MODELLING DISTRIBUTED SYSTEMS

Akinori Yonezawa and Carl Hewitt
Artificial Intelligence Laboratory, M.I.T.
Cambridge. Mass. 02139 U.S.A.

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 systems 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 behavior 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 between 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 for 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 its application to Artificial Intelligence. In section 2 and 3, we discuss a model of distributed systems and its specification and verification techniques. We introduce a simple example of air line reservation systems in Section 4 and illustrate our specification and verification techniques for this example in the subsequent sections. Then we discuss our further work.

2. A Model of Distributed Systems

The actor model of computation[Greif&Hewitt75, Greif75, HewittfcBaker77] 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 is activated 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. 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 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 can be 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 chips can be viewed as actors.

Knowledge and intelligence can be embedded as actors in a modular and distributed fashion. For example, /frames[Minsky75, Kuipers75], units[Bobrow&-Winograd76], beings[Lenat75], stereotypes[Hewitt75] e.t.c. 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 result of the receipt of the message should be sent. By virtue of continuations in messages, the message-passing in the actor model of computation realizes a universal and yet flexible control structure without using implicit mechanisms such as push down stacks. Various forms of control structures such as go-to's, procedure calls, and coroutines can be viewed as particular patterns of message passing [Hewitt76].

This model of computation has been implemented as a programming language PLASMA[Hewitt76]. The script of an actor can be written as a PLASMA program. We believe that PLASMA will provide a basis for programming languages for distributed systems. In section 5, an example of PLASMA programs 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 behavior of the distributed system. Also we need sound techniques for demonstrating that implementations of the system meet its specification. Below we give some of the central ideas of our specification and verification techniques based on the model introduced in the previous section. The more detailed work will be found in [Yonezawa77].

In specifying the behavior 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 behavior of actors which behave like data structures[Yonezawa&Hewitt76]. We have found them very useful to express states of modules in distributed systems at varying levels of abstraction and also from various view points. The basic motivation of conceptual representations is to aid in providing a specification language which serves as a good interface between programmers and the computer and also between users and implementors. Conceptual representations are intuitive clear and easy to understand, yet their rigorous interpretations are provided. Instead of going into details of syntactic constructs of conceptual representations, we give examples. Below |<exp> is the unpack operation on <exp> which means writing out all elements denoted by <exp> individually.

(CELL A) *;a cell containing* A *at its content*.*
[QUEUE A B C) *;a queue with element** A B C .
(NODE (car. A)(c<fr; B)) *;a LISP node containing* A *and* B.
(CUSTOMER (letter: {lm])(*~of-*tompt-nccdcd:* n))
*;a cuttomer vi*iting a po*t office*
*;u>ho carriet letter** !m *and want** n **tamps.*
(POST-OFFICE (customer: {!c}) *(collector:* {!cl}))
;a pott office which contain cuttomer** gc *and mail collector** !cl.

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

(Q *it-a (QUEUE* A B O)

is used. 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 reseachers, for example [Boyer&Moore75, Bursta1l&Darlington75, Hewitt&Smith75, Yonezawa75, King76].

Our method for symbolic evaluation of distributed systems is an extention of the one developed for symbolic evaluation of programs written in SIMULA-like 1anguages[Yonezawa&Hewitt76l. One of the main techinques we employ in symbolic evaluation is the introduction of a notion of situations[McCarthy&Haves69]. A situation is the *local* state of an actor system at a give moment. The precise definition of locality in the actor model of computation is found in [Hewitt&Baker77], By relativiiing states of modules with *situational tags* which denote situations, relations and assertions about states of modules in different situations can be expressed. Explicit uses of situational tags seem to be very powerful in symbolic evaluation of distributed systems. A simple example is given in Section 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. Section 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 specifications."

The precedes ordering has two kinds of suborderings, 1) the activation ordering, *"activated,* which is the causal relation among events, and 2) the arrival ordering, *''arrive*-hefore^{M},* 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 the arrival ordering is used in Section 7.

