ソフトウェア基礎論　３３－１１
プログラミング言語　２３－１１
（1989. 12. 8）

分散プログラムのデバッグにおける大域的条件について

真鍋 義文　　今瀬 真
NTT ソフトウェア研究所

あらまし

再現をベースにする分散プログラムデバッガにおいて、ブレークポイント、トレース条件の指定に複数のプロセスにまたがる大域的条件を導入する。ブレークポイント条件式が単一プロセス条件の積の形の場合に、条件が成立する最初の地点で停止するアルゴリズムを示す。また、条件式が単一プロセス条件の和の形の場合には、最初の地点で停止するアルゴリズムは存在しないことを示す。また、積の形、和の形それぞれのトレース条件式についてトレースを行うアルゴリズムを示す。

Global Condition in Debugging Distributed Programs

Yoshifumi Manabe　　Makoto Imase
NTT Software Laboratories
3-9-11 Midori-cho, Musashino-shi, Tokyo 180 Japan

ABSTRACT

This paper describes facilities for a distributed program debugger based on the instant replay technique. Most distributed program debuggers restrict the predicate for specifying the breakpoints and selective trace conditions to an expression for one process. This paper introduces two kinds of distributed global predicates (Conjunctive and Disjunctive Predicates) which are expressions related to plural processes. An algorithm is shown which halts at the first global state in which a Conjunctive Predicate is satisfied. It is shown that it is impossible to halt at the first state, but possible to halt at some state in which a Disjunctive Predicate is satisfied. An algorithm for a selective tracing function is also shown when a Conjunctive or Disjunctive Predicate selection condition is given.

1 Introduction

A distributed computing system is a collection of communicating and cooperating processes which work towards a common goal. The software for these systems is a collection of programs, each of which corresponds to a process. These programs operate in an integrated fashion to achieve the common system goal.

Debugging distributed programs is considerably more difficult than debugging a sequential program, because the execution behavior in response to a fixed input may be indeterminate, with the results depending on a particular resolution of race conditions existing among processes.

The indeterminism of the distributed system prevents some distributed program debugging packages from recreating the event sequences that precede errors [1] [8] [9] [15]. Some packages record a trace and offer the programmer different facilities for analyzing the trace information [1] [8]. Other packages provide a mechanism for stopping execution with an assertion violation [9] [15].

The disadvantage of these techniques is that all information needed to diagnose the program errors that might arise must be collected during a single execution. There is no mechanism for gathering additional information about an error after it is observed.

As another approach, some packages have a mechanism that guarantees reproducible behavior of the distributed programs and provides cyclic debugging techniques, in which the programs are executed until an error manifests itself, the programmer then postulates a set of underlying causes for the error, trace statements or additional breakpoints are inserted to gather more information about the causes of the error, and the program is reexecuted [4] [5] [12] [14]. This paper also considers a debugger based on a cyclic debugging technique.

To allow re-execution, BugNet [5] and Recap [14] record all input values, such as the contents of messages received, the values of shared memory locations referenced, the values of timers referenced, and all events that affect the state of the process for each process. This approach is convenient in the sense that each process can be reexecuted independently. However, logging all events requires a large amount of storage[14].

If the distributed programs do not contain nondeterministic statements such as asynchronous interrupts or time dependent statements such as reading a clock, the size of the log can be reduced drastically. When each process is supplied the same input values, which correspond to the contents of messages received or the values in shared memory locations referenced, in the same order during successive executions, it will produce the same output values in the same order. Each of those output values may then serve as an input value for some other process. Therefore, by ensuring that each process sees the same input values at every step of execution, successive executions will exhibit the same behavior.

'Instant replay'[12] is based on this premise. In the monitoring phase, the order of input events such as receiving messages and referencing shared memories is recorded for each process. In the replay phase, this mechanism ensures that each process reads the inputs in the order recorded.

Cooper proposes recording of the order of receive message events based on message passing. According to his research, the effect of this recording on the message-receiving mechanism is to slow it down by at most 2.5%[4]. Thus, the recording mechanism can also be used as a constant background monitoring of programs believed to be correct. The record is used when an error is detected which was not found in the debugging phase. For systems based on shared-memory communication, LeBlanc and Mellor-Crummey propose an effective algorithm for recording a partial order of shared memory access events, and show the possibility of constant monitoring[12]. Takahashi proposes an improved algorithm and compares the efficiency and power of various algorithms[16]. This paper discusses various facilities for a debugger based on the instant replay technique.

