MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 727

April, 1983

Analyzing the Roles of Descriptions and Actions in Open Systems

Carl Hewitt
Peter de Jong

This paper analyzes relationships between the roles of descriptions and actions in large scale, open ended, geographically distributed, concurrent systems. Rather than attempt to deal with the complexities and ambiguities of currently implemented descriptive languages, we concentrate our analysis on what can be expressed in the underlying frameworks such as the lambda calculus and first order logic. By this means we conclude that descriptions and actions complement one another; neither being sufficient unto itself. This paper provides a basis to begin the analysis of the very subtle relationships that hold between descriptions and actions in Open Systems.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Major support for the research reported in this paper was provided by the System Development Foundation. Major support for other related work in the Artificial Intelligence Laboratory is provided, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505.

1. Open Systems

Problems and opportunities associated with describing and taking action in "open systems" will be increasingly recognized as a central line of computer system development. In the future many computer applications will be based on communication between systems which will have been developed separately and independently. These systems are open-ended and incremental--undergoing continual evolution. The only thing that the components of an open system hold in common is the ability to communicate with each other. In this paper we study description and actions in Open Systems from the viewpoint of Message Passing Semantics, a research programme to explore issues in the semantics of communication in parallel systems.

In an open system it becomes very difficult to determine what objects exist at any point in time. For example a query might never finish looking for possible answers. If a system is asked to find all the current mail address of people who have graduated from MIT since 1930, it might have a hard time answering. It can give all the mail address it has found so far, but there is no guarantee that another one can't be found by more diligent search. These examples illustrate how the "closed world assumption" is intrinsically contrary to the nature of Open Systems. We understand the "closed world assumption" to be that the information about the world being modeled is complete in the sense that *all and only* the relationships that can possibly hold among objects are those implied by the local information at hand (cf. [Reiter 82]). Systems based on the "closed world assumption" typically assume that they can find all the instances of a concept that exist by searching their local storage. In contrast we desire that systems be accountable for having evidence for their beliefs and be explicitly aware of the limits of their knowledge. At first glance it might seem that the closed world assumption, almost universal in the A.I. and database literature, is smart because it provides a ready default answer for any query. Unfortunately the default answers provided become less realistic as the Open System increases in size.

2. Message Passing Semantics

2.1. Descriptions of Behavior

Message Passing Semantics builds on Actor Theory as a foundation. Actor Theory describes important aspects of parallelism and serialization from the view point of message passing objects. It goes beyond the sequential coroutine message passing developed in systems like Simula and SmallTalk. An *actor system* is composed of abstract objects called *actors*. *Actors are defined by their behavior when they accept communications.* When a communication is accepted an actor can perform the following kinds of actions concurrently: make simple decisions (such as whether some actor it received in the communication is the same as one of its acquaintances), *create* new actors, *transmit* more communications to its own acquaintances as well as the acquaintances of the communication accepted, and change its behavior for the next message

accepted (i.e. change its local state) subject to the constraints of certain laws [Hewitt, Baker 77].

Below we describe the behavior of a simple actor which is a shared checking account which we will call ACCOUNT43. One kind of description might be a *partial* description of what happened when the actor ACCOUNT43 with a balance of $10 accepted a request to make a deposit with amount $2 for customer c2, and as a result created the actor $12, sent a Completion report to c2, and became an account with balance $12:

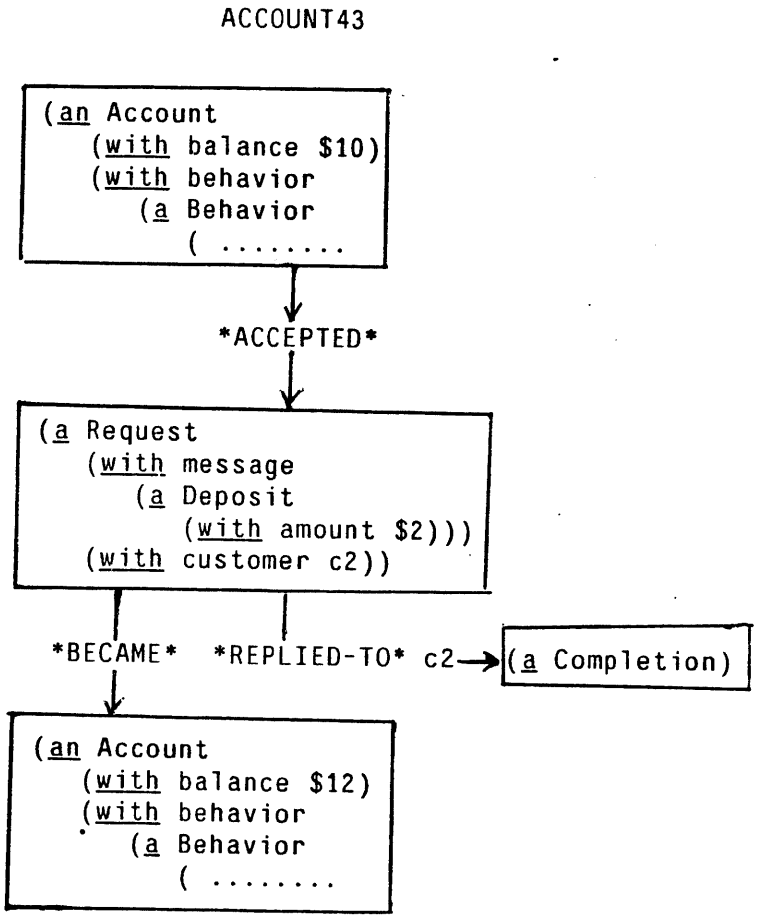

Figure 2-1: A Happening

Later in the paper we will describe how the behavior of an actor is written in the language Act2.

2.2. Shared Resources

The modeling of shared resources is fundamental to Open Systems. Actor systems aim to provide a clean way to implement *effects* (not "side-effects" a pejorative term that has been used as a kind of curse by proponents of purely applicative programming in the lambda calculus). By an *effect* we mean a local state change in a shared actor which causes a change in behavior that is visible to other actors. For example sending a deposit to an account shared by multiple users should have the effect of increasing the balance in

the account.

ACCOUNT43 is an example of the kind of shared resource which is important in the conceptual modeling of Open Systems. Such shared resources should be suitable for use by a growing collection of users. To a given user, a shared account will exhibit indeterminacy in the balance depending on the deposits and withdrawals made by other users. The indeterminacy arises from the indeterminacy in the arrival order of messages at the shared account. A common bug in attempting to model computation in open systems is to assume that an agent sending messages can determine the order in which they will be seen by the recipient. It is important to realize that the above assumption is contrary to the nature of Open Systems. Also implementing shared resources inherently involves being open to outside communications, even those put forth by parties which joined the Open System after the request being considered was received.

2.3. Multiple-Inheritance

A description, such and `(a SharedAccount)`, can inherit attributes and behavior from other descriptions. In this example, a shared account inherits from *both* the description of an `Account` *and* the description of a `Possession`.

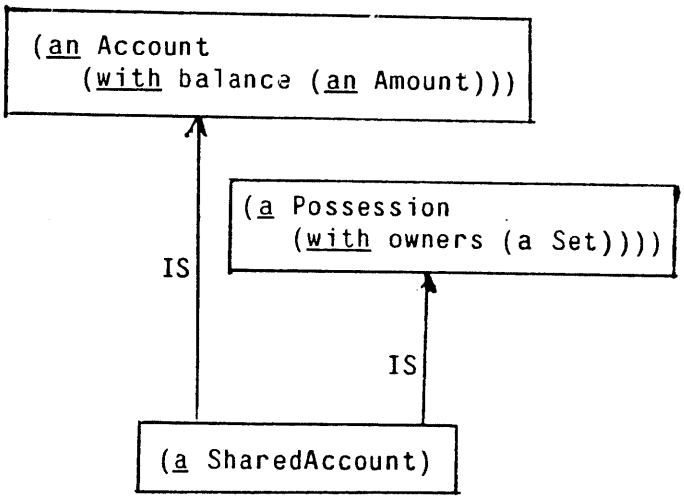

Figure 2-2: Multiple Inheritance

Dealing with the issues raised by the possibility of being a specialization of more than one description has become known as the "Multiple Inheritance Problem". A number of approaches have been developed in the last few years including the following: [Weinreb, Moon 81], [Curry, Baer, Lipkie, Lee 82], [Borning, Ingalls 82], [Bobrow, Stefik 82] and [Borgida, Mylopoulos, Wong 82]. Our approach differs in that it builds on the theory of an underlying description system [Attardi, Simi 81]. This theory axiomatizes the relationship between the multiple inherited descriptions. This theory also maintains the parallelism inherent in actor

theory, and thus is suitable for describing highly distributed open systems.

