A Method for Structural Testing of Ada Concurrent Programs Using the Event Interactions Graph

Tetsuro Katayama
Graduate School of
Information Science,
Nara Institute of
Science and Technology.
8916-5 Takayama Ikoma,
Nara 630-01, Japan.
kat@is.aist-nara.ac.jp

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

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

Abstract

Software testing generally proceeds as follows: generating test-cases, selecting test-data, executing a test target program, inspecting execution result and evaluating whether testing has already been sufficient or not yet. As for methods for structural testing of programs, the way using a coverage, where the coverage means what extent given testing criteria are satisfied, is noted. At the evaluating step, whether or not we finish the testing is determined in view of the coverage.

This paper proposes a method for structural testing of concurrent programs written in Ada programming language, especially, test-case generation and execution of the programs. The Event InterActions Graph (EIAG) is used as a model for concurrent programs. The EIAG consists of Event Graphs and Interactions. An Event Graph is a control flow graph of a program unit in a concurrent program. The Interactions represent interactions between the program units. Program units are such as procedures, functions and task-types. After generating test-cases on the EIAG, a method for selecting test-data is described and measures to cope with infeasible test-cases with which are generated in this step is clarified. And a forced execution of a test target concurrent program in order to solve the nondeterministic execution is investigated. The nondeterministic execution is characteristic of concurrent programs.

Keywords: software testing, Ada programming language, concurrent programs, structural testing, test-cases, Event InterActions Graph (EIAG), testing criteria, nondeterministic execution.

1 Introduction

Software testing generally proceeds as follows: generating test-cases, selecting test-data, executing a test target program, inspecting execution result and evaluating whether testing has already been sufficient or not yet. For proving correctness of a program with testing, it must utilize all elements in the input domain of the program[1]. But it is not practical. In practical testing, therefore, we execute a test target program with test-data which are elements selected in a subset divided with conditions on the input domain. The conditions are called test-cases. In order to finish testing in finite testing period, we need criteria to detail the conditions. The criteria are called ‘testing criteria[2].’ As for a method for structural testing of programs, the way of using a coverage[3] - [5] is noted, where the coverage means “to what extent given testing criteria are satisfied.” At the evaluating step, in view of the coverage, we determine whether or not we have finished the testing. If we set up testing criteria clearly before we test a program, we can generate test-cases satisfying the testing criteria from its source code. And if we test the program with selected test-data based on the test-cases, we can judge that this testing is sufficient (this testing satisfies the testing criteria) and we finish the testing at the moment.

In recent years concurrent programs are frequently written and used[6]. Their reliability need to be improved because of practical usage of the concurrent programs.

In this paper, we propose a testing method for structural testing of concurrent programs written in Ada.
programming language, especially, test-case generation and execution of the programs. We generate test-cases, which satisfy testing criteria, from a source code. We use the Event InterActions Graph (EIAG) [7][8] as a model for concurrent programs. The EIAG describes program units as Event Graphs and Interactions between the program units. Program units are such as procedures, functions and task-types. We generate test-cases on the EIAG.

Then, we select test-data satisfying the test-cases. In this step we are faced with the problem of infeasible test-cases. We discuss measures to cope with the test-cases. Moreover, in executing a test target program, we are faced with the problem of non-deterministic execution; test-data does not guarantee to determine the execution order of statements in a program. By the non-deterministic execution, even if we execute a test target program with appropriate test-data, a test-case which includes a execution order of statements might not be executed. We consider a forced execution of target program in order to realize the test-case.

This paper is organized as follows. Section 2 presents the Event InterActions Graph (EIAG) as a model for concurrent programs, defines test-cases by using the EIAG and outlines testing criteria for testing of concurrent programs. [7][8]. Section 3 describes a method for selecting test-data and clarifies measures to cope with infeasible test-cases which we are faced in the step of selecting test-data. And then, we consider a forced execution of a target program in order to avoid the non-deterministic execution. Section 4 analyzes the feasibility of test-cases or test-data obtained by the method proposed in this paper and evaluates the capability of detecting errors, and then discusses the forced execution. Section 5 presents our conclusion.

