ソフトウェア工学 76－3
（１９９０．１２．４）

並列オブジェクト指向言語*A-MODEL/PL*の設計と実現

野呂昌満* 原田賢一**

*南山大学情報管理学科 **慶應義塾大学理工学部計測工学科

*A-MODEL/PL*は基本ソフトウェア等の開発のために設計した並列オブジェクト指向言語である。言語設計上の最大の目標は問題領域におけるオブジェクトとその関係を自然にモデル化するための機能をできうるかぎり提供することである。このために、筆者らは*A-MODEL/PL*を

－オブジェクトが並列処理のためのシナリオを持つ、

－クライアントーサーバモデルに基づく言語である、

－オブジェクト間通信の手段として放送機能を持つ、

等の特長のある言語として設計した。また、大きなソフトウェアを開発するさいの言語の実用製を考慮して、

－強い型付けの機構を導入し、

－オブジェクトのシナリオおよびオブジェクト内の操作は手続き向きな記述を行なうことにした。

Design and Implementation of the Object-Oriented Concurrent Programming Language *A-MODEL/PL*

Masami Noro* (Masami@niq0.nanzan-u.ac.jp) Ken'ichi Harada** (harada@cs.keio.ac.jp)

* Department of Information Systems and Quantitative Sciences, Nanzan University
18 Yamazato-Cho, Showa-Ku, Nagoya, 466 JAPAN

** Department of Instrumentation Engineering, Faculty of Science and Technology, Keio University
4-1-1, Hiyoshi, Kohoku-Ku, Yokohama, 211 JAPAN

A-MODEL/PL is an object-oriented concurrent programming language for the development of production quality software. To naturally model objects and their relationships in the problem space within the software solution space, we interpret an object as an entity with its own scenario. We adopt the client-server model as the underlying computational model and a broadcasting mechanism as one of the inter-object communication methods. *A-MODEL/PL* employs a procedure-oriented description within objects and a strong typing facility is incorporated as much as possible.

1. Introduction

The most attractive feature inherent to object-oriented programming languages is that they support the modeling of objects and their relationships in the problem space within the software solution space very naturally [BOO86]. We call this concept *natural modeling*. Taking advantage of this feature, we circumvent the troublesome task of structurally transforming the problem to the solution as is necessary in traditional imperative programming languages.

The introduction of concurrency into the language is indispensable for natural modeling because the problem space contains parallelism. An absence of concurrency would lead us to less natural designs. Indeed, the major problem found inpractical object-oriented languages such as Objective-C [COX86], Class C [DEN84], Object Pascal [SCH86], C++ [STR86] and so forth, which are extension of procedure-oriented languages, is that they lack concurrency.

Although several languages have introduced concurrency [ISH86, YOK87, YON86], they still have drawbacks with respect to natural modeling:

1. They do not distinguish active objects from passive objects while both exist in the problem space.
2. An object's autonomy is not complete; objects do not have their own scenarios making it necessary to write additional driver objects.
3. They do not provide an inter-object broadcasting mechanism useful, for instance, in error message propagation.

A-MODEL/PL (*A*da' -based *MO*dular *DE*scription *L*anguage/ *P*rogramming *L*anguage) [NOR88] is an object-oriented concurrent programming language designed for the development of basic software such as compilers, editors, debuggers. Its basis is an Ada-like procedure-oriented language. The main goal of *A-MODEL/PL* is to facilitate to a great extent natural modeling by addressing the drawbacks of previous languages.

This paper discusses the design and implementation issues of*A-MODEL/PL*. In particular, how the designaddresses the shortcomings outlined above.

2. Design Issues of *A-MODEL/PL*

In designing *A-MODEL/PL*, the features needed for natural modeling were considered first. Then language features were added to address the goals of understandability and modifiability of software and language practicality.

2.1 Issues of Natural Modeling

To eliminate the drawbacks of previous object-oriented concurrent programming languages, we first examined their underlying computational models and now they effect the languages.

2.1.1 Computational Model

The underlying computation model for *A MODEL/PL* prescribing object discrimination can be summarized as follows:

1. There exist two types of object: active and passive called clients and servers, respectively.
2. Clients and servers run concurrently and communicate with one another.
3. In the computational model, a client triggers a server to perform a task and to receive the results when needed.

