MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I.Memo 399 December 1976

SYMBOLIC EVALUATION USING CONCEPTUAL REPRESENTATIONS FOR PROGRAMS WITH SIDE-EFFECTS

AKINORI YONEZAWA

and

CARL HEWITT

ABSTRACT

Symbolic evaluation is a process which abstractly evaluates an program on abstract data. A formalism based on *conceptual representations* is proposed as a specification language for programs with *side-effects*. Relations between algebraic specifications and specifications based on conceptual representations are discussed and limitations of the current algebraic specification techniques are pointed out. Symbolic evaluation is carried out with explicit use of a notion of *situations*. Uses of *situational tags* in assertions make it possible to state relations about properties of objects in different situations. The proposed formalism can deal with problems of side-effects which have been beyond the scope of Floyd-Hoare proof rules and give a solution to McCarthy's frame problem.

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

I. 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 has 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, King 1976]. The main purpose of symbolic evaluation is not simply to verify that implementations meet their specification. Rather the purpose is to provide sufficient information for answering questions about dependencies between program modules in order to understand implications of the proposed changes in both specifications and implementations. To accomplish these purposes, symbolic evaluation must be based on an adequate coherent formalism which consists of:

1) a formal language for writing specifications of modules (procedures or data structures) which may change their behavior (side-effects) [Burstall 1972, Greif & Hewitt 1975, Yonezawa 1975],
2) a formal system for reasoning in situations where side-effects are involved, and
3) a formal language for expressing commentary about modules.

Since the Programming Apprentice should provide an environment where programmers can easily communicate and cooperate with the system in an interactive manner, it is equally important that the formalism used in the Programming Apprentice should be intuitively clear and easily understood by programmers. This paper describes a formalism which attempts to meet these requirements.

Our research is closely related to the research on program verification systems. Previous program verification systems[King 1969, Deutch 1973, Igarashi London & Luckham 1973, Suzuki 1974, Boyer & Moore 1975, Good London & Bledsoe 1975, King 1976] have been limited in their ability to deal with programs which change their behavior (side-effects) because of the inadequacy of the formal systems on which these implementations were based. Furthermore, previous research on algebraic and axiomatic techniques [Hoare 1972, Spitzen & Wegbreit 1975, Zilles 1975, Guttag 1976, Wegbreit & Spitzen 1976] for specifying data-structures has concentrated on data-structures without 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 more storage. Also there is a certain type of side-effect in communication between concurrent processes which is impossible to realize without side-effects. In the early sections of this paper, we will present a new formalism which can deal with side-effects and discuss limitations of previous work on program verification and algebraic (or axiomatic) specification techniques. Then

in the later sections, we will illustrate how the symbolic evaluation is carried in our formalism.

2. CONCEPTUAL REPRESENTATIONS

In this section we introduce the idea of conceptual representations as an important part of our specification language. Conceptual representations can be used for specifying a wide range of data structures at various levels of abstraction[Liskov & Zilles 1975]. The basic motivation of conceptual representations is to aid in providing a specification language which serves as a good interface between programmers and the computer and also between users and implementors. A "good" interface language should allow programmers to easily express and understand their intuitive concept of a data structure and how it behaves for various operations. In this sense the "language" of diagrams using boxes and arrows is a very good language in which people can exchange their intuitive ideas, but it is not rigourous and unambiguous enough for the computer to understand without many hidden assumptions. Conceptual representations can be viewed as a language which can express graphical specifications of data structures. Different degrees of awareness about the implementation of a data structure are required in the different activities of implementing a system such as the initial design, coding, and the subsequent evolution. Conceptual representations should be flexible enough to express just the details that are important in each activities. In this section we give a simple example of the use of conceptual representations.

Consider queues as an illustrative example. Programmers envisage a queue a linear sequence of elements which are enqueued at one end and dequeued at the other end. Suppose we have a queue consisting three elements, a, b, and c, where a is its front element and c is its last element. We can use the following notation for the conceptual representation of this queue.

(*QUEUE* a b c)

To express a queue which has an indefinite number (including zero) of elements, the notation:

(*QUEUE* !x)

is used where !x is an abbreviation of the "unpack" operation on x whose result is equivalent to writing out all of elements denoted by x individually. For example, if x denotes a sequence [b c d], then the sequence [a !x] is equivalent to [a b c d] whereas the sequence [a x] is equivalent to [a [b c d]]. Thus (*QUEUE* !x) is equivalent to (*QUEUE* b c d). If y denotes an empty sequence [], then (*QUEUE* !y) is equivalent to (*QUEUE*) which is the conceptual representation of an empty queue. More elaborate example are

(*QUEUE* a !z) and (*QUEUE* !z b)

which are the conceptual representations of queues with a as its front element and with b as its last element, respectively.

A complete specification of queues is given by describing how a queue is created and how it behaves when certain legitimate operations are applied. First, we specify the domains and ranges of the operations:

f1. create-queue: ---> *queue*
f2. enqueue: *queue* x *item* ---> *queue*
f3. dequeue: *queue* ---> *item* x *queue* *or* *error*
f4. is-empty: *queue* ---> *boolean*

The following is a formal specification of queues using the conceptual representations.

c1. create-queue() = (*QUEUE*)

c2. enqueue((*QUEUE* !x), A) = (*QUEUE* !x A)

c3. dequeue((*QUEUE*)) = ERROR: attempt-to-dequeue-empty-queue

c4. dequeue((*QUEUE* A !x)) = ⟨A , (*QUEUE* !x)⟩

c5. is-empty((*QUEUE*)) = *true*

c6. is-empty((*QUEUE* A !x)) = *false*

Previous investigators have used algebraic and axiomatic techniques for the formal specifications of data structures [Hoare 1972, Sptizen & Wegbreit 1975, Zilles 1975, Guttag 1976, Wegbreit & Spitzen 1976]. To compare it with our approach, we present an algebraic specification of queues.

a1. is-empty(create-queue()) = *true*

a2. is-empty(enqueue(q, A)) = *false*

a3. dequeue(create-queue()) = ERROR: attempt-to-dequeue-empty-queue

a4. *if* is-empty(q) *then* dequeue(enqueue(q, A)) = ⟨A, q⟩

a5. *if* ¬is-empty(q) ∧ dequeue(q) = ⟨B, q'⟩
then dequeue(enqueue(q, A)) = ⟨B, enqueue(q', A)⟩

The above axioms are easily derived from our specification of queues based on conceptual representations. (For the derivation of the axiom a5, see Appendix I.) We believe that specifications using conceptual representations are often easier for programmers to both write and understand than algebraic or axiomatic specifications. Because the conceptual representation approach directly describes what effects take place in data structures (at the conceptual level) when the operations are applied, whereas the algebraic and axiomatic specifications describes the effects of the operations indirectly in terms of relations (or equations) among the operations as illustrated in the above axioms a1 to a5. Such indirect specifications are often difficult to grasp. Thus the author or reader of such an indirect specification of a data type D tends to be less confident whether or not the specification completely describes the desired properties of the data type D. Furthermore the current algebraic and axiomatic techniques do not capture an important difference between data structures without side-effects and data structures with side-effects. (This difference will be explained in Section 3 and Section 4.) As we will see in Section 6, the specification technique using conceptual representations can easily capture this difference.

Besides queues discussed above, conceptual representations of other simple data structure such as cells and sequences will be used in this paper. More detailed discussions and examples of the conceptual representations of more complicated data structures such as sets, bags, arrays, lists, symbol tables, file directories e.t.c. will be found in [Yonezawa 1977].

3. ACTORS, PURE AND IMPURE

Since our symbolic evaluation is based on the actor model of computation [Hewitt and Baker 1976, Greif & Hewitt 1975], 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. Actors interact by sending 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 description of how it should behave when it is sent a message. Its acquaintances are a finite set of actors that it directly *knows about*. If an actor A 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 data and 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 1976a]. Thus a complete model of computation can be constructed with a system of actors. An implementation of this model is realized as a programming language PLASMA[Hewitt 1976b].

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 with time. 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 behaves like a mathematical function. The only primitive impure actor we will use is the "cell". A cell-actor accepts a message (*update:* v) 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 a pure actor is a sequence-actor. One can retrieve elements of a sequence-actor, but cannot change its elements; instead 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 shared record. This type of side-effect 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 Section 5, serious problems arise when impure actors which behave like data structures are shared between actors.

4. PURE QUEUES AND IMPURE QUEUES

In this section, we introduce a pure actor and an impure actor, both of which behave like a queue. Using them as an illustrative example we discuss the problems of previous work on program verification systems and formal specification techniques in dealing with side-effects. (cf. Section 5)

Both pure queue-actors and impure queue-actors accept the same two kinds of messages: one is (*nq:* x) which is a request to enqueue a new elements x, and the other is (*dq:*) which is a request to take out the front element of the queue and return it with the remaining queue. However if the queue is empty, it returns a complaint message (*exhausted:*). The important difference between a pure queue-actor and an impure queue-actor is whether or not a new queue-actor is created when (*nq:* ...) and (*dq:*) are sent. When (*nq:* x) is sent to a pure queue-actor PQ, a new pure queue-actor PQ' which has x as the last element of the queue is created and returned. The original queue-actor PQ still has the same elements as before. When (*nq:* x) is sent to an impure queue-actor IQ, x is absorbed inside IQ and enqueued at rear of the previous elements. So IQ itself is extended and returned. No new queue-actors are created. (See Figure 1.) When (*dq:*) is sent to a pure queue-actor PQ (which is not empty), a new pure queue-actor PQ' whose elements are all elements of PQ except the front element of PQ is created and the front element of PQ and

the new pure queue-actor PQ' are returned. Again the original PQ is intact and has the same elements as before. When (*dq:*) is sent to IQ (which is not empty), then the front element of IQ and IQ itself which now has the rest of the original elements are returned. No queue-actors are created.

It might be helpful to see LISP analogy in understanding this difference between pure queues and impure queues. Suppose that a queue is implemented as a list L. Then sending (*nq:* x) to a pure queue-actor corresponds to (append L (list x)) whose result is a totally new list constructed from a copy of L and x and sending (*nq:* x) to an impure queue-actor corresponds to (nconc L (list x)) whose result does not consist of a copy of L.