4. Modelling an Air Line Reservation System

- A specification of an Air Line Reservation System -

As an illustrative 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 an air line reservation system as a flight actor F which behaves as follows. The flight actor F accepts two kinds of message, *(reterve-a-*cat:)* and *(cancel-o-teat:).* When F receives *(reterve-a-*cat:)$_{t}$* if the number of free seats is zero, a message *(no-more-teat*:)* is returned. Otherwise a message *(ok-itt-reterved:)* is returned and the number of free seats is decreased by one. When F receives *(cancel-a-*eat:),* if the number of free seats is less than the maximum number of seats of the flight, a message *(ok-it*-cancelled:)* is returned and the number of free seats is increased by one, otherwise *(too-many-canceIt:)* is returned. Furthermore requests by *(re*erve-a-*eat.)* and *(cancel-a-teat:)* are served on a f irst-come-first-served base.

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 (seals free

which describes the state of a flight actor. The number of free seau is <m> and <*> 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 Figure 1 below.

```
(event: (create-flight <■ S)
  (pre-cond: (S > 0) >
  (return: F* >
  (post-eond: (F is-a {FLIGHT {teatt-free: S) {tize: S)))»

(event: (F <« {reterve-a-teat:))
  {cate-h
    (pre-cond: (F is-e (FUCHT {teatt-free: 0) {tize: S)))>
    (next-cond: (F it-a 9FLIGHT {teatt-free: 0) (me: $)))>
    (return: {no-more-matt:) >)
  (caie-2:
    (pre-cond:
      (F is-a {FLIGHT {teatt-free: N) (size: S)))
      (N>0) >
    <next-cond: (F is-a {FLIGHT {teatt-free: N - 1) (size: S)))>
    <return; (ok- its- reserved.) >)>

<event: (F <■ (cancel-a-ieat:))
  {caae-l:
    (pre-cond: (F is-o {FLIGHT {teatt-free: S) (size; S)))>
    <next-cond. (F if-a {FLIGHT {teatt-free: S) (size S)))>
    (return: {too-many-cancel*:) >)
  (case-2.
    <pre-cond;
      (F is-a {FLIGHT {teatt-free: N) (size: S)))
      (N<$) >
    <next-con<f: (F it-a {FLIGHT {teatt-free: N ♦ 1) (size: S)))>
    <relurn: (o*-ili-canc«//ed:) > ) >

(for-eventt: E, E*
    whsere E=(F<= M), E' - (F = M')
  <pre-cond:
    (F is-a {FLIGHT {teatt-free: ...) (me:...)))
    (E arrivet-before E')>
  <caused-evenu: reply-for[E], rep/y-/or[E']>
  (post-cond: {reply-for[t] precedet reply-for[V]) »
```

Figure 1 A Specification of the Air Line Reservation System (A Specification for the Flight Actor)

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 *{reterve-a-teat:)*. When the number of free seats is zero *(Cate-l)*, 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-coiuf:...>~clause. The notation in Figure 1:

```
<event; (T <« M)
    (pre-cond: ... >
    (next-cond: ... <astertion> ... >
    (return: <ector> > >
```

means that when an event (T <« M) takes place, if the preconditions are satisfied, <astertion>s in the <next-con\..>-clause hold immediately after the event until the next message arrives at T. <actor> in the <resurn.\..>-clause is returned as the result of the event. *<next-cond:...>* differs from *<pott-cond:...>* in that assertions in <poii-conrf;...>-clause hold at the time <actor> is returned, whereas assertions in <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. The *(for-eventt:* ...>-clause states that requests (messages) received by the flight actor are served on the first-come-first-served base. Namely, the replying events for events E and E' take place in the same order as E and E\

5. Implementing the Air Line Reservation System

Our strategy to implement 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 Figure 1 on the condition that it is always activated serially Then we put some protecting (or scheduling) mechanism on the flight-data actor so that the protected flight-data actor may satisfy the specification of the air line reservation system

