Actor Induction and Meta-evaluation

Carl Hewitt
Peter Bishop
Irene Greif
Brian Smith
Todd Matson
Richard Steiger

"Programs should not only work,
but they should appear to work as well."

PDP-1X Dogma

The PLANNER project is continuing research in natural and effective means for embedding knowledge in procedures. In the course of this work we have succeeded in unifying the formalism around one fundamental concept: the ACTOR. Intuitively, an ACTOR is an active agent which plays a role on cue according to a script. We use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. Data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, demons, processes, contexts, and data bases can all be shown to be special cases of actors. All of the above are objects with certain useful modes of behavior. Our formalism shows how all of these modes of behavior can be defined in terms of one kind of behavior: sending messages to actors. An actor is always invoked uniformly in exactly the same way regardless of whether it behaves as a recursive function, data structure, or process.

"It is vain to multiply Entities beyond need."
William of Occam

"Monotheism is the Answer"

The unification and simplification of the formalisms for the procedural embedding of knowledge has a great many benefits for us. In particular it enables us to substantiate properties of procedures more easily.

INTENTIONS: Furthermore the confirmation of properties of procedures is made easier and more uniform. Every actor has an INTENTION which checks that the prerequisites and the context of the actor being sent the message are satisfied. The intention is the CONTRACT that the actor has with the outside world. How an actor fulfills its contract is its own business. By a SIMPLE BUG we mean an actor which does not satisfy its intention. We would like to eliminate simply debugging of actors by the META-EVALUATION of actors to show that they satisfy their intentions. By this we do not necessarily mean a proof in the first order quantificational calculus for input-output assertions written in the first-order quantificational calculus. The rules of deduction to establish that actors satisfy their intentions essentially take the form of a high level interpreter for abstractly evaluating the program in the context of its intentions. This process [called META-EVALUATION] can be justified by a form of induction. In general in order to substantiate a property of the behavior of an actor system some form of induction will be needed. At present, actor induction for an actor configuration with audience E can be tentatively described in the following manner:

1. The actors in the audience E satisfy the intentions of the actor to which they send messages.

and

2. For each actor A the intention of A is satisfied => the intentions for all actors sent messages by A are satisfied

---

Therefore

The intentions of all actions caused by E are satisfied
(i.e. the system behaves correctly)

Computational induction [Manna], structural induction [Burstall], and Peano induction are a special cases of ACTOR induction. Actor based intentions have the following advantages:

The intention is decoupled from the actors it describes.

Intentions of concurrent actions are more easily disentangled.

We can more elegantly write intentions for dialogues between actors.

The intentions are written in the same formalism as the procedures they describe. Thus intentions can have intentions.

Because protection is an intrinsic property of actors, we hope to be able to deal with protection issues in the same straight forward manner as more conventional intentions.

Intentions of data structures are handled by the same machinery as for all other actors.

Syntactic Sugar

"What's the good of Mercator's North Poles and Equators, Tropics,
Zones and Meridian Lines?"
So the Bellman would cry: and the crew would reply
"They are merely conventional signs!"
Lewis Carroll

Thus far in our discussion we have discussed the semantic issues intuitively but vaguely. We would now like to proceed with more precision. Unfortunately in order to do this it seems necessary to introduce a formal language. The precise nature of this language is completely unimportant so long as it is capable of expressing the semantic meanings we wish to convey. For some years we have been constructing a series of languages to express our evolving understanding of the above semantic issues. The latest of these is called PLANNER-73.

Meta-syntactic variables will be underlined. We shall assume that the reader is familiar with advanced pattern matching languages such as SNOBOL4, CONVERT, PLANNER-71, QA4, and POPLER.

We shall use (%A M%) to indicate sending the message M to the actor A. We shall use [s1 s2 ... sn] to denote the finite sequence s1, s2, ...sn. A sequence s is an actor where (%s i%) is element i of sequence s. For example (%[a c b] 2%) is c. We will use ( ) to delimite the simultaneous synchronous transmission of more than one message so that (A1 A2 ...An) will be defined to be (%A1 [A2 ... An]%). The expression [%a1 a2 ... an%] (read as "a1 then a2 ... finally send back an") will be evaluated by evaluating a1, a2,..., and an in sequence and then sending back ["returning"] the value of an as the message.

Identifiers can be created by the prefix operator =. For example if the pattern =x is matched with v, then a new identifier is created and bound to v.

"But 'glory' doesn't mean ' a nice knock-down argument,'" Alice objected.

"When I use a word, "Humpty Dumpty said, in rather a scornful tone, "it means just which I choose it to mean-neither more nor less."

"The question is," said Alcie, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master-that's all."

Lewis Carroll

Humpty Dumpty propounds two criteria on the rules for names:

Each actor has complete control over the names he uses.

All other actors must respect the meaning that an actor has chosen for a name.

We are encouraged to note that in addition to satisfying the criteria of Humpty Dumpty, our names also satisfy those subsequently proposed by Bill Wulf and Mary Shaw:

The default is not necessarily to extend the scope of a name to any other actor.

The right to access a name is by mutual agreement between the creating actor and each accessing actor.

