Working Paper 83

November 1974

# PROTECTION and SYNCHRONIZATION in ACTOR SYSTEMS

Carl Hewitt

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

November 1974

# Protection and Synchronization in Actor Systems

**Professor Carl Hewitt**
Department of Electrical Engineering
M.I.T
(617) 253-5873

Room 813
545 Technology Square
Cambridge, Mass. 02139

**Keywords:** Protection, Synchronization, Actors, Data Base Access, Data Definition Languages

## Abstract

This paper presents a unified method [called ENCASING] for dealing with the closely related issues of synchronization and protection in actor systems [Hewitt et al. 1973a, 1973b, 1974a; Greif and Hewitt 1975]. Actors are a semantic concept in which no active process is ever allowed to treat anything as an object. Instead a polite request must be extended to accomplish what the activator [process] desires. Actors enable us to define effective and efficient protection schemes. Vulnerable actors can be protected before being passed out by ENCASING their behavior in a guardian which applies the appropriate checks before invoking the protected actor. Protected actors can be freely passed out since they will work only for actors which have the authority to use them where authority can be decided by an arbitrary procedure. Synchronization can be viewed as a [time-variant] kind of protection in which access is only allowed to the encased actor when it is safe to do so.

## Introduction

Currently there are a variety of protection schemes that have been tried on various machines: domains [Schroeder, Needham, etc.], access control lists [MULTICS], objects [Wulf 1974] and capabilities [Dennis, Plummer, Lampson]. Actors are at least as powerful a protection mechanism as the above mechanisms in the sense that all of the above mechanisms can be realized modularly and efficiently using actors. The protection mechanisms of the above authors are all designed to be added onto conventional machines. Our approach is to build protection into the very basic primitives of a new kind of machine architecture. In this paper we show how to efficiently implement more traditional password [lock and key] schemes using the new primitives.

The actor transmission primitive provides a universal communication primitive that is suitable for use within a given machine as well as for communicating between machines. The same communication primitive can be used to efficiently get the next element of a list, to return an answer to a query or request, and to call a procedure with an argument. Since the same communication primitive is always used, a programmer need not be concerned whether the actor being sent the message resides on the same machine which sends the message or not. Actor transmission has the potential of helping us to develop systems in which computation is easily distributed across machines in an efficient, flexible, protected way.

Using actors, mutually suspicious subsystems which are guaranteed to confine the information entrusted to them are easily and efficiently implemented. Because actors can locally perform an arbitrary computational test whenever a message is passed and cannot be coerced there is reason to believe that they may be a semi-universal protection mechanism in the sense that any other protection mechanism can be efficiently defined using actors.

Important issues in privacy and protection that remain unsolved are those involving intent and trust. Here the concept of justification plays an important role. A protected subsystem that provides an answer should be able to justify that the answer is correct. Another set of important unsolved problems revolves around the conflict between the goal of privacy and the goals of simplicity and flexibility in sharing information. We are currently considering ways in which our model can be further developed to address these problems.

## Relationship to Knowledge-Based Systems

Currently successful research in knowledge-based systems is critically dependent on the choice of a "micro-world" in which to illustrate general principles of organizing intelligence. In recent years it has become apparent that relatively small self-contained programs are a rich source of micro-worlds. We are working to develop a Programming Apprentice to more systematically explore the programming micro-world. The actor semantics which we have developed have proved to be a very good foundation for expressing the semantics involved in very diverse kinds of processing. Actor semantics can be used to justify the principle of Actor Induction, which in turn provides the overall justification for the meta-evaluation process. One of the purposes of this paper is to extend the actor semantics into the areas of protection and synchronization which have previously not been systematically explored. Contracts and intentions for the programs in this paper need to be developed so we can determine what kind of extensions to our semantics might need to address the issues of protection and synchronization. Irene Greif is working on the development of axioms for cells and serializers, contracts for systems which allow parallel access to the information which they contain, and the planning rules that are needed to carry out the meta-evaluation of programs on the level of those presented in this paper. If we can successfully write contracts and carry out the meta-evaluation for the examples in this paper, it will be further evidence [together with the example of queues in "Towards a Programming Apprentice"] that programs are an interesting micro-world. In our experience it takes at least a half hour for most good programmers to digest one of these examples to the level of detail that is expounded in meta-evaluation.

For some time we have adopted an approach which has become known as the Thesis of Procedural Embedding of Knowledge: Knowledge of a domain is inextricably bound up with the procedures for its use. The development of actor semantics has clarified and re-reinforced this view. Every actor has a procedural aspect and a data aspect which are inextricably bound together. If the Procedural Embedding of Knowledge is to be anything more than a cleaned up version of the "Hairy Kludge a Month Paradigm" [which characterized many of the programs constructed in the first decade of artificial intelligence research], then the structure of the procedural knowledge that goes into large knowledge-based programs must be carefully delineated. We hope that our research on actor semantics and the development of a Programming Apprentice can make a contribution toward this end.

