MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

WORKING PAPER 212 December 1980

GUARDIANS FOR CONCURRENT SYSTEMS

Carl Hewitt and Giuseppe Attardi

ABSTRACT

In this paper we survey the current state of the art on fundamental aspects of concurrent systems. We discuss the notion of concurrency and discuss a model of computation which unifies the lambda calculus model and the sequential stored program model. We develop the notion of a guardian as a module that regulates the use of shared resources by scheduling their access, providing protection, and implementing recovery from hardware failures. A shared checking account is an example of the kind of resource that needs a guardian. We introduce the notions of a customer and a transaction manager for a request and illustrate how to use them to implement arbitrary scheduling policies for a guardian. A proof methodology is presented for proving properties of guardians, such as guarantee of service for all requests received.

Table of Contents

I -- Introduction

Two computational activities A1 and A2 will be said to be **concurrent** if they do not have a necessary temporal ordering with respect to one another i.e. A1 might precede A2, A2 might precede A1, or they might even overlap in time. Concurrency can arise from a variety of sources including multiplexing individual processors as well as from the interaction of multiple processors. Thus concurrent systems include time sharing systems, multiprocessor systems, and distributed systems. In this paper we discuss fundamental issues in the design, implementation, and analysis of concurrent systems. We do not assume that the reader has any prior knowledge of actors, message-passing or recent proposals of languages based on communicating sequential processes.

II -- Background

For thirty years the lambda calculus and the sequential stored program have coexisted as important bases for software engineering. Systems (such as Pure Lisp and ISWIM) based on the lambda calculus provide a sound basis for constructing independent immutable objects (functions and functional data structures). Systems based on the lambda calculus have inherent concurrency which is constrained only by the speed of communications between processing elements. Lambda calculus systems provide important ways to realize the massive parallelism that will be made possible by the development of very large scale integrated circuits.

Unfortunately the lambda calculus is not capable of realizing the need in concurrent systems for shared objects such as checking accounts which must change their behavior during the course of their lifetime. The stored program computer provides a way to make the required changes through its ability to update its global memory. However the concurrency of the stored program computer is limited because only one access to the memory can occur at a time. The variable assignment command (e.g. SETQ in LISP or the := command of the Algol-like languages) incorporates this ability in higher level languages. The attendant cost is that the higher level language becomes inherently sequential.

In the early seventies an important step was made toward the unification of the two approaches, by developing the concept of *object*. An object consists of a local state and procedures to operate on the object. SIMULA [Birtwistle et al. 1973] was the first language in the ALGOL family which introduced objects in the form of *class* instances. A

class declaration specifies the structure of each object (in term of the variables which constitute its local state) and a set of procedures which can be invoked from outside to operate on the object.

SIMULA designers realized that objects were a useful idea to simulate systems with inherent concurrency as those modeled in simulation. SIMULA objects are in fact allowed to run in pseudo-parallelism by two different mechanisms:

1. Objects can execute in coroutine fashion by resuming each other.

2. The simulation package allows a simple form of interaction with the system scheduler on the basis of delays with respect to a simulated time.

In the LISP languages, objects are embodied by closures. A closure is a function plus an environment. The environment, which keeps the values associated to variables used within the function, represents the local state of the closure. When the closure is invoked, its function is applied in that environment, therefore it can modify the state by means of assignment.

In the case of truly concurrent systems, however, the assignment command is not suitable by itself as the basis for change of behavior because it does not deal with the problem of scheduling access to a shared object, so that timing errors can be avoided. To deal with this problem C.A.R. Hoare [Hoare 1974] proposed an adaptation of the Simula class construct called *monitor* as an operating systems construct for higher level languages. By imposing the constraint that only one invocation can be active at one time, monitors provide a mean for achieving synchronization and mutual exclusion. Monitors retained most of the aspects of sequential programming languages including:

Use of the assignment command to update variables of the monitor

Requiring sequential execution within the procedures of a monitor

However, one of the most criticized aspect of monitors is the use of low level *wait* and *signal* primitives to manipulate the queues of the scheduler of the operating system of the computer. The effect of the execution of such instructions is the release of the monitor by the process presently executing inside the monitor and transfer of control to some other processes. Thus control "jumps around" inside a monitor in a way which is not obvious from the structure of the code.

Monitors do a good job of incorporating important abilities of the operating system of a sequential computer in a high level language. As an "operating systems structuring concept", monitors are intended as a mean of interaction among parts of an operating systems. These components are all at the same level and each of them bears some responsibility for the correct behavior of the system as a whole. In fact monitors basically support the ability for processes to synchronize and notify each other when some action is performed on shared data. However, the correct use of the resource, or the consitency of its state when a process leaves the monitor or the guarantee that each process will eventually release the monitor, usually cannot be established from the code of the monitor alone.

In the setting of a distributed system, it seems more appropriate that the responsibility in the use of a shared resource be delegated to a specific guardian for the resource. All users will act as inferiors to the guardian. In fact it is unreasonable to expect that each user is aware of the protocols to be followed in accessing the resource and that each user will provide a guarantee on its "correct" usage.

The abstraction that has been developed for this purpose has been termed *guardian* [Hewitt, Attardi and Lieberman 1979; Svobodova, Liskov and Clark 1979, Dennis 1980]. The purpose of a guardian is to provide an interface to users for performing operations on a protected resource. The guardian is responsible to synchronize possibly concurrent requests, to schedule the access to the resource, provide protection and ability to recover from failures. The job of a guardian is to accept requests for operations and act on behalf of the requesters to carry out such operations. Those requesting service are not allowed to act directly on the resource; a property which we call *absolute containment.*

We will further discuss the concept of guardians and what is required from a programming language to support a guardian abstraction. Some of the recently proposed programming languages, such as ADA [Ichbiah 1980] and CSP [Hoare 1978], are not adequate to support such an abstraction. The discussion will focus on a specific language, called **Act1**, which is based on the Actor Model of Computation.

Actors are a generalization for concurrent systems of both the functional objects of the lambda calculus and the Simula objects which can change state. **They solve the problem of providing both inherent concurrency and the ability for shared objects to change state.** Message passing is the uniform means of communication between actors. Other known mechanisms for interaction, such as procedure invocation or coroutining, can be interpreted in term of message passing, as specific patterns of communications [Hewitt

1975]. Serialized actors are a novel construct which combines both the synchronization facilities of primitives like monitors or Hoare's guards, and the ability to perform state changes of assignment commands. Serialized actors might change their behavior in the course of a computation. When a serialized actor receives a message, it can in fact designate a replacement for itself to receive the next message delivered to it.

III -- The Actor Model of Computation

The actor model is based on fundamental principles that must be obeyed by all physically realizable communication systems. Computation in the Actor Model is performed by a number of independent computing elements called *actors*. Hardware modules, subprograms, and entire computers are examples of things that may be thought of as actors.

A computation is carried out by actors that communicate with each other by *message passing*. Examples of such communications are electrical signals, parameter passing between subroutines of a program and messages transferred between computers in a geographically distributed network. A hardware module might receive operands and function codes on a bus, while the subprograms might receive values or locations of parameters, and the computer might receive communications in packets.

Conceptually one actor communicates with another using a *mail address*. The operation which can be performed on a mail address is to send communications. Thus a mail address is quite a different concept from a machine address which has read and write as the defined operations. There are a variety of ways to physically implement mail addresses including copper wires, machine addresses, and network addresses.

An actor performs computation as a result of receiving a *communication*. The actor model refers to the arrival of a communication at an actor as an *event*. As a result of receiving a communication, an actor may produce other communications to be sent to other actors. Speaking in term of events, this means that an event may *activate* some other events. Events are hence related by an *activation* ordering [Hewitt and Baker 1977].

Also as an effect of an event, an actor can create some new actors and can also designate another actor to take its place, to receive the next delivered communication.

An actor can take the following actions in processing a message received:

1. It can create new actors;

2. It can send more messages;

3. It can specify a replacement actor which it will become and which will handle the next communication received.

All such actions are specified by the *behavior* of the actor. A behavior is a function which maps a communication received to a three tuple consisting the actors created as a result of receiving the message, the communications sent as a result of receiving the messages, and a next behavior.

Communications received by each actor are related by the *arrival* ordering, which expresses the order in which communication are received by the actor. The arrival ordering of a serialized actor is a total ordering, i.e. for any two communications received by an actor, it always specifies which one arrived first. Some form of arbitration is usually necessary to implement the arrival ordering for shared actors.

A computation in the actor model is a partial order of events obtained by combining the activation ordering and all of the arrival orderings. The actor model incorporates properties that any physically realizable communication system must obey. In fact, not all partially ordered sets of events can be physically realized. For instance, no physically realizable computation can contain two events which have a chain of infinitely many events in between them, each one activating the next one.

IV -- Related Work

A number of models have been recently developed for concurrent systems. These models differ in their conception of communication, in what can be communicated, and in the ability to dynamically create new computational agents.

IV.1 --- Nature of Communication

For some of these models [Hoare 1978, Milner 1979], the mechanism of communication resembles a telephone system, where communication can occur only if the called party is available at the time when the caller requests the connection, i.e. when both parties are simultaneously available for communicating. For the actor model, however, message passing resembles mail service, so that messages may always be sent but are subject to variable delays en route to their destinations. Communication via a mail system has important properties that cause it to differ with "hard-wired" connection:

> **Asynchronony:** The mail system decouples the sending of a message from its arrival. It is not necessary for the recipient to rendezvous with the sender of a messages.
>
> **Buffering:** The mail system buffers messages between the time they are sent and the time they are accepted by the recipient.

We have found the properties of asynchrony and buffering to be fundamental to the widespread applicability of actor systems. They enable us to disentangle the senders and receivers of messages raising the level of the description.

