A prototype of a concurrent behavior monitoring tool for testing of concurrent programs

Eisuke ITOH
Dept. of Computer Science and Communication Engineering,
Kyushu University.
Hakozaki 6-10-1, Higashi-ku,
Fukuoka, 812-81, Japan.
itou@csce.kyushu-u.ac.jp

Zengo FURUKAWA
Educational Center for Information Processing,
Kyushu University.
Hakozaki 6-10-1, Higashi-ku,
Fukuoka, 812-81, Japan.
zengo@ec.kyushu-u.ac.jp

Kazuo USHIJIMA
Dept. of Computer Science and Communication Engineering,
Kyushu University.
Hakozaki 6-10-1, Higashi-ku,
Fukuoka, 812-81, Japan.
ushijima@csce.kyushu-u.ac.jp

Abstract

Testing of concurrent programs is much more difficult than that of sequential programs. A concurrent program behaves nondeterministically, that is, the program may produce different results with the same input data according to execution timings of the program. In testing of concurrent programs, test data must specify not only input data but also sequences of statements.

Ordered Sequence Testing Criterion for length k (OSC_k), which was proposed by the authors, requires execution of all sequences of length k of concurrency statements which cause concurrent actions in a concurrent program. A monitoring tool has been developed for applying the testing criterion OSC_k to the testing of C concurrent programs on UNIX system. The tool measures coverage with regard to k-tuples of concurrency statements (OSC_k) in source codes of a C concurrent program using a probe insertion method.

The analysis of the tool's output for a practical C concurrent program shows not only applicability of the tool for testing of concurrent program but also the necessity of a supporting tool for forcing execution of concurrency statements.

1 Introduction

Testing of concurrent programs is more difficult than that of sequential programs. In execution of a concurrent program, a process in the program communicates with, synchronizes and waits for nondeterministically other processes through execution of concurrency statements related to interprocess communication, synchronization and waiting. Therefore, execution sequences with the same input data may be different due to execution timing of concurrency statements. In testing of concurrent programs, we have to specify not only input data but also sequences of statements for execution as testing conditions of the concurrent programs.

A testing criterion specifies conditions for termination of testing. We proposed testing criteria OSC (Ordered Sequence Testing Criteria) for concurrent programs in [1, 2], which are based on sequences of concurrency statements. An OSC_k requires execution of all sequences of length k of the statements at least once (k is a natural number.)

We have developed a prototype of a concurrent behavior monitoring tool on UNIX for testing based on OSC . The tool monitors execution of a concurrent program and measures coverage with regard to pairs of concurrency statements in source codes of a C concurrent program. Many concurrent programs are developed with C language on UNIX system, and then we developed the prototype for C concurrent programs.

In this paper, we describe the prototype tool and the result of an experiment. We analyze the experiment results and discuss the effectiveness of OSC. Section 2 briefly explains testing criteria, OSC. The OSC_k are based on source code, and it is only interested in execution order of interprocess communication, synchronization and waiting. Section 3 describes our monitoring tool. This section shows how to transform a source code, how to work the monitor and record concurrent behavior, and how to calculate coverage. Section 4 shows an experiment of our prototype tool. We applied the tool to the `phone` program [3] and recorded the behavior. Section 5 analyzes the experiment results. Section 6 discusses our tool for practical use. Section 7 concludes this paper.

2 OSC

The concurrent programs that we are interested in testing include those written in languages such as Ada, CSP, and we are especially interested in C concurrent programs on UNIX system. And we consider the behavior of concurrent programs on the interleaving model[4]. The interleaving model has been used to represent the behavior of concurrent programs running on a single processor computer.

A concurrent program consists of some computation units (processes), each of which behaves like a sequential program. The processes communicate, synchronize, and/or wait with each other. The processor selects a process arbitrarily and executes one of statements in the process. However, the selection is constrained in case of synchronization, communication, and waiting between concurrency statements. These features may cause nondeterministic behavior by a computer system.

A concurrent program can be supplied with the same input data set on two different executions, yet exhibit different behavior. This behavior represents the effects of the computer system making different choices in response to conditions external to the program, such as the load on the machine on which the program runs.