Debuggers should provide the following dynamic tracing functions[7].

- setting breakpoints and halting
- tracing
- single-step execution
- examination of state

The examination of state for a distributed debugger can be easily implemented by a method similar to that used in sequential program debuggers. The other functions require further prudent consideration.

To set breakpoints, many sequential program debuggers introduce logical predicates, which are written at the source-code level to describe relationships among components of the current program state (e.g., $J < 10$ and $A(I) = M$). When the specified predicate becomes true, the execution is stopped and the control is transferred to the programmer's terminal. Most distributed program debuggers restrict the predicate to an expression for one process[4][12]. However, distributed processes may cause many kinds of errors in which the over-all system has an error but each individual process has no error. To cope with such errors, global predicates, that is, expressions related to plural processes, should be introduced. Global predicates for non-replay debuggers are discussed in [13]. However, there is no discussion of global predicates for instant-replay debuggers. This paper considers global predicates for specifying breakpoints and proposes a halting algorithm for the breakpoints.

Some debuggers have a selective trace function, which prints, during execution, the values of specific variables and

the executed statement numbers, not at every step, but at certain points of interest. This paper introduces a selective trace function based on point selection according to global predicates.

In Section 2, after presenting models for a distributed system, the instant-replay based debugger is explained. Section 3 discusses halting by breakpoints and Section 4 covers selective traces. Section 5 discusses single-step execution and other facilities peculiar to the distributed debugger. Section 6 presents conclusions and describes further studies.

2 Model Definitions

2.1 Distributed System Model

In this paper, it is assumed that values exchanged between processes depend only on the initial values in each process and the order in which processes communicate. That is, the deterministic nature of processes is assumed. Stated somewhat differently, there are no nondeterministic statements such as asynchronous interrupts and no time dependent statements such as reading a clock and timing out from a semaphore in the program.

The distributed system execution model is the same as that proposed by Lamport[10], which is based on message-passing communication. A distributed system consists of a finite set of processes and a finite set of channels. Channels are assumed to have infinite buffers, to be error-free and to be FIFO. The delay experienced by a message in a channel is arbitrary but finite.

Each process consists of a sequence of distinct events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one event. We are assuming that the events of a process form a sequence, where a is before b in this sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.

Sending a message and receiving a message are considered as events and are called a *send event* and a *receive event*, respectively. We can then define the "happened before" relation, denoted by "$<$", as follows. (In [10], $\rightarrow$ is used rather than $<$)

Definition 1 *The relation "$<$" on the set of events of a system is the minimum relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, $a < b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a < b$. (3) If $a < b$ and $b < c$ then $a < c$. Two distinct events a and b are said to be concurrent and denoted by "$a \odot b$" if not $a < b$ and not $b < a$."*

For two events a and b, $a \leq b$ if $a < b$ or $a = b$.

The distributed program execution can be expressed by a 2-tuple $\langle E, R\rangle$, where E is a set of events and R is a partial order on E. For given programs and inputs, if two executions have the same $\langle E, R\rangle$, they will produce the same result and can be considered identical from the assumption that the programs do not contain any nondeterministic statements. A set of events E is partitioned into $E_1, E_2, \ldots, E_N$, where E_i is a set of events in process i. E_i has a total ordering.

In some distributed systems, processes communicate via shared memory [12]. For such a system, writing values to a shared memory location can be simulated by sending a message and reading the values from that location can be simulated by receiving the message. This paper refers only to a message-passing based system for simplicity of discussions, but the results are also applicable to systems communicating through shared memory.

Next, global states of the distributed system are defined.

Definition 2 *For $\langle E, R\rangle$, an N-tuple of events of processes $s = (t_1, t_2, \ldots, t_N)$ $(t_i \in E_i)$ is said to be a global state if $t_i \odot t_j$ for any distinct events t_i and t_j.*

Let U be the set of all the global states for $< E, R >$.

The global state is intuitively considered as a set of concurrent events for some execution with $< E, R >$.

The "happened before" relation for global states is defined as follows.

