Event-Constraint Model of a Concurrent Program
for Test-Case Generation

Tetsuro KATAYAMA, Zengo FURUKAWA, and Kazuo USHIJIMA

Department of Computer Science and Communication Engineering, Kyushu University.
Hakozaki 6-10-1, Fukuoka 812, JAPAN.
E-mail: katayama, zengo, ushijima@cse.kyushu-u.ac.jp

Abstract

Test-cases play an important role in determining the quality of software. For a sequential program we have practical methods of generating test-cases based on a source code and specification of a program. Few studies on test-cases for a concurrent program have been documented. Concurrent programs are often written to solve practical problems, therefore, the quality of testing for such programs must be upgraded. In this paper, firstly, in order to generate test-cases for a concurrent program we introduce an Event-Constraint Model (ECM) describing the behavior of a concurrent program. It is prepared based on concepts of an Event Graph and Constraints expressing a simultaneous execution. Secondly, we define cooperated paths (CoPaths) on the ECM as test-cases for a concurrent program. Finally, we give testing criteria for a concurrent program and describe algorithms used to generate test-cases on the ECM.

1 Introduction

Software testing is expensive; it accounts for approximately half of a software system development[1]. 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. Unless the number of test-cases is adequate, it is likely that bugs will appear. If the test-cases overlap, testing costs increase and many bugs in the program will remain. For a sequential program we have practical methods of generating test-cases, based on a source code and specification of a program (e.g. [7]). For a concurrent program few studies on test-cases have been reported[2][3][4]. It is necessary to improve testing quality for a concurrent program, since a concurrent program is written to solve a practical problem.

Taylor and Killy proposed the concept of structural testing of concurrent programs[2]. They defined the concurrent state graph as a model of a concurrent program. They also suggested testing criteria based on the coverage of nodes and/or edges in the concurrent state graph. Tai reported the reproducible test execution of a concurrent program and the synchronization sequences for definition of behavior and faults of a concurrent program[3]. Chang et al. developed the testing support system for a concurrent program of a communication system[4].

In section 2, to generate test-cases for a concurrent program we introduce an Event-Constraint Model (ECM) describing behavior of a concurrent program by using concepts of an Event Graph and Constraints. Section 3 defines test-cases by using this ECM. Section 4 states testing criteria for concurrent programs and shows algorithms generating test-cases on the ECM. In Section 5 the ECM is discussed. Figure 1 shows a sample program (Producer_Consumer Problem) written in Ada language. We use this program as an example throughout the paper.

2 Modeling a Concurrent Program

In this section, we introduce an Event-Constraint model (ECM) describing behavior of a concurrent program. The ECM consists of an Event Graph (EG) and Constraints.

2.1 Event Graph

A concurrent program consists of tasks which communicate with each other. EG represents abstract control flows of a task or a program unit in a concurrent program. Because each task or program unit is regarded as being sequential, we can deduce a control flow graph from the source code. Nodes in an EG denote concurrent event statements and flow-control statements which include concurrent event statements. A concurrent event statement is one which characterizes concurrent behavior in a concurrent program. For example, in an Ada concurrent program, an entry call, an accept statement, a generation statement of a new instance of a task type, etc are concurrent event statements. Edges in an EG express transfer of control between concurrent event statements.

$$EG = \langle N, E, s, f \rangle,$$

where $N$ is a set of nodes in an $EG$, and $E$ is a set of edges in an $EG$. If $e = (u, v) \in E$, then $u, v \in N$, where $u$ is an in_node, and $v$ is an out_node, in the edge $(u, v)$ respectively. $s$ is a start node and $f$ is a final node, $s$ and
with TEXT_IO; use TEXT_IO;
procedure Producer_Consumer is
  task Producer;
  task Consumer;
  task Buffer is
    entry P(x: in character);
    entry C(x: out character);
  end;

  task body Producer is
    x: character;
    begin
      1 loop
        get(x);
        2 Buffer.P(x);
        3 exit when x = ascii.eot;
      end loop;
    end Producer;

  task body Consumer is
    y: character;
    begin
      1 loop
        Buffer.C(y);
        2 exit when y = ascii.eot;
        put(y);
      end loop;
    end Consumer;

