Object-Oriented Concurrent Programming: Introducing a New Programming Methodology

Jean-Pierre BRIOT*

LITP, IBP
Université Pierre et Marie Curie (Paris VI)
4 place Jussieu, 75252 Paris Cedex 05, France
E-mail: `briot@litp.ibp.fr`

(invited paper) in Proceedings of the 7th International Meeting of Young Computer Scientists (IMYCS'92), Topics in Computer Science Series, Gordon & Breach, 1993.

Abstract

This paper discusses the introduction of new programming methodologies to computer science and practice. Programming is ongoing deep evolution because the use of computers is getting larger and wider. As a consequence, programs are becoming more complex, dynamic, and resource demanding. This implies new methodologies to help decomposing large and complex programs and running them efficiently onto new parallel machines. *Object-oriented concurrent programming* (OOCP at short) is one of the leading methodology to match these requirements. It is based on the notion of active and autonomous entities (named *objects*) which run concurrently and communicate by exchanging *messages*.

This paper introduces the concepts of OOCP methodology and discusses its advantages. It also surveys current application fields and research issues. We then focus on the issue of introducing and teaching OOCP. We describe a platform which we designed in order to help describing and experimenting with OOCP. We finally discuss the experience of using this platform for pedagogical purposes and experiments.

*Currently visiting Yonezawa Lab., Dept. of Information Science, Faculty of Science, University of Tokyo, Hongo, Bunkyo-ku, Tokyo 113, Japan. E-mail: `briot@is.s.u-tokyo.ac.jp`

1 New requirements for Programming

The use of computers is now becoming ubiquitous in our society. Early use of computers was restricted to scientific computation and data processing, but application fields are now wide open. We expect computer systems to solve more efficiently more complex problems. This leads to more sophisticated programs also more demanding in computing resources.

In order to program large and complex applications whose characteristics and environment may be dynamically changing, we are in needs for novel programming methodologies helping for decomposition and evolution. Resulting programs should also run efficiently. New generation processors have ever increasing speed, but the only way to achieve large performance is by taking advantage of new multiprocessor architectures. This implies having *concurrent* programs which exhibit potential parallelism of computation which then can be scaled down onto various parallel architectures.

2 Object-Oriented Concurrent Programming

Object-oriented concurrent programming (OOCP at short, also named *concurrent object-oriented programming*) is a very promising methodology to match these requirements. It describes computation through a collection of small self-contained modules (named *objects*) which compute and interact concurrently through some unified communication protocol (named *message passing*). This provides a good foundation for decomposing programs and running them efficiently onto multiprocessors.

2.1 Objects and Messages

Object-oriented concurrent programming takes its ground from the *object-oriented programming* (OOP at short) methodology. Object-oriented programming has been recognized as the leading methodology for developing high-level and reusable software.

This methodology is conceptually simple and general enough to be applied widely. It is based on two main concepts: *objects* which identify knowledge (that is data and services), and a unified communication/activation protocol between them, called *message passing*. Objects represent various entities and concepts composing the problem domain. They will be animated like for a simulation. There is no global algorithm but a cooperation of various objects which exchange information and requests through messages. Note that it is a transposition of the way various human communities solve problems in a cooperative way. Other useful concepts of object-oriented programming are *abstraction* (the *class* concept) and *inheritance* (the *subclass* concept), which allow to specify, classify and reuse object descriptions[1].

The first occurrence of a programming language based on these concepts came out in the late 60's. The concepts were initially derived from a simulation language into a general framework for programming (as simulation of the world). Simula-67 [BDMN73] was the first object-oriented programming language and was quickly followed by Smalltalk which added the interaction dimension through its wonderful interactive programming environment [GR83]. The Smalltalk system was a direct influence of Apple's Macintosh, which became a key step to popularize this new vision of the computer. The concepts

[1] To be more precise, not all OOP languages rely on the class concept, although this is the case for most of them. A few other ones are based on the alternative notion of *prototype*. [Weg87] proposed a more precise taxonomy/terminology of OOP languages whether they support or not concepts of class and inheritance.

of OOP were then clearly identified, and the methodology became validated by software engineering as a significant improvement in program development, quality, maintenance, and reusability [Mey88].

