MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 124 April 1976

SYMBOLIC-EVALUATION AS AN AID TO PROGRAM SYNTHESIS

AKINORI YONEZAWA

ABSTRACT

Symbolic-evaluation is the process which abstractly evaluates an actor program and checks to see whether the program fulfills its *contract* (specification). In this paper, a formalism based on the *conceptual representation* is proposed as a specification language and a proof system for programs which may include *change of behavior* (side-effects). The relation between algebraic specifications and the specifications based on the conceptual representation is discussed and the limitation of the current algebraic specifications is pointed out. The proposed formalism can deal with problems of side-effects which have been beyond the scope of Floyd-Hoare proof rules. Symbolic-evaluation is carried out with explicit use of the notion of *situation* (local state of an actor system). Uses of *situational tags* in assertions make it possible to state relations holding between objects in different situations. As an illustrative example, an *impure* actors which behave like a queue is extensively examined. The verification of a procedure which deals with the queue-actors and the correctness of its implementations are demonstrated by the symbolic-evaluation. Furthermore how the symbolic-evaluation serves as an aid to program synthesis is illustrated using two different implementations of the queue-actor.

This report describes research done at the Artificial Intelligence laboratory of the massachusetts Institute of Technology. Support for the laboratory's artificial intelligence reseach is provided in part by the Advance Research Projects Agency of the Department of Defence under Office of Naval Research contract N00014-75-C0522.

INTRODUCTION

Our goal is to construct a software system called a Programming Apprentice [Hewitt & Smith 1975] which aids expert programmers in various aspects of programming activities, such as verification, debugging and refinement of programs. As its major component, the Programming Apprentice is expected to have a very high level interpreter which abstractly evaluates a program on abstract data and tries to see whether the program satisfies its contracts (specifications). We call this process "*symbolic-evaluation*" [Hewitt et al 1973, Boyer & More 1975, Burstall & Darlington 1975, Rich & Shrobe 1975, Yonezawa 1975]. Our purpose of symbolic-evaluation is not simply to verify program modules, but also to provide sufficient information for answering questions about dependencies between program modules and implications of changes in both specifications and program modules. To accomplish these purposes, the symbolic-evaluation must be based on an adequate coherent formalism which consists of:

1) a formal language for writing contracts of modules which may change their behavior (side-effects) [Greif & Hewitt 1975, Yonezawa 1975],
2) a formal language for expressing intentions of programmers,
3) a formal language for representing domain specific knowledge, and
4) a formal system for reasoning.

In addition, since the Programming Apprentice is used in an interactive manner, the formalism should be intuitively clear and easily understood by human users. The purpose of our research is to develop a formalism which meets these requirements.

Obviously our research is closely related to the currently active research on proving properties of programs. But all previous program verification systems[King 1969, Deutch 1973, Igarashi London & Luckham 1973, Suzuki 1974, Boyer & Moore 1975] have not been able to deal with programs with change of their behavior (side-effects) because of the inadequacy of the formal systems on which these implementations were based. Furthermore, the previous research on algebraic and axiomatic techniques [Hoare 1972, Wegbreit & Spitzen 1975, Zilles 1975, Guttag 1976] for specifying data-structures has not dealt with data-structures with side-effects. Although a program with side-effects can sometimes be transformed into a program without side-effects [Greif & Hewitt 1975], the transformation decreases efficiency and requires several times the storage. Also there is a certain type of side-effect in communication between concurrent processes which is impossible to realize without side-effects. So the exclusion of programs which change their behavior from the research domain reduces the range of application. In the early sections of this paper, we will discuss the limitations of previous work on program verification and algebraic specification techniques and then propose a new formalism. It will be seen that the proposed formalism naturally meets the requirements for successful symbolic-evaluation.

PURE ACTORS AND IMPURE ACTORS

Since our symbolic-evaluation is based on the actor model of computation [Hewitt et al 1973], we will begin with a brief explanation of actors.

An actor is a potentially active piece of knowledge (procedure) which is activated when it is sent a message by another actor. No actor can treat another actor as an object to be operated on; instead it can only send messages to other actors. Each actor decides itself how to respond to messages sent to it. An actor is defined by its two parts, script and acquaintances. Its script is a procedural description or code of how it should behave when it is sent a message. Its acquaintance is a finite set of actors that it directly knows about. If an actor A directly knows about another actor B, A can directly send a message to B. The behavior of an actor can be roughly characterized by stimuli (messages as questions) and responses (messages as answers). In the actor paradigm, the traditional concepts of procedure and data-structure are unified. Furthermore various kinds of control structures such as go-to's, procedure calls, and coroutines can be viewed as particular patterns of message passing [Hewitt 1976]. Thus a complete model of computation can be constructed with a system of actors.

All actors are classified into two categories depending upon their behavior. Actors which belong to one category never change their behavior. They are called *pure actors*. Actors which belong to the other category are called *impure actors* and their behavior may change during message passing. More precise definitions are as follows.

> An actor is pure if it always gives the same response to the same message.
>
> An actor is impure (not pure) if it does not always give the same response to the same message.

From this definition, it can be said that a pure actor behave like a mathematical function. The only primitive impure actor we will use is the "cell". A cell-actor accepts a message which updates its contents and a message (*contents?*) which retrieves its contents. A cell-actor may change its behavior because it can give different answers to the (*contents?*) message, depending upon what it contains at the moment. An example of pure actors is a sequence-actor. One can retrieve elements of a sequence-actor, but cannot change its elements. To change its elements a completely new sequence-actor must be created. So a sequence-actor is pure. The notion of impureness of actors is closely related to that of side-effects in the traditional programming languages. A typical example of "side-effects" is the effect of updating components of a record which is shared by objects. This type of side-effects can be viewed as the change of behavior of actors which behave like data-structures. Cells do not make serious trouble 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 actors.

PREVIOUS WORK

An automatic theorem prover for pure LISP functions [Boyer & Moore 1975] is considered to be based on interpretive semantics. The following examples illustrate how some LISP functions are symbolically evaluated in their system:

```
(CONS A B) --> (A . B), (CAR A) --> (CAR A), (CAR (CONS A B)) --> A
(EQUAL A A) --> T, (EQUAL A B) --> NIL
(CDR (A B C)) --> (B C) where A, B, and C are free variables.
```

So far as pure LISP functions are concerned, their formalism is sufficient for proving theorems for the following reasons:

1. Pure LISP functions are constructed solely by the composition of pure primitive 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 primitive functions of 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, their symbolic evaluation confronts a serious problem. Let us consider a non-pure function RPLACA. In their formalism the symbolic evaluation of RPLACA could be expressed as follows:

```
(RPLACA (A . B) C) --> (C . 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 their system 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 the 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 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. Hoare uses the notation P{Q}R to express the above meaning. The following figure illustrates how an assignment is treated in VCG [Igarashi London & Luckham 1973].

```
        P { A } Q(e)
-------------------------
      P { A ;  x ← 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 shared structures are used. The reason is obvious. This simple substitution does not take account of shared data. In the following example, the above rule cannot tell the correct final value of x[2]. (A is a two-dimensional array and X is a one-dimensional array.)

```
x ← A[1] ;  ;a slice of A is assigned to X.
   ...
(A[1])[2] ← 4;
   ...
```

By extending Floyd's proof system, R. Burstall [1971] proposed some techniques which are able to handle list processing languages. But his formalism has not suffucient expressive power for our purpose.

PURE QUEUES AND IMPURE QUEUES

Queues are a common type of data-structure in the traditional programming languages. As we pointed out earlier, since every object is an actor in the actor model of computation, a queue is also an actor which has its script and acquaintances. The queue-actor will be used as a very convenient illustrative example throughout this paper.

A pure queue-actor behaves as follows: a queue accepts two kinds of messages, (*nq:* x) which is a request to enqueue a new element x and (*dq:*) 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 message (*complaint:* (*exhausted:*)). Below we give a contract (specification) of pure queues based on a *conceptual representation*. The notation (*PURE-QUEUE* [!b]) represents a pure queue-actor which has [!b] as its queuees (i.e. members of the queue). ! 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]. If [a b c d] = [a !y], then y has to denote [b c d]. For a more comprehensive explanation, see [Hewitt &

Smith 1975]. The following is a contract for pure queue-actors.

```
A)  (cons-pure-queue) = (PURE-QUEUE [])              ; the evaluation of (cons-pure-queue)
                                                                  ; creates an empty queue.
B)  ((PURE-QUEUE [!b]) <= (nq: X)) = (PURE-QUEUE [!b X])
                                                         ; the enqueue message (nq:...) is sent.
C)  ((PURE-QUEUE []) <= (dq:)) = (complaint: (exhausted:))
                                            ;the dequeue message (dq:) is sent to an empty queue.
D)  ((PURE-QUEUE [Y !c]) <= (dq:)) = (next: Y (rest: (PURE-QUEUE [!c])))
                                         ;the dequeue message (dq:) is sent to a non-empty queue.
```

Figure 1

(*PURE-QUEUE* ...) is an example of conceptual representations. The conceptual representation is a new technique for formal specifications of data-structures. In a later section we will give another contract for pure queues which also uses the conceptual representation. The techniques which have been actively investigated are algebraic and axiomatic ones [Wegbreit & Sptizen 1975, Zilles 1975, Guttag 1976]. To compare it with our approach, we give the following algebraic specification of pure queues.

a) Cons-queue: --> *queue*
b) Enqueue: *queue* x *item* --> *queue*
c) Dequeue: *queue* --> *item* x *queue* / *error*

I) Dequeue(Cons-queue) = Error
II) *if* Dequeue(q) = <b, q'>,
        then Dequeue(Enqueue(q,a)) = <b, Enqueue(q', a)>.

a), b) and c) specify the domains and ranges of the operations associated with a queue. I) and II) are axioms of the operations. These axioms are easily derived from the above contract for pure queues based on the conceptual representation. For the derivation of the axiom II), see Appendix I.

