COOL: Kernel Support for Object-Oriented Environments*

Sabine Habert†, Laurence Mosseri
INRIA, BP 105, 78153 Rocquencourt Cedex, France

Vadim Abrossimov
Chorus-systèmes, 6 rue Gustave Eiffel, 78181 St.-Quentin-en-Yvelines, France

Abstract

The Chorus Object-Oriented Layer (COOL) is an extension of the facilities provided by the Chorus distributed operating system with additional functionality for the support of object-oriented environments. This functionality is realized by a layer built on top of the Chorus V3 Nucleus, which extends the Chorus interface with generic functions for object management: creation, deletion, storage, remote invocation and migration. One major goal of this approach was to explore the feasibility of general object management at the kernel level, with support of multiple object models at a higher level. We present the implementation of COOL and a first evaluation of this approach with a C++ environment using the COOL mechanisms.

1 Introduction

COOL is a distributed object-oriented system, built on top of the Chorus[1] V3 minimal kernel, or Nucleus [23], alongside Chorus Unix[2] [4]. It runs on a local network of Sun 3/60 workstations.

COOL is a joint project between Chorus systèmes and INRIA, with a partial support from SEPT[3], a french PTT's research center.

The first goal of this project was to provide an object-oriented environment for the support of the CIDRE distributed document application [15] of SEPT.

Another major goal was to evaluate the feasibility of basic object management at the kernel level, which is able to support run-time systems with different object models. Our assumption was that a large part of the basic object management functionality can be shared by various object models, while specific features can be realized in a higher layer. Another assumption was that to place that management at the system level can improve the control on the system resources and the overall efficiency. To do so, we intended to map the abstractions offered by Chorus and to extend them in a generic and minimal way, while keeping their open features.

1.1 Related systems

As previously stated, in COOL, we intend to be able to support various object models. In order to meet

*Joint ECOOP/OOPSLA Conference, Ottawa (Canada), October 1990

†Author's current address: Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195, USA.

[1] Chorus is a trademark of Chorus-systèmes

[2] Unix is a trademark of AT&T Bell Laboratories

[3] SEPT is the french acronym for "Service d'Etudes Communes des Postes et Télécommunications"

that requirement, it is necessary to define a generic architecture that supports a large spectrum of existing object-oriented models. Existing distributed object-oriented systems can be roughly divided in two trends:

1. Systems that do not provide a uniform object model, such as Argus [19, 20, 21], Clouds V2 [14], Eden [3, 11, 18], Hermes [12] and SOS [24, 25, 26].
2. Systems that do provide a uniform object model, such as Amber [13], Emerald [9, 10, 16] and Guide [7, 8, 17].

Systems with a non-uniform object model are typically designed for specialized distributed applications with, for example, strong requirements in terms of data consistency (Argus, Clouds and Eden). They distinguish between language objects, which are small, local and passive, and system objects, which are large, global, and potentially active. The syntax of those two categories is slightly different. Only the system objects benefit from built-in reliability constructs and their granularity is, most of the time, of the order of the address space. Another reason to choose a non-uniform object model is the desire to be language independent (SOS, Hermes). In SOS, as in Hermes, the system objects are mobile and medium-grained. Object invocation relies on a proxy mechanism.

Systems with uniform object-oriented models are coupled with an object-oriented language. Such systems generally provide fine-grained object management along with thread and object mobility. While the advantage of such an approach is obvious (uniformity and fine-granularity at the programming level), there are several drawbacks. As the language and the system are strongly coupled, all the layers of object management are typically redefined and implemented for each programming language that you want install on the underlying operating system. Also, in Emerald and Amber, for efficiency, there is only one address space per site. This approach is not suitable for a large number of co-existing large applications.

In COOL, we have made the choice of a layered architecture. A basic layer implements the generic object management functionality. Higher layers will refine the facilities it offers and implement the semantics of their specific model. In order to do this, the basic functionality must not enforce any policy for object management. Also, the proposed mechanisms must be general enough to fit a large spectrum of requirements. In a first approach, we allow the co-existence of multiple address spaces within each site of the system, with objects modeled as segments of those address spaces (see section 2). We feel that this model is general enough to fit with most of the object models exposed above.