But note that because of the sequential nature of conventional processors and programs, autonomous activity of each object leading to simultaneous activities, although implicit in the metaphor, has usually not been addressed by OOP systems. As a result, most of OOP languages still remain sequential. One object is active at a time, and activity is transferred from one to another along message passing.

2.2 Concurrency

Meanwhile, as stated in Section 1, the ability to express potential parallelism of activities through concurrent programs is becoming a key issue for future computer systems. The emergence of new multi-computers which offer parallel execution brings promises for increasing efficiency. Another motivation for concurrent programming lies in the distribution of knowledge, as illustrated by computer networks, and the multiple interaction aspects of large interactive programs (such as for process control, where input and control may be multiple and distributed). Furthermore, many problems may be naturally expressed in terms of concurrent and coordinating activities, whereas finding serial algorithms to compute them may be not trivial.

Concurrent programming allows to abstract from specific execution rates on specific architectures. Therefore *concurrency* (logical simultaneity of activities) should be distinguished from *parallelism* (physical simultaneity). *Synchronization* is the way to ensure coordination and non-interference of activities by specifying constraints on the ordering of events [AS83].

2.3 Objects + Concurrency = OOCP

In order to achieve optimal integration of OOP and concurrency, it is fruitful to identify objects as the unit of activity and to associate synchronization between objects at the message passing level. These *active objects* are also oftenly called *actors*. This unification achieves integration of concepts of OOP and concurrency and frees the programmer from having to explicitly take care of most of synchronization discipline. This also preserves modularity and simplicity of OOP while enforcing self-containedness and autonomy of objects.

This integration of object-oriented programming with concurrent programming is named *Object-oriented concurrent programming* [OOCP87]. It appears as the natural generalization of object-oriented programming by giving further autonomy to objects. (We may then reformulate that sequential OOP was a technological restriction of the more general concepts of OOCP!).

Autonomous activity of each object exhibits concurrency between them. This is called *inter-object* concurrency. One may also bring finer grain concurrency by allowing an object to handle several messages concurrently. This is called *intra-object concurrency*. In this case, there is the need to control the scheduling of message acceptance. This is usually specified through conditions to enable/disable acceptance of various patterns of messages. Such control is also useful to possibly specify delay of message acceptance based on the state of the object (e.g., a data structure would delay put requests while it is full). Note that finding optimal specifications to control scheduling of message acceptance is still an active research issue. Current focus is on how to provide specifications which are highly generic and reusable (see Section 2.7).

2.4 Interaction Protocols

Message passing is the means for interaction and coordination between objects. All various patterns of interaction: informing, requiring some information, requesting some computation... are subsumed by the message passing concept. Sending a message triggers the activation of one of the *methods* (operation associated to a pattern of message) of the receiver with the arguments conveyed in the message.

Because of the potential competing activities of the sender and the receiver, it is useful to buffer messages in order to: possibly support distinct communication speeds within sender and receiver, and free the sender of unnecessary wait when no reply is needed. Therefore each active object owns a *message queue* (also called mailbox) which buffers its incoming messages. This implies the existence of a mail delivery system which conveys and delivers the messages once sent.

In some interaction patterns (like informing), no reply nor synchronization is needed. Thus the sender may resume its activity as soon as after sending the message. This is called *asynchronous message passing*. In some other patterns, the sender waits for a reply or acknowledgement. This is called *synchronous message passing*. Asynchronous message passing increases concurrency by desynchronizing sending, delivery, acceptance, and reply of a message. OCCP languages usually provide these two types of message passing, and possibly other ones, like the eager-reply type (also called *future*) which immediately replies a promise of the future reply.