In contrast to the pure queue, let us consider an impure actor which also behaves like a queue. This actor accepts the same messages, namely (*nq:* x) and (*dq:*), but it behaves in a different way. When it receives the (*nq:*...) message, it does not create a new queue-actor. When it receives the (*dq:*) message, it returns its front element and itself as the remaining queue. Suppose Q is an actor which is created by evaluating (cons-impure-queue). At this moment Q is an empty queue. If a message (*nq:*a) is sent to Q, then a is absorbed inside Q and Q itself is returned, but no new queue actors are created during this process. Q responds to (*dq:*) in two different ways: if Q receives (*dq:*), (*next:* a (*rest:* Q)) is sent back, and then if Q receives (*dq:*) again, (*complaint: exhausted:*)) is returned. Therefore Q is an impure actor. In a later section, we will give a contract for this impure queue in a formalism based on the conceptual representations. It should be noted that either of the above two specifications of <u>pure</u> queues does not correctly specify the behavior of this <u>impure</u> queue. Because, for example, in B) of the contract, it is not specified that (*PURE-QUEUE* [!b]) and (*PURE-QUEUE* [!b X]) represent the same queue-actor. The following example will clarify this point.

```
(let (queue-1 = (cons-impure-queue))          ;an empty queue is created and bound to queue-1
   then
    (let (queue-2 = (queue-1 <= (nq: 2)))                      ;(nq: 2) is sent to queue-1 and
                              ;the same actor which has been bound to queue-1 is bound to queue-2
      then
       (queue-1 <= (nq: 3))...))                                  (nq: 3) is sent to queue-1
```

In the above example, in order to tell that the length of queue-1 is equal to that of queue-2 after the third statement, we have to know that queue-1 and queue-2 refer to the same actor. This would not be the case if queue-1 created and returned a new queue-actor when it received the message (*nq:* 2). Floyd-Hoare proof rules cannot deal with this type of problems.

EVENTS AND SITUATIONS

As has been discussed in the preceding sections, we need some device to describe the more global state at a given moment in order to deal with side-effects. Since our symbolic-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, a message actor, m, and an activator, ac. Since we are primarily concerned with an actor system without parallelism [Greif 1975], we will not consider activators. An event is defined as a transmission of an message actor m to a target actor t which will sometimes be denoted by the notation (t <= m) borrowed from the PLASMA syntax [Smith & Hewitt 1975]. This definition of events is slightly simplified one, but sufficient for our purpose. A situation S can now be defined as the local state of an actor system at the moment when 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 assertions which are true at that moment. The following examples illustrate how the *situational tags* [Hewitt 1975a] are used.

(*length* a-queue$_S$) = 8, ;*the length of* a-queue *in a situation S is 8.*
((t <= m) *in* S$_8$), ;*the event* (t <= m) *occurs in a situation* S$_8$.
(*contents* a-cell$_S$) = 1984, ;*the contents of* a-cell *in a situation S is 1984.*

If we are to state some relations between facts which hold in 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 an implementer of a module and users of the module. In symbolic-evaluation of an actor we are checking to see that an implementation satisfies its contract.

In our formalism for writing contracts, the following conventions are observed: variables prefixed with "=" are pattern variables or formal arguments; each variable in upper case letters denotes an actor, and each variable in lower case letters denotes a sequence of actors or an empty sequence. Note that a variable in lower case letter does not denote a sequence-actor!! !x (unpack operation on x) expresses the juxtaposition of actors denoted by x.

The first thing we have to state in the contract is how an impure queue-actor is created. We state it in our formalism as follows.

```
((cons-impure-queue) creates-an-actor
                     (Q where {(Q is (IMPURE-QUEUE []))})
```

Namely, an actor Q is created by evaluating (cons-impure-queue) and the property that Q is an empty queue is expressed in the notation (Q *is* (*IMPURE-QUEUE* [])). This notation is a special case of (Q *is* (*IMPURE-QUEUE* [!a])) which asserts that Q is a queue with [!a] as its queuees (i.e. members of the queue). The notation (*IMPURE-QUEUE* [!a]) is a *conceptual representation* of an impure queue-actor. As will be seen later, this notation is also used as assertions in the date base for the symbolic-evaluation.

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

```
(result-of
     ((Q <= (nq: =X)) where {(Q is (IMPURE-QUEUE [!b]))})
   is
     (Q where {(Q is (IMPURE-QUEUE [!b X]))}))
```

This notation claims that if an event (Q <= (*nq*: =X)) (namely, a message (*nq*:...) is sent to Q) 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 behind 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 symbolic-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 introduce an (*either* (*if* ...)...) expression as below. Each clause in an (*either* ...) expression is mutually exclusive with the other clauses and the clauses are all inclusive.

```
(result-of                                        ;the result of the following event:
  (Q <= (dq:))                                                    ;(dq:) is sent to Q
 is                                                                               ;is
  (either                                                                     ;either
   (if  (Q is (IMPURE-QUEUE []))                     ;if Q is an empty impure queue
     (then:                                                                     ;then
      (complaint: (exhausted:))))               ;the complaint message is returned
   (if  (Q is (IMPURE-QUEUE [Y !c]))       ;or if Q is not empty and its queue is [Y !c]
     (then:                                                                     ;then
      ((next: Y (rest: Q))                  ;(next:...) is returned and in this situation
           where  {(Q is (IMPURE-QUEUE [!c]))}))) ))            ; Q has [!c] as its queuees.
```

Suppose that (Q <= (*dq:*)) (namely, a message (*dq:*) is sent to Q) happens in a certain situation and Q is not empty (namely, (Q *is* (*IMPURE-QUEUE* [Y !c])) holds in the situation). Then (*next:* Y (*rest:* Q)) should be returned in the next situation and Q has !c as its queuees. For the case where Q is empty, namely (Q *is* (*IMPURE-QUEUE* [])) holds, (*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.

In an earlier section, we gave a contract for pure queues in Figure 1 using a conceptual representation (*PURE-QUEUE*...). In that contract the notation (*PURE-QUEUE*...) represents a pure queue-actor itself. But in the contract for impure queues in this section, the notation (*IMPURE-QUEUE*...) is used to represent a state of an impure queue-actor. By using (*PURE-QUEUE*...) as a notation which represent a state of a pure queue-actor, we can write a contract for pure queues by slightly changing the contract for impure queues in this section. Namely, all occurences of (*IMPURE-QUEUE*...) should be replaced by (*PURE-QUEUE*...) and the second *if*-clause in the *result-of*-statment for the dequeuing should be replaced by the following *if*-clause.

```
(if  (Q is (PURE-QUEUE [Y !c]))
  (then:
    ((next: Y (rest: Q'))
        where  {(Q is (PURE-QUEUE [Y !c]))
                (Q' is (PURE-QUEUE [!c]))})))
```

A crucial point is that a different pure queue-actor Q' is returned and Q remains in the same state. A similar change in the *result-of*-statement for the enqueuing is also necessary.

The complete contract for an impure queue is depicted in Figure 2. In Appendix II a contract for a cell-actor in the same formalism is given. The contract for cells will be used in the symbolic-evaluation of concrete implementations of impure queues. Furthermore, a contract for an actor whose behavior depends upon a *history* of its incoming message is discussed in Appendix III.

```
[contract-for impure-queue ≡
 (((cons-impure-queue)  creates-an-actor                              ;an actor Q is created
       (Q   where  {(Q is (IMPURE-QUEUE []))}))             ;where Q is an empty impure queue.

 (result-of                                                ;the result of the following event:
   ((Q <= (nq: =X))                                        ;(nq:...) is sent to Q in the situation
     where {(Q is (IMPURE-QUEUE [!b]))})                     ; where Q has [!b] as its queuees
  is                                                                                        ;is
    (Q                                                     ;Q is returned and in this situation
       where  {(Q is (IMPURE-QUEUE [!b X]))}))                        ;Q has [!b X] as its queuees.

 (result-of                                                ;the result of the following event:
   (Q <= (dq:))                                                                 ;(dq:) is sent to Q
  is                                                                                        ;is
   (either                                                                              ;either
     (if  (Q is (IMPURE-QUEUE []))                               ;if Q is an empty impure queue
      (then:                                                                              ;then
       (complaint: (exhausted:))))                        ;the complaint message is returned
     (if  (Q is (IMPURE-QUEUE [Y !c]))          ;or if Q is not empty and its queue is [Y !c]
      (then:                                                                              ;then
       ((next: Y (rest: Q))                         ;(next:...) is returned and in this situation
          where  {(Q is (IMPURE-QUEUE [!c]))})))) ))                   ;Q has [!c] as its queuees.
```

Figure 2

As pointed out earlier, the actor model of computation can serve as underlying semantics of various programming languages such as SIMULA-67[Dahl et al 1968], CLU[Liskov 1974, Schaffert, Snyder & Atkinson 1975] and ALPHARD[Wulf 1974]. The fact that the above contract is precisely interpreted in terms of the message passing. (namely, the actor model of computation) assures that this contract is not biased by the languages in which the implementations are written.

