META-EVALUATION
OF
ACTORS WITH SIDE-EFFECTS

by

AKINORI YONEZAWA

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 which allow sharing of data is presented. Typical examples of actors with side-effects are the cell, actor counterparts of the LISP function replace and replace, and procedures whose computation depends upon their input history. Meta-evaluation of actors with side-effects 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). 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 Advance Research Projects Agency of the Department of Defence under Office of Naval Research contract N000-14-74-C-0643.

Working paper are informal papers intended for internal use.
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 [Greif & Hewitt 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 needs 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 verification 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 message 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 we will use 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 "contents?" 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 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 terms of these functional semantics, although data-structures without side-effects could be expressed by giving axioms for the operations on data structures. Thus within the framework 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, the following is a theorem proven by this system.

\[(\text{EQUAL} \ (\text{REVERSE} \ (\text{REVERSE} \ A)) \ A).\]

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

\begin{align*}
(CAR \ A) & \rightarrow \ (\text{CAR} \ A), \ (\text{CAR} \ (\text{CONS} \ A \ B)) \rightarrow \ A \\
(\text{EQUAL} \ A \ A) & \rightarrow \ T, \ (\text{EQUAL} \ A \ B) \rightarrow \ \text{NIL} \\
(\text{CDR} \ (A \ B \ C)) & \rightarrow \ (B \ C) \text{ where } A, B, \text{ and } C \text{ are free variables.}
\end{align*}

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. (That is, pure LISP is an applicative language.)
2. The parameter mechanism of pure 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 \text{RPLACA}. The symbolic evaluation of \text{RPLACA} could be expressed as follows:

\[(\text{RPLACA} \ (A \ B) \ C) \rightarrow \ (C \ B)\]

but this description does not capture the most important fact which distinguishes \text{RPLACA} from \text{CONS}. Namely \((\text{CONS} \ 'a \ 'b)\) creates a new dotted pair \((a;b)\) while the result of \((\text{RPLACA} \ '(a;b) \ 'c)\) i.e. \((c;b)\) is the same dotted pair as the first argument of \text{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 are 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].

\[
\begin{align*}
P & \{ A \} Q(e) \\
\cdots\\
P & \{ A ; \ x = e \} Q(x)
\end{align*}
\]

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 shared structures are used. The reason is obvious. This simple substitution does not take account of changes to shared data. In the following example, the above rule cannot tell the final value of $x[2]$.

\[ x + A[1]; \]
\[ tf x[2] = 3 \text{ then } (A[1])[2] + 4; \]

R. Burstall [1971] proposed some techniques which are able to handle list processing languages by extending Floyd's proof system. He introduced the notation

\((x \rightarrow_{y} b \rightarrow_{c} n!)\)

to denote the following linear list structure.

\[
\begin{array}{c}
\text{x} \\
\text{----------------> a ---> b ---> c nil}
\end{array}
\]

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, \((aq; x)\) which is a request to enqueue a
new element \( x \) and \((dq:1)\) which is a request to return the front element of the queue and the remaining queue. However, if the queue is empty, it returns a complaint of why it cannot provide the next element. The essence of their 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-evaluation 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 \((nq; x)\) on a queue represented as \((\text{queue } !q)\).

\[
(nq \text{ a at-rear-of (queue } !q)) = (\text{queue } !q \ a)
\]

But it should be noted that \((\text{queue } !g)\) and \((\text{queue } !g \ a)\) denote different actors. (%! stands for the "unpack" operation on a sequence. For example, if \( x \) denotes a sequence \([a \ b \ c]\), then \([!x \ d]\) becomes \([a \ b \ c \ d]\) instead of \([!a \ b \ c \ d]\). For more comprehensive explanation, see [Hewitt & Smith 1975].)

**Inpure 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 \((nq; x)\) and \((dq;1)\), but it behaves in a different way. When it receives the \((dq; \ldots)\) message, it does not create a new queue-actor. And when it receives the
(dq:) message, it returns its front element and itself 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 in PLASMA is given in Figure 2. (<packager>:<elements>) is an expression which stands for an actor called a "package". The meta-syntactic variable "packager" serves as a name for the package. The <elements> which are contained in the package can be other packages. When packages are used in a message or a pattern, the ordering of components is unimportant because the elements are tagged using the packagers. Some components may be optional. Some examples of packages in Figure 2 are (nq:=x), (dq:=), and (next:front(rest:=self)).

