WORKING PAPER 101

META-EVALUATION OF ACTORS WITH SIDE-EFFECTS

by

AKINORI YONEZAWA

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

June 1975

Abstract

Meta-evaluation is a process which symbolically evaluates an actor and checks to see whether the actor fulfills its *contract* (specification). A formalism for writing contracts for actors with side-effects is presented. Meta-evaluation of actors with side-effets is carried out by using *situational tags* which denotes a *situation* (local state of an actor systems at the moment of the transmissions of messages). And also it is illustrated how the situational tags are used for proving the termination of the activation of actors.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-70-A-0362-0004.

INTRODUCTION

The purpose of this research is to develop a formalism which is both intuitively clear and convenient for carrying out the "meta-evaluation" [Hewitt et al 1973] of programs with side-effects based on actor concepts [Hewitt & Greif 1975].

"Meta-evaluation" is a process which symbolically evaluates a piece of code and shows whether the code fulfills its specification. (This specification needs to be represented in a good formalism which is one of the topics of our investigation.) Since meta-evaluation is expected to be a major component in a software system called the Programming Apprentice [Hewitt & Smith 1975] which assists expert programmers in various programming activities, it must provide sufficient information for answering questions about various properties of programs as well as for showing their correctness.

Meta-evaluation is closely related to the semantics of programming languages and verification or proving the correctness of programs. These topics have been extensively investigated. But all previous program verifiers have not been able to deal with programs with real side-effects because of the inadequacy of the formal systems on which these implementations were based. Although a program with side-effects can sometimes be transformed into a program without side-effects [Greif & Hewitt 1975], the transformation decreases efficiency and need several times the storage. And also there is a certain type of side-effect in communication between concurrent processes which is impossible to realize without side-effects.

Therefore the need for a formalism which is able to treat side-effects is obvious. In what follows, we will discuss the limitations of previous works on program verfication in dealing with programs with side-effects and propose a new formalism which can cope with this problem.

Before starting our discussion, we will more precisely define a side-effect.

WHAT IS A SIDE-EFFECT?

A definition of side-effect can be stated very clearly in terms of actors. Furthermore meta-evaluation is based on actor concepts. So we will begin with a brief description of actors.

An actor is a potentially active piece of knowledge (procedure) which is activated when sent a message by another actor. No actor can treat another actor as an object to be operated on; instead it can only send actors as a message to other actors. Each actor decides itself how to respond to messages sent to it. An actor can be characterized by stimuli (messages as questions) and responses (messages as answers). In this actor paradigm, the traditional concepts of procedure and data-structure are now unified. Furthermore various kinds of control mechanisms such as go-to's, procedure calls, and coroutines can be thought as particular patterns of messages passing. Thus a complete model of computation can be constructed with a system of actors.

We now define a side-effect in terms of actors:

> *An actor has a side-effect if it does not always give the same response to the same message.*

For example, an actor "random" which produces a random number when it gets a request is an actor with a side-effect. The only primitive actor with a side-effect is the "cell". A cell accepts a message which updates its contents and a message which asks about its contents. Thus a cell has a side-effect because it can give different answers to the "content?" message, depending upon what it contains at the moment. Updating a cell corresponds to an assignment statement in traditional programming languages. Cells do not make serious troubles for program verification if sharing is not involved. But as will be seen in the next section, serious problems arise when data-structures such as lists, stacks, queues, bags etc. are shared between procedures.

PREVIOUS WORK

Previous works on the implementation of program verification systems are based on essentially three different ways of defining the semantics of programming languages.

An implementation of LCF (Logic for Computable Functions) [Milner 1972] is based on the functional semantics proposed by D. Scott and C. Strachey

[1971]. They define the semantics of a program as a mathematical object, namely a function. As a result of this definition, in order to show some properties of a simple program, at first we have to find a function for which the program is supposed to represent and then show that the function has those properties. There is no easy way to deal with data-structures with side-effects in this functional semantics, although data-structures without side-effects could be expressed by giving axioms for the operations on data structures. Thus within the frame-work of LCF it is often difficult to capture many interesting and important facts which must be dealt with in meta-evaluation. There is some attempt [Cadiou & Levy 1973] to describe several properties related to parallel processes in LCF. But it has not been fully developed. And also some attempt is made to model parallel systems by applying the functional semantics [Cohen 1975]. However a verification system based on this model has not been developed and the model seems very complicated.

An automatic theorem prover for pure LISP functions [Boyer & Moore 1975] is considered to be based on the interpretive semantics. The semantics of LISP is defined as an evaluator of LISP. In this system a theorem is stated in LISP itself. For example, suppose a theorem to be proven by this system is:

(EQUAL (REVERSE (REVERSE $\underline{A}$)) $\underline{A}$).

The system tries to prove it by evaluating a definition of REVERSE symbolically. The following examples illustrate how symbolic evaluation works on some LISP functions:

(CAR $\underline{A}$) --> (CAR $\underline{A}$), (CAR (CONS $\underline{A}$ $\underline{B}$)) --> $\underline{A}$

(EQUAL $\underline{A}$ $\underline{A}$) --> T, (EQUAL $\underline{A}$ $\underline{B}$) --> NIL

(CDR ($\underline{A}$ $\underline{B}$ $\underline{C}$)) --> ($\underline{B}$ $\underline{C}$)   where $\underline{A}$ $\underline{B}$ and $\underline{C}$ are free variables.

So far as pure LISP functions are concerned, symbolic evaluation is a good tool for proving theorems for the following reasons:

1. Pure LISP functions are constructed solely by the composition of functions. (namely pure LISP is an applicative language.)
2. The parameter mechanism of LISP is call by value.
3. There are no side-effects in pure LISP.