An access right to an actor and one of its acquaintances is decoupled.

It is possible to distinguish different types of access.

The definition of a name, access to a name, and allocation of storage are decoupled.

The use of the prefix = does not imply the allocation of any storage.

One of the simplest kinds of ACTORS is a cell. A cell with initial contents V can be created by evaluating (cell V). Given a cell x, we can ask it to send back its content by evaluating (contents x) which is an abbreviation for (x #contents). For example (contents (cell 3)) evaluates to 3. We can ask it to change its contents to v by evaluating (x <- v). For example if we let x be (cell 3) and evaluate (x <- 4), we will subsequently find that (contents x) will evaluate to 4.

The pattern (by-reference P) matches object E if the pattern P matches (cell E) i.e. a "cell" [see below] which contains E. Thus matching the pattern (by-reference =x) against E is the same as binding x to (cell E) i.e. a new cell which contains the value of the expression E. We shall use => [read as "RECEIVE MESSAGE"] to mean an actor which is reminiscent of the actor LAMBDA in the lambda calculus. For example (=> =x body) is like (LAMBDA x body) where x is an identifier. An expression (=> pattern body is an abbreviation for (receive(message pattern) body) where receive is a more general actor that is capable of binding elements of the action in addition to the message.

Evaluating

(%(=> pattern body) the-message%), i.e. sending
(=> pattern body) the-message,

will attempt to match the-message against pattern. If the-message is not of the form specified by pattern, then the actor is NOT-APPLICABLE to the-message. If the-message matches pattern, the body is evaluated.

Evaluating (%(cases [f1 f2 ... fn]) arg%) will send f1 the message arg and if it is not applicable then it will send f2 the message arg, etc. until it findsone that is applicable. The message [#not - applicable] is sent back if none were applicable. Evaluating (%(cases {f1 f2 ... fn}) arg%)will send f1 the message arg, ..., and send fn the message arg concurrently.

The following abbreviations will be used to improve readability:

```
(rules object clauses) for
    ((cases clauses) object)

(where object pattern-for-message body) for
    ((=> [pattern-for-message] body) object)
            ; for example (where  1 + 2)   x (x + 1)) is 4

(let
      {
       [x0 <= expression0]
       [x1 <= expression1]
       ...
       [xn <= expressionn]}
      body) for
  ((=> [=xo = x1 ... =xn] body)
      expressiono
      expression1
      ...
      expressionn)
            ; for example
                 (let
                     {
                        [x <= ( 2 + 1)]
                        [y <= ( 2 * 2)]}
                     ( x + y)) is 7
```

Sending Messages and Creating Actors

The world's a theatre, the earth a state,
Which God and nature do with actors fill.
Thomas Heywood 1612

Conceptually at least a new actor is created every time a message is sent. Consider sending to a target T a message M and a continuation C.

```
(send T
    (message M
        [#continuation C]))
```

The transmission (%T M%) is an abbreviation for the above where C is defaulted to be the caller. If the target T is the following:

```
(receive
    (message the-body
        [#continuation =the-continuation])

    the-body)
```

then the-body is evaluated in an evironment where the-message is bound to M and the-continuation is bound to C.

We define an EVENT to be a quadruple of the form [C T M N] where C is the continuation of the caller,T the target, and M the message thereby creating a new actor N. We define a HISTORY to be a strict partial order of events with the transitive closure of the partial ordering ->[read as PRECEDES] where

[c1 t1 m1 n1] -> [c2 t2 m2 n2] if

{n1} intersect {c2 t2 m2} is nonvoid

The above definition states that one action proecedes another if any of the actors generated by the first event are used in the second event. The relation -> can be thought of as the "arrow of time." Notice that we do not require a definition of global simultaneity; i.e. we do not require the two arbitrary events be related by ->. We define the BEHAVIOR of a history with respect to an AUDIENCE [a set of actors] E to be the subpartial ordering of the history consisting of those quadruples [C T M N] where C or T is an element of the audience E. The REPERTOIRE of a configuration of actors is the set of all behaviors of the configuration for all interpretations of the actors in the audience. The REPERTOIRE of a configuration defines what the configuration does as opposed to how it does it. Two configurations of actors will be said to be EQUIVALENT if they have the same REPERTOIRE

We can name an actor H with the name A in the body B by the notation (label {[A <= H]} B). More precisely, the behavior of the actor (label {[f <= (E f)]} B) is defined by the MINIMAL BEHAVIORAL FIXED POINT of (E F) i.e. the minimal repertoire F such that (E F) = F. In the case where F happens to define a function, it will be the case that the repertoire F is isomorphic with the graph [set of ordered pairs] of the function defined by F and that the graph of F is also the least (lattice-theoretic) fixed point of Park and Scott.

Many happy returns

Many actors who are executing in parallel can share the same continuation. They can all send a message ["return"] to the same continuation. This property of actors is heavily exploited in meta-evaluation and synchronization. An actor can be thought of as a kind of virtual processor that is never "busy" [in the sense that it cannot be sent a message].

