Ordered Sequence Testing Criteria for Concurrent Programs and the Support Tool

Eisuke Itoh*, Yutaka Kawaguchi*, Zengo Furukawa** and Kazuo Ushijima*
* Dept. of Computer Science and Communication Engineering,
** Educational Center for Information Processing,
Kyushu University.
6-10-1 hakozaki, Higashi-ku, Fukuoka, 812, JAPAN.
E-mail: itou, kawaguti, zengo, ushijima@csce.kyushu-u.ac.jp

Abstract

Testing of programs is important to increase reliability of the programs. Coverage is a ratio of the number of worked test-events to all test-events, and it is used as a metric of testing sufficiency and reliability. The test-events are defined by a testing criterion.

Some testing criteria are proposed for evaluating testing sufficiency of sequential programs. However, the criteria are inadequate for concurrent programs. New testing criteria must be introduced for concurrent programs.

*This paper proposes new testing criteria, Ordered Sequence Criteria (*OSC *for short) for concurrent programs.* OSC *are concerned with interprocess communication and synchronization. An* OSC_k *selects k-length sequences of statements related to communication or synchronization. The sequences should be executed at least once in testing.* OSC_k *presents various levels of testing according to values of k. The* OSC_2 *is reliable for a program which is correct or which includes communication errors. A prototype is implemented for coverage measuring based on* OSC_2.

1 Introduction

Testing has been playing an important role in program development. Most programs are tested to increase their reliability. Generally, a program is tested through the following 5 steps.

1. Test case generation.
2. Test data selection.
3. Program testing.
4. Judgment of results.
5. Evaluation of testing sufficiency.

We are interested in the 5th step; evaluation of testing sufficiency. Many testers use coverage for evaluating testing sufficiency. Coverage is a ratio of worked test-events to all test-events, which should be executed in the program testing step. Coverage reveals us the testing sufficiency quantitatively, and the test-events are defined by a testing criterion.

Various testing criteria and tools based on the criteria have been used to increase confidence for conventional sequential programs. For examples, four well-established testing criteria for sequential programs are statements testing (C_0), branch testing (C_1), path testing (C_∞), and all-du-path testing. The path testing criterion (C_∞) is the strongest one of all control-flow testing criteria[5]. The all-du-path criterion is the strongest one of data-flow path selection criteria[2].

In recent years, concurrent programs are used practically. It is obvious that only using testing criteria proposed for sequential programs is inadequate for evaluating testing reliability of a concurrent program, because the testing criteria proposed for sequential programs do not care the two characteristics of concurrent program. The first characteristic is nondeterministic execution and the second is interprocess communication or syn-

chronization.

There are some testing criteria for concurrent programs. Taylor *et al.*[10] proposed the concept of structural testing of concurrent programs. They defined the concurrency state and graph as a model of concurrent programs, and propose some testing criteria based on the concurrency graph. The concurrency graph consists of nodes and edges. A node denotes a combination of states of each process, and an edge denotes state transition. The concurrency state and graph has two difficulties.

1. The larger the number of states of each process, the larger the size of the concurrency graph.

2. The number of processes must be fixed before the concurrency state and graph is constructed.

Most concurrent programs generate dynamic ally process. Testing criteria based on the concurrency graph, therefore, are not always suitable for testing of concurrent program.

Tai *et al.*[9] developed an approach to reproducing the *entry* call arrival and rendezvous sequence (called *Syn-sequence*) of an Ada program using an added *task* for controlling the execution order. And they proposed a testing criterion based on the Syn-sequence.

Furukawa *et al.*[4] proposed a testing criterion for Ada programs, named rendezvous path testing criterion. The rendezvous path testing criterion requires execution of pairs *entry* call statement and *accept* statement. But the rendezvous path testing criterion can only apply to Ada programs testing.

The rest of this paper is organized as follows. In section 2, we define some terms and notations. In section 3, we consider an execution sequence of concurrent program based on interleaving model. In section 4, we propose new testing criteria, named ordered sequence criteria (OSC for short), and show the test-events of OSC in an example concurrent program. In section 5, we discuss reliability of OSC , discuss feasibility of a test-event of OSC , and study subsumption relations OSC with another testing criteria. In section 6, we describe a prototype of a coverage measuring system and experience of OSC . Finally, we conclude this paper in section 7.