1.2 Outline of the document

The remainder of this paper describes the COOL abstractions. The next section presents the Chorus Nucleus abstractions and the architectural choices of the COOL system. The basic object model is described in section 3. Section 4 then presents the inter-object communication mechanisms, i.e., invocation and migration. The following section discusses our approach for object persistence. Section 6 presents an initial run-time of our system, the COOL/C++ library. Section 7 presents our first evaluation of the system on the basis of that run-time environment. We finally conclude in section 8.

2 The COOL architecture

COOL has been built as a member of the Chorus operating systems family. Each system in this family is built as a set of independent system servers based on a minimal real-time Nucleus, which provides the basic services such as activity scheduling, network transparent IPC, memory management and real-time event handling. Chorus also allows integration of various sub-systems with its Nucleus. Such sub-systems can be viewed as servers running in the Nucleus address space and are accessible via systems calls.

2.1 Nucleus basic abstractions

The physical support for a system is composed of a set of machines or *sites*, interconnected by a communication network. There is one Nucleus per site. The *actor* is the unit of distribution of the system. It defines a protected address space supporting the execution of one or more flows of control, or *threads*, which are scheduled by the Nucleus as independent entities. A given site can support many simultaneous actors.

The thread is the unit of execution in the system and is characterized by its context, corresponding to the state of the processor (registers, program counter, stack pointer, privilege level, etc.). A thread is always tied to one and only one actor, which constitutes its execution environment.

While the threads of an actor can communicate and synchronize using the shared memory of their common address space, they can also communicate with any other thread of any site via Inter-Process Communication (IPC) facilities. The Chorus IPC allows threads to exchange messages either *asynchronously* or by *demand/response*, also called *Remote Procedure Call* (RPC).

Messages are not addressed directly to a thread, but to an intermediate *port* attached to the thread's actor. A queue, associated to the port, holds the messages received but not yet consumed by the threads. A port can only be attached to a single actor at a time, but it can be successively attached to different actors. One can thus *migrate* the port from one actor to another, along with the queued messages.

A *group* of ports connects them to a *multicast* facility: messages can be sent either from one thread to an entire group of ports, or in "functional" mode; a port is then selected from the group of (equivalent) ports, therefore providing functional access to a service. A group is built by dynamic insertion and removal of ports. Ports and groups are globally designated with location independent, *Unique Identifiers* (UI's), whose scope is a Chorus network.

The Nucleus memory management [1] provides separate address spaces associated with actors. The address space of an actor is constituted of a set of non-overlapping *regions*, which form its valid portions. These regions are mapped (generally) to secondary storage objects, called *segments*.

A segment is implemented by an independent actor, its *mapper*. Segments are designated by *capabilities* containing the mapper's port UI and a key. The key is opaque data of the mapper, allowing it to manage and protect segment access. A mapper exports a standard read/write interface, invoked using the IPC mechanisms. Some mappers are known to the Nucleus as *defaults*; these export an additional interface for the allocation of *temporary* segments.

When the Nucleus decides (e.g., on a page fault or a segment operation) to make available a fragment of a segment in the form of physical memory, it extracts the segment mapper port name from the segment capability and sends a read request to the mapper. The mapper responds with a message containing the data.

The Nucleus encapsulates the physical memory holding portions of the segment data in a per segment *local cache* object. A local cache object is designated by its capability and its server is the Nucleus. Using the local cache capability, a mapper is able to distinguish between the local caches, on different sites, of the same segment, and to implement distributed consistency maintenance protocols.

2.2 COOL overview

In COOL, the execution domain, or *context*, is an address space local to a site. Several contexts can coexist on the same site. Each context is composed of objects and threads. Objects and threads are seen as orthogonal entities by the system, which associates no relationship between them.

COOL does not add any functionality to the Chorus thread model, which provides for thread creation, scheduling, synchronization and deletion. Each object consists of one code and one data segment mapped into a particular context. While threads always remain local to their creation context, objects are mobile and can move between different contexts during their lifetime.

The notion of context maps that of the Nucleus actor: a COOL context is an actor, created with an initial, user-specified object.