These three facts guarantee that all information necessary for carrying out a proof are passed through as arguments (or parameters) and a returned value of each function which is an element of composition. But once the limitation of pure LISP is thrown away, namely where non-pure functions are dealt with, symbolic evaluation confronts a serious problem. Let us consider a non-pure function RPLACA. The symbolic evaluation of RPLACA could be expressed as follows:

(RPLACA ($\underline{A}$ $\underline{B}$) $\underline{C}$) --> ($\underline{C}$ $\underline{B}$)

but this description does not capture the most important fact which distinguishes RPLACA from CONS. Namely (CONS 'a 'b) creates a new dotted pair (a.b) while the result of (RPLACA '(a b) 'c) i.e. (c b) is the same dotted pair as the first argument of RPLACA. The following example illustrates the difference more clearly.

```
(SETQ x (CONS 'a 'b))        ;x becomes (a.b)
(SETQ y (CONS 'c x))         ;y becomes (c.(a.b))
(RPLACA x 'd)                ; ???
```

The real effect is, of course, that x becomes (d.b) and y becomes (c.(d.b)). But what we can expect from the symbolic evaluation is that x becomes (d.b) while y remains (c.(a.b)), because the information passed through the arguments does not reflect the fact that y is sharing the same list with x. To get around this problem, we need some device to pass more global information to a called function besides the arguments themselves.

Other program verification systems [King 1969, Deutch 1973, Igarashi London & Luckham 1973, Suzuki 1974] are based on axiomatic semantics originally proposed by R. Floyd [1967] for flow-chart like languages, and by C. A. R. Hoare [1969] for Algol-like languages. The main idea of this approach is as follows: Suppose that some assertion P holds before the execution of statement Q. Then the semantics of statement Q is defined as the strongest assertion R among those which hold after executing Q. C. A. R. Hoare uses the notation P{Q}R to express the above meaning. This way of defining semantics is quite natural for a program written in an imperative language whose structure is the juxtaposition of statements (or commands) rather than the composition of functions. The following figure illustrates how an assignment is treated in VCG [Igarashi London & Luckham 1973].

$$\frac{P \; \{ A \} \; Q(e)}{P \; \{ A \; ; \; x \leftarrow e \} \; Q(x)}$$

where A is an arbitrary statement.

This rule claims that after x is assigned the value e, valid assertions for e are also valid assertions for x. But this sort of simple substitution of x for e in Q does not work correctly if pointer or list structure is introduced. The reason is obvious, this simple substitution does not take account of shared data. Suppose x and y are sharing the number 3. After assigning 5 to x, y also has to be 5, but the above rule has no way of telling that y became 5.

R. Burstall [1971] proposed some techniques which are able to handle list processing languages by extending Floyd's proof system. He introduced the notation $(x \xrightarrow{a} y \xrightarrow{bc} nil)$ to denote the following linear list structure.

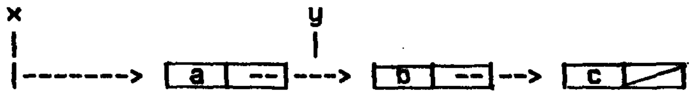

Figure 1

Although his technique is useful for statement-type list processing languages, the lack of the concept of situation which we will introduce into our formalism limits the expressive power of his notation.

QUEUES WITHOUT SIDE-EFFECTS

C. Hewitt and B. Smith [1975] succeeded in the meta-evaluation of two implementations of queues as actors without side-effects. A queue-actor is characterized as follows: a queue accepts two kinds of messages, (nq: x) which is a request to enqueue a new element x and (dq: (*else-complain-to:* the-complaint-dept)) which is a request to return the front element of the queue and the the remaining queue. However if the queue is empty, the-complaint-dept handles the situation. The essence of his implementation is that every time the message (nq: x) is sent, a new queue actor is created which contains x as the rear element. The old queue is unchanged after the operation and will respond the same way if sent (nq: x) again. Therefore it has no side-effects. We observe that this implementation of the queue uses the successive composition of actors in the same sense as the successive composition of functions. The meta-valuation of this sort of implementation can be carried out by an idea similar to the symbolic evaluation of pure LISP functions. In fact C. Hewitt and B. Smith [1975] used the following notation to express the effect of (nq: x).

(*nq* <u>a</u> *at-rear-of* (queue !<u>q</u>)) = (queue !<u>q</u> <u>a</u>)

But it should be noted that (queue !<u>q</u>) and (queue !<u>q</u> <u>a</u>) denote different actors. ! stands for the "unpack" operation (See Appendix-I.).

IMPURE QUEUES

In contrast to the queue without side-effects in the previous section, let us consider an actor with side-effects which also behaves like a queue. This actor accepts the same messages, namely (*nq*: <u>x</u>) and (*dq*: (*else-complain-to*: <u>cd</u>)), but it behaves in a different way. When it receives the (*nq*:...) message, it does not create a new queue-actor. And when it receives the (*dq*:...) message it returns its front element and <u>itself</u> as the remaining queue. Hereafter we call this actor an *impure* queue. In the following section we will give a rigorous description (i.e. a contract) of the behavior of this impure queue. An example of a concrete implementation of such an actor is given in Figure 2. (A brief explanation of the PLASMA syntax is given in Appendix-I.)

```
[(cons-impure-q !=initial-elements) ≡
   (let {[queuees = (cons-cell initial-elements)]}              ;a cell which
             ;contains initial-elements is created and denoted by queuees.
      (self ≡          ;a queue-actor is defined here and denoted by self.
        (cases
          (≡> (nq: =x)                      ;when received the enqueuing message
            (queuees <- [!$queuees x])            ;the new element x is stored
                             ;in the cell queuees with the old elements.
             self)                                          ;self is returned.
          (≡> (dq: (else-complain-to: =the-complaint-dept))
                                            ;when received the dequeuing message
            (rules $queuees
              (≡> [] (exhausted! => the-complaint-dept))
                                 ;if queuee is empty, exhausted! message is
                                                ;sent to the complaint-dept.
              (≡> [=front !=rest]                                 ;otherwise
                (queuees <- rest)       ;the contents of queuees is updated.
                (next: front (rest: self)))           ;(next:...) is returned.
                                             )) ))))]
```

Figure 2

Let us look at an example of the behavior of the above actor. Suppose Q is an actor which is created by (cons-impure-q a). If a message (*nq*: b) is sent to Q, then the cell, queuees, contains [a b], but no new actors are created. If Q receives (*dq*:...), a is sent back and the contents of queuees becomes [b], and if Q receives (*dq*:...) again, b is sent back and the contents of queuees becomes []. Thus Q has side-effects. For this implementation the notation used in [Hewitt & Smith 1975] for a queue without side-effects does not fully reflect the effect of sending the (*nq*:...) message. It does not indicate that the same actor is returned, after sending the (*nq*:...) to Q. The following example will clarify this point.

```
(let {[queue-1 = (cons-impure-q 1)]}
   (let {[queue-2 = (queue-1 <= (nq: 2))]}))
```

The effect of the above code is as follows: a cell which contains 1 is created and bound to queue-1 and then (*nq*: 2) is sent the cell and the cell is bound to queue-2. In the above example, in order to tell that the length of queue-1 is equal to that of queue-2 after the two let-statements, we have to know that queue-1 and queue-2 refer to the same actor. This would not be the case if (cons-impure-q ...) made a queue without side-effects.