A 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 members (i.e. queuees) of one impure queue to another impure queue. This code and contract will be used to illustrate symbolic-evaluation in the next section. We present the contract for this actor (Figure 3) 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.

```
[contract-for (empty ...  into ...) ≡
  (result-of                                                  ;the result of the following event:
    ((empty =Q1 into =Q2)                            ;Q1 and Q2 are sent to (empty...into...) where
       where {(Q1 is (IMPURE-QUEUE [!w1]))                       ;Q1 has [!w1] as its queuees,
              (Q2 is (IMPURE-QUEUE [!w2]))                       ;Q2 has [!w2] as its queuees,
              (Q1 not-eq Q2)})                          ;Q1 and Q2 are not the same actor.
  is                                                                                      ;is
    ((done: (emptied: Q1) (extended: Q2))                          ;(done:...) is returned where
       where  {(Q1 is (IMPURE-QUEUE []))                                 ;Q1 is an empty queue
               (Q2 is (IMPURE-QUEUE [!w2 !w1]))})))]        ;Q2 has [!w1 !w2] as its queuees.
```

Figure 3

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 constuctor, STRUCT, a program could easily destroy the internal structure of such queues by using an access mechanism provided for STRUCT. Recently they remedied this problem by adopting the class concept of SIMULA-67 [Wegbreit & Spitzen 1976]. Any module which uses queues relies implicitly on their integrity. For example, (*empty...into...*) strongly relies on the integrity of queues.

Figure 4 below shows an implementation of this actor.

```
((empty =q1 into =q2) ≡                   ;two impure queues are sent to (empty...into...)
                                                               ;and bound to q1 and q2.
   (rules (q1 <= (dq:))                         ;the dequeuing message is sent to q1.
                                                                     ;if q1 is not empty
      (≡> (next: =front-q1                               ;the front element of q1 and
            (rest: =dequeued-q1))                       ;remaining queue are received
                                                 ;and bound to front-q1 and dequeued-q1.
         (nq front-q1 at-rear-of q2)   ;front-q1 is enqueued in q2 behind its previous queuees.
         (empty q1 into q2))                         ;q1 and q2 are sent to (empty...into...).

      (≡> (complaint: (exhausted:))         ;if q1 is empty, the complaint message is received
         (done: (emptied: q1) (extended: q2))) ))                             ;emptied q1 and
                                                                ;extended q2 are returned.
```

Figure 4

One should note that the implementation in Figure 4 crucially depends on the fact that queue-actors referred to by q1 and q2 are impure actors. Suppose that these queue-actors are pure actors. Every time (*dq:*) or (*nq:...*) messages are sent, a new queue-actor would be created but q1 and q2 would still refer to the same queue-actors to which they originally referred. Therefore after completing of the evaluation of (*empty* q1 *into* q2), completely new queue-actors would be returned as (*done:* (*emptied:* q1') (*extended:* q')) and the original queue-actors referred by q1 and q2 would remain intact. This violates the contract in Figure 3. Also note that q1 and dequeued-q1 always denote the same queue-actor.

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

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 symbolic-evaluation of a larger program should be carried out by using only the contracts of its modules instead of being bothered by implementation details of the 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 symbolic-evaluation given below of the actor (*empty...into...*).

In general we assume that the symbolic-evaluator has a conceptually uniform data base (i.e. without the context mechanism of QA4 or Conniver) in which assertions are entered. If some assertions hold in a particular situation, they are asserted in the data base with tags which indicate the situations where they hold. In addition, the following rule, called *inheritance-rule* is used in the symbolic-evaluation: Suppose that some event takes place in a situation *S*. Then assertions which hold in *S*, but which are irrevelant in that event automatically hold in S'. Furthermore, we assume that notational devices which indicate the causal relation between situations are used. For example, one of such notations (S' ***is-caused-by*** <event> ***in*** S) states that a situation S' is caused by an <event> which occured in a situation S. But these notations are not explicitly shown in the subsequent symbolic-evaluation.

Now let us consider the symbolic-evaluation of *(empty...into...)* actor. In order to aid the symbolic-evaluation process the augmented code for *(empty...into...)* shown in Figure 5 is given to the symbolic-evaluator. (Actually this commentary of the code should be done through the interaction mode between users and the Programming Apprentice.) The large capital letters $S_{...}$ between the lines denotes the situations in which events occur. They will be used as situational tags for assertions in the data base.

```
  - S_initial -
((empty =q1 into =q2)                     ;two impure queues are sent to (empty...into...)
                                                               ;and bound to q1 and q2.
   - S_dq -
   (rules (q1 <= (dq:))                          ;the dequeuing message is sent to q1.
                                                                   ;if q1 is not empty
      - S_next-8 -
     (=> (next: =front-q1                                ;the front element of q1 and
             (rest: =dequeued-q1))                          ;remained queue are received
                                              ;and bound to front-q1 and dequeued-q1.
         - S_next-1 -
       (nq front-q1 at-rear-of q2)    ;front-q1 is enqueued in q2 behind its previous queuees.
           - S_next-2 -
       (empty q1 into q2))                         ;q1 and q2 are sent to (empty...into...).

      - S_else-8 -
     (=> (complaint: (exhausted:))         ;if q1 is empty, the complaint message is received
         - S_else-1 -
       (done: (emptied: q1) (extended: q2))) ))                              ;emptied q1 and
                                                              ;extended q2 are returned.
```

Figure 5

For example, the S_{initial} at the top of Figure 5 denotes the situation in which the transmission of two impure queues to *(empty...into...)* occurs and the $S_{\text{next-8}}$ denotes the situation in which the transmission of *(next:* actor-1 *(rest:* actor-2)) to the continuation of the dequeuing message to q1 occurs.

What follows is a detailed demonstration of the symbolic-evaluation of the augmented code cited in Figure 5 against the contract for *(empty...into...)* in Figure 3. The contract for impure-queue in Figure 2 is used extensively. The notation *in* $S_{\dots}$: <assertions> is used to mean that the situational tag $S_{\dots}$ is attached to each of the <assertions>.

First, by reading the contract of *(empty...into...)* in Figure 3 the symbolic-evaluator enters the following assertions which are the pre-requisites of *(empty...into...)* in the data base. Q1, Q2, x1 and x2 are newly generated identifiers.

in $S_{initial}$:

(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 program variables q1 and q2, respectively. Such binding of actors to program variables are generally expressed by assertions of the form (<identifier> = <actor>). So the following assertions are newly entered in the data base.

in S_{dq} :

(q1 = Q1)
(q2 = Q2)

Then the dequeuing message is sent to the actor bound to q1 in S_{dq}. By interpreting the (*result-of...*) clause for dequeuing in the contract in Figure 2, there are two cases to be considered, 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-0} and S_{else-0} , are considered as the next situations of S_{dq}. For S_{else-0}, the symbolic-evaluator asserts the following assertion.

in S_{else-0} :

(x1 = [])

Now the message (*complaint:* (*exhausted:*)) is returned. The next situation has the same assertions as S_{else-0}.

S_{else-1} = S_{else-0}.

Then in S_{else-1} the transmission of (*done:* (*emptied:* Q1) (*extended:* Q2)) to the implicit continuation that arrived with the original message to (*empty..into...*). Note that to get (*done:* (*emptied:* Q1) (*extended:* Q2)) from (*done:* (*emptied:* q1) (*extended:* q2)) we have used the assertions (q1 = Q1) and (q2 = Q2) which are inherited from S_{dq} through S_{else-0} and S_{else-1}. The requirements of the contract of (*empty..into...*) in Figure 3, namely:

(Q1 *is* (*IMPURE-QUEUE* []))
(Q2 *is* (*IMPURE-QUEUE* [!x2 !x1]))

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

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

So the case where q1 is empty is done.

For the other case S_{next-0}, the symbolic-evaluator enters the following assertion with a tag S_{next-0} where W and z are newly generated identifiers.

in S_{next-0} :
(x1 = [W !z])
(Q1 *is* (*IMPURE-QUEUE* [!z]))

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

in S_{next-1} :
(front-q1 ≡ W)
(dequeued-q1 ≡ Q1)

The (*nq:* W) message is sent to Q2 in S_{next-1}. By the inheritance rule (Q2 *is* (*IMPURE-QUEUE* [!x2])) holds in S_{next-1}, and from the (*result-of...*) clause for the enqueuing message in the contract in Figure 2, the symbolic-evaluator enters the following assertion with a tag S_{next-2}. Note that the crucial fact is that Q1 and Q2 are distinct impure queues.

in S_{next-2} :
(Q2 *is* (*IMPURE-QUEUE* [!x2 W]))

Now the symbolic-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 referred to. Since the pre-requisites of the (*empty...into...*), namely:

(Q1 *is* (*IMPURE-QUEUE* [!z])) and
(Q2 *is* (*IMPURE-QUEUE* [!x2 W]))

hold in S_{next-2}, the contract guarantees that (*done:* (*emptied:* Q1) (*extended:* Q2)) is returned and

(Q1 *is* (*IMPURE-QUEUE* [])) and
(Q2 *is* (*IMPURE-QUEUE* [![!x2 W] !z])) hold.