It is widely acknowledged that parallel processing can make a significant contribution to developing practical visual and audio input processing systems. The HEARSAY-II at Carnegie-Mellon University system is investigating parallelism for speech processing. Marilyn McLennan is investigating the use of parallelism in visual information processing for understanding pictures of plants. At the current level of development of the state of the art, it is not clear how high it is desireable to try to extend the low level parallelism into higher level reasoning. Nor is it known how to interface the low level parallel processing with the more serial computation that takes place in higher level reasoning. We originally developed serializers as a modular synchronization primitive for actor systems to aid investigating these issues. We now feel that parallelism may have a more important role than previously realized in explicating the structure of higher level symbolic processing. Until recently it has

been widely accepted that parallelism is not suited for the higher level symbolic processing of problem solving. Most psychological evidence seems to point to humans being almost entirely serial in their high level problem solving. The efficiency that can be gained by using parallelism for higher level processing is currently not known: Minsky has conjectured that n processors will usually only bring a speed up of (log n).

However the development of actor semantics has brought about a shift in our paradigm for problem solving. Early programs which tried to be expert in some domain were thought to be analogous to an individual human expert who was expert in the domain. Most programs were developed on the basis that there should be a unified coherent intelligence which was directing all aspects of the problem solving in a serial fashion. The development of the actor model of computation has encouraged us to attempt to develop a paradigm based on a society of experts communicating by passing messages. This switch in paradigm has provided us with a rich source of ideas for problem solving strategies: the societies in which we live. We have developed a dialogue style of programming which places its emphasis on the modular distribution of knowledge and clean means of communication between pieces of knowledge. Thinking and programming in this new paradigm has in turn caused us to re-evaluate the case for parallelism. We note that societies often make good use of parallelism for a variety of purposes. In serial processing since only one thing can be done next, the choice of what to do next is critical. However, the mechanism that is used to choose what to do next must be relatively efficient or the problem solver will spend most of its time deciding what to do next. The criteria of efficiency vs. criticallity for a method for selecting what to do next often come into conflict. This conflict may have been the source of the large number of overly simplistic decision mechanisms [such as the decision tables in GPS] in early A.I. programs.

An alternative to trying very hard to make the very best choice is to pursue in parallel all the reasonable alternatives [for which there are enough processing resources] relying on those which run into severe difficulties to record the nature of the difficulty and commit suicide. This alternative parallel style fits well with what has been learned about desirable properties of problem solving modules:

> A problem solving module must be able to work with partial information.
>
> A problem solving module should not be dependent on any one crucial feature in order to be able to run.
>
> A problem solving module should be prepared to get stuck, try other avenues of approach, and then restart when there is more information available at a later time.

The additional programming burden imposed by parallelism is the task of synchronizing the above activities of modules running in parallel. We hope that by analyzing the structure of problem solvers

that attempt to use parallelism at the highest levels of problem solving that we can further explicate the structure of problem solving. This paper contributes toward that aim through the explication of a modular synchronization primitive.

## Theoretical Basis

This work is presented using ACTORS, which are potentially active chunks of knowledge that can communicate by sending messages. The actor model is a precise model of computation in the same sense that the Turing Machine and the PDP-10 are precise models of computation. The primitive kind of action for a Turing Machine consists of writing a symbol on its tape, shifting right or left, and changing state. The primitive kind of action for a PDP-10 is executing a single instruction. The primitive kind of action in the actor model of computation is sending a message actor to another actor which can make sense out of the message. Just as a PDP-10 requires several different kinds of primitive instructions in order to run, so also there are several different kinds of primitive actors.

> A sequence is a primitive kind of actor that allows actors to be grouped in a linear structure. If a sequence S is sent an integer n then it sends back its nth element $S_n$. For example if [x 'b 4] is sent a 2 then it replies with 'b.
>
> A cell is a primitive kind of actor which allows actors to be constructed which have a behavior that varies with time. If a cell C is sent a message **contents?** then it sends back its current contents. If C is sent a message of the form [← new-contents] then thereafter (until it is sent another such message) when it is sent the message **contents?** it replies with new-contents.

In the actor model of computation, no active process is ever allowed to treat anything as an object; instead a polite request must be extended to accomplish what the activator desires. The basic (and only) kind of event that occurs in the actor semantics is that a message actor is passed to a target actor. The act of message-passing is like a handshake -- the message actor reaches out to make contact, and the target actor reaches out to receive it. As long as the hands mesh sufficiently, the information can pass between them.

## The ACTOR Model of Computation

The actor model of computation consists of three tightly related and mutually consistent components:

Axiomatic behavioral specifications of the primitive actors.

A graphical notation for actor computations which represents the relationship between the events that occur in the middle of actor computations.

A modular formalism based on actor semantics in which to write programs. The programming formalism [called **PLASMA** for **PLA**NNER-like **S**ystem **M**odeled on **A**ctors] attempts to achieve the following goals:

> For any given piece of knowledge there is a natural place in the formalism to incorporate the knowledge so that it will be used when it is appropriate and not used when it is inappropriate.
>
> As much knowledge as desired can be put in any actor.
>
> A system can be directly programmed without circumlocution once the behavior of objects and the messages passed between them has been specified.
>
> As many properties as possible of the behavior of a program should be manifest in the structure of the code.