Events and Situations

As been discussed in the preceding sections, in order to be able to deal with side-effects, we need some device to describe the local state of the world concerned at a given moment. Since our meta-evaluation is carried out on an actor system, we are interested in the state of the world at the time of message transmissions. I. Greif and C. Hewitt [1975] introduced a notion of event for the purpose of defining their behavioral semantics. An event consists of a target actor t, an envelope actor m, an activator ac, and an event counter ec with respect to ac. Since we are primarily concerned with an actor system without parallelism [Greif 1975], we will not consider activators. Furthermore we will not need to introduce event counters into our formalism initially. An event is defined as a transmission of an envelope actor m to a target actor t which will sometimes be denoted by the notation (t <= m) borrowed from the PLASMA synatx. A situation S can now be defined as the local state of an actor system at the moment an event E occurs. In general the complete description of the state of an actor system is not only impossible, but irrelevant. So a situation S will be used as a tag for referring to a moment of a transmission to state fragmental assertions which are true at the moment. The following examples illustrate how the situational tags are used.

(*length* a-queue$_S$) = 8, ;*the length of a-queue in a situation S is 8.*
((t <= m)) *in* S_e), ;*the event (t <= m) occurs in a situation S_e.*
(*content* a-cell$_S$) = 1984;*the content of a-cell in a situation S is 1984.*

If we are to state some relations between facts which hold at different situations --for example, a certain order relation for showing the termination of a program --, the concept of situations is quite powerful.

A CONTRACT FOR IMPURE QUEUES

Now we will illustrate how a *contract* for impure queues is written in our formalism. We use the term "contract" instead of "specification" to emphasize the fact that it is an agreement between the implementer of a module and users of the module. In meta-evaluation of an actor we are checking to see that it meets its contracts.

The first thing we have to state in the contract is how an actor which behaves as a queue with side-effects is created. We state it in our formalism as follows (note that in the contract variables prefixed with "=" are pattern variables or formal arguments as in the PLASMA syntax and that underscored variables denote free variables.)

```
((cons-impure-q !=a) creates-an-actor
                   Q  where {(Q is (Impure-queue !a))})
```

Namely an actor Q is created by (cons-impure-q !=a) and the property that Q is a queue with queuees !a is expressed in the notation (Q *is* (*Impure-queue* !a)). As will be seen later, this notation is also used as assertions in the date base for the meta-evaluation.

The next thing to state in the contract is how the actor Q responds to the (*nq*:...) and (*dq*:...) messages. The important idea is, as we pointed out earlier, that these messages do not cause the creation of new actors, but rather that they cause the behavior of Q to change. For the (*nq*:...) message, we express its response as follows.

```
(to-simplify
    ((Q <= (nq: =x))  where {(Q is (Impure-queue !b))})
  try-using
    (Q where {(Q is (Impure-queue !b x))}))
```

This notation claims that if an event (Q <= (*nq*: =x)) happens in a situation where (Q *is* (*Impure-queue* !b)) holds, then in the succeeding situation the actor Q is returned and (Q *is* (*Impure-queue* !b x)) holds. (*Impure-queue* !b x) indicates that a new element x is enqueued at the rear of the previous queuees !b. It should be pointed out that the notion of situations is not explicitly introduced into the contract; instead *where*-clauses are used. But in the process of the meta-evaluation the notion of situations is indispensable.

For the (*dq*:...) message the response is slightly complicated, because it depends on whether Q is empty or not. So we must split the cases. For this purpose we introduced an (*either* (*consider* ...)) clause as below.

```
(to-simplify
    (Q <= (dq: (else-complain-to: =cd)))
  try-using
   (either
     (consider {(Q is (Impure-queue))}
       (then:
         (exhausted! => cd)))
     (consider {(Q is (Impure-queue x !c))}
       (then:
         ((next: x (rest: Q)) where {(Q is (Impure-queue !c))})))) ))
```

Suppose that (Q <= (*dq*: (*else-complain-to*: ...)) happens in a certain situation and if Q is not empty, namely (Q *is* (*Impure-queue* x !c)) holds in the situation, then (*next*: x (*rest*: Q)) should be returned in the next situation. For the case where Q is empty, namely (Q *is* (*Impure-queue*)) holds, (exhausted! => cd) should happen in the next situation. By not stating the property of Q in the new situation we implicitly assume that the property of Q which held in the previous situation still holds.

The whole contract for an impure queue is depicted in Figure 3. One might be encouraged to compare the code in Figure 2 and this contract. In Appendix II a contract for a cell actor in the same formalism is given.

```
[contract-for impure-queue ≡

  (((cons-impure-q !=a) creates-an-actor
                Q  where {(Q is (Impure-queue !a)}))

   (to-simplify
      ((Q <= (nq: =x))  where {(Q is (Impure-queue !b)}))
     try-using
      (Q where {(Q is (Impure-queue !b x)})))

   (to-simplify
      (Q <= (dq: (else-complain-to: =cd)))
     try-using
      (either
         (consider ((Q is (Impure-queue)))
           (then:
             (exhausted! => cd)))
         (consider ((Q is (Impure-queue y !c)))
           (then:
             ((next: y (rest: Q)) where {(Q is (Impure-queue !c)}))))) ))))]
```

Figure 3

The CODE AND CONTRACT FOR (EMPTY QUEUE-1 INTO QUEUE-2)