  task body Buffer is
    BufferSize: constant integer := 100;
    buffer: array(0..BufferSize - 1) of character;
    i,j: integer range 0..BufferSize -1 := 0;
    count: integer range 0..BufferSize := 0;
    begin
      1 loop
        when count < BufferSize ->
          3 accept P(x: in character) do
            buffer(i) := x;
            end P;
            i := (i + 1) mod BufferSize;
            count := count + 1;
          or
          when count > 0 ->
            4 accept C(x: out character) do
              z := buffer(j);
              end C;
              j := (j + 1) mod BufferSize;
              count := count - 1;
            or
            5 terminate;
        end select;
      end loop;
    end Buffer;
  begin
    null;
  end Producer_Consumer;

f satisfy the following expressions:

\[ s \in N \land \forall u [u \in N \rightarrow u, s \notin E] \]
\[ f \in N \land \forall u [u \in N \rightarrow f, u \notin E] \]

A concurrent program has multiple Event Graphs. Here, we express a set of Event Graphs corresponding to a concurrent program as \( EGs \).

\[ EGs = \{ \text{EG} | \text{EG} \rightarrow \langle N, E, s, f \rangle \} \]

2.2 Constraints

When two tasks \( T_x \) and \( T_y \) communicate with each other, an Event Graph \( EG_{T_x} \) has a node set \( N_{T_x} \) and \( EG_{T_y} \) has a node set \( N_{T_y} \). We define a set \( Con \), which satisfies the following expression, consisting of the pairs of elements of each set. A pair \( (\alpha, \beta) \) in \( Con \) represents a simultaneous execution in the concurrent program.

\[ Con(EG_{T_x}, EG_{T_y}) = \{ (\alpha, \beta) | \alpha \in N_{T_x}, \beta \in N_{T_y} \} \]

where \( \alpha, \beta \) have the relation of a simultaneous execution.

We let \( Constraints \) denote a set of the all pairs of simultaneous executions in a concurrent program.

\[ Constraints = \{ Con(X,Y) | \forall X, Y \exists [X,Y \in EGs] \} \]

For example, in an Ada concurrent program, a constraint is constructed of both an entry call statement node and an accept statement node for the same entry, or of both a generation statement node of a new task instance and a start node in the EG of the task type.

an Event-Constraint Model (ECM) consists of an Event graph and Constraints. The ECM represents behavior of a concurrent program.

That is:

\[ ECM = \langle EGs, Constraints \rangle \]

The following is the ECM of the program in Figure 1:

Producer
\[ N = \{0,1,2,3,4\} \]
\[ E = \{(0,1),(1,2),(2,3),(3,1),(3,4)\} \]
\[ s = 0 \]
\[ f = 4 \]

Buffer
\[ N = \{0,1,2,3,4,5,6\} \]
\[ E = \{(0,1),(1,2),(2,3),(3,1),(2,4),(4,1),(2,5),(5,6)\} \]
\[ s = 0 \]
\[ f = 6 \]

Consumer
\[ N = \{0,1,2,3,4\} \]
\[ E = \{(0,1),(1,2),(2,3),(3,1),(3,4)\} \]
\[ s = 0 \]
\[ f = 4 \]

Constraints
\[ \{ (2 \in N \in EG_{Producer}, 3 \in N \in EG_{Buffer}). \}
\[ \{ 2 \in N \in EG_{Consumer}, 4 \in N \in EG_{Buffer} \} \]

Figure 2 is a graphical form of the ECM of the sample program.
3 Method used to generate Test-Cases

To generate test-cases from an Event-Constraint Model, we firstly consider test-cases on an Event Graph.

3.1 Test-Cases on an Event Graph

We define test-cases on an Event Graph in a similar manner for sequential programs, and define that test-cases are paths on its graph. Firstly, we define a subpath on an Event Graph as following:

\[
\text{subpath} = \text{the sequence of the nodes on } \text{EG} \rightleftharpoons N, E, s, f, \text{ and all pairs of nodes, side by side, are elements of the edge set } E. \text{ The } \text{subpath} \text{ is of which the first node is the start node } s \text{ and the last node is the final node } f \text{ is called path.}
\]

The method to generate test-cases on an Event Graph is as follows.

