OBJECT ORIENTED CONCURRENT PROGRAMMING AND INDUSTRIAL SOFTWARE PRODUCTION

Akinori Yonezawa* and Yoshihiro Matsumoto**

*) Department of Information Science
Tokyo Institute of Technology
Ookayama Meguro-ku, Tokyo 152

**) Toshiba Corporation
Heavy Apparatus Eng. Lab.
1, Toshiba-Cho, Fuchu, Tokyo 183

0. Abstract

A framework of object oriented parallel computations is presented and a programming language called ABCL whose semantics faithfully reflects this computation model is illustrated. A methodology for industrial software production based upon the computation model is discussed.

1. Introduction

Objects in the object oriented programming are conceptual entities which model the functions and knowledge of "things" that appear in problem domains. The fundamental aim in the object oriented programming is to make the structure of a solution as natural as possible by representing it as interactions of objects.

Currently proposed formalisms for object oriented programming (e.g., [GR83] [WM81]) confine themselves in the sequential world. This is too restrictive. Parallelism is ubiquitous in our problem domains. Behaviors of computer systems, human information processing systems, corporative organizations, scientific societies etc. are results of highly concurrent (independent, cooperative or contentious) activities of their components. To model and study such systems, or design various software systems and solve problems by the metaphor of such systems, it is necessary to develop an adequate formalism in which various concurrent activities and interactions of "objects" can be naturally expressed and which is also executable as computer programs.

We have already proposed such a formalism, namely a programming language called ABCL[YO84]. The problem domains to which we apply our formalism include distributed problem solving and planning in AI, modeling human cognitive processes, designing real-time systems and operating systems, and designing and constructing office information systems. Some of characteristic example programs in these domains are also given in [YO84].

2. A Model of Computation

Our computation model for object oriented concurrent programming is a direct descendant of the Actor computation model which was proposed and studied by C. Hewitt and his group at MIT[HE73][HB77][YO77][YO79]. This section gives an intuitive account of what we mean by "objects". The properties of objects explained in this section are those which are inherited from the notion of "actors".

In our computation model, computations are performed by concurrent message passing among procedural modules called objects. Objects model conceptual or physical entities which appear in problem domains. Messages correspond to requests, inquiries or replies.

Each object has its own processing power and it may have its local memory. An object is always in one of two modes, active or inactive and it becomes active when it receives a message. Each object has its own description which determines what messages it can accept and what computations it performs. Upon receiving a message, an object can make simple decisions, send messages to objects (including itself), create new objects and change its local memory according to its description. After performing the described computation, an object becomes inactive until it receives a new message.

Though message passings in a system of objects may take place concurrently, we assume message arrivals at an object be linearly ordered. No two messages cannot arrive at the same object simultaneously and a single message queue sorted in the arrival order is assumed for each object. When a message arrives at an object, if the object is not active and no messages are in the queue, then the message is received by the object. If the object is active or messages are already in the queue, the message is put at the end of the queue.

There are two classes of objects, "serialized" objects and "unserialized" objects. A serialized object is activated by one message at a time. While a serialized object is being activated by a message, it is locked and cannot receive a new message. We do not assume this property for unserialized objects. In the subsequent discussion we focus our attention on serialized objects.

3. Types of Message Passings and Continuations

To study the versatility of our model of computation, we modeled and described various parallel or real-time systems using a simple set of notations. In the course of this process, we made a simple assumption on the message arrival and also found it sometimes necessary to distinguish three types of message passings which are not included in the original Actor model of computation.

[Arrival Ordering Preservation Assumption]

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

This assumption was not included in the original Actor model of computation[HB77]. Without this, however, we cannot model, for example, a computer terminal or displaying device as an object. The terminal object must receive character messages in the same order as their transmissions from an object that models an output handling program in an operating system.

["Past" Type Message Passings]

> Suppose an object O is being 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 (provided that the transmission of M is not the last action during the current activation of O).

