4.1 Design Principles

The primary design principles of our language 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, represented for example by Smalltalk, in which it is attempted to represent every single concept in computation purely in terms of objects and message passing. In describing the behavior of an object, 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.

4.2 Parallelism and Synchronization Mechanisms

The following summarizes the language mechanisms of ABCL/1 designed for exploiting parallelism and realizing synchronization.

Parallelism

1. Concurrent activations of multiple, independent objects.

   (a) Even if messages are transmitted sequentially, when the receiver objects are different, the activations of the receiver objects overlap in time.

   (b) Messages can be simultaneously sent by an object. ABCL/1 expresses this operation by the following notation called *parallel construct.*

{message passing ... message passing}

The semantics of the parallel construct is that the subsequent action is not performed until all the message passing specified in the construct complete. Accordingly, if now type message passings appear inside the construct {...} it has the same effect as **cobegin** ...**coend**, which is often useful for realizing some kinds of synchronization.

2. Parallelism between the actions of an object which sends a request message and the actions of an object which receives the request in past type or future type message passing.

3. Multicast in Past Type and Future Type: In ABCL/1, the operation of simultaneously sending, in past type or future type, the same message with the same reply destination or the same future variable to multiple objects is denoted by:

   [list of objects `<=` *message]*

   The *list of objects* may be an expression which is evaluated to a list of objects.

Synchronization

1. One at a time: An object always performs a single sequence of actions in response to a single acceptable message. It does not execute more than one sequence of actions at the same time.

2. Waiting mode: Executing `(wait-for ...)` makes an object change into waiting mode.

3. Now Type and Future Type: In now type, the activation of a message sending object is suspended until the corresponding reply comes back. In future type, when a message sending object accesses the future variable, if the future object has not received any value yet, the object has to wait for some value to arrive at the future object.

4. Parallel Constructs: As described above.

5 A Scheme of Distributed Problem Solving: – A Programming Example –

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 modeling and programming language in the concurrent object-oriented paradigm.

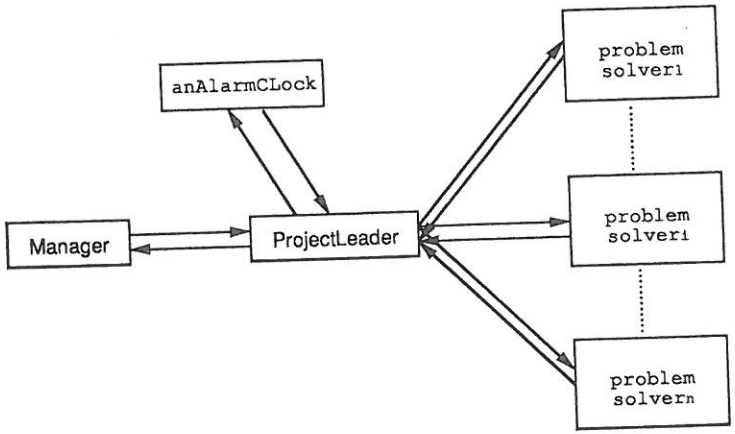

Figure 10: A Scheme for Distributed Problem Solving

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 specification 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).

The definition of the project leader object is given in Figure 11. When the project leader object receives a `[:solve ...]` message from the manager object, it requests an alarm clock object `anAlarmClock` (defined in Figure 9) to start and set the alarming time. Then, the project leader object *multicasts* to the project team members a message that contains the problem specification and ask them to start working on the specified problem. Note that this multicasting the message is done in future type. If a problem solver finds a solution, it sends the solution to the future object


```
[object ProjectLeader
  (state [team-members := nil] [bestSolution := nil]
         [Solutions := (make-future)])
  (script
    (=> [:add-a-team-member M]
        [team-members := (cons M team-members)])

    (=> [:solve SPEC :within INTERVAL]
        (temporary  [mySolution := nil])     ; temporary variable

        [anAlarmClock <= [:start-and-wake Me :after (- INTERVAL 20)]]
        [team-members <= [:solve SPEC] $ Solutions] ;; multicast in future type
        (while (and (not (ready? Solutions)) (null mySolution))
          (progn ... 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] from Sender where (= Sender anAlarmClock)
        (temporary new-deadline)

        (when (null bestSolution)
          [new-deadline := [Manager <<== [:can-extend-deadline?]]]
          (if (null new-deadline)
              (progn
                [team-members <<= [:stop-your-task]]
                (suicide))
              [anAlarmClock <<= [:wake Me :after new-deadline]])))

    (=>> [:you-are-too-late] from Sender where (= Sender Manager)
        (when (null bestSolution)
          [team-members <<= [:stop-your-task]]
          (suicide))))]
```