2 Event InterActions Graph

In this section, we introduce the Event InterActions Graph (EIAG), define test-cases for concurrent programs using the EIAG and consider testing criteria which must be satisfied in testing for concurrent programs. [7][8]

2.1 Model for Concurrent Programs

The Event InterActions Graph (EIAG) describing behavior of a concurrent program. The EIAG consists of Event Graphs and Interactions between tasks.

2.1.1 Event Graphs in the EIAG

A concurrent program consists of tasks which communicate with each other. An Event Graph (EG) represents abstract control flows of a task or a program unit in a concurrent program. Because each program unit is regarded as being sequential, we can deduce a control flow graph from a source code. Nodes in the Event Graph denote concurrent event statements and flow-control statements which include the concurrent event statements. Concurrent event statements characterize concurrent behavior of a concurrent program. In an Ada concurrent program, concurrent event statements are such statements as entry calls, accept statements and new instance generation statements of a task-type. Edges in the Event Graph express transfer of control between nodes. That is:

$$EG = < N, E, s, f >,$$

where $N$ is a set of nodes in $EG$, and $E$ is a set of edges in $EG$. If $e = (u, v) \in E$, then $u, v \in N$. $s$ is the start node and $f$ is the final node.

A concurrent program has multiple program units; it has multiple Event Graphs. Here, we express a set of Event Graphs corresponding to a concurrent program $P$ as $EGs$.

$$EGs(P) = \{ EG_i = (N_i, E_i, s_i, f_i) | 1 \leq i \leq numProc(P) \},$$

where $numProc(P)$ denotes the number of processes (tasks) in $P$.

2.1.2 Interactions in the EIAG

When two tasks $T_A$ and $T_B$ synchronize, let Event Graphs $EG_A$ and $EG_B$ represent tasks $T_A$ and $T_B$, respectively. The Event Graph $EG_A$ has a node set $N_A$ and the $EG_B$ has a node set $N_B$. We define a set $Sync$, which satisfies the following expression, consisting of the pairs of elements of each set. A triplet $< a, b, X >$ in the $Sync$ represents a simultaneous execution with an identifier $X$ in a concurrent program.

$$Sync(EG_A, EG_B) = \{ Sync = < a, b, X > | a \in N_A, b \in N_B \},$$

where $< a, b, X >$ represents simultaneous execution with an identifier $X$.

In an Ada concurrent program, $Sync$ is constructed from both an entry call statement node and an accept statement node for the same entry, or from both a generation statement node of a new task-instance and a start node in the Event Graph of the task-type.
procedure sample is
  task T1;
  task T2;
  task T3 is
    entry E1;
    entry E2;
  end;
  task body T1 is
    begin
      T3,E1;
      end T1;
  task body T2 is
    begin
      loop
        T3,E2;
      end loop;
      end T2;
  task body T3 is
    begin
      loop
        select
          accept E1;
        or
          accept E2;
        end select;
      end loop;
      end T3;
    begin
      null;
      end sample;

Figure 1. A sample program written in Ada language.

We let $Syncs$ denote a set of all triplets of simultaneous executions in a concurrent program.

$Syncs(EGs) = \{ <a, b, X \mid \exists A, \exists B \langle a, b, X \rangle \in Sync(A, B) \land A, B \in EGs] \}$.

The Event Interactions Graph (EIAG) consists of Event graphs and Interactions\(^1\). The EIAG represents behavior of a concurrent program. That is:

$Interactions(P) = \{ Syncs(EGs) \}$,

$EIAG(P) = \{ EGs(P), Interactions(P) \}$.