We call this type of message passings "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 by the following notation.

$$[T <= M] \tag{1}$$

The past type corresponds to the 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 passings substantially increase concurrency of activities within a system. (Past type message passings can further be divided into two kinds which may reflect two different implementation strategies. In one kind, an object which transmits a message does not continue its computation until the arrival of the message is assured, while in the other kind, the object continues its computation as soon as the message leaves the object. Of course the latter kind allows higher concurrency than the former one, but may sacrifice the robustness against various unexpected errors in the system's components.)

["Now" Type Message Passings]

> 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 return some information to O. If T does not return anything, O waits until T's current activation caused by M ends.

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 during the same activation caused by M. A now type message passing is denoted by

$$[T <== M] \tag{2}$$

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 that his message was surely received by the object though he may waste time in waiting. The returned information (certain values or signals) is denoted by the same notation as the message passing. Namely, (2) denotes not merely an action of sending M to T by a now type message passing, but also denotes the information returned by T. If the activation of T ends without returning any information, we assume, by convention, (2) denotes some special value (e.g. nil).

Now type message passings provide a quite convenient means to synchronize concurrent activities performed by independent objects when it is used together with the parallel construct that will be discussed in a later section. (It should be warned that recursive now type message passings cause local deadlock.)

["Future" Type Message Passings]

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, O does not have to wait for T to return the result after the transmission of M. It continues its computation immediately. Later on when O needs that result, it checks O's internal memory area that was specified at the time of the transmission of M. If the result is ready, it is used. Otherwise O waits there until the result is obtained.

A future type message passing is denoted by

[x := [T <= M]] (3)

where x is the specified memory area (or a variable). A system's concurrency is increased by the use of future type message passings. If the now type were used instead of future type, O would have to waste time by waiting for the currently unnecessary result to be produced. The future type message passing feature has been incorporated in previous object oriented programming languages [LI81][FU84].

Since the now type and future type message passings are not allowed in the Actor computation model, an actor A which sends a message to a target actor T and expects a response from T must terminate its current activation and A must wait for the response to arrive as just one of incoming messages. To discriminate T's response from other incoming messages, A must make some provision before it sends the message to T. Also the necessity of the termination of A's current activation causes unnatural breaking down of A's task into small pieces.

In the above discussion, the contents of a message was left vague. We should make it clear in order to make a more precise account of how various information flows among objects through message passings. A message consists of two parts, an RR-part and a C-part. An RR-part which stands for a request/reply part tells the message receiving object about the contents of a request or it is used to carry a reply or result of a requested task.

When a message is sent by a "past" type message passing to request an object to do some task, it is sometimes useful for the message sending object to have a means to specify a destination object where the result of the requested task should be sent. We call this destination object a "continuation". A C-part which stands for a continuation-part provides this means. (Without an explicit indication of the destination, the only thing one can do is either to have the object which carries out the requested task keep the result within itself, or to have the result sent to some default object.) In our notational convention, a message is expressed by a pair whose first and second parts are separated by a period. The first part and second part correspond to its RR-part and C-part, respectively. Namely, it is of the form

[<RR-part> . <C-part>].

When the C-part of a message need not be specified, it is left blanked. In this case the message is a Lisp singleton list of the following form

[<RR-part>]

where the period after <RR-part> is omitted. In fact, the C-parts of messages sent by "now" or "future" types must be void, because the destination to which the result is supposed to be sent is predetermined. Namely, a "future" type message passing itself specifies a part of the internal memory (or a variable) of the message sending object. For the case of "now" type, we can view this type of message passings as a special case of "now" type message passings.

4. A Language ABCL

In order to describe behaviors of objects in more precise and concrete terms, we need to develop a language. We have tentatively designed and implemented a programming language called ABCL (An object-Based Core Language). The purpose of designing this language is manifold. It is intended to serve as an experimental programming language to construct software in the framework of object-based concurrent programming. The kind of the application domain we emphasize includes the AI fields and we plan to use this language as an executable thought-tool for developing the paradigm of distributed problems solving [YO84][SI81] and cognitive models. It is also intended to serve as an executable language for modeling and designing of various parallel or real time systems. Thus ABCL serves as a language for rapid prototyping[SI82].