## A Brief Introduction to PLASMA Syntax

Before introducing any formal examples, we must briefly introduce some of the common PLASMA syntax which is used in the rest of the paper. In particular there are several constructs which need to be explained at this point. Meta-syntactic variables will be underlined.

### Sequences and Bags

We note initially that [$A_1$ $A_2$ ... $A_N$] means a ordered sequence of the actors $A_1$ through $A_N$ whereas {$A_1$ $A_2$ ... $A_N$} means a unordered bag of the actors $A_1$ through $A_N$. Thus [3 "b"] is not equivalent to ["b" 3] although {3 "b"} is equivalent to {"b" 3}. Also bags behave differently from mathematical sets in that {3 "b" 3} is not equivalent to {3 "b"}.

### Transmitters

Simple syntax for sending a message M to an actor T is:

(T <= M)

or, which is entirely equivalent,

(M => T).

Thus,

(["this" "is" "a" "simple" "sentence"] => parser)

will send a sequence of 5 words to the actor called parser.

We need to have some syntactic mechanism to denote special syntactic forms ($E_1$ $E_2$ ... $E_n$) that are not ordinary procedure calls with procedure $E_1$ and arguments $E_2$, ..., and $E_n$. We will use italicized symbols for this purpose to increase the readability of PLASMA. By using italicized symbols, PLASMA avoids the introduction of FEXPRS as in LISP or reserved words. FEXPRS are undesirable because they can violate the privacy and security of actors making it more difficult for the programming apprentice to understand.

For example both => and <= are italicized symbols. The italicized symbols => and <= are read as forms of the verb "send". For example (an-actor <= [1 3]) would be read as " an-actor is sent the sequence 1 3", or "a sequence of 1 and 3 is sent to an-actor."

There are actually many flavors of "sends". These tend to deal with specialized aspects of actor control structure, which is more general than the usual recursive control structure used in ALGOL. We have developed a simple scheme for multiple heads and shafts which is quite friendly once it is understood. However, the above syntax is the simplest and much the most common.

Sometimes it is not convenient to explicitly write out either of the "send" symbols <= or =>. Therefore,

($E_1$ $E_2$ ... $E_n$)

where none of the $E_i$ are italicized symbols is taken to be an ordinary procedure call where $E_1$ is the procedure and [$E_2$ ... $E_n$] is the sequence of its arguments. In this case the above expression is completely equivalent to

($E_1$ <= [$E_2$ ... $E_n$])

which as we remarked above is completely equivalent to

([$E_2$ ... $E_n$] => $E_1$)

If any of the $E_i$ are italicized then ($E_1$ $E_2$ ... $E_n$) then its meaning is determined by the definition of the form of the italicized symbols in it.

For example

| | | |
|---|---|---|
| (factorial 3) | is equivalent to | (factorial <= [3]) |
| (generate) | is equivalent to | ([] => generate) |
| (* 3 4 (1 + 2)) | is equivalent to | (* <= [3 4 (1 <= [+ 2])]) |

Note that when the <= or => are written out explicitly in a special syntactic form, there is always one expression before the arrow and one after it.

Also note that arithmetic can be expressed in infix notation [as well as the LISP prefix notation], producing the unusual semantics of sending "+" to a number, rather than the other way around. This infix notation is used freely.

The use of parentheses is similar to that of languages such as LISP, except that by using the arrows one can be more explicit about who is being sent what message. Thus, the meaning in PLASMA of any expression of the form

$$(\underline{\text{a-function}}\ arg_1\ arg_2 \ldots arg_n)$$

will correspond closely to its usual meaning in LISP. The usual LISP meaning would be to apply the function a-function to list of $arg_1$ through $arg_n$. The PLASMA meaning is to send a-function the same arguments, in a sequence wrapped in a message indicating that this is a normal application. I.E. it could also be written

$$(\underline{\text{a-function}} <= [arg_1\ arg_2 \ldots arg_n])$$

The PLASMA syntax allows one to specify things other than sequences of arguments, and gives the flexibility of making the code more readable by putting the message first.

## Receivers

Reminiscent of the LISP lambda expression is the ACTOR message-receiver:

(=> pattern
body)

where the italicized symbol => is read as "receives". This means that if an actor with this definition is sent a message which matches pattern it will evaluate body in the environment resulting from the

pattern match. Patterns will often use a notation by which names are bound in an environment. For example, if 3 is matched against the pattern =x, then x will be bound to 3.

For example, the following routine adds one to any message it is passed:

```
(≡> =x
    (x + 1))
```

## Conditionals

Conditionals take two standard forms. The first is known as the rules expression and has the form:

```
(rules an-expression
   (≡> pattern1 body1)
   (≡> pattern2 body2)
      ...
   (≡> patternn bodyn))
```

The expression is matched against the successive patterns until it matches one of them; then the corresponding body is evaluated in the environment resulting from the pattern match. For example,

```
(rules (3 + 4)
   (≡> (even)
      yes)
   (≡> (odd)
      no))
```

evaluates to no.

A similar construct, more convenient at the outer level of message reception in an ACTOR definition, is the cases statement:

```
(cases
   (≡> pattern1 body1)
   (≡> pattern2 body2)
      ...
   (≡> patternn bodyn))
```

in which the incoming message is matched directly against the successive patterns until a match is found, whereupon the corresponding body is evaluated in the resultant environment.