Figure 1 shows a sample program written in Ada language. The numbers in front of statements are node numbers. The node number '0' denotes the start node, and the number '−1' denotes the final node. Figure 2 shows the EIAG of the program in Figure 1:

\(^1\)We defined Interactions are synchronizations, communications and waits in [8]. In this paper, we treat with only synchronizations as Interactions because concurrent programs written in Ada language are represented by only synchronization.

2.2 Test-cases for Concurrent Programs

In order to generate test-cases from an EIAG, we firstly consider test-cases on an Event Graph.

2.2.1 Test-Cases on an Event Graph

We define test-cases as Paths on an Event Graph in a similar manner for sequential programs. Firstly, we define Subpaths on an Event Graph as follows:

Definition 1: Subpaths is a set of sequences of the nodes on $EG = (N, E, s, f)$, and all pairs of side by side nodes in the sequences are elements of the edge set $E$:

$Subpaths(EG) = \{ \alpha | \alpha \in Seq(N) \land Arc(\alpha, EG) \}$,

$Arc(\alpha, EG) = \forall [1 \leq i < |\alpha| \rightarrow \alpha(i), \alpha(i + 1) \in E]$,

where $Seq(N)$ represents the sequence of nodes, $|\alpha|$ is length of the sequence $\alpha$, and $\alpha(i)$ is the $i$-th element of the sequence $\alpha$.

Definition 2: Paths is a set of Subpaths' elements whose first node is the start node $s$ and last node is the final node $f$:

$Paths(EG) = \{ \alpha | \alpha \in Subpaths(EG) \land \alpha(1) = s \land \alpha(|\alpha|) = f \}$. An element of Subpaths is called a subpath and an element of Paths is called a path.
2.2.2 Test Cases on an EIAG

By using test cases on Event Graphs and being based on the Interactions, we generate test cases on an EIAG. Firstly, we define Copath (cooperated path) between two Event Graphs.

**Definition 3**: Suppose that $A, B \in EGs$ and that $\alpha$ and $\beta$ are the elements of Paths($A$) and Paths($B$) respectively. Copath is a set of pairs $< \alpha, \beta >$, and if $< a, b, X >$ is an element of Sync($A, B$), the paths have property that the number of $a$'s is equal to the number of $b$'s, where $a$ is an element of $\alpha$ and $b$ is an element of $\beta$:

$$Copath(A, B) =$$

$$\{ < \alpha, \beta > | \alpha \in \text{Paths}(A) \land \beta \in \text{Paths}(B) \land \text{Suc}( \alpha, \beta, \text{Interactions}) \}$$

$$\text{Suc}( \alpha, \beta, \text{Interactions}) =$$

$$\forall < a, b, X > [ < a, b, X > \in \text{Interactions} \land [\text{Num}(\alpha, a) = \text{Num}(\beta, b)]],$$

where $\text{Num}(\alpha, a)$ represents the number of $a$'s in the sequence $\alpha$.

In a concurrent program, we define Copaths between any two Event Graphs if there are more than two Event Graphs as follows. If a concurrent program has $m$ tasks, Copaths consists of a set of $m$ paths.

$$\text{Copaths}(\text{EGs}) =$$

$$\{ < \alpha_1, \alpha_2, \ldots, \alpha_m > |$$

$$\forall i, j [1 \leq i, j \leq m, i \neq j] \land < a_i, a_j > \in \text{Copath}(\text{EG}_i, \text{EG}_j) \land \text{EG}_i, \text{EG}_j \in \text{EGs} \}.$$  

We can define that elements of Copaths denote test cases on an EIAG.

**Definition 4**: We define that elements of Copaths denote test cases on an Event InterActions Graph. That is:

$$\text{TestCases( EIAG )} = \text{Copaths( EGs )}$$

An element of Copaths is called a copath. Figure 3 shows a sample copath of the EIAG in Figure 2.

### 2.3 Testing Criteria for Concurrent Programs

In this section, we use the EIAG and Copaths as a model for concurrent programs and their test cases.

