Design and Implementation of Test-case Generation for Concurrent Programs

Tetsuro Katayama  
Graduate School of  
Information Science,  
Nara Institute of Science and Technology.  
8916-5 Takayama Ikoma,  
Nara 630-0101, Japan.  
akat@is.aist-nara.ac.jp

Zengo Furukawa  
Faculty of Engineering,  
Kagawa University.  
1-1 Sachi Takamatsu,  
Kagawa 760-8526, Japan.  
zengo@eng.kagawa-u.ac.jp

Kazuo Ushijima  
Graduate School of  
Information Science and Electrical Engineering,  
Kyushu University.  
Hakozaki 6-10-1, Fukuoka 812-8581, Japan.  
ushijima@csce.kyushu-u.ac.jp

Abstract

Test-cases play an important role for high quality of software testing. Inadequate test-cases may cause bugs remaining after testing. Overlapped ones lead to the increases in testing costs. This paper proposes the Event InterActions Graph (EIG) representing behavior of concurrent programs including any task-type and the cooperated paths (copaths) on the EIG as test-cases, and describes the test-case generation tool (T`Cgen) for concurrent programs written in Ada programming language. The EIG 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 such as synchronizations between the program units. T`Cgen generates test-cases as copaths from an Ada concurrent program. The generated copaths satisfy given testing criteria. They can find some communication errors in testing and detect unreachable statements which concern interactions. It is, however, necessary to validate feasibility of the generated copaths.

Keywords: software testing, concurrent programs, structural testing, test-cases, Event InterActions Graph (EIG), testing criteria, task-types.

1 Introduction

A process of generating test-cases should be systemized. Software testing is expensive; it accounts for approximately half of a software system development[6]. One of the problems involved in software testing is the process of generating test-cases. Test-cases play an important role in determining the quality of software. If the number of test-cases is not adequate, it is likely that bugs would appear in usage of programs after testing. If the test-cases overlap, testing costs increase.

Concurrent programs are frequently written and used in recent years[8]. It is necessary to improve their reliability. For sequential programs we have practical methods of generating test-cases, based on a source code or specification of a program. It is obvious that only using the methods for sequential programs is inadequate for evaluating reliability of concurrent programs. Testing criteria proposed for sequential programs do not care the two characteristics of concurrent programs. One is nondeterministic execution and the other is interactions such as synchronizations and communications between processes.

A concurrent program may include task-types. A task-type is formally defined in Ada programming language. In C programming language, a process can be generalized as a task-type. A task-type in concurrent programs is a template of task-instances dynamically generated in execution of the programs. Each of task-instances is executed sequentially, but task-instances may be able to communicate with each other.

This paper describes a test-case generation tool for concurrent programs including any task-type. In section 2, we introduce the Event InterActions Graph (EIG)[3] describing behavior of concurrent programs, and then define cooperated paths (copaths) as test-cases on the EIG. Moreover, we discuss testing criteria for concurrent programs and show algorithms generating copaths on the EIG. In section 3, we explain the test-cases generation tool T`Cgen for Ada[11] concurrent programs. In section 4 we discuss and evaluate the generated test-cases.

2 A Model for Concurrent Programs

In this section, firstly we introduce an Event InterActions Graph (EIG) describing behavior of a concurrent program[3], and then define cooperated paths (copaths) as test-cases on the EIG. Furthermore, we discuss test-
ing criteria for concurrent programs and show algorithms generating copaths on the EIAG.

2.1 Event InterActions Graph (EIAG)

The EIAG consists of Event Graphs and Interactions between processes (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. A control flow graph can be deduced from source code because each task or program unit is regarded as being sequential. 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 in a concurrent program. For example, in an Ada concurrent program, concurrent event statements are such statements as entries, accept statements and generation statements of a new task-instance of a task-type. Edges in the Event Graph express transfer of control between nodes.

$$EG \equiv (N, E, s, f),$$

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. We express a set of Event Graphs corresponding to a concurrent program $P$ as $EG_s$.

$$EG_s(P) \equiv \{ EG_i = (N_i, E_i, s, f_i) | 1 \leq i \leq numProc(P) \},$$

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

2.1.2 Interactions in the EIAG

When two tasks $TA$ and $TB$ synchronize, let two Event Graphs $EG_A$ and $EG_B$ represent tasks $TA$ and $TB$, 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 pairs of elements in each node set. A triplet $< a, b, X >$ in the $Sync$ represents a simultaneous execution with an identifier $X$ in a concurrent program.

