Arche: A Framework for Parallel Object-Oriented Programming Above a Distributed Architecture

M. Banâtre Y. Belhamissi V. Issarny I. Puaut J.P. Routeau

INRIA / IRISA
Campus de Beaulieu
35042 Rennes Cédex, France

Abstract

This paper sketches our experience with the design and implementation of a parallel object-oriented language and its distributed run-time system. The language integrates two original mechanisms for concurrency control: a synchronization mechanism that does not interfere with inheritance nor with subtyping, and a mechanism that serves for managing object groups. Because of the increasing power of inter-connection networks, the language's run-time system has been designed for a distributed architecture instead of a single multiprocessor machine. Furthermore, in order to ease the development of correct applications, we have chosen to rely on the run-time system to provide the required efficiency instead of offering the programmer low level primitives to be used for producing efficient code.

1 Introduction

Although the design of programming support for building parallel applications has been studied for many years, it is still an active area of research due to the wide range of issues that need to be addressed. In particular, applications must execute efficiently. Furthermore, as for any other programming context, the programming model must facilitate the design of correct programs. In this paper, we introduce a concurrent programming language, called Arche, aimed at easing the development of correct parallel programs, together with its distributed run-time system. The language is based on the object paradigm, which is now recognized as a powerful notion for program design. Moreover, we have chosen to rely on the language's run-time system to provide the required efficiency instead of offering the programmer low level primitives to be used for producing efficient code. Although our approach leads to less flexible programming, we claim that it is less error prone and thus suits more the correctness goal. The increasing power of inter-connection networks tends to allow execution of applications on a network of processors as if they were running on a loosely coupled multiprocessor, and hence enables execution of parallel applications on distributed architectures. Therefore, the language's run-time system has been designed for a distributed architecture. Furthermore, our past experience in distributed programming has led us to integrate facilities embedded within distributed systems, which we believe are as useful for parallel programming as they are for distributed programming. In particular, the run-time system provides implicit support for persistence.

This paper is organized as follows. An overview of the Arche language is first presented. Main contributions of our work come from three points: (*i*) proper integration of concurrency control within an object-oriented language; in particular, the mechanism for conditional synchronization that does not interfere with inheritance; (*ii*) a novel mechanism for multi-party interaction that eases management of object groups; and (*iii*) an exception handling mechanism that helps the design of fault-tolerant parallel software. The next section defines the Arche concurrency model; it details the first two contributions cited above. Section 3 then presents main features of the Arche distributed run-time system. In particular, we discuss its garbage collector for active objects that does not induce strong synchronization among local garbage collectors located on the system's nodes. Implementation status of the run-time system is then given. Finally, comparison with related work, and directions for future work are offered in Section 4.

1063-6927/94 $03.00 © 1994 IEEE

2 Overview of the Arche Language

The design of the Arche programming language has mainly been guided by our objective to minimize sources of programming errors. This has further led to the decision to address the efficiency problems at the run-time system level rather than offer in the language low level primitives for producing efficient code. Even though our approach may be considered as leading to rigid programming compared to existing parallel object-oriented programming systems (e.g., Presto [1]), we claim that the resulting programming language better meets the correctness requirement. For instance, although not enforced by the language, associating pre- and post-assertion to methods is as easy to achieve as with the Eiffel sequential object-oriented programming language [2]. Furthermore, definition of concurrency control as an integrated component of the language eases specialization and re-use of process behavior through respectively subtyping and inheritance since the language semantics defines all the language constituents. This section introduces main features of the Arche language related to concurrency control. For detailed presentation and assessment of the language, the interested reader is referred to [3]; a description of the Arche exception handling mechanism aimed at software fault tolerance may be found in [4].

2.1 Object Model

Our concern with safe programming has led us to define a language belonging to the Algol family with respect to block structures, scope rules and type-checking. The Arche language is hybrid; in addition to declaration of object descriptions, usual types (e.g., integer, record) are offered. An object description is made up of two components, namely a *view* and a *class*. The type constructor *view* defines an entity akin to an abstract data type. The declaration of a view $\mathcal{V}$ embeds the signatures of the methods that may be applied on objects of type $\mathcal{V}$. A *class* then defines a view implementation. Such an implementation must declare a procedure for every method of the view; it may further embed additional procedures, state variables, and an initialization method that is executed when an instance of the class is created. Finally, *objects* are instances of classes, and communicate exclusively by method calls. Implicitly, objects are persistent and their deletion relies on the use of a garbage collector provided by the underlying run-time system.

