

Chapter 20

Directions in Object-Oriented Research¹

D.C. Tsichritzis
O.M. Nierstrasz

Introduction

Object-oriented systems as an approach have inherited concepts, methods and tools from many other areas in computer science. We can venture to say that object-oriented systems consist of a repackaging and relabeling of a cross section of computer science. As a result many different researchers see projections of their ideas within object-oriented systems. This situation explains two phenomena. First, in a short period object-oriented systems evolved from an esoteric, exotic way of programming in certain artificial intelligence applications to a diverse and expanding area of research. Second, there are disagreements among researchers on basic definitions; for example, “What is an object?” Researchers coming from programming languages, artificial intelligence and databases, among other fields, have their own and conflicting ideas on when an “object” can be called an object. There is no reason to worry. Loose definitions are inevitable and sometimes welcome during a dynamic period of scientific discovery. They should and will become more rigorous during a period of consolidation that will inevitably follow.

Directions in object-oriented research to some extent follow the roots of the researchers. We see, therefore, object-oriented research related to programming languages, to operating systems, to databases, etc. This does not necessarily mean that object-oriented researchers are rediscovering the results already present in other areas. On the contrary, researchers in other areas are finding a new territory in which to apply their ideas. The net effect is, however, that the directions in object-oriented research can be clearly categorized according to the traditional areas in computer science to which they are related. In the first part of this chapter we shall discuss the directions that we will call *traditional*. They are advanced but closely related to areas already well-understood. In the second part of the chapter, we will permit ourselves to dream. In this way we can explore some exciting problems that we consider very important for the success of the object-oriented approach. Some of these problems existed before, but an object-oriented setting gives them different emphasis and a new light. The solutions may therefore take on a very different flavour. We will call the research directions related to these problems *exotic*.

1 Traditional Research Directions

Whenever a new technology becomes available, we must re-evaluate what we already know with respect to these new ideas. With the gradual introduction and acceptance of object-oriented tech-

¹In *Object-Oriented Concepts, Databases and Applications*, ed. Kim and Lochovsky, ACM Press and Addison Wesley, 1989, pp. 523-536.

niques, we find ourselves forced to pursue several directions of research along traditional lines, in order that we may understand how objects require us to re-interpret established results.

We can identify (at least) five major directions that researchers are currently investigating:

1. Programming languages
2. Concurrency and distribution
3. Object management
4. Software management
5. User environments

This list is not intended to be exhaustive, but rather to identify broad classifications of current research. We shall overview each of these areas and briefly discuss how object-oriented techniques pose new problems.

1.1 Programming languages

Objects affect our understanding of programming languages in three fundamental ways. First, our languages have a different “look” to them, that is, we need and use different language constructs and mechanisms. Second, our programming methodology must adapt, since we must use different disciplines to build our applications. Third, we need to develop new formal models that can help us to describe and analyze the systems we build.

If object-oriented programming were simply an issue of substituting an entirely new programming paradigm, we should perhaps have an easier job of it. Instead, we have discovered that the idea of “objects” has always been around, except that it has been lurking in the shadows under various guises, and using obscure names. Now our big problem is that of *language integration*: we would like to continue to be able to do all the things we have discovered to be good in the past, and we want to be able to use the new object-oriented techniques that have become available as well. We can only do this if we discover how to cleanly and consistently integrate disparate object models in our programming languages.

The first major “discovery” of this sort is that Smalltalk-style objects look a lot like abstract data types. That is, perhaps we can identify object classes and object *types*. This identification introduces a host of problems. What does strong-typing mean in this context? How can (static) strong-typing cope with dynamic binding of variables to object instances? How can a type theory for objects adequately cope with single/multiple class inheritance? Should class inheritance be permitted to violate encapsulation by permitting subclass methods to access inherited instance variables? How can we understand type-casting and coercion in an object-oriented language? How can we exploit encapsulation to support heterogeneous objects, that is, environments in which different objects may be implemented in various languages with, say, a common object-oriented interface language? Various answers to these problems have been proposed, and even implemented [Meyer 1986; Nierstrasz 1987a; Schaffert et al. 1986].

