Working Paper 186B

June 1979

Preliminary Design of the APIARY for VLSI Support of Knowledge-Based Systems

Carl Hewitt

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Working Papers are informal papers intended for internal use.

This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

© MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1979

Preliminary Design of the APIARY for VLSI Support of Knowledge-Based Systems

Carl Hewitt
Room 813
545 Technology Square
Cambridge, Mass. 02139

(617) 253-5873

ABSTRACT

Knowledge-based applications will require vastly increased computational resources to achieve their goals. We are working on the development of a VLSI Message Passing Architecture to meet this need. As a first step we present the preliminary design of the APIARY system in this paper. The APIARY is currently in an early stage of implementation at the MIT Artificial Intelligence Laboratory.

I -- Support for Knowledge-Based Systems

The development of Knowledge-based systems (MACSYMA, DENDRAL, Programming Apprentices, Therapy Advisors, Tutors, Trouble-shooters) will require increasing hardware support to realize their goals. Knowledge-based systems need to provide quick response in order to interact effectively with their users. In these systems there is a vast amount of potential concurrency which we would like to exploit.

II -- Goals of Message Passing Architecture

The primary source of our interest in developing a VLSI message passing architecture is to provide a suitable hardware and software base for such knowledge-based applications. We are working to create a VLSI Message Passing Architecture to support our knowledge-based systems research. Message Passing Architecture [Hewitt, Bishop, and Steiger: 1973; Steiger: 1974; Bishop: 1977; Baker: 1978; Halstead: 1978; Ward: 1978] provides for the connection of a large number of **workers**. Each **worker** consists of a communications processor, work processor, integrated circuit memory, and communications interface.

The system which we propose to construct is called the **APIARY** because it is reminiscent of the social organization of bees. Like that of the honeybee, our APIARY consists of a large number of workers closely cooperating to accomplish the tasks at hand.

Message Passing Architecture differs from conventional multiprocessor architecture (MULTICS, Burroughs 6700, C.mmp, etc.) in which processors communicate through shared memory using a crossbar switch. The workers of an APIARY do not share any physical memory. They communicate with each other by sending and receiving messages.

Pipelined machines such as the CRAY-1 and the CDC STAR are not suitable for the implementation of knowledge-based systems because the large number of conditional branches disrupts any ability to keep a pipeline full.

III -- Actors

The APIARY is intended to supply an execution environment for objects called **actors** which specify the actual work of computation. An actor is a virtual object which is defined by its behavior when it receives messages. An actor *conceptually* consists of a **script** which specifies what it does when it receives a message and a vector of network addresses (called the **acquaintances**). The *only* operation which can be performed on a network address is to use it to send a message to the actor which is named by the network address. Each message is an actor.

Each actor x obeys the **principle of locality** which states that if x receives a message m, then the only actors which can be directly sent messages are the acquaintances of x and the acquaintances of m. We plan to heavily exploit the **principle of locality** in our implementations of the APIARY. For example our algorithms for the real time garbage collection of the storage for **inaccessible** actors are dependent on the principle of locality. An **inaccessible** actor is one which can no longer be sent a message (either directly on indirectly) by the users of the APIARY.

Implementations of actor systems are allowed to use any methods whatsoever *provided* that the effects are indistinguishable from those which would result from actually sending messages (which is called the **principle of indistinguishable effects**). Our implementations will depend heavily on this principle in order to gain efficiency. For example actors implemented directly in hardware do not consist of two separate objects for the **script** and **acquaintances**. The implementations of primitive actors (such as integers and sequences) is supported by micro-code so that in most cases messages do not have to be sent. Some users might want to implement some actors which subsume the behavior of integers and sequences (perhaps for the purpose of instrumenting certain actors). These actors will in fact receive messages. This capability can be implemented without significantly slowing down the primitive integers and sequences.

The implementations we are developing have the property that it is impossible to determine the *physical* representation of any actor (which is called the **principle of independence of physical representation**). This principle is important because it allows us to extend the behavior of any actor at any time without affecting the behavior of any other actor. Using the **principle of indistinguishable effects, independence of physical representation** can be achieved *without* significantly sacrificing efficiency.

IV -- Serialized and Unserialized Actors

An **unserialized** actor can process arbitrarily many messages concurrently. Functions such as factorial, tangent, and eval (which performs evaluation given an expression and environment) are examples of unserialized actors. On the other hand, a **serialized** actor can process only one message at a time.

A serialized actor has a **current state** which can change as a result of the messages received. At any given point in time a serialized actor can be either **locked** or **unlocked.** When it is created it is **unlocked.** When the first message arrives, the serialized actor becomes **locked** and its **script** is executed. While **locked** the actor might possibly send some messages to other actors. The **next state** of the serialized actor is specified and and then the actor becomes **unlocked** and thus able to **accept** another message. An important consideration in the design of efficient serialized actors is that they should remain **locked** for as brief a time as possible.

Note that there are three separate events which must occur before a message can be **accepted** by a serialized actor I. First it must be sent in a transmission event. Next it must arrive in an arrival event. Hardware modules called arbiters are used to establish an arrival ordering for all messages delivered to I. Finally it must be accepted in an acceptance event. Messages are **accepted** in the order in which they arrive. The acceptance marks a transition in which the target changes from **unlocked** to **locked.** Thus if a serialized actor becomes **locked** then no more messages can be accepted until it **unlocks.** Messages which arrive at a serialized actor when it is **locked** are queued.

V -- Workers

A **worker** consists of a communications processor, work processor, integrated circuit memory, and a communications interface.

The communications processor of a worker provides the following functions:

accepts messages from the work processor for delivery to other workers.

accepts messages destined for actors that reside in the worker and inserts an entry in the work load of the work processor to cause the message to be processed.

forwards messages along the route from worker to worker.

The primary function of a **worker** is to provide storage, processing power, and communications services for the actors which reside in it. To accomplish these goals, each worker performs the following housekeeping functions:

local real time garbage collection of actors: We are using the algorithm of [Baker: 1978].

dynamic load balancing: Initially we are using a simple algorithm in which pairs of neighboring workers periodically compare work loads and shift any excess from the more heavily loaded worker.

migration: In many cases it is desirable to move an actor between workers or between a worker and secondary storage such as disks. This function is performed by a part of the APIARY called the MOVER [Bishop: 1977].

communication: messages must be routed to reach the destination.