2 Test-events and Coverage

In this section, we present some definitions of terms and notations.

The first, we define a set of test-events with a testing criterion. A test-event is an event that tester ought to execute in the program testing step and the test-events are defined by a testing criterion.

Definition 1 Test-events set : $TE(cri)$
Let cri is a testing criterion. We express a set of test-events defined by cri as $TE(cri)$. □

For example, let consider a program and testing criterion C_0(statements testing). The C_0 requires execution of all statements of the program at least once. The test-events set of the program with C_0 is

$$TE(C_0) = \{all\ statements\ in\ the\ program\}.$$

Next, we define *coverage*. Coverage is a ratio of the number of worked test-events to the number of test-events in TE. Definition of coverage Cov is presented.

Definition 2 Coverage : Cov

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

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

Next, we define a *concurrency statement* and a set of the concurrency statements.

Definition 3 *Concurrency statement* :
We call a statement related to interprocess communication or synchronization as a *concurrency statement*.

□

The instances of concurrency statement are the *fork()* statement which is an UNIX system call, or *entry call* and *accept* statements of Ada rendezvous, or definition and use statements of a shared variable.

We define a set consists of concurrency statements, and express this set as *Sync*.

Definition 4 *Sync* :

$$Sync = \{all\ concurrency\ statements\ in\ a\ program\ \}\ .$$

□

3 Execution Sequences of Concurrent Programs

In this section, we consider *execution sequences* of a concurrent program on the interleaving model[1]. The interleaving model have been used to represent a behavior of a concurrent program running on a single processor computer.

On the interleaving model, a concurrent program behaves as follows. A concurrent program consists of some *processes*, each of which behaves like a sequential program. The processes communicate and/or synchronize each other. A processor selects one statement from a process and executes it. A statement selection is arbitrary except the selection of concurrency statement.

We express a concurrent program **P** :

$$\mathbf{P} = (P_1, P_2, \cdots P_n)\ ,$$

where P_i $(1 \leq i \leq n)$ is a *process* and n is the number of processes in **P**.

We describe a *process* as a control flow graph. In the control flow graph, each node represents a statement, each edge between nodes represents the flow of control from one node to another. We express a process P_i :

$$P_i = (N_i, E_i, s_i, f_i), 1 \leq i \leq n\ ,$$

where N_i is a set of nodes, E_i is a set of edges, s_i is a start node and f_i is a terminate node of the process P_i.

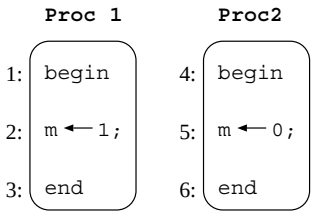

Figure 1. A concurrent program.

An execution of a process is described as a *path* through the control flow graph of the process. We express a path :

$$path =< a_1, a_2, \cdots a_k >\ ,$$

where $a_j \in N_i$, $a_1 = s_i$, $a_k = f_i$ and $<a_j, a_{j+1}> \in E_i$ $(1 \leq j < k)$.

An *execution sequence* of a program is described as a sequence of statements in the program, and this sequence is made from shuffling of the *paths* of processes. Provided that any execution sequence must satisfy the constraints with the interprocess communication and synchronization. We express an execution sequence *exc_seq* :

$$exc\_seq =< b_1, b_2, \cdots, b_l >\ ,$$

where b_j $(1 \leq j \leq l)$ is a statements in the program, l is the length of *exec_seq*.

We can define a set of all execution sequences :

$$Exc\_Seq(\mathbf{P}) = \{exe\_seq\}.$$

For example, **P** is a program in Figure 1. The **P** has two processes, `Proc1` and `Proc2`. The *path* of `Proc1` is <1,2,3>, and the *path* of `Proc2` is <4,5,6>. The set of execution sequences of **P** is

$$\begin{aligned} Exc\_Seq(\mathbf{P}) = \{ & \\ & < 123456 >, \\ & < 124356 >, < 124536 >, < 124563 >, \\ & < 142356 >, < 142536 >, < 142563 >, \\ & \quad < 145236 >, < 145263 >, < 145623 >, \\ & < 412356 >, < 412536 >, < 412563 >, \\ & \quad < 415236 >, < 415263 >, < 415623 >, \\ & < 451236 >, < 451263 >, < 451623 >, \\ & < 456123 > \}. \end{aligned}$$

If all *exc_seq*s in a program are executed (covered), it is possible to detect all errors in the program. However, the number of elements of Exc_Seq is generally infinite, then all *exc_seq*s are never covered.