Definition 3 *For the two global states $s = (t_1, t_2, \ldots, t_N)$ and $s' = (t'_1, t'_2, \ldots, t'_N)$,*
$s \leq s'$ if $t_i \leq t'_i$ for every $i (1 \leq i \leq N)$.
$s < s'$ if $s \leq s'$ and $t_j < t'_j$ for some $j (1 \leq j \leq N)$.

2.2 Debugger Model

The instant-replay based debugger **D** considered here consists of one global debugger GD and local debuggers LD_i corresponding to the processes (Fig. 1). Every pair of GD and LD_i has a communication channel. Process i is controlled by LD_i, which is similar to a sequential program debugger. LD_i is controlled by GD. **D** executes the programs twice. The first execution is called a *monitoring phase* and the second is called a *replay phase*.

In the monitoring phase, each LD_i stores the communication history of the process to its local storage independently. The communication history is a sequence of items corresponding to receive events. The item is a process number or a *Null* symbol. When the process receives a message from process j in the α-th receive event, the α-th item of history is j . *Null* means that no message was received in this receive event.

In the replay phase, message receiving is controlled by LD_i such that the order of message arrival at the process is

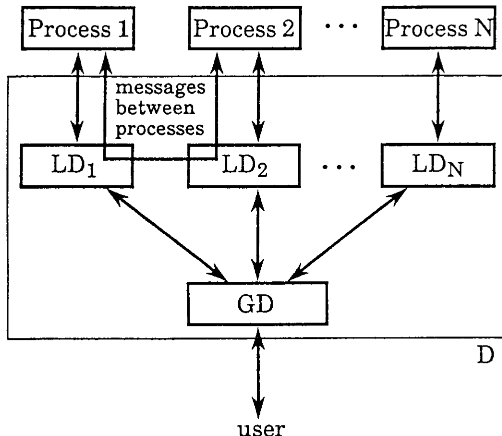

Fig.1 The Instant-Replay Based Debugger

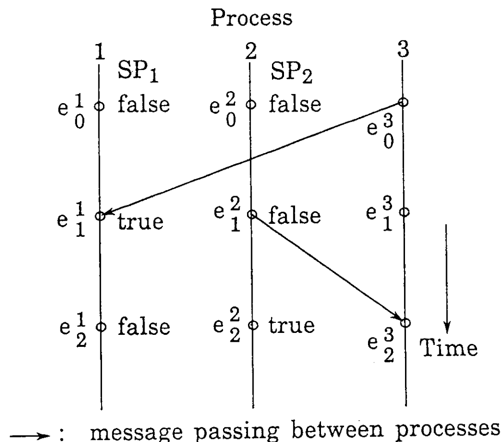

Fig.2 3-process distributed system

the same as in the original execution. From the deterministic nature of the processes, the execution behavior of this replay is the same as that of the original execution.

LD_i controls process i, in the replay phase, with the following operations:

Definition 4 *LD_i can do the following operations to process i in the replay phase.*

1. *see the type of event to be executed next, where the type is "receive event", "send event" or "normal event",*
2. *buffer the messages sent to i and to see the messages sent from i,*
3. *execute the next event and halt the process (if the executed event is a receive event, also specify which message should be read), and*
4. *examine the current state of the process.*

These facilities are generally installed in sequential debuggers. Appendix A shows the functions that are installed in LD_i. They would be used in the algorithms proposed here.

3 Halting by breakpoints

This section discusses how to set breakpoints and halt the system. First, predicates are defined to specify breakpoints.

3.1 Global Predicate

Simple predicates[13] are typically used in sequential program debuggers. They can be expressed in terms of the boolean values of variables in the program and the instruction address variables. The simple predicate is instantaneous in the meaning that its validity does not depend on the execution history but on the state at a instant. The restriction is that a simple predicate is based on the state local to a single process. Let the simple predicate for process i be SP_i. The value of SP_i right after the execution of event e is denoted by $SP_i(e)$.

Two kinds of global predicates are introduced. One is called a Disjunctive Predicate (DP), which consists of single predicates joined by disjunctive operators "$\bigcup$". The other is a Conjunctive Predicate (CP) consisting of single predicates and Conjunctive operators "$\bigcap$". Note that global predicates are functions mapping from U to $\{true, false\}$. The predicate should be supposed to be given after the monitoring phase.

When a global predicate P is given to specify a breakpoint as the command, "stop if P", there are many global states satisfying P. Let $G(P)$ be $\{s \in U | P(s) = true\}$. At which global state in $G(P)$ should the debugger halt the system?

