ソフトウェア工学 86－2
（1992．7．20）

並行処理プログラムのためのテストケース生成系の試作

片山徹郎　菰田敏行　古川善吾　牛島和夫
九州大学 工学部 情報工学科

テストケースは、ソフトウェアの信頼性向上に重要な役割を果たす。テストケース作成技法については、逐次処理プログラムの場合、様々な方法が実用化されている。しかしながら、並行処理プログラムの場合、テストケースの考え方すらほとんど研究されていない。並行処理プログラムが実用化されるようになり、並行処理プログラムのテストの質を向上させることが重要になっている。本稿では、並行処理プログラムのテストケースの定義、およびその生成系（*tcgen*）の試作と利用経験について述べる。事象制約モデル（*ECM*）上の協調路（*Copath*）を、並行処理プログラムのテストケースと定義する。*tcgen* は、系統的にテストケースを生成するので、テストケースの漏れや重複を少なくできる。

Prototype of Test-Case Generation System for Concurrent Programs

Tetsuro Katayama, Toshiyuki Komoda, Zengo Furukawa
and Kazuo Ushijima

Department of Computer Science and Communication Engineering,

Kyushu University.

6-10-1 Hakozaki, Fukuoka 812, Japan.

Test-cases play an important role in improving the reliability of software. For sequential programs, we have practical methods for generating test-cases. Few studies on test-cases for concurrent programs have been performed. Recently, concurrent programs are used for solving practical problems, therefore, it is necessary to improve the quality of testing for concurrent programs. In this paper, we define the test-case for concurrent programs. And we describe a test-case generation system (*tcgen*) and report experience in using it. We define that test-cases of concurrent programs are *Copaths* on an Event-Constraint Model (*ECM*). The *tcgen* makes sufficient test-cases there are no overlapping.

1. はじめに

テストの目的は、プログラム内の誤りを見つけることと、プログラムの正しさに対する確信を増すことである。一般に、テストでは、実行した入力データ(テストデータ)に対してのみプログラムの正しさを保障できるので、プログラムの品質は、どのようなテストデータに基づいてテストしたかに依存する。そこで、テストデータを選定する時には、テストデータについての条件を記述したテストケースの作成が重要になる。テストケースの作成は、従来、人手によるものが多く、個人の経験や勘に頼っているため、漏れや重複が多かった。テストケースに漏れや重複があると、プログラムにバグが残り、ソフトウェアの信頼性が低下する恐れがある。テストケース生成技法は、逐次処理プログラムの場合、そのソースコードや仕様に基づいた、様々な研究がなされている(例えば[1])。しかしながら、並行処理プログラムの場合、テストケースの考え方すらほとんど研究されていないのが現状である[2][3][4]。並行処理プログラムは、逐次処理プログラムに比較すると動作が複雑であるので、逐次処理プログラムのテスト手法をそのまま利用するだけでは不十分である。しかしながら、これまで提案されてきたテスト法の多くは、並行処理プログラムのテスト法として実用的なものとは言い難い。多重プロセッサシステムやLAN等が普及し、並行処理プログラムが多く書かれるようになってきた昨今[5]、並行処理プログラムのテストの質を向上することが重要である。

本稿では、並行処理プログラムのテストケースの定義、およびその生成系の試作と利用経験について述べる。並行処理プログラムのモデルとして、事象制約モデル(ECM)を用いた[7][8]。2章では、事象制約モデルを、事象グラフ(EG)と制約関係($Constraints$)を用いて簡単に紹介する。3章では、この事象制約モデルを用いて、テストケースを定義する。4章では、並行処理プログラムのテスト基準について述べる。5章では、テストケース生成ツール($tcgen$)の紹介をする。6章では、評価を行なう。

2. 並行処理プログラムのモデル化

この章では、並行処理プログラムの動作を表す事象制約モデル[7][8]を簡単に述べる。事象制約モデルは、事象グラフと制約関係からなる。

図1は、言語Adaで書かれた、「5人の哲学者の食事問題」である。このプログラムを例にとり、以後説明する。問題の簡単化のため、哲学者が2人の場合を考えていく。また、Adaの用語は断りなしに使用する(詳しくは[6])。

2.1 事象グラフ(Event Graph)