Then the following knowledge about sequences is used to simplify the above two assertions

[![!x2 W] !z] is equivalent to [!x2 W !z],
[![!x2 W] !z] is equivalent to [!x2 !x1] if x1 is equivalent to [W !z], which holds in S_{next-2}.

So symbolic-evaluator can claim 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 4 is guaranteed to meet its contract in Figure 3. 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 symbolic-evaluation. For a detailed demonstration of the convergence, see [Yonezawa 1975].

AN IMPLEMENTATION CONTRACT FOR IMPURE-QUEUE

In the symbolic-evaluation of (*empty...into...*), the properties of impure queues used to demonstrate its correctness were only the ones given in the contract for impure queues in Figure 2. This fact guarantees that the (*empty...into...*) works correctly on any implementations of impure queues as long as the implementations satisfy the contract in Figure 2. Now we give an example of a concrete implementation of impure queues which is supposed to satisfy the contract in Figure 2. The code depicted in Figure 6 is such an implementation written in PLASMA [Smith & Hewitt 1975]. A similar implementation written in CLU [Schaffert, Snyder & Atkinson 1975] is presented in Appendix IV. These two implementations exhibit the same computation sequence in terms of the actor model of computation (the message passing paradigm).

In Figure 6, a cell which is a typical example of impure actors is used in this implementation. (cons-cell a) is an expression which creates a cell-actor which contains an actor a. (a-cell ← new-contents) replaces the current contents of a cell-actor, a-cell, by an actor new-contents. $a-cell is an abbreviation of the expression (a-cell <= (*contents?*)) which retrieves the current contents of the a-cell. (<packager>: elements) is an expression which stands for an actor called a "package". Packagers are analogous to records in some languages. The meta-syntactic variable <packager> serves as a name for the package. When packages are used in a message or a pattern, the ordering of components is unimportant because the elements are tagged using packagers. Some components may be optional. Some examples of packages in Figure 6 are (*nq:* =new-element), (*dq:*), and (*next:* front (*rest:* the-queue-itself)). ! is the unpacked operation on sequences that we explained in the earlier section. (A brief explanation of the PLASMA syntax is found in [Hewitt & Smith 1975].)

```
((cons-impure-queue) ≡
  (let (queuees = (cons-cell []))              ;a cell which contains an empty sequence is created
     then
   (the-queue-itself ≡                      ;a queue-actor is defined as the following cases-clause
                                                               ;and denoted by the-queue-itself.
    (cases
     (≡> (nq: =new-element)       ;whenever an enqueue message with new-element is received,
       (queuees ← [!$queuees new-element])                                ;new-element is stored
                                                  ;in the cell queuees behind the previous elements.
       the-queue-itself)                                  ;and then the-queue-itself is returned.

     (≡> (dq:)                                           ;whenever an dequeue message is received,
      (rules $queuees                                                   ;if the contents of queuees
        (≡> [] (complaint: (exhausted:)))                ;is empty, then the message is returned.
        (≡> [=front !=rest]                      ;otherwise the first element is bound to front
                                                     ;and the rest of the elements is bound to rest.
          (queuees ← rest)                                      ;the contents of queuees is updated.
          (next: front (rest: the-queue-itself))) )) ))))                  ;(next:...) is returned.
```

Figure 6

The idea of this implementation is quite simple. When (cons-impure-queue) is evaluated, an actor Q which knows about a cell is created. This cell will contain the members of Q as a sequence. A more formal and precise description for the idea of this implementation is given as an *implementation contract* in Figure 7.

```
[implementation-contract-for impure-queue ≡
 ((Q  where  {(Q is (IMPURE-QUEUE [!a]))})             ;an queue-actor Q with its queuees [!a]
    is-implemented-as                                                        ;is implemented as
     (Q  with-acquaintances  {queuees, the-queue-itself}     ;an actor with acquaintances {...}
      where {(the-queue-itself ≡ Q)                        ;where Q is bound to the-queue-itself
             (queuees is (CELL S))                     ; queuees is a cell-actor with its content S
             (S is (SEQUENCE [!a]))})))]      ;and S is a sequence-actor with its elements [!a].
```

Figure 7

This implementation contract reads as follows: Q denotes an actor created by evaluating (cons-impure-queue). When Q is an impure queue with its queuees [!a] (namely, (Q *is* (*IMPURE-QUEUE* [!a])) holds), Q is an actor whose *acquaintances* are queuees and the-queue-itself and the following assertions hold: Q is bound to a program variable the-queue-itself (namely, (the-queue-itself ≡ Q) holds), queuees is a cell-actor which contains a sequence-actor S (namely, (queuees *is* (*CELL* S)) holds), and S has !a as its elements (namely, (S *is* (*SEQUENCE* [!a])) holds).

A notation (*CELL* S) is a conceptual representation of a cell-actor which has S as its contents. When the contents of the cell-actor is updated by some other actor, say SS, the conceptual representation of the cell-actor becomes (*CELL* SS). (A contract of cell is found in Appendix II.) Similarly, (*SEQUENCE* [!a]) is a conceptual representation of a sequence-actor whose elements are [!a].

As one might notice, program variables (e.g. queuees,and the-queue-itself) are used in implementation contracts. To avoid the problems of scope rules of program variables, implementation contracts will be inserted between lines of the codes when they are refered to in the course of the symbolic evaluation. A diagram in Figure 8 will encourage the intuitive understanding of this implementation. Arrows in the diagram indicate the knows-about relation.

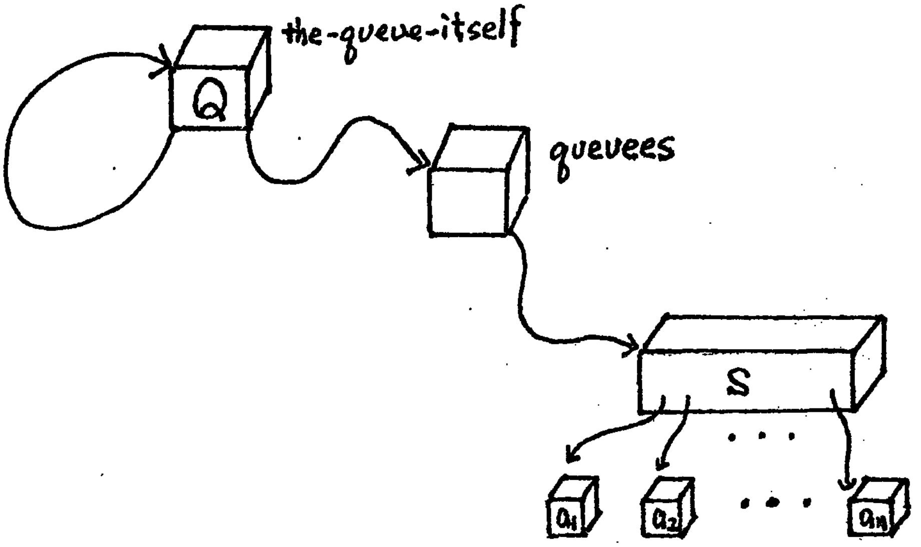


Figure 8

The conceptual representation of impure queues (*IMPURE-QUEUE* [!a])) characterized in the contract in Figure 2 is used in the above implementation contract. There is virtually an infinite number of different implementations which could be represented by this (*IMPURE-QUEUE* [!a]). The implementation contract in Figure 7 specifies how this particular implementation in Figure 6 (and/or in Appendix IV) is realized. The use of the implementation contracts allows us to introduce a hierarchy of the conceptual representations of actors which behave as data-structures. This is another useful technique for data abstraction [Liskov & Zilles 1975]. The implementation contract in Figure 7 should not be public to those who are only interested in the external behavior of the modules. For external uses only the ordinary contracts such as the one for impure queues in Figure 2 and the one for (*empty...into...*) in Figure 3 are sufficient. But the implementation contracts should be available to those who are concerned with how modules are internally represented and how they work. The topics in the subsequent sections are for such people.

SYMBOLIC-EVALUATION OF IMPLEMENTATION OF IMPURE-QUEUE

Now we proceed to the symbolic-evaluation of **cons-impure-queue** in Figure 6 against its contract in Figure 2. But this time our emphasis is not only on the correctness of the code, but also on the internal structure of the code which will be exposed during the process of symbolic-evaluation. The code of **cons-impure-queue** augumented with the situation symbols and the intention statement are depicted in Figure 9. Notice that The implementation contract given in Figure 7 is inserted in Figure 9 as an intention statement. The following three cases, namely, creation, enqueuing and dequeuing, will be treated separately. We follow the same convention for situational tags of assertions as in the symbolic-evaluation of *(empty...into...)*. Instead of attaching a situational tag to each assertion, the notation *in* $S_{...}$ will be used.

```
 - $S_{pre-creation}$ -
((cons-impure-queue) ≡
  (let (queuees = (cons-cell []))               ;a cell which contains an empty sequence is created
     then
     - $S_{8}$ -
         - $S_{nq-or-dq-initial}$
     (Intention:
           ((Q  where  {(Q is (IMPURE-QUEUE [!a]))})
                is-implemented-as
                     (Q with-acquaintances {queuees, the-queue-itself}
                          where {(the-queue-itself ≡ Q)
                                 (queuees is (CELL S))
                                 (S is (sequence [!a]))}))) -

   (the-queue-itself ≡                          ;a queue-actor is defined as the following case-clause
                                                                    ;and denoted by the-queue-itself.
     (cases
      (≡> (nq: =new-element)         ;whenever an enqueue message with new-element is received,
                   - $S_{nq-8}$ -
        (queuees ← [!$queuees new-element])                                ;new-element is stored
                                                     ;in the cell queuees behind the previous elements.
                   - $S_{nq-final}$ -
        the-queue-itself)                                    ;and then the-queue-itself is returned.

      (≡> (dq:)                                            ;whenever an dequeue message is received,
                   - $S_{dq-8}$ -
       (rules $queuees                                                   ;if the contents of queuees
         (≡> []  (complaint: (exhausted:)))   ;is empty, then the complaint message is returned.
                        - $S_{dq-a-final}$ -
         (≡> [=front !=rest]                         ;otherwise the first element is bound to front
                                                     ;and the rest of the elements is bound to rest.
                        - $S_{dq-b-8}$ -
          (queuees ← rest)                                  ;the contents of queuees is updated.
                        - $S_{dq-b-final}$ -
          (next: front (rest: the-queue-itself))) )) ))                    ;(next:...) is returned.
    - $S_{post-creation}$ - ))
```