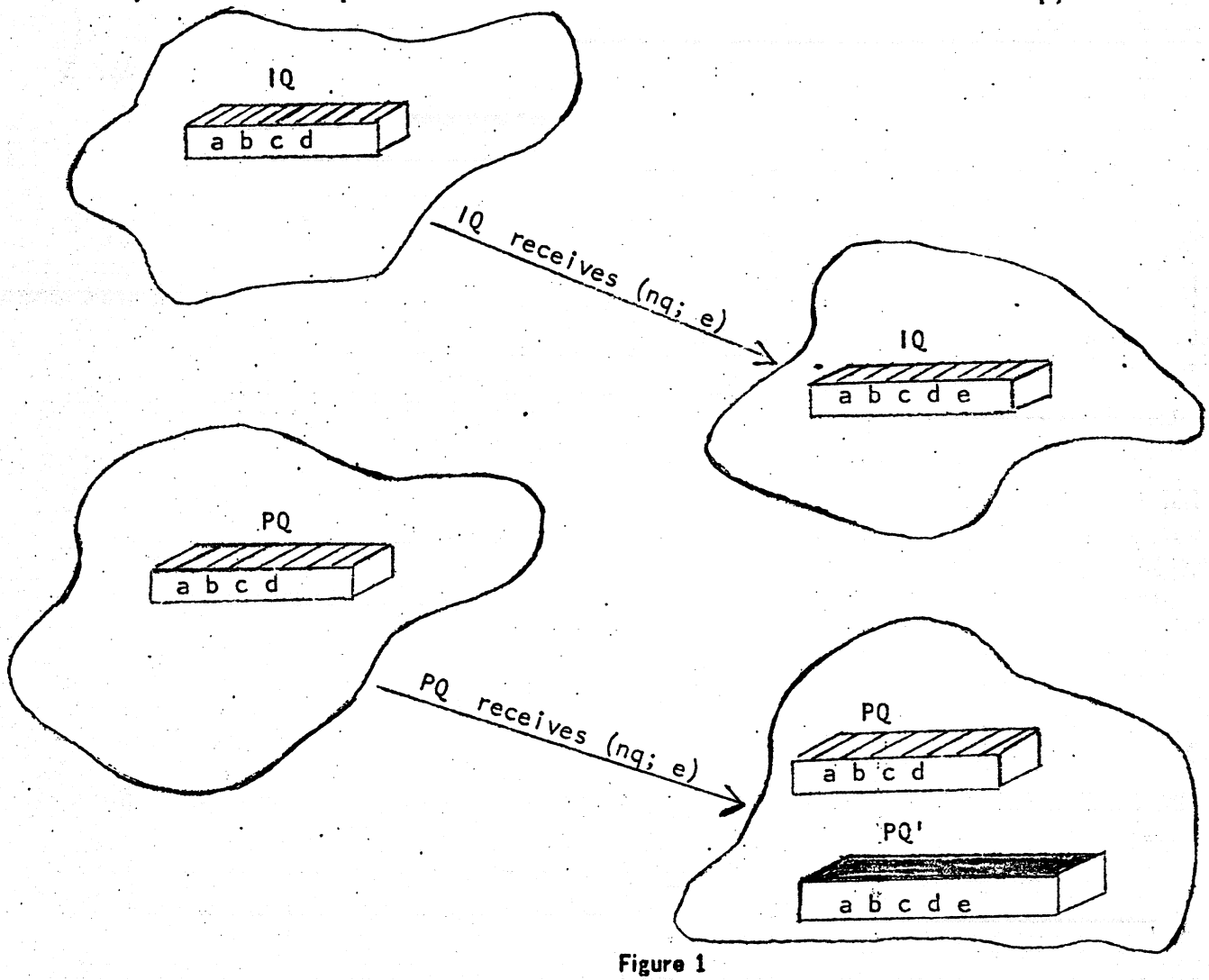


Figure 1

The two formal specifications of queues in Section 2, one based on the conceptual representations and the other by algebraic specifications, do not distinguish the above important difference between impure queues and pure queues. In Section 6 by slightly

extending the interpretation of the conceptual representations, we will give a specification of queues which captures this distinction.

5. SITUATIONS AND SIDE-EFFECTS

In order to deal with behavior that changes with time, we need a powerful coherent formalism for specifications and reasoning. A general technique we employ to cope with this kind of problems is to explicitly introduce a notion of *situation*[McCarthy & Hayes 1969] into the formalism.

A situation is the local state of an actor system at a given moment. (If an actor system has no concurrency, there is no distinction between the local states and the global states of the actor system.) For example, the contents of a cell-actor C changes from time to time according to the update messages which have been most recently sent to C. Suppose that in a situation S the contents of C is 3 and that C receives (*update:* 4) message in S. Then in the situation S' after C receives the message, the contents of C is 4. (See Figure 2.)

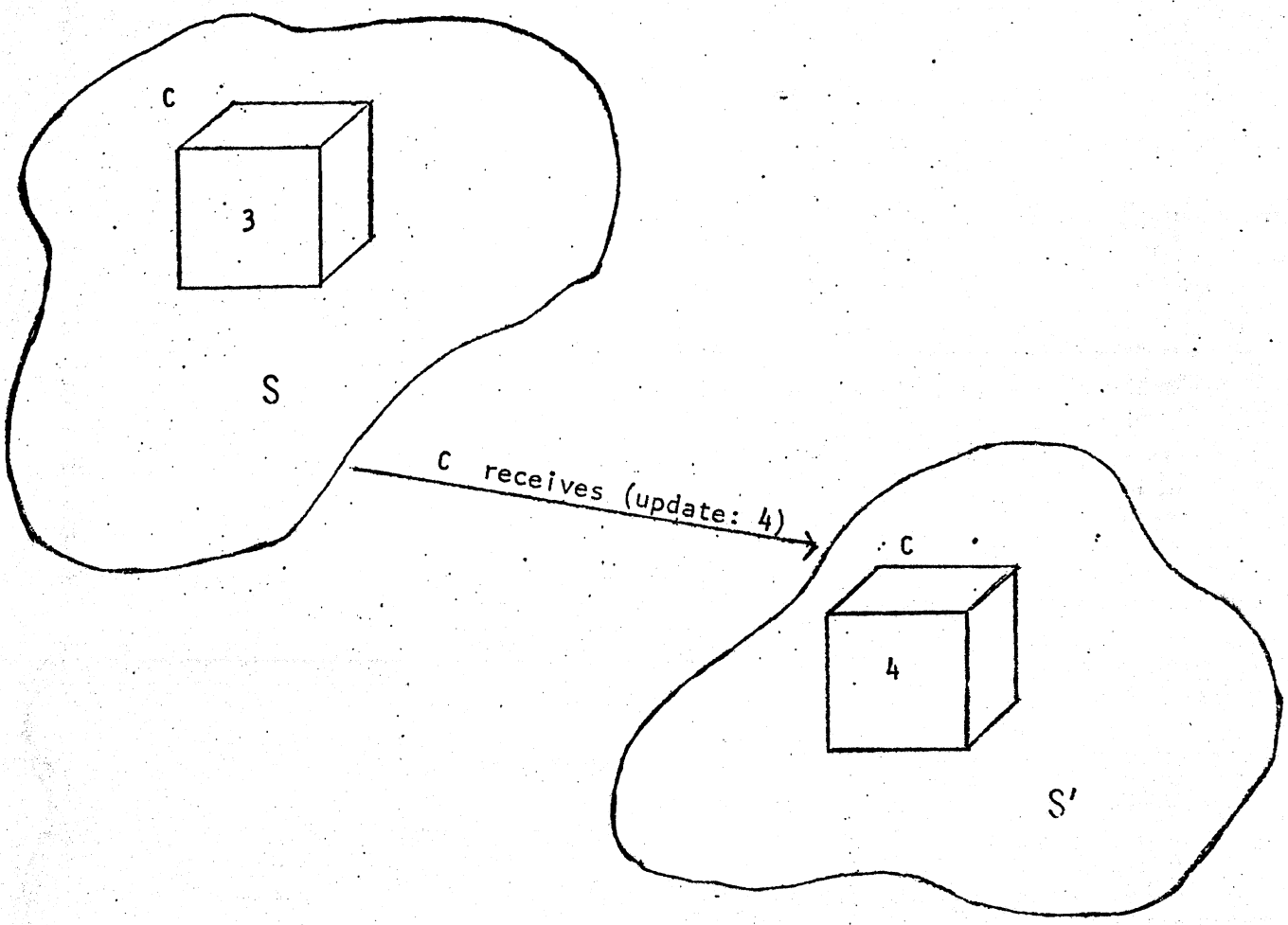


Figure 2

By using a symbol S to denote a situation, we can refer to the contents of C in the situation in the following manner:

(*contents* C) *in* S.

We call a symbol such as S, which is used to refer to a situation, a *situational tag*. In the later sections, we will see how the notion of situations is used in writing specifications of modules (Section 6) and in carrying out symbolic evaluation (Section 8, 10 and 12) where side-effects are involved. In the rest of this section we restrict ourselves to giving simple examples of how the notion of situations is introduced into our formalism.

The previous work on formal proof rules and program verification systems[King 1969, Deutch 1973, Igarashi London & Luckham 1973, Suzuki 1974, Boyer & Moore 1975, Good London & Bledsoe 1975] has difficulties in dealing with the problems of side-effects. To illustrate such difficulties, let us consider the following piece of PLASMA program.

```
(let (queue-1 = (create-queue))                ;an empty queue is created and bound to queue-1
  then
   (let (queue-2 = (create-queue))             ;an empty queue is created and bound to queue-2
    then
     (let (queue-3 = (queue-1 <= (nq: 2)))
                              ;(nq: 2) is sent to queue-1 (i.e. 2 is enqueued at rear of queue-1)
                                                ;and the result is bound to queue-3
       then
        (queue-1 <= (nq: 3))    (nq: 3) is sent to queue-1 (i.e. 3 is enqueued at rear of queue-1.)
                          .
                          . )))
```

Suppose that a verification system is asked whether or not the length of queue-1 is equal to that of queue-3 after the last statement. A usual way of answering this question is to find out which queue-actor queue-1 and queue-3 refer to and get the length of each queue and then compare each length. To do so, the system has to know at least the following things:

1) what kind of queue-actors are created (e.g. pure or impure),
2) whether or not and how queue-actors are shared and
3) which queue-actors are affected by each event (i.e. message passing).

Our formalism that uses the notion of situations can easily deal with this kind of problem. Let S_{post} denote the situation after the last statement in the above code is executed. The question whether or not the length of queue-1 is equal to that of queue-3 in that situation is stated as follows:

((*length* queue-1) *is-equal* (*length* queue-3)) *in* S_{post}.

By distributing the situational tag S_{post}, the same question can be stated in the following two different manners:

((*length* queue-1) *in* S_{post}) *is-equal* ((*length* queue-3) *in* S_{post}) or

(*length* (queue-1 *in* S_{post})) *is-equal* (*length* (queue-3 *in* S_{post}))

Since situational tags allow us to relativize facts, relations between facts holding in different situations can be easily stated in our formalism. For example, an assertion that the length of queue-1 in S_{post} is greater than the length of queue-1 in the situation S_{pre} before the last statement is executed is stated as:

((*length* queue-1) *in* S_{post}) *is-greater* ((*length* queue-1) *in* S_{pre})

This kind of assertion is quite useful to show the termination of programs[Yonezawa 1975]. Furthermore a question about the identity of the queues is easily stated as:

(queue-1 *in* S_{post}) *is-eq* (queue-3 *in* S_{pre})

We will see how our reasoning system deals with these statements in the later sections.

6. CONTRACTS FOR IMPURE QUEUES AND PURE QUEUES

In this section we will illustrate how contracts for impure queues and pure queue can be expressed in our formalism based upon the conceptual representations. We use the term "contract" instead of "specification" to emphasize the fact that it is an agreement or a "treaty" between implementors of a module and its users. Users of a module M should only rely on properties stated in the contract of M. On the other hand, when implementors implement a module M, they need only satisfy the contract of M.(See Figure 3) In the symbolic evaluation of a program which uses a module N, the system should only rely on the properties of N which are derived from the contract of N.