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

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

Similarly, we define two sets $Comm$ and $Wait$.

$$Comm(EG_A, EG_B) \equiv \{ \text{comm} = < a, b, Y > | a \in N_A, b \in N_B \},$$

where $< a, b, Y >$ represents communication from $a$ to $b$ with an identifier $Y$.

$$Wait(EG_A, EG_B) \equiv \{ \text{wait} = < a, b, Z > | a \in N_A, b \in N_B \},$$

where $< a, b, Z >$ represents that there is possibility of $a$ waiting with an identifier $Z$. $Wait$ has two states for waiting and for not waiting.

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

$$Syncs(EG_s) \equiv \{ < a, b, X > | \exists A, \exists B, < a, b, X > \in Sync(A, B) \land A, B \in EG_s \}. $$

Similarly, we describe $Comm$s and $Wait$s:

$$Comm(EG_s) \equiv \{ < a, b, Y > | \exists A, \exists B, < a, b, Y > \in Comm(A, B) \land A, B \in EG_s \},$$

$$Wait(EG_s) \equiv \{ < a, b, Z > | \exists A, \exists B, < a, b, Z > \in Wait(A, B) \land A, B \in EG_s \}. $$

Synchronization between two statements means that one statement necessarily waits the other statement.

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

$$Interactions(P) \equiv \{ Syncs(EG_s), Comm(EG_s), Wait(EG_s) \},$$

$$EIAG(P) \equiv < EGs(P), Interactions(P) > .$$

Table 1 shows a part of a program to solve the producer-consumer problem written in Ada programming language. This program consists of three tasks: the task producer, the task consumer and the task buffer. The task producer generates one character and puts it in the buffer. The task consumer gets one character from the buffer and extinguishes it. The task buffer controls elements in the buffer.

Figure 1 shows the EIAG of the program. Table 1 shows correspondence of the program to nodes in the EIAG. Node numbers are not continuous because in the steps constructing Event Graphs we give a node number to each of the statements of the program and then remove the nodes which do not relate to the concurrent event statement. In Figure 1, the circles denote the nodes, the solid arrows denote the edges, the dashed arrows denote communications, which are elements of $Comm$s, and node 0's and node -1's are start and final nodes, respectively.

1These nodes are elements of $Syncs$ also. In order to simplify graph we omit those elements.
Table 1. A part of a program to solve the producer-consumer problem and correspondence of the program to nodes in the EiAG.

<table>
<thead>
<tr>
<th>task</th>
<th>node</th>
<th>statements of the program</th>
</tr>
</thead>
<tbody>
<tr>
<td>producer</td>
<td>0</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>buffer.put(x);</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>exit when x = ASCII.eot;</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>end loop;</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>end;</td>
</tr>
<tr>
<td>consumer</td>
<td>0</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>buffer.get(y);</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>exit when y = ASCII.eot;</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>end loop;</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>end;</td>
</tr>
<tr>
<td>buffer</td>
<td>0</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>select</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>accept put(z: in character) do</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>accept get(z: out character) do</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>terminate;</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>end select;</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>end loop;</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>end;</td>
</tr>
</tbody>
</table>

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 the 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.

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 $E$:

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

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

Paths is a subset of Subpaths' element whose first node is the start node $s$ and last node is the final node $f$:

$$Paths(EG) \equiv \{ \alpha | \alpha \in Subpaths(EG), s, f \}.$$ 

An element of Subpaths is called a subpath and an element of Paths is called a path.

2.2.2 Test-Cases on the EiAG

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

Suppose that $A, B \in EG$s 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 $\beta$:

$$Copath(A, B) = \{ < \alpha, \beta > | \alpha \in Path(A) \land \beta \in Path(B) \land \lnot \exists a \in \alpha, b \in \beta \}.$$ 

$$Suc(< \alpha, \beta >, Interactions) = \forall a, b, X \in \alpha \land \beta \in Interactions \rightarrow \sum_{\alpha} Num(\alpha, a) = \sum_{\beta} Num(\beta, b),$$

where $Num(\alpha, a)$ 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.

