An Environment to Support Fragmented Active Objects

L. Courtrai J.F. Roos* J.M. Geib J.F. Méhaut

Laboratoire d'Informatique Fondamentale de Lille - URA 369 CNRS
Université des Sciences et Technologies de Lille
Bâtiment M3 - F - 59655 Villeneuve d'Ascq Cedex (France)
Phone: (+33) 20 43 47 27 - Fax: (+33) 20 43 65 66
E-mail: {courtrai roos geib mehaut}@lifl.fr

Abstract

This paper describes a medium level tool to implement active objects. This programming environment uses Communicating Active Components (Cac/s) and hides low-level mechanisms of the target machine for the programmer. An object-based concurrent application may be defined as a set of Cac/s. We propose several distributed representations of fragmented active objects. Our goal is intra-object parallelism and distribution.

1 Introduction

Highly decentralized architectures, especially multicomputers[1], are very promising. But they need high level tools for users and programmers. Many researches exhibit the Active Object notion as a very structuring tool to control the design of parallel programs and the exploitation of these programs on parallel machines. Concurrency and distribution may (must?) be object-oriented!

Many Object-Oriented Parallel Languages (for example *Pool* [3], *Act++* [13], *Guide* [14] and *Parallel-Eiffel* [11]) were implemented with the UNIX[2] operating system. The UNIX choice was justified by the development facilities, the network software and the portability. But the assessment of the prototypes shows some performances limitations due to this system (for example the memory management or the communication technique). A new approach ([5], [12]) uses a microkernel (*Mach* [1] or *Chorus* [17]). Consequently, the designer and the implementer of active objects based high level software need some general features issued from a run-time support [12] [15]. This run-time must provide an "*object abstraction*" between the Operating System Kernel and a specific implementation of active objects. In this paper, we propose a medium level structuring run-time. We call it the Cac model, it introduces **Communicating Active Components**. The Cac model unifies objects, processes and communications in a single entity. This level is easily constructed onto the standard mechanisms of operating systems, taking advantage of modern functionalities like the kernel distribution. It supports various active object models. It is well suited for medium grained distribution.

From the Cac environment, we implement an Object layer. Our Active Objects may be fragmented to allow a high parallelism degree. The fragments may be distributed over the nodes. Our work is a part of the **Vcp** project [8].

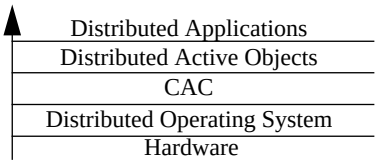

figure 1.0 Vcp Architecture

1.1 Active Objects

Active objects integrate processes in the object structure. Each object represents an autonomous task of an application, executed by an internal process. Processes are created at instantiation time, they run until deletion time. An important advantage is that processes and objects are manipulated as a whole[3] by the application programmer. So the active object notion is the unique structuring tool for data and activities of parallel applications.

1. A multicomputer is a M.I.M.D. (Multiple Instruction Multiple Data) computer without shared memory.
2. UNIX is trademark of AT&T Bell Laboratories.
3. In the family of Object-Oriented Parallel Languages, an other (and oldest) approach is to separate object and process into two distinct entities.

* This work is supported by «Région Nord/Pas de Calais» (n° 91030068) in R&D program on Advanced Communication «Ganymède».

There are many works on Active Objects. We propose a classification in accordance of the two following criteria: 1) Is the internal activity mono or multi-threaded ? 2) Is the object representation compact or distributed ?

Single-threaded active objects: In a first model, a process, linked to an object, executes the method invocations. This process manages the parallel environment of the object: the object receives concurrent requests from other objects and serializes their treatments to ensure its integrity.

Actors [2] are also active objects. Actors cooperate by sending messages to actor mailboxes. The process of an actor runs the actor script and sequentially processes the messages of its mailbox.

The parallelism degree can be defined by the number of created processes. With single-threaded objects, this degree equals the number of created objects. There is only **inter-object parallelism**.

Multi-threaded active objects: A problem related to single-threaded objects is the strong serialization of the methods processing. One object can be a bottleneck in an application. The **intra-object parallelism** clears up this problem and increases the application parallelism. Now, an Object can concurrently respond to several requests and several processes will be concurrently active in the object. The parallelism degree is greater than the object number.

Fragmented multi-threaded objects: The representation of single-threaded objects is conventionally compact: the code and the object attributes are gathered in single chunks of memory. There is no **intra-object distribution**, only **inter-object distribution**. An object is fully located on one node The code can be duplicated on several sites [3].

The intra-object parallelism of multi-threaded objects will be real if an object may be split into fragments distributed on different nodes at execution time [4],[18]. Each fragment contains a part of the object representation, so the set of fragments constitutes the object. Processing is really concurrent inside the different fragments of the object.