(A brief explanation of the PLASMA syntax is found in [Hewitt & Smith 1975].)

```lisp
(defun cons-impure-q (initial-elements)
  (let ((queuees (cons-cell initial-elements)))
    (self =
      ;; a queue-actor is defined as the following
      ;; case-clause and denoted by "self".
      (cases
        (nq:=x)
        (queuees <- (cons $queuees x)) ;; the new element x is stored
        ;; in the cell queuees with the old elements.
        self
      )
      (s> (dq:=)
        ;; when receiving the dequeuing message
        (rules $queuees
          (=> () (complaint: exhausted)) ;; if the contents of queuees is empty.
          (=> [=front !=rest]
              ;; exhausted message is returned.
              (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 \((\text{cons-impure-q a})\). If a message \((\text{nq:b})\) is sent to \(Q\), then the cell queues contains \([a b]\), but no new actors are created. If \(Q\) receives \((\text{dq:})\), \(a\) is sent back and the contents of the cell queues becomes \([b]\), and if \(Q\) receives \((\text{dq:})\) again, \(b\) is sent back and the contents of the cell queues 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 \((\text{dq:...})\) message. It does not indicate that the same actor is returned, after sending the \((\text{dq:...})\) to \(Q\). The following example will clarify this point.

\[
\begin{align*}
\text{(let \{[queue-1 = (cons-impure-q 1)]\)} \\
\text{(let \{[queue-2 = (queue-1 <= (nq: 2))]\)})}
\end{align*}
\]

The effect of the above code is as follows: a queue which contains 1 is created and bound to \(\text{queue-1}\) and then \((\text{nq:2})\) is sent to \(\text{queue-1}\) and the result is bound to \(\text{queue-2}\). In the above example, in order to tell that the length of \(\text{queue-1}\) is equal to that of \(\text{queue-2}\) after the two let-statements, we have to know that \(\text{queue-1}\) and \(\text{queue-2}\) refer to the same actor. This would not be the case if \(\text{queue-1}\) made a new queue when it received the message \((\text{nq:2})\).

**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 $(t \leftarrow m)$ borrowed from the PLASMA syntax (for the double shafted arrow $\leftarrow$, see the footnote in Appendix-II). 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 physically 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 [Hewitt 1975a] are used.

\begin{align*}
\text{(length } a\text{-queue}_S) &= 8, & \text{the length of } a\text{-queue in a situation } S \text{ is 8.} \\
\text{((t } \leftarrow m) \text{ in } S_8) &= \text{the event } (t \leftarrow m) \text{ occurs in a situation } S_8. \\
\text{(content } a\text{-cell}_S) &= 1984; \text{the content of } a\text{-cell in a situation } S \text{ is 1984.}
\end{align*}

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 an implementation satisfies its contract.

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 are considered as universally quantified variables.)

\[
((\text{cons-impure-q } !a) \ \text{creates-an-actor } Q \ \text{where} \ ((Q \ \text{is} \ (\text{Impure-queue} \ !a)))))
\]

Namely, an actor \( Q \) is created by \((\text{cons-impure-q } !a)\) and the property that \( Q \) is a queue with queues \( !a \) is expressed in the notation \((Q \ \text{is} \ (\text{Impure-queue} \ !a))\). In other words we are using \((\text{Impure-queue} \ !a)\) as the abstract representation. As will be seen later, this notation is also used as assertions in the database for the meta-evaluation.

The next thing to state in the contract is how the actor \( Q \) responds to the \((dq:\ldots)\) 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 \((dq:\ldots)\) message, we express its response as follows.

\[
\begin{align*}
&\ \text{(to-simplify)} \\
&\ ((Q <= (\text{nq: } =\!x)) \ \text{where} \ ((Q \ \text{is} \ (\text{Impure-queue} \ !b)))) \\
&\ \text{try-using} \\
&\ (Q \ \text{where} \ ((Q \ \text{is} \ (\text{Impure-queue} \ !b \ x))))
\end{align*}
\]
This notation claims that if an event \((Q <= (nq: x))\) happens in a situation where \((Q \text{ is } \text{Impure-queue } !b)\) holds, then in the succeeding situation the actor \(Q\) is returned and \((Q \text{ is } \text{Impure-queue } !b \ x)\) holds. \(\text{Impure-queue } !b \ x\) 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 situation is not explicitly introduced into the contract; instead where-clauses are used. But in the process of the meta-evaluation situations are used explicitly in the reasoning.

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 \((\text{either } \text{consider } \ldots \ldots \text{then})\) expression as below. Each clause in an \((\text{either } \ldots \) expression is mutually exclusive with the other clauses and the clauses are all inclusive.

\[
\begin{align*}
\text{(to-simplify)} \\
(Q <= (dq:)) \\
\text{try-using} \\
(\text{either}) \\
(\text{consider } Q \text{ is } \text{Impure-queue}) \\
(\text{then:}) \\
(\text{complaint: } \text{exhausted}) \\