From the standpoint of concurrency, a process is associated to each object. An object creation leads to the creation of a process and then to the asynchronous call of the object's initialization method. As discussed in [5], expressing synchronization constraints in a decentralized way seems to be a key approach for integrating both inheritance and parallelism in an object-oriented language. This solution has therefore been retained in Arche. More precisely, concurrency control relies on four points:

- Any (non initialization) method call is synchronous, method acceptance by the invoked object being implicit;
- A N-readers/1-writer policy is implemented within each object;
- A conditional synchronization mechanism compatible with inheritance, is provided; and
- A mechanism aimed at easing management of object groups is offered.

The two last mechanisms are discussed in the following subsections.

2.2 Conditional Synchronization

The conditional synchronization mechanism of Arche relies on the specification of mutable sets of available methods as previously proposed in [5] and [6]. The advantage of this approach is that sets of methods declared in a class may be inherited and enriched in subclasses.

Unlike the mentioned references, sets of methods are defined in object types in Arche, and hence are taken into account in the definition of the subtyping relation. This aspect is crucial due the principle of substitutability that is inherent to subtyping [7]. In the parallel framework, this boils down to use a process in the replacement of another one. Since synchronization statements largely determine process behavior, the subtyping relation should be enriched so that process behavior may be taken into account when resorting to the substitutability principle.

Synchronization information are integrated within object types (views) through the definition of *synchronization states*. A synchronization state s of a view $\mathcal{V}$ specifies $\mathcal{V}$'s methods that are accessible when an object of type $\mathcal{V}$ is in state s. Moreover, *post-states* are introduced to indicate synchronization states that may be reached from execution of view methods. State transitions appear within method implementation through the use of a specific command that names the state in which the transition occurs. A state transition then becomes effective only if and when the method terminates. However, if no transition state takes place while a method executes, the

previous object state remains valid. The Arche mechanism for conditional synchronization may be related to path expressions [8] in that a part of the specification of constraints on the execution of operations is separated from the implementation of operations. However, compared to path expressions, our solution relies on the expression of synchronization information (i.e., state transition) within method implementation, and hence does not suffer from the limitations of path expressions for condition synchronization discussed in [9].

For illustration purpose, let us take a simple example. We consider the management of a dictionary whose associated view type **Dictionary** declares the following methods: **Find** that returns all the information pertained to a word passed as input parameter; **Add** that adds the definition passed as input parameter within the dictionary; and **Remove** that removes the word passed as input parameter from the dictionary. The synchronization of the type **Dictionary** is then defined as follows. First, **Find** is defined as a reading method while **Add** and **Remove** are writers. As far as conditional synchronization is concerned, **Add** can always be executed but the execution of **Find** or **Remove** has to be delayed if the called dictionary object is empty. We introduce two synchronization states: **empty** that allows only executing **Add** and **all** that enables any method execution. The definition of **Dictionary** is given hereafter. The following syntax is used: key words **OBSERVER**, **STATE** and **POST** respectively precede declarations of reading methods, synchronization states and method post-states; and synchronization states given in the header of the type declaration defines those that may be reached after termination of the initialization method.

```
TYPE Dictionary = VIEW () {empty}
      Add:  (def:  DElmt) ();
      Remove:  (w:  String) ()
   OBSERVER
      Find:  (w:  String) (def:  DElmt)
   STATE
      empty:  {Add}; all:  {Add, Remove, Find}
   POST
      Add:  {all}; Remove:  {empty, all}; Find:  {all}
END
```

Finally, let us remark that the rules of the subtyping relation for views given in [3] state conditions under which a subtype conforms to the behavior of its supertype

