プログラミング—言語・基礎・実践— 3－14
（1991. 7. 17）

A Debugger for the Parallel, Object-Oriented Language $\mathcal{A}'\mathcal{UM}$-90

Charles Fabian, Tsutomu Maruyama, Koichi Konishi, Akihiko Konagaya

NEC Corporation, C&C Systems Laboratory

1-1, Miyazaki 4-Chome Miyamae-ku Kawasaki Kanagawa 213 JAPAN

A major obstacle of concurrent programming is the complexity involved in program debugging. A debugger for such a system must be powerful enough to overcome non-determinism and other difficulties associated with debugging such systems. It is also important that the debugger does not alter execution behavior. We have implemented an event-based debugger for $\mathcal{A}'\mathcal{UM}$ which provides execution histories, deterministic execution replay and a graphical interface. Our debugger provides this powerful functionality while minimizing its effect on performance. This balance is achieved through separation of debugger operation into history recording and active debugging modes of operation. In this report we present the design of this debugger as well as some preliminary analysis of its overhead.

並列オブジェクト指向言語 $\mathcal{A}'\mathcal{UM}$-90 のデバッガ

Charles Fabian, 丸山 勉、 小西 弘一、小長谷 明彦

日本電気(株) C&C システム研究所

並列言語のデバッグは、並列言語の持つ非決定性等のためにより困難である。このため、並列言語のデバッグには、非決定性およびそれに伴う種々の問題点を効率的に扱うことができるデバッガが不可欠である。我々は、並列オブジェクト指向言語 A'Um 用のデバッガを作成した。本デバッガは、イベントに基づく実行ヒストリの記録、それを用いた実行過程の再現およびグラフィックインタフェス等の機能を持つ。本デバッガでは、デバッグ機能を、実行ヒストリ管理を行なう部分と、それを用いて実際にデバッグを行なう部分とに分けるにより、デバッガによる実行時への影響を最小限に抑えている。本報告では、デバッガの設計方針とその評価について述べる。

1 Difficulties of Distributed Debugger Design

There are three major issues to be considered for the design of an effective debugger for a parallel, distributed system such as $\mathcal{A}'\mathcal{UM}$. First, the debugger must be powerful enough to help the user to overcome the added difficulties associated with debugging concurrent computation. Potential non-determinism means that, even given the same inputs, a program may have different results for different executions. Erroneous behavior may be difficult to repeat, perhaps occurring only once out of every one hundred executions — a debugging nightmare. The debugger should provide aids, such as deterministic re-execution, to help overcome non-determinism.

The second issue is that a debugger must minimize the *probe effect*. The probe effect refers to the interference by an external influence, such as a debugger, has on the computation and the possibility that such interference will alter the behavior of computation. In the parallel case, the probe effect is particularly hazardous, as a slight difference in execution timing may greatly alter execution behavior. While it is impossible to design a debugger which fully avoids the probe effect, the probe effect may be reduced to a tolerable level by minimizing the memory and computation cycle requirements of the debugger.

Finally, of course the debugger must be easy to use. While this is not easily quantifiable, a well designed user interface is a major factor when considering usability. While ease of use is important for both sequential and concurrent debuggers, it is particularly helpful for the concurrent case because of the much greater complexity of parallel/distributed systems.

There is a direct conflict between the second goal of reducing the *probe effect* (i.e. minimizing the debugger) and the other two goals of providing a powerful and easy to use debugger (i.e. increasing the size and complexity of the debugger). Thus, the difficulty of designing a debugger is finding a good balance between reducing the probe effect and providing powerful functionality.

Despite this apparent trade-off we feel that it *is* possible to design a debugger which is quite powerful and at the same time reduces the probe effect to tolerable levels such that program behavior remains consistent with actual execution. We believe that this may be achieved through the realization that debugger operation should be separated into two distinct modes of operation. We call these modes the history recording and the active debugging modes.[1] We have attempted to design such a debugger for $\mathcal{A}'\mathcal{UM}$. But before describing our debugger we must first briefly introduce $\mathcal{A}'\mathcal{UM}$.

2 Introduction to $\mathcal{A}'\mathcal{UM}$

$\mathcal{A}'\mathcal{UM}$ is a stream based, object-oriented language for parallel execution, similar in many ways to the actors model. [1]

Each object is a separate process, with its own thread of control — many objects may be executed concurrently. It has an input queue for messages, and when a message arrives on this queue the object is awakened to be scheduled for execution. If the physical processor is busy, the object may have to wait before being scheduled for execution; while waiting newly arriving messages will continue to enqueue on the queue. An object executes its messages in the strict order of their arrival on its queue. Note that this is independent of the order which these messages may have been sent.

