Test-case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences

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

Eisuke Itoh
Computer Center,
Kyushu University.
6-10-1 Hakozaki,
Fukuoka 812-8581, Japan.
itou@cc.kyushu-u.ac.jp

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

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

Abstract

Test-cases play an important roll for high quality of software testing. Inadequate test-cases may cause bugs remaining after testing. Overlapped test-cases lead to the increases in testing costs. This paper proposes a test-case generation method with the EIAG(Event InterActions Graph) and the ISTC*(Interaction Sequences Testing Criteria). The EIAG represents behavior of concurrent programs. It 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 (synchronizations, communications and waits) between the program units. The* ISTC *are proposed which are based on sequences of Interactions. The cooperated paths (copaths) on the EIAG as test-cases satisfying the* ISTC *are generated. The generated copaths can detect dead (unreachable) statements which concern interactions, and they can find some communication errors and deadlocks in testing. It is, however, necessary to validate feasibility of the generated copaths.*

Keywords: software testing, concurrent programs, structural testing, test-cases, Event InterActions Graph(EIAG), testing criteria, Interactions Sequences Testing Criteria(ISTC).

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[7]. 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 the 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 (synchronizations, communications and waits) between processes (tasks).

We proposed a test-case generation method for concurrent programs before[3][6]. This method, however, does not always obtain sufficient results in testing for concurrent programs because of our adopted tesitng criteria. A testing criterion specifies conditions for ter-

mination of testing. In testing of concurrent programs, we have to specify not only input data but also sequences of statements for execution as testing conditions of the concurrent programs.

The testing criteria OSC (Ordered Sequence Testing Criteria) are proposed for concurrent programs in [2], which are based on sequences of concurrency statements. An OSC_k requires execution of all sequences of length k of the statements at least once (k is a natural number). We propose new testing criteria ISTC(Interaction Sequences Testing Criteria) using a concept of the OSC.

This paper describes a test-case generation method for concurrent programs with the ISTC as new testing criteria. In section 2, firstly we introduce the *Event InterActions Graph* ($EIAG$) as a model describing behavior of a concurrent program, and cooperated paths (copaths) as test-cases on the EIAG[3], and then our adopted testing criteria for concurrent programs in [3][6] are described. In section 3, we express the OSC for concurrent programs and show test-case generation with the EIAG and the ISTC. In section 4 we discuss and evaluate our proposed method.

2 A Model for Concurrent Programs

In this section, we introduce an Event InterActions Graph (EIAG) as a model describing behavior of a concurrent program, and cooperated paths (copaths) as test-cases on the EIAG[3]. Furthermore, we describe our adopted testing criteria for concurrent programs in [3][6].

2.1 EIAG (Event InterActions Graph)

The EIAG consists of Event Graphs and Interactions between processes (tasks).

2.1.1 Event Graphs in the EIAG

The concurrent programs that we are interested in testing include those written in languages such as Ada, CSP, and C concurrent programs on UNIX system. A concurrent program consists of some program units (processes or tasks) which communicate with each other. An Event Graph(EG) represents abstract control flows of a process (task) or a program unit in a concurrent program. A control flow graph can be deduced from source code because each process 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 entry calls, 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 $\mathbf{P}$ as EGs.

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

where $numProc(\mathbf{P})$ denotes the number of processes in $\mathbf{P}$.

2.1.2 Interactions in the EIAG

When two processes T_A and T_B synchronize with each other, let Event Graphs EG_A and EG_B represent processes 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 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.

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

where (a, b, X) represents simultaneous execution of a and b with an identifier X.

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

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

where (a, b, Y) represents communication from a to b with an identifier Y.