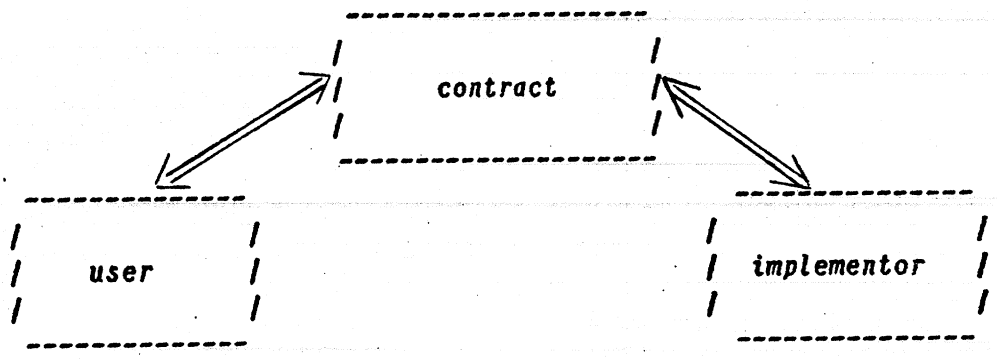


Figure 3

Conceptual representations can be used to capture the difference between the pureness and impureness of queue-actors. To do this the conceptual representation of queues must distinguish the state of a queue-actor in a situation from its identity. Our strategy for capturing the distinction between the pureness and impureness is to let the conceptual representation describe the state of an actor in a particular situation. For example, to state the fact that an impure queue-actor has elements a, b and c in a situation S, we use the following assertion:

(Q *is-a* (*IMPURE-QUEUE* a b c)) *in* S

where Q represents the identity of the queue-actor and (*IMPURE-QUEUE* a b c) is the description of the state of the queue-actor. In order to differentiate the identity of an actor, the predicate "*is-eq*" is used. When there is a queue Q' such that

(Q' *is-a* (*IMPURE-QUEUE* a b c)) *in* S,

it may or may not be true that

(Q *is-eq* Q').

A contract for impure queues includes specifications of events relevant to impure queue-actors. (In the actor model of computation, a transmission of a message actor M to a target actor T is called an "event" and is denoted by the expression (T <= M) borrowed from PLASMA syntax.) In our formalism, a specification of an event is expressed in the following form.

(*event:* (<target-actor> <= <message-actor>)
(*pre-conditions:* ...<assertion>...)
(*returns:* <actor>)
(*post-conditions:* ...<assertion...))

This form states that if the assertions in (*pre-conditions:* ...)-clause hold in the situation where the event takes place, then the assertions in (*post-conditions:*...)-clause hold in the situation where the actor in (*returns:*...)-clause is returned (or sent to the continuation). It should be noted that the above form guarantees the return of the actor.

Using this form, a specification of how an impure queue-actor is created is given as follows.

(*event:* (create-impure-queue <= [])
(*returns:* (*new-actor* Q))
(*post-conditions:* (Q *is-a* (*IMPURE-QUEUE*))))

When an actor create-impure-queue receives an empty sequence [], it returns a new actor Q where Q is an empty impure queue. Since there are no pre-conditions for this event, (*pre-conditions:*...)-clause is not used. The notation (*new-actor* Q) indicates that Q is a newly created actor which is not *eq* (in the LISP sense) to other actors ever created before. (*IMPURE-QUEUE*) is the conceptual representation of empty impure queues.

A specification of the event where an impure queue receives a message (*nq:*...) is as follows.

(*event:* (Q <= (*nq:* A))
(*pre-conditions:* (Q *is-a* (*IMPURE-QUEUE* !x)))
(*returns:* Q)
(*post-conditions:* (Q *is-a* (*IMPURE-QUEUE* !x A))))

Note that the same queue-actor Q is returned and the state of Q has changed as asserted in the (*post-conditions:*...). The specification of the event where an impure queue-actor receives the message (*dq:*) is given in the complete contract of impure queues in Figure 4 below.

Besides specifications of events relevant to the actor concerned, what has to be stated in contracts is the conservation of validity of assertions, from one situation to another, used in the specifications of events. For example, the validity of the assertion (Q *is-a* (*IMPURE-QUEUE* a b c)) does not change after some other queue-actor QQ receives (*dq:*) or (*nq:*..) messages. In other words, such an event does not cause side-effects which change the validity of the assertion. Also the validity of this assertion does not change even after an event where one of the elements of the queue, say b, receives some message, provided that b is not the same actor as Q. To state this kind of conservation of validity in contracts, we list all events which do affect the validity of the assertions using the conceptual representation. This can be viewed as a solution to McCarthy's frame problem[McCarthy & Hayes 1969]. In the case of assertions of the form:

(Q *is-a* (*IMPURE-QUEUE* ...)),

(Q <= (*nq*:...)) and (Q <= (*dq*:)) are such events. No other events affect the conservation of the validity of the assertions using the conceptual representation for impure queues. This is expressed in our formalism as follows.

```
(for-assertion: (Q is-a (IMPURE-QUEUE...))
    (only-affecting-events-are:
            {(Q <= (nq:...)) (Q <= (dq:))}    ))
```

As will been seen in Section 8, 10 and 12, the reasoning in situations where side-effects are involved depends heavily on how statements are affected by going from situation to situation.

```
[contract-for (IMPURE-QUEUE...) ≡

  (event: (create-impure-queue <= [])
      (returns: (new-actor Q) )
      (post-conditions: (Q is-a (IMPURE-QUEUE )) ) )

  (event: (Q <= (nq: A))
      (pre-conditions: (Q is-a (IMPURE-QUEUE !x)) )
      (returns: Q)
      (post-conditions: (Q is-a (IMPURE-QUEUE !x A)) ) )

  (event: (Q <= (dq:))
      (case-1:
        (pre-conditions: (Q is-a (IMPURE-QUEUE )) )
        (returns: (exhausted:) )
        (post-conditions: (Q is-a (IMPURE-QUEUE )) ) )

      (case-2:
        (pre-conditions: (Q is-a (IMPURE-QUEUE B !y)) )
        (returns: (next: B (rest: Q)) )
        (post-conditions: (Q is-a (IMPURE-QUEUE !y)) ) ))

  (for-assertion: (Q is-a (IMPURE-QUEUE...))
      (only-affecting-events-are:
            {(Q <= (nq:...)) (Q <= (dq:))}  )) ]
```

Figure 4

In contrast to the above contract, we give a complete contract for pure queues in

Figure 5.

```
[contract-for (PURE-QUEUE...) ≡

    (event: (create-pure-queue <= [])
        (returns: (new-actor Q) )
        (post-conditions: (Q is-a (PURE-QUEUE )) ) )

    (event: (Q <= (nq: A))
        (pre-conditions: (Q is-a (PURE-QUEUE !x)) )
        (returns: (new-actor Q') )
        (post-conditions:
              (Q' is-a (PURE-QUEUE !x A))
              (Q is-a (PURE-QUEUE !x)) ) )

    (event: (Q <= (dq:))
        (case-1:
          (pre-conditions: (Q is-a (PURE-QUEUE )) )
          (returns: (exhausted:) )
          (post-conditions: (Q is-a (PURE-QUEUE )) )

        (case-2:
          (pre-conditions: (Q is-a (PURE-QUEUE B !y)) )
          (returns: (next: B (rest: (new-actor Q') ))
          (post-conditions:
                (Q' is-a (PURE-QUEUE !y))
                (Q is-a (PURE-QUEUE B !y)) ) )) ]
```

Figure 5

The important fact stated in the above contract is that every time a pure queue-actor receives (*dq:*) or (*nq:...*) messages, a new pure queue-actor is created and that the state of the original queue-actor does not change. In fact the state of each pure queue-actor never changes after it is created, which implies that assertions about the state of pure queue-actors are always valid. Therefore the (*for-assertions:...*)-clause is not necessary in the contract for pure queues. In Appendix II a contract for cell-actors is given in the same formalism.

It should be remarked that since the actor model of computation serves as the underlying semantics of various programming languages such as SIMULA-67[Dahl et al 1968], CLU[Liskov 1974, Schaffert, Snyder & Atkinson 1975], ALPHARD[Wulf 1974] and SMALL TALK[Learning Research Group 1976], these contracts are not biased by the language in which implementations are written, because the contracts can be precisely interpreted in terms of the actor model of computation.

7. A CONTRACT FOR EMPTY-ONE-QUEUE-INTO-ANOTHER

In this section we will give the code and contract for an actor which is supposed to transfer elements (i.e. queuees) of one impure queue to another impure queue. This code and contract will be used to illustrate the symbolic evaluation in Section 10. We present the contract for this actor (Figure 6) before presenting its concrete implementation. Other modules which use empty-one-queue-into-another below should rely only on properties that can be derived from the contract given bellow.

```
[contract-for empty-one-queue-into-another ≡

    (event: (empty-one-queue-into-another <= [Q1  Q2])
        (pre-conditions:
            (Q1 is-a (IMPURE-QUEUE !x1))
            (Q2 is-a (IMPURE-QUEUE !x2))
            (Q1 not-eq Q2) )
        (returns: (done: (emptied: Q1) (extended: Q2))  )
        (post-conditions:
            (Q1 is-a (IMPURE-QUEUE ))
            (Q2 is-a (IMPURE-QUEUE !x2 !x1))  ) ) ]
```

Figure 6

Figure 7 below shows an implementation of this actor in PLASMA. (<name>: <elements>) is an expression which stands for an actor called a "package". Packages are analogous to records in some languages. Examples of packages in Figure 7 are (*dq*:), (*next*:...(*rest*:...)), (*nq*:...) and (*done*: (*emptied*:...) (*extended*:...)). A similar implementation in CLU[Schaffert, Snyder & Atkinson 1975] is found in Appendix III.

```
(empty-one-queue-into-another ≡
 (≡> [=q1 =q2]                ;two impure queues are received by empty-one-queue-into-another
                                                          ;and bound to q1 and q2.
   (rules (q1 <= (dq:))                                   ;the dequeuing message is sent to q1.

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

     (≡> (next: =front-of-q1                              ;if q1 is not empty, the front element of q1 and
         (rest: =dequeued-q1))                                        ;the remaining queue are received
                                                          ;and bound to front-of-q1 and dequeued-q1.
       (q2 <= (nq: front-of-q1))                                  ;front-of-q1 is enqueued at rear of q2.
       (empty-one-queue-into-another <= [dequeued-q1 q2])) ) ))
                                ;dequeued-q1 and q2 are sent to empty-one-queue-into-another.
```

Figure 7

One should note that the implementation in Figure 7 crucially depends on the fact that queue-actors referred to by q1 and q2 are impure actors. If q1 and q2 are pure actors, then 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-one-queue-into-another, the original queue-actors referred to by q1 and q2 would remain intact, which would violate the contract in Figure 6. Also note that if the original queue-actor refered to by q1 is a pure queue-actor, then q1 and dequeued-q1 would always denote different queue-actors.