Testing criteria designed for application to sequential programs can be applied to concurrent programs. However, they fail to directly address the testing problems peculiar to concurrent programs. The nondeterministic behavior presents one key difficulty.

For example, let's consider a concurrent program **P** which is shown in Figure 1. **P** consists of 3 processes: P1, P2, and P3. P1 and P2 send a string to P3 once. P3 receives both strings and sets both strings in the same data cell "m". The execution result is unpredictable which string is set.

On the program **P** , the execution result value of "m" is in accordance with the execution order of the communication statements. The possible execution orders of communication statement are as follows:

$$<1,3,2,3>; \quad \longrightarrow \quad m = \text{“}bb''\ ,$$

$$<2,3,1,3>; \quad \longrightarrow \quad m = \text{“}aa''\ .$$

If **P** is executed once, then the all-paths testing criterion (C_∞) [5] is covered. But it is not sufficient. **P** must be executed in a different execution sequence.

The nondeterministic execution of concurrent program can be modeled as ordered sequences of statements in the program. In testing of concurrent programs, we have to specify not only input data but also sequences of statements for execution as testing conditions of concurrent programs. However, it is not practical to execute all possible execution orders of the statements in testing.

```
    P1                 P2
1: Send_data(aa); 2: Send_data(bb);
    P3
   repeat
3: m:= Received_data;
   until nodata;
```

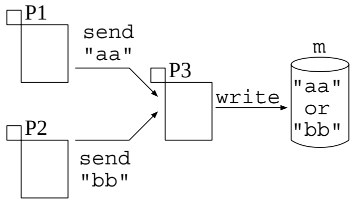

Figure 1. A concurrent program.

The characteristics of concurrent programs are interprocess communication, synchronization and waiting. We call the statement related with interprocess communication, synchronization and waiting as "a concurrency statement." A nondeterministic execution of a concurrent program can be abstracted as an execution order of concurrency statements.

Therefore, we proposed testing criteria, named OSC [1, 2]. The criteria OSC are concerned with the execution order of concurrency statements. The formal definition of OSC is as follows.

> OSC_k :
> Suppose that there is a concurrent program and *Sync* is a set of all concurrency statements in the source code. Construct ordered sequences of length k of the concurrency statements, where k is a natural number. An OSC_k requires execution of all the ordered sequences of length k at least once.

A set of test-events of OSC_k ($k \geq 1$) is described as the following set,

$$TE(\mathsf{OSC}_k) = \{< s_1, s_2, \cdots s_k > | s_i \in Sync, 1 \leq i \leq k\}\ ,$$

where the $TE(Cri)$ is a set of test-events of a testing criterion Cri.

If $k = 2$, we call the OSC_2 the *ordered pair* testing criterion. A set of test-events of OSC_2 is described as the following set.

$$TE(\mathsf{OSC}_2) = \{< s_i, s_j > | s_i, s_j \in Sync\}.$$

If all test-events of OSC_2 are executed, then all communications between any two processes are tested at least once.

3 A prototype of a monitoring tool

We have developed a prototype of a monitoring tool which measures coverage of pairs of concurrency statements in C concurrent programs. This coverage is a metric for evaluating testing sufficiency based on the testing criterion OSC_2.

C language does not offer interprocess communication features. C concurrent programs cooperate with an operating system for realizing the interprocess communication. The monitoring tool can deal with C concurrent programs on UNIX operating system. There are some kinds of mechanisms for interprocess communication such as semaphore, FIFO, pipe and socket[6, 7]. These mechanisms are implemented as *system calls* in UNIX. The monitoring tool is able to deal with the system calls for semaphore, pipe and socket mechanisms currently.

In this section, we firstly explain *system calls* for the socket mechanism because monitoring behavior is easy to understand. Then, we desicribe the outline of the monitoring tool.

3.1 Socket

The *socket* is developed for interprocess communication in 4.3BSD UNIX at first[6]. The socket is available on some other operating systems at present[7]. Our tool makes a log of execution of the socket system calls. There are some features in the socket. This subsection briefly describes the socket of UNIX.

The socket supports two data types of communication: stream type and datagram type. The stream type socket guarantees message arrival. The datagram type socket does not guarantee message arrival. In communication with the stream type socket, messages are sent with system calls `write()` or `send()`, and messages are received with system calls `read()` or `recv()`. In communication with the datagram type socket, messages are sent with the system call `sendto()`, and messages are received with the system call `recvfrom()`.