## Definitions

ACTOR definitions are usually given labels by using:

[a-label ≡ a-definition]

which means that a-label is taken as the name of the procedural call-by-value fixed point of a-definition. For example, we can define fibonacci as follows:

```
[fibonacci ≡
   (cases
      (≡> 1
          1)
      (≡> 2
          2)
      (≡> [≡n]
          (+
             (fibonacci (n - 1))
             (fibonacci (n - 2))))))]
```

A set of mutually recursive definitions can be given using the italicized *labels* symbol using the form:

```
(labels
  {[name1 ≡ D1]
   [name2 ≡ D2]
      ...
   [namen ≡ Dn]}
  body)
```

which evaluates body in an environment with each $name_i$ bound to the value of $D_i$. The equations are mutually recursive in that any occurrence of a $name_i$ within a $D_k$ refers to $D_i$.

A different way of binding names is in the use of the italicized *let* symbol, which takes the form:

```
(let
  {[name1 ≡ E1]
   [name2 ≡ E2]
      ...
   [namen ≡ En]}
  body )
```

which evaluates body in an environment with each $name_i$ bound to the value of $E_i$. The equations of a *let* expression are not mutually recursive. For example:

```
(let
    {[x = 3]
     [y = 5]}
    (let
        {[y = (x + y)]}
        (x * y)))
```

will return 24.

## The Password Protection Problem

In this section we show how to use the intrinsic protection that is inherent in the actor concept to efficiently implement more traditional password [lock and key] schemes. We show how to efficiently implement protected actors which can be freely passed out since they will work only for actors and/or processes which have the authority to use them.

For example suppose that we wish to entrust a certain actor to a community but don't want unauthorized users to be able to interrogate or harm the actor. We decide that it will be necessary for an authorized user to have a password [key] in order to use the protected actor. Following the lead of Morris we define an actor cons-security which constructs a pair of actors [seal unseal] such that the only way to extract x from (seal x) is to apply the actor unseal. The fundamental idea we need is that (seal x) should ENCASE the behavior of x in a container which will only disgorge x if given the proper password. n the program below we will find it convenient to make use of an actor construct-a-new-actor which returns a new actor every time it is invoked. The only properties of the values of construct-a-new-actor on which we rely are that it always returns a new actor and there is no other way to create an actor which is a value of construct-a-new-actor. We make an initial attempt to define cons-security as follows:

```
[cons-security ≡                          ;define the actor cons-security to have the following behavior
   (≡> []                                                                  ;receive no arguments
      (let
         {[the-password ≡ (construct-a-new-actor)]}   ;construct-a-new-actor always generates a new actor
         (labels                                                          ;define the following actors
            {
               [seal ≡                                                                ;define an actor seal
                  (≡> [≡x]                ;such that when it receives an argument x it returns an actor which
                     (≡> [≡the-alleged-password]                    ;when it receives an alleged password
                        (rules the-alleged-password                          ;checks to see if it is
                           (≡> (identical the-password)                      ;identical to the password
                              x)                                             ;and if so sends back x
                           (else    ;else take the appropriate action for an attempted protection violation
                              (attempted-password-violation)))))]
               [unseal ≡                                                              ;also define an actor unseal
                  (≡> [≡the-alleged-container]          ;such that when it receives an alleged container
                     (the-alleged-container the-password))]}
                                                          ;then the alleged container is sent the password
            [seal unseal])))]                                ;send back the pair of actors [seal unseal]
```

The actor cons-security defined above has the nice property that if it is called twice producing two pairs of actors $[s_1\ u_1]$ and $[s_2\ u_2]$, then actors sealed by $s_1$ cannot be unsealed by $u_2$. The reason for this is that every time the actor cons-security is called a new actor is created which serves as the-password.

Unfortunately, we can write a program cons-fake-container which given a sealed-container can

molest the contents by pretending to be a sealed container in order to find out from unseal what the password is and then opening the real container itself! That is an unscrupulous party can molest the contents of containers even though it has not been given either the sealer or the unsealer for that kind of container. This can be done by simply making "fake containers" and sending them [or simply leaving them lying around in public message boxes] for others to unseal. The actor **cons-fake-container** makes fake containers which contain real containers inside them.

```
[cons-fake-container ≡                    ;define the actor cons-fake-container which behaves as follows
   (≡> [=a-sealed-container]                  ;receive an argument called a-sealed-container
                                              ;return an actor which behaves as follows
      (≡> [=the-password]                     ;receive the password
         (MOLEST                              ;molest the actor obtained by
            (a-sealed-container the-password)))]
                                           ;sending a-sealed-container the password obtained above
```

An unauthorized member of the community which has access to real containers but does not have access to the unseal actor for the container can hope to violate protection by enclosing real containers in fake containers and sending them back out to the community.

```
[an-unauthorized-user ≡
   (≡> =the-real-container
      (the-community <= (cons-fake-container the-real-container)))]
```

Our solution to the password protection problem is closely related to the solution using trademarks that has been developed by James H. Morris. It differs from the trademarks of Morris in that our system each actor is born with a archetype which can be checked by interested parties instead of having interested parties being able to trademark any object whatsoever for future reference as Morris does. The actor approach is to build archetypes into the very basis of the system whereas trademarks can be added onto an already existing system. On an actor machine, the implementation of archetypes is extremely efficient. Although archetypes by themselves do not seem to have all the generality of the trademark mechanism of Morris, it is not clear that there are any circumstances in which the extra generality is really needed.

