3

Modelling Distributed Systems

A. Yonezawa† and C. Hewitt
Artificial Intelligence Laboratory
Massachusetts Institute of Technology, USA

1. INTRODUCTION

Distributed systems are multi-processor information processing systems which do not rely on the central shared memory for communication. The importance of distributed systems has been growing with the advent of "computer networks" of a wide spectrum: networks of geographically distributed computers at one end, and tightly coupled systems built with a large number of inexpensive physical processors at the other end. Both kinds of distributed system are made available by the rapid progress in the technology of large-scale integrated circuits. Yet little has been done in the research on semantics and programming methodologies for distributed information processing systems.

Our main research goal is to understand and describe the behaviour of such distributed systems in seeking the maximum benefit of employing multi-processor computation schemata.

The contribution of such research to Artificial Intelligence is manifold. We advocate an approach to modelling intelligence in terms of cooperation and communication among knowledge-based problem-solving experts. In this approach, we present a coherent methodology for the distribution of active knowledge as a knowledge representation theory. Also this methodology provides flexible control structures which we believe are well suited to organizing distributed active knowledge. Furthermore, we hope to make technical contributions to the central issues of problem solving, such as parallel versus serial processing, centralization versus decentralization of control and information storage, and the "declarative-procedural" controversy.

This paper presents ideas and techniques in modelling distributed systems and their application to Artificial Intelligence. In Secs. 2 and 3, we discuss a

†Now with the Department of Information Science, Tokyo Institute of Technology (Tokyo Kogyo Diagaku), Oh-Okayama, Meguro, Tokyo, Japan.

model of distributed systems and its specification and proof techniques. In Sec. 4 we introduce the simple example of an air line reservation system and illustrate our specification and proof techniques by this example in the subsequent sections. Then we discuss our further work.

2. A MODEL OF DISTRIBUTED SYSTEMS

The actor model of computation (Grief and Hewitt 1975, Grief 1975, Hewitt and Baker 1977) has been developed as a model of communicating parallel processes. The fundamental objects in the model of computation are *actors*. An actor is a potentially active piece of knowledge (procedure) which becomes active when it is sent a message which is also an actor. Actors interact by sending messages to other actors. More than one transmission of messages may take place concurrently. Two events will be said to be *concurrent* if they can possibly occur at the same time. Each actor decides how to respond to messages sent to it. An actor is defined by its two parts, a *script* and a set of *acquaintances*. Its script is a description of how it should behave when it is sent a message. Its acquaintances are a finite set of actors that it directly *knows about*. If an actor A knows about another actor B, A can send a message to B directly. The concept of an *event* is fundamental in the actor model of computation. An event is an *arrival* of a message from actor M at a target actor T and is denoted by the expression $[\![T <= M]\!]$. A computation is expressed as a partially ordered set of events. We call this partial order the *precedes* ordering. Events which are unordered in the computation are concurrent. Thus the partial order of events naturally generalizes the notion of serial computation (which is a sequence of events) to that of parallel computation.

A collection of actors which communicate and cooperate with each other in a goal-oriented fashion can be implemented as a single actor. In essence, actors are procedural objects which may or may not have local storage. Some may behave like procedures, and some may behave like data structures. Modules in distributed systems are modelled by actors and systems of actors. In this regard, IC (integrated circuit) chips can be viewed as actors.

Knowledge and intelligence can be embedded as actors in a modular and distributed fashion. For example, *frames* (Minsky 1975), (Kuipers 1975), *units* (Bobrow and Winograd 1976), *beings* (Lenat 1975), *stereotypes* (Hewitt 1975) etc. which represent modular knowledge with procedural attachments, are modelled and implemented as actors. In the context of electronic mail systems and business information systems, objects such as forms, documents, customers, mail collecting stations, and mail distributing stations are easily modelled and implemented as actors.

Messages which are sent to target actors usually contain *continuation* actors to indicate where the replies to the messages should be sent. By virtue of continuations in messages, the message-passing in the actor model of computation realizes a universal, yet flexible control structure without using implicit mechanisms such as push-down stacks. Various forms of control structure such as go-to's,

procedure calls, and co-routines can be viewed as particular patterns of message passing (Hewitt 1977).

This model of computation has been implemented as a programming language, PLASMA (Hewitt 1977). The script of an actor can be written as a PLASMA program. We believe that message-passing semantics provide a basis for programming languages for distributed systems. In Sec. 5, an example of a PLASMA program is given as a script of a flight-data actor in the model of a simple air line reservation system.

