A Universal Modular ACTOR Formalism for Artificial Intelligence

Carl Hewitt
Peter Bishop
Richard Steiger

Abstract

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired in a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, or semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

"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 the 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:

FOUNDATIONS: The concept puts procedural semantics [the theory of how things operate] on a firmer basis. It will now be possible to do cleaner theoretical studies of the relation between procedural semantics and set-theoretic semantics such as model theories of the quantificational calculus and the lambda calculus.

LOGICAL CALCULAE: A procedural semantics is developed for the quantificational calculus. The logical constants FOR-ALL, THERE-EXISTS, AND, OR, NOT, and IMPLIES are defined as actors.

KNOWLEDGE BASED PROGRAMMING is programming in an environment which has a substantial knowledge base in the application area for which the programs are intended. The actor formalism aids knowledge based programming in the following ways: PROCEDURAL EMBEDDING of KNOWLEDGE, TRACING BEHAVIORAL DEPENDENCIES, and SUBSTANTIATING that ACTORS SATISFY their INTENTIONS.

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 fullfills 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 simple debugging of actors by the META-EVALUATION of actors to show that they satisfy their intentions. Suppose that there is an external audience of actors E which satisfy the intentions of the actors to which they send messages. Intuitively, the principle of ACTOR INDUCTION states that the intentions of all actions caused by E are in turn satisfied provided that the following condition holds:

```
If for each actor A
    the intention of A is satisfied =>
    that the intentions of all actors sent messages by A are satisfied.
```

Computational induction [Manna], structural induction [Burstall], and Peano induction are all 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 for example 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.

COMPARATIVE SCHEMATOLOGY: The theory of comparative power of control structures is extended and unified. The following hierarchy of control structures can be explicated by incrementally increasing the power of the message sending primitive:

iterative--→recursive--→backtrack--→determinate--→universal

EDUCATION: The model is sufficiently natural and simple that it can be made the conceptual basis of the model of computation for students. In particular it can be used as the conceptual model for a generalization of Seymour Papert's "little man" model of LOGO. The model becomes a cooperating society of "little men" each of whom can address others with whom it is acquainted and politely request that some task be performed.

LEARNING and MODULARITY: Actors also enable us to teach computers more easily because they make it possible to incrementally add knowledge to procedures without having to rewrite all the knowledge which the computer already possesses. Incremental extensions can be incorporated and interfaced in a natural flexible manner. Protocol abstraction [Hewitt 1969; Hart, Nilsson, and Fikes 1972] can be generalized to actors so that procedures with an arbitrary control structure can be abstracted.

EXTENDABILITY: The model provides for only one extension mechanism: creating new actors. However, this mechanism is sufficient to obtain any semantic extension that might be desired.

PRIVACY and PROTECTION: Actors enable us to define effective and efficient protection schemes. Ordinary protection falls out as an efficient intrinsic property of actors. The protection is based on the concept of "use". Actors can be freely passed out since they will work only for actors which have the authority to use them. Mutually suspicious "memoryless" subsystems are easily and efficiently implemented. ACTORS are at least as powerful a protection mechanism as domains [Schroeder, Needham, etc.], access control lists [MULTICS], objects [Wulf 1972], and capabilities [Dennis, Plummer, Lampson]. Because actors are locally computationally universal and cannot be coerced there is reason to believe that they are a universal protection mechanism in the sense that all other protection mechanisms can be efficiently defined using actors. The most important issues in privacy and protection that remain unsolved are those involving intent and trust. We are currently considering ways in which our model can be further developed to address these problems.

SYNCHRONIZATION: It provides at least as powerful a synchronization mechanism as the multiple semaphore P operation with no busy waiting and guaranteed first in first out discipline on each resource. Synchronization actors are easier to use and substantiate than semaphores since they are directly tied to the control-data flow.