2.5 Advantages

Thanks to its inner decomposition and modularity, OOCP helps decomposing large programs into a collection of small cooperative modules, and running them efficiently by exploiting new multiprocessor architectures. Note that message passing is a communication metaphor independent of assumptions of shared memory, and consequently appropriate for distributed memory multiprocessors.

Main advantages of OOCP may be stated as following. Remark how they match the requirements discussed in Section 1:

- **high-levelness**

  The programmer focuses on the roles of the interactions between the various kinds of conceptual entities describing the problem. Therefore he abstracts from low-level implementation concerns.

- **implicit decomposition of concurrent activities**

  Units for concurrency are clearly identified through the decomposition of programs into a collection of interacting objects. Each object is autonomous, leading to concurrency between objects. Further concurrency may be gained through internal concurrency within objects.

- **transparent synchronization**

  Synchronization between concurrent activities takes place in the design of their interaction and coordination patterns. Designers don't have to be concerned with low-level synchronization.

- **locality and self-containedness**

  Objects are self-contained entities grouping knowledge, actions, and resources. Locality of objects eases their parallel execution and dynamic reallocation on different processors.

- **dynamicity and openness**

  Objects may be dynamically created and recovered. They may acquire new knowledge and change their acquaintances along the information they receive. Organizations of objects may be dynamically reconfigured and new objects may be created along the needs for new resources during the course of computation.

- **multi-granularity**

  Objects of various granularities (size, knowledge, internal concurrency) may cohabit and cooperate within a same system. This helps decomposing the design of a complex system into various levels. Examples may be found in integrated intelligent architectures (from large-grained scene recognition blackboards down to fine-grained neurons), see Section 2.6.

2.6 Applications

Object-oriented concurrent programming is being applied to a growing number of fields. Applications are specially growing strong in the following fields: distributed operating systems [Chorus88], (distributed) artificial intelligence [DAI88], distributed simulation [SY90], distributed data-bases [Agh88], office information systems [Hew86], real-time systems and (distributed) process control [Bar89]. Significant results may also be found in other fields, e.g., text parsing [YO90], and computer music [CBS87]. OOCP has also main impact on new multi-processor architectures, like the J-Machine [Dal89].

Object-oriented concurrent programming fits specially well with the new area of artificial intelligence, called distributed artificial intelligence (DAI in short) [DAI88], which takes much input from sociology and organization theories. DAI intends at solving problems in a distributed way through a collection of *agents* which coordinate their knowledge and behavior (consequently they are also often called *multi-agent* systems). Objects are a very good base to build-up intelligent agents. But agents need higher-level and more explicit representations of each-other's knowledge, beliefs, and course of action. [GB92] discusses the relations between (active) objects and agents.

Uniformity of communication and interaction mechanisms as well as encapsulation enable modeling various granularities of objects (size, knowledge, internal concurrency...). For example objects can model very fine entities such as neurons [WA91], or large components of expertise such as found in large-grained multi-agent systems (e.g., MACE [GBH87]). This helps constructing multi-level/granularity architectures such as an architecture for brain simulation [WA91].

The OOCP metaphor of communities of interactive entities fits also very well to support alternative programming models and systems. These new *metaphorical* programming models are the results of transferring scientific theories from the real world to provide alternative computation models. A famous example is the case of (artificial) neural networks [PDP86], abstraction of neurophysiology. Other examples include abstractions from chemistry (e.g., the chemical abstract machine [BB90]), genetics (genetic algorithms [Gol89]), ethology (e.g., ant nests), economy (e.g., market-based models), and other various kinds of self-organizing systems [Eco88].

2.7 Research Issues

Being a novel field, object-oriented concurrent programming generates much effort in various research issues.

