Object-Oriented Concurrent Programming in ABCL/1

Akinori Yonezawa Jean-Pierre Briot and Etsuya Shibayama

Department of Information Science
Tokyo Institute of Technology
Ookayama, Meguro-ku, Tokyo 152
(03)-726-1111 ext. 3209

Abstract

An object-oriented computation model is presented which is designed for modelling and describing a wide variety of concurrent systems. In this model, three types of message passing are incorporated. An overview of a programming language called ABCL/1, whose semantics faithfully reflects this computation model, is also presented. Using ABCL/1, a simple scheme of distributed problem solving is illustrated. Furthermore, we discuss the reply destination mechanism and its applications. A distributed "same fringe" algorithm is presented as an illustration of both the reply destination mechanism and the future type message passing which is one of the three message passing types in our computation model.

1. Introduction

Parallelism is ubiquitous in our problem domains. The behavior of computer systems, human information processing systems, corporative organizations, scientific societies, etc. is the result of highly concurrent (independent, cooperative, or contentious) activities of their components. We like to model such systems, and design AI and software systems by using various metaphors found in such systems [Smith 1985] [Special Issue 1981] [Yonezawa and Tokoro 1986] [Brodie et al. 1984]. Our approach is to represent the components of such a system as a collection of *objects* [Stefik and Bobrow 1986] and their interactions as *concurrent* message passing among such objects. The problem domains to which we apply our framework include distributed problem solving and planning in AI, modelling human cognitive processes, designing real-time systems and operating systems, and designing and constructing office information systems [Tschritzis 1985].

This paper first presents an object-based model for parallel computation and an overview of a programming language, called ABCL/1 [Yonezawa et al. 1986] [Shibayama and Yonezawa 1986a], which is based on the computation model. Then, schemes of distributed problem solving are illustrated using ABCL/1. Though our computation model has evolved from the Actor model [Hewitt 77] [Hewitt and Baker 1977], the notion of *objects* in our model is different from that of *actors*.

2. Objects

Each *object* in our computation model has its own (autonomous) processing power and it may have its local persistent memory, the contents of which represent its *state*. An object is always in one of three modes: *dormant*, *active*, or *waiting*. An object is initially dormant. It becomes active when it receives a message that satisfies one of the specified patterns and constraints. Each object has a description called *script* (or a set of methods) which specifies its behavior: what messages it accepts and what actions it performs when it receives such messages.

When an active object completes the sequence of actions that are performed in response to an accepted message, if no subsequent messages have arrived, it becomes dormant again. An object in the active mode sometimes needs to stop its current activity in order to wait for a message with specified patterns to arrive. In such a case, an active object changes into the waiting mode. An object in the waiting mode becomes active again when it receives a required message. For instance, suppose a buffer object accepts two kinds of messages: a [:get] message from a consumer object requesting the delivery of one of the stored products, and a [:put <product>] message from a producer object requesting that a product (information) be stored in the buffer. When the buffer object receives a [:get] message from a consumer object and finds that its storage, namely the buffer, is empty, it must wait for a [:put <product>] message to arrive. In such a case the buffer object in the active mode changes into the waiting mode.

An active object can perform usual symbolic and numerical computations, make decisions, send messages to objects (including itself), create new objects and update the contents of its local memory. An object with local memory cannot be activated by more than one message at the same time. Thus, the activation of such an object takes place one at a time.

As mentioned above, each dormant object has a fixed set of patterns and constraints for messages that it can accept and by which it can be activated. To define the behavior of an object, we must specify what computations or actions the object performs for each message pattern and constraint. To

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-204-7/86/0900-0258 75¢

write a definition of an object in our language ABCL/1, we use the notation in Figure 1. Figure 2 shows a skeletal definition of an object.

```
[object object-name
 (state representation-of-local-memory )
 (script
  (=> message-pattern where constraint
        ...action... )
   .
   .
   .
  (=> message-pattern where constraint
        ...action... ))]
```

Figure 1. Object Definition

```
[object Buffer
 (state ... )
 (script
  (=> [:put ... ]  ... )


  (=> [:get]  ...  ))]
```

Figure 2. Buffer

(state ...) declares the variables which represent the local persistent memory (we call such variables *state* variables) and specifies their initialization. *object-name* and the construct "where *constraint*" are optional. If a message sent to an object defined in the notation above satisfies more than one pattern-constraint pair, the first pair (from the top of the script) is chosen and the corresponding sequence of actions is performed.

An object changes into the waiting mode when it performs a special action. In ABCL/1, this action (i.e., the transition of an object from the active mode to the waiting mode) is expressed by a *select*-construct. A select construct also specifies the patterns and constraints of messages that are able to reactivate the object. We call this a *selective message receipt*.

```
(select
  (=> message-pattern where constraint      ... action ...)
   .
   .
   .
  (=> message-pattern where constraint      ... action ...))
```

Figure 3. Select Construct

As an example of the use of this construct, we give, in Figure 4, a skeleton of the definition of an object which behaves as a buffer of a bounded size.

```
[object Buffer
 (state declare-the-storage-for-buffer )
 (script
  (=> [:put aProduct]          ; aProduct is a pattern variable.
   (if the-storage-is-full
    then (select               ; then waits for a [:get] message.
             (=> [:get]
               remove-a-product-from-the-storage-and-return-it )))
   store-aProduct   )

  (=> [:get]
   (if the-storage-is-empty
     then (select              ; then waits for a [:put ...] message.
              (=> [:put aProduct]
                send-aProduct-to-the-object-which-sent-[:get]-message ))
     else remove-a-product-from-the-storage-and-return-it )) )]
```

Figure 4. An Example of the Use of Select Constructs

Suppose a [:put <product>] arrives at the object Buffer. When the storage in the object Buffer is found to be full, Buffer waits for a [:get] message to arrive. When a [:get] message arrives, Buffer accepts it and returns one of the stored products. If a [:put] message arrives in *this* waiting mode, it will not be accepted (and put into the *message queue* for Buffer, which will be explained in §3). Then, Buffer continues to wait for a [:get] message to arrive. A more precise explanation will be given in the next section.

As the notation for a select construct suggests, more than one message pattern (and constraint) can be specified, but the ABCL/1 program for the buffer example in Figure 4 contains only one message pattern for each select construct.

3. Message Passing

An object can send a message to any object as long as it knows the name of the target object. The "knows" relation is dynamic: if the name of an object T comes to be known to an object O and as long as O remembers the name of T, O can send a message to T. If an object does not know or forgets the name of a target object, it cannot at least directly send a message to the target object. Thus message passing takes place in a point-to-point (object-to-object) fashion. No message can be broadcast.