To generate paths on an Event Graph, first, we get a path (named trunk) by linking edges from the start node to the final node. Second, we get a subpath of which the first node and the last node (are called a fork node and a confluence node, respectively) exist on the firstly generated path. We can get the next path by replacing the subpath from the fork node to the confluence node of the first generated trunk with the second generated subpath.

The test-cases on an Event Graph is as follows:

\[
\text{path(EG)} = \{ r | r \in \text{subpath(EG)} \land r(1) = s \land r(|r|) = f \},
\]

\[
\text{subpath(EG)} = \{ r | r \in \text{Seq}(N) \land \text{Arced}(r, \text{EG}) \}.
\]

\[
\text{Arced}(r, \text{EG}) = \forall i [1 \leq i \leq |r| \rightarrow < r(i), r(i + 1) > \in E],
\]

where \( \text{Seq}(N) \) represents the sequence of nodes, \(|r|\) is length of the sequence \(r\), and \(r(i)\) is the \(i\)-th element of the sequence \(r\).

The following is paths of the task Buffer in the program in Figure 1:

\[
\text{trunk} = (0,1,2,5,6)
\]

\[
\text{subpath} = (2,3,1,2,4,1,2,5,6).
\]

\[
\text{path} = (0,1,2,5,6), (0,1,2,3,1,2,5,6), (0,1,2,4,1,2,5,6), (0,1,2,3,1,2,5,6), (0,1,2,3,1,2,3,1,2,5,6).
\]

3.2 Test-Cases on an Event-Constraint Model

By using test-cases on Event Graphs we generate test-cases on an Event-Constraint Model, based on its Constraints. Firstly, we define a CoPath based on two Event Graphs \((X, Y)\). We assume that \(X \in \text{EGs}\) and that \(r_x\) and \(r_y\) are the elements of \text{path}(X) and \text{path}(Y) respectively.

\(\text{CoPath} = \) the pair of paths, when \((x_j, y_k)\) is an element of \text{Con}(X, Y), that have properties that \(x_j\)'s and \(y_k\)'s, elements of \(r_x\)'s and \(r_y\)'s respectively, are equal in number and in order of appearances.

When we generate a CoPath, we must select \(r_x\) from \text{path}(X), and \(r_y\) from \text{path}(Y) as \(x_j\) and \(y_k\) are equal in number. The CoPath is as follows:

\[
\text{CoPath}(X, Y) = \{ s | r_x, r_y > s \in \text{path}(X) \land r_y \in \text{path}(Y) \land \text{Suc}(s, \text{Con}(X, Y)) \}.
\]

\[
\text{Suc}(< r_x, r_y >, \text{Con}(X, Y)) = \forall x, y [x < y \in \text{Con}(X, Y) \rightarrow \text{Paired}(< r_x, r_y >, < x, y >)]
\]

\[
\text{Paired}(< r_x, r_y >, < x, y >) = \left[ \\text{Num}(r_x, x) = \text{Num}(r_y, y) \right] \land \forall i [1 \leq i \leq \text{Num}(r_x, x) \rightarrow r_x(j_i) = x] \land \text{NUM}(r_y, x) \land \text{Num}(r_y, y) \land \text{NUM}(< r_y(j_i), x >, < r_y(k_i), y >, < x, y >)
\]

where \(j_0 = k_0 - r(1) = s\), \(\text{NUM}(r, x)\) means that there is no \(x\) in the sequence \(r\), and \(\text{Num}(r, x)\) represents the number of \(x\)'s in the sequence \(r\).

In a concurrent program, we define a CoPath between any two Event Graphs when there are more than two Event Graphs.

That is:

\[
\text{CoPath}(\text{EGs, Con}(\text{EGs})) = \{ \text{CoPath}(\text{EG}_i, \text{EG}_j)| 1 \leq i, j \leq |\text{EGs}| \}
\]

When a concurrent program has \(m\) tasks, CoPath is a set of \(m\) paths. Even if a concurrent program has any task type, we can find Constraints between \(\text{EG}_i\) and \(\text{EG}_j\).
4 Testing Criteria and Algorithms

In section 3, we defined CoPaths on an ECM as test-cases for a concurrent program. In order to generate paths, we used a trunk path in an EG and an exchangeable subpath in the trunk path. If a loop exists in an EG, the number of the generated paths becomes infinite. Testing criterion specifies conditions for test-case generation and for completion of testing. This section deals with testing criteria for a concurrent program and describes algorithms to generate test-cases on an ECM to satisfy this criteria.