The basic mechanism of sending a message preserves all relevent information and is entirely free of side effects. Hence it is most suitable for purposes of semantic definiton of special cases of invocation and for debugging situations where more information needs to be preserved. However, if fast write-once optical memories are developed then it would be suitable to be implemented directly in hardware.

The following is an overview of what appears to be the behavior of the process of a running actor R sending a target T the message M specifying C as the continuation. If C is not explicitly specified by R then a representative of R must be constructed as the default.

1: Call the banker of R to approve the expenditure of resouces by the caller.

2: The banker will probably eventually send a message to the scheduler of T.

3: The scheduler will probably eventually send a message to the monitors of T.

4: The monitors will probably eventually send a message to the intentions of T.

5: The intentions of T will probably eventually send the message M to T.

6: T will finally attempt to get some real work done.

There are several important things to know about the process of sending a message to an actor:

1: Conceptually at least, whenever a target is passed a message a new actor is constructed which is the target instantiated with a message. When ever possible we reuse old actors where the reuse cannot be detected by the behavior of the system.

2: Sending messages between actors is a universal control primitive in the sense that control operations such as function calls, iteration, coroutine invocations, resource seizures, scheduling, synchronization, and continuous evaluation of expressions are special cases.

3: Actors can conduct their dialogue directly with each others: they do not have to set up some intermediary such as ports [Krutar, Balzer, and Mitchell] or possibility lists [McDermott and Sussman] which act as pipes through which conversations must be conducted.

4: Sending a message to an actor is entirely free of side effects such as those in the message mechanism of the current SMALL TALK machine of Alan Kay, the port mechanism of Krutar and Balzer, and possibility lists. Being free of side effects allows us a maximum of parallelism and allows an actor to be engaged in several conversations at the same time without becoming confused.

5: Sending a message to an actor makes no presupposition that the actor sent the message will ever send back a message to the continuation. The unidirectional nature of sending messages enables us to define iteration, monitors, coroutines, etc. straight forwardly.

6: The ACTOR model is not an [environment-pointer, instruction-pointer] model such as the CONTOUR model. A continuation is a full blown actor [with all the rights and privileges]; it is not a program counter. There are no instructions [in the sense of present day machines] in our model. Instead of instructions, an actor machine has certain primitive actors built in hardware.

Static Data Structures

Data structures are special cases of ACTORS. For example consider the following definition of the list nil:

```
[nil <=
    (cases
        {(=> [#out =stream]
                ;"this is a comment"
                ;"to print nil: print the
                string '(list)' to stream"
            (out stream
                (print-open "(")
                (print-string "list")
                (print-close ")")))

         (=> [#empty?]
                ;"it is empty"
            true)
         (=> [#equivalent =x =overlord =the-complaint-dept]
            (rules nil
                [(=> x
                    true)
                (else
                        (not-equal the-complaint-dept))]))
          (=> [#structure?]
                ;"it is a structure"
             true)
          ( => [#next =the-complaint-dept]
             (exhausted the-complaint-dept))
```

We also define the function output:

```
[output <=
    (=> [=x =stream]
        ( x #out stream ))]
```

The above is an operational definition of nil which is the null list. For example (nil #structure?) is true. Evaluating (output nil s) will cause "(list)" to be printed to the stream s. However from an operational point of view nil is not very interesting because it is completely static. What we need to ask our-selves is what are the useful modes of behavior that are embodied in the usual notion of a list structure and define an object which behaves in this way. So let us try to give an operational definition of an arbitrary list: In order to do this we need to be able to make changes in the world. We will use the primitive actor CELL to realize these changes.

<u>Definition of LISP-like List Structure</u>

```
[cons-list <=
   (=> [(by-reference =first-of-list)
        (by-reference =rest-of-list)]
        (cases
            {(=>  [#first]
                      ;"the first element  of
                      the list is contents of first-of-list"
                 (contents first-of-list)))
            (=> [#rest]
                     ;"the rest is contents of rest-of-list"
                (contents rest-of-list))
            (=> [#first <- =new-first]
                (first-of-list <- new-first))
            (=> [#rest <- =new-rest]
                (rest-of-list <- new-rest))
            (=> [#constructor]
                     ;"a constructor for this
                         kind of behavior is list"
                list)
            (=> [#next =the-complaint-dept]
                (stream
                    (contents first-of-list)
                    (contents rest-of-list)))
            (=> [#equivalent =x =overlord =the-complaint-dept]
                [%
                    (overlord
                        (first x)
                        (contents first-of-list)
                        the-complaint-dept)
                    (overlord
                        (rest x)
                        (contents rest-of-list)
                        the-complaint-dept)%])
            (=> [#out =the-customer]
                (out the-customer
                        ;"to print the list first print open-delimiter ("
                    (print-open "(")
                        ;print that it is a list"
                    (print-string "list")
                        ;"print the first element"
                    (print (contents first-of-list))
                        ;"print the rest of the elements in the list"
                    (print-elements (contents rest-of-list))
                        ;"the function print-elements is defined below"
                        ;"print the close )"
                    (print-close ")")))
            (=> [structure?]
                true)
            (=> [empty?]
                false)}))]
```

The above definition is much more interesting. For one thing there is a subtle bug in that if cons-list is implemented as a lambda calculus closure then it will hang onto too much storage since any actor which hangs onto a piece of list structure will hang onto the creator of that list structure. We will deal with this bug later. But let's see how it works anyway: Let x be (cons-list 6 nil). Thus x is an instantiated CASES statement in the definition of CONS-LIST with first-of-list the name of a new cell which contains 6 and rest-of-list the name of a new cell which contains nil.

```
now ( x #first) evaluates to 6.
but suppose we execute ( x #first <- B) causing
        [#first <- B] to be matched against
        the patterns in the CASES till it matches
        [#first <- =new-first]
now (x #first) evaluates to B.
```

The reason is that there is a side effect in the evaluation of
(x #first <- B) which changes the first element of x to B.