All the message transmissions in our computation model are asynchronous in the sense that an object can send a message whenever it likes, irrespective of the current state or mode of the target object. Though message passing in a system of objects may take place concurrently, we assume message arrivals at an object be linearly ordered. No two messages can arrive at the same object simultaneously. Furthermore we make the following (standard) assumption on message arrival:

[Assumption for Preservation of Transmission Ordering]

> When two messages are sent to an object T by the same object O, the temporal ordering of the two message transmissions (according to O's clock) must be preserved in the temporal ordering of the two message arrivals (according to T's clock).

This assumption was not made in the Actor model of computation. Without this, however, it is difficult to model even simple things as objects. For example, a computer terminal or displaying device is difficult to model as an object without this assumption because the order of text lines which are sent by a terminal handling program (in an operating system) must be preserved when they are received. Furthermore, descriptions of distributed algorithms would become very complicated without this assumption.

In modelling various types of interactions and information exchange which take place among physical or conceptual components that comprise parallel or real-time systems, it is often necessary to have two distinct modes of message passing: *ordinary* and *express*. Correspondingly, for each object T, we assume two message queues: one for messages sent to T in the ordinary mode and the other for messages sent in the express mode. Messages are enqueued in arrival order.

[*Ordinary* Mode Message Passing]

> Suppose a message M sent in the ordinary mode arrives at an object T when the message queue associated with T is empty. If T is in the dormant mode, M is checked as to whether or not it is acceptable according to T's script. When M is acceptable, T becomes active and starts performing the actions specified for it. When M is not acceptable, it is discarded. If T is in the active mode, M is put at the end of the *ordinary* message queue associated with T.

If T is in the waiting mode, M is checked to see if it satisfies one of the pattern-and-constraint pairs that T accepts in *this* waiting mode. When M is acceptable, T is reactivated and starts performing the specified actions. When M is not acceptable, it is put at the end of the message queue.

In general, upon the completion of the specified actions of an object, if the ordinary message queue associated with the object is empty, the object becomes dormant. If the queue is not empty, then the first message in the queue is removed and checked as to whether or not it is acceptable to the object according to its script. When it is acceptable, the object stays in the active mode and starts performing the actions specified for the message. If it is not acceptable, the message is discarded and some appropriate default action is taken (for instance, the message is simply discarded, or a default failure message is sent to the sender of the message). Then if the queue is not empty, the new first message in the queue is removed and checked. This process is repeated until the queue becomes empty. When an object changes into the waiting mode, if the ordinary message queue is not empty, then it is searched from its head and the first message that matches one of the required pattern-and-constraint pairs is removed from the queue. Then the removed message reactivates the object. If no such message is found or the queue itself is empty, the object stays in the waiting mode and keeps waiting for such a message to arrive. Note that the waiting mode does not imply "busy wait".

[*Express* Mode Message Passing]

Suppose a message M sent in the *express* mode arrives at an object T. If T has been previously activated by a message which was also sent to T in the *express* mode, M is put at the end of the *express* message queue associated with T. Otherwise, M is checked to see if it satisfies one of the pattern-and-constraint pairs that T accepts. If M is acceptable, T starts performing the actions specified for M even if T has been previously activated by a message sent to T in the *ordinary* mode. The actions specified for the previous message are suspended until the actions specified for M are completed. If so specified, the suspended actions are aborted. But, in default, they are resumed.

An object cannot accept an *ordinary* mode message as long as it stays in the active mode. Thus, without the express mode message passing, no request would be responded to by an object in the active mode. For example, consider an object which models a problem solver working hard to solve a given problem (cf. §7). If the given problem is too hard and very little progress can be made, we would have no means to stop him or make him give up. Thus without the express mode, we cannot monitor the state of an object (process) which is continuously in operation and also cannot change the course of its operation. More discussion about the express mode will be found in §5.3, §10.2, and §10.3.

As was discussed above, objects are autonomous information processing agents and interact with other objects only through message passing. In modelling interactions among such autonomous objects, the convention of message passing should incorporate a *natural* model of synchronization among interacting objects. In our computation model, we distinguish three types of message passing: *past*, *now*, and *future*. In what follows, we discuss each of them in turn. The following discussions are valid, irrespective of whether messages are sent in the ordinary or express mode.

[*Past* Type Message Passing] (send and no wait)

Suppose an object O has been activated and it sends a message M to an object T. Then O does not wait for M to be received by T. It just continues its computation after the transmission of M (if the transmission of M is not the last action of the current activity of O).

We call this type of message passing *past* type because sending a message finishes before it causes the intented effects to the message receiving object. Let us denote a past type message passing in the ordinary and the express modes by:

[T <= M]   and   [T <<= M],

respectively. The past type corresponds to a situation where one requests or commands someone to do some task and simultaneously he proceeds his own task without waiting for the requested task to be completed. This type of message passing substantially increases the concurrency of activities within a system.

[*Now* Type Message Passing] (send and wait)

When an object O sends a message M to an object T, O waits for not only M to be received by T, but also waits for T to send some information back to O.

This is similar to ordinary function/procedure calls, but it differs in that T's activation does not have to end with sending some information back to O. T may continue its computation after sending back some information to O. A now type message passing in the ordinary and express modes are denoted by:

[T <== M]   and   [T <<== M],

respectively. Returning information from T to O may serve as an acknowledgement of receiving the message (or request) as well as reporting the result of a requested task. Thus the message sending object O is able to know for certain that his message was received by the object T though he may waste time waiting. The returned information (certain values or signals) is denoted by the same notation as that of a now type message passing. That is, the above notation denotes not merely an action of sending M to T by a now type message passing, but also denotes the information returned by T. This convention is useful in expressing the assignment of the returned value to a variable. For example, [x := [T <== M]].

Now type message passing provides a convenient means to synchronize concurrent activities performed by independent objects when it is used together with the parallel construct. This construct will not be discussed in this paper. It should be noted that recursive *now* type message passing causes a local deadlock.

[*Future* Type Message Passing] (reply to me later)

Suppose an object O sends a message M to an object T expecting a certain requested result to be returned from T. But O does not need the result immediately. In this situation, after the transmission of M, O does not have to wait for T to return the result. It continues its computation immediately. Later on when O needs that result, it checks its special *private* object called *future object* that was specified at the time of the transmission of M. If the result has been stored in the future object, it can be used.

Of course, O can check whether or not the result is available before the result is actually used. A future type message passing in the ordinary and express modes are denoted by:

[T <= M $ x]   and   [T <<= M $ x],

respectively, where x stands for a special variable called *future variable* which binds a future object. We assume that a future object behaves like a queue. The contents of the queue can be checked or removed *solely* by the object O which performed the future type message passing. Using a special expression "(ready? x)", O can check to see if the queue is empty. O could access to the first element of the queue with a special expression "(next-value x)", or to all the elements with "(all-values x)". If the queue is empty in such cases, O has to wait. (Its precise behavior will be given in §6.2.).