The execution of a message is an atomic event, an object executes these messages sequentially. [2]

By *stream based*, we mean that messages are channeled to objects through queues which we call *streams*. Messages are sent in a "send-no-wait" asynchronous manner. The primary work of an

[1] Halstead uses the terms *production mode* and *trace mode* for his Multivision system. [4]

[2] For brevity, in this paper refer to the "execution of the method for a particular method" as "execution of the message." Likewise, "execution of an object" refers to the "execution of a method for the object."

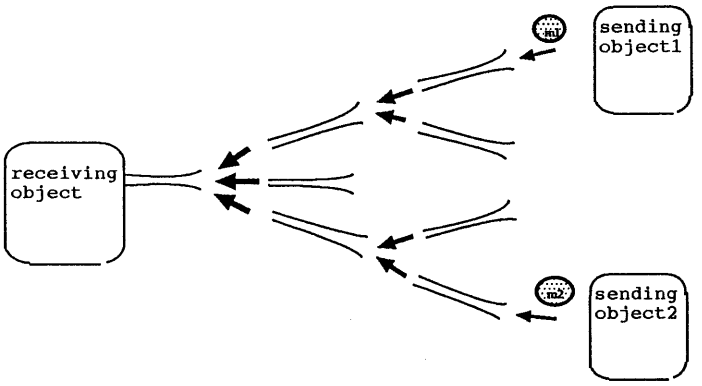

Figure 1: Stream Network

$\mathcal{A}'\mathcal{UM}$ method is to build up a network of streams by connecting them one to another. The above mentioned input queues of objects are also streams.

In addition to building up this stream network, methods may also create objects and, of course, send messages. Messages may carry a number of arguments; these arguments are references to objects or streams. An object is terminated when it receives a close message on its input stream.

Objects may also have a sense of state — state is represented by fixed length list of streams which the object has reference to. (Each such stream reference is called a *slot*.)

Because objects execute message exactly in the order they are found on its input stream, the only source of non-determinism is at the "joints" of the stream network, where the arrival of messages from two separate streams can not be determined.

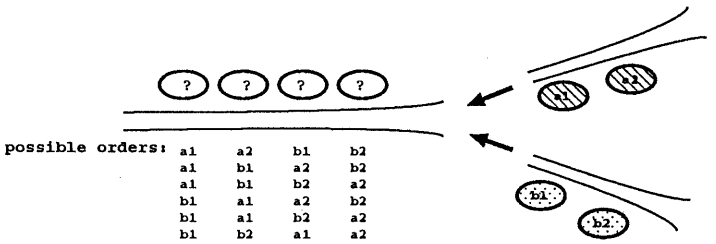

Figure 2: Message Ordering

3 The Debugger

Our debugger is an event-based debugger [5] which provides an execution history, deterministic replay and a graphical user interface. Traditional breakpoint and tracing mechanism are also provided, as well as message level and instruction level execution stepping. The user may examine arguments, reorder messages, and exercise control over the scheduler to try and isolate and correct erroneous program behavior.

Debugger operation is divided into two modes: history recording and active debugging modes. During the recording mode, execution occurs just as in normal $\mathcal{A}'\mathcal{UM}$ execution, except that a history of synchronization events is recorded for possible execution replay.

3.1 History Recording

The debugger records an execution history for each object, called an *object history*. The only events that must be recorded in this history are message executions. From the order of message execution, the entire state of the object can be reconstructed.