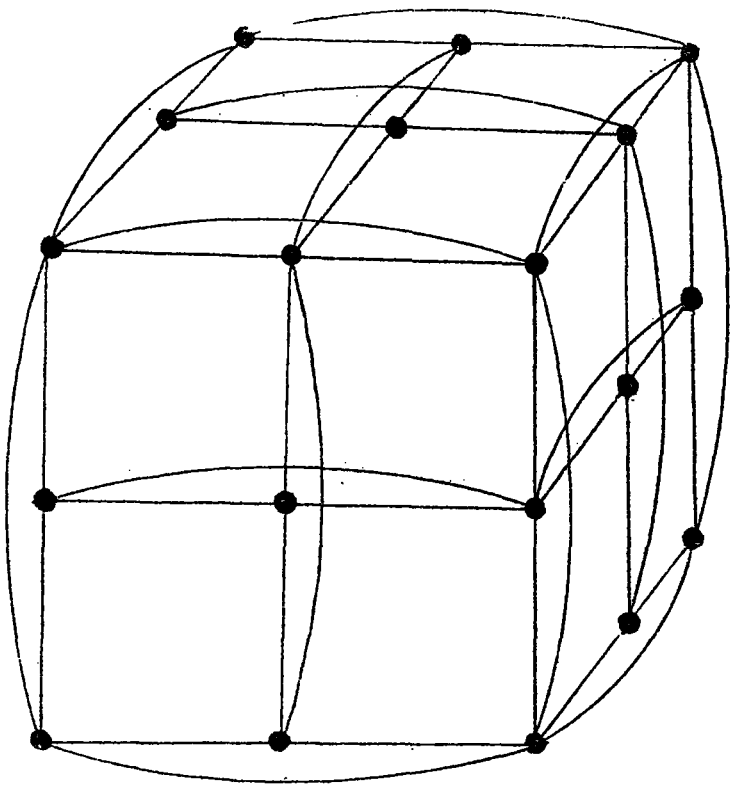
The **migration**, **communication**, and **load balancing** mechanisms are tightly intertwined with each other so that efficient implementation involves many important research issues which are not yet resolved.

VI -- Interconnection of Workers

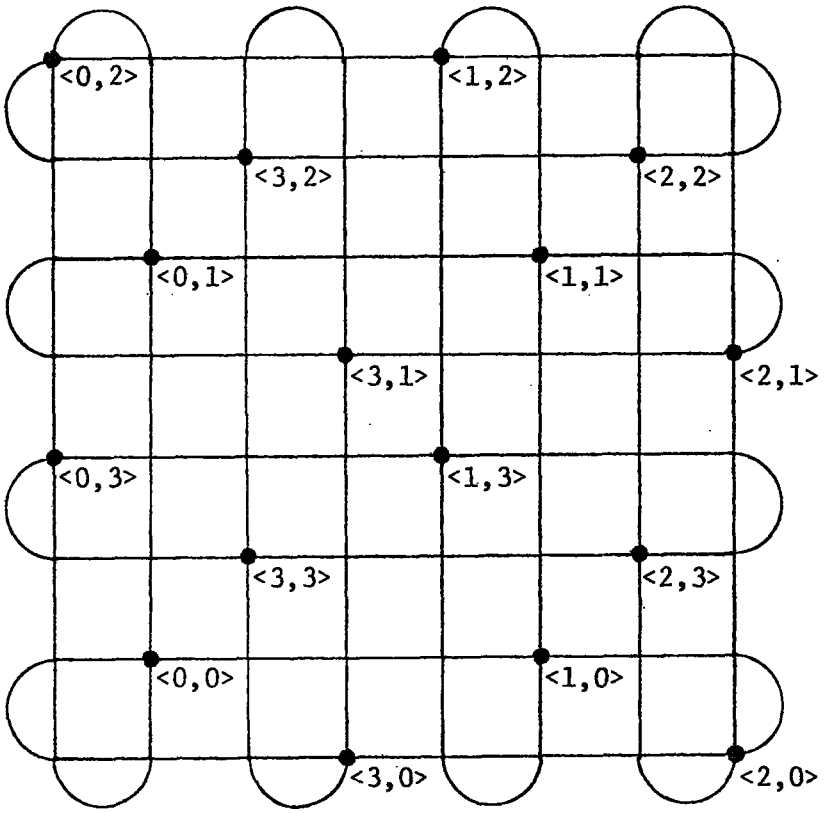
Each worker is connected to a number of other workers called its **neighbors**. We propose to investigate several different technologies for communicating between workers. The possibilities include electrical busses, coaxial cables (using the CHAOSNET protocols which are a further development of the ETHERNET, ARPANET, and ALOHA protocols), and fiber optics.

We plan to experiment with several interconnection topologies for the workers in an APIARY including the Batcher sorting network [Batcher: 1968 and Moravec: 1978], Boolean n-dimensional hypercube [Pease: 1977 and Sullivan and Bashkow: 1977], and another family of topologies which we call the **hypertorus**.

A **Cartesian hypertorus** is obtained by giving each worker three integer coordinates $\langle x, y, z \rangle$ such that $1 \leq x \leq l$, $1 \leq y \leq w$, and $1 \leq z \leq h$ where l, w, and h are the length, width, and height respectively. Each such node is connected to the six others with coordinates $\langle x \pm_l 1, y \pm_w 1, z \pm_h 1 \rangle$ where $+_u$ and $-_u$ are respectively addition and subtraction modulo u. The hypertorus is topologically the same as a three dimensional cube with the opposite faces identified. It can be visualized as three orthogonal interlocking doughnuts. Below is a diagram that shows the connections for three faces of a $3 \times 3 \times 3$ Cartesian hypertorus:



Unfortunately packaging suggested by the above diagram requires a large number of very long wires between opposite faces. Richard Zippel has developed a very clever way to package the Hypertorus topology so that essentially no long wires are required. Below we show the connections for a **4 x 4 two dimensional** Cartesian hypertorus.