並行処理プログラムは、互いに通信する複数のタスクによって構成されている。逐次的に実行されるプログラム単位(ユニット)毎に、有向グラフを考える。この有向グラフを事象グラフEGと呼び、次に述べる並行事象文とその文を含む分岐文とを節点とし、その制御の流れを枝とする。並行事象文は、並行処理を特徴づける文である。例えば、言語Adaにおいて並行事象文は、エントリの呼び出し文や、アクセプト文、タスクの生成文などである。

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

N はグラフ EG の節点集合、E はグラフ EG の枝集合で、$e =< u, v > \in E$ ならば、$u, v \in N$ である。ここで節点 u を枝 $e =< u, v >$ における枝元節点、v を枝 $e =< u, v >$ における枝先節点と呼ぶ。s と f は、それぞれ以下の式を満足し、グラフの開始節点、および終了節点と呼ぶ。

$$s \in N \wedge \forall u[u \in N \rightarrow < u, s > \notin E].$$

$$f \in N \wedge \forall u[u \in N \rightarrow < f, u > \notin E].$$

並行処理プログラムにおいて、事象グラフは複数存在する。プログラムから抽出した全ての事象グラフの集合をEGsで表す。

$$EGs = \{EG | EG =< N, E, s, f >\}.$$

図2は、図1のプログラムにおいて哲学者が2人の場合の事象グラフである。

2.2 制約関係(Constraints)

2つのタスク T_x, T_y 間で通信を行なう際に、それぞれの事象グラフ EG_{T_x}, EG_{T_y} の、節点集合 N_{T_x} と N_{T_y} の要素の対として、以下の集合 Con を考える。集合 Con の要素に含まれる節点の対 (α, β) は、並行処理プログラムの実行時に、同時に実行されることを表す。

$Con(EG_{T_x}, EG_{T_y}) = \{(\alpha, \beta) | \alpha \in N_{T_x}, \beta \in N_{T_y}\}$,
ただし、(α, β) は同時に実行される節点対である。

ある並行処理プログラムにおいて、その中の考えられる全ての通信の組を表す集合を、制約関係 $Constraints$ と呼ぶ。

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

例えば、言語Adaにおいて制約関係は、同じ名前のエントリの呼びだし文とアクセプト文との対や、タスクの生成文と生成されるタスクを表す事象グラフの開始点との対、などである。

事象グラフと制約関係とを用いることにより、並行処理プログラムを事象制約モデル ECM によって、モデル化する。これは、以下のように表せる。

$$ECM =< EGs, Constraints > .$$

図3は、サンプルプログラムにおいて哲学者が2人の場合の事象制約モデルを図式化したものである。

3. テストケースの定義

事象制約モデル上でテストケースを定義するために、まず事象グラフ上での、テストケースについて考える。

```
  procedure dining_philosophers is

  seats : constant :=5;
  type seat_assignment is range 1 .. seats;

  task type fork is
       entry up;
       entry down;
  end fork;

  forks : array (seat_assignment) of fork;                    -- active all fork tasks

  task body fork is
 0 begin
 1      loop
 2           accept up;
 3           accept down;
 4      end loop;
-1 end fork;

  generic
       N : in seat_assignment;                  -- used to identify each philosopher task
  package philosopher is
  end philosopher;

  package body philosopher is
       eat, think : constant := 10.0;                    -- seconds

       task p;
       task body p is
 0     begin
 1          loop
 2                    forks(N).up;                -- acquire left fork ...
 3                    forks(N mod seats + 1).up;                -- and right fork
 4                    delay eat;
 5                    forks(N).down;                -- put down left fork ...
 6                    forks(N mod seats + 1).down;                -- and right fork
 7                    delay think;
 8          end loop;
-1      end p;
  end philosopher;

  package Aquinas is new philosopher(1);                    -- activate philosopher tasks
  package Bonhoeffer is new philosopher(2);
  package Kierkegaard is new philosopher(3);
  package Schaeffer is new philosopher(4);
  package Tilich is new philosopher(5);

 0 begin
       null;
-1 end dining_philosophers;
```

図 1: 言語 Ada で書かれた、哲学者の食事問題プログラム。statement の先頭の番号は、節点番号を表し、'0' は開始節点、'−1' は終了節点を表す。

3.1 事象グラフのテストケース

事象グラフは、プログラム単位毎の有向グラフである。このため、事象グラフ自体は、逐次処理プログラムである。そこで、事象グラフのテストケースは、逐次処理プログラムのテストケースと同様、事象グラフ上の路 ($Path$) として定義する。まず、事象グラフ上の「部分路 ($Subpath$)」を定義する。

