Chapter 2.

An Object-Oriented Concurrent Computation Model ABCM/1 and its Description Language ABCL/1

Akinori Yonezawa
Etsuya Shibayama
Yasuaki Honda
Toshihiro Takada
Jean-Pierre Briot

An object-oriented computation model ABCM/1 is presented which is designed for modeling and describing a wide variety of concurrent systems and concurrent algorithms. 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. For more details of the language ABCL/1, see the ABCL/1 User's Guide given in the appendix of this volume.

1 Introduction

This chapter introduces our concurrent computation (or information processing) model called ABCM/1[82] and gives an overview of our description/programming language ABCL/1[78, 83] whose semantics is based on our computation model.

The two fundamental notions in ABCM/1 are anthropomorphic information processing agents called *objects* and forms of their mutual interactions. The forms of mutual interactions are abstracted from actual communication forms found in human or social organizations in such a way that they can be realized by the current computer technology without great difficulties. In our approach, the system to be

modeled/designed/implemented is represented as a collection of *objects* and the interaction of the system's components are represented as *concurrent* message passing among such objects. The application domains of ABCM/1 and ABCL/1 include distributed problem solving, modeling of human cognitive process, modeling and implementation of realtime systems and operating systems, design and implementation of office information systems, distributed event simulation, scientific (numerical) computation etc.

The subsequent account of our computation model is given rather intuitively, but most parts can be described mathematically. However, an elegant mathematical model which completely characterizes our computation model ABCM/1 is still subject to research. In order to avoid ambiguity, the notations of our description language ABCL/1 will be used to explain concepts in ABCM/1, thus introducing the corresponding language constructs in ABCL/1.

ABCM/1 has evolved from the early Actor formalism[34, 35, 76, 79] proposed by C. Hewitt and his group at MIT. Comparisons with the early Actor formalism and the recent Actor model[2, 3] are made in Section 8.5. Though an overview of the language ABCL/1 is given in this chapter, its details and the user's guide are found in the appendix of this volume.

2 Objects

2.1 State and Basic Operations

In our computation model, computation or information processing is performed by a collection of abstract entities called *objects* which become active when they receive messages, and computation proceeds as message transmissions[1] among objects. More than one message transmission may take place in parallel and more than one object may become active simultaneously.

Each *object* in our computation model has its own (autonomous) processing power and it may have its local persistent memory, the contents of which can be accessed only by itself. (It is possible for an object not to have such memory.) The state of an object at a given time is defined as the contents of its local memory at that time.

Upon receiving a message, an object executes a sequence of the following four kinds of basic actions.

- the kinds of message passing described in Section 3.
- creation of objects.

[1] More precisely, computation can be characterized as a partially ordered set of message transmissions. See [60]


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

Figure 1: Basic Object Definition

- referencing and updating of the contents of its local memory.
- various operations (such as arithmetic operations and list processing) on values that are stored in its local memory and passed around in messages.

The messages that an object can accept are determined by the message patterns, the values contained in the messages, and the current state (and mode[2]) of the object. Therefore, in order to define an object, we must specify:

- how its local persistent memory is represented,
- on what conditions messages are accepted, and
- the sequence of actions to be performed when a message is accepted.

To write a definition of an object in our language ABCL/1, we use the notation in Figure 1. `(state ...)` declares the variables which represent the local memory and specifies their initialization. We call such variables *state* variables. *Object name* and the construct `where` *constraint* are optional. The object definition form given in Figure 1 is a basic one. The full-fledged form is found in the User's Guide in the appendix of this volume.

2.2 Three Modes

An object is always in one of three modes: *dormant*, *active*, or *waiting*. An object is initially dormant. The patterns and constraints of messages that an object can accept in its dormant mode are specified by the notation shown in Figure 1. An object becomes active when it receives a message that satisfies one of the specified patterns and constraints. If a message sent to an object defined in the notation satisfies more than one pattern-constraint pair, the first pair (from the top of the script)

[2] See the next subsection.

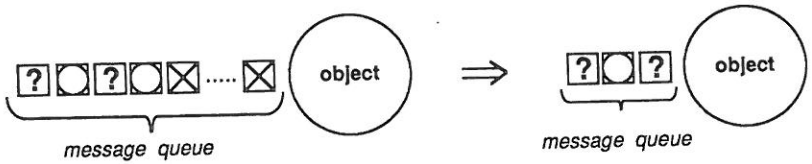

Figure 2: Transition from Dormant Mode to Active Mode

is chosen and the corresponding sequence of actions is performed. As mentioned above, 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. When an active object completes the sequence of actions that are performed in response to an accepted message, if no subsequent messages have arrived yet, it becomes dormant again.