Figure 9

I. CREATION OF impure-queue

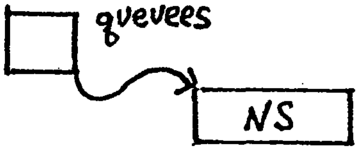
The symbolic-evaluator reads the first *result-of* clause in the contract for impure-queue in Figure 2 and finds that there are no initial assumptions for evaluating (cons-impure-queue). So in the initial situation no assertions are entered in the data base.

in $S_{\text{pre-creation}}$: <u>empty</u>

By the *let* statement a cell which has an empty sequence NS as its contents is created and bound to queuees. NS is newly created identifier.

in S_8 :

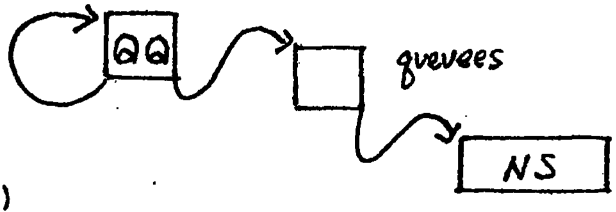
(queuees *is* (*CELL* NS))
(NS *is* (*SEQUENCE* []))



Then in this situation an actor, say QQ, whose script is the code given after (the-queue-itself ≡ ... is returned. So (the-queue-itself ≡ QQ) holds. Furthermore by looking for free variables in the script of QQ, the acquaintances of QQ are found: in this case the acquaintances are queuees and the-queue-itself. To record this, the assertions (QQ *knows-about* queuees) and (QQ *knows-about* the-queue-itself) are used. After QQ is returned the following assertions are entered.

in $S_{\text{post-creation}}$:

(the-queue-itself ≡ QQ)
(QQ *knows-about* queuees)
(QQ *knows-about* the-queue-itself)



What the contract for impure-queue in Figure 2 requires is that the returned actor Q be (Q *is* (*IMPURE-QUEUE* [])). By virtue of the implementation contract, (Q *is* (*IMPURE-QUEUE* [])) is translated into the following assertions. Note that (Q *with-acquaintances* {queuees, the queue-itself}...) in the implementation contract is interpreted as assertions (Q *knows-about* queuees) and (Q *knows-about* the-queue-itself).

(the-queue-itself ≡ Q)
(Q *knows-about* queuees)
(Q *knows-about* the-queue-itself)
(queuees *is* (*CELL* S))
(S *is* (*SEQUENCE* []))

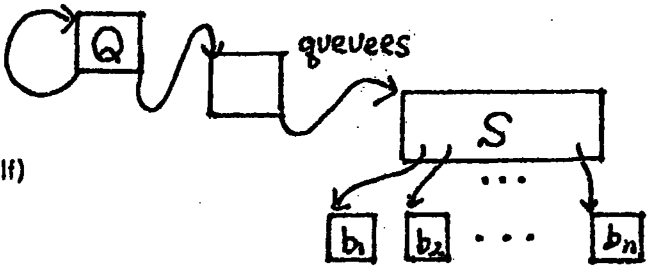
Since the three assertions in $S_{\text{post-creation}}$ and assertions inherited from S_8:
(queuees *is* (*CELL* NS)) and
(NS *is* (*SEQUENCE* []))
match against the above translated assertions, it is concluded that the returned actor QQ has

the correct internal structure prescribed by the implementation contract. So the creation of impure queues is correct.

II. ENQUEUING

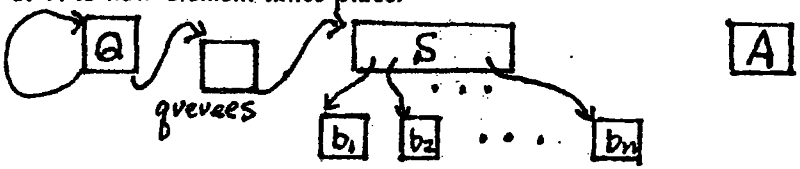
From the contract for impure-queue in Figure 2, it is assumed that (Q *is* (*IMPURE-QUEUE* [!b])) holds in the initial situation. As noted before, the above assertion is translated into the following five assertions and entered in the data base.

in $S_{\text{nq-or-dq-initial}}$:
(the-queue-itself ≡ Q)
(Q *knows-about* queuees)
(Q *knows-about* the-queue-itself)
(queuees *is* (*CELL* S)
(S *is* (*SEQUENCE* [!b]))



Now the message (*nq:* A) is sent to Q. The message sent to Q matches the first clause of the case statement. So the binding of A to new-element takes place.

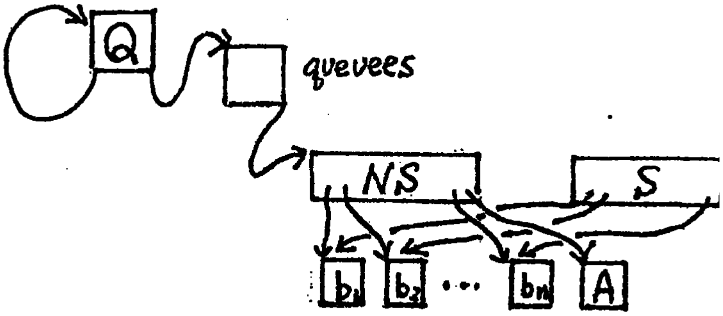
in $S_{\text{nq-0}}$:
(new-element ≡ A)



Then the contents of queuees is updated by a newly created sequence-actor NS with its elements [!$queuees new-element]. $queuees is the contents of queuees in $S_{\text{nq-0}}$, namely, S. !$queuees is the result of the unpack operation on the sequence S. So the new sequence-actor NS is represented as (*SEQUENCE* [!b A]). By the update, (queuees *is* (*CELL* S)) is replaced by (queuees *is* (*CELL* NS)). So the following assertions hold in the next situation.

in $S_{\text{dq-final}}$:
(queuees *is* (*CELL* NS))
(NS *is* (*SEQUENCE* [!b A]))

In this situation Q is returned. Note that the contract for impure-queue in Figure 2 requires that Q is returned and that (Q *is* (*IMPURE-QUEUE* [!b A])) holds. To make all assertions holding in $S_{\text{dq-final}}$ explicit by applying the inheritance rule, the following assertions are obtained.



```
(queuees is (CELL NS))
(NS is (SEQUENCE [!b A]))
(new-element ≡ A)
(the-queue-itself ≡ Q)
(Q knows-about queuees)
(Q knows-about the-queue-itself)
(S is (SEQUENCE [!b])) <-- S is now garbage!!.
```

It is easy to see that the assertions obtained by translating (Q *is* (*IMPURE-QUEUE* [!b A])) through the implementation contract are satisfied by the above assertions. So the enqueuing is correct.

Besides the correctness of the implementation, a very important fact is revealed by the above symbolic-evaluation. The sequence S is created by this implementation and never passed out. S was initially contained in the cell queuees in $S_{\text{nq-initial}}$ but it is not contained by queuees and there are no acquaintances of Q which know about it in $S_{\text{nq-final}}$, namely, S is just floating in the air. Thus there will be no chance for S to be used later. S is subject to the garbage collection. In this implementation every time the enqueuing takes place, a garbage sequence is produced.

III. DEQUEUING

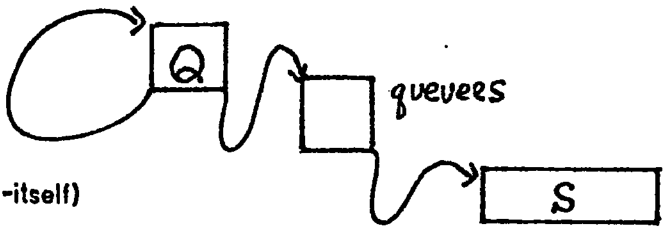
As is indicated in the contract for impure-queue in Figure 2, there are two cases needed to be considered: case 1) where the queue is empty, and case 2) where the queue is not empty.

Case 1)

Assuming that (Q *is* (*IMPURE-QUEUE* [])) holds in the initial situation, the implementation contract in the code specifies the following assumption in the same manner as before.

in $S_{\text{nq-or-dq-initial}}$:

```
    (the-queue-itself ≡ Q)
    (Q knows-about queuees)
    (Q knows-about the-queue-itself)
    (queuees is (CELL S))
    (S is (SEQUENCE []))
```



Now the (*dq:*) message is sent to Q. Then the message matches against the second clause of the case statement. Since no binding takes place, all assertions are inherited from $S_{\text{nq-or-dq-initial}}$ to $S_{\text{dq-0}}$.