(\text{consider } Q \text{ is } \text{Impure-queue } x !c) \\
(\text{then:}) \\
((\text{next: } x (\text{rest: } Q)) \text{ where } ((Q \text{ is } \text{Impure-queue } !c)))) \\
\end{align*}
\]

Suppose that \((Q <= (dq:))\) happens in a certain situation and if \(Q\) is not empty, namely \((Q \text{ is } \text{Impure-queue } x !c)\) holds in the situation, then \((\text{next: } x (\text{rest: } Q))\) should be returned in the next situation. For the case where \(Q\) is empty, namely \((Q \text{ is } \text{Impure-queue})\) holds, \((\text{complaint: exhausted})\) should be returned 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 I a contract for
a cell actor in the same formalism is given.

(contract-for cons-impure-q =
  ((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:))
  try-using
  (either
    (consider (Q is (Impure-queue)))
    (then:
      (complaint: exhausted)))
    (consider (Q is (Impure-queue y !c)))
    (then:
      ((next: y (rest: Q))
       where ((Q is (Impure-queue !c)))))
  ))

Figure 3

Instead of using an abstract representation for queues, J. Spitzen and B. Wegbreit [1975] state their specification of queues in terms of a relation which holds between the enqueueing and dequeueing operations. Namely, if a queue is not empty, the enqueueing and dequeueing operations are commutative. This relation between the \textit{nq} and \textit{dq} operations is easily derived from our contract for impure queues.
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. We present the contract for this actor (Figure 4) before presenting its concrete implementation. Other modules which use the (empty... into...) below should only rely on properties that can be derived from the contract.

\[
\text{[contract-for (empty ... into ...)} =
\]

\[
\text{(to-simplify)
}\]

\[
\text{(empty -Q1 into -Q2)
}\]

\[
\text{where (Q1 is (Impure-queue !w1))
}\]

\[
\text{(Q2 is (Impure-queue !w2))
}\]

\[
\text{(Q1 not-eq Q2)}
\]

\[
\text{try-using
}\]

\[
\text{(done: (emptied: Q1)(extended:Q2))
}\]

\[
\text{where (Q1 is (Impure-queue))
}\]

\[
\text{(Q2 is (Impure-queue !w2 !w1)))}
\]

\[
\]

Figure 4

The implementation of queues using pointers by J. Spitzen and B.Wegbreit [1975] is not protected from illegitimate accesses. Since their queues are implemented as a non-primitive mode using a mode constructor, STRUCT, a program could easily destroy the internal structure of such queues by using an access mechanism provided for STRUCT.

Any module which uses queues relies implicitly on their integrity. For example, (empty... into...) strongly relies on the integrity of queues.

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 abstract operations on a queue rather than directly in terms of the transmission of \((dq:)\) and \((dq:...)\) 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.

\[
\begin{align*}
(nq = x \text{ at-rear-of the-queue}) & \Rightarrow \\
\text{the-queue} <= (nq: x)
\end{align*}
\]

Furthermore in order to impose a certain constraint on the types of incoming actors, a new syntactic device \(<\text{pattern}> \text{is-a} (<\text{type}>)\) is introduced. For example \((-q1 \text{ is-a Impure-queue ...})\) requires that the type of actors which are bound to \(q1\) should be Impure-queue (i.e. a queue with side-effects). One should note 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. Every time \((dq:)\) or \((dq:...)\) 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 \((\text{empty q1 into q2})\), completely new actors would be returned as \((\text{done: emptied:q1'})(\text{extended:q2'})\) 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 (=q2 is-a (Impure-queue ...))) ; two queue-actors
; with side-effects are sent
; and bound to q1 and q2.
(dq q1) ; the dequeuing message is sent to q1.
; if q1 is not empty
(m> (next: =front-q1
    (rest: =dequeued-q1)) ; the front element of q1 and remained queue are received
    ; and bound to front-q1 and dequeued-q1.
    (mq front-q1 at-rear-of q2) ; front-q1 is enqueued.
    (empty q1 into q2)) ; q1 and q2 are sent to empty.
    ; if q1 is empty
(m> (complaint: exhausted) ; exhausted message is received
    (done:
        (emptied: q1)
        (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 implementation details 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.
\[-S_{\text{initial}}-\]

\[
((\text{empty } (=q_1 \text{ is-a Impure-queue ...}) \text{ into} \\
= q_2 \text{ is-a Impure-queue ...})) \Rightarrow \]

\[-S_{\text{dq}}-\]

(dq q1)

\[-S_{\text{next-0}}-\]

\[
(\Rightarrow (\text{next: } =\text{front-q1} \\
\text{rest: } =\text{dequeued-q1}))
\]

\[-S_{\text{next-1}}-\]

\[
(nq \text{ front-q1 at-rear-of q2})
\]

\[-S_{\text{next-2}}-\]

\[
(\text{empty q1 into q2}))
\]

\[-S_{\text{else-0}}-\]

\[
(\Rightarrow (\text{complaint: } \text{exhausted})
\]

\[-S_{\text{else-1}}-\]

\[
(\text{done:} \\
\text{emptied: q1} \\
\text{extended: q2}))
\]

Figure 6
