A computational model for an object-oriented operating system

Lourdes Tajes Martínez, Fernando Álvarez García, Marian Díaz Fondón, Darío Álvarez Gutiérrez y Juan Manuel Cueva L?ovelle

***Abstract*--The design of an object-oriented operating system involves the design of a model that governs the execution of the tasks. In this paper we show the design goals to build a computational model for an object-oriented operating system based in an object-oriented abstract machine. We propose to adopt an active object model and we consider features like uniformity, homogeneity or self-contained are essential for objects. Finally, our proposal is to achieve the above using reflectivity as a key characteristic of the system.**

***Index Terms*--Object-Oriented Operating System (*OOOS*), Object-Oriented Abstract Machine (*OOAM*), Object Computation, Object Synchronization, Active Object Model**

I. INTRODUCTION

The aim of the project Oviedo3[1] [1] is to develop an object-oriented integral system where every layer on the system is designed and developed using the object-oriented paradigm. The two lowest layers of Oviedo3 are an abstract machine named Carbayonia and an operating system, named SO4.

The main goal is the design of the operating system following the Object-Oriented paradigm in all the components of the system, including the computational subsystem. SO4 is intended to be an object-oriented OS that fulfils the following goals:

1. **Uniformity**. It must provide the object as the one abstraction and it must offer a vision of the system as a set of objects. These objects communicate with each other using a message-passing mechanism.
2. **Unique identity**, **homogeneity**, **self-contained** and **complete semantic** are some important characteristics we wish to impose on objects [2].
3. Besides, the OS transparently must achieve a set of important features: **security**, **persistence**, **distribution** and **concurrency**.

A part of the project will be the definition of the computational model offered by the operating system and the abstract machine. We try to endow the system with a concurrency model [3] that maximizes the parallelism degree in a secure way. We have to permit concurrency between objects, between methods of the same object and between several instances of the same method of an object, in a secure way.

In the next section the aims of the computational model are presented. Then the different alternatives in the object model are shown and, finally, we describe the way Carbayonia and SO4 cooperate to offer the necessary mechanisms to support the chosen model.

II. GOALS IN THE DESIGN OF THE COMPUTATION SYSTEM FOR SO4

SO4 is intended to be an object-oriented operating system. So, the computational support must be designed in a way that the object-oriented paradigm is respected. The design goals are [4]:

1. **Unique, simple and powerful abstraction***:* SO4 must provide the object as the only abstraction. The operating system does not support a process like abstraction that executes the methods provided by the objects. It must be the object itself that executes a method when another object requests it. Conceptually, this is a straightforward idea. But it must be supported by a powerful object abstraction. Each object must encapsulate the processing it is doing.

The authors are part of the staff of the Computer Science Department in the University of Oviedo. Facultad de Ciencias. C/Calvo Sotelo, s/n, Oviedo. Spain

[1] This work has been supported in part by the 2nd regional plan (FICYT) of research of the Principado de Asturias, Spain, under project PBP-TIC97-01 "Sistema Integral Orientado a Objetos: OVIEDO3"

2. **Inter-Object Concurrency**: SO4 must allow the possibility of several objects executing methods simultaneously.
3. **Intra-Object Concurrency**: Every object must be able to serve more than one method invocation (possibly several invocations to the same method) simultaneously.
4. **Object Autonomy**: Each object must maintain and protect its own internal state. So, if intra-object concurrency is allowed, each object will have to decide when to execute the requested methods. If the object has to serve more than one request and they are incompatible, the object must be able to decide, independently of any other object in the system, what to do. It must be able to delay or resume its work.
5. **Scheduling**: To achieve the idea of only abstraction that encapsulates the computation, every object has to fulfil two goals.
   Schedule jobs when a petition arrives. Each object must be able to start a new activity and to schedule the existing ones based on the knowledge of its own inner state.
   Schedule jobs when a reply arrives. The object must react when a reply arrives resuming the adequate job.
6. **Flexible computation mechanism**: Another goal is to offer the computational mechanism in a flexible way [5]. An object or set of objects will be able to adapt it to their particular needs.
7. **Portability**: This model is portable because it is based upon an abstract machine. This machine offers the basic computational mechanisms.

III. Object Model for Computation

When designing the computational model there are two different alternatives [3, 6]. They define two different models and two different sets of characteristics for the objects.