2.3 Management of Object Groups

Multi-party synchronization mechanisms have been recognized as a powerful tool for parallel programming (e.g., see [10]). The Arche language provides such a facility through the notion of object groups on which methods can be executed. An object group constitutes itself an object, called *sequence object*, and is declared as a dynamic sequence of objects having a common supertype in the subtyping hierarchy. Methods of a sequence object are called *multi-operations* and their semantics follows from the one of the *multiprocedure* notion introduced in [11]. A call to a multi-operation M of a group $\mathcal{G}$ is carried out as follows:

(i) All the objects belonging to $\mathcal{G}$ are synchronized;
(ii) Input parameters are distributed among $\mathcal{G}$'s components;
(iii) Method M is executed in parallel by all the components of $\mathcal{G}$;
(iv) Objects are synchronized and their respective contributions to the multi-operation's result –if any– are collected;
(v) Result –if any– is made out of each component contribution and made available to the caller;
(vi) Objects of $\mathcal{G}$ become available to execute further calls.

A multi-operation may issue a *coordinated call*, which is a natural extension of the method call mechanism. All the multi-operation components then join together to call a multi-operation and are all synchronized. When the call is terminated, result –if any– is made available to the caller's components before their parallel activities are resumed.

As a simple illustration of the multi-operation notion, our dictionary example may be enhanced from the standpoint of efficiency. Let us consider a partitioning of the dictionary into a group of basic dictionaries. Retrieving the definition of a word may be achieved efficiently by exploiting parallelism inherent to multi-operations. Given a partitioned dictionary **PartDic** declared as a sequence of dictionary partitions, and assuming that there is at most one definition for a word in the dictionary, the call to the multi-operation **Find** of **PartDic** is expressed as:

```
x := Filter(PartDic.Find(w))
```

where: it is assumed that the method **Find** returns the object **NIL** of type **NULL** (i.e., the smallest type of the type hierarchy) if there is no definition associated to **w**; **Filter** is a locally defined function that takes a sequence of elements of type **DElmt** as input parameter and returns the only element of type **DElmt** within the sequence by discarding elements of type **NULL**; and **x** is a variable of type **DElmt**. Finally, the above call uses the following notation convention with respect to the

type of w: when an expression of type T is passed as a parameter in a multi-operation call, the parameter is *multicast* to every component of the multi-operation.

2.4 Programming Environment

Among built-in Arche classes, the class **aggregate** is provided to aggregate objects within protection domains (e.g., process, task) offered by the underlying operating system. The creation of an instance of a subclass of **aggregate** leads to the creation of a new protection domain $\mathcal{P}$ and then to the creation of the class' instance within $\mathcal{P}$. On the other hand, creation of an Arche object whose class is not a subclass of **aggregate** boils down to the creation of an object within the enclosing protection domain.

From the perspective of implementation, an Arche compiler has now been operational for about a year; it generates code, written in the C language, aimed at executing above the distributed run-time system described hereafter.

3 Overview of the Arche Distributed Run-Time System

Our goal in the design of the Arche distributed run-time system was to allow efficient execution of Arche parallel applications on a local area network of workstations. The Arche distributed run-time system has been designed as an object-based system [12]. This choice has been motivated in part by the Arche programming model but more particularly by the advantages of the object approach for system structuring. In particular, object-based systems rely on higher level abstractions than the ones used for standard systems.

A straightforward solution for supporting Arche applications would be to define run-time system entities mimicking Arche objects. However, it is well known that such an approach results in an inefficient run-time system due to the resulting granularity for system entities. We therefore chose to define large grain system entities, called *protection delegates*.

3.1 Protection Delegates

A protection delegate defines a protected domain having its own virtual address space. Protection delegates are persistent; a delegate is removed from the system once it no longer contains pertinent information. As far as naming is concerned, associating unique identifiers (UIDs) to protection delegates may be done in two ways depending on whether UIDs are independent from virtual addresses or not. Although the second approach is becoming realistic due to the appearance of processors with 64-bit virtual address space (e.g., see [13]), we have decided not to undertake it as, in our opinion, it is not yet sufficiently mature and requires further investigations.

From the perspective of parallelism, a protection delegate embeds a dynamic set of threads and hence may be compared to the usual notion of server. However, in order to enforce actual parallel execution within protection delegates, a delegate may be distributed on a set of nodes. The distributed nature of delegates provides the system support for actual parallel execution on a distributed architecture; the nodes on which execute the threads of a given delegate may be seen as the constituents of a virtual multiprocessor. This further suggests communication by distributed shared memory for threads of a delegate. On the other hand, the communication model selected to handle interaction between protection delegates directly follows from the one adopted by systems based on the client-server paradigm; delegates communicate by remote procedure calls.

