マルチメディア通信と分散処理56－13
（1992. 7. 10）

# 再演をベースとした分散プログラムのデバッグ手法について

真鍋 義文　　　　　青柳 滋己
NTT ソフトウェア研究所

## あらまし

プロセス生成 / 消滅、および通信路の設定 / 解放を動的に行なう分散プログラムに対するデバッグ手法について述べる。分散プログラムのデバッグの困難さの主な原因は動作の非決定性および、プロセスの相互関係理解の困難さにあると考えられる。分散プログラムの開発支援用に、(1) 非決定性の原因となるプロセス生成・通信路設定・通信の再演機能、(2) 大域的条件によるブレークポイント・トレース設定機能、(3) 与えられた大域的条件に対する最小限動作機能を持つ、分散プログラム用デバッガのプロトタイプ版 ddbx-p を開発した。動作の非決定性がある場合、(1) の機能より同じ状況を繰り返して試験可能である。また、(2)(3) の機能より、プロセスの相互関係に関するバグの発見が容易である。

# A Distributed Program Debugger Besed on a Replay Technique

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

## ABSTRACT

This paper describes a debugger for distributed programs which may dynamically fork child processes and open and close communication channels between processes. Debugging distributed programs is more difficult than debugging sequential programs, since its behavior might be nondeterministic and behavior relation among processes is difficult to understand. We develop a prototype debugger ddbx-p. Ddbx-p has (1) replay mechanism for process fork, channel creation, and asynchronous communication, (2) breakpoint and selective trace commands using global predicate conditions, and (3) halting mechanism at the first state satisfying the condition. By facility (1), users can test the same execution repeatedly. By facility (2)(3), users can detect errors concerned with the process behavior relation.

## 1 Introduction

Distributed programs are much more difficult to debug than sequential programs, because there might be a bug related to multiple processes. If every bug were only related to a single process, then conventional sequential program debuggers could be used to debug distributed programs.

Thus, one of the main problems in debugging distributed programs is how to detect bugs related to multiple processes[14][16]. One of the most basic and useful tools is a mechanism for detecting predicate satisfaction related to multiple processes. For example, let us consider that process $p_i (i = 1, 2)$ has a variable $x_i$, and that $x_i = 1$ means $p_i$ has the right to access some common data. In this case, satisfying $x_1 = 1$ and $x_2 = 1$ at the same time is a bug. In order to detect this bug, it is convenient if there is a mechanism to detect the satisfaction of the predicate "$x_1 = 1 \bigcap x_2 = 1$". We call a predicate related to multiple processes a global predicate. This paper considers how to detect global predicate satisfaction in distributed programs.

Two kinds of global predicates were introduced in [10]. One is Disjunctive Predicate (DP), which consists of simple predicates joined by disjunctive operators "$\bigcup$", where a simple predicate is a predicate whose true/false state can be detected by a single process. The other is Conjunctive Predicate (CP) consisting of simple predicates and conjunctive operators "$\bigcap$". In this paper, both these predicates are considered.

There are three major requirements for detecting global predicate satisfaction. One is that the probe effect is small. Since distributed programs are asynchronous, if the timing is different, they might behave differently. Thus, if some additional execution for satisfaction detection is done during execution, the behavior might change. Thus, the additional execution for debugging must be as small as possible.

Second, distributed debuggers should let a user see the state just after a given predicate is satisfied. Since the given predicate defines a bug condition, execution after the predicate is satisfied might be meaningless for the user. In addition, the extra execution might hide the real cause of the bug from the user. For example, if the extra execution is exiting from some subroutine, then local variables whose values indicate the real reason of the bug might be completely hidden by the exiting. Thus, debuggers must show the user the exact state when the predicate is satisfied. Consider the case in Fig. 1(a) that the given global predicate $P$ is "$x_1 = 1 \bigcap x_4 = 1$" ($x_1$ and $x_4$ are the variables of processes $p_1$ and $p_4$, respectively). If the predicate is satisfied, $p_1$ and $p_4$ should halt at the exact state in which the condition is satisfied. However, this predicate has no condition concerning the other processes. Where should we halt the other processes? The cause for the predicate becoming true might not only be in $p_1$ or $p_4$, but in the other processes, since these processes are working together. The message from $p_6$ might cause $P$ to be true. Thus, $p_6$ should stop at the state when it has sent a message to $p_1$, which is the last event for $p_6$ to make $P$ to be true. The same situation might occur for the other processes. Therefore, all processes should halt at the state in which each process has executed the minimum requirements for making $P$ true.