Previous object oriented languages [GOL83, ISH86, MOO86, YON86] do not discriminate between the active and passive objects that can be thought to exist in problem space. The reason they do not discriminate between them originated from a desire for simplicity in their underlying computational models. However, since we can identify two types of objects in problem space, we can recognize this discrimination. Thus, we employ these two types of objects in *A-MODEL/PL*.

Objects and Their States

An object in *A-MODEL/PL* is an entity which can be created dynamically. It consists of a private memory and the operations on it. A server owns a set of operations (called *visible operations*) that access the contents of its memory. Only clients may access the visible operations of servers. Servers cannot access the visible operations of other servers. Clients have no visible operations. This differentiation provides some type checking capability.

Although there are these two types of objects: clients and servers. However, hybrid objects having features of both, are introduced for programming ease. Sometimes it is too restrictive to insist that each object be either a client or server and the developer may forfeit the type-checking benefit of distinct types.

Objects in *A-MODEL/PL* start in an *active* state immediately after being created whereas, in other languages objects become active only when they receive a message [GOL83, YON86]. We call an object's activities the behavior of that object. It is natural to think of objects as autonomous entities which have independent behaviors because true objects in problem space usually do. Without behavior for objects, we would be forced to implement a driver object to supervise. This is not desirable in terms of natural modeling.

Besides being active, objects can be *waiting*. Waiting occurs when:

1. A client is forced to wait when a server is not ready in synchronized communication.
2. A client waits for the termination of a operation during synchronized communication.
3. A server waits for one of its operations to be invoked by a client.

Inter-Object Communication

In general, there are two kinds of inter-object communication methods: one-to-one and one-to-many. Many types of one-to-one communication methods have been devised and employed in previous languages [DOD83, GOL83, INM84, ISH86, YOK87, YON86]. These can be categorized into one of four types: synchronous and asynchronous data passing, and synchronous and asynchronous data communication. Data passing is one-way while data communication is two-way. Objects may be forced to wait in synchronous type

[1]Ada is a registered trademark of U.S. Government

communication but not in the asynchronous type. These four types of methods have been implemented in previous languages but few languages provide all four. For the sake of natural modeling, all four are adopted in our computational model.

Meanwhile, one-to-many communication has never been explicitly realized in previous languages. However, it is sometimes needed, for instance, to deliver an error message to multiple objects. Our problem space requires such communication. Hence, one-to-many communication is necessary for natural modeling. We employ broadcasting as our one-to-many communication method.

In addition to the above, interrupts are also crucial for natural modeling as seen in the express mode message passing in ABCL/1. We use interrupt mode data communication as well.

2.1.2 Language Features Supporting Natural Modeling

To group objects having similar properties, *A-MODEL/PL* provides classes as templates of objects. Hence, all features dealing with objects are class features.

Object Discrimination
To discriminate between active and passive objects, *A-MODEL/PL* provides client and server classes. A hybrid class is provided for hybrid objects.

Fig.1(a), (b), (c) and (d) dipict outlines of the class definitions. Each class has a specification and an implementation part. The specification presents the interface of the class. That is,

1. the classes used are ennumerated in the client's specification,
2. the visible operations are listed in a server's specification, and
3. both are listed for a hybrid specification.

Implementation parts define the realization of the specifications.

Object Autonomy
As in Fig.1, the implementation part of each class has a behavior. The client's behavior includes the order of invocation of the visible operations of the servers it uses. The server's behavior specifies the order in which invocations are accepted. Orderings can be structured by concatenation, selection, and iteration. Mutual exclusion among objects is achieved in server by a (conditional) selective wait as in Ada tasks.

Inter-Object Communication
Since *A-MODEL/PL* was designed under the client-server model, all communication is limited to between clients and servers. To realize all the communication methods in our computational model, *A-MODEL/PL* provides the following primitives:

1. Synchronous operation calls: A client tries to synchronizes with a server by calling one of the server's operations. The client waits until the server is ready and for the server to reply. As
   ac2->p2(...) **Ignore**
   in Fig.1(a), the client can ignore replies using the {\bf ignore} option.
2. Asynchronous operation calls: A client need not wait until the server replies. It may obtain the reply later via the **receive** statement. If the server still has not replied by this time, the client is forced to wait. A pair of **Issue** and **receive** statement,
   Issue *ac1->f*(...);
   and
   x := **receive** *ac1->f*;
   is an example of the asynchronous operation calls. The **Ignore** option can be used here as well.
3. Broadcasting: A client can broadcast to send data to all designated objects at the same time. The description
   broadcasting(...) *ac1*, *ac2*;
   in Fig.1(a) specifies an example of broadcasting to two objects. The default designation is to broadcast to all instantiatedobjects.

