Since every object has a name, "know about" means an object knows the name of its meta-object by a variable denoting it and a meta-object knows the name of the objects by object descriptors (i.e. variables) denoting them.

The reflective computation of an object by the meta-object is accomplished either implicitly or explicitly. Inter-object communication and paging facilities for the local storage of an object are examples of implicit reflection, while sending a message to its meta-object (i.e. a system call) is an example of explicit reflection. Computation of an object by the meta-object can be reflected as follows:

- Since the meta-object knows the status of the objects, it can change its own behavior to manage the objects efficiently,
- The meta-object can change a meta-object which has other functions to delegate requests upon reception, and
- An object can migrate to a new meta-object which is compatible with the old one.

In any case, a meta-object should know the way to change computation of an object.

Since a meta-object knows the status of its associated objects, computation of an object can be reflected by the meta-object at any time. On the other hand, since the status of the meta-object cannot be known to objects at any time because of protection reasons, an object must issue a request to the meta-object to obtain status information.

To summarize the section, we give a figure (Figure 3) which combines meta hierarchy and class hierarchy. The relationship between objects and a server (or a daemon) in other systems such as V-system, Mach, and Amoeba seems to be similar to the relationship between objects and a meta-object in Muse. However, the followings are different:

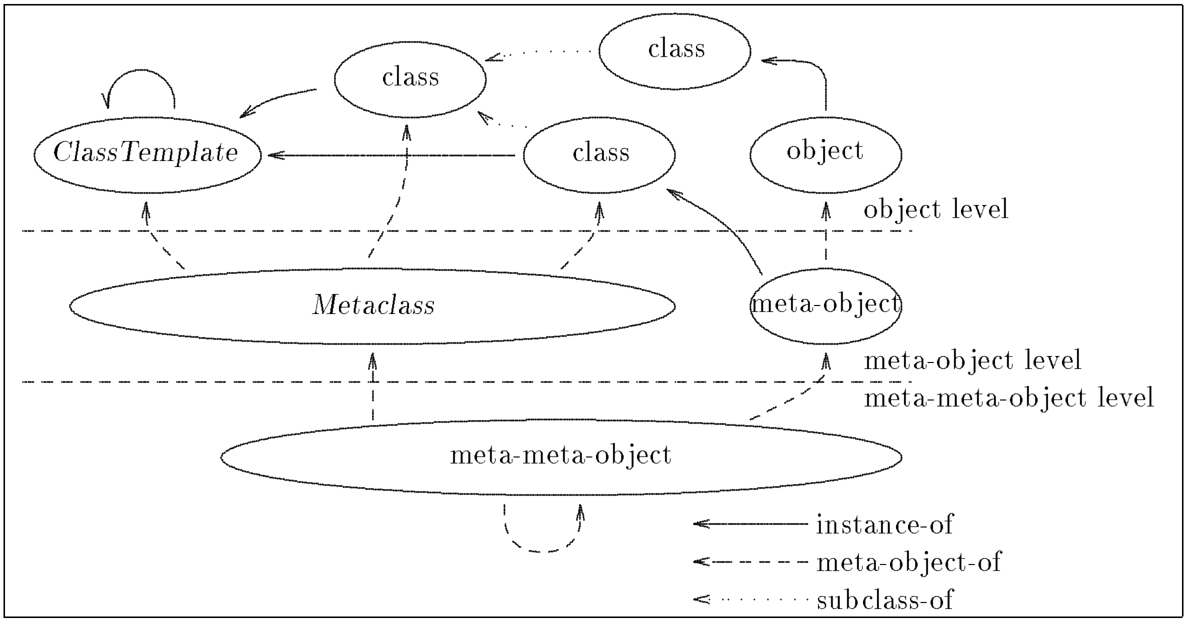

Figure 3: The Summary of the Model

- An object in Muse is an active entity in that one and only one activity is conceptually always associated with it. In contrast, objects in other systems are passive, and two or more threads can be activated on a passive object. Therefore one has to explicitly describe synchronization among activities on those systems.
- Objects and their meta-objects are causally connected, while objects and the server process are not. A meta-object can influence computation of objects implicitly.
- Since objects can be accessed only by the server process, applications have to issue explicit requests to use the functions provided by the server. Applications in Muse can share in the benefit of the meta-object implicitly.
- A meta-object can define computation of its objects. For example, the semantics of the send operation in objects is defined in the meta-object.

