



The Muse Object Architecture: A New Operating System Structuring Concept

Yasuhiko Yokote[†], Fumio Teraoka[†], Atsushi Mitsuzawa[‡],
Nobuhisa Fujinami[†], and Mario Tokoro^{†‡}

E-mail: {ykt,tera,mituzawa,fnami,mario}@csl.sony.co.jp, {mituzawa,mario}@keio.ac.jp

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

[‡]Department of Computer Science
Keio University
3-14-1 Hiyoshi, Kohoku-ku,
Yokohama, 223 JAPAN

ABSTRACT

A next generation operating system should accommodate an ultra large-scale, open, self-advancing, and distributed environment. This environment is dynamic and versatile in nature. In it, an unlimited number of objects, ranging from fine to coarse-grained, are emerging, vanishing, evolving, and being replaced; computers of various processing capacities are dynamically connected and disconnected to networks; systems can optimize object execution by automatically detecting the user's and/or programmer's requirements. In this paper, we investigate several structuring concepts in existing operating systems. These structuring concepts include layered structuring, hierarchical structuring, policy/mechanism separation, collective kernel structuring, object-based structuring, open operating system structuring, virtual machine structuring, and proxy structuring.

We adjudge that these structuring concepts are not sufficient to support the environment described above because they lack the abilities to handle dynamic system behavior and transparency and to control dependency. Thus, we propose a new operating system structuring concept which we call the Muse object architecture. In this architecture, an object is a single abstraction of a computing resource in the system. Each object has a group of meta-objects which provide an execution environment. These meta-objects constitute a meta-space which is represented within the meta-hierarchy. An object is causally connected with its meta-objects: the internal structure of an object is represented by meta-objects; an object can make a request of meta-computing; a meta-object can reflect the results of meta-computing to its object. We discuss object/meta-object separation, the meta-hierarchy, and reflective computing of the architecture. We then compare the Muse object architecture with the existing structuring concepts.

We also demonstrate that the Muse object architecture is suitable for structuring future operating systems by presenting several system services of the Muse operating system such as class systems, a real-time scheduler with hierarchical policies, and free-grained objects management. Class systems facilitate programming by several classes of programming languages. A real-time scheduler with hierarchical policies can meet various types of real-time constraints presented by applications. Free-grained objects management can suit the object granularity to the application, so that an object is efficiently managed according to its granularity. Finally, we present the implementation of the Muse operating system which is designed based on the Muse object architecture. Version 0.3 of the Muse kernel is running on the MC68030 based Sony NEWS workstations.

1 Introduction

New hardware and software technologies increase the expectations and demands of computer users. With respect to operating systems and the applications they support, there are several specific expectations that the next generation of operating systems should meet:

- **Ultra-large scale.** Accurately predicting the number of entities — workstations, portable and mobile computers, persistent objects, devices, activities, address spaces, etc. — to be manipulated at system configuration time is impossible. Therefore, the number of entities that a system can manage should not be limited. For example, limiting a system to 2^{32} entities may force a programmer into unnatural programming styles that reduce the number of manipulated entities.
- **Highly distributed.** As networking becomes even faster and internationalization continues, the need for geographically distributed operating systems will increase. In contrast, most existing operating systems are restricted, often to a single local area network. In addition, since users and programmers are increasingly mobile, it is critical that their static and dynamic computing environment be independent of the location of the computer they are using. In addition, users and programmers should be as protected as possible from changes to network topology, file system configurations, new versions of tools, etc.
- **Open.** To remain useful, operating systems, like all software, must evolve. It must be relatively easy to create new operating system functions, integrate new services, and add new processors. Since a system provides no distinction between the levels of protection for systems and users, we can easily use well-designed system objects.
- **Rich user and programmer interaction.** Users should be largely unconstrained with respect to how they interact with the system. For instance, they should not be constrained to a particular style of input/output device or to a particular user interface style (such as command-based or menu-based). Similarly, programmers should be allowed to manipulate objects in the system naturally and efficiently.
- **Self-advancing.** Just as the number of entities needed is difficult to predict, the kinds of services, the resource capacity for computing, and the desired communication path are also difficult to predict prior to execution. Thus a system should evolve to provide an optimal execution environment in spite of such unpredictability.

This paper introduces an operating system called Muse, whose design is intended to address these and other basic requirements for contemporary operating systems such as real-time support, fault-tolerance, reliability, compatibility with existing software (i.e., UNIX¹), and heterogeneity (i.e., offering gateways to different type of computers and operating systems). This paper details Muse's approach to simultaneously solving these problems. The current version of Muse, which is more limited, is described later in the paper.

Muse, an object-oriented distributed operating system, can be characterized by two notable features.

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

- **Meta-objects.** Each object resides in the context of a collection of meta-objects to handle dynamic system behavior, to deal with transparency, to reduce constraints about distribution, etc. These meta-objects define the environment for computation and constitutes a meta-space. By adapting the meta-environment, the underlying object or objects can be protected from many changes to the operating system environment.
- **Reflection.** To provide an open and self-advancing environment, Muse provides reflective computing that presents facilities for self-modifying an object with its environment and for inspecting the meta-computing environment of an object. In addition, to define meta-objects as objects, we introduce a meta-hierarchy composed of meta-spaces. Thus the relationship between an object and its meta-space is relative.

This paper defines the object architecture which provides the above features. We call this architecture the Muse object architecture. The ability of Muse to satisfy the needed requirements comes not from meta-objects and reflection alone, but from the kinds of structures we can build using them together. For example, while most conventional structuring concepts — including layering, hierarchical structuring, collective kernel structuring, proxy structuring, and others — do not make it easy or possible to handle objects of many different grain-sizes efficiently and naturally, Muse can do so by carefully combining meta-objects and reflection. In addition, the structuring techniques of Muse allow us to more easily take advantage of advancing network technology such as wide-area networks, etc.

In this paper, we introduce the Muse object architecture and discuss the structuring concept for such operating systems. Section 2 discusses several structuring concepts of existing operating systems in terms of their pluses and minuses in fulfilling the above requirements. In Section 3, the Muse object architecture is described including the definitions of objects, meta-objects, and reflective computing. Section 4 gives example applications designed and implemented using the Muse object architecture. In Section 5, the prototype implementation of Muse is described briefly. Section 6 summarizes this paper.

2 Operating System Structuring Concepts

This section argues that existing operating system structuring concepts are not sufficient for developing the kinds of systems motivated in the previous section. They include layered structuring, hierarchical structuring, policy/mechanism separation, collective kernel structuring, object-based structuring, open operating system structuring, virtual machine structuring, and proxy structuring. Finally, we summarize this section.

2.1 Layered structuring

Some operating systems are designed as sequences of layers. Each layer of the system provides a set of functions to the layer above and is implemented in terms of functions provided by the layer below. Each layer contains concurrent processes that implement the layer's functions. Dividing the system functions into layers makes it easier to conceptualize (and thus maintain) the system, since a process belonging to one layer cannot use a process belonging to higher layers. Specifically,

a given layer can be understood entirely in terms of the layer directly below; no knowledge is needed about how the lower layer is implemented (in terms of additional underlying layers) or of how the given layer is used by upper layers. The most well-known examples of this are the THE system[Dijkstra 68] and Multics[Organick 72].

Despite its attractiveness, layered structuring makes it inherently hard to construct a complex system. For many good designs, it is not always possible to assign processes to layers while ensuring strict layering. For example, many modern operating systems rely heavily on the ability of low-level system functions (such as Unix signals) to invoke user-level processes; this is impossible to describe using strict layering techniques, since it would involve a circular dependency between layers.

2.2 Hierarchical structuring

As with layered structuring, hierarchical structuring demands that strict relationships be imposed on the system's functions. In contrast to layered structuring, modules — which are used to hide design decisions about data structures, such as segment or process tables — are permitted to span several levels of the functional hierarchy [Habermann *et al.* 76]. The Cal system [Lampson and Sturgis 76], Pilot[Redell *et al.* 80], and Cedar[Swinehart *et al.* 86] are examples of systems constructed with hierarchical structuring².