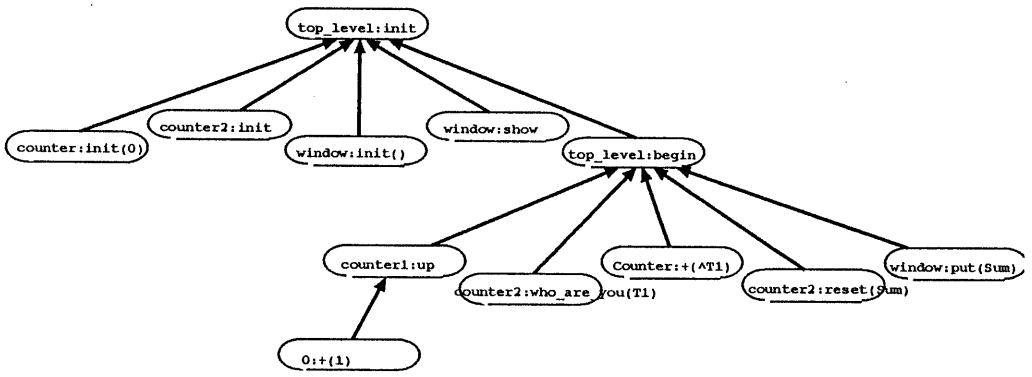

Figure 3: Message Backtrack History

Complementing the object histories is a *message hierarchy tree*. This is merely a representation of the causality of a program. The *parent* of each message is recorded — the parent being defined as the message whose execution resulted in the creation and sending of this message. The result is a tree representing the control structure of the program, analogous to the procedure invocation stack of a language such as LISP. Figure 3 is an illustration of a message hierarchy tree.

The object histories provide only a partial ordering over message execution. Each history only specifies the execution ordering of messages executed by that object. The object histories imply no ordering constraints between messages executed by different objects. However, a complete ordering over all message executions would be expensive to record, considering that on a distributed system there is usually no global clock. Further, such an ordering would enforce non-existing constraints on execution ordering. For example, there are many cases where two messages executing on different objects whose execution ordering are independent; either one could execute first, or they may even execute simultaneously, without affecting the execution outcome. A complete ordering would unnecessarily place the constraint that such messages execute in the same order in which they executed during the original execution.

The only constraints between messages executing on different objects are the result of message creation. With the addition of the *message hierarchy tree*, which reflects these creation dependencies, it is possible to deterministically reconstruct execution from the object histories. Using this scheme, only actual constraints are specified for re-execution.

Besides forming the foundation of execution replay, the histories are also useful aids in understanding and debugging the program. For example, during the active debugging mode, the user may use the message hierarchy tree to examine the causality of the program. The user may also backtrack up this tree, like backtracking up the call stack, to examine in more detail the other messages in the causality chain.

3.2 Active Debugging

The active mode of the debugger is for interactive examination of program execution. This mode may be entered immediately upon encountering an execution error during history recording mode, or active mode may entered with the replay of execution.

The execution replay mechanism uses the recorded histories to deterministically re-execute the program. With this mechanism, the user can confidently reproduce erroneous program behavior. It is possible that two messages which do not have timing constraints between them will execute in a different relative order during re-execution. However, this would not have any effect on the execution outcome, any erroneous behavior would be reproduced (provided that there is not a bug in the actual system).

The debugger is also equipped with a message reordering mechanism. While the execution is suspended, the user is able to reorder the messages waiting in an object's input queue. In this way, the user can experiment with different event orderings, producing different execution outcomes. Thus, the user may have an active role in debugging.

The user can also exercise control over the scheduler, which determines the order in which "ready" objects execute. As with message reordering, the user may use this ability to experiment with different possible pathes of execution.

The debugger is also equipped with the traditional breakpoint and tracing mechanisms. Breakpoints may be set to occur on several different conditions, such as scheduling an object or execution of a particular method.

3.3 Graphical Interface

In order to enhance use of the debugger a visual interface is also provided. Users select certain objects to be displayed on the screen while they perform active debugging. The displayed object, along with its execution history and its input stream is graphically displayed. From this display, the user may selectively step through the execution, reorder messages in the input queue, and examine the arguments of messages in more detail. While stepping through message execution, changes in an object's state or input queue is automatically reflected by updating the display.

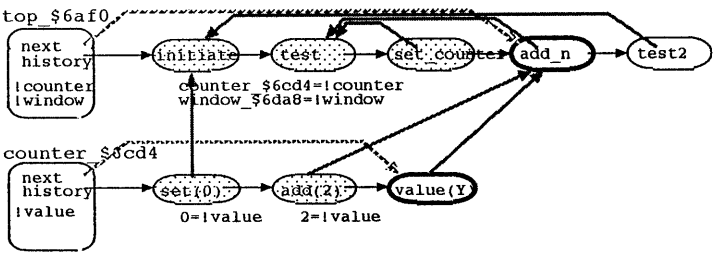

Figure 4: Example Visual Display

Figure 4 illustrates the basic concept of the graphical interface. In the illustration two separate objects are being displayed. Each object's history, and input queue are displayed. The shaded messages have already been executed. The currently executing messages are outlined in bold. The remaining messages are waiting in their respective object's queue. Changes in slot values are shown below the messages which caused their change. Also, an arrow represents the parent of messages wherever possible.