4 OSC for concurrent programs

In this section, we propose new testing criteria, OSC (Ordered Sequence Criteria). An OSC_k is defined with respect to execution order of concurrency statements.

4.1 Definition of OSC

If all execution sequences of a concurrent program are covered in testing, it is possible to detect all errors in the program. But the execution sequences are so many that it is impossible to cover all of them. The characteristics of concurrent programs are nondeterministic execution and interprocess communication and synchronization. We are only interest in the characteristics as a test-event.

In execution of a concurrent program as illustrated in the previous section Figure 1, the difference of the order of execution of concurrency statements may lead to a different output. We consider that the order of execution of concurrency statements should be a test-event for concurrent program testing. In the concrete, we consider that a k-length ($k \geq 1$) ordered sequence which consist of concurrency statements should be a test-events.

Some programming languages have functions for dynamic process generation, for example, *task type* of Ada, and *fork() system call* of UNIX. Most concurrent programs accordingly have dynamic process generation. The generated processes (clones) have a same source but their behavior may be different. If a clone process communicates and/or synchronizes with another clone, as shown in Figure 2, a same statement but in another process may be executed continuously. Then it is possible to appear a subsequence which has a same statement continuously in an execution sequence. Thus, it is necessary to append some ordered sequences which include subsequence consist of same statements to the TE.

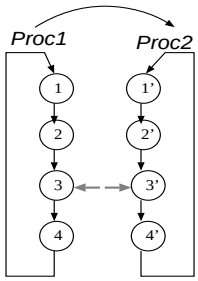

Figure 2. Clone processes. Where a circle denotes a node of a control flow graph. A solid arrow denotes a control flow, and a dash arrow denotes communication or synchronization.

Now we propose new testing criteria, named OSC . An OSC_k requires that at least once execution of k-length ordered sequences in testing, which a sequence consists of k concurrency statements.

Definition 5 OSC_k :
Suppose that **P** is a concurrent program and $Sync$ is a set of all concurrency statements in **P**. Construct k-length ordered sequences with concurrency statements. An OSC_k requires execution of the all k-length ordered sequences at least once. □

Test-events on OSC_k ($k \geq 1$) is described as the following expression.

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

OSC_k presents various levels of testing according to values of k. OSC_k has the following effects according to values of k.

1. $k = 1$
   OSC_1 requires execution of all concurrency statements at least once. The set $TE(\mathsf{OSC}_1)$ is accordingly equal to $Sync$. If a program satisfies OSC_1 in testing, then all concurrency statements are tested at least once. Where, a term *satisfy* means that the coverage of the testing criterion is 100% in the program testing.

2. $k = 2$
   OSC_2 requires execution of 2-length ordered sequences of concurrency statements at least once. Here we call a 2-length sequence as an *ordered pair*. If a process communicates or synchronizes with another process, the communication or synchronization is represented as an ordered pair in the execution sequence. If a program satisfies OSC_2 in testing, then all communication and synchronization between two processes are tested at least once.

3. $k = \infty$
   If a concurrent program has some loops, the length of an execution sequence may be infinite. And the length of the sequence which is extracted concurrency statements from the execution sequence may be also infinite. OSC_∞ can be defined as a testing criterion which requires at least once execution of infinite length ordered sequences. However the OSC_∞ is never satisfied.

4.2 An example : The Producer-Consumer Problem

As further clarification of the ideas presented above subsection, we present the test-events of OSC from the Producer-Consumer Problem. The control flow graph of the Producer-Consumer Problem using semaphore is illustrated in the Figure 3. In the program, the Producer process products a data and puts it into a buffer, the Consumer process gets the data from the buffer and consumes it. The two processes repeat their works. The Buffer is shared with the Producer and the Consumer.

The set $Sync$ of the program is

$$Sync = \{2, 3, 5, 6, 7, 8, 10, 11\}.$$

Now, we express the $TE(\mathsf{OSC}_k)$ (k=1,2,3) of Producer-Consumer Problem. The test-events according to the value of k are expressed as follows.