The system calls receive messages in two modes : blocking and nonblocking. In the blocking mode, if there is no readable message, then the process which is to receive the message waits until the message arrival. In the nonblocking mode, the process does not wait but executes another statement immediately.

A socket can receive messages from multiple sockets and other devices. The system call `select()` can be used for multiple message receiving. In the system call `select()`, a process which receives the message can wait a fixed time for message arrival.

After that, the statements which include the system calls, `write()`, `send()`, `read()`, `recv()`, `sendto()`, `recvfrom()`, and `select()` are the concurrency statement of UNIX C concurrent program, where the system calls must use a socket. (The system calls `write()` and `read()` can use for any device.)

3.2 The outline of the monitoring tool

We describe the outline of the prototype of our monitoring tool. We take a program transformation approach to monitor concurrent behavior [8, 9]. Figure 2 shows the outline of our monitoring tool. The tool consists of three programs: the source code transformation program, the monitor program, and the coverage calculation program.

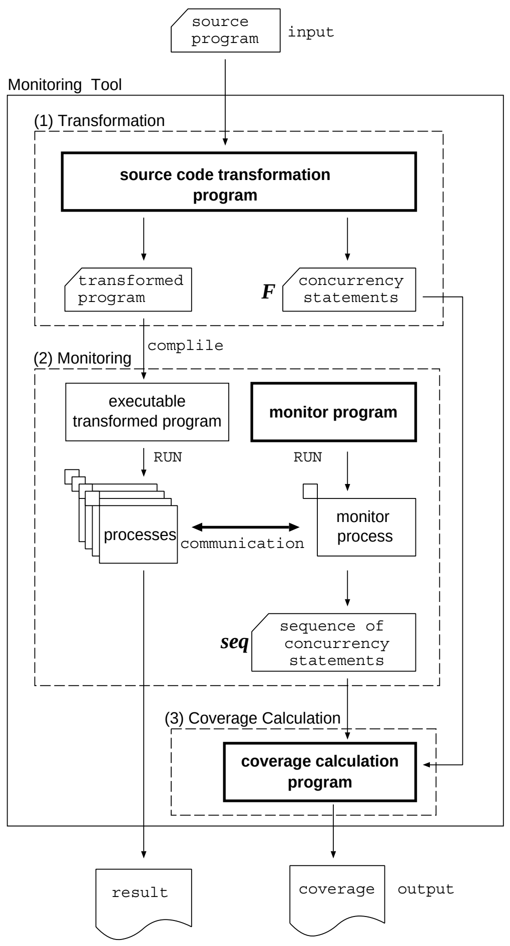

Figure 2. An outline of the monitoring tool.

(1) Source Code Transformation Program

Figure 3 shows a source code transformation. The input of the transformation program is a C source code of a concurrent program, and the output is a transformed C source code and a file F. The file F includes a table of all concurrency statements. A concurrency statement (a system call for communication) is identified by two items: file name which includes the statement and line number where the statement appears. Then every concurrency statement can be given a unique identifier. The transformation program assumes that one line has only one statement in the input program. Then, the input program must be organized beforehand.

The transformed sources are constructed by inserting probes before and after every communication statement. Figure 3 shows how to insert probes before and after a communication statement. In Figure 3, the function `probe_b()` is the inserted probe before a communication statement, and the function `probe_a()` is the after probe. A probe is a communication function between existing processes and the monitor process.

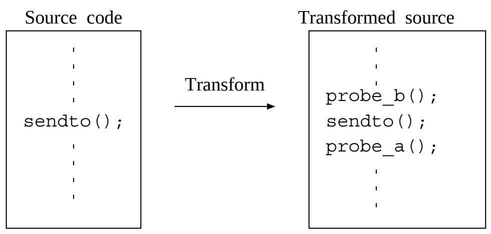

Figure 3. Program transformation. (Probe insertion.)

(2) Monitor Program

The monitor program controls communications between processes in the tested program, and records communications in a file *seq*. The program proceeds according to the following steps. The first step is to compile the transformed sources. The next is to run the monitor program, and then the monitor process will be generated. The last is to run the complied program, and then some target processes will be generated. We call the processes which are generated from the transformed program (tested program) as the target processes. The monitor program must be running before execution of target program, otherwise, execution behaviors are not recorded.