A. *Passive object Model*

This model divides the objects, from the computational point of view, in passive and active objects. The first ones only contain data and methods. They do not have any capability for the execution and, so, they depend on other objects for executing their methods. The second ones provide the computational ability and they are able to execute methods. They represent the execution flow and they can involve methods offered for more than one object. So, their life is independent from the objects they traverse in their execution.

B. *Active object model*

This model defines an object with more semantic content. The objects are defined in a homogeneous way like live entities which contains not only data and access methods but their own computing resources. That is, the objects are endowed with computational power.

In this model, the object is an autonomous entity which interacts with another objects in the system. The mechanism used for the communication is the message passing.

C. *Comparison*

The main advantage obtained by the adoption of an active object model is *homogeneity* [7]. The active model offers a powerful abstraction of the object that fits well with our goals. These objects will be homogeneous, that is, all the objects will have the same properties. Besides, the objects will be self-contained; every object must contain its own private activities.

Because homogeneity, the same communication mechanism can be used to communicate and synchronized any two objects.

In a passive object model, homogeneity is lost. Two kinds of objects are presented. Because this lack of the homogeneity, the interaction between two active objects can not be made with the same mechanism used to communicate two passive objects.

Object homogeneity provides a lot of additional advantages. The active object model is a powerful help to the distribution and persistent mechanisms. The active object model provides a compact view of the object. The computation in the object is transparent to those mechanisms. So, they do not need to develop special mechanisms to distribute or persist computation.

Another mechanisms as object migration and load balancing will benefit too.
The extension of the passive object model to a distributed environment is difficult. And

some form of hidden message passing will be necessary.

IV. COMPUTATIONAL MODEL: ADDING REFLECTIVITY

Computational system offered by the operating system must provide the mechanisms needed for scheduling, concurrency and synchronization. SO4 collaborates with Carbayonia to get all these mechanisms. SO4 extends the default behavior of Carbayonia in some areas like scheduling, concurrency control and method invocation.

A. Carbayonia Computation Mechanism

Carbayonia is the abstract machine that supports all the objects of the rest of the system and offers the basic object model. The architecture of the Carbayonia abstract machine consists of four areas: class area, references area, instance area, and system references area (See Figure 1). The main characteristic of the machine is that every action upon an object is made using a reference to it.

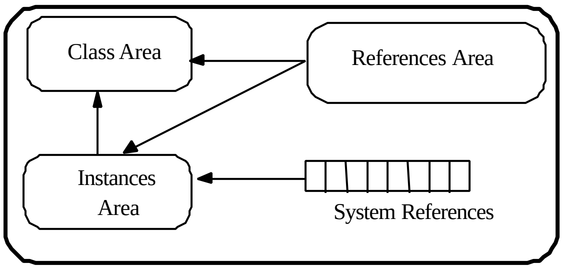

Figure 1. Architecture of the Carbayonia abstract machine.

1. **Class area**. Maintains the description of each class. There is a set of primitive classes defined permanently in this area.
2. **References area**. Stores the references. Every reference has a type (relates to the class area) and points to an object of this type (relates to the instance area).
3. **Instance area**. Stores objects created. At run time, the information of its class can be accessed in the class area.
4. **System references area**. Contains references with specific functions in the machine.

Each area can be considered as an object in charge of the management of its data. That is, Carbayonia is designed in a reflective way.

Carbayonia is the lowest support level. It provides the objects with the most basic mechanisms for the execution of their methods and the communication.

Because Carbayonia is an Object-Oriented Abstract Machine, it offers its mechanisms as a hierarchy of basic classes and a reduced instruction set. Carbayonia executes directly the methods of these basic classes without interruption and no synchronization is needed. Carbayonia maintains a basic class, named *thread* that represents the execution of a specific method in a specific object. The methods in this object are: *suspend*, *resume*, *start*, ... with the habitual meaning. The invocation of these methods allows the abstract machine to suspend, resume, ... the execution of a job giving the basis for concurrent execution. Specific policies can be implemented over this basic mechanism.

Each thread maintains the reference of the object in which it is executing a method, a reference to the method and the specific instruction Carbayonia must execute to complete this method.

So, when an object receives a new message, it can create a new thread to represent the execution of the request.

Threads are machine objects. They are implemented inside the abstract machine. The invocation of their run method provokes the machine starts to execute the specific method. So, threads are a mechanism the object can use to ask the machine to do something. (See figure 2).

To fulfil the self-contained goal, the objects maintain a list of references to their thread objects and manage them.