COOL is organized as a number of servers. There is one COOL manager per COOL site. The COOL manager is implemented as a sub-system within the Nucleus address space, and provides basic context and object management on the site. It handles hardware traps used by objects to access the COOL services, and it uses Chorus IPC to communicate with other COOL servers. All the low-level management is done at the Nucleus level. We can implement group operations, such as migration of several objects, more efficiently in the Nucleus than in external servers. Context management involves creation, deletion, and persistence. Object management consists mainly of creation, copy, deletion, remote communication and migration.

The other servers currently available are a class server (see subsection 3.2), a segment mapper, and a name server. The segment mapper collaborates with the class server in order to provide segments containing an object's code and initial data state. The name server associates a symbolic name with an object capability. The COOL names are integrated in the Unix naming hierarchy, so that COOL objects can also be accessed from the Unix environment. Inversely, Unix files are also visible from the COOL environment, and it is possible for COOL objects to use the Unix I/O mechanisms. We thus spare the effort to implement such functionality. A storage server is also planned for the future. Although we can rely on the Unix file system functionality, object storage requires higher level mechanisms than the storage of uninterpreted byte streams.

Although it is possible to use the raw COOL facilities, it is not very convenient. The COOL manager is intended to support run-time systems. A C++ [27] run-time environment has been implemented as a library linked with users applications.

3 The basic object model

3.1 Object representation

COOL objects are passive and medium-grained. By this, we mean that a COOL object is much more lightweight than a process or an address space (in fact many COOL objects can coexist in any typical address space), but still too heavyweight to make every user-visible entity into a COOL object. A set of attributes, associated with an object, determines whether it is *globally* known and whether it is *persistent*. A global object may request receipt of messages sent by remote objects. A persistent object can only be explicitly destroyed; it survives system shutdowns.

Figure 1 shows the different object components. A system descriptor, located in the sub-system address space, handles the user part of the object representation and the object attributes.

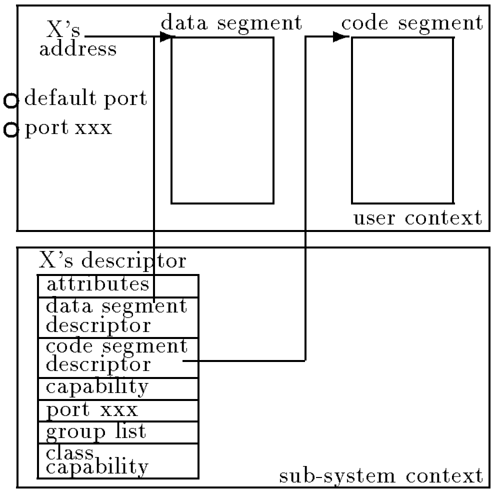

Figure 1: Object Representation

The user part of the object consists of two segments, each one mapped in an individual region of the containing context:

- The code segment contains the code of the object methods. This segment is shared by all the instances of the same class in the system.

- The data segment contains the object state. It constitutes the private "heap" of the object, and can grow or shrink following the object's dynamic (de)allocations. Its segment mapper can be selected at object creation time.

3.2 The class object

The class of an object is itself an immutable object, which carries the information needed by the COOL manager for object creation. There is a unique instance for each class[4], managed by the class server.

A class object publishes a functional interface which provides a class description; this description can be used for type-checking[5] by languages that require it. A class description is a structure that contains at least the following information:

- class attributes,

- initial data and code segment descriptors.

Object regions are created and initialized on the basis of the information furnished by their class description. The only class attribute significant for the COOL manager is the global attribute: if present, a capability and a port are created for each instance of the class.

4 Inter-object communication

4.1 Object invocation

Object interaction is mapped on the underlying Chorus communication facilities. ¿From the COOL manager point of view, the only way objects can communicate is message passing, either in a synchronous or asynchronous way. An object created with the global attribute is assigned a capability and a port tied to the capability. A global object can thus be the target of remote invocations. It must request message reception *explicitly*. COOL also allows the construction of object groups for asynchronous multicast communication.

4.2 Object migration

Object migration is piggy-backed on message transmission. Several objects can migrate along with a message, either on request or on reply message transmissions. One simply provides a list of objects to be moved or copied in the target object context. Such an interface does not enforce any migration policy. Runtime systems can build upon it either explicit migration primitives, or argument passing with call-by-move or call-by-visit semantics, as in Emerald [16]. Figure 2 illustrates synchronous remote object invocation with migration.

