# Arts – Flexible Runtime Support for Massively Parallel Systems

Lars Büttner, Jörg Nolte
GMD/FIRST
Rudower Chaussee 5, Bldg. 13.10, 12489 Berlin
{lb,jon}@first.gmd.de

**Abstract**

Parallel object oriented programming languages employ a number of different programming models, namely active/passive objects and data/control parallelism or a combination of those. A flexible and efficient runtime platform is therefore required to exploit the capabilities of a given programming model in practice. Arts is an open object oriented framework that provides customizable runtime services for parallel processing within a global distributed object space. In this paper we present the fundamental mechanisms of Arts including performance measurements. We focus especially on location independent invocation as well as migration mechanisms, two key concepts for dynamic load-balancing.

## 1 Introduction

A parallel runtime system has to fill the gap between high-level programming models and the parallel computing platform. This task is not easy to accomplish. There is neither a commonly accepted parallel object paradigm nor can a single runtime system be able to perform all requirements of different object models equally efficient in terms of runtime overhead and resource consumption. Most runtime systems are therefore very specialized implementations for a single programming model.

This is acceptable for established programming paradigms and languages but causes a lot of extra development overhead when experimental language design and exploitation of parallel programming paradigms are concerned. Realizing these facts we designed Arts (*Adaptable RunTime System*) as a highly efficient object support system on top of established as well as experimental system platforms to exploit a variety of object based programming models. Arts is therefore organized as an open object oriented framework that provides customizable runtime services for parallel processing within a global distributed object space. Special emphasis is put on efficient location independent invocation mechanisms and object mobility, two premises for dynamic load-balancing algorithms.

In section 2 we present the design of the Arts framework and pay close attention to explain our design choices. An in-depth look at object migration mechanisms is taken in section 3. Section 4 discusses performance measurements for object invocation and migration mechanisms. Finally we draw conclusions to justify our framework design.

## 2 The Arts Framework

Originally Arts was a remote object invocation (ROI) [1] system designed as an extension of the parallel Peace operating system family [2]. In Peace system services as well as C++ applications are described by means of annotated C++-classes. These classes are pre-processed by a tool called Dog [3] which produces stub routines for remote object invocation and uses any C++ compiler as back-end. Efficiency and minimal abstractions were major design goals such that even frequently used low-level system services could apply high-level cooperation paradigms instead of crude message passing.

Today ARTS is an open object oriented framework which provides the basic services for distributed and parallel processing within a global object space. The global space is populated with both *active* and *passive* objects that reside within possibly distributed *domains*. Active objects are statically associated with an individual flow of control (a thread) whereas passive objects can only be manipulated by active objects. Domains are virtual processors (servers) providing logically encapsulated object spaces with controlled access to all objects inside. Dependent on a server's configuration, many domains may share an address space or may have separate address spaces either on the same processing node or somewhere in a network.

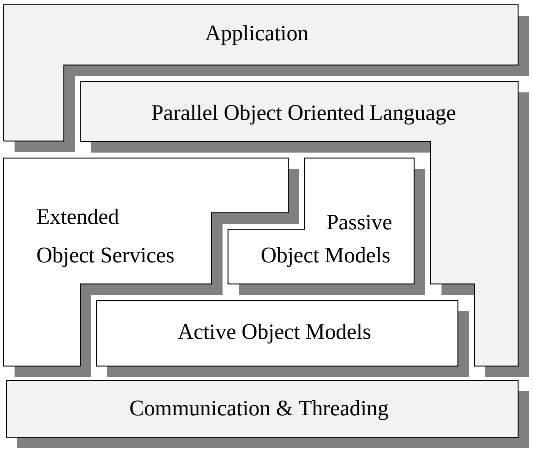


Figure 1: Building Blocks

Figure 1 depicts the building blocks of ARTS. On top of a basic execution platform active object models form the base of ARTS since they provide the most natural way to express object parallelism. Furthermore, only active entities can carry out activities and communicate physically in a distributed system. Passive object paradigms are therefore based on active object models in ARTS. Compilers for a parallel language generate code adapting either active or passive models or both. Extended object services provide support for naming and binding as well as bulk data transfer and object migration. These services can be directly applied by applications to customize for instance load balancing strategies and object placement.