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

where (a, b, Z) represents that there is possibility of node a waiting b 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.

$$Syncs(EGs) \equiv \{(a,b,X) | \exists A, \exists B \\ [(a,b,X) \in Sync(A,B) \wedge A, B \in EGs]\}.$$

Similarly, we describe $Comms$ and $Waits$:

$$Comms(EGs) \equiv \{(a,b,Y) | \exists A, \exists B \\ [(a,b,Y) \in Comm(A,B) \wedge A, B \in EGs]\},$$

$$Waits(EGs) \equiv \{(a,b,Z) | \exists A, \exists B \\ [(a,b,Z) \in Wait(A,B) \wedge A, B \in EGs]\}.$$

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:

$$Interactions(\mathbf{P}) \equiv \\ \{Syncs(EGs), Comms(EGs), Waits(EGs)\},$$

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

Table 1. A part of a program to solve the producer_consumer problem and correspondence of the program to nodes in the EIAG.

task	node	statements of the program
	0	**begin**
	1	**loop**
producer	3	buffer.put(x);
	4	**exit when** x = ascii.eot;
	5	**end loop**;
	−1	**end**;
	0	**begin**
	1	**loop**
consumer	2	buffer.get(y);
	4	**exit when** y = ascii.eot;
	5	**end loop**;
	−1	**end**;
	0	**begin**
	1	**loop**
	2	**select**
	4	**accept** put(z: **in** character) **do**
buffer	10	**accept** get(z: **out** character) **do**
	15	**terminate**;
	16	**end select**;
	17	**end loop**;
	−1	**end**;

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 $Comms$[1], and node 0's and node -1's are start and final nodes, respectively.

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 as follows:

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:

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

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

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

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

$$Paths(EG) \equiv \\ \{\alpha | \alpha \in Subpaths(EG) \wedge \alpha(1) = s \wedge \alpha(|\alpha|) = f\}.$$

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

2.2.2 Test-Cases on 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 $A, B \in EGs$ and that α and β are the elements of $Paths(A)$ and $Paths(B)$ respectively. *Copath* is a set of pairs (α, β), and if (a, b, X) is an

[1] These nodes are elements of *Syncs* also. In order to simplify graph we omit those elements.

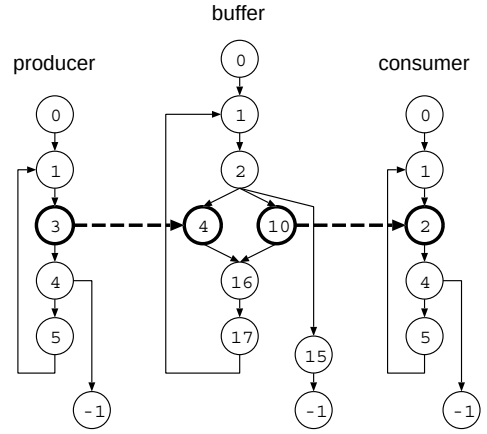

Figure 1. The EIAG of the program to solve the producer_consumer problem.

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 α and b is an element of β:

$$Copath(A, B) = \{(\alpha, \beta) |\ \alpha \in Path(A) \wedge \beta \in Path(B) \wedge Suc((\alpha, \beta), Interactions),$$

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

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

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 processes, $Copaths$ consists of a set of m paths.

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

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

$$TestCases(EIAG) \equiv Copaths(EGs)$$

An element of $Copaths$ is called a copath. Figure 2 shows a sample copath of Figure 1's EIAG.

2.3 Testing Criteria

Testing criteria specify not only termination conditions of test-cases generation but also reliability

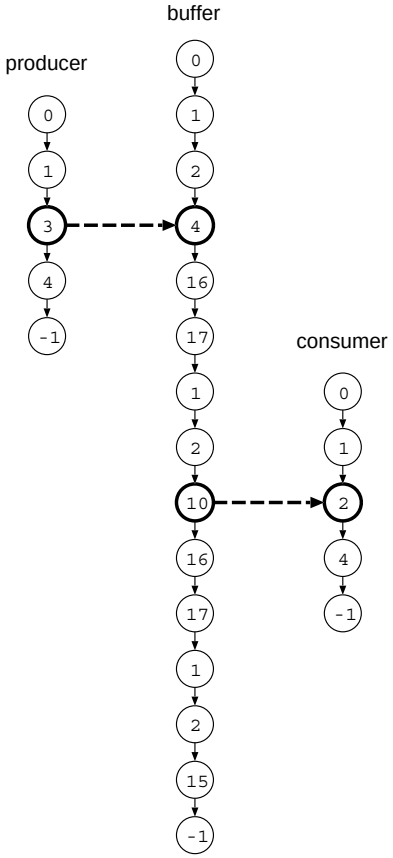

Figure 2. A sample copath.