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

```
(craata-flight-data =s) zreate-flight-data receives a tize s of flight.
 {let (tize initially s)                ;a variable size it set to s.
      (teatt-free initially t)     ;a variable seats-free it net to s.hen ;the following ca
                          ;returned at an actor which behavet at a flight-data.
    (cases
      (=> {reserve-a-teat:) ;when a {reterve-...) message it received,
        (rules teats-free
          (E> 0                                  ;if seats-free it zero,
            {no-more-seats:))              ;(no-.«) mettage it returned.
          (=                 else                         ;otherwite
            (seats-free «- (teats-free - 1))
                                        ;teats-free it decreated by one.
            {ok-itt-reterved:))))           ;{ok-„) mettage it returned.
      (s> {cancel-a-teat:) ;when a (cancel-...) mettage it received,
        (rules seats-free
          (i> size                        ;if teatt-free it equal to tize,
            {too-many-cancel t:))             ;{too-...) it returned,
          (i> elte                                        ;otherwite
            (teats-free «- (teatt-free ♦ 1»
                                        ;seatt-free it increated by one.
            {ok-itt-cancelled:)))) ))               ;{ok-...) it returned.
```

Figure 2

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

{FLICHT-DATA {teatt-free: <m» *{tixe:* <t»)

which describes the state of a flight-data actor. The number of free seats is <m> and <$> is the size of the flight in terms of the number of seats. Note that if FO were sent more than one message concurrently, anomalous results would be caused. For example, in the implementation in Figure 2, if *(retserve-a-seat:)* and *(cancel-a-teat:)* messages are sent concurrently, *(no-more-teatt:)* 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 interference between different activations does not take place. As suggested in the beginning of this section, the restriction we impose is that FD must be used serially in the sense that FD is not allowed to receive a message until the activation by the previous message is completed. Now the flight-data actor can be used to implement the air line reservation system under this restriction. We give a formal specification for the flight-data actor in Figure 3 below.

```
<event: (creste>-flight-data <= S)
   <pre-cond: (S > 0)>
   <return: FD* >
   <post-cond: (FO it-a (FLIGHT-DATA (teatt-free: S) (tixe: $)))»

ievent: (FD <» (reterve-a-teat:))
   {cat*-1:
      <pre-cond:
         (FD it-uted-terially)
         (FD it-a {FLIGHT-DATA (sett-free: 0) (tize: S)))>
      <returm (no-more-teatt:) >
      <post-cond:
         (FD ii-a (FLIGHT-DATA {seots-free: 0) (tixe: $))) »
   (cate-2:
      <pre-cond:
         (FD it-uted-terially)
         (FD it-a (FLIGHT-DATA (teatt-free: N) (tixe: S»)
         (N>0) >
      <returm (ok-itt-reterved:) >
      <pott-eond:
         (FD it-a (FLIGHT-DATA (teatt-free: N - 1) (tize: S))) »>

<event: (FD <= (cancel-a-teat:))
   (cate-1:
      <pre-cond:
         (FD it-uted-terially)
         (FD it-a (FLIGHT-DATA (teatt-free: S) (tixe: S))) >
      <return: (too-many-cancelt:) >
      <post-cond:
         (FD it-a (FLIGHT-DATA (teatt-free: S) {size: S))) »
   (cate-2:
      <pre-cond:
         (FD it-uted-terially)
         (FD it-a (FLIGHT-DATA (teatt-free: N) {site: $)))
         (N < S) >
      return: (ok-itt-cancelled:) >
      <pott-eond:
         (FD it-a (FLIGHT-DATA (teatt-free: N ♦ 1) (tize: S))) »>
```

Figure- 3 A Specification for the Flight-data Actor

In this specification, the restriction of the serial use is expressed in the following notation,

(FD *it-uted-terially)*

stated as a precondition for events. In contrast to the specification above, there are no such preconditions in the specification of the air line reservation system (the flight actor) in Figure I. Thus the reservation system is specified to work properly even *if* It is accessed concurrently. Also notice that the specification above has no statements about scheduling such as the first-come-first-served scheduling which is stated as <for events:...>-clause in the specification of the air line reservation system.