8. SYMBOLIC EVALUATION AND SITUATIONS

In this section we give an overview of symbolic evaluation. Also important rules in our reasoning system, called *trans-situational rules*, are explained. As briefly mentioned before, symbolic evaluation is a process which abstractly evaluates a module on abstract data in the context of its contract. If the symbolic evaluation of a module M encounters an invocation of some module N, then the contract of N is used to continue the symbolic evaluation. The implementation of N is not used! The purpose of symbolic evaluation is to gather information for various purposes such as verification that the implementation satisfies its specification, question answering for debugging, perturbation analysis[Hewitt & Smith 1975], and refinement of the implementation.

In symbolic evaluation, the code of a program is interpreted step by step according to either pre-defined semantics of language primitives or contracts of modules invoked in the program. Each such step is triggered by the symbolic evaluation of an expression in the

code. The state of the program at each moment before or after such interpretation steps is refered as a *situation*. The symbolic evaluator has a data base to record what events happen, what facts hold and what is assumed in each situation. Facts that hold in a situation S are recorded as assertions associated with the situation S. Since the interpretation of each expression is performed on abstract data, when a conditional expression is interpreted, the subsequent symbolic execution path must split in the usual fashion. For example, consider the symbolic evaluation of

if (P x) *then* ... *else* ....

After the symbolic evaluation of the expression (P x), the symbolic execution path splits into two branches: one for then-clause and the other for else-clause. To start the subsequent symbolic evaluation, (P x) must be assumed for then-clause and ¬(P x) for else-clause. If the evaluation of (P x) has no side-effects, the assertions holding in the situation where (P x) is evaluated are inherited for both clauses.

Symbolic evaluation can be viewed as a process which evaluates the code forward along the execution path and produces a tree structure whose nodes correspond to situations. At each node of such a tree structure, assertions which hold in the corresponding situations can be entered. We call this tree structure a *situational tree*. (See Figure 8.) The assertions in the situational tree are used as the primary source of information for answering questions about the implementation.

In what follows, we illustrate how a situational tree is used. In general questions about a given situation are answered by reasoning backward to previous situations. Suppose that the system is asked whether or not a certain impure queue, say Q, has elements a b and c in a situation S_8. First, the system tries to find an assertion (Q *is-a* (*IMPURE-QUEUE* a b c)) at the node of the situational tree which corresponds to S_8. If it is found there, the answer is "yes", but if it is not found there, the system looks for the assertion (Q *is-a* (*IMPURE-QUEUE* a b c)) backward along the branch of the situational tree from the node of S_8. Suppose that the assertion is found at the node of a situation S_3. (See a dotted line in Figure 8) All we know, at this point, is that the assertion holds in S_3, but it is not sure that the assertion holds in S_8, because some events which destroy the validity of this assertion in S_8 might have happended between S_3 and S_8.

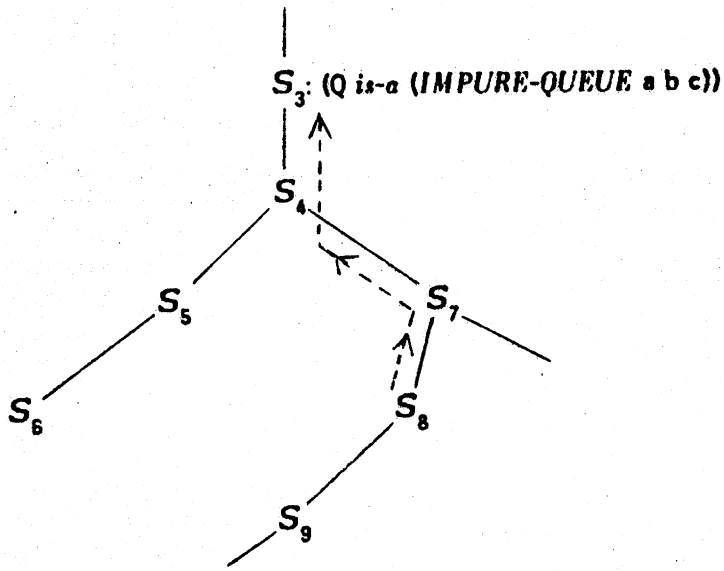


Figure 8.

So the system has to check whether or not such events have happened. In this case, in order to know what events destroy the validity of the assertion, the system consults the contract of (*IMPURE-QUEUE*...) given in Figure 4 and finds out the events (Q <= (*dq*:)) and (Q <= (*nq*:...)) in the (*for-assertions*:...)-clause. Thus the answer to the original question whether or not (Q *is-a* (*IMPURE-QUEUE* a b c)) holds in S_8 is "yes" if neither (Q <= (*dq*:)) nor (Q <= (*nq*:...)) have happened between S_3 and S_8.

The process working forward in interpreting each expression also relies on the process retrieving the situational tree backward. Suppose that an event (Q <= (*nq*: d)) takes place in S_8 in the above example. To get the state of the actor Q in the next situation S_9, the system wants to know the state of Q in S_8, which is not found the at the node of S_8. So the system tries to retrieve the information using the situational tree backward in the same manner as above.

In general the information which indicates what conditions assure the conservation of validity of assertions from one situation to another is called a ***trans-situational rule***. For particular assertions or particular types of assertions, appropriate trans-situational rules are necessary for correct reasoning. In what follows, we will give trans-situational rules which will be used in the examples of symbolic evaluation in this paper.

Assertions of the form: (<identifier> ≡ <actor>)

which states that <actor> is bound to <identifier>, can be passed unchanged between any two situations within the scope of <identifier>.

Assertions of the forms: (<actor1> *eq* <actor2>) and (<actor1> *not-eq* <actor2>),

which state the identity of actors can be always inherited from one situation to another without any conditions.

Assertions of the forms: (<sequence1> = <sequence2>) and (<sequence1> ≠ <sequence2>),

which state the equality sequences which appear in conceptual representations can be also inherited without any conditions. (Note that <sequence1> and <sequence2> are not sequence-actors but mathematical sequences. All mathematical facts can be inherited without any conditions. This is a special case.)

Assertions of the form: (<actor> *is-a* (*SEQUENCE* !x))

which states that <actor> is a sequence-actor whose elements are !x can be inherited without any conditions because a sequence-actor is a pure actor which never changes its state.

Assertions of the form: (<actor1> *knows-about* <actor2>)

which state the knows-about relation between actors can be inherited without any conditions.

Assertions of the form: (<variable> *has-contents* <actor>)

which state that <variable> has <actor> as its contents in some situation S can be inherited to a situation T if no assignments to this <variable> take place between S and T.

9. SYMBOLIC EVALUATION AND NAMES

Before we give a concrete example of symbolic evaluation, we will explain the use of names in symbolic evaluation. Names in PLASMA fall into two classes: ***identifiers*** and ***variables***. An identifier x can be declared and bound to the value of an expression E by

(*let* (x = E)...).

A variable x can be declared and initialized with the value of an expression E by

(*let* (x *initially* E)...).

The above statement is implemented by creating a new cell-actor C whose initial contents are the value of E and binding x to C. Evaluation of x is implemented by retrieving the contents of C. The value of x can be changed only by an expression of the form:

(x ← E)

and this is implemented by updating the contents of C with the value of E. PLASMA has been carefully designed so that there is no way to release the cell-actor C to other actors. In order to state that a variable x has an actor A as its value in a certain situation, the following assertion

(x *has-contents* A)

is used in symbolic evaluation. When an expression

(x ← E)

is interpreted in a situation S, the following assertion

(x *has-contents* B)

is entered in the situation following S where B is the value of E in S.

In the symbolic evaluation of a module , an identifier x in the code of module can be always regarded as the actor which is bound to x, because one identifier does not bind more than one different actor throughout the symbolic evaluation of M. This is guaranteed by:

1) the restriction on the syntax of PLASMA that no names are allowed to be rebound inside a module (referential transparency) and

2) the fact that symbolic evaluation passes over each expression in a module exactly once.

For example, let us consider the following piece of code.

```
(let (queue-1 = (create-impure-queue))
     then
       (let (queue-2 = queue-1) .....
```

In the first statement an empty impure queue-actor Q is created and bound to an identifier queue-1. To record this event, instead of using two assertions:

(Q *is-a* (*IMPURE-QUEUE*))
(queue-1 ≡ Q),

one assertion:

(queue-1 *is-a* (*IMPURE-QUEUE*))

suffices our purposes. This type of assertions will be frequently used in Section 12. By the second statement in the above code, the assertion:

(queue-2 *eq* queue-1)

will be entered in the data base and it tells the actor bound to queue-2 is the same actor denoted by queue-1. The question about the identity of queue-2 and queue-1 is answered by this assertion. (cf. Section 5)

10. AN EXAMPLE OF SYMBOLIC EVALUATION

As an example of symbolic evaluation, let us consider the symbolic evaluation of the code of empty-one-queue-into-another in Figure 7 in Section 7. In this example, we assume that the system tries to verify the implementation of empty-one-queue-into-another against its contract given in Figure 6 in Section 7. In order to ease the interaction between the system and users and also to aid the symbolic evaluation, programs should be augumented with commentary which denotes situations or identify important parts of code. In Figure 9 the augumented code of empty-one-queue-into-another is given. (This augumentation in Figure 9 is solely for the purpose of the presentation of this paper. For our existing system, somewhat different augumentations are used.) The large capital letters S... between the lines denotes situations.

```
(empty-one-queue-into-another ≡
 (≡> [=q1 =q2]                ;two impure queues are received by empty-one-queue-into-another
                                                          ;and bound to q1 and q2.
   - S received-queues -
   (rules (q1 <= (dq:))                                   ;the dequeuing message is sent to q1.
      (≡> (exhausted:)                                    ;if q1 is empty, the complaint message is received
         - S exhausted-q1 -
         (done: (emptied: q1) (extended: q2)) )           ;then emptied q1 and
                                                          ;extended q2 are returned.
      (≡> (next: =front-of-q1                             ;if q1 is not empty, the front element of q1 and
           (rest: =dequeued-q1))                          ;the remaining queue are received
                                                          ;and bound to front-of-q1 and dequeued-q1.
         - S dequeued-q1 -
      (q2 <= (nq: front-of-q1))                           ;front-of-q1 is enqueued at rear of q2.
         - S enqueued-q2 -
      (empty-one-queue-into-another <= [dequeued-q1 q2])) ) ))
                                   ;dequeued-q1 and q2 are sent to empty-one-queue-into-another.
```

Figure 9

Symbolic evaluation of a module takes place in the context of the specifications of the module. Thus in the case of this example, the symbolic evaluator reads the contract of empty-one-queue-into-another in Figure 6 and enters the following assertions which are the pre-requisites of empty-one-queue-into-another in the data base.

in $S_{initial}$:

(Q1 *is-a* (*IMPURE-QUEUE* !x1))
(Q2 *is-a* (*IMPURE-QUEUE* !x2))
(Q1 *not-eq* Q2)

After the symbolic pattern matching is performed, Q1 and Q2 are bound to identifiers q1 and q2, respectively. So the following assertions are entered in the data base.

in $S_{received\text{-}queues}$:

(q1 ≡ Q1)
(q2 ≡ Q2)

Then the expression (q1 <= (*dq*:)) in the rules-statement is interpreted. Namely, the

dequeuing message (*dq:*) is sent to Q1 which is bound to q1. To know the result of this event, the symbolic evaluator must consult the (*event...*) clause for dequeuing in the contract:

```
(event: (Q1 <= (dq:))
    (case-1:
      (pre-conditions: (Q1 is-a (IMPURE-QUEUE )) )
      (returns: (exhausted:) )
      (post-conditions: (Q1 is-a (IMPURE-QUEUE )) ) )
    (case-2:
      (pre-conditions: (Q1 is-a (IMPURE-QUEUE B !y)) )
      (returns: (next: B (rest: Q1)) )
      (post-conditions: (Q1 is-a (IMPURE-QUEUE !y)) ) ))
```

Note that the above clause is an instantiation of the (*event...*)-clause for the dequeuing in the contract for impure queues in Figure 4, which is obtained by substituting Q1 for Q. Now the symbolic evaluator has to consider two case: one case where Q1 is empty and the other case where Q1 is not empty. (See the situational tree for this example in Figure 10.)