IV.2 --- Functionality of Mail System

The implementation of an actor system entails the use of a mail system to perform communications. Such mail system will transport and deliver the communication by invoking hardware modules, activating actors defined by software, or sending messages through the network as appropriate. The following functionality is provided by the mail system.

Routing. The mail system routes a message to the recipient over whatever route seems most appropriate. For example it may be necessary to route the message around certain components which are malfunctioning. The use of a mail system contrasts with systems which require a direct connection in order for communication to take place.

Forwarding. The mail system must also forward messages to actors which have migrated. Migration can be used to perform computational load balancing, to relieve storage overpopulation, and to implement automatic real-time storage reclamation.

IV.3 --- Sending Mail Addresses in Messages

An important innovation is that mail addresses of actors can be sent in messages. This ability provides the following important functionalities:

Public Access. The receiver of a message does not have to anticipate arrival of message in contrast to systems (such as CSP) which require that a recipient know the name of the sender before any message can be received, or more in general to systems where element interconnections are fixed and specified in advance.

Reconfiguration. Actors can be put in direct contact with one another after they are created since actors can be sent in messages.

In some models the mobility of objects is limited. For instance CSP and CCS only allow communications composed of elementary data types such as integers, reals and character strings. Processes or other nonprimitive objects cannot be transmitted. Such limitation is related to the corresponding restriction on the reconfiguration of the system. Reconfiguration is not possible in systems such as CSP which require that a process be created knowing exactly the processes with which it will be able to communicate throughout its entire existence.

IV.4 --- Dynamic Creation of Actors

New actors can be dynamically created as a result of an actor receiving a message. The creator of an actor is provided with a mail address that can be used to communicate with the new actor. Reconfiguration (see above) enables previously created actors to communicate with the new ones.

IV.5 --- Mathematical Models

Mathematical models for actor systems rigorously characterize the underlying physical realities of communication systems. In this respect they share a common motivation with other mathematical models which have been developed to characterize physical phenomena. The actor model differs in motivation from theories developed for reasons of pure mathematical elegance or to illustrate the application of pre-existing mathematical theories (modal logic, algebra etc.) The actor model of computation has been mathematically characterized pragmatically [Greif 1975], axiomatically [Hewitt and Baker 1977], operationally [Baker 1978], and in terms of power domains [Clinger 1981].

An important innovation of the actor model is to take the arrival ordering of communications as being fundamental to the notion of concurrency. In this respect it differs from systems such as Petri Nets and CSP which model concurrency in terms of nondeterministic choice (such as might be obtained by repeatedly flipping a coin). As will be seen later in this paper, the use of arrival ordering has a decisive impact on the ability to deal with fundamental issues of software engineering such as being able to prove that a concurrent system will be able to guarantee a response for a request received.

IV.6 --- Generality

As a consequence of flexibility provided by the mail system and the ability to dynamically create actors, the actor model can be used to analyze a wide spectrum of computer systems. We have found it suitable for describing hardware and software aspects of multiprocessor systems including distributed systems. It also seems to be quite appropriate to describe the kind of *self-timed systems* that arise in VLSI technologies [Seitz 1979]. The generality of the actor model distinguishes it from other models of concurrent systems developed for more narrow contexts.

V -- The Actor Language Act1

When you speak a new language you must see if you can translate all of the poetry of your old language into the new one.
Dana Scott

Act1 is an experimental language based on the actor model of computation. Act1 is universal in the sense that any physically realizable actor system can be implemented in it.

Henry Lieberman has implemented a preliminary version of Act1 on the PDP-10 and conducted some interesting experiments in its use.

V.1 --- Notation

Act1 borrows from LISP the syntactic notation for expressions, i.e. each expression is enclosed in parentheses with the elements of the expression separated by white space. This notation, which is essentially a parenthesized version of Polish prefix notation, has the advantage that all expressions have a uniform syntax at the level of expression boundaries. This is however only a superficial resemblance between Act1 and LISP. Most of the new semantic notions in Act1 (such as serializers, customers, transaction managers, etc. which are not present in Lisp) stem from the fact that Act1 was designed to implement concurrent systems whereas Lisp was designed and has evolved to implement sequential procedures on a sequential computer.

V.2 --- Omega Descriptions

Act1 embeds a subset of the Description System Omega [Hewitt, Attardi, and Simi 1980]. Omega combines ideas from the predicate calculus and the theory of types. From the former it derives the ability to express arbitrary properties; from the latter the idea of inheritance of properties. Descriptions are used to express properties, attributes, and relations between objects.

Data types in programming languages types have come to serve more and more purposes in the course of time. Type checking has become a very important feature of compilers to provide type coercion, to help in optimization, and to aid in checking consistent use of data. The lack of power and flexibility in the type systems of current programming

languages limits the ability of the languages to serve these purposes. Omega helps to overcome these limitations.

We use Omega to express assumptions and the constraints on objects manipulated by programs in **Act1**. These descriptions are an integral part of the programs and can be used both as checks when the programs are executing and as useful information which can be exploited by other systems which examine the program such as translators, optimizers, indexers, etc. We believe that bugs occurring in programs are frequently caused by the violation of implicit assumptions about the environment in which the program is intended to operate. Therefore many advantages can be drawn by a system that encourages the programmer to express such assumptions explicitly, and by a system which is able to detect when they are violated.

In this paper we use Omega mainly to describe the data, procedures, and messages used in our programs. Examples of Omega descriptions are presented in the next section for describing communications.

V.3 --- Communications and Customers

Communications are the units of information that are transmitted from an actor to another. There are different kinds of communications each one with possibly different attributes. **Act1** provides mechanisms to enable an actor to distinguish between kinds of communications which it receives and to select their attributes, by means of simple pattern matching.

A request is a kind of communication which always contains a message and a customer:

```
(a Request) is (a Request (with message (a Message)) (with customer (a Customer)))
```

The notion of customer generalizes the concept of **continuation** introduced in the context of denotational semantics [Strachey and Wadsworth 1974, Reynolds 1974] to express the semantics of sequential control mechanisms in the lambda calculus. In that context a continuation is a function which represents "the rest of the computation" to which the value of the current computation will be given as an argument. A customer is analogous to a continuation, in that a reply is sent to the customer when the transaction activated by the request is completed.

A Response is another kind of communication and can be either a Reply or Complaint:

(*a* Response) *is* (*a* Reply (*with* message (*a* Message)))
or
(*a* Complaint (*with* reason (*a* Message)))

The first reply received by a customer is usually treated differently than any subsequent reply. In general subsequent replies will be treated as errors and generate complaints.

We would like to remark that the notion of customers subsumes and unifies many less well defined concepts such as a "suspended job" or "waiting process" in conventional operating systems. In fact the ability to deal explicitly with customers unifies all levels of scheduling by eliminating the dichotomy between programming language scheduling and operating system scheduling found in most current systems.

V.4 --- Communication Primitives

Act1 provides primitives to perform unsynchronized communication. This means that an actor sending the communication simply gives it to the electronic mail system. It will arrive at the recipient at some time in the future. I.E. an actor can transmit and receive communications while messages that it has sent are in transit to their destinations.

Executing a command of the form

(*SendTo* t c)

results in the transmission to the target actor t of a communication c.

Transmitting communications using this primitive is a very convenient method for causing more parallelism. The usual method in other languages for creating more parallelism entails creating processes (cf. ALGOL-68, PL-1, Communicating Sequential Processes etc.). The ability to engender parallelism by transmitting communications is one of the differences between actors and the usual processes in other languages.

A few higher level communication commands are also provided, which can be however expressed in term of the previous ones.

A very frequent pattern of use of communications is to send a request to an actor and receive the reply back from it for use in further computations. A special notation is used for this common case of two way communication:

(ask t r)

results in the transmission to the target actor t of a request communication containing r as message. The appropriate customer is automatically created so that the expression *(ask* t r) can be used as denoting the value of the reply.

To send a reply, a command of the form

(ReplyTo t v)

can be executed, which is an abbreviation for

(SendTo t (*a* Reply (*with* message v)))

Replies indicate the successful completion of a request. The *ReplyTo* command packages up the reply value v as a Reply so that an arbitrary actor can be sent as a reply and still have the recipient understand that it is receiving a reply indicating successful completion of the request. To indicate that a request has not been successfully completed, a command of the form

(ComplainTo t m)

can be used to send a complaint with message m to the target actor t i.e. it is an abbreviation for

(SendTo t (*a* Complaint (*with* message m)))

V.5 --- Behaviors

A behavior is what characterizes an actor. The behavior of an actor determines which communications the actor can accept, which action it will take as an effect of receiving a communication, and what will be its subsequent behavior.

V.5.a --- Description of Behaviors

A behavior can be described using an expression of the following form:

(*a* Behavior
  (⟨PatternForCommunication⟩$_1$ *communication* ⟨body⟩$_1$)
  ...
  (⟨PatternForCommunication⟩$_n$ *communication* ⟨body⟩$_n$))

V.5.b --- Creation of Behaviors

To create an new actor with a specified behavior, the following notation is used:

(*the* Behavior
  (⟨PatternForCommunication⟩$_1$ *communication* ⟨body⟩$_1$)
  ...
  (⟨PatternForCommunication⟩$_n$ *communication* ⟨body⟩$_n$))

When an actor with a behavior of this kind receives a communication which matches any of the patterns, then the corresponding body is executed. If the communication matches more than one of the patterns, then an arbitrary one of the corresponding bodies is selected to be executed. If this is the case, it is a good programming practice to ensure that the results are the same in both cases or that it is immaterial which one is selected.

V.6 --- Serializers

A serialized actor can only process one message at a time. A suitable implementation might rely on a first-in-first-out queue where communications received when the serializer is processing a previous message are queued, until they can be accepted and a scheduling decision made.