We can define a function which will print the elements of objects which behave like lists as follows:

```
[print-elements <=
    (=> [=supply =send-to]
        (next
            supply
                ;"else let =element be the next element and
                    =remainder-of-supply be the remainder of the supply"
            (=>
                (stream  =element =remainder-of-supply)

                    (out send-to
                        (print element)
                        (print-elements remainder-of-supply))
            (=> [#exhausted]
                    ;"if the supply is exhausted, do nothing"
                nothing))))]
```

The function next calls up the supply and asks it for the "next".

```
[next <=
    (=> [=the-supply =the-customer =the-complaint-dept]
        (the-customer
            (the-supply #next the-complaint-dept)))]
```

Note that to get the second [and subsequent] elements out of a stream s the continuation received by n for (next s n) must be used.

Let w be (cons-list 3 (cons-list 4 nil)). The following expression will create a circular structure when evaluated:

```
(w  #rest <- w)
(output w s)
```

The printing will look like "(list 3 3 3 3 ... to s and will never cease. It will never get to print the ")".

The reader might be puzzled why we proceed in this "backward" way. Why don't we write a FUNCTION rest which takes the rest of a list like any ordinary programming language does? For example

```
(rest
    (cons-list
        3
        (cons-list B nil)))
```

would be "(list B)". People who have taken the approach of attempting to define such functions have come to realize that it is desireable to have some independence in the representation of data objects so they have tried to define REST as a "polymorphic" operator. This means for example that REST would attempt to operate on vectors as well as lists. But then in any modeling situation in which a kind of object is desired for which we would like to be able to compute the REST, the extrinsic functional definition of REST would have to change. The definition of REST must keep changing in a nonmodular way in order to add new knowledge. For example we might create strings and want to be able to take the REST of a string. Of course the following definitions of REST and FIRST as functions will work:

```
[rest <=
    (=> [=z]
        (z #rest))]

[first <=
    (=> [=z]
        (z #first))]
```

These are in fact the definitions that we use. Note that we have two semantically related names: #REST and REST. We use #REST as a message and REST for the function which sends the message # REST to its argument. Making the above definitions of FIRST and REST and using them instead of directly passing the messages #first and #rest does, however, increase the modularity of our formalism and so we shall adopt them. For example the definitions of FIRST and REST enable us to monitor these operations.

The reason that we have discussed the actor cons-list in such great detail is that it provides a paradigm for the way in which we will define actors in general. For example our definition of EVAL [the actor which evaluates forms] is:

```
[eval <=
    (=>
        [=x =the-environment]
        (x #eval the-environment))]
```

In other words EVAL passes the buck to each kind of expression which is expected to either know how to evaluate itself or to further delegate the responsibility.

There remains the problem of dividing up the responsibility and knowledge in a reasonable way. At this point we have only a few heuristics to offer. We hope to become more definitive as we gain more practical experience with actors. In general we program each actor to field those requests for which it feels most qualified because the information needed is most immediately at hand. For example we have not included #length among messages fielded by list-structures but rather have preferred to write:

```
[length <=
   (=> [=the-supply]
        (send
            the-supply
            (message [#length]
            [#alternate
                (=> [#not-applicable]
                    (next
                        the-supply
                        (=>
                            (stream ? =the-remaining-supply)

                                ;"the answer is"
                            (1 + (length the-remaining-supply)))
                        (=> [#exhausted]
                                ;"if the supply is exhausted then 0"
                            0)))]
```

There is a complete duality between operands and operators in the actor formalism. In many cases the precise organization seems more a matter of taste then anything else.

The data type cons-list is the class of all actors that have the behavior defined above. Certain properties of the data type can be derived immediately from the definition. For example

```
(where (cons-list x y) =z
    (first z) is x)

(where (cons-list x y) =z
    (rest z) is y)

(where (cons-list x y) =z
    (z #first <- x') is z
    and z is equal to (cons-list x' y))

(where (cons-list x y) =z
    (z #rest <- y') is z
    and z is equal to (cons-list x y'))
```

McCarthy has given the above formulas as axioms for lists. In his system the data type list is the class of all structures that obey the above axioms. However if nothing is known about the actor dragons then

```
(where (cons-list x y) =z
    [%
        (dragons z)
        (first (z #first <- x')) is unknown!%])
```

The reason is that dragons may have swallowed the list z and passed it to some actor which is still acting concurrently. Thus we don't know that the first of z is x' even though we just stored x' there!