4 Implementation

Implementation of the message hierarchy tree requires increasing the internal representation of messages by one field to hold the reference to the parent message.

The object history also can be maintained easily: the input queue is basically the same form as the history. Instead of removing messages from the front of the queue as they are executed, a

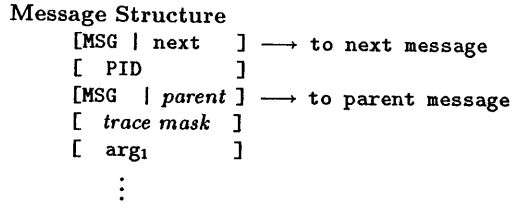

Figure 5: Modified Message Structure.

pointer is moved along the queue to indicate the next un-executed message. Thus the implementation requires adding just one field to the internal representation of the object structure.

Of course the memory used to internally represent messages may not be reclaimed once the message has executed, unlike in regular execution. On the other hand, the accurate replay of execution requires that the entire history be preserved. This would present no problem if unlimited memory were available, but this is not the case.

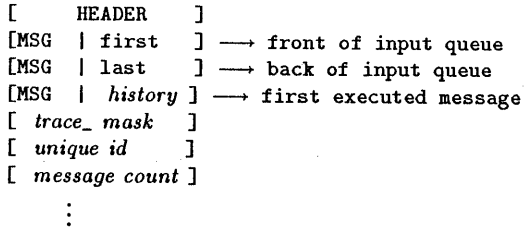

Figure 6: Modified Object Structure.

The execution replay mechanism is also straightforward to implement. Only the main loop of each processor must be modified to guarantee that messages execute in the correct order. Instead of the normal behavior of executing the first message waiting in an object's queue, the object first consults the history to determine which message should be executed next. If the correct message is not yet available on the queue, the object enters the waiting state, as if it had no messages on its queue.

5 Logging Schemes

Preserving a history of execution is fairly costly in terms of memory. In normal $\mathcal{A}'\mathcal{UM}$ execution, the memory used to internally represent a message may be reclaimed as soon as the message is executed. With the debugger, that memory space not be reclaimed, the message instead remains in the object's history of executed messages.

In order to avoid running out of memory we can save the object histories to a disk file. However, writing to a file is a slow operation, so we are faced with a time-space trade-off. If it is necessary to write the histories to disk, the information written to disk should be minimized as much as possible.

To be able to accurately recreate execution, for each message we must be able to determine its destination object, its relative execution order, and its parent message. (As stated earlier, a complete ordering on message executions is not required to be able to recreate execution behavior. Only specification of the relative ordering of messages executed by the same object is required.)

While it is possible to write a log entry for each message as it is executed, we do not feel that this would be a good solution. In order to uniquely describe each message we would have to devise some identification scheme, which would require a lengthy entry to describe each message.

Instead, log entries for messages are written only when the executing object is terminated. Thus, all messages in the same object history are logged at the same time. This has two advantages:

1. Writes to disk occur less often, with more data written at each write.
2. Identification of each message is simplified by the fact that all messages executed by the same object are recorded together; for each message only the its parent and its relative position in the object history need to be recorded. The relative position is taken care of by the log

For this scheme, each object is assigned a unique identifier. Each message is identified by its executing object's ID and the execution number of the message. For log entries, a message is identified by its parent and a small integer to distinguish it from other children of the parent. Thus, the log for a history consists of the object's own ID, followed by a list of parent IDs, one for each message in the history.

6 Reduction of Probe Effect

As we stated, the *probe effect* is the chance that the debugger may alter execution behavior. With a concurrent system, even the slightest change in timing may result a drastically different execution behavior; thus, it is impossible to fully avoid the probe effect when designing a debugger. However, we can reduce the probe effect to a tolerable level; *tolerable* has different meanings with regard to active mode and recording mode.

In active mode, we wish to reduce the probe effect such that

1. with the debugger enabled, all execution behavior exhibited will be *correct* with respect to the semantics of the $\mathcal{A}'\mathcal{UM}$ language.
2. any execution behavior that is possible with the debugger disabled would also be attainable with the debugger. In other words, using the debugger the user should be able to examine and experiment with every possible execution behavior that the actual system might exhibit.

These conditions do *not* guarantee that execution behavior exhibited with and without the debugger will always be the same, but this is acceptable because in a non-deterministic system it is also not guaranteed that two separate executions will have the same result. On the other hand, the above conditions do guarantee that all legal execution outcomes are attainable, if the user wished to experiment with them. This is the most that can be asked of a debugger tool.