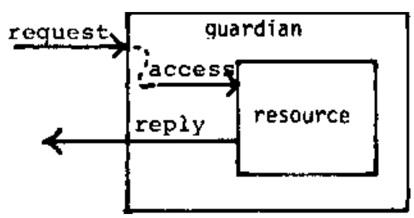
8. One-at-a-tIme

In this section, we consider how the serial use of a flight-data actor is realized in environments where communicating parallel processes try to *use* the flight-data actor. Our approach is to surround a flight-data actor FD with some mechanism which arbitrates parallel requests to the flight-data actor FO and passes these requests to FD in the serial fashion. We call this protection mechanism a one-at-a-time guardian. A ono-at-a-time- guardian can be easily implemented by a serial izer[Atkinson&Hewitt77] which is a general synchronization mechanism in the actor model of computation.

Now we give a specification for one-at-a-time guardians. A one-at-a-time guardian is created in an event where an actor one-at-a-tIme receives a resource (a flight-data actor in this case). The one-at-a-time guardian thereby created will then contain the received resource. The following <event;...>-clause expresses this.

```
<event: (one at time <■ RESOURCE)
      (return: G* >
      <post-cond: (G it-a (ONE-AT-A-TIME RESOURCE)) »
```

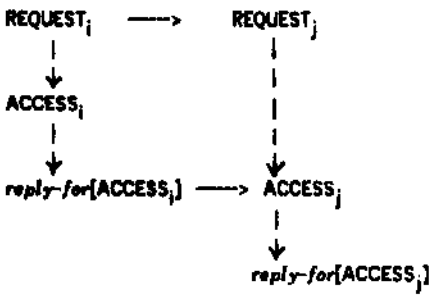
where *(ONE-AT-A-TIME* <retource>) is the conceptual representation for a one-it-a-tima guardian which contains <*resoure*. Next, we specify how a one-at-a-time guardian G behaves. In general a *request* to the guardian G, which *is* an arrival of a message M at G, eventually causes an invocation (or use) of RESOURCE. The invocation of RESOURCE begins with an access to RESOURCE which is an arrival of the same message M at RESOURCE and ends with a *reply* for the *access* which is a return of some result of the invocation. (See the figure 4 below.)



Our aim of using a one-at-a-time guardian G is to control invocations of RESOURCE by parallel requests so that only one invocation of RESOURCE takes place at a time. In order to do so, if we have two concurrent requests, the end of the invocation by one request should always precede the beginning of the invocation of the other request. This intuitive description of the desired behavior of a one-at-a-time guardian can be described in terms of the order of the events *request*, *access* and *reply* introduced above. Suppose we have two requests, REQUEST} which is an arrival of a message M_i at G, and REQUEST, which is an arrival of a message M1 at G. Then REQUEST^ causes ACCESS^ which is an arrival of Mk at RESOURCE resulting in *reply-for[*$ACCESS_K$], in this order (where K stands for either *i* or j). To ensure the one-at-a-time property of invocations of a resource, the following ordering relation must be satisfied:

> *if* REQUEST1 precedes REQUESTj,
> *then* rep/r/orfACCESSj] must precede $ACCESS_j$".

Since $REQUEST_R$ always precedes $ACCESS_K$ and $ACCESS_K$ always precedes rep/y-for[$ACCESS_K$], the desired ordering relation can be expressed by the following diagram.