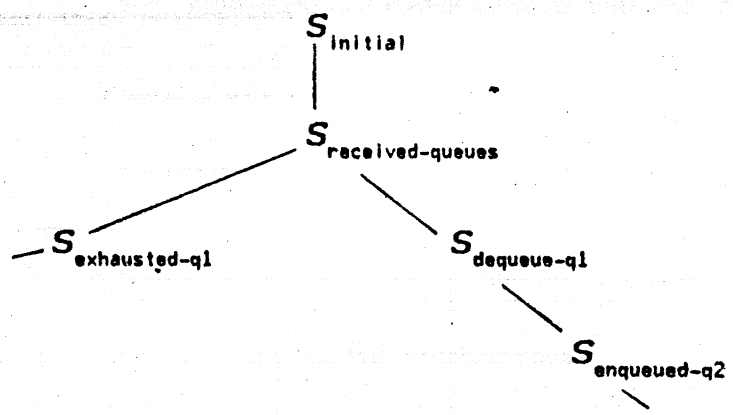


Figure 10

Case 1: (Q1 *is-a* (*IMPURE-QUEUE*))

In this case, the contract specifies that the (*exhausted:*) message is returned. This message matches against the first (≡>...)-statement inside the (rules...) statement. To follow this path, the symbolic evaluator needs to assume that x1 = []. So in $S_{exhausted\text{-}q1}$, the following assertions are entered in the data base.

in $S_{\text{exhausted-q1}}$:
(x1 = [])
(Q1 *is-a* (*IMPURE-QUEUE*))

Then in $S_{\text{exhausted-q1}}$ the message (*done:* (*emptied:* q1) (*extended:* q2)) is returned. At this point the contract of empty-one-queue-into-another in Figure 6 requires three things:

r1: (*done:* (*emptied:* Q1) (*extended:* Q2)) must be returned
r2: (Q1 *is-a* (*IMPURE-QUEUE*)) must hold and
r3: (Q2 *is-a* (*IMPURE-QUEUE* !x2 !x1)) must hold.

It is easy to show that each requirement is satisfied in $S_{\text{exhausted-q1}}$:

for r1, since the trans-situational rules for binding allow the inheritance of the assertions (q1 ≡ Q1) and (q2 ≡ Q2) from $S_{\text{received-queues}}$ to $S_{\text{exhausted-q1}}$, the required message is returned in $S_{\text{exhausted-q1}}$.

for r2, since the assertion (Q1 *is-a* (*IMPURE-QUEUE*)) is entered in $S_{\text{exhausted-q1}}$, it holds in $S_{\text{exhausted-q1}}$ and

for r3, (Q2 *is-a* (*IMPURE-QUEUE* !x2)) hold in $S_{\text{exhausted-q1}}$ because it can be inherited from S_{initial} by the trans-situational rule for (<actor> *is-a* (*IMPURE-QUEUE* ...)) and the following fact about sequences can be used because x1 = [] holds in $S_{\text{exhausted-q1}}$.

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

So (Q2 *is-a* (*IMPURE-QUEUE* !x2 !x1)) holds in $S_{\text{exhausted-q1}}$. Thus Case-1 is done.

Case-2: (Q1 *is-a* (*IMPURE-QUEUE* B !y))

In this case, the contract specifies that (*next:* B (*rest:* Q1)) is the result of (q1 <= (*dq:*)) where the following assertions are assumed.

(x1 = [B !y])
(Q1 *is-a* (*IMPURE-QUEUE* !y))

So (*next:* B (*rest:* Q1)) is matched against the pattern in the first (≡>...) statement inside of the (rules...) statement. The symbolic evaluator also asserts the binding information in $S_{\text{dequeued-q1}}$.

in $S_{\text{dequeued-q1}}$:
(front-of-q1 ≡ B)
(dequeued-q1 ≡ Q1)
(x1 = [B !y])
(Q1 *is-a* (*IMPURE-QUEUE* !y))

Then in this situation (*nq:* B) message is sent to Q2 which is bound to q2 and the assertion (Q2 *is-a* (*IMPURE-QUEUE* !x2)) holds because it can be inherited from $S_{initial}$ by the trans-situational rule for (<actor> *is-a* (*IMPURE-QUEUE*...)). From the instantiated clause for the event (Q2 <= (*nq:* B))...) of the contract in Figure 4 (note the substitutions of Q2 for Q, x2 for x, and B for A.) :

(*event:* (Q2 <= (*nq:* B))
(*pre-conditions:* (Q2 *is-a* (*IMPURE-QUEUE* !x2)))
(*returns:* Q2)
(*post-conditions:* (Q2 *is-a* (*IMPURE-QUEUE* !x2 B))))

the symbolic evaluator enters the following assertion in $S_{enqueued\text{-}q2}$.

in $S_{enqueued\text{-}q2}$:
(Q2 *is-a* (*IMPURE-QUEUE* !x2 B))

Now the symbolic evaluator encounters the transmission of Q1 and Q2 to empty-one-queue-into-another in $S_{enqueued\text{-}q2}$. Then in order to know the behavior of the empty-one-queue-into-another, the symbolic evaluator refers to the contract for empty-one-queue-into-another in Figure 6. (Note that this is a "recursive" use of the contract.) Since the assertions:

(Q1 *is-a* (*IMPURE-QUEUE* !y)) and (Q1 *not-eq* Q2)

can be inherited from $S_{dequeued\text{-}q1}$ by the trans-situational rules for

(<actor> *is-a* (*IMPURE-QUEUE*...)) and (<actor1> *not-eq* <actor2>),

respectively, the following pre-conditions of empty-one-queue-into-another are satisfied.

(Q1 *is-a* (*IMPURE-QUEUE* !y))
(Q2 *is-a* (*IMPURE-QUEUE* !x2 B)) and
(Q1 *not-eq* Q2)

Therefore the post-conditions of the contract for empty-one-queue-into-another guarantees that (*done:* (*emptied:* Q1) (*extended:* Q2)) is returned and that the following assertions:

(Q1 *is-a* (*IMPURE-QUEUE*)) and
(Q2 *is-a* (*IMPURE-QUEUE* ![!x2 B] !y))

hold in the situation following $S_{enqueued\text{-}q2}$. Then the following knowledge about sequences is used to simplify the above assertions:

[![!x2 B] !y] is equivalent to [!x2 B !y],
[!x2 B !y] is equivalent to [!x2 !x1] if x1 = [B !y].

Since x1 = [B !y] can be inherited from $S_{dequeued\text{-}q1}$ by the trans-situational rule for (<sequence1> = <sequence2>), the symbolic evaluator can claim that

(Q1 *is-a* (*IMPURE-QUEUE*)) and
(Q2 *is-a* (*IMPURE-QUEUE* !x2 !x1)).

Thus the post-conditions for empty-one-queue-into-another are also satisfied in Case-2.

Since the requirements stated in the contact for empty-one-queue-into-another are satisfied, we conclude that the implementation of empty-one-queue-into-another in Figure 7 is guaranteed to meet its contract in Figure 6. It should be noted that the above assertions are guaranteed to hold in a given situation only if control passes through that situation. There is no guarantee that any of situations after $S_{received\text{-}queues}$ above will ever be reached. Therefore the demonstration of convergence is another part of the symbolic evaluation that can be incorporated into the above symbolic evaluation. As we mentioned in Section 5, explicit use of situational tags is powerful to the formal demonstration of the convergence. For a detailed demonstration of the convergence, see [Yonezawa 1975].

II. AN IMPLEMENTATION FOR IMPURE-QUEUE

In the symbolic evaluation of empty-one-queue-into-another, the only properties of impure queues used were only the ones given in the contract for impure queues in Figure 4. This fact guarantees that empty-one-queue-into-another satisfies its specification on <u>all</u> implementations of impure queues that satisfy the contract in Figure 4. Now we give an example of an implementation of an impure queue which is supposed to satisfy the contract in Figure 4. The PLASMA code depicted in Figure II is such an implementation. Note that queuees is an example of a PLASMA <u>variable</u> explained in Section 9. So the value of queuees changes, but other names such as new-element, the-queue-itself, front etc. are identifiers and thus are bound to the same actor throughout their scope.

```
(create-impure-queue ≡
 (≡> []                                   ;create-impure-queue receives an empty sequence.
  (let (queuees initially [])             ;a variable queuees is declared
   then                                   ;and initialized with an empty sequence.
   (the-queue-itself ≡                    ;a queue-actor is defined by the cases-statement given below
                                          ;and denoted by the-queue-itself.
    (cases
     (≡> (nq: =new-element)               ;when an enqueue message with an element is received,
                                          ;the element is bound to new-element.
      (queuees ← [!queuees new-element])  ;a new sequence-actor whose elements are
                                          ;the unpack of the value of queuees and new-element
                                          ;is created and stored in queuees.
      the-queue-itself)                   ;and then the-queue-itself is returned.

     (≡> (dq:)                            ;when a dequeue message is received,
      (rules queuees                      ;if the contents of queuees
       (≡> [] (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 value of queuees is updated.
        (next: front (rest: the-queue-itself))  ) )) ))))   ;(next:...) is returned.
```

Figure 11

The basic idea of this implementation is presented as a diagram in Figure 12. An impure queue-actor the-queue-itself consists of a variable queuees which has a sequence-actor S as its value. The elements of this sequence-actor S are the members of the queue. In the diagram arrows indicate which actors directly reference others (i.e. the knows-about relation). (cf. Section 3) Since the-queue-itself returns itself, the-queue-itself must know-about the-queue-itself.