4.1 Two Testing Criteria

Testing criteria for a concurrent program are as follows:

1. Sequential Program Testing Criterion — testing criteria derived for sequential programs are applied to every task or program unit in a concurrent program.

2. Constraints Cover Criterion — all Constraints of a concurrent program are executed at least once.

The criterion 1 guarantees observation and test-case generation because each task or program unit is executed independently.

By setting this criterion, the test-cases have the following conditions when we generate test-cases on an Event Graph.

1. Cover Condition — all edges of an Event Graph are executed at least once.

2. Loop Condition — if a program has iteration, we consider two cases of zero and one repetitions.

By setting condition 2, generated paths may not become infinite. In order to generate CoPaths, firstly generate test-cases on EGs on the above conditions, and then combine them so that they satisfy Constraints. However, there may be no generation of CoPaths. In this case, let the generating CoPath take priority. That is: we generate such paths on an EG all over again as CoPaths are generated well.

Criterion 2 does not include which operations Constraints correspond to in a concurrent program. Constraints have to be defined in any language or by using a model. For example, we proposed the ‘rendezvous path’ [5] and the ‘global data flow’ [6] criteria, where we used Ada as the language and defined a remote procedure call and common variables, respectively, as constraints.

4.2 Algorithms

In this section, we show two algorithms in order to generate test-cases on an ECM. One is the path generation algorithm for an EG, and the other is the CoPath generation algorithm.

4.2.1 Path generation algorithm

This algorithm is roughly described as follows:

1. get edges from an EG, and link them in lists.
The path generation algorithm.

**input**: $E - A$ set of edges of an EG. ($s$ is the start node and $f$ is the final node.)

**output**: $p_i - A$ set of paths of the EG.

**variable**: $t_i -$ Lists that express subpath. (The list $t_0$ denotes trunk.)

$j -$ the number of lists.

**method**:

$t_0 \leftarrow c_1 - (s, \bullet)$. $j \leftarrow 0$, $E \leftarrow E - c_1$.

for $i \leftarrow 2, |E|$ {

$\ldots \{1\}$

$E \leftarrow E - c_i$.

Find $k$ and $l$ that satisfy the expression $x = t_k(l)$.

if $l \neq |t_k|$ {

$t_k \leftarrow t_k + y.$

} else {

$j \leftarrow j + 1$. $t_j \leftarrow (x, y)$. Mark ‘1’ in the node $t_k(l)$.

}. 

while $t_0([t_0]) \neq f$ do {

$\ldots \{2\}$

Find $k$ that satisfies the expression $t_k([t_k]) = f$.

Find $l$ and $i$ that satisfy the expression $t_k(1) = t_i(1)$.

Replace $t_k$ with $t_i[|t_i|]$.

}. 

for $i \leftarrow 1, j$ {

$\ldots \{3\}$

if $t_i(1) \notin t_0$ Find $k$ and $l$ that satisfy the expression $t_i(1) - t_k(l)(k < i)$. $t_i \leftarrow t_i(1 \sim l) + t_k.$

if $t_i([t_i]) \notin t_0$ Find $k$ and $l$ that satisfy the expression $t_i([t_i]) - t_k(l)(k < i)$. $t_i \leftarrow t_i + t_k(1 \sim |t_k|)$.

Find $l$ and $m$ that satisfy the expression $t_i(1) = t_j(l)$ and $t_j([t_j]) = t_0[m]$.

if $l > m$ {

$t_i \leftarrow t_i + t_j(0 \sim m - l)$.

}. 

$l \leftarrow 0$.

repeat 

$\ldots \{4\}$

$i, k, n \leftarrow 0$, $l \leftarrow l + 1$.

repeat

$i \leftarrow i + 1, v \leftarrow t_k(i)$. $p_i \leftarrow p_i + v$.