Given the notion of protection delegate, two orthogonal issues need to be addressed in providing support for execution of Arche applications: the design and implementation of delegates themselves, and the implementation of the Arche run-time support within a delegate. The former is currently under study; the remainder of this section focuses on the latter where a (distributed) protection delegate is emulated by a set of cooperating servers communicating by distributed shared memory. The next subsection addresses run-time support for object management within one protection delegate. Subsection 3.3 then addresses implementation of Arche objects, and is followed by a presentation of one of the run-time constituents that is the system's garbage collector. Finally, implementation status of the run-time system is sketched.

3.2 Object Implementation within Protection Delegates

Within a protection delegate, objects are managed as follows. An object is named by the UID of its enclosing protection delegate followed by the object's address within the delegate. An object state contains simple values (e.g., integer) and references naming other objects (i.e., objects' UIDs). The object state is persistent; the lifetime of an object state is independent from the lifetime of the object creator. As detailed further in subsection 3.4, an object is implicitly

removed from its enclosing protection delegate when (and only when) the object is no longer used.

From the perspective of parallelism, an object is active and hence embeds a (dynamic) set of threads for method execution. Objects may interact through both synchronous and asynchronous method calls. In the former case, the caller is blocked until termination of the called method. A method call may carry parameters, and the execution of a method called synchronously may return a result. Synchronization is supported through system-managed synchronization objects, implementing a N-readers/1-writer locking policy. Operations offered by a synchronization object are: *lock* and *unlock* that implement respectively lock acquisition and release; and *suspend* and *resume* that are used for conditional synchronization.

Finally, an algorithm for placing method execution is implemented within a protection delegate. Algorithms for method placement offered by existing distributed object-based systems (e.g., Emerald [14], Guide [15]) are simple; they select the node to be used for executing a method among one of the following: object storing node, calling node, and node explicitly mentioned by the programmer. One of the drawbacks of the existing implicit solutions is that they do not exploit node load. On the other hand, the approach that consists in leaving the programmer to specify the node on which a method should be executed allows her/him to choose the most appropriate node with respect to the node's load and resulting communication costs. As discussed previously, one of our goals in the design of Arche and its distributed run-time system was to make as transparent as possible management of system resources to the programmer. Therefore, in order to provide implicit support for placement of method execution while exploiting node load, we have designed a novel algorithm for method placement. The proposed placement algorithm, detailed in [16], adapts to processor load and to object characteristics, the latter allowing to approximate the exact cost of method's remote execution.

3.3 Implementation of Arche Objects

Given the implementation of objects within protection delegates discussed in the previous subsection, the implementation of Arche objects is straightforward. An Arche object (including an Arche class) translates into a protection delegate's object.

Aggregation of Arche objects within protection delegates relies on the use of the built-in Arche class **aggregate** (see § 2.4). It follows that the creation of an instance of an Arche class leads to the creation of an object within either a new protection delegate, or the enclosing one depending on whether the class is a subclass of **aggregate** or not. Once the creation of an object is finished, the object reference is returned to the object creator that may thereupon resume its activity. The creator then calls asynchronously the initialization method of the created Arche object.

Arche methods are implemented through addition of a postlude and a prelude to each method, which manage object synchronization by means of a synchronization object. Furthermore, a multi-operation of a sequence object is implemented as follows:

(*i*) The corresponding method is called asynchronously on all the components of the object group, which calls carry a result object as input parameter;
(*ii*) The sequence object blocks until termination of the method execution by all the components;
(*iii*) Called methods notify their termination and send their result –if any– through a synchronous call to the result object;
(*iv*) Once step (*iii*) is terminated, the sequence object collects the result –if any, and resumes its execution.