An object in active mode sometimes needs to stop its current execution in order to wait for a message with specified patterns to arrive. In such a case, an active object changes into waiting mode. An object in waiting mode becomes active again when it receives a required message. Each object is assumed to have a conceptually infinite queue (or buffer) for storing messages in the order of arrival.[3] These messages wait in the buffer to be processed by the object. This buffer is considered to exist outside the object so that messages arrive regardless of the operation being currently performed inside the object. When an object is in dormant mode (or it has completed a sequence of actions), if the queue is not empty, then the object takes out of the queue the first acceptable message that has already arrived, and discards [4] the (unacceptable) messages which had arrived before the acceptable message. The object starts performing the sequence of actions specified by the acceptable message. Figure 2 illustrates this, where [x] stands for a message which cannot be accepted, [o] for an acceptable message, and [?] stands for any message.

As mentioned above, an object in active mode sometimes needs to stop its current activity in order to wait for a message with specified patterns to arrive. 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 be stored in the

[3]For more details of message arrival, see Section 3.1.

[4]This is sufficient as a computation model. In a programming language, however, it is desirable that some kind of error message be sent to the objects who sent an unacceptable messages.


```
(wait-for
   (=> message pattern where constraint ... action ...)
      ⋮
   (=> message pattern where constraint ... action ...))
```

Figure 3: Wait-for Construct

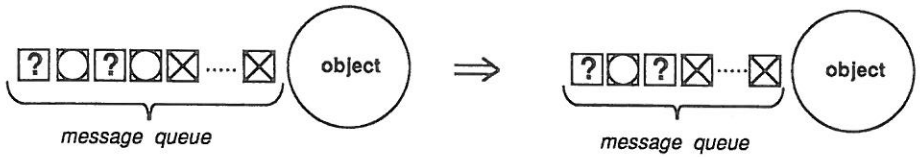

Figure 4: Transition from Waiting Mode to Active Mode

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 active mode changes into waiting mode.

The transition of an object from active mode to waiting mode takes place by performing a special action. In ABCL/1, this action is expressed by a *wait-for*-construct. A wait-for construct specifies the patterns and constraints of messages that are able to reactivate the object. The ABCL/1 notation for this construct is given in Figure 3. Upon a receipt of a message that satisfies one of the message patterns and constraints specified, the object becomes active again. We call this *selective message receipt.*

Dormant mode and waiting mode differ in the handling of messages in the queue. What is common to these modes is that the acceptable message closest to the head of the queue, that is, the message which arrived earliest is selected from all the acceptable messages. In waiting mode, however, only the selected message is taken out of the queue and all the other messages remain where they are. In dormant mode, messages which arrived before the selected message are also removed from the queue and they are simply discarded without being processed. Figure 4 shows transition from waiting mode to active mode. Symbols have the same meanings as in Figure 2.

As an example of the use of wait-for construct, we give, in Figure 5, a skeletal definition of an object which behaves as a buffer of a bounded size. Suppose a `[:put` *product*`]` arrives at the object Buffer. When the storage in the object Buffer is found

```
[object Buffer
  (state  declare-the-storage-for-buffer)
  (script
    (=> [:put aProduct]    ;;aProduct is a pattern variable.
        (when the-storage-is-full
            (wait-for      ;;waits for a [:get] message.
              (=> [:get]
                  remove-a-product-from-the-storage-and-return-it)))
        store aProduct)
    (=> [:get]
        (when the-storage-is-empty
            (wait-for      ;;waits for a [:put ...] message.
              (=> [:put aProduct]
                  send-aProduct-to-the-object-which-sent-[:get]-message))
        remove-a-product-from-the-storage-and-return-it)))]
```

Figure 5: An Example of the Use of Wait-for Constructs

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. As the notation for a wait-for construct suggests, more than one message pattern (and constraint) can be specified, but the ABCL/1 program for the buffer example in Figure 5 contains only one message pattern for each `wait-for` construct.

The rule of mode transition among dormant mode, active mode, and waiting mode is illustrated in Figure 6.

3 Message Passing

3.1 General Characteristics

The general characteristics of message passing between objects in ABCM/1 are given as follows:

1. **No Broadcasting**: An object X cannot send a message directly to an object Y unless X "knows" the name of Y at the time of message transmission. Message passing takes place in a point-to-point (object-to-object) fashion.