$Copaths(EGs) \equiv \{ < \alpha_1, \alpha_2, \cdots, \alpha_{|EGs|} > \mid \forall i, j [1 \leq i, j \leq |EGs|, i \neq j] \rightarrow < \alpha_i, \alpha_j > \in Copath(EG_i, EG_j) \land EG_i, EG_j \in EGs \}.$

We can define that elements of Copaths denote test-cases on an EiAG. That is:

$$TestCases(EiAG) \equiv Copaths(EGs).$$
2.3 Testing Criteria and Algorithms

In section 2.2.2, we defined Copaths on the EIAG as test-cases for concurrent programs. If a loop exists in an Event Graph on an EIAG, the number of paths in the Event Graph becomes infinite. Because of avoiding the infinity, the number of loop iterations should be restricted as a testing criterion, which specifies conditions for test-case generation and for completion of testing. This section deals with testing criteria for concurrent programs and describes algorithms satisfying the criteria.

2.3.1 Testing Criteria for Concurrent Programs

Testing criteria specify not only termination conditions of test-cases generation but also reliability of testing[1]. Many testing criteria have been proposed for sequential program testing, but a few testing criteria have been proposed for concurrent program testing[2][9][10]. We adopt a simple testing criterion for test-cases generation of concurrent programs, because of easy implementation and as a practical method. We generate test-cases for "execution of all interactions of a concurrent program at least once."

which we call Interaction Coverage Criterion[3]. Reliability of this criterion is discussed in section 4.1.

A concurrent program may include task-types. Another criterion for such a concurrent program is necessary. When a task-type is included in a concurrent program, it is possible to dynamically generate task-instances in execution of the program although there is only one original task-template in the program. If we wish to apply a model which was proposed for testing of concurrent programs to such concurrent programs including task-types, we need to rewrite a model over and over according as how many task-instances are generated because it must be determined before modeling the programs.

We can apply the EIAG to a program including task-types[5]. If task-instances generated from a task-type communicate with each other the graph is constructed as two tasks in view of the task-type. When task-instances generated from a task-type do not communicate with each other, it is sufficient to make one task. If we generated test-cases for all possible task-instances, the number of the test-cases would become large and the test-cases could not be practically executed. Restriction of the number of task-instances reduces the number of test-cases. For example, Figure 3 shows a program to solve the dining-philosopher problem[10].

At generation of paths of Event Graphs, we use two criteria; all edges in an Event graph are executed at least once in testing(Edge Coverage Criterion) and we consider two cases of zero and one repetitions for loops in an Event Graph (Loop Coverage Criterion).

In summary, we adopt three testing criteria for 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 iter-
tion, we consider two cases of zero and one repetitions in testing.

3. Interaction Coverage Criterion — All interactions of a concurrent program are executed at least once in testing.

2.3.2 Algorithms of generating the copaths

In this section, we describe two algorithms satisfying three criteria in section 2.3.1. One is the path generation algorithm for the Event Graphs, and the other is the copath generation algorithm.

The following algorithm can generate paths on an Event Graph.

§Path Generation Algorithm

Step1. Find one path from the start node to the final node by the depth-first search.

Step2. Find a subpath of which the first node (named fork node) and the last node (named join node) are on already found paths. If we cannot find such a subpath, this algorithm ends.

Step3. Replace the subpath from the fork node to join node on the paths with the subpath found in Step2. Hence, we get another path.

Step4. Go to Step2.

The following algorithm can generate copaths for a concurrent program which has two Event Graphs A, B.

§Copath Generation Algorithm

Step1. Select a path α from a set Path(A).

Step2. Count the number of nodes identified by a in the path α, where a is one component of an element < a, b, X > of Interactions(A, B).

Step3. Find a path β from a set Path(B), where the number of nodes identified by b in the path β are equal to the number counted in Step2. The pair of the path α and β is a copath. If we cannot find such a path β, go to Step5.

Step4. Go to Step1.

Step5. Make a new path β' by combining subpaths with the path β so that it can satisfy the condition in Step3. The pair of the path α and β' is a copath. If we cannot make such a new path β' and we select all paths of a set Path(A), this algorithm ends.

Step6. Go to Step1.

<table>
<thead>
<tr>
<th>philosopher(1)</th>
<th>fork(1)</th>
<th>fork(2)</th>
<th>philosopher(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>fork up</td>
<td>accept up</td>
<td>accept up</td>
<td>fork up</td>
</tr>
<tr>
<td>fork up</td>
<td>accept up</td>
<td>accept up</td>
<td>fork up</td>
</tr>
<tr>
<td>fork down</td>
<td>accept down</td>
<td>accept down</td>
<td>fork down</td>
</tr>
<tr>
<td>end loop</td>
<td>end loop</td>
<td>end loop</td>
<td>end loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>accept up</td>
<td>fork up</td>
<td>accept down</td>
<td>fork down</td>
</tr>
<tr>
<td>accept down</td>
<td>fork up</td>
<td>accept up</td>
<td>fork up</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Figure 4. A sample test-case.