The Hypertorus topologies have a number of potential advantages which we would like to explore. In contrast to sorting networks and Boolean n-cubes, the number of immediate neighbors of a worker does not increase with the total number of workers in the APIARY. A worker in a Cartesian hypertorus always has exactly 6 neighbors regardless of the number of workers. In contrast a worker in a Boolean n-cube has (log N) neighbors where N is the number of workers. A Hypertorus is **homogeneous** and **isotropic** without any annoying boundary effects. The property of **homogeneity** is that the workers look pretty much the same: there are no preferred locations. Similarly the property of **isotropy** is that there are no preferred directions: the APIARY looks the same in all directions. The properties of **homogeneity** and **isotropy** are important because they vastly simply the software systems since complicated optimizations do not have to be performed to take boundary effects and other irregularities into account. A Hypertorus has no bottlenecks of the kind which can limit the parallelism possible in tree-structured topologies [Keller, Lindstrom, and Patil 1979]. The Hypertorus topology will become particularly attractive if methods are developed to interconnect chips directly in three dimensions as opposed to the two dimensions provided by chips connected through boards.

Ultimately with the development of *ultra* large scale integration (millions of gates on a chip), we will be able to package many workers in a single chip. This development will make an APIARY with thousands of workers practical for realistic applications.

VII -- Migration of Actors

Allowing actors to migrate from worker to worker is an important capability of the APIARY. It is essential to the implementation of real time garbage collection over the entire APIARY. When an actor can no longer be accessed from the worker on which it currently resides, then it is often desirable to move it to another worker which is still interested.

Actors can also be moved to improve the efficiency of the system. Copies of unserialized actors can be moved to workers in which the space-time tradeoff favors keeping a local copy over communicating with an already existing copy. A serialized actor might be moved to bring it closer to workers which are using it intensively.

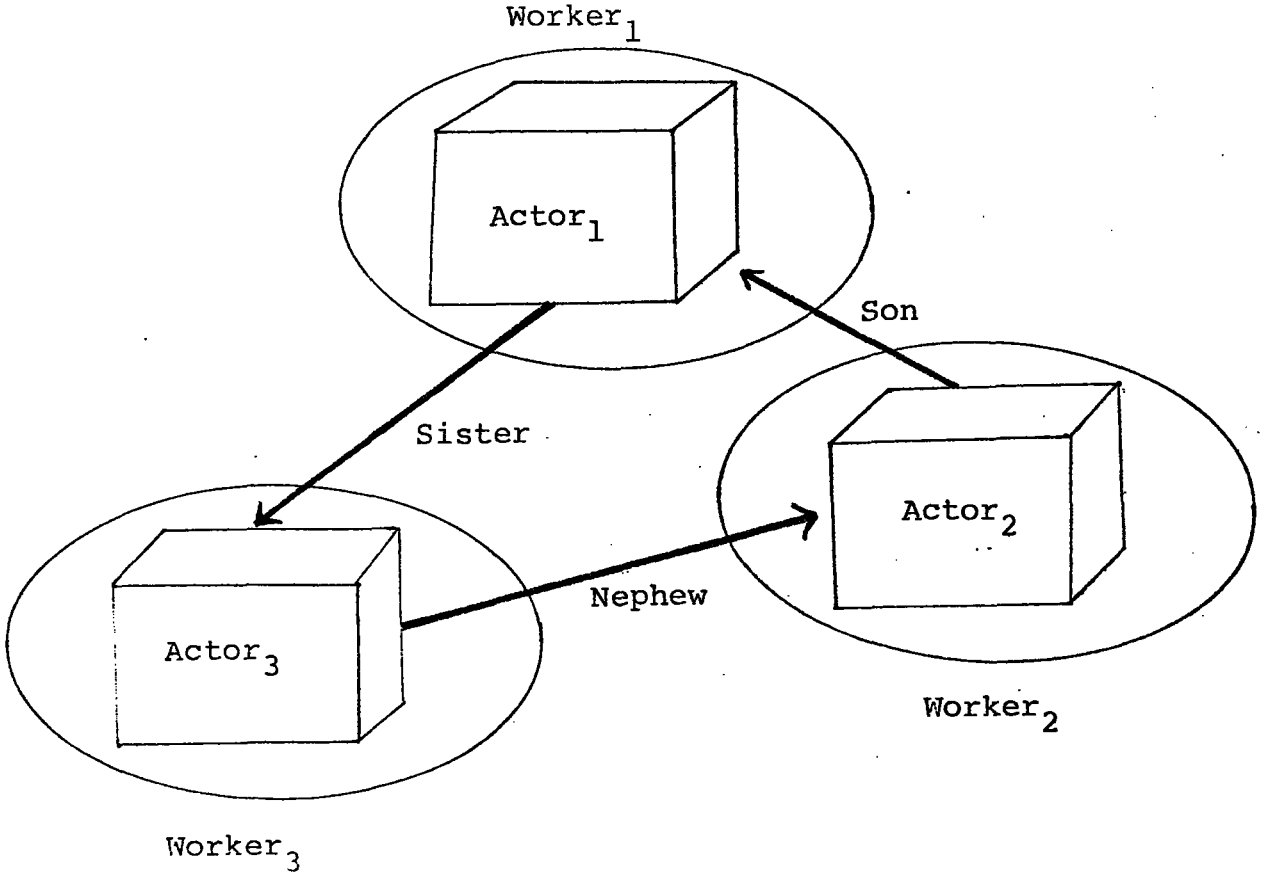
The MOVER needs to insure that the actor being moved will continue to receive its messages at its new location. This is accomplished using a combination of several mechanisms. One mechanism is to simply forward the mail from the old location. Another complementary mechanism is to maintain an **interest group** for an actor that migrates. The **interest group** of an actor is the set of workers who are interested in

sending messages to the actor. When an actor migrates to a new location, all of the members of its **interest group** will be notified.