2.4. Change

The degree of parallelism is partially determined by whether the actor can change its local state or not. Actors which can change their local state are called *serialized* actors. A serialized actor accepts only one message at a time for processing; it will not accept another message for processing until it has dealt with the one which it has received. A communication received by a serialized actor is processed in order of arrival; although, not necessarily in the order sent. Actors which can never change their local state are called *unserialized* actors. These actors are able to process arbitrarily many messages at the same time. Actors such as the square root function and the text of Lincoln's Gettysburg Address are unserialized.

Open systems do not have well defined global states. Effects are implemented by an actor changing its *own local state* using a become command [Hewitt, Attardi, Lieberman 79]. There are *no* assignment commands. Our conceptual model of change contrasts with the usual computer science notion in which change is modeled by updating the state components of a universal global state [Milne, Strachey 76]. The absence of a well defined global state is a fundamental difference between the actor model and classical sequential models of computation (viz. [Turing 37], [Church 41], etc.) *Actor systems can perform nondeterministic computations for which there is no equivalent nondeterministic Turing Machine* [Clinger 81]. The nonequivalence points up the limitations of attempting to model parallel systems as nondeterministic sequential machines [Milne, Strachey 76]. This is not to say that actor systems can implement functions which are not recursively computable (like solving the Halting Problem). Instead the point is that recursive functions do not provide an adequate model of parallelism. I.E. an Open System cannot be adequately modeled as a recursive function which maps global states to global states because at any given point in time an Open System does not in general have a well defined global state. Will Clinger has developed an elegant mathematical theory [Clinger 81] (called Actor Theory) which accounts for capabilities of actor systems which go beyond those of nondeterministic Turing Machines. *We claim that nondeterministic Turing machines are an unsatisfactory model of the computational capabilities of large-scale open systems.*

2.5. Concurrency

Concurrency in Open Systems stems from the parallel operation of an incrementally growing number of multiple, independent, communicating agents. Sites can join an Open system in the course of its normal operation--sometimes even affecting the results of computations initiated *before* they joined. *Actor Theory has been designed to accurately model the computational properties of Open Systems.* It is a consequence of the Actor Model that purely functional programming languages based on the lambda calculus *cannot* implement shared accounts in Open Systems. The continuation technique promoted by Strachey and Milne [Milne,

Strachey 76] for simulating some kinds of parallelism in the lambda calculus does not apply to Open Systems. The lambda calculus simulation is sequential whereas Open Systems are inherently parallel. Concurrency in the lambda calculus stems from the concurrent reduction/evaluation of various parts of a *single* lambda expression with an environment which is fixed when the lambda expression is created. In Open Systems independent agents can incrementally and independently spawn ongoing computations so the evaluation of an expression can be affected by actions which were initiated after the evaluation of the expression is begun.

2.6. Truth and behavior

Message Passing Semantics takes a different perspective on the meaning of a sentence from that of truth-theoretic semantics. In truth-theoretic semantics [Tarski 44], the meaning of a sentence is determined by the models which make it true. For example the conjunction of two sentences is true in a model exactly in which both of its conjuncts are true in the model. *In contrast Message Passing Semantics takes the meaning of a message to be the effect it has on the subsequent behavior of the system.* In other words the meaning of a message is determined by how it affects the recipients. Each partial meaning of a message is constructed by a recipient in terms of how it is processed (c.f. [Reddy 79]). The meaning of a message is open ended and unfolds indefinitely far into the future as other recipients process the message.

At a deep level, understanding always involves categorization, which is a function of interactional (rather than inherent) properties and the perspective of individual viewpoints. Message Passing Semantics differs radically from truth-theoretic semantics which assumes that it is possible to give an account of truth in itself, free of interactional issues, and that the theory of meaning will be based on such a theory of truth [Lakoff, Johnson 80].