Pilot, for example, consists of a collection of (Mesa) modules composing the hierarchical structure. They include (from bottom-up) machine simulating Mesa bytecodes, a low-level part of Mesa run-time support, primitive services such as network drivers, file systems, and virtual memory systems stream services, and so on. File and virtual memory systems in Pilot are based on what we call the manager/kernel pattern: a kernel provides primitives and the manager extends its functions using these primitives. In Pilot, functions filer, swapper, file manager, and virtual memory manager (from bottom-up) compose the function hierarchy, while modules the file and virtual memory systems are intermingled.

This structuring concept can be elaborated using the object-oriented paradigm and the notion of classes. Classes represent functions and compose the hierarchical structure: subclasses implement the details of their superclasses; subclasses can use the functions of their superclasses. We benefit from the class hierarchy at compile-time. This scheme encourages modularization, customization, code reuse, maintainability, extendability, and so on.

Choices[Campbell *et al.* 87] uses the class hierarchy to implement operating system functions such as memory management and process management. Objects which implement kernel functions are represented within the C++ class hierarchy. For example, process management in Choices uses the following class hierarchy [Russo *et al.* 88]:

```
Object
  Process
    ProcessContainer
      SingleProcessContainer
        CPU
```

²For this paper, we regard mono-lingual computing environments such as Cedar and Smalltalk-80 as operating systems.

FIFOScheduler
RoundRobinScheduler

Process implements independent execution of programs. *ProcessContainer* and its subclasses implement process scheduling. *CPU* represents an actual CPU in which actual scheduling mechanisms are implemented.

Hierarchical structuring is based on the experience with layered structuring. Benefits from class hierarchy, however, are exclusively inside operating systems. We need the uniform object view to be applied both inside and outside of systems.

2.3 Policy/mechanism separation

A mechanism provides a set of primitives intended to allow the definition and implementation of several significantly different policies. Separating mechanism from policy increases system flexibility by making it easy to change policies without modifying the underlying mechanisms. HYDRA[Wulf *et al.* 74] is a classic example of a system designed with policy/mechanism separation. HYDRA's kernel provides several mechanisms, including capability based protection, creation and representation of new types of objects, and primitive object type (procedure, local name space, and process) management. HYDRA demonstrates that a wide variety of useful and interesting security policies can be built on top of its carefully defined kernel-level capability-based protection mechanism [Cohen and Jefferson 75].

ARTS[Tokuda and Mercer 89] also employs this concept for real-time scheduling. The kernel provides the scheduling mechanisms and system primitives for binding a scheduling policy and setting its attributes. Each scheduling policy module is implemented as an object and the kernel mechanism upcalls a specified policy object to determine the current scheduling policy.

Policy/mechanism separation is a variant on hierarchical structuring. Policy modules are defined in the higher layer while mechanisms are implemented in the lower layer. Although this separation increases the flexibility of systems for users, it is difficult to decide what mechanisms should be provided in order to implement any policies users require.

2.4 Collective kernel structuring

Many modern operating systems are designed as a collection of largely independent processes. In this structuring style, one of the key responsibilities of the nucleus (or micro kernel) is to support interaction among the processes which implement operating system services. Examples of this approach include Chorus[Zimmermann *et al.* 81], Mach[Accetta *et al.* 86], and V-kernel[Cheriton 88].

Chorus divides the system into three layers: applications, subsystems, and the micro kernel. An application program is a collection of objects (or actors in Chorus terminology) that have their own execution environment (or subsystems in the Chorus terminology). A subsystem is also defined as a collection of actors. The subsystems are supported by the micro kernel (or nucleus in the Chorus terminology) which is located at each host [Rozier *et al.* 88]. Chorus/MIX[Armand *et al.* 90] is a subsystem that simulates System-V compatible UNIX. The UNIX subsystem is a collection of actors each of which is responsible for process management, memory management, and event handling to simulate the UNIX semantics.

Mach also defines a micro kernel in which the IPC facility, virtual memory management, task/thread management, processor management, and resource management are incorporated. There are several operating systems, such as BSD 4.3 UNIX, System-V 4.0, MS-DOS, the Macintosh operating system, and a real-time operating system, implemented on top of the Mach micro kernel. Camelot[Spector *et al.* 87], which is also implemented on top of Mach, provides run-time libraries that present programmers with transaction processing facilities.

The V-kernel acts as a software backplane, providing network transparency and memory management for lightweight processes and interprocess communication (IPC). Functions other than IPC management are implemented as kernel servers which are defined within the micro kernel. These include time management, process management, memory management, and device drivers.

Collective kernel structuring is the current state of the art for designing operating systems. In some ways, it takes advantage of the earlier structuring techniques. For example, it uses policy/mechanism separation, the micro kernel implements mechanisms while processes carry out policies. Since collective kernel structuring defines a way of structuring operating systems (i.e. operating system services are implemented on top of the micro kernel), it needs a discipline or rule, such as an object-oriented framework on which construct the system, to give a better perspective. Further, although this structuring enables an existing service to be replaced with new one, we need a way for a service to evolve itself or to acquire new functions for a service. For example, when a portable host with restricted user interface is re-connected to a network, a shell program should acquire new services to provide users with richer user interface. Collective structuring also has the benefit of separating the role of kernel into two parts: mechanisms that manipulate the system resources (such as physical memory, address space, and I/O), and the programming paradigm that helps programmers use the system effectively [Tokuda 90]. Of course, different programming paradigms can be built relatively easily, as is also true with virtual machines.

2.5 Object-based structuring

Operating system services are implemented as a collection of objects which are defined as segments protected by capabilities. Each object has a type which designates properties of the object: processes, directories, files, etc. An object has a set of operations by which its internal segment can be accessed and altered. Before a user requests an object, that user must acquire its capabilities including rights permitting operations. The kernel of the system generally has the responsibility to protect capabilities against malicious access. It also validates capabilities supplying by objects. HYDRA, StarOS[Jones *et al.* 79], Medusa[Ousterhout 80], iMAX 432[Intel 82], Eden[Almes *et al.* 85], Amoeba[Mullender 87], and Clouds[Spafford 86] are examples of systems designed with object-based structuring.

Object-based structuring includes the issues similar to those in collective kernel structuring. It also requires a discipline to organize operating system services as a collection of objects. Objects in these systems are static, coarse-grained, and expensive. We need objects that are dynamic, free-grained, and cheap.

2.6 Open operating system structuring

In open operating system structuring³, there is no distinction between user objects and system objects. The system is written in such a way that we can design, implement, access, and modify system objects in the same way as user objects. This structuring is adequate for systems that are used for experimental or testbed operating systems. Almost all systems based on this concept are used for the personal workstation: only a single user can access the system at a time. Also, almost all systems provide programmers with a mono-lingual environment (that is, users can use only single programming language). Pilot, Smalltalk-80[Goldberg and Robson 83], and Cedar are examples of such systems designed. This approach is dangerous and may lead to catastrophic failure, since the user can unintentionally or erroneously access system objects. A system that does not provide mechanisms to protect against such an attack will, of course, be unsuitable for large-scale use.

Since open operating system structures have a strong tendency to be mono-lingual, they cannot meet our basic requirement that all users will be able to compute successfully using any language. Although it is possible for mono-lingual systems to invoke functions or subsystems written in other languages — a recent version of Smalltalk-80 adds facilities to support precisely this activity — current attempts of achieving this are still largely *ad hoc*.

2.7 Virtual machine structuring

A virtual machine structure provides a set of abstract machines, each of which acts almost identically to the underlying hardware. This structure works by separately simulating each abstract machine on the underlying real machine. For instance, a simulation of the underlying card reader (and printer and disk) is used to produce (multiple) virtual card readers (and printers and disks). Thus, to each user of the system, it appears as if he has his own copy of the underlying hardware, so protection and security is relatively straightforward. VM370[Creasy 81] is a well-known example of this approach. One problem with this approach is that the performance of the operating system tends to degrade, since simulation can be costly. Recent technology for implementing virtual machines reduces this performance degradation using special assist facilities such as VMA.