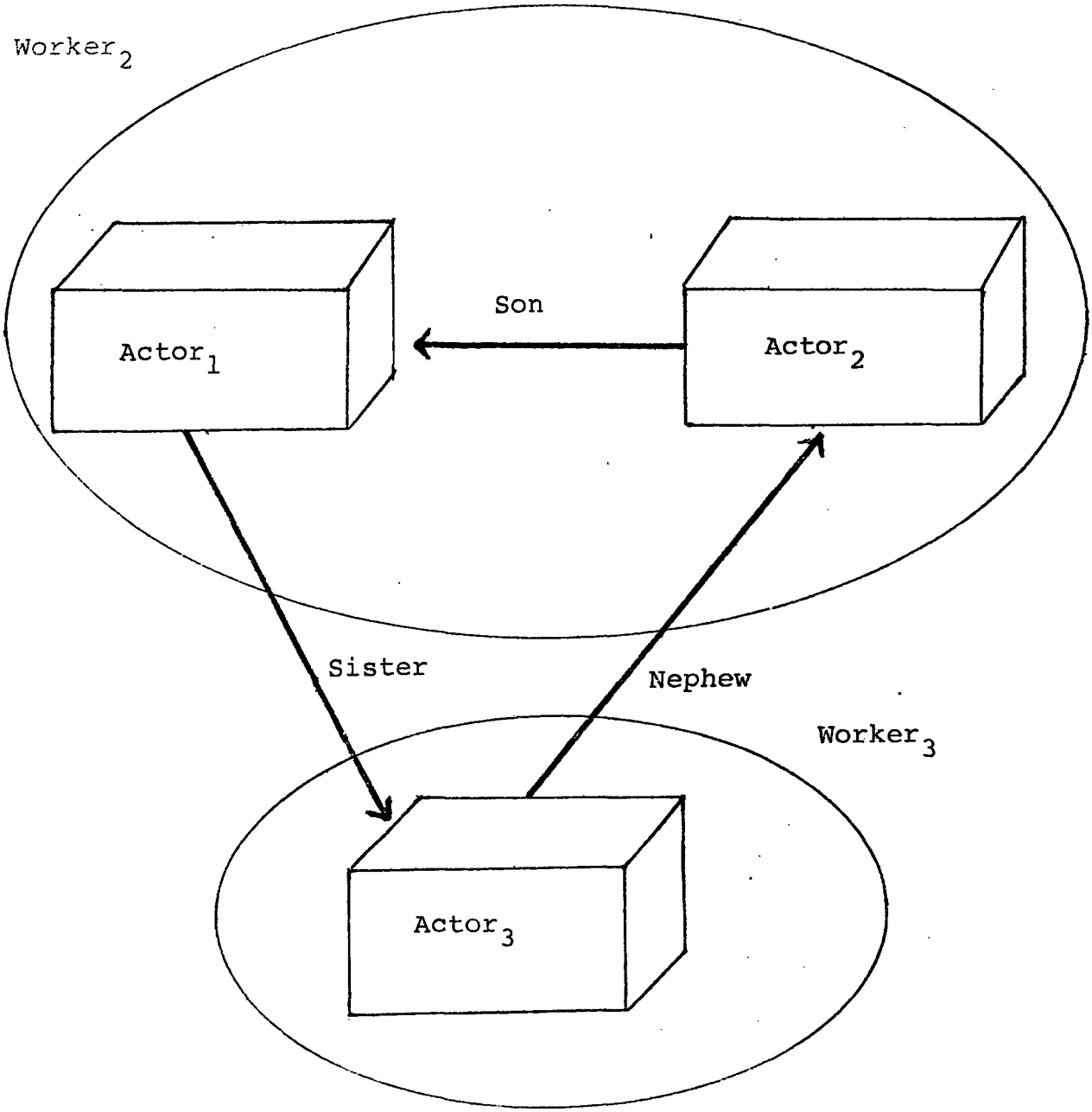
One of the primary design goals of the APIARY is that workers should be able to quickly communicate with one another. This means that messages from an actor in one worker to an actor in a different worker should not travel via an inefficient path. For this reason **interest groups** are a separate mechanism from **message transport** in contrast to [Halstead: 1978] where the **reference tree** mechanism performs both functions. The function of **message transport** is to efficiently deliver messages from one actor to another. The function of **interest groups** is to ensure that messages will be delivered to actors which migrate.

Using the ability to migrate actors, the problem of **real time** garbage collection on a multiprocessor system can be solved using the following algorithm which is a slight adaptation of [Bishop: 1977]. The major problem is to garbage collect cycles of actors which contain each others network addresses where the cycle spans more than one worker.