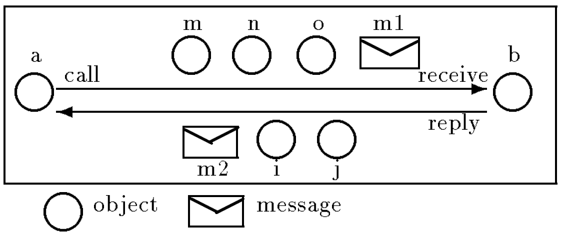

Figure 2: Synchronous inter-object communication, with migration

[4] It can be replicated for availability.

[5] The manager doesn't perform any type-checking itself.

Although coupled with remote invocation, object migration is based on the mechanisms of the distributed virtual memory management. Object migration simply consists of unmapping the object's segments from its source context, transmitting its descriptor, and then mapping its segments in the target context. As COOL runs on a homogeneous network, we don't need to carry class information with an object. Its code segment is simply mapped (if not already present) in the destination context. However, an object's descriptor holds its class capability. This capability can be used by run-times for type-checking purpose. Also, it may be useful in the future, if we decide to adapt COOL for an heterogeneous environment: the class manager could handle different code segments for the different machine architectures of the network.

The algorithm of migration with asynchronous communication is the following:

1. At transmission time, for each object to be migrated,
   (a) its segments are unmapped from its source context,
   (b) its descriptor is removed from the objects list of the source context and concatenated at the end of the user message.
2. The message is then sent.
3. At reception time, for each object identified by the descriptor list of the message,
   (a) Its segments are mapped in the target context.
   (b) If the object is global, its port is migrated from the source context.
   (c) Its descriptor is then inserted in the target context list.

In case of synchronous message transmission, step 1a occurs at the beginning of the receipt session, so that in case of a transmission error, the object is still mapped in the source context. In case of migration of a copy, step 1a is skipped, and the object descriptor is duplicated. Step 3a is also different; in this case, a copy of the source is mapped in the target context, and, if the object is global, a new capability/port pair is allocated.

Inter-site port migration is more complicated than the intra-site migration. Currently, Chorus does not implement the remote migration of ports with queued messages. We thus have to forward the messages queued on the source site, without guaranteeing that they will be received before messages sent more recently on the remote site. This feature calls into question our initial design of object implementation, since associating a private port with each global object only relieves us of the burden of forwarding pending messages for intra-site migrations.

5 Persistence

In COOL, object persistence is achieved by context persistence. A context that contains one or more persistent objects inherits the persistent attribute.

A persistent context cannot be destroyed until it loses its persistent property, i.e., all its persistent objects are migrated or explicitly deleted. At shutdown time, the state of persistent contexts is checkpointed on storage. Objects and threads are saved, messages are lost. Persistent contexts are then automatically restarted at boot-time. The current state of work does not address the problem of site or context crashes. Additional functionality, such as explicit checkpointing, is needed.

While COOL provides object persistence, it does not solve all problems associated with long-term object storage. In particular, it does not handle conversion of internal object pointers, since it is not possible to achieve this without the assistance of a language and/or a run-time system [2, 5, 6, 22], or without explicit programmer intervention.

6 The COOL/C++ library

The COOL/C++ library, used by the CIDRE distributed document application, is our first validation of the COOL manager. We tried to bring additional functionality to the COOL manager, while matching closely the abstractions it provides without any special language effort.

The library provides a *member* mechanism which enables the application of operations such as migration and deletion[6] to groups of related objects. This mechanism, inspired by the Emerald "attach" mechanism, exploits the grouped migration mechanism allowed by the manager.

The library also provides relocatable typed pointers for the internal data of an object. As all the data of a given object, including dynamically allocated data, are located in a single contiguous area of address space, it is fairly easy to provide relocatable pointers. Their data part contains the relative offset of the referenced object. Dereferencing computes the absolute address by a simple addition.

[6] And also storage, in the future.

Finally, the library manages the relationship between objects and threads, which is not taken into account at the sub-system level.

6.1 The COOL/C++ object

A COOL object is a C++ object whose root base class is the *cool* class. It can run its own thread if it defines the main method, and it thus has the *active* attribute.