Now any object which behaves like a list can be used in place of a list. For example we can construct an object which is indistinguishable from an arbitrary list Z except that it will print out whenever its first element is changed. To do this we will give a general definition of a monitor.

Monitors

Every actor can have monitors which get to read every message that is sent to the actor. Monitors are mainly useful for metering and debugging. A monitor can be constructed by

(cons-monitor _pattern_ _in-going-action_ _out-going-monitor_) where pattern is the specification of the in going message, _in-going-action_ is what to do, and _out-going-monitor_ [which by the way is optional] an out going monitor.

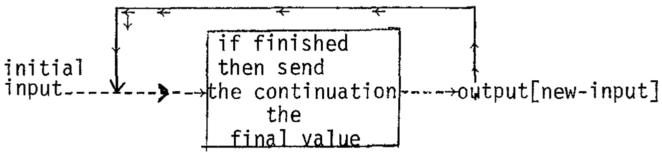
For example we can define a monitor for factorial that keeps adding one to the contents of number-of-calls-to-factorial every time that factorial is called and prints out the [input output] pairs on the stream history-of-factorial for each call.

```
[monitor-for-factorial <=
    (cons-monitor
        (message =input)
        (number-of-calls-to-factorial
            <-
                (1 + (contents (number-of-calls-to-factorial))))
        (cons-monitor
            (message =output)
            (out history-of-factorial
                 (print [input output])))))]
```

The system actor NEW-MONITOR is used to install a new monitor in an actor. For example (new-monitor factorial monitor-for-factorial) installs monitor-for-factorial as a new monitor for factorial. After which if (factorial 3) is evaluated then the contents number-of-calls-to-factorial will be increased by 3 and the stream history-of-factorial will be sent "[[1] 1]" then "[[2] 2]", and finally "[[3] 6]".

Iteration

Iteration is a special case of sending messages to oneself. We envisage a finite state machine with inputs on one side and outputs on the other.



The iteration statement is due to Nick Pippinger and has the syntax:

```
(iterate name
         initial-input
         definition)
```

For example an iterative factorial program can be written as:

```
[iterative-factorial <=
    (=> [=n]
        (iterate counting-up
            [1 1]
            (=> [=counter =accumulator]
                (rules counter
                    [(=> n accumulator)
                    (else
                        (counting-up
                            (counter + 1)
                            (counter * accumulator)))]))))]
```

Notice that there are no assignment statements in the above program!
The behavior of iterative-factorial is the same as if it were defined as follows:

```
[iterative-factorial <=
   (= > [=n]
        (label
            {[counting-up <=
                (receive
                    (message [=counter =accumlator]
                        [#continuation =c])
                    (rules counter
                        [(=> n
                            accumlator)
                        (else
                            (send
                                counting-up
                                (message
                                    [
                                        (counter + 1)
                                        (counter * accumulator )]
                                    [#continuation c]))))]))]}
           (counting-up 1 1)))]
```

We use

```
(cycle name
    body)
```

as an abbreviation for

```
(iterate name
    []
    (=> [] body))
```

Meta-Evaluation

Meta-evaluation is the process of binding actors to their intentions and then evaluating the actors abstractly on abstract data. Using actor induction we will show that if the meta-evaluation of a configuration of actors succeeds then the intentions of the actors will be satisfied in the subsequent execution of the configuration. If the meta-evaluation cannot proceed it will stop at the point where it cannot confirm that an actor always satisfied its intention and ask for help. At this point there are several possibilities:

There really is an inconsistency:

The inconsistency is between the way the actor is being attempted to be used and its intention.

The inconsistency is between the intention of the actor and its actual implementation.

The intentions for a configuration of actors are not mutually consistent.

There is no inconsistency but:

There are hidden assumptions being made about actors that should be made explicit.

There is hidden domain dependent knowledge that the actor is using which should be made explicit.

The intentions are not being sufficiently explicit as to why they are expected to be satisfied.

Convergence of Actors

Meta-evaluation can be used to show that certain inputs must eventually generate outputs. The basic technique is the principle of induction over well-founded partial orders invented by mathematicians and elegantly formalized by John von Neumann. The technique is a special case of actor induction. At present, actor induction for an actor configuration with input audience I and output audience O can be tentatively described in the following manner:

1. There is a well-founded input-output partial order P. That is, there is no sequence s of distinct elements of P such that for every i we have (s i) {P}(s (i + 1)).

2. The actors in the input audience I assign an element p of P to each input message m. We will denote this by the notation P<m,p>.

3. For each actor A if P<m,p> is received by A, then A must send a message P<m',p'> such that p{P}p'.

Therefore

Every message from the input audience must eventually result in a message being sent to the output audience.

A Simple Example Illustrating How A Diligent But Moderately Dumb Apprentice Can Help

We would like to give a simple concrete example to illustrate our techniques in action. Consider the probelm of writing a program to shift the gears of a truck with a manual transmission. We apologize for the necessity for introducing new syntax but the following concepts are crucial to the discussion which follows:

1: Definitions

```
[x <=
    (=>  [=y]
         body)]
```

is actor syntax which at a rough intuititve level means: define an actor x which, when it is called with an argument (to which y is bound) executes body.