SIMULTANEOUS GOALS: The synchronization problem is actually a special case of the simultaneous goal problem. Each resource which is seized is the achievement and maintenance of one of a number of simultaneous goals. Recently Sussman has extended the previous theory of goal protection by making the protection guardians into a list of predicates which must be re-evaluated every time anything changes. We have generalized protection in our model by endowing each actor with a scheduler. We thus retain the advantages of local intentional semantics. A scheduler actor allows us to program EXCUSES for violation in case of need and to allow NEGOTIATION and re-negotiation between the actor which seeks to seize another and its scheduler. Richard Waldinger has pointed out that the task of sorting three numbers is a very elegant simple example illustrating the utility of incorporating these kinds of excuses for violating protection.

RESOURCE ALLOCATION: Each actor has a banker who can keep track of the resources used by the actors that are financed by the banker.

STRUCTURING: The actor point of view raises some interesting questions concerning the structure of programming.

STRUCTURED PROGRAMS: We maintain that actor communication is well-structured. Having no goto, interrupt, semphore, etc. constructs, they do not violate "the letter of the law." Some readers will probably feel that some actors exhibit "undisciplined" control flow. These distinctions can be formalized through the mathematical discipline of comparative schematology [Patterson and Hewitt].

STRUCTURED PROGRAMMING: Some authors have advocated top down programming. We find that our own programming style can be more accurately described as "middle out". We typically start with specifications for a large task which we would like to program. We refine these specifications attempting to create a program as rapidly as possible. This initial attempt to meet the specifications has the effect of causing us to change the specifications in two ways:

1: More specifications [features which we originally did not realize are important] are added to the definition of the task.

2: The specifications are generalized and combined to produce a task that is easier to implement and more suited to our real needs.

IMPLEMENTATION: Actors provide a very flexible implementation language. In fact we are carrying out the implementation entirely in the formalism itself. By so doing we obtain an implementation that is efficient and has an effective model of itself. The efficiency is gained by not having to incur the interpretive overhead of embedding the implementation in some other formalism. The model enables the formalism to answer questions about itself and to draw conclusions as to the impact of proposed changes in the implementation.

ARCHITECTURE: Actors can be made the basis of the architecture of a computer which means that all the benefits listed above can be enforced and made efficient. Programs written for the machine are guaranteed to be syntactically properly nested. The basic unit of execution on an actor machine is sending a message in much the same way that the basic unit of execution on present day machines is an instruction. On a current generation machine in order to do an addition an add instruction must be executed; so on an actor machine a hardware actor must be sent the operands to be added. There are no goto, semaphore, interrupt, etc. instructions on an ACTOR machine. An ACTOR machine can be built using the current hardware technology that is competitive with current generation machines.

"Now! Now!" cried the Queen. "Faster! Faster!"
Lewis Carroll

Current developments in hardware technology are making it economically attractive to run many physical processors in parallel. This leads to a "swarm of bees" style of programming. The actor formalism provides a coherent method for organizing and controlling all these processors. One way to build an ACTOR machine is to put each actor on a chip and build a decoding network so that each actor chip can address all the others. In certain applications parallel processing can greatly speed up the processing. For example with sufficient parallelism, garbage collection can be done in a time which is proportional to the logarithm of the storage collected instead of a time proportional to the amount of storage collected which is the best that a serial processor can do. Also the architecture looks very promising for parallel processing in the lower levels of computer audio and visual processing.

"All the world's a stage,
And all the men and women merely actors.
They have their exits and their entrances;
And one man in his time plays many parts."

"If it waddles like a duck, quacks like a duck, and otherwise behaves like a duck; then you can't tell that it isn't a duck."

Adding and Reorganizing Knowledge

Our aim is to build a firm procedural foundation for problem solving. The foundation attempts to be a matrix in which real world problem solving knowledge can be efficiently and naturally embedded. We envisage knowledge being embedded in a set of knowledge boxes with interfaces between the boxes. In constructing models we need the ability to embed more knowledge in the model without having to totally rewrite it. Certain kinds of additions can be easily encompassed by declarative formalisms such as the quantificational calculus by simply adding more axioms. Imperative formalisms such as actors do not automatically extend so easily. However, we are implementing mechanisms that allow a great deal of flexibility in adding new procedural knowledge. The mechanisms attempt to provide the following abilities:

PROCEDURAL EMBEDDING: They provide the means by which knowledge can easily and naturally be embedded in processes so that it will be used as intended.