Object-oriented concurrent programming has been initially defined conceptually and not formally. There is currently much activity to provide more formal descriptions. It is non trivial challenge, because OOCP is highly dynamic (creation, reconfiguration during computation) while computer theory has mainly focused on statically definable and synchronous systems. A computation model for actors has been based on transitions by Gul Agha [Agh86], and a theory of communicating concurrent objects has been proposed by Robert Milner [Mil89]. One of the important issue is how to express communicating labels and behaviors of active objects. This is important for objects to dynamically acquire and reconfigure their acquaintances and to possibly change behavior. Some recent work in this direction is the π-calculus [MPW89].

In order to model self-descriptive and self-adaptive computer systems, *reflection* has been proposed by Brian Smith [Smi84] as a methodology for a system to describe and control its structure and course of computation. The principle is to provide a representation of some key aspects of the system within the system itself, and to causally connect this representation with the actual system. Reflection makes implementation and control of computation becoming explicit at the level of the system itself. They may be accessed and changed with the same high-level metaphor used at object-level computation. In object-oriented (concurrent) programming systems, there is the issue of self-representing the whole system or individually each object. In fact there are needs for combining reflection at the individual level (a self-representation of an object is called a *meta-object* [Mae87]) and at a *group* level in order to grasp also interactions and sharing of resources (e.g., allocation, scheduling) between objects [MWY91].

Object-oriented programming is famous for the way it helps managing modular and reusable programs, thanks to the inheritance mechanism. Unfortunately, in object-oriented concurrent programming, using inheritance to reuse synchronization specifications (on scheduling of message acceptance) turns out not to always work smoothly. This problem has recently been coined as *inheritance anomaly*. Finding the most expressive, generic and reusable synchronization scheme is currently an active field of research. Recent results (see [MTY93] for a very good study and proposal) show that synchronization specifications need to be: abstract enough, apart from method definitions, among various synchronization schemes proposed (because there is (currently) no single synchronization scheme which is a silver bullet), and, last but not the least, efficient.

Another important issue is the needs for programming environments. The abundance and competition of activities makes it difficult to apprehend the course of computation(s). We need multi-representation-based sophisticated environments to visualize and control activities and cooperation of objects. Some work has been done on specific aspects (visualization, debugging...) but we need to work on a good integration of a set of generic tools. The integrated programming environment of Smalltalk-80 is a good example in the context of serial object-oriented programming. (This is the reason why our project chose Smalltalk-80 as the basis to prototype OOCP environment tools, see Section 3.4 and [LBB91].)

To conclude this quick survey, let's notice that the frontier between programming language, operating system, and data-base is blurring. There is a high research activity to achieve optimal integration between these distinct but complementary perspectives.

3 Teaching

In this section we will discuss the issue of introducing this new programming methodology to students and conventional programmers. This discussion is based on our personal experience in teaching object-oriented (concurrent) programming in courses and seminars at universities, engineering schools, and companies. We also developed a system to help describing object-oriented concurrent programming, classifying various programming languages of that kind, and experimenting with them.

3.1 Introducing Objects

Object-oriented programming is more conceptual than conventional programming. It focuses on interaction between entities, abstracting from their representation and implementation details. Consequently we believe that this makes it easier for programmers and non-programmers to catch up with this methodology. Our opinion is supported by our experience of introducing the concepts of object-oriented programming to various audiences.

3.2 From Objects to Actors

Object-oriented programming is getting now heavily disseminated in the programming community. A hot issue is how to transfer this expertise into object-oriented concurrent programming. As pointed out in Section 2.3, object-oriented concurrent programming is a generalization of object-oriented programming. However some of the mechanisms developed for object-oriented programming need some evolution to fit well with the concept of concurrency.

Another problem is the relative immaturity of the field of OOCP. This results in the abundance of prototype OOCP languages [OOCP87]. We remarked that it is not always easy to analyze and compare these languages and discuss their design decisions. Comparing various semantics needs to abstract from various syntax and implementation supports.