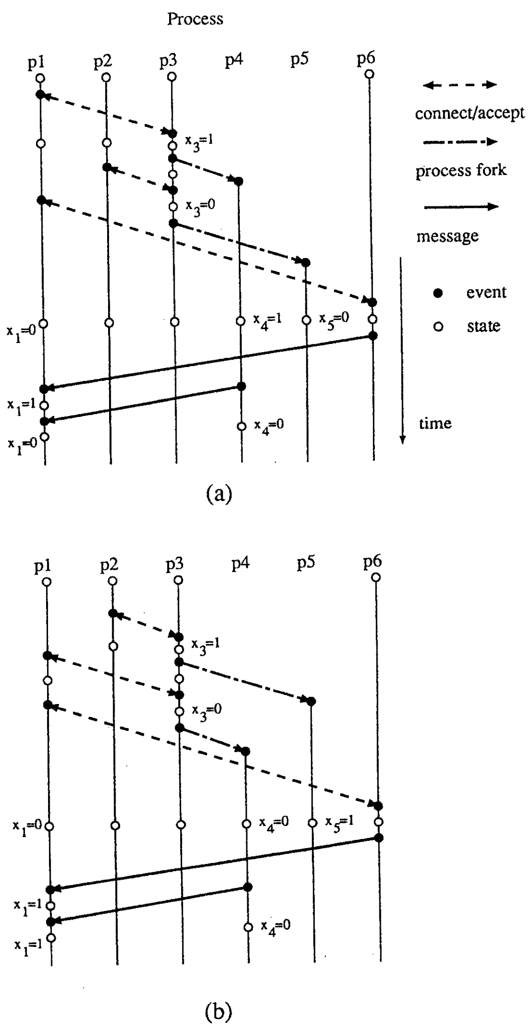

Fig. 1 Examples of distributed program behavior.

Lastly, they should be able to test the same execution repeatedly. Cyclic debugging is one of the most common way of debugging[8]. First, the user sets some breakpoints and executes the program once. If the execution is stopped by a breakpoint, it means that a bug has been found. In order to see the cause of the bug, the user looks at the status and sets other breakpoints and executes the program again. If the program is halted, the user try to see the cause of the new condition by repeating the above procedure. For cyclic debugging, the behavior must be the same in every execution. However, 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. For example, there might be another behavior in Fig. 1(b) for the same distributed program, because of a connect request delay. This behavior has no bug, since "$x_1 = 1 \bigcap x_4 = 1$" does not hold. A similar situation might occur due to a message transmission delay (for example, the message from $p_4$ arrives earlier than that from $p_6$). Thus, in one execution a bug is found, but in the next execution when user tries to find its cause, the behavior might be different and the bug might occur at another point (or not occur at all). This makes debugging very difficult. Thus, for cyclic debugging, repeatedly testing the same execution is necessary.

Current research about detecting global predicate satisfaction falls into three categories: detection during execution, after execution, and during replay.

Detection during execution tests the process state during execution[5][10][15]. This needs no replay mechanism and does not have to execute the whole programs once in advance. Miller and Choi[15] have presented an algorithm to detect a global predicate called a linked predicate. Linked predicate has the form "$SP_1 \rightarrow SP_2 \rightarrow \ldots SP_n$", where $SP_i$ is a simple predicate. The arrow "$\rightarrow$" is Lamport's "happens before" relation[11]. Thus, "$SP_1 \rightarrow SP_2 \rightarrow \ldots SP_n$" means "$SP_1$ is satisfied, and after that $SP_2$ is satisfied and, ..., and after that $SP_n$ is satisfied". This predicate is relatively easy to detect, since the predicate can be piggybacked onto messages between processes. However, the probe effect is not small since it tries to detect satisfaction during execution. And the exact state when the predicate is satisfied cannot be obtained, since this algorithm lets the process with $SP_1$ do an extra execution for a given condition "$SP_1 \rightarrow SP_2$".

Haban and Weigel[10] proposed an algorithm to halt the processes for a non-replay debugger. Their algorithm considers Disjunctive and Conjunctive Predicates, not restricted to the Linked Predicates. However, it is also impossible for their algorithm to halt the processes just when the predicate is satisfied. In addition, the probe effect problem exists.