Figure 2 depicts the major classes of the ARTS framework as well as their relation to the building blocks of ARTS (fig. 1). Active object invocation protocols integrate platform specific threading models and network communication protocols. Passive object invocation protocols extend the basic communication based addressing scheme of active objects to a global pointer concept similar to CC++[12]. Passive objects are accessed indirectly through communication with active objects that act as message dispatchers and control concurrent access to passive objects.

Complex operations covering object groups like in pC++[13] are not implemented in ARTS but can be provided by reusable topology classes using language-level multicast operations pre-processed by the DOG tool. We have chosen such an approach because complex protocols can be implemented much easier *and* still

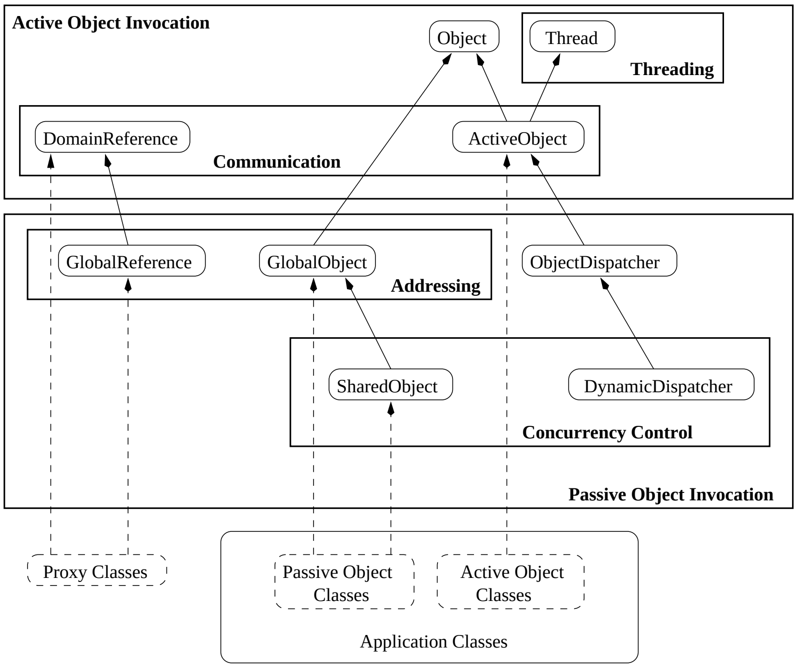


Figure 2: The ARTS framework

efficiently with language level support for distributed and parallel computation. Therefore ARTS focuses only on basic object models needed to implement a reconfigurable global object space.

## 2.1 Active Objects

Active objects are the primary way in ARTS to model parallel or concurrent activities and implement object domains. Active objects invoke each other through synchronous or asynchronous messages which are processed sequentially. In order to invoke an active object its `DomainReference` must be known. The `DomainReference` class provides a generic message passing interface for both synchronous and asynchronous remote object invocation. In the ARTS implementation for the PEACE system a `DomainReference` consists of a `unique` identifier that implicitly contains a node and a thread identification for communication between distributed threads[1]. Instances of `DomainReference` can be passed around through message passing or can be exported to a name service [4] that acts as a remote binding agent.

Active objects are instances of classes derived from `ActiveObject`. This class integrates a platform specific control flow model (`Thread`) with an ARTS compliant invocation interface (`Object`). Active objects provide an `action` method which specifies the main activity of the object. The default implementation of that method receives and processes messages in an endless loop. The `Object` class specifies a pure virtual `decode` method called by the `action` loop to convert received messages into local method calls. Concrete `decode` methods are typically generated automatically for application classes and perform at least usual tasks like parameter unmarshaling and local method invocation.

[1]If ARTS would be implemented e.g. on top of MPI or PVM a node number plus a local thread identification would be sufficient to identify an active object uniquely.

Active objects can either be *autonomous* or *recurrent* (fig.3). Autonomous objects redefine the default `action` method and can accept new messages explicitly during method execution using an `accept` method (fig.3). Thus an autonomous object always has both a dynamic stack context and a (passive) object context. In contrast recurrent objects do not redefine the default `action` method of `ActiveObject` and never accept new messages before the execution of a method is completed. Therefore the stack context of a recurrent object is (logically) undefined between two method calls and a recurrent object can be restarted from scratch when the object is migrated (section 3).

