MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. MEMO 781 September 1984

Linguistic Support of Receptionists for Shared Resources*

Carl Hewitt
Tom Reinhardt
Gul Agha
Giuseppe Attardi
The Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

September 15, 1984

Abstract

This paper addresses linguistic issues that arise in providing support for shared resources in large scale concurrent systems. Our work is based on the Actor Model of computation which unifies the lambda calculus, the sequential stored-program and the object-oriented models of computation. We show how *receptionists* can be used to regulate the use of shared resources by scheduling their access and providing protection against unauthorized or accidental access. A shared financial account is an example of the kind of resource that needs a receptionist. Issues involved in the implementation of scheduling policies for shared resources are also addressed. The modularity problems involved in implementing servers which multiplex the use of physical devices illustrate how delegation aids in the implementation of parallel problem solving systems for communities of actors.

*The 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 the System Development Foundation and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505.

1 Background

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: A1 might precede A2, A2 might precede A1, or they might overlap in time. Concurrency can arise from a variety of sources including the multiplexing of individual processors as well as the interaction of multiple processors. Thus concurrent systems include time sharing systems, multiprocessor systems, distributed systems, and integrated circuits. *Parallelism* is the exploitation of concurrency to cause activities to overlap in time.

For thirty years, the lambda calculus and the sequential stored program models have coexisted as important bases for software engineering. Systems based on the lambda calculus [McCarthy 62, Landin 65, Kahn 81, Friedman and Wise 76, Hewitt and Baker 77 and Backus 78] provide a sound basis for constructing independent immutable objects (functions and functional data structures) that have inherent concurrency which is constrained only by the speed of communications between processing elements. Such 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 shared checking accounts which must change their behavior during the course of their lifetimes. Moreover, checking-accounts are good examples of *objects created for open-ended, evolving systems* in that their behavior must be specified independently of the number of teller machines.

While it has been shown that by treating input and output to functions as infinite, continuous streams, functional languages can create objects capable of changing their local states, it is also known that this is only adequate for "closed systems." In an *Open System* (That is, a system which is open ended and continuously evolving, see [Hewitt 83]), the inter-stream ordering is indeterminate and subject to interactive external influences: the Brock-Ackerman anomaly [Brock-Ackerman 81] is an example of a system where one is unable to abstract the arrival order information from the purely functional behavior of a concurrent system (Agha, private communication 1984).

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 as only one access to the memory occurs at a time. The variable assignment command (e.g., SETQ in Lisp or the := command in the Algol-like languages) incorporates this ability in higher level languages with the attendant cost that they become inherently sequential.

In the early seventies, an important step was made toward unifying of the two approaches by developing the concept of *objects*. An object consists of a local state and procedures to operate on it. SIMULA 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.

In Lisp, objects are embodied by closures, sometimes called "funargs". A closure is a function plus an environment. The environment keeps the values associated with variables used within the function and represents the local state of the closure. When the closure is invoked, its function is applied in that environment. Closures can modify their states by means of an assignment command.

In the case of truly concurrent systems, however, the assignment command is not suitable as the basis for change of behavior because it doesn't address 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 74] proposed an adaptation of the Simula class construct called a *monitor* as an operating systems construct for higher level languages. By imposing the constraint that only one invocation can be active at a time, monitors provide a means for achieving synchronization and mutual exclusion. Monitors retain 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

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 of it 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 means of interaction among parts of an operating system. These components are all at the same level, and each bears some responsibility for the correct behavior of the system as a whole. 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 consistency of its state when a process leaves the monitor, or the guarantee that each process will eventually release the monitor, cannot usually be established from the code of the monitor alone.

In the setting of a distributed system, it seems more appropriate that the responsibility for the use of a shared resource be delegated to a specific abstraction for the resource which regulates its use. 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 follow them.

We call the abstractions that have been developed for this purpose *receptionists* [Hewitt, Attardi and Lieberman 79a and Atkinson and Hewitt 79]. Receptionists solve the problem of scheduling access to shared objects by unifying the notions of *opacity* found in lambda-calculus based systems and *mutability* provided by the assignment command in imperative languages.

The purpose of a receptionist is to provide an interface to users for performing operations on a protected resource. The receptionist is responsible for serializing concurrent requests, for scheduling the access to the resource, and for providing protection and ability to recover from failures. Receptionists 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*.