CONSERVATIVE EXTENSION: They enable new knowledge boxes to be added and interfaced between knowledge boxes.

MODULAR CONNECTIVITY: They make it possible to reorganize the interfaces between knowledge boxes.

MODULAR EQUIVALENCE: They guarantee that any box can be replaced by one which satisfies the previous interfaces.

Actors must provide interfaces so that the binding of interfaces between boxes can be controlled by knowledge of the domain of the problem. The right kind of interface promotes modularity because the procedures on the other side of the interface are not affected so long as the conventions of the interface are not changed. These interfaces aid in debugging since traps and checkpoints are conveniently placed there. More generally, formal conditions can be stated for the interfaces and confirmed once and for all.

Unification

We claim that there is a common intellectual core to the following (now somewhat isolated) fields that can be characterized and investigated: digital circuit designers, data base designers, computer architecture designers, programming language designers, computer system architects.

"Our primary thesis is that there can and must exist a single language for software engineering which is usable at all stages of design from the initial conception through to the final stage in which the last bit is solidly in place on some hardware computing system."
Doug Ross

The time has come for the unification and integration of the facilities provided by the above designers into an intellectually coherent manageable whole. Current systems which separate the following intellectual capabilities with arbitrary boundaries are now obsolete.

"Know thyself".

We intend that our actors should have a useful working knowledge of themselves. That is, they should be able to answer reasonable questions about themselves and be able to trace the implications of proposed changes in their intentions. It might seem that having the implementation understand itself is a rather incestuous artificial intelligence domain but we believe that it is a good one for several reasons. The implementation of actors on a conventional computer is a relatively large complex useful program which is not a toy. The implementation must adapt itself to a relatively unfavorable environment. Creating a model of itself should aid in showing how to create useful models of other large knowledge based programs since the implementation addresses a large number of difficult semantic issues. We have a number of experts on the domain that are very interested in formalizing and extending their knowledge. These experts are good programmers and have the time, motivations, and ability to embed their knowledge and intentions in the formalism.

"The road to hell is paved with good intentions."

Once the experts put in some of their intentions they find that they have to put in a great deal more to convince the auditor of the consistency of their intentions and procedures. In this way we hope to make explicit all the behavioral assumptions that our implementation is relying upon. The domain is closed in the sense that the questions that can reasonably be asked do not lead to a vast body of other knowledge which would have to be formalized as well. The domain is limited in that it is possible to start with a small superficial model of actors and build up incrementally. Any advance is immediately useful in aiding and motivating future advances. There is no hidden knowledge as the formalism is being entirely implemented in itself. The task is not complicated by unnecessary bad software engineering practices such as the use of gotos, interrupts, or semaphores.

Intrinsic Computation

We are approaching the problem from a behavioral [procedural] as opposed to an axiomatic approach. Our view is that objects are defined by their actors rather than by axiomatizing the properties of the operations that can be performed on them.

"Ask not what you can do to some actor;
but what the actor can [will?] do for you."

Alan Kay has called this the INTRINSIC as opposed to the EXTRINSIC approach to defining objects. Our model follows the following two fundamental principles of organizing behavior:

Control flow and data flow are inseparable.

Computation should be done intrinsically instead of extrinsically i.e. "Every actor should act for himself or delegate the responsibility [pass the buck] to an actor who will."

Although the fundamental principles are very general they have definite concrete consequences. For example they rule out the goto construct on the grounds that it does not allow a message to be passed to the place where control is going. Thus it violates the inseparability of control and data flow. Also the goto defines a semantic object [the code following the tag] which is not properly syntactically delimited thus possibly leading to programs which are not properly syntactically nested. Similarly the classical interrupt mechanism of present day machines violates the principle of intrinsic computation since it wrenches control away from whatever instruction is running when the interrupt strikes.