A system's concurrency is increased by the use of future type message passing. If the now type is used instead of the future type, O has to waste time waiting for the currently unnecessary result to be produced. Message passing of a somewhat similar vein has been adopted in previous object-oriented programming languages. Act1, an actor-based language developed by H. Lieberman [1981] has a language feature called "future," but it is different from ours. The three types of message passing are illustrated in Figure 5.

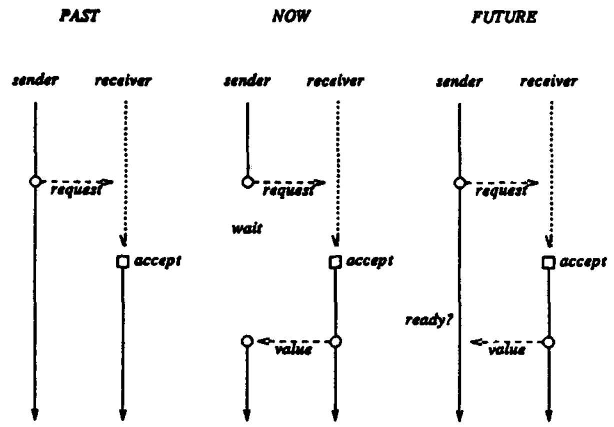

Figure 5. The Three Message Passing Types

Though our computation model for object-oriented concurrent programming is a descendant of the Actor computation model which has been proposed and studied by C. Hewitt and his group at MIT [Hewitt 1977] [Hewitt and Baker 1977] [Yonezawa and Hewitt 1979] [Lieberman 1981], it differs from the Actor computation model in many respects. For example, in our computation model, an object in the waiting mode can accept a message which is not at the head of the message queue, whereas, in the actor computation model, a (serialized) actor can only accept a message that is placed at the head of the message queue. Furthermore, now type and future type message passing are not allowed in the Actor computation model. Therefore, an actor A which sends a message to a target actor T and expects a response from T must terminate its current activity and receive the response as just one of any incoming messages. To discriminate T's response from other incoming messages arriving at A, some provision must be made before the message is sent to T. Also the necessity of the termination of A's current activity to receive T's response causes unnatural breaking down of A's task into small pieces.

4. Messages

We will consider what information a message may contain. A message is composed of a singleton or a sequence of *tags*, *parameters*, and/or *names of objects*. Tags are used to distinguish message patterns. (In the buffer example mentioned in Figure 4, :get and :put are tags, and "aProduct" denotes a parameter in the [:put ...] message.) Object names contained in a message can be used for various purposes. For example, when an object O sends a message M to an object T requesting T to do some task, and O wishes T to send the result of the requested task to a specified object C1, O can include the name of C1 in the message M. Objects used in this way correspond to "continuation" (or customer) in the Actor computation model. Also, when O requests T to do some task in cooperation with a specified object C2, O must let T know the name of C2 by including it in the message M.

Besides the information contained in a message itself, we assume two other kinds of information can be transmitted in message passing. One is the *sender name* and the other is the *reply destination*. When a message sent from an object O is received by an object T, it is assumed that the name of the sender object O becomes known to the receiver object T. (We denote the sender name by "&sender" in ABCL/1.) This assumption considerably strengthens the expressive power of the model and it is easy to realize in the implementation of our computation model. A receiver object can decide whether it accepts or rejects an incoming message on the basis of who (or what object) sent the message.

When an object T receives a message sent in a now or future type message passing, T is required to reply to the message or return the result of the requested task (or just an acknowledgement). Since the destination to which the result should be returned is known at the time of the message transmission, we assume that such information about the destination is available to the receiver object T (and this information can be passed around among objects). We call such information the *reply destination*. To specify the object to which the result should be returned, the *reply destination* mechanism provides a more uniform way than simply including the name of the object in the request message. This mechanism is compatible with the three types of message passing, and enables us to use both explicit reply destinations in case of past type message as well as implicit ones in case of now or future type messages (cf. §6 and §9). Furthermore, the availability of the reply destination allows us to specify continuations and implement various *delegation* mechanisms [Lieberman 1986] uniformly. This will be discussed in the §8.

The fact that sender names and reply destinations can be known to message receiving objects not only makes the computation model powerful, but also makes it possible that the three different types of message passing: *past*, *now*, and *future*, be reduced to just one type of message passing, namely the *past* type message passing. In fact, a now type message passing in an object T can be expressed in terms of past type message passing together with the transition into the waiting mode in the execution of the script of the object T. And a future type message passing can be expressed in terms of past and now type message passing, which are in turn reduced to past type message passing. These reductions can be actually demonstrated, but to do so, we need a formal language. Since the programming language ABCL/1 to be introduced in the subsequent sections can also serve this purpose, we will give an actual demonstration after the explanation of ABCL/1 (cf. §6). The reply destination mechanism plays an important role in the demonstration.

5. An Overview of the Language ABCL/1

5.1. Design Principles

The primary design principles of our language, ABCL/1, are:

[1] [Clear Semantics of Message Passing] The semantics of message passing among objects should be transparent and faithful to the underlying computation model.

[2] [Practicality] Intentionally, we do not pursue the approach in which every single concept in computation should be represented purely in terms of objects and message passing. In describing the object's behavior, basic values, data structures (such as numbers, strings, lists), and invocations of operations manipulating them may be assumed to exist as they are, not necessarily as objects or message passing. Control structures (such as *if-then-else* and looping) used in the description of the behavior of an object are not necessarily based upon message passing (though they can of course be interpreted in terms of message passing).

Thus in ABCL/1, *inter*-object message passing is entirely based on the underlying object-oriented computation model, but the representation of the behavior (script) of an object may contain conventional *applicative* and *imperative* features, which we believe makes ABCL/1 programs easier to read and write from the viewpoint of *conventional* programmers. Since we are trying to grasp and exploit a complicated phenomenon, namely parallelism, a rather conservative approach is taken in describing the internal behavior of individual objects. Various applicative and imperative features in the current version of ABCL/1 are expressed in terms of Lisp-like parenthesized prefix notations, but that is not essential at all; such features may be written in other notations employed in various languages such as C or Fortran.