As with collective structuring, virtual machine structuring makes it possible to implement several types of operating systems on top of each virtual machine. This is one of the key requirements for our environment. The structure is not, however, sufficient because the operating systems on each virtual machine are disjoint. Even with support for communicating across different emulated operating systems, each operating system is a distinct entity. This prohibits the rich kinds of interaction, sharing, and communication that we require.

2.8 Proxy structuring

Proxy structuring[Shapiro 86] is intended to ease the construction of distributed systems based on server/client design. In proxy structuring, a client object must acquire the proxy object that represents the server object. Then the client object can communicate with the server object by

³We do not use the term “open” to mean interconnectability of heterogeneous hardware, as it has recently been used in the commercial field. Rather, we use it to describe open-ended systems.

locally invoking the client's proxy object. When the client and server are executed on different hosts, a proxy is created on the client's host. A proxy behaves like a capability protecting the server object. Although a proxy object knows the location of the server object and is locally accessible from the client object, it is defined as a part of the server. By using a proxy object, users benefit from security of the system and location transparency which eases object migration.

SOS[Shapiro *et al.* 89] is an example of system based on proxy structuring. In SOS, an elementary object is a basic entity managed by the system. A fragmented object is a group that crosses contexts and is implemented using a proxy object.

Proxy structuring can hide differences between programming languages by supporting objects implemented in different programming languages in a uniform way. However, it is difficult to write programs for proxy objects, because they differ from normal objects.

2.9 Summary

The above structuring concepts are interrelated. In many cases, one structuring concept can or does employ other structuring concepts to overcome some of its drawbacks. Although the above have been suitable for constructing existing operating systems, we need a versatile and flexible structuring concept — not a rigid and unadaptable one — in order to construct environments meeting our requirements. We need a structuring concept with the following abilities, for example:

- To handle dynamic system behavior such as creating/destroying objects, varying network topology, and managing fine to coarse-grained objects,
- To deal with transparency such as location transparency, network transparency, and persistency transparency, and
- To control relationships among objects, such as the “knows about” dependency, the “uses” dependency, and the consistency dependency.

The above structurings are not enough to create an open and self-advancing distributed environment because they lack these abilities. In such an environment, there is an unlimited number of objects emerging and replacing old ones. These objects have various types of grain, lifespans, and real-time constraints. Furthermore, portable computers are dynamically connected and disconnected to the network. Mobile computers are connecting to the system while moving networks. Systems should have the capability to handle such dynamic behavior of objects. The above structuring techniques, however, cannot handle dynamic system behavior, since the system structure is unadaptable: for example, it is difficult to migrate an object to a new environment since the dependency among objects cannot be easily controlled.

We cannot provide users and programmers with a single level of transparency, for example:

- Although location transparency is useful for an object, it should be possible for an object to know location and/or distance of target objects and communication delay between them. This information is valuable for the object, for example, to deliver a message to one of the replicas and to make a decision on object migration policies.
- Although persistent objects are helpful for using object-oriented database systems, it should be possible for programmers to control persistency when they construct database systems.

Thus we need a level which does not provide transparency in order to implement services with transparency.

Objects are independent. However, we need a means of determining dependency among objects. For example, if an object migrates to another location, it might not be benefit from that migration because other objects which depend on the migrating object are still in the original location. Also, when an object migrates to a different environment, the system has to know what is dependent on the original environment. A transaction manager should keep track of objects which are parts of the transaction to maintain consistency. Thus we need ability to control dependency among objects. The above structuring can implement the mechanism of dealing with transparency and controlling dependency but only in an *ad hoc* way.

We introduce object-oriented concurrent computing [Yonezawa and Tokoro 87] as a basic model for overcoming the above issues. We provide a way to state differences of abstraction in a natural way by introducing reflective computing [Smith 84][Maes and Nardi 88], which we call the Muse object architecture. The next section describes the details of the Muse object architecture.

3 The Muse Object Architecture

The Muse object architecture, a new structuring concept designed to overcome the weaknesses identified in the previous section, enables us to achieve the requirements for the kind of environment described in the introductory section. Muse provides levels of abstraction: we can, for example, provide location transparent services for objects on top of location dependent services. The Muse object architecture explicitly defines levels of abstraction in such a way that a meta-object is separated from an object. Meta-objects define a higher level of abstraction than objects. Since a group of meta-objects composes meta-space and the relationship between an object and its meta-objects is relative, meta-spaces are represented within a hierarchical structure (or meta-hierarchy). This section presents the details of the Muse object architecture and compares it with existing structuring concepts.

3.1 A Meta-object is separated from an object.

Our basic computational model is *object-oriented concurrent computing*. Applications, devices (such as disks or network connections), and even storage are defined as objects or collections of objects. In the Muse object architecture, each object consists of:

- local storage,
- methods that access the local storage, and
- virtual processor(s) that execute(s) the methods.

The presence of virtual processors, which distinguishes Muse objects from many other object-based systems, allows each object to have its own execution environment. This style allows the introduction of *reflective computing* into object-oriented concurrent computing. Reflective computing provides ability to alter and/or change the behavior/computation of objects dynamically.

Each Muse object has one or more meta-objects. The purpose of meta-objects is to provide an optimal execution environment for the object. In particular, the meta-objects are the computational

units that simulate an object’s virtual processor(s). An object is “causally connected” with its meta-objects: the internal structure of an object is represented by meta-objects. Meta-computing, the computation of meta-objects in the meta-level, can alter the behavior of objects. Because meta-objects have knowledge of the status of their objects, they can optimally provide services for the object’s execution. As far as the authors understand, Muse is the first operating system designed based on reflective computing. Each Muse object is defined in such a reflective computing framework.

Figure 1 depicts the conceptual view of the Muse object architecture. Another figure showing the actual implementation of the architecture in the Muse operating system can be found in Section 5. Each object is supported by one or more meta-objects which constitute its meta-space. A meta-

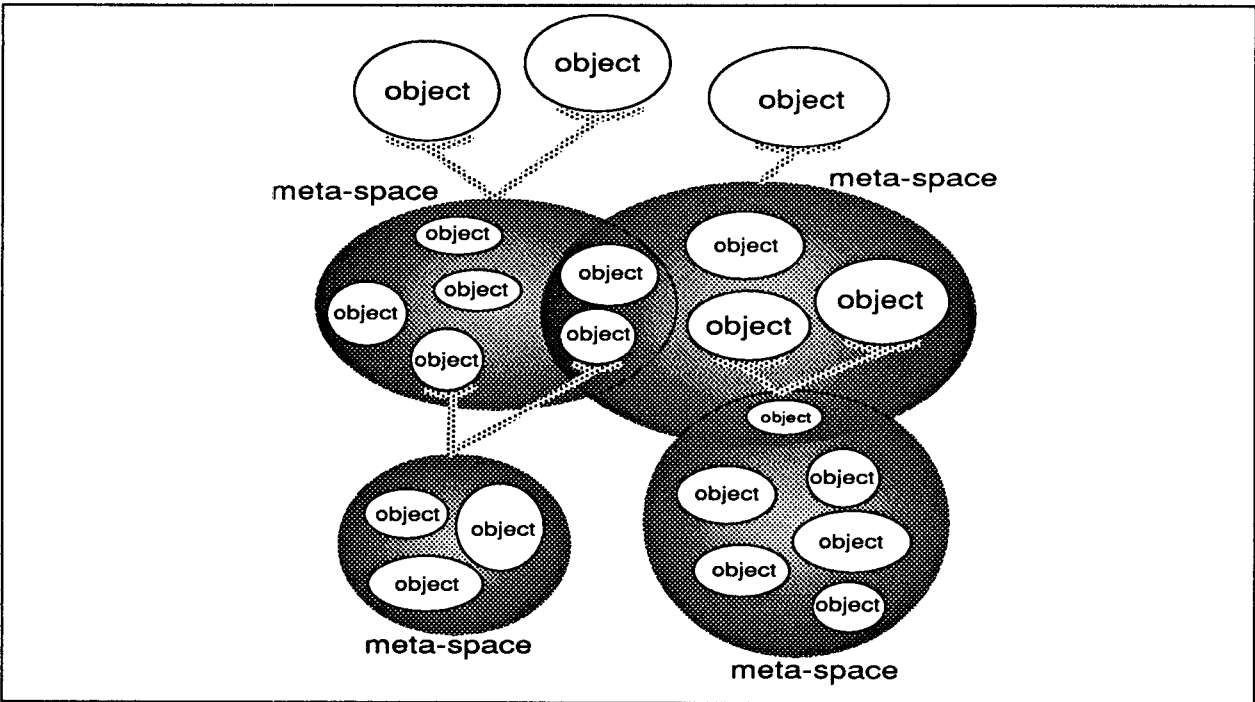


Figure 1: A Conceptual View of the Muse Object Architecture

space can be viewed as:

- a dedicated virtual machine for the object, or
- an optimized operating system for the object.