Many distributed program debuggers do not seem to attend to this problem. The sequential program debugger halts the process at the first state in which the simple predicate is satisfied. According to this strategy, the processes should be stopped as soon as possible after P becomes true. For example, consider a 3-process distributed system with P given as $SP_1 \bigcap SP_2$. This predicate has no conditions concerning process 3. Thus, in the case that execution is as shown in Fig. 2, there are two different global states (e_1^1, e_2^2, e_1^3), (e_1^1, e_2^2, e_2^3) which satisfy P. The cause for P becoming true is contained not only in process 1 or 2, but also in process 3. The message from 3 might cause P to be true. If process 3 proceeds beyond the sending event, its contribution to the cause might be destroyed and the real reason for the error might be hidden. Therefore, it would be best to stop the processes at the first global state in $G(P)$. The first global state satisfying the predicate is defined by $Inf(P)$ as shown below.

$Inf(P) = \{s | s \in G(P)$ and $s' \notin G(P)$ for any $s' \in U$

such that $s' < s\}$,
where P is a global predicate. Note that $Inf(P)$ may include plural global states.

For non-replay based debuggers such as described in [9] or [15], it is not possible to stop the processes at some $s \in Inf(P)$ even when P is a simple predicate. Miller and Choi propose an algorithm which stops them at some $s \in G(P)$ when P is a DP and shows that it is impossible to stop them when P is a CP [13]. Section 3.2 and 3.3 show that it is possible to stop the processes at $s \in Inf(P)$ when P is a CP and at $s \in G(P)$ when P is a DP.

3.2 Halting for CP

This section proposes an algorithm which stops the processes at $Inf(P)$ when P is a CP.

For each process, we introduce two states, "*active*" and "*passive*". If the process is being executed, it is called active. A passive process does not become active by itself, but only when another active process activates it. System halting means that all processes are passive and never become active.

Initially, processes with a simple predicate are active and the others are passive. An active process with a simple predicate becomes passive when the predicate becomes true. Suppose an active process tries to read a message M. The message M may not have arrived, because its sender S is passive or the execution of S is delayed. At that time the receiver R sends a control message to ask S to send M. Then S becomes active and executes the program to send M. After sending M, S returns to a passive state. In other words, a process is active when its predicate is false or it must send a message. The part that checks the process status and manages program execution is called a local controller, denoted by LC_i. LC_i is installed in LD_i. The detailed LC_i algorithm is shown in Appendix B.

In a control message, it is necessary to specify which message R needs in order to proceed. Thus the control message contains, as an identity, the message number, which is a sequence number of the messages going through the channel from S to R. LC_i counts the number of messages sent to or received from every channel. Note that it is unnecessary to send the message number attached to the message.

We should be able to detect when every process is passive and does not become active. This is one variation of the distributed termination detection problem proposed by Dijkstra and Scholtem[6] and many algorithms have been proposed for different assumptions regarding the system. The part that detects the termination is called a termination detector. LC_i reports the status to the termination detector when the status changes. To simplify termination detection, control messages go through the termination detector, which can be implemented in either a distributed fashion or a centralized fashion. The distributed algorithm shown in [3] can be used for this termination detector.

Proof of the correctness of this algorithm is beyond the scope of this paper. It is shown only that the above algorithm stops the processes at some $s \in inf(P)$. For preparation, a definition and two lemmas are presented.

Definition 5 *For two events e and e' in one process, let $min(e,e')$ be the event which was not happened after. For two global states $s = (e_1, e_2, \ldots, e_N)$ and $s' = (e'_1, e'_2, \ldots, e'_N)$ $min(s, s')$ is defined as*

$$min(s, s') = (min(e_1, e'_1), min(e_2, e'_2), \ldots, min(e_N, e'_N)).$$

Lemma 1 *If $s, s' \in U$, $min(s, s') \in U$.*

Proof Assume that $min(s, s') = (e''_1, e''_2, \ldots, e''_N) \notin U$. There is a pair of i and j such that $e''_i < e''_j$ from the definition of U. Since $e''_i = min(e_i, e'_i)$, $e''_i = e_i$ or $e''_i = e'_i$. Without loss of generality, it can be assumed that $e''_i = e_i$. Then

$$e_i = e''_i < e''_j = min(e_j, e'_j) \leq e_j,$$

which contradicts $s \in U$. ■

Lemma 2 *If P is a CP $|Inf(P)| \leq 1$*

Proof Let the predicate P be $SP_{i_1} \bigcap SPi_2 \bigcap \ldots SP_{i_k}$.