3. Description and Action

3.1. Limitations of Descriptions

The distinction between *doing* and *describing* is different from the usual distinction made in the Artificial Intelligence literature [McCarthy, Hayes 69] [Hayes 77] between the *epistemological adequacy* of a system (its *accuracy* with respect to truth-theoretic semantics [Tarski 44]) and the *heuristic adequacy* (the *efficiency* of its inferential procedures in proving theorems). Thus the distinction between epistemological adequacy and heuristic adequacy is founded on the basis of truth-theoretic semantics. Our simple example can help clarify the distinction. An epistemologically adequate theory of financial accounts gives an accurate description of the rules that govern them. An heuristically adequate system can derive theorems in the theory of financial accounts fast enough to answer queries. Both kinds of adequacy are concerned with the *description* of financial accounts: the former with its accuracy; the later with the efficiency with which the description can be

used to answer questions.

Neither kind of adequacy actually *accomplishes* creating a shared account with $10 in it so a growing set of geographically dispersed users can make deposits and withdrawals. We could *describe* a certain kind of account with an axiom such as the following:

```
((d > 0) implies
    (replies-after-acceptance
       (a SharedAccount (with balance b))
       (a Deposit (with amount d))
       (a CompletionReport
          (with ResultingBalance (b + d)))))
```

where the variables b and d are universally quantified and the predicate `replies-after-acceptance` takes three arguments which are a description of the local state of an actor when it accepts a request message, a description of the request accepted, and a description of the reply to the request, respectively. Now it should be clear that hypothesizing that ACCOUNT43 satisfies the following description:

```
(a SharedAccount
   (with InitialBalance $20)
   (with Place BankOfLondon)
   (with Time "Midnight, December 31, 1987"))
```

does not by itself actually create such an account. Indeed the above assertion is ambiguous between the following interpretations:

- *Hypothesis Interpretation*: We have just discovered that ACCOUNT43 is already such an account and want to explicitly record this in the data base.

- *Goal Interpretation*: We want to declare our goal of having ACCOUNT43 be such an account and we will devote some effort to establishing the goal.

To actually create the account requires providing the money for the initial deposit in the account. Making assertions by itself will not suffice. Similarly asserting the ACCOUNT43 is not a `SharedAccount` does not in itself destroy an account which is located elswhere.

3.2. Taking Action

Knowing that something is true and taking action to make it come true are two different things. Both are important and they should not be confused. In this section we discuss how actors can take action to supplement the previous sections discussion of how they can manipulate descriptions.

Actions such as creating a new shared account with balance $12 and owner Ken can be performed by evaluating an expression such as the one below in the programming language Act2 [Theriault 83], [Lieberman

81a]:

```
(new SharedAccount
   (with Balance $12)
   (with Owners {Ken}))
```

Below we present *part* of the implementation of `Sharedaccount`.

```
(Define (new SharedAccount
           (with balance =b)
           (with owners =s))

  (create
     (is-request (a balance) do
        (reply (a SharedAccount
                  (with balance b))))

     (is-request (a deposit
                    (with amount =a) do
        (become (new SharedAccount
                  (with balance (+ b a))))
        (reply (a deposit-receipt
                  (with amount a)))))
     ...
     ...
     ...)))
```

At this point we would like to take note of several unusual aspects of the above implementation. The ACT2 language is an open system, i.e. each command and expression parses and evaluates itself. New commands and expressions can easily be added to the language. They are also actors, so they can execute in parallel. For example, in the communication handler for `Deposit` requests above, the following two commands are executed concurrently:

```
(become (new SharedAccount
          (with balance (+ b a))))
(reply (a deposit-receipt
          (with amount a)))
```

The principle of maximizing concurrency is fundamental to the design of actor programming languages. It accounts for many of the differences with conventional languages based on communicating sequential processes.