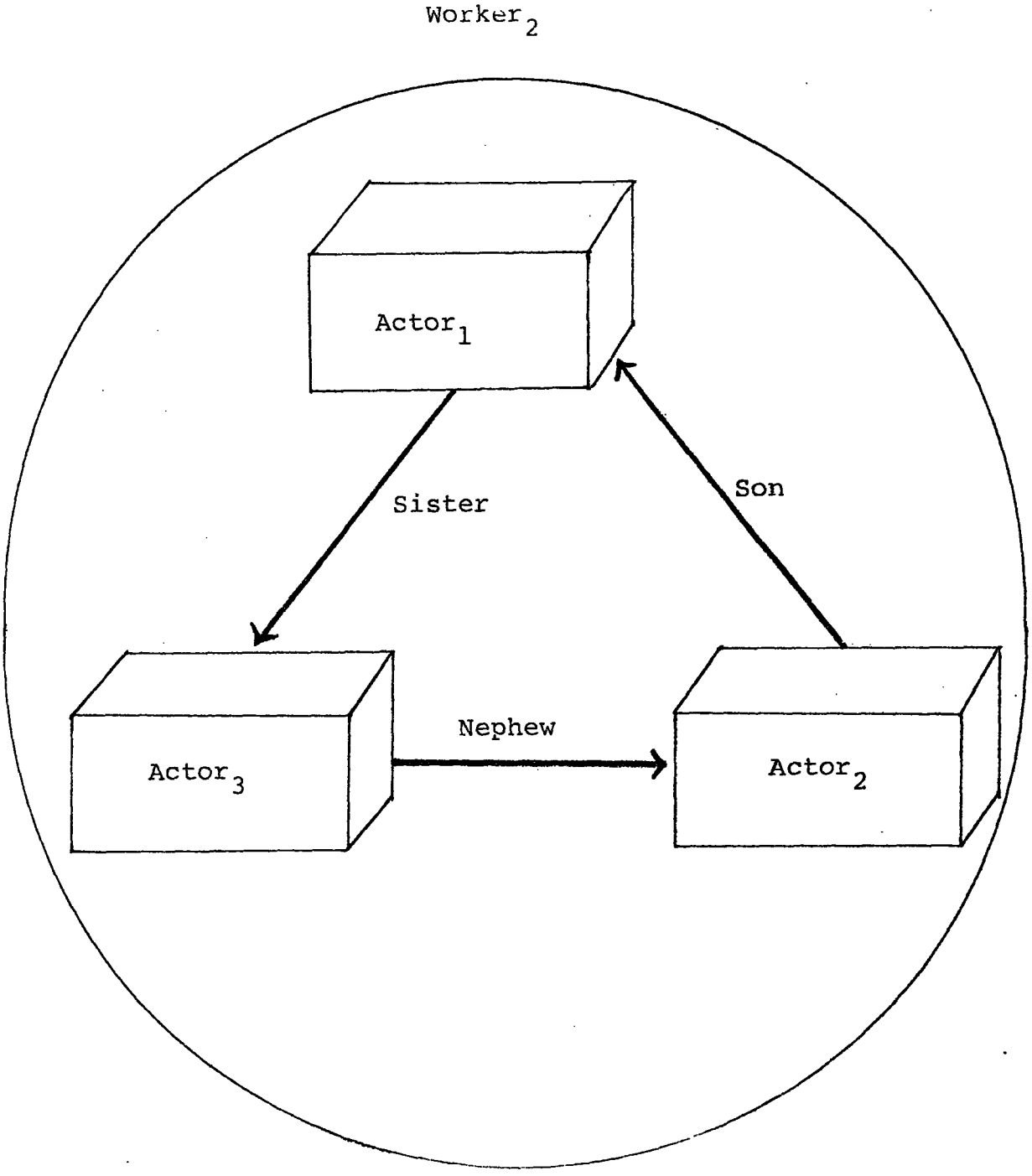
The diagram below depicts a situation in which actor_1 resides in worker_1, actor_2 in worker_2, and actor_3 in worker_3. Furthermore actor_1 has an acquaintance named **sister** which is the network address of actor_3; actor_2 has an acquaintance named **son** which is the network address of actor_1; and actor_3 has an acquaintance named **nephew** which is the network address of actor_2.



Suppose that worker$_1$ completes a phase of its local **real-time** garbage collection so that it knows that actor$_1$ is not referenced from elsewhere within itself. It would like to move actor$_1$ to another worker which is interested in using it. So a member of the **interest group** of actor$_1$ is selected and actor$_1$ is moved to the worker selected (in this case worker$_2$) producing the situation depicted below:

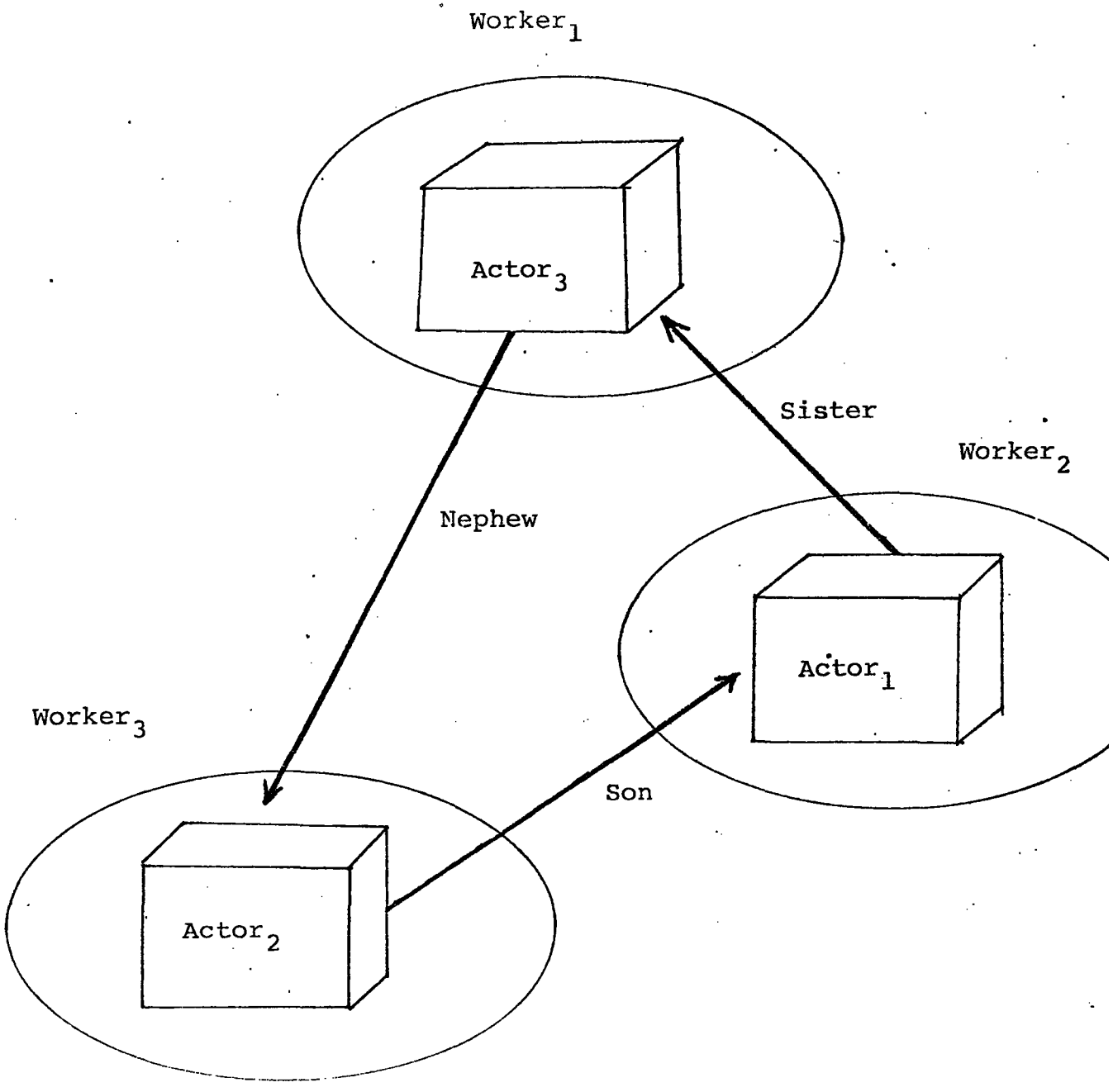


Next suppose that $worker_3$ completes a phase of its local real time garbage collection so that it knows that $actor_3$ is not referenced from elsewhere within itself. As before a member of the interest group of $actor_3$ is selected and $actor_3$ is moved to the worker selected (in this case also $worker_2$) producing the situation depicted below:



Notice that the entire cycle is now contained within $worker_2$. Storing entire cycles in a single worker should not be much of a problem since a worker can store some of its actors on secondary storage. When $worker_2$ completes the next phase of its local real time garbage collection the space for $actor_1$, $actor_2$, and $actor_3$ is reclaimed.