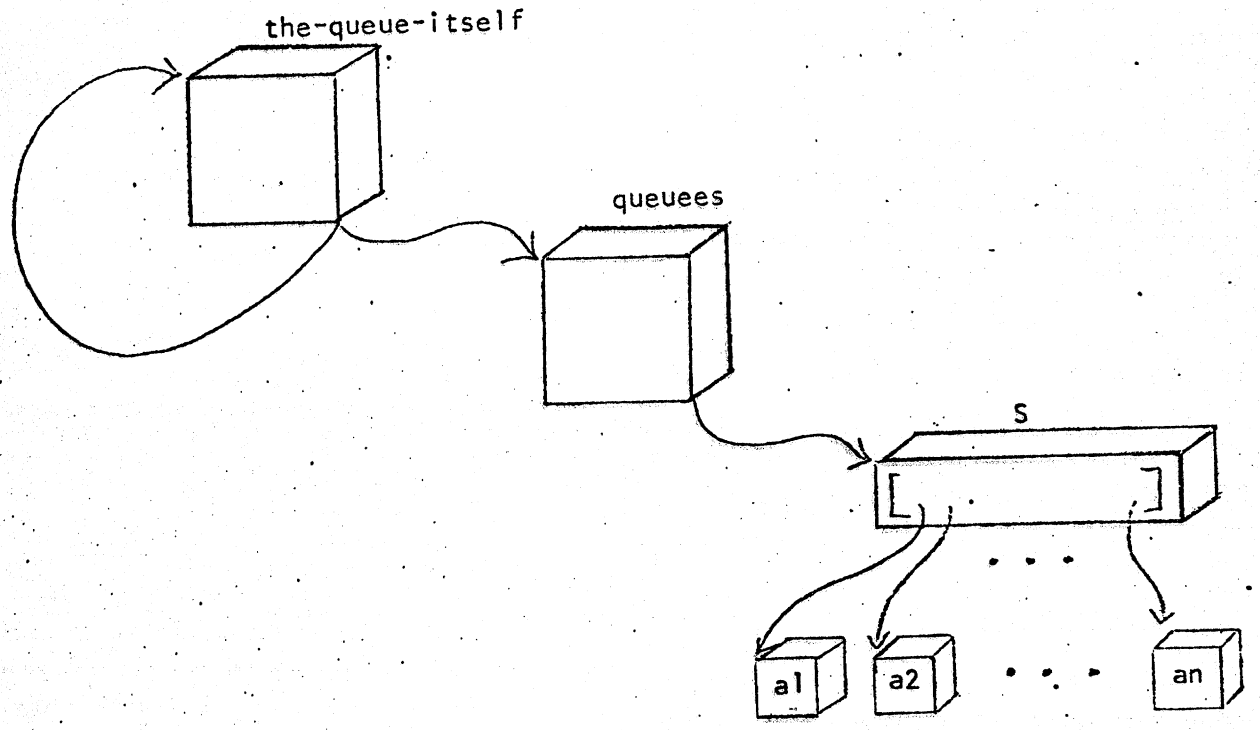


Figure 12

The diagram in Figure 12 is not only a partial and static description of the above implementation, but it also expresses invariant or integrity conditions which must be satisfied among the constituents of the implementation before and after each invocation. Thus an important property of the implementation can be expressed formally as the following *invariant statement*.

(*Invariant:* (the-queue-itself *is-a* (*IMPURE-QUEUE* !a))
where (the-queue-itself *knows-about* queuees)
(the-queue-itself *knows-about* the-queue-itself)
(queuees *has-contents* S)
(S *is-a* (*SEQUENCE* !a)))

This states that the assertion (the-queue-itself *is-a* (*IMPURE-QUEUE* !a)) is an invariant. When this assertion holds, the-queue-itself has acquaintances queuees and the-queue-itself {namely, (the-queue-itself *knows-about* queuees) and (the-queue-itself *knows-about* the-queue-itself) hold}, the variable queuees has a sequence-actor S as its value {namely, (queuees *has-contents* S) } and S has !a as its elements. A notation (*SEQUENCE*...) is the conceptual representation of a sequence-actor.

A similar implementation written in CLU is found in Appendix IV. These two implementations exhibit the same computation sequence in terms of the actor model of computation.

12. SYMBOLIC EVALUATION OF IMPURE-QUEUE

Now we proceed to the symbolic evaluation of **create-impure-queue** in Figure 11 against its contract in Figure 4. This time our interest is not only in the demonstration that the implementation satisfies its specification, but also in the internal structure of the code which will be exposed during the process of symbolic evaluation. The code of **create-impure-queue** in Figure 13 is augumented with situational tags and the (*Invariant:*) statement which was introduced in Section 11. In general (*Invariant:...*) statements not only express the invariant or integrity conditions of actors which behave like data structures, but also they express inductive assertions when they are used inside iterative programs.

```
(create-impure-queue ≡
 (≡> []                                   ;create-impure-queue receives an empty sequence.
  (let (queuees initially [])             ;a variable queuees is declared
    then                                  ;and initialized with an empty sequence.
     - S initialized-queuees -

  (the-queue-itself ≡
       - S nq-or-dq-initial -
    (Invariant: (the-queue-itself is-a (IMPURE-QUEUE !a))
                where (the-queue-itself knows-about queuees)
                      (the-queue-itself knows-about the-queue-itself)
                      (queuees has-contents S)
                      (S is-a (SEQUENCE !a))  )

  ;a queue-actor is defined by the cases-statement given below and denoted by the-queue-itself.
    (cases
     (≡> (nq: =new-element)               ;when an enqueue message with an element is received,
                                          ;the element is bound to new-element.
                 - S received-nq -
      (queuees ← [!queuees new-element])  ;a new sequence-actor whose elements
                                          ;are the unpack of the value of queuees and new-element
                                          is created and is stored in queuees.
                 - S updated-queuees-nq -
      the-queue-itself)                   ;and then the-queue-itself is returned.

     (≡> (dq:)                            ;when an dequeue message is received,
                 - S received-dq -
      (rules queuees                      ;if the value of queuees
       (≡> []                             ;is empty
                 - S empty-queuees -
        (exhausted:) )                    ;then the complaint message is returned.

       (≡> [=front !=rest]                ;otherwise the first element is bound to front
                                          ;and the rest of the elements is bound to rest.
                 - S non-empty-queuees -
        (queuees ← rest)                  ;the contents of queuees is updated.
                 - S updated-queuees-dq -
        (next: front (rest: the-queue-itself)) ) )) ))  ))                ;(next:...) is returned.
```

Figure 13

The symbolic evaluation of the code in Figure 13 is carried out in the context of the contract of impure queues in Figure 4. There are four clauses in the contract: one clause

each for the creation, enqueuing and dequeuing, and one clause for the trans-situational rule for assertions of the form (Q *is-a* (*IMPURE-QUEUE*...)). As seen in the symbolic evaluation of empty-one-queue-into-another in Section 10, trans-situational rules are vital for reasoning in situations where side-effects are involved. Therefore it is necessary to establish the trans-situational rule in the contract for impure queues as well as the specifications of the creation, enqueuing, and dequeuing. In what follows, the four clauses in the contract are established separately.

I. Establishing the CREATION specification

The symbolic evaluator reads the first (*event:*...) clause in the contract for (*IMPURE-QUEUE*...) in Figure 4:

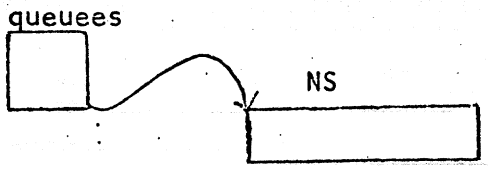
(*event:* (create-impure-queue <= [])
(*returns:* (*new-actor* Q))
(*post-conditions:* (Q *is-a* (*IMPURE-QUEUE*)))).

Since there are no pre-conditions for this event, no assertions are entered in the data base for the initial situation.

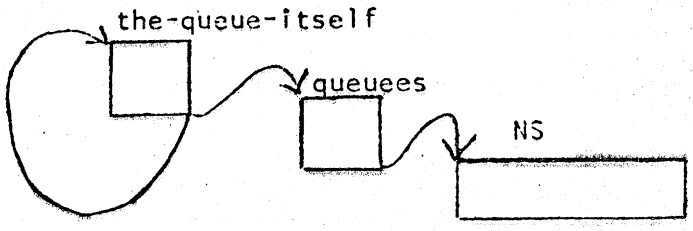
in $S_{\text{pre-creation}}$: empty

The *let* statement declares and initializes a variable queuees with an empty sequence NS. The following assertions are entered.

in $S_{\text{initialized-queuees}}$:
(queuees *has-contents* NS)
(NS *is-a* (*SEQUENCE*))



Then in this situation an actor whose script (i.e. code) is given as the (cases...) statement after (the-queue-itself ≡ ... is newly created and returned. This actor is denoted by the-queue-itself. Furthermore by looking for free names (variables or identifiers) in the script of the-queue-itself, its acquaintances are found: in this case the acquaintances are queuees and the-queue-itself. To record this, the following assertions are entered in the next situation $S_{\text{post-creation}}$.



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

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

The contract for the creation requires that the returned actor Q be newly created and have the property (Q *is-a* (*IMPURE-QUEUE*)). Since the returned actor is the-queue-itself, what has to be shown is that (the-queue-itself *is-a* (*IMPURE-QUEUE*)) holds. This assertion is translated using the assertions in *where*-clause in the invariant statement. Now it has to be shown that the following assertions hold in $S_{post\text{-}creation}$. (Note that the assertions in the *where*-clause are instantiated by substituting an empty sequence [] for !a.)

(the-queue-itself *knows-about* queuees)
(the-queue-itself *knows-about* the-queue-itself)
(queuees *has-contents* S)*
(S *is-a* (*SEQUENCE*))*

The first two assertions hold in $S_{post\text{-}creation}$ because they are entered in $S_{post\text{-}creation}$. The two assertions entered in $S_{initialized\text{-}queuees}$:

(queuees *has-contents* NS) and (NS *is-a* (*SEQUENCE*))

can be inherited to $S_{post\text{-}creation}$ by the trans-situational rules for types of assertions (<variable> *has-contents*...)) and (<actor> *is-a* (*SEQUENCE*...)) and they are matched against the assertions marked with *. Therefore it is concluded that the returned actor the-queue-itself has the correct internal structure prescribed by the invariant statement. So the result of (create-impure-queue <= []) satisfies its specification.

II. Establishing the ENQUEUING specification

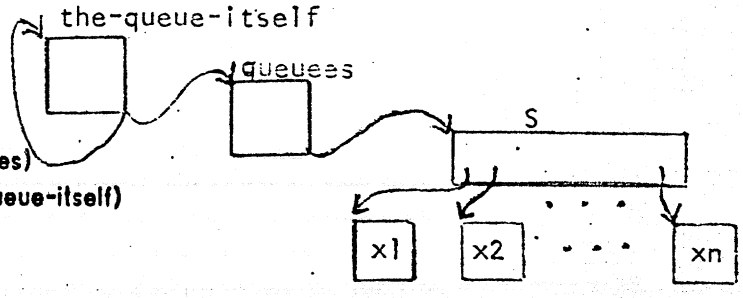
From the instantiation of the specification for enqueueing:

(*event:* (the-queue-itself <= (*nq:* A))
(*pre-conditions:* (the-queue-itself *is-a* (*IMPURE-QUEUE* !x)))
(*returns:* the-queue-itself)
(*post-conditions:* (the-queue-itself *is-a* (*IMPURE-QUEUE* !x A)))

which is obtained by substituting the-queue-itself for Q in the contract for (*IMPURE-QUEUE*...) in Figure 4, it is assumed that

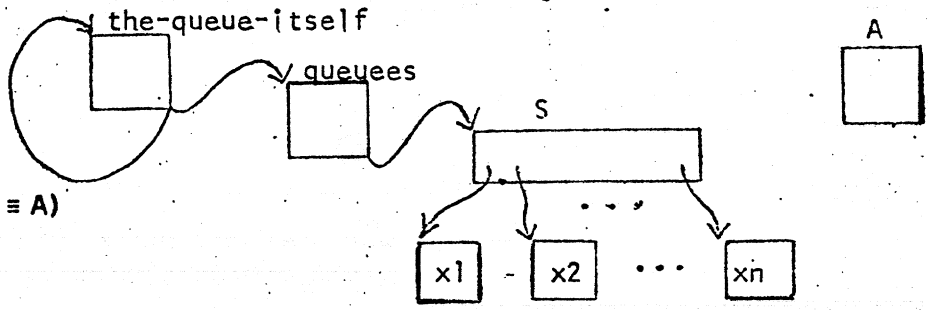
(the-queue-itself *is-a* (*IMPURE-QUEUE* !x))

holds in the initial situation. By the invariant statement in the code, this assumption is translated into the following four assertions. (Note that x is substituted for a in the invariant statement.)