The Act1 syntax for the creation of an actor is:

(*the* Behavior <MessageHandler>$_1$ ... <MessageHandler>$_n$)

As we mentioned before, an actor after processing a message can specify the actor which it will become in order to process the next message received. The replacement actor is designated using an expression of the following form

(*become* <ExpressionForReplacement>)

The replacement is the actor which will handle the next communication received.

An important special case occurs where none of the message handlers have *become* commands. Actors whose behavior has no *become* commands are called unserialized and are treated specially by the implementation. Actors such as the text of the Lincoln's Gettysburg Address and the square root function are unserialized.

An important efficiency consideration is that a serialized actor should specify the actor which is its replacement as quickly as possible. In fact the *only* limitation on parallelism in systems constructed using Act1 derives from the speed with which replacement actors can be computed.

Accepting communications in the order in which they arrive does not involve any loss of generality because the communication need not be acted on when it is accepted. For example a request to print a document need not be acted on by a guardian when it accepts the request. The guardian can remember the request for future action and in the meantime accepts messages concerning other activities. Processing of a request can resume at any time by simply retrieving it from where it is stored. The ability to handle customers like any other actors allows guardians to organize the storage of requests in progress in a variety of ways. Unlike monitors guardians are not limited to the use of a couple of predefined specialized storage structures such as queues and priority queues.

In other languages which do not support the concept of a customer (such as Communicating Sequential Processes and ADA), the acceptance of a request must be delayed until the proper conditions are met for processing it. This usually requires complicated programming constructs to guard the acceptance of messages.

VI -- Guardians

A **guardian** is an actor that can be used to regulate the use of shared resources by scheduling access, providing protection, and implementing recovery from hardware failures. A shared checking account is an example of the kind of resource that needs a guardian. A guardian must be able to implement any appropriate scheduling policy to process a request. We introduce the notions of a customer and transaction manager for a request and illustrate how to use them to implement arbitrary scheduling policies for a guardian.

In this paper we show a programming methodology for the construction of guardians that is based on these principles. We start by defining the data structures which describe the state of the resource. We then specify the constraints on these data structures which have to be maintained by the guardian. Then we specify the operations that are allowed on the resource. A scheduling policy is then devised for the operations on the resources. Finally code for the guardian can be produced that embodies the scheduling policy and provides service for all requests while fulfilling the requirement of consistency expressed by the constraints.

VII -- A Simple Checking Account Guardian

As a first example of how Act1 can be used to implement guardians, we give the implementation of a very simple checking account guardian which ensures that timing errors do not occur when concurrent attempts are made to deposit or withdraw money in an account.

In our simple case a single value is sufficient which represents the balance of the account will be sufficient to characterize the state of the account. The appropriate constraint that the guardian has to maintain is the fact that the balance is not allowed to become negative. The operations allowed on the account are withdrawal and deposit operations. They are to be performed as soon as they are accepted; therefore no particular scheduling policy has to be implemented. We first present an abstracted partial description (interface specification)

```
((an Account (constraint balance (a NonNegativeUSCurrency)))
   is (a Behavior
          ((a Request (with message (or (a Deposit) (a Withdrawal))) (with customer c))
                communication
                      (ReplyTo c (a CompletionReport)))))
-- ; Responses to deposit and withdrawal messages are guaranteed
```

which is separate from the implementation.

From this analysis of the problem, we produce the implementation of the checking account guardian given below. In order to aid more complete understanding of the foundations of message passing, the implementation is written out in full without using any of the syntactic short cuts that are available in Act1 to make the procedure more concise.

The notation for the implementation of an actor is close to the Omega notation. An important difference from Omega is that Act1 uses the definite article *the* to create a new serialized actor. For instance (*the* Account) written below is a specific implementation of the generic (*an* Account) described above. Whenever an expression like (*the* Behavior ...) is evaluated, a new actor is created with the corresponding behavior. An appendix of this paper contains an explanation of the Act1 *CaseFor* primitive.

```
((the Account (constraint balance (a NonNegativeUSCurrency)))
  is
   (the Behavior
        ((a Request (with message (a Withdrawal (with amount w))) (with customer c))
             communication
             (CaseFor w
                       (is (≤ balance) then
                            (ReplyTo c (a CompletionReport))
                            (become (the Account (with balance (balance - w)))))
                       (is (> balance) then
                            (ComplainTo c (an OverdraftComplaint))
                            (become (the Account (with balance balance))))))
        ((a Request (with message (a Deposit (with amount d))) (with customer c)) communication
                  (ReplyTo c (a CompletionReport))
                  (become (the Account (with balance (balance + d)))))))
```

Below we present the very same implementation written using abbreviations which elide the explicit mention of the customers involved. This is accomplished by using communication handlers of the form

```
(<PatternForMessage> request
                  ...
                  (reply <ExpressionForReply>)
                  ...
                  (complain <ExpressionForComplaint>)
                  ...)
```

which is an abbreviation for

```
((a Request (with message <PatternForMessage>) (with customer c)) communication
        ...
        (ReplyTo c <ExpressionForReply>)
        ...
        (ComplainTo c <ExpressionForComplaint>)
        ...)
```

We can also elide mentioning the new behavior, in the case it is not changed.

```
((the Account (constraint balance (a NonNegativeUSCurrency)))
  is
    (the Behavior
      ((a Withdrawal (with amount w)) request
          (CaseFor w
                    (is (≤ balance) then
                          (reply (a CompletionReport))
                          (become (the Account (with balance (balance - w)))))
                    (is (> balance) then
                          (complain (an OverdraftComplaint)))))
      ((a Deposit (with amount d)) request
                (reply (a CompletionReport))
                (become (the Account (with balance (balance + d)))))))
```

VIII -- Data Structures

Data structures in Act1 derive from the description system Omega. In this section we show how to characterize simple First-In First-Out queues in Omega. Various kind of queues are quite useful in the implementation of concurrent systems. Later in the paper we show how to extend these ideas to more realistic situations.

Omega facilitates providing multiple partial descriptions of objects. For example (*a* Queue (*with* front 0)) is a description of an **instance** of a queue whose front is 0. Note that we have used the indefinite article "*a*" to mark descriptions of instances of a concept. Descriptions can in turn be multiply described. For example the following says that an empty queue is a queue:

```
((an EmptyQueue) is (a Queue))
```

Note that the *is* statement is asymmetric so we cannot *incorrectly* deduce that

```
((a Queue) is (an EmptyQueue))
```

which is fortunate since not every queue is empty. We can write equations using bidirectional inheritances so that in general

```
((d1 same d2) ⇔ ((d1 is d2) ∧ (d2 is d1)))
```

We can also use prepositional phrases to specialize instance descriptions as in the following statements:

```
((a Queue (with front 3)) is (a Queue))
¬((a Queue (with front Something)) is (an EmptyQueue))
```

A NonEmptyQueue can be described as follows:

```
((a NonEmptyQueue) same
        (a Queue (with front Something) (with AllButFront (a Queue))))
```

Note that by using the concept NonEmptyQueue on the left hand side of the above equation we have specified that *every* NonEmptyQueue has two attributes front which can be anything and AllButFront which must be a Queue. Furthermore every queue with a front and an AllButFront is in turn a NonEmptyQueue.

We can describe a queue with front x and all but front an empty queue as being the same as a queue with rear x and all but rear an empty queue:

```
((a Queue (with front x) (with AllButFront (an EmptyQueue))) same
  (a Queue (with rear x) (with AllButRear (an EmptyQueue))))
```

Local identifiers like x play a role in Omega similar to the role played by free identifiers in formulas in the quantificational calculus: they can be bound to any object. For example

```
((a Queue (with front 3) (with AllButFront (an EmptyQueue))) same
  (a Queue (with rear 3) (with AllButRear (an Emptyqueue))))
```

The above descriptions express some of the relations that hold for queues. We believe that it is important to allow information to be presented in an incremental fashion. For example it should be possible for the user to later *further* describe properties of queues such as the following which states an important relationship between the front, rear, and middle of a queue.

```
((a Queue
        (with front f)
        (with AllButFront (a Queue (with rear r) (with AllButRear m))))
 same
  (a Queue
        (with rear r)
        (with AllButRear (a Queue (with front f) (with AllButFront m)))))
```

The above description together with the ones given early completely describes queues in the same sense that axiomatizations of queues in the quantificational calculus or data algebras can completely describe queues. It is important to realize that in giving the above descriptions the user is not making any commitments as to the physical representation of queues. The possibility is still open that queues will be physically represented as doubly linked lists, arrays, some mixture, or still some other alternative physical representation. It is even possible that more than one physical representations will **cohabit** the same system.

Queues of the kind described are highly suited for use concurrent systems. Their use enables us to increase the concurrency in the server defined below since the activity of constructing a new queue with an additional element can be concurrent with the server processing other requests. The construction of a queue with a new element x in addition to the elements already in a queue q is specified to place the new element at the rear of q as follows:

```
((an Enqueuing (with NewEntry x) (with OldQueue q)) is
        (a Queue (with rear x) (with AllButRear q)))
```

IX -- Implementation of a Hard Copy Guardian

Implementing a module to service printing requests on a continuously available system provides a concrete example to illustrate the flexibility of guardians. The example illustrates how to implement a guardian that protects more than one resource (in this case two printers).

IX.1 --- Interface of a Hard Copy Guardian

Below we give a behavior specification for the hardcopy guardian. We would like to contrast this kind of specification with alternatives such as state specifications or specification in terms of mathematical functions. Mathematical functions are appropriate for specifying properties of modules where the major interest lies in specifying the output as a function of the input. Specification of the hard copy guardian as a mathematical function is not satisfactory because the output is always always the same: a reply that the printing request has been complete. State change is an appropriate way to specify properties of a command where the major interest lies in specifying what changes as a result of executing the command. For example the assignment command x:=3 can be

precisely described by saying that it changes the value of x to be 3. Specification of the hard copy guardian in terms of a state change is unsatisfactory because its functionality is only distantly related to its change in state from the time before executing a command to print a document and the time when a reply is received that the document has been printed.