2 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*. Various control structures are hence viewed as "patterns of message passing" [Hewitt 77]. Examples of such communications are electrical signals, parameter passing between subroutines of a program and messages transferred between computers in a geographically distributed network.

Conceptually, one actor communicates with another using a *mail address*. Mail addresses may be the *targets* of communications and may also be sent as messages within communications. Thus a mail address differs fundamentally from a machine address which has read and write as the defined operations. Mail addresses may be physically implemented in a variety of ways 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, $\mathcal{K}$, at an actor, $\mathcal{A}$, as an *event*. Symbolically, an Event, $\mathcal{E}$, may be represented:

$$\mathcal{E} = [\mathcal{A} \Longleftarrow \mathcal{K}]$$

As a result of receiving a communication, an actor may produce other communications to be sent to other actors (trivially, this includes itself). In terms of events, this means that an event may *activate* some other events. Events are hence related by an *activation* ordering [Hewitt and Baker 77], and computation occurs between such events.

While processing a communication, an actor can create new actors and can also designate another actor to take its place to receive the next delivered communication.

In summary, an actor, $\mathcal{A}$, can take the following actions upon receipt of a communication:

- It can make simple decisions.
- It can create new actors.
- It can send communications to actors.
- It can specify a replacement actor which will handle the next communication accepted.

Communications received by each actor are related by the *arrival* ordering, which expresses the order in which communications 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 arrived first. Some form of arbitration is usually necessary to implement 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. Although the actor model incorporates properties that any physically realizable communication system must obey, 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 activating the next.

2.1 The Nature of Actor Communication

For some of the recently proposed models of concurrent systems [Hoare 78, Milner 79], the communication mechanism 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, in that communications may always be sent but are subject to variable delays *en route* to their destinations. Communication via a mail system has important properties that distinguish it from "hard-wired" connections:

- *Asynchrony*: The mail system decouples the sending of a communication from its arrival. It is not necessary for the recipient to rendezvous with the sender of a communication.
- *Buffering*: The mail system buffers communications between the time they are sent and the time they are delivered to the recipient.

We have found the properties of asynchrony and buffering fundamental to the widespread applicability of actor systems; they enable us to disentangle the senders and receivers of communications, thus raising the level of the description.

2.2 Functionality of the Mail System

The implementation of an actor system entails the use of a mail system to effect communication. The mail system transports and delivers communications by invoking hardware modules, activating actors defined by software, or sending communications through the network as appropriate. The mail system provides the following functionality:

- *Routing*. The mail system routes a communication to the recipient over whatever route seems most appropriate. For example, it may be necessary to route the communication 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 communications 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.

2.3 Sending Mail Addresses in Communications

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

- *Public Access*. The receiver of a communication does not have to anticipate its arrival, in contrast to systems which require that a recipient know the name of the sender before any communication can be received, or, more generally, 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 the mail address of a newly created actor can be sent to pre-existing actors.

In some models, e.g., those which only allow messages composed of elementary data types such as integers, reals and character strings, the mobility of processes is limited. Processes or other non-primitive objects cannot be transmitted: a limitation which is closely related to an important restriction on the reconfiguration of the system. Reconfiguration is not possible in such systems which require that a process be created knowing exactly the processes with which it will be able to communicate throughout its entire existence.

New actors can be dynamically created as a result of an actor receiving a communication. 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.

2.4 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 [Grief 75], axiomatically [Hewitt and Baker 77], operationally [Baker 78], and in terms of power domains [Clinger 81].

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). 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 within the mathematical model.

3 The Actor Language Act3

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)

A number of languages have been recently developed for concurrent systems. These languages differ in their conception of communication, in what can be communicated, and in their ability to dynamically create new computational agents. Design decisions in Act3 have been determined by the need to provide support for parallel problem solving in our applications. Our applications require that we be able to efficiently create, garbage collect and migrate large numbers of actors from place to place in the network. This circumstance causes it to differ with other languages and systems.

Act3 is an experimental language based on the actor model of computation. Act3 is universal in the sense that any physically realizable actor system can be implemented in it.

3.1 Notation

Act3 generalizes Lisp's syntactic notation for expressions (i.e. each expression is enclosed in parentheses with the elements of the expression separated by white space) by allowing infix operations to be defined. This 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 Act3 and Lisp. Most of the new semantic notions in Act3 (such as receptionists, delegation, proxies, etc.) are not present in Lisp. Act3 was designed to implement concurrent systems, whereas Lisp was designed and has evolved to implement sequential procedures on a sequential computer.