In this section we will give the code and contract for an actor which is supposed to transfer the queuees in one impure queue to another impure queue. This code and contract will be used to illustrate meta-evaluation in the next section. This time we present the contract for this actor

(Figure 4) before presenting its concrete implementation because the contract clearly states what this actor is supposed to do.

```
[contract-for (empty ... into ...)  ≡

  ((to-simplify
      ((empty Q1 into Q2)
             where((Q1 is (Impure-queue !q1))
                   (Q2 is (Impure-queue !q2))
                   (Q1 not-eq Q2)))
   try-using
      ((done: (emptied: Q1)(extended:Q2))
             where((Q1 is (Impure-queue))
                   (Q2 is (Impure-queue !q2 !q1)))))]
```

Figure 4

Figure 5 shows an implementation of this actor which is written not directly in terms of passing messages. To facilitate its readability we adopt extended syntax in which enqueuing and dequeuing look like operations on the queue actor rather than the transmission of (*dq:*...) and (*nq:*...) messages to it. The effect of such operations are easily translated into the standard form of actor message passing. For example, in the case of enqueuing, the translation is as follows.

```
  [(nq =x at-rear-of =the-queue ≡
                     (the-queue <= (nq: x)]
```

Furthermore in order to impose a certain constraint on the types of incoming actors, a new syntactic device (<pattern> *is-a* (<type>)) is introduced. For example (=q1 *is-a* (*impure-queue* ...)) requires that the type of actors which are bound to q1 should be ***impure-queue*** (i.e. a queue with side-effects). It should be pointed out that the implementation in Figure 5 crucially depends on the fact that queue actors referred by q1 and q2 have side-effects. Suppose that these queue actors had no side-effects. Everytime (*dq*:...) or (*nq*:...) messages are sent, a new actor would be created but q1 and q2 would still refer to the same queue actors as they originally referred. Therefore after completing of the evaluation of (*empty* q1 *into* q2), completely new actors would be returned as (*done*: (*emptied*: q1') (*extended*: q2')) and the original actors referred by q1 and q2 would remain intact. This violates the contract in Figure 4.

```
[(empty (=q1  is-a (impure-queue ...)) into             ;two queue-actors
        (=q2  is-a (impure-queue ...))     ≡  ;with side-effects are sent
                                                   ;and bound to q1 and q2.
    (dq q1                           ;the dequeuing message is sent to q1.
       (next-to:                                        ;if q1 is not empty
          (≡> (next: =front-q1)              ;the front elememt of q1 and
                (rest: =dequeued-q1))         ;remained queue are received
                               ;and bound to front-q1 and dequeued-q1.
             (nq front-q1 at-rear-of q2)            ;front-q1 is enqueued.
             (empty q1 into q2)))            ;q1 and q2 are sent to empty.
       (else-complain-to:                                   ;if q1 is empty
          (≡> exhausted!                 ;exhausted! message is received
             (done:                                        ;emptied q1 and
                (emptied: q1)                  ;extended q2 are returned.
                (extended: q2)) )) )]
```

Figure 5

META-EVALUATION OF (EMPTY QUEUE-1 INTO QUEUE-2)

Meta-evaluation is a process which abstractly evaluates actors on abstract data and checks to see whether the actors meet their contracts. As briefly mentioned before, a contract is a kind of summary or advertisement of a program for those who use it as a module in writing a larger program. The meta-evaluation of a larger program should be carried out by using only the contracts of its modules instead of being bothered by the gorry details of the implementations of these modules. Of course every program should have an explicit contract. The modularity of contracts should reflect the modularity of programs. We will get some flavor of such modularity in the meta-evaluation given below of the actor (*empty* ...*into*...).

In general we assume that the meta-evaluator has a large uniform data base (i.e. without the context mechanism of QA4 or Conniver) in which assertions are made. If some assertions hold in a particular situation, they are asserted in the data base with tags which indicate the situations where they hold. Now let us consider the meta-evaluation of (*empty* ...*into*...) actor as an illustrative example.

In order to aid the meta-evaluation process the augmented code for (*empty* ...*into*...) shown in Figure 6 is given to the meta-evaluator. (Actually this augmentation of the code may be done in the interactive mode between users and the meta-evaluator.) The large capital letter $S_{...}$ between the lines denotes the situations in which events occur. It will be used as a situational tag for assertions in the data-base.

```
- $S_{initial}$ -

[(empty (=q1 is-a (impure-queue ...)) into
        (=q2 is-a (impure-queue ...)) )        =

  - $S_{dq}$ -

    (dq q1
        (next-to:

            - $S_{next-0}$ -

            (=> (next: =front-q1
                    (rest: =dequeued-q1))

                - $S_{next-1}$ -

                (nq front-q1 at-rear-of q2)

                - $S_{next-2}$ -

                (empty q1 into q2)))
        (else-complain-to:

            -$S_{else-0}$-

            (=> exhausted!

                - $S_{else-1}$ -

                (done:
                  (emptied: q1)
                  (extended: q2)) )) )]
```

Figure 6

For example, the $S_{initial}$ at the top of Figure 6 denotes the situation in which the transmission of two impure queues to (*empty...into...*) occurs and the $S_{next-\theta}$ denotes the situation in which the transmission of (*next:* actor-1 (*rest:* actor-2)) to the continuation of the dequeuing message to q1 occurs.

In what follows a detailed demonstration of the meta-evaluation of the augmented code cited in Figure 6 against the contract for (*empty...into...*) in Figure 4 is shown. The contract for impure-queue in Figure 3 is used extensively. For the convenience of explanation the situations are described as a collection of assertions instead being used as tags.

First, by reading the contract of (*empty...into...*) in Figure 3 the meta-evaluator asserts the following assertions in the data base. Q1, Q2, x1 and x2 are newly generated identifiers because they correspond to free variables underscored in the contract.

$S_{initial}$ ≡
```
      {(Q1 is (Impure-queue !x1)) (Q2 is (Impure-queue !x2))
       (Q1 not-eq Q2)}
```

After actors Q1 and Q2 are sent to (*empty...into...*) and the pattern matching is performed, Q1 and Q2 are bound to identifiers q1 and q2, respectively. Such binding of actors to identifiers is generally expressed by an assertion of the form (<identifier> ≡ <actor>).