4 THE STRUCTURE OF THE MUSE OPERATING SYSTEM

The Muse object model presented in the previous section will be implemented in the Muse operating system. This section describes the structure of the system in terms of primitives to implement a reflective architecture, the meta hierarchy, the class system, scheduling of objects, communication, storage management, and bootstrapping of Muse.

4.1 Primitives

The following two alternatives can be considered in implementing a reflective architecture:

1. A meta-object is defined as a virtual machine as in Smalltalk-80. Computation of an object is defined in such a way that the meta-object simulates intermediate codes such as bytecodes. To communicate with another object is to execute a *send* intermediate code. To access the local storage is to execute intermediate codes *load* and *store*.

2. A meta-object is defined as an implementation of primitive functions such as allocation of local storage and accessing the external devices. Computation of an object is defined in such a way that a physical processor executes native codes (e.g. compiled from sources) and a meta-object accepts requests from objects to execute primitive functions.

While alternative 1 is a straightforward implementation of a reflective architecture and attractive in the heterogeneous environment, we chose alternative 2. This is because in the object-oriented computing model, computation can be reduced to exchanging messages between objects and mutation of objects upon receipt of requests, and can be realized by the following two kinds of primitives: communication primitives and mutation primitives. Since mutation of an object is performed by the underlying hardware, we introduce primitives which help mutation such as forwarding a request when memory fault occurs.

Primitive inter-object communication methods must:

- be atomic operations. They cannot create new objects in order to communicate between objects. If a new object were created, we would have to introduce a new primitive communication method to communicate with that object. They also cannot invoke the scheduler since the scheduler would be recursively invoked when the communication methods were used in the scheduler.
- provide high speed communication, since they are heavily used in implementing higher level functions.

Thus, we introduce the following three primitive inter-object communication methods which are based on synchronous and asynchronous request send with implicit request reception:

1. Invoking a method of an object. A caller object sends a request to a called object and waits for a result from the called object. The called object starts to execute a method if it is dormant. Otherwise, the request is put into the queue of the called object. A result is returned to the caller object when the called object executes the method.
2. Sending a request. A caller object sends a request to a called object and continues to run. A called object starts to execute a method if it is dormant. Otherwise, the request is put into the queue of the called object. The caller object does not receive any result later. A result returned from the called object is discarded.
3. Sending a result to the caller object. A called object returns a result and can continue to run.

4.2 The Meta Hierarchy

As depicted in Figure 4, the Muse operating system uses the three-level structure: object level, meta-object level, and meta-meta-object level.

4.2.1 The object level

This level can be thought as the user level in conventional meaning. Application programs are collections of objects on this level. Objects on this level are interpreted by the meta-objects which reside on the next lower layer: many primitives which are issued by objects on this level are defined on the meta-object level.