Figure 11: Definition of `ProjectLeader` Object

bound to a future variable `Solutions` defined in the project leader object. While the project leader engages himself in the problem solving, he periodically checks the future object/variable by executing `(ready? Solutions)` to find if it contains 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 will be seen in Section 7, 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 the current elements in the queue. Note that the sequence of actions from the selection of the best solutions to the termination of the tasks of the team members is enclosed by "`(atomic`" and "`)`" in Figure 11. Thus, the sequence of actions is not interrupted by messages sent in express mode. If no solution is found within the time limit the project leader himself has set, a `[:time-is-up]` message is sent by the alarm clock object in *express* mode. Then, the project leader asks the manager object 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.

6 Reply Destination and Object Creation

What we have presented in the preceding sections is an outline of our computation model ABCM/1 and language ABCL/1. This section discusses several finer points which are important in understanding our computation model and necessary in reading and writing descriptions in ABCL/1.

6.1 Information Passed Around

Let us summarize the information passed around in message passing in our computation model. A message can contain tags, elements in the specified domains of values (e.g., numbers, character strings, lists, etc.), and object names. The object name sent in a message is used in various ways. For example, suppose an object O sends a message M to an object T, and requests T to do some task and return its result of the requested task to a specified object C_1, or O requests T to do the task in cooperation with an object C_2. In such cases, O sends T a message which contains the object name C_1 or C_2 accompanied with a tag indicating the purpose (e.g., `[... :reply-to` C_1 `...]`).

Besides explicit information contained in a message, we assume in our computation model that a receiver object is able to know the name of the sender of the message. In other words, 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. A receiver object can decide whether it accepts or rejects an incoming message

on the basis of who (or what object) sent the message. Therefore, this assumption considerably reinforces the expressive power of the ABCM/1 model. Furthermore, it is extremely easy to realize this assumption. In describing the behavior of such a receiver object, ABCL/1 provides a language construct which allows the sender name to be bound to a variable at the beginning of a script.

(=> *message pattern* `from` *sender-var* `where` *constraint*
... *action* ...)

The name of the sender object which sends a message that satisfies the pattern-constraint pair is bound to *sender-var*. This variable can, of course, appear in *constraint* and ... *action* ....

6.2 Reply Destination

When requested to do some task in now type or future type message passing, the receiver must make a reply by returning either an acknowledgement of the message or the result of the requested task. Accordingly, the receiver of a message must always know which object it must send a reply to. In future type message passing, an object to which a reply should be made is obviously the future object that was sent with the message. However, in now type message passing, an object which receives a reply is not the object which sent the message. (Details of this operation will be explained in Section 7.) In any case, the name of the object to which a reply is sent is called *reply destination*. Since the reply destination is sent with a request message, it is convenient for the receiver object to be able to make the reply destination bound to a dedicated pattern variable as an extended message pattern. In ABCL/1, the extended message pattern is denoted in the following manner.

(=> *message pattern* `@` *destination variable* ...)

When such a form is included in the definition of an object O, a future type message passing:

`[`O `<=` M `$` x`]`

makes x bound to the *destination variable*. As will be explained in Section 7, a certain object is bound to the *destination variable* in now type message passing `[`O `<==` M`]`. Moreover, in past type message passing, any object C is allowed to be sent with a message M as the reply destination in the following manner:

`[`O `<=` M `@` C`]`

Then, of course, C will be bound to the *destination variable*.