Figure 4 shows how the monitor process repeats four phases : request waiting phase, recording phase, acceptance phase, and confirmation phase. They are explained as follows.

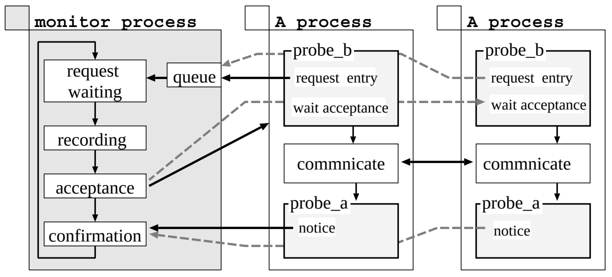

Figure 4. Monitor process.

1. Request waiting phase.
   The monitor process waits for a request from a target process. When a target process executes a communication, then the inserted before probe `probe_b()` sends a request to the monitor process. The request is queued.

2. Recording phase.
   The monitor process takes a request from the queue, and records the request in the file *seq*.

3. Acceptance phase.
   The monitor process sends an acceptance message to the target process. The target process continues execution. The target process can communicate with other processes.

4. Confirmation phase.
   After the communication with the target process, the after probe `probe_a()` sends a notification message of communication end. The monitor process waits for the notification. If the monitor process receives the notification, then it returns to the first phase.

The monitor process suspends communications of other target processes. If the mode of the receiving system call is the blocking mode, then all processes may get into deadlock. A target process sends a request for receiving a message to the monitor process. The monitor process accepts the request and waits for the notification of the receiving end. If the mode of the receiving system call is the blocking mode, then the target process waits for message arrival from other processes. However, if the monitor process prevents the other process's communication, then no messages will arrive at the waiting system call. Then all processes get into deadlock.

To avoid deadlock, the monitor process examines the state of the target process in the 4th phase[7]. When the target process gets into the message waiting state, the monitor process can know it. Then the monitor process confirms that the target process terminates the receiving system call, and the monitor process returns to the request waiting phase.

The monitor process can also monitor communication with *pipe*. The socket system is a revision of the pipe system. Then, the monitor process is able to monitor pipe communications in the same way as in socket.

(3) Coverage calculation program.

The present version of the coverage calculation program calculates coverage of OSC_2 using two files, F and *seq*. The file F produced by the transformation program represents a table of all concurrency statements (communication system calls), and the file *seq* produced by the monitor process represents a sequence of the concurrency statements.

At first, the coverage calculation program constructs a set $TE(\mathsf{OSC}_2)$ using the file F. Next, it picks up the executed test-events (ordered pairs) from the file *seq*, and appends these worked pairs to the set W. For example, if the content of the file *seq* is

$$seq =< a, b, c, d, e >,$$

where $a, b, c, d, e \in Sync$, then W (worked pairs) will be

$$W = \{< a, b >, < b, c >, < c, d >, < d, e >\},$$

provided that W has no identical pairs.

In the case of OSC_k, the executed k-length sequence can be picked from the file *seq* in the same way as in the case of OSC_2. But, we have not implemented OSC_k coverage calculation function yet.

Finally, this program calculates coverage according to the definition of coverage *Cov*,

$$Cov = \frac{|W|}{|TE(Cri)|} \times 100\ (\%),$$

where W is a set of worked test-events, $TE(Cri)$ is a set of test-events of a testing criterion Cri, and $|\cdot|$ represents the size of the set.

4 An experiment

We applied the monitoring tool to the `phone` program[3]. The `phone` program is a character based conversation program on terminals. For people with Internet access, the source of the `phone` program is available in `phone.tar` through `ftp` from some ftp site. We developed our monitoring tool and executed the `phone` on a Sun3/80 workstation with SunOS.4.1.1. In this section, we report the experiment and the results.

4.1 `phone` program

The `phone` program consists of three subprograms : client(`client`), master daemon (`masterd`), and conversation daemon (`convd`). One master daemon must stay on each UNIX machine. A client process is created when a user types a `phone` command. A conversation daemon process is created for every conversation by the master daemon.