What we have presented leads to the hierarchical classification of active object models in fig. 1.4. The distributed multi-threaded active object model seems to be the most promising for parallel programs on parallel machines. Nevertheless, we think that a powerful environment must simultaneously provide the different kinds of active objects behind a uniform protocol, i.e. the internal representation must only depend specific needs. In the second part we introduce the Cac model to support the different active object models.

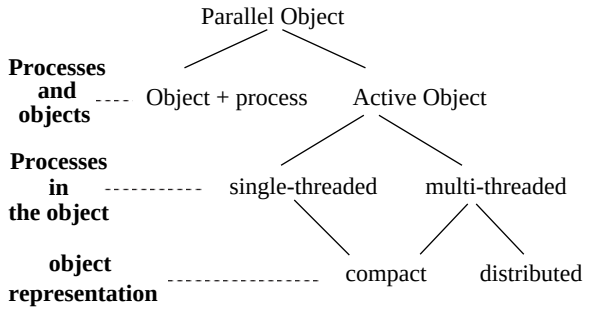

figure 1.1 Classification of Parallel Objects

2 Distributed representations of Objects

In this part, we describe a medium level tool to implement active objects. This programming environment uses *Communicating Active Component*s (Cac/s) and hides low-level mechanisms of the target machine for the programmer. An object-based concurrent application may be defined as a set of Cac/s. After the introduction of the Cac structure, we propose several distributed representations of active objects. Our goal is intra-object parallelism and distribution. So the different aspects of an active object (naming, instance variables, activities) are viewed as potential units of parallelism.

2.1 Communicating Active Components

We introduce a unique structuring entity for multicomputer programming: we call it **Cac** (**Communicating Active Component**) [8,9,10].

The **C**ommunicating **A**ctive **C**omponent stands for the concurrency unit of an application. A Cac is designed by its behavioral function and local data. A Cac is an active and single-threaded structure. It communicates with other Cac/s by message sending.

The component structure is close to the serialized actor structure defined by Agha [2]. A communicating active component is made up of three parts: a **process** (processing part), a **mailbox** (communicating part) and a **local environment** (local memory part) (figure 2.0).

The process: Each component has an autonomous activity running as an internal process. The process code represents the behavior of the component. This activity must be designed by the programmer as a behavioral function which will handle the local environment and the communications with the other components. The Cac type denotes its behavioral function.

The mailbox: All communications between components use a specialized communication structure. This structure, linked with each component, is a mailbox which stores all the messages for the component. A communication is asynchronous and consists of sending a message to the mailbox of the recipient component. The system takes the message, unlocks the sender, and carries the message to the receiver's mailbox. The message passing is reliable. The process of the component may extract messages from its mailbox. A mailbox is created at component creation time and its name identifies the component in the system.

The local environment: The local environment is the local memory of the component. It gathers all the local data (variables or local functions). This environment is dynamically created by the component using two run-time calls: memory allocation and memory liberation. Only the owner Cac can access this local memory and a protection mechanism of local data is not necessary.

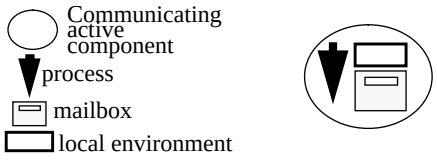

figure 2.0 Structure of Cac

The Cac model is independent of the target machine. On a parallel machine, the Cac/s are distributed over the different nodes. With the following basic requirements:

- 1 - The node must support the structure of Cac. The node's operating system must be multi-tasked for to receive many Cac/s. The mailboxes and the local environments of Cac/s are allocated on the node memory.

- 2 - A Cac is located on a node of a machine. It cannot migrate to another node.

The Cac structure is easily realizable on a modern distributed operating system.

A Cac implementation was realized onto a Transputer-based multicomputer: the Multicluster-2 by Parsytec under Helios Operating System [16]. The Cac prototype uses the main functionalities of Helios: task manager, memory allocation, process creation and communication by pipes. The distribution step is achieved by a specialization of the Component[4] Distribution Language (CDL) of the system. This implementation and its performances are described in [10].

A second prototype is being developed on a network of SUN workstations by using UNIX processes, lightweight processes [19] and sockets. We also plan an implementation of a Cac run-time on a microkernel (*Mach* or *Chorus*).

The object interface: From the Cac/s, we propose several distributed representations for the single and the multi-threaded objects. A Cac is not a complete active object, it is just a "proto-active-object". An active object uses a standard interface allowing it to communicate. This interface is made up of 1) a **name** which is used to communicate with the object without knowledge about its representation, and 2) a **method invocation protocol** which is used to express method requests.

All of our objects have got a uniform interface. So objects of different representations can exist together in the same application:

- First, each object is linked to a specific Cac, called *Orc* (Object Representation Component) which identifies the object. So the name of the object is always that of its *Orc*.