Note that description and classification abilities are among main strengths of object-oriented programming. This led us to design some platform based on OOP (namely Smalltalk-80 [GR83]) to model OOCP languages.

3.3 The Actalk Platform

The Actalk platform introduces (OOCP) actors into the world of objects. We chose the Smalltalk-80 programming language and environment as the OOP basis for reasons discussed below. Consequently we named our platform *Actalk*, which stands for *act*ors (or *act*ive objects) in Small*talk* [Bri89]. Actalk provides support for a comprehensive description of object-oriented concurrent programming. It has been used as the experimental basis of numerous courses on OOCP.

We will first discuss the design decisions which led to the actual system. The we will discuss its pedagogical purposes, and survey the status of the system.

3.4 Design Decisions

To design a system for integrating various OOCP languages into a single environment, we had the following goals in mind:

- *uniformity and modularity*
  We wanted to model in a modular way various OOCP languages into some common and unified programming environment. Therefore we chose the object-oriented

paradigm as a very good support for these uniformity and modularity requirements. We smoothly introduce actors into standard (sequential) OOP, by defining a sub-world of actors embedded into the world of objects, and without changing the underlying object system.

- *minimality and extensibility*
  We wanted some minimal kernel expressing the most general semantics of OOCP languages, and to further extend it in order to simulate various existing languages. Therefore we chose a minimal architecture based on a minimal kernel. We then further extend it by inheritance to simulate and classify various OOCP models.

- *an integrated environment*
  We didn't want to restrict our system to some semantic model and raw implementation, but also to provide a full environment for pragmatic experiment with OOCP. Therefore we chose Smalltalk-80 because it is the most achieved and flexible OOP system with a fully integrated programming environment. We extend it further to prototype programming environment tools for OOCP [LBB91].

By choosing Smalltalk-80 we achieve a minimal implementation of the system, because all entities needed to build actors: objects, classes and messages, and to express concurrency: processes and semaphores, are provided by Smalltalk-80. Thus implementation is both high level and minimal (the kernel of the platform fits within 2 pages of code [Bri89]). This allows focusing on the semantics of OOCP and the differences with standard OOP, rather than dealing with low level concerns.

3.5 Pedagogy

Actalk is an integrated environment, embedded into Smalltalk-80, used as a testbed to classify and design OOCP languages. Because of the optimal integration of actors within the world of objects, Actalk is also useful as a basis for studying relation and combination between objects and actors. This is of interest because many OOCP systems are hybrid in that they combine active objects and more traditional programming which may be encapsulated within active objects. Actalk has been successfully used in many courses, seminars and tutorials to describe how to move from object-oriented programming to object-oriented concurrent programming. The minimal implementation of the kernel of Actalk helps making concrete and manageable the concepts of OOCP.

Actalk has also been successfully used to describe the key concepts and constructs of some of the most representative OOCP computation models and languages. These extensions use inheritance to customize the basic OOCP semantics defined by the Actalk kernel. Because the kernel and its extensions are related by inheritance, one could easily compare them. Examples of extensions include the Actor model of computation defined by Gul Agha [Agh86] and the ABCL/1 programming language designed by A. Yonezawa and his team [ABCL90]. Simulation of other OOCP languages, like POOL [Ame87], and other concurrent languages, like CSP, have also been conducted by students in the team of Jean Bézivin at University of Nantes.

3.6 Experiments

Experiments on some of the current OOCP research issues with Actalk are also being conducted: modeling reflective computation to simulate adaptive behavior (the ReActalk system [GS91]), and prototyping programming environments for OOCP [LBB91].

Actalk has also been used to build various prototype multi-agent systems in the team of Jacques Ferber at the Laforia research lab in our university. One of them is the Mages multi-agent testbed [Bou90]. These prototype multi-agent systems have been applied to various combinatorial and cooperation problems.

All these experiences convinced us of the goods of our approach based on a progressive and layered description of OOCP concepts, and practical experiment with them, thanks to the Actalk platform (built on the wonderful environment provided by Smalltalk-80).