3.2 Descriptions

Descriptions serve a role in Act3 that corresponds to the role of data types and structures in more conventional languages. Descriptions are used to express properties, attributes and relations between objects. Our description language includes first order logic as a sublanguage.

Data types in programming languages have come to serve more and more purposes in the course of time. Type checking has become a very important feature of compilers, providing type coercion, helping in optimization and aiding 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. Descriptions help overcome these limitations.

We use descriptions to express assumptions and the constraints on objects manipulated by programs in Act3. These descriptions are an integral part of programs and can be used both as checks when programs are executing and as useful information which can be exploited by other systems which examine programs such as translators, optimizers, indexers, etc.

3.3 Communications and Customers

Communications are actors that embody the units of information that are transmitted from one actor to another. There are different kinds of communications, each with possibly different attributes. Act3 provides mechanisms that 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. The notion of customer generalizes the concept of a *continuation*, introduced in the context of denotational semantics [Strachey and Wadsworth 74 and Reynolds 74] 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 actor analogue 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`. 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.

This notion of customers subsumes and unifies many less well defined concepts such as a "suspended job" or "waiting process" in conventional operating systems. 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 existing systems.

3.4 Communication Primitives

Act3 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. That is, an actor can transmit and receive communications (continue computing) while communications that it has sent are in transit to their destinations.

Transmitting communications using commands provides a very convenient method for spawning more parallelism. The usual method in other languages for creating more parallelism entails creating processes (as in ALGOL-68, PL-1, and Communicating Sequential Processes, etc.). The ability to engender parallelism simply by transmitting more communications is one of the fundamental differences between actors and the languages based on communicating sequential processes.

3.5 Transactions

The notion of an *Event* is now further elaborated to take into account the various communication types:

$$\mathcal{E}_1 = [\mathcal{A} \Longleftarrow (A\ Request\ (With\ Message\ \mathcal{M}\ (With\ Customer\ \mathcal{C})))]$$

describes the reception of a `Request` by some actor, $\mathcal{A}$. Note, that in place of a symbol representing the communication, the communication type and its primary components are specified. As a result of $\mathcal{E}_1$, some computation is performed and, at some later time, a Reply is sent to the customer, $\mathcal{C}$, specified in the Request.

$$\mathcal{E}_n = [\mathcal{C} \Longleftarrow (A\ Reply\ (With\ Message\ \mathcal{M}))]$$

Events $\mathcal{E}_1$ and $\mathcal{E}_n$ are thus causally related and together comprise a *transaction*. For a given computation, transactions may encompass sub-transactions. For instance, sending a Request to an actor, $\mathcal{F}$, that recursively calculates the factorial of the incoming message might result in a series of events:

$$\begin{aligned}
\mathcal{E}_1 &= [\mathcal{F} \Longleftarrow (A\ Request\ (With\ Customer\ \mathcal{C}_1)\ (With\ Message\ 3))] \\
\mathcal{E}_2 &= [\mathcal{F} \Longleftarrow (A\ Request\ (With\ Customer\ \mathcal{C}_2)\ (With\ Message\ 2))] \\
\mathcal{E}_3 &= [\mathcal{F} \Longleftarrow (A\ Request\ (With\ Customer\ \mathcal{C}_3)\ (With\ Message\ 1))] \\
\mathcal{E}_4 &= [\mathcal{F} \Longleftarrow (A\ Request\ (With\ Customer\ \mathcal{C}_4)\ (With\ Message\ 0))] \\
\mathcal{E}_5 &= [\mathcal{C}_4 \Longleftarrow (A\ Reply\ (With\ Message\ 1))] \\
\mathcal{E}_6 &= [\mathcal{C}_3 \Longleftarrow (A\ Reply\ (With\ Message\ 1))] \\
\mathcal{E}_7 &= [\mathcal{C}_2 \Longleftarrow (A\ Reply\ (With\ Message\ 2))] \\
\mathcal{E}_8 &= [\mathcal{C}_1 \Longleftarrow (A\ Reply\ (With\ Message\ 6))]
\end{aligned}$$

The top-level transaction begins with $\mathcal{E}_1$ and ends with $\mathcal{E}_8$, whereas $\mathcal{E}_2$ and $\mathcal{E}_7$, $\mathcal{E}_3$ and $\mathcal{E}_6$, and $\mathcal{E}_4$ and $\mathcal{E}_5$ delimit nested sub-transactions.