- Second, each method call is materialized by a message sending to the *Orc* of the target object. The message structure is made up of a **method** name and an argument list (**args**). The message also contains the calling method name or by continuation another **proxy** method. Two invocation modes are possible: *synchronous call* which blocks the calling method, and *asynchronous call* which does not block the calling method (the calling and requested methods run concurrently).

2.2 Single-threaded active objects

A single-threaded object sequentially processes the incoming requests. It can be represented by one Cac, which is a single-threaded structure. Such an object is a specialization of a Cac with the same behavior.

The representation uses the *Orc* of the object interface: the *Orc* contains the local attributes and it processes the external (method invocations) and internal (self invocations) requests.

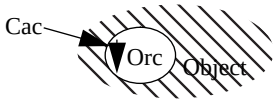

figure 2.1 The Object representation component

The *Orc* component is a Cac specialization. Each part of the Cac structure materializes a functionality of a single-threaded object. The *Orc* structure contains:

- 1 - A **mailbox** containing all requests to this object.

- 2 - A local **environment** containing all instance variables.

4. In this context, this word is synonym of Helios task.

- 3 - A **process** which initializes the object, extracts the messages, executes the methods, and manages the instance variables located in the local environment.

The designer of an object defines the list of instance variables and the set of methods. In our environment the code of the *Orc* Cac for single threaded objects is predefined according to our object interface.

2.3 Multi-threaded active objects

The structure of a multi-threaded object is more complex than a Cac structure: these active objects contain several activities when a Cac is single-threaded. So, now each active object will be materialized by a dynamic gathering of Cac/s. Cac/s do not share memory, so the object representation (i.e Cac/s) may be distributed in the memory.

Remote method execution: The *Orc* of a multi-threaded active object is the root Cac. The *Orc* does not execute the method executions itself, but it delegates these executions to specific Cac/s, called Method Execution Components (*Mec*). The *Orc* processes an incoming message by creating a *Mec* which executes the requested method. The local variables of the method are stored in the local environment of the *Mec*. The *Mec* exists during the method execution and is destroyed after the method execution. The separation of the method execution and the request reception leads to intra-object parallelism.

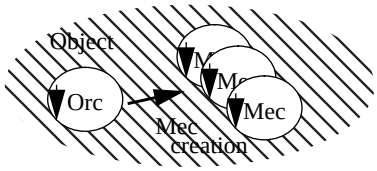

figure 2.2 The Method execution component

The *Mec/s* of an object are distributed on the nodes of the target machine. The object is distributed during the execution, moreover it dynamically changes according to the method executions on the object. The object is not a compact entity, it is a dynamic aggregate of Cac/s.

Remote access to instance variables: The instance variables of an object are stored in the *Orc* local environment. When a method wants to access an instance variable, the *Mec* must cooperate by messages with its *Orc*. This creates a communication overhead, and furthermore, an *Orc* may become a bottleneck in the object.

We have two answers for this problem. First the communication overhead is balanced by the intra-object real parallelism. Second, because the *Orc* creates the Mec/s and also manages the instance variables, we can extract one of these two tasks from this component and give this task to a specific Cac.

We introduce a new Cac type that manages instance variables. We call it an Attribute Manager Component (*Amc*). Each request by a *Mec* to an instance variable is processed by an *Amc* (fig 2.3). Note that an *Amc* may be located on any machine node.

An *Amc* serializes requests to instance variables. This avoids the use of a bottom level synchronization: the instance variables are not directly accessible by the *Mec*/s. More, an *Amc* may implement any specific strategy to protect or reserve some data for a particular use.

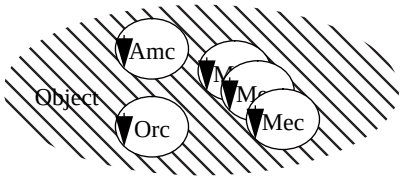

figure 2.3 The Attributes manager component

2.4 Fragmented multi-threaded active objects

We can go a step further in intra-object distribution. In the section 1.3, we have presented several interests of object fragmentation. The processing in our multi-threaded active object is already distributed, but the representation of instance variables remains compact in the *Amc*. This is true if a single *Amc* is used. Several *Amc/s* coexist in our fragmented multi-threaded active objects. Each *Amc* manages some instance variables. The *Amc*/s are created at object instantiation, and the *Amc*/s are known by the related *Mec*/s which can directly communicate with them.

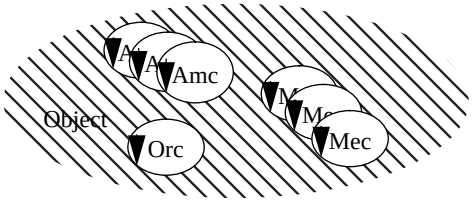

figure 2.4 A fragmented object

The creation of several sub-sets of the instance variables in different *Amc/s* presents other interests. First, this allows

concurrent access to different instance variables, so addressing the previous bottleneck problem.