2. **Dynamicity of Connection Topology**: The "knows-about" relation is dynamic. An object may know a particular object since the time of its birth, come to know it later, or forget it. However, an object must always know about itself.

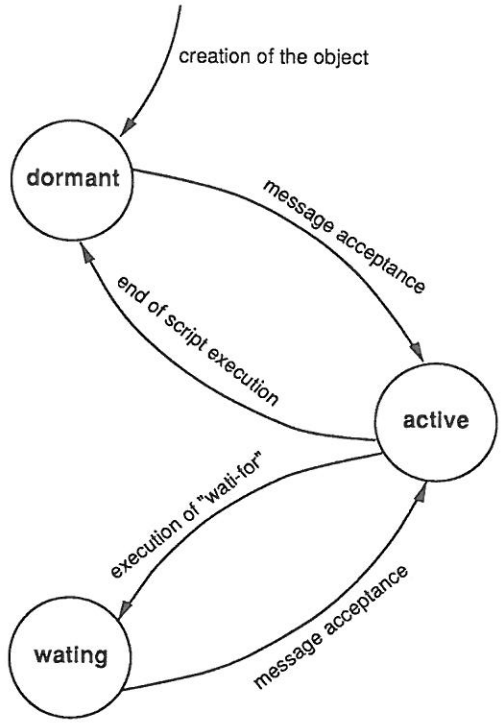

Figure 6: Mode Transition

3. **Asynchrony**: When an object X sends a message to an object Y, X can send the message anytime, irrespective of the current state and mode of Y.

4. **Guaranteed Arrival and Buffered Communication**: A message sent by an object always arrives at the destination within a finite time, and arrived messages are stored in a unique queue (or buffer) associated with the object.

5. **Linearity of Arrival Order**: Messages sent to an object are put in the object's unique queue in the order of arrival. No simultaneity is assumed in message arrival and message arrivals at an object are always linearly ordered.

6. **Preservation of Transmission Ordering**: When an object X sends two messages M and M' to an object Y, M and M' arrive at Y in the same order as the order of the transmissions of the two messages from X.

7. **No Global Clock**: The existence of a system-wide global clock cannot be assumed. In general, unrelated events are considered to take place *concurrently*.

(4) is not guaranteed when a target object does not exist or it has extinguished, but the description language ABCL/1 does an appropriate exception handling in such a case. The length of the object's queue can conceptually be unlimited. Preservation of transmission ordering, namely (6), is not guaranteed in general computer networks, but some new multicomputer architectures guarantee this[6, 24]. It is obvious that the description of even an extremely simple system becomes very cumbersome without this assumption. 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. (7) is a usual assumption for distributed systems where no global simultaneity can be assumed.

3.2 Three Types

As mentioned in the beginning, the message passing forms in ABCM/1 are abstracted from human communication conventions in such a way that parallelism can be exploited and their implementation on computer systems can be done with relative ease. In our computation model, we distinguish three types of message passing: *past*, *now*, and *future*. In what follows, we will discuss each of them in turn.

[**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 immediately 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 following ABCL/1 notation.

$$[T \texttt{ <= } M]$$

The past type corresponds to situations where one requests or commands someone to do some task, and simultaneously proceeds with one's own task without waiting for the requested task to be completed. Also the past type corresponds to a situation where information is simply transferred from one person to another. This type of message passing substantially increases the concurrency of activities within a system. (A variation of past type message passing will be discussed in Section 6.)

[**Now Type Message Passing**] (send and wait) :
When an object O sends a request message M to an object T, O waits not only for M to be received by T, but also waits for the reply to the request T to be sent back to O.

This is similar to ordinary function/procedure calls, but it is more general and differs in the following two points:

- T's activation does not have to end with sending the reply to O. T may continue its computation after sending the reply.
- The reply to the request does not necessarily have to be sent back by T. T can delegate the responsibility of replying to other objects.

These differences are illustrated by Figure 7.

A now type message passing is expressed by the following ABCL/1 notation:

$$[T \texttt{ <== } M]$$

The reply to O may be the result of the requested task or simply an acknowledgment of receiving the request message. 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 reply (i.e., some value or a signal) 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 be briefly mentioned in Section 4.2, but for its details see the Appendix of this book. It should be noted that recursive *now* type message passing causes a local deadlock.