$S_{dq-0} = S_{nq-or-dq-initial}$

Since the contents of queuees is S which is an empty sequence, we reach the situation $S_{dq-a-final}$.

$S_{dq-a-final} = S_{dq-0}$

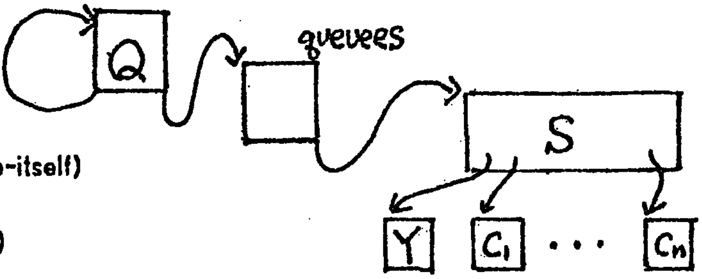
Then the (*complaint:* (*exhausted:*)) message is returned and since all assertions assumed in $S_{nq-or-dq-initial}$ hold in $S_{dq-a-final}$, Q maintains its legitimate internal structure for which (*IMPURE-QUEUE* []) stands. This is what is required.

Case 2)

For this case, (Q *is* (*impure-queue* [Y !c])) is assumed in the contract in Figure 2 and so the translated assertions are as follows.

in $S_{nq-or-dq-initial}$

(the-queue-itself ≡ Q)
(Q *knows-about* queuees)
(Q *knows-about* the-queue-itself)
(queuees *is* (*CELL* S))
(S *is* (*SEQUENCE* [Y !c]))



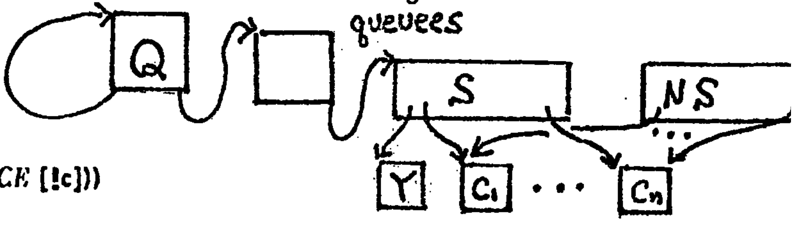
Then all assertions are inherited from $S_{nq-or-dq-initial}$ to S_{dq-0} as in case 1).

$S_{dq-0} = S_{nq-or-dq-initial}$

The contents of queuees which is S matches the pattern [=front !=rest], because (S *is* (*SEQUENCE* [Y !c])) holds. By this matching a new sequence, say NS, with its elements [!c] is created and bound to rest and Y is bound to front. So the binding information is added in the next situation.

in S_{dq-b-0} :

(front ≡ Y)
(rest ≡ NS) (NS *is* (*SEQUENCE* [!c]))



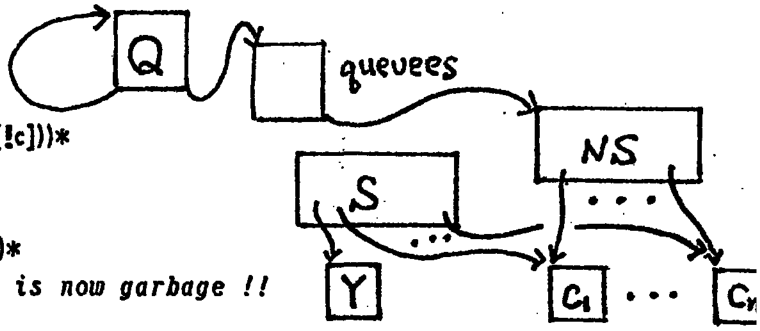
After the updating of the content of queuees, (queuees *is* (*CELL* S)) is replaced by (queuees *is* (*CELL* NS)).

in $S_{dq-b-final}$:

(queuees *is* (*CELL* NS))

All assertions which hold in this situation are as follows:

(queuees *is* (*CELL* NS))*
(front ≡ Y)
(rest ≡ NS) (NS *is* (*SEQUENCE* [!c]))*
(the-queue-itself ≡ Q)*
(Q *knows-about* queuees)*
(Q *knows-about* the-queue-itself)*
(S *is* (*SEQUENCE* [Y !c])) <-- S *is now garbage !!*



In this situation (*next:* Y (*rest:* Q)) is returned. The five assertions marked with * guarantee that (Q *is* (*IMPURE-QUEUE* [!c])) holds through the implementation contract. So all cases are proved.

As in the case of the enqueuing, it is also revealed by the symbolic-evaluation that a sequence S initially contained in queuees becomes a garbage sequence at the end of the dequeuing.

REFINEMENTS OF IMPLEMENTATION OF IMPURE-QUEUE

The preceding symbolic-evaluation have revealed that the implementation based on the implementation contract in Figure 7 turns out to be very inefficient in terms of the amount of space to be consumed. In order to save the wasted space, let us consider the following refinement of the implementation. A specific cause of the inefficiency is that every time the enqueuing and dequeuing take place (except in the case where the queue is empty when it gets (*dq:*) message), the previous contents of the cell queuees (namely, a sequence-actor which contains the members of the queuee) becomes garbage and that a completely new sequence has to be created and be put in the cell queuees. To alleviate this deficiency, instead of keeping all the members of the queue in a single long sequence-actor, we try another implementation which keeps the current members of the queue in a chain of short sequence-actors each of which contains only one member of the queue. Additionally, we will use two cells, say front-cell and rear-cell, one of which maintains the front end of that chain and the other the rear end.

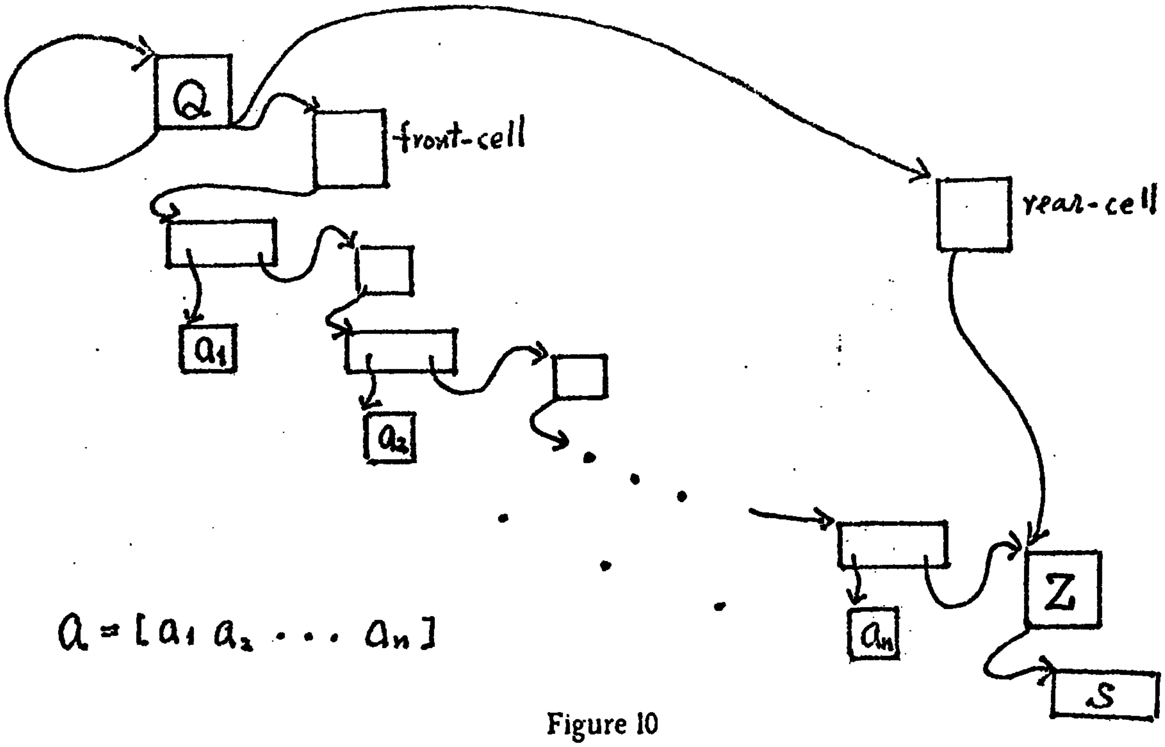


Figure 10

A diagram for this new implementation and its formal description (.i.e implementation contract) are given in Figure 10 and Figure 11, respectively.

```
[implementation-contract-for impure-queue-a ≡
   ((Q where {(Q is (IMPURE-QUEUE [!a]))})
        is-implemented-as
           (Q with-acquaintances {front-cell, rear-cell, the-queue-itself}
                 where {(front-cell contains [!a] with-end Z)
                          (rear-cell is (CELL Z))
                          (Z is (CELL S)) (S is (SEQUENCE []))})})

   (to-simplify
           {(front-cell contains [] with-end Z)}
        try
           {(front-cell eq Z)
              (front-cell is (CELL S)) (S is (SEQUENCE []))})

   (to-simplify
           {(front-cell contains [A !y] with-end Z)}
        try
           {(front-cell is (CELL S))
              (S is (SEQUENCE [A C]))
              (C contains [!y] with-end Z)})

   (to-simplify
           {(front-cell contains [!y A] with-end Z)}
        try
           {(front-cell contains [!y] with-end Y)
              (Y is (CELL S))
              (S is (SEQUENCE [A Z]))}) )]
```

Figure 11

To-simplify clauses in this implementation contract are the characterizations of the assertion (front-cell *contains* [!a] *with-end* Z). An important property stated in this characterization which is difficult to see in the diagram in Figure 10 is that if [!a] is [], as stated in the first *to-simplify* clause, front-cell and Z turn out to be the same actor. In fact, the assertion (front-cell *contains* [!a] *with-end* Z) can be viewed as abstractly representing a cell front-cell which contains the chain of short sequences. The role of the cell front-cell is quite similar to that of the cell queuees in the previous implementation. The cell rear-cell is introduced to quickly get to the end of the chain when the enqueuing is being accomplished, instead of tracing down the chain from the front end. In Figure 12 is the PLASMA code which realizes this new implementation idea. More details will be revealed by the symbolic-evaluation of this code.