This behavior of the one-at-a-tima guardian is formally described as a specification in Figure 5 below. Note that RESOURCE must be guaranteed to reply.

```
<event: (one-at-a-time <= RESOURCE)
  <return: G* >
  <post-cond: (G is-a (ONE-AT-A-TIME RESOURCE)) >>

<for-events: REQUEST_i, REQUEST_j
    where REQUEST_i = (G <= M_i),  REQUEST_j = (G <= M_j)
  <pre-cond:
    (G is-a (ONE-AT-A-TIME RESOURCE))
    (RESOURCE is-guaranteed-to-reply)
    (REQUEST_i precedes REQUEST_j) >

  <caused-events: ACCESS_i, ACCESS_j,
        reply-for[ACCESS_i], reply-for[ACCESS_j]
    where
      ACCESS_i = (RESOURCE <= M_i),   ACCESS_j = (RESOURCE <= M_j)>
  <post-cond:
    (REQUEST_i precedes ACCESS_i)
    (REQUEST_j precedes ACCESS_j)
    (ACCESS_i precedes reply-for[ACCESS_i])
    (ACCESS_j precedes reply-for[ACCESS_j])
    (reply-for[ACCESS_i] precedes ACCESS_j)>>
```

Figure 5 A Specification for the One-at-a-Time Actor

7. Symbolic Evaluation of the Air Line Reservation System

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

(craata-flijht at) a (ona-at-a-tima (craata-flijht-data «)).

(Equivalent^,
(cr^ata-flight =«) i (on«-at-a-tima <« (craata-flight-data <= s)).)

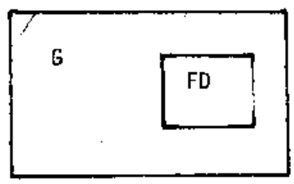
In this section we demonstrate that the above code meets the specification of the air line reservation system given in Figure 1. Our method for the demonstration is symbolic evaluation.

The symbolic evaluation of the code

(ona-at-a-time (craata-flifht-data •))

reveals the following facts:

1) an actor FD is created by (craata-flight-data <= s),
2) G is created by (one-at-a-time <= FD) and returned, and
3) the two actors satisfy the following assertions immediately after the creation of G
   (FD *i*-e (FLIGHT-DATA (trait-fret:* t) *(*ite:* s)))
   (G *i*-a (ONE-AT-A-TIME* FD)).



This means that the flight actor is created as a one-at-a-time guardian G which contains a flight-data actor FD with s free seats. In what follows, we will establish that the ona-at-a-tima guardian G satisfies the specification for the flight actor in Figure 1.

The <even:..>c1ause in the specification for the flight actor in Figure 1 specifies the behavior of G in terms of the conceptual representation

(G *it-a {FLIGHT (seats-free:...)(*ize:...)))*

(Notice that F in the specification for the flight actor is instantiated as G.) On the other hand, G is implemented as a onav-at-a-tima) guardian which contains the flight-data actor FD. This means that we have two views of G and correspondingly two different conceptual representations are used to describe the state of G. In order to show that the implementation satisfies the specification, we need to establish some relation between the state of G expressed by

(FLIGHT (seats- free:,,.) (tize:..,))

and the state of FD expresssed by

*(FLIGHT-DATA (*eat*-free:...)(*i*e:..)).*

The relation we need is:

"If G satisfies the assertion

(G *ii-a {FLIGHT beau-free:* n) *(tize:* s)))

in a situation where G receives a message M, then FD always satisfies the assertion

(FD *it-a (FLIGHT-DATA (teatt-free:* n) (size:s)))

in the situation where FD receives the same message M (through the ore-at-a-time. guardian), and vice versa."

This relation is expressed formally as follows:

<implementation-commentary:
(G is-a (*FLIGHT* (seats-free: n) (size: s))) *in* S
where S = *Sit*[(G <= M)]
if-and-only-if
(FD is-a (*FLIGHT-DATA* (seats-free: n) (size: s))) *in* S'
where S' = *Sit*[(FD <= M)] >.