We are a long way, however, from having standard solutions, because the object models supported by different languages emphasize different properties of objects. For example, another

important “discovery” is the similarity between Smalltalk-style objects and active entities, or processes. However, not all concurrent object models view objects as active entities, as we shall see in the following subsection. We are still at the stage of enumerating the mechanisms supported by different object models, and attempting to understand their interactions [Wegner 1987].

Now that we have object-oriented programming languages that enable us to exploit encapsulation in various ways, we have discovered that we can no longer program in the way that we are used to. Our concerns with encapsulation in the past were mostly limited to problem decomposition and minimal interference between program modules. Now if we wish to exploit object reusability, polymorphism, class inheritance, parameterized types, and so on, we must organize our applications in such a way as to be consistent with these possibilities. We have developed techniques for decomposing problems into relatively independent components, but we still lack a methodology for viewing applications in terms of *objects*. Such a methodology will only arise through years of experience programming with these languages, followed by reflection and attempts to pass this knowledge on to other programmers. An extremely interesting research issue is whether it will be possible to construct software tools that encourage programmers to conform to such a methodology – this is related to the *object design* problem we shall discuss below.

Finally, we must re-evaluate our formal models of computation with respect to object-oriented concepts. This is already happening with type theory [Cardelli and Wegner 1985], models of concurrency [Agha 1986; Hoare 1985], and semantics of programming languages and applications [Wolczko 1987]. As object-oriented languages and systems gain acceptance, we shall see a similar re-evaluation of our models of security, distribution and persistence, since in each of these cases we are concerned with the properties of environments that have been partitioned into interacting components, and with interaction across environments. In distributed object-oriented systems, we will have to be able to understand and model security and persistence of objects communicating across environments.

1.2 Concurrency and distribution

We have already alluded to the importance of concurrent and distributed object-oriented applications. As in the past, object-oriented approaches to concurrency typically adopt one of two models of concurrency. The first is to view objects as passive entities manipulated and shared by concurrent application programs. The object management system controls and synchronizes access to these objects. This we call the “database” world-view of objects and concurrency.

The second approach is to consider objects as active entities themselves. Objects thus replace the notion of a “process”. Passive objects are effectively “server” processes that do nothing except wait for and serve incoming requests. Objects communicate by passing “messages”, and so synchronize themselves without the aid of an object manager. This is the “actors” world-view of objects and concurrency [Agha 1986].

Both these views have their advantages and disadvantages, but they are not so clearly at odds with one another as they might first seem. The database view offers explicit threads of control (encapsulated in the applications), and can support the abstraction mechanisms of passive objects (à la monitors, for example). The main difficulty with this view has to do with the separation of the world into passive entities (objects) and “applications”, which are not objects. This means that applications cannot be described by such an object model. In particular, it is not possible for applications to directly communicate with one another, unless we invent an independent communication

model for applications.

The active object model eliminates this problem by adopting a homogeneous view in which there is no distinction between objects and “applications”. On the other hand, active objects lack an explicit notion of threads of control. (Despite our claims of homogeneity, threads are not objects!) We must restrict ourselves to specific communication protocols, or message-passing patterns, if we are to be able to program effectively. By analogy, programming with *sends* and *accepts* is as dangerous as undisciplined programming with *gotos*. Note also that the active object model does not say whether objects are persistent or not, whereas this is an implicit assumption of the database view.

Distribution is also viewed differently by the two models. The (traditional) database approach is to provide an illusion of global shared database of objects, with communication between remote sites being handled automatically by the system. With active objects, it is more natural to expose the network, and even to allow objects to move themselves between sites. We can imagine, however, object-oriented databases that support migrations, or, conversely, active object systems that hide the network or automatically perform load-balancing.

More fundamental is the different approach taken to communication and synchronization. The database view is that applications perform operations on passive data and that the data must be protected from conflicts between concurrent applications. As a consequence it is natural to adopt a mechanism for mutual exclusion based on, for example, locks or transactions. Synchronization is viewed as orthogonal to an object’s interface. With active objects, on the other hand, communication is modelled by message-passing, so there is no need for an explicit synchronization mechanism. Mutual exclusion is not an issue since objects themselves decide when they are ready to accept a message.

These differences have more to do with modeling than with implementation, however. Distributed transactions in an object-oriented database necessarily require message-passing between cooperating object managers. Similarly, active objects can be implemented in a centralized system by coroutines (representing execution threads) sharing passive object data structures. Message-passing between active objects is thus a communication paradigm rather than an implementation requirement.