$TE(\mathsf{OSC}_1)$ is

$$TE(\mathsf{OSC}_1) = \{< 2 >, < 3 >, < 5 >, < 6 >, < 7 >, < 8 >, < 10 >, < 11 >\},$$

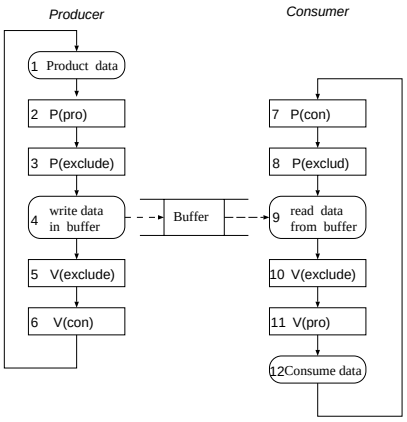

Figure 3. The control flow graph of the Producer-Consumer Problem. A solid arrow denotes an edge, a dash arrow denotes a communication.

where a number in the expression corresponds to a concurrency statement number in Figure 3. The size of $TE(\mathsf{OSC}_1)$ is 8 ($=|Sync|$).

$TE(\mathsf{OSC}_2)$ is

$$\begin{aligned}
TE(\mathsf{OSC}_2) = \{ & \\
& < 2,2 >, < 2,3 >, < 2,5 >, < 2,6 >, \\
& \quad < 2,7 >, < 2,8 >, < 2,10 >, < 2,11 >, \\
& < 3,2 >, < 3,3 >, < 3,5 >, < 3,6 >, \\
& \quad < 3,7 >, < 3,8 >, < 3,10 >, < 3,11 >, \\
& < 5,2 >, < 5,3 >, < 5,5 >, < 5,6 >, \\
& \quad < 5,7 >, < 5,8 >, < 5,10 >, < 5,11 >, \\
& \qquad \vdots \\
& < 11,2 >, < 11,3 >, < 11,5 >, < 11,6 >, \\
& \quad < 11,7 >, < 11,8 >, < 11,10 >, < 11,11 >\}.
\end{aligned}$$

The size of $TE(\mathsf{OSC}_2)$ is 64 ($=|Sync|^2$) .

$TE(\mathsf{OSC}_3)$ is

$$\begin{aligned}
TE(\mathsf{OSC}_3) = \{ & \\
& < 2,2,2 >, < 2,2,3 >, \cdots \quad < 2,11,11 >, \\
& < 3,2,2 >, < 3,2,3 >, \quad < 3,11,11 >, \\
& < 5,2,2 >, < 5,2,3 >, \quad < 5,11,11 >, \\
& \vdots \\
& < 11,2,2 >, < 11,2,3 >, \cdots \quad < 11,11,11 >\} .
\end{aligned}$$

The size of $TE(\mathsf{OSC}_2)$ is 512 ($=|Sync|^3$).

After all, the size of $TE(\mathsf{OSC}_k)$ is equal to $|Sync|^k$.

5 Discussion

In this section, we discuss the reliability of OSC for concurrent programs, the feasibility of test-events on OSC , and the subsumption relations OSC with another testing criteria.

5.1 Reliability

Howden[5] defined a term *reliable* as follow. If a program satisfies a testing criterion Cri and all errors in the program are detected, then Cri is *reliable* for the program. However, the testing criterion which is reliable for any program is only *exhaustive test*[11]. Any practical testing criterion is only reliable for a program which is correct or includes some particular errors.

To analyze the reliability of OSC for programs, we analyze errors of concurrent programs. An error of concurrent programs can be typed as follows[1, 3].

1. Inner process errors.
   Inner process errors like the errors of sequential programs.

2. Communication errors.

3. Synchronization errors.

OSC only interest interprocess communication and synchronization. Therefore, OSC are not reliable for a program which includes inner process errors. We consider the reliability of OSC for a program which includes communication errors. A communication error is further typed as follows.

(2-a) *Complete communication errors*
If a process always communicates error data with another process, then this communication is a *complete communication error*.

(2-b) *Partial communication errors*
If a process often communicates error data with another process, then this communication is a *partial communication error*.

OSC_2 is reliable for a program which is correct or which includes perfect communication errors. We proof.

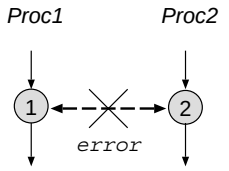

Figure 4. Communication error.