From the former view point, the semantics of execution of an object is given by its meta-space. For instance, the semantics of communication between objects — such as synchronous, asynchronous, and real-time⁴ — is defined by a meta-space. From the latter view point, the environment within which an object is evaluated is provided by the meta-space. For example, in order to support objects with real-time constraints, a scheduler meta-object with the ability of real-time scheduling and a

⁴Real-time communication guarantees the communication delay between objects.

memory meta-object with the ability to pin objects in memory provide an execution environment for the objects.

3.2 The meta-hierarchy

As shown in Figure 1, each meta-space contains one or more meta-objects. The Muse object architecture defines meta-objects as Muse objects. This means we need meta-objects of meta-objects. These meta-objects constitute another meta-space. In this way meta-spaces compose the hierarchical structure called the meta-hierarchy. The meta-hierarchy conceptually extends to infinite depth (Figure 1 breaks the meta-hierarchy for clarity).

Some meta-objects can be shared between different meta-spaces. For a given object, some meta-objects can be shared between same level meta-spaces in its meta-hierarchy, and some meta-objects can be shared between different levels. For example, mechanisms for a real-time scheduler are implemented as a meta-object that is shared between several meta-spaces while scheduling policies are separated between meta-spaces.

An object has ability to change its meta-space. We call this object migration. Since an object can be a meta-object of another object, and a meta-object can be an object which is supported by another meta-space, and since an object can change its meta-space, the relationship between an object and its meta-objects is *relative*.

An object migrates to a different environment by designating a meta-space that is to provide the new execution environment. If a meta-space cannot support the incoming object, the meta-space can integrate a new meta-object that can support the object; this is possible since the meta-space can query the state of the object. Changing meta-spaces allows us to deal with problems such as transparency in an effective way. In particular, we can construct a meta-hierarchy in which the lower-level meta-objects provide location transparency, and the higher-level meta-objects explicitly know the location of objects. This way, an object can change meta-spaces to the higher-level meta-spaces when location information is needed, but, otherwise, can compute in the context of the lower-level meta-objects.

3.3 What meta-objects exist in meta-space?

Here, we define a collection of key meta-objects that constitute meta-spaces:

Mailer meta-objects. Mailer meta-objects have the responsibility of delivering a message to a target object. At present, the mailer implements the remote procedure call style of synchronous and asynchronous communication. A network mailer delivers a message to a target object on a remote host via the underlying protocol handling and network hardware, since the mailer cannot send a message directly to a remote object.

Scheduler meta-objects. Scheduler meta-objects schedule objects that are supported by the meta-space. Scheduler meta-objects are represented within the meta-hierarchy if they do not have real-time constraints. Each processor has a scheduler meta-object that handles real-time constraints, while policy meta-objects for real-time scheduling compose the hierarchical structure: each meta-space contains a policy meta-object. The details are discussed in Subsection 4.2.

Memory meta-objects. Memory meta-objects manage physical memory and virtual memory. There are several memory meta-objects, each with its own memory management policy: for example, implementing virtual address space without paging for real-time computing, implementing distributed shared virtual memory as in Kai Li's [Li 86], and implementing object memory with automatic garbage collection.

Decision maker meta-objects. Decision maker meta-objects are used for making a decision about object migration policies: which objects should migrate, when should they migrate, and where should they migrate to. Decision maker meta-objects gather information such as machine load and network load by using the facilities of Muse-IP[Teraoka *et al.* 89].

3.4 Reflective Computing

The above definitions give the structure of the Muse object architecture. This subsection defines the execution model of this architecture. We introduce the following primitives to facilitate interaction between an object and its meta-objects:

- P1: for an object to make a meta-computing request (that is, a request to a meta-object), and
- P2: for a meta-object to reflect the result of meta-computing to its object.
- P3: to maintain the causal connection link between an object and its meta-space.

The Muse object architecture provides these primitives as basic functions, and the Muse operating system is constructed using these primitives. For example, we can implement inter-object communication in such a way that:

1. A sender object requests that a mailer meta-object delivers a message to a target object using primitive P1.
2. A mailer meta-object retrieves a message from the sender object and determines the target object according to the contents of the message. Then, the message is stored in the target object. The internals of an object such as a message queue, are represented as meta-objects due to the causal connection between an object and its meta-space (maintained by primitive P3), these tasks are represented as meta-computing, i.e. executed by communicating meta-objects.
3. A mailer meta-object activates the activity of the target object by using primitive P2. This means that the target object is activated by receiving an incoming message.

3.5 Summary

The Muse object architecture subsumes the existing structuring concepts discussed in Section 2. It contains collective kernel structuring and object-based structuring: a system consists of a collection of objects that comprise the meta-hierarchy. Unlike a system based on collective kernel structuring, an object is a fundamental entity. Unlike a system with object-based structuring, an object is defined within the meta-hierarchy: an object is defined in the framework of reflective

computing (or the Muse object architecture). The Muse object architecture contrasts with collective kernel structuring in that the meta-meta-space⁵ can be thought of as a micro kernel and meta-spaces can be thought of as an operating system emulating objects in systems like Mach and Chorus.

Unlike hierarchical structuring, a group of objects is represented within the meta-hierarchy. The meta-hierarchy is orthogonal to policy/mechanism separation. Policy/mechanism separation can be applied within the same meta-space or between the meta-spaces.

The Muse object architecture elaborates virtual machine structuring. Unlike virtual machine structuring, a meta-space is a collection of objects and composes a meta-hierarchy.

The Muse object architecture, thus, is a new operating structuring concept that supports the environments' requirements described in the introductory section. Since the system is dynamic in nature, separating meta-objects from objects and reflective computing facilitate handling the dynamic behavior of objects. For instance, we can provide a meta-object that works differently on objects of the same kind but different size.

For objects, we can deal with transparency and control dependency among objects using the mechanism of changing meta-spaces or object migration. For instance, we can manage portable and/or mobile hosts using a virtual network[Teraoka *et al.* 90]. In a virtual network, the network layer is divided into two sublayers: one is a virtual network sublayer, which hides location and movement of hosts; the other is the physical network sublayer, which knows the location of hosts. Thus, we can implement the physical network sublayer as a meta-object for the virtual network sublayer.

4 The Muse Operating System — Examples of the Muse Object Architecture

Many operating system services are difficult to implement using existing structuring techniques. This section shows how many of these services can be built easily using the Muse object architecture. The services we describe are:

- multi-language programming facilitated by class systems,
- a real-time scheduler with hierarchical policies, each of which is suitable for scheduling applications with real-time constraints, and
- free-grained objects that adapt their granularity to suit the application and by which storage of the object is efficiently managed according to its granularity.

Each of these services is a key part of the Muse operating system.

4.1 Class systems

Each object has a class that acts as a static immutable template of the object. A class contains both machine independent and dependent information including the structure of the object, binary images of the text, and the format of the data representation. Classes are also used for data and

⁵See Section 5.

binary image conversion when objects migrate to heterogeneous hardware: a meta-object converts the binary image to the new one (a *Loader* meta-object is responsible for such tasks).