Second, that also allows to regulate in some way the distribution of the components: *Amc/s* and *Mec/s* may be coupled at the distribution step to preserve privileged relationships between some of them. So we optimize the access time to instance variables and the global time of the application.

Finally, that also introduces the concept of active fragment: a fragment is often an active entity in an object. This entity may be defined by the programmer as an is-part-of component of the final object (Fragment-Oriented Parallel Languages). In our environment, these fragments can be represented by the union of an *Amc* and a *Mec*. A fragment is made up of instance variables (local data of the *Amc*), a name which identifies the fragment (the *Amc* name) and an activity (the *Mec*).

3 Conclusion

Operating Systems Object Layers must cope with different kinds of active object models. We introduce the Communicating Active Component to abstract the medium level between Object-Oriented Parallel Languages and Operating Systems. This tool is easily built on any distributed operating system providing lightweight processes, memory management and a communication protocol. Because it hides low levels, the Cac run-time increases the portability of parallel software, only the Cac run-time needs to be rewritten to execute Cac applications on another operating system.

A large scope of applications are described by the Cac model: parallel applications and especially run-times of concurrent languages for which the Cac environment is an efficient support for active object design. We mix several distributed representations of active objects. These representations preserve the design parallelism and create a real intra-object parallelism. Our current prototype integrates an execution model with method invocations on distributed objects. The first measures confirm the good properties of these representations.

Our current work concerns a high level Object-Oriented Parallel Language (called *Vcp)* for multicomputers. This future language uses fragmented multi-threaded active objects. We are also focusing on the synchronization constraints in this language [6].

References

[1] M.J. Accetta, R.V. Baron, D.B. Golub, R.F. Rashid, A. Tevanian, M.W. Young *Mach: A New Kernel Foudation for UNIX Developpment* in Proceedings of Summer Usenix, July 1986

[2] G. Agha, *Actors. A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1986, 144 Pages

[3] P. America, *POOL-T : A Parallel Object-Oriented Language*, in *Object-Oriented Concurrent Programming*, The MIT Press, 1987, pp. 199-220

[4] J.P. Banâtre, M. Banâtre, *Les systèmes distribués - Expérience du Projet GOTHIC*, InterEditions 1991

[5] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont, *Supporting an object-oriented distributed system: experience with UNIX, Mach and Chorus*, BULL-IMAG Technical Report 7-90, December 1990

[6] L. Courtrai, J.M. Geib, J.F. Méhaut, *Inheritance of Synchronization Constraints in the VCP system*, in EastEurOOPe'91 Proceedings, pp. 57-60, September 1991

[8] L. Courtrai, *Les Composants Actifs de Communication: outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD*, Thèse de doctorat de l'Université de Lille I (F), 1992

[9] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Méhaut, *The implementation of Actor environment on transputer under Helios: The Communicating Active Components*, Poster, in Transputers'92, Besançon (F), May 1992

[10] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Méhaut, *Communicating Active Components: An environment for concurrent applications on parallel machines*, accepted in EUROMICRO'92 to appear, Paris, September 1992

[11] C. Gransart, J.M. Geib, *Reusability and Concurrency in Parallel Eiffel*, X ième international Eiffel User Conference, Dortmund (Germany), April 1992

[12] S. Habert, L. Mosseri, V. Abrossimov *COOL: Kernel Support for Object-Oriented Environments*, OOPSLA/ECOOP'90 Conf. on Object-Oriented Programming, European Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990, pp 269-277

[13] D.G. Kafura, K.H.Lee, ACT++: Building a Concurrent C++ with Actors, JOOP Journal of Object-Oriented Programming, May/June 1990 , pp. 25-37

[14] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, *Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications*, JOOP Journal of Object-Oriented Programming, September/October 1990 , pp. 11-21

[15] J. Alves Marques, P. Guedes, *Extending the Operating System to Support an Object-Oriented Environment*, OOPSLA'89 Conf. Proc., Object-Oriented Programming Systems, Languages, and Applications, New Orleans, October 1989, Special issue of SIGPLAN Notices, Vol. 24, N° 10, October 1989, pp. 113-122

[16] Perihelion Software, *The HELIOS Operating System*, Prentice-Hall, 1989

[17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser, *The Chorus distributed operating system* ,Chorus Distributed Operating System, Chorus System, february 1989, pp. 305-370

[18] M. Shapiro, Y. Gourhant, S. Habert, L. Misseri, M. Ruffin, C. Valot, *SOS: An object-oriented operating system - assessment and perspectives*, Computing Systems, 2(7), 1989, pp. 287-337

[19] Sun Microsystems, *System Services Overview. Chapter 6 : Lightweight Processes*,1988, Part Number 800-1753-10

