Debugging Dynamic Distributed Programs Using Global Predicates

Yoshifumi Manabe Shigemi Aoyagi

NTT Basic Research Laboratories
3-9-11 Musashino-shi, Tokyo 180 Japan

Abstract

This paper describes a debugger for distributed programs based on a replay technique. Distributed programs may dynamically fork child processes and open and close communication channels between processes. This debugger features breakpoint setting and selective trace commands with global predicate conditions called Conjunctive Predicate and Disjunctive Predicate, which are related to multiple processes. It can halt or test the processes at the first global state for a given Conjunctive Predicate breakpoint condition.

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 related to only 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[13][15]. 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 $p_i (i = 1,4)$ 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 $x_4 = 1$ 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 "$x_1 = 1 \bigcap x_4 = 1$". 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 [9]. One, the Disjunctive Predicate (DP) 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, the Conjunctive Predicate (CP) consists of simple predicates and conjunctive operators "$\bigcap$". This paper considers both types of predicates.

There are three major requirements for detecting global predicate satisfaction. One is that the probe effect must be small. Distributed programs are asynchronous so their behavior varies with the timing of events, and the additional execution required for satisfaction detection may alter the timing. Thus, the additional execution required for debugging must be minimized.

Another requirement is that it should be possible to test the same execution repeatedly. Cyclic debugging is one of the most common methods[7]. In the example in Fig. 1(a), when the program is halted by the predicate "$x_1 = 1 \bigcap x_4 = 1$", the user might set another breakpoint and execute the program again to determine the cause of the bug. For cyclic debugging, the behavior must be the same in every execution. However, the execution behavior might be nondeterministic. For example, there might be another behavior, such as shown 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 the one from p_6). Thus, in one execution a bug is found, but in the next execution, in which user tries to find the cause of the bug, the behavior might be different and the bug might occur at another point (or not at all). This makes debugging very difficult. Thus, for cyclic debugging, repeatedly testing of the same execution is necessary.

The third requirement is that distributed debuggers should let a user see the state just after a given predicate is satisfied. Since the predicate defines a bug condition, the execution that follows after it is satisfied might be meaningless for the user. The extra execution might also hide the real cause of the bug from the user. Thus, debuggers must show the user the exact state where the predicate is satisfied. Consider the case in Fig. 1(a) that the given global predicate P is "$x_1 = 1 \bigcap x_4 = 1$". This predicate has no condition concerning processes other than p_1 or p_4. 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. 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.

Current research on detecting global predicate satisfaction falls into three categories: detection during execution, after execution, and during replay. Detection during execution tests the process state dur-

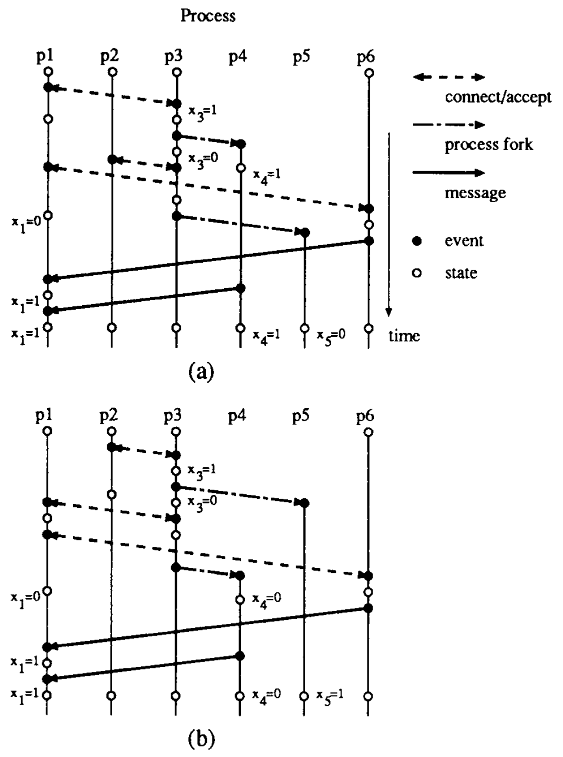

Figure 1: Examples of distributed program behavior.

ing execution[5][9][14]. This needs no replay mechanism or execution of the whole programs in advance. Miller and Choi[14] have presented an algorithm for detecting a global predicate called a Linked Predicate. The 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[10]. Thus, "$SP_1 \rightarrow SP_2 \rightarrow \ldots SP_n$" means that "$SP_1$ is satisfied, and then SP_2 is satisfied and so on, and then 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's algorithm[9] 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. The probe effect problem also 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, so 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 case the predicate P is "$x_1 = x_2$". Process p_1 has the variable x_1 whose initial value is 1. Process p_2 has the variable x_2 whose initial value is 2. When the debugger lets p_1 execute one step, it might happen that $x_1 \neq 1$ is always true and $x_2 = 1$ holds after some steps in p_2. When it lets p_2 execute one step, it might happen that $x_2 \neq 2$ is always true 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.