Current research into concurrent object-oriented languages generally adopts either a database view [Moss and Kohler 1987] or an actors view [Yonezawa et al. 1986], though there have been attempts to integrate these approaches into a single model [Nierstrasz 1987b].

1.3 Object management

As our object models and object-oriented languages evolve, we discover that it is useful to have object-oriented environments that provide us with run-time support for our objects. A sample of the kinds of support one might desire follows:

- *persistence*: automatic saving of objects between sessions, either through persistent virtual memories or object-oriented databases
- *garbage collection*: automatic deletion of unreferenced objects
- *concurrency*: multiple concurrent threads; communication and concurrency control through message-passing or object-oriented databases

- *distribution*: global object-naming, and remote message-passing or method invocation
- *security*: ownership of objects; “access rights”, or permission to invoke objects’ operations

Depending on the intended application domains, these kinds of functionality can be provided either by operating systems or by object-oriented databases. The database approach is valid for data-intensive applications that deal with large numbers of similarly structured objects (the number of objects will be much larger than the number of object classes). Content addressability is fundamental in order to support querying and to provide a basis for organizing the object store (i.e., for indexing, etc.). For application domains that violate these assumptions, there must be a greater emphasis on providing support for non-database objects. There is clearly a need for object-oriented systems that can support a mix of these two application paradigms. We will eventually have to address an integration problem in bringing object-oriented databases closer to object-oriented programming languages. The basic choice will be whether we decide to maintain the barrier between “the applications” and “the database”, thus viewing an object-oriented database as a strictly separate component that can be interfaced to the rest of the system, or whether we will move towards truly integrated object management systems that provide standard support (for persistence etc.) for all applications, and additionally provide database support for the applications that need it.

1.4 Software management

We mentioned the *object design* problem earlier in this section. This is the problem of deciding what objects one needs to put together an application. The strategy for decomposing an application into objects depends on the properties required of the resulting partitioning. Do we wish merely to write a minimum of code, or to produce a prototype in the shortest time possible; or do we need an extendible application whose objects are reusable and portable to other environments? For these questions there is no substitute for experience. The issue is whether we can take short cuts by learning from other people’s experience. The development of a methodology for object-oriented programming will help in the object design problem.

We may now well ask what software tools would be useful to a developer of an object-oriented application? In other words, can we realistically speak of computer-aided design of software objects? If so, is object design similar to other design tasks? The first example that comes to mind is that of database design. Here too we are concerned with decomposition into semi-independent “objects”. Can we develop tools (say, graphical tools) that drive the decomposition and design task?

Once we have a plan of our application in terms of objects, we must find ways to implement them. Again, experience is invaluable. But object-oriented programming heavily emphasizes reusability. That means we can expect extremely large collections of reusable objects to be available to us. “Browsing” tools are popular for navigating through a collection of object classes, but they may be severely limited when dealing with tens of thousands of reusable classes. Since the mapping from object specifications to plausible implementation components is effectively “expert knowledge”, we should ask whether expert systems are appropriate tools for helping programmers to find their way through databases of reusable object classes.

At a more technical level, as software is being developed, we must keep track of the dependencies between object classes, and between instances and their realizations. In a distributed environment, we must also manage the distribution of evolving software. We already know how to manage evolving software written in traditional programming languages. In object-oriented systems, software

not only evolves, but reusable software is expected to have a very long lifetime. Furthermore, we expect that objects with their well-defined boundaries will be easier to manage than traditional software. These points suggest that tools for managing object-oriented software will quickly become standard as their need arises.

1.5 User environments

Object-oriented concepts have a very natural application in user interfaces. *Direct manipulation* is a fundamentally object-oriented idea that demonstrates both encapsulation and polymorphism (the “same” operation can apply to different kinds of objects). Instead of issuing a command to perform some function, and then later checking to see how that function was performed, one is presented with a view of the “object” one wishes to manipulate, and then one performs an action “directly” on the object (that is, on its presentation). The feedback is typically immediate. This approach is best seen in WYSIWYG (“What you see is what you get”) applications, such as document editor-formatters.