2: Rules

```
    (rules x
        (=> =y1 body1)
        (=> =y2 body2)
        ...
        ...)
```

roughly means: take x, and if it matches y1, execute body1; otherwise if it matches y2, execute body 2, etc...

3: Intentions

```
    [x <=
        (intention [n]
            i1
            definition
            i2)]
```

is an elaboration of 1, meaning that when x is called with n, then i1 is the intention of the incoming call and i2 is the intention when x calls out again.

Our first try at a shift procedure might be:

Primitive-shift-to: when called with a target gear checks to see if it is 1, 2, 3, or 4 and calls the appropriate select; upper-left, upper-right, or lower-right respectively.

```
[primitive-shift-to <=
    (=> [=target-gear]
        (rules target-gear
            (=> 1
              (select-upper-left))
            (=> 2
              (select-lower-left))
            (=> 3
              (select-upper-right))
            (=> 4
              (select-lower-right))))]
```

Now we consider the various select routines and their intentions. Each of the select functions has an incoming intention that the clutch be disengaged. Furthermore each of them has code (delimited by *) to do the selecting. When a selector calls out, we fully intend for the truck to be in the gear appropriate to that selection.

```
[select-upper-right <=
    (intention []
        (clutch disengaged)
        *code-for-select-upper-right*
        (in-gear 3))]

[select-upper-left <=
    (intention []
        (clutch disengaged)
        *code-for-select-upper-left*
        (in-gear 1))]


[select-lower-right <=
    (intention []
        (clutch disengaged)
         *code-for-select-lower-right*
         (in-gear 4))]

[select-lower-left <=
    (intention []
        (clutch disengaged)
        *code-for-select-upper-right*
        (in-gear 2))]
```

Our apprentice notices that for each one that there is a physical constraint that the clutch must be disengaged before shifting. He queries us abut this and so we decide to modify the function PRIMITIVE-SHIFT-TO to first disengage the clutch.

```
[primitive-shift-to <=
    (=> [=target-gear]
        (disengage clutch)
        (rules target-gear
            (=> 1
              (select-upper-left))
            (=> 2
              (select-lower-left))
            (=> 3
              (select-upper-right))
            (=> 4
              (select-lower-right))))]

         (engage clutch))]
```

Now the code for primitive-shift-to is to first disengage the clutch, then do the selecting as before, and finally engage the clutch.

We also write functions to disengage and engage the clutch.

```
[disengage <=
    (intention [=clutch]
        (clutch engaged)
        *code-for-disengage*
        (clutch disengaged))]

[engage <=
     (intention [=clutch]
         (clutch disengaged)
         *code-for-engage*
         (clutch engaged))]
```

Now our apprenctice is mollified. However, the engineers dealing with the transmission come to us with some additional constraints. For example to select third gear the constraints are now that the clutch must be disengaged and the truck must be in either second or fourth gear. The other constraints are similar.

```
[select-upper-right <=
    (intention
        (and
            (clutch disengaged)
            (or
                (in-gear 2)
                (in-gear 4)))
        *code-for-select-upper-right*
        (in-gear 3))]

[select-upper-left <=
    (intention
        (and
            (clutch disengaged)
            (stopped))
        *code-for-select-upper-left*
        (in-gear 1))]

[select-lower-right <=
    (intention
        (and
            (clutch disengaged)
            (in-gear 3))
        *code-for-select-lower-right*
        (in-gear 4))]

[select-lower-left <=
    (intention
        (and
            (clutch disengaged)
            (or
                (in-gear 1)
                (in-gear 3)))
```

```
        *code-for-select-lower-left*
        (in-gear 2))]
```

The new requirements say that (temporarily at least) the truck has to be stopped to shift into gear 1 and no gears can be skipped in shifting while running. (Note you can shift directly from any gear to first if the truck is stopped.) So we have to write some new procedures to meet these new intentions. We now write our top-level shifting function:

SHIFT-TO: when called with a target gear considers in order the following rules for the target gear:

> If it is first gear, then do a primitive-shift-to first gear.
>
> If it is either one greater than the current gear or one less than the current gear then do a primitive-shift-to the target gear.
>
> If it is greater than the current gear then shift-to one less than the target gear and then primitive-shift-to the target gear.
>
> If it is less than the current gear then shift-to one greater than the target gear and then primitive-shift-to the target gear.

```
[shift-to <=
    (=> [=target-gear]
        (rules target-gear
            (=> 1
                (primitive-shift-to 1))
            (=> (either
                    (current-gear + 1)
                    (current-gear - 1))
                (primitive-shift-to target-gear))
            (=> (greater (current-gear))
                (shift-to (target-gear-1))
                (primitive-shift-to target-gear))
            (=> (less (current-gear))
                (shift-to (target-gear + 1))
                (primitive-shift-to target-gear))))]
```

We ask our apprentice to meta-evaluate our program. It thinks for a while and sees two problems:

> It can only shift to gear 1 if the truck is stopped.
>
> It should not be asked to shift to the gear that it already is in. [the procedure shift-to does not work if it is asked to shift to the current gear.]