S_{dq} ≡
```
      {(q1 ≡ Q1) (q2 ≡ Q2)
       (Q1 is (Impure-queue !x1)) (Q2 is (Impure-queue !x2))
       (Q1 not-eq Q2)}
```

Then the dequeuing message is sent to the actor bound to q1 in S_{dq}. By interpreting the (*to-simplify...*)-clause for dequeuing in the contract in Figure 3 the meta-evaluator considers two cases, namely one case where q1 is empty and the other case where q1 is not empty. Corresponding to these two cases, two different situations, $S_{next-\theta}$ and $S_{else-\theta}$, are considered as the next situation of S_{dq}. For $S_{else-\theta}$, the meta-evaluator asserts the following assertions.

$S_{\text{else-0}}$ =

 {(x1 = []) (Q1 *is* (*Impure-queue* !x1))
 (q1 ≡ Q1) (q2 ≡ Q2)
 (Q2 *is* (*Impure-queue* !x2))
 (Q1 *not-eq* Q2)}

Now the exhausted! message is sent to the complaint department. But since no binding of actors occurs, the next situation is the same as $S_{\text{else-0}}$.

$S_{\text{else-1}} = S_{\text{else-0}}$.

Then in $S_{\text{else-1}}$ the transmission of (*done*: (*emptied*: Q1)(*extended*: Q2)) to the implicit continuation in the original message to (***empty...into...***) occurs. Note that we have used the assertions (q1 ≡ Q1) (q2 ≡ Q2). It is easily seen that what the contract of (*empty...into...*) in Figure 4 requires, namely:

(Q1 *is* (*Impure-queue*))
(Q2 *is* (*Impure-queue* !x2 !x1))

are satisfied by using knowledge about the sequences (See Appendix I for PLASMA syntax):

[!x2 !x1] is equal to [!x2] if x1 is equal to [].

So the case where q1 is empty is done.

For the other case, the meta-evaluator asserts the following assertions with a tag $S_{\text{next-0}}$ where z1 and z2 are newly generated identifiers.

$S_{\text{next-0}}$

 {(x1 = [z1 !z2]) (Q1 *is* (*Impure-queue* !z2))
 (q1 ≡ Q1) (q2 ≡ Q2) (Q1 *not-eq* Q2)
 (Q2 *is* (*Impure-queue* !x2))}

In $S_{\text{next-0}}$, (*next*: z1 (*rest*: Q1)) is transmitted and the pattern matching is performed. So the meta-evaluator asserts the binding information with a tag $S_{\text{next-1}}$.

S_{next-1} =

```
    {(front-q1 = z1) (dequeued-q1 = Q1)
     (x1 = [z1 !z2]) (Q1 is (Impure-queue !z2))
     (q1 = Q1) (q2 = Q2) (Q1 not-eq Q2)
     (Q2 is (Impure-queue !x2))}
```

The (*nq*: z1) message is sent to Q2 in S_{next-1}. By using the (*to-simplify*: ...)-clause for the enqueuing message in the contract in Figure 3, the meta-evaluator asserts the following assertions with a tag S_{next-2}. Note that the crucial fact is that Q1 and Q2 are distinct impure queues.

S_{next-2} =

```
    {(Q2 is (Impure-queue !x2 z1))
     (front-q1 = z1) (dequeued-q1 = Q1)
     (x1 = [z1 !z2]) (Q1 is (Impure-queue !z2))
     (q1 = Q1) (q2 = Q2) (Q1 not-eq Q2)}
```

Now the meta-evaluator encounters the transmission of Q1 and Q2 to (*empty...into...*) in S_{next-2}. Then in order to know the behavior of the (*empty...into...*), its contract is refered to. The contract gives:

```
    (done: (emptied: Q1) (extended: Q2)) is returned !!!

       where  (Q1 is (Impure-queue))
              (Q2 is (Impure-queue ![!x2 z1] !z2)).
```

Again using knowledge about the sequences:

[![!x2 z1] !z2] is equal to [!x2 !x1] if x1 is equal to [z1 !z2], which holds in S_{next-2}.

the meta-evaluator claims that

```
    (Q1 is (Impure-queue)) and
    (Q2 is (Impure-queue !x2 !x1)) also hold for this case.
```

Since the requirements stated in the contact for (*empty...into...*) are satisfied for both cases, we conclude that the implementation of (*empty...into...*) in Figure 5 is guaranteed to meet its contract in Figure 4. In fact the justification of this conclusion is essentially based on induction on the sequence, namely the first case corresponds to the induction base and the second case corresponds to the induction step and the contract for (*empty*...*into*...) is used as an induction hypothesis. Note that all of these conditions hold when control passes through the situation. There is no guarantee that the situation will ever be reached. The demonstration of convergence is another part of meta-evaluation which is treated it in the next section.

CONVERGENCE OF (EMPTY ...INTO...)

In this section we focus our attention on the convergence of (*empty* ...*into*...) in Figure 5 as a special case of the more general concept of the convergence defined in terms of events (For this general definition and a general proof technique for the convergence see Appendix III). In the following discussion we will not distinguish the identifiers q1 or q2 in Figure 5 from the queue-actors which are bound to q1 or q2.

We can claim that the activation of (*empty* ...*into*...) always converges, if for any q1 and q2 (*done*: (*emptied*: q1) (*extended*: q2)) is always returned, provided that the pre-requisites of (*empty* ...*into*...) in the contract are satisfied. I.e. that q1 and q2 are both impure queues and not the same actor. In showing the convergence of (*empty* ...*into*...), it is enough to check that the number of the messages sent to (*empty* ...*into*...) in S_{next-2} is bounded. In fact, the number of such transmissions corresponds to the number of elements contained in q1 (i.e. the length of the queue) at the moment where the two queues are sent to (*empty* ...*into*...) in $S_{initial}$. So the number of the messages is bounded by the length of q1. What should be done here is just to present a more formal and explicit account for this correspondence. Our technique is to show that the length of q1 in S_{next-2} is <u>strictly</u> less than the length of q1 in S_{dq}.