The specification of a guardian can be expressed as a behavioral specification.

```
((a HardCopyGuardian (constraint device1 (a Printer)) (constraint device2 (a Printer)))

 is
     (a Behavior
          ((m WhichIs (a PrintRequisition)) request (reply (Ask (a Printer) m)))))
```

This specification expresses the fact that all print requests will be handed to one of the printers and the reply from the printer sent back to the requestor.

The hard copy guardian relies on receiving a response from a printer when the printing is completed. More formally the guardian relies on the following behavior specification of printers:

```
((a Printer) is
          (a Behavior
               ((a Request (with message (a PrintRequisition)) (with customer c))
                    communication
                         (ReplyTo c (a PrintingCompletedReport)))))
```

IX.2 --- Modularity of Guardians

We have modularized the implementation of the guardian for a hard copy server into five separate modules:

1. The **Guardian** which provides the external interface to the outside world.

2. **Servers** which deal with the issues of managing transactions for a shared resource.

3. **A Scheduling Structure** which keeps a record of pending work to be done.

4. **Transitions** which computes what should be done as result of a server entering a new state.

5. **Transaction Managers** which handle the information for particular transactions and their communications.

We have found that the above modularization to be generally applicable to problems in implementing shared resources. The same modularity will be preserved in a follow on paper to this one in which we deal with issues of unreliable communication.

IX.3 --- Diagrams of a Simple Transaction

The Hard Copy guardian starts performing a transaction when it receives a communication which satisfies the following description:

```
(a Request
    (with message (PR WhichIs (a PrintRequisition)))
    (with customer C))
```

The transaction will be eventually carried on by sending one of the printing devices a communication which satisfies the description

```
(a Request
    (with message PR)
    (with customer M))
```

where M is a transaction manager. The transaction will be completed when the reply produced by the printer will be sent back to the original customer C:

```
(a Reply (with message V))
```

The complete sequence of events taking place during the processing of a transaction is presented in the following diagrams.

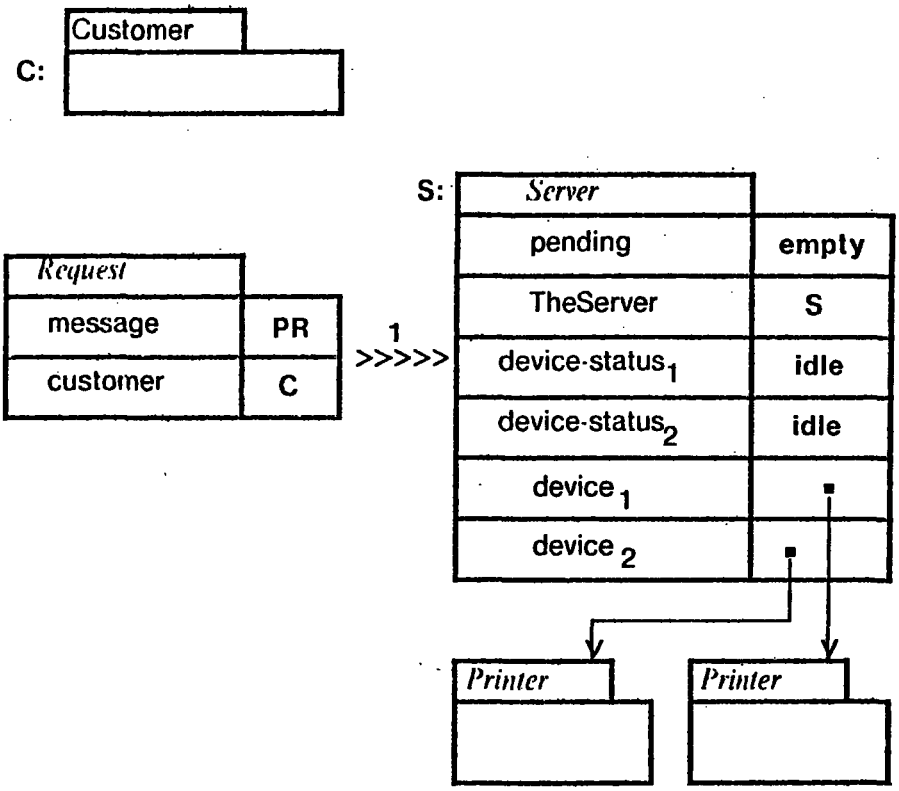

Figure 1.
The first request is accepted by the server G.

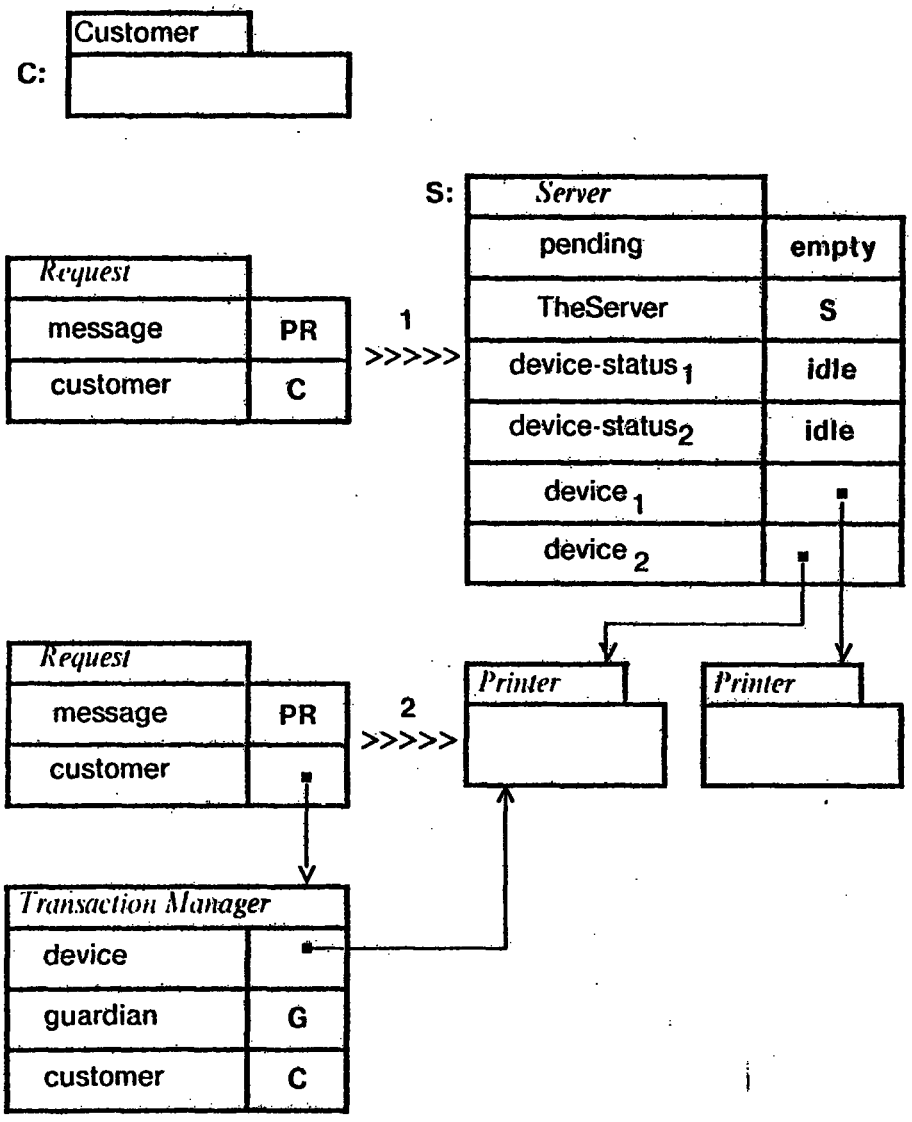

Figure 2.
A manager is created for handling the transaction and the request is passed to a printer.