4. Related Work

The object-oriented programming languages (e.g. [Birtwistle, Dahl, Myhrhaug, Nygaard 73], [Liskov, Snyder, Atkinson, Schaffert 77], [Shaw, Wulf, London 77], and [Ichbiah 80]) are built out of objects (sometimes called "data abstractions") which are completely separate from the procedures in the language. Similarly the lambda calculus programming languages (e.g. [McCarthy 62], [Landin 65], [Friedman, Wise 76], [Backus 78], and [Steele, Sussman 78]) are built on functions and data structures (viz. lists, arrays, etc.) which are separate. SmallTalk [Ingalls 78] is somewhat a special case since it simplified Simula by leaving out the procedures entirely, i.e. it has only classes. The Simula-like languages provide effective support for coroutines but not for concurrency. In contrast the Actor Model is designed to aid in conceptual modeling of shared objects in a highly parallel open systems. Actors serve to provide a unified conceptual basis for functions, data structures, classes, suspensions, futures, procedure invocations, exception handlers, objects, procedures, processes, etc. in all of the above programming languages [Baker, Hewitt 77], [Lieberman 81b], [Hewitt, Attardi, Lieberman 79]. For example sending a request communication generalizes the traditional procedure invocation mechanism which requires that control return to the point of invocation. A request communication contains the mail address of a customer to which the response to the request should be sent as well as the message specifying the task to be performed. In this way, exception handlers [Liskov, Snyder, Atkinson, Schaffert 77], [Ichbiah 80] and co-routines [Birtwistle, Dahl, Myhrhaug, Nygaard 73] are conveniently unified with other more general control structures. The Actor Model unifies the conceptual basis of the lambda calculus and the object-oriented schools of programming languages--being mathematically defined, it is independent of all programming languages.

Prolog based systems such as Intermission [Kahn 82] and Concurrent prolog [Shapiro 83] are pragmatically useful and interesting experiments in their own right. Unfortunately, neither as yet has a well developed mathematical semantics which would make it possible to directly analyze them as we have done for the lambda calculus and first order logic. *To the extent that Intermission and Concurrent Prolog are based on first order logic and the lambda calculus, they inherit limitations discussed in this paper.*

5. Conclusion

The fundamental thesis of this paper is that knowing that something is the case and taking action to make it the case are two different things which should not be confused. Asserting a proposition P can mean that we have established that P is the case and want to explicitly record this in the data base. Or it can mean that we want to declare that P is our goal and we will devote some effort to planning how to do it. Neither of the above possibilities inherently involves taking any action. The capability to take action as well as to describe the world is very important. The relationships between description and action are quite subtle.

We claim that actors (unlike lambda expressions, logical implications, etc.) are the universal objects of concurrent systems and that they can serve as a efficient interface between the hardware and software. Actors provide an absolute conceptual interface between the software and hardware of parallel computer systems. The function of the hardware is to efficiently implement the primitive actors and the ability to communicate in parallel. Software systems in turn can be implemented in terms of actors completely independently of the hardware configuration. The actor concept itself is defined mathematically and is thus logically independent of all programming languages and hardware architectures. A system consisting of multiple processors -- called the APIARY -- is being developed to use the inherent parallelism of actor systems to increase the efficiency of computation [Hewitt 80].

Message Passing Semantics deals coherently with both doing and describing whereas truth-theoretic semantics only addresses some of the issues of describing. We claim that it is impossible to implement shared resources for Open Systems using description systems such as first order logic and the lambda calculus because they lack the necessary communication capabilities.

Description languages based on first order logic and/or the lambda calculus have been designed to express properties but are incapable of taking action. On the other hand procedural languages (such as current dialects of Lisp and Ada) have been designed to efficiently take action but they suffer from a lack of descriptive capabilities. We need good ways to integrate the roles of descriptions and actions in our systems. Some of the ideas in this paper have been applied to the analysis of the relationship between the roles of descriptions and actions in organizational work [Barber, de Jong, Hewitt 83].

6. Acknowledgments

Much of the work underlying our ideas was conducted by members of the Message Passing Semantics group at MIT. We especially would like to thank Jon Amsterdam, Jerry Barber, Henry Lieberman, and Dan Theriault. Extensive discussions with our collaborators Elihu Gerson and Leigh Star have been of fundamental importance in improving the organization and content of this paper. A chat with Marvin Minsky and Danny Hillis helped to clarify the nature of the differences between actions and descriptions.

The development of the Actor Model has benefited from extensive interaction with the work of Jack Dennis, Bob Filman, Dan Friedman, Bert Halstead, Tony Hoare, Gilles Kahn, Dave MacQueen, Robin Milner, Gordon Plotkin, Steve Ward, David Wise over the past half decade. The work on Simula and its successors SmallTalk, CLU, Alphard, etc. has profoundly influenced our work. We are particularly grateful to Alan Kay, Peter Deutsch, Laura Gould, and the other members of the Learning Research group for interactions and useful suggestions.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Major support for the research reported in this paper was provided by the System Development Foundation. Major support for other related work in the Artificial Intelligence Laboratory is provided, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505. We would like to thank Charles Smith, Mike Brady, and Patrick Winston for their support and encouragement.