We believe that programmers have an idea why the code they write should terminate, and that it should be explicitly stated in the contract. In the case of (*empty* ...*into*...), a clause for the convergence like:

```
(to-show-convergence:
        ((ordering: less-than) in (domain: (length-of Q1))))
```

should be put in the contract in Figure 4. A definition or characterization of (*length-of* ...) should be given by the programmer if the meta-evaluator does not know it. And to aid the meta-evaluator in demonstrating the convergence, the following (*Intention:*...)-statement is inserted just after -S_{next-2}- in Figure 6.

(*Intention:*
    ((*length-of* $q1_{S_{next-2}}$) less-than (*length-of* $q1_{S_{dq}}$)))

In general (*Intention:*...)-statements serves as formal statements about what is intended to be true at the places in the code where they are inserted [Goldstine & von Neumann 1963, Naur 1966, Floyd 1967, Hoare 1969]. Here we use them as an aid for showing the convergence.

An actual demonstration of the convergence by the meta-evaluator depends upon the formalisms adopted for the definition of (*length-of* ...). So rather than going through the formal details, we restrict ourselves to stating the essential facts used in the demonstration. These facts are:

(*length-of* $q1_{S_{next-2}}$) is the *length* of !z2.
(*length-of* $q1_{S_{dq}}$) is the *length* of !x1.
x1 is equal to [z1 !z2] in S_{next-2}.

(The definition of the *length* of a "sequence" is given in the simplification plans in Figure 8.) Before we leave this section it should be pointed out that the whole argument on the convergence of (*empty*...*into*...) relies on the pre-requisite that Q1 and Q2 are distinct.

A CONTRACT FOR "AVERAGE"

Let us consider how a contract for an actor whose behavior depends upon the history of incoming messages is written in our formalism. Obviously such actors have side-effects. An example of actors of this type is the "average" actor. It receives a (new-elements: $\underline{x}$) message which enters a number x into the data base, and a message average? which asks for the average element of all the numbers currently in the data base. Figure 7 below is a contract for this actor.

```
[contract-for average ≡

  (((average =initial-element) creates-an-actor D
         where {(D has (History initial-element))})

   (to-simplify
       ((D <= (new-element: =x)) where {(D has (History !a))})
     try-using
       (D where {(D has (History !a x))}))

   (to-simplify
       ((D <= average?) where {(D has (History !b))})
     try-using
       (average !b)))]
```

Figure 7

The idea is simple. We introduced a property that the actor q has the history !a and expressed it in the notation (D ***has*** (***History*** !a)). This idea is similar to that of M. Clint[1973] who introduced a "mythical pushdown stack" to have the history recorded. The definition or characterization of the notation (***average*** !b) used in the contract should be given together with the contract. A characterization of (***average*** ...) in the form of the simplification-plan will be found in Figure 8. One might be invited to meta-evaluate an implementation of "average" in Figure 9 against the contract in Figure 7.

```
(to-simplify (length []) try-using  0 )

(to-simplify (length x !y) try-using (1 + (length !y)))

(to-simplify  (average !x) try-using ((sigma !x)/(length !x)))

(to-simplify (sigma []) try-using  0)

(to-simplify (sigma x !y) try-using (x + (sigma !y)))
```

Figure 8

```
[(average =initial-elements) ≡
   (let
     {[current-average = (cons-cell initial-elememt)]          ;a cell which
         ;contains initial-element is created and bound to current-average.
      [counter = (cons-cell 1)]}
     (self ≡                  ;the following case-clause is defined as self.
         (cases
           (≡> (new-element: =x)                  ;when received a new data x
             (counter ← ($counter + 1))          ;counter is incremented by 1
             (current-average            ;the current average is calculated
                 ← (($current-average * ($counter - 1) + x)/$counter))
                                       ;and store in the cell current-average.
             self)                                          ;self is returned.
           (≡> average?                               ;when received average?.
             $current-average)            ;the content of current-average.
                                                             ; is returned.
                                  ))))]
```

Figure 9

FURTHER WORK

Using the "queues" and "average" as examples we have discussed the meta-evaluation of actors with side-effects. It is rather straightforward to apply our techniques to other types of actors with side-effects such as stacks, sets, bags, tables, lists and trees.

One of the contributions of our work done so far is an explicit introduction of the notion of situations in the context of meta-evaluation. The successful meta-evaluation of actors with side-effects and the demonstration of the convergence crucially depends on the use of situational tags which explicitly denote situations. As an extention of our work, we would like to develop the idea of using the notion of situations more thoroughly. In what follows, we propose three more sophisticated examples of domains where the idea is expected to be successfully extended.

We plan to construct a Programming Apprentice [Hewitt & Smith 1975] which will aid expert programmers in various activities such as debugging, maintenance, and program understanding [Rich & Shrobe 1974] in large software construction. In these activities one of the essential kinds of information required is the dependency between or within modules. For example, suppose that a certain module in a large system is changed or replaced by another module. In order to know what kinds of changes will appear in the overall behavior of the whole system, we must have precise information about the dependency between modules. We will pursue the development of a formalism in which these dependencies can be easily described using the notion of situations.

Recently several garbage collection algorithms using parallel processing have been proposed [Steel 1975, Dijkstra 1975]. All the currently used garbage collection algorithms assume that when a garbage collector is running, no other programs operate on the whole storage area being garbage collected. The proposed algorithms remove this restriction. Namely the garbage collector and other programs can be running concurrently and working on the same storage area. Since a precise formulation of the required properties for such a parallel garbage collector does not exist yet, we will first try to write its contract. We then hope to meta-evaluate implementations of these proposed algorithms using the notion of situations.

The third example we plan to pursue is the problem of writing a specification for a time-sharing file system. An intuitive description of the specification is that no two files should attempt to use the same disk track and that the track usage table should be consistent with the users file directories. This problem was originally raised in [Hewitt & Smith 1975] as an example of a specification which is difficult to express in declarative languages such as the first order logic while it is fairly easy to give a procedural specification. We will try to formulate this problem using the notion of situations in the hope to clarify the kinds of specifications that can be used for such problems.

ACKNOWLEDGEMENT

I would like to express my deep appreciation to Carl Hewitt who suggested this topic, pruned irrelevant branches which I climbed or was about to climb during this research, and carefully read drafts of this paper. Thanks are also due to Keith Nishihara who made comments and has been correcting my Engilsh.