5.2. Creating Objects and Returning Messages

In our computation model, objects can be dynamically created. Usually, when an object A needs a new object B, A sends, in a now or future type message passing, some initial information to a certain object which *creates* B. Then B is returned as the value (or result) of the now/future type message passing. This way of creating an object is often described in ABCL/1 as follows:

```
[object CreateSomething
 (script
  (=> pattern-for-initial-info   ![object ... ] ) )]
```

where [object ....] is the definition of an object newly created by the object CreateSomething. The CreateAlarmClock object defined in Figure 6 creates and returns an alarm clock object when it receives a [:new ...] message containing the person (object) to wake. The time to ring is set by sending a [:wake-me-at ...] message to the alarm clock object. It is supposed to keep receiving [:tick ...] messages from a clock object (called the Ticker and which will be defined in the next subsection). When the time contained in a [:tick ...] message is equal to the time to ring, the alarm clock object sends a [:time-is-up] message to the person to wake in the express mode.

```
[object CreateAlarmClock
 (script
  (=> [:new Person-to-wake]

   ![object
     (state [time-to-ring := nil])
     (script
      (=> [:tick Time]
       (if (= Time time-to-ring)
         then [Person-to-wake <<= [:time-is-up]]))

      (=> [:wake-me-at T]
       [time-to-ring := T]) )] ) )]
```

Figure 6. Definition of CreateAlarmClock Object

Note that the "Person-to-wake" variable in the script of the alarm clock object to be created is a free variable (it is not a state variable nor a message parameter). It will be "closured" when creating this object, which implies that the scope rule of ABCL/1 is lexical. The notation using ! is often used in ABCL/1 to express an event of returning or sending back a value in response to a request which is sent in a now or future type message passing. In the following fragment of a script:

(=> *pattern-for-request*   ... !*expression* ...   ),

where is the value of *expression* returned? In fact, this notation is an abbreviated form of a more explicit description which uses the reply destination. An equivalent and more explicit form is:

(=> *pattern-for-request* @ *destination*   ... [*destination* <= *expression*] ...)

where *destination* is a pattern variable which is bound to the reply destination for a message that matches *pattern-for-request*. When a message is sent in a past type message passing, if we need to specify the reply destination, it can be expressed as:

[T <= *request* @ *reply-destination*].

Note that *reply-destination* denotes an object. In the case of now or future type message passing, pattern variables for reply destination are matched with certain objects that the semantics of now/future type message passing defines. (See §6.) Thus the programmer is not allowed to explicitly specify reply destinations in now or future type message passing. So the following expressions [*target* <== *message* @ *reply-destination*], and [*target* <= *message* @ *reply-destination* $ x] are illegal.

There is another way to create an object. That is, an object can be obtained by copying some object. We can use the copy instantiation model [Briot 1984] after defining a prototype [Lieberman 1986], rather than defining a generator object (analog to a class). Each object can invoke a primitive function "self-copy" whose returning value is a copy of the object itself (Me), which will be exemplified in §9.

5.3. Ordinary Mode and Express Mode in Message Passing

The difference between the ordinary mode and express mode in message passing was explained in §3. The notational distinction between the two modes in message transmission is made by the number of "<", one for the ordinary mode and two for the express mode (namely <= and <==, vs. <<= and <<==). The same distinction should be made in message reception because a message sent in the ordinary mode should not be interpreted as one sent in the express mode. To make the distinction explicit, we use the following notation for expressing the reception of a message sent in the express mode.

(=>> *message-pattern* where *constraint* ... *action* ...),

The reception of a message sent in the *ordinary* mode is expressed by the following notation as explained above:

(=> *message-pattern* where *constraint* ... *action* ...)

This notational distinction protects an object from unwanted express mode messages because the object accepts only messages that satisfy the patterns and constraints declared after the notation "(=>>". Express mode messages which do not satisfy such patterns and constraints are simply discarded.

Suppose a message sent in the express mode arrives at an object which has been currently activated by an ordinary mode message. If the script of the object contains the pattern and constraint that the message satisfies, the current actions are temporarily terminated (or suspended) and the actions requested by the express mode message are performed. If the object is accessing its local persistent memory when the express mode message arrives, the current actions will not be terminated until the current access to its local memory is completed. Also, if the object is performing the actions whose script is enclosed by "(atomic" and ")" in the following manner:

(atomic ... *action* ...),

they will not be terminated (or suspended) until they are completed. And if the actions specified by the express mode message are completed and no express mode messages have arrived yet at that time, the temporarily terminated actions are resumed by default. But, if the actions specified by the express mode message contains the "non-resume" command, denoted by:

(non-resume),

the temporarily terminated actions are aborted and will not be performed any more.

Note that, in the above explanation, the actions temporarily terminated by an express mode message are the ones that are activated (specified) by an ordinary mode message. When an object is currently performing the actions specified by an express mode message, no message (even in the express mode) can terminate (or suspend) the current actions.

To illustrate the use of express mode, we give the definition of the behavior of a clock object Ticker which sends [:tick ...] messages to all the alarm clocks he knows about (the value of its state variable "alarm-clocks-list"). The definition of the Ticker object is given in Figure 7. The two state variables of Ticker, "time" and "alarm-clocks-list", respectively contain the current time and a list of alarm clocks to be "ticked". When Ticker receives a [:start] message, it starts ticking and updating the contents of "time".

[alarm-clocks-list <= [:tick...]]

means sending [:tick ...] messages to each member of "alarm-clocks-list" simultaneously. We call this way of sending messages *multicast*. When Ticker receives a [:stop] message sent in the express mode, it stops ticking by the effect of (non-resume). This message must be sent in the express mode because Ticker always stays in the active mode to keep ticking (in the while loop). An [:add ...] message appends new alarm clock object to the "alarm-clocks-list" in Ticker. This message also should be sent in the express mode for the same reason.

```
[object Ticker
  (state [time := 0] [alarm-clocks-list := nil])
  (script
   (=> [:start]
    (while t do
      (if alarm-clocks-list
         then [alarm-clocks-list <= [:tick time]])
      [time := (1+ time)]))

   (=>> [:add AlarmClock]
    [alarm-clocks-list := (cons AlarmClock alarm-clocks-list)])

   (=>> [:stop] (non-resume)) )]
```

Figure 7. Definition of Ticker Object

The definition of the CreateAlarmObject (which appeared in Figure 6) should be slightly changed in order for a newly created alarm clock object to be known by Ticker. The description of an alarm clock object is the same as in Figure 6, but when created it will now be bound to a temporary variable "AlarmClock". Then, after the created object is sent to Ticker to be appended to Ticker's "alarm-clocks-list", it is returned to the sender of the [:new ...] message as in the case of Figure 6.

```
[object CreateAlarmClock
  (script
   (=> [:new Person-to-wake]
    (temporary
     [AlarmClock := [object   description of an alarm clock object   ]])

    [Ticker <<= [:add AlarmClock]]
    !AlarmClock) )]
```

Figure 8. New Definition of CreateAlarmClock Object

6. A Minimal Computation Model

Below we will demonstrate that

[1] A now type message passing can be reduced to a combination of past type message passing and a selective message reception in the waiting mode, and

[2] A future type message passing can also be reduced to a combination of past type message passing and now type message passing.