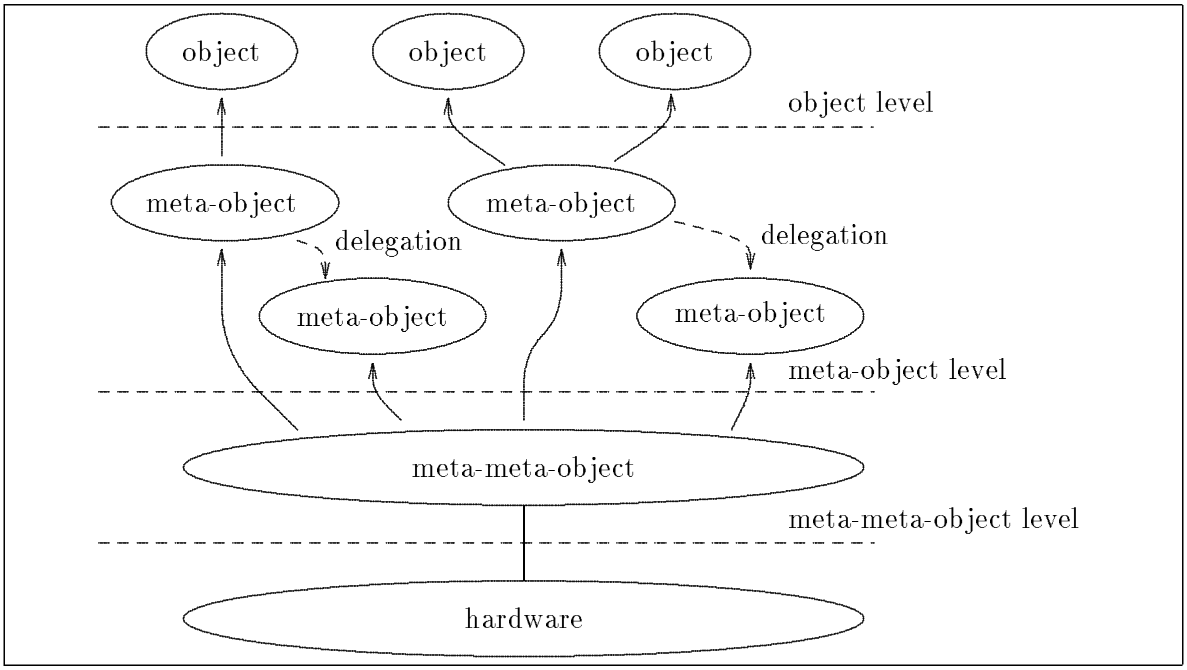

Figure 4: The Structure of Muse

4.2.2 The meta-object level

The meta-object level provides a virtual machine environment to the objects which reside on the object level. From the viewpoint of the object level, one virtual machine is realized by one meta-object. Since a meta-object can obtain properties by delegation, a virtual machine may consist of some meta-objects. Objects on the object level can even define new virtual machines by modifying existing meta-objects or by adding new meta-objects.

In this level there are two types of meta-objects. One includes meta-objects which interpret/manage the execution of the objects on the object level. The other includes meta-objects which provide various functions to other meta-objects by delegation. Examples are a default scheduler, a default pager, communication protocols handling, transaction processing, etc. The default scheduler provides default scheduling facilities to meta-objects which do not implement scheduling facilities by themselves. A default pager provides default paging facilities to meta-objects which do not implement paging facilities.

4.2.3 The meta-meta-object level

The meta-meta-object level is a memory resident part of the Muse operating system. One meta-meta-object exists per CPU and constitutes a kernel in conventional meaning. A meta-meta-object can provide a common platform to meta-objects by hiding the hardware heterogeneity. Meta-meta-objects are divided into two types: machine dependent and machine independent. The machine dependent one has the following functions:

- MMU manipulation, such as reading and writing page table entries,
- Device management and drivers, such as activating and deactivating devices, handling hardware interrupts, and so on.

The Muse operating system can be ported by modifying only the machine dependent part for target machines. The machine independent part has the following functions:

- Primitives described in Subsection 4.1,
- Trap handling, such as handling of page fault traps, and so on.

4.3 The Class System

A class is viewed as a static and immutable entity in Muse. Class version control is performed by the programming environment. In Emerald[Jul 88], a template is used for keeping the representation of the storage of the object. XDR[Sun 86] and Courier[Xerox 81] have intermediate representations for data conversion between heterogeneous hardware. In Muse, a class holds the following information:

name: Name of the class is specified.

superclasses: List of names of superclasses is specified.

meta: Name of the meta-object or name of the class of a meta-object. This field is used as a hint for deciding which meta-object is responsible of creating a new object and defining computation of a new object.

template: A template for an object which is created by the class is described. A template is used for creating an object, for determining the format of storage area, and for migrating an object from/to another host. When an object migrates to another host, the internal representation of an object is converted to one which is suitable for the target hardware.

virtual code: Intermediate codes compiled from the source may be stored. The codes can be used for migrating an object: if the target host is heterogeneous from the original host, the virtual code is dynamically translated to the target binary code. This field can be omitted. The detailed discussion of the virtual code will be described in another paper.

binary code: Hardware dependent binary codes (i.e. methods) compiled from the source text are stored. Methods are mapped into local storage of the object.

source code: Source texts of methods are stored. This field can be omitted for binary distribution of a program.

There are two representations of a class in Muse: independent and collapsed representations. In the independent representation, every class has its own template and methods. When the superclass of a class is changed, the subclasses need not to be recompiled. For efficiency sake, we introduce the collapsed representation. All of the templates and combined methods of the superclasses are collected into the class. That is to say, methods of superclasses are collected into the class itself, and templates are combined. Typically, product quality classes are stored in collapsed representation. When inheritance and delegation are employed at the same time, inheritance is preceded by delegation, since the names of methods which are defined in the superclasses are held in the object.

4.4 Inter-object Communication

Objects on the object level can communicate with each other in a uniform way regardless of whether the target object exists on the same machine or not. Muse provides location independent communication facilities to users. Whether the communication request from an object is local or remote is determined by the meta-object of the object.