The primary design principles of this language are:

[1] Clear semantics: the semantics of the language should be as close to the simple underlying computation model as possible.

[2] Practicality: various features of Lisp can be directly utilized to exploit efficiency and programming ease as long as the framework of the object oriented programming style is maintained.

The purpose of the present paper is not to introduce the details of the language, we keep its explanation minimum. For those who are interested in the language, see [YO84][MY84].

4.1. Defining Objects

Each object has a fixed set of message patterns it accepts. To define the behavior of an object, we must specify what computations or actions it will perform for each such message pattern. The description of computation for each message pattern is called a "script". If an object has its local memory, its computations may be affected by the current contents of such memory. Thus in order to define an object with local memory, we must also describe how the object's local memory is represented. Representations of local memory are variables or internal objects which have its own local memory.

To write a definition of an object in ABCL, we use a notation of the following form (4). (state: ...) declares the representation of local memory and initializes it. Scripts are basically expressed in terms of message passings, referencing to variables and calculating values or manipulating list structures using Lisp functions. These actions are performed sequentially unless special parallel execution constructs are used.

```
[object <object-name>
   (state: <representation of memory> )

   (=> [<pattern>]   <script>   )                                    (4)
    ...
    ...
   (=> [<pattern>]   <script>   )    ]
```

As an illustrative example, let us consider an object which models the behavior of a semaphore. A semaphore has a counter to store an integer with a certain initial value (say 1) and also it has a queue for waiting processes which is initially empty. We represent the counter as a variable and the queue as an (internal) object which behaves as a queue. A semaphore accepts two patterns of messages, [p-op: . C] and [v-op: . C] which correspond to the P-operation and V-operation. In ABCL, symbols ending with a colon in messages or message patterns are constants, whereas symbols starting with a capital letter or "_" are pattern variables which bind components of incoming messages. (p-op: and v-op: are constants. C is a pattern variable which binds the C-part (, namely the continuation) of an incoming matching message.)

Using the notation (4), a definition of the semaphore object is shown below.

```
[object aSemaphore
  (state: [counter := 1]                         ; := means assignment.
          [process_q = [CreateQ <== [new:]]])    ; = means binding.

  (=> [p-op: . C]  <script for P-operation>      )

  (=> [v-op: . C]  <script for V-operation>      ) ]
```

Note that a "now" type message passing is used to create a queue object and it is bound to a symbol process_q.

4.2. Creating Objects

CreateQ in the above example is an object which creates and returns a new object which behaves as a queue. We assume it is defined elsewhere. CreateQ can be viewed as a class of queues and the created queue object as an instance of the queue class (if we use the terminology of AI or SmallTalk[GB83]). In ABCL, rather complicated notions such as classes and meta-classes are unified as the notion of objects, which allows us to manipulate classes and meta-classes as objects.

Objects which create and return an object are often defined in the following fashion.

```
[object CreateSomething
  (=> [<initial-information> . <continuation>]

    [<continuation> <= [[object ...                  ; a newly created object is
                          (=> [...] ...)              ; sent to <continuation>
                            ...
                          (=> [...] ...)]

                        nil]]
  )]
```

Namely, a message whose RR-part is a newly created object defined by [object ....] and whose C-part is nil is sent to <continuation>. Creating a new object and sending it back to the continuation is one of typical situations where a message with its C-part being nil is sent to the original continuation. A simple abbreviated notation in ABCL expresses this scheme of message transmission.

```
        (=> [<request>]     ... !<expression> ...),
```

This is equivalent to

```
     (=> [<request> . <continuation>]
            ...  [<continuation> <= [<expression> . nil]] ...).
```

5. Parallelism and Synchronization

5.1 Parallelism

Using the abbreviated notation explained in the previous section, the object which creates and returns a semaphore object is defined in Figure 1. In the script for p-op:, (subl counter) is an invocation of a lisp function subl and the result updates the contents of counter. When process_q object is empty in executing the script for v-op:, a [go:] message is sent to the continuation which is bound to C; otherwise the first process that has been waiting is removed from the queue and [go:] messages are sent to this process and the continuation simultaneously.