Sit[E] expresses the situation where an event E takes place. The above *implementation commentary* formally describes the basic idea of the implementation. It can be viewed as the counterpart of an "invariant" in parallel process environments, which was first introduced by [Hoare 1972] to show correctness of implementation of data structures which are supposed to be used serially.

It should be noted that the first-come-first-served based scheduling by the guardian G guarantees the above relation. If the guardian does more complicated scheduling, the relation needed for the demonstration may not be so simple. For more general scheduling cases, see [Yoneiawa77].

I. Establishing the *<event:* (G *<■ (reserve-a-ieat:))...>-c\dLUse*

There are two cases to be considered. We only consider the (Case-.2...)-c1ause.

Case-2: (G is-*a [FLIGHT* (seats- free: n) *(tize:* t))), (n > 0)

The guardian G receives a *(reterve-a-neat:)* message M. To know the result of this event, the specification for one-at-a-time in Figure 5 is used. Since the flight-data actor FD is guaranteed to reply, the specification for one-at-a-time guarantees that the *(reserve-a-seat:)* message M is received by FD. To know the state of the flight-data actor FD at the time of the arrival of M, the above implementation commentary is used. Since the state of G at the time of the arrival of M at G is described as:

(G *i$-a (FLIGHT ($eatt-free:* n) *(siize:* s)).

the state of FD at the arrival of M at FD is described as

(FD *i$-a {FLIGHT-DATA (eat$-free:* n) *{size: s))).*

Then the (Case--2...)-clause in the <event.\..>-clause of the specification for flight-data actors in Figure 3 is referred to. Since the precondition that FD must be used serially is satisfied (because FD is contained inside the one-at-a-time G), the (Case-2~)-clause of the specification for flight-data actors tells us that

(1) *(ok-iu-reterved:)* is returned, and
(2) the state of FD is now expressed as:
(FD it-a *(FLIGHT-DATA (seat-free:* n - 1) *(tize:* ■))).

(1) is what the <return:..>-clause in the specification for the flight in Figure 1 requires. Since the state of FD expressed as

(FD is-a (*FLIGHT-DATA* (seat-free: n - 1) (size: s)))

remains unchanged until the next message M' arrives at FD. by using the implementation commentary in the other direction this time, we know that the state of G remains unchanged as

(G is-a (*FLIGHT* (seats-free: n - 1) (size: s)))

until the message M' arrives at G. This is what *<next-cond:..»>* clause in the specification for the flight actor in Figure 1 requires. Thus Case-2 is shown. Case-1 may be shown analogously. It should be noted that induction on the order of arrival of messages is used.

II. Establishing the *<event:* (G *<■ (cancel-a-teai:))..>-c*\ise*

The argument for this event is analogous to that of I.

III. Establishing the *<for-eventt:J><\zuie*

The event where the flight actor G receives a message means that the one-at-a-time guardian receives the same message. Suppose that M and M' arrive at G in this order. The specification for the one-at-a-time guardian specifies that M' is not received by FD until the reply from FD for M is completed. Therefore the reply to M' always takes place after the reply to M. This is what the specification requires.

IV. Establishing the Confinement of the flight-data actor FD

The discussion in I, II and III above assumes that no one can access the flight-data actor FD except through the guardian G. This assumption always holds because the flight-data actor FD created by (create-flight-data <« t) is never released outside the one-at-a-time actor.

8. Further Work

We are currently working to establish a coherent methodology for demonstrating that a distributed message-passing system will meet its specifications. By using the technique of symbolic evaluation, we would like to analyze 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 as images on the TV terminal 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 in 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.

9. 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.

10. Bibliography

Atkinson, R. and Hewitt, C. E. "Synchronization in Actor Systems" 4-th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angles, January. 1977.

Birtwistle, C, Dahl, O.-J., Myhrhang, B, and Nygaard, K. SIMULA Begin Auerbach, Phil. Pa., 1973.

Bobrow, R. S. and Winograd, T. "An Overview of KRL, a Knowledge Representation Language" CSL-76-4. Xerox Palo Alto Research Center, July, 1976.