Direct manipulation is well-suited to object-oriented applications, since it is quite easy to identify the objects that must be presented to the user. *How* an object should be presented, however, is often not so easy! The paradigm of direct manipulation starts to break down when we have to perform complicated actions on objects. Inserting text at the current location by pressing the appropriate character key is straightforward enough, as is deleting the current object by hitting the *delete* key. But if we want to construct and remember a more complicated procedure, such as deleting obsolete addresses from several address lists of people on another list, then we must resort to “programming”. If we can package this procedure in some convenient way, then we can apply it whenever the occasion recurs. To do this, we need a direct manipulation analogue of shell programming. This is what is often referred to as “programming-by-example” [Halbert 1984]. It remains to be seen how far object-oriented interfaces can be pushed to replace the need for command shells.

The development and popularization of window systems to a large extent solves the problem of managing multiple concurrent threads. We can not only present to the user applications with concurrently executing objects, but the user can switch his attention between tasks by moving from window to window without interrupting or terminating the current one. We are also beginning to see “nested” windows (replacing directory hierarchies) and “remote” windows for distributed applications.

Perhaps with the continuing development of computer games, we will see the kinds of objects that we now play with begin to appear as characters in the interfaces we use to do our work on. As we become more desperate to find good paradigms for direct manipulation, we shall probably find that plausible solutions already exist in terms of presentation objects that jump, fight, crash, and blow up.

2 Exotic Research Directions

By *exotic* directions we do not necessarily mean impossible or unnecessary problems. We simply mean research directions that are particularly relevant when we think in terms of objects. We will present the direction as a series of interesting problems. Most of these problems manifest themselves when we deal with a distributed, open-ended environment of active objects.

2.1 Acquaintance problem

The acquaintance problem is related to the binding problem in programming. Consider a population of independent objects that can move around in a network, where this population is continuously evolving and expanding. An object at its birth knows about a number of objects in its immediate environment. We can think that these objects' names have been passed down as capabilities from the creator of the object. For an object to be effective, however, especially in a network environment, the object may have to communicate with other totally unknown objects. The problem is how two objects unknown to each other can get introduced. We are not discussing the location problem, i.e., how you find an object already known to you. We are discussing how you obtain the name of an object you would like to talk to.

Two immediate solutions come to mind, one from the user and one from the object manager. They both have severe limitations. The user does not and should not know all the objects around. The object manager knows only the objects in its jurisdiction. We need, therefore, to solve a problem of context. We need to define a context in which we will try to select some interesting objects. Second, we need to specify the criteria according to which an object should be selected. An unknown object cannot be selected according to its internal structure or values because they are hidden. Its methods do not have the proper selectivity since many objects share them. We should probably select it according to its *past behaviour*. We need, therefore, to define and implement a notion of past behaviour. We can then select a new acquaintance within a context in terms of what it did (good or bad) during its lifetime.

2.2 Evolution problems

Consider an object that can dynamically inherit new methods, i.e., that can *evolve*. Apart from the mechanics of changing an object we have two very difficult problems. First, how does an object decide that it is advantageous to inherit or modify a method or a part of itself? The object may be forced to do so, which means that its independence was inherently limited, i.e., it was not fully encapsulated. Or the object may have no alternative, which means it has detected a malfunction. Finally, the object may inadvertently inherit a "bad" trait. In this case "bad" may mean incompatible, or inconsistent with some of its other characteristics. Evolving means not only adding but also rejecting methods and parts. This suggests another interesting problem: How does an object decide that a particular method or trait is disadvantageous and should be rejected? In special cases a metric may exist to evaluate new methods or shed existing ones. For instance, we have implemented objects for which rules relating to buying and selling stocks were evaluated according to their performance in the market [Tschritzis et al. 1987]. In general, the situation is much more complex, unless we take the approach that each object is a total actor and its methods are always passed in the messages.

2.3 Global behaviour problem

Consider a population of independent objects. The behaviour of each object is defined by its methods. We can specify what an object is supposed to do by properly defining it and controlling it during its lifetime. The global behaviour of the expanding and evolving population is implicitly defined by the behaviour of its parts (the objects). This implies that we cannot specify global

behaviour. The notion of a formal specification of global behaviour being used to generate the object population is in conflict with the idea of free-wheeling, independent objects.