¿From the run-time point of view, the public methods of an object are its virtual methods. This feature has been dictated by the thread management policy exposed in 6.3. We also wanted to avoid the cost of dynamic linking of object methods, both at creation time and at migration time. In order to do so, we compile objects as executables, linked at fixed addresses. The only unresolved references are the calls to other COOL objects. As all those calls are indirected via the C++ virtual mechanisms, they can be resolved at execution time. The C++ class description contains two additional fields to the fields mentioned above in 3.2, the addresses of the class *constructor* and of the main method, and two attributes, active and *monitor*. All the virtual methods of a monitor object will be executed in critical sections (see 6.3).

As we wanted to avoid duplicating an object definition, depending on whether or not it had to be known by the system, COOL objects are not automatically known by the system[7]. In order to be known by the system, an object has to be created using the COOL primitive for object creation (see below). This primitive allows selection of a segment mapper for the data segment of the object. It is thus possible to associate different mapping policies to objects. The COOL manager creates the object segments before calling the constructor, whose address is furnished by the class description.

```
c_objCreate (classDesc, attribute, segmentMapper)
                → object
```

6.2 The member mechanism

The member mechanism allows programmers to simply form composed objects for automatic grouped migration and deletion. As shown in table 1, to attach a member to an object one has only to declare and assign a member pointer. The member pointer can then be used as a C pointer. Detachment occurs by reset or reassignment of the pointer.

An object may only be the member of a single composed object. On the other hand, one object can have several members. It is thus possible to build member trees. A member cannot be migrated independently of its root object. But one can migrate copies of a member either in a shallow way (only the member itself) or in a deep way (with all its own members).

[7]Nonetheless, it may be inadequate to create a simple C++ object of a class which owns the global or active attribute.

```
member (type) memberPointer;
type* objectPointer;
---
memberPointer = objectPointer;
memberPointer->method();
```

Table 1: Member pointers interface

6.3 Thread management

In COOL/C++, the only way to create threads is to create active objects[8]. The attachment of a thread to an object occurs at object creation time. Once the object regions have been created by the manager, the constructor is called, then a new thread is started with the object's main entry point. The thread data is part of the user data region.

Active objects are likely to be *resident* objects, as it is difficult to provide thread mobility without efficient language support [16]. They are used to start a context and to provide pseudo-parallelism inside a context. However, as objects are mobile, we wanted to provide a restricted thread mobility. When an object has to migrate, its thread (if any) is destroyed in the source context, and restarted at the main entry point in the target context. If other threads are currently executing an object's method, the migration is refused and an error code is returned.

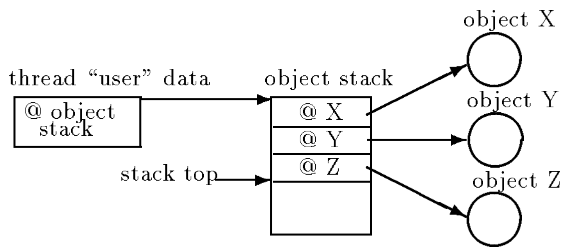

Figure 3: Relationship between threads and objects

The system object management primitives are not methods of the COOL objects. Instead, for each thread we maintain the notion of the current object, to which system calls have to be applied. To retrieve the address of the current object, we could rely on the

[8]There are no predefined thread objects, but one can simply define "pure" thread objects.

C++ feature that places the address of the invoked object as the first argument of a method call. But this feature is not safe, since a system call can be issued by a C procedure, or in the CIDRE application, by a Lisp function. Instead, an object stack is associated with each thread, which contains the addresses of all objects actually crossed by the thread.

The thread's current object is pushed on the top of the object stack before each call to a virtual object method. This is done by a logical intra-context trap, in a transparent way. The trap occurs via an indirection table, global to the context. Each table entry contains the address of a procedure which implements:

1. a prologue (push of object's address on the stack, increment of the object's activity count, monitor management),

2. the call of the object's method,

3. an epilogue (pop of object's address, decrement of the activity count, etc.).

All the information needed by the trap procedures are members of the *cool* base class. The indirection is realized at object creation time, the address of the object's real method table is saved in a field of the *cool* class.