Note that the receiver object O can learn the reply destination from the value of the *destination variable* regardless of the type of message passing — whether it is past, now, or future. This fact gives us a great advantage: If a script of an object is defined using the destination variable, then as long as arriving messages satisfy the pattern-constraint pair of the script, the same script can be used to process the messages regardless of any of the following message passing types being used :

- past type accompanied with the reply destination,
- now type, and
- future type.

This means that in defining a script of an object the programmer need not be concerned with which type of message passing is used to send messages that trigger the execution of the script.

Notice that future type differs from past type in one regard. That is, the future object, or a special local object, is the reply destination in future type message passing while any object can be specified as the destination in the past type.

6.3 Delegation

The reply destination sent with a message can be transferred to another object. Suppose an object O requests an object T to do some task by sending a message T with the reply destination C. Let T request another object T' to do this task. In such a case, if T makes a request to T' and sends a message with the destination C, T' can send the result of the task directly (not via T) to C specified by O. This mechanism provides the basic tool for implementing various delegation strategies[49, 14]. The explicit use of destination variables 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 which delegates to an object B all *unacceptable* messages sent to A. The pattern variable any appearing in the definition of A will match any message not matched by the other patterns in the script of A. The destination variable R will bind the reply destination sent with an unknown message. Thus, any kind of (request) message, namely past type with or without reply destination[5], now type, or future type, is matched and delegated to the object B, which will in turn send some answer directly to the reply destination of the original message sent to A.

[5] R binds nil in this case.

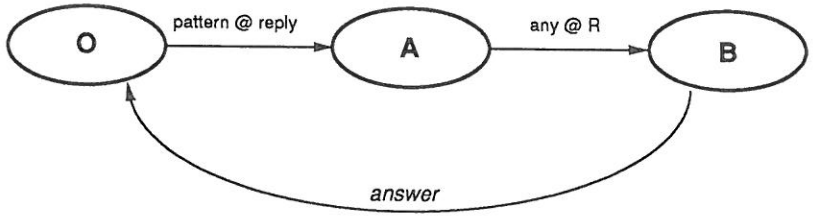

Figure 12: Illustration of Basic Delegation

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

```
[object B
  (state ...)
  (script
    (=> [:request ...] @ RR
      ... [RR <= [:answer ...]] ...)
    ...
    (=> patternBp ...))]
```

Figure 12 illustrates that the answer is delivered directly to the requester O without coming back through A, assuming that the reply destination is specified as the original sender O of the request message.

6.4 Creating Objects and Replying

As mentioned earlier, an object can create another object dynamically in our computation model. In the language ABCL/1, executing an expression `[object ...]` creates an object whose behavior is defined by this `[object ...]`-notation. For instance,

$$[[\texttt{object } A \ \ldots] \texttt{ <= } M]$$

means to create an object named A and send a message M to this object. Similarly,

$$[B \texttt{ <= } [\texttt{object } C \ \ldots]]$$

means to create an object C and send this object to B as a message.

In modeling and simulating various systems, we often need a collection of objects, each performing similar actions or having similar characteristics. In our computation model, a standard way of creating such objects is to define an object which *creates* such similar objects and send the object a message which contains information necessary for initialization of new objects. In ABCL/1, this way of creation is often denoted by

```
[object SomeCreator
  (script
    (=> initializing information pattern @ destination variable
        [destination variable <= [object ...]]))]
```

When a message M for triggering creation is sent in now type to the object `SomeCreator`, the created object will be returned as the result. (The object to be created is denoted by `[object...]`.) That is, when

```
[aNewObject:= [SomeCreator <== M]]
```

is executed, the created object is assigned to the variable `aNewObject`. Here, the object `SomeCreator` roughly corresponds to the *"class"* notion in Simula[23] or Smalltalk-80[32].