Finally, the coordinated call facility is implemented in a simple and centralized manner as follows. A system-managed object, called *coordinator*, is associated to each sequence object. Within a component of a multi-operation of a group $\mathcal{G}$, a coordinated call translates into a synchronous call to the method *coord* of the coordinator associated to $\mathcal{G}$; this call specifies the (multi-)operation actually called and will return the corresponding result. The coordinator collects calls to *coord* issued by $\mathcal{G}$'s multi-operation components. There are two cases to be considered depending on whether it receives matching calls from all the components or not. In the first case, the coordinator synchronously calls the method specified by *coord* parameters and then forwards the result to the calling multi-operation's components. On the other hand, it signals an exception to the multi-operation's components in the second case.

3.4 Garbage Collection

This subsection focuses on one of the constituents of the run-time support for Arche objects within protection delegates, that is, the garbage collector used for removing unneeded objects from delegates. Garbage collection in systems managing active objects is crucial: as active objects not only consume memory space

but also processing capacity, it is imperative to identify quickly garbage active objects. Garbage collection in object-based programming systems raises three distinct problems: distinguishing references from other data within objects, detecting garbage objects, and reclaiming the resources allocated to the garbage objects. In the following, we focus on the second issue, which is the most critical one in garbage collection.

Garbage collection in systems of active objects was first addressed for Actor-based languages [17, 18]. Intuitively, an object is garbage if its absence from the system cannot be detected through external observation, excluding memory and processor resources consumption. In particular, this implies that an object currently executing can be removed from the system if there is no link between the object and a root object. However, it should be noticed that an object that cannot call a root object at a given time (either because it is inactive[1] or does not embed a reference to a root object) may do so later as it may get a reference on a root object from another object through parameter passing on method call. In the following, we discuss the basic principles of the garbage collector implemented within the distributed run-time system of Arche, which has been described and assessed in detail in [19]. The garbage collector extends the one proposed in [18] for a centralized system of active objects, to a distributed framework. Compared to the distributed extension previously proposed in [20], the algorithm sketched hereafter does not require strong synchronization between nodes involved in the garbage collection process.

The set of objects is divided into memory *spaces*, each being associated to a component of the enclosing (distributed) protection delegate. The garbage collector is comprised of a collection of *local* garbage collectors (one per space), loosely coupled to a *global* garbage collector. The local garbage collector, associated to a space S, identifies and reclaims the objects that can be detected to be garbage without requiring communication with spaces other than S. A local garbage collector proceeds by marking the objects belonging to S following the algorithm proposed by Kafura in [18]. Initially, the root objects of S are marked as well as the objects that are assumed to be needed due to the lack of global information. These last objects may be called by/call objects of other spaces, as they embed references to objects of spaces different from S (objects containing *remote* references) or they are named by objects of spaces different from S (*remotely referenced* objects). By applying the marking algorithm, a subset of the system's garbage objects is detected without requiring any synchronization between the local garbage collectors.

To ensure that all garbage is detected requires having information on inter-space references. The global collector detects garbage objects whose identification requires inter-space communication in the following way. Periodically, each local garbage collector sends a message to the global garbage collector. This message carries a compact representation of the subgraph of the object graph of the sender's space needed by the global garbage collector, and references to objects sent in not yet received messages. The subgraph is computed by applying the sequential marking algorithm of Kafura; the initial marking applies to every object containing a remote reference and every remotely referenced object. The global garbage collector is a logically centralized service that maintains a graph G. When receiving a message from a local garbage collector, the global garbage collector merges the received subgraph with G. In order to check whether G represents a consistent system state, the message sent to the global garbage collector is timestamped with logical clock vectors [21]. If a consistent state is detected, G is marked following Kafura's marking algorithm; messages are then sent to the spaces containing the detected garbage objects. The stability property of garbage together with the properties of logical clock vectors [22, 21], ensure the correctness of the algorithm.

3.5 Implementation Status

We are currently implementing a prototype of the Arche distributed run-time system above the Mach micro-kernel [23] so that we can study the system's actual behavior. The current prototype supports a single protection delegate that is implemented by means of a distributed set of Mach tasks. This prototype has been operational since December 1992 on top of a local area network of SUN-3 workstations and a local area network of PC-compatible machines; it has been implemented using Mach 3.0, the communication link being a 10Mb/s Ethernet. The whole run-time system is made of about 9000 lines of C++ code, and took approximately one man-year to develop.