【定義 1】 部分路 ($Subpath$)—— 事象グラフ $EG =< N, E, s, f >$ 上の節点の列で、隣合う節点の対は枝の集合 E に含まれる。部分路の最初の節点を始点と呼び、最後の節点を終点と呼ぶ。

【定義 2】 路 ($Path$)—— 部分路の始点と終点が、それぞれ事象グラフの開始節点 s と終了節点 f である部分列。

事象グラフ上の路 ($Path$) を作成するには、開始節点から出発して、終了節点に至る路 (幹と呼ぶ) を最初に 1 つ見つけ出す。次に、作成した路の上で、分岐している節点 (分岐点) を始点として、すでに作成した路上の節点 (合流点) を終点とする部分路を見つけ出す。最初に作成した幹の分岐点から合流点までの部分路を、次に作成した部分路で置き換えることによって、次の路を作成する。

事象グラフの路は、以下のように表せる。

$Path(EG) = \{r | r \in Subpath(EG) \wedge r(1) = s \wedge r(|r|) = f\}$,

$Subpath(EG) = \{r | r \in Seq(N) \wedge Arced(r, EG)\}$,

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

ただし、$Seq(N)$ は節点の列、$|r|$ は列 r の長さ、$r(i)$ は列 r の i 番目の要素を表す。

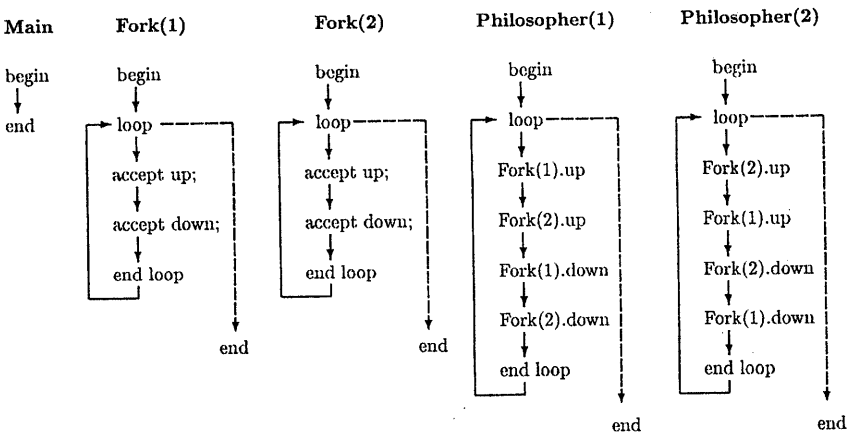

図 2: 哲学者が 2 人の場合の事象グラフ。実線は枝、破線は loop からの仮想出口を表す。

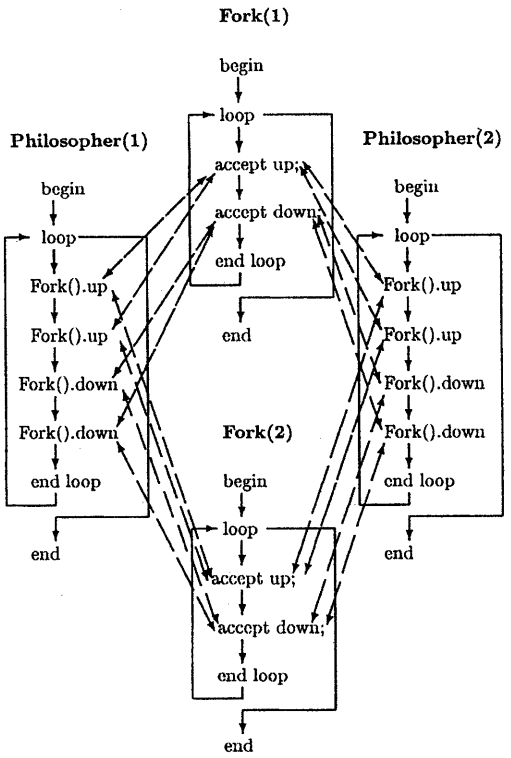

図 3: 哲学者が 2 人の場合の事象制約モデル。実線は枝、破線は制約関係を表す。

3.2 事象制約モデルのテストケース

事象グラフの路を用い、その制約関係に基づき、事象制約モデル ECM のテストケースを定義する。まず、事象グラフが 2 つの場合の「協調路 ($CoPath$)」を定義する。