An Environment to Support Fragmented Active Objects

L. Courtrai J.F. Roos* J.M. Geib J.F. Méhaut

Laboratoire d'Informatique Fondamentale de Lille - URA 369 CNRS
Université des Sciences et Technologies de Lille
Bâtiment M3 - F - 59655 Villeneuve d'Ascq Cedex (France)
Phone: (+33) 20 43 47 27 - Fax: (+33) 20 43 65 66
E-mail: {courtrai roos geib mehaut}@lifl.fr

Abstract

This paper describes a medium level tool to implement active objects. This programming environment uses Communicating Active Components (Cac/s) and hides low-level mechanisms of the target machine for the programmer. An object-based concurrent application may be defined as a set of Cac/s. We propose several distributed representations of fragmented active objects. Our goal is intra-object parallelism and distribution.

1 Introduction

Highly decentralized architectures, especially multicomputers[1], are very promising. But they need high level tools for users and programmers. Many researches exhibit the Active Object notion as a very structuring tool to control the design of parallel programs and the exploitation of these programs on parallel machines. Concurrency and distribution may (must?) be object-oriented!

Many Object-Oriented Parallel Languages (for example *Pool* [3], *Act++* [13], *Guide* [14] and *Parallel-Eiffel* [11]) were implemented with the UNIX[2] operating system. The UNIX choice was justified by the development facilities, the network software and the portability. But the assessment of the prototypes shows some performances limitations due to this system (for example the memory management or the communication technique). A new approach ([5], [12]) uses a microkernel (*Mach* [1] or *Chorus* [17]). Consequently, the designer and the implementer of active objects based high level software need some general features issued from a run-time support [12] [15]. This run-time must provide an "*object abstraction*" between the Operating System Kernel and a specific implementation of active objects. In this paper, we propose a medium level structuring run-time. We call it the Cac model, it introduces **Communicating Active Components**. The Cac model unifies objects, processes and communications in a single entity. This level is easily constructed onto the standard mechanisms of operating systems, taking advantage of modern functionalities like the kernel distribution. It supports various active object models. It is well suited for medium grained distribution.

From the Cac environment, we implement an Object layer. Our Active Objects may be fragmented to allow a high parallelism degree. The fragments may be distributed over the nodes. Our work is a part of the **Vcp** project [8].

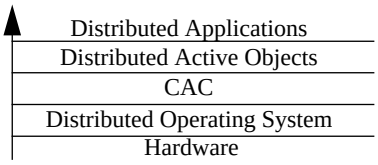

figure 1.0 Vcp Architecture

1.1 Active Objects

Active objects integrate processes in the object structure. Each object represents an autonomous task of an application, executed by an internal process. Processes are created at instantiation time, they run until deletion time. An important advantage is that processes and objects are manipulated as a whole[3] by the application programmer. So the active object notion is the unique structuring tool for data and activities of parallel applications.

1. A multicomputer is a M.I.M.D. (Multiple Instruction Multiple Data) computer without shared memory.
2. UNIX is trademark of AT&T Bell Laboratories.
3. In the family of Object-Oriented Parallel Languages, an other (and oldest) approach is to separate object and process into two distinct entities.

* This work is supported by «Région Nord/Pas de Calais» (n° 91030068) in R&D program on Advanced Communication «Ganymède».

There are many works on Active Objects. We propose a classification in accordance of the two following criteria: 1) Is the internal activity mono or multi-threaded ? 2) Is the object representation compact or distributed ?

Single-threaded active objects: In a first model, a process, linked to an object, executes the method invocations. This process manages the parallel environment of the object: the object receives concurrent requests from other objects and serializes their treatments to ensure its integrity.

Actors [2] are also active objects. Actors cooperate by sending messages to actor mailboxes. The process of an actor runs the actor script and sequentially processes the messages of its mailbox.

The parallelism degree can be defined by the number of created processes. With single-threaded objects, this degree equals the number of created objects. There is only **inter-object parallelism**.

Multi-threaded active objects: A problem related to single-threaded objects is the strong serialization of the methods processing. One object can be a bottleneck in an application. The **intra-object parallelism** clears up this problem and increases the application parallelism. Now, an Object can concurrently respond to several requests and several processes will be concurrently active in the object. The parallelism degree is greater than the object number.

Fragmented multi-threaded objects: The representation of single-threaded objects is conventionally compact: the code and the object attributes are gathered in single chunks of memory. There is no **intra-object distribution**, only **inter-object distribution**. An object is fully located on one node The code can be duplicated on several sites [3].

The intra-object parallelism of multi-threaded objects will be real if an object may be split into fragments distributed on different nodes at execution time [4],[18]. Each fragment contains a part of the object representation, so the set of fragments constitutes the object. Processing is really concurrent inside the different fragments of the object.