**case** The mark that node $v$ has.

$\ldots \{4.1\}$

**The mark is ‘1’**: */v is a fork node,*

$v$’s mark $\leftarrow ‘2’.$

**The mark is ‘2’*/Replace with a subpath.*

$\{ \text{for } q \leftarrow 1, j \{ \}$

Find $m$ that satisfies the expression $t_m(1) = v$.

if $t_m(i)$’s mark $\neq ‘3’$ { $k \leftarrow m$, $i \leftarrow 1$, $n \leftarrow 1.$ }.

if $n = 0$ { $v$’s mark $\leftarrow ‘0’$. }.

if $v = t_k([t_k])$ { $v$’s mark $\leftarrow ‘3’$. Find $r$ and $s$ that satisfy the expression $v = t_r(s)$. $i \leftarrow r$, $k \leftarrow s$.

$\ldots \{4.3\}$

until $v = f$.

if $3 \in t_q([t_q])$’s mark $= ‘3’$ and $(t_q(1), t_q(2), \ldots, t_q([t_q]) = 1)$’s mark $= ‘0’$ { $t_q(1)$’s mark $\leftarrow ‘3’.$ }.

until $|t_q([t_q])|$’s mark $= ‘3’$.

Figure 4: The path generation algorithm. The mark is for characterized nodes. (0... a normal node/1...a (not replaced) fork node/2...for replacing with a subpath/3...for used subpaths)

2. get trunk.

3. supplement edges for subpaths.

4. get nodes from lists, and make paths.

(a) make paths.

(b) for used subpaths.

(c) for used fork nodes.

Details are given in Figure 4.

For example, we give an EG of the task Buffer of the program in Figure 1 as the input of this algorithm. Block 1 gets edges from an EG in order, links in lists them and marks a fork node. If block 1 ends,

$$
\begin{align*}
& t_0 = (0,1,2,3,1) \\
& t_1 = (2,4,1) \\
& t_2 = (2,5,6)
\end{align*}
$$

the node 2’s mark ‘1’.

Block 2 replaces the lists, and makes up a trunk. If block 2 ends,

$$
\begin{align*}
& t_0 = (0,1,2,5,6) \\
& t_1 = (2,4,1) \\
& t_2 = (2,3,1)
\end{align*}
$$

the node 2’s mark ‘1’.

Block 3 supplements edges for subpaths such as their first node and last node exist in the trunk, and as the first node is placed before the last node in the trunk. If block 3 ends,

$$
\begin{align*}
& t_0 = (0,1,2,5,6) \\
& t_1 = (2,4,1,2) \\
& t_2 = (2,3,1,2)
\end{align*}
$$

the node 2’s mark ‘1’.

Block 4 gets nodes from lists, and makes paths. Firstly, recognize the trunk($t_0$) as a path. Secondly, get the next
paths by replacing the subpath from the fork node to the
confluence node of the trunk with other subpaths. This
algorithm ends when all subpaths are used:

\[
4 \begin{cases}
\text{path} = (0,1,2,5,6), \\
(0,1,2,4,5,6), \\
(0,1,2,3,5,6).
\end{cases}
\]

Paths generated by this algorithm automatically satisfy
’Semantic Program Testing Criterion’ described in the
previous section.

4.2.2 The CoPath generation algorithm

This algorithm is roughly described as follows:

1. preparation
2. select one EG.
3. select paths from EGs and make CoPaths.
   (a) select such paths from two EGs that satisfy \(cn \notin \text{Constraints}\),
   (b) make a new path so that it satisfies \(cn\) with replacement,
   (c) cope with the new path,
   (d) make CoPaths.

Details are given in Figure 5.