3.6 Receptionists

Receptionists are actors that can accept only one communication at a time for processing. Communications that arrive while the receptionist is receiving another communication are service in the order in which they arrive.

Accepting communications in the order in which they arrive does not involve any loss of generality since a 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 receptionist when it accepts the request. The receptionist can remember the request for future action, and, in the meantime, accept communications 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 as any other actors allows receptionists to organize the storage of requests in progress in a variety of ways. Unlike monitors, receptionists 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, 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 communications.

4 Implementation of a Hard Copy Server

Implementing a module to service printing requests for two printing devices provides a concrete example to illustrate the flexibility of receptionists. The following example illustrates the implementation of a receptionist that protects more than one resource (in this case two printers).

```
(Define (New HardCopyServer
                          (With device-1 ≡d-1)
                          (With device-2 ≡d-2))
    (New HardCopyReceptionist
                     (With device-1 d-1)
                     (With device-2 d-2)
                     (With pending (A (New waiting-queue)))
                     (With device-status-1 idle)
                     (With device-status-2 idle)))
```

We have modularized the implementation of the receptionist for a hard copy server into two kinds of modules:

1. *Receptionists* which provide the external interfaces to the outside world.
2. *Proxies* which deal with the issues of delegated communications for a shared resource.

The hard copy server is provided with the mail addresses of two devices (which are printers) when it is created. The function of the receptionist is to set up and initialize a hard copy server. The receptionist has to maintain a fundamental constraint that no device should be idle if there are pending requests. 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 schedule the printers.

We can bind an identifier to a new instance of an actor using the `Define` construct.

```
(Define Server-64 (New HardCopyServer
                     (With device-1 Dover)
                     (With device-2 LGP)))
```

4.1 Receptionist Implementation

A first implementation of the Receptionist provides that print requests submitted to the receptionist will be served in the order in which they are received. In general, though, requests cannot be served immediately for no printer may be available at that time. Hence, a FIFO queue will be used as appropriate scheduling structure for pending requests. Later, we will evolve this implementation into one which uses a more sophisticated scheduling structure.

Prior to code generation, a *template* program is constructed (see code below). Such a template will serve as guide to the actual implementation.

```
(Define (New HardCopyReceptionist <attributions>)
  (Is-Request <pattern> Do
    ;;the incoming-print-request Joins the
    ;;pending queue.
    (Join p ;;the current transaction joins p
      (Then Do
        (Release-from p IF <some-device-is-idle>))
        (After-Released Do ;;the current transaction has been released from p
          <record-starting> ;device is now busy
          ;;After delegating the print request to a proxy
          (After <delegate-request>
            <record-stopping> ;record its completion
            <reply-to-customer>
            ;;the next pending print-request is
            ;;Released-From the pending queue
            ;;If a device is free
            (Release-From p If <some-device-is-idle>)))))))
```

Operationally, the behavior of the HardCopyReceptionist is as follows:

1. A print request arrives and its transaction immediately joins the pending queue.

2. If device is free, the receptionist releases a transaction from the pending queue.

3. The HardCopyReceptionist updates its state to reflect the fact that one of printers is busy concurrently with delegating the print message to the chosen device.

4. Eventually, a reply is received from the printer causing the receptionist to update its local state to reflect that the device is no longer busy.

5. The reply is forwarded to the customer of the transaction concurrently with releasing another transaction from the pending queue if some device is free.

It is important to understand that the `After` construct frees up the receptionist for other processing while the printers are operating in parallel.

4.2 Actual Implementation

The actual code for the `HardCopyReceptionist` is presented below. Readers not interested in studying it may safely skip the rest of this section.

Act3 is homogeneous; all data are Actors. A program in Act3 is a collection of actors that, taken together, specify a behavior for the top-level actor —in this case, `HardCopyReceptionist`. Specifically, the Act3 language is comprised of *commands*, *expressions* and *communication-handlers*.

The terms *command* and *expression* are used in the ordinary semantic sense: Commands are actors whose execution results in some effect, whereas evaluating an expression results in the production of more actors.