4.4.1 Local communication

We can implement higher level inter-object communication methods such as remote procedure calls, streams, message passing, and communication with an ambiguous target by using the primitives described in Subsection 4.1. In the implementation of such higher level inter-object communication, the control flows as depicted in Figure 5 appear. In the figure, solid lines indicate a path of communication, thin dotted lines

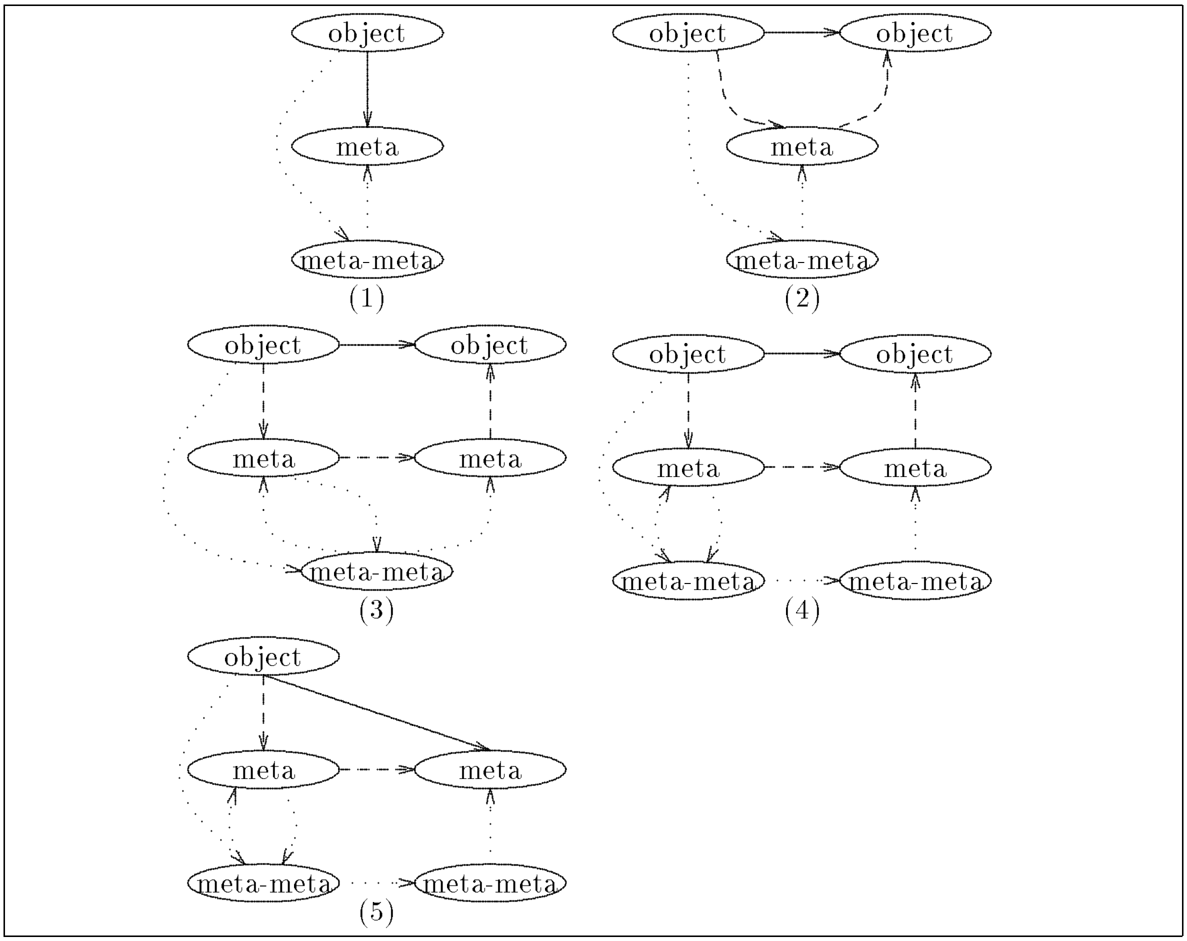

Figure 5: Control Flows of the Muse Communication System

indicate execution path of primitive inter-object communication methods, and thick dotted lines indicate path of communication executed by meta-objects.

(1) Communication between an object and a meta-object of the object. A caller object sends a request to a meta-object associated with the caller object. The caller object executes a primitive inter-object communication method, and the control of execution passes to the meta-meta-object which sends the request to the target meta-object. The meta-object finds out that the request is for itself.

(2) Communication between objects which run on the same meta-object. The meta-object receives a request from the caller object as (1). It finds out that the request is for another object on itself, and it sends the request to the called object.