We consider testing criteria that testing methods for concurrent programs must satisfy. Testing criteria specify conditions for test case generation and for the completion of testing. In general, testing criteria show the lowest condition for testing. If we test a program based on a testing criterion, we may expect to detect a kind of error in the program. Testing criteria are factors characterizing testing methods.

Furukawa and Usijima discussed in [9] testing criteria for testing concurrent programs as follows:

1. **Edge Coverage Criterion** — All edges in a model are executed at least once in testing.

2. **Loop Coverage Criterion** — If a program has iteration, we consider two cases of zero and one repetitions in testing.

3. **Interaction Coverage Criterion** — All interactions between processes/tasks of a concurrent program
are executed at least once in testing.

The edge coverage criterion applied to the EIAG means that all edges of Event Graphs must be executed at least once. The loop coverage criterion is for the case that Event Graphs have iterations. These two criteria are applied to sequential programs. Furthermore, because of the loop coverage criterion, we can guarantee that loops as a special target are tested and that the paths are finite.

With regard to the interaction coverage criterion, we can select Interactions as a target of testing. That is, all synchronizations which exist in a program must be executed at least once in testing. We construct paths satisfying both criteria of the edge coverage and the loop coverage from Event Graphs and then combine them to satisfy the Interactions in order to get copaths.

3 Execution of Testing

After we generate test-cases, which satisfy the three testing criteria in the previous section, from a source code of a test target program, we select test-data satisfying each of the test-cases and then execute the program with the test-data. In this section, we describe two problems. One is feasibility of test-cases in selecting test-data. The other is the nondeterministic execution of concurrent programs in executing the programs.

3.1 Test-data Selection

We select test-data that can execute programs according to each of generated test-cases. The steps of selecting test-data are as follows:

Step 1 We get paths on each of the Event Graphs according to a generated test-case (copath).

Step 2 If any of the paths have branch statements, we pick up all expressions of conditions on the branch statements from the source code.

Step 3 We calculate test-data satisfying all of the expressions.

Step 3 forces us to solve simultaneous inequalities. These inequalities consist of expressions of conditions on the branch statements from the source code. Generally speaking, most of them may have domain of solution, but do not have only one solution. Hence, All programs can not be automatically selected test-data; we must manually select test-data. With regard to the step of generating test-cases, the generated test-cases may be infeasible. That is, it is possible that appropriate test-data to satisfy the generated test-cases do not exist because the simultaneous inequalities may not have domain of solution. This problem arises even in case of structural testing of sequential programs.

In case of no appropriate test-data for a test-case, feasibility of the test-case is not guaranteed. In this case, we must delete an infeasible copath, modify the paths and then generate copaths again.

As for satisfying test-cases, excepting the method for selecting test-data as mentioned above, we think of a method for testing a program with selecting test-data at random. We execute the program with various test-data until the test-cases which we intend to satisfy are satisfied. This method takes fewer times in the step of selecting test-data, but causes a problem of no guarantee that the test-cases which we intend to satisfy can be certainly satisfied. Also, it remains the problem that the method cannot determine whether or not the generated test-cases are feasible.

We may randomly select test-data from domains which are gotten by solving inequalities for feasible test-cases. Here, the test-data are input to a test target program. However, test-cases may not be satisfied. Because a concurrent program may have nondeterministic behavior, results of the program may be different for same input data. It is necessary to force the execution of the program along the test-cases. This forced execution is discussed next subsection.

After we execute a test target program with selected test-data, we compare outputs from the program with expected outputs which are specified in the specification of the program, and then we judge whether or not the outputs from the program is equal to what we intend. If the outputs are different from what we expect, we infer that some errors must exist in the program. When we execute the program with all selected test-data based on the test-cases, we can finish the testing.