In order to get copaths in case that more than two Event Graphs exist, this algorithm must be executed between any two Event graphs.

A concurrent program may include a task-type. If the number of task-instances generated from a task-type is statically determined (obviously declaration in source code), we generate a set of paths whose number is equal to the number of task-instances declared in the source code. For example, if the number of task-instances is determined as ‘2’ by the source code, we can get copaths after we extract a set of four paths on the modified EIG (two paths from the task ‘fork’ and two paths from the task ‘philosopher.’) Figure 4 shows a copath of the program.

If the number of task-instances is determined by input-data, we can realize copaths by selecting test-data which generate the same number of task-instances as the copaths declare. In the case of the program, a copath generated from the EIG (Figure 3) is as Figure 5.

3 Outline of the test-case generation tool

We have developed the test-case generation tool (T C gen) in order to implement the algorithms in section 2.3.2. T C gen generates copaths from the source code of a concurrent program written in Ada programming language automatically. T C gen consists of four parts as follows (see Figure 6).

5
1. making control flow graphs
   This part produces control flow graphs and Interactions from a source code of concurrent programs written in Ada programming language.

2. making Event Graphs
   The control flow graph has nodes of which each corresponds to all statements of the program. This part eliminates nodes which are not the concurrent event statements (See section 2.1), and then constructs Event Graphs.

3. making paths
   This part generates paths from Event graphs based on the Path Generation Algorithm.

4. making copaths
   This part generates copaths on EIAG from paths and Interactions based on the Copath Generation Algorithm.

We make the part of making control flow graphs by Lex and Yacc. And, we implement the other parts by C programming language.

Table 2 shows approximate execution times\(^2\) of TC\textit{gen} when we apply three programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>producer_consumer_pair</td>
<td>0.079</td>
</tr>
<tr>
<td>five_dining_philosophers</td>
<td>0.064</td>
</tr>
<tr>
<td>prime_number_sieve</td>
<td>0.276</td>
</tr>
</tbody>
</table>

In Table 2, the \textit{producer\_consumer\_pair} represents a program to solve the \textit{producer\_consumer} problem. The \textit{five\_dining\_philosophers} represents a program to solve the five dining philosophers problem. The \textit{prime\_number\_sieve} represents a program to calculate all prime numbers which are less than a given integer.

4 Discussion and Evaluation

4.1 Characteristic and Reliability of Test-cases

Howden[1] defined a term \textit{reliable} as follow. If a program satisfies a testing criterion \textit{Cti} and all errors in the program are detected, then \textit{Cti} is \textit{reliable} for the program.

The testing criterion which is reliable for any program is only \textit{exhaustive test}[12]. Any practical testing criterion is only reliable for a program which is correct or includes some particular errors.

Probably discovered errors on communications are classified into two types[3].

1. \textit{Complete communication error} — If a process always communicates error data with another process, then this communication error is \textit{complete}. Interpretation of the data is distinguished in sender and receiver processes respectively.

\(^2\)The approximate execution times in Table 2 are measured on DEC AlphaStation 200.
2. Partial communication error — If a process may communicate error data with another process, then this communication error is partial.

The Interaction Coverage Criterion is ‘reliable’ for the complete communication errors. If a concurrent program includes the complete communication errors, testing with test-cases which satisfy the Interaction Coverage Criterion always discovers the errors. On the other hand, the criterion is “partially reliable” for the partial communication error; the errors may be discovered corresponding to values of test-data.

Other types of errors, such as synchronization error, are not guaranteed to be discovered by the test-cases satisfying the Interaction Coverage Criterion. Of course, errors which occur in a process may be discovered according to the testing criteria for Event Graphs: deadlocks caused by a concurrent event statement and unreachable concurrent event statements.

In Figure 5, we can find a deadlock. Because two interactions in Figure 5 cross, it is recognized that this copath cannot be executed and that the program comes into the deadlock.