(3) Communication between objects which run on different meta-objects. The meta-object associated with the caller object receives a request from the caller object as (1), finds out that the request is for another object which exists on another meta-object, and sends the request to that meta-object. The meta-object of the called object sends the request to the called object.

(4) and (5) are described in the following subsection.

4.4.2 Remote communication

If a meta-object recognizes that the target object exists on another host, the meta-object executes network communication methods. On the meta-object level, there are some meta-objects that provide network communication facilities which cover under the session layer protocol such as TCP/IP and VMTP[Cheriton 86]. Users can implement new communication protocols by adding new meta-objects.

In Figure 5, (4) and (5) show the control flows of remote communication.

(4) Communication between objects which run on different meta-objects and meta-meta-objects. A meta-object receives a request from the caller object, finds out that the request is for an object which exists on another meta-meta-object, and sends the request to the meta-object of the called object. The meta-object of the called object sends the request to the called object.

(5) Communication between an object and a meta-object which run on different meta-object and meta-meta-object. A meta-object receives a request from the caller object, finds out that the request is for a meta-object which exists on another meta-meta-object, and sends the request to the meta-object.

4.5 Scheduling of Objects

Muse provides two levels of scheduling: object scheduling and meta-object scheduling.

object scheduling. Objects are scheduled by the meta-object with which objects are associated. Scheduling of objects on the same meta-object is independent of that on other meta-objects. Scheduling facilities are either inherited from other classes at programming/compile-time by inheritance or gotten from other meta-objects at run-time by delegation. Users can define their own scheduling strategies which are suitable for executing the application. Users can dynamically change the scheduling strategies of objects by changing scheduler objects, possibly using the delegation mechanism.

meta-object scheduling. Execution of all meta-objects in the same host are scheduled by the meta-meta-object. Basically a meta-meta-object can give a time quota to a meta-object. A meta-object relinquishes CPU when the time quota is exceeded, when an object waits for events, or when the execution of an object is preempted.

4.6 Storage Management

One of our goals in designing storage management is to efficiently implement fine grain objects.

Object storage management. Storage of objects is managed by their meta-objects. Meta-objects define the properties of storage of an object, such as private or sharable and protected or unprotected. Usage of a virtual address space is also managed by meta-objects. One virtual address space might be occupied by only one object, or by several objects. Paging strategies are inherited either from other classes by inheritance or from other objects by delegation.

Meta-object storage management. Storage of meta-objects is managed by their meta-meta-objects. The following functions have to be implemented in meta-meta-objects since they provide a hardware dependent methods:

- getting the status of a page table entry to decide what pages should be swapped out,
- invalidating the page table entries of the candidates,
- swapping out the candidate pages using the disk driver, and so on.

4.7 Bootstrap of Muse

Before the system begins operation, the meta hierarchy and the class hierarchy must be established as follows:

1. The *Skeleton* object is created by another system such as UNIX. The object *Skeleton* is a special object which can be viewed as both a class and a meta-object. The first time, *Skeleton* is allocated on the meta-meta-object level, so that *Skeleton* becomes a meta-meta-object.
2. *Skeleton* creates the object, meta-object *SmallMetaclass*. Meta-object *SmallMetaclass* creates the object, class *SmallClassTemplate*. The former has a method to create a class. The latter has a template for classes to be created.
3. To create a new class, the message *name:subclassOf:in:* is sent to *SmallClassTemplate*. Since a class is static, the structure of a class is determined at this bootstrap time.
4. Meta-object *SmallMetaclass* and class *SmallClassTemplate* are reincarnated to meta-object *Metaclass* and class *ClassTemplate*, respectively.

The above operation will be accomplished once.

Since Muse supports persistent objects, after the Muse world has been built, the system is booted as follows:

1. A machine dependent boot program loads the meta-meta-object either from local disk storage or from another host, and passes control to the meta-meta-object.
2. The meta-meta-object activates meta-object *Startup* on the meta-object level.
3. Meta-object *Startup* activates some meta-objects which represent the virtual machine environment according to a configuration specification.

5 PROJECT STATUS

The Muse project is a five-year project. The effort consists of five phases:

1. **Basic architecture design:** The basic architecture of Muse is designed, such as basic abstraction of the system, storage architecture, communication mechanisms, protection mechanisms, reliability and availability mechanisms. One of the results of this phase is to present the Muse object model described in this paper.

2. **Prototype implementation:** A prototype kernel of the Muse operating system is being implemented on several SONY NEWS workstations which consist of MC68030 with 4MB real memory and without cache.