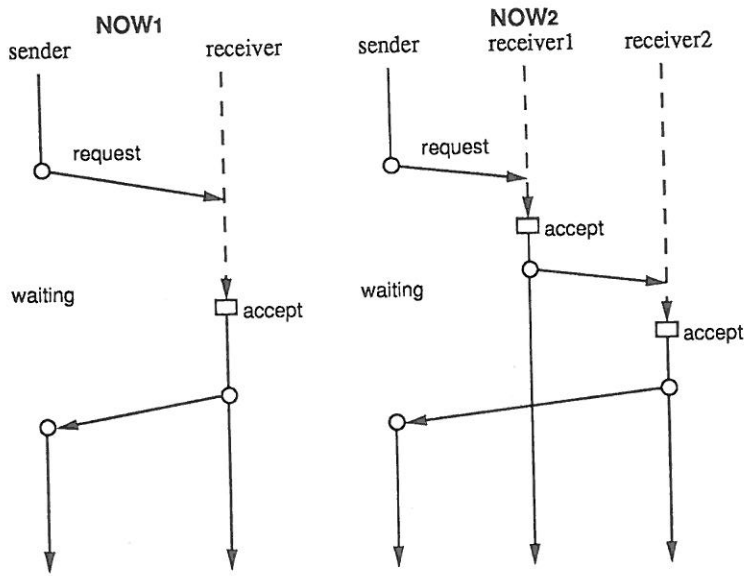

Figure 7: Now Type Message Passing

[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 sent back. But O does not need the result immediately. In this situation, after the transmission of M, O does not have to wait for the result to be returned. 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 is denoted by the following ABCL/1 notation:

$$[T \texttt{ <= } M \texttt{ \$ } x]$$

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 the first element of the queue with a special expression `(next-value x)`, or 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 Section 7.2).

The degree of concurrency in a system is increased by the use of future type message passing. If now type is used instead of 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. For example, Act1, an actor-based language developed by H. Lieberman[47] has a language feature called "future", but it is different from ours. Past and future types of message passing are illustrated in Figure 8. In Section 7, we will see the three types of message passing reduced to one type of message passing.

3.3 Two Modes

So far we have assumed that an object cannot accept any messages as long as it stays in active mode. Once a sequence of actions specified by a message has started, the sequence cannot be interrupted or controlled from the outside unless the object executes a wait-for command and changes into waiting mode. In human communication, however, a person working in an office is often interrupted by a phone call. In such a case, he has to stop his current work to answer the phone call. Depending upon the degree of urgency of the request, he may have to suspend his work to finish the requested task. Modeling such a situation, and designing and implementing a system using the metaphor, we introduce a new mode of message passing called *express*

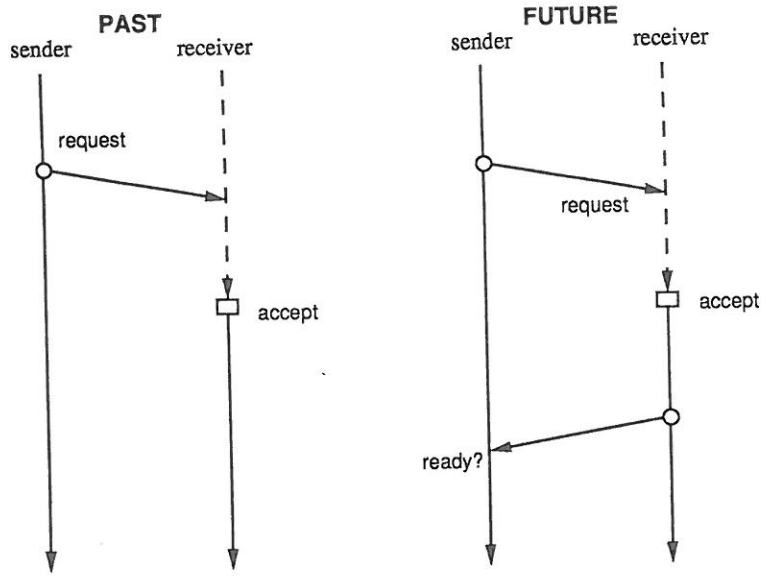

Figure 8: Past and Future Type Message Passing

mode. Accordingly, the mode of message passing that has been discussed is renamed *ordinary mode.* A message sent to an object in express mode externally interrupts the task being executed by the object.

The motivation of the introduction of the express mode is to conduct experiments on description. In other words, we wish to see how the introduction of the express mode simplifies and facilitates the modeling, designing, and realization of a system or how it makes them more complicated and difficult. However, it is obvious that express mode is a serious obstacle to the construction of a pure mathematical model. Consequently, we assume that the express mode explained below exists only in the realm of the language ABCL/1, namely, outside the ABCM/1 model. Section 8.2 suggests another approach to the express mode.