The description of the definition of the object `SomeCreator` is a typical example of message passing in which the *destination variable* is given as an extended message pattern, and something (in this case, a new object) is returned to the reply destination bound to this variable. To simplify the description of such a typical message passing, ABCL/1 allows the following abbreviation.

```
(=> message pattern ... !expression ...)
```

This is an abbreviation of

```
(=> message pattern @ destination variable
    ... [destination variable <= expression]...)
```

The *expression* which immediately follows "`!`" is an expression denoting some value. In the above description, this value represents an object to be created. Accordingly, creation and returning of the created object can be described by

```
[object SomeCreator
  (script
    (=> initializing information pattern    ![object...]))]
```

Besides the above way in which similar objects are created, they can be created by copying a prototypical object. Namely, first define an object, then send a message to request this object to make a copy of itself and return it. This method is suggested in [13] and [47]. Moreover, when a request for a copy is made, it is possible to change the present state of the object and make a copy with the intended state. To realize this, ABCM/1 allows an object to have the function to generate copies of itself (or clones) as well as the function to kill itself. In ABCL/1, these functions are performed by executing `(self-copy)` and `(suicide)`, respectively.

7 A Minimal Computation Model

In this section, we will demonstrate that

1. Now type message passing can be reduced to a combination of past type message passing and a selective message reception in waiting mode, and
2. 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 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" type message passing.)

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

Recall that the script of T can be abbreviated as:

```
(=> pattern for M ... !expression ...)
```

Now we introduce a new object `New-object` which just passes any received message to A, and also introduce a *wait-for*-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]
(wait-for from Sender
   (=> 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 waiting mode and waits for a message that is passed by the `New-object`. The constraint

```
where (= Sender New-object)
```

in the wait-for-construct means that only the messages sent by `New-Object` can be accepted. `New-object` serves as a unique identifier for the message transmission:

```
[T <= M @ New-object]
```

from A to T in past type.