Thus both kinds of message passing can be expressed in terms of past type message passing and selective message reception in the waiting mode, which means that now type message passing and future type message passing are derived concepts in our computation model. (The rest of this section could be skipped if one is not interested in the precise semantics of "now" and "future" types message passing.)

6.1. Reducing Now Type

Suppose the script of an object A contains a now type message passing in which a message M is sent to an object T. Let the object T accept the message M and return the response (i.e., send the response to the reply destination for M). This situation is described by the following definitions for A and T written in ABCL/1.

```
[object A
  ...
 (script
  ...
  (=> message-pattern          ... [T <== M] ...         ) ... )]

[object T
  ...
 (script
  ...
  (=> pattern-for-M @ R    ... [R <= expression ] ...    ) ... )]
```

** Note that the script of T can be abbreviated as:
(=> *pattern-for*-M ... !*expression* ...)

We introduce a new object "New-object" which just passes any received message to A, and also introduce a *select*-construct which receives only a message that is sent from "New-object". The behavior of the object A can be redefined without using now type message passing as follows:

```
[object A
 (script
  ...
  (=> message-pattern
   (temporary  [New-object := [object (script (=> any  [A <= any]))] ])
   ...
   [T <= M @ New-object]
   (select
    (=> value where (= &sender New-object)
        ... value ... ))     ... ) ... )]
```

Note that the message M is sent by a past type message passing with the reply destination being the newly created "New-object." Immediately after this message transmission, the object A changes into the waiting mode and waits for a message that is passed by the "New-object". The constraint
"where (= &sender New-object)"
in the select-construct means that the messages sent by New-Object can only be accepted. "New-object" serves as a unique identifier for the message transmission from A to T in past type: [T <= M @ New-object].

6.2. Reducing Future Type

Suppose the script of an object A contains a future type message passing as follows:

```
[object A
 (state ... )
 (future ... x ... )          ; declaration of a future variable x.
 (script
  ...
  (=> message-pattern
   ... [T <= M $ x] ...
   ... (ready? x) ... (next-value x) ... (all-values x) ... ) ... )]
```

Then we consider the future variable x in A to be a state variable binding a special object created by an object CreateFutureObject. (In general, such a object, namely a future object, is created for each future variable if more than one future variable is declared.) Also we rewrite the accesses to x by now type message passing to x as follows:

```
[object A
 (state ... [x := [CreateFutureObject <== [:new Me]]] ... )
 (script
  ...
  (=> message-pattern
   ... [T <= M @ x] ... [x <== [:ready?]] ...
   ... [x <== [:next-value]] ... [x <== [:all-values]] ... ) ... )]
```

Note that the future type message passing [T <= M $ x] is replaced by a past type message passing [T <= M @ x] with the reply destination being x. Thus, the future type message passing is eliminated. The behavior of the future object is defined in Figure 9. As mentioned before, it is essentially a queue object, but it only accepts message satisfying special pattern-and-constraint pairs. A queue object created by CreateQ accepts four kinds of messages: [:empty?], [:enqueue...], [:dequeue], and [:all-elements].

```
[object CreateFutureObject
 (script
  (=> [:new Creator]

   ![object
     (state [box := [CreateQ <== [:new]]])
     (script
      (=> [:ready?] where (= &sender Creator)       ; if [:ready?] is sent
       !(not [box <== [:empty?]]))  ; by the Creator,
                         ; and if the box is non-empty, t is returned.

      (=> [:next-value] @ R where (= &sender Creator)
       (if [box <== [:empty?]]
        then (select  ; waits for a message to come, not sent by the
              (=> message  where (not (= &sender Creator))  ; Creator.
                [R <= message]))             ; it is returned
                         ; to the reply destination for a [:next-value] message.
        else  ![box <= [:dequeue]]))
      ; removes the first element in the queue and returns it.

      (=> [:all-values] @ R  where (= &sender Creator)
       (if [box <== [:empty?]]
        then (select  ; waits for a message to come, not sent by the
              (=> message  where (not (= &sender Creator))  ; Creator.
                [R <= [message]]))           ; sends a singleton list.
        else  ![box <== [:all-elements]]))
      ; removes all the elements in the queue and returns the list of them.

      (=> returned-value
       [box <= [:enqueue returned-value]]) )] ) )]
```

Figure 9. Definition of Future Object

Note the fact that the contents of the queue object stored in "box" can be checked or removed *solely* by the object which is bound to the pattern variable "Creator". Furthermore, if the queue is empty, the object which sends messages [:next-value] or [:all-values] has to wait for some value to arrive.

7. Project Team: A Scheme of Distributed Problem Solving

In this section, we present a simple scheme of distributed problem solving described in ABCL/1. In doing so, we would like to show the adequacy of ABCL/1 as a modelling and programming language in the concurrent object-oriented paradigm.

Suppose a manager is requested to create a project team to solve a certain problem by a certain deadline. He first creates a project team comprised of the project leader and multiple problem solvers, each having a different problem solving strategy. The project leader dispatches the same problem to each problem solver. For the sake of simplicity, the problem solvers are assumed to work independently in parallel. When a problem solver has solved the problem, it sends the solution to the project leader immediately. We assume the project leader also tries to solve the problem himself by his own strategy. When either the project leader or some problem solvers, or both, have solved the problem, the project leader selects the best solution and sends the success report to the manager. Then he sends a *stop* message to all the problem solvers. If nobody has solved the problem by the deadline, the project leader asks the manager to extend the deadline. If no solution has been found by the extended deadline, the project leader sends the failure report to the manager and commits suicide. This problem solving scheme is easily modeled and described in ABCL/1 without any structural distortions. (See Figure 10.)