Assume two global states s and $s' \in Inf(P)(s \neq s')$. If $s < s'$, then $s' \notin Inf(P)$, which implies $s \not< s'$. Thus $min(s, s') < s$.
$P(min(s, s')) = SP_{i_1}(min(t_{i_1}, t'_{i_1})) \bigcap SP_{i_2}(min(t_{i_2}, t'_{i_2}))$
$\bigcap \ldots SP_{i_k}(min(t_{i_k}, t'_{i_k})) = true$.
$min(s, s') \in U$ from the above lemma.

Thus $min(s, s') \in G(P)$. The fact that $min(s, s') < s$ and $min(s, s') \in G(P)$ contradicts $s \in Inf(P)$. ■

Let $inf = (t_1, t_2, \ldots, t_n)$ be the global state in $Inf(P)$. To show that the system halts at inf, the following properties must be demonstrated.

- The processes do not terminate at any $s \in U$ such that $s < inf$.
- Process i does not go over t_i.

It is clear that the system halts at some $s \in U$. Every SP_i is true at s. Thus s is contained in $G(P)$. If $s < inf$, the definition of $Inf(P)$ is contradicted.

Next we show the latter proposition. Process i executes the next event if and only if one of the following conditions is satisfied.

- i has an SP_i and $SP_i = false$
- i received a control message requesting a message and has not yet sent the message.

Suppose that the system halts at s such that some process i goes over t_i. Let s' be a global state during the replay such that $s' = (s_1, s_2, \ldots, s_N) \leq inf$, and for some process

i which goes over t_i in s, $s_i = t_i$ in s. Let S_{inf} be the set of processes which satisfy $s_i = t_i$. No process in S_{inf} executes the next event by the first of above conditions because $s_i = t_i$. Thus, they do not execute the next events unless some process $j \notin S_{inf}$ sends a control message to a process $k \in S_{inf}$ to ask for a message which has not been sent at t_k. Let the send and receive events of the requested message be s_k and r_j. Since $t_k < s_k$, $r_j \leq t_j$ and $s_k < r_j$, $t_k < t_j$. This contradicts $inf \in U$.

Therefore, no process moves beyond t_i. ■

3.3 Halting for DP

When the given predicate is a DP, it is impossible for **D** to stop the processes at $s \in Inf(P)$ as shown below.

Theorem 1 *There is no debugger* **D** *which stops the processes at $s \in Inf(P)$ when P is a* DP.

Proof Let us consider the following example. The number of processes n is 2, and $P = SP_1 \bigcup SP_2$. Processes 1 and 2 do not satisfy SP_1 and SP_2, respectively, in the initial state. Processes 1 and 2 never communicate with each other. From the initial state, assume that **D** lets process 1 move one step. In this case, it may occur that process 1 never satisfies SP_1 and process 2 satisfies SP_2 after executing some events. Thus, **D** cannot stop at $Inf(P)$. A similar situation may occur if **D** lets process 2 move one step. Therefore, the proposition is valid. ■

If we want to stop the processes at some $s \in G(P)$, we can obtain a simple algorithm by modifying the termination algorithm for CP. In this case, the termination detector is not used. In addition to the *active* and *passive* states, a *halt* state is introduced. The "passive" or "halt" processes do not try to execute the program. Initially the processes with a simple predicate are active and the others are passive. The state transition rule is:

- The "active" process with a simple predicate becomes "halt" and the halt message is broadcasted to the others when the predicate becomes true.
- The "passive" process without a predicate becomes "active" when it is asked to send the message M and it has not yet sent M, and returns to "passive" when it sends M.
- Every process becomes "halt" when it receives the halt message, and becomes neither "active" nor "passive" even when it receives a message.

4 Selective Tracing

Many sequential debuggers have the following selective tracing facility.

trace [condition] print [expression]

This command means that the debugger displays the value of [expression] whenever [condition] is satisfied. The expression consists of some variables, the program counter, and arithmetic and logical operators. In the distributed debugger, selective tracing is considered to be display of the value of the global expression when the global condition is satisfied. The global condition is given by the global predicate, and the global expression consists of plural process variables and program counters.