For example, the $S_{\text{initial}}$ at the top of Figure 6 denotes the situation in which the transmission of two impure queues to (empty... into...) occurs and the $S_{\text{next-}}$ 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 cons-impure-q 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.

$$S_{\text{initial}} =
\{(Q1 \text{ is } \text{Impure-queue } \!x1)) \ (Q2 \text{ is } \text{Impure-queue } \!x2))
\ (Q1 \text{ not-eq } Q2)\}$$

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>).

$$S_{\text{dq}} =
\{(q1 = Q1) \ (q2 = Q2)
\ (Q1 \text{ is } \text{Impure-queue } \!x1)) \ (Q2 \text{ is } \text{Impure-queue } \!x2))
\ (Q1 \text{ not-eq } Q2)\}$$

Then the dequeuing message is sent to the actor bound to q1 in $S_{\text{dq}}$. 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, $S_{\text{next-}}$ and $S_{\text{else-}}$, are considered as the next situation of $S_{\text{dq}}$. For $S_{\text{else-}}$, the meta-evaluator asserts the following assertions.

$$S_{\text{else-}} =
\{(|x1 = [])) \ (Q1 \text{ is } \text{Impure-queue } \!x1))
\ (q1 = Q1) \ (q2 = Q2)
\ (Q2 \text{ is } \text{Impure-queue } \!x2))
\ (Q1 \text{ not-eq } Q2)\}$$
Now the message (complaint: exhausted) is returned. But since no binding of actors occurs, the next situation is the same as $S_{\text{else-5'}}$.

\[ S_{\text{else-1}} = S_{\text{else-5'}} \]

Then in $S_{\text{else-1}}$ the transmission of \((\text{done: emptied:01})(\text{extended:02})\) to the implicit continuation in the original message to \(\text{(empty... into...)}\) occurs. Note that we have used the assertions \((q1 = 01), (q2 = 02)\). It is easily seen that what the contract of \(\text{(empty... into...)}\) in Figure 4 requires, namely:

\[
\begin{align*}
(Q1 \text{ is } \text{Impure-queue}) \\
(Q2 \text{ is } \text{Impure-queue} !x2 !x1)
\end{align*}
\]

are satisfied by using knowledge about the sequences (See [Hewitt & Smith 1975] for PLASMA syntax):

\[ ![x2] !x1 \text{ is equal to } ![x2] \text{ if } x1 \text{ is equal to } []. \]

So the case where \(q1\) is empty is done.

For the other case, the meta-evaluator asserts the following assertions with a tag $S_{\text{next-8}}$ where \(z1\) and \(z2\) are newly generated identifiers.

\[ S_{\text{next-8}} \]

\[
\begin{align*}
((x1 = [z1 !z2]) & \ (Q1 \text{ is } \text{Impure-queue} !z2)) \\
(q1 = q1) & \ (q2 = 02) \ (Q1 \text{ not-eq } 02) \\
(02 \text{ is } \text{Impure-queue} !x2))
\end{align*}
\]

In $S_{\text{next-8'}}$ \((\text{next: } z1 \ (\text{rest: } Q1))\) is transmitted and the pattern matching is performed. So the meta-evaluator asserts the binding information with a tag $S_{\text{next-1'}}$:

\[ S_{\text{next-1'}} \]

\[
\begin{align*}
((\text{front-q1} = z1) & \ (\text{dequeued-q1} = 01)) \\
(x1 = [z1 !z2]) & \ (Q1 \text{ is } \text{Impure-queue} !z2)) \\
(q1 = q1) & \ (q2 = 02) \ (Q1 \text{ not-eq } 02) \\
(02 \text{ is } \text{Impure-queue} !x2))
\end{align*}
\]

The (mq:z1) message is sent to 02 in $S_{\text{next-1'}}$. By using the \(\text{(to-simplify...)}\)-clause for the enqueuing message in the contract in Figure 3, the meta-evaluator asserts the following assertions with a tag $S_{\text{next-2'}}$. Note that the crucial fact is that 01 and 02 are distinct impure queues.
\[ S_{\text{next-2}} = \]
\[
((Q2 \ is \ (Impure-queue \ !x2 \ z1)))
(front-q1 = z1) (dequeued-q1 = Q1)
(x1 = [z1 !z2]) (Q1 is (Impure-queue !z2))
(q1 = Q1) (q2 = Q2) (Q1 not-eq Q2))
\]

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

\[(\text{done; (emptied:01)(extended:02)) is returned} \]

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

Again knowledge about the sequences is used:

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

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 the conditions of a situation hold when control passes through the situation. There is no guarantee that the situation described will ever be reached.

The demonstration of convergence is another part of meta-evaluation which is
treated 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 II). 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. That is, 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 strictly 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:
   ($(length-of q_{S_{next-2}}) \text{less-than} (length-of q_{S_{dq}}))$

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:

$(length-of q_{S_{next-2}})$ is the length of $[!z2]$.
$(length-of q_{S_{dq}})$ 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 below.

(to-simplify (length []) try-using 0 )
(to-simplify (length x !y) try-using (1 + (length !y)))

Before we leave this section it should be pointed out that the whole argument on the
convergence of (empty... into...) relies heavily on the pre-requisite that q1 and q2 are
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 \( \text{new-element: } x \) message which enters a number \( x \) into the data base, and a message \( \text{average?} \) which asks for the average of all the numbers currently in the data base. Figure 7 below is a contract for this actor.