Protection and privacy applications often require that some positive means of identification be provided before certain actions be taken or information divulged. To this end it is useful to have an actor [which we will name "archetype"] such that for any actor x, **(archetype x)** is an unforgeable identification of the kind of behavior that x has. The only useful behavior of an archetype is to determine whether it is identically the same as some other archetype. Thus if **(archetype x)** is recognized as being the archetype of an an actor whose behavior is known, then an unforgeable password may be entrusted to x since knowledge of the archetype of x determines the action that x will take when it receives a message. Thus two mutually suspicious actors can establish a safe basis for communication without either being hostage to the other. We will now introduce a convenient syntax for binding archetypes:

name-to-be-bound-to-archetype-of-the-actor-which-follows <≡>$_{name\text{-}of\text{-}label\text{-}set}$ code-for-an-actor

The above syntax does not actually introduce any new primitive mechanisms; it simply makes it more convenient to use the archetype machinery which already exists. Use of archetypes for protection purposes is similar to the way types can be used in SIMULA-67 for protection purposes. An important difference is that archetypes are actors and therefore can be passed around during program execution whereas types in SIMULA-67 are not values and cannot be passed during program execution.

Types in SIMULA-67 also used to define the correspondence between an abstract operation on objects and the code bodies that actually implement the operation. Actors define the semantics of the correspondence in terms of passing a message to the object asking it to perform the operation.

```
[cons-security ≡                          ;define the actor cons-security to have the following behavior
   (≡> []                                                                  ;receive no arguments
      (let
         {[the-password ≡ (construct-a-new-actor)]}    ;construct-a-new-actor always generates a new actor
         (labels strong-box                    ;define the following actors in a label set named strong-box
            {
               [seal ≡                                                              ;define an actor seal
                  (≡> [≡x]                ;such that when it receives an argument x it returns an actor which
                   sealed-container <≡>strong-box                ;with archetype named sealed-container bound
                                                          ;to the actor below in the label set named strong-box
                        (≡> [≡the-alleged-password]                ;when it receives an alleged password
                           (rules the-alleged-password                          ;checks to see if it is
                              (≡> (identical the-password)                      ;identical to the password
                                 x)                                             ;and if so sends back x
                              (else    ;else take the appropriate action for an attempted protection violation
                                 (attempted-password-violation))))))]
               [unseal ≡                                                        ;also define an actor unseal
                  (≡> [≡y]                                          ;such that when it receives an argument y
                     (rules y                                                   ;then the rules for y are
                        (≡>  (has-archetype sealed-container)
                                                                ;that if it has the archetype sealed-container
                           (y the-password))                                    ;then y is sent the password
                        (else                    ;else if y does not have the archetype sealed-container then
                           (attempted-container-violation)))))]}
                                          ;take the appropriate action for an attempted-container-violation
         [seal unseal]))))]                                          ;send back the pair of actors [seal unseal]
```

As we mentioned above our definitions are slightly different from those of Morris because we rely on the fact that every actor is born with an unforgeable archetype instead of on the ability to trademark objects. In practical terms this means that we envisage a system in which the above kind of protection capability is built in to the very heart of the system instead of being added on to an already existing scheme. Notice the constraint in the definition of unseal that the argument to unseal must be bound to an actor which has the archetype sealed-container. Suppose the new actor cons-security

defined above it is called twice producing two pairs of actors [$s_1$ $u_1$] and [$s_2$ $u_2$]. Then it will be the case for all actors x and y that

(archetype ($s_1$ x)) = (archetype ($s_2$ y)).

Without the constraint the program **cons-fake-container** given above can molest the contents of a real container by pretending to be a sealed container in order to find out from **unseal** what the password is and then opening the real container itself. The author of **cons-fake-container** might still attempt to violate protection using the following slight modification to her program:

```
[cons-fake-container ≡                    ;define the actor cons-fake-container which behaves as follows
   (≡> [=a-sealed-container]              ;receive an argument called a-sealed-container
                                          ;return an actor which behaves as follows
 sealed-container <≡>                     ;the archetype of a fake container is named sealed-container
      (≡> [=the-password]                 ;receive the password
         (MOLEST                          ;molest the actor obtained by
            (a-sealed-container the-password)))]
                                          ;sending a-sealed-container the password obtained above
```

However, the author of the program **cons-fake-container** cannot break the security simply by choosing the name "sealed-container" for the archetype of fake containers. The name "sealed-container" in the environment of **cons-fake-container** will be bound to a different actor than in the environment of the interior of **cons-security**.

Of course, the party who calls **cons-security** and receives a pair [s u] must be careful as to whom it gives the seal s and the unseal u. If given to unscrupulous parties, s can be used to counterfeit messages and u can be used to commit vandalism or to violate privacy. Within the actor theory of computation we would like to rigorously formulate and prove a result to the effect that an actor sealed using s can only be accessed by unsealing it with u.