Cooper and Marzullo[5] considered halting at "Currently $P$", which means halting at the state when $P$ is satisfied. Their algorithm blocks some processes, thus the probe effect is large. In addition, for some predicate $P$, the algorithm cannot halt at the state when the condition is satisfied. Consider the following example where the predicate $P$ is "$x_1 = x_2$". Now, process $p_1$ is at some state and $x_1 = 1$ holds. Process $p_2$ is at some state and $x_2 = 2$ holds. When the debugger lets $p_1$ execute one step, it might happen that $x_1 \neq 1$ holds forever and $x_2 = 1$ holds after some steps in $p_2$. When it lets $p_2$ execute one step, it might happen that $x_2 \neq 2$ holds forever and $x_1 = 2$ holds after some steps in $p_1$. Thus, it is impossible to halt at a state where currently $P$ is true. For a Disjunctive Predicate $P$, it is impossible to halt at the first state where $P$ is true[13].

Detection after execution first gathers traces of event sequences for each process independently and tests the execution afterwards[3][6][9]. The log storing algorithm during execution is simple and any complicated analysis can be done afterwards. However, it is necessary to specify before execution which values should be recorded. Thus, the log tends to be big and the probe effect problem exists. In addition, if some information (which was not specified before) proves to be necessary to detect the cause of the bug, the successive execution to get the information might have a different behavior and no error might occur.

Detection during replay is as follows. During the first execution, the minimum information necessary to replay is corrected and after that, the execution is replayed using the stored information. The global predicate satisfaction is detected by the second execution[13]. If a distributed program contains neither nondeterministic statements such as asynchronous interrupts nor time dependent statements such as reading a clock, its execution can be replayed according to a small log kept during execution; thus, the probe effect can be small. The replay technique discussed in [1][12][17] is based on the above premise. They only consider the replay technique, and detecting global condition is not considered.

Detecting global predicates based on this replay method is considered in [13]. It can halt the processes at the first state for a given Conjunctive Predicate condition. However, the algorithm has a restriction that no dynamic process fork and no dynamic communication channel creation is allowed in distributed programs. Dynamic process fork and channel open/close is commonly used in client-server type distributed programs. In this paper, an algorithm to detect global predicate satisfaction is shown for dynamically process fork and open/close connection distributed programs.

Section 2 presents the model of the distributed system and debugger. Section 3 shows a replay method for dynamic distributed programs. Section 4 gives a minimum execution algorithm to detect a given Conjunctive Predicate. Section 5 presents an implementation. Section 6 discusses further study.

## 2 Model Definition

### 2.1 Distributed System Model

This paper assumes that values exchanged between processes depend only on the initial values in each process and the order in which processes communicate. That is, processes are assumed to be deterministic. Stated somewhat differently, there are no nondeterministic statements, such as asynchronous interrupts, and there are no time dependent statements, such as reading a clock or getting a time out.

The distributed system execution model, based on message-passing communication, is the same as that proposed by Lamport[11]. The system consists of processes and 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.

Forking a child process, connecting a channel, accepting a connect request, closing a channel, sending a message, and receiving a message are considered to be special events and are called a *fork event*, *connect event*, *accept event*, *close event*, *send event*, and *receive event*, respectively. The other events are called *internal events*. The *initial state* of each process is also considered to be an internal event. For a child process (that is, a process that did not exist at the beginning), the *initial state* is considered to be the state before creation. We can then define the "happened before" relation[11] for dynamic systems, denoted by "$<$", as follows (In [11], $\rightarrow$ is used rather than $<$). The relation without conditions (3), (4) below is the original "happened before" relation.

**Definition 1** *The relation "$<$" on the set of events of a system is the minimum relation satisfying the following five conditions: (1) If $a$ and $b$ are events in the same process, and $a$ comes before $b$, then $a < b$. (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 $a$ is a connect event and $b$ is the corresponding accept event, $a < b$ and $b < a$. (4) If $a$ is a fork event for a parent process and $b$ is the corresponding child event (that is, a child process initialization event), $a < b$. (5) If $a < b$ and $b < c$, then $a < c$.*

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

In some distributed systems, processes communicate via shared memory[12]. Such systems can be simulated by a message-passing system[13]. This paper refers only to a message-passing system, but the results are also applicable to systems which communicate through shared memory.