For example, the input of this algorithm as paths of the
program in Figure 1 is as follows:

\[
\begin{aligned}
\text{input}&: \quad \\
p_{1,1} &= (0,1,2,3,4) \\
p_{1,2} &= (0,1,2,3,1,2,3,4) \\
p_{2,1} &= (0,1,2,5,6) \\
p_{2,2} &= (0,1,2,4,1,2,5,6) \\
p_{2,3} &= (0,1,2,3,1,2,5,6) \\
p_{3,1} &= (0,1,2,3,4) \\
p_{3,2} &= (0,1,2,3,1,2,3,4) \\
cn_1 &= \{2 \in p_{1,3} \notin p_{2,1}\} \\
cn_2 &= \{2 \in p_{1,4} \notin p_{2,1}\}.
\end{aligned}
\]

\[
\begin{aligned}
p_1 &= \text{Producer} \\
p_2 &= \text{Buffer} \\
p_3 &= \text{Consumer}
\end{aligned}
\]

Block 1 is preparation. Block 2 selects one EG. Now, con-
sider \(p_1\). Block 3 selects paths that satisfy Constraints,
and makes CoPath. Firstly, select \(p_{1,1}\) that has node 2
\((cn_1)\). Secondly, select path, in the other EG, that has
the other node of the pair \(cn_1\). That is node 3 in \(p_2\), select \(p_{2,3}\)
that has the node in \(p_2\). Next, consider \(p_{1,2}\) that has node 2
\((cn_2)\). The other node of the pair \(cn_2\) exists in \(p_2\) (node 4).
The previously selected \(p_{2,3}\) in \(p_2\) does not have node 4.
Here, consider replacement with subpath used in the
path generation algorithm. With replacement, make up a
new path that satisfies Constraints. That is: replace \(p_{2,3}\)
with subpaths, and make up a new path that has node 4.
This implies that loops are often executed in the former
program.

As above, we get \(\text{Cop}_1\):

\[
\text{output} \quad \begin{cases}
p_1 &= (0,1,2,3,4) \\
p_2 &= (0,1,2,3,1,2,3,4,1,2,5,6) \\
p_3 &= (0,1,2,3,4)
\end{cases}
\]

This algorithm basically ends when all paths (inputs)
are selected. However, if replacement is made, then there
may be paths that do not satisfy Constraints, thereby im-
plying a dead-lock. For example, consider \(p_{2,1}\). With this
path no CoPath can be generated.

5 Discussion

In this section, the order of the algorithms and usefulness
and limitations of an ECM are given attention.

We firstly consider the order of the path generation algo-
rithm. Block 1 is \(\Sigma[\text{E}]\). Block 2 is \([\text{sub}]\) that denotes
‘the number of the subpaths’ \(\times [\text{sub}_2]\) that denotes ‘the
number of nodes in all subpaths’. Block 3 is \([\text{sub}] \times [\text{sub}_2]\).
Block 4 is \([\text{sub}] \times \text{[sub}_2\times[\text{sub}_2] + [\text{sub}] + [\text{sub}]\)’ (Three
terms in the parentheses correspond to block 4a, 4b, 4c
in order). We consider the worst condition: \([\text{sub}] - \text{[E]}\)
and \([\text{sub}_2] - \text{[E]}\). That is, the order of the path gen-
eration algorithm is \(O([\text{E}]^3)\).

Concerning the order of the CoPath generation algo-
rithm, block 1 is \([p] \times \{p\}_i\) ‘the number of calling
mark process’ \(\{p\}_i\) denotes the number of EGs, and \([p]_i\)
denotes the number of paths in the \(i\)-th EG). Block 2 is \([p]_i \times \text{‘the number of calling mark process’}\). Block 3 is
\([cn] \times ([p] \times [p]_i + [\text{sub}_2] - \text{[E]} + [p] \times [p]_i + \text{‘the
number of calling mark process’}]\) (Four terms in the paren-
theses correspond to block 3a, 3b, 3c, 3d in order).
We consider the worst condition: \([p]_i - \text{[E]}\), ‘the number
of calling mark process’ \(\{p\}_i\) and ‘the number of calling
mark process’ \([p]_i\). That is the order of the CoPath
generation algorithm is \(O([p]_i^2 \times [\text{E}]^3 \times [cn])\).

Taylor and Killy proposed the concept of structural test-
ing of concurrent programs[2]. They defined the concurre-
ent state graph as a model of a concurrent program. 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 suggested test-
ing criteria based on the coverage of nodes and edges in
the concurrent state graph. However, the number of tasks
must be defined before one can generate the concurrent
state graph. The larger the number of states of each task,
the larger the size of the concurrent state graph. There-
fore, the concurrent state graph is not always realistic in
a model for a concurrent program. Further, if we use testing
criteria based on information in an executed program, it
can be difficult to understand the program. An ECM is
fundamentally based on the source code of a concurrent
program. Therefore, one can easily compare it with the
former program and can cope with a concurrent program
that has the task type.