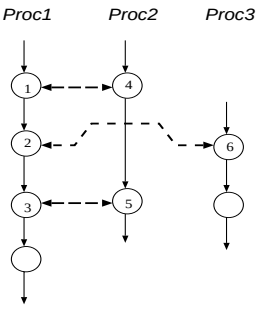

Figure 5. A concurrent program which have unexecutable ordered sequences. A numbered circle denotes a concurrency statement.

Let consider communication from statement 1 to 2 shown in Figure 4. And let the communication is an *complete communication error*. If the communication occurs in an execution of the program, the execution order of statement 1 and 2 is only either $< 1, 2 >$ or $< 2, 1 >$. The two orders must be included in $TE(\mathsf{OSC}_2)$. Therefore, if the program satisfies OSC_2 in testing, the error must be actualized. Hence, OSC_2 is *reliable* for a program which is correct or includes complete communication errors.

5.2 Feasibility

In this subsection, we consider the feasibility of the test-event of OSC .

There may be many unexecutable events in a set $TE(\mathsf{OSC})$, because we define a test-event for OSC_k as a $k-$tuple of concurrency statements.

Let consider a concurrent program illustrated in Figure 5 and the OSC_2. $TE(\mathsf{OSC}_2)$ of the program is

$$
\begin{aligned}
TE(\mathsf{OSC}_2) = \{ & \\
& < 1,1 >, < 1,2 >, < 1,3 >, \cdots, < 1,6 > \\
& < 2,1 >, < 2,2 >, < 2,3 >, \cdots, < 2,6 > \\
& \qquad \vdots
\end{aligned}
$$

$< 6,1 >, < 6,2 >, < 6,3 >, \cdots, < 6,6 >$ } .

In the set $TE(\mathsf{OSC}_2)$, there are many unexecutable ordered sequences. The same number pairs like $< 1,1 >$ or $< 2,2 >$ are never executed. The ordered pairs like $< 1,3 >, < 2,1 >, < 3,1 >$, which have reverse or skipping order through a control flow graph of a process, are also never executed. The coverage of OSC_2 consequently never amounts to 100% in this program.

The same number pairs and reverse or skipping ordered pairs are introduced for testing of programs which have dynamic process generation. If a tested program has no dynamic process generation, then it is necessary to remove the same number pairs and the reverse or skipping ordered pairs from the test-event set.

5.3 Subsumption

In this section, we study subsumption relations of OSC with another testing criteria. First, we define a term *subsume* as follows.

Definition 6 *Subsume*
If a program satisfies a testing criterion A, and the program also satisfies a testing criterion B, then the A *subsumes* the B. And we express this relation as $A \supset B$. □

The following subsumption relations obviously stand.

$$C_{i+1} \supset C_i,\ i \geq 1.$$

$$\mathsf{OSC}_{i+1} \supset \mathsf{OSC}_i,\ i \geq 1.$$

The set $TE(C_0)$ consists of all statements in a program **P**, and) the set $TE(\mathsf{OSC}_1)$ is equal to *Sync* of **P**. That is, $TE(\mathsf{OSC})$ is a subset of $TE(C_0)$. Hence,

$$C_0 \supset \mathsf{OSC}_1 \ .$$

Because OSC_2 considers the execution orders of the concurrency statements in the other process, then the C_m ($m \geq 1$) don't subsume the OSC_k ($k \geq 2$).

$$C_m \not\supset \mathsf{OSC}_k\ (m \geq 1,\ k \geq 2) \ .$$

We are able to consider a super testing criterion, which requires execution of all sequences in $Exc\_Seq(\mathbf{P})$ at least once. We define this super testing criterion as CC_∞.

Definition 7 CC_∞ :
A testing criterion which requires at least once execution along with all possible execution sequence of a program in testing. □

A set of test-events of CC_∞ is described as

$$TE(CC_\infty) = Exc\_Seq(\mathbf{P}) \ .$$

On the CC_∞, the two following subsumption relations stand.

$$CC_\infty \supset C_\infty \ \wedge \ CC_\infty \supset \mathsf{OSC}_\infty \ .$$

Figure 6 shows all subsumption relations discussed in this subsection. In Figure 6, an arrow represents a subsumption relation. That is, a testing criterion on tail of an arrow subsumes the criterion which on the arrow head.