The procedure illustrated above tends to shrink the number of workers involved in a cycle of inaccessible actors. As pointed out by [Halstead: 1978], in certain cases the procedure may not reclaim the storage of all inaccessible actors. For example if all three workers in the initial situation complete phases of their garbage collection at approximately the same time then the following situation can result:



Notice that the above situation does *not* represent any improvement over the original in recovering the storage of inaccessible actors. Hopefully dynamically maintained cycles of inaccessible actors will not persist for long periods of time in practice. We plan to investigate this problem carefully.

VIII -- A Computational Environment

The techniques of **virtual memory** and **real time garbage collection** have proven to be extremely important in the development of efficient useful storage management systems. We propose to develop algorithms for **virtual workers** and the **real time allocation of work** to aid in the management of thousands of VLSI APIARY workers. The actor concept provides a good basis for the design of systems with these goals.

Message Passing Architecture must *efficiently* provide the following capabilities:

storage for actors which have been created

transport for messages from one actor to another

processing power for actors which have received messages

migration of actors to locations where they can be efficiently utilized

recovery of the storage of actors which are inaccessible

transport for actors to and from secondary storage

We propose to develop automatic mechanisms to efficiently provide the above capabilities in real time. This means that in no case will a worker be disabled for a significant amount of time in order to perform the above housekeeping functions.

In pursuit of these goals, we have developed algorithms for a real-time distributed garbage collection and migration. The purpose is to maintain a suitable environment in which the actors can do their work.

IX -- Programming Environment

The APIARY will support implementations of EXTENDED LISP, SCHEME, ACT1, ETHER, and ultimately ADA when it reaches a suitable state of development. EXTENDED LISP [Weinreb and Moon: 1978] is a greatly extended dialect of MACLISP that features support for multiple processes, multiple overlapping windows [Engelbart: 1970, ITS TECO and EMACS: 1977, Swinehart: 1974, Ingalls: 1978, Teitelman: 1977, CADR: 1978], real-time garbage collection (using adaptations of the algorithms in [Baker: 1978] and [Bishop: 1977]), and message passing as in SMALLTALK and actors. SCHEME [Sussman and Steele: 1978] is a dialect of LISP which features lexical scoping and tail recursion (in these respects similar to PLASMA [Hewitt and Smith: 1975]). ACT1 [Hewitt, Attardi, and Lieberman: 1979] has been developed to make it more convenient to implement actors on distributed systems such as the APIARY. ETHER [Kornfeld: 1979] is a PLANNER-like system that features **parallel pattern-directed invocation** together with flexible resource control capabilities. ADA is a new language being developed by the DoD. Our goal is to have programs in *all* of the above languages be able to efficiently communicate with each other in the same APIARY.

We plan to continue the development of high-level languages (such as ACT1 and ETHER) with inherent parallelism in order to support the development of knowledge-based applications. We expect that it will be necessary to develop efficient broadcast protocols in order to support the parallel pattern-directed invocation in ETHER. The experience of developing successive generations of the APIARY should have an important effect on the evolution of these languages.

X -- Successive Generations of the APIARY

We are proceeding to develop an implementation of APIARY-0 on the MIT CADR MACHINE. This system consists of 64 workers each with 1 megabyte of local memory. The area mechanism on CADR is being used to partition the memory between the workers.

APIARY-0 will serve as a testbed for mechanisms proposed for incorporation in future hardware implementations using VLSI. The APIARY-0 implementation will help us determine the effect on performance of the following parameters:

number of workers
incorporation of specialized workers
topology of the APIARY
amount of memory for each worker

Subsequently we plan to use APIARY-0 to investigate the robustness of the system in the face of hardware failures of workers and communication links. We have designed a checkpoint and recovery system [Hewitt: 1979] for serialized actors which we plan to implement. Our recovery system builds on the ideas in [Gray: 1976] and [Lampson and Sturgis: 1979]. In order to facilitate the initial implementation of checkpoint and recovery, we plan to impose the following limitation: **At any given point a serialized actor resides in at most one worker in contrast to an unserialized actor which can reside in arbitrarily many workers.** In some circumstances there are efficiency advantages in allowing a serialized actor to reside on more than one worker.

The interaction between load balancing and locality of reference must be carefully investigated.

An APIARY will be said to be **overheated** if more concurrent activities have been spawned than can be efficiently handled using the available memory and processing power in the APIARY. We expect that **overheating** will be a significant problem which will have to be dealt with using appropriate **cooling** mechanisms [Hewitt: 1978 Kornfeld: 1979] to control the rate at which parallelism can be created.

The development of APIARY-0 will help us determine the specifications and kernel code size needed to provide

load balancing with neighbors
migration of actors to neighbors
local garbage collection
communication with neighbors

in each worker. The APIARY-0 implementation will enable us to more intelligently design hardware support for the above functions in VLSI for subsequent generations of the APIARY.

APIARY-1 (the successor to APIARY-0) will be a more general system that can run concurrently on several CADR machines. APIARY-1 will be a redesign which incorporates the lessons learned from the implementation and operation of APIARY-0. It will be a transitional system that serves as the bridge from APIARY-0 to later systems that are implemented in VLSI.