A research report collects papers on the Actalk project conducted in our lab [Actalk92] and describes more extensively various aspects of this platform.

Conclusion

In this paper we introduced the object-oriented concurrent programming (OOCP) methodology, which helps decomposing programs and running them onto new multiprocessors. We described its basic concepts, surveyed its current and prospective applications, and summarized the main research issues. We then discussed the issue of introducing this new leading methodology to programmers and non-programmers. Therefore we described how we designed a platform for experiments with OOCP, named Actalk. This platform has been validated as an useful tool for describing and experimenting with OOCP. We hope it could help triggering further experiments and feedback on this novel programming methodology.

This paper has been based on an invited lecture given at the European Conference on Education and Computer Science, in Salerno, Italy, in November 1991. We would like to thank Vincenzo Loia and Antonio Gisolfi for providing us the initial opportunity of discussing the issue of teaching new programming methodologies. We greatly thank all people, and Loïc Lescaudron in the first place, who took part in the development of Actalk, feedbacked on it, and reused it in various contexts.

References

[ABCL90] *ABCL: an Object-Oriented Concurrent System.* Edited by A. Yonezawa, Mit-Press, 1990.

[Actalk92] J.-P. Briot, L. Lescaudron, and M. Bouabsa, Selection of Publications on the Actalk Project. *LITP 92-68 Research Report*, Institut Blaise Pascal, Paris, France, September 1992.

[Agh86] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence, MIT Press, 1986.

[Agh88] G. Agha, The Relation Between Problems in Large-Scale Concurrent Systems and Distributed Databases. International Symposium on Databases in Parallel and Distributed Systems, IEEE, Austin TX, USA, December 1988.

[Ame87] P. America, POOL-T: A Parallel Object-Oriented Language. In [OOCP87], pages 199–220, 1987.

[AS83] G.R. Andrews and F.B. Schneider, Concepts and Notations for Concurrent Programming. *Computing Surveys*, Vol. 15, No 1, pages 3–43, March 1983.

[Bar89] B. Barry, Prototyping a Real-Time Embedded System in Smalltalk. Proceedings of OOPSLA'89, *Special Issue of Sigplan Notices*, Vol. 22, No 12, pages 255–265, October 1989.

[BB90] G. Berry and G. Boudol, The Chemical Abstract Machine. *ACM Conference on Principle of Programming Languages (POPL'90)*, San-Francisco CA, USA, 1990.

[BDMN73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug and K. Nygaard, *Simula Begin.* Petrocelli Charter, New York NY, USA 1973.

[Bou90] T. Bouron, J. Ferber, and F. Samuel, Mages: a Multi-Agent Testbed for Heterogeneous Agents. Proceedings of MAAMAW'90, *Decentralized AI*, Vol. II, North-Holland, 1991.

[Bri89] J.-P. Briot, Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. *Proceedings of ECOOP'89*, pages 109–129, Cambridge University Press, 1989.

[Chorus88] M. Rozier et al., Chorus Distributed Operating Systems. *Computing Systems Journal*, Vol. 1, No 4, pages 305–370, Usenix, December 1988.

[CBS87] P. Cointe, J.-P. Briot, and B. Serpette, The Formes Language: A Musical Application of Object-Oriented Concurrent Programming. In [OOCP87], pages 221–258, 1987.

[DAI88] *Readings in Distributed Artificial Intelligence.* Edited by A. Bond and L. Gasser, Morgan Kaufman, 1988.

[Dal89] W.J. Dally et al., The J-Machine: a Fine-Grain Concurrent Computer. *Proceedings of Information Processing Congress (IFIP'89)*, pages 1147–1153, August 1989.

[Eco88] *The Ecology of Computation.* Edited by B.A. Huberman, North-Holland, 1988.