Hierarchies

The model provides for the following orthogonal hierarchies:

SCHEDULING: Every actor has a scheduler which determines when the actor actually acts after it is sent a message. The scheduler handles problems of synchronization. Another job of the scheduler [Rulifson] is to try to cause actors to act in an order such that their intentions will be satisfied.

INTENTIONS: Every actor has an intention which makes certain that the prerequisites and context of the actor being sent the message are satisfied. Intentions provide a certain amount of redundancy in the specifications of what is supposed to happen.

MONITORING: Every actor can have monitors which look over each message sent to the actor.

BINDING: Every actor can have a procedure for looking up the values of names that occur within it.

RESOURCE MANAGEMENT: Every actor has a banker which monitors the use of space and time.

Note that every actor has all of the above abilities and that each is done via an actor!

"A slow sort of country!" said the Queen. "Now here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

Lewis Carroll

The previous sentence may worry the reader a bit as she [he] might envisage an infinite chain of actions [such as banking] to be necessary in order to get anything done. We short circuit this by only requiring that it appear that each of the above activities is done each time an actor is sent a message.

"There's no use trying," she said: "one can't believe impossible things."

"I daresay you haven't had much practice," said the Queen. "When I was your age, I always did it for half-an-hour a day. Why, sometimes I've believed as many as six impossible things before breakfast."

Lewis Carroll

Each of the activities is locally defined and executed at the point of invocation. This allows the maximum possible degree of parallelism. Our model contrasts strongly with extrinsic quantificational calculus models which are forced into global noneffective statements in order to characterize the semantics.

"Global state considered harmful."

We consider language definition techniques [such as those used with the Vienna Definition Language] that require the semantics be defined in terms of the global computational state to be harmful. Formal penalties [such as the frame problem and the definition of simultaneity] must be paid even if the definition only effectively modifies local parts of the state. Local intrinsic models are better suited for our purposes.

Hardware

Procedural embedding should be carried to its ultimate level: the architecture of the machine. Conceptually, the only objects in the machine are actors. In practice the machine recognizes certain actors as special cases to save speed and storage. We can easily reserve a portion of the name space for actors implemented in hardware.

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, QA4, and PLANNER-71.

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 the sequence s. For example (%[a c b] 2%) is c. We will use ( ) to delimit 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 what I choose it to mean--neither more nor less."
"The question is," said Alice, "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 acquantances 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 contents 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, then 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,..., and send fn the message arg

The following abbreviations will be used to improve readability:

```
(rules object clauses) for
    ((cases clauses)object)
(let object pattern-for-message body) for
    (%(=> pattern-for-message body) object%)
```

```
                        ;for example (let 3=x ( x+1)) is 4
        (let-reception object pattern-for-reception body)
            (%(receive pattern-for-reception body) object%)
                        ;let is a special case of let-reception
```

Sending Messages and Creating Actors

The world's a theatre, the earth a stage,
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 a message to a target T with message M and continuation C.

```
        (send
            T
            {
                [#message the-body]
                [#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 M]
                [#continuation =the-continuation]}
            the-body)
```

then the-body is evaluated in an environment 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 precedes 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" which we require to be a strict partial order. Notice that we do not require a definition of global simultaneity; i.e. we do not require that two arbitrary events be related by →. We define 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 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 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 relevant information and is entirely free of side effects. *Hence it is most suitable for purposes of semantic definition* 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 resources 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 the continuation of T.
6: The continuation of 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:

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

1: Sending messages between actors is a universal control primitivein the sense that control operations such as function calls, iteration, coroutine invocations, resource seizures, scheduling, sychroniztion, and continous evaluation of expressions are special cases.

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

3: Sending a message to an actor is entirely free of side effects such as those in the message mechanisms of the current SMALL TALK machine of Alan Kay and the port mechanism of Krutat and Balzer. 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.

4: 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.