When the predicate P is a CP, the selective tracing can be implemented by Algorithm 1. When the processes halt at $Inf(P)$, the debugger calculates the expression value at the global state and prints it out. When the printing is finished, the debugger returns the process states to the initial value, "active" or "passive", and executes the programs again (under the control of Algorithm 1).

When P is a DP, it is impossible to halt at $Inf(P)$ as shown in Section 3.3. However, to print out the expression value it is necessary only to save the values of the variables used in the expression and calculate the expression value when $Inf(P)$ is detected afterwards. The saved values become useless after the calculation is finished and can be discarded. The tracing algorithm is outlined, focusing on the data saving and discarding mechanisms.

The tracing algorithm in LD$_i$ is implemented as follows. Let P be $SP_{j_1} \bigcup SP_{j_2} \bigcup \ldots \bigcup SP_{j_k}$. Let e_{j_l} be the state of process j_l at some instant during execution. The first global state for e_{j_l} is defined as $Inf[e_{j_l}] = \{s = (s_1, s_2, \ldots, s_N) \in U | s_{j_l} = e_{j_l}$ and $s'_{j_l} < e_{j_l}$ for any $s' = (s'_1, s'_2 \ldots, s'_N) < s$ and $s' \in U\}$. LD$_i$ prepares k virtual program counters $PC_i(j_l)$, $l = 1, 2, \ldots, k$, which correspond to SP_{j_l}.

All the LD$_i$'s cooperate to maintain $PC_i(j_l)$ such that $Inf[e_{j_l}] = (PC_1(j_l), PC_2(j_l), \ldots, PC_N(j_l))$. LD$_i$ such that process i has a variable in the expression maintains the variable value at $PC_i(j_l)$ for each j_l. When SP_{j_l} becomes true, LD$_{j_l}$ collects the variable values at $PC_i(j_l)$ and calculates the expression.

If P has only one simple predicate SP_{j_1}, LD$_i$ maintains $PC_i(j_1)$ in a way similar to Algorithm 1. To maintain k virtual program counters, k programs for Algorithm 1 are executed simultaneously. During the execution, the send and receive event sequence is recorded.

The control message contains the predicate number j_l. When process S receives a control message with predicate number j_l from process R, S moves $PC_s(j_l)$ to the next send event to process R according to the send and receive event sequence.

In Algorithm 1, the process sends the program message to R in response to the control message from R. Here, instead of the program message, process S sends a pseudo message which consists of the predicate number l_j and the message number.

The variable values at events which all PC_{l_j} have passed

are not used any more and are discarded.

During the tracing, the total number of control messages and pseudo messages is $2k$ times of the number of program messages.

5 Other Commands

5.1 Single step execution

Single step execution for sequential program debugger means that the debugger executes the next step of the program and stops. For distributed program debuggers, single step execution has two different concepts. Either concept might be used, according to the purpose.

- next step for any process.
- next step for the specified process the user is interested in.

The latter execution is not always single step for the entire system. If the next event of the process is a receive event and the sender has not sent the message, many steps might have to be executed. This single step execution can be achieved by applying the control message sending mechanism in Algorithm 1.

5.2 Reverse execution

Some distributed debuggers are capable of reverse execution [5] [14]. In the cyclic debugging technique, after a bug is detected and the system is stopped by a breakpoint, the user must look for the bug, which is sometimes located in some process before the halted state. To find it, the user generally has to reexecute the replay from the beginning. However, the replay re-execution might take a long time, because distributed programs run for a long time. Reverse execution logically goes back to see an earlier state of the execution. Reverse execution is implemented by checkpointing global states periodically[5][14]. The reverse execution for this distributed debugger can be implemented in the same manner.

6 Conclusion

This paper introduced global conditions for some commands of distributed program debuggers. Conjunctive and disjunctive predicates were considered. General global predicates, that is, any predicate consisting of simple predicates, disjunctive operators, and conjunctive operators, can be considered as global conditions. Halting at $Inf(P)$ when P is a general global predicate was shown to be impossible. Suppose the program is executed twice in the replay phase. During the first replay, a history of each simple predicate value is recorded. After the first replay, $Inf(P)$ can be calculated from the history. Then during the second replay, each process can stop at $Inf(P)$. However, it is NP-hard to calculate $Inf(P)$, because the satisfiability problem is NP-complete.