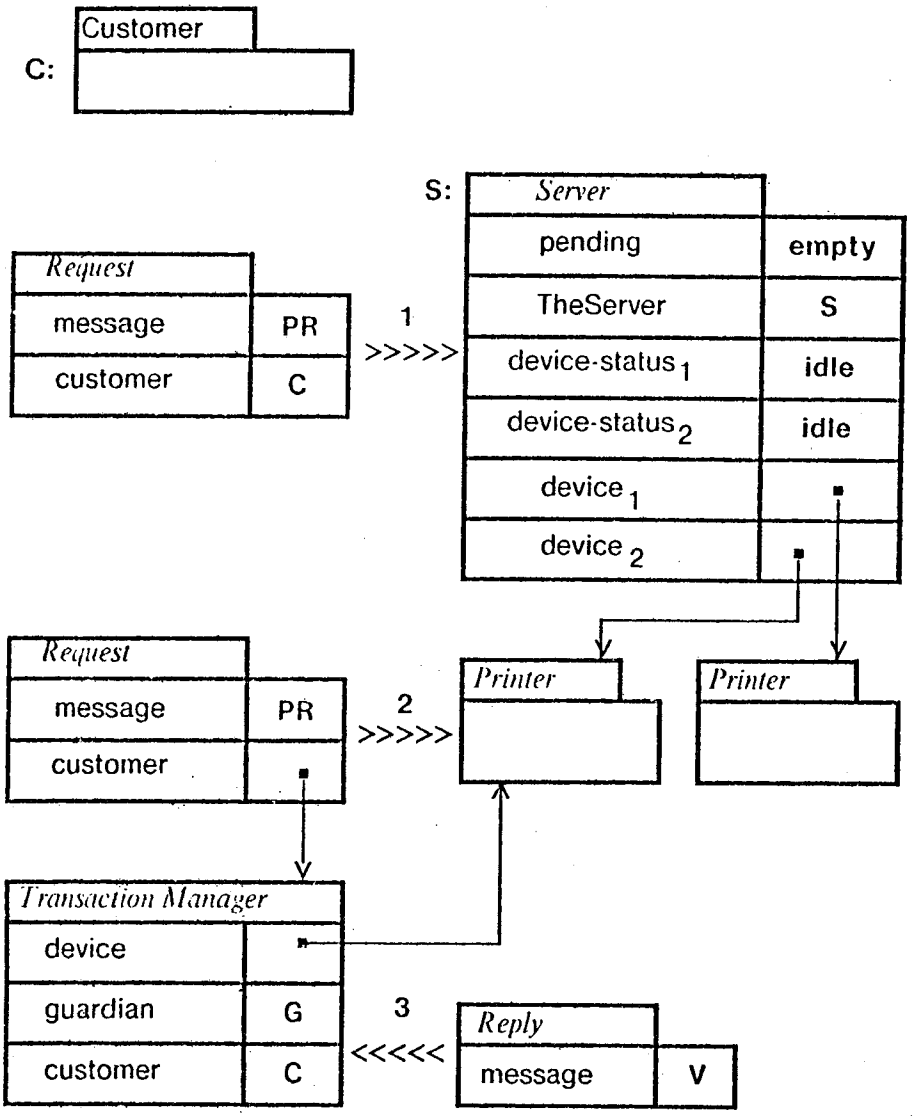

Figure 3.
The reply from the printer is received by the transaction manager

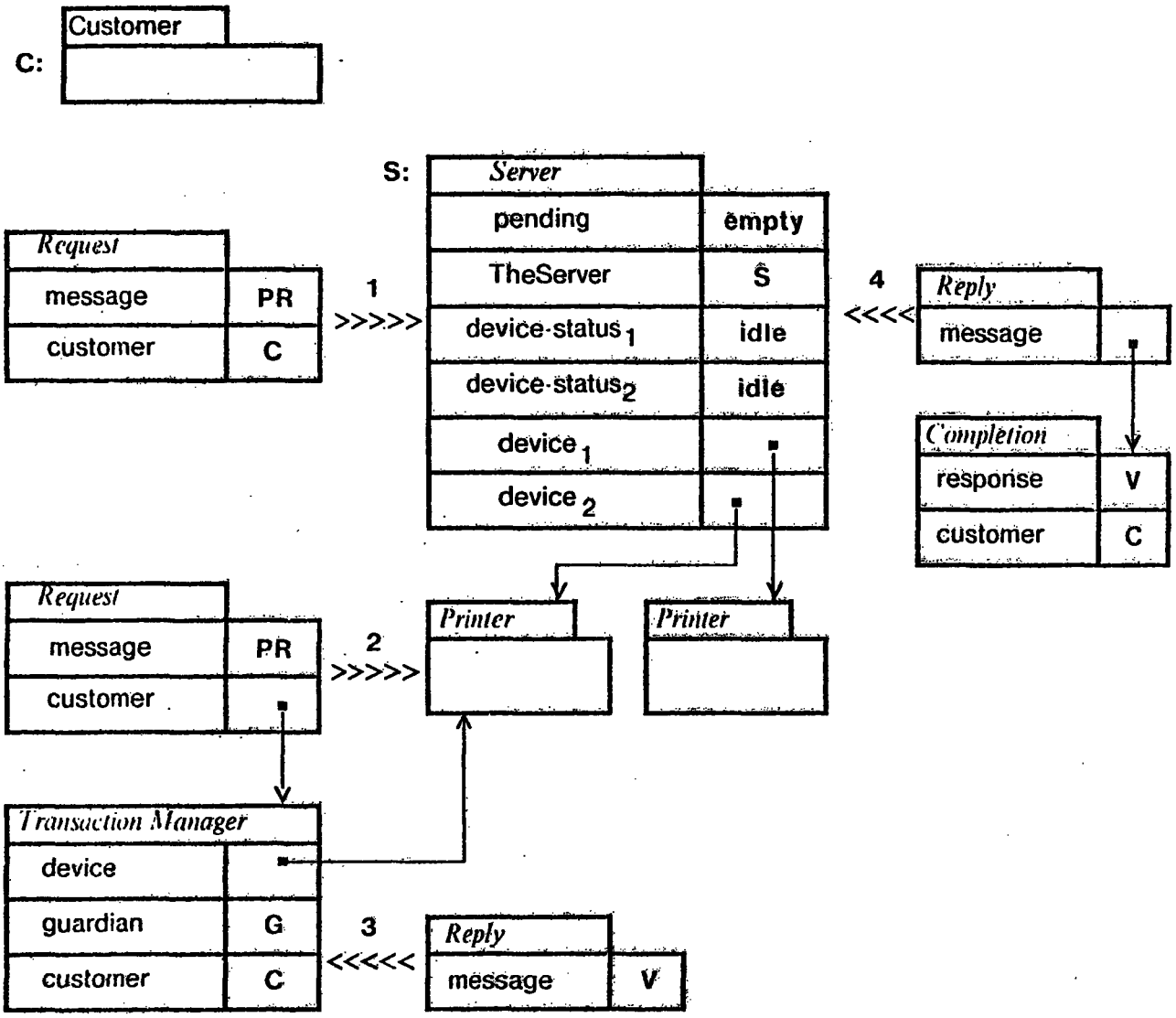

Figure 4.
The server accepts the reply from the manager stating that the transaction has been completed

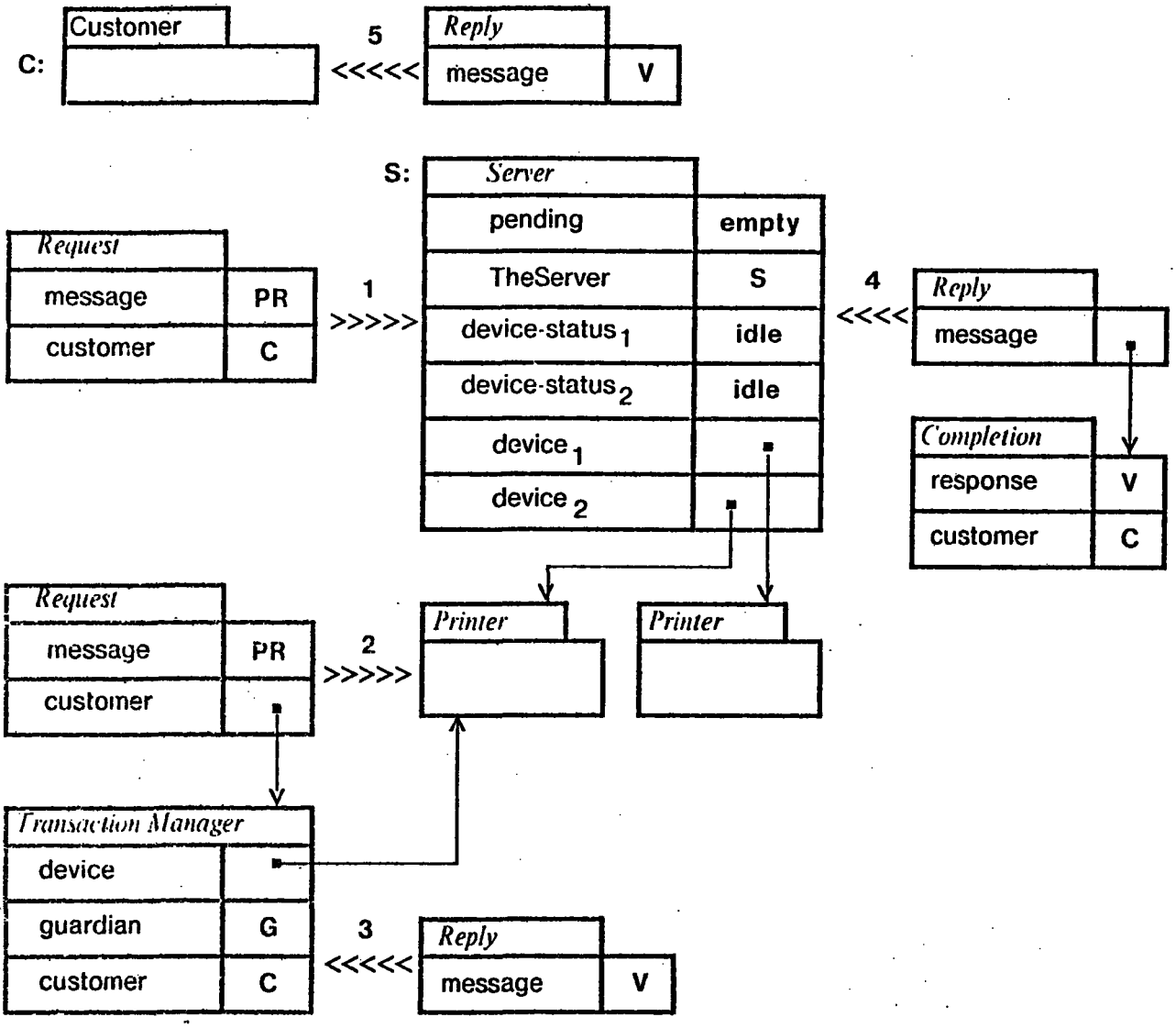

Figure 5.
The customer C receives the reply to the original request

In the actor model sending messages is decoupled from their arrival. It is not necessary for any kind of "rendezvous" to take place between the sender and receiver. A considerable period of time might pass between the time when an message is sent and the time when it arrives. This circumstance is modeled in the behavior specification for the hard copy guardian given below by having separate events for the sending and arrival of a communication.