5: 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.

Logic

"It is behavior, not meaning, that counts."

We would like to show how actors represent formulas in the quantificational calculus and how the rules of natural deduction follow as special cases from the mechanism of extension worlds. We assume the existence of a function ANONYMOUS which generates a new name which has never before been encountered. Consider a formula of the form (every phi) which means that for every x we have that (phi x) is the case. The formula has two important uses: it can be asserted and it can be proved. We shall use an actor >=> [read as "ACCEPT REQUEST"] with the syntax

(>=> pattern-for-request body) for procedures to be invoked by pattern directed invocation by a command which matches pattern-for-request.

Our behavioral definitions are reminiscent of classical natural deduction except that we have four introduction and elimination rules [PROVE, DISPROVE, ASSERT, and DENY] to give us more flexibility in dealing with negation.

"Then Logic would take you by the throat, and force you to do it!"
Lewis Carroll

Data Bases

Data bases are actors which organize a set of actors for efficient retrieval. There are two primitive operations on data bases: PUT and GET. A new virgin data base can be created by evaluating (virgin). If we evaluate (w ← (virgin)) then (contents w) will be a virgin world. We can put an actor (at John airport) in the world (contents w) by evaluating (put(at John airport) {[#world(contents w)]}). We could add further knowledge by evaluating

(put (at airport Boston) {[#world (contents w)]}) to record that the airport is at Boston.

(put (city Boston) {[#world (contents w)]}) to record that Boston is a city.

If the constructor EXTENSION is passed a message then it will create a world which is an extension of its message. For example

```
(put
        [(on John (flight 34))
        (extension-world ← (contents w))])
```

will set extension-world to a new world in which we have supposed that John is on flight #34. The world (contents w) is unaffected by this operation. On the other hand the extension world is affected if we do (put [(hungry John) (contents w)]). Extension worlds are very good for modeling the following:

WORLD DIRECTED INVOCATION

The extension world machinery provides a very powerful invocation and parameter passing mechanism for procedures. The idea is that to invoke a procedure, first grow an extension world; then do a world directed invocation on the extension world. This mechanism generalizes the previous pattern directed invocation of PLANNER-67 several ways. Pattern directed invocation is a special case in which there is just one assertion in the wish world. World Directed Invocation represents a formalization of the useful problem solving technique known as "wishful thinking" which is invocation on the basis of a fragment of a micro-world. Terry Winograd uses restriction lists for the same purpose in his thesis version of the blocks world. Suppose that we want to find a bridge with a red top which is supported by its left-leg and its right-leg both of which are of the same color. In order to accomplish this we can call upon a genie with our wish as its message. The genie uses whatever domain dependent knowledge it has to try to realize the wish.

```
(realize
    (utopia
        {top left-leg right-leg color-of-legs}
            ;"the variables in the uptopia are listed above"
        {
            (color top red)
            (supported-by top left-leg)
            (supported-by top right-leg)
            (left-of left-leg right-leg)
            (color left-leg color-of-legs)
            (color left-leg color-of-legs)}))
```

LOGICAL HYPOTHETICALS are logically possible alternatives to a world.

By the Normalization Theorem for intuitionistic logic our actor definition of the logical constant IMPLIES is sufficient to mechanize logical implication. The rules of natural deduction are a special case of our rules for extension worlds and our procedural definition of the logical connectives.

ALTERNATIVE WORLDS are physically possible alternatives to a world.

PERCEPTUAL VIEWPOINTS can be mechanized as extension worlds. For example suppose rattle-trap is the name of a world which describes my car. Then (front rattle-trap) could be a world which describes my car from the front and (left rattle-trap) can be the description from the left side. We can also consider a future historian's view of the present by (view-from-1984 world-of-1972). Minsky [1973] considers these possibilities from a somewhat different point of view.