More generally, the condition might need to be global such as $V_1 = V_2$, where V_i is a variable in process i. This type of condition seems to be no easier to deal with than a general global predicate, because if these variables have a finite domain $\{a_1, a_2, \ldots, a_n\}$, the condition can be written as $\bigcup_{i=1}^{n}(V_1 = a_i \bigcap V_2 = a_i)$.

For further study, other global conditions which consider the time relationship between events, such as linked predicate[13], must also be considered.

Acknowledgements. The authors would like to thank Masaru Takesue for their guidance and encouragement. They also wish to thank Dr. Naohisa Takahashi for his useful discussions.

Reference

[1] P. Bates: "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behaviors", Proc. of the ACM Workshop on Parallel and Distributed Debugging, pp. 11–122 (May 1988).

[2] K. M. Chandy and L. Lamport: "Distributed Snapshots: Determining Global States of Distributed Systems," ACM Trans. on Computer Systems, vol. 3, no. 1, pp. 63–75 (February 1985).

[3] S. Cohen and D. Lehmann: "Dynamic Systems and Their Distributed Termination", Proc. of 2nd ACM Symp. on Distributed Computing, pp. 29-33 (1982).

[4] R. Cooper: "Pilgrim: A Debugger for Distributed Systems", Proc. of 7th Int. Conf. on Distributed Computing Systems, pp. 458–465 (Sep. 1987).

[5] R. Curtis and L. Wittie: "BugNet: A Debugging System for Parallel Programming Environments," Proc. of 3rd Conf. on Distributed Computing Systems pp. 394–399 (Aug. 1982).

[6] E. W. Dijkstra and C. S. Scholten: "Termination Detection for Diffusing Computations," Inform. Process. Lett., vol. 11, no. 1, pp. 1-4 (1980).

[7] R. E. Fairley: "Software Engineering Concepts", McGraw-Hill, pp. 288–289.

[8] H. Garcia-Molina F. Germaro Jr. and W. H. Kohlar: "Debugging a Distributed Computing System", IEEE Trans. Software Engineering, vol. SE-10, no. 2, pp. 210–219 (Mar. 1984).

[9] P. K. Hartre Jr., D. M. Heimbiigner and R. King: "IDD: An Interactive Distributed Debugger", Proc. of 5th Int. Conf. on Distributed Computing Systems, pp. 498–506 (May 1985).

[10] L. Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, vol. 21, no. 7, pp. 558-565 (July 1978).

[11] B. Lazzerini and L. Lopriore: "Abstraction Mechanisms for Event Control in Program Debugging," IEEE Trans. on Software Engineering, vol. SE-15, no. 7, pp. 890–901 (July 1989).

[12] T. J. LeBlanc and J. M. Mellor-Crummey: "Debugging Parallel Programs with Instant Replay," IEEE Trans. on Comput., vol. C-36, no. 4, pp. 471–480 (April 1987).

[13] B. P. Miller and J.-D. Choi: "Breakpoints and Halting in Distributed Programs," 8th Int. Conf. on Distributed Computing Systems, pp. 316-323 (June 1988).

[14] D. Z. Pan and M. A. Linton: "Supporting Reverse Execution of Parallel Programs", Proc. of the ACM Workshop on Parallel and Distributed Debugging, pp. 124-129 (May 1988).

[15] M. Spezialetti and J. P. Kearns: "A General Approach to Recognizing Event Occurrences in Distributed Computations", 8th Int. Conf. on Distributed Computing Systems, pp. 300-307 (June 1988).

[16] N. Takahashi: "Partial Replay of Parallel Programs Based on Shared Objects", IEICE Techinical Report COMP89 (Dec. 1989) (In Japanese).