Note that Act3 maintains the distinction between *describing* an actor and *creating* it. Basically, expressions of the form:

```
(A <Concept> (With <Att-Relation> <Att-Filler>))
```

are called *Instance-Descriptions*. Evaluating an instance-description results in the creation of an `<A-Expression>` actor whose `<Att-Relation>`s and `<Att-Filler>`s are determined by the encompassing environment. `<New-Expressions>` have exactly the same syntax, except that the first token must be the symbol, `New`. Evaluating a `<New-Expression>`, however, results in the creation of an instance of the specified actor. *Communication-handlers* are expressions of the form:

```
(Is-Request <Pattern> Do <Commands>)
```

Note, they contain a `<Pattern>` which is an expression and a series of one or more `<Commands>`. In general, commands are preceded by the keywords, `Do` or `In`. Besides `Is-Request`, `Is-Reply` and `Is-Complaint`, an `Is-Communication` handler type is provided for complete generality.

Forms typed in at the top level are "asked to parse themselves", hence, "actor programs" are intrinsically distributed, and, depending upon context, many Act3 constructs can appear as both commands and expressions.

```
(Define (New HardCopyReceptionist
             (With device-1 ≡d-1)
             (With device-2 ≡d-2)
             (With pending ≡p)
             (With device-status-1 ≡ds-1)
             (With device-status-2 ≡ds-2))
 (Is-Request (≡pm Which-Is (A PrintRequestMessage)) Do
   (Join p
    ;;suspend this transaction to wait in queue,p
    (Then Do
          ;;if some device is idle, then resume the next
          ;;request to be processed
         (Release-From p
                  If (∨ (ds-1 Is idle) (ds-2 Is idle))))
    (After-Released Do
     ;;when this transaction is resumed
      (Let (((An IdleDeviceFound
                 (With device-status ≡chosen-device-status)
                 (With device-chosen ≡chosen-device))
             Match
             (Call ChooseIdleDevice
                   (With device-status-1 ds-1)
                   (With device-status-2 ds-2)))
       ;;Call ChooseIdleDevice to bind the Chosen Device
       ;;and the device-status.
       In
       (Become (New HardCopyReceptionist
                    (With chosen-device-status busy)))
       (After (Ask chosen-device pm)
        Cases
        (Is (≡r Which-Is (A PrinterReplyMessage)) Do
         (Reply (Call MakePrintingCompletedReport
                    (With printer-reply r)))
         (Become (New HardCopyReceptionist
                      (With chosen-device-status idle)))
          ;;if some requests are pending
          ;;then resume the next one to continue processing
         (Release-From p
                  If (∨ (ds-1 Is idle) (ds-2 Is idle))))))))))
```

4.3 An Operational HardCopyReceptionist

Act3 is inherently parallel; concurrency is the default. Communication handlers, commands and sub-expressions are evaluated in parallel.

A key part of the above code is the `After` command which we have duplicated below:

```
(After (Ask chosen-device pm)
    Cases
      (Is (≡r Which-Is (A printer-replyMessage)) Do
           (Reply (Call MakePrintingCompletedReport
                      (With printer-reply r)))
           (If (non-empty p)
              ;;if some requests are pending,
              (Then Do
                     ;;then resume the next one to continue processing
                     (Release-From p)))))
```

The `After` command works as follows:

1. The `Become` command executes concurrently with the `After` command making a new `HardCopyReceptionist` with `device-status-i` set to busy. Hence, communications arriving at the `HardCopyReceptionist` during the interim will have the information that it is not available.
2. The `HardCopyReceptionist` becomes sensitive to incoming communications while `(Ask chosen-device pm)` is taking place invoking device, `chosen-device`. In effect, then, the implementation of the `Hard-CopyReceptionist` concurrently processes printing requests and guarantees service.
3. After the reply from `(Ask chosen-device pm)` has been accepted, the bindings of `d-1`, `d-2`, `p`, `ds-1` and `ds-2`, are *bound* to the values they have *when the reply is accepted* which may be different from what they were when `chosen-device` was invoked. The reply `r` is processed by the `Cases` handler in the `After` command.
4. The processing consists of replying with a printed completed report and then checking to see if there are any pending printer requests.

5 Methodology

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

5.1 Absolute Containment

With receptionists it is possible to implement computational abstractions which have a property called *absolute containment* of the protected resource. This concept was proposed by [Hewitt 75] and further developed in [Atkinson and Hewitt 79 (cf. [Hoare 76], for a similar idea using the `inner` construct of SIMULA). The idea is to send a communication with directions to the receptionist. This, 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 out the resource. 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 allocate resources protected by receptionists. Note that the correct behavior of a receptionist which implements absolute containment depends only on the behavior of the resource and the code for the receptionist which implements the receptionist, not on the programs which call it.

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, no problem arises in retrieving the devices from users who have seized them since they are never given out.