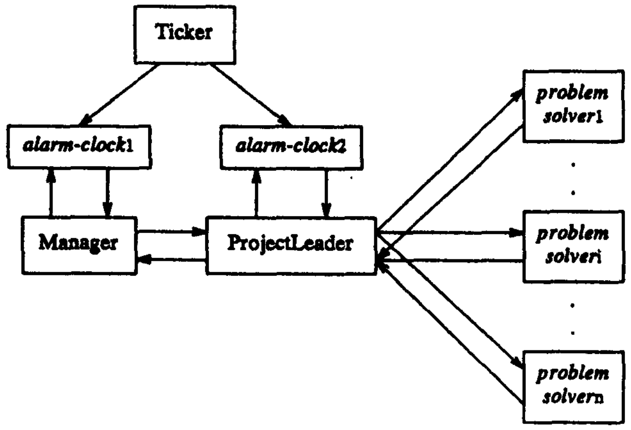

Figure 10. A Scheme for Distributed Problem Solving

The definition of the project leader object is given in Figure 11. Initially it creates an alarm clock object which will wake the project leader, and keeps it in a state variable "time-keeper". "Me" is a reserved symbol in ABCL/1 which denotes the innermost object whose definition contains the occurrence of "Me". We assume that the Ticker defined in Figure 7 is now ticking. When the project leader object receives a [:solve...] message from the manager object, it requests its alarm clock (time-keeper) to wake itself at certain time. Then, the project leader object *multicasts* to the project team members a message that contains the problem description. Note that dispatching the problem to each problem solver is expressed as a *multicast* of the problem specifications and also the message passing is of *future* type. If a problem solver finds a solution, it sends the solution to the future object bound tò "Solutions" of the project leader object. While the project leader engages himself in the problem solving, he periodically checks the variable by executing "(ready? Solutions)" as to if it may contain solutions obtained by problem solvers. Note that there is a fair chance that more than one problem solver sends their solutions to the future object bound to "Solutions". As defined in the previous section, solutions sent by problem solvers are put in the queue representing the future object in the order of arrival. "(all-values Solutions)" evaluates to the list of all the elements in the queue. Note that the sequence of actions from selecting the best solutions to terminating the team members' tasks is enclosed by "(atomic" and ")" in Figure 11. Thus, the sequence of actions is not terminated (or suspended) by an express mode message.

```
[object ProjectLeader
 (state [team-members := nil] [bestSolution := nil]
   [time-keeper := [CreateAlarmClock <== [:new Me]]])
 (future Solutions)
 (script
  (=> [:add-a-team-member M]
    [team-members := (cons M team-members)])

  (=> [:solve SPEC :by TIME]
    (temporary [mySolution := nil]) ; temporary variable

    [time-keeper <= [:wake-me-at (- TIME 20)]]
    [team-members <= [:solve SPEC] $ Solutions]
                                  ; multicast in future type
    (while (and (not (ready? Solutions)) (null mySolution))
      do ... try to solve the problem by his own
             strategy and store his solution in mySolution ...)
    (atomic
     [bestSolution := (choose-best mySolution (all-values Solutions))]
     [Manager <<= [:found bestSolution]]
     [team-members <<= [:stop-your-task]]))

  (=>> [:time-is-up] where (= &sender time-keeper)
    (temporary new-deadline)

    (if (null bestSolution)
      then
       [new-deadline := [Manager <<== [:can-extend-deadline?]]]
       (if (null new-deadline)
         then [team-members <<= [:stop-your-task]] (suicide)
         else [time-keeper <= [:wake-me-at new-deadline]])))

  (=>> [:you-are-too-late] where (= &sender Manager)
    (if (null bestSolution)
     then [team-members <<= [:stop-your-task]] (suicide))) )]
```

Figure 11. Definition of ProjectLeader Object

If no solution is found within the time limit the project leader himself has set, a [:time-is-up] message is sent by his time keeper (an alarm clock object) in the *express* mode. Then, the project leader asks the manager about the possibility of extending the deadline. If the manager answers "no" (i.e., answers "nil"), it sends a message to stop all the problem solvers and commits suicide.

Though the definition of the manager object (denoted by "Manager" in Figure 11) and problems solvers are easily written in ABCL/1, we omit them here.

8. Delegation

The *reply destination* mechanism explained in §4 and used in §6 is the basic tool to provide various delegation strategies [Lieberman 1986]. The explicit use of pattern variables for reply destinations enables us to write the script of an object which delegates the responsibility of returning a requested result to another object.

Below we define an object A, and an object B which will delegate all unknown messages to A. The pattern variable "any" will match any message not matched by the other patterns in the script of B (this is analog to the last clause with predicate t in a Lisp cond construct). The variable R will match the reply destination. So any kind of message, namely past type with or without reply destination, or now type, or future type message, will be matched and fully delegated to the object A, which could in turn, also delegate it to another object.

```
[object A                       [object B
  (state ... )                    (state ... )
  (script                         (script
    (=> patternA1                   (=> patternB1
       ... )                           ... )
     ...                              ...
    (=> patternAn                   (=> patternBp
       ... ))]                         ... )
                                    (=> any @ R   [A <= any @ R]) )]
```

This is illustrated by Figure 12, showing an answer is delivered directly to the asker without coming back through B.

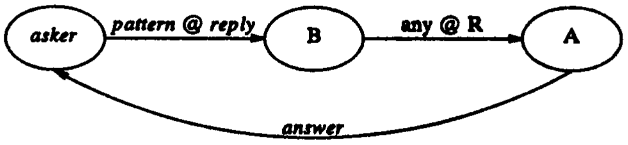

Figure 12. Illustration of Basic Delegation

9. A Distributed Algorithm for the Same Fringe Problem

The same fringe problem is to compare the fringes of two trees (Lisp lists). We will present a solution of the same fringe problem in ABCL/1, which will permit us to illustrate the use of both *future* type messages and *reply destinations*.

Our approach to the problem is similar to the one proposed by B. Serpette in [Serpette 1984]. Basically, there are three objects in this model:

- two tree extractors, extracting recursively the fringe of each tree,
- one comparator, comparing the successive elements of the two fringes.

These three objects will work in parallel. (See Figure 13.)

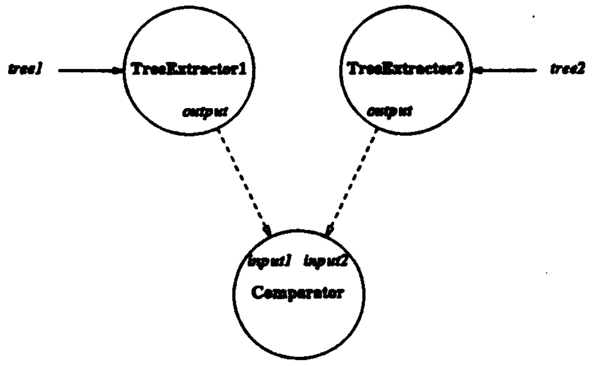

Figure 13. The Same Fringe: Tree Extractors and Comparator

The two tree extractors are linked to the comparator through two dashed arrows. Each one represents the data-flow of the successive elements of the fringe extracted by each tree extractor.