3. **Evaluation of the prototype and redesign:** Some distributed applications will be designed and implemented on the prototype Muse system to evaluate the architecture designed at Phase 1. We will redesign the architecture based on the evaluation.

4. **Implementation of Muse:** The Muse system will be implemented on NEWS and other workstations based on the architecture designed at Phase 3.

5. **Distribution of Muse:** The Muse system will be distributed as public domain software.

The project is currently in Phase 2. Before implementing the prototype Muse system, specification of several kernel functionalities are being described in ConcurrentSmalltalk[Yokote 87].

6 CONCLUSION

This paper presented the Muse object model which provides a reflective architecture for an object-oriented distributed operating system called Muse. The object model is not intended as a model for programming languages but as a tool to build distributed operating systems which will be used practically in the near future. The model not only provides a clear and unique view to programmers and users but also helps to efficiently manage physical resources.

Some of the important features of the Muse object model are as follows:

- The model makes a clear distinction between meta hierarchy and class hierarchy. The meta hierarchy is the hierarchy of interpretation, or the hierarchy of virtual machines. The meta hierarchy contributes to building multiple virtual machines. The class hierarchy is the hierarchy used at programming time (and therefore at compile-time). This facilitates differential programming and code reuse.

- The model provides both inheritance and delegation. Inheritance is a compile-time facility as described above, while delegation is a run-time facility. Since inheritance is a static relationship, it enables us to optimize the code for a class. That is, inheritance contributes to efficient operation of meta-objects. On the other hand, delegation is a dynamic relationship which enables us to build a flexible system architecture.

- The model provides a reflective mechanism. This enables us to dynamically change a virtual machine so that the operating system can be self-advancing.

An overview of the Muse operating system was presented based on the Muse object model. A prototype implementation of the Muse operating system is being done so that some experiences in programming the operating system and some results from the prototype implementation will be reflected in the final design and implementation of the Muse operating system.

ACKNOWLEDGMENTS

We give great thanks to Dr. Hideyuki Tokuda of Carnegie-Mellon University and Rik Smoody of Sony Computer Science Laboratory Inc. Several discussions with them were helpful to us in designing the system and deciding on the implementation strategy of the system.

REFERENCES

[Anderson 88] David P. Anderson and Domenico Ferrari. *The DASH Project: An Overview.* Technical Report UCB/CSD 88/405, Computer Science Division (EECS), University of California, Berkeley, February 1988.

[Campbell 87] Roy Campbell, Vincent Russo, and Gray Johnston. The Design of a Multiprocessor Operating System. In *USENIX C++ Conference*, USENIX Association, November 1987.

[Cheriton 86] David R. Cheriton. VMTP: A Transport Protocol for the Next Generation of Communication Systems. In *Proceedings of SIGCOMM 86*, ACM, August 1986.

[Cheriton 88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, Vol.31, No.3, pp.314–333, March 1988.

[Cointe 87] Pierre Cointe. Metaclasses are First Class: the ObjVlisp Model. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1987*, ACM, October 1987.

[Dasgupta 86] Partha Dasgupta. A Probe-Based Monitoring Scheme for an Object-Oriented, Distributed Operating System. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1986*, ACM, September–October 1986.

[Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[Jul 88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, Vol.6, No.1, February 1988.

[Lieberman 86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1986*, ACM, September–October 1986.

[Maes 87] Pattie Maes. *COMPUTATIONAL REFLECTION.* Technical Report TR-87-2, VUB AI-LAB, 1987.

[Mullender 85] Sape J. Mullender. *PRINCIPLES OF DISTRIBUTED OPERATING SYSTEM DESIGN*. Technical Report, Vrije Universiteit, 1985.

[Ousterhout 87] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch. An Overview of the Sprite Project. *;login:*, Vol.12, No.1, pp.13–17, January/February 1987.

[Sun 86] Sun. *External Data Representation Protocol Specification*. Sun Microsystems, Inc., February 1986. Revision B.

[Swinehart 86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, Vol.8, No.4, October 1986.

[Watanabe 88] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1988*, ACM, September 1988.

[Xerox 81] Xerox. *Courier: The Remote Procedure Call Protocol*. Xerox Corporation, December 1981. XSIS 038112.

[Yokote 87] Yasuhiko Yokote and Mario Tokoro. Experience and Evolution of ConcurrentSmalltalk. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1987*, ACM, October 1987.

[Young 87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*. Technical Report CMU-CS-87-155, Department of Computer Science, Carnegie-Mellon University, August 1987.