Detection after execution first gathers traces of event sequences for each process independently and tests the execution afterwards[3][8]. The log storing algorithm during execution is simple and any complicated analysis can be done. However, it is necessary to specify before execution which values should be recorded during execution. 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 be different and no error might occur.

Detection during replay is as follows. During the first execution, the minimum information necessary to replay is collected and after that, the execution is replayed using the stored information. The global predicate satisfaction is detected by the second execution[12]. 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], [11], and [16] is based on the above premise. They only consider the replay technique, and detecting global conditions is not considered.

Detecting the global predicate based on this replay method is considered in [12]. It can halt the processes at the first state for a given Conjunctive Predicate condition. However, the algorithm has the restriction that no dynamic process fork and no dynamic communication channel creation is allowed in distributed programs. Dynamic process fork and channel open/close are commonly used in client-server type distributed programs. In this paper, an algorithm for detecting global predicate satisfaction is shown for dynamically process fork and open/close connection distributed programs.

Section 2 presents the model of the distributed system. Section 3 shows a replay method for dynamic distributed programs. Section 4 gives a halting algorithm for detecting a given Conjunctive Predicate. Section 5 presents an implementation. Section 6 summarizes the paper and 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 time out.

The distributed system execution model, based on message-passing communication, is the same as that proposed by Lamport[10]. 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. Note that in some distributed systems, processes communicate via shared memory[11]. Such systems can be simulated by a message-passing system[12]. This paper refers only to a message-passing system, but the results are also applicable to systems which communicate through shared memory.

Forking a child process, connecting a channel, accepting a connect request, closing a communication 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[10] for dynamic systems, denoted by "$<$", as follows. (In [10], $\rightarrow$ is used rather than $<$.) The relation without conditions (3) and (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 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 $a = b$.

Next, for the proof, Chandy-Lamport's "meaningful global state"[2] of the distributed system is formally defined using the "happened before" relation. 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 $s = (e_1, e_2, \ldots, e_N)$ $(e_i \in E_i)$ is said to be a global state iff 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.

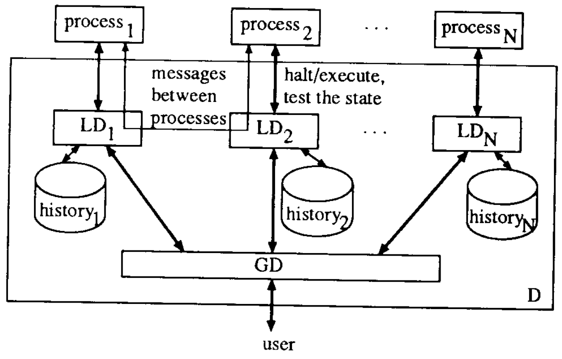

Figure 2: Distributed program debugger.

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

Lastly, the "first" global state for a predicate is defined as follows.

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

2.2 Debugger Model

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. 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 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. Consider the communication and process fork primitives in Fig. 3. Here, `mode` is blocking or non-blocking. If `mode` is non-blocking and no message is available, it returns an error. If **mode** is blocking, the routine is blocked until a message arrives.

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 from any channel
- **int rcvs(mode, sockno_set, message)**
  : receive from any channel in a channel set

Figure 3. Communication/process fork routines.

In this algorithm, there is the restriction that every channel must be a one-to-one connection. When a child process fork occurs, one socket is connected to two sockets (the parent's and child's sockets) and it can receive from both of them. 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 or parent process just after the fork statement.

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' (where 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**, next 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 socket number from which the current message is received. 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 **socket**, **close**, and **snd** need not to be stored in the log.

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*. Since connect/accept and message transmission are asynchronous, the arrival order of connect request or message might be different in the monitoring and replay phases. Thus, using the event log, the execution is blocked until the proper request or message arrives. The replay algorithm is similar to the instant replay algorithm[11] and its detail will be described in the final paper.

```
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)

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

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.

For a given P, 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 in which a process activates another process.

- channel connection
- message transmission
- child process fork

First consider channel connection. Suppose an active process p tries to connect a socket to process p''s socket. Process p sends a control message to p' that to ask p' to execute the corresponding 'accept'. If p' is passive, it becomes active upon receiving this control message and executes until its simple predicate becomes true after the 'accept'. The message transmission is the same as channel connection where 'connect' and 'accept' correspond to 'receive' and 'send'.

The last case is child process fork. The previous cases supposed that process p' exists. There is a case in which 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 $\mathrm{LD}_{p'}$ 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 of process p'. Process p'' becomes active upon receiving the control message and executes until its simple predicate becomes true after the 'fork'.

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

We should be able to detect the state in which every process is passive. This is one variation of the distributed termination detection problem[6], 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.

