Unification of Active and Passive Objects in an Object-Oriented Operating System

Kenichi Murata[†,‡], R. Nigel Horspool[‡], Eric G. Manning[‡], Yasuhiko Yokote[*], Mario Tokoro[†,*]

† Department of Computer Science Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama 223 JAPAN

‡ Department of Computer Science, University of Victoria
Victoria, British Columbia V8W 3P6 CANADA

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

Abstract

This paper proposes the unification of active and passive objects in object-oriented operating systems. With such a unification, programmers can concentrate on programming an algorithm without having to consider how objects are used and executed. Also, all objects can be migrated in a distributed system. To achieve this unification, meta level scheduling control is introduced, and the scheduling policy is determined at object creation time. We named this system Cognac. In Cognac, the execution domain of an object is an object cluster, in which some metaobjects control the execution of member objects at run-time. The scheduler metaobject manages method invocation and the status of its member objects. Since all objects' attributes are managed by metaobjects, mutual exclusion is guaranteed.

1 Introduction

The object-oriented paradigm has become popular and it is now widely accepted as a good tool for constructing large, complex systems, because of its excellent modularity. It is also helpful in building concurrent systems.

Several concurrent object-oriented programming languages have been designed and implemented, based on the concurrent object model[1], where each object is an active entity. From the viewpoint of the operating system, each object is a process, with its own single thread of control.

Apertos[2] is an object-oriented operating system which supports concurrent objects and which is itself composed of concurrent objects. At first, in programming for Apertos, C++ was used as the primary language. However, we needed to write a lot of stub code to represent the active objects. In the latest version of Apertos, we use MC++[3] for programming an Apertos object, where we can declare an active/passive attribute in the C++ class definition. In MC++, internal data structures are also modeled as objects, where the objects are passive entities and are only accessed in a single active object. However, the distinction between active and passive objects is confusing to programmers, and it makes programming more difficult.

From our experience with programming using MC++, we felt that there was a great need for system support where all objects are viewed as equal entities and have equal ability to migrate amongclusters to find a suitable execution environment. To achieve unification of active and passive objects, we introduced run-time object scheduling suitable for this set of objects, and plug-in intermediate method code. We named this system *Cognac*[1].

2 Concurrency in Object Models

2.1 Object/Thread Model

To incorporate concurrent constructs in a

1. The name has no acronymic significance.

```
// programmed first
class Class_A {
private:
      int      x;
public:
      void     Method_1 ( ) { x = 0; }
               // must be modified when
               // subclass Class_B is added
};

// programmed later
class Class_B : public Class_A {
public:
      void     Method_2 ( ) {
               Lock (x);
               x = 100;
               // conflict with Class_A::Method_1
               Unlock (x);
      }
};
```

Figure 1: Mutual Exclusion Problem in Inheritance

conventional object-oriented system, the notion of execution threads was introduced in some systems, e.g. Clouds[4]. A model in which concurrency is introduced in this way is called an *object / thread model*.

In this model, objects are passive entities, and each thread can execute object method code at any time. Good response times and good performance are expected. However, since threads execute independently, a mutual exclusion problem might occur when more than two threads simultaneously execute the same method. To avoid this problem, a locking mechanism such as a semaphore must be used to control access to critical sections in a method.

This model has the advantage of familiarity, because a conventional programming language is used as the base language. However, there are two great disadvantages in programming. First, since the original base language does not have ability to control concurrent execution, a new external (unprogrammable) entity that represents a thread must be introduced. Second, the locking mechanism makes programming difficult, especially when inheritance is used. For example, when Class_B is defined in Figure 1, Method_1 in Class_A, which was previously programmed and is inherited by Class_B, must be modified to use locking for variable x.

2.2 Active Object Model

The active object model views an object as an active entity which communicates with other objects by sending messages. In this model, an object is composed from instance data, a communication module, an incoming message queue and its own execution context. Each message arriving at the object is, at first, spooled in the message queue by the communication module. Each object also has the ability to select a message from the queue to be executed next. If the object decides not to handle a message at this moment, the message can be pushed back into the message queue. Since each active object communicates with others only by sending messages, they can run concurrently.

An active object is called *atomic* if there is exactly one thread of control. In this case, since execution context is unique for each object, messages are processed sequentially, and the programmer of the object need not consider mutual exclusion. In this paper, all active objects are assumed to be atomic.

Some programming languages have been implemented using this model, as seen in [1]. Most actor-base languages, like ABCL/1[1], AL-1/D[5], manage internal instance data as primitive data which is not viewed as an object, and is accessed by special predefined operations. If the object is fine-grained, there is much message passing among objects, and it causes much context switching. This leads to a large performance penalty in a big system.