What we have presented leads to the hierarchical classification of active object models in fig. 1.4. The distributed multi-threaded active object model seems to be the most promising for parallel programs on parallel machines. Nevertheless, we think that a powerful environment must simultaneously provide the different kinds of active objects behind a uniform protocol, i.e. the internal representation must only depend specific needs. In the second part we introduce the Cac model to support the different active object models.

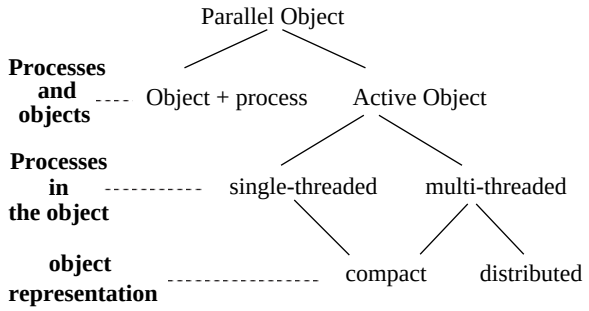

figure 1.1 Classification of Parallel Objects

2 Distributed representations of Objects

In this part, we describe a medium level tool to implement active objects. This programming environment uses *Communicating Active Component*s (Cac/s) and hides low-level mechanisms of the target machine for the programmer. An object-based concurrent application may be defined as a set of Cac/s. After the introduction of the Cac structure, we propose several distributed representations of active objects. Our goal is intra-object parallelism and distribution. So the different aspects of an active object (naming, instance variables, activities) are viewed as potential units of parallelism.

2.1 Communicating Active Components

We introduce a unique structuring entity for multicomputer programming: we call it **Cac** (**Communicating Active Component**) [8,9,10].

The **C**ommunicating **A**ctive **C**omponent stands for the concurrency unit of an application. A Cac is designed by its behavioral function and local data. A Cac is an active and single-threaded structure. It communicates with other Cac/s by message sending.

The component structure is close to the serialized actor structure defined by Agha [2]. A communicating active component is made up of three parts: a **process** (processing part), a **mailbox** (communicating part) and a **local environment** (local memory part) (figure 2.0).

The process: Each component has an autonomous activity running as an internal process. The process code represents the behavior of the component. This activity must be designed by the programmer as a behavioral function which will handle the local environment and the communications with the other components. The Cac type denotes its behavioral function.

The mailbox: All communications between components use a specialized communication structure. This structure, linked with each component, is a mailbox which stores all the messages for the component. A communication is asynchronous and consists of sending a message to the mailbox of the recipient component. The system takes the message, unlocks the sender, and carries the message to the receiver's mailbox. The message passing is reliable. The process of the component may extract messages from its mailbox. A mailbox is created at component creation time and its name identifies the component in the system.

The local environment: The local environment is the local memory of the component. It gathers all the local data (variables or local functions). This environment is dynamically created by the component using two run-time calls: memory allocation and memory liberation. Only the owner Cac can access this local memory and a protection mechanism of local data is not necessary.

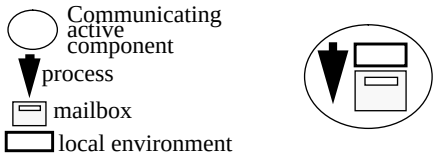

figure 2.0 Structure of Cac

The Cac model is independent of the target machine. On a parallel machine, the Cac/s are distributed over the different nodes. With the following basic requirements:

- 1 - The node must support the structure of Cac. The node's operating system must be multi-tasked for to receive many Cac/s. The mailboxes and the local environments of Cac/s are allocated on the node memory.

- 2 - A Cac is located on a node of a machine. It cannot migrate to another node.

The Cac structure is easily realizable on a modern distributed operating system.

A Cac implementation was realized onto a Transputer-based multicomputer: the Multicluster-2 by Parsytec under Helios Operating System [16]. The Cac prototype uses the main functionalities of Helios: task manager, memory allocation, process creation and communication by pipes. The distribution step is achieved by a specialization of the Component[4] Distribution Language (CDL) of the system. This implementation and its performances are described in [10].

A second prototype is being developed on a network of SUN workstations by using UNIX processes, lightweight processes [19] and sockets. We also plan an implementation of a Cac run-time on a microkernel (*Mach* or *Chorus*).

The object interface: From the Cac/s, we propose several distributed representations for the single and the multi-threaded objects. A Cac is not a complete active object, it is just a "proto-active-object". An active object uses a standard interface allowing it to communicate. This interface is made up of 1) a **name** which is used to communicate with the object without knowledge about its representation, and 2) a **method invocation protocol** which is used to express method requests.

All of our objects have got a uniform interface. So objects of different representations can exist together in the same application:

- First, each object is linked to a specific Cac, called *Orc* (Object Representation Component) which identifies the object. So the name of the object is always that of its *Orc*.