5.2 Evolution

An important consideration in the design of a receptionist is the likely direction in which it will need to evolve to meet future needs. For example, 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 $n_2 - n_1$.

Simply replacing the queues in the original implementation of the hard copy receptionist with floating priority queues will accomplish the desired change.

5.3 Guarantee of Service

In our applications we want to be able to implement receptionists which guarantee that a response will be sent for each request received. This requirement for a strong guarantee of service is the concurrent system's 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 a response to communications received.

If one can prove that each *individual* serialized actor in an actor system will specify a replacement for itself for each communication that it processes, then that actor system is guaranteed to be deadlock free. In general, the proof that a receptionist always designates a replacement might depend on assumptions on the behavior of other actors. In our example, the property of the server was achieved by relying on the well defined behavior of each printer. In cases where such dependencies constitute a partial ordering, such a proof can be performed without difficulties. If there are loops in the dependencies, then a more complex analysis is necessary.

Proving a guarantee of service (i.e., every request received will generate a response) is not trivial. Note that it is impossible to prove the property of guarantee of service in some computational models, such as Petri nets 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 principal advantages of using the actor model of computation in our applications.

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 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. Modeling concurrency using nondeterministic choice implies that all systems must have bounded nondeterminism. However, a system, such as our hard copy server which guarantees service for requests received, can be used to implement systems with unbounded nondeterminism. Actor systems impose a constraint on all implementations that all mail sent must be delivered to the target actor. Whether the target actor ever accepts the communication and acts on it is a separate matter which is not of concern to the mail system.

- *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 constructing theories in which the ability to guarantee service can be proved (e.g., [Dijkstra 77], pg. 77).

- *Semantic Level.* The axiomatic and power domain characterizations of actor systems are closely related and represent a unification of operational and denotational semantics. The axioms which characterize actor computations that are physically realizable are entirely different from those 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. It provides a model theory to support proofs in which properties such as guarantee of service can be proved.

One criticism of guarantee of service is that it does not give any indication of when the service will be performed. Of course, this theoretical problem has been with us for a long time since it occurs even for sequential programs. In the case of concurrent systems, we can do somewhat better by transmitting progress reports as the computation proceeds as well as estimates when the request will be accomplished. For example, we can modify the communication handler of our receptionist for the hard copy server so that it produces a report of the number of print requests queued before the one which has been submitted using the `Report-Status` command (see example below).

```
(Define (New HardCopyReceptionist
                    (With device-1 ≡d-1)
                    (With device-2 ≡d-2)
                    (With pending ≡p)
                    (With device-status-1 ≡ds-1)
                    (With device-status-2 ≡ds-2))
    (Is-Request (≡pm Which-Is (A print-request-message)) Do
      (Report-Status
           (A Report
              (With no-of-previous-requests (length p))))
           (Join p (With priority (length (document pm)))
           ...)))
```

The idea for incorporating this modification in our example comes from a suggestion of Hoare [private communication 1981] and is, in fact, similar to the way that the current hard copy server for our laser printer at MIT works. Using such ideas, we can incorporate stronger performance criteria into our mathematical semantics.

6 Concurrency

Concurrency is the default in Act3. 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 communication received.

Concurrency occurs among all the following activities:

- Within the activities of processing a single communication for a given serialized actor. The serialized actor which receives a communication can concurrently create new actors, send communications, and designate its replacement (cf. [Ward and Halstead 80] for the application of this idea in a more limited context).

- Between the activities of processing a communication for a serialized actor and a successor communication received by the same serialized actor. The ability to pipeline the processing of successive communications is particularly important for a serialized actor which does not change state as a result of the communication 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 communication using "eager evaluation" [Baker and Hewitt 77]. 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, no limitation whatsoever exists 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 receptionist 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 since special measures must be taken to force actions to be sequential.

7 Related Work

The work on actors has co-evolved with a great deal of work done elsewhere. Important differences have emerged in part because of different motivations and intended areas of application. The driving force behind the our work has been the needs of parallel problem solving systems for communities of actors [Kornfeld and Hewitt 81, Barber, DeJong and Hewitt 83].

The work on Simula and its successors SmallTalk, CLU, Alphard, etc., has profoundly influenced our work. We are particularly grateful to Alan Kay and the other members of the Learning Research group for interactions and useful suggestions over the last few years.