A conversation on `phone` is constructed in order of the following steps. First, a user types a command "`phone` *username*", then a client process is created. The client process sends a request message to the master daemon process. If the destination user exists on the same local machine, then the master daemon on the machine calls the destination user, or else the local master daemon calls a remote master daemon process which stays on the same machine which is used by the destination user. Then the destination user is called by the master daemon process which stays at the destination user using the machine. When the destination user types "`phone`", then a client process is created for the destination user. The destination user's client sends a message to the caller's master daemon. When the first master daemon receives the message, it creates a conversation daemon, then both client daemons are connected with the conversation daemon. Finally, a conversation starts. Figure 5 shows the outline of a conversation with `phone` program.

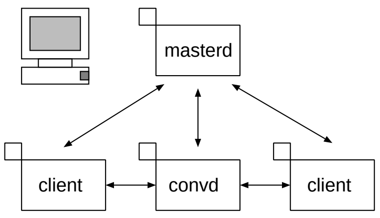

Figure 5. `Phone` **program.**

4.2 Results

In the `phone` program source, there are 44 communication statements. The number of test-events of OSC_2 is 1,936 ($= 44^2$) , consequently. Table 1 shows the communication statements in each program. And, Figure 6 shows the contents of the file F, produced by the transformation program.

We executed the `phone` program 37 times. The total count of communication statement execution is 44,017. Tables 2 and 3 describe the results of the experiment.

Table 2 describes the executed communication statements. Table 2 shows that there are 31 executed com-

Table 1. Communication statements in each source of `phone` program.

	`read()`	`write()`	`recvfrom()`	`sendto()`	`select()`	Total
`client`	3	6	1	2	2	14
`masterd`	0	0	1	17	1	19
`convd`	2	8	0	0	1	11
Total	5	14	2	19	4	44

Table 2. Executed communication statements.

`client`						
	`read()`	`write()`	`recvfrom()`	`sendto()`	`select()`	Total
statements	3	6	1	2	2	14
executed statements	3	6	1	1	2	13
non-executed statements	0	0	0	1	0	1
`masterd`						
	`read()`	`write()`	`recvfrom()`	`sendto()`	`select()`	Total
statements	0	0	1	17	1	19
executed statements	0	0	1	7	1	9
non-executed statements	0	0	0	10	0	10
`convd`						
	`read()`	`write()`	`recvfrom()`	`sendto()`	`select()`	Total
statements	2	8	0	0	1	11
executed statements	2	6	0	0	1	9
non-executed statements	0	2	0	0	0	2
Total of non-executed statements						13

munication statements and then 13 non-executed communication statements. The OSC_1 requires execution of all concurrency statements (communication statements) at least once. (A sequence of length 1 of concurrency statement means a statement.) $TE(\mathsf{OSC}_1)$ is the set of all concurrency(communication) statements. Then, the coverage of OSC_1 is ,

$$\frac{31}{44} \times 100 = 70\%.$$

Table 3 describes the number of executed ordered pairs. Table 3 shows that there are 666 executed ordered pairs in $TE(\mathsf{OSC}_2)$. After all, the coverage of OSC_2 is

$$\frac{666}{1936} \times 100 = 34.4\%.$$

Table 3. Executed ordered pairs.

before\ after	`client`	`masterd`	`convd`
`client`	126	89	94
`masterd`	85	20	58
`convd`	84	51	59

5 Analysis

The 34.4% coverage is not sufficient. We analyze all non-executed communication statements and ordered pairs in the `phone` program whether they are executable or not. The number of all pairs is 1,936, and 666 pairs were executed in the experiment. 1,276 pairs have not been executed. Table 4 shows the breakdowns of the analysis.

Table 4. Analysis of ordered pairs.