Bibliography

Boyer, R.S. and Moore, J.S. "Proving Theorems about LISP Functions" JACM. 22. 1. January, 1975.

Burstall, R.M. "Some Techinques for Proving Correctness of Programs Which Alter Data Structures" Machine Intelligence 7. 1972.

Cadiou, J.M. and Levy, J.J. "Mechanizable Proofs about Parallel Processes" IEEE Conference record of 14th Annual Symposium on Switching and Automata Theory. 1973.

Clint, M. "Program proving: Coroutines" Acta Informatica 2. 1973.

Cohen, E.S. "A Semantic Model for Parallel Systems with Scheduling" Proc. of ACM SIGPLAN-SIGACT Conference Palo Alto, California, January 1975.

Deutch, L.P. "An Interactive Program Verifier" Ph.D Thesis. University of California at Berkeley. June, 1973.

Dijkstra, E.W. "A Parallel Garbage Collector" Unpublished Memo 1975.

Floyd, R.W. "Assigning Meaning to Programs" Mathematical Aspect of Computer Science. J.T.Schwartz (ed.) Vol.19. Am.Math.Soc. Providence Rhode Island. 1967.

Goldstine, H.R. and von Neumann, J. "Planning and coding problems for electronic computer instrument". Collected Works of John von Neumann. Macmillan. New York 1963.

Greif, I. "Semantics of Communicating Parallel Processes" Ph.D Thesis MIT Forthcoming 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. "A PLASMA PRIMER" MIT AI Lab. Working Paper in preparation. 1975.

Hewitt, C.E et.al. "Actor Induction and Meta-evaluation" Conference Record of ACM Symposium on Principles of Programming Lamguages. Boston. October, 1973.

Hewitt, C.E. and Smith, B.C. "Towards a Programming Apprentice" IEEE Transaction on Software Engineering, Vol. SE-1 No. 1. March, 1975.

Hoare, C.A.R. "An Axiomatic Basis for Computer Programming" CACM 12, October, 1969.

Igarashi, S., London, R.L.,and Luckham, D.C. "Automatic Program Verification I: A Logical Basis and Implementation" Stanford A.I. Memo.200. 1973.

King, J. "A Program Verifier" Ph.D Thesis. Carnegie-Mellon University. 1969.

McCarthy, J. "A Basis for a Mathematical Theory of Computation" Computer Programming and Formal System. North Holland. Amsterdam. 1963.

Milner, R. "Implementation and Applications of Scott's Logic for Computable Function" Proc. of ACM Conference on Proving Assertions about Programs, New Mexico. January, 1972.

Naur, P. "Proof of algorithms by general snapshots" BIT, Vol.6, No.4 1966.

Rich, C. and Shrobe, H.E. "Understanding Lisp Programs: Towards a Programmer's Apprentice" MIT AI Lab. Working Paper 82. December 1974.

Scott, D. and Strachey, C. "Toward a Mathematical Semantics for Computer Languages" Oxford University Computing Laboratory. Technical Monograph PRG-6. August, 1971.

Steel, G.L. "Multiprocessing Compactifying Garbage Collection" CACM forthcoming.

Suzuki, N. "Automatic Program Verification II: Verifying Programs by Algebraic and Logical Reduction" Stanford A.I. Memo.255 December, 1974.

APPENDIX-I

A SHORT TOUR ON PLASMA SYNTAX

For the sake of self-containedness of this paper the minimum explanation of PLASMA syntax which is sufficient for understanding of codes cited in the main content is given below. The most complete explanation for PLASMA is found in A PLASMA PRIMER [Hewitt 1975]. The meta-syntactic variables are underlined.

<Sequences>

[$\underline{a1}$ $\underline{a2}$ ... $\underline{an}$] is an expression which creates an actor called "sequence". It is an ordered sequence of actors $\underline{a1}$,$\underline{a2}$,...$\underline{an}$. [] is also a sequence called the "empty sequence". If a sequence receives a number as a message, say k, it will returns its k-th element $\underline{ak}$.

<Cells>

(cons-cell $\underline{a}$) is an expression which creates an actor called a "cell" which contains an actor $\underline{a}$. If a cell receives a message "content?", it returns its content and if it receives a message [← $\underline{b}$], it replaces its current content by an actor $\underline{b}$. $c is an abbreviation of sending content? message to c where c denotes a cell.

<Transmitters>

($\underline{T}$ <= $\underline{M}$) and ($\underline{T}$ => $\underline{M}$) are equivalent expressions called "transmitters". When a transmitter is evaluated, a message actor M is sent to a target actor T. The following expressions are abbreviations.

($\underline{T}$ $\underline{M}$) is equivalent to ($\underline{T}$ <= $\underline{M}$).
($\underline{E1}$ $\underline{E2}$ ... $\underline{En}$) is equivalent to ($\underline{E1}$ <= [$\underline{E2}$ ... $\underline{En}$]).

For example

```
(cons-cell)  is equivalent to (cons-cell []).
(factorial 3)  is equivalent to (factorial <= 3).
(a-cell ← 1984)  is equivalent to (a-cell <= [← 1984]).
```

<Receivers>

(≡> pattern body) is an expression actor called a "receiver". If a receiver is sent a message which matches pattern, an evaluation of body will start in an environment resulting from the pattern match. For example,

((≡> =n (factorial n)) <= 4) returns 24.

=n is an example of patterns and a prefix = is an actor which binds a message actor to an identifier. So in the above example 4 is bound to n.

[=x a =y b c] is an example of patterns which expects a sequence of 5 elements whose 2nd, 4th and 5th elements are equal to a, b, and c, respectively.

(≡> [=x1 ... =xn] body)

is analogous to LISP form

(lambda (x1 ... xn) body).