【定義 3】 協調路 ($CoPath$)—X,Y を事象グラフとし、r_X, r_Y をそれぞれ X,Y 上の路とする時 ($r_X \in Path(X)$, $r_Y \in Path(Y)$)、$Con(X,Y)$ の要素 (x_j, y_k) に対して、r_X と r_Y 内の x_j と y_k の個数が同じでかつ出現順序が等しい路の対。

すなわち、協調路を作成する時は、x_j と y_k の個数が r_X と r_Y 内で、等しくなるよう、r_X と r_Y を、それぞれ $Path(X)$ と $Path(Y)$ から、選び出さなければならない。協調路は以下のように表せる。

$CoPath(X,Y) = \{< r_X, r_Y > | r_X \in Path(X) \wedge r_Y \in Path(Y) \wedge Suc(< r_X, r_Y >, Con(X,Y))\}$,

$Suc(< r_X, r_Y >, Con(X,Y)) = \forall < x,y > [< x,y > \in Con(X,Y) \rightarrow (Paired(< r_X, r_Y >, < x,y >)]$,

$Paired(< r_X, r_Y >, < x,y >) = [Num(r_X, x) = Num(r_Y, y)] \wedge \forall i [1 \leq i \leq Num(r_X, x) \rightarrow [r_X(j_i) = x] \wedge NE(< r_X(j_{i-1}, \ldots, r_X(j_i - 1) >, x) \wedge [r_Y(k_i) = y] \wedge NE(< r_Y(k_{i-1}) .. r_Y(k_i - 1) >, y)]$,

ただし、$j_0 = k_0 = r(1) = s$、$NE(r,x)$ は列 r の中に x が存在しないことを表し、$Num(r,x)$ は列 r の中に含まれる x の要素数である。

並行処理プログラムにおいて、タスクが 2 つ以上存在する時、すなわち事象グラフが 2 つ以上の場合は、任意の 2 つの事象グラフの間で協調路を定義していく。

$CoPath(EGs, Con(EGs)) =$
$\{CoPath(EG_i, EG_j) | 1 \leq i,j \leq |EGs|\}$ $(i \neq j)$.

並行処理プログラムが、m 個のタスクから成る場合、協調路は、m 個の路の組である。

【定義 4】 協調路を事象制約モデル ECM のテストケースとする。

$$TestCases(ECM) = CoPath(EGs, Constraints).$$

図 4 は協調路の例である。

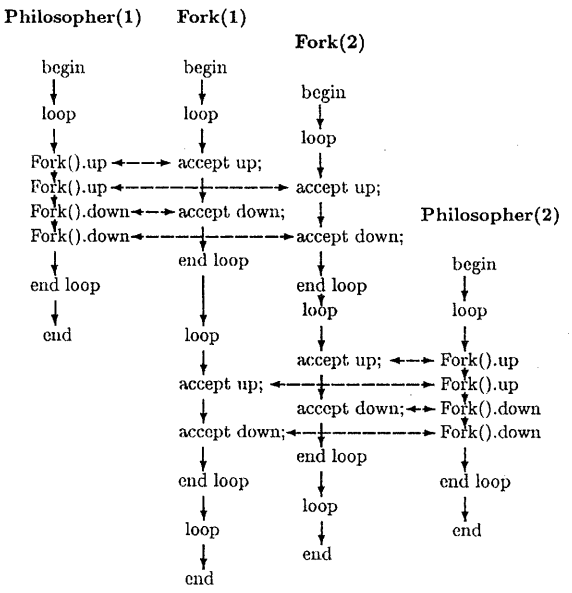

図 4: 哲学者が 2 人の場合のテストケース (協調路) の例。実線は実行順序を表す。点線は制約関係を表し、同時に実行されることを意味する。

4. テスト基準

3 章では、事象制約モデル上での協調路を並行処理プログラムのテストケースと定義した。路を生成するために、事象グラフにおける幹と、幹内の置き換え可能な部分路とを用いた。事象グラフ内に、ループが 1 つでも存在すれば、無限個数の路が生成される。テスト基準は、テストケース生成、および、テストの完了のための条件を定めたものである。この章では、並行処理プログラムのテスト基準を示す。