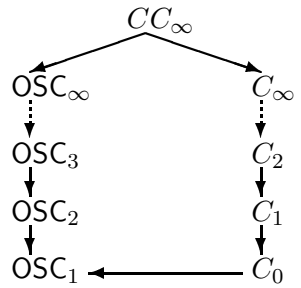

Figure 6. Subsumption relations.

6 Support Tool

To verify the effectiveness of OSC , we construct a prototype of measuring coverage system[6, 7]. The target of our system is C concurrent programs which run on UNIX and which include *system calls* related with semaphore operations. Since many C concurrent programs are running on UNIX, we expect to use our system in testing of many programs.

Concurrent processing of a C concurrent program is implemented through making use of *system call*s, which are library functions archived in

UNIX. There are some system calls for interprocess communication. For instance, semaphore, message pass, FIFO[8] and socket[12]. Since the semaphore is the most primitive function for concurrent processing, we select it for the first target. The semaphore is provided with a system call `semop` on the UNIX SYSTEM V[12].

6.1 Outline of the system

To note and record worked test-events in a single run, we adopt a program transformation approach. Figure 7 illustrates the outline of our system. Our system is constructed with 3 subsystems.

(1) Program Transformation part

The first subsystem's input is a source program **P** ; the outputs are a transformed program **P**' and a file F. This subsystem scans the **P** . If the subsystem finds a concurrency statement, then writes its statement number to the file F. At the same time, this subsystem adds a monitoring process to **P** and inserts many communication statements into existing processes. We call the inserted communication statements as a *probe*. **P**' exhibits all behavior of **P**.

(2) Execution part

The second subsystem makes the transformed program **P**' run. In the test execution, a single run will not generate complete coverage for any testing criterion except on the most trivial of programs. Thus the coverage statistics need to be retained and correlated over a set of runs. In a running, every concurrency statement must request permission from the monitor through a probe before proceeding; The monitor records the permission duly and permits their proceedings. This subsystem repeats the transformed program running and appending the activities to the file *seq*.

(3) Coverage Calculation part

The third subsystem's inputs are file F and *seq*, where F is output by the Program Transformation part and *seq* is output by the Program Execution part; this subsystem outputs the coverage of OSC_2. The file F hold all concurrency statement numbers in the tested program. This subsystem constructs a set $TE(\mathsf{OSC}_2)$, at the next, this subsystem picks up the worked test-events from the file *seq* and appends these worked events to the set W. For example, if the contents of the file *seq* is

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

where $a, b, c, \cdots \in Sync$. Then, W (worked test-events) is the following ordered pairs.

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

provided that the each of elements of W has no same one.

Finally the third subsystem calculates coverage according to the definition of coverage.

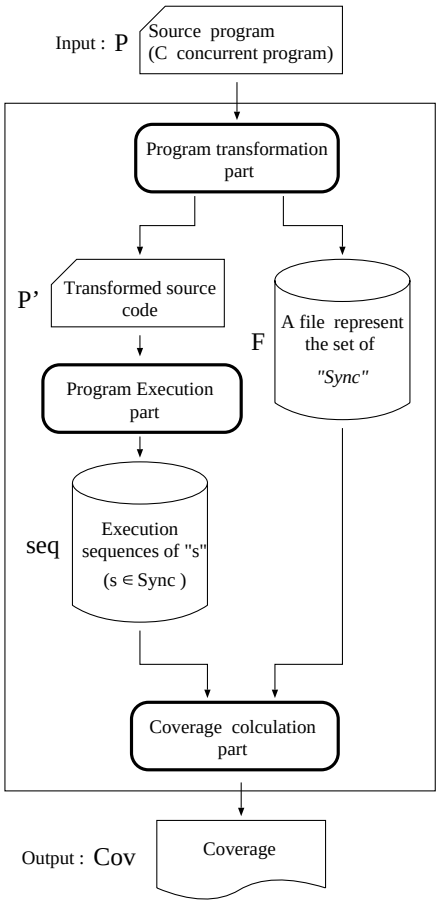

Figure 7. The outline of the measuring coverage system.

Table 1. Coverages of Producer-Consumer Problem.