A Common Functions in LD$_I$

```
/* Some data types and procedures installed in LDi */
const
  I = ..., /* this process number */
  N = ..., /* the number of processes */
  PredicateExist = ... ,
    /* true if a simple predicate exists for this process */
  Null = 0,
  Infinity = ..., /* ∞ */
  Forever =false;
type
  boolean = {true, false};
  simple_predicate = ...;
  process = 0 .. N;
  status = {passive, active};
  message_no = 0 .. Infinity /* the message number */
  event_type = { send, receive, others };
  message = record   /* messages between the processes */
    sender : process; receiver : process; data : ... end;
function NextEvent : event_type;
  begin /* Return the type of the next event */ end;
procedure LastEvent(var e : event_type, var p : process);
  begin /* Return the event type last executed. If e = send,
  p is receiver's process number. If others, p = Null. */ end;
procedure ExecuteNextEvent(rm : message);
  begin /* Execute next event. If it is a receive event,
  execute with the message. */ end;
function JudgePredicate(SP : simple_predicate) : boolean;
  begin /* Return the predicate validity*/ end;
function ReadProgramMessage : message;
  begin /* read a message */ end;
procedure InsertQueue(m : message)
  begin /* store the message into queue */ end ;
function ReadQueue(i : process) : message ;
  begin /* return the first received message from i if one exists.
  If no message exists, message = Null */ end;
function ReadHistory : process;
  begin /* return the next item of communication history
  and increment the pointer where the item shows the sender*/
  end
```

B Algorithm for LC$_I$

```
program HaltAtBreakpoint
/* This is the program for LCI to halt the system for a CP */
type
  external_event_type = { MessageArrival, ControlMessage-
    Arrival, ExecFinish, TeminationDetected };
  /* ExecFinish means one event execution has finished.*/
  control_message = record
      /* request to send the message */
    dest : process; soc : process; mno : message_no end
procedure wait(var event : external_event_type);
  begin /* Wait for an external event to happen */ end;
procedure SendControlMessage(req : control_message);
  begin /* send a control message */ end;
functionReadControlMessage : control_message;
  begin /* receive a control message */ end;
procedure SendStatus(stat : status);
  begin /* send the status to the termination
  detector*/ end;
function TerminationTest
    (sent, mustsend : array [process] of message_no,
    SP : simple_predicate) : status;
  begin /* return "active" if SP is false or
  sent[i] < mustsend[i] for some process i*/ end

procedure MAIN;
var
  sent, /* the number of messages sent to i. */
  received, /* the number of mesages received from i.*/
  mustsend: /* i requested that the message be sent */
    /* I is active if sent[i] < mustsend[i] */
    array [process] of message_no;
  waitp : process;
      /* waiting for the message from waitp*/
  bstat, cstat : status; /* previous and current status */
  extevent : external_event_type;
  SP : simple_predicate;
  e : event_type ;
  m : message; cm : control_message; i : process
begin
  for i := 1 to N do begin
    sent(i) := 0; received(i) := 0; mustsend(i) := 0 end;
  waitp := Null ;
  cstat := TerminationTest(sent, mustsend, SP);
  if cstat =active
    then TrytoExecuteNext(recieved, waitp);
  repeat
    wait(extevent);
    case extevent of
```

```
      ControlMessageArrival:
        begin
          cm:=ReadControlMessage;
          mustsend[cm.soc] := cm.mno;
          bstat := cstat;
          cstat := TerminationTest(sent, mustsend, SP);
          SendStatus(cstat);
          if bstat =passive and cstat =active
            then TrytoExecuteNext(recieved, waitp)
        end;
      ExecFinish:
        begin
          LastEvent(e, i);
          if e = send then sent(i) := sent(i) + 1;
          if e = receive then received(i) := received(i) + 1;
          cstat := TerminationTest(sent, mustsend, SP);
          if cstat =passive then SendStatus(cstat) ;
          if cstat =active
            then TrytoExecuteNext(recieved, waitp)
        end;
      MessageArrival:
        begin
          m := ReadProgramMessage;
          if waitp = m.sender then
            begin
              waitp := Null;
              ExecuteNextEvent(m)
            end
            else InsertQueue(m);
        end;
      TeminationDetected: /*halt the system*/;
    end
  unitl Forever
end

function TrytoExecuteNext
    (received : array[process]ofmessage_no, var waitp : process);
var s, r, : process; m : message; cm : control_message
begin
  if NextEvent =send or NextEvent =others
      then ExecuteNextEvent(Null);
  if NextEvent =receive then
    begin
      s := ReadHistory;
      if s = Null then ExecuteNextEvent(Null);
      if s ≠ Null then
        begin
          m := ReadQueue(s);
          if m ≠ Null thenExecuteNextEvent(m)   else
            begin
              cm.soc := s; cm.dest := I;
              cm.mno := recieved(s) + 1;
              SendControlMessage(cm); waitp:=s
            end
        end
    end
end
```