並行処理プログラムのテストケースが満足すべきテスト基準としては以下のものが考えられる [9]。

i) 逐次処理テスト基準 — 並行処理プログラム内の全てのタスクについては、それぞれのタスク毎に逐次処理プログラムとしてのテスト基準を満足する。

ii) 制約被覆基準 — 並行処理プログラム内の全ての制約は、テストを行なう際に、少なくとも 1 回は実現される。

iii) タスク型テスト基準 — 並行処理プログラム内にタスク型が存在する場合は、そのタスクが 2 個生成されるものとしてテストを行なう。

逐次処理テスト基準は、タスクが独立に動作するものとして計測あるいはテストケース生成が行われることを保証するものである。この基準より、事象グラフのテストケースを作成する際、以下の条件を満足するようにする。

i) 被覆条件 — 事象グラフの全ての枝を少なくとも 1 回は通る

ii) ループ条件 — ループを含む場合は 0 回と 1 回繰り返す場合を考える

ループ条件を用いることにより、生成される路の数が、無限になることはない。無限ループを含む場合も、ループからの仮想出口を考えることにより (図 2参照)、この基準で有限にすることができる。協調路を生成するためには、事象グラフ上から、上記の条件を満たす路を作り、制約関係を満足するようにそれらの路を組み合わせる。しかしながら、事象制約モデルのテストケースを作成する際、協調路が作成できない可能性がある。その際は、協調路を作ることを優先する。すなわち、協調路が作成できるように、事象グラフ上の路を作り直す。

制約被覆基準は、制約として並行処理プログラム内のどの操作を定義するかを規定していない。制約は言語や計算モデルに応じて定義しなければならない。例えば、先に検討した Ada 並行処理プログラムについての「ランデブー通路」基準 [10][11] や「広域データフロー」テスト基準 [12] は制約基準を Ada に特定し、動作として遠隔手続き呼び出しあるいは共有変数方式に限ったときのテスト基準である。

タスク型テスト基準は、被テストプログラムがタスク型を含んでいる場合の基準である。タスクは動的に生成されるが、その雛型は 1 つしか存在しない。タスク型テスト基準を設けることにより、プログラムの実行モデルを最小限に抑え、かつ実用的なテストが行なえる。

5. ツールの概要

この章では、並行処理プログラムから、協調路をテストケースとして作成するツール (*tcgen*) について説明する。*tcgen* は、Ada のソースコードから事象制約モデル上のテストケース (協調路) を自動的に生成する。*tcgen* は、以下に示す 4 つの部からなる (図 5 参照)。

i) 制御フローグラフ作成部
並行処理プログラムから、制御フローグラフと、制約関係とを取り出す。入力は、Ada のソースコードであり、出力は、プログラム単位毎の制御フローグラフと制約関係である。

ii) 事象グラフ作成部
制御フローグラフは、プログラム中の全ての文 (statement) を節点に持つ。事象グラフ作成部は、制御フローグラフを事象グラフに縮約する。入力は制御フローグラフであり、出力はその制御フローグラフを縮約した事象グラフである。

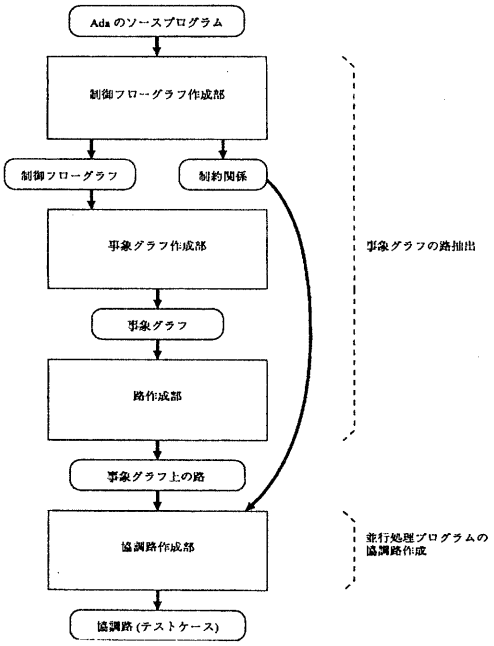

図 5: ツール *tcgen* の概要

iii) 路作成部
事象グラフから、事象グラフ上の路を作成する。入力は事象グラフであり、出力はその事象グラフ上の路である。

iv) 協調路作成部
事象グラフ上の路から、制約関係に基づき、協調路を作成する。入力は事象グラフの路と制約関係であり、出力は協調路である。

制御フローグラフ作成部は、Yacc[13] と Lex[14] を用いて作成した。すなわち、Ada の構文規則 [15] の中に記述したアクションによって、制御フローグラフと、制約関係とを取り出した。その他の 3 つの部は、言語 C で記述した。