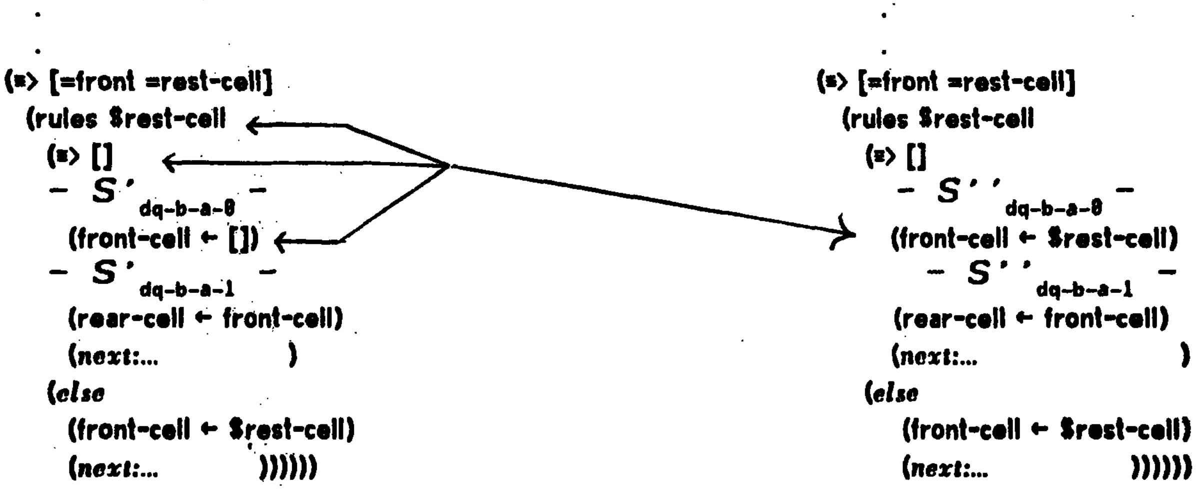
```
((cons-impure-queue-a) ≡
  (let (front-cell = (cons-cell []))                      ;a cell with an empty sequence is created.
   then
    (let (rear-cell = (cons-cell front-cell))                                ;a cell which contains
     then                                                ;the actor bound to front-cell is created.
     (the-queue-itself ≡                   ;a queue-actor is defined as the following cases-clause
       (case                                                      ;and bound to the-queue-itself.
        (≡> (nq: =new-element)                                                    ;whenever (nq:...),
          (let (new-cell = (cons-cell []))       ;a cell-actor with an empty sequence is created.
           then
            ($rear-cell ← [new-element new-cell])          ;the contents of the contents of rear-cell
                                                         ;is updated by a newly created sequence
                                                        with its elements new-element and new-cell.
            (rear-cell ← new-cell)                                 ;the contents of rear-cell is updated.
            the-queue-itself))                    ;the actor bound to the-queue-itself is returned.

        (≡> (dq:)                                                         ;whenever (dq:...) is received
          (rules $front-cell                                            ;if the content of front-cell
            (≡> []                                                              ;is an empty sequence,
              (complaint: (exhausted:))                                 ;then the complaint is returned.
            (≡> [=front =rest-cell]                             ;if it is a sequence of two elements,
              (rules $rest-cell                                  ;then check the content of rest-cell.
                (≡> []                                                      ;if it is an empty sequence,
                  (front-cell ← [])                                ;the contents of front-cell is updated.
                  (rear-cell ← front-cell)                          ;the contents of rear-cell is updated.
                  (next: front (rest: the-queue-itself)))                       ;(next:...) is returned.
                (else                                                                          ;otherwise
                  (front-cell ← $rest-cell)                     ;the contents of front-cell is replaced
                                                                          by the contents of rest-cell.
                  (next: front (rest: the-queue-itself)) ))) ))    ))    )))       ;(next:...) is returned.
```

Figure 12

However, instead of going through the symbolic-evaluation here, we restrict ourselves to stating some interesting observations obtained by the symbolic-evaluation. One observation is the similarity between two implementations. First of all, both implementations are behaviorally equivalent, because both satisfy the contract for impure queues in Figure 2. Furthermore, the clause (≡> (*nq*:...)...) in the second implementation (see the expression inside the upper solid rectangle) are behaviorally equivalent to the clause ((≡> (nq:...)...) in the first implementation in Figure 6. The same is true for ((≡> (*dq*:)...) clauses in both implementations. As pointed out before, the role of queuee in the first implementation and that of front-cell are similar. So when a (*dq*:) message is received, in order to check for whether the queue is empty or not, the contents of **queuee** is tested in one implementation and the contents of **front-cell** is tested in the other. Since the first implementation has been already shown to be correct, these similarities make the justification of the second implementation easier.

Another interesting observation is that the result of symbolic-evaluation of the second implementation suggests us the optimization of the code in Figure 12. In Figure 13 a relevant part of the code in Figure 12 is given. In $S'_{\text{dq-b-a-0}}$, **\$rest-cell** (i.e. the contents of **rest-cell**) is an empty sequence and in the next situation $S'_{\text{dq-b-a-1}}$ the contents of **front-cell** becomes an empty sequence by **(front-cell ← [])**. So the replacement of **(front-cell ← [])** by **(front-cell ← \$rest-cell)** does not change the assertions which hold in $S_{\text{dq-b-a-1}}$.



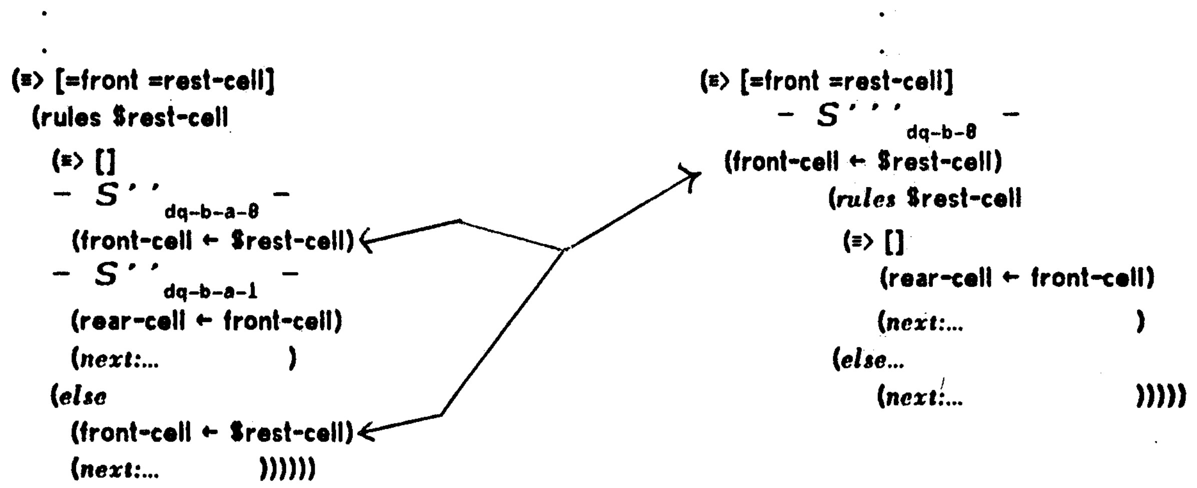
```
.
.
(≡> [=front =rest-cell]
  (rules $rest-cell
   (≡> []
   - S'            -
       dq-b-a-0
    (front-cell ← [])
   - S'             -
       dq-b-a-1
    (rear-cell ← front-cell)
    (next:...           )
   (else
    (front-cell ← $rest-cell)
    (next:...         ))))))
```

```
.
.
(≡> [=front =rest-cell]
  (rules $rest-cell
   (≡> []
    - S''            -
        dq-b-a-0
     (front-cell ← $rest-cell)
      - S''            -
          dq-b-a-1
     (rear-cell ← front-cell)
     (next:...               )
   (else
     (front-cell ← $rest-cell)
     (next:...           ))))))
```

Figure 13

This fact is easily checked by starting the symbolic-evaluation from $S''_{\text{dq-b-a-0}}$ in Figure 13 and observing the assertions which hold in $S''_{\text{dq-b-a-1}}$. This is a simple example of applying symbolic-evaluation for checking the implications of changes in codes.