[GBH87] L. Gasser, C. Braganza, and N. Herman, MACE: A Flexible Testbed for Distributed AI Research. *Distributed Artificial Intelligence*, edited by M.N. Huhns, pages 119–152, Pitman - Morgan Kaufmann, 1987.

[GB92] L. Gasser and J.-P. Briot, Object-Based Concurrent Programming and Distributed Artificial Intelligence. In *Distributed Artificial Intelligence: Theory and Praxis*, edited by N. Avouris and L. Gasser, pages 81–107, Kluwer, 1992.

[GS91] S. Giroux and A. Senteni, Taylorizing the Behavior of an Actor: an Organizational Approach of Computational Reflection. *Proceedings of East EurOOP'91*, edited by A. Mrasik, pages 12–26, Bratislava, Slovakia, September 1991.

[GR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Series in Computer Science, Addison-Wesley, 1983.

[Gol89] D.E. Goldberg, *Genetic Algorithms.* Addison-Wesley, 1989.

[Hew86] C.E. Hewitt, Offices are Open Systems. *ACM Transactions on Office Information Systems*, Vol. 4, No 3, pages 271–287, 1986.

[LBB91] L. Lescaudron, J.-P. Briot, and M. Bouabsa, Prototyping Programming Environments for Object-Oriented Concurrent Languages: a Smalltalk-Based Experience. *Proceedings of the 5th Conference on the Technology of Object-Oriented Languages and Systems (TOOLS USA'91)*, pages 449–462, Prentice-Hall, August 1991.

[Mae87] P. Maes, Concepts and Experiments in Computational Reflection. Proceedings of OOPSLA'87, *Special Issue of Sigplan Notices*, Vol. 22, No 12, pages 147–155, December 1987.

[MWY91] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. Proceedings of ECOOP'91, *LNCS*, No 512, pages 231–250, Springer Verlag, 1991.

[MTY93] S. Matsuoka, K. Taura, and A. Yonezawa. Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages. Proceedings of OOPSLA'93, *Special Issue of Sigplan Notices*, Vol. 28, No 9, September 1993.

[Mey88] B. Meyer, *Object-Oriented Software Construction.* Series in Computer Science, Prentice-Hall, 1988.

[Mil89] R. Milner, *Communication and Concurrency.* International Series in Computer Science, Prentice-Hall, 1989.

[MPW89] R. Milner, J. Parrow, and D. Walker, A Calculus of Mobile Processes, Parts I and II. *Reports ECS-LFCS 89–85 and 89–86*, Computer Science Dept., University of Edinburgh, U.K., March 1989.

[OOCP87] *Object-Oriented Concurrent Programming.* Edited by A. Yonezawa and M. Tokoro, MIT Press, 1987.

[PDP86] *Parallel Distributed Processing.* Edited by D.E. McClelland and J.L. Rumelhart, Vol. 1 and 2, MIT Press, 1986.

[SY90] E. Shibayama and A. Yonezawa, Distributed Discrete Event Simulation in ABCL/1. In [ABCL90], pages 157–181, 1990.

[Smi84] B. Smith, Reflection and Semantics in Lisp. *Proceedings of POPL'84*, ACM, pages 23–35, 1984.

[Weg87] P. Wegner, Dimensions of Object-Based Language Design. Proceedings of OOPSLA'87, *Special Issue of Sigplan Notices*, Vol. 22, No 12, pages 168–182, December 1987.

[WA91] A. Weitzenfeld and M.A. Arbib, A Concurrent Object-Oriented Framework for the Simulation of Neural Networks. Proceedings of the OOPSLA/ECOOP'90 Workshop on Object-Based Concurrent Programming, *OOPS Messenger*, Vol. 2, No 2, pages 120–124, ACM Sigplan, April 1991.

[YO90] A. Yonezawa and I. Ohsawa, Object-Oriented Parallel Parsing for Context-Free Grammars. In [ABCL90], pages 209–232, 1990.