Figure 2 shows the conceptual view of class systems in the Muse operating system. Since classes

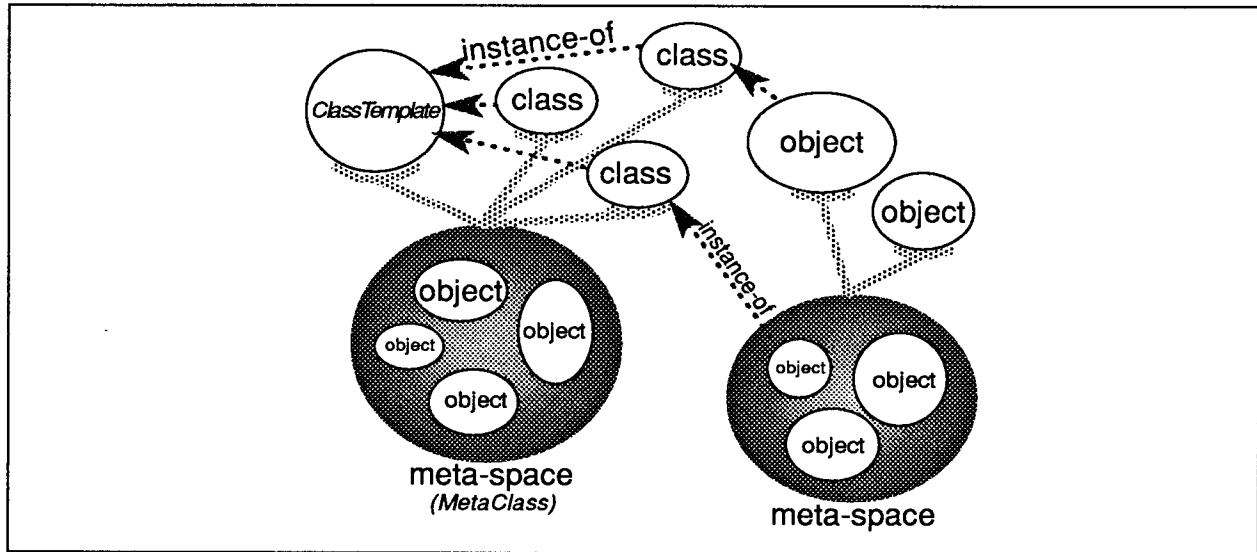


Figure 2: A Conceptual View of Class Systems

are Muse objects, each class has a class and a meta-space. We call these *ClassTemplate* and *MetaClass*, respectively. *MetaClass* provides classes with the immutability. It designates the default meta-space for a new object when a meta-space is not given by a creation message. *ClassTemplate* defines the structure of each class.

Classes can be represented within a (single and/or multiple) hierarchy. The class hierarchy is independent of the meta-hierarchy. The class hierarchy is static and is an asset at compile-time[Yokote *et al.* 89]. We introduce a *delegation* mechanism[Lieberman 86] to obtain properties of other objects at run-time. Unlike [Lieberman 86], the Muse delegation mechanism is restricted: an object cannot access variables defined in the object to which the message is delegated. The delegation is initiated by a mailer meta-object and archived by a *MetaClass* meta-object.

In the current implementation, a class system provides compile-time and run-time facilities for the C++ programming language. It also provides tools that assist in programming. Each C++ class is represented as a Muse class object. That is, a *ClassTemplate* object defines the internal structure of a C++ class, while class hierarchy, size of an object, names of variables, etc. are defined in Muse class objects. A *MetaClass* meta-object provides a facility for navigating the class hierarchy to find a class required for compilation: a C++ compiler uses this facility to collapse the class hierarchy.

We can implement class systems that handle any type of programming language: class systems are extended to support a versatile multi-lingual environment. We can provide a *MetaClass* meta-object and a *ClassTemplate* object for each programming language. These can encapsulate any language dependent information.

We can also implement class systems in such a way that each class has a specification part

and an implementation part. This separation facilitates different implementations with the same interface and accommodates heterogeneity. The specification part of a class defines the interface to the class, while the implementation part of a class defines the concrete realization of the class. Muse differs, however, in that there might be several implementation parts of a class, each of which implements a specification according to desired algorithm and hardware dependency. In such a class system, while a class internally has two or more parts, only one specification part is externally visible. These parts can be represented as Muse classes. A *ClassTemplate* object defines the structure of such classes and a *MetaClass* meta-object implements access to the specification part and the implementation parts. Since meta-objects provide the execution environment for objects, they can select the desired implementation part of the class.

HCS takes a similar approaches[Notkin 90]. HCS employs proxies to accommodate hardware and/or system heterogeneity. Proxies are objects which encapsulate hardware dependency and act as stubs. A proxy is created by a class represented within the class hierarchy. We can implement this scheme using the Muse class system: proxies can be implemented as meta-objects and are created by Muse classes.

4.2 A real-time scheduler with hierarchical policies

Each meta-space contains a scheduler for the objects that meta-space supports. This structure enables us to provide a suitable scheduler for applications which, as stated above, consist of a collection of objects. This structure leads us to the hierarchical scheduler, since there are two or more schedulers in the system which are represented within meta-spaces composing a meta-hierarchy. Since each meta-space presents a virtual computing environment for an application, the scheduler defined in a meta-space cannot meet real-time constraints.

Figure 3 shows our real-time scheduler with hierarchical policies. To meet real-time constraints we separate policy objects that define scheduling policies into intended policy objects and a base policy object. Intended policy objects comprise the hierarchical structure and pass the following information down to the base policy object:

- *importance*, used for deciding which object is a candidate for failing when several objects cannot all meet their deadlines,
- *arrival rate*, a cycle for an object to be activated,
- *worst case processing time*, an estimate of the time within which an object is expected to finish execution, and
- *deadline time*, the time left until an object has to finish execution.

Based on this information, a base policy object decides which scheduling algorithms — rate monotonic scheduling, earliest deadline first scheduling, etc. — to use. The adaptation mechanism taken in real-time operating systems such as Spring-kernel[Stankovic and Ramamritham 89] should be integrated into our real-time scheduler in order to handle dynamic objects.

4.3 Free-grained objects

In environments like ours, there are various sizes of objects due to the diversity of programming languages and applications. The system should provide a mechanism that efficiently manages