Thus, one-to-one communication methods can be realized as:

1. Synchronous data passing is realized as a synchronous operation call while ignoring the server's reply.
2. Asynchronous data passing corresponds to an asynchronous operation call ignoring the reply.
3. Synchronous data communication is the same as the synchronous operation call.
4. The asynchronous operation call is used for the asynchronous data communication.

To communicate between clients, *A- MODEL/PL* provides a predefined class named *pipe* implemented with the above constructs. The *pipe* is a data buffer which mediates between clients. There are four types of *pipe*:: the various combinations of one-way or two-way, and synchronous or asynchronous. Synchronous data passing between clients, for example, is implemented as communication via a one-way, synchronous *pipe.*

Ows_Pipe: *one_way_syn_pipe*;

in Fig.1(a) is a declaration of the one-way and synchronous *pipe* and

Ows_Pipe->Put(...);

is an example of its usage.

An object may include a description on how to react to a broadcast. The description starting with a set of reserved words, **broadcast Is** in both Fig.1(a) and (b) defines an action to a broadcast. In the absence of a description, the object ignores the broadcast.

The exception handling facility is used to realize interrupts. For interrupts, only synchronous data communication is possible. While the interrupt is being processed, no other interrupts can occur. A procedure, *e1* in Fig.1(b) is an example of an exception operation. A broadcast operation can be stated as an exception as in Fig.1(b) **broadcast**(...) **Is exception**;.

2.2 Writing Understandable and Modifiable Software

To achieve the goal of understandability and modifiability of software, *A-MODEL/PL* offers constructs for data abstraction and modularity, the software engineering principles underlying these goals [BOO86].

```
-- specification part
class c is client
  use c1, c2, one_way_syn_pipe;
  broadcast is exception;
end c;


-- implementation part
class body c is
  -- variables, procedures and functions
  -- local to c
  Ows_Pipe : accessone_way_syn_pipe;
                -- pipe declaration
  x : integer;

  -- initialization procedure
  initialization(a1 : access c1; a2 : access c2) is
  begin
    ac1 := a1; ac2 := a2; ...;
  end initialization;

  -- finalization procedure
  finalization is begin ... end finalization;

  -- action taken if interrupted and broadcasted
  broadcast(...) is exception ... end broadcast;

begin -- behavior part
  issue ac1 → f(...); -- asynchronous operation call
  loop
    ac2 → p1(...); -- synchronous operation call
    broadcasting(...) ac1, ac2; -- broadcasting
    ac2 → p2(...) ignore;
                -- synchronous operation call with
                -- ignore option
    Ows_Pipe → Put(...);
                -- pipe usage
  end loop;
  x := receive ac1 → f; -- get the reply
  raise ac2 → ex(...); -- raise exception
  raise broadcasting(...);
                -- broadcast to all objects
                -- parameters are checked in
                -- receiver's own context,
                -- the broadcasting is ignored
                -- if type crach occurs
end c;
```

(a) The Client Class

```
-- specification part of a super class
class Super is server
  -- visible operations
  procedure p1(...) is pseudo;
                -- it is a pseudo operation
                -- can be redefined in subclass

  function f1(...) return x_type;
  procedure e1(...) is exception pseudo;
                -- it is an exception and a pseudo

  -- this class can accept both type of broadcasting
  broadcast(...) is pseudo;
                -- normal mode, can be redefined
  broadcast(...) is exception;
                -- interrupt mode
end Super;

-- implementation part of a super class
class body Super(...) is
  -- local declarations

  procedure p1(...) is pseudo ... end p1;
  ...
  procdure e1(...) is exception ... end e1;

  broadcast(...) is ... end broadcast;
  broadcast(...) is exception ... end broadcast;

  initialization(...) ... end initialization;
  finalization ... end finalization;

begin -- behavior part
  accept broadcast;
  loop
    select -- selective wait
      when ... ⇒ -- selective condition
        accpet f1;
    or
      ...
    end select;
    pseudo;  -- subclass's behavior will be expanded
                -- if subclass's instance
  end loop;
  accept ;
end Super;
```