Boyer, R.S. and Moore, J.S. "Proving Theorems about LISP Functions" JACM. Vol.22. No.l. January, 1975.

Burstall, R.M and Darlington, J "Some Transformations for Developing Recursive Functions" Proc. of International Conference on Reliable Software. Los Angles, 1975.

Greif, I. "Semantics of Communicating Parallel Processes" PhD Thesis MIT, also Technical Report TR-154. Laboratory for Computer Science (formerly Project MAC), September, 1975.

Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73" Proc. of ACM SIGPLAN-SIGACT Conference. Palo Alto, California. January, 1975.

Hewitt, C. E. "How to Use What You Know" IJCAI-75, USSR, September 1977.

Hewitt, C.E. "Viewing Control Structures as Patterns of Passing Messages" AI-MEMO No.410, Artificial Intelligence Lab. MIT, December. 1976., also to appear in the Journal of Artificial Intelligence.

Hewitt, C.E and Baker, H. "Laws for Communicating Parallel Processes" Proceeding of IFIP-77, Toronto, August. 1977, also Working Paper 134. Artificial Intelligence Laboratory MIT. December 1976.

Hewitt, C.E. and Smith, B.C. "Towards a Programming Apprentice" IEEE Transaction on Software Engineering, Vol. SE-INo. 1. March, 1975.

Hoare, C.A.R. "Proof of Correctness of Data Representation" Acta Informatica Vol. 1. pp271-281.1972

King, J. "Symbolic Execution and Program Testing" CACM Vol.19 No. 7 July 1976.

Kulpers, B. J. "A Frame for Frames: Representating Knowledge for Recognition" in D. G. Bobrow & A. M. Collins (Ed.) Representation and Understanding, New york , Academic Press, 1975.

Learning Research Group "Personal Dynamic Media" SSL-76-1. Xerox Palo Alto Research Center, April 1976

Lenat, D. B. "Beings: Knowledge as Interacting Experts" IJCAI-75. USSR, September, 1975.

McCarthy, J. and Hayes, P. "Some Philosophical Problems from the Standpoint of Artificial Intelligence" Machine Intelligence Vol.4. American Elsevier New York 1969.

Minsky, M. "A Framework for Representing Knowledge" in P. H. Winston (Ed.) The Psychology of Computer Vision, New york, McGraw-Hill, 1975.

Rich, C. and Shrobe, H.E. "Understanding Lisp Programs: Towards a Programmer's Apprentice" Masters' Thesis, Electrical Engineering and Computer Science, MIT August, 1975., also AI-TR No.354, Artificial Intelligence Laboratory, MIT.

Steiger, R. "Actor Machine Architecture" Master's Thesis Department of Electrical Engineering and Computer Science, MIT, June 1974.

Yonezawa, A. "Meta-evaluation of Actors with Side-effects" Working paper 101. Artificial Intelligence Laboratory MIT. June, 1975.

Yonezawa, A. Forthcoming Ph.D. Thesis., Department of Electrical Engineering and Computer Science. MIT. 1977.

Yonezawa, A., and Hewitt, C. "Symbolic Evaluation using Conceptual Representations for Programs with Side-effects." AI-Memo. No.399, Artificial Intelligence Laboratory, MIT. December 1976.

UNDERSTANDING AND IMPROVING LISP PROGRAMS

Harald WERTZ
Universite Paris 8
Departement d'Informatique
75012 PARIS

We are currently investigating the creation of an didactic environment for the teaching of a programming language to approximately 1500 students each year. At this effect we are constructing a robust and perspicious system : VISION & CAN & PHENARETE (GREUSSAY 1977, GOOSSENS 1977, WERTZ 1976), a system designed to help the individual apprentice in the process of developing, writing and debugging programs.