7 Evaluation

In the current state of the work, we can only give a preliminary evaluation of our work with a C++ environment. For example, we have not yet done any performance measurement; it would not be very significant, because our run-time is implemented on top of a simulator. Unfortunately, at the time the project started, Chorus Unix was not yet available on Sun workstations. As it was not convenient to develop on the native Chorus system, a COOL prototype has been implemented on top of a Chorus simulator. This prototype is currently being ported to the native Chorus.

A positive observation is that the library was easy to implement on top of the COOL manager. We were able to provide high-level functionality without any special language effort, for example:

- The member mechanism was implemented with the group migration mechanism of the COOL manager.

- Variable-sized object mobility was provided, based on the object's "private data segment" feature coupled with a relocatable pointer mechanism. It allows migration of a composed object, such as a linked list, to a remote context or storage, without the cost of encoding its state in an external representation.

Unfortunately, the segmented approach does not map very well with the granularity of language objects, at least for the data part of the object. We have tried to design the COOL/C++ object to fit the segment granularity when possible, by inserting the thread state in the data segment of active objects. In general, this is not sufficient, however it is not a major problem for the CIDRE application, which encapsulates C++ and Lisp objects in COOL objects. CIDRE deals with rather large objects, but it implies an unnatural way of programming. Also, while an object is composed of a code and a data part, the actual one-to-one matching between regions and segments and the choice of two regions per object is somewhat arbitrary. For the code part, for example, it does not match the sharing semantics of the class hierarchies of object-oriented languages. We feel that further work in that area is needed in order to make our model more general.

Finally, we also feel the need for additional functionality such as object location. As objects are uniquely identified and location transparency is provided by ports, we didn't feel the need, at the sub-system level, to locate objects. Noneless, in order to exploit distribution, some applications may need explicit knowledge of object location. We must therefore provide some location functionality and finding mechanisms.

In order to test the generality of our system, it would be interesting to see how COOL abstractions map with the object-oriented systems mentioned above in 1.1, both in terms of granularity and functionality.

At a first glance, our model maps well with non-uniform object-oriented systems such as Argus, Clouds or Eden, which provide large-grained objects. Problems arise as the granularity becomes finer. While COOL can be used for medium-grained object management as provided in SOS and Guide, it is clearly inadequate for run-time environments such as Emerald and Amber.

Our communication model, based on message passing semantics, is not general enough. It does not match the local invocation model based on the procedure call semantics, because objects have to request the receipt of messages. This feature can be easily fixed. In order to allow remote procedure call semantics, we have to couple ports with objects in a loose way. We still have to assign a capability to each object but, for example, we can associate the default port of the object's context (see figure 1) with that capability.

As they provide object mobility, systems like Amber and Emerald also implement thread mobility while executing an object operation. We don't provide it in COOL. In Amber, thread mobility is made easier by the implementation of a distributed global address space. Although it would be nice to provide such a feature in COOL, we don't want to impose it because it would not fit our requirement of generality. In Emerald, thread mobility is made easier by the help of the compiler which provides templates of object layouts and of context dependencies such as registers. This functionality is realized at a higher level than the COOL manager. Our belief is that it is not safe to deal with thread mobility at the kernel level, since the COOL manager does not hold enough information to move threads without leaving residual dependencies in the source context.

8 Conclusion

We have designed and implemented COOL, a system layer for distributed object management. A prototype is currently available on a local network of Sun 3/60 workstations. It is used by SEPT to support a distributed document application. The goal of the COOL project was to be able support a large spectrum of object-oriented run-times. The COOL layer provides address space management and basic functionality for object management, such as creation, deletion, invocation, migration and persistence. In a first approach, we have chosen a segmented architecture as the basis of the object representation. Relying on distributed virtual memory management brings a lot of nice features. One can associate different mapping policies with objects by choosing the segment mapper for their data at creation time. Object migration simply consists of changing the mapping of the object components and is not restricted to fixed-sized objects. Noneless a segmented approach does not scale very well with small object management.

We investigate a redesign of COOL on the basis of those preliminary observations. Language objects can hardly be the unit of segment management. On the other hand, the segmented model allows us to rely on Chorus' underlying mechanisms for basic storage management and sharing. Also, we are not convinced that it is possible to efficiently manage fine-grained entities at the kernel level. Our current idea is that the regions which constitute an address space can be viewed as *container* entities that host collections of related objects. Containers are composed of one or several segments, one or several ports. They may be a unit of migration and persistence. Our investigation focuses on the way to provide container assembly and migration.