To manage the granularity of the objects, internal instance data can be also defined as an object, called a passive object. There are two different ways to introduce passive objects: (1) using another language to describe passive objects, and (2) defining passive objects using the same language in which the active object is described.

MC++ uses passive objects in the latter manner to reduce context switching among active objects. However, it is the programmer's responsibility to use a passive object only inside a single active object to avoid execution conflicts. The programmer therefore always has to consider the choice between active and passive for each object. This means that messages exchanged between active objects cannot contain a pointer to an internal passive object. As it is difficult to share passive objects correctly, programming is difficult.

3 Unification of Active and Passive Object

The programming task becomes much simpler if the distinction between active and passive objects can be eliminated.

A necessary but not sufficient condition for an object O to be passive is, that it is used by only one active object. This is, if O is shared among two or more active objects, it must be active (to deal with possible mutual exclusion problems). And, this condition is necessary but not sufficient because, for example, a stack object must

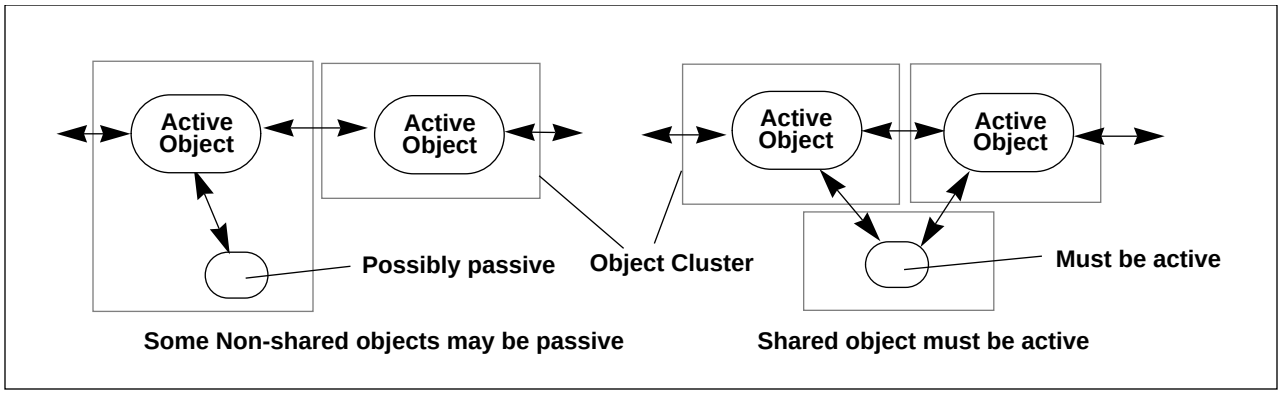

Figure 2: Active/Passive Object Determination

reject a pop request when it has no data on the stack. In this situation, such an object should be explicitly placed by the programmer in an object cluster which supports the active object model. Also, if an object is constant (i.e. all internal data is stored only at creation time, and it is never modified after creation), the object can be passive. However, it is difficult to determine this property at compile time. Therefore, we determine it at run-time.

Using this unification, the distinction between active and passive objects is no longer necessary. Also, objects can be modeled as mobile entities, so that each object can migrate between address spaces.

4 Cognac Architecture

4.1 Object Cluster and Metaobjects

To achieve the unification described in the previous section, we separate the concept of execution domain from object programming, and provide a way to execute a method as a passive object. Also, we introduce dynamic object scheduling control based on object status. This provides a mechanism of mutual exclusion. Our execution domain is called an *object cluster*. In an object cluster, member objects share the same address space and the same scheduling policy.

The object cluster includes some meta objects. These are: a *scheduler* metaobject, a *message queue* metaobject, and *object status* metaobjects. The scheduler metaobject determines the execution scheduling of member objects. the message queue metaobject is used to spool the incoming messages from outside the cluster; and the object status metaobjects hold the execution status of each member object in the cluster.

Membership in a cluster is initially determined from static information provided by the Cognac compiler. Membership can be dynamically changed by migrating an object to another cluster to reduce communication overhead. Dynamic object migration is managed by the object monitor metaobject.

These metaobjects are themselves supported by meta-meta objects which provide system wide resource management such as memory management and message handling between object clusters.

4.2 Interface to meta-meta system

The meta-meta system supports the execution of metaobjects, such as inter-meta space communication, memory management and naming management. Each object cluster has an interface to the meta-meta system. For example, the `Invoke` interface is provided to allow invocation of an object from outside of the cluster. This interface depends on the kind of model which the cluster supports.