As noted earlier, a script is usually executed sequentially. But when a special construct denoted by

```
             { E1 , ... , Ek }
```

is executed, the executions of E1,...,Ek start simultaneously. The execution of this construct, which we call a "parallel construct", does not end until the executions of all the components E1,...,Ek end. When the components of a parallel construct are all past type message passings, the degree of parallelism caused among the message receiving objects is not much greater than the degree of parallelism caused by the sequential execution of the components because of very small time cost of a message transmission. But if a parallel construct contains now type message passings, the possibility of exploitation of parallelism among the message receiving objects is very high.

```
[object CreateSemaphore
  (=> [init: N]              ;when [init: ...] is sent, the following object
                             ;is created and returned.
    ![object                      ; definition of a semaphore object begins.
        (state:
           [counter := N]
           [process_q = [CreateQ <== [new:]]])

        (=> [p_op: . C]
          [counter := (sub1 counter)]
          (case (> 0 counter)
             (is t               ; if counter is negative
               [process_q <= [enqueue: C]])
             (otherwise
               [C <= [go:]])))

        (=> [v_op: . C]
          [counter := (add1 counter)]
          (case [process_q <== [dequeue:]]
           (is nil                             ; if process_q is empty
               [C <= [go:]])
           (is FrontProcess    ; the head of process_q is bound to FrontProcess
             { [FrontProcess <= [go:]],
               [C <= [go:]] }    )))
     ] )]
```

Figure 1. Defining a Semaphore Object

After having explained parallel constructs, it is an appropriate time to review the basic types of parallelism provided in ABCL.

[1] concurrent activations of independent objects.

[2] parallelism caused by past type message passings.

[3] parallelism caused by parallel constructs.

5.2. Synchronization

Parallel constructs are also powerful in synchronizing the behaviors of objects because the semantics of a parallel construct requires that its execution completes only when the executions of all the components complete. When a parallel construct, in a script, contains a now type message passing, all the intended actions of the message receiving objects must be completed before going on to the execution of the rest of the script. (Note that we need no synchronization if all the components of a parallel construct are past type message passings.)

For example, suppose the movement of a robot arm is actuated by three step motors, each being responsible for the movement along different coordinates[KS84] and for each motor there is an object operating it. In order to pick up something by the fingers attached to the arm, the control program sends signals to the three objects in parallel, and it must wait until the rotations of all the three motors stop. See a fragment of the program below.

```
 ... { [motorX <== [step: 100]], [motorY <== [step: 150]], [motorZ <== [step: -30]] }
     <command to pick up>  ...
```

We conclude this section to remind one that ABCL provides the following four basic mechanisms for synchronization.

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

[2] now type message passing: it does not end until a certain result is returned or the activation of the message receiving object comes to end.

[3] future type message passing: when the specified variable is referred to, the execution is suspended if the contents is not updated yet.

[4] parallel construct: as discussed above.

Although we have shown an implementation (or modeling) of semaphores in terms of the object paradigm, we think semaphores are too primitive and unstructured as a basic synchronization mechanism. Thus we have no intention of using semaphore objects to synchronize behaviors of objects. Our experience of writing programs which require various types of synchronization suggests that combinations of the four mechanisms listed above seem sufficiently powerful for dealing our current problems.

6. Use of ABCL for Industrial Software Production

Software design in the practical software manufacturing is a refinement process in which requirements specified in a problem domain are gradually transformed into programs. In the waterfall type lifecycle models, the refinement is done through multiple numbers of consecutive phases. In an earlier phase, a specification may be transformed into the second specification which has more concreteness, or less abstraction. In later phases, a program written in a higher language will be translated into the program written in a lower language. Dijkstra described the concept of abstract machine M(i) and description D(i) on abstract level i such that execution of D(i) on M(i) satisfies the purpose of code D(0) which is to be executed on a real machine M(0). At the next lower level, level(i-1), D(i-1) is executed on M(i-1) [DI72].