The following general principles hold for the use of extension worlds:

Each independent fact should be a separate assertion. For example to record that "the banana banl is under the table tabl" we would assert:

```
        (banana banl)
        (table tabl)
        (under banl tabl)
```

instead of conglomerating [McDermott 1973] them into one assertion:

```
        (at
            (the banl (is banl banana))
            (place
                (the tabl (is tabl table))
                under))
```

A person knowing a statement can be analyzed into the person believing the statement and the statement being true. So we might make the following definition of knowing:

```
    [know <=
        (=> [= person = statement]
            (and
                (believes person statement)
                (true statement)))]
```

Thus the statement [Moore 1973] "John knows Bill's phone number" can be represented by the assertion:

```
    (knows John (phone-number Bill pn0005))
```

where pn0005 is a new name and (phone-number Bill pn0005) is intended to mean that the phone number of Bill is pn0005. The assertion can be expanded as follows:

```
    (believes John (phone-number Bill pn0005))
    (true (phone-number Bill pn0005))
```

However the expansion is optional since the two assertions are not independent of the original assertion.

"Whatever Logic is good enough to tell me is worth writing down," said the Tortoise. "So enter it in your book, please."
Lewis Carroll

Each assertion should have justifications [derivations] which are also assertions and which therefore ...

Extraneous factors such as time and causality should not be conglomerated [McDermott 1973] into the extension world mechanism. Facts about time and causality should also be separate assertions. In this way we can deal more naturally and uniformly with questions involving more than one time. For example we can answer the question "How many times were there at most two cannibals in the boat while the missionaries and cannibals were crossing the river?" Also we can check the consistency of two different narratives of overlapping events such as might be generated by two people who attended the same party. Retreival of actors from data bases takes facts about time and causality into account in the retreival. Thus we still effectively avoid most of the frame problem of McCarthy. The ability to do this is enhanced by the way we define data bases as actors.

A CONTEXT mechanism was invented for QA4 to generalize the property list structure of LISP. Rulifson explained it by means of examples of its use to mechanize identifiers. By use of the functions PUSH-CONTEXT and POP_CONTEXT and an EPAM discrimination net [Feigenbaum and Simon] the context mechanism can be used to mechanize a version of tree-structured worlds. The tree-structured worlds of PLANNER-71 were invented to get around the problem of having only one global data base not realizing that a context mechanism could be used to implement something like that. The tree-structured worlds were defined directly in terms of the hash-coding mechanism of PLANNER which had the advantage of decoupling them from the identifier structure of PLANNER. In addition by not conceiving an extension world analogue of POP_CONTEXT large gains in efficiency over the context mechanism are possible.

Worlds can ask the actors put in them to index themselves for rapid retreival. We also need to be able to retrieve actors from worlds. Simple retrieval can be done using patterns. For example

```
(locations ← (get (at (?) (?)){[#world (contents w)]}))
```

will set locations to an actor which will retrieve all the actors stored in (contents w) which match the pattern (at (?) (?)). Now (next locations) will thus retrieve either (at airport Boston) or (at John airport). Actually, the above is an over simplification. We shall let $reality stand for the current world at any given point and $utopia stand for the world as we would like to see it. We do not want to have to explicity store every piece of knowledge which we have but would like to be able to derive conclusions from what is already known: We can distinguish several different classes of procedures for deriving conclusions.

"McCarthy is at the airport." (put (at McCarthy airport)) If a person is at the airport, then the person might take a plane from the airport.

```
  [put-at <=
      (>=> (put (at = person airport))
          (put (might (take-plane-from person airport))))]
```

"McCarthy is not at the airport." (deny (at McCarthy airport)) If a person is not at th airport then he can't take a plane from the airport.