(b) The Server Class: Superclass

Fig.1. Class Description in *A-MODEL/PL*

```
-- specification part of a subclass
-- inherits properties of Super
class Sub is server inherits Super
  procedure p1(...); -- redfining procedure;
  ...
end Sub;

-- implementation part of a subclass
class body Sub is
  ...
begin
  -- this behavior will be replaced with
  -- superclass's pseudo statement
  loop ... end loop;
end Sub;
```

(c) The Server Classes: Subclass

```
-- specification of hybrid class
class h is hybrid inherits Super
  use c1, c2, c3;
  procdure p1(...); -- redefining procedure;
  ...
end h;
```

(d) The Hybrid Class

```
-- specification part of type class
class Type_Scheme(...) is type
  procedure p(...) is pseudo;
  function f(...) return integer;
  ...
  -- may also export variables
  a : array[1..100] of character;
  aType_Scheme : access Type_Scheme;
  ...
end Type_Scheme;


-- body part of type class
class body Type_Scheme is
  -- variable declarations
  procedure p(...) is pseudo ... end p;
  ...
  initialization(...) is ... end initialization;
  -- no behavior part here
end Type_Scheme;
```

(e) Class as Type Scheme

Fig.1. Class Description in A-MODEL/PL (Cont.)

Data abstraction is done by taking advantage of the separation of the specification and implementation part of the server clas. The internal data structure and details of the operations are encapsulated in the implementation part, and internal memory of the server may be accessed only through the visible operations listed in the specification part.

For modularity, class hierarchies and a mechanism for single inheritance are provided. In the hierarchy, the common properties of several classes are grouped together as a superclass and the different properties distinguish subclasses.

In addition to class hierarchy and inheritance, the virtual resource concepts as in SIMULA-67 [BIR79] are implemented with pseudo operations yielding modifiability. A pseudo operation declared in a superclass can be redefined in a subclass at runtime.

2.3 Language Practicality

Strong Typing
A-MODEL/PL is designed as a strongly typed language and has primitive types as with Ada's standard packages. One exception is the introduction of a late-binding mechanism for *MODEL/PL* the intra-class description is written in a traditional procedural way.

Class as Type Constructor
Classes can be used as type constructors as well as object templates. The reasons are:

1. It is not practical to have only one construct for concurrent entities and to use it to represent both types and object templates since the granularity of objects may sometimes be too small.
2. For the sake of simplicity, we do not want to have two different kinds of constructs as in Ada [DOD83] or Argus [LIS83].

Fig.1(e) sketches the class description for type constructor. Different from an object template, it includes no behavioral part and can have variable declarations in the specification part. However, it is similar to an object template and provides hierarchies, inheritance, and the pseudo concept.

3. Implementation

A first prototype of *A-MODEL/PL* has been implemented as a pre-processor to C under Sun UNIX\footnote[2] . Below we describe several of the implementation issues.

3.1 Redefinition of Pseudo Operations

We implement the redefinition of pseudo operation by taking advantage of pointers in C. A pointer to an integer is provided for each pseudo operation to hold the redefined function. The pointer is assigned to the provided pointer at the time of instance creation. It is cast to the required type. Runtime routines take care of these setup operations and the other operations needed for instance extinction.

[2]. UNIX is a trademark of AT&T Bell Laboratory

3.2 Inter-Object Communication

Sockets [SUN86], a monitor call in BSD UNIX, are used to implement inter-object communication. UNIX has two types of socket: *stream sockets* and *datagram sockets*. We select the stream socket for our communications as the order of the transmitted data needs to be preserved.

Implementation of Operation Call
Fig.2(a) shows the implementation of a synchronized operation call. For each server process, a queue called an *operation-call queue* is made. Fig.2(b) presents a detailed algorithm for a server to handle operation calls. The request found nearest to the head of the queue is selected.

An asynchronous operation call can be implemented by putting the code that is between the **issue** statement and **receive** statement, after the client's step 5 in Fig.2(a). The **ignore** option of the operation call is implemented by omitting step 8 and 9 in the client object and step 7 through 9 in the server object.

Implementation of Broadcasting
Broadcasting is implemented as an operation call to the supervisor process. There is only one supervisor process at runtime in an *A-MODEL/PL* program. It manages all object creations and terminations. This process owns the process table which includes a list of all created objects. If the process accepts a broadcasting inquiry, it delivers the broadcast data to all designated objects in the order of the objects' creations.