We decide to give the following intention to SHIFT-TO: If the target-gear is first gear then the truck must be stopped; otherwise the target-gear must be 2, 3, or 4 and not be the current gear.

```
[shift-to <=
    (intention [=target-gear]
        (rules target-gear
            (=> 1
                (stopped))
            (=> (or 2 3 4)
                (target-gear ≠ current gear))
            (else
                (not-applicable)))
         *code-for-repeatedly-shift-to*
        (in-gear target-gear))]
```

To summarize we have used intentions in the following somewhat distinct ways:

> As a <u>contract</u> that the actor has with its external environment. How it carries the <u>contract</u> is its own business.
>
> As a formal statement of the <u>conditions</u> under which the actor will fullfill its contract.

The above example does not deal with all of the computational issues that our apprentice will be faced with. For example it does not have sophisticated data structures and has no concurrency or parallelism. We deal with these problems in the technical report.

Acknowledgements

> "Everything of importance has been said before by somebody who did not discover it."
> Alfred North Whitehead

This research was sponsored by the MIT Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research. We would be very appreciative of any comments, criticisms, or suggestions that the reader might care to offer. Please address them to:

Carl Hewitt
Room 813
545 Technology Square
M.I.T. Artificial Intelligence Laboratory
Cambridge, Massachusetts 02139

The topics discussed in this paper have been under intense investigation by a large number of researchers for a decade. In this paper we have merely attempted to construct a coherent managable formalism that embraces the ideas that are currently "in the air".

We would like to acknowledge the help of the following colleagues: Bill Gosper who knew the truth all along: "A data structure is nothing but a stupid programming language". Alan Kay whose FLEX and SMALL TALK machines have influenced our work. Alan emphasized the crucial importance of using intentional definitions of data structures and of passing messages to them. This paper explores the consequences of generalizing the message mechanism of SMALL TALK and SIMULA-67; the port mechanism of Krutar, Balzer, and Mitchell; and the previous CALL statement of PLANNER-71 to a universal communication mechanism. Alan has been extremely helpful in discussions both of overall philosophy and technical details. Nick Pippenger for his very beautiful ITERATE statement and for helping us to find a fast economical decoding net for our ACTOR machine. John McCarthy for making the first circular definition of an effective problem solving formalism and for emphasizing the importance of the epistemological problem for artificial intelligence. Seymour Papert for his "little man" metaphor for computation. Allen Newell whose kernel approach to building software systems has here perhaps been carried to near its ultimate extreme along one dimension. David Marr whose penetrating questions led us to further discoveries. Rudy Krutar, Bob Balzer, and Jim Mitchell who introduced the notion of a PORT which we have generalized into an ACTOR. Robin Milner is tackling the problems of L-values and processes from the point of view of the lambda calculus. He has emphasized the practical as well as the theoretical implications of fixed point operators. Robin's puzzlement over the meaning of "equality" for processes led to our definition of behavior. Edsger Dijkstra for a pleasant afternoon discussion. Jim Mitchell has patiently explained the systems implementation language MPS. Jeff Rulifson, Bruce Adnerson, Gregg Pfister, and Julian Davies showed us how to clean up and generalize certain aspects of PLANNER-71. Peter Landin and John Reynolds for emphasizing the importance of continuations for defining control structures. Warren Teitleman who cleaned up and generalized the means of integrating editors and debuggers in higher level languages. Peter Landin, Arthur Evans, and John Reynolds for emphasizing the importance of "functional" data structures. Danny Bobrow and Ben Wegbreit who originated an implementation method that cuts down on some of the overhead. We have simplified their scheme by eliminating the reference counts and all of their primitives. C.A.R. Hoare is independently investigating "monitors" for data structures. Jack Dennis for sharing many of our same goals in his COMMON BASE LANGUAGE and for his emphasis on logical clarity of language definition and the importance of parallelism. Bill Wulff for our "." notation on the conventions of the values of cells and for being a strong advocate of exceptional cleanliness in language. Pitts Jarvis and Richard Greenblatt have given us valuable help and advice on systems aspects. Chris Reeve, Bruce Daniels, Terry Winograd, Jerry Sussman, Gene Charniak, Gordan Benedict, Gary Peskin, and Drew McDermott for implementing previous generations of these ideas in addition to their own. J.C.R. Licklider for emphasizing the importance of mediating procedure calls. Butler Lampson for the notion of a banker and and for the questions which led to our criteria for separating an actor from its base. Richard Wehyharauch for pointing out that logicians are also considering the possibility of procedural semantics for logic. He is doing some very interesting research in the much abused field of "computational logic". Terry Winograd, Donald Eastlake, Bob Frankston, Jerry Sussman, Ira Goldstein, and others who made valuable suggestions at a seminar which we gave at M.I.T. John Shockey for helping us to eradicate a infestation of bugs from this document. Greg Pfister, Bruce Daniels, Seymour Papert, Bruce Anderson, Andee Rubin, Allen Brown, Terry Winogard, Dave Waltz, Nick Horn, Ken Harrenstien, David Marr, Ellis Cohen, Ira Goldstein, Steve Zilles, Roger Hale, and Richard Howell made valuable comments and suggestions on previous versions of this paper.