IX.4 --- An Implementation

The operation of the hard copy guardian is illustrated for a typical transaction by the diagrams given in this paper. They give an overview of salient events which occur in the course of a printing transaction. We recommend that you look over them briefly now and then refer to them more carefully as you read through the implementation.

The overall state of the resource protected by the hard copy guardian can be described by such a queue of pending requests, and by the state of the two printers. This information will be kept in three state variables described as follows:

pending *is* (*a* Queue (*constraint* element (*a* PrintRequisition)))
DeviceStatus$_1$ *is* (*or* idle printing)
DeviceStatus$_2$ *is* (*or* idle printing)

The guardian has to maintain the fundamental constraint among these data, corresponding to the fact that no device should be idle if there are pending requests:

pending *is* ((*an* EmptyQueue) *provided* (v (DeviceStatus$_1$ *is* idle) (DeviceStatus$_2$ *is* idle)))

The description connective *provided* can be translated into logical implication as follows:

((v (DeviceStatus$_1$ *is* idle) (DeviceStatus$_2$ *is* idle)) ⟹ (pending *is* (*an* EmptyQueue)))

The hard copy guardian is provided with the mail addresses of two devices which are printers when it is created. The function of the guardian is to set up and initialize a hard copy server. The hard copy server accepts printing requests and communicates with the printing devices. It maintains records of the status of the printing devices and of pending requests in order to the schedule the printers.

```
(the HardCopyGuardian (constraint device1 (a Printer)) (constraint device2 (a Printer)))

 is
  (label S                          ; S is the name of the following serialized actor
       (the Server
            (with pending (an EmptyQueue))
            (with DeviceStatus1 idle)
            (with DeviceStatus2 idle)
            (with TheServer S)))
```

IX.5 --- Scheduling Structure

In our first implementation of the guardian, we will provide that print requests submitted to the guardian will be served in the order in which they are received. In general though, they cannot be served immediately, because no printer might be available at that time. This means that a FIFO queue will be used as appropriate scheduling structure for pending requests. This version will use as a scheduling structure the queue data structure as described in the section on data structures above.

IX.6 --- Server Behavior

As in the case of the guardian, we first give an interface specification for a server and then present its implementation.

IX.6.a --- Interface Specification

```
((a Server
    (constraint device1 (a Printer))
    (constraint device2 (a Printer))
    (constraint pending
            (a Queue (constraint element (a PrintRequisition)))
            ((an EmptyQueue) provided (v (DeviceStatus1 is idle) (DeviceStatus2 is idle))))
    (constraint DeviceStatus1 (or idle printing))
    (constraint DeviceStatus2 (or idle printing))
    (constraint TheServer (a HardCopyServer)))

    is
        (a Behavior
            ((m WhichIs (a PrintRequisition)) request
                    (Ask (a Printer) m))
            ((a CompletionReport (with customer c)) reply
                    (ReplyTo c (a PrintingCompletedReport)))))
```

IX.6.b --- Implementation

```
(the Server
    (constraint device1 (a Printer))
    (constraint device2 (a Printer))
    (constraint pending (a Queue (constraint element (a PrintRequisition))))
    (constraint DeviceStatus1 (or idle printing))
    (constraint DeviceStatus2 (or idle printing))
    (constraint TheServer (a HardCopyServer)))

    is

      (the Behavior
       ((a Request (with message (m WhichIs (a PrintRequisition))) (with customer c)) communication
          (become (the NextServer
                       (with pending (an Enqueuing
                                          (with OldQueue pending)
                                          (with NewEntry
                                                (a Request
                                                     (with message m)
                                                     (with customer c))))))))
                  ;the next server is the value of the (the NextServer) expression above

       ((a CompletionReport (with device devicei) (with response r) (with customer c)) reply
                  ;a completion reply means that devicei has stopped printing
             (ReplyTo c r)
                  ;the reply is transmitted to the customer
             (become (the NextServer (with DeviceStatusi idle)))))
                  ;the replacement is the NextServer with devicei idle
```

We have adopted in the above code and in our language a useful convention for giving default values to missing attributions in a description or missing argument in a function call. For instance in the above code the expression

```
(the NextServer
     (with pending (an Enqueuing
                        (with OldQueue pending)
                        (with NewEntry
                             (a Request
                                  (with message m)
                                  (with customer c))))))
```

is considered to be equivalent to

```
(the NextServer
     (with pending (an Enqueuing
                         (with OldQueue pending)
                         (with NewEntry
                                 (a Request
                                       (with message m)
                                       (with customer c)))))
     (with DeviceStatus1 DeviceStatus1)
     (with DeviceStatus2 DeviceStatus2)
     (with TheServer TheServer))
```

This convention allows us to shorten our notation by avoiding the repetition of all the attributions that are left unchanged.

Below we define the NextServer which computes the next replacement.

IX.7 --- Transitions

The concept of a NextServer serves to modularize the change of behavior of a *Server*.

The program below illustrates how nondeterminism can manifest itself in actor systems. If both devices are idle when a print request is received, then a nondeterministic choice is made which printer to use since it doesn't matter which one is chosen. The actual choice is resolved by the actual physical implementation such as by the arrival order of communications in a highly parallel architecture or perhaps by the choice of compiler.

```
((the NextServer
    (constraint device1 (a Printer))
    (constraint device2 (a Printer))
    (constraint pending (a Queue (constraint element (a PrintRequisition))))
    (constraint DeviceStatus1 (or idle printing))
    (constraint DeviceStatus2 (or idle printing))
    (constraint TheServer (a HardCopyServer)))

is

    (if (pending is
            (a Queue (with front (a Request (with message r) (with customer c)))))
        ∧ (DeviceStatus(i WhichIs (or 1 2)) is idle)
     then
        (SendTo devicei
                (a Request (with message r)
                           (with customer
                                  (a TransactionManager
                                         (with device devicei)
                                         (with customer c)))))
        (return (the Server (with pending AllButFront) (with DeviceStatusi printing)))

    (Else (the Server))))
```

The construct

```
(return <expression>)
```

is used to designate the expression whose value is to be returned. This notation is necessary because several expressions can be evaluated concurrently.

Note that a new transaction manager is created to manage each printing request for the printing devices.

Also note that each use of *SendTo* corresponds to the generation of a concurrent activity. So for instance, the first *SendTo* above starts the printing on one device, while the server proceeds in parallel to look for more requests to be accepted.

IX.8 --- Transaction Managers

The transaction manager expects to receive a reply from the printer (indicating completion of the print request). The transaction manager in turn sends the completion report to the server with additional information about the transaction such as the device and customer involved.

```
((a TransactionManager
    (constraint TheServer (a HardCopyServer))
    (constraint device d)
    (constraint customer c))
 is
 (a Behavior
     ((a PrintingCompletedReport) reply (ReplyTo TheServer (a CompletionReport)))))
```

The transaction manager is implemented as a customer with a behavior which when a printing completed report R is received, packages up the report R together with the name of the device d, and the customer c and sends them off in a reply to the server. Note that the transaction manager is itself implemented as a customer that designates c as the sponsor who should pay for the resources used in printing the document.

```
((the TransactionManager
    (constraint TheServer (a HardCopyServer))
    (constraint device d)
    (constraint customer c))
 is
    (the Customer
        (with behavior
            (the Behavior
                ((R WhichIs (a PrintingCompletedReport)) reply
                    (ReplyTo TheServer
                        (a CompletionReport
                            (with device d) (with response R) (with customer c))))))
        (with sponsor c)))
```

X -- Serializer Induction

Serializer induction is an inductive method for proving that a serialized actor S always satisfies a specified property P. The base step of the induction is to show that S satisfies P when it is created. The induction step is to show that if S satisfies P then the replacement of S will satisfy P.

We first show that certain constraints on the behavior of the hard copy server are always met. These invariants will be used in the rest of the proof.

The second part of the proof shows that the server computes a replacement for each message which it receives. This will be a preliminary result for proving that the rest of the constraints for the hard copy server always hold. Finally we prove that the server always replies to the requests which it receives.

X.1 — Checking Constraints of the Behavior

First we verify that the following constraints on the behavior of the hard copy server always hold:

```
((a HardCopyServer) is
  (a Server
     (with pending
           (a Queue (constraint element (a PrintRequisition)))
           ((an EmptyQueue) provided (v (DeviceStatus1 is idle) (DeviceStatus2 is idle))))
     (with DeviceStatus1 (or idle printing))
     (with DeviceStatus2 (or idle printing))
     (with TheServer (a HardCopyServer))))

((a NextServer) is
   ((a NextServer
     (constraint pending (a Queue (constraint element (a Request))))
     (with DeviceStatus1 (or idle printing))
     (with DeviceStatus2 (or idle printing))
     (with TheServer (a HardCopyServer)))))
```

The proof that these constraints *always* hold is by serializer induction.

1. Show that the constraints are met when the hard copy server is created.

2. Assuming that the constraints are true, show that, whatever message is received the resulting replacement server will meet the constraints.

The general result can be established by case analysis for each message received. For instance if the server receives a PrintRequisition **r**, then the request **r** will be added to the rear of pending and NextServer will be invoked with a nonempty queue to designate the new server. There are two cases to be considered:

1: One of the devices is idle. Therefore by the constraint, pending contains only the request r. The request r is removed from the queue of pending requests and the appropriate message is sent to the idle device. The replacement satisfies the constraints because pending is once again empty.

2: Both devices are printing and therefore the conditional in *NextServer* is false, so that the *Else* clause applies. Since pending was not empty, this means that none of the devices was idle. Then the server becomes a server with neither of the devices idle. Therefore the invariant will hold also in this case.

The proof that the constraints always hold is similar for the Completion communications.

Except for the proof of the condition on the emptiness of pending this part of the proof is not very different from the kind of static type checking usually performed by a compiler.