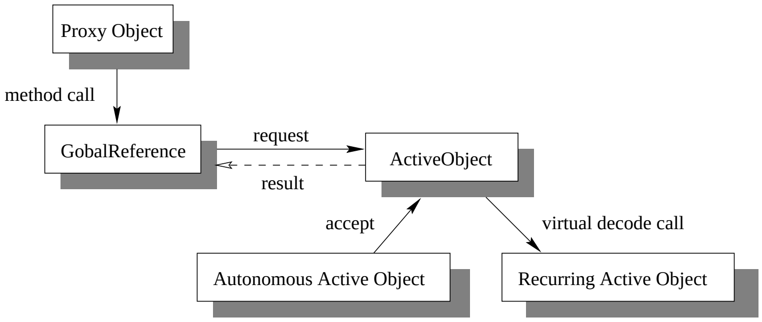


Figure 3: Invocation of Active Objects

Note, that the basic invocation mechanisms assume statically typed message formats and cannot transmit e.g. dynamically marshaled graphs. This fundamental design decision has been made because most methods calls can in fact be mapped to statically typed message formats and only a very few require dynamic argument marshaling. Consequently we separated dynamic and static messages to avoid unnecessary overhead for the most common cases. If dynamic data structures are involved this data can be dynamically marshaled into a separate buffer and only a pointer to that buffer as well as the buffer size is transmitted. Then we apply remote data transfer services at the receiving site to fetch that buffer. Thus the basic communication layers do not have to cope with dynamic buffering and can be implemented most efficiently.

## 2.2 Passive Objects

In ARTS passive object models are expressed as extensions to the basic active object model. Invocation of passive objects requires an additional level of indirection through an `ObjectDispatcher` (fig.4). `ObjectDispatcher` instances are special active objects which pass incoming messages to passive objects (instances of classes derived from `GlobalObject`) which in turn execute the requested methods under the control of the dispatcher. Thus all objects managed by the same dispatcher are automatically encapsulated in a monitor. This model has been applied to most PEACE system services because it is space efficient and does not cost any extra synchronization overhead caused by object locking.

In contrast the `DynamicDispatcher` class implements a dynamic processing model. For each incoming message a thread is dynamically created[2] to process the request. The `SharedObject` class provides locking mechanisms to implement mutual exclusion of readers and writers on a per instance basis. Instances of classes

[2]In fact an elastic thread pool is applied such that threads are only created if the pool is exhausted.

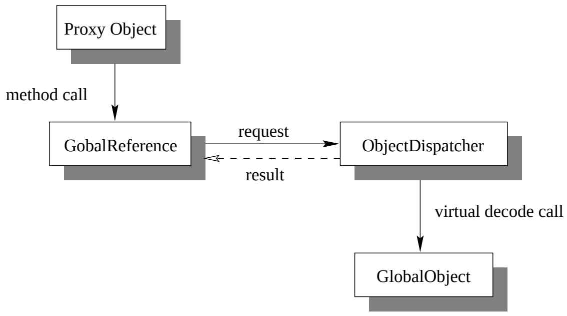


Figure 4: Invocation of Passive Objects

derived from `SharedObject` can therefore be managed by a `DynamicDispatcher`. Object locking is not performed by the dispatcher itself but in the `decode` methods of the classes derived from `SharedObject`. Thus a parallel programming language can control object locks according to the synchronization model of the language.

Each `GlobalObject` instance holds a unique object identifier which is generated during object creation. This identification never changes during the lifetime of an object and consists of a domain identifier and a randomly chosen sparse value. Thus these identifiers are hard to guess object *capabilities* stored in `GlobalReference` instances. Capability based addressing costs a small overhead but eases migration and helps to detect randomly forged messages caused by client failures. If this overhead cannot be tolerated it is very easy to provide a reference class and a dispatcher which directly uses the virtual address of an object as identification. In fact in the future we will implement both `GlobalReference` as well as `ObjectDispatcher` as template classes such that object addressing can be controlled through template parameters.

## 2.3 Remote Object Creation

In ARTS object creation is performed by instances of classes derived from `TypeObject`. Passive `TypeObject` instances are a runtime representation of classes capable to decode object creation requests. Specialized `TypeObject` classes are automatically generated by a translator or compiler.