## Serializers

Serializers are generalizations of the "secretary" concept which was conceived by Dijkstra and later developed into "monitors" by Hansen, and Hoare. They are used to arbitrate access to shared actors with side effects. Roughly speaking, serializers are analogous to monitors in the same way that actors are analogous to SIMULA-67 classes. A serializer is an actor that will allow only one process to be executing inside it at a time whereas a monitor is a SIMULA-67 class that will allow only one process to be executing inside it at a time. Actors differ from SIMULA-67 classes in the way in which the correspondence is defined between an abstract operation on objects and the code bodies that actually implement the operation. Actors define the correspondence in terms of passing a message to the object asking it to perform the operation. SIMULA-67 requires that the class of each object be declared in the text of the program and the correspondence is made by checking the type of the object within its declared class to determine the code body which is applicable.

A serializer bears an analogy to a railway switchyard in which only one train [process] is allowed to move at a time. A general principle of efficient operation that is applicable to both serializers and monitors is to try to keep the serializer [monitor] unlocked as much of the time as possible to keep it from being a bottleneck in the operation of a larger system. Our serializers have an important advantage over "monitors" [Hansen 1973, Hoare 1973] in that they completely encase the actor to which access is supposed to be controlled. Thus control can possess a serializer, then temporarily yield possession of the serializer and travel into the actor encased in the serializer, repossess the serializer when the encased actor replies, and finally release the serializer with the reply from the encased actor. The above behavior for serializers is illustrated by a solution to the readers-writers problem. We will contrast our solution to the problem with one given by Hoare using monitors. The example illustrates how serializers using the concept of encasing a protected actor can prevent the possibility of certain obscure kinds of timing errors from being possible.

## Monitors

The following description of a monitor implementation of the readers writers problem is taken from Hoare's paper:

"As a more significant example, we take a problem which arises in on-line real-time applications such as airspace control. Suppose that each aircraft is represented by a record, and that this record is kept up to date by a number of "writer" processes and accessed by a number of "reader" processes. Any number of "reader" processes may simultaneously access the same record, but obviously any process which is updating (writing) the individual components of the record must have exclusive access to it, or chaos will ensue. Thus we need a class of monitors; an instance of this class local to each individual aircraft record will enforce the required discipline for that record. If there are many aircraft, there is a strong motivation for minimizing local data of the monitor; and if each read or write operation is brief, we should also minimize the time taken by each monitor entry.

When many readers are interested in a single aircraft record, there is a danger that a writer will be indefinitely prevented from keeping that record up to date. We therefore decide that a new reader should not be permitted to start if there is a writer waiting. Similarly, to avoid the danger of indefinite exclusion of readers, all readers waiting at the end of a write should have priority over the next writer. Note that this is a very different scheduling rule from that propounded in [Courtois, Heymans, and Parnas], and does not seem to require such subtlety in implementation. Nevertheless, it may be more suited to this kind of application, where it is better to read stale information than to wait indefinitely!

The monitor obviously requires four local procedures:

| | |
|---|---|
| **startread** | entered by reader who wishes to read. |
| **endread** | entered by reader who has finished reading. |
| **startwrite** | entered by writer who wished to write. |
| **endwrite** | entered by writer who has finished writing. |

We need to keep a count of the number of users who are reading, so that the last reader to finish will know this fact:

```
readercount:integer
```

We also need a **Boolean** to indicate that someone is actually writing:

```
busy:Boolean;
```

We introduce separate conditions for readers and writers to wait on:

```
OKtoread, OKtowrite:condition;
```

The following annotation is relevant:

```
OKtoread ≡ ¬ busy
Oktowrite ≡ ¬ busy ∧ readercount = 0
invariant: busy → readercount = 0
```

```
class readers and writers:monitor
   begin
      readercount:integer;
      busy:Boolean;
      OKtoread, OKtowrite:condition;
     procedure startread;
        begin
           if busy v OKtowrite.queue
                        then OKtoread.wait;
           readercount := readercount + 1;
           OKtoread.signal;
           comment Once one reader can start, they all can;
        end startread;
     procedure endread;
        begin
           readercount := readercount - 1;
           if readercount = 0 then OKtowrite.signal;
        end endread;
     procedure startwrite;
        begin
           if readercount ≠ 0 v busy
                        then OKtowrite.wait;
           busy := true;
        end startwrite;
     procedure endwrite;
        begin
           busy:= false;
           if OKtoread.queue
                        then OKtoread.signal
                        else OKtowrite.signal;
        end endwrite;
     readercount := 0;
     busy := false;
  end readers and writers;"
```

A major structural difficulty with monitors is brought out by the readers and writers example. The problem is that in the **readers and writers** monitor there is no guarantee in the monitor itself that a reader of the data base does not call the procedure **endread** before calling the procedure **startread**, It would clearly be a mistake to do this and the code of the readers and writers monitor implicitly assumes that this will never be done. For example if calling **endread** is the very first procedure of the monitor which is called then the reader count will become negative! If this kind of mistake is accidentally made then it can cause exactly the kind of obscure form of time-dependent coding error that monitors were introduced to avoid. In the serializer concept which we introduce below we introduce a mechanism which because of its very nature guarantees that this kind of error cannot occur.