Implementation of Exception Handling
Exception handling is implemented using the *kill* system call used by a UNIX process to send a signal to another process, and the *sigvec* system call which is used to set handler routines for signals. The **raise** statement is translated into a **kill** system call and exception routines are set by the *sigvec* system call. A process aborting facility is implemented as a *kill* system call to the runtime supervising process. The supervisor takes care of aborting the designated process and its propagation.

3.3 Problems in the First Prototype and Required Improvement

The current prototype has an inter-object communication and a late-binding mechanism for pseudo operations. However, there was no consideration given for the efficiency of produced code in the prototype. In particular, parameter and return value type checks in an operation call are performed at run time even though they can be done at compilation. In this sense, a strong typing scheme
is really not implemented in the prototype as was planned in the design.

Even if the static check were implemented, run time checking would still be needed for a late-bound pseudo operation. We employ a late-binding mechanism at the cost of runtime efficiency but with a gain in the flexibility. This disadvantage, however, can be avoided by providing a compiler option to not include run time checking. Once we know the code is correct, we do not have to pay the run time cost any more. This option may be used to compile only completed programs. It should not be used during development of programs. This option will be incorporated in the next version of the implementation.

4. Conclusion

In this paper, we gave an overview of *A-MODEL/PL* designed for the development of production quality basic software. Central to our design is to enable natural modeling. Towards this end, we interpret an object as a entity with a scenario, we distinguish active objects from passive objects, and we introduce a broadcasting mechanism as an inter-object communication aid. These features have not been previously realized in object-oriented concurrent programming languages.

Programming with objects without scenarios requires us to write a driver routine to a driver object. Adding such an object introduces a structural change when going from the problem to the solution space. Another way to control objects without scenarios is to provide an operation for the scenario for each object and to invoke this operation at instance creation. This approach is close to employing objects with built-in scenarios, but we feel that it still does not provide a natural model.

Distinguishing between clients and servers provides additional benefits to natural modeling. It helps static detection of invalid communication between objects. For example, an attempt of a server to invoke operations of another server can be detected as an error. This can be regarded as a kind of strong typing scheme.

From the standpoint of the reuse of objects (classes), servers are more reusable than clients because servers are independent of their context while clients are not. Thus, to distinguish between clients and servers allows us to extract reusable classes from a specific problem space. This becomes even more important if we move to a style of differential programming [MOO86] where reusing existing objects is a key concern.

Broadcasting is a powerful inter-object communication method describing system-wide data communication. For example, an error message propagated from a parser to all other objects of a compiler, or a network-wide message delivered to every process in a certain region of a local area network. If we try to write such software without broadcasting, we are forced to use one-to-one communication. However, it is not guaranteed that the object issuing the message knows all the other objects receiving the message. To do this, we have to introduce an additional object knowing about all of instantiated objects in the system. It is, again, an unnatural model.

A-MODEL/PL was designed as a specialized programming language for the software development environment *STEP* [HAR86, NOR85, NOR88]. A prototype *STEP* has been written in C code and fully compatible with the object codes of *A-MODEL/PL*. This has further supported the practicality of our language. We are currently planning to implement an *A-MODEL/PL* compiler written in *A-MODEL/PL*.

Based on the above discussion, *A-MODEL/PL* qualifies as yet another object-oriented concurrent programming language. But, based on experience, the authors believe it provides a more complete and natural model of the problem space.

Client

Server.Op1(x, y, z);

1. Open sockets CC to call $Op1$ and CR to get the return value from $Op1$.
2. Bind them to the names CCN and CRN respectively.
3. Connect CC to SA with SAN.

4. Write *operation-call data* to SA.
   Operation-call data
   the operation tag $Op1$, parameters x, y, z, and CRN, the name of the socket on which to receive the return value.
5. Close CC.

8.1 Listen for connection to CR.
8.2 Accept the connection to CR
   $SR' \leftarrow$ **accept**CR;
9. Read return value from SR'.
10. Close CR and SR'.

Server

accept $Op1$;

0. Open socket SA to accept all operation calls to this server and bind it to the name SAN.

3.1 Listen for connection to SA.
3.2 Accept the connection SA.
   $SA' \leftarrow$ **accept**SA;
4. Read the *operation-call data* through SA'.

5. Close SA'.
6. Execute the corresponding operation.
7. Open a socket SR for the return value and bind it to SRN.
8. Connect SR to CR with CRN.

9. Write return value to SR.
10. Close SR.

(a) Operation Call Realized with Socket

-- Let *CRet_Sock* and *CRet_SN* be
-- a socket and its name, respectively, which a client
-- set up to receive a return value.