3. TECHNIQUES FOR SPECIFICATION AND VERIFICATION

In designing and implementing a distributed (message-passing) system, it is desirable to have a precise specification of the intended behaviour of the distributed system. Also we need sound techniques for demonstrating that implementations of the system meet its specifications. Below, we give some of the central ideas of our specification and proof techniques based on the model introduced in the previous section. More detailed work will be found in Yonezawa (1977).

In specifying the behaviour of a distributed system, it is not only practically infeasible, but also irrelevant to use global states of the entire system or the global time axis which governs the uniform time reference throughout the system. We are concerned with states of modular components of a distributed system which interact with each other by sending messages. Thus we are interested in the states of actors participating in an event at the instance at which the message is received.

In our specification language, *conceptual representations* are used to express local states of actors (modules). Conceptual representations were originally developed to specify the behaviour of actors which behave like data structures (Yonezawa and Hewitt 1976). We have found them very useful to express states of modules in distributed systems at varying levels of abstraction and from various view-points. The basic motivation of conceptual representations is as an aid in the provision of a specification language which serves as a good interface between programmers and the computer, and also between users and implementers. Conceptual representations are intuitively clear and easy to understand, yet their rigorous interpretations are provided. Instead of going into the details of syntactic constructs of conceptual representations, we shall give a few examples. Below !⟨exp⟩ is the unpack operation on ⟨exp⟩, that is individually writing out all the elements denoted by ⟨exp⟩.

```
(CELL (contents: A))                    ;a cell containing A as its contents.
(QUEUE (elements: [A B C]))             ;a queue with elements A B C.
(NODE (car: A)(cdr: B))                 ;a LISP node containing A and B.
(CUSTOMER (letters: {!m})(#-of-stamps-needed: n))
                                        ;a customer visiting a post office
                                        ;who carries letters !m and wants n stamps.
(POST-OFFICE (customers: {!c}) (collectors: {!cl}))
                ;a post office which contains customers !c and mail collectors !cl.
```

It should be noted that a conceptual representation does not represent the identity of an actor. It only provides a description of the local state of an actor. Thus to say that an actor Q is in the state expressed by a conceptual representation (*QUEUE* (*elements:* [A B C])), an assertion of the following form is used:

(Q *is-a* (*QUEUE* (*elements:* [A B C])))

Some examples of specification using conceptual representation are given in the later sections.

Symbolic evaluation is a process which interprets a module on abstract data to demonstrate that the module satisfies its specification. Symbolic evaluation differs from ordinary evaluation in that (1) the only properties of input that can be used are the ones specified in the pre-requisites, and (2) if the symbolic evaluation of a module M encounters an invocation of some module N, the specification of N is used to continue the symbolic evaluation. The implementation of N is not used. The technique of symbolic evaluation has been studied by a number of researchers, for example Boyer and Moore (1975), Burstall and Darlington (1975), Hewitt and Smith (1975), Yonezawa (1975), King (1976).

Our method for symbolic evaluation of distributed systems is an extension of the one developed for symbolic evaluation of programs written in SIMULA-like languages (Yonezawa and Hewitt 1976). One of the main techniques we employ in symbolic evaluation is the introduction of a notion of **situations** (McCarthy and Hayes 1969). A situation is the *local* state of an actor system at a given moment. The precise definition of locality in the actor model of computation is found in Hewitt and Baker (1977). By relativizing assertions with situations, relations and assertions about states of modules in different situations can be expressed. Explicit uses of situational tags seems to be very powerful in symbolic evaluation of distributed systems. A simple example is given in Sec. 7.

Another technique we employ in symbolic evaluation is the use of *actor induction* to prove properties holding in a computation. Actor induction is a computational induction based on the *precedes* ordering (cf. Sec. 2) among events. It can be stated intuitively as follows:

> "For each event E in a computation C, if *preconditions* for E imply *preconditions* for each event E′ which is immediately caused by E, then the computation C is carried out according to the overall specification."

The precedes ordering has two kinds of suborderings, (1) the activation ordering, *"activates"*, which is the causal relation among events, and (2) the arrival ordering, *"arrives-before"*, which expresses ordering among events which have the same target actor. Thus there are two kinds of actor induction according to these suborderings. An example of the induction based on arrival ordering is used in Sec. 7.