The phases, descriptions and abstract levels in our practical software production are defined as follows [MA84]:

i=4; D(4): requirements specification,
i=3; D(3): functional design specification or external design specification,
i=2; D(2): program design specification or detail design specification,
i=1; D(1): program text,
i=0; D(0): binary code.

phase(4): requirements analysis and definition,
phase(3): functional design,
phase(2): program design,
phase(1): programming,
phase(0): translation or compilation.

It has been our consistent desire that the designer does not have to go down into too much low level description. In this sense, we owe compilers and assemblers for staying away from level 0, but the actual productivity value is still much to be improved. If we could have a language in which we could write a specification of a higher abstract level (i.e., the functional design or program design level) and translate it into binary code, it will bring us a great benefit for increasing the productivity. The descriptions of higher abstract levels reflect the designer's internal models or concepts more directly than those of lower levels do. Therefore the languages which could meet our desire of describing specifications of higher levels should be those which could describe human (expert) knowledge in more direct forms. Our object oriented concurrent language ABCL seems to be one candidate of such desired languages.

We will give a simple example selected from our actual software and show how ABCL has been used for this example. This example is a part of software to print out the operational guidance messages for human operators in local dispatch control stations of an electric power system.

In response to requests from the central supervisory office, an operator disconnects facilities which are required to be checked, repaired or cleaned. The facilities may be transmission lines, electric bus lines in a substation, or transformers. When a facility is disconnected, it will become out of service. Therefore the operator must put alternative facilities into service so that they will substitute for the functions of the disconnected facility. By setting up an alternative circuit which rounds about the facility to be disconnected, the consumer of electric power will not be affected.

The programs written in ABCL in Figure 2 and Figure 3 show a part of the software which reasons the method to set up this round about operation. The programs describe a knowledge to disconnect a transmission line named "LINE-N". Figure 3 describes the knowledge before it is instantiated. The description in Figure 2 instantiates it so that the newly created object whose name is now Disconnect_operation_for_LINE_N contains the knowledge to disconnect the transmission line "LINE-N". This knowledge is modelled as the object described in line 4 to line 17 in Figure 3, where the value of the pattern variable "Line_name" is "LINE-N".

```
1  [Disconnect_operation_for_LINE_N =
2          [Create_line_disconnect_knowledge <== [for: LINE-N]]]

3  [Disconnect_operation_for_LINE_N <=
4     [disconnect_transmission_line_and_report_to: . Supervisor]]
```

Figure 2.

```
1  [object Create_line_disconnect_knowledge
2    (=> [for: Line_name]
3
4     ![object Me
5        (state:
6          [caller_name := nil]
7          [load_facility_list := (func$fetch_local_facility_for  Line_name)]
8          [substation_operation = [Create_substation_operation <= [new:]]])
9
10       (=> [disconnect_transmission_line_and_report_to: . Caller]
11         [caller_name := Caller]
12         [substation_operation <=
13            [handle_outservice_of: load_facility_list and_report_to: . Me]])
14
15       (=> [substation_operation_finished:]
16         (func$disconnect_transmission_line  Line_name)
17         [caller_name <= [transmission_line_disconnected:]])] )]
```

Figure 3.

The object "Disconnect_operation_for_LINE_N" is activated when it receives messages with the patterns shown in line 10 and 15. When it receives a message of the pattern in line 10, the name of the object which is bound to the pattern variable "Caller" is stored in the variable "caller_name", and it activates the object "substation_operation" by sending the message shown in line 13, and then it becomes inactive. When the substation object finishes its operation, it is supposed to send back to Disconnect_operation_for_LINE_N a message of the pattern shown in line 15. Thus Disconnect_operation_for_LINE_N becomes active again and does the disconnecting of transmission lines for LINE-N (line 16) and finally it sends to the original caller (here, Supervisor) a message indicating the end of the operation (line 17).

The programs shown reflect the knowledge of the operator more directly because it simulates the behavior of the operator who first becomes active by the instruction from the central office, analyzes it, transmits his instructions to his subsidiaries, disconnects his facility and then reports the completion to the central office.