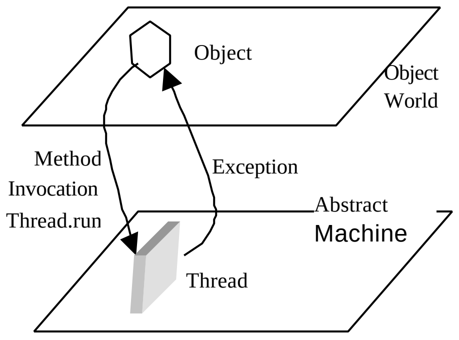

Figure 2. Abstract machine - object communication.

Carbayonia also offers a set of instructions. The most important for computations are those regarded to method invocation and exceptions.

1. **Message Passing***:* This instruction informs to an object that the execution of one of its methods the requested by another object. Conceptually, a new thread will be created in the target object to serve this request.
2. **Exception Mechanism***:* The machine can communicate with the objects with a mechanism similar to the message passing. This is the exception mechanism. The machine raises an exception when a specific condition is reached. This provokes the thread tries to capture the exception and handle it. If the current thread cannot manage the exception will be necessary to propagate it in the call-reply chain.

B. Reflectivity: Adding flexibility to the system

Traditionally, programs had to make its task in a limited computational environment. But, today applications present an increasing complexity. New applications need mechanisms for multithreading, distribution, persistence, migration, load balancing, etc.

Some ad-hoc extensions to traditional systems have been implemented for support these mechanisms by means of system calls or distinguished system objects. But this solution is quite inflexible. There must be a way to change the manner the operating system offers a specific functionality.

The applications, then, could define the behavior of its environment and modify or extend it if needed. Some systems like Exokernel [8] and SPIN [9] have explored this area. But we propose reflection as the way to get it [10].

Reflectivity can be expressed in two ways: *structural reflectivity* and *behavioral* or *computational reflection. Structural reflection* reifies structural aspects of a program like inheritance. An example is the Java Reflection API [11].

Computational reflection reifies the computational aspect and defines the environment where the base actions are executed. This is the most interesting aspect for us. The *meta-object* concept is basic to express computational reflection [12]. Since the meta-objects know the internal organization and structure of application-defined objects or base objects, there are able to control certain aspects of the execution of base-objects.

C. Computational Model in SO4

The environment where object activities are executed must provide the objects some abilities. These abilities traditionally, were part of the operating system functionality. Our approximation is very different because we think reflectivity can help us to get a flexible environment for objects.

Carbayonia offers the basic mechanisms to implement a computational system: communication (*Call instruction*), scheduling (time interrupts) and concurrency (machine objects called thread) but the exact behavior of the machine can be modified using reflection.

Reflectivity is used to extend the machine behavior and is achieved extending object concept and giving it more semantic meaning.

The key idea in the design will be to divide the object vision in two levels: base level and meta-level. Base level is composed of application-defined of base objects. Base objects are the unit of computation in SO4.

The meta-level will consist of a set of meta-objects that control the way basic-objects are executed. SO4 will implement meta-objects as objects so uniformity is maintained.

The set of objects, which transform an object in a superobject, will be named the *object environment.*

D. SO4 Object Environment

The key idea in the design will be to *divide the object world in two levels*: base level and meta-level (See figure 3).

Each object in the base level will be supported by a set of meta-objects. Each one of them will describe some aspect of the base-level behavior of the object. This is a popular idea already studied in [13] and [14].

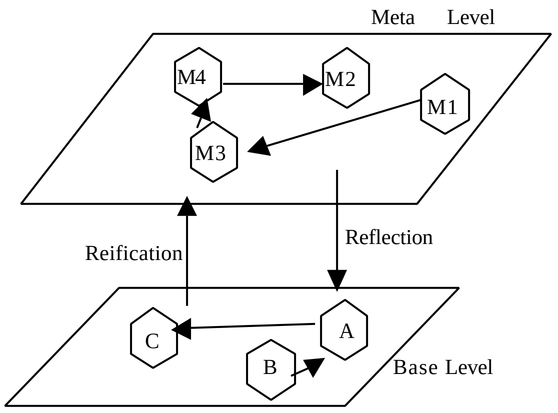

Figure 3. Behavioral Reflection. The computation of objects A, B and C transfers control to the meta-level

In SO4, the object environment can be subdivided in three different meta-objects:

1. *Scheduler meta-object.* It contains a set of ready threads in the object. It can schedule them and decides which one is the most adequate to execute.
2. *Communication meta-object.* It takes charge of sending and receiving messages. It can propagate exceptions.
3. *Synchronization meta-object.* It decides when is secure to pass a thread to the scheduler meta-object or when is convenient to delay a thread.