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[9][10]. We adopted simple testing criteria for test-cases generation of concurrent programs as below, because of easy implementation and a practical method in [3][6].

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 of a concurrent program are executed at least once in testing.

We have developed the test-case generation tool($TCgen$)[6]. It generates copaths satisfied our adopted testing criteria as mentioned above from the

source code of a concurrent program written in Ada programming language automatically.

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

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

```
    P1                    P2
1: Send_data(aa);    2: Send_data(bb);

    P3
3: repeat
4: m:= Received_data;
5: until nodata;
```

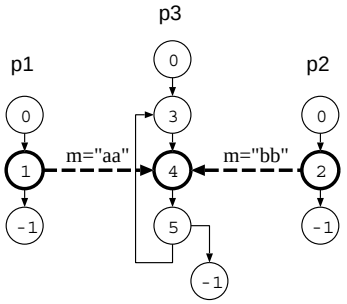

Figure 3. An example concurrent program.

On the program, the Interactions are the following:

$$(1, 4, m), (2, 4, m) \in Comms,$$

The execution result value of "m" is in accordance with the execution order of the elements in $Comms$. The possible execution orders are as follows:

$$Seq1: \ < (1,4,m),(2,4,m) >; \longrightarrow \ m = \text{“}bb\text{”},$$
$$Seq2: \ < (2,4,m),(1,4,m) >; \longrightarrow \ m = \text{“}aa\text{”}.$$

If the program is executed once; either $Seq1$ or $Seq2$ is executed, then the all-paths testing criterion (C_∞) [1] is covered and our adopted three testing criteria as mentioned above are satisfied. But it is not sufficient. It should be executed in a different execution sequence. We must reconsider the Interaction Coverage Criterion.

3 ISTC(Interaction Sequences Testing Criteria) and Copath

3.1 ISTC

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

The OSC (Ordered Sequence Testing Criteria) are proposed[2]. The OSC are concerned with the execution order of concurrency statements. In [2], the formal definition of OSC is as follows:

> OSC_k :
> Suppose that there is a concurrent program and S is a set of all concurrency statements in the source code. Construct ordered sequences of length k of the concurrency statements, where k is a natural number. An OSC_k requires execution of all the ordered sequences of length k at least once.
>
> A set of test-events of OSC_k ($k \geq 1$) is described as the following set,
>
> $$TE(\mathsf{OSC}_k) \equiv \{< s_1, s_2, \cdots s_k > | s_i \in S, 1 \leq i \leq k\},$$
>
> where the $TE(Cri)$ is a set of test-events of a testing criterion Cri.
>
> If $k = 2$, the OSC_2 is called the *ordered pair* testing criterion. A set of test-events of OSC_2 is described as the following set.
>
> $$TE(\mathsf{OSC}_2) \equiv \{< s_i, s_j > | s_i, s_j \in S\}.$$
>
> If all test-events of OSC_2 are executed, then all communications between any two processes are tested at least once.

A concept of the OSC is applied the EIAG; we propose new testing criteria ISTC(Interaction Sequences

Testing Criteria) which are based on sequences of Interactions.

$$TE(\mathsf{ISTC}_k) \equiv \{\omega | \omega \in Seq(Interactions) \wedge |\omega| = k\}$$

When $k = 1$, the ISTC_1 is equivalent to the Interaction Coverage Criterion.

If $k = 2$, the following are gotten.

$$TE(\mathsf{ISTC}_2) \equiv \{< (a,b,X),(c,d,Y) > | \forall (a,b,X),(c,d,Y) \in Interactions\}.$$

By adopting ISTC_2 as a testing criterion instead of the Interaction Coverage Criterion, both $Seq1$ and $Seq2$ on the example in the Figure 3 must be executed.

3.2 Algorithms of copath generation

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

Both algorithms are proposed in [3]. The path generation algorithm can generate paths on an Event Graph. It is not needed to modify because it is satisfying not the Interaction Coverage Criterion but the other criteria.

§**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 stops.

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.

This algorithm gets the first path and exchangeable subpaths by scanning edges once, and scans edge once more in order to search fork nodes and join nodes. Therefore, the order of the algorithm is $O(|E|+|E|) = O(|E|)$. We consider the worst condition: $|E| = |N|^2$. That is, the order of the algorithm is $O(|N|^2)$.