4. MODELLING AN AIR-LINE RESERVATION SYSTEM

A specification of an air line reservation system

As an example of distributed systems, let us consider a very simple air-line reservation system. Suppose we have just one flight which has a non-negative number of seats†. A number of travel agencies (parallel processes) independently try to reserve or cancel seats for this flight, possibly concurrently. We model the air-line reservation system as a flight actor F which behaves as follows. The flight actor F accepts two kinds of message, (*reserve-a-seat:*) and (*cancel-a-seat:*). When F receives (*reserve-a-seat:*), if the number of free seats is zero, a message (*no-more-seats:*) is returned. Otherwise a message (*ok-its-reserved:*) is returned and the number of free seats is decreased by one. When F receives (*cancel-a-seat:*), if the number of free seats is less than the maximum number of seats of the flight, a message (*ok-its-cancelled:*) is returned and the number of free seats is increased by one, otherwise (*too-many-cancels:*) is returned. Furthermore, requests by (*reserve-a-seat:*) and (*cancel-a-seat:*) are served on a first-come-first-served basis.

To write a formal specification of the air-line reservation system, we need to describe the states of the flight actor. For this purpose, we use the following conceptual representation:

$$(FLIGHT\ (seats\text{-}free:\ \langle m\rangle)\ (size:\ \langle s\rangle))$$

The number of free seats is ⟨*m*⟩, and ⟨*s*⟩ is the size of the flight in terms of the total number of seats. The formal specification of the air-line reservation system using this conceptual representation is depicted in Fig. 1.

The first ⟨*event:*...⟩-clause states that a new flight actor F is created by an event where the create-flight actor receives a positive number S. ⟨Actor⟩* means that ⟨actor⟩ is newly created. The second ⟨*event:*...⟩-clause has two cases according to the number of free seats at the moment when the flight actor F receives (*reserve-a-seat:*). When the number of free seats is zero (*Case-1*), the state of F does not change. When it is positive (*Case-2*), the number of free seats decreases by one as stated by the assertion in the ⟨*next-cond:*... ⟩1-clause. The notation in Fig. 1:

⟨*event:* [[T ⟨= M]]
  ⟨*pre-cond:* ... ⟩
  ⟨*next-cond:* ... ⟨assertion⟩ ... ⟩
  ⟨*return:* ⟨actor⟩ ⟩ ⟩

means that when an event [[T ⟨= M]] takes place, if the preconditions are satisfied, ⟨assertion⟩s in the ⟨*next-cond:*...⟩-clause hold immediately after the event

†A model of air-line reservation systems which deal with more than one flight is discussed in Yonezawa (1977).

until the next message arrives at T. ⟨Actor⟩ in the ⟨*return:...*⟩-clause is returned as a result of the event. A ⟨*next-cond:...*⟩-clause differs from a ⟨*post-cond:...*⟩-clause in that assertions in a ⟨*post-cond:...*⟩-clause hold at the time ⟨actor⟩ is returned, whereas assertions in a ⟨*next-cond:...*⟩-clause hold at the time the next message arrives. The next message may arrive at T before or after a reply for the previous message is returned. The third ⟨*event:...*⟩-clause is for the cancelling event, which is interpreted in a similar way.

```
⟨event: [[create-flight ⟨= S]]
    ⟨pre-cond: |(S ⟩ 0)⟩
    ⟨return: F*⟩
    ⟨post-cond: (F is-a (FLIGHT (seats-free: S) (size: S)))⟩⟩

⟨event: [[F ⟨= (reserve-a-seat:)]]
  (case-1:
    ⟨pre-cond: (F is-a (FLIGHT (seats-free: 0) (size: S)))⟩
    ⟨next-cond: (F is-a (FLIGHT (seats-free: 0) (size: S)))⟩
    ⟨return: (no-more-seats:)⟩)
  (case-2:
    ⟨pre-cond:
      (F is-a (FLIGHT (seats-free: N) (size: S)))
      (N ⟩ 0)⟩
    ⟨next-cond: (F is-a (FLIGHT (seats-free: N − 1) (size: |S|)))⟩
    ⟨return: (ok-its-reserved:)⟩)⟩

⟨event: [[F ⟨= (cancel-a-seat:)]]
  (case-1:
    ⟨pre-cond: (F is-a (FLIGHT (seats-free: S) (size: |S|)))⟩
    ⟨next-cond: (F is-a (FLIGHT (seats-free: S) (size: S)))⟩
    ⟨return: (too-many-cancels:)⟩)
  (case-2:
    ⟨pre-cond:
      (F is-a (FLIGHT (seats-free: N) (size: S)))
      (N ⟨ S)⟩
    ⟨next-cond: (F is-a (FLIGHT (seats-free: N + 1) (size: |S|)))⟩
    ⟨return: (ok-its-cancelled:)⟩)⟩
```

Fig. 1 – A specification of the air line reservation system (a specification for the flight actor).