Next, for the proof of the later lemmas, Chandy-Lamport's "meaningful global state"[2] of the distributed system is formally defined using the "happened before" relation(In [2], a global state is defined by a set of event sequences. Our definition is equivalent to that one). Let $N$ be the number of processes.

**Definition 2** *Let $E_i$ be the set of events in process $i$. An $N$-tuple of events of processes $s = (e_1, e_2, \ldots, e_N)$ $(e_i \in E_i)$ is said to be a global state if and only if for all $e'_i \in E_i$: $e'_i > e_i$ implies $e'_i \not< e_j$ for any $j(1 \leq j \leq N)$.*

*Let $U$ be the set of all the global states.*

Global state $s = (e_1, e_2, \ldots, e_N)$ is intuitively considered as a set of concurrently occurring events for some timing occurrence, and we consider it as the state when each process $i$ has just finished the execution of $e_i$. The "happened before" relation for global states is defined as follows.

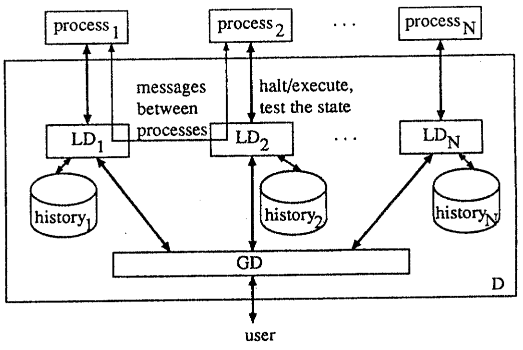

Fig. 2 Distributed Program Debugger.

**Definition 3** *For two global states $s = (e_1, e_2, \ldots, e_N)$ and $s' = (e'_1, e'_2, \ldots, e'_N)$,*
*$s \leq s'$ if and only if $e_i \leq e'_i$ for every $i(1 \leq i \leq N)$, and*
*$s < s'$ if and only if $s \leq s'$ and $e_j < e'_j$ for some $j(1 \leq j \leq N)$.*

For a predicate $P$, let $G(P)$ be $\{s \in U | P(s) = true\}$. The "first" global state in $G(P)$ can be defined as follows.

**Definition 4** *For a predicate $P$,*
*$Inf(P) = \{s | s \in G(P)$ and $s' \notin G(P)$ for any $s' \in U$ such that $s' < s\}$.*

## 2.2 Debugger Model

This section gives a general model for a replay-based debugger. A replay-based debugger **D** consists of one main debugger GD and local debuggers LD$_i$ corresponding to the processes (Fig. 2). Every LD$_i$ is connected to GD by a communication channel. LD$_i$ halts process $i$, lets it execute events, and tests its status. GD sends commands to LD$_i$, receives its replies, and displays the results to the user. **D** executes the programs iteratively. The first execution is called the *monitoring phase* and the others are called *replay phases*.

After the monitoring phase, debugger commands such as "stop if $P$" are given to GD by the user.

In the replay phase, LD$_i$ controls process $i$ with the following operations which are offered by sequential program debuggers: (1) see the type of the event to be executed next, (2) store a received message in a buffer, (3) execute the next event, (4) examine the current state.

## 3 Replay Method for Dynamic Distributed Programs

This section presents a replay method for distributed programs which dynamically fork child processes and open/close connections between processes.

C system call routines

- `int fork()` : create child process
- `int socket()` : create a socket
- `int bind()` : bind a name to a socket
- `int listen()` : listen for a connection
- `int accept()` : accept a connection request
- `int connect()` : initiate a connection
- `int close()` : close a socket

Message transmission Routines

- `int snd(sockno, message)` : send a message to a channel
- `int rcv(mode, sockno, message)` : receive a message from a channel
- `int rcva(mode, message)` : receive a message from any channel
- `int rcvs(mode, sockno_set, message)` : receive a message from any channel in a channel set

Fig. 3 Communication/process fork routines.