In this note we will describe some aspects of PHENARETE, the convivial part of the system : PHENARETE receives as input a student proposition of a program - which may contain as well syntactic as semantic errors - and will deliver as output her improved propositions of this program : a finite sequence of approximations.

With the help of our system, the learning of a programming language is done in a cycle :

user ^proposition ~~*" improvement and approximation ■_system_____\ PHENARETE —- propositions ~"ot -PHENARETE

Before constructing PHENARETE we observed for one year our students beginning to learn LISP, to see which kinds of errors are statiscally the most current. There we found principally five sorts of errors :

- absence of variables
- inversion of variables
- grouping errors
- absence of the conditionnal instruction
- non-representation of the termination of a computation.

Our system is particularly efficient in the detection of these kinds of errors. It doesn't try to verify the program with respect to the intentions of the programmer, but it tries to understand the programming language constructs used in the program, to see the interaction of the different parts of the program and to detect inconsistencies, with a particular emphasis on the verification of the termination of the program. In some way PHENARETE may be considered as an intelligent interpreter : she does not execute but she injerpiejjLthe program, and the result of an interpretation are some propositions (propositional interpreter). PHENARETE incorporates the programming knowledge of a programming apprentice.

Let me illustrate the reasoning involved in an interpretation of a simple LISP program (main steps only) : suppose a student has submlted the following program (the numbers are for references in the tetf).

```
.1. (DE REV (X Y)
.2.     ((REV X (CONS (CAR X) Y)))
.3.     ((NULL Y) X) )
```

First, (line 2 and 3) the grouping of the body of the function suggests that the user has omitted the COND-function call, so PHENARETE introduces (line 2-b) «COND». Then (line 2) we find as the first clause of the COND immediately a recursive call of the function REV, and following (line 3) another clause. Knowing that line 3 can never be attained during an execution of the program we can invert the two clauses. This gives :

```
.1'. (DE REV (X Y)
.2'.     (COND
.3'.         ((NULL Y) X)
.4'.         (T (REV X (CONS (CAR X) Y)))))
```

So far the surface improvements. Now a closer look at line 4' tells us that X and Y will be lists, when the function is invoked, and that the first argument (X) is inchanged and the second argument (Y) will grow longer. In line 3' the case second argument — NIL is solved, but this is the only case where the computation of REV will terminate. If Y is different of NIL, the only constructive computation done in the program is the creation of the new list argument 2. So, when hypothesising that the result of REV will be this list just created, we have to force a recursion stop. Let us try to call REV with the CDR of argument 1 :

```
(T(REV (CDR X) (CONS (CAR X) Y)))
```

but the recursion will not stop either. So let's try to introduce a supplementary test ((NULL X)). And, always under the hypothesis that Y will be the result of the computation, this gives us :

```
((NULL X) Y).
```

This will be the first proposition of PHENARETE ;

Proposition 1

```
(DE REV (X Y)
    (COND
      ((NULL Y) X)
      ((NULL X) Y)
      (T (REV (CDR X) (CONS (CAR X) Y)))))
```

But this new line is just the line 3' with X and Y inversed, so let's try without line 3'. Ok, that is also a recursive procedure which stops with the correct algorithm, so another proposition :

```
(DE REV (X Y)
    (COND
      ((NULL X) Y)
      (T (REV (CDR X) (CONS (CAR X) Y)))))
```

Finding no more possible interpretation, PHENARETE will stop, after these two propositions. During the interpretation-phase she justifies all the modifications she proposes in a way similar we did.

The system is implemented in VLISP-10 and is currently used by about 1500 students.

REFERENCES :

CHAILLOUX J., VLISP-10, Manuel de references, Dpt. Informatique. Universite Paris 8, RT-17-76, 1976
GOOSSENS D., CAN, RT-03-77, 1977
GREUSSAY P., Difference et repetition dans les systemes VISION,, RT-06-77, 1977
WERTZH.PHENARETE Springer VERLAG' GI 6 Jahrestagung,Stuttgart,