```
program $HaltAtBreakpoint$ /* Program for LD$_I$. */
  function $TermTest$;
    if $s[i] \geq ms[i]$ for all $i \in sset$ and
        $SP = true$ and $cpid = \{\}$
      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
        to socket $Psock(pid, i)$ in process $Peer(pid, i)$. */
    if $e$ is fork then begin /* let $pid$ be the child. */
      Execute $e$; Send 'Forked, $sset$, $s$, $r$, $ms$ ' to LD$_{pid}$ ;
      For all 'close($i$)' 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 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 to fork */
  $sset := \{\}$; /* current socket set */
  $exit := ThisProcessExistsFromTheBeginning$;
  while($exit$=false) 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 $rsset$, $rs$, $rr$, $rms$' is arrived:
      begin $exit$ :=true; $sset := sset \bigcup rsset$;
      For all $i \in rsset$ begin $s[i] := rs[i]$; $r[i] := rr[i]$;
        $ms[i] := rms[i]$ end;
      For all 'close($i$)' after fork $sset := sset - \{i\}$
    end
  end; /* end of while loop */
  $cstat := TermTest$; Send $cstat$ to TermDetector;
  if $cstat$ =active then $waitp := TryExec$;

  When execution of event $e$ is finished: begin
    if $e$ = send/connect/accept to $i$ then $s[i] := s[i] + 1$;
    if $e$ = receive/connect/accept to $i$ then $r[i] := r[i] + 1$;
    $cstat = TermTest$;
    if $cstat$ =active then $waitp := TryExec$
      else Send $cstat$ to TermDetector;
  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 TermDetector; $waitp := TryExec$ 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 TermDetector; $waitp := TryExec$ 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 TermDetector; $waitp := TryExec$ end
  end;
  When Termination is detected: Halt;
end.
```

Figure 5. Halting algorithm for CP.

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

Lemma 1 [12] *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.

- 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 or 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 it is impossible to halt at $Inf(P)$ if P is a DP [12]. 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 UNIX[1] 4.2 BSD. The **ddbx-p** commands related to the global predicates are as follows.

- **stop if [global predicate]** :set a breakpoint
- **trace [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. The switch command changes the primary. For the other predicates, **ddbx-p** reports satisfaction after $Inf(P)$ and halts.

Another command trace prints out the value of some expression whenever the condition is satisfied and continues execution. It is not necessary to stop at $Inf(P)$, 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 [12].

The **ddbx-p** receive primitive takes about 153 μs with monitoring and 196 μs without monitoring on SUN4. Thus, the time for monitoring is about 43 μs per receive event. Therefore, if receive occurs less than once within 5 ms, the overhead for monitoring is less than 1%. (Generally, the other events are not called as frequently as receive.) Therefore, the probe effect might not be so great for some programs. The log size is a few bytes per event. (Strings such as **connect** and **fork** can be encoded to reduce the size.) Thus, the size of the log is not too great for ordinary programs.

[1]UNIX is a registered trademark of AT&T.

6 Conclusion

We presented a global predicate satisfaction detection algorithm for distributed programs which dynamically fork child process and open or close connections. We developed a prototype debugger, ddbx-p, and on the basis of experience using it in our group, we will refine by adding other commands which are more useful for debugging distributed programs.

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

References

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

[2] Chandy, K. M., and Lamport, L. Distributed Snapshots: Determining Global States of Distributed Systems, *ACM TOCS*, 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, *Tech. Report 786, Univ. of Wisconsin-Madison, Computer Science Department* (Aug. 1988).

[4] Cohen, S., and Lehmann, D. Dynamic Systems and Their Distributed Termination, *2nd PODC* (1982), pp. 29–33.

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

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

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

[8] Garcia-Molina, H., Germano, F. Jr., and Kohler, W.H. Debugging a Distributed Computing System *IEEE Trans. Software Eng.*, Vol.SE-10, No.29(Mar. 1984), pp.210–219.

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

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

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

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

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

[14] Miller, B. P., and Choi, J.-D. Breakpoints and Halting in Distributed Programs, *8th ICDCS* (1988), pp.316–323.

[15] 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.

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