We may want, however, to *constrain* the global behaviour. Consider, for instance, the notion of a flock of birds [Maruichi et al. 1987]. The global behaviour of the flock results from the combined behaviour of its independent objects (birds). We have, for example, a notion of a flock flying south. There are two ways, at least, to coordinate objects. In one approach we can define scripts that participating objects should follow [Fiume et al. 1987]. These scripts are constraint mechanisms comparable to a scenario in a play: an actor is capable of acting many different parts, any one of which can be performed in a variety of ways. But in any given play the actor must conform to the constraints of that play's script. Scripts are very valuable, especially for temporal coordination of objects.

Another way to coordinate objects is through the definition of fields [Pintado and Fiume 1988]. A field is a population of objects each of which is subject to a force defined in the point it occupies. In this way, we can define fields where objects can be attracted or repulsed. A field does not totally define the behaviour of each object. It partially influences their collective behaviour according to some desired overall goal.

2.4 Presentation problem

Consider a dynamic, evolving object. We need to portray the different states that the object can reach in a suitable fashion. This is especially important for users to be able to visualize and monitor the progress and interplay of objects. An active object cannot be meaningfully portrayed by a static icon. We need to represent it by an animated character. In this way object states will be mirrored in the character's appearance. The animated character behaves in a fashion that reflects the progress of the object it represents and its coordination with other objects. The user can then be trained to look for abnormal object behaviour. The same script that coordinates object behaviour can define the behaviour of the corresponding animated objects [Fiume et al. 1987]. We are readily immersed in problems of event-driven simulation and real-time animation. In addition, we have complete freedom in portraying animated objects. If they correspond to real-world objects, we can choose to represent them by animation of their real-world counterparts. For objects that are completely imaginary, we need to create an artificial reality that properly portrays them. Objects not only have to do the right things, but they have to look good.

2.5 Defense problem

It is probably improper to talk about *protection* of objects. Dynamic, moving objects are not passive units that need protection. They can actively organize their own defense. An object can refuse to allow certain operations. It can hide information. It can provide disinformation. It can move outside a context where it cannot be reached. It can completely erase itself. An object can even defend itself by directly or indirectly attacking the intruding object. All these possibilities give protection mechanisms a new meaning.

We can consider special objects with proper defense mechanisms that guard more sedate but critical objects. There is no limit in the ways that objects may defend themselves, just as there is no limit in the ways that objects can attack other objects. Proper object behaviour is not supervised by any one user, but by a combination of object managers who can watch out for signs of aggression.

2.6 Temporal problem

So far objects operate independently, perhaps in coordination with other objects. We may, however, want to force objects to be at certain points or states at certain times. We need to introduce, therefore, a notion of time, real or artificial, to force not only ordering of operations but rendez vous of objects. This is especially important when objects are interfaced with machines and robots. Objects operating these processes need to abide by real-time requirements. Their interaction, therefore, with the other free-wheeling objects has to be subjected to some strict temporal constraints. An event-driven script controlled by independent samples is a plausible mechanism for forcing a rendez vous [Dami et al., 1988] (provided, of course, that the machines and the networks perform well enough to support object coordination). The same issues come up with the interfaces of video for the visualization of objects. The operations of the objects have to be rendered in animation with specific audiovisual characteristics (frames/sec., voice or sound time delay, etc). The only difference is that people for whom the objects are rendered may be more flexible in terms of real-time constraints than machines.

Summary

The paradigm of object-oriented programming has proven its value in many areas, such as enhancing software reusability, maintainability and reliability. Despite the obvious benefits of the object-oriented approach, it is often unclear how object models affect traditional issues such as strong-typing, security, concurrency, and distribution. We are faced not only with the challenge of integrating object-oriented and traditional approaches, but we must constantly be prepared to discover new ways in which object models can help us to find better solutions to old problems. What we have classified as traditional research directions are the most immediate and obvious problems of fitting the object-oriented approach into the traditional body of known computer science.

All the exotic problems we have discussed carry connotations of science fiction. However, they are not really new. They all have counterparts in programming problems known and sometimes unresolved. We propose the following correspondence.

The acquaintance problem is related to selection and binding in programming. The evolution problem is related to software maintenance and modification. The global behaviour problem is related to constraint specification and verification. The presentation problem is related to multimedia user interfaces. The defense problem is related to protection and security issues. The temporal problem is related to real-time issues in operating systems, simulation and animation.