"McCarthy is not at the airport." (deny (at McCarthy airport)) If a person is not at the airport then he can't take a plane from the airport.

```
        [deny-at<=
            (>=> (deny (at =person airport))
                (put (can't (take---plane---from person airport ))))]
```

"It is not known whether McCarthy is at the airport." (erase (at McCarthy airport)) If it is not known whether a person is at the airport then erase whatever depends on previous knowledge that the person is at the airport.

```
        [erase-at <=
            (>=> (erase (at =person airport))
                (find (depends---on =s (at person airport))
                    (erase s)))]
```

"Get McCarthy to the airport." (achieve {(at McCarthy airport)}) To achieve a person at a place:

```
        Find the present location of the person.
        Show that it is walkable from the present location to the car.
        Show that it is drivable from the car to the place.
    [achieve-at <=
        (>=> (achieve [(at =person =place )])
            (achieve
                (find [(at person =present-location)]
                    (show  {(walkable present-location car)}
                        (show {(drivable car place)}))))))]
```

"Show that McCarthy is at the airport." (show {(at McCarthy airport)}) To show that a thing is at a place show that the thing is at some intermediate and the intermediate is at the place.

```
        [show-at <=
            (>=> (show {(at =thing =place)})
                (show {(at thing =intermediate)}
                    (show {(at intermediate place)}))))]
```

The actor show-at is simply transitivity of at.

Is Anything Really Better Than Anything Else?

CONNIVER can easily be defined in terms of PLANNER-73. We do this not because we believe that the procedures of CONNIVER are particularly well designed. Indeed we have given reasons above why these procedures are deficient. Rather we formally define these procedures to show how our model applies even to rather baroque control structures.

CONNIVER is essentially the conglomeration of the following ideas: Landin's non-hierarchical goto-71, the pattern directed construction, matching, retrieval, and invocation of PLANNER, Landin's streams, the context mechanism of QA4, and Balzer's and Krutar's ports.

In most cases, two procedures in CONNIVER do not talk directly to each other but instead are required to communicate through an intermediary which is called a possibilities list. The concept of a POSSIBILITIES LIST is the major original contribution of CONNIVER.

"What are these
So wild and withered in their attire,
That look not like the inhabitants
O' the earth,
and yet are on't?"

Macbeth: Act I, Scene III

Substitution, Reduction, and Meta-evaluation

"One program's constant is another program's variable."

Alan Perlis

"Programming [or problem solving in general] is the judicious postponement of decisions and commitments!"

Edsger W. Dijkstra [1969]

"Programming languages should be designed to suppress what is constant and emphasize what is variable."

Alan Perlis

"Each constant will eventually be a variable!"

Corollary to Murphy's Law

We never do unsubstitution [or if you wish decompilation, unsimplification, or unevaluation]. We always save the higher level language and resubstitute. The metaphor of substitution followed by reduction gives us a macroscopic view of a large number of computational activities. We hope to show more precisely how all the following activities fit within the general scheme of substitution followed by reduction:

EVALUATION [Church, McCarthy, Lnadin] can be done by substituting the message into the code and reducing [execution].

DEDUCTION [Herbrand, Godel, Heyting, Prawitz, Robinson, Hewitt, Weyhrauch and Milner] can be done by procedural embedding. In this paper we have extended our previous work by defining the logical constants to be certain actors thus providing a procedural semantics for the quantificational calculus along the lines indicated by natural deduction.

CONFIRMING the CONSISTENCY of ACTORS and their INTENTIONS [Naur, Floyd, Hewitt

1971, Waldinger, Deutsch] can be done by substituting the code for the actors into **their** intentions and then meta-evaluating the code.

AUTOMATIC ACTOR GENERATION. An important corollary of the Thesis of Procedural Embedding is that the Fundamental Technique of Artificial Intelligence is automatic programming and procedural knowledge base construction. It can be done by the following methods:

PARAMETERIZATION [Church, McCarthy, Landin, McIntosh, Manna and Waldinger, Hewitt] of canned procedure templates.

COMPILATION [Lombardi, Elcock, Fikes, Daniels, Wulff, Reynolds, and Wegbreit] can be done by substituting the values of the free variables in the code and then reducing [optimizing]. For examples we can enhance the behavior of the lists which were behaviorally defined above to vectors which will run more efficiently on current generation machines.