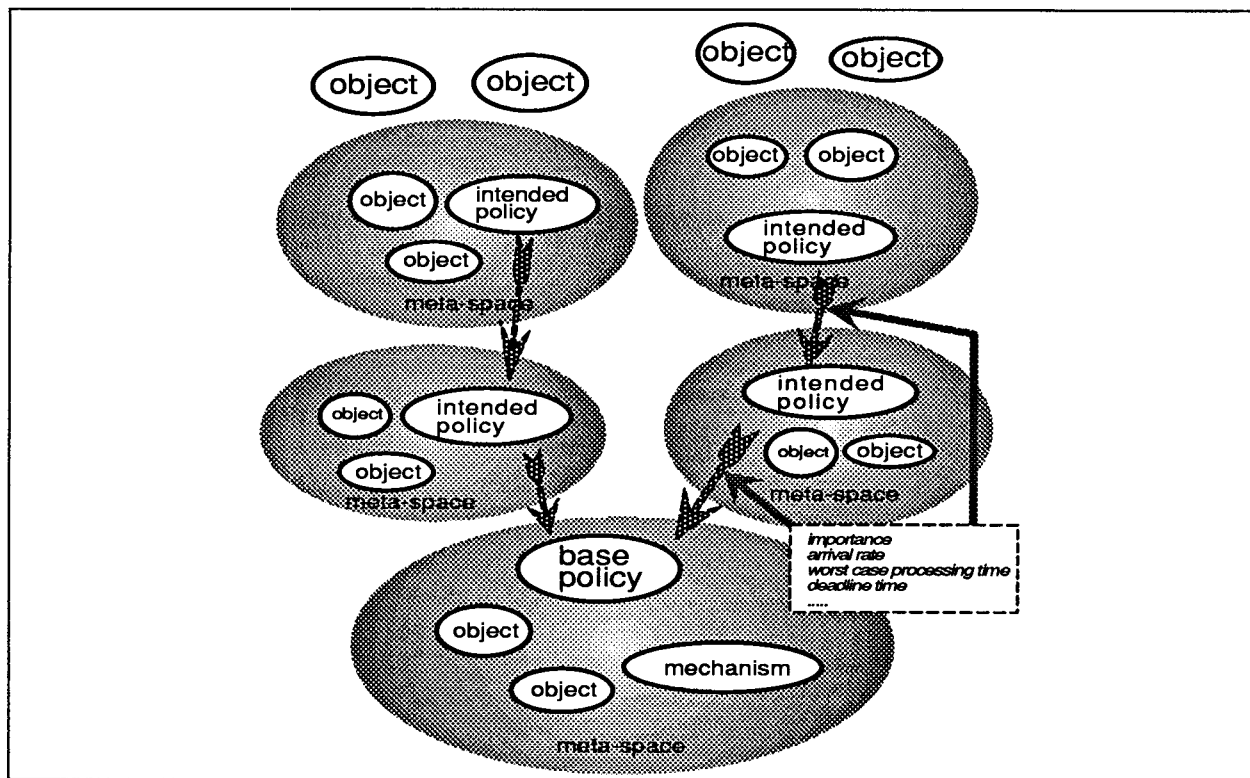


Figure 3: The Simplified View of the Hierarchical Policies

storage for objects of various sizes. Muse solves this problem by providing several memory meta-objects that implement different types of memory management policies. For example, if an object is fine-grained, a memory meta-object may locate two or more objects within the same address space, so that communication overhead, which is mainly caused by context switching, can be reduced. Figure 4 shows five objects located in three address spaces. Objects A and B and objects C and D reside in the same address spaces, respectively, while object E occupies its own address space. If object C communicates with objects A and B, this causes context switching since they are located in different address spaces. If they communicate with each other frequently, they should be incorporated into the same address space using mechanisms provided by the memory meta-object. This idea is derived from the task forces in StarOS and Medusa. Unlike the task forces, this scheme is fully dynamic and not visible to programmers. A memory meta-object performs this task in cooperation with other meta-objects such as a mailer meta-object and a decision maker meta-object. This scheme is also applied to reduce the context switching overhead when two objects are located on different hosts.

Object migration like this may be dangerous. In particular, it might increase, rather than decrease, context switching overhead. For example, assume that the left most address space and other address spaces are located at different hosts. In Figure 4, passing a message between objects C and E means delivering the message across hosts. This increases overhead for communication

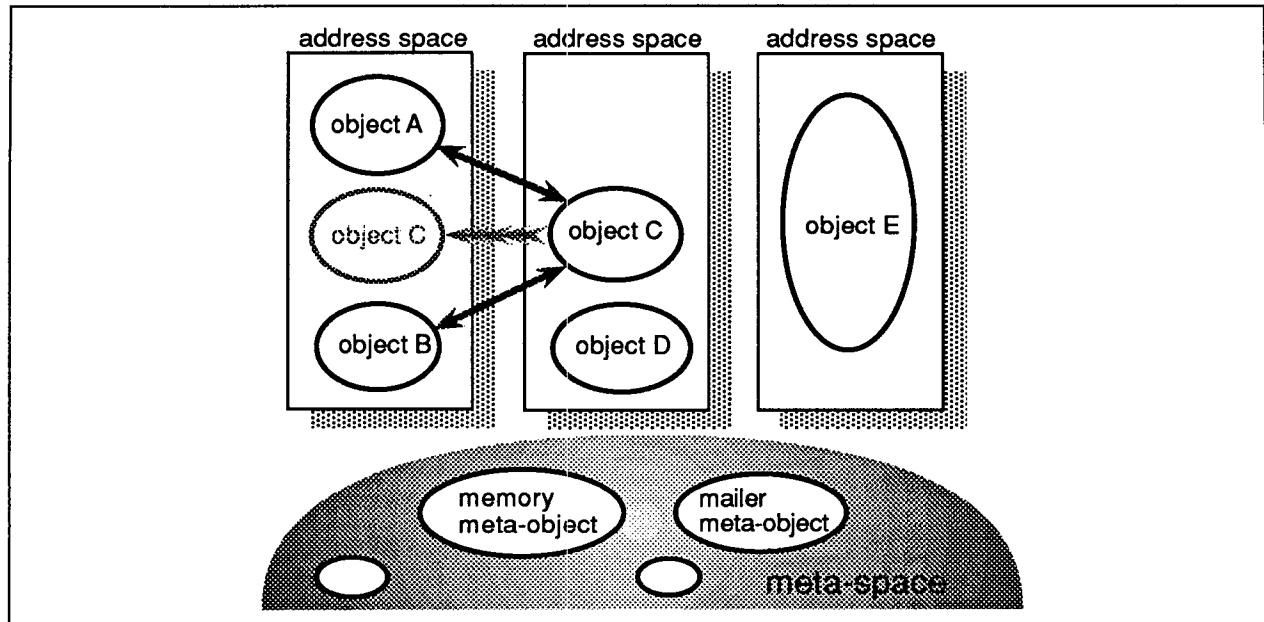


Figure 4: Address Spaces and its Memory Meta-object

between them after moving object C to A and B’s address space. To handle this potential problem of migration, we use a new computing model called the Computational Field Model (CFM for short), which is best described in terms of some common terms from physics [Tokoro 90]. CFM defines a metric space composed of:

- *mass*, representing the size of an object,
- *distance*, representing the logical distance between objects: communication bandwidth and latency are considered along with geographical distance,
- *gravitational forces*, representing communication frequency between objects and the amount of data objects transmit to one another,
- *repulsive forces*, representing the computing load around an object, and
- *inertia* or *friction*, representing the cost of object migration.

CFM facilitates a decisions about object migration.

Furthermore, in cooperation with a mailer meta-object, communication between objects can sometimes be translated to local procedure calls. Since a mailer meta-object can trace paths of communication between objects, if the semantics of a communication is the same as local procedure calls, this communication is a candidate for bypassing meta-objects: a method is invoked by the local procedure call without assistance of a mailer meta-object.

Most large-scale distributed systems rely on each object having a unique ID. There are several issues that arise when this approach is taken. An ID is usually assigned according to an object’s lifetime and granularity. It may also depend on whether an object is shared with other objects. Unique IDs for coarse-grained objects must be large, since there are a large number of objects in

the environment. For fine-grained objects, though, it is difficult to use the same ID format. It is too big, in practice. For instance, our current implementation, which is not unusual, uses 96-bit unique ID's for external reference; of course, 96-bit references for local objects are simply not acceptable.

If we give up using unique ID to identify fine-grained objects, we have to introduce another mechanism that can distinguish them. There are at least two methods for doing this:

- The uniqueness of an identifier can be guaranteed within a domain but not throughout the whole system. This makes it difficult to determine whether two objects are identical and to determine which objects belong to different domains.
- A unique ID can be assigned when an object is referenced. This reduces the utilization of the unique ID. However, this increases the difficulty of object migration, since an object containing local IDs is only valid in the original domain.

In the current implementation, we introduce two or more ID management meta-objects which guarantee the uniqueness of object IDs. We designate objects with IDs issued by an ID management meta-object in the meta-space supporting the object. Also, an object ID is assigned when the object is referenced by others.

4.4 Summary

Muse has several features which distinguish it from other object-based operating systems such as HYDRA, Amoeba, Clouds, ARTS, and so on. First, Muse objects are active, while existing object-oriented operating systems generally have passive objects. For instance, in the usual approach, an object is defined passively, such a file or memory object, and requires separate processes (which in some systems are also objects) as an execution environment for executing its methods. Second, Muse provides reflective facilities based on the Muse object architecture. Each object has its own meta-space. This allows the system to evolve, since each object can monitor its own status (and that of other objects) to tailor the system for the most efficient object execution. Third, Muse provides an inheritance mechanism that encourages object-oriented programming. The class hierarchy is used at compile-time while a delegation mechanism is used at run-time. According to the definition of [Wegner 87], Muse is an object-oriented operating system.