References

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

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

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

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

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

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

[Borgida, Mylopoulos, Wong 82]
Borgida, A., Mylopoulos, J. L., Wong, H. K. T.
Generalization as a Basis for Software Specification.
In Brodie, M. L., Mylopoulos, J. L., Schmidt, J. W., editor, *Perspectives on Conceptual Modeling*. Springer-Verlag, 1982.

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

[Church 41]
Church, A.
The Calculi of Lambda-Conversion.
In *Annals of Mathematics Studies Number 6*. Princeton University Press, 1941.

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

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

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

[Hayes 77]
Hayes, P. J.
In Defense of Logic.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 559-565. Cambridge, Ma, 1977.

[Hewitt 80]
Hewitt C. E.
The Apiary Network Architecture for Knowledgeable Systems.
In *Conference Record of the 1980 Lisp Conference.* Stanford University, Stanford, California, August, 1980.

[Hewitt, Attardi, Lieberman 79]
Hewitt C., Attardi G., and Lieberman H.
Specifying and Proving Properties of Guardians for Distributed Systems.
In *Proceedings of the Conference on Semantics of Concurrent Computation.* INRIA, Evian, France, July, 1979.

[Hewitt, Baker 77]
Hewitt, C. and Baker, H.
Laws for Communicating Parallel Processes.
In *1977 IFIP Congress Proceedings.* IFIP, 1977.

[Ichbiah 80]
Ichbiah, J. D.
Reference Manual for the Ada Programming Language.
November 1980 edition, United States Department of Defense, 1980.

[Ingalls 78]
Ingalls D.
The SmallTalk-76 Programming System, Design and Implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.* ACM, Tucson, Arizona, January, 1978.

[Kahn 82]
Kahn, K. M.
Intermission - Actors in Prolog.
In *Logic Programming*. Academic Press, 1982.

[Lakoff, Johnson 80]
Lakoff, G., Johnson, M.
Metaphors We Live By.
The University of Chicago Press, 1980.

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

[Lieberman 81a]
Lieberman, H.
A Preview of Act-1.
A.I. Memo 625, MIT Artificial Intelligence Laboratory, 1981.

[Lieberman 81b]
Lieberman, H.
Thinking About Lots of Things At Once Without Getting Confused: Parallelism in Act-1.
A.I. Memo 626, MIT Artificial Intelligence Laboratory, 1981.

[Liskov, Snyder, Atkinson, Schaffert 77]
Liskov B., Snyder A., Atkinson R., and Schaffert C.
Abstraction Mechanism in CLU.
Communications of the ACM 20(8), August, 1977.

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

[McCarthy, Hayes 69]
McCarthy, J. and Hayes, P. J.
Some Philosophical Problems from the Standpoint of Artificial Intelligence.
In *Machine Intelligence 4*, pages 463-502. Edinburgh University Press, 1969.

[Milne, Strachey 76]
Milne, R. and Strachey, C.
A Theory of Programming Languages.
John Wiley & Sons, New York, 1976.

[Reddy 79]
Reddy, M.
The Conduit Metaphor.
In Ortony, A., editor, *Metaphor and Thought.* Cambridge University Press, 1979.

[Reiter 82]
Reiter, R.
Towards a Logical Reconstruction of Relational Database Theory.
In Brodie, M. L., Mylopoulos, J. L., Schmidt, J. W., editor, *Perspectives on Conceptual Modeling.* Springer-Verlag, 1982.

[Shapiro 83]
Shapiro, E.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, ICOT, January, 1983.

[Shaw, Wulf, London 77]
Shaw, M., Wulf, W. A., London, R. L.
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators.
Communications of the ACM 20(8), August, 1977.

[Steele, Sussman 78]
Steele G. L. Jr., Sussman, G. J.
The Revised Report on SCHEME: A Dialect of LISP.
AI Memo 452, MIT, January, 1978.

[Tarski 44]
Tarski, A.
The Semantic Conception of Truth.
Philosophy and Phenomenological Research 4 :341-375, 1944.

[Theriault 83]
Theriault.
Issues in the Design and Implementation of Act2.
Master's thesis, Massachusetts Institute of Technology, 1983.

[Turing 37]
Turing, A. M.
Computability and λ-definability.
Journal of Symbolic Logic 2:153-163, 1937.

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