5. IMPLEMENTING THE AIR LINE RESERVATION SYSTEM

Our strategy for implementing the air line reservation system (specified in the previous section) is as follows. First, we implement a flight-data actor which satisfies the specification in Fig. 1.

In Fig. 2 we give an implementation of the flight-data actor in PLASMA.

```
(create-flight-data =s) ≡                ;create-flight-data receives a size s of flight.
  (create-serialised-actor (size initially s)   ;a variable size is set to s.
    (seats-free initially s)             ;a variable seats-free is set to s.
    then                                 ;the following cases-clause is
                                         ;returned as an actor which behaves as a flight-data.
    (receivers
      (≡) (reserve-a-seat:)              ;when a (reserve-...) message is received,
        (rules seats-free
          (≡) 0                          ;if seats-free is zero,
            (no-more-seats:))            ;(no-...) message is returned.
          (≡) (> 0)                      ;otherwise
            (seats-free ← (seats-free − 1))   ;seats-free is decreased by one.
            (ok-its-reserved:))))        ;(ok-...) message is returned.
      (≡) (cancel-a-seat:)               ;when a (cancel-...) message is received,
        (rules seats-free
          (≡) size                       ;if seats-free is equal to size,
            (too-many-cancels:))         ;(too-...) is returned,
          (≡) (< size)                   ;otherwise
            (seats-free ← (seats-free + 1))   ;seats-free is increased by one.
            (ok-its-cancelled:)))) ))    ;(ok-...) is returned.
```

Fig. 2 – An implementation of a flight actor.

It is fairly straightforward to write a specification for this flight F by using a conceptual representation:

(*FLIGHT* (*seats-free:* ⟨*m*⟩) (*size:* ⟨*s*⟩))

which describes the state of a flight actor. The number of free seats is ⟨*m*⟩ and ⟨*s*⟩ is the size of the flight in terms of the number of seats. Note that if F were sent more than one message concurrently, anomalous results would be caused unless we take precautions. For example, in the implementation in Fig. 2 if (*reserve-a-seat:*) and (*cancel-a-seat:*) messages are sent concurrently, a (*no-more-seats:*) message might be returned even if there are vacant seats. Therefore in order to model the air line reservation system by using the above implementation of a flight-data actor, the way it is used must be restricted so that inference between different activations may not take place. As suggested in the beginning of this section, the restriction we impose is that F must be *used serially* in the sense that F is not allowed to receive a message until the activation by the previous message is completed. Now the flight actor can be used to implement the air line reservation system under this restriction.

7. SYMBOLIC EVALUATION OF THE AIR LINE RESERVATION SYSTEM

Our implementation of the air line reservation system is expressed by the following simple PLASMA code:

```
(create-a-flight S)
```

which creates a flight which initially has the following conceptual representation:

(FLIGHT (seats-free: S)(*size:* S))

This establishes the first clause of the specification of the air line reservation system. The other clauses are established in the same way using symbolic evaluation.

8. FURTHER WORK

We are currently working to establish a coherent methodology for demonstrating that a distributed message-passing system will meet its task specifications. As an example, an actor model of a simple post office is studied in Yonezawa (1977). It is shown that the overall task specifications of the post office are implied by specifications of the individual behaviour and mutual interaction of actors in the model.

By using the technique of symbolic evaluation, we would like to analyse the relationships and dependencies between modules in a distributed system. This approach will be instrumental in assisting us with the evolutionary development of distributed systems.

We are also working on the application of procedural objects (such as actors) to the area of business automation. In order to replace paper forms and paper documents, we use "active" forms and "active" documents which are displayed on the TV terminal as images accompanied by procedures. Active forms and documents are sent from one site to another whereby clerks are requested to provide necessary information with the guidance of the accompanying procedures. Such procedures may also check the consistency of filled items and point out errors and inconsistencies to persons who are processing forms. Thus active forms and documents accompanied by procedures enormously increase the flexibility and security of message and document systems. Furthermore, we propose to use the "language" of forms and documents as the basis for the user to communicate with the information processing system. One of the ultimate objectives of our research is to develop a methodology for the construction of real-time distributed systems which can be efficiently and effectively used by non-programmers.