Conceptually, each object asks its meta-objects to do something, sending a message to it. The methods in the meta-object are executed in a sequential, not interruptible and exclusive way. So, corruption of the inner state is avoided and secure processing of the messages are guaranteed.

To avoid the overhead caused for too many threads in the base and meta-level and for the message-passing operations to communicate the base-level with the meta-level, the same thread that is executing the base-level method can execute the meta-level method too.

E. *Scheduler meta-object*

Scheduler objects will realize scheduling tasks taking charge of managing the ready threads in the object. It manages the threads waiting in a synchronous invocation too. The set of threads is divided into two disjoint sets and the threads can be moved between these two sets.

1. *List of ready threads.* Some scheduling policy will be applied over the threads in this list. The scheduling policy is particular to each object. Even, it is possible for an object to decide to avoid the overhead caused by a private scheduler. So, the object can delegate in a global scheduler object or in a per-application scheduler object the task to schedule its threads.
2. *List of waiting threads.* These threads will be waiting in a synchronous call. They will be moved to the above list when the reply arrives.

These scheduler objects are not different of any other objects in the system because the homogeneous object model provided. So, it is possible to insert new scheduler objects to provide scheduler functionality to a particular set of objects. It is possible to change a scheduler object for another, which implements a different policy. Even, it is possible to remove a scheduler object.

This meta-object communicates with the high-level scheduler object and with the abstract machine. When the high-level scheduler decides one object is the most adequate to execute, the execution meta-object of the chosen object will decide which thread to execute. This meta-object will send the message run to the thread object. The abstract machine will start to execute the instructions pointed by this thread.

F. *Communication meta-object.*

It takes charge to send and receive the messages and to manage them. Each message can be a request, a reply or an exception and there is only one method to manage all of them.The propagation of the exceptions is considered as a particular case of the call-reply chain.

This meta-object builds the message and executes the adequate instruction machine.

Hand-off is used in the message passing. So, when an object sends a message to other, the first will give up some time to the second. The target object will execute the operations needed to receive a message. It must evaluate its inner state and decide to create a thread to serve the request or delay it.

G. *Synchronization meta-object*

Two synchronization mechanisms are offered to an object. The first is a coarse-grained mechanism. It establishes exclusion between incompatible methods. A similar idea has been implemented in Guide [15].

The second is a fine-grained one. The operating system offers a semaphore class with the typical functionality and semantic.

Synchronization meta-object covers the first option. It maintains, for each object method, m, a list of other methods M. M represents the set of methods whose execution is incompatible with the execution of m. To avoid dangerous executions, when a method is invoked in an object, this meta-object decides to execute or to delay it, based in the information above.

When a thread finishes, the synchronization meta-object tests if any other thread in the specific list can continue. This meta-object can specify a special order to evaluate the messages. So, user objects construct some aspects of the abstract machine. For example, when some instructions of the abstract machine (Call) are executed, the actions for this instruction are specified for a meta-object. (Figure 4).

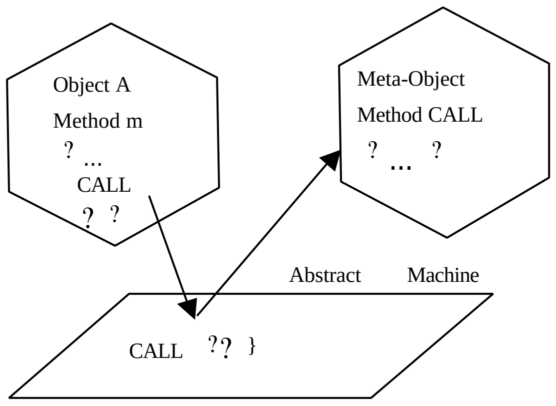

Figure 4. The object environment extends the abstract machine

V. SCHEDULING

By default, there is a general scheduler object in the system. The abstract machine knows it and its reference is one of the system references. The abstract machine will give it quanta of time. The scheduler distributes this time following a specific and modifiable policy. It is possible to insert new schedulers that implement a special policy.

Each scheduler maintains a set of thread objects and it gives up machine time to them. Each thread can be serving a request made by an object or it can be doing inner tasks in the object, for example, inner scheduling.

In the same way, every scheduler object installed by a subsystem or application, will be an object schedulable by other schedulers. The high-level scheduler will give execution time to them. And they distribute it between their objects.