It is quite clear that VLSI will be required to make it feasible to have an APIARY with thousands of workers. Since the implementation of APIARY-0 and APIARY-1 will be on the CADR MACHINE, one natural route for the evolution of VLSI implementations to take is the development of a VLSI CADR MACHINE. Design studies will be undertaken to determine the feasibility of a VLSI APIARY worker based on a modification of the CADR MACHINE architecture. Our initial goal is to get an entire worker (consisting of a processor, adequate memory, and communication interface) on one board. We expect that it will take us several years to accomplish this goal. A pre-requisite is the development of good VLSI design tools such as the ones proposed in [Sussman, Holloway, and Knight: 1979].

XI -- CONCLUSIONS

The needs of knowledge-based systems have provided the driving force behind the initial design of the APIARY. In addition they will be the most significant factor guiding its subsequent evolution. Systems like the APIARY are needed to provide the quick response needed by interactive knowledge-based systems. Our approach to these applications distinguishes the APIARY from other multiprocessor systems that are currently being proposed by other research groups.

An important design constraint is that the hardware must efficiently provide for the implementation of knowledge-based systems. One of the ways in which the APIARY meets this design constraint is by multiplexing hardware devices. Packet-switching is used instead of circuit-switching in communication to make more efficient use of the available communication bandwidth. The processor of a worker is multiplexed in time so that a single worker can provide processing power for a large number of actors. We rely on load balancing between workers instead of attempting to partition programs into units that are suitable for individual processors.

Another distinguishing feature of our approach is that we believe that considerable special purpose VLSI will be necessary to create an implementation of the APIARY that provides a suitable base for knowledge-based systems in the 1980's. Therefore we propose to develop specialized VLSI chips for the APIARY. We expect the architecture of the VLSI workers which we eventually develop will be quite different from current generation computers.

XII -- ACKNOWLEDGEMENTS

Extensive conversations with Tom Knight, Richard Zippel, Jeff Schiller, and Jack Holloway have contributed to removing numerous bugs from the preliminary design of the APIARY. Howard Cannon, John Cocke, Bert Halstead, Danny Hillis, Jack Holloway, Ken Kahn, Tom Knight, Glen Miranker, Jim Stansfield, Gerry Sussman, and Dick Waters helped to improve the presentation. Peter Bishop, Henry Baker, and Bert Halstead originated many of the algorithms used in the design of the APIARY as part of their thesis research. Bert Halstead showed how to properly diagram the two-dimensional embedding of the Hypertorus Topology.

The preliminary design of the APIARY presented here is made possible only because of the hardware and software base provided by the MIT CADR MACHINE which was conceived and initially designed under the leadership of Richard Greenblatt, Jack Holloway, and Tom Knight. A preliminary version of the manual [Weinreb and Moon: 1978] describes many of its current capabilities. Henry Lieberman has developed a preliminary implementation of primitive serializers on the machine. Bill Kornfeld is transferring his implementation of ETHER-0 from the PDP-10 to the CADR MACHINE.

Message Passing Architecture has been pioneered in the actor machine architectures of [Hewitt, Bishop, and Steiger: 1973; Steiger: 1974, Bishop: 1977, and Baker: 1978] and in the Mu-Network of [Ward: 1978 and Halstead: 1978]. **Interest groups** are generalizations and further developments of **inter-area links** [Bishop: 1977] and **reference trees** [Halstead: 1978]. The **interest group** concept was developed jointly with Tom Knight.

The Columbia Homogeneous Parallel Processor Design [Sullivan and Bashkow: 1977 and Sullivan, Bashkow, and Klappholz: 1977], the architecture of the CM* projects at CMU [Swan, Fuller, and Siewiorek: 1977], and data flow machines [Dennis and Misunas: 1975, Rumbaugh: 1975, Arvind and Gostelow: 1977] have also strongly influenced our work.

Message Passing Architecture builds on recent work in message passing systems including the following:

quasi-parallel message passing languages [Dahl and Nygaard: 1968; Ingalls: 1978; K. Kahn: 1978]

parallel message passing languages [Hewitt and Smith: 1975; Kahn and MacQueen: 1977, Hewitt and Atkinson: 1978;

Hoare: 1978; Hewitt, Attardi, and Lieberman: 1979; Kornfeld: 1979; Svobodova, Liskov, and Clark: 1979; Ward: 1978]

message passing theories [Hewitt and Baker: 1977, Francez, Hoare, Lehmann, and de Roever: 1978, Milne and Milner: 1979, Hewitt, Attardi, and Lieberman: 1979]

The concept of an actor has been developed in an attempt to synthesize a unified system that combines the message passing, pattern matching, and pattern directed invocation and retrieval in PLANNER [Hewitt: 1969; Sussman, Charniak, and Winograd: 1972; Hewitt: 1971], the modularity of SIMULA [Dahl and Nygaard: 1968], the message passing ideas of an early design for SMALLTALK [Kay: 1972], the functional data structures in the lambda calculus based programming languages, the concept of concurrent events from Petri Nets (although the actor notion of an event is rather different than Petri's), and the protection inherent in the capability based operating systems with their protected entry points. The subclass concept originated in [Dahl and Nygaard: 1968] and adapted in [Ingalls: 1978] has provided useful ideas.

XIII -- BIBLIOGRAPHY

Arvind and Gostelow, K. "Some Relationships between Asynchronous Interpreters of a Dataflow Language" IFIP Working Conference on Formal Description of Programming Concepts. 31 July to 5 August 1977.

Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" CACM. August 1978. pp. 613-641.

Batcher, K. E. "Sorting Networks and their Applications" SJCC. April 1968. pp 307-314.

Baker, H. J. Jr. "Actor Systems for Real-time Computation" MIT Ph.D. Thesis. Laboratory of Computer Science Technical Report MIT/LCS/TR-197. March 1978.

Baker, H. J. Jr. and Hewitt, Carl "Incremental Garbage Collection of Processes" MIT A.I. Memo 454. December, 1977.

Bishop, P. B. "Computer Systems with a Very Large Address Space and Garbage Collection" MIT Ph.D. Thesis. Laboratory of Computer Science Technical Report MIT/LCS/TR-178. May 1977.

Dahl,O. J. and Nygaard, K. "Class and Subclass Declarations" In Simulation Programming Languages J. N. Buxton (Ed.) North Holland. 1968. pp. 158-174.