4.3 Meta interface compatibility

To support object migration across object clusters, we introduce a notion of compatibility of interface to metaobjects. In Cognac, the class of an object cluster is defined in a hierarchical class system. A subclass object cluster supports operations which are defined in its super-class object cluster, so that objects can move to the subclass object cluster at any time. We call such compatibility *meta interface compatibility*, because the interface of the operation is compatible but the semantics of the operation might be different.

5 Implementation

A prototype version of Cognac has been implemented on top of UNIX. Each object cluster is implemented as a UNIX process, which contains the code for the necessary metaobjects. Since the meta-meta system is the UNIX kernel and the semantics of message

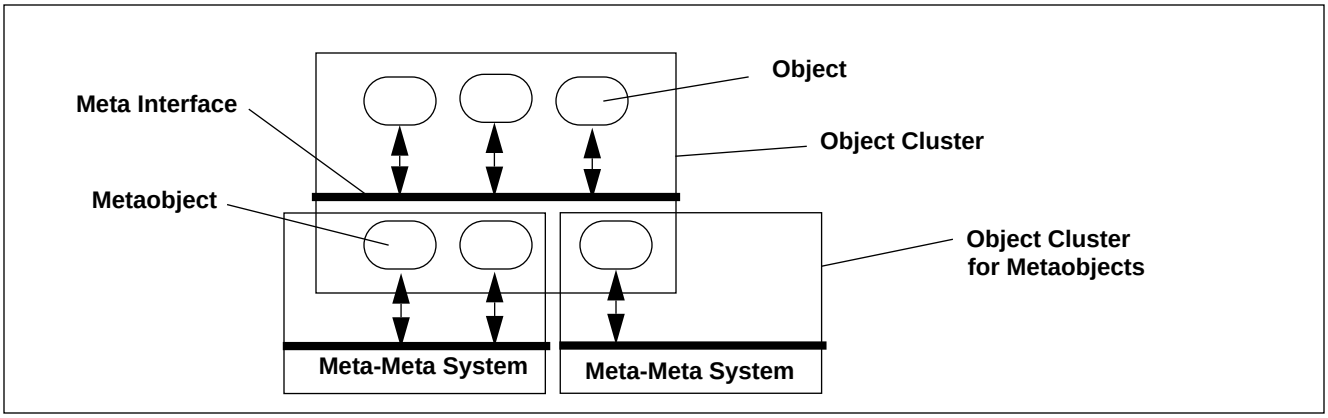

Figure 3: Structure of Cognac Object Clusters

sending between object clusters is determined by UNIX inter-process communication, all metaobjects must share the same meta-meta system. We now plan to implement Cognac on Apertos, which provides a hierarchical meta-space architecture, so that metaobjects can be programmed on top of different meta-meta spaces.

6 Related work

Apertos has a hierarchal meta architecture. A set of metaobjects is called a meta space and the meta space is managed by a special object named a *reflector* in Apertos. Since the Cognac design is based on Apertos, the reflector which defines the interface to the meta space for objects looks similar to the object cluster in Cognac. However, all Apertos objects are active, so that there is no direct invocation of objects. Also, reflectors manage not only object scheduling, but also the object's environment, including memory management, message delivering, and so on. In this sense, an Cognac object cluster is a subset of an Apertos reflector. We are now trying to merge Cognac into Apertos.

7 Future work

Our current goals are:

1. To implement Cognac on Apertos.
2. To establish the formal semantics of the unified object model.
3. To achieve reflection by dynamic modification of metaobject definition.
4. To implement an object migration support.

8 Conclusion

In this paper, we describe the unification of active and passive objects in object-oriented operating systems. This unification simplifies programming and simplifies the migration of objects between execution environments. A meta-level scheduling strategy is used to achieve unification. The scheduling policy is determined at run-time.

Acknowledgement

We would like to thank Mantis Cheng and Robert Bryce for helpful discussions.

References

[1] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.

[2] Yasuhiko Yokote. *Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach*, In Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS), 1993.

[3] Takao Tenma. *MC++ Language Manual*, In Apertos Manuals. 1994

[4] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. *The Clouds Distributed Operating System.* In Computer. pp34-44, Vol. 24, No. 11, November 1991.

[5] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. *AL-1/D: A Distributed Programming System with MultiModel Reflection Framework.* In Proceedings of the International Workshop on New Models for Software Architecture'92 Reflection and Meta-level Architecutre, November, 1992.