The Comparator object, defined in Figure 14, owns two state variables: "Extractor1" and "Extractor2" binding the two tree extractors, and two *future* variables "input1" and "input2" which are used for receiving the fringes from these two extractors. "Extractor1" will be bound to the object TreeExtractor defined in Figure 15, the second extractor ("Extractor2") will be created by requesting TreeExtractor to copy itself. When the Comparator object receives the [tree1 :and tree2] message, it will send a future type message [:fringe tree] to each TreeExtractor in order to request it to compute the fringe of each of the trees. Comparator assumes that Extractor1 and Extractor2 will reply the successive elements of the fringes, which will be enqueued in the future objects bound to input1 and input2, respectively.

```
[object Comparator
  (state
   [Extractor1 := TreeExtractor]
   [Extractor2 := [TreeExtractor <== [:copy]]])
  (future  input1 input2)
  (script
   (=> [tree1 :and tree2]
     [Extractor1 <= [:fringe tree1] $ input1] ; future type message
     [Extractor2 <= [:fringe tree2] $ input2] ; future type message
     [Me <= [:eq (next-value input1) :with (next-value input2)]])

   (=> [:eq atom1 :with atom2]
     (if (eq atom1 atom2)
        then  (if (eq atom1 'EOT)
                 then  (print "same fringe")
                 else  [Me <= [:eq (next-value input1)
                                  :with (next-value input2)]])
        else (print "fringes differ"))) )]
```

Figure 14. The Same Fringe Comparator

When two values from the two extractors become available to Comparator through input1 and input2, Comparator sends an [:eq (next-value input1) :with (next-value input2)] message to itself. Note that if one of the two queues (i.e., the future objects bound to variables input1 and input2) is empty, Comparator has to wait until both queues become non-empty. (See the definition of a future object in §6.2.) If the two elements are equal, Comparator will compare next elements unless they were equal to the special atom EOT (as End Of Tree), which indicates the end of the extraction. If both are EOT, the two fringes are declared to be the same. On the other hand, if the two elements differ, Comparator will declare the two fringes to be different.

We could have defined a CreateTreeExtractor object, as generator of the tree extractors, but (to show a different way of creating objects) we will rather define the prototype object TreeExtractor, and later copy it to create the second tree extractor we need. The TreeExtractor object, defined in Figure 15, owns a single state variable "output" to remember the reply destination to which it has to send the successive elements of the fringe during the extraction.

The script [:copy] will return a copy of itself. This will be a *pure* (exact copy of the original object) copy of TreeExtractor. The [:fringe tree] script will bind the reply destination to the variable "Pipe". This reply destination is a future object which was bound to the future variable "input1" or "input2" of Comparator. It will be assigned to the state variable "output", thus connecting† its "output" with one "input" of the Comparator (like in the Figure 13). Then it will send to itself the message [:extract tree] with itself being the reply destination.

† like the communication pipes in the ObjPive model [Serpette 1984], inspired by the Un*x pipes. In contrast, these "pipes" are virtual (no assumption of shared memory).

```
[object TreeExtractor
  (state  output)
  (script
    (=> [:copy]   !(self-copy))

    (=> [:fringe tree] @ Pipe
      [output := Pipe]
      [Me <= [:extract tree] @ Me])

    (=> [:extract tree] @ C
     (cond
       ((null tree)   [C <= [:continue]])
       ((atom tree)   [output <= tree] [C <= [:continue]])
       (t   [Me <= [:extract (car tree)]
               @ [object
                    (state [Extractor := Me])
                    (script
                      (=> [:continue]
                        [Extractor <= [:extract (cdr tree)] @ C]))] ]) ))

    (=> [:continue]  [output <= 'EOT]) )]
```

Figure 15. The Same Fringe TreeExtractor

To extract the fringe of a tree, the continuation-based programming style is adopted, which is in contrast to iterative or recursive ones. This model was initiated by Carl Hewitt [Hewitt et al. 1974], who gave a solution of the same fringe problem using continuations in a coroutine style. In contrast, our algorithm is fully parallel. The "[:extract tree] @ C" message script will bind the variable C to the reply destination, which represents the continuation, i.e., the object which will do the following:

- If the tree is null, the tree extractor just activates the continuation C, by sending it the message [:continue].
- If the tree is atomic, then this element is sent to the output, (so the corresponding "input1/2" of Comparator will receive a new element) and the continuation will be activated.
- The last case means that the tree is a node (a Lisp cons). We have to extract its left son (car), and then its right son (cdr). This second part to be performed later is specified in a dynamically created object (a new continuation), which will request the tree extractor to extract the cdr of the tree, when receiving the [:continue] message. The bindings of variables "tree" and "C" are memorized in the new continuation because of the lexical scoping of ABCL/1.

When the tree extractor receives the [:continue] message, that means the end of the extraction. So it will send EOT to the output, and stop there.

Note that in this algorithm if the two fringes are found to be different, the two extraction processes go on. Comparator could then send a *stop* message to either "freeze" or kill them. To deal with such a situation, we could devise various strategies which are related to the issues of objects' "capability" and garbage collection. This will be a subject for further study.

10. Concluding Remarks

10.1. Importance of the Waiting Mode

The computation model presented in this paper has evolved from the Actor computation model. One of the important differences is the introduction of the waiting mode in our computation model. As noted at the end of §3, without now type (and/or future type) message passing, module decomposition in terms of a collection of objects tends to become unnatural. Thus the now type message passing is essential in structuring solution programs. In our computation model, the now type message passing is derived from the waiting mode and the past type message passing in a simple manner as demonstrated in §6.1. In contrast, the realization of a now type message passing in the Actor computation model forces the unnatural decomposition of actors and requires rather cumbersome procedures for identifying a message that corresponds to the return (reply) value of now type message passing.

10.2. Express Mode Message Passing

We admit that the introduction of the express mode message passing in a high-level programming language is rather unusual. The main reason of introducing the express mode is to provide a language facility for *natural* modelling. Without this mode, the script of an object whose activity needs to be interrupted would become very complicated. When an object is continuously working or active, if no express mode message passing is allowed, there is no way of interrupting the object's activity or monitoring its state. One can only hope that the object terminates or suspends its activity itself and gives an interrupting message a chance to be accepted by the object. But this would make the structure of the script of the object unnatural and complicated. It should also be noted that the express mode message passing is useful for debugging because it can monitor the states of active objects.

10.3. Interrupt vs. Non-Interrupt

Our notion of *express* mode message passing is based on a very simple *interrupt* scheme. Even in this simple scheme, we must sometime protect the activity of an object from unwanted interruptions by using the "(atomic ...)" construct. (See the script of ProjectLeader in Figure 11.) Appropriate uses of this construct sometimes requires skills.

An alternative scheme might be what we call the *mail priority* model. In this model, objects are not interrupted during their activities. An express mode message sent to an object arrives at the express queue without interrupting the object. When the object is ready to check its message queues, it always first consult its express queue (with first priority), and consult its ordinary queue only when there is no (more) message in the express queue. Now there is no fear of *bad* interruptions that the programmer has to take care of. But, on the other hand, as noted in the previous subsection, the activity of an object cannot be stopped or monitored when it is in progress. To alleviate this situation, we can introduce a built-in primitive, say "(check-express)", with which an object can check to see whether an express mode message has arrived while the object is carrying out its actions. "(check-express)" can be placed in the script of an object and it is invoked as one of the actions performed by the object. When it is invoked, if a message is in the express queue and it satisfies one of the pattern-and-constraint pairs in the script, the execution of the actions specified for the message pattern intervenes.