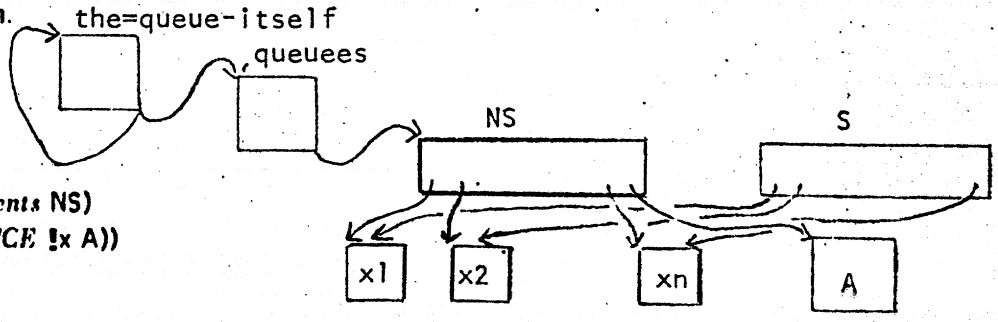
in $S_{\text{nq-or-dq-initial}}$:
(the-queue-itself *knows-about* queuees)
(the-queue-itself *knows-about* the-queue-itself)
(queuees *has-contents* S))
(S *is-a* (*SEQUENCE* !x))

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



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

Then the value of queuees is updated by a newly created sequence-actor NS with its elements [!queuees new-element]. Since the value of queuees in $S_{\text{received-nq}}$ is a sequence-actor S, !queuees is the result of the unpack operation on S, which is !x. So the new sequence-actor NS is represented as (*SEQUENCE* !x A). For the assignment of NS to queuees, the new assertion (queuees *has-contents* NS) is entered in the data base. So the following assertions hold in the next situation.



in $S_{\text{updated-queuees-nq}}$:
(queuees *has-contents* NS)
(NS *is-a* (*SEQUENCE* !x A))

In this situation the-queue-itself is returned. Note that the specification for the enqueueing requires that the-queue-itself is returned and that

(the-queue-itself *is-a* (*IMPURE-QUEUE* !x A)).

By making all assertions holding in $S_{\text{updated-queuees-nq}}$ explicit by applying the trans-situational rules for (<actor1> *knows-about* <actor2>) and (<identifier> ≡ <actor>), the following assertions are obtained.

(queuees *has-contents* NS)*
(NS *is-a* (*SEQUENCE* !x A))*
(new-element ≡ A)
(the-queue-itself *knows-about* queuees)*
(the-queue-itself *knows-about* the-queue-itself)*
(S *is-a* (*SEQUENCE* !x)) <-- S *is no longer referenced.*

It is easy to see that the assertions obtained by translating (the-queue-itself *is-a* (*IMPURE-QUEUE* !x A)) through the invariant statement are matched against the above assertions marked with *. So enqueuing satisfies its specification.

Besides the fact that the implementation of enqueuing meets its specification, a very interesting fact is revealed by the above symbolic evaluation. The sequence S is created by this implementation and never passed out. S was initially the value of the variable queuees in $S_{\text{nq-or-dq-initial}}$ but in $S_{\text{updated-queuees-nq}}$ it is not the value of queuees and there are no acquaintances of the-queue-itself which know about it in $S_{\text{updated-queuees-nq}}$. So 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 enqueuing takes place, a garbage sequence is produced.

III. Establishing the DEQUEUING specification

As is indicated in the instantiation of the specification of the dequeuing:

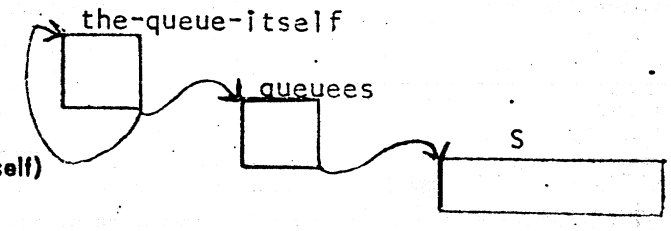
(*event:* (the-queue-itself <= (*dq:*))
(*case-1:*
(*pre-conditions:* (the-queue-itself *is-a* (*IMPURE-QUEUE*)))
(*returns:* (*exhausted:*))
(*post-conditions:* (the-queue-itself *is-a* (*IMPURE-QUEUE*))))
(*case-2:*
(*pre-conditions:* (the-queue-itself *is-a* (*IMPURE-QUEUE* B !y)))
(*returns:* (*next:* B (*rest:* the-queue-itself)))
(*post-conditions:* (the-queue-itself *is-a* (*IMPURE-QUEUE* !y)))

which is obtained by substituting the-queue-itself for Q in the contract for impure queues in Figure 4, 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: (the-queue-itself *is-a* (*IMPURE-QUEUE*))

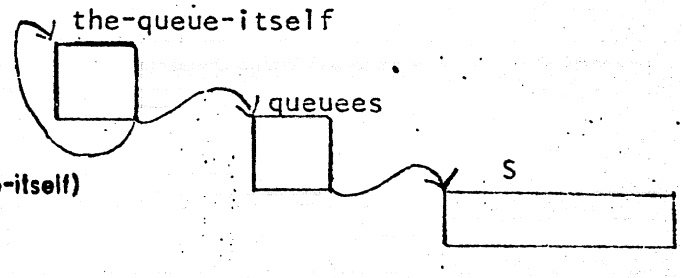
Assuming that (the-queue-itself *is-a* (*IMPURE-QUEUE*)) holds in the initial situation, the following assertions are obtained through the invariant statement and entered in the data base.



in $S_{\text{nq-or-dq-initial}}$:
(the-queue-itself *knows-about* queuees)
(the-queue-itself *knows-about* the-queue-itself)
(queuees *has-contents* S)
(S *is-a* (*SEQUENCE*))

Now the (*dq:*) message is sent to the-queue-itself. Then the message matches against the second clause in the (cases...) statement and $S_{\text{received-dq}}$ is reached. Since the value of queuees is S which is an empty sequence, the first (≡>...) clause in the rules statement is matched and the situation $S_{\text{empty-queuees}}$ is reached.

Then the (*exhausted:*) message is returned. At this point the contract requires that (the-queue-itself *is-a* (*IMPURE-QUEUE*)) holds. Again by the invariant statement, the following assertions are required to hold.

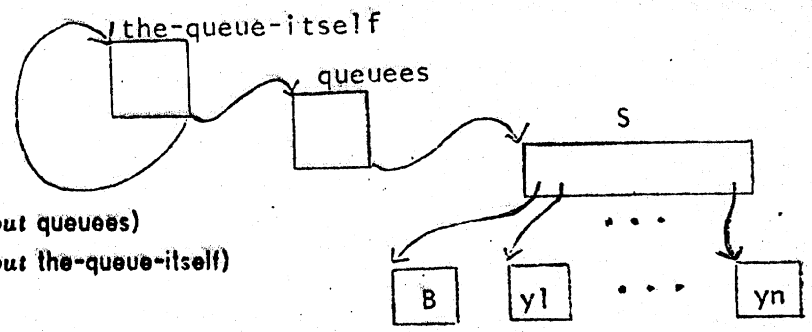


(the-queue-itself *knows-about* queuees)
(the-queue-itself *knows-about* the-queue-itself)
(queuees *has-contents* S)
(S *is-a* (*SEQUENCE*))

The above assertions are matched against the ones which hold in $S_{\text{nq-or-dq-initial}}$ and no events which affect the conservation of validity of the assertions in $S_{\text{nq-or-dq-initial}}$ happened between $S_{\text{empty-queuees}}$ and $S_{\text{nq-or-dq-initial}}$. So what the contract requires is satisfied for Case 1.

Case 2: (the-queue-itself *is-a* (*IMPURE-QUEUE* B !y))

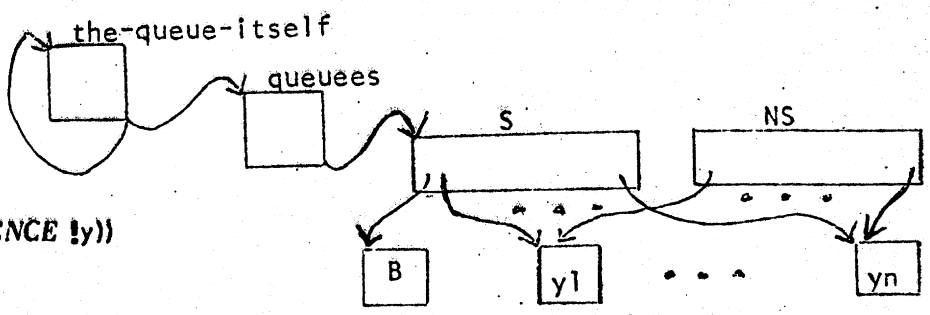
For this case, (the-queue-itself *is-a* (*IMPURE-QUEUE* B !y)) is assumed in the contract and so the assertions translated by the invariant statement are as follows.



in $S_{\text{nq-or-dq-initial}}$
(the-queue-itself *knows-about* queuees)
(the-queue-itself *knows-about* the-queue-itself)
(queuees *has-contents* S)
(S *is-a* (*SEQUENCE* B !y))

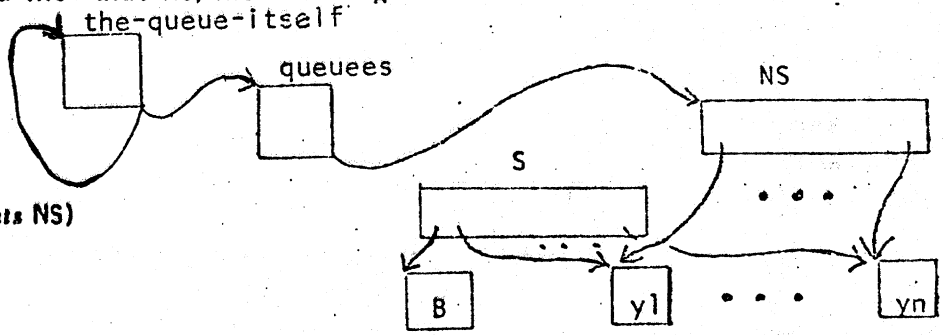
The (*dq:*) message is sent to the-queue-itself and the situation $S_{\text{received-dq}}$ is reached.