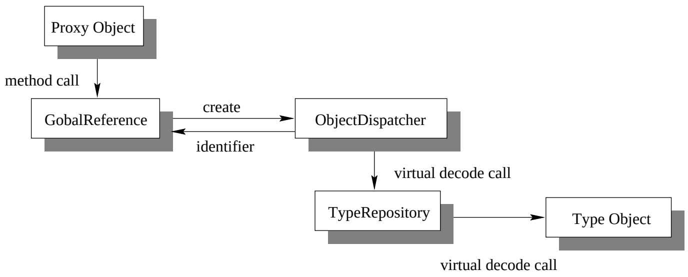


Figure 5: Object Creation

If an entity wants to create an object within the global space (fig.5) it sends a creation request to the domain where the object shall be created. This request is composed of a class name and the constructor arguments. Since a domain can handle instances of many classes all `TypeObject` instances are organized within a `TypeRepository`. The `TypeRepository` is a passive object with a known object identifier (a zero). Each `ObjectDispatcher` instance that is associated with such a repository is able to handle remote creation requests. Incoming requests to object zero reach the `TypeRepository` which in turn selects the suitable `TypeObject` instance by name and passes the `decode` request to that object which finally decodes the message and creates an instance of the requested type.

# 3 Object Mobility

Object Mobility is the key concept for dynamic load balancing [5] [6] and reconfiguration issues. ARTS therefore provides migration mechanisms to move objects between domains during the application runtime. The basic model of a fixed global object space is extended to a dynamically reconfigurable space.

## 3.1 Migration of Passive Objects

Instances of classes derived from `MobileObject` are free to move between domains. Class `MobileObject` is an extension of `GlobalObject` and provides a migration interface. Migration is initiated through a `migrate()` operation on the object. That is, the migration itself is executed by the control flow handling all other operations on this object. Therefore the migration is not preemptive. Non-preemptively object migration has the major advantage that synchronization of object migrations can use the basic synchronization of method invocations and thus costs no extra overhead.

Upon migration a remote copy of the object is created by means of a virtual `clone()` method that needs to be defined by the application class or can be automatically generated. Figure 6 shows all migration phases in detail. If the mobile object has no dynamic data phases 2, 4 and 5 are omitted (depicted with dashed lines). Migration phases are numbered to indicate their execution order.

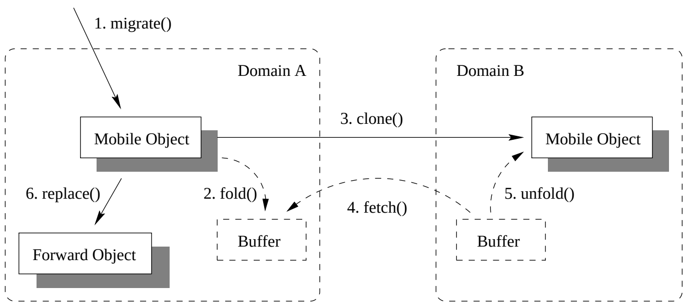


Figure 6: Migration mechanism

The remote copy of the object is created by means of a remote copy constructor. The new clone is initialized with the data of the original object. That data may of course contain references to other objects in the global space. Thus all remote references are retained. Only if the original object holds machine level pointers to objects in the original domain serialization has to be performed (`fold()`) before the remote copy constructor is executed. This constructor then needs to fetch the serialized object by means of a remote `fetch()` operation and unfold the serialized data.

After this cloning procedure a `Forward` instance with the same identifier like the cloned object is created to replace the original object (`replace`). The original object is deleted. This forward object replies to all messages with a message that causes the client to send the message to the new destination (fig. 7). This protocol is implemented by a `TrackingReference` that is derived from `GlobalReference`.

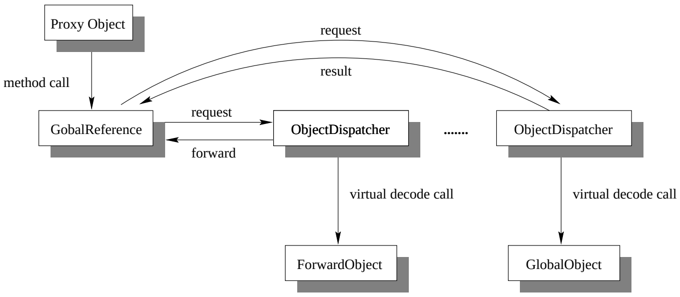


Figure 7: Message Forwarding

If an object moves multiple times the forwarding chain will naturally grow. Therefore we apply dynamic chain shortening. Every `TrackingReference` instance caches the reference of the domain which finally replies to the message. Thus the direct connection is automatically reestablished after the first call through the forwarding chain. The forward objects in the chain are finally collected when the migrated object is destroyed. The collection of forward instances is initiated by sending a destruction request to the head of the chain to remove the chain recursively.