- Second, each method call is materialized by a message sending to the *Orc* of the target object. The message structure is made up of a **method** name and an argument list (**args**). The message also contains the calling method name or by continuation another **proxy** method. Two invocation modes are possible: *synchronous call* which blocks the calling method, and *asynchronous call* which does not block the calling method (the calling and requested methods run concurrently).

2.2 Single-threaded active objects

A single-threaded object sequentially processes the incoming requests. It can be represented by one Cac, which is a single-threaded structure. Such an object is a specialization of a Cac with the same behavior.

The representation uses the *Orc* of the object interface: the *Orc* contains the local attributes and it processes the external (method invocations) and internal (self invocations) requests.

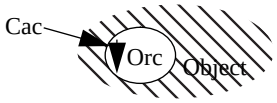

figure 2.1 The Object representation component

The *Orc* component is a Cac specialization. Each part of the Cac structure materializes a functionality of a single-threaded object. The *Orc* structure contains:

- 1 - A **mailbox** containing all requests to this object.

- 2 - A local **environment** containing all instance variables.

4. In this context, this word is synonym of Helios task.

- 3 - A **process** which initializes the object, extracts the messages, executes the methods, and manages the instance variables located in the local environment.

The designer of an object defines the list of instance variables and the set of methods. In our environment the code of the *Orc* Cac for single threaded objects is predefined according to our object interface.

2.3 Multi-threaded active objects

The structure of a multi-threaded object is more complex than a Cac structure: these active objects contain several activities when a Cac is single-threaded. So, now each active object will be materialized by a dynamic gathering of Cac/s. Cac/s do not share memory, so the object representation (i.e Cac/s) may be distributed in the memory.

Remote method execution: The *Orc* of a multi-threaded active object is the root Cac. The *Orc* does not execute the method executions itself, but it delegates these executions to specific Cac/s, called Method Execution Components (*Mec*). The *Orc* processes an incoming message by creating a *Mec* which executes the requested method. The local variables of the method are stored in the local environment of the *Mec*. The *Mec* exists during the method execution and is destroyed after the method execution. The separation of the method execution and the request reception leads to intra-object parallelism.

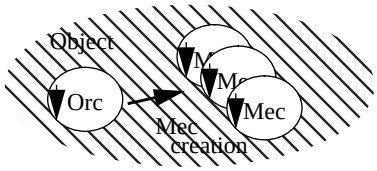

figure 2.2 The Method execution component

The *Mec/s* of an object are distributed on the nodes of the target machine. The object is distributed during the execution, moreover it dynamically changes according to the method executions on the object. The object is not a compact entity, it is a dynamic aggregate of Cac/s.

Remote access to instance variables: The instance variables of an object are stored in the *Orc* local environment. When a method wants to access an instance variable, the *Mec* must cooperate by messages with its *Orc*. This creates a communication overhead, and furthermore, an *Orc* may become a bottleneck in the object.

We have two answers for this problem. First the communication overhead is balanced by the intra-object real parallelism. Second, because the *Orc* creates the Mec/s and also manages the instance variables, we can extract one of these two tasks from this component and give this task to a specific Cac.

We introduce a new Cac type that manages instance variables. We call it an Attribute Manager Component (*Amc*). Each request by a *Mec* to an instance variable is processed by an *Amc* (fig 2.3). Note that an *Amc* may be located on any machine node.

An *Amc* serializes requests to instance variables. This avoids the use of a bottom level synchronization: the instance variables are not directly accessible by the *Mec*/s. More, an *Amc* may implement any specific strategy to protect or reserve some data for a particular use.

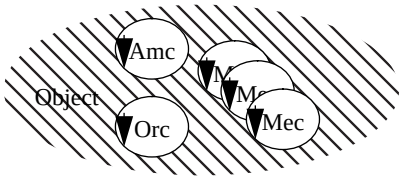

figure 2.3 The Attributes manager component

2.4 Fragmented multi-threaded active objects

We can go a step further in intra-object distribution. In the section 1.3, we have presented several interests of object fragmentation. The processing in our multi-threaded active object is already distributed, but the representation of instance variables remains compact in the *Amc*. This is true if a single *Amc* is used. Several *Amc/s* coexist in our fragmented multi-threaded active objects. Each *Amc* manages some instance variables. The *Amc*/s are created at object instantiation, and the *Amc*/s are known by the related *Mec*/s which can directly communicate with them.

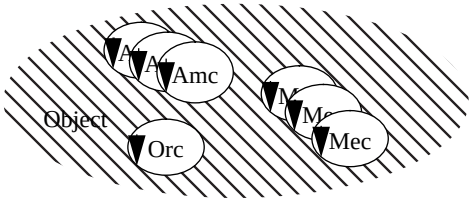

figure 2.4 A fragmented object

The creation of several sub-sets of the instance variables in different *Amc/s* presents other interests. First, this allows

concurrent access to different instance variables, so addressing the previous bottleneck problem.