X.2 --- Proof of Guarantee of Service

We can prove that service is guaranteed to all printing requests. If the server receives a request when one of the devices is idle, the request will be immediately passed on, since pending will be empty according to the constraints for the server behavior. If none of the devices is idle, the request will be queued.

The following assertion is proved by induction on n:

> If n requests precede a request **R** in pending, then **R** will be passed to one of the devices after n completion communications have been received by the server.

A completion is either one of the following communications:

(*a* CompletionReport (*with* device $device_1$) (*with* response ...) (*with* customer ...))
(*a* CompletionReport (*with* device $device_2$) (*with* response ...) (*with* customer ...))

The implementation of the server has the property it will always receive a communication back for each of the requests it sends to a device. By the constraints for the server, we know that if **R** is pending, then there is a request outstanding for either $device_1$ or $device_2$, and a completion reply will be received by the server.

The first such communication will be received after a number p of print requests have been received by the server. p is finite because of the law of finite chains in the arrival ordering of actor systems [Hewitt and Baker 1977].

We can show that each of these p print requests will leave unchanged the first n elements in pending and will not alter the status of the devices. Consider then the effect of the next completion received by the server. We show that either the number of requests preceding **R** is decreased by one in the next replacement or the request **R** is sent to one of the printing devices. Clearly one effect of the completion is that one of the devices will become idle. Therefore the next request will be removed from pending and passed to the free device. Therefore if n is 0, the request **R** is served. On the other hand if n is bigger than 0,. then removing the first element from pending reduces by one the number of elements preceding **R** in the queue.

XI -- Methodology

In the following sections we present a more thorough treatment of important methodological issues raised by our treatment of the hard copy guardian presented above.

XI.1 --- Absolute Containment

With serializers it is possible to implement guardians which have a property called absolute **containment** of the protected resource. This concept was proposed by [Hewitt: 1975] and further developed in and in [Atkinson and Hewitt: 1979] (cf. [Hoare: 1976] for a similar idea using the inner construct of SIMULA). The idea is to send a message with directions to the guardian. This one in turn will pass it to the resource so that it can carry out the directions without allowing the user to deal directly with the resource. An important robustness issue arises with the usual strategy of giving the resource out. In fact it is not easy to recover the use of the resource from a situation in which the user process has failed for any reason to complete its operations.

We have found that absolute containment produces more modular implementations than schemes which actually gives out resources protected by guardians. Note that the correct behavior of a guardian which implements absolute containment depends only on the behavior of the resource and the code for the serializer which implements the guardian, but not on the programs which call the guardian.

Our hard copy server implements absolute containment by never allowing others to have direct access to its devices. Thus there is no way for others to depend on the number of physical devices available. Furthermore there is no problem retrieving the devices from users who have seized them since they are never given out.

From a broader philosophical view, absolute containment is a reflection of the logical development of the message passing theory of modularity which was initiated by the lambda calculus and Simula.

XL2 --- Evolution

An important consideration in the design of a guardian is the likely direction in which it will need to evolve to meet future needs. For example the users may decide that smaller documents should be given faster service than larger documents. A simple scheme for accomplish this is to assign floating priorities to the documents based on their length. The idea is to assign an initial priority equal to the length of the document. When a printer is free, the document with highest priority (i.e. with the smallest priority number) is served next. If a print requisition for a document D_1 of length n_1 is received when there is a document D_2 at the rear of pending with priority n_2 which is greater than n_1, then D_1 is placed in front of D_2. In addition the priority of D_2 is changed to n_2-n_1. The above property of floating priority queues can quite easily be specified as follows:

```
((an Enqueuing
    (with NewEntry (an Entry (with item D1) (with priority n1)))
    (with OldQueue
      (a FloatingQueue
          (with Rear (an Entry (with item D2) (with priority n2)))
          (with AllButRear q))))

 is

 (if (n2 > n1)
   then (a FloatingQueue
              (with Rear (an Entry (with item D2) (with priority (n2 - n1))))
              (with AllButRear
                (an Enqueuing
                     (with NewEntry (an Entry (with item D1) (with priority n1)))
                     (with OldQueue q))))
   else
     (a FloatingQueue
        (with rear (an Entry (with item D1) (with priority n1)))
        (with AllButRear
              (a FloatingQueue
                   (with Rear (an Entry (with item D2) (with priority n2)))
                   (with AllButRear q))))))
```

Simply replacing the queues in the original implementation of the hard copy guardian with floating priority queues will accomplish the desired change. As illustrated by the above example, we have found that it is quite easy evolve guardians to meet new requirements. It is also quite easy to evolve the proof given above to show that the new hard copy server still provides a guarantee of service. The ease with guardians can evolve to meet new requirements is testimony to the power of the concept.

XI.3 --- Guarantee of Service

In our applications we want to be able to implement guardians which guarantee that a response will be sent for each request received. This requirement for a strong guarantee of service is the concurrent analogue to the usual requirement in sequential programming that subroutines must return values for all legitimate arguments. In our applications it would be *incorrect* to have implementations which did not guarantee to respond to messages received.

Serializers have the important advantage that it is possible to guarantee absence of deadlock in actor systems by simply assuring that each *individual* actor will specify a replacement for itself for each message that it processes. In many cases (such as the programs in this paper) it is quite easy to make this check.

Proving a guarantee of service (i.e. every request received will generate a response) is not quite so trivial. Note that it is impossible to prove the property of guarantee of service in some computational models such as Petri nets, CCS, and CSP in which processes communicate via synchronized communication. We consider the ease with which we can prove guarantee of service to be one of the principle advantages of using the actor model of computation.

We recognize that our conclusions concerning the issue of guarantee of service are at variance with the beliefs of some of our colleagues. These disagreements appear to be fundamental and have their genesis in the inception of the field in the early 1970's. The disagreements can be traced to different hypotheses and assumptions on conceptual, physical, and semantic levels.

> **Conceptual Level.** As we mentioned earlier, one of the innovations of the actor model is to take the arrival ordering of communications as being fundamental to the notion of concurrency. In this respect it differs from systems such as Petri Nets and CSP which model concurrency in terms of nondeterministic choice (such as might be obtained by repeatedly flipping a coin). Modeling concurrency using nondeterministic choice implies that all systems must have bounded nondeterminism. However a system such as our hard copy guardian which guarantees service for requests received, can be used to implement a system with unbounded nondeterminism. For this reason, guarantee of service was rejected for inclusion in CSP producing a fundamental difference with actor systems.

Physical level. A careful analysis of the physical and engineering realities leads to the conclusion that guarantee of service can be reliably implemented in practice. Worries about the possibility of implementing guarantee of service have caused others to shrink from providing the ability to guarantee service.

Semantic level. The axiomatic and power domain characterizations of actor systems are closely related and represent a unification of operational and denotational semantics. It is important to note that the axioms which characterize actor computations that are physically are of an entirely different kind from the ones which have been developed by von Neumann, Floyd, Hoare, Dijkstra etc. to characterize classical programming languages. The power domain semantics for actor computations developed by Clinger is grounded on the underlying physical realities of communication based on the use of a mail system. The physical grounding of the ordering using in Clinger's model causes it to differ with others such as Egli-Milner ordering which inherently preclude the possibility of modeling guarantee of service.

XII -- Concurrency

Concurrency is the default in Actl. Indeed maximizing concurrency, minimizing response time, and the avoidance of bottlenecks are perhaps the most fundamental engineering principles in the construction of actor systems. The only limitation on the concurrency of a serialized actor is the speed with which the replacement can be computed for a message received.

Concurrency occurs among all the following activities:

> Within the activities of processing a single message for a given serialized actor. The serialized actor which receives a message can concurrently create new actors, send messages, and designate its replacement (cf. [Ward and Halstead 1980] for the application of this idea in a more limited context).
>
> Between the activities of processing a message for a serialized actor and a successor message received by the same serialized actor. The ability to pipeline the processing of successive messages is particularly important for a serialized actor which does not change state as a result of the message which it has received and thus can easily designate its successor. Another important case occurs where the computation for constructing the replacement can occur concurrently with the replacement processing the next message using "eager evaluation" [Baker and Hewitt 1978]. For example a checking account can overlap the work of constructing a report of all the checks paid out to the Electric Company during the previous year with making another deposit for the current year.

Of course there is no limitation whatsoever on the concurrency that is possible between the activities of two different serialized actors. For example two separate checking accounts can be processing withdrawals at exactly the same time.

Unlike communicating sequential processes, the commands in a serializer do not have to be executed sequentially. They can be executed in any order or in parallel. This difference stems from the different ways in which parallelism is developed in the actor model and communicating sequential processes. In the latter parallelism comes from the combination of sequential processes which are the fundamental units of execution. In the actor model concurrent events are the fundamental units and sequential execution is a derived notion.

XIII -- Summary

Act1 has a number of important advantages over previous systems for the implementation of concurrent systems. In this paper we have demonstrated how it facilitates important aspects of the design, implementation, documentation, proof of specified properties, and evolution of concurrent systems.

XIV -- Acknowledgements

Our colleagues at the MIT Artificial Intelligence Laboratory and Laboratory for Computer Science provided intellectual atmosphere, facilities, and constructive criticism which greatly facilitated our work. Major support for both laboratories is provided by the Advanced Research Projects Agency of the DoD. Additional support for this research was provided by the Office of Naval Research. We would like to thank Bill Carlson, Mike Dertouzos, Bob Engelmore, Bob Grafton, Bob Kahn, and Pat Winston for their encouragement and support.

Henry Lieberman has implemented a preliminary version of **Act1** on a DEC PDP-10 on the M.I.T. ITS system. We are working to develop a successor implementation on the MIT CADR Distributed system building on the foundation which Jeff Schiller has developed software for load balancing and migration of actors making use of the Chaos packet switched network (which is similar to the Ethernet). Ultimately we will develop an implementation of Act1 on a follow-on system called the APIARY [Hewitt: 1980, Schiller: 1980] which is currently under development. Phyllis Koton [Koton and Hewitt 1980] has participated in the development of the Portal communications chip to facilitate multiway communication on the Apiary. Gerald Barber and Maria Simi helped to formalize the property of guarantee of service which is fundamental to actor systems.