7.2 Reducing Future Type

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

```
[object A
   (state ... [x := (make-future)] ...) ;;creating 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 future object created by `CreateFutureObject`. Also we replace 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 13. As mentioned before, it is essentially a queue object, but it only accepts messages satisfying special pattern-and-constraint pairs. A queue object created by `CreateQ` accepts four kinds of messages: `[:empty?]`, `[:enqueue...]`, `[:dequeue]`, and `[:remove-all-elements]`. Note 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.

8 Concluding Remarks

8.1 Importance of the Waiting Mode

The computation model presented in this paper has evolved from the Actor computation model. As will be discussed later, one of the important differences is the introduction of waiting mode in our computation model. As noted at the end of Section 3, without now type (and/or future type) message passing, module decomposition in terms of a collection of objects tends to become unnatural. Thus now type message passing is essential in structuring solution programs. In our computation model, now type message passing is derived from waiting mode and past type message passing in a simple manner as demonstrated in Section 7.1.

8.2 Express Mode Message Passing

We admit that the introduction of the express mode message passing in a high-level programming language is rather unusual and also makes it almost impossible to construct simple mathematical foundations for the language. The main reason for introducing the express mode is to provide a language facility for *natural* modeling. 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 express mode message

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

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

         (=> [:all-values] @ R from Sender where (= Sender Creator)
             (if [box <== [:empty?]]
                 (wait-for       ;;waits for a message which is
                                 ;;not sent by Creator.
                       (=> message from Sender where (not (= Sender Creator))
                           [R <= [message]]))  ; sends a singleton list.
                 ![box <== [:remove-all-elements]]))
                   ;;removes all the elements in the queue and returns them.

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

Figure 13: Definition of Future Object

passing is useful for debugging because it can monitor the states of active objects. Further use of express mode message passing will be found in Chapter 9 of this book.

There is an alternative approach to express mode which does not rely on the "interrupt". In this approach, each object is assumed to have a dedicated queue where messages arriving in express mode are stored in the order of arrival, and no interrupts are granted even when a message arrives in express mode. Interrupts are granted only when an object is in dormant or waiting mode or when an object in active mode executes a special command such as `(check-express-queue)` command. Upon execution of this command, if the queue storing express messages is not empty, the urgent task specified by the first message in the queue is executed. If the queue is empty, the object continues its work. In this manner, the object receiving an express message can exercise complete control over the timing at which it accepts an urgent work requested by the express message. In this case, `(atomic...)` becomes unnecessary. Besides this, no changes need to be made in both ABCM/1 and ABCL/1, and it is possible to define fairly simple mathematical semantics. However, this method is slightly inferior in terms of the convenience of modeling.

8.3 Inheritance

The inheritance mechanism employed by Simula and Smalltalk is said to be an important characteristic of so-called "object-oriented" languages. We agree that this function can effectively reduce the volume of description. However, we have not yet reached a convincing conclusion as to what semantics is necessary to implement this function in a parallel language. For this reason, the inheritance mechanism is not incorporated in ABCL/1. However, the delegation mechanism proposed by Lieberman[49] can be introduced naturally by making use of the "reply destination" as explained in Section 6.3. We have several ideas on the details of the delegation mechanism, semantics of various types of inheritance mechanism, etc. Some of such ideas are discussed in [14].

8.4 Issues from the Standpoint of Relativism

Generally, existence of the global time or information is often rejected in pure frameworks of distributed systems. We take the same stand for ABCM/1. In a sense, however, this forces us to take a relativistic point of view. The real world on which we try to model our system is not actually so extreme in many cases, so we need not take a purely relativistic position. Therefore, we can think of a model which guarantees at least the global time, that is, a model in which each object can refer to the globally synchronized time. To what extent global information should be allowed in our model is an important theme to be studied in the future.

8.5 Relationship to Other Work

Actor Formalisms

Our computation model for object-oriented concurrent programming has evolved from the early Actor formalism which was proposed and studied by C. Hewitt and his group at MIT[34, 35, 79, 47]. Our computation model differs from the Actor formalism in many respects. For example, in our computation model, an object in waiting mode can accept a message which is not at the head of the message queue, whereas, in the Actor formalism, an actor can only accept a message that is placed at the head of the message queue.

Efforts to describe various parallel systems and parallel problem solving algorithms in the framework of the Actor formalism often result in rather unnatural descriptions. One reason for this is that, in the early Actor formalism, all messages are exchanged solely in past type (of ABCM/1). 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. Another reason for unnatural descriptions is that the Actor formalism lacks the "preservation of transmission ordering" described in Section 3. The Actor formalism does not assume this because it tried to model very general parallel computations.

A new Actor computation model has been evolved by G. Agha and C. Hewitt[2, 3]. They combined the advantages of object-oriented programming and those of functional programming, and banished the notion of explicit "states" of an actor by introducing the notion of "behavior replacement". This is mathematically elegant and serves as a basic computation model for object-oriented concurrent programming. But we feel that the notion of "states" is much easier for the designers and programmers to understand, and is more natural for them to model and describe various parallel systems. The exclusion of the notion of "states" required the notion of "insensitive actors" that was introduced in the new Actor computation model. Compared to the notion of "waiting mode" in our computation model, the notion of insensitive actors seems artificial.

Besides the differences in the computation models, it is fair to note that the early actor formalism or new computation model do not have a well-developed description/programming language which is extensively used as our language ABCL/1 has been.

Monitors

Our notion of concurrent objects superficially resembles Hoare's monitor concept[36]. Both have the one-at-a-time property and encapsulation property. But, the use of a monitor, namely, a monitor call is based on inherently bilateral call/return protocols between a caller process and a monitor which is actually a callee. On the other hand, an object is autonomous and has its own thread of control, and message passing between objects is mutually equal (no caller-callee relation) and fundamentally unilateral. Thus an object which receives a request message from another object can immediately send a request message to the other object without replying to the original request message.

Communicating Sequential Processes

Compared to Hoare's Communicating Sequential Processes [37], our language ABCL/1 has characteristics of a more dynamic nature: objects can be created dynamically, message transmission is asynchronous, and the "knows-about"-relation among objects (i.e., network topology) changes dynamically. Thus, ABCL/1 is more suitable for describing systems which are dynamic and open-ended.