The architecture of the programming system on a given node is composed of three software layers. The bottom layer is the Mach 3.0 micro-kernel. The middle software layer of the programming system is composed of three server tasks: an object server, a memory manager (external pager, following the Mach terminology),

[1] An object is said to be inactive if and only if its threads of control are inactive.

and a Unix BSD 4.3 server. The first two servers make up the object-based run-time system. The object server implements object creation and communication between objects. The single protection delegate is made of the set of distributed object servers. When calling a method on an object, the appropriate node for the method execution is first selected by the object server. The object's memory constituents are then mapped in the address space of the selected object server –if necessary. Should the placement algorithm choose to execute the method on the caller's node, neither message passing nor context switches are required. The memory manager solves page faults on the objects stored on disk and keeps these objects consistent according to the causal consistency model presented in [24]. The run-time system uses some of the features of the Unix BSD 4.3 server; in particular, it exploits the Mach/Unix message server, implementing the TCP/IP protocols. Finally, the top software layer is composed of Arche objects whose code is generated by the Arche compiler.

4 Conclusion

Our work may be compared to existing distributed object-based programming systems such as Emerald [25], Guide [15] and POOL/DOOM [26, 27]. The programming languages offered by the referenced systems do not integrate synchronization information within types nor do they support management of object groups. Given the substitutability principle inherent to subtyping, integration of synchronization information within types is essential; it allows to take into account the process behavior in the definition of the subtyping relation. Although not demonstrated in this paper, our mechanism for conditional synchronization further offers the advantage to not interfere with inheritance. While Emerald does not support inheritance, inheritance of synchronization properties is restrictive in the Guide and Pool-family languages: either all the constraints have to be inherited as a whole and cannot be modified, or they are not inherited at all, which leads to rewrite synchronization information already declared in the superclass.

The Arche distributed run-time system relies on the implementation of large grain, distributed system entities, called protection delegates, that embed a dynamic set of objects. In our current implementation, a protection delegate is emulated by a set of cooperating servers. A protection delegate may be compared to the set of servers implemented in the Emerald system to execute an application. In the same way, the servers of a protection delegate share a common virtual address space. However, unlike our proposal, Emerald does not support object persistence. Our run-time system also differs from Emerald in its support for garbage collection [14]. Although Emerald provides active objects, the garbage collector designed for this system is based exclusively on object reachability (all objects that are executing are designated as being root objects, and hence none of them will be deleted). It follows that an object currently executing but not linked with a root object will not be removed from the system although it is no longer useful. On the other hand, the garbage collector of the Arche system has been designed for a system of active objects. Finally, our system differs from referenced ones in its support for placement of method execution. Whereas placement of method execution in Emerald and DOOM relies on information provided by the programmer, it is implicit in Guide. With respect to our design goal to make transparent to the programmer issues related to system resource management, placement is implicit in our system. However, unlike the placement algorithm of Guide that consists in executing an object method on the object's storing node, ours takes into account node load in order to minimize method execution time.

Our major concern with the Arche programming system is the design and implementation of protection delegates as defined in subsection 3.1. Our current implementation integrates a single delegate that is emulated by a set of cooperating servers. One study relates to the integration of protection delegates within Mach so as to efficiently handle the distributed nature of delegates. Other work concerns the provision and interaction of multiple delegates; in particular, this includes garbage collection of protection delegates.

Acknowledgments: The authors are grateful to Jean-Pierre Banâtre and Marc Benveniste for their contribution to the design of the Arche programming system. They also acknowledge the support of the FTM group [28] at IRISA for letting us modify their memory manager.

References

[1] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Presto: A system for object-oriented parallel programming," *Software Practice and Experience*, vol. 18, no. 8, pp. 713–732, 1988.

[2] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall International, 1988.

[3] M. Benveniste and V. Issarny, "Concurrent programming notations in the object-oriented language Arche," Research Report 1822, INRIA, December 1992.

[4] V. Issarny, "An exception handling mechanism for parallel object-oriented programming: Towards the design of reusable, and robust distributed software," *Journal of Object-Oriented Programming*, vol. 6, pp. 29–39, October 1993.

[5] D. G. Kafura and K. H. Lee, "Inheritance in actor based concurrent object-oriented languages," *The Computer Journal*, vol. 32, no. 4, pp. 297–303, 1989.