Total number of Ordered pairs	1,936
Executed pairs	666
Non-executed pairs	1,270
1,270 Non-executed pairs	
(a) Non-executable pairs	87
(b) Communication between machines	405
(c) Signal procedures	215
(d) Error messages	427
Subtotal of (a),(b),(c),(d)	1,134
(e) The rest	136

We describe what each enumerated columns in Table

```
connect_daemon 36 stream 12
keyboard 36 stream 12
keyboard 38 tochild 12
main 113 dummy 16
readctl 7 ctl 11
readstream 11 stream 13
readchild 5 fromchild 13
readchild 21 stream 12
sendit 8 ctl 10
sendit 14 dummy 16
sendit 18 ctl 13
sigstop 6 stream 12
sigstop 9 stream 12
who 23 ctl 10
sigchld 25 misc 10
daemon 9 misc 10
daemon 21 misc 10
daemon 26 misc 10
daemon 32 misc 10
daemon 35 misc 10
daemon 46 misc 10
inquire 6 misc 10
answer 5 misc 10
dopage 18 misc 10
service 16 dummy 16
service 21 sock 11
page 9 misc 10
page 33 misc 10
reinvite 9 misc 10
who 9 misc 10
who 21 misc 10
who 29 misc 10
who 31 misc 10
main 30 dummy 16
main 40 cslot->fd 13
main 68 slots[i].fd 12
service 19 new 12
service 23 new 13
sendit 4 slots[i].fd 12
intro 9 fd 12
intro 14 fd 12
intro 30 fd 12
intro 32 fd 12
fatal 16 slots[s].fd 12
```

Figure 6. Contents of communication statements in the phone **program. (First field: function name, second field: line number, third field:socket)**

4 mean as below.

(a) non-executable pairs. : 87 pairs.
As shown in Table 2, there are 13 non-executed communication statements. One of the 13 non-executed statements is never an executable statement. This statement is reserved for future extension in advance. Then, $44 \times 2 - 1 = 87$ ordered pairs, which include the non-executable statement, were not executed.

(b) Communication statements between processes on different machines. : 405 pairs.
As mentioned in section 3, our monitoring tool can not monitor an interprocess communication between different machines yet. The tool can not record the communications between machines. However, if the tool supports the recording of the communications between different machines, these 405 pairs will be covered.

There are 5 communication statements which are only used for communications between different machines. There are $(\ (44-1) \times 2\ ) \times 5 - 5^2 = 405$ ordered pairs which include the statements of communication between machines. The 405 pairs were not recorded.

(c) Communication statements in *signal* processing procedures. : 215 pairs.
In the `phone` program, there are 4 communication statements in *signal* processing procedures. The monitoring tool ignores signal processing, because the tool must keep equivalence between the real execution order of concurrency statements and the order of concurrency statements which are recorded in the file *seq*. If a target process receives a signal, then the process will be forced to call a procedure to process the signal. The target process will be free from the control of the monitor process, and the monitor process will exit. 215 ordered pairs which include those 4 statements were not executed.

(d) Communication statements for error message. : 427 pairs.
In the `phone` program, 7 communication statements are used for error messages. Error messages were not issued in this experiment. $(\ (44-1-5-4) \times 2\ ) \times 7 - 7^2 = 427$ ordered pairs were not executed, where the pairs which already appear in (a), (b) and (c) are not counted.

(e) The rest : 136 pairs.
There are 136 pairs which are the rest of the non-executed ordered pairs. We analyzed these pairs.

Then we found out that all of the 136 pairs are executable pairs, but they were not executed. If the tool can arbitrarily change the execution timing of each communication, then the 136 pairs will be executed. To execute the 136 pairs, an execution timing controller is required.

In the above analysis, pairs in (a) are not executable. However, if the monitoring tool is improved and it can monitor communications between different machines and communications in signal procedures, then all pairs in (b) and (c) will be executed. To execute the pairs in (d), it is necessary to produce errors in execution of the `phone` program. And if the tool has an execution timing controller, then the pairs in (e) will be executed. Table 5 shows the summary of the above analyses.

Table 5. Ordered pairs of the `phone` program.

Total number of ordered pairs	1,936
Executed pairs in the experiment	666
Pairs in (b),(c)	620
Pairs in (d),(e)	563
Possibly, executable pairs $(= 666 + 620 + 563)$	1,849
Non-executable pairs : (a)	87

6 Discussion

In the previous section, we showed that the prototype tool can monitor and record concurrent behavior (the execution of concurrency statements). The sequence *seq* recorded by the monitor program can not only be used for testing but for debugging information. And we also presented that the prototype tool can calculate the coverage of OSC_2. And then, we expected that our monitor tool could be used for practical testing. However, the tool still has some problems for practical use. In this section, we discuss the remaining problems of the prototype tool. And we also discuss other applications of the tool.