図 6は、図 1に示した、言語 Ada で書かれた哲学者の食事問題プログラムをツール *tcgen* に適用した結果である。[Constraints] は、数字の 4 つ組 $(pr1, n1, pr2, n2)$ から構成され、プロセス $pr1$ の節点 $n1$ と、プロセス $pr2$ の節点 $n2$ とが、同時に実行されることを表している。ここで、プロセスの番号は、0, 1 がタスク Fork を、2, 3 がタスク Philosopher を、それぞれ表す。[Event Graphs] は、プロセスごとに表され、n (n は偶数) 番目の数字を枝元節点とし、$n+1$ 番目の数字を枝先節点とする枝を表す。[Paths] は、プロセスごとに表され、数字の並びが、路 (前章のテスト基準を満たす) を構成する節点番号である。[Subpaths] は、プロセスごとに表され、事象グラフ内での置き換え可能な部分路である。[Test-Cases] は、路の

```
[Constraints]
2 2 0 2
2 2 1 2
3 2 0 2
3 2 1 2
2 5 0 3
2 5 1 3
3 5 0 3
3 5 1 3
2 3 0 2
2 3 1 2
3 3 0 2
3 3 1 2
2 6 0 3
2 6 1 3
3 6 0 3
3 6 1 3

[Event Graphs]
pr=0  0 1 1 2 2 3 3 4 4 1 1 -1
pr=1  0 1 1 2 2 3 3 4 4 1 1 -1
pr=2  0 1 1 2 2 3 3 5 5 6 6 8 8 1 1 -1
pr=3  0 1 1 2 2 3 3 5 5 6 6 8 8 1 1 -1

[Paths]
pr=0
0 1 -1
0 1 2 3 4 1 -1
pr=1
0 1 -1
0 1 2 3 4 1 -1
pr=2
0 1 -1
0 1 2 3 5 6 8 1 -1
pr=3
0 1 -1
0 1 2 3 5 6 8 1 -1
[Subpaths]
pr=0
1 2 3 4 1
pr=1
1 2 3 4 1
pr=2
1 2 3 5 6 8 1
pr=3
1 2 3 5 6 8 1

[Test-Case0]
pr:0 - 0 1 -1
pr:1 - 0 1 -1
pr:2 - 0 1 -1
pr:3 - 0 1 -1
[Test-Case1]
pr:0 - 0 1 2 3 4 1 -1
pr:1 - 0 1 2 3 4 1 -1
pr:2 - 0 1 2 3 5 6 8 1 -1
pr:3 - 0 1 -1
[Test-Case2]
pr:0 - 0 1 2 3 4 1 -1
pr:1 - 0 1 2 3 4 1 -1
pr:2 - 0 1 -1
pr:3 - 0 1 2 3 5 6 8 1 -1
[Test-Case3]
pr:0 - 0 1 2 3 4 1 1 2 3 4 1 -1
pr:1 - 0 1 2 3 4 1 1 2 3 4 1 -1
pr:2 - 0 1 2 3 5 6 8 1 -1
pr:3 - 0 1 2 3 5 6 8 1 -1
```

図 6: 哲学者の食事問題を *tcgen* に適用した結果

組で表される。

6. 議論および評価

この章では、作成したツール *tcgen*、生成したテストケース(協調路)、および、事象制約モデルを用いたテスト法、それぞれについて評価を行なう。

6.1 ツール

ここでは、Ada 並行処理プログラムからテストケースを作成するツール *tcgen* について述べる。

表 1: ツール *tcgen* の実行時間

program	tasks	statements	time (s)
Producer_Consumer	3	58	0.15
Dining_Philosopher	4	59	0.13
SIEVE	4	78	0.12

表 1 は、ツール *tcgen* を、Ada 並行処理プログラムに適用、その際にかかった時間である[1]。Producer_Consumer は、いわゆる生産者消費者問題と呼ばれるもので、共有バッファがあり、生産者がそこに書き込み、消費者が先頭からそれを読んでいくものである。Dining_Philosopher は、図 1に示したものである。SIEVE は、与えられた正の整数に対して、その数までに含まれる素数を検索し、出力するものである。