Bibliography

Balzer, R. M., "Ports--A Method for Dynamic Interporgam Communication and Job Control" The Rand Coporation. 1971.

Bishop, Peter. "Data Types for Programming Generality" M.S. June 1972. M.I.T.

Bobrow D., and Wegbriet Ben. "A Model and Stack Implementation of Multiple Environments. March 1973

Burstall, R.M. "Proving Properties of Programs by Strucural Induction" Computer Journal Vol 12. pp. 41-48 (1969).

Courtois, P.J., Heymans, F. Parnass D.L. "Concurrent Control with 'READERS' and WRITERS'" Comm. ACM 14 10 pp667-668. Oct. 1971

Daniels, Bruce. "Automatic Generation of Compilers from Interpreters". M.S. Forthcoming 1973.

Davies, D.J.M. "POPLER: A POP-2 PLANNER" MIP-89. School of A-I. University of Edinburgh.

Deutsch L.P. "An Interactive Program Verifier" Phd. University of California at Berkley. June 1973. Forthcoming.

Earley, Jay. "Relational Level Data Structures for Programming Languages". March 1972.

Elcock, E.W.; Foster, J.M.; Gray, P.M.D.; MacGregor, H.H.; and Murrary A.M. Abset, a Programming Language Based on Sets: Motivation and Example. Machine Intelligence 6. Edinburg University Press.

Evans, A.E. "PAL--Pedagogic Algorithmic Language, A Reference Manual and Primer" Dept. of Electrical Engineering. M.I.T. 1970.

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

Floyd R. W. "Assigning Meaning to Programs" Mathematical Aspects of Computer Science. J.T. Schwarts (ed.) Vol 19. Am. Math. Soc. pp. 19-32. Pro-idence Rhode Island 1967.

Greif I.G. "Induction ti Proofs about Programs" Project MAC Technical Report 93. Feb, 1972.

Habermann A.N. "Synchronization of Communicating Processes" Comm. ACM 15 3 pp. 177-184. March 1970.

Hewitt, C. and Patterson M. "Comparative Schematology" Record of Project MAC Conference on Concurrent Systems and Parallel Compuation. June 2-5, 1970. Available from ACM.

Hewit, C., Bishop P., and Steiger, R. " A Universal Modular ACTOR Formalism for Artificial Intelligence" PLANNER Technical Report 3. December 1972. Revised March 1973 and June 1973.

Hewitt, C., Bishop P., and Steiger, R. "A Universal Modular Actor Formalism for Artificial Intelligence" IJCAI III. Stanford, Calif. Aug 1973 Forthcoming.

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

Kay, Alan C. Private Communication.

Krutar, R. "Conversational Systems Programming (or Program Plagiarism made Easy)" First USA-Japan Computer Conference. October 1972.

Landin, P.J. "The Next 700 Programming Languages" Comm. ACM 9. (March, 1966). pp 790-793.

Manna, Z.; Ness, S.; Vuillemin J. "Inductive Methods for Proving Properties of Programs" Proceeding of an ACM Conference on Proving Assertions about Programs" January 1972.

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"

McCarthy, J. "Definitions of New Data Types in ALGOL X" ALGOL Bulletin. OCT. 1964.

Milner, R. Private communication.

Mitchell, J.G. "A Unified Sequential Control Structure Model" NIC 16816. Forthcoming.

Morris J. H. "Protection in Programming Languages" CACM. Jan, 1973.

Naur. P. "Proffs of Programs by General Snapshots" BIT. 1967.

Newell, A.; Freeman P.; McCracken D.; and Robertson. G. "The Kernel Approach to Building Software Systems." Carneigie-Mellon University Computer Science Research Reviews. 1970-71.

Park, D. "Fixpoint Induction and Proofs of Program Properties" Machine Intelligence 5. Edinburgh University Press. 1969.

Patil, S. "Closure Properties of Interconnection of Determinate Systems" Project MAC Conference on Concurrent Systems and Parallel Processing. June 1970.

Perlis, A.J. "The Syntesis of Algorithmic Systems" JACM. Jan 1967.

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.

Rulifson Johns F., Derksen J.A., and Waldinger R. J. "QA4: A Procedural Calculus for Intuitive Reasoning" Phd. Stanford. November 1972.

Scott, D. "Data Types as Lattices" Notes. Amsterdam, June 1972.

Standish, T. "A Data Definition Facility for Programming Languages" Phd. Carnegie. 1967.

Steiger, R. "Actors". M.S. 1973. Forthcoming.

Waldinger R. Private Communication.

Wang A. and Dahl O. "Coroutine Sequencing in a Blcok Structured Environment" BIT 11 425-449.

Weyhrauch, R. and Milner R. "Programming Semantics and Correctness in a Mechanized Logic." First USA-Japan Computer Conference. October 1972.

Wulf W. and Shaw M. "Global Bariable Considered Harmful" Carnegie-Mellon University. Pittsburgh, PA. SIGPLAN Bulletin. 1973.