Consider the communication and process fork primitives in Fig. 3. These primitives are considered common for message passing communication. The first group are C system call routines to create connection between processes, close connections, and fork child processes. The second group are message passing communication routines: **snd** sends a message via a socket specified by **sockno**. **rcv** receives a message from a channel. **mode** is blocking or non-blocking. If the mode is non-blocking and no message is available, it returns an error. If the mode is blocking, the routine is blocked until a message arrives in the channel. **rcva** receives a message from any one of the currently connected channels. The mode is the same as that in which **rcv**. **rcvs** receives a message from any one of the channels in the set defined by **sockno_set**.

In this algorithm, there is one restriction that every channel must be a one-to-one connection. When a child process fork occurs, one socket is connected to two sockets (parent's and child's sockets) and it can receive messages from two sockets. This case is not considered for simplicity (Note that if additional information is stored in the log, such communication can also be replayed). When a process fork occurs, each socket must be closed by the child process or parent process just after the fork system call.

In the monitoring mode, when these routines are called, the event log is stored as follows. The log is stored for each process as a file named 'historyX' (X is the process id). The first line shows whether the process is created by the user or by another process. If the line begins with **parent**, it shows it is a child process and the number is the parent process number. If not, the line is the program name and its arguments. For the other lines, if it begins with a number, it is connect (**connect** follows), accept (**accept** follows), or receive (number only). The first number is the process's socket number. In the cases of **accept** or **connect**, the first number is its local socket address and the others are the remote hostname and its socket address. These values can be obtained by calling **getsockname()** and **getpeername()** after it is connected or accepted. In the case of **receive**, the number shows the received socket number. If the value is **-1**, it means that no message is received by the receive command.

The line beginning with **fork** shows that a child process fork is done and the child process number is logged. The example in Fig. 4 is the log of the execution in Fig. 1(a). Note that another execution in Fig. 1(b) might also occur for the same program. However, the execution is known to be the one in Fig. 1(a) from the log. Note that **socket**, **listen**, **close**, and **snd** need not to be stored in the log, since they are not nondeterministic.

```
client_a arg_for_a            parent 20001
3 connect 4001 host2 3001
4 connect 4002 host3 5001
4
3
   history10001(p1)              history20002(p4)

client_b arg_for_b            parent 20001
3 connect 4010 host2 3002
   history10002(p2)              history20003(p5)

server_x arg_for_x            server_y arg_for_y
4 accept 3001 host1 4001      4 accept 5001 host1 4002
fork 20002
4 accept 3002 host1 4010
fork 20003
   history20001(p3)              history30001(p6)
```

Fig. 4 History of the behavior in Fig. 1(a).

After the monitor mode, the execution is replayed according to the event log. When the processes are created, the process number is different from the original execution. Thus, each process has a replay process number *rpid* and channel connection request and its reply is handled using *rpid*.

Every connect relation is known because for each connection, the socket address can be obtained from the history. For the example in Fig. 4, process 20001 first accepts the connection request from host1 with socket address 4001 (which is the connection request from process 10001). During the replay, the connection request with socket address 4010 might arrive earlier than that with socket address 4001. However, it is known from the log that the request with socket address 4001 must be accepted first. Thus, if the connection with socket address 4010 is accepted first, it tries another accept until it accepts the connection request with socket address 4001. The connection between 4010 is used when it replays the next accept (with socket address 4010) and this accept call becomes a dummy execution.

For each receive event, the sender socket number is known from the log. Thus, the receive event is replayed by a similar procedure to the one shown above[13]. By using the log, the execution in Fig. 1(a) is replayed by the log in Fig. 4.

## 4 Global Predicate Satisfaction Detection Algorithm

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

For each process, we introduce "*active*" and "*passive*" states. If the process is being executed, it is called active. A passive process becomes active only when another active process makes it active. System halting means that all processes are passive.

Suppose processes without a simple predicate have a special predicate which is always true. Initially, processes whose simple predicate is false are active and the other processes are passive. There are three cases for a process to activate another process.

The first case is channel connection. Suppose an active process $p$ tries to connect a socket to another process's socket. Let the peer process number be $p'$. $p'$ might not have executed 'accept' because $p'$ is passive or the execution of $p'$ is delayed. In that case, $p$ sends a control message to $p'$ ($p'$ can be got from the history) to ask $p$ to execute 'accept'. Then $p'$ becomes active and continues execution until the predicate becomes true after executing 'accept'. A similar case occurs when $p$ tries to accept and $p'$ has not executed 'connect'.