The copath generation algorithm can generate copaths for a concurrent program which has two Event Graphs A, B. It should be modified because it is satisfying the Interaction Coverage Criterion. We construct a new algorithm with ISTC_k.

§**Copath Generation Algorithm**

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

Step2. If $k \geq 2$ and two subpaths on the path α can be exchanged, the exchanged path is generated as a new path in advance.

Step3. Count the sum of the number of nodes corresponding to concurrent event statements in the path α. If the sum is equal to or more than k, go to Step4. Otherwise, make a new path α' by combining subpaths with the path α or replacing a subpath with a part of the path α so that the sum can be equal to k. Here, count each a in the path α and one component of an element (a,b,X) of $Interactions(A,B)$.

Step4. 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 each a. If we cannot find such a path β, go to step6. Otherwise, the pair of the path $\alpha(\alpha')$ and β is a copath.

Step5. Go to Step8.

Step6. Make a new path β' by combining subpaths with the path β or replacing a subpath with a part of the path β so that it can satisfy the condition in Step4. If we cannot make such a new path β', go to Step8.

Step7. The pair of the path $\alpha(\alpha')$ and β' is a copath.

Step8. When a new path in which the sum of number of nodes corresponding to concurrent event statements in the path α can be equal to k cannot be generated in the set $Path(A)$, it is replaced with the set $Path(B)$. If all paths are used, this algorithm stops. Otherwise, go to Step1.

The order of the copath generation algorithm is $O(|Path|^2)$ because it corresponds to combination of paths. We consider the worst condition: $O(|Path|) = O(|E|) = O(|N|^2)$. That is, the order of the algorithm is $O(|N|^4)$. In order to get copaths in case that more than two Event Graphs exist, this algorithm must be executed between any two Event graphs. Hence, the order of the algorithm is $O(|N|^{4m})$ if a concurrent program has m processes.

4 Discussion and Evaluation

In this section, we discuss the reliability of the copaths, models for concurrent programs and the feasibility of test-cases on an EIAG.

4.1 Characteristic and Reliability of the Copath

We qualitatively discuss reliability of concurrent programs tested with the generated test-cases. Howden[1] 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. 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.

Concerning errors in communication between processes, we can roughly characterize two kinds, one is a complete communication error which causes error states for all data can be detected in communication, and the other is a partial communication error where we may detect errors to some data in communication.

The ISTC are 'reliable' for the complete communication errors in a concurrent program. If a concurrent program includes the complete communication errors, testing with test-cases which satisfy the ISTC 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 deadlock and starvation, are not guaranteed to be discovered by the test-cases satisfying the ISTC. Of course, errors which occur in a process may be discovered according to the testing criteria for Event Graphs: unreachable (dead) concurrent event statements. And some deadlocks can be found[5][6].

The larger the value of k in ISTC_k, is, the more complex copaths to use in testing is. The number of test-events of ISTC_k is $|TE(\mathsf{ISTC}_k)|^k$. When $k = 1$, the ISTC_1 is equivalent to the Interaction Coverage Criterion; it is the same as the method proposed in [3][6]. It is practical but not accurate in cases of concurrent programs such as Figure 3. If $k \geq 2$, the cases can be solved.

There is a correlation between the number of test-cases and an accurate description of program behavior. An accurate description may increase the number of test-cases. In contrast, decrease in the number of test-cases may not well reflect program behavior. On the other hand, Increasing the number of test-cases may cause the feasibility problem of test-cases may cause (see Section 4.3). We should determine the value of k according to characteristics or behavior of the program and the time to take in testing.

4.2 Models for Concurrent Programs

We used the EIAG as a model of concurrent programs. The EIAG 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 process (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 a task-type which is a template of task-instances which are dynamically generated in execution of the programs. 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.

The EIAG is fundamentally based on the source code of a concurrent program. It can cope with concurrent programs including task-types[5][6].

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 test-cases actually generated based on the method. In sequential programs this case may occur.

For example, Figure 4 is a copath for the program to solve the producer_consumer problem. The copath must be executed to satisfy the ISTC_2. However, 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. The larger the value of k in ISTC_k is, the more the number of infeasible paths is. In view of such copaths, we will verify the feasibility by forcing execution of the program[4].