6.1 Improvement

The coverage of OSC_2 is about 34%, which is unsatisfactory. However, as mentioned in section 5, if we improve the prototype tool, the coverage of OSC_2 may possibly increase to 95.5% $(= 1,849/1936)$. The prototype tool must be enhanced to support the following functions for practical use in the future.

1. Communications between different machines.

   The prototype tool can not record the execution of concurrency statements on another machine. The communication between the monitor and the probe is implemented with the *shared memory*, the *semaphore*, and the *message queue* system calls of UNIX[7]. These system calls can work on the same machine.

   The monitor and the probe should be implemented with the socket to record the execution order of concurrency statements executed on different machines.

2. Signal procedures.

   The signals are used for sudden interruption, cyclic routine and error routine. When a process receives a signal, the process is interrupted and goes to the procedure which corresponds to the signal.

   If there is a concurrency statement in a signal procedure of a process in the tested program, the program may possibly be in deadlock. The monitor process forces sequential execution of all concurrency statements.

3. Error simulator.

   It is difficult to occur error states for `phone` program by users. For supporting program execution at testing, a test environment should have a simulation feature of input errors.

4. Execution timing controller.

   To execute the non-executed ordered sequences, a mechanism to force execution is needed. Tai *et al.*[9] developed an approach to reproduce the *entry* call arrival and rendezvous sequence (called *Syn-sequence*) of an Ada program using an added *task* for controlling the execution order. Our tool needs a mechanism to force an execution sequence deterministically at the execution of the tested program.

Feature 1. and 2 require improvement of our tool. Feature 3. and 4 require a program execution supporting tool as a test environment.

6.2 Probe effect

The program transformation approach must generate overhead arising from the insertion of the probes and the monitoring process.

1. Justification.

   Our tool transforms the tested program, inserts probes and adds the monitor process. Then the execution of concurrency statements in a transformed program is sequentialized to record the execution order. But the transformation doesn't introduce a new error. If an error occurs in the execution of a transformed program, the error also occurs in the execution of the original.

2. Overhead.

   Our tool charges large time overhead to the transformed program. We measured the execution time from program beginning until session establishment of both the original `phone` program and the transformed `phone` . Table 6 shows the mean time of execution. We executed the `phone` program 10 times, both the original and the transformed, on the same machine Sun3/80. The times were measured with the `time` command of UNIX.

Table 6. Execution time.

	Original (sec)	Transformed (sec)
Real time	9.4	270
User time	0.5	0.5
System time	0.5	0.5

Table 6 shows that the execution time of the transformed program is about 27 times larger than that of the original. One of the reasons for the large overhead is the recording. The monitor process records the execution sequence of concurrency statements in the file *seq* as shown in Figure 7. The access time to the file is much longer than that to the main memory.

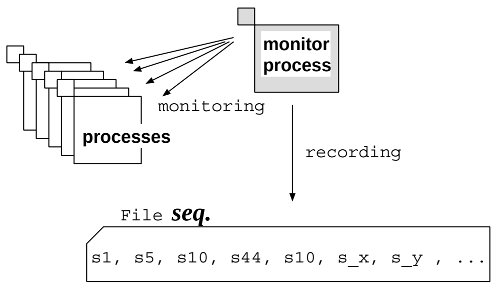

Figure 7. Present system. The monitor process records execution sequence in a file.

One solution to reduce the overhead is for all records to be on the main memory. In this way, the monitor doesn't record the execution sequence, because the sequence becames huge very quickly. Instead of sequence, the monitor system allocates an array of boolean for each test-event as shown in Figure 8. The monitor checks executed pairs (or k-length sequence). The access time to the main memory is much shorter than that to the file. Then, we can hope to reduce the time overhead.

However, it also produces scale problems. The size of the array is in proportion to the number of test-events. The number of test-events of OSC_k is $|TE(OSC_k)|^k$, an exponential number[1]. If k is large, the size of array becomes huge. Then, it is desirable to take the option to select both methods according to the scale of the test-events. In the case of the OSC_2, the number of test-events is the square of the number of all concurrency statements, and it may be possible to allocate the test-event array on memory.