```plaintext
(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 \( D \) 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 (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-element) =
  (let
    [(current-average = (cons-cell initial-element))] ; 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)
        (counter += (counter + 1)); counter is incremented by 1
        (current-average += ((current-average * (counter - 1) + x)/counter)) ; the current average is calculated
        ; and store in the cell current-average.
        self) ; self is returned.
      (=> average?
        (current-average) ; when received 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 extension 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 [Steele 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, and pruned irrelevant branches which I was about to climb during this research. Thanks are also due to Keith Nishihara and Guy L. Steele Jr. who made comments and have been correcting my English.
BIBLIOGRAPHY


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


APPENDIX-I

A CONTRACT FOR CELLS

(contract-for cons-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))))))}
APPENDIX - II

A DEFINITION OF CONVERGENCE AND PARTIAL ORDERING

From the viewpoint 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:

\[ A \leftarrow \text{apply: } \begin{array}{l}
message \\
(\text{then-to: } \text{continuation}) \\
(\text{else-complaint-to: } \text{complaint-dept})
\end{array} \]

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

\[ \text{continuation } \leftarrow m \text{ and } \text{complaint-dept } \leftarrow m' \]

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 \( \text{continuation } \leftarrow m \) and \( \text{complaint-dept } \leftarrow m' \) are the minimal events in the ordering R.

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

\[ \text{(a-target } \leftarrow \text{a-message) is an abbreviation for} \]

\( \begin{array}{l}
(\text{apply: } \text{a-message}) \\
(\text{then-to: defaulted}) \\
(\text{else-complaint-to: defaulted})
\end{array} \).