At the lower level, even each object can have an inner scheduler to schedule its own threads following a specific policy. But, if the object decides to avoid this overhead, it can employ a pre-existent scheduler.
A scheduler hierarchy is created in a transparent way [16, 17].

When the time finishes, the thread will return the machine control to the scheduler that gave it the time.

VI. CONCLUSIONS

This model provides an important advantage. Because the computational model will be expressed providing some special characteristics to the objects, every object will be able to manage its own computation (active object model). And because the special characteristics are offered as a set of objects, uniformity is maintained.

Computation system is a key component of any object-oriented operating system.

Because the uniform application of the object-oriented paradigm in all the components of the system, we endow the objects with computational power. An environment or set of objects will support each object. These objects provide the basic behavior to the base object and allow it to execute, synchronize and communicate.

Reflective architecture of the operating system is a promising way to provide a flexible, modifiable and extensible environment.

VII. REFERENCES

[1] Juan Manuel Cueva Lovelle and others. Sesion ?Sistemas Operativos Orientados a Objetos: Seguridad, Persistencia, Concurrencia y Distribucion? (Object-Oriented Operating Systems: Security, Persistence, Concurrency and Distribution). II Jornadas sobre Tecnologias Orientadas a Objetos. Oviedo, Spain, March 1996. (in spanish).

[2] Fernando Alvarez Garcia, Dario Alvarez Gutierrez, Lourdes Tajes Martinez, Marian Diaz Fondon, Raul Izquierdo Castanedo, Juan Manuel Cueva Lovelle. An Object-Oriented Abstract Machine as the substrate for an Object-Oriented Operating System. Workshop on Object-Orientation and Operating Systems. 11th European conference on Object-Oriented Programming (ECOOP'97). Jyvaskyla (Finland), June 1997.

[3] R. S. Chin, S. T. Chanson. Distributed Object-Based Programming Systems. ACM Computing Surveys. 23(1): 91--124, March 1991.

[4] O. Nierstrasz. Composing Active Objects. The Next 700 Concurrent Object-Oriented Languages. In Research Directions in Object-Based Concurrency. Ed. G. Agha, P. Wegner, A. Yonezawa. MIT Press, 1993.

[5] V. Cahill. Flexibility in Object-Oriented Operating Systems: A Review. 3rd CaberNet Radicals Workshop. Connemara (Ireland), May 1996.

[6] J-P. Briot, R. Guerraoui. A classification of various approaches for Object-Based parallel and Distributed Programming. Technical Report. University of Tokyo and Ecole Polytechnique Federale de Lausanne, 1996.

[7] O. Nierstrasz. A Survey of Object-Oriented Concepts. In OO Concepts, Databases and Applications. Ed. W. Kim and F. Lochovsky. ACM Press and Addison-Wesley, 1989.

[8] D. R. Engler, J. O?Toole Jr., F. Kaashoek. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Proceedings of the Fifteenth Symposium on Operating Systems Principles, December 1995.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. Proceedings of the Fifteenth Symposium on Operating Systems, 1995.

[10] G. Kiczales, J. Lamping. Operating Systems: Why Object-Oriented?. In Proceedings of the 3rd Workshop on Object Orientation in Operating Systems (IWOOOS'93), 1993.

[11] URL: http://java.sun.com/docs/books/ tutorial/reflect/TOC.html

[12] P. Maes. Concepts and Experiments in Computational Reflection. In Proceedings of the 2nd Conference on Object-Orientation Programming Systems, Languages and Applications (OOPSLA'87), 1987.

[13] M. Golm, J. Kleinoder. MetaJava- A platform for adaptable Operating-System Mechanisms. Workshop on Object-Orientation and Operating Systems. 11th European conference on Object-Oriented Programming (ECOOP ?97). Jyvaskyla (Finland), June 1997.

[14] J. McAffer. Meta-Level Programming with CodA. 9th European conference on Object-Oriented Programming (ECOOP ?95). Aarhus (Denmark), 1995.

[15] M. Riveill. Synchronising Shared Objects. Distributed Systems Engineering Journal. 2(2), 1995.

[16] T. Riechmann, J. Kleinoder. User-Level Scheduling with Kernel Threads. Technical Report TR-I4-96-05. Computer Science Department. Friedrich-Alexander-University, 1996.

[17] B. Ford, S. Susarla. CPU Inheritance Scheduling. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'96), 1996.