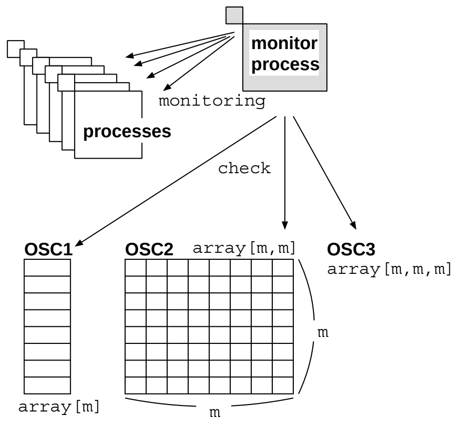

Figure 8. One solution. Allocate array according to OSC_k.

6.3 For another application

The prototype tool records the execution sequence in a file. The sequence information can be used to a debugging information. If a concurrency statement has an error in execution and the monitor suspends execution of all processes in the program, then it can specify which statement has the error. And the execution sequence which has the error can be used to reproduce the error. If our tool supports deterministic execution according to a given sequence, then the user can trace the previous execution.

7 Conclusion

We have developed the monitoring tool of concurrent program execution. The tool monitors communication between processes in a C concurrent program and reports the coverage of OSC_2 (Ordered Sequence Testing Criterion) which is used for testing evaluation. This paper describes the tool and the experience.

The monitoring tool measures the coverage of OSC_2 on the `phone` program which supports character based conversation in a computer network. 34% ordered pairs of the concurrency statements in the `phone` program are executed in testing. It is possible to record concurrent behavior and to evaluate testing sufficiency with the monitoring tool. However, the following improvement of the tool and new execution support tool are necessary for practical usage in testing.

1. Monitoring communication between different machines.
2. Recording execution of concurrency statements in signal procedures.
3. Input simulator for occurrences of error states in C concurrent programs.
4. Execution timing controller for forced execution of C concurrent programs.

It is possible to increase the coverage of `phone` program until 95.5% by those new features.

We are improving the monitoring tool and are planing the new test environment which includes input simulator and execution timing controller. In the future, we will try empirical studies of improvement of concurrent program reliability.

References

[1] E. Itoh, Y. Kawaguchi, Z. Furukawa and K. Ushijima : *Ordered Sequence Testing Criteria for Concurrent Programs and The Support Tool*, Proc. of APSEC'94, pp.236-245, 1994.

[2] E. Itoh, Y. Kawaguchi, Z. Furukawa and K. Ushijima : *Reliability of Testing based on Ordered Sequence Testing Criteria for Concurrent Programs*, Trans. IPSJ, Vol.36, No.9, pp.2195-2205, 1996 (in Japanese).

[3] Original : Jonathan C. Broome (`broome@ucb-vax.berkeley.edu`), Japanized : H.Tachibana (`tachi@cs.titech.junet`), K.Odajima (`odajima@mt.cs.keio.ac.jp`) :`phone program`, `ftp://ftp.csce.kyushu-u.ac.jp/pub/ok-phone1.2.tar.gz`

[4] M. Ben-Ari : *Principles of Concurrent Programming,* Prentice Hall International, Inc. 1982.

[5] W. E. Howden : *Reliability of the Path Analysis Testing Strategy*, IEEE Trans. Softw. Eng., Vol.SE-3, No.4, pp.226-278, 1976.

[6] S. J. Leffler, M. K. Mckusuck, M. J. Karels and J. S. Quarterman : *The Design and Implementation of the* 4.3*BSD UNIX Operating System*, Addison-Wesley Publishing Co. Inc. 1989.

[7] W. R. Stevens : *UNIX NETWORK PROGRAMMING*, Prentice Hall Inc., 1990.

[8] R. N. Taylor,D. L. Levine and C. D. Kelly : *Structural Testing of Concurrent Programs*, IEEE Trans. Softw. Eng., Vol 18, No.3, pp.206-215, 1992.

[9] K. C. Tai, R. H. Carver and E. E. Obaid : *Debugging Concurrent Ada Programs by Deterministic Execution*, IEEE Trans. Softw. Eng., Vol.17, No.1, pp.45-63, 1992.