## How Serializers Work

A serializer with name serializer-name is constructed by an expression of the form

```
(serializer serializer-name
   (queues: set-of-queues-for-the-serializer)
   (after-possession-to: receiver-for-messages))
```

If the actor defined by an expression of the form given above is sent a message M then M will be sent to receiver-for-messages after the activator [process] which sent the message gets possession of the serializer.

When an activator executes an expression of the following form

```
(enqueue the-queue (afterwards: the-continuation))
```

it is queued on the-queue where it remains dormant until it reaches the front of the-queue and some other activator specifies that the-queue should be served next, which causes the enqueued activator to resume with the-continuation. The enqueuing of the activator can be made conditional in the following way:

```
(enqueue the-queue
   (if: condition-for-entering-the-queue)
   (afterwards: the-continuation))
```

If condition-for-entering-the-queue is false then the activator proceeds directly with the execution of the-continuation without releasing possession of the serializer. The reader should note that the treatment of queues in serializers is somewhat different from the treatment in monitors. We originated the above syntax for resuming the execution of dequeued activators after Edsger Dijkstra [private communication] pointed out to us that the way in which monitors dequeue in the middle of a monitor has potential difficulties in that local variables can become bound and other assumptions made, some of which may not be valid when the enqueued activator resumes execution. The above syntax for the **enqueue** primitive makes manifest the scope of the code affected by the possible change in the world induced by being enqueued on the-queue and then emerging to resume execution with the-continuation after the state of the serializer has changed. We are considering using the usual scope rules of our language to deal with the potential world shift involved in this case and also another very similar case which is introduced by the yield primitive which is described below.

When possession of a serializer is yielded it is possible to specify which activator should get possession next.

```
(enqueue the-queue the-message
   (serve-next: another-queue)
   (afterwards: the-continuation))
```

The activator which is at the head of another-queue immediately gets possession of the serializer before any other activator. There is an implicit queue in front of every serializer which for convenience is given the same name as the serializer. Every activator which gains possession of a serializer must pass at least once through this implicit queue.

Within the serializer, an expression of the following form temporarily yields possession of the serializer so that the-expression can be evaluated without possession of the serializer.

```
(yield the-serializer in-order-to the-expression
   (then-after-repossession-to: receiver-for-value-of-the-expression)
   (serve-next: a-queue))
```

In parallel with evaluating the-expression the next activator on a-queue is served with execution resuming from the point at which it was enqueued. We mentioned above that every serializer has an implicit queue with the same name as the serializer. When an activator yields possession of the serializer then it must again pass through the same implicit queue in order to repossess the serializer. It may be more efficient in certain applications to give priority to those activators seeking to repossess a serializer over those seeking possession of the serializer for the first time.

Within the serializer, an expression of the following form releases the serializer after the value of the-expression has been computed. The value returned by the-serializer is the value of the-expression.

```
(release the-serializer to-return the-expression
   (serve-next: a-queue))
```

The polymorphic operators read and write defined below can be used to read and write in data bases regardless of whether or not they have serializers in front of them.

```
[read ≡                          ;define the operator read which reads a data base according to directions
  (≡> [=the-data-base =the-directions]          ;receive a data base and directions for reading
      (the-data-base <= (read: the-directions)))]
                                   ;request to read the data base according to the directions

[write ≡                         ;define the operator write which writes into a data base according to directions
  (≡> [=the-data-base =the-directions]          ;receive a data base and directions for writing
      (the-data-base <= (write: the-directions)))]
                                   ;request to write in the data base according to the directions
```

Below we define an operator readers-and-writers such that (readers-and-writers a-data-base) returns a-data-base protected inside a scheduler which works according to the criteria enumerated above. This way of encasing a-data-base has very desirable properties of modularity in that it separates the scheduling policy of a-data-base from how it is actually implemented.

In the solution we will make use of the following notations:

| | |
|---|---|
| $c | *for the contents of the cell* c |
| (c ← v) | *to update the contents of* c *to be* v |
| *q | *for the length of the queue* q |