<Package>

(packagename: a-content) is an expression which creates an actor called a "package". A package is considered as an actor which attaches a name packagename to an actor a-content. When a package is used in a message or a pattern, we do not have to worry about the order of components in a message or pattern simply because a packagename stands for a selector of a component. And also some components can be optional.
For example, suppose

```
(dq: (next-to: =continuation)
     (else-complain-to: =complaint-dept))
```

is used as a pattern, then

```
(dq: (else-complain-to: (≡> pattern-e body-e))
     (next-to: (≡> pattern-t body-t)))
```

is a package which matches the above pattern and it also matches against the following patterns.

```
(dq: )
(dq: (next-to: =continuation))
(dq: (else-complain-to: =complaint-dept))
```

<Unpack>

There is an operator called "unpack" which manipulates linear data-structures. An unpack is abbreviated as an exclamation mark.
For example,

If x is bound to [9 8],then [1 !x 4] evaluates to [1 9 8 4].
The following analogies to LISP functions might help.

[x y z]	is analogous to (LIST X Y Z).
[x !y]	is analogous to (CONS X Y).
[!x y]	is analogous to (APPEND X (LIST Y)).
[!x !y]	is analogous to (APPEND X Y).

An unpack is extremely useful in a pattern. If [1 9 8 4] is matched with the following patterns,

[1 !=x],	x is bound to [9 8 4];
[!=y 8 4],	y is bound to [1 9];
[!=z],	z is bound to [1 9 8 4];
[!=x 8 !=y],	x and y are bound to [1 9] and [4], respectively.
[=x !=y],	x and y are bound to 1 and [9 8 4], respectively.
[!=x =y],	x and y are bound to [1 9 8] and 4, respectively.

<Conditionals>

```
(cases
  (=> pattern_1 body_1)
  (=> pattern_2 body_2)
        .
        .
  (=> pattern_n body_n))
```

is an expression called a "case-statement". When an case-statement is sent a message m, if m matches against <u>pattern</u>$_1$, then <u>body</u>$_1$ is evaluated, and if not, the next pattern <u>pattern</u>$_2$ is tried and so on.
For example,

```
(1984 =>
  (cases
    (=> 1776 independence)
    (=> 1976 bicentennial)
    (=> 1984 hate-year)))          will return hate-year.
```

```
(rules expression
   (≡> pattern_1 body_1)
            .
            .
   (≡> pattern_n body_n))
```

is an abbreviation for

```
(expression =>
   (cases
      (≡> pattern_1 body_1)
               .
               .
      (≡> pattern_n body_n))).
```

<Definitions>

The simplest way of defining an actor is:

```
[name ≡ definition]
```

which means that name is the name of the procedural call-by-name fixed point of definition. Note name may occur in definition. For example,

```
[factorial ≡
   (≡> =n
     (rules n
       (≡> [0] 1)
       (≡> ? (n * (factorial n-1)))]
```

is a definition of factorial where ? denotes a pattern which gets matched against any messages.

<Labels>

```
(labels
   {[name_1 ≡ definition_1]
            .
            .
    [name_n ≡ definition_n]}
    body)
```

is an expression which creates an actor with the following behavior. When it is evaluated, body is evaluated in an environment with each name defined as definition.

For example,

```
(labels
   {[factorial ≡
        (cases
          (≡> [0] 1)
          (≡> [=n] (n * (factorial n - 1)))]}
     (factorial 3))
```

evaluates to 6.

(name ≡ body)   is an abbreviation for

```
(labels
   {[name ≡ body]} name)
```

The value of the above expression is the value of body, except that occurrences of name in body refer to the whole of body. For example, if

```
(self ≡
  (cases
    (≡> (nq: =x)
      (queuees  [$queuees x])
      self)))
```

receives (nq: a), then it returns the case clause which the cell queuees contains a sequence of the previous content of queuees and x.

(name ≡ body) is also used in an iteration. For example,

```
[iterative-factorial ≡
    (≡> [=n]
      ([0 1] =>
        (loop ≡
          (≡> [=count =accumulation]
            (rules count
              (≡> n accumulation)
              (≡> ?
                (loop <= [(count + 1) (accumulation * count)]))))))))]
```

After a message is sent to the above expression and it gets bound to n, [0 1] is sent to (≡> [=count =accumulation]...). If count is equal to n, the result of iterative-factorial is the value of accumulation, and otherwise [(count + 1) (accumulation * count)] is sent to (≡> [=count =accumulation] ...).

<Let>

Besides the use of binding prefix = in pattern, another way of binding a value to name is the following expression.

```
(let
  {[name_1 = expression_1]
   [name_2 = expression_2]
          .
          .
   [name_n = expression_n]}
  body)
```

When this is evaluated, at first each expression is evaluated and each value is bound to name and body is evaluated in the resulted environment. The mutual recursion within equations is not allowed.

For example

```
(let
  {[a-number = 3]
   [a-cell = (cons-cell 1984)]}
   (a-cell ← a-number))
```

the content of a-cell becomes 3.

A P P E N D I X - II

A CONTRACT FOR CELLS

```
[contract-for cell ≡
   (((cons-cell =a) creates-an-actor  C where {(C is (Cell a))})

    (to-simplify
         ((C <= contents?) where {(C is (Cell b))})
       try-using
         b)

    (to-simplify
         (((C <= [← d]) where {(C is (Cell e))})
       try-using
         (C where {(C is (Cell d))})]
```

A P P E N D I X - III

A DEFINITION OF CONVERGENCE AND PARTIAL ORDERING

From the view point of the actor concept the "convergence" or "termination" of the activation of an actor (procedure) A is stated in terms of events. Suppose A gets activated in the following event:

```
[ A <==** (apply: message
                   (then-to: continuation)
                   (else-to: complaint-dept))]
```

Then the activation of A always converges if in the succeeding events there always happens one of the following events:

```
[continuation <= m]  and  [complaint-dept <= m']
```

where m and m' are arbitrary messages.

The general technique of showing the convergence is to find a partial ordering R in the events where the above events [continuation <= m] and [complaint-dept <= m'] are the ***minimal*** events in the ordering R.

) The double shafted arrow <== is called the ***apply-level-send.* The apply-level-send is used for making the continuation and compalint-department explicit in the transmission of messages. In the main contents of this paper the ordinal single shated arrow is used to express the transmission of messages with defaulted continuation or complaint-department. In fact, (a-target <= a-message) is an abbreviation for

```
(a-target <==
      (apply: a-message
           (then-to: defaulted)
           (else-to: defaulted)))
```