3.2 Forced Execution

In testing of concurrent programs, we are faced with the problem of the nondeterministic execution that does not guarantee test-data to determine the execution order of statements in a test target program. By the nondeterministic execution, even if we choose an appropriate test-data which we intend to satisfy a test-case, we may not execute the program so that the test-case may be satisfied. That is, the execution order required by the test-case may not be followed.
The nondeterministic execution of concurrent programs written in Ada language has two cases as follows[10]:

(1) By entry call statements.
If many entry call statements call one accept statement in program execution, the rendezvous is realized between the accept statement and the earliest executed entry call statement. In every executing, hence, the order of the rendezvous may change by differences of each task execution time.

(2) By select statements.
If many accept statements which accept entry call statements exist in a program, an arbitrary accept statement is executed. The specification of Ada language does not provide which rendezvous is realized earlier.

Figure 4 and 5 show examples of nondeterministic execution. The program in Figure 4 has the entry call statement (a1) in task A and the entry call statement (c1) in task C. If (a1) is executed earlier than (c1), two rendezvous are executed in order of \(<(a1),(b1),E>\), \(<(c1),(b2),E>\). Otherwise, two rendezvous are executed in order of \(<(c1),(b1),E>\), \(<(a1),(c2),E>\).

The program in Figure 5 has the select statement (b1). When the statement is executed, we do not know which rendezvous \(<(a1),(b2),E1>\) or \(<(c1),(b3),E2>\) is executed. (But, if either the entry call statement (a1) or the (c1) is waited, the rendezvous of the waited entry call statement and the accept statement corresponded to it is executed.)

By the nondeterministic execution, a test target program may not be executed along generated copaths. Hence, we need to confirm that the program execution behaves as same as the execution order which is specified by the copaths. Therefore, we must be able to investigate which paths and interactions are executed. Otherwise, we must forcibly execute the program so that the program will satisfy the copaths; the program execution becomes deterministic by removing causes of the nondeterministic execution, and then we execute testing. We describe a method for the forced execution.

If only one entry call statement calls when an accept statement or a select statement is executed, the rendezvous between the entry call statement and the accept statement corresponding to the entry call statement is executed. Therefore, a rendezvous is deterministically executed if we prevent entry call statements from being executed until an entry call statement of

\begin{verbatim}
procedure example1 is
task A;
task B is	entry E;
end;
task C;
task body A is
  begin
    B.E;
  end A;
task body B is
  accept E;
  accept E;
end B;
task body C is
  begin
    B.E;
  end C;
begin
  null;
end example1;
\end{verbatim}

Figure 4. An example of nondeterministic execution(1)

the rendezvous which we intend to realize finishes the rendezvous.

In order to make deterministic the order of execution among entry call statements, we insert new rendezvous in the source code of a test target program as follows:

To execute rendezvous in order of \(r_1,r_2,\cdots,r_n\) corresponding to entry call statements \(e_1,e_2,\cdots,e_n\), respectively, where each of \(e_1,e_2,\cdots,e_n\) is different, we insert two nodes for one nondeterministic execution. One is inserted where is executed immediately after \(e_i\) and the other is inserted where it is executed immediately before \(e_{i+1}\) for \(i=1,2,\cdots,n-1\), and then one of the nodes synchronizes with another node.

In the program in Figure 4 to execute two rendezvous \(<(a1),(b1),E>\) and \(<(c1),(b2),E>\), we have only to insert two nodes. One is inserted where it is executed immediately after (c1) and the other is inserted...
procedure example2 is
  task A;
  task B is
    entry E1;
    entry E2;
    end;
  task C;
  task body A is
    begin
      B.E1;
      (a1)
      B.E;
      end A;
  task body B is
    begin
      loop
        select
          accept E1;
          (b1)
          or
          accept E2;
          (b2)
        end select;
        end loop;
        end B;
  task body C is
    begin
      B.E2;
      (c1)
      B.E;
      end C;
  begin
    null;
  end example2;