A create-object such as the one shown in Figure 3 is defined for each type of facility. For example, we have Create_transformer_disconnect_knowledge for the transformer. The collection of such objects is called "knowledge base" in our system. This knowledge base is accessed and maintained through the knowledge base management system.

The object defined in Figure 3 illustrates just one type of objects which represent "knowledge chunk" in the form of state changes. We have other types of objects. For example, a type of object represents a block of production rules. Another type of object can represent a set of fuzzy logic. We are convinced that all these types of objects are required in order to implement our system which supports plant operators.

7. Concluding Remarks

7.1. Programming Environments

The first stage of (concurrent) programming in the object oriented style is to determine, at a certain level of abstraction, what kinds of objects are necessary and natural to have in solving the problem concerned. At this stage, message passing relations (namely what objects send messages to what objects) are also determined.

Since it is often useful or even necessary to effectively overview the structure of a solution or result of modeling, those identified objects and message passing relations should be recorded and be retrieved or even manipulated graphically. For this purpose, we are currently designing and implementing a programming aid system on a SUN-II Workstation with multi-window facilities and a standard pointing device. A typical action using this system might be to add a node to a graph which represents message passing relations among objects (where nodes correspond to objects), point the node by a mouse to get a pop-up menu and select/perform operations such as editing and compiling the program for the object.

7.2.Other Examples

A wide variety of example programs have been written in ABCL and we are fairly convinced that essential part of ABCL is robust enough to be used in the intended domains. Examples we have written include distributed problem solving by a project team[YO84], parallel discrete simulation[YO84a], deamons, production rules, robot arm control[KS84], bounded buffers, integer tables[HO78], simulation of data flow computations, process schedulers etc. Also a simplified example of a mill speed control program[MA84] written in ABCL is given in the Appendix below.

Acknowledgements

The first author expresses his deep appreciation to H. Matsuda and E. Shibayama who made various contributions to the present work including the design and implementation of ABCL.

References

[DI72] Dijkstra, E.W.: Notes on Structured Programming, Structured Programming, (Eds. O.J. Dahl, et al.), Academic Press, 1972.

[FU84] Fukui, S.: An Object Oriented Parallel Language, Proc. Hakone Programming Symposium, (1984), in Japanese.

[GR83] Goldberg, A. and Robson, D.: SmallTalk80 - The Language and its Implementation -, Addison Wesley, 1983.

[HB77] Hewitt, C. and Baker, H.: Laws for Parallel Communicating Processes, IFIP-77, Toronto, (1977).

[HE73] Hewitt, C. et al.: A Universal Modular Actor Formalism for Artificial Intelligence, Proc. Int. Jnt. Conf. on Art. Int., (1973).

[HO78] Hoare, C.A.R.: Communicating Sequential Processes, CACM, Vol. 21 No. 8, 1978.

[KS84] Kerridge, J. M. and Simpson, D.: Three Solutions for a Robot Arm Controller Using Pascal-Plus, Occam and Edison, Software - Practice and Experience - Vol. 14, (1984), pp.3-15.

[LI81] Lieberman, H.: A Preview of Act-1, AI-Memo 625, MIT AI Lab., (1981).

[MA84] Matsumoto, Y.: Management of Industrial Software Production, IEEE Computer Vol. 17, No. 2, (1984), pp.59-72.

[MY84] Matsuda, H. and Yonezawa, A.: ABCL User's Manual, Internal Memo, Dept. of Information Science, Tokyo Institute of Technology, November 1984.

[SI82] Special Issue on Rapid Prototyping, ACM SIG Software Engineering Notes Vol. 7, No. 5, December 1982.

[SP81] Special Issue For Distributed Problem Solving, IEEE Trans. on Systems, Man and Cybernetics, Vol. SMC-11, No.1, (1981).

[WM81] Weinreb, D. and Moon, D.: Flavors: Message Passing in the Lisp Machine, AI-Memo 602, MIT AI Lab., (1981).