We should not dismiss the traditional directions as old and uninteresting just because the exotic directions are new and exciting. They are both equally important. The main difference is that the traditional directions emphasize getting an object right, whereas the exotic directions emphasize the societal problems in a population of objects, that is, getting a set of objects to do something together. It is well-accepted in computer science that getting autonomous objects, programs, people or whatever to work together is far more difficult. Without mechanisms for cooperation, however, we are limited to what can be done by a single object, program or person. Recent interest in computer-supported cooperative work points out that getting persons and programmed objects to cooperate for an overall goal can be a critical issue.

References

- [Agha 1986] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

- [Dami, et al. 1988] L. Dami, E. Fiume, O. Nierstrasz and D. Tsichritzis, *Temporal Scripts for Objects*, 1988, (submitted for publication).
- [Cardelli and Wegner 1985] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, vol. 17, no. 4, pp. 471-522, Dec 1985.
- [Fiume, et al. 1987] E. Fiume, D.C. Tsichritzis and L. Dami, "A Temporal Scripting Language for Object-Oriented Animation", *Proceedings of Eurographics 1987 (North-Holland)*, Elsevier Science Publishers, Amsterdam, 1987.
- [Halbert 1984] D.C. Halbert, "Programming by Example", Ph.D. Thesis, Dept. of EE and CS, University of California, Berkeley CA, 1984.
- [Hoare 1985] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
- [Maruichi, et al. 1987] T. Maruichi, T. Uchiki and M. Tokoro, "Behavioral Simulation Based on Knowledge Objects", *Proceedings of the European Conference on Object-oriented Programming*, pp. 257-266, Paris, France, June 15-17, 1987.
- [Meyer 1986] B. Meyer, "Genericity versus Inheritance", *ACM SIGPLAN Notices Proceedings OOPSLA '86*, vol. 21, no. 11, pp. 391-405, Nov 1986.
- [Moss and Kohler 1987] J.E.B. Moss and W.H. Kohler, "Concurrency Features for the Trellis/Owl Language", *Proceedings of the European Conference on Object-oriented Programming*, pp. 223-232, Paris, France, June 15-17, 1987.
- [Nierstrasz 1987a] O.M. Nierstrasz, "Hybrid – A Language for Programming with Active Objects", in *Objects and Things*, ed. D.C. Tsichritzis, pp. 15-42, Centre Universitaire d'Informatique, University of Geneva, March 1987.
- [Nierstrasz 1987b] O.M. Nierstrasz, "Active Objects in Hybrid", *ACM SIGPLAN Notices Proceedings OOPSLA '87*, vol. 22, no. 12, pp. 243-253, Dec 1987.
- [Pintado and Fiume 1988] X. Pintado and E. Fiume, "Grafields: Field-Directed Dynamic Splines for Interactive Motion Control", *Proceedings of Eurographics 1988 (North-Holland)*, Elsevier Science Publishers, Amsterdam, 1988, (to appear).
- [Schaffert, et al. 1986] C. Schaffert, T. Cooper, B. Bullis, M. Killian and C. Wilpolt, "An Introduction to Trellis/Owl", *ACM SIGPLAN Notices Proceedings OOPSLA '86*, vol. 21, no. 11, pp. 9-16, Nov 1986.
- [Tsichritzis, et al. 1987] D.C. Tsichritzis, E. Fiume, S. Gibbs and O.M. Nierstrasz, "KNOs: KNowledge Acquisition, Dissemination and Manipulation Objects", *ACM TOOIS*, vol. 5, no. 1, pp. 96-112, Jan 1987.
- [Wegner 1987] P. Wegner, "Dimensions of Object-Based Language Design", *ACM SIGPLAN Notices, Proceedings OOPSLA '87*, vol. 22, no. 12, pp. 168-182, Dec 1987.
- [Wolczko 1987] M. Wolczko, "Semantics of Smalltalk-80", *Proceedings of the European Conference on Object-oriented Programming*, pp. 119-131, Paris, France, June 15-17, 1987.

[Yonezawa, et al. 1986] A. Yonezawa, J-P Briot and E. Shibayama, “Object-Oriented Concurrent Programming in ABCL/1”, ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 258-268, Nov 1986.