Counts of `semop`	The number of executed test-events	Coverage (%)
215	7	10.9
586	16	25.0
1171	17	26.6
2341	20	31.2
3512	21	32.8
9363	21	32.8

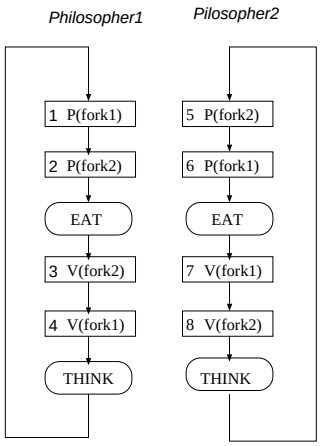

Figure 8. Flow graph of Two Dining Philosophers.

6.2 Experiments of the system

We measured coverage of OSC_2 of two concurrent programs, Producer-Consumer Problem and Dining philosophers with this system. The two programs are run on SUN Sparc Station ELC.

At first, we measured coverage of OSC_2 of Producer-Consumer Problem. The flow graph of Producer-Consumer Problem is illustrated in Figure 3. Our system transformed this program to record all concurrency activities, ran the transformed program several times and calculated coverage. The results of coverage is presented in Table 1.

Next, we measured the coverage of Two Dining Philosophers. Figure 8 shows the flow graph of Two Dining Philosophers.

The following expression represents $TE(\mathsf{OSC}_2)$ of Two Dining Philosophers, provided that a number in the expression corresponds to a number in Figure 8.

$$
\begin{aligned}
TE(\mathsf{OSC}_2) = \{ & \\
& <1,1>, <1,2>, \cdots <1,7>, <1,8>, \\
& <2,1>, <2,2>, \cdots \quad\quad <2,8>, \\
& \vdots \qquad\qquad \ddots \qquad \vdots \\
& <8,1>, <8,2>, \cdots \quad\quad <8,8> \}.
\end{aligned}
$$

Table 2. Coverage of Two Dinig Philosopher.

Counts of `semop`	The number of executed test-events	Coverage (%)
82	15	23.4
166	24	37.5
622	28	43.8
1012	28	43.8
1468	29	45.3

Table 2 shows the coverage of OSC_2 of Two Dining Philosophers.

Table 1 and Table2 shows us that there are bound of coverage. The reasons of the bound is the following two things:

1. As discussed in section5.2, there are unexecutable test-events in $TE(\mathsf{OSC}_2)$.
   It is necessary to remove the unexecutable test-events from TE.

2. In $TE(\mathsf{OSC}_2)$, there are some test-events which executable but yet covered.
   It is necessary to make an test data set to be cover the event or necessary to force a execution order.

The program transformation approach must appear an overhead incurred arising from the insertion of the probes and the monitoring process. We measured the execution time overhead of Producer Consumer Problem and Table 3 describes the time overhead. The time is measured with *time* command of UNIX.

Table 3 shows that the execution time of P' has very large overhead than that of original. The reason of the large overhead is structure of monitor. In the system, if a concurrency statement in **P**' is about to execute, then the monitor process

Table 3. Execution time table of original Producer-Consumer Problem and the translated program.

Produced Data (byte)	**P** (sec)	**P**' (sec)	Count of `semop`
40	0.0†	7.6	322
365	0.3	57.6	2922
1208	0.7	208.6	9600

† : less than 0.1 sec

write the event in the file *seq*. The execution time is consequently increased by the file inputs. It is necessary to tune up recording part of the added monitoring process.

6.3 An strategy for effective use of OSC

Described in above section, the coverage of OSC may not be 100%. It is necessary to devise the effective use of the coverage for evaluation of testing sufficiency. To use the coverage of OSC efficiency, we recommend a strategy, in which there is the three steps. We show the steps.

First step:

Execute testing a concurrent program with a set of test data and measure the coverage of OSC. If the coverage is not 100%, then the tester list up the yet covered test-events.

Second step:

Decide every yet covered test-event whether it is an executable event or not. If a test-event is not executable, then remove it from the test-events set, else remain it.

Third step:

To cover the remainder events, run the program and measure coverage again. But to cover the remainder events, it needs two works, test data selection and forcing an execution ordered sequence.

Let consider the test data selection. Test data commonly are not able to select automatically, then the tester has to select test data by oneself.

Next, let consider the forcing execution. Control of execution is one of the difficulties encountered in the testing of concurrent programs. It is nondeterministic which execution sequence is covered in an running and consequently it is nondeterministic which ordered sequence is covered. The processor nonpredictably choose one of the statement.