Since both schemes have various advantages and disadvantages and they depend on the application areas of our language, we need more experiments to draw a firm conclusion.

10.4. Parallelism and Synchronization

Let us review the basic types of parallelism provided in ABCL/1:

[1] Concurrent activations of independent objects.

[2] Parallelism caused by past type and future type message passing.

[3] Parallelism caused by the *parallel* constructs [Yonezawa et al. 1986] (we did not explain in this paper) and *multicasting* (cf. §5.3 and §7).

Furthermore, ABCL/1 provides the following four basic mechanisms for synchronization:

[1] Object: the activation of an object takes place one at a time and a single first-come-first-served message queue for ordinary messages is associated with each object.

[2] Now type message passing: a message passing of the now type does not end until the result is returned.

[3] Select construct: when an object executes a select construct, it changes into the waiting mode and waits only for messages satisfying specified pattern-and-constraint pairs.

[4] Parallel construct: see [Yonezawa et al. 1986].

10.5. Relation to Other Work

Our present work is related to a number of previous research activities. To distinguish our work from them, we will give a brief summary of ABCL/1. Unlike CSP [Hoare 1978] or other languages, ABCL/1 has characteristics of *dynamic* nature: objects can be created dynamically, message transmission is asynchronous, and the "knows"-relation among objects (i.e., network topology) changes dynamically. An object in our computation model cannot be activated by more than one message at the same time. This "one-at-a-time" nature is similar to that of Monitors [Hoare 1974], but the basic mode of communication in programming with monitors is the call/return bilateral communication, whereas it is unilateral in ABCL/1.

10.6. Other Program Examples

A wide variety of example programs have been written in ABCL/1 and we are convinced that the essential part of ABCL/1 is robust enough to be used in the intended areas. The examples we have written include parallel discrete simulation [Yonezawa et al. 1984] [Shibayama and Yonezawa 1986], inventory control systems [Kerridge and Simpson 1984] [Shibayama et al. 1985] à la Jackson's example [Jackson 1983], robot arm control, mill speed control [Yonezawa and Matsumoto 1985], concurrent access to 2-3 trees and distributed quick sort [Shibayama and Yonezawa 1986].

Acknowledgements

We would like to thank Y. Honda and T. Takada for their implementation efforts on Vax/11s, Sun workstations, and a Symbolics.

References

[Briot 1984] Briot, J-P., *Instanciation et Héritage dans les Langages Objets*, (thèse de 3ème cycle), LITP Research Report, No 85-21, LITP - Université Paris-VI, Paris, 15 December 1984.

[Brodie et al. 1984] Brodie, M., J. Mylopoulos, J. Schmidt (Eds.), On Conceptual Modelling, Springer, 1984.

[Hewitt et al. 1974] Hewitt, C., et al., *Behavioral Semantics of Nonrecursive Control Structures*, Proc. Colloque sur la Programmation, Paris, April, 1974.

[Hewitt 1977] Hewitt, C., *Viewing Control Structures as Patterns of Passing Messages*, Journal of Artificial Intelligence, Vol. 8, No. 3 (1977), pp.323-364.

[Hewitt and Baker 1977] Hewitt, C., H. Baker, *Laws for Parallel Communicating Processes*, Proc. IFIP-77, Toronto, 1977.

[Hoare 1974] Hoare, C.A.R., *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, No. 10 (1974), pp.549-558.

[Hoare 1978] Hoare, C.A.R., *Communicating Sequential Processes*, Communications of the ACM, Vol. 21 No. 8 (1978), pp.666-677.

[Jackson 1983] Jackson, M., System Development, Prentice Hall, 1983.

[Kerridge and Simpson 1984] Kerridge, J. M., D. Simpson, *Three Solutions for a Robot Arm Controller Using Pascal-Plus, Occam and Edison*, Software - Practice and Experience - Vol. 14, (1984), pp.3-15.

[Lieberman 1981] Lieberman, H., *A Preview of Act-1*, AI-Memo 625, Artificial Intelligence Laboratory, MIT, 1981.

[Lieberman 1986] Lieberman, H., *Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems*, Proc. of 3rd Workshop on Object-Oriented Languages, Bigre+Globule, No. 48, Paris, January 1986.

[Serpette 1984] Serpette, B., *Contextes, Processus, Objets, Séquenceurs: FORMES*, (thèse de 3ème cycle), LITP Research Report, No. 85-5, LITP - Université Paris-VI, Paris, 30 October 1984.

[Shibayama et al. 1985] Shibayama, E., M. Matsuda, A. Yonezawa, *A Description of an Inventory Control System Based on an Object-Oriented Concurrent Programming Methodology*, Jouhou-Shori, Vol. 26, No. 5 (1985), pp.460-468. (in Japanese)

[Shibayama and Yonezawa 1986] Shibayama, E., A. Yonezawa, *Distributed Computing in ABCL/1*, in "Object-Oriented Concurrent Programming" edited by A. Yonezawa and M. Tokoro, MIT Press, 1986.

[Shibayama and Yonezawa 1986a] Shibayama, E., A. Yonezawa, *ABCL/1 User's Manual*, Internal Memo, 1986.

[Smith 1985] Smith, R. G., *Report on the 1984 Distributed Artificial Intelligence Workshop*, The AI Magazine Fall, 1985.

[Special Issue 1981] Special Issue on Distributed Problem Solving, IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-11, No.1, 1981.

[Special Issue 1982] Special Issue on Rapid Prototyping, ACM SIG Software Engineering Notes Vol. 7, No. 5, December 1982.

[Stefik and Bobrow 1986] Stefik, M. K., D. G. Bobrow, *Object-Oriented Programming: Themes and Variation*, The AI Magazine, 1986

[Tschritzis 1985] Tschritzis, D. (Ed.), *Office Automation*, Springer, 1985.

[Yonezawa and Hewitt 1979] Yonezawa, A., C. Hewitt, *Modelling Distributed Systems*, Machine Intelligence, Vol. 9 (1979), pp.41-50.

[Yonezawa et al. 1984] Yonezawa, A., H. Matsuda, E. Shibayama, *Discrete Event Simulation Based on an Object-Oriented Parallel Computation Model*, Research Report C-64, Dept. of Information Science, Tokyo Institute of Technology, November 1984.

[Yonezawa et al. 1985] Yonezawa, A., Y. Matsumoto, *Object-Oriented Concurrent Programming and Industrial Software Production*, Lecture Notes in Computer Science, No.186, Springer-Verlag, 1985.

[Yonezawa et al. 1986] Yonezawa, A., E. Shibayama, T. Takada, Y. Honda, *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*, in "Object-Oriented Concurrent Programming" edited by A. Yonezawa and M. Tokoro, MIT Press, 1986.

[Yonezawa and Tokoro 1986] Yonezawa, A., M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press 1986 (in press).