Prepare a queue for operation-call data;
Open socket *AccConn_Sock* to accept all operation calls to the server and bind it to the name *AccConn_SN*;

loop
   -- search the operation-call queue
   Search_Queue(*list of tags for operations to execute*);
   if corresponding operation-call data are in the queue **then**
      exit;
   else
      Listen for connection to *AccConn_Sock*;
      Accept the connection to *AccConn_Sock*;
      DataGet_Sock ← **accept** *AccConn_Sock*;
      Read the operation-call data through *DataGet_Sock*;
      Close *DataGet_Sock*;
      Enqueue operation-call data that have just been received;
   end if;
end loop;

Execute the corresponding operation;
Open a socket *RetVal_Sock* in which to put return value and bind it to *RetVal_SN*;
Connect *Ret_Val_Sock* to *CRet_Sock* using *CRet_SN*;
Write return value through *Ret_Val_Sock*;
Close *Ret_Val_Sock*;

(b) Algorithm for Server to Accept Operation Call

Fig.2 Implementation of Operation Call

Acknoeledgements

We would like to thank Professor Victor R. Basili of the University of Maryland who gave us valuable comments on writing and organizing this paper.

References

[BIR79] G. M. Birtwistle *et al.*, *Simula Begin*, Lund, Sweden: Studentlitteratur, 1979.

[BOO86] G. Booch, *Software Engineering with Ada*, 2nd ed. Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1987.

[COX86] B. J. Cox, *Object Oriented Programming*, Reading, MA: Addison-Wesley Publishing Company, 1986.

[DEN84] R. J. DeNatale, "Class C -- Extensions to Object-Oriented Programming," *IBM Technical Journal*, Watson Research Center, Feb. 1984.

[DOD83] DoD, *Reference Manual for the Ada Programming Language*, American National Standard Institute, Jan. 1983.

[GOL83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Reading, MA: Addison-Wesley Publishing Company, 1983.

[HAR86] K. Harada and M. Noro, "An Interactive Software Design and Implementation Support System STEP," Institute of Information Science, Keio University, KIIS-85-01, Feb. 1986.

[HOA78] C. A. R. Hoare *et al.*, "Communicating Sequential Processes," *Commun. ACM*, Vol. 21, No. 8, Aug. 1978, pp. 323-334.

[INM84] INMOS Limited, *Occam Programming Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1984.

[ISH86] Y. Ishikawa and M. Tokoro, "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Feature and Implementation," *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986, pp. 232-241.

[LIS83] B. Liskov and R. Scheifler: "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transaction on Programming Languages and Systems*, Vol.5, No.3, July 1983, pp.381-404.

[MOO86] D. A. Moon, "Object-Oriented Programming with Flavors," *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986, pp. 1-8.

[NOR85] M. Noro and K. Harada, "A Design of the Programming Environment STEP Based on Stepwise Refinement," in *Proc. COMPSAC'85*, Oct. 1985, pp. 334-341.

[NOR88] M. Noro, *Design and Implementation of a Process-Oriented Software Development Environment*, Ph.D. Dissertation, Keio University, 1988.

[SCH86] K. Schmucker, "Object-Oriented Languages for the Macintosh," *Byte*, Aug. 1986, pp. 177-185.

[STR86] B. Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley Publishing Company, 1986.

[SUN86] Sun Microsystems Inc., "Inter-Process Communication Primer," in Networking on the Sun Workstation, Feb. 1986.

[THA85] D. Thalmann, *MODULA-2*, Berlin: Springer-Verlag,1985.

[TOU87] H. Touati, "Is Ada an Object Oriented Programming Language?," *SIGPLAN Notices*, Vol. 22, No. 5, May 1987, pp. 23-26.

[WEG87] P. Wegner, "Dimensions of Object-Oriented Language Design," *SIGPLAN Notices*, Vol. 22, No. 11, Nov. 1987, pp. 168-182.

[YOK87] Y. Yokote and M. Tokoro, "Concurrent Programming in Concurrent Smalltalk in *Object-Oriented Concurrent Programming*," A. Yonezawa and M. Tokoro *Eds.*, Cambridge: The MIT Press, 1987, pp. 129-158.

[YON86] A. Yonezawa *et al.*, "Object-Oriented Concurrent Programming in ABCL/1," *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986, pp. 258-267.