Guaranteeing that only semantically correct execution behavior is exhibited is no more difficult than guaranteeing that the actual $\mathcal{A}'\mathcal{UM}$ system produces only legal behavior. The message reordering mechanism and manual control over the object scheduler allow the user to generate all possible execution behavior, thus guaranteeing the second condition.

When in recording mode, tolerance of the probe effect involves another condition in addition to the above. The execution must require a noticeably longer amount of execution time. Unless the history recording is enabled before the execution begins, the debugger's replay mechanism would be unable to recreate the execution behavior. Therefore we recommend that the debugger always be *enabled* during the program development cycle. However, this advice will not be heeded unless performance is not greatly affected by the debugger.

It can not be quantitatively stated whether this condition has been met; however in the next section we describe some preliminary analysis of the overhead introduced by the debugger. We

would like to point at this stage that an important decision in the design of our debugger is the distinction between recording mode and active mode. During active mode there is a higher tolerance for debugger overhead, and thus we are able to provide greater functionality. On the other hand, during recording mode, functionality is not important; we must minimize the interference of the debugger to minimize the probe effect. This concept may be generalized to the design of debuggers for any concurrent system.

7 Analysis

We have conducted some measurements to quantify the overhead of the $\mathcal{A}'\mathcal{UM}$ debugger. The critical comparison to make is between the performance of the system in record mode and the performance of the system without the debugger. For the active mode, it is only necessary to verify that execution speed is adequate for human interaction. The user should not experience any noticeable waiting time if possible. Both the memory and time requirements for active mode is expected to be quite large.

We examine the dynamic memory consumption of the $\mathcal{A}'\mathcal{UM}$ debugger in recording mode and compare it with that of regular execution.

7.1 Space Analysis

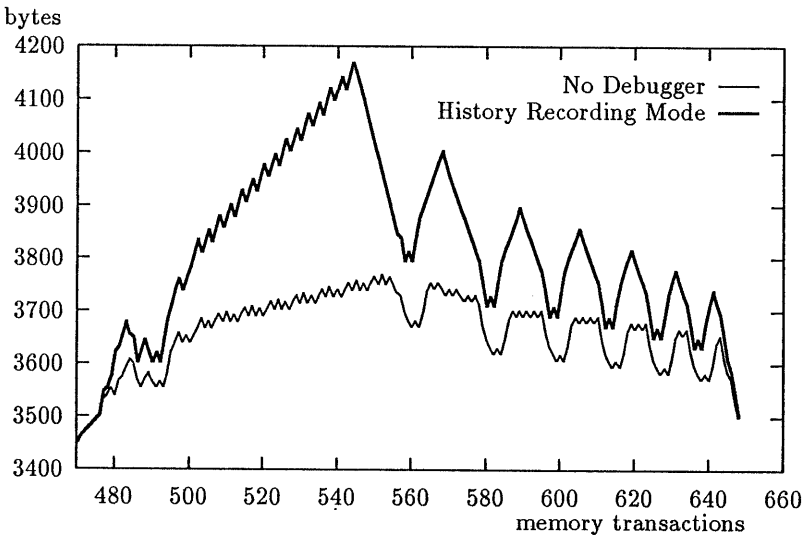

Figure 7: Dynamic Memory Usage for Primes 20

As mentioned, we expect the memory requirements of the debugger to increase, particularly the dynamic memory usage of the system. How much more memory used would depend upon the nature of the application. The worst case would be for computations which have objects with long history chains. The memory used to represent the messages of the history would be collected soon after the message's execution under normal execution; with the debugger's histories, such memory can not be reclaimed until the entire history is through executing.

We measure the memory requirements for the debugger using the benchmark program for finding the first n prime numbers using the sieve of erosthothenes. This program acts as an approximation for "worst case" conditions of memory consumption because the objects' histories are fairly long chains of messages. Our dynamic memory usage measurements are results from executing the program on

only one process. This way, it is easier to see the total memory usage; the analysis can be extended to execution on multiple processors.