The value of queuees which is S matches the pattern [=front !=rest], because (S *is-a* (*SEQUENCE* B !y)) holds. By this matching a new sequence, say NS, whose elements are !y is created and bound to rest and B is bound to front. So the binding information is also added in the next situation.



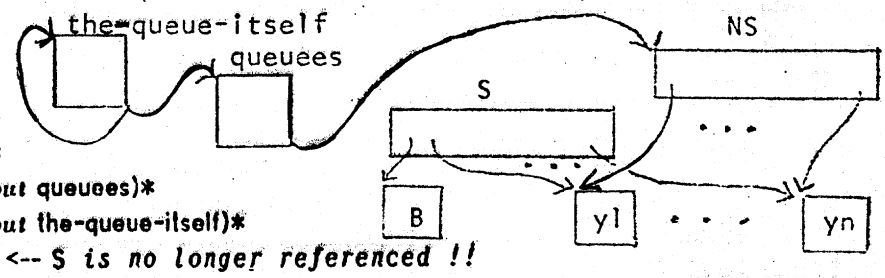
in $S_{\text{non-empty-queuees}}$:
(front ≡ B)
(rest ≡ NS)
(NS *is-a* (*SEQUENCE* !y))

After queuees is assigned the value NS, the following assertion is entered in the next situation.



in $S_{\text{updated-queuees-dq}}$:
(queuees *has-contents* NS)

Now in this situation all assertions which can be inherited from the previous situations by the appropriate trans-situational rules are as follows:



(queuees *has-contents* NS)*
(front ≡ B)
(rest ≡ NS)
(NS *is-a* (*SEQUENCE* !y))*
(the-queue-itself *knows-about* queuees)*
(the-queue-itself *knows-about* the-queue-itself)*
(S *is-a* (*SEQUENCE* B !y)) <-- S *is no longer referenced !!*

In this situation (*next:* B (*rest:* the-queue-itself)) is returned. The four assertions marked with * guarantee that (the-queue-itself *is-a* (*IMPURE-QUEUE* !y)) holds through the invariant statement. So all cases are shown.

As in the case of the enqueuing, it is also revealed by the symbolic evaluation that a sequence S which was the initial value of queuees becomes a garbage sequence at the end of the dequeuing.

IV. Establishing the TRANS-SITUATIONAL RULE for (the-queue-itself *is-a* (*IMPURE-QUEUE* ...))

The instantiation of the trans-situational rule:

(*for-assertion:* (the-queue-itself *is-a* (*IMPURE-QUEUE*...))
(*only-affecting-events-are:*
{(the-queue-itself <= (*nq:*...)) (the-queue-itself <= (*dq:*))}))

which is obtained by substituting the-queue-itself for Q in the (*for-assertion:*...)-clause in the contract for impure queues in Figure 4 says that the only events which may destroy the conservation of validity of assertions of the form (the-queue-itself *is-a* (*IMPURE-QUEUE*...)) are (the-queue-itself <= (*nq:*...)) and (the-queue-itself <= (*dq:*)).

This is established by the following facts:

1) The preceding symbolic evaluation has shown that the-queue-itself always maintains the invariant or integrity conditions before and after each invocation by (*nq:*...) and (*dq:*) because the following facts are shown by the symbolic evaluation.
   a) immediately after the creation of the-queue-itself, it maintains the invariant condition: (the-queue-itself *is-a* (*IMPURE-QUEUE*...)) and
   b) if the invariant condition is satisfied before invocations by (*nq:*...) or (*dq:*), the-queue-itself still maintains its invariant condition after the invocations.

2) The preceding symbolic evaluation has shown that no internal constituents of the-queue-itself are released outside. Note that there is no way of releasing the cell-actor with which the variable queuees is implemented. (cf. Section 9)

3) the-queue-itself accepts only messages of the forms (*nq:*...) and (*dq:*) and the state of the-queue-itself expressed by the conceptual representation (*IMPURE-QUEUE*...) can be changed only by events of the form:

(the-queue-itself <= (*nq:*...)) and (the-queue-itself <= (*dq:*)).

Since the four clauses in the contract for impure queues have been established, it follows that the implementation of impure queues in Figure 11 (or Figure 13) satisfies the contract for impure queues in Figure 4.

13. CONCLUSIONS

The work presented in this paper is based upon two main ideas: ***conceptual representations*** and the explicit use of ***situations***.

Conceptual representations serve as a notational devise to describe not only states of individual data structures, but also how individual data structures are interrelated at various levels. Since conceptual representations are able to directly express states of data structures, specifications of data structures by conceptual representations are often easier to write and understand than algebraic specifications. By separating the state of an object from its identity, conceptual representations can describe sharing structures of data and can easily specify the behaviors of data structures with side-effects.

We introduced a notion of situations which was the state of a system at a given moment. By relativizing states of objects in the system with situational tags, relations and assertions about states of objects in different situations can be expressed. Our assertion language can describe systems from various points of view and emphasis by appropriate choices of conceptual representations and situations. Thus the expressive power of our formalism is strong enough to cover a wide range of the statements which are needed in debugging, question-answering, and the evolutional developments of programs.

The traditional approach of program verification is that given a set of assertions holding in a situation, verification conditions are generated either in backward direction[Hoare 1968, Dijkstra 1976] or in forward direction[Floyd 1967]. In contrast to the traditional approach, our approach is:

First, by symbolically evaluating in the ***forward*** direction, a "partial" description of each situation in terms of conceptual representations is produced. (Here a "partial" description is used in two senses: 1) a description of a situation at a certain level of details, and 2) in creating a description of one situation, the part of the description of the previous situation which has not changed is not necessarily duplicated.)

Then for answering questions, the retrieval is performed in the ***backward*** direction using ***trans-situational rules***.

Descriptions of situations in terms of conceptual representations enable us to deal with side-effects and working in both forward and backward directions allows us the flexibility to answer questions.

14. ACKNOWLEDGEMENTS

The comments and suggestions of B. Liskov were valuable. Thanks are also due to Chuck Rich, Ron Pankiewicz, Ken Kahn, Dick Steiger, Pat Greussay and Harold Weltz who carefully read an early version of this paper and made comments.

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.

15. BIBLIOGRAPHY

Boyer, R.S., Elspas, B. and Levitt, K.N. "SELECT -- A Formal System for Testing and Debugging Programs by Symbolic Execution." Proc. of International Conference on Reliable Software. Los Angles, 1975.

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

Burstall, R.M. "Some Techniques 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.

Birtwistle, Dahl, Myhrhang and Nygaard. SIMULA Begin Auerbach. 1973

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

Dijkstra, E. W. A Discipline of Programming Prentice-Hall, Englewood Cliffs, N.J. 1976

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.

Good, D.I., London, R.L. and Bledsoe, and W.W. "An Interactive Program Verification System." IEEE Transaction on Software Engineering, Vol. SE-1 No. 1. March, 1975.

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" to appear in the Journal of Artificial Intelligence.

Hewitt, C.E and Baker, H. "laws for Communicating Parallel Processes" Working Paper 134. Artificial Intelligence Laboratory MIT. December 1976.

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

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

Learning Research Group "Personal Dynamic Media" Technical Report. Xerox Palo Alto Research Center. 1976.

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 Techniques for Data Abstractions" IEEE transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

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

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

Spitzen, J. and Wegbreit, B. "The Verification and Synthesis of Data Structures." Acta Informatica 4. 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. "Conceptual Representations -- A Specification Technique for Data Structures" Working paper in preparation Artificial Intelligence Laboratory, MIT.

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

APPENDIX I DERIVATION OF AXIOM a5 FROM THE CONTRACT FOR PURE QUEUES

Axiom a5: *if* is-empty(q) = *false* ∧ dequeue(q) = <B, q'>
then dequeue(enqueue(q, A)) = <B, enqueue(q', A)>

is derived from the specification of queues based on the conceptual representation in Section 2. In the following derivation c2, c3, c4, c5, and c6 refer to the lines in specification base on the conceptual representation.

1) is-empty(q) = *false* ;*given as the premise of the axiom* a5).
2) dequeue(q) = <B, q'> ;*given as the premise of the axiom* a5).
3) q <---> (*QUEUE* B !x) ;*from* 1), 2), and c6).
4) q' <---> (*QUEUE* !x) ;*from* 2), 3) and c4).
5) dequeue(enqueue(q, A)) ;*the left side of the axiom* a5).
<---> dequeue(enqueue ((*QUEUE* B !x), A)) ;*from* 3).
= dequeue((*QUEUE* B !x A)) ;*from* c2).
= <B , (*QUEUE* !x A)> ;*from* c4).
= <B , enqueue((*QUEUE* !x), A)> ;*from* c2).
<---> <B , enqueue(q', A)> q.e.d. ;*from* 4).

APPENDIX II. A CONTRACT FOR CELLS

[contract-for (*CELL* ...) ≡

(*event:* (create-cell A)
(*returns:* (*new-actor* C))
(*post-conditions:* (C *is-a* (*CELL* A))))

(*event:* (C <= (*contents?*))
(*pre-conditions:* (C *is-a* (*CELL* B)))
(*returns:* B)
(*post-conditions:* (C *is-a* (*CELL* B))))

(*event:* (C ← D))
(*pre-conditions:* (C *is-a* (*CELL* E)))
(*returns:* C)
(*post-conditions:* (C *is-a* (*CELL* D))))

(*for-assertion:* (C *is-a* (*CELL* ...))
(*only-affecting-events-are:* { (C <= (*update:*...)) }))]

APPENDIX III empty-one-queue-into-another in CLU

```
empty-one-queue-into-another =
   proc(q1: impure-queue, q2: impure-queue)
            returns record[emptied: impure-queue, extended: impure-queue];
      front-of-q1: any;
      dequeued-q1: impure-queue;
      front-of-q1, dequeued-q1 := impure-queue$dq(q1)
                              except exhausted: return {emptied: q1, extended: q2} end;
      impure-queue$nq(q2, front-of-q1);
      empty-one-queue-into-another(dequeued-q1, q2);
   end empty-one-queue-into-another;
```

APPENDIX IV impure-queue in CLU

```
impure-queue = cluster is create, nq, dq;
     rep = record[queuees: sequence];
create = proc() returns(cvt);
   return {queuees: sequence$create()};
end create;
%
nq = proc(a-queue: cvt, new-element: any) returns(cvt);
   a-queue.queuees := sequence$extendh(a-queue.queuees, new-element);
   return a-queue;
end nq;
%
dq = proc(a-queue: cvt) returns (any, cvt) signals exhausted;
   if sequence$empty(a-queue.queuees)
         then signal exhausted;
         else
            front: any := sequence$first(a-queue.queuees);
            a-queue.queuees := sequence$rest(a-queue.queuees);
            return front, a-queue;  % multi-valued return
end dq;
%
end impure-queue;
```