Second, that also allows to regulate in some way the distribution of the components: *Amc/s* and *Mec/s* may be coupled at the distribution step to preserve privileged relationships between some of them. So we optimize the access time to instance variables and the global time of the application.

Finally, that also introduces the concept of active fragment: a fragment is often an active entity in an object. This entity may be defined by the programmer as an is-part-of component of the final object (Fragment-Oriented Parallel Languages). In our environment, these fragments can be represented by the union of an *Amc* and a *Mec*. A fragment is made up of instance variables (local data of the *Amc*), a name which identifies the fragment (the *Amc* name) and an activity (the *Mec*).

3 Conclusion

Operating Systems Object Layers must cope with different kinds of active object models. We introduce the Communicating Active Component to abstract the medium level between Object-Oriented Parallel Languages and Operating Systems. This tool is easily built on any distributed operating system providing lightweight processes, memory management and a communication protocol. Because it hides low levels, the Cac run-time increases the portability of parallel software, only the Cac run-time needs to be rewritten to execute Cac applications on another operating system.

A large scope of applications are described by the Cac model: parallel applications and especially run-times of concurrent languages for which the Cac environment is an efficient support for active object design. We mix several distributed representations of active objects. These representations preserve the design parallelism and create a real intra-object parallelism. Our current prototype integrates an execution model with method invocations on distributed objects. The first measures confirm the good properties of these representations.

Our current work concerns a high level Object-Oriented Parallel Language (called *Vcp)* for multicomputers. This future language uses fragmented multi-threaded active objects. We are also focusing on the synchronization constraints in this language [6].

References

[1] M.J. Accetta, R.V. Baron, D.B. Golub, R.F. Rashid, A. Tevanian, M.W. Young *Mach: A New Kernel Foudation for UNIX Developpment* in Proceedings of Summer Usenix, July 1986

[2] G. Agha, *Actors. A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1986, 144 Pages

[3] P. America, *POOL-T : A Parallel Object-Oriented Language*, in *Object-Oriented Concurrent Programming*, The MIT Press, 1987, pp. 199-220

[4] J.P. Banâtre, M. Banâtre, *Les systèmes distribués - Expérience du Projet GOTHIC*, InterEditions 1991

[5] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont, *Supporting an object-oriented distributed system: experience with UNIX, Mach and Chorus*, BULL-IMAG Technical Report 7-90, December 1990

[6] L. Courtrai, J.M. Geib, J.F. Méhaut, *Inheritance of Synchronization Constraints in the VCP system*, in EastEurOOPe'91 Proceedings, pp. 57-60, September 1991

[8] L. Courtrai, *Les Composants Actifs de Communication: outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD*, Thèse de doctorat de l'Université de Lille I (F), 1992

[9] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Méhaut, *The implementation of Actor environment on transputer under Helios: The Communicating Active Components*, Poster, in Transputers'92, Besançon (F), May 1992

[10] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Méhaut, *Communicating Active Components: An environment for concurrent applications on parallel machines*, accepted in EUROMICRO'92 to appear, Paris, September 1992

[11] C. Gransart, J.M. Geib, *Reusability and Concurrency in Parallel Eiffel*, X ième international Eiffel User Conference, Dortmund (Germany), April 1992

[12] S. Habert, L. Mosseri, V. Abrossimov *COOL: Kernel Support for Object-Oriented Environments*, OOPSLA/ECOOP'90 Conf. on Object-Oriented Programming, European Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990, pp 269-277

[13] D.G. Kafura, K.H.Lee, ACT++: Building a Concurrent C++ with Actors, JOOP Journal of Object-Oriented Programming, May/June 1990 , pp. 25-37

[14] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, *Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications*, JOOP Journal of Object-Oriented Programming, September/October 1990 , pp. 11-21

[15] J. Alves Marques, P. Guedes, *Extending the Operating System to Support an Object-Oriented Environment*, OOPSLA'89 Conf. Proc., Object-Oriented Programming Systems, Languages, and Applications, New Orleans, October 1989, Special issue of SIGPLAN Notices, Vol. 24, N° 10, October 1989, pp. 113-122

[16] Perihelion Software, *The HELIOS Operating System*, Prentice-Hall, 1989

[17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser, *The Chorus distributed operating system* ,Chorus Distributed Operating System, Chorus System, february 1989, pp. 305-370

[18] M. Shapiro, Y. Gourhant, S. Habert, L. Misseri, M. Ruffin, C. Valot, *SOS: An object-oriented operating system - assessment and perspectives*, Computing Systems, 2(7), 1989, pp. 287-337

[19] Sun Microsystems, *System Services Overview. Chapter 6 : Lightweight Processes*,1988, Part Number 800-1753-10