For forcing an ordered sequence in a running of a concurrent program, we should avail the added monitor process. Tai *et. al.*[9] have developed an approach to reproducing the entry call arrival and rendezvous sequence of an Ada program using an added task to control the execution sequence. Tai's approach can avail to our system.

It is necessary for forcing an ordered sequence to improve our system. The improvement is as follows. The system transforms the tested program **P** into **P**', such that **P**' allows as input not only test data but also ordered sequences of concurrency statements which tester want to cover. Of course, the **P**' may has moniter process and probes. The added monitor force the execution order of concurrency statements along with given ordered sequence. Every concurrency statements must request permission from monitor through probe before proceeding; the monitor permits the statement's proceeding which is next statement of given ordered sequence.

7 Conclusion

In this paper, we proposed new testing criteria OSC for evaluation of testing reliability. The OSC is simple and portable testing criteria for using in testing. The OSC is fundamentally based on the source code of a concurrent program, therefore, the number of test-events must be finite. We also discussed the reliability of the OSC for a concurrent program. The OSC_2 is reliable for a program which is correct or which includes complete communication errors. Furthermore we discussed the subsumption relation OSC with another testing criteria. We also implemented a measuring coverage system and measured the coverage of OSC_2 on two concurrent programs with this system. Since OSC_2 may request execution with unexecutable ordered sequences, the coverage of OSC_2 may not

be 100%. We consider a strategy for effective use the coverage of OSC.

There are some future problems:

- Improve the coverage evaluate system.
  - Apply another system calls.
    Our system only correspondence semaphore (`semop` system call) on UNIX SYSTEM V. There are many interprocess communication system call on UNIX. It is necessary to deal with these functions. Now, we are improving our system to record an function `socket`, which is a interprocess communication interface on BSD UNIX.
  - Expand length of ordered sequence.
    Our system only measures coverage of OSC_2. It is necessary to improve the system to measure coverage of OSC_k ($k \geq 3$).
- Expansion of the OSC .
  - Remove unexecutable sequences.
    As discussed in subsection5.2, the OSC_k may request executions of unexecutable ordered sequences. It is necessary to remove the unexecutable ordered sequences from the test-events set.
  - Forcing an ordered sequence.
    The system must allow the ordered sequences as input for forcing ordered sequences of concurrency statements.
- Distinction all dynamically generated processes in program.
  Our system is constructed on UNIX. On UNIX, any running process has an unique process numbers. For utilizing the process number, it is possible to more definite test.

Acknowledgment

We thank Prof. J. Cheng, Prof. H. Taniguchi for encouragement and valuable comments. We also thank T.Katayama, K.Simozono and H.Kashima for discussion about testing and system implementation.

References

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

[2] Clarke, L. A., Podgurski, A., Richardson, D. J. and Zeil, S. J. : *A Formal Evaluation of Data Flow Path Selection Criteria*, IEEE Trans. Softw. Eng., Vol.15, No.11, pp.1318-1332, 1989.

[3] Furukawa, Z. and Ushijima, K. : A Testing Method for Concurrent Programming, Proc. of 6th JSSST, pp.185-188, 1989 (in Japanese).

[4] Furukawa, Z. and Ushijima, K. : Reliability of the Rendezvous Path Testing Criterion for Ada Concurrent Programs, Trans. IEICE, Vol.J75-D-I, No.5, pp.288-297, 1992 (in Japanese).

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

[6] Itoh, E., Kawaguchi, Y., Furukawa, Z. and Ushijima, K. : Testing Criteria for Interprocess Communication in C Concurrent Programs. IPSJ SIG Notes, 93-SE-90, pp.9-16, 1993. (in Japanese)

[7] Kawaguchi, Y., Itoh, E., Furukawa, Z. and Ushijima, K. : On constructing testing reliability evaluation system with ordered sequence testing criteria, IPSJ SIG Notes, 96-SE-14, pp.107-114, 1994 (in Japanese).

[8] Rockkind, M. j. : *Advanced UNIX Programming*, Prentice Hall International, Inc. 1985.

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

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

[11] Weyuker, E. J. : *The Complexity of Data Flow Criteria for Test Data Selection*, Information Processing Letters, Vol.19, pp.103-109, 1984.

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