One of our achievements is to unify procedure and data objects into the single notion of an actor. The Simula-like languages provide effective support for coroutines but not for concurrency. Through its receptionist mechanism, Act3 provides effective support for shared objects in a highly parallel distributed environment.

In the area of semantics, parallels can be found with the recent work of Hoare, Milner, and Kahn and MacQueen. From the outset, an important difference between the Actor Model and the process models of Hoare and

Milner has been that process models have not allowed direct references to processes to be stored in data structures or communicated in messages. The Actor Model differs in that all objects are actors and all computation takes place via communication.

A fundamental part of the motivation of the work reported in this paper is to provide linguistic support for receptionists in open systems [Hewitt 83]. One of the fundamental properties of open systems is that they do not have well defined global states. In this respect our work differs from previous work on concurrent access to data bases using the notion of *serializability*.

The only thing that is visible to an actor is the communication which it has just accepted. Other communications which might be on the way will not have been noticed yet. We have deliberately made nothing else visible so that a variety of scheduling procedures can be implemented such as pipes, queues, multiple queues, priority queues, floating priority queues, etc.

8 Future Work

Dealing with the issues raised by the possibility of an actor being a specialization of more than one description has become known as the "Multiple Inheritance Problem". A number of approaches have been developed in the last few years including the following: [Weinreb and Moon 81, Curry, Baer, Lipkie and Lee 82, Borning and Ingalls 82, Bobrow and Stefik 82 and Borgida, Mylopoulos and Wong 82].

Our approach differs in that it builds on the theory of an underlying description system [Attardi and Simi 81] and in the fact that it is designed for a parallel message passing environment in contrast to the sequential coroutine object-oriented programming languages derived from Simula. In this paper we have shown how *receptionists* can be used to regulate the use of shared resources by scheduling their access and providing protection against unauthorized or accidental access. The work in this paper needs to be combined with work on relating descriptions and actions in concurrent systems [Hewitt and DeJong 83] to provide a general framework for addressing problems of multiple inheritance.

9 Acknowledgments

Much of the work underlying our ideas was conducted by members of the Message Passing Semantics group at MIT. We especially would like to thank Jon Amsterdam, Jerry Barber, Peter de Jong, Henry Lieberman, Tim McNerney, Elijah Millgram, Chunka Mui, and Dan Theriault. We are indebted to Henry Lieberman for his insistence that the tough problems associated with delegation and proxies can and must be worked through to a satisfactory resolution and for his earlier work on the problem [Hewitt, Attardi and Lieberman 79b].

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.

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 have proved to be of great value in shaping our concepts. The first author would like to thank Luigia Aio 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, Larry Tesler, Deepak Kapur and the members of the Message Passing Systems Seminar have given us valuable feedback and suggestions on this paper.

The receptionist construct used in this paper is a further development of the construct in [Atkinson and Hewitt 79]. 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 receptionists as well as to implement new facilities that were not provided before. Additional flexibility comes form the fact that receptionists can explicitly deal with the *customer* of a communication.

[Atkinson and Hewitt 79] Atkinson, R. and Hewitt, C. Specification and Proof Techniques for Serializers. IEEE Transactions on Software Engineering SE-5 No. 1, IEEE, January, 1979.

[Attardi and Simi 81] Attardi, G. and Simi, M. Semantics of Inheritance and Attributions in the Description System Omega. Proceedings of IJCAI 81, IJCAI, Vancouvèr, B. C., Canada, August, 1981.

[Backus 78] Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM 21*, 8 (August 1978), 613-641.

[Baker 78] Baker, H. Actor Systems for Real-Time Computation. Technical Report 197, Mit Laboratory for Computer Science, 1978.

[Baker and Hewitt 77] Baker, H. and Hewitt, C. The Incremental Garbage Collection of Processes. Conference Record of the Conference on AI and Programming Languages, ACM, Rochester, New York, August, 1977, pp. 55-59.

[Barber, de Jong, and Hewitt 83] Barber, G. R., de Jong, S. P., and Hewitt, C. Semantic Support for Work in Organizations. Proceedings of IFIP-83, IFIP, Sept., 1983.

[Birtwistle, Dahl, Myhrhaug, and Nygaard 73] Birtwistle, G. M., Dahl, O-J., Myhrhaug, B., Nygaard, K. *Simula Begin.* Van Nostrand Reinhold, New York, 1973.

[Bobrow and Stefik 82] Bobrow, D. G., Stefik, M. J. Loops: An Object Oriented Programming System for Interlisp. Xerox PARC, 1982.