During the spring of 1978, the first author participated in a series of meetings with the Laboratory of Computer Science Distributed Systems Group. These meetings were quite productive and strongly influenced both this paper and the Progress Report of the Distributed Systems Group [Svobodova, Liskov, and Clark: 1979].

Conversations with Jean-Raymond Abrial, Ole-Johan Dahl, Jack Dennis, Edsger Dijkstra, David Fisher, Dan Friedman, Stein Gjessing, Tony Hoare, Jean Ichbiah, Gilles Kahn, Dave MacQueen, Robin Milner, Birger Moller-Pedersen, Kristen Nygaard, Jerry Schwarz, Steve Schuman, Bob Tennent, and David Wise. The first author would like to thank Luigia

Aiello and Gianfranco Prini and the participants in the summer school on Foundations of Artificial Intelligence and Computer Science in Pisa for helpful comments.

Robin Stanton has provided constructive criticism and great encouragement. Peter Deutsch made valuable suggestions on how to organize the early sections of this paper. Maria Simi, Phyllis Koton, Valdis Berzins, Alan Borning, Richard Fikes, Gary Nutt, Susan Owicki, Dan Shapiro, Richard Stallman, Larry Tesler, Deepak Kapur and the members of the Message Passing Systems Seminar have given us valuable feedback and suggestions on this paper. Vera Ketelboeter has independently developed a notion of "responsible agents" that is very close to the transaction managers described in this paper. Jerry Barber and Maria Simi have developed methods for proving that actor systems implemented with internal concurrency will respond properly to the messages which they receive.

Although we have criticized certain aspects of monitors and Communicating Sequential Processes in this paper, both proposals represent extremely important advances in the state of the art of developing more modular concurrent systems and both have deeply influenced our work.

The serializer construct used in this paper is a further development of the construct in [Atkinson and Hewitt: 1979]. It has been simplified by removing most of the built-in machinery of the previous version. However the more basic capabilities of the new construct allow us to efficiently implement the facilities (such as queues) that were provided by previous serializers as well as to implement new facilities that were not provided before. Additional flexibility comes form the fact that primitive serializers can explicitly deal with the customer of a communication.

XV -- Bibliography

Atkinson, R. and Hewitt, C. "Specification and Proof Techniques for Serializers" IEEE Transactions on Software Engineering SE-5. No. 1. January 1979. pp 10-23.

Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs" Communications of the ACM, Volume 21, Number 8, August 1978, pages 613-641.

Baker, H. "Actor Systems for Real-Time Computation" MIT Laboratory for Computer Science Technical Report 197. March 1978.

Birtwistle, G. M.; Dahl, O.; Myhrhaug, B.; and Nygaard, K. "SIMULA Begin" Auerbach. 1973.

Brinch Hansen, P. "The Programming Language Concurrent Pascal" IEEE Transactions on Software Engineering SE-1 pp 199-207. 1975.

Brinch Hansen, P. "Distributed Processes: A Concurrent Programming Concept" CACM. Vol 21. No 11. November 1978. pp 934-940.

Borning, A. H. "THINGLAB -- A Constraint-Oriented Simulation Laboratory", Stanford PhD thesis, March 1979. Revised version to appear as Xerox PARC SSL-79-3.

Brinch Hansen, P. "The Programming Language Concurrent Pascal" IEEE Transactions on Software Engineering. June, 1975. pp 199-207.

Clinger, W. "Foundations of Actor Semantics" MIT PhD. Thesis. 1981. forthcoming.

Dennis, J.B. "The APPL Language and its Interpreter", MIT DSG Memo, August 1980.

Dijkstra, E. W. "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs" CACM. Vol. 18. No. 8. August 1975. pp 453-457.

Greif, I. "Semantics of Communicating Parallel Processes" Project MAC Technical Report 154. September 1975.

Gjessing, S. "Compile Time Preparations for Run Time Scheduling in Monitors" Research Report No. 17, Institute of Informatics, University of Oslo, June 1977.

Hewitt, C. E. "Protection and Synchronization in Actor Systems" ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication, March 1975.

Hewitt, C. E. "Viewing Control Structures as Patterns of Passing Messages" Journal of Artificial Intelligence. 8-3, 323-364, June 1977.

Hewitt, C., Attardi, G. and Lieberman, H. "Specifying and Proving Properties of Guardians for Distributed Systems" in Semantics of Concurrent Computations (Ed. G. Kahn), *Lecture Notes in Computer Science* No. 70, Springer-Verlag, Berlin, 1979.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Security and Modularity in Message Passing" MIT AI Lab Working Paper 180, also in First International Conference on Distributed Computing Systems, Huntsville, Alabama, October 1979.

Hewitt, C. and Baker, H. "Laws for Communicating Parallel Processes" MIT Artificial Intelligence Working Paper 134, December 1976. Invited paper at IFIP-77.

Hewitt, C. E. "The Apiary Network Architecture for Knowledgeable Systems" Proceedings of Lisp Conference Stanford. August 1980. pp 107-118.

Hewitt, C. E., Attardi, G., and Simi, M. "Knowledge Embedding in the Description System Omega" Proceedings of AAAI Conference Stanford. August 1980. pp 157-164.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" CACM. October 1974.

Hoare, C. A. R. "Language Hierarchies and Interfaces" *Lecture Notes in Computer Science* No. 46, Springer-Verlag, 1976. pp 242-265.

Hoare, C.A.R. "Communicating Sequential Processes" CACM, Vol 21, No. 8. August 1978. pp. 666-677.

Ichbiah, J. et al. "Reference Manual of the ADA Programming Language", SigPlan Notices 16, 6, 1980.

Ingalls, D. H. H. "The SmallTalk-76 Programming System Design and Implementation" *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, January 1978, pp. 9-16.

Koton, P. and Hewitt, C. "Portal: A Multiway Communicator" MIT AI Lab Working Paper. December 1980.

Liskov, B. "Primitives for Distributed Computing", *Proceedings of the Seventh Symposium on Operating Systems Principles*", Pacific Grove, California, 1979.

Kerns, B. "Towards a Better Definition of Transactions" M.I.T. AI Memo. 1979.

Kristensen, B. B.; Madsen, O. L.; Moller-Pedersen, B.; and Nygaard, K. "A Definition of the BETA Language" TECHNICAL REPORT TR-8. Aarhus University. February 1979.

Manna, Z. and McCarthy, J. "Properties of Programs and Partial Function Logic" Machine Intelligence 5 B. Meltzer and D. Michie, editors. Edinburgh Univ. Press. 1970. pp 27-37.

Milner, R. "Flowgraphs and Flow Algebras" JACM. Vol 26, No. 4. 1979.

Owicki, S. "Verifying concurrent Programs With Shared Data Classes" Formal Description of Programming Concepts edited by E. J. Neuhold. North Holland. 1978.

Reynolds, J.C. "On the Relation Between Direct and Continuation Semantics" Proceedings of the Second Colloquium on Automata Language and Programming, Saarbruecken, Springer-Verlag, Berlin 1974.

Schiller, J. "Progress on the Implementation of an Apiary" MIT AI Working Paper. December 1980.

Seitz, C. "System Timing" in Introduction to VLSI Systems" by Mead and Conway. Addison-Wesley. 1980. pp 218-262.

Strachey, C. and Wadsworth, C.P. "Continuations - a Mathematical Semantics for Handling Full Jumps", Technical Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

Svobodova, L.; Liskov, B.; and Clark, D. "Distributed Computer Systems: Structure and Semantics" MIT Laboratory for Computer Science TR-215. March 1979.

Ward, S. and Halstead, R. "A Syntactic Theory of Message Passing" JACM. Vol 27, No. 2, April 1980. pp 365-383.

APPENDIX XVI --- Conditional Constructs of Act1

XVI.1 --- A Concurrent Case Expression

In Act1 the conditional case expressions has the following form:

(*CaseFor* expression
...
(*is* pattern_for_value$_i$ *then* body$_i$)
...
(*complaint* pattern_for_complaint$_i$ *then* body$_i$)
...
(*NoneOfAbove* alternative_body))

In order to evaluate this construct, expression is evaluated first. If the evaluation produces a value V which matches any of the patterns then the corresponding body is executed. If the value V matches more than one of the patterns then an arbitrary one of the corresponding body$_i$ is selected to be executed. If the evaluation produces a complaint (exception) instead of a value and the complaint matches one of the patterns for complaints, then one of the corresponding bodies is selected for execution.

The rule of concurrent selection of which body to execute in the case where a value matches more than one pattern has the advantage that it makes each body more modular since it depends only on its pattern, making it easy to add more selections later. Thus the rule of *concurrent* consideration of cases encourages the construction of programs which are more modifiable. The programs are also more robust since the addition of new cases is less likely to introduce bugs in already existing cases.

The concurrent case statement facilitates efficient implementation by allowing concurrent matching of expression against the patterns. This ability is important in applications where a large amount of time is required to determine whether or not conditions hold. In this case the implementation could execute the body of the clause whose condition is first determined to hold meanwhile abandoning effort on the other clauses. Thus the rule of concurrent consideration of cases enables some programs to be implemented more efficiently.

If the value V is determined to match none of the patterns then alternative_body is executed. This rule provides the ability to have the patterns represent special cases leaving the alternative_body to deal with the general case if none of the special cases apply.

Allowing pattern matching in the case clauses is a useful feature that is derived from the pattern directed programming languages PLANNER, QA-4, POPLER, CONNIVER, etc.