Tai reported the reproducible test execution of a concurrent program and the synchronization sequences for definition of behavior and faults of a concurrent program[9]. Concurrent executable concurrent events, however, are placed in one sequence since the synchronization sequences express behavior for programs by the total order. A copath on the EIG is a set of paths. An explosion in the number of test-cases for a concurrent program can be avoided because copaths don’t express behavior for programs in the total order.

4.2 Models for Concurrent Programs

We have used the EIG as a model of concurrent programs. The EIG can express various mechanisms for concurrency.

Taylor et al. proposed the concept of structural testing of concurrent programs[10]. They defined the concurrent state graph. The concurrent state graph consists of nodes that denote a combination of states of each task, and edges that denote transfers of states of each task. They also proposed testing criteria based on the coverage of nodes and/or edges in the graph. The graph cannot be, however, adapted to concurrent programs including task-types. The number of task-instances must be determined before the graph is constructed. The larger the number of states of each task, the larger the size of the graph. The graph is not always realistic as a model for a concurrent program. An EIG is fundamentally based on the source code of a concurrent program. It can cope with concurrent programs including task-types[5].

Furthermore, we can easily compare the EIG with the original program. If we select test-cases based on the concurrent state graph, it is difficult to understand the original program. Notomi and Murata proposed the hierarchical reachability graph of bounded Petri nets for concurrent software analysis[7]. Because this graph is constructed by combining and compressing flow graphs based on the source code, it is not simple to find correspondence between this graph and the original program.

4.3 Feasibility of Test-cases

We described how to automatically generate test-cases, without interpreting semantics of a program. Hence, there is no guarantee that the program can be executed in actual test-cases; the program may not be executed for some generated test-cases. In sequential programs this case may occur.

For example, Figure 7 is a copath for the example program. If we execute the program so that this copath can be satisfied, the program must execute the get statement in the task consumer and then execute the put statement in the task producer. There are no test-data for this
copath. In view of such copaths, we will verify the feasibility by forcing execution of the program \[4\]. The forcing execution of the program may solve the problem nondeterministic execution which is a characteristic of concurrent programs.

### 4.4 Restriction of \(TC_{gen}\)

\(TC_{gen}\) generates copaths from concurrent programs automatically in a few seconds (see Table 2). The current \(TC_{gen}\) cannot deal with a program including exceptions. Concurrent programming languages such as Ada have the function of exception handling. For example, numeric errors can be dealt with in a typical exception handler of Ada. We consider it better to generate edges from the exception handler to each assignment statement in the program. The EIA has no nodes that denote assignment statements. Even though the EIA has such nodes, it may not be practical because the number of edges greatly increases.

Also, there is an abort statement which kills another task in Ada programming language. The current \(TC_{gen}\) cannot deal with any program including the abort statement.

### 5 Conclusion

We introduced the Event InterActions Graph (EIA) to describe the test-case generation for concurrent programs. We defined the copath (cooperated path) on the EIA as a test-case for the programs which is a set of paths in program units. Furthermore, we developed a test-case generation tool (\(TC_{gen}\)) for Ada concurrent programs. \(TC_{gen}\) generates copaths from a concurrent program including any task-type. We expect that the number of insufficient or overlapping test-cases will decrease since copaths generated by algorithms presented in this paper are made up systematically. The generated copaths are reliable for detecting unreachable statements which are concurrent event statements, for the complete communication error, and for some deadlocks, which we discussed in section 4.1. The EIA is fundamentally based on the source code of a concurrent program, therefore, it can be found correspondence to the original program easily. Furthermore, an explosion in the number of test-cases for a concurrent program can be avoided because copaths don’t express behavior for programs in the total order.

Future issues are as follows:

- **Solving feasibility of test-cases.**

\(TC_{gen}\) automatically generates test-cases from an Ada program, without interpreting semantics of a tested program. Hence, the program may not be executed on some generated test-cases. There may be no test-data for this test-case. We will take measures to confirm the feasibility by forcing the program to be actually executed. The forcing execution of the program may solve the problem nondeterministic execution which is a characteristic of concurrent programs.

- **Applying \(TC_{gen}\) to various programs.**

We applied some programs written in Ada programming language. \(TC_{gen}\) need to be applied to more concurrent programs in order to show usefulness of itself.

- **Expanding \(TC_{gen}\).**

\(TC_{gen}\) cannot cope with a concurrent program that has an exception handler or an abort statement. And, the present \(TC_{gen}\) cannot be applied to all concurrent programs. We must expand \(TC_{gen}\) so that it can be applied to concurrent programs written in other programming languages such as C.

### References