Note added in proof

Since this paper was written the message-passing semantics group has continued to develop the preliminary ideas in this paper. Recent results are reported in

Hewitt, C. (1978). Concurrent systems need *both* sequences and serialisers, *Working Paper 179*, Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology; Hewitt, C., Attardi, G. and Liebermann, H. (1978), *Working Paper 172*, Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology; Hewitt, C., Attardi, G. and Liebermann, H. (1978), *Working Paper 180*, Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Acknowledgements

Conversations with Jeff Rulifson have been helpful in further developing the notion of an "active" document. This research was conducted at the Artificial Intelligence Laboratory and Laboratory for Computer Science (formerly Project MAC), Massachusetts Institute of Technology, under the sponsorship of the Office of Naval Research, contract number N00014-75C0522.

BIBLIOGRAPHY

Atkinson, R. and Hewitt, C. (1977). Synchronisation in actor systems. Paper given at 4th ACM (SIGPLAN-SIGACT) Symposium on Principles of Programming Languages, Los Angeles. New York: Association for Computing Machinery (in press).

Birtwhistle, G., Dahl, O.-J., Myrhang, B. and Nygaard, K. (1973). *SIMULA Begin*. Philadelphia: Auerbach.

Bobrow, D. G. and Winograd, T. (1976). An overview of KRL, a knowledge representation language. *Cognitive Science*, **1**, No. 1, 3–46.

Boyer, R. S. and Moore, J. S. (1975). Proving theorems about LISP functions. *JACM*. 22, 129–144.

Burstall, R. M. and Darlington, J. (1975). Some transformations for developing recursive programs. *Proc. of 1975 International Conference on Reliable Software*, Los Angeles. pp. 456–472. Also (1977) *JACM*, **24**, 44–67.

Greif, I. and Hewitt, C (1975). Actor semantics of PLANNER-73. *Proc. of 2nd ACM (SIGPLAN-SIGACT) Symposium on Principles of Programming Languages*, pp. 67–77. New York: Association for Computing Machinery.

Hewitt, C. (1975). How to use what you know. *Advance Papers Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)* Tbilisi, USSR, pp. 189–198. Cambridge, Mass: Artificial Intelligence Laboratory, MIT.

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Artificial Intelligence*, **8**, 323–64. Also *AI-Memo* No. 410. Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Hewitt, C. and Baker, H. (1977). Laws for communicating parallel processes. *Proc. of IFIP-77*. Toronto. Also *Working Paper* 134. Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Hewitt, C. and Smith, B. C. (1975). Towards a programming apprentice. *IEEE Transactions on Software Engineering*, **SE-1**, No. 1, pp. 26–45.

Hoare, C. A. R. (1972). Proof of correctness of data representation. *Acta Informatica*, **2**, 271–81.

King, J. (1976). Symbolic execution and program testing. *CACM*, **19**, No. 7, 385–394.

Kuipers, B. J. (1976). A frame for frames: representing knowledge for recognition. In *Representation and Understanding* (eds. Bobrow, D. G. and Collins, A. M.), New York: Academic Press.

Learning Research Group (1976). Personal dynamic media. *SSL-76-1*. Palo Alto: Xerox Palo Alto Research Center.

Lenat, D. B. (1975). Beings: Knowledge as interacting experts. *Advance Papers of Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)* Tbilisi, pp. 126–133. Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence* 4, pp. 463–501 (eds. Meltzer, B. and Michie, D.). Edinburgh: Edinburgh University Press.

Minsky, M. (1975). A framework for representing knowledge. In *The Psychology of Computer Vision* (ed. Winston, P. H.) pp. 211–277. New York: McGraw Hill.

Rich, C. and Shrobe, H. E. (1975). Understanding LISP programs: towards a programmers apprentice. *AI-TR No. 354.* Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Steiger, R. (1974). Actor machine architecture. M.Sc. Thesis. Cambridge, Mass: Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Yonezawa, A. (1975). Meta-evaluation of actors with side-effects. *Working Paper 101,* Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Yonezawa, A. (1977). Specification and verification of programs based on message-passing semantics. Ph.D thesis, Cambridge, Mass: Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Yonezawa, A. and Hewitt, C. (1976). Symbolic evaluation using conceptual representations for programs with side-effects. *AI-Memo No. 399.* Cambridge, Mass: Artificial Intelligence Laboratory, Massachusetts Institute of Technology.