An complexity analysis of object finding in distributed systems is given in [7]. We have chosen a chain model similar to [8] because a home based model requires extra communication during the migration phase. Our application analysis shows that chains are usually short (almost always one hop only). This stems from the fact that parallel applications typically use all of their remote references at least once during a computation phase. If object redistribution is applied between computing phases at most one hop is inserted for all moved objects. This hop is removed after the first use of the respective reference. Therefore we wanted to minimize the migration overhead itself to make load balancing of highly dynamic systems feasible.

## 3.2 Migration of Active Objects

We can also migrate recurring active objects and replace them with active forward objects with a minimal stack. Migration of recurrent active objects is essentially the same as passive object migration because recurring active objects do not have a stack context at migration time. At the destination site the thread of the recurrent object can therefore be restarted from scratch.

On top of this approach additional mechanisms have to provide the handling of runtime contexts. Autonomous active objects can only be moved if either a compiler is able to generate code to handle the stack layout at the new destination or the autonomous object can recover its dynamic state from the passive part of the object's state.

# 4 Performance Measurements

All measurements were done with our system running with PEACE on MANNA. MANNA is a massively parallel computer with a 50 MHz i860 processor and a 50 MBytes/sec crossbar connection. We used the most complex migration mechanism, that is, object serialization has to be performed. Table 1 lists the execution times of PEACE and the total time of language level calls using ARTS. All operations are executed between separated address spaces on different MANNA nodes. PEACE provides an inter-process communication (IPC) that realizes a rendezvous protocol consuming 165 $\mu$sec exchanging a fixed size packet of 64 bytes. Bulk data transfer is possible using the fetch operation. The caller reads a consecutive memory area from the callee's address space. The minimum time for this operation is 126 $\mu$sec. Depending on the memory area size additional transfer time is needed.

| Operation | Time ($\mu$sec) |
|---|---|
| Inter-Process Communication (IPC) | 165 |
| bulk data transfer (fetch) | 126 |
| ROI with argument size not exceeding an IPC packet | 175 |
| ROI with implicit bulk data transfer | 318 |
| remote object creation | 266 |

Table 1: Times of basic operations (PEACE and applications using ARTS)

The last three lines of table 1 list the total times of application level operations on top of ARTS, that is they include compiler generated stub code, ARTS operations and PEACE calls. A remote object invocation with argument size not exceeding an IPC packet needs 175 $\mu$sec. We approximated the basic overhead of our platform to be 10 $\mu$sec by subtracting communication times from an empty ROI. If caller and callee reside in the same address space (local object invocation) and the same PEACE communication primitives are applied, the overhead drops to 5 $\mu$sec. This is caused by caching effects. Thus the measured 10 $\mu$sec overhead describes almost the worst case. A remote object invocation including an implicit bulk data transfer is mapped to one IPC to synchronize caller and callee and a following fetch operation. Thus the overhead of this ROI call is 17 $\mu$sec.

| Phase | Description | Time ($\mu$sec) |
|---|---|---|
| migrate() | complete object migration | 1162 |
| fold() | memory allocation + serialization of dynamic data | 18 |
| clone() | clone operation including fold(), fetch() and unfold() | 870 |
| unfold() | copy static data + memory allocation and rebuilding dynamic data structure | 42 + 17 |
| replace() | create and install forward object | 21 |

Table 2: Detailed times of migration phases (mobile object with 400 bytes dynamic and 400 bytes static data)

We evaluated the worst case times of our migration mechanism. That is, all phases are executed in synchronous sequential order and no inherent parallelism is used. Table 2 shows the performance of our mechanism. The left column “phases” refers to the migration steps depicted in figure 6. The table lists the worst case times for every migration phase if we migrate an object with 400 bytes static data and serialize its 400 bytes dynamic data[3].

We measured the pure overhead migrating an empty object using the sequential migration mechanism with 790 $\mu$sec[4]. If the object contains 8 KBytes, the migration consumes 2338 $\mu$sec, for 40 KBytes 6675 $\mu$sec. The cost increases linearly with the object size, that is, a high speed interconnection network is absolutely necessary to achieve low migration times with bigger objects.