The second case is a process fork. The previous case supposed that process $p'$ exists. There is a case that $p'$ does not exist since $p'$ is a child process of a process $p''$, and $p''$ has not executed fork $p'$. In that case, when LD$_{p'}$ receives the control message sent to $p'$, it sends another control message 'fork $p'$' to $p''$. The parent process number $p''$ can be obtained from the history for process $p'$. When LD$_{p''}$ receives the control message 'fork $p'$', if $p''$ does not exist, the same procedure is executed for $p''$ in advance. If $p''$ already exists or is forked by the previous procedure, $p''$ becomes active and executes fork $p'$. Process $p''$ remains active and continues execution until the predicate becomes true after forking $p'$.

The last case is message transmission. Suppose an active process $p$ tries to receive a message $M$ via a socket $s$. The message $M$ may not have arrived. If $M$ has not arrived, $p$ sends a control message to the peer of $s$ to ask it to send $M$. Then $s$'s peer becomes active and executes the program to send $M$. In this case, the peer process always exists because the connection is already established before trying to receive a message. However, the sender process is not known in advance, since a process fork might occur and whether the sender is the parent process or child process is not stored in the history (Note that the sender process number can also be stored in the history, but considering the probe effect, the amount of stored information should be as small as possible). Thus, the receiver process $p$ just sends a control message via $s$. The current peer process $p'$ or one of its descendants is $M$'s sender. Thus, $p'$ becomes active and continues execution until it sends $M$ or it forks a child and closes socket $s$. In the former case, $p'$ becomes passive when its simple predicate becomes true after the sending. In the latter case, $p'$ becomes passive until its simple predicate becomes true after the fork. The child process becomes active and continues execution until its simple predicate becomes true after the sending (or it forks another child process). Note that in order to activate a process correctly, it is necessary to assume that after forking a child process, every socket is closed by at least one of parent process or child process, and sockets are closed immediately after the fork statement.

In other words, a process is active when its predicate is false, or when it must connect/accept a socket, send a message, or fork a child process.

In a control message, it is necessary to specify which message $p'$ must send. Thus the control message contains an identifying message number, which is the sequence number of the messages going through the channel. LD$_i$ counts the 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 will not become active. This is one variation of the distributed termination detection problem proposed by Dijkstra and Scholtem[7], and many algorithms have been proposed for different assumptions regarding the system. The part that detects the termination is called a termination detector. Status changes are reported to the termination detector when they occur. To simplify termination detection, we assume that control messages go through the termination detector. The termination detector can be implemented either in a distributed fashion (in LD$_i$ ) or in a centralized fashion (in GD). The distributed algorithm shown in [4] can be used for this termination detector. The halt algorithm is outlined in Fig. 5.

Now we show that the algorithm can halt the processes at $Inf(P)$.