For the sieve of erosthothenes, at the point of maximum memory usage the available memory requirement approximately doubles. (This corresponds to what might be the expected behavior for the algorithm.) Figure 7 plots the dynamic memory usage for both the original $\mathcal{A}'\mathcal{UM}$ system and the debugging system in history mode for finding the first 20 primes. In this plot the x axis is the number of memory allocate and deallocate transactions done by the system. The creation of an object, message or stream would result in memory allocation transactions. When the memory for representing an object, message or stream is reclaimed, a deallocate transaction results. The y axis represents the total amount of memory in use. Figure 8 plots the dynamic memory consumption for

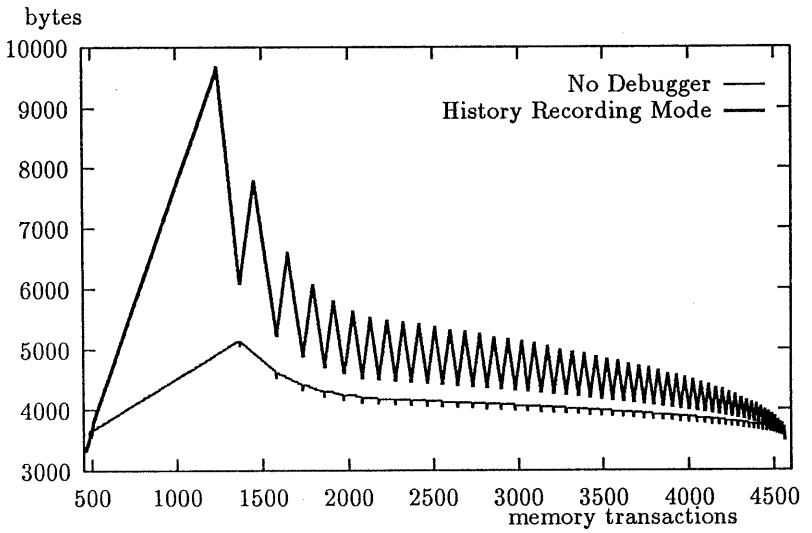

Figure 8: Dynamic Memory Usage for Primes 250

finding the first 250 primes. We find that for the sieve of erosthothenes, the maximum amount of memory used is consistently less than three times that required by the original system.

8 Conclusion

We have presented the design of a debugger for the concurrent system $\mathcal{A}'\mathcal{UM}$. The debugger, with its object histories, message passing history and replay mechanism, has been designed to overcome program non-determinism by allowing the user to deterministically replay execution. The debugger's graphical interface aids the user in visualizing computation, somewhat reducing the complexity of concurrent debugging. The debugger has also been designed to minimize the probe effect.

We have been able to achieve a good balance of the trade-off between providing high functionality and minimizing the probe effect by separating operation of the debugger into two modes of operation: history recording mode and active debugging mode. During history recording mode the computation of the debugger, and thus, its interference on the actual computation, is minimized. During interactive mode, when computation power is expected to be abundantly available on a parallel machine, the expensive computation of the debugger, such as providing a graphical interface, is performed.

The debugger design is not without flaws. Large programs, which already push the $\mathcal{A}'\mathcal{UM}$ system to its memory limits, would certainly cause problems for the debugger. However, such conditions

may be easily checked. In the future we would like to more closely examine other logging schemes, such as the immediate logging of messages when executing. We may also incrementally logging histories, say every 1000 message executions.

Further possible areas of research for the $\mathcal{A}'\mathcal{UM}$ debugger project includes further improving the user interface and possibly providing filters which can recognize, group, and classify events. Also another area of research would be ways of integrating static analysis, compile time information about the program into the $\mathcal{A}'\mathcal{UM}$ debugger.

References

[1] Agha, G.A., *Actors: a Model of Concurrent Computation in Distributed Systems,* M.I.T. Press (1986).

[2] Peter Bates., Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior, *Proceedings of Workshop on Parallel and Distributed Debugging.* SigPlan Notices Vol. 24, No 1. January 1989.

[3] Elshoff, I.J.P., A Distributed Debugger for Amoeba, *Proceedings of Workshop on Parallel and Distributed Debugging.*

[4] Halstead, Robert H., Kranz, David, and Sobalvarro, Patrick, "Multivision: A Tool for Visualizing Parallel Program Execution," Talk given at Tokyo University, April 8, 1991.

[5] Lin, Chu-Chung and LeBlanc, Richard J., Event-based Debugging of Object/Action Programs *Proceedings of Workshop on Parallel and Distributed Debugging.*

[6] McDowell, Charles E. and Helmbold, David P., Debugging Concurrent Programs, *ACM Computing Surveys* Vol. 21, No 4. December 1989.

[7] Yoshida, Kaoru, *$\mathcal{A}'\mathcal{UM}$: A Stream-Based Concurrent Object-Oriented Programming Language* Keio University, PhD thesis. 1990.