There are several issues in designing object-oriented operating systems and services. They include: object granularity, treatment of language objects and system objects, object sharing for efficiency, dealing with transparency, and controlling dependency. Each of the services described above has unique features that can be implemented relatively easily based on the Muse object architecture. Providing objects *MetaClass* and *ClassTemplate* makes it possible to define objects in a uniform way independent of the programming languages in which the objects are implemented. We can also use class semantics which separate the specification and its implementations. We can meet real-time constraints of application even though schedulers compose a hierarchical structure. Our scheme for hierarchical scheduling can also apply to systems based on collective kernel structuring. The Muse object architecture makes it far easier to implement free-grained objects than conventional approaches. Meta-objects facilitate the definition of objects with any granularity and the optimization of object execution such as hand-off scheduling and identifier management. Systems such as Emerald[Jul *et al.* 88] and Amber[Chase *et al.* 87] support fine-grained objects and their

migration. These systems, however, provide these services for a single programming language. Muse can define objects in such a way that there is no distinction between objects supported by operating systems and objects supported by programming languages in a multi-lingual environment.

5 Implementation

To give a better feel for the Muse object architecture, we will briefly describe the prototype implementation of the Muse operating system. Figure 5 shows a simplified view of the object structure of the Muse operating system version 0.3. Although the architecture can support more,

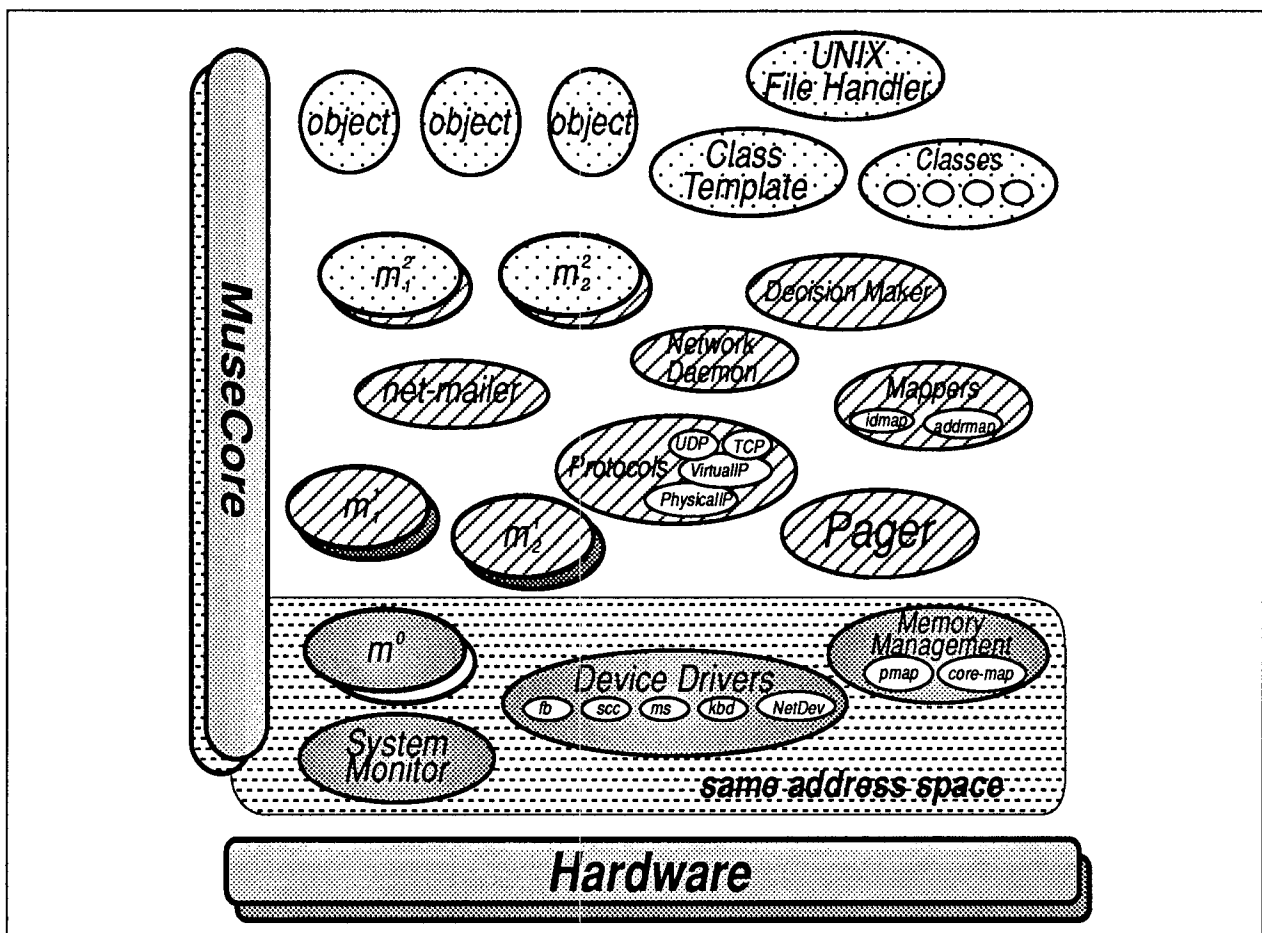





Figure 5: The Simplified View of the Object Structure of the Muse Operating System Version 0.3

version 0.3 uses only three layers of meta-hierarchy: an object space, a meta-space of the objects in an object space, and a meta-meta-space of the objects in a meta-space. Meta-objects in the meta-meta-space implement hardware dependent functions.

To implement the Muse object architecture, we introduce the following fundamental facilities:

- *MuseCore* provides a virtual machine environment in which objects can utilize the Muse object architecture. It employs a *Context* structure which is used for mapping object execution to CPU execution. It also maintains the relationship between an object and its meta-objects and manages the transfer of control between them.
- A *reflector* (indicated by m_m^n , which specifies the m -th instance of a reflector in the n -th layer of the meta-hierarchy in Figure 5) is an entry point through which objects, explicitly or implicitly, invoke meta-computing. An object is connected to its meta-space by a reflector: an object can know the name of meta-objects defined in the meta-space through this reflector. In the implementation, a reflector also contains parts of the functions of mailer and scheduler meta-objects for efficiency.

In Figure 5, a shaded oval  denotes meta-objects constituting the Muse kernel. In the implementation, these meta-objects are assembled into the same address space. They make up one meta-space that can be seen through reflector m^0 . We call this meta-space the meta-meta-space. A stripped oval  denotes the meta-objects that implement Muse system functions: for example, pagers for virtual memory management, protocol handlers for network communication, and decision makers for object migration. In the implementation, there are several meta-spaces that can be seen through reflectors (m_1^1 and m_2^1 in the figure). Some meta-objects might be shared between two or more meta-spaces. The meta-objects in these meta-spaces are supported by the meta-meta-space. A dotted oval  denotes objects that constitute applications. Each object is supported by one of the meta-spaces.

For reflective computing, *MuseCore* implements the primitives described in Subsection 3.4. In implementation, we introduce the following three primitives:

- **M**: an object makes a request of meta-computing: an object invokes a method of a meta-object in its meta-space through its reflector,
- **R**: a meta-object reflects the result of meta-computing: it resumes object execution, and
- **C**: objects designate parameters of *MuseCore* which is used for managing a *Context* structure: for example, creating and destroying it and binding *Context* to external events.

6 Conclusion

Given the rapid advances in network technology, we crave the advent of ultra large-scale, open, self-advancing, and distributed environments. Existing structuring concepts for operating systems are, however, insufficient for developing such environments. We propose the Muse object architecture and demonstrate the Muse operating system, which is intended to support environments like these. The novel features of this architecture include:

- The notion of an object and its meta-objects gives programmers a clear abstraction of the system. We distinguish the meta-level abstraction from the object-level abstraction in a natural way.

References

- Reflective computing provides a basic mechanism to realize a self-advancing system. This is important in the kinds of environments in which we are interested. The system can be tailored to suit the objects that comprise an application.
- The system is flexible and adaptable. Several operating system policies can coexist in the system.