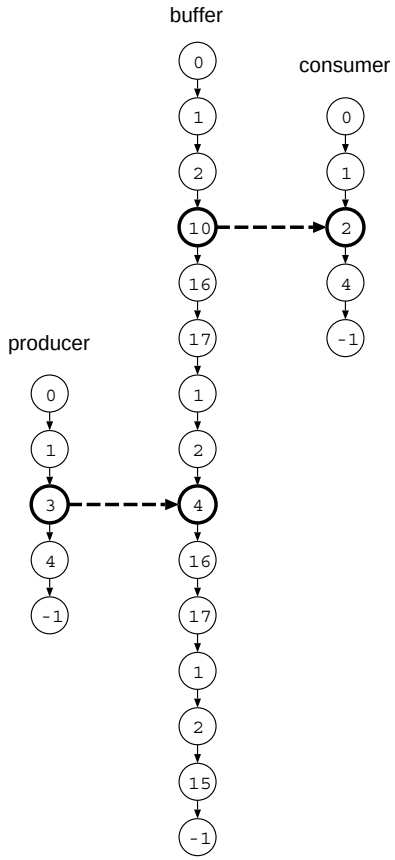

Figure 4. An infeasible copath.

5 Conclusion

We described a test-case generation method for concurrent programs with the ISTC (Interactions Sequences Testing Criteria) as new testing criteria. We introduced the EIAG (Event InterActions Graph) which represents behavior of concurrent programs and propose the ISTC based on sequences of Interactions. The cooperated paths (copaths) on the EIAG as test-cases satisfying the ISTC are generated. We expect that the number of insufficient or overlapping test-cases will decrease since copaths generated by the algorithms presented in this paper are made up systematically. The generated copaths are reliable for detecting unreachable (dead) concurrent event statements, for the complete communication error, and for some deadlocks. We should determine the value of k in ISTC_k according to characteristics or behavior of the program and the time to take in testing.

Future issues are as follows:

- Solving feasibility of test-cases.
  We described how to automatically generate test-cases, 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. In the future, 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[4].

- Expanding *TCgen*.
  We have implemented the tool TCgen (Test-case generator) that automatically generates copaths from concurrent programs written in Ada language[6]. We need to expand the tool TCgen so that it can automatically generate copaths satisfying the ISTC and that it can generate them from various concurrent programs.

References

[1] W. E. Howden: "Reliability of the Path Analysis Testing Strategy," *IEEE Trans. Softw. Eng.*, Vol.2, No.3, pp.208-215, 1976.

[2] E. Itoh, Z. Furukawa and K. Ushijima: "A Prototype of a Concurrent Behavior Monitoring Tool for Testing Concurrent Programs," *Proc. 1996 Asia-Pacific Softw. Eng. Conf. (APSEC'96)*, pp.345-354, 1996.

[3] T. Katayama, Z. Furukawa and K. Ushijima: "Event Interactions Graph for Test-case Generation of Concurrent Programs," *Proc. 1995 Asia-Pacific Softw. Eng. Conf. (APSEC'95)*, pp.29-37, 1995.

[4] T. Katayama, Z. Furukawa and K. Ushijima: "A Method for Structural Testing of Ada Concurrent Programs Using the Event Interactions Graph," *Proc. 1996 Asia-Pacific Softw. Eng. Conf. (APSEC'96)*, pp.355-364, 1996.

[5] T. Katayama, Z. Furukawa and K. Ushijima: "A Test-case Generation Method for Concurrent Programs Including Task-types," *Proc. Joint 1997 Asia-Pacific Softw. Eng. Conf. and Int'l Comp. Sci. Conf. (APSEC'97/ICSC'97)*, pp.485-494, 1997.

[6] T. Katayama, Z. Furukawa and K. Ushijima: "Design and Implementation of Test-case Generation for Concurrent Programs," *Proc. 1998 Asia-Pacific Softw. Eng. Conf. (APSEC'98)*, pp.262-269, 1998.

[7] G. Myers: "The art of software testing," John Wiley & Sons, 1979.

[8] C.R. Snow: "Concurrent Programming," Cambridge University Press, 1992.

[9] K.C. Tai: "On Testing Concurrent Programs," *Proc. Compsac'85,* pp.310-317, 1985.

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

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