ABSTRACT IMPOSSIBILITIES REMOVAL can be done by binding the alternatives with the code and deleting those which can never succeed. What we have in mind are situations such as having simultaneous subgoals (on a b) and (on b c) where we can show by meta-evaluation that the order given above can never succeed. Gerry Sussman has designed a program which attempts to abstract this fact from running on concrete examples. We believe that in this case and many others it can be abstractly derived by meta-evaluation.

EXAMPLE EXPANSION [Hart, Nilsson, and Fikes 1971; Sussman 1972; Hewitt 1971] can be done by binding the high level goal oriented language to an example problem and then reducing [executing and expanding to the paths executed] using world directed invocation [or some generalization] to create linkages between the variablized special cases.

PROTOCOL ABSTRACTION [Hewitt 1969, 1971] can be done by binding together the protocols, reducing the resulting protocol tree by identifying indistinguishable nodes.

ABSTRACT CASE GENERATION to distinguish the methods to achieve a goal can be done by determining the necessary pre-conditions for each method by reducing to a decision tree which distinguishes each method.

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 manageable 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 communications mechanism. Alan has been extremely helpful in discussions both of overall philosophy and technical details. Nick Pippenger for his very beautiful ITERATE statememt 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 enphasized 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 Anderson, 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. Todd Matson, Brian Smith, Irene Grief, and Henry Baker are aiding us in the implementation. Chris Reeve, Bruce Daniels, Terry Winograd, Jerry Sussman, Gene Charniak, Gordon 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 for the question which led to our criteria for separating an actor from its base. Richard Weyhrauch 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 Shockley for helping us to eradicate an infestation of bugs from this document. Greg Pfister, Bruce Daniels, Seymour Papert, Bruce Anderson, Andee Rubin, Allen Brown, Terry Winograd, 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 Interprogram Communication and Job Control"The Rand Corp., 1971.

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

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

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

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

Earley, Jay. "Toward an Understanding of Data Structures" Computer Science Department, University of California, Berkeley.

Elcock, E.W.; Foster, J.M.; Gray, P.M.D.; McGregor, H.H.; and Murray A.M. Abset, a Programming Language Based on Sets: Motivation and Examples. Machine Intelligence 6. Edinburgh, University Press.

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

Gentzen G. "Collected Papers of Gerhard Gentzen".North Holland. 1969.

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

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

Hewitt, C., Bishop P., and Steiger R. "The Democratic Ethos or 'How a Society of Noncoercable ACTORS can be Incorporated into a Structured System'" SIGPLAN-SIGOPS Interface Meeting, Savannah, Georgia. April, 1973.

Hewitt, C., and Greif,I. "Actor Induction and Meta-Evaluation"ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Boston, Mass. Oct. 1973. Forthcoming.

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

Kay, Alan C. Private Communication.

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

Lampson, B. "An Overview of CAL-TSS". Computer Center, University of California, Berkeley.

Liskov, B.H. "A Design Methodology for Reliable Software Systems" The Last FJCC. Dec.1972. Pt. 1, 191-199.

McDermott D.V. "Assimilation of New Information by a Natural Language-Understanding System" M.S. MIT. Forthcoming 1973.

McDermott, D.V. and Sussman G.J. "The Conniver Reference Manual" A.I. Memo no. 259. 1972.

Milner, R. Private Communication.

Minsky, Marvin. "Frame-Systems: A Theory for Representation of Knowledge" Forthcoming 1973.

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

Newell, A. "Some Problems of Basic Organization in Problem-Solving Programs." Self-Organizing Systems. 1962.

Papert S. and Solomon C. "NIM: A Game-Playing Program" A.I. Memo no. 254.

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.

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

Sussman, G.J. "Teaching of Procedures-Progress Report" Oct. 1972. A.I. Memo no. 270.

Waldinger R. Private Communication.

Wang A. and Dahl O. "Coroutine Sequencing in a Block 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.

Winograd, T. "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language" MAC TR-84. February 1971.

Wirth, N. "How to Live without Interrupts" or some such. Vol. 12 No. 9, pp. 489-498.

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