Figure 5. An example of nondeterministic execution(2)

procedure example1 is
  task A;
  task B is
    entry E;
    end;
  task C is
    entry inst_syn_1;
    ***
    end;
  task body A is
    begin
      B.E;
      (a1)
      C.inst_syn_1;
      (a2)***
      B.E;
      end A;
  task body B is
    begin
      accept E;
      (b1)
      accept E;
      (b2)
      end B;
  task body C is
    begin
      accept inst_syn_1;
      (c2)***
      B.E;
      (c1)
      end C;
  begin
    null;
  end example1;

Figure 6. An example of changing the program for forced execution(1)

that time each condition of the when condition is difference.

In the program in Figure 5 to execute two rendezvous \(<(a1),(b2),E1>\) and \(<(c1),(b3),E2>\), we have only to insert two nodes. One is inserted where is executed immediately before \(b2\) and the other is inserted where is executed immediately before \(b3\), and at that time each condition of the when condition is different. Figure 7 shows the source code of the program with inserted nodes. The mark \(***\) on the right side of the source code denotes inserted statements.

We determine the order of the rendezvous in program execution by inserting nodes as mentioned above, and then can execute the test target program so that the program will satisfy the copaths.
procedure example2 is
    task A;
task B is
    entry E1;
    entry E2;
end;
task body A is
begin
    B.E1;
end A;
task body B is
    acc_cont: integer;
begin
    loop
        get(acc_cont);
        select
            when acc_cont = 1 =>
                accept E1;
            when acc_cont = 2 =>
                accept E2;
        end select;
    end loop;
end B;
task body C is
begin
    B.E2;
end C;
begin
    null;
end example2;

Figure 7. An example of changing the program for forced execution(2)

4 Discussion

In this section, we discuss the reliability of the testing method using the EIAG and the forced execution.

4.1 Reliability of the Testing Method

Howden[11] defined the term reliable as follows. If a program satisfies a testing criterion $C_Ti$ and all errors in the program are detected, then $C_Ti$ is reliable for the program. However, the testing criterion which is reliable for any program is only exhaustive test[1] that utilizes all data in the input domain. Any practical testing criterion is only reliable for a program which is correct or includes some particular errors.

4.1.1 Reliability in Generating Copaths

Copaths may not satisfy the testing criteria (as described in Section 2.3). We point out three reasons as follows:

Figure 8. Two examples having an error in synchronization.

Figure 9. An EIAG containing two infeasible pairs.

1. The case that some errors exist in the order or the number of nodes, which are elements of $Sync$.

We consider the case that errors in synchronization[9] exist in the program. Figure 8 shows two examples of this case. Figure 8(a) shows that an error in the order of nodes of elements of $Sync$ exists, and (b) shows that an error in the number of those exists. We cannot execute correctly the rendezvous in both (a) and (b). In the case of (a), we can detect errors with only static analysis in generating copaths. In the other, we may detect errors in the step of generating test-cases because we generate copaths so that the number of synchronization statements in each copath can be equal.
2. The case that some elements of Sync are infeasible.
This case arises if accept statements or entry call statements of the same entry name exist. Because the pair of an entry call statement and an accept statement of the same entry name is regarded as an element of Sync in the EIG, this case arises even if the test target program has no errors. In this case, we delete such elements (pairs) and select test-data. Figure 9 shows an example of this case. Figure 9(a) shows an EIG containing two infeasible pairs of \( <a_1, b_2> \) and \( <a_2, b_1> \). We delete the pairs so that Figure 9(b) shows.