The following invariants are relevant to understanding the solution:

```
(non-negative-integer $num-writers)
(non-negative-integer $num-readers)
($num-writers = 0)  v  ($num-writers = 1)
($num-readers > 0)  →  ($num-writers = 0)
($num-writers = 1) → ($num-readers = 0)
```

```
[readers-and-writers =          ;readers-and-writers constructs a guardian of the data base which is its argument
  (=> [=the-data-base]                                                    ;receive a data base
    (let {[num-writers = @0]          ;let the number of writers in the data base be a new cell initialized to 0
          [num-readers = @0]}         ;let the number of readers in the data base be a new cell initialized to 0
      (serializer s                                                       ;s is the name of the serializer which has
          (queues: {readers-q writers-q})          ;two queues called the readers-q and the writers-q
          (after-possession-to:                                       ;after obtaining possession of the serializer
            (cases                                                    ;there are two cases
              {(=> (read: =the-directions)          ;the request is to read the data base using the directions
                 (enqueue readers-q                                   ;enqueue in the readers-q
                    (if:                                              ;if either
                       (∨ ($num-writers = 1)                          ;there is a writer in the data base or
                          (#writers-q > 0)))                          ;there are waiting writers
                    (afterwards:
                       (num-readers ← ($num-readers + 1))    ;increment the count of the readers in the data base
                       (yield s in-order-to (read the-data-base the-directions)
                                                                      ;yield s to read the data base
                          (serve-next:                                ;specifying that
                             (if (∧ (#readers-q > 0) (#writers-q = 0))
                                                                      ;if both there are waiting readers and no waiting writers
                                 then readers-q                       ;then the readers-q should be served next
                                 else s))
                          (then-after-repossession-to:          ;after regaining possession of the serializer
                             (=> =the-output                          ;receive the output
                                (num-readers ← ($num-readers - 1))    ;decrement the count of the readers
                                (release s to-return the-output       ;release s to return the output
                                  (serve-next:                        ;specifying that
                                    (if (∧ ($num-readers = 0)         ;if both the number of readers is 0
                                           (#writers-q > 0))          ;and there are waiting writers
                                        then writers-q                ;then serve the writers-q next
                                        else s)))))))))
               (=> (write: =the-directions)          ;receive a request to write in the data base using the directions
                  (enqueue writers-q                                  ;enqueue in the writers-q
                     (if:                                             ;if
                        (∨ ($num-writers = 1)                         ;there is a writer in the data base
                           (#writers-q > 0)                           ;or there are waiting writers
                           ($num-readers > 0)))                       ;or there are readers in the data base
                    (afterwards:
                       (num-writers ← 1)                              ;set the number of writers to 1
                       (yield s in-order-to (write the-data-base the-directions)
                                                                      ;yielding s to write in the data base
                          (then-after-repossession-to:          ;after obtaining repossession of the serializer
                            (=> =the-output                           ;receive the output
                              (num-writers ← 0)                       ;set the number of writers to 0
                              (release s to-return the-output         ;release the serializer s to return the output
                                 (serve-next:                         ;specifying that
                                   (if (#readers-q > 0)               ;if there are waiting readers
                                     then readers-q                   ;then the readers-q should be served next
                                    else-if (#writers-q > 0)          ;else if there are waiting writers then
                                     then writers-q                   ;then the writers-q should be served next
                                     else s)))))))))))))]             ;else serve the serializer s next
```

## Acknowledgements

This research was sponsored by the M.I.T. Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research. Our syntax for binding names to archetypes was suggested by Guy Steele. Irene Greif, Howie Shrobe, Ben Kuipers, Marilyn McLennan, and Jim Rumbaugh made many helpful comments and suggestions that materially improved the presentation of the material in this paper. Conversations with E. W. Dijkstra, Tony Hoare, John Reynolds, and Ole-Johan Dahl have helped us to crystallize our ideas on these issues.

# Bibliography

Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes" Acta Informatica. 1971.

Dijkstra, E. W. "The Humble Programmer" CACM. October, 1972.

Dijkstra, E. W. "Notes on Structured Programming" Aug. 1969.

Fisher, D. A. "Control Structures for Programming Languages" Ph. b.. Carnegie. 1970.

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

Hansen, P.B. "Operating System Principles" Prentice-Hall. 1973.

Hewitt, C., Bishop P., and Steiger, R. "A Universal Modular Actor Formalism for Artificial Intelligence" IJCAI-73. Stanford, Calif. Aug, 1973. pp. 235-245.

Hewitt, Carl et al. "Actor Induction and Meta-evaluation" Conference Record of ACM Symposium on Principles of Programming Languages. Boston. Oct, 1973.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" Stanford. 1973.

Hoare, C. A. R. "An Axiomatic Definition of the Programming Language PASCAL" February 1972.

Kay,Alan C. "FLEX, A Flexible Extendible Language" CS Tech. Rept. U. of Utah. 1968.

Kay,Alan C. "Reactive Engine" Ph. D. thesis at University of Utah, 1970.

Kay,Alan C. and the Learning Research Group. "The SMALL TALK Note Book" Forthcoming.

Landin, P. J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation" CACM. February, 1965.

McCarthy, J.; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; and Levin, Michael I. "Lisp 1.5 Programmer's Manual, M. I. T. Press"

Mitchell, J. G. "The Modular Programming System: Processes and Ports" NIC 7359. June, 1971.

Morris, J. H. "Verification-oriented Language Design" Berkeley Technical Report 7. December, 1972.

Reynolds, J. C. "GEDANKEN-A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept" CACM, 1970.

Reynolds, J. C. "Definitional Interpreters for Higher-Order Programming Languages" Proceedings of ACM National Convention 1972.

Smith, Brian and Hewitt, Carl. "Towards a Programming Apprentice" AISB Conference. July 1973.

Steele, G. L. "Multiprocessing Compactifying Garbage Collection" September, 1974.

Tesler, L. G.; Enea, H. J.; and Smith, D. C. "The LISP70 Pattern Matching System" IJCAI-73. August 1973.

WangA. and Dahl O. "Coroutine Sequencing in a Block Structured Environment" BIT 11 425-449.

Wirth, N. "Program Development by Stepwise Refinement" CACM 14, 221-227. 1971.