After the above replacement is made, the first statements in the two clauses of the ***rules*** statement are identical. So now two identical statements can be collapsed together and can be pushed up before the **(rules...)** (See Figure 14.). To justify this change, again the symbolic-evaluation starts from $S'''_{\text{dq-b-0}}$.



```
(=> [=front =rest-cell]
  (rules $rest-cell
   (=> []
    - S''      -
         dq-b-a-8
     (front-cell ← $rest-cell)
    - S''      -
         dq-b-a-1
     (rear-cell ← front-cell)
     (next:...          )
   (else
     (front-cell ← $rest-cell)
     (next:...        ))))))
```

```
(=> [=front =rest-cell]
      - S'''      -
            dq-b-8
  (front-cell ← $rest-cell)
          (rules $rest-cell
           (=> []
            (rear-cell ← front-cell)
            (next:...              )
           (else...
            (next:...              )))))
```

Figure 14

As suggested by the examples above, the use of symbolic-evaluation for analysis of the implications of changes in codes (*perturbation analysis*) provides with us a veryuseful tool in the process of debugging and optimization.

FURTHER WORK

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 impure actor programs 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. The following examples are more sophisticated domains where the idea is expected to be successfully extended.

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 [Yonezawa 1976]. We then hope to symbolically evaluate implementations of these proposed algorithms using the notion of situations and show their correctness.

Another example we plan to pursue is the problem of writing a specification for a file system in a time-sharing environment. 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 express in a procedural specification. We will try to formulate this problem using the notion of situations in hopes of clarifying the kinds of specifications that can be used for such problems.

Furthermore we believe that the above examples are good research targets for the Programming Apprentice to extend its domain of application to programs with parallelism and synchronization.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Carl Hewitt who suggested this topic, and pruned irrelevant limbs on to which I was about to climb during this research. The comments and suggestions of B. Liskov and J. Sussman were valuable. Thanks are also due to Chuck Rich, Ron Pankiewicz, Keith Nishihara and Guy L. Steele Jr. who carefully read the early version of this paper and made comments. Russ Atkinson was helpful in writing the CLU code in Appendix III.

This research was conducted at the Artificial Intelligence Laboratory and Laboratory for Computer Science (formerly Project MAC), Massachusetts Institute of Technology under the sponsorship of the Office of Naval Research, contract number N00014-75C0522.

BIBLIOGRAPHY

Boyer, R.S. and Moore, J.S. "Proving Theorems about LISP Functions" JACM. Vol.22. No.1. January, 1975.

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

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

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

Dahl, O. J., Nygaard, K., and Myhrhang, B. SIMULA 67 Common Base Language Norwegian Computing Center, Forskningsveien 1B, Oslo, 1968

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. American Mathematical Society, Providence Rhode Island. 1967.

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

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

Guttag, J. "Abstract Data Types and the Development of Data Structures" Proc. of ACM SIGPLAN-SIGMOD Conference, Salt Lake City, Utah. March, 1976.

Hewitt, C.E. "How to Use What You Know" Proc. of International Joint Conference on Artificial Intelligence U.S.S.R. September, 1975.

Hewitt, C.E. "Viewing Control Structures as Patterns of Message Passing" Working paper 92, Artificial Intelligence Laboratory, MIT revised in March, 1976.

Hewitt, C.E et.al. "Actor Induction and Meta-evaluation" Conference Record of ACM Symposium on Principles of Programming Languages. 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. "Proof of Correctness of Data Representation" Acta Informatica Vol. 1. pp271-281. 1972

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.

Liskov, B "A Note on CLU" Memo 112. Computation Structure Group, Laboratory for Computer Science (formerly Project MAC) MIT, November, 1974

Liskov, B. and Zilles, S. N. "Specification Techiniques for Data Abstractions" IEEE transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Rich, C. and Shrobe, H.E. "Understanding Lisp Programs: Towards a Programmer's Apprentice" Masters' Thesis, Electrical Engineering and Computer Science, MIT August, 1975.

Schaffert, C., Snyder, A. and Atkinson, R. "The CLU Reference Manual" Laboratory for Computer Science (formaly Project MAC), MIT September, 1975

Smith, B. and Hewitt, C.E. "A PLASMA PRIMER" Working Paper in preparation. Artificial Intelligence Laboratory, MIT.

Spitzen, J. and Wegbreit, B. "The Verification and Synthesis of Data Structures." Acta Informatica 4. 1975.

Steele, G.L. "Multiprocessing Compactifying Garbage Collection" CACM 18. September, 1975.

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

Wegbreit, B. and Sptizen, J. M. "Proving Properties of Complex Data Structures" to appear in JACM, 1976.

Wulf, W. A. "ALPHARD: Towards a Language to Support Structured Programming" Department of Computer Science, Carnegie-Mellon University, April 1974.

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

Yonezawa, A. "A Specification Language for Synchronization Problems" Working paper in preparation Arificial Intelligence Laboratory, MIT.

Zilles, S. N. "Abstract Specifications for Data Types" IBM Research Laboratory, San Jose, California. 1975.

APPENDIX I DERIVATION OF AXIOM II FROM CONTRACT FOR PURE QUEUE

We assume the following correspondences between notations in the contract for pure queues and the algebraic specification of pure queues.

c1) (cons-pure-queue) <----> Cons-queue
c2) (Q <= (*nq:* a)) <---> Enqueue(Q, a)
c3) (Q <= (*dq:*)) <---> Dequeue(Q)
c4) (*next:* a (*rest:* Q)) <---> <a, Q>

The axiom II) is derived from the contract for pure queues as follows:

1) Dequeue(q) = <b, q'> *;given as the premise of the axiom* II).
2) ((*PURE-QUEUE* [b !x]) <= (*dq:*)) = (*next:* b (*rest:* (*PURE-QUEUE* [!x])) *;from* D).
3) q <---> (*PURE-QUEUE* [b !x]) *;from 1) and 2).*
4) q' <---> (*PURE-QUEUE* [!x]) *;from 1) and 2).*
5) Dequeue(Enqueue(q,a)) *;the left side of the axiom* II).
<---> (((*PURE-QUEUE* [b !x]) <= (*nq:* a)) <= (*dq:*)) *;from 3) c2) and c3).*
= ((*PURE-QUEUE* [b !x a]) <= (*dq:*)) *;from* B).
= (*next:* b (*rest:* (*PURE-QUEUE* [!x a]))) *;from* D)
= (*next:* b (*rest:* ((*PURE-QUEUE* [!x]) <= (*nq:* a))))) *;from* B).
<---> (*next:* b (*rest:* Enqueue(q', a))) *;from 4) and c2).*
<---> <b, Enqueue(q', a)> *;from c4).*
Therefore Dequeue(Enqueue(q, a)) = <b, Enqueue(q', a))>. *q.e.d.*

APPENDIX II. A CONTRACT FOR CELLS

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

   (result-of
         ((C <= (contents?))  where  {(C is (CELL B))})
       is
         B)

   (result-of
         (((C ← =D)  where  {(C is (CELL E))})
       is
         (C  where  {(C is (CELL D))})))]
```

APPENDIX III A CONTRACT FOR "AVERAGE"

Let us consider how a contract for another type of actor is written in our formalism. In this appendix, we focus on actors whose behavior depends upon the history of their incoming messages. Obviously such actors are impure. An example of actors of this type is the "**average**" actor. It receives a (*new-element:* X) message which contains a number X, and a message (*average?*) which asks for the average of all the numbers which have been sent to it. Figure 15 below is a contract for this actor.

```
[contract-for average ≡
  ((cons-average <= (initial-element: A)) creates-an-actor
                 (D  where  {(D has (HISTORY [A]))})

  (result-of
    ((D <= (new-element:  =B))
              where  {(D has (HISTORY [!a]))})
       is
         (D
                 where  {(D has (HISTORY [!a B]))}))

  (result-of
    ((D <= (average?))
         where {(D has (HISTORY [!b]))})
       is
         (average [!b]))

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

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

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

Figure 15

The idea is simple. We introduced a property that the actor D has a history of incoming messages !a and expressed it in the notation (D *has* (*HISTORY* [!a])). Again (*HISTORY* [!a]) is an example of the conceptual representations. This idea is similar to that of M. Clint[1973] who introduced a "mythical pushdown stack" to have the history recorded. The characterization of the notation (*average* !b) used in the contract is given the above contract.

APPENDIX IV AN IMPLEMENTATION OF IMPURE QUEUES IN CLU

```
%
%
impure-queue = cluster is cons-impure-queue, nq, dq;
%
returned-package = record[next: any, rest: impure-queue];
complaints = record[complaint: string];
%
%
%
cons-impure-queue = oper() returns(cvt);
   return cell$cons(nil);
end cons-impure-queue;
%
%
%
nq = oper(a-queue: cvt, new-element: any) returns(cvt);
   old-queuees: sequence:= cell$contents(a-queue);
   cell$update(a-queue, sequence$cons([sequence$unpack(old-queuees), new-element]));
   return a-queue;
end nq;
%
%
%
dq = oper(a-queue: cvt) returns (union[complaint, returned-package]);
   queuees: sequence:= cell$contents(a-queue);
   if queuees = nil then return {complaint: 'exhausted};
   front: any:= sequence$first(queuees);
   rest: any:= sequence$but-first(queuees);
   cell$update(a-queue, rest);
   return {next: front, rest: a-queue};
end dq;
%
end impure-queue;
%
%
```