[6] C. Tomlinson and V. Singh, "Inheritance and synchronization with enabled-sets," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 103–112, 1989.

[7] P. Wegner and S. B. Zdonik, "Inheritance as an incremental modification mechanism or what like is and isn't like," in *Proceedings of the European Conference on Object-Oriented Programming*, pp. 55–77, Springer Verlag, 1988. Lecture Notes in Computer Science, volume 322.

[8] R. H. Campbell and A. N. Habermann, "The specification of process synchronization by path expressions," in *Lecture Notes in Computer Science, vol. 16*, pp. 89–102, Springer-Verlag, 1974.

[9] T. Bloom, "Evaluating synchronization mechanisms," in *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pp. 24–32, 1979.

[10] H. Jordan, M. Benten, G. Alaghband, and R. Jakob, "The Force: A highly portable parallel programming language," in *Proceedings of the International Conference on Parallel Processing*, pp. II–112–II–117, 1989.

[11] J. P. Banâtre, M. Banâtre, and F. Ployette, "The concept of multi-functions, a general structuring tool for distributed operating system," in *Proceedings of the Sixth Distributed Computing Systems Conference*, 1986.

[12] R. S. Chin and S. S. Chanson, "Distributed object-based programming systems," *ACM Computing Surveys*, vol. 23, pp. 91–124, March 1991.

[13] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single address space operating system," *ACM Trans. Computer Systems*, 1994. to appear.

[14] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system.," *ACM Trans. Computer Systems*, vol. 6, pp. 109–133, February 1988.

[15] S. Krakowiak, M. Meysembourg, H. Nguyen van, M. Riveil, and C. Roisin, "Design and implementation of an object oriented strongly typed language for distributed applications," *Journal of Object-Oriented Programming*, vol. 3, pp. 11–22, September 1990.

[16] Y. Belhamissi, *Allocation de processeurs dans un système distribué à objets*. Phd. thesis, université de Rennes I, 1994. to appear.

[17] G. Agha, *Actors : A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[18] D. Kafura, D. M. Washabaugh, and J. Nelson, "Garbage collection of actors," in *Proc. of the 1990 ECOOP/OOPSLA Conference*, pp. 126–133, 1990.

[19] I. Puaut, "Distributed garbage collection of active objects with no global synchronization," in *Proceedings of the International Workshop on Memory Management*, Lecture Notes in Computer Science 637, pp. 148–164, Springer Verlag, 1992.

[20] D. M. Washabaugh and D. Kafura, "Distributed garbage collection of active objects," in *Proc. of 11th International Conference on Distributed Computing Systems*, pp. 369–376, May 1991.

[21] F. Mattern, "Virtual time and global states in distributed systems," in *Proc. Int. Conf. on Parallel and Distributed Algorithms*, pp. 215–226, North-Holland Publishing, 1988.

[22] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," *ACM Trans. Computer Systems*, vol. 3, pp. 63–75, February 1985.

[23] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for Unix development," in *Proc. of Usenix 1986 Summer Conference*, pp. 93–112, July 1986.

[24] M. Ahamad, P. W. Hutto, and R. John, "Implementing and programming causal distributed shared memory," in *Proceedings of the eleventh International Conference on Distributed Computing Systems*, pp. 274–281, 1991.

[25] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul, "Emerald: A general purpose programming language," *Software Practice and Experience*, vol. 21, no. 1, pp. 91–118, 1991.

[26] P. America, "Pool-T: a parallel object-oriented language," in *Object-Oriented Concurrent Programming*, pp. 199–220, MIT Press, 1987.

[27] E. A. M. Odijk, "The doom system and its applications: A survey of esprit415 subproject a," in *Proc. of PARLE - Parallel Architectures and Languages Europe*, vol. 259 of *Lecture Notes in Computer Science*, (Eindhoven), pp. 461–479, Springer Verlag, 1987.

[28] M. Banâtre, P. Heng, , G. Muller, N. Peyrouze, and B. Rochat, "An experience in the design of a reliable object based system," in *Proc. of the 2th Conference on Parallel and Distributed Information Systems*, (San Diego, California), January 1993.