[YO77] Yonezawa, A.: Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, (Ph.D. Thesis), TR-191 Laboratory for Computer Science, MIT, 1977.

[YO79] Yonezawa, A. and Hewitt, C.: Modelling Distributed Systems, Machine Intelligence, Vol. 9 (1979).

[YO84] Yonezawa, A, Matsuda, H and Shibayama, E: An Object Oriented Approach for Concurrent Programming, Research Report C-63, Dept. of Information Science, Tokyo Institute of Technology, November 1984.

[YO84a] Yonezawa, A.: 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.

Appendix   Mill Speed Control Program

A simple ABCL program for controlling mill roller speed is given below. Input_Handler object receives sensor data which consist of 6 heat sensor values (hl-h6), a load cell value (ls), and an emergency stop flag (stp). Speed_Selection object determines the right speed of the mill roller by considering the current position of the slab and the data sent from Input_Handler. Speed_Control object sets the actual speed of the roller. Note the concurrency among the three object.

```
                                      *****************
        Sensor Value      ==>         * Input_Handler *
                                      *****************

                                             ||
                                             vv

    *****************          *******************
    * Speed_Control *   <==    * Speed_Selection *
    *****************          *******************
```

```
[object Input_Handler
  (=> [sensor_value: Frame]
    (case Frame
      (is [hl: 0 h2: 0 h3: 0 ls: 0 h4: 0 h5: 0 h6: 0 stp: 0]
        [Speed_Selection <= [input: 'il]] )

      (is [hl: 1 h2: 0 h3: 0 ls: 0 h4: 0 h5: 0 h6: 0 stp: 0]
        [Speed_Selection <= [input: 'i2]] )

      (is [hl: 0 h2: 1 h3: 0 ls: 0 h4: 0 h5: 0 h6: 0 stp: 0]
        [Speed_Selection <= [input: 'i3]] )

       ...  <cases for 'i4 to 'i8 are omitted>  ...

      (is [hl: 0 h2: 0 h3: 0 ls: 0 h4: 0 h5: 1 h6: 0 stp: 0]
        [Speed_Selection M= [input: 'i9]])

      (is [hl: 1 h2: 0 h3: 0 ls: 0 h4: 0 h5: 0 h6: 1 stp: 0]
        [Speed_Selection <= [input: 'il0]] )

      (is [hl: _ h2: _ h3: _ ls: _ h4: _ h5: _ h6: _ stp: 1]
        [Speed_Selection <= [input: 'il1]])

      (otherwise
        [Speed_Selection <= [input: 'il2]])  ))]
```

```
[object Speed_Selection
  (state: [current_slab_loc := 'no_slab])

  (=> [input: Status]
    (case (list current_slab_loc Status)

      (is ['no_slab 'i1]
        [Speed_Control <= [speed: idle:]] )

      (is ['no_slab 'i2]
        [current_slab_loc := 'coming]
        [Speed_Control <= [speed: low:]] )

      (is ['coming  _I]
        (case (member _I '(i3 i4 i5))
          (is t
            [Speed_Control <= [speed: low:]])))

      (is ['coming  'i6]
        [current_slab_loc := 'rolling]
        [Speed_Control <= [speed: high:]] )

      (is ['rolling 'i6]
        [Speed_Control <= [speed: high:]] )

      (is ['rolling 'i7]
        [current_slab_loc := 'leaving]
        [Speed_Control <= [speed: low:]] )

      (is ['leaving  _I]
        (case (member _I '(i7 i8 i9 i10))
          (is t
            [Speed_Control <= [speed: low:]])))

      (is ['leaving  'i1]
        [current_slab_loc := 'no_slab]
        [Speed_Control <= [speed: idle:]] )

      (otherwise
        [Speed_Control <= [speed: stop:]]) )) ]


[object Speed_Control
  (=> [speed: SP]
    (case SP
      (is idle:     (set_roller_speed 'idle))

      (is low:      (set_roller_speed 'low))

      (is high:     (set_roller_speed 'high))

      (is stop:     (set_roller_speed 'stop)) ))]
```