Dennis, J. B. and Misunas, D. P. "A Preliminary Architecture for a Basic Data Flow Processor" The 2nd Annual Symposium on Computer Architecture. Houston. pp. 126-132.

Engelbart, D. C. "Advanced Intellect-Augmentation Techniques" SRI Final Report. July 1970.

Feldman, J. A. "A Programming Methodology for Distributed Computing (among other things)" TR9. Computer Science Dept. Univ. of Rochester. September 1976.

Francez, N.; Hoare, C. A. R.; Lehmann, D. J.; and de Roever, W. P. "Semantics of Nondeterminism, Concurrency, and Communication" Working Paper. August 1978.

Friedman, D. P. and Wise, D. S. "A Constructor for Applicative Multiprogramming" Technical Report No. 80. Indiana University. January 1979.

Gray,J. N. "Notes on Data Base Operating Systems" in Operating Systems, An Advanced Course American Elsevier. 1978.

Halstead, B. "Multiple-Processor Implementations of Message-Passing Systems" MIT Laboratory of Computer Science Technical Report MIT/LCS/TR-198. 1978.

Hewitt, C. "PLANNER: A Language for Proving Theorems in Robots" IJCAI-69. Washington, D. C. May 1969. pp 295-302.

Hewitt, C. "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot" Unpublished doctoral dissertation. MIT. 1971.

Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages" A.I. Journal. Vol. 8. No. 3. June 1977. pp. 323-364.

Hewitt, C. and Smith, B. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, #1. March 1975. pp. 26-45.

Hewitt, C. and Baker, H. "Laws for Communicating Parallel Processes" Proceedings of IFIP Congress 77. Toronto, August 8-12, 1977. pp. 987-992.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Specifying and Proving Properties of Guardians for Distributed Systems" MIT AI Lab Working Paper 172. December 1978. Revised April 1979. Proceedings of International Symposium on Semantics of Concurrent Computation. Evian les Bains, France. July 1979.

Hewitt, C. "Concurrent Systems Need Both Sequences and Serializers" MIT AI Lab Working Paper 179. December 1978. Revised February 1979.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Security and Modularity in Message Passing" MIT AI Lab Working Paper 180. December 1978. Revised February 1979.

Hewitt, C. "Checkpoint and Recovery in Actor Systems" MIT AI Lab Working Paper 180. Forthcoming 1979.

Hibbard, P. G.; Hisgen, A.; and Rodeheffer, T. "A Language Implementation Design for a Multiprocessor Computer System" IEEE/ACM 5th Annual Conference on Computer Architecture. April 1978. pp 66-72.

Hoare, C.A.R. "Communicating Sequential Processes" CACM, Vol 21, No. 8. August 1978. pp. 666-677.

Ingalls, D. H. H. "The Smalltalk-76 Programming System Design and Implementation" Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. January 23-25, 1978. Tucson, Arizona. pp. 9-16.

Kahn,K. M. "DIRECTOR Guide" MIT AI Memo 482. June 1978.

Kahn,G. and MacQueen, D.. "Coroutines and Networks of Parallel Processes" IFIP-77. Toronto. August 8-12, 1977. pp. 987-992.

Keller, R. M., Lindstrom, G., and Patil, S. "A Loosely-coupled Applicative Multi-processing System" 1979 NCC. pp 861-870.

Kornfeld, W. A. "Using Parallel Processing for Problem Solving" IJCAI-79. Tokyo, Japan. August 1979.

Lampson, B. W. and Sturgis, H. E. "Crash Recovery in a Distributed Data Storage System" Forthcoming. 1979.

Milne, G. and Milner, R. "Concurrent Processes and their Syntax" JACM. Vol 26, No. 1. pp 302-321. April 1979.

Moravec, H. P. "Intelligent Machines: How to get there from here and What to do afterwards" Computer Science Dept. Stanford University.

Pease, M. C. "The Indirect Binary n-Cube Microprocessor Array" IEEE Transactions on Computers. Vol. C-26. No. 5. May 1977.

Rich,C.; Shrobe, H. E.; Waters, R. C.; Sussman, G. J.; and Hewitt, C. E. "Programming Viewed as an Engineering Activity" MIT A.I. Memo 459. January 1978.

Rumbaugh, J. E. "A Parallel Asynchronous Computer Architecture for Data Flow Programs" MIT Ph.D. dissertation. Project MAC Technical Report TR-150. May 1975.

Steele, G. L. and Sussman, G. J. "The Revised Report on SCHEME a Dialect of LISP" Artificial Intelligence Memo 452. January 1978.

Steiger, Richard, "Actor Machine Architecture" Unpublished Masters Thesis. MIT EECS Dept. June 1974.

Sullivan, H. and Bashkow, T. R. "A Large Scale, Homogeneous, Full Distributed Parallel Machine, I" Proceedings of 4th Annual Symposium on computer Architecture. March 1977. pp 105-117.

Sullivan, H.; Bashkow, T. R.; and Klappholz, D. "A Large Scale, Homogeneous, Full Distributed Parallel Machine, II" Proceedings of 4th Annual Symposium on computer Architecture. March 1977. pp 118-124.

Sussman, G. J.; Winograd, T.; and Charniak, E. "MICRO-PLANNER Reference Manual" MIT AI Memo 203A. December 1972. Cambridge, Mass.

Sussman, Holloway, and Knight. "Computer Aided Evolutionary Design for Submicron Digital Technology" MIT AI Memo 526. May 1979.

Swan,R. J., Fuller, S. H., and Siewiorek, D. P. "Cm*--A Modular, Multi-Microprocessor" AFIPS Conference Proceedings 46. 1977.

Swinehart, D. "COPILOT: A Multiple Process Approach to Interactive Programming Systems" Stanford AI Memo 230. July 1974.

Ward,S. A. "The MuNet: A Multiprocessor Message-Passing System Architecture" Seventh Texas Conference on Computing Systems. Houston, Texas. October, 1978.

Ward,S. A. "An Approach to Real-Time Computation" Seventh Texas Conference on Computing Systems. Houston, Texas. October, 1978.

Weinreb, D. and Moon, D. "LISP MACHINE Manual" Preliminary version. November 1978.