**Lemma 1** *[13] If $P$ is a* CP, $|Inf(P)| \leq 1$.

**Theorem 1** *The algorithm can halt the processes at $Inf(P)$ for a given* CP*$P$.*

**(Proof)** 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.

```
program $HaltAtBreakpoint$ /* Program for $LD_I$. */
  function $TermTest$;
    if $s[i] \geq ms[i]$ for all $i \in sset$ and $SP = true$ and $cpid = \{\}$
    /* If $SP$ does not exist for this process, $SP = true$ */
      then return (passive) else return (active) end;
  function $TryExec$; /* let $e$ be the next event */
    if $e$ is create socket $i$ then begin
      Execute $e$; $sset := sset \bigcup \{i\}$; return ($Null$) end
    if $e$ is connect or accept for socket $i$ then begin
      if $ms[i] = 0$ then Send 'Connect $Psock(pid, i)$' to $Peer(pid, i)$;
      Execute $e$; return ($Null$) end /* make a connection
      between process $Peer(pid, i)$'s socket $Psock(pid, i)$. */
    if $e$ is fork then begin /* let $pid$ be the child process number */
      Execute $e$; Send 'Forked, $sset$, $s$, $r$, $ms$ ' to $LD_{pid}$ ;
      For all 'close($i$)' statement after fork $sset := sset - \{i\}$ ;
      $cpid := cpid - \{pid\}$; return ($Null$) end
    if $e$ is receive from $i$ and there is no program message from $i$
        in the local queue then
      begin Send control message "$r[i] + 1$" to $i$; return($i$) end
    else /* the other events */ begin Execute $e$; return ($Null$) end
  end;
begin /* MAIN */
  for $i := 0$ to $NumberOfSockets$ do begin
    $s[i] := 0$; /* The number of messages sent to $i$ */
    $r[i] := 0$; /* The number of messages received from $i$ */
    $ms[i] := 0$ /* $i$ requested to send up to $ms[i]$ */ end;
  $cpid := \{\}$; /* process number which requested to fork */
  $sset := \{\}$; /* current socket set */
  $exit := ThisProcessExistsFromTheBeginning$;
  while ($exit$) do begin
    When 'Connect $i$' is arrived: begin
      $ms[i] = 1$; $sset := sset \bigcup \{i\}$; Send 'Fork $I$' to $LD_{parent(I)}$
    end ;
    When 'Fork $pid$' is arrived: begin
      Send 'Fork $I$' to $LD_{parent(I)}$; $cpid := cpid \bigcup \{pid\}$
    end ;
    When 'Forked $rsset$, $rs$, $rr$, $rms$' message is arrived:
      begin $exit := true$; $sset := sset \bigcup rsset$;
      $s[i] := rs[i]$; $r[i] := rr[i]$; $ms[i] := rms[i]$ for all $i \in rsset$;
      For all 'close($i$)' statements just after fork $sset := sset - \{i\}$
    end
  end ;

  $cstat := TermTest$; Send $cstat$ to TerminationDetector;
  if $cstat$ =active then $waitp := TryExec$;

  When execution of event $e$ is finished: begin
    if $e$ = send or connect or accept to $i$ then $s[i] := s[i] + 1$;
    if $e$ = receive or connect or accept to $i$ then $r[i] := r[i] + 1$;
    $cstat = TermTest$;
    if $cstat$ =active then $waitp := TryExec$
      else Send $cstat$ to TerminationDetector;
  end ;
  When program message $m$ is arrived from $i$: begin
    Insert $m$ to local queue;
    if $waitp = i$ then $waitp := TryExec$
  end;
  When control message "$k$" is arrived from $i$ : begin
    $ms[i] := k$; $bstat := cstat$; $cstat := TermTest$;
    if $bstat$ =passive and $cstat$ =active then begin
      Send $cstat$ to TerminationDetector; $waitp := TryExec$ end
  end ;
  When 'Connect $i$' is arrived: begin
    $ms[i] = 1$; $sset := sset \bigcup \{i\}$; $bstat := cstat$; $cstat := TermTest$;
    if $bstat$ =passive and $cstat$ =active then begin
      Send $cstat$ to TerminationDetector; $waitp := TryExec$ end
  end ;
  When 'Fork $pid$' is arrived: begin
    $cpid := cpid \bigcup \{pid\}$; $bstat := cstat$; $cstat := TermTest$;
    if $bstat$ =passive and $cstat$ =active then begin
      Send $cstat$ to TerminationDetector; $waitp := TryExec$ end
  end ;
  When Termination is detected: Halt;
end.
```

Fig. 5. Halting algorithm for a Conjunctive Predicate.

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

Assume 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. The former proposition is therefore true.

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 child process fork and has not yet forked the process.
- $i$ received a control message requesting connect/accept.
- $i$ 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 $i$ executes beyond $t_i$. 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 executes beyond $t_i$ 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 $k \in S_{inf}$, before it executes $t_j$, to ask for a child process fork, connect, accept, or message sending whose corresponding event $k$ has not executed at $t_k$. Let the corresponding events at $k$ and $j$ be $e_k$ and $e_j$, respectively. Thus, $e_k > t_k$ and $e_k < e_j < t_j$ holds, which contradicts $inf \in U$.

Therefore, no process executes beyond $t_i$. ■

Note that as shown in [13], it is impossible to halt at $Inf(P)$ if $P$ is a DP. It is also impossible for other predicates which cannot be converted to a CP(for example, predicate $x_1 = x_2$, where $x_i$ is a variable in $p_i$).

## 5 Prototype Implementation

We developed a prototype distributed debugger ddbx-p on SUN-4. In order to use ddbx-p, a user must write programs with the routines shown in Fig. 3 and special routine **ddbx_init** and **ddbx_term**. **ddbx_init** must be called at the top of the programs to initiate logging. **ddbx_term** must be called at the end of the programs to close log files.