Our migration mechanism has some operations that can be executed in parallel. Optimization merely exploits this parallelism by overlapping some migration phases. Table 3 lists three possible variants.

| Overlapped phases | Description | Time ($\mu$sec) |
|---|---|---|
| migrate() request<br>migrate() execution | non-blocking migration request<br>blocking cloning operation | 62<br>870 |
| migrate() request<br>migrate() execution | non-blocking migration request<br>blocking cloning operation with class specific new/delete for clone object creation (destination domain) | 62<br>846 |
| migrate() request<br>migrate() execution<br>clone() execution | non-blocking migration request<br>non-blocking cloning operation | **62**<br>**160** |

Table 3: Optimized migration variants

One obvious improvement can be done in every case. Migration activation should be non-blocking. This drastically reduces the migration time. The caller is 62 $\mu$sec blocked, the control flow of migration source domain (callee) 870 $\mu$sec. Another optimization is class specific new/delete operators. They are somewhat useful in case of multiple migrations. This reduces the blocking time of the control flow of the migration source domain by another 24 $\mu$sec.

If the object has no dynamic data the migration reaches its maximum speed. Referring to figure 6 all migration phases shown with dashed lines can be omitted. Since the migration destination domain has not to be synchronized with the source domain (`fetch()` is not executed) the clone object creation is non-blocking. The caller of `migrate()` is blocked for 62 $\mu$sec as before, whereas the migration source control flow is busy for only 160 $\mu$sec. This case could be used for migration of small objects. Our experiences with numerical applications show that either the object is small and contains solely static data or it contains dynamic data and is at least of medium size [9].

# 5 Conclusions

Reconfigurable global object spaces are provided by numerous other systems like XENOOPS [8] and COOL-2 [10]. The ARTS framework in turn was deliberately restricted to very few key concepts providing basic mechanisms to implement this reconfigurable global object space. The design of our framework is strongly influenced by family based systems like Choices [11] and PEACE [2]. Therefore we

[3]Basically this “dynamic data structure” is a pointer to an array

[4]The actual object contains a few bytes because of ARTS dependent data, for instance the object identifier

implemented the reconfigurable global object space as extension to the static global object space. The result is that the application has only to pay for actually used functionalities. We provide only those features which are absolutely needed to form a runtime and resource efficient object support system. Common object oriented programming models are supported through combinations of ARTS components. The design decisions assure that the additional runtime overhead caused by this framework is very small. In particular statically typed message formats removed unnecessary dynamic memory management operations. Furthermore unmarshaling of invocation arguments is often executed without copy operations, because arguments are directly referenced from the message buffer. Our experience shows that the ARTS framework is suitable to realize a distributed and parallel C++ dialect.

## References


[1] H. M. Levy, E. D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software—Practice and Experience*, vol 21(1), January 1991.

[2] W. Schröder-Preikschat. The Logical Design of Parallel Operating Systems. *Prentice Hall International*, 1994.

[3] J. Nolte, W. Schröder-Preikschat. An Object-Oriented Computing Surface for Distributed Memory Architectures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, HICSS-26, January, 1993.

[4] H. Schmidt. Using Naming Strategies to Make Massively Parallel Systems Work. *Scientific Programming* vol 3(4), 1994.

[5] D. L. Eager, E. D. Lazowska, J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering* vol 12(5), May 1986.

[6] J. Kramer, O. Kremien. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, vol 3(6), November 1992.

[7] R. J. Fowler. The Complexity of Using Forwarding Addresses for Decentralized Object Finding. In *Proceedings of 5th ACM Symposium on Principles of Distributed Computation*, August 1986.

[8] S. Bijnens, W. Joosen, P. Verbaeten. Object Parallelism in XENOOPS. In *Proceedings of the Conference on Object-Oriented Numerics*, April 1993.

[9] T. Stickdorn, U. Haack, L. Büttner, W. Schröder-Preikschat. Dual, Mobile and Reactive Objects - Operating System Support for Load Balancing Parallel VR Applications. In *Proceedings of 13th Annual Hawaii International Conference on System Sciences*, January 1997.

[10] R. Lea, P. Amaral, C. Jacquemont. COOL-2: an Object Oriented Support Platform built above the CHORUS Micro-Kernel. In *Proceedings of the International Workshop on Object Orientation in Operating Systems, IWOOOS*, 1991.

[11] R. Campbell, G. Johnston, V. Russo. Choices – Class Hierarchical Open Interface for Custom Embedded Systems. In *Operating Systems Review*, vol 21(3), July 1987.

[12] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[13] Francois Bordin, Peter Beckman, Dennis Gannon, Srinivas Narayana, Shelby X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. Journal article from Scientific Programming, vol 2(3), Fall 1993.