9 Acknowledgements

The SEPT team of the CIDRE project was involved in the design of the C++ run-time environment. Rodger Lea provided useful feedback as our first, patient and exacting user. Marc Shapiro and Marc Guillemont have given generously of their time and experience, participating in early discussions of the COOL model. We would also like to thank Hank Levy for his helpful comments on numerous versions of this paper.

References

[1] V. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 123–136, Litchfield Park AZ (USA), December 1989. ACM.

[2] O. Agesen, S. Frolund, and M. Hoffman Olsen. Persistent and shared objects in Beta. Daimi ir - 77, University of Aarhus, April 1989.

[3] Guy Almes, Andrew Black, Edward Lazowska, and Jerry Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1), January 1985.

[4] F. Armand, M. Gien, F. Hermann, and M. Rozier. Revolution '89 or distributing Unix brings it back to its original virtue. In *Proceedings of Distributed and Multiprocessor Systems*, Ft Lauderdale (USA), 1989.

[5] M. Atkinson, J. Lucking, R. Morrison, and G. Pratten. PISA club rules. Persistent programming research report 47, Universisty of St. Andrews, Scotland, August 1987.

[6] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4), 1983.

[7] R. Balter, J. Bernadat, D. Decouchant, S. Krakowiak, M. Riveill, and X. Rousset de Pina. Modèle d'exécution du système Guide. Rapport Guide R-3, Laboratoire de Génie Informatique, Saint-Martin-d'Hères (France), December 1987.

[8] R. Balter, S. Krakowiak, M. Meysembourg, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandôme. Principes de conception du système d'exploitation réparti GUIDE. Rapport Guide R1, Laboratoire de Génie Informatique, Saint-Martin-d'Hères (France), April 1987.

[9] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, October 1986.

[10] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–77, January 1987.

[11] A. P. Black. Supporting distributed applications: Experience with Eden. In *10th ACM Symposium on Operating System Principles*, volume 19, pages 2–12, Orcas Island WA (USA), December 1985.

[12] Andrew P. Black and Yeshayahu Artsy. Implementing location independent invocation. In *Proc. 9th Int. Conf. on Distributed Computing Systems*, pages 550–559, Newport Beach, CA USA, June 1989. IEEE.

[13] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, Arizona USA, December 1989. ACM.

[14] Partha Dasgupta, Richard J. Leblanc, Jr., and William F. Appelbe. The Clouds distributed operating systems: Functional description, implementation details and related work. In *Proc. 8th Int. Conf. on Distributed Computing Systems*, pages 2–9, S. José CA (USA), June 1988. (IEEE).

[15] J. Deshayes, V. Abrossimov, and R. Lea. The CIDRE distributed object system based on Chorus. In *Proceedings of the TOOLS'89 conference*, 1989.

[16] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[17] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, and C. Roisin. Design and implementation of an object-oriented, strongly typed language for distributed applications. To appear in *Journal of Object-Oriented Programming*, 1990.

[18] E. Lazowska, H. Levy, G. Almes, M. Fisher, R. Fowler, and S. Vestal. The architecture of the Eden system. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, pages 148–149, December 1981.

[19] B. Liskov. Overview of the Argus language and system. Technical report, MIT, February 1984. Programming Methodology Group Memo 40.

[20] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the 11h ACM Symposium on Operating Systems Principles*, pages 111–122, Austin TX (USA), November 1987. ACM.

[21] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1988.

[22] P. O'Brien, B. Bullis, and C Schaffert. Persistent and shared objects in Trellis/Owl. In *International Workshop on Object-Oriented Database Systems*, 1986.

[23] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, Kaiser C., S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.

[24] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[25] Marc Shapiro. Prototyping a distributed object-oriented OS on Unix. In Eugene Spafford, editor, *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale FL (USA), October 1989. USENIX. Also available as Rapport de Recherche INRIA no. 1082.

[26] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(7), December 1989.

[27] Bjarne Stroustrup. *The C++ Programming Language*. Number ISBN 0-201-12078-X. Addison Wesley, 1985.