[Borgida, Mylopoulos, and Wong 82] Borgida, A., Mylopoulos, J. L., Wong, H. K. T. Generalization as a Basis for Software Specification. Perspectives on Conceptual Modeling, Springer-Verlag, 1982.

[Borning and Ingalls 82] Borning, A. H., Ingalls, D. H. Multiple Inheritance in Smalltalk-80. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1982.

[Brock and Ackerman 78] Brock, J. D. and Ackerman, W.B. An Anomoly in the Specifications of Nondeterminate Packet Systems. Tech. Rep. Computation Structures Group None 33-1, M.I.T., January, 1978.

[Clinger 81] Clinger, W. D. Foundations of Actor Semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May, 1981.

[Curry, Baer, Lipkie, and Lee 82] Curry, G., Baer, L., Lipkie, D., Lee, B. Traits: An Approach to Multiple-Inheritance Subclassing. Conference on Office Information Systems, ACM SIGOA, June, 1982.

[Dijkstra 77] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, 1977.

[Friedman and Wise 76] Friedman, D. P., Wise, D. S. The Impact of Applicative Programming on Multiprocessing. Proceedings of the International Conference on Parallel Processing, ACM, 1976, pp. 263-272.

[Greif 75] Greif, I. Semantics of Communicating Parallel Processes. Technical Report 154, MIT, Project MAC, 1975.

[Hewitt 75] Hewitt, C.E. Protection and Synchronization in Actor Systems. SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications, ACM, March, 1975.

[Hewitt 77] Hewitt, C.E. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence 8-3* (June 1977), 323-364.

[Hewitt 83] Hewitt, C.E. Open Systems. Perspectives on Conceptual Modeling, Springer-Verlag, 1983.

[Hewitt and Baker 77] Hewitt, C. and Baker, H. Laws for Communicating Parallel Processes. 1977 IFIP Congress Proceedings, IFIP, August, 1977, pp. 987-992.

[Hewitt and de Jong 83] Hewitt, C., de Jong, P. Analyzing the Roles of Descriptions and Actions in Open Systems. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1983.

[Hewitt, Attardi, and Lieberman 79a] Hewitt C., Attardi G., and Lieberman H. Specifying and Proving Properties of Guardians for Distributed Systems. Proceedings of the Conference on Semantics of Concurrent Computation, Vol. 70, INRIA, Springer-Verlag, Evian, France, July, 1979, pp. 316-336.

[Hewitt, Attardi, and Lieberman 79b] Hewitt, C. E., Attardi, G., and Lieberman, H. Delegation in Message Passing. Proceedings of First International Conference on Distributed Systems, ACM, Huntsville, October, 1979.

[Hoare 74] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *CACM* (October 1974).

[Hoare 76] Hoare, C.A.R. Language Hierarachies and Interfaces. In *Lecture Notes in Computer Science,* Springer-Verlag, 1976.

[Hoare 78] Hoare, C. A. R. Communicating Sequential Processes. *CACM 21,* 8 (August 1978), 666-677.

[Kahn 81] Kahn, K. Uniform--A Language Based Upon Unification which Unifies (much of) Lisp, Prolog, and Act 1. University of Uppsala, March, 1981.

[Kornfeld and Hewitt 81] Kornfeld, W. A. and Hewitt, C. The Scientific Community Metaphor. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11,* 1 (January 1981).

[Landin 65] Landin, P. A Correspondence Between ALGOL 60 and Church's Lambda Notation. *Communication of the ACM 8,* 2 (February 1965).

[McCarthy 62] McCarthy, John. *LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, Ma., 1962.

[Milner 79] Milner, R. Flowgraphs and Flow Algebras. *JACM 26. No. 4* (1979).

[Reynolds 74] Reynolds, J.C. On the Relation Between Direct and Continuation Semantics. Proceedings of the Second Colloquium on Automata, Language and Programming, Springer-Verlag, 1974.

[Strachey and Wadsworth 74] Strachey, C. and Wadsworth, C.P. *Continuations - A Mathematical Semantics for Handling Full Jumps.* Univeristy of Oxford, Programming Reseach Group, 1974.

[Ward and Halstead 80] Ward, S. and Halstead, R. A Syntactic Theory of Message Passing. *JACM 27, No. 2* (1980), 365-383.

[Weinreb and Moon 81] Weinreb, D. and Moon D. *LISP Machine Manual.* MIT, 1981.