The express mode introduced here is one of the simplest forms of so-called "interrupt." This mode has the following characteristics:

1. The condition for accepting an interrupt initiated by a message sent in express mode are specified by an object which receives the message. (Here "accept an interrupt" implies that the sequence of actions specified by the interrupt message starts immediately.)

2. Interrupts have only one level of priority: There are no multiple-level interrupts. When an object is executing a sequence of actions requested by an *express* mode message, the decision of whether or not to accept a subsequent message arrived in express mode is postponed until the current sequence of actions is completed. That is, an interrupt is not accepted during the handling of the preceding interrupt.

Conditions for accepting an interrupt can be expressed by modifying the notation of object definition shown in Figure 1. The following fragment:

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

in Figure 1 is interpreted as the description of a sequence of actions performed in response to a message sent in the *ordinary mode.* Then, we allow the following notation in the `(script...)`-notation.

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

This specifies that an object accepts an express mode message which satisfies both *message pattern* and *constraint*, and performs ... *action* ... as an urgent task (i.e., interrupt handling).

The timing at which an interrupt is accepted by an object executing the task specified by a message sent in ordinary mode is determined by the detailed definition of the language ABCL/1. Generally, no interrupt is accepted while an object is accessing its local memory. In addition, description of a sequence of actions may be enclosed with "(atomic" and ")" to prevent the sequence from being interrupted during execution in the following manner:

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

Furthermore, it is desirable that the programmer be allowed to specify whether or not to resume the actions which had been performed in response to an ordinary mode message before they were interrupted by an express mode message. For this reason, ABCL/1 provides the non-resume command, which is denoted by:

(non-resume).

The interrupted task will be aborted when this command is executed in the middle of the sequence of actions specified by the message sent in express mode. If this command is not executed by the end of the sequence of actions, the interrupted task resumes.

For each of the three *types* of message passing, it is possible to have the two modes of message passing. Thus there are six different modes/types of message passing. They are denoted by the following ABCL/1 notations where the double-hatted arrows indicate the express mode message passing of the corresponding types, namely *past*, *now*, and *future*.

[T <= M]	[T <<= M]	(past)
[T <== M]	[T <<== M]	(now)
[T <= M $ x]	[T <<= M $ x]	(future)

As an example of the use of express mode message passing, the ABCL/1 definition of an object which models the behavior of an alarm clock is given in Figure 9. In this definition, person-to-wake and count in (state...) declare variables which represent the state of this object. This object receives messages of three patterns: [:start-and-wake ...], [:wake...], and [:stop]. The last two patterns are those for express mode messages. In the description of the message patterns, a symbol starting from a colon (:) represents a tag, and all the other symbols but a string of numbers are pattern variables. The symbol := stands for the assignment operation. Suppose an object does the following past type message transmission for the first time (to this object).

[anAlarmClock <= [:start-and-wake *A* :after 12]]


```
[object anAlarmClock
  (state  person-to-wake  count)
  (script
    (=> [:start-and-wake Person :after time]
      [person-to-wake := Person]
      [count := time]
      (while (> count 0)
        (progn
          (consume-unit-time)
          [count := (sub1 count)]))
      [person-to-wake <<= [:time-is-up]])

    (=>> [:wake Person :after time]
      (non-resume)
      [Me <= [:start-and-wake Person :after time]])

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

Figure 9: Definition of an Alarm Clock Object in ABCL/1

Then, this object changes into active mode. After the object name A is assigned to person-to-wake and 12 to count, this object keeps decrementing the count value by 1 at every unit interval in the while loop. When the count reaches 0, the anAlarmClock object changes into dormant mode immediately after sending the [:time-is-up] message to *A* in express mode. Note that the expression

[person-to-wake <<= [:time-is-up]]

denotes an *express* past type transmission of [:time-is-up] message to the object which is bound to person-to-wake. In the meantime, if a [:stop] message is sent in express mode, the clock stops as the result of executing (non-resume). If the [:wake...] message is sent in express mode, the clock stops ticking so that the receiver of [:time-is-up] and the wake-up time can be changed. The symbol Me has a special meaning in ABCL/1: Me stands for the object which executes the operation in which the "Me" appears.

4 An Overview of the Language ABCL/1

So far the ABCM/1 model has been described using the notations of the language ABCL/1, which may have helped to give an outline ABCL/1. This section discusses the design principles of the programming language ABCL/1 and several important facilities which have not been explained. For more details of ABCL/1, please refer to the appendix of this volume.