Tai reported the reproducible test execution of a concur-
rent program and the synchronization sequences for defi-
nition of behavior and faults of a concurrent program[3].
However, concurrently executable concurrent events are
placed in one sequence since the synchronization sequences
The CoPath generation algorithm.

**input**: $P_i(j)$ - the $j$-th (sub)path in the $i$-th EG.

$|P|$ denotes the number of EGs, and $|P_i|$ denotes the number of paths in the $i$-th EG.

$cn_i$ - the $i$-th element of Constraints.

**output**: CoPaths - A set of CoPaths.

**variable**: stack - the value of all mrk_pa kept for a time

use_pa - for a path used for test-cases,(1..used path/0..unused path)

mrk_pa - for a path satisfying Constraints,(1..satisfied path/0..unsatisfied path)

mrk_pa1 - for $X \in$ EG.(1..mark_process(X) is executed/0..not)

mrk_pa2 - for $Y \in$ EG.(1..investigated whether $Y$ satisfies Constraints(X,Y)/0..not)

$x$ - for a path having $cn_i$(1..had the node $cn_i$/0..not)

$z$ - for paths satisfying Constraints,(1..all satisfied paths exist/0..not)

$z$ - for exchangeable paths,(1..exchangeable paths exist/0..not)

**method**:

main {
    $a \leftarrow 0$. ... (1)
}

for $i = 1,|P|$
    $\forall p_1$’s mrk_pa1 $\leftarrow 0.$ $\forall p_2$’s mrk_pa $\leftarrow 1.$

for $j = 1,|P_i|$
    if $p_j$’s use_pa $= 1$ 
        $p_j$’s mrk_pa $\leftarrow 0.$

mark_process($p_i$); 

sub mark_process($p_i$) 

    $p_i$’s mrk_pa1 $\leftarrow 1.$ Push all mrk_pa’s in the stack.

for $j = 1,|P_i|$
    if $p_j$’s mrk_pa $\leftarrow 0.$ $\forall p_2$’s mrk_pa $\leftarrow 0.$ $p_j$’s mrk_pa $\leftarrow 1.$ mark_path($p_i$,$p_j$); 

Pop all mrk_pa from the stack, $p_i$’s mrk_pa1 $\leftarrow 0.$

return }

sub mark_path($p_i$, $p_j$) 

    $x$, $y$, $z$ $\leftarrow 0.$

for $k = 1,|cn_i|$
    if $p_j$ has a node in $cn_i$
        ... (3a)

Find $l$ that satisfies the condition: $p_l$ has the other node of the pair $cn_i$.

$p_l$’s mrk_pa1 $\leftarrow 0$.

for $m = 1,|P_i|$
    if The number of a node in $cn_k$ in $p_l$ is not equal to one in $p_m$.
        $p_m$’s mrk_pa $\leftarrow 0.$
    else 
        $\forall p_2$’s mrk_pa $\leftarrow 1.$ $y \leftarrow 1.$

if $y$ $= 0,$ and $p_l$’s mrk_pa1 $\leftarrow 0$ 

... (3b)

if $\exists l,m$ that satisfy the expression; ‘the number of a node in $cn_k$ in $p_l$’ $\neq$ ‘in $p_m$’ $+$ ‘in $P_{x8xh}$’

    if $\exists p_{|x|+1}$, a new path $p_{|x|+1}$ that satisfies the above expression 
        $p_{|x|+1}$ $\leftarrow$ the new path. $p_{|x|}$’s mrk_pa $\leftarrow 1.$ $y \leftarrow 1.$ $z \leftarrow 1.$

if $z = 1.$

... (3c)

for $q = 1,|P|$
    for $r = 1,|P|
        if $p_y$’s mrk_pa1 $\leftarrow 1,$ and $p_r$’s mrk_pa $\leftarrow 1$
            $y \leftarrow 0.$ $\forall p_2$’s mrk_pa $\leftarrow 0.$

if $\exists l,k,x$ such that $(<p_k,p_y>'$ $\cap$ $\forall cn_k \in$ Constraints) 