Version 0.3 of the Muse operating system is running on Sony NEWS workstations each of which has 25MHz MC68030 CPU and a minimum 4MB of physical memory without external cache. The system is written in the C++ programming language. We experiments using version 0.3 of the Muse operating system on, for example:

- negotiation between conflicting meta-objects,
- supporting portable (and mobile) hosts, and
- porting the system to different types of hardware including a Sun3, Sony RISC-NEWS workstations with MIPS R3000 CPU, the Symmetry shared memory multiprocessor machine, and a portable computer.

Acknowledgments

We give our great thanks to Dr. Hideyuki Tokuda of Carnegie Mellon University and the members of Sony Computer Science Laboratory Inc. Several discussions with them were helpful to us in inventing the architecture, designing the system, and determining the implementation strategy of the system. We also thank Professor David Notkin of the University of Washington who read an early draft of this paper carefully and give us valuable comments to improve the quality of this paper. We could not have finished this paper without his efforts.

References

- [Accetta *et al.* 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *USENIX 1986 Summer Conference Proceedings*. USENIX Association, June 1986.
- [Almes *et al.* 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January 1985.
- [Armand *et al.* 90] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier. Multi-threaded Processes in Chorus/MIX. In *Proceedings of EUUG Spring'90 Conference*, April 1990.
- [Campbell *et al.* 87] Roy Campbell, Gray Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, Vol. 21, No. 3, July 1987.

References

- [Chase *et al.* 87] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 11th ACM Symposium on Operating System Principles*. ACM, November 1987.
- [Cheriton 88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, Vol. 31, No. 3 pp. 314–333, March 1988.
- [Cohen and Jefferson 75] Ellis Cohen and David Jefferson. Protection in the HYDRA Operating System. In *Proceedings of the 5th ACM Symposium on Operating System Principles*. ACM, November 1975.
- [Creasy 81] R. J. Creasy. The Origin of the VM/370 Time-Sharing Systems. *IBM Journal of Research and Development*, Vol. 25, No. 5 pp. 483–490, 1981.
- [Dijkstra 68] Edsger W. Dijkstra. The Structure of the “THE” — Multiprogramming System. *Communications of the ACM*, Vol. 11, No. 5, May 1968.
- [Goldberg and Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [Habermann *et al.* 76] A. N. Habermann, L. Flon, and L. Coopride. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, Vol. 19, No. 5, May 1976.
- [Intel 82] Intel. *iMAX 432 Reference Manual*, 1982. Order Number: 172103-002.
- [Jones *et al.* 79] Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl. StarOS, a Multiprocessor Operating System for the Support of Task Forces. In *Proceedings of the 7th ACM Symposium on Operating System Principles*. ACM, December 1979.
- [Jul *et al.* 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.
- [Lampson and Sturgis 76] Butler W. Lampson and Howard E. Sturgis. Reflections on an Operating System Design. *Communications of the ACM*, Vol. 19, No. 5, May 1976.
- [Li 86] Kai Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. Technical report, Yale University, December 1986.
- [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. also appeared in SIGPLAN NOTICES, Vol.21, No.11.
- [Maes and Nardi 88] Pattie Maes and Daniele Nardi, editors. *META-LEVEL ARCHITECTURE AND REFLECTION*. North-Holland, 1988.

References

- [Mullender 87] Sape J. Mullender, editor. *The Amoeba distributed operating system: Selected papers 1984 – 1987*. Centre for Mathematics and Computer Science, 1987. CWI Tract 41.
- [Notkin 90] David Notkin. Proxies: A Software Structure for Accommodating Heterogeneity. *Software Practice and Experience*, Vol. 20, No. 4, April 1990.
- [Organick 72] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. The MIT Press, 1972.
- [Ousterhout 80] John K. Ousterhout. Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa. Technical Report CMU-CS-80-112, Department of Computer Science, Carnegie-Mellon University, April 1980.
- [Redell *et al.* 80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, Vol. 23, No. 2, February 1980.
- [Rozier *et al.* 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, Vol. 1, No. 4, Fall 1988.
- [Russo *et al.* 88] Vincent Russo, Gary Johnston, and Roy Campbell. Process Management and Exception Handling in Multiprocessor Operating Systems using Object-Oriented Design Techniques. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1988*. ACM, September 1988. also appeared in SIGPLAN NOTICES, Vol.23, No.11.
- [Shapiro 86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986.
- [Shapiro *et al.* 89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An Object-Oriented Operating System — Assessment and Perspectives. *Computing Systems*, Vol. 2, No. 4, 1989.
- [Smith 84] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, January 1984.
- [Spafford 86] Eugene Howard Spafford. *Kernel Structures for a Distributed Operating System*. PhD thesis, Georgia Institute of Technology, May 1986.
- [Spector *et al.* 87] A. Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. J. Bloch. Camelot: A Distributed Transaction Facility for Mach and the Internet – An Interim Report. Technical Report CMU-CS-87-129, Department of Computer Science, Carnegie-Mellon University, June 1987.
- [Stankovic and Ramamritham 89] John A. Stankovic and Krithi Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *Operating Systems Review*, Vol. 23, No. 3, July 1989.

References

- [Swinehart *et al.* 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.
- [Teraoka *et al.* 89] Fumio Teraoka, Yasuhiko Yokote, and Mario Tokoro. Muse-IP: A Network Layer Protocol for Large Distributed Systems with Mobile Hosts. In *Proceedings of 4th International Joint Workshop on Computer Communications*, July 1989. also available as SCSL-TR-89-003 of Sony Computer Science Laboratory Inc.
- [Teraoka *et al.* 90] Fumio Teraoka, Yasuhiko Yokote, and Mario Tokoro. Virtual Network: Towards Location Transparent Communication in Large Distributed Systems. In *Proceedings of 5th International Workshop on Computer Communications*, June 1990. also appeared in SCSL-TR-90-005 of Sony Computer Science Laboratory Inc.
- [Tokoro 90] Mario Tokoro. Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment. In *Proceedings of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems*, September 1990. also appeared as Technical Report SCSL-TR-90-006.
- [Tokuda 90] Hideyuki Tokuda. Private Communication, August 1990.
- [Tokuda and Mercer 89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A Distributed Real-Time Kernel. *Operating Systems Review*, Vol. 23, No. 3, July 1989.
- [Wegner 87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1987*. ACM, October 1987. also appeared in SIGPLAN NOTICES, Vol.22, No.12.
- [Wulf *et al.* 74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, Vol. 17, No. 6, June 1974.
- [Yokote *et al.* 89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of European Conference on Object-Oriented Programming*, July 1989. also appeared in SCSL-TR-89-001 of Sony Computer Science Laboratory Inc.
- [Yonezawa and Tokoro 87] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.
- [Zimmermann *et al.* 81] Hubert Zimmermann, Jean-Serge Banino, Alain Caristan, Marc Guillemont, and Gérard Morisset. Basic Concepts for the Support of Distributed Systems: The Chorus Approach. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*. IEEE, 1981.

Glossary

Objects. An object is a fundamental entity in the system. It has local storage and methods. It also provides uniform interface with the outside of the object. The local storage of an object is accessed by a method invoked by an incoming request message.

Concurrent objects. A concurrent object is an object which encapsulates local storage, methods, and a virtual processor. Local storage of a concurrent object is accessed by methods executed by a virtual processor: one and only one activity is conceptually associated with a concurrent object to execute a method. A concurrent object facilitates synchronization problems: concurrent requests are synchronized at the entry point of the object.

Classes. A class describes the similarity of a set of objects. For example, it contains methods which can access the internal structure of an object. A class also acts as a template for creation of an object.

Meta-objects. A meta-object is itself an object which provides an environment for executing the object. Meta-objects implement the meta-functions of objects such as scheduling, communication, and object management.

Reflective computing. Reflective computing allows an object to alter its meta-functions, usually represented as meta-objects, during its execution. An object and its meta-objects are causally connected: the internals of the object are represented as meta-objects.

Class inheritance. A class can be defined as a subclass of another class: classes compose a hierarchical structure to represent difference and to encourage differential programming. Inheritance is usually a programming and a compile-time facility.

Delegation. Delegation is a mechanism which forwards an incoming message to a designated object. The environment of the object is also forwarded to the designated object. Delegation is usually a run-time facility.