The ddbx-p commands related to global predicate are:

- `stop if [global predicate]` :set a breakpoint
- `trace [process expression] if [global predicate]` :set a trace condition
- `switch [commandno], [predicateno]` : change primary

```
[global predicate]::= [conjunctive predicate] |
        [conjunctive predicate] or [global predicate]
[conjunctive predicate] ::= [simple predicate] |
        [simple predicate] and [conjunctive predicate]
```

Since it is impossible to halt at $Inf(P)$ for a DP, ddbx-p can halt at $Inf(P)$ for only one CP. This CP is called a primary predicate. Users can specify one CP as the primary among the breakpoint conditions. Switch command changes the primary. For the other predicates, ddbx-p reports satisfaction after $Inf(P)$ and halts.

Another command trace also uses a global condition. It prints out the value of some expression whenever the condition is satisfied and continues execution. It is unnecessary to stop at $Inf(P)$ to print out an expression value, if the values of the variables in the expression are saved during replay. Thus, trace can print out the expression value at $Inf(P)$ even if $P$ is a DP. The algorithm is similar to that in [13].

## 6 Conclusion

We presented a global predicate satisfaction detection algorithm for distributed programs which dynamically fork child process and open/close connections. We developed a prototype debugger, ddbx-p, and based on experience using it in our group, we will refine it with other commands which would be more useful for debugging distributed programs.

**Acknowledgments** The authors would like to thank Dr. Rikio Onai for his encouragement and suggestions. They also thank Dr. Makoto Imase and Dr. Naohisa Takahashi for discussions about early versions of this work.

## ■ 参考文献

[1] Carver, R. H., and Tai, K. Reproducible Testing of Concurrent Programs Based on Shared Variable, *Proc. 6th Int. Conf. on Distributed Computing Systems*(May 1986), pp. 428–433.

[2] Chandy, K. M., and Lamport, L. Distributed Snapshots: Determining Global States of Distributed Systems,*ACM Trans. on Computer Systems*, 3, 1(Feb. 1985), pp. 63–75 .

[3] Choi, J.-D., Miller, B.P., and Netzer, R. Techniques for Debugging Parallel Programs with Flowback Analysis, *Technical Report 786, Univ. of Wisconsin-Madison, Computer Science Department* (Aug. 1988).

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

[5] Cooper, R. and Marzullo, K.: Consistent Detection of Global Predicates, *Proc. of Workshop on Parallel and Distributed Debugging* (May 1991), pp.167–174.

[6] Denning, A. and Schonberg, E. The task recycling technique for detecting access anomalies on-the-fly, *Technical Report RC 15385 (68543), IBM T.J. Watson Research Center* (Jan. 1990).

[7] Dijkstra, E.W., and Scholten, C.S. Termination Detection for Diffusing Computations, *Inform. Process. Lett.* 11, 1(1980), pp.1–4.

[8] Fairley, R. E. *Software Engineering Concepts*, McGraw-Hill.

[9] Garcia-Molina, H., Germano, F. Jr., and Kohler, W.H. Debugging a Distributed Computing System *IEEE Trans. on Software Eng.* , vol.SE-10, no.29(March 1984), pp.210–219 .

[10] Haban, D. and Weigel, W. Global Events and Global Breakpoints in Distributed Systems, *21st Hawaii International Conference on System Sciences*, (Jan. 1988), pp. 166–175.

[11] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21, 7(July 1978), pp. 558–565.

[12] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging Parallel Programs with Instant Replay, *IEEE Trans. on Comput.*, C-36, 4(April 1987), pp. 471–480.

[13] Manabe, Y. and Imase, M. Global Conditions in Debugging Distributed Programs, *J. of Parallel and Distributed Computing*, 15, (1992) pp. 62–69.

[14] McDowell, C.E. and Helmbold, D.P. Debugging Concurrent Programs, *ACM Computing Surveys*, Vol.21, No.4, (Dec. 1989) pp.593–622.

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

[16] Pancake, C.M. and Utter, S. A Bibliography of Parallel Debuggers, 1990 Edition, *ACM SIGPLAN Notices*, Vol.26, No.1 (Jan. 1991) pp.21–37.

[17] Takahashi, N. Partial Replay of Parallel Programs Based on Shared Objects, *IEICE Technical Report* COMP89–98 (Dec. 1989) (In Japanese).