    $p_k$’s mrk_pa $\leftarrow 1.$ $y \leftarrow 1.$

if $y = 0.$ $q \leftarrow |P|.$

... (3d)

if $x = 1$ or $y = 0$

    if $\forall p_2$’s mrk_pa1 $\leftarrow 1$

    if $z = 0,$ and at least one use_pa $= 0$ in $\forall p_2$’s mrk_pa $\leftarrow 1$

        CoPath $\leftarrow$ CoPath + $\forall p_2$’s mrk_pa $\leftarrow 1.$ $a \leftarrow a + 1.$ $\forall (p_2$’s mrk_pa $\leftarrow 1)$’s use_pa $\leftarrow 1.$

else 
    if $\exists l,k,x$ such that $(<p_k,p_y>'$ $\cap$ $\forall cn_k \in$ Constraints) 

else 
    if $\exists l,k,x$ such that $(<p_k,p_y>'$ $\cap$ $\forall cn_k \in$ Constraints) 

if $\exists l,p_y$ is a new path in subroutine ‘mark_path’

    All $p_y$’s are deleted.

return }

Figure 5: The CoPath generation algorithm.
expresses the total order. Because a CoPath on an ECM is a set of paths and satisfies constraints in an ECM, it preserves the partial order of nodes in each EG and corresponds to multiple paths of an ECM, which represent the total order of all nodes in the ECM. By a CoPath, the partial order itself, of statement executions in a concurrent program, is preserved. Therefore, an explosion in the number of test-cases for a concurrent program can be avoided using a CoPath. However, it is necessary for synchronization execution of constraints.

Now, we consider the feasibility of generated test-cases on an ECM. We described how to generate test-cases mechanically, hence, there is no guarantee that the program can be executed in actual test-cases. For example, the following is a CoPath in the sample program:

```plaintext
\[
\begin{align*}
\text{Producer} &= (0,1,2,3,4) \\
\text{Buffer} &= (0,1,2,4,1,2,3,1,2,5,6) \\
\text{Consumer} &= (0,1,2,3,4)
\end{align*}
\]
```

If, as this CoPath expresses, we execute the sample program, the program must execute Consumer and then execute Producer. That is: there are no test-data for this test-case.

Concurrent programming languages such as Ada have the function of exception handling. The current ECM cannot deal with a program including exceptions. Numeric errors can be dealt with in a typical exception handler of Ada. We consider it better to generate edges whose in-node is each assignment statement in the program and whose out-node is the exception handler. However, an ECM has no nodes that denote assignment statements, and even though ECM has nodes, the number of edges greatly increases. There is an abort statement killed an other task in Ada. The current ECM cannot deal with any program including abort statement.

6 Conclusion

We introduced an Event-Constraint Model to describe the principle of test-case generation for a concurrent program. Based on the concepts of an EG and Constraints, we set up a concurrent program with an ECM to generate test-cases. We expect that insufficient or overlapping test-cases will decrease since the test-cases generated by algorithms presented in this paper are made up logically and systematically. We investigated the order of the algorithms and the usefulness of the ECM. The ECM is fundamentally based on the source code of a concurrent program, therefore, it can be compared with the former program. Further, because a CoPath on an ECM is a set of paths and satisfies constraints in an ECM, it preserves partial order of nodes in each EG and corresponds to the multiple paths of an ECM which represent the total order of all nodes in the ECM.

Future problems:

- Automatic generation of test-cases.
- Confirmation that the test-cases generated using the introduced method can actually be executed.
- Expansion of the ECM to cope with a concurrent program that has exception handler.

In the present paper, we described how to generate test-cases mechanically, hence, there is no guarantee whether the program can actually be executed in test-cases. That is: there may be no test-data for this test-case. This problem must urgently be resolved.

Acknowledgment

We thank T. Komoda for assistance, Y. Miura, H. Itoh, T. Nishi and M. Ohara for encouragement and comments. This work was supported in part by the Hara Foundation for Intellectual Systems, in Japan.

References