3. The case that a path which cannot satisfy Sync is made.
When we generate copaths, we generate paths satisfying both criteria of the edge coverage criterion and the loop coverage criterion from an Event Graph and then combine them satisfying Interactions (as described in Section 2.3). At the moment, we may have generated paths so that we cannot combine them satisfying Sync. In this case, we replace the paths and then generate copaths again. For example, in the EIG of Figure 10(a), when the paths \( \alpha_1, \alpha_2, \beta_1, \beta_2 \) of (b) are generated, we cannot generate copaths by combining the paths. We replace the paths \( \beta_1, \beta_2 \) of (b) with the paths \( \beta'_1, \beta'_2 \) of (c). Then we can generate copaths \( <\alpha_1, \beta'_1>, <\alpha_2, \beta'_2> \) and satisfy the testing criteria.

In summary, in case 1, because we can detect errors in a program, we modify the program and then generate copaths again. In case 2, we delete infeasible pairs as mentioned above and then generate copaths again. In case 3, we replace paths and generate copaths again as mentioned above. In all cases, after we finish generating copaths, we proceed to next step which is to select test-data.

4.1.2 Reliability in Executing Testing
Concerning errors in communication between tasks, we can roughly characterize two kinds, one is a complete communication error where we can detect errors to any test-data in communication, and the other is a partial communication error where we may detect errors to some test-data in communication[9].

Copaths are reliable for complete communication errors because all interactions of a tested program are executed at least once in testing by the interaction coverage criterion. Furthermore, copaths are reliable for deadlocks which occur without regard to variables and sequences of concurrent event statements in a program.

However, we do not know how reliable the testing method using the EIG is for practical concurrent programs. In future, we must evaluate the effectiveness of this testing method by applying it to various concurrent programs.

4.2 Forced Execution

In investigating the nondeterministic execution of Ada concurrent programs, Tai proposed a method of synchronization sequences (SYN-sequences)[12]. He reported the reproducible testing execution of concur-
rent programs with the SYN-sequences for debugging. In executing a target program, a programmer gets out a sequence of synchronization executed in the program and then executes the program with the sequence (SYN-sequence) as similar. Tai realized the method by inserting a control task into the target program. The method treats all synchronizing nodes in sequential order and lets the nodes execute along a particular order by demanding permission for a rendezvous of the control task just before executing the rendezvous.

We adopt the deterministic execution of a test target program so that copaths will be satisfied. We devise the method, for the deterministic execution, that inserts nodes to deterministically execute the order of entry call statements or accept statements. Because the method inserts nodes for only synchronization of the nondeterministic execution, the number of inserted nodes by our method is less than that by Tai’s method. However, if an entry call statement exists in a loop, the loop must be unfolded and the loop coverage criterion must ease to execute synchronization whose order change in each time of the loop.

After we insert nodes, an original program is transformed. That is, we test the rewritten program which is different from the original one. In the method as we mentioned, the order of statements in each task of the rewritten program is as same as one of the original. However, we need to be careful that the inserted nodes may change the execution time and affect real-time processes.

5 Conclusion

In this paper, we describe a method for structural testing of Ada concurrent programs. We set up testing criteria clearly before we execute a test target program for testing. And then we generate copaths as test-cases, satisfying the testing criteria, from the source code of the program, select test-data satisfying the test-cases and execute the program. If we execute the program with selected test-data based on the test-cases, we can judge that this testing is sufficient (this testing satisfies the testing criteria). Therefore, we can finish the testing at the moment.

We are faced with the problem of the nondeterministic execution in testing for concurrent programs, and then consider how to solve the problem. We showed that the execution can become deterministic by inserting synchronization points in the source code.

In future, we must evaluate the effectiveness of this testing method by applying it to various concurrent programs. And we need to set up more adequate testing criteria based on the result of the evaluation.

We have implemented the tool TCgen (Test-case generator) that automatically generates copaths, which are test-cases for a concurrent program, from concurrent programs written in Ada language[13]. We are planning to enhance the tool TCgen because TCgen cannot treat some programs including abort statements, task-types and so on. Moreover, we need to develop a tool for supporting automatic selection of test-data and insertion of nodes to avoid the nondeterministic execution.

References