Ada のような並行処理言語は、例外処理の機能を持つ。現在の *tcgen* では、例外処理を含むプログラムを取り扱うことはできない。例えば、Numeric errors は、Ada において典型的な例外処理である。この場合は、事象制約モデルにおいて、プログラム中の全ての代入文を枝元節点とし、例外処理部を枝先節点とする枝を作ればよい。しかしながら、事象制約モデルは、代入文に対応する節点を持たない。たとえ、持つように定義しても、枝の数が非常に多くなる。

さらに Ada には、他のタスクを強制終了させるアボート文がある。現在の *tcgen* では、このアボート文を含むプログラムも処理することができない。

6.2 テストケース

ここでは、ツールによって生成したテストケース(協調路)について議論する。

事象制約モデル上で生成された、テストケースの実行可能性を検討する。今回提案した方法は、テストケースを機械的に生成するものである。それ故、作られたテストケースに基づいて、プログラムが実行できる保証はない。すなわち、ツールによって作成されたテストケースの通りに、被テストプログラムを実行させるためのテストデータが、必ず存在する、とは言えない。例えば、図 6の [Test-Case1] および [Test-Case2] を実行できるようなテストデータは存在しない。

[1]表 1, 2, 3 において実行時間の計測には、SUN SparcStation 2 を使用した。ただし、表 2は、文献 [3] からの抜粋である。

表 2: 並行状態グラフの構成にかかる時間

philosophers	tasks	states	edges	time (s)
2	4	19	28	2.4
3	6	84	186	2.8
4	8	375	1112	3.5
5	10	1653	6130	5.4
6	12	7282	32412	21.6
7	14	32063	166502	648

表 3: 事象制約モデルの構成にかかる時間

tasks	nodes	edges	Constraints	time (s)
4	28	28	16	0.09

6.3 テスト法

ここでは、事象制約モデルを用いたテスト法と、他に提案されたテスト法との比較を行なう。

Taylor らは、並行処理プログラムにおける構造的テスト法の概念を提案した [3]。並行処理プログラムのモデルとして、並行状態グラフを定義した。並行状態グラフは、各タスクの状態の組である並行状態と、その間の制御の移行を表す枝とからなる。彼らは、並行状態グラフ上の、状態や枝の被覆率に基づいたテスト基準を提案した。しかしながら、この並行状態グラフは、(1) タスクの数が予め分かっていなければ作ることができない。(2) タスクの状態数が増えるにつれ、並行状態の数が膨大になる、という問題点がある。そのため、並行状態グラフは、並行処理プログラムのモデルとして、実用的なものではない。表 2は、文献 [3] より抜粋した、哲学者問題における、並行状態グラフの構成にかかる時間と、実際の並行状態数である。表 3に、事象制約モデルを構成する時間を示した。哲学者の食事問題プログラムは、タスク型を含んでいる。4 章で定めたテスト基準に基づくと、哲学者が何人であっても、事象制約モデルは図 3で表され、節点数が増える事はない。

さらに、被テストプログラムに基づいて、テストケースを選ぼうとしても、並行状態グラフは、元のプログラムとの対比が難しい。事象制約モデルは、並行処理プログラムのソースコードに基づき作られる。このため、元のプログラムと容易に対応づけることができる。

Tai は並行処理プログラムの再実行可能なテストを提案し、同期列を用いた [2]。同期列は、並行に実行可能な文も一つの列で表現する。事象制約モデルの協調路は、制約関係を満足する路の組である。このため、各々の事象グラフにおいて、半順序関係を満足する実行系列を表現している。協調路を用いることによって、並行処理プログラム内の半順序関係を守ることができる。さらに、並行処理プログラムのテストケースの数が、膨大なものになることを防止できる。しかしながら、事象制約モデルを用いた場合、テストを行なう際に、被テストプログラムを、何らかの形

で与えられたテストケース通りに、強制的に実行させることが必要である。

7. おわりに

本論文では、並行処理プログラムのテストケースについて、その定義と作成系を述べた。事象グラフ EG と制約関係 $Constraints$ を用い、事象制約モデル ECM として並行処理プログラムをモデル化し、テストケース(協調路)を作成する。このようにして作成されたテストケースは、論理的かつ系統的に作られるため、漏れや重複の減少が期待できる。さらに、並行処理プログラムからテストケース(協調路)を作成するツール *tcgen* を作成し、評価を行なった。事象制約モデルは、プログラムのソースコードに基づき生成される。そのため、容易に、元のプログラムと対応づけることができる。さらに、生成した協調路は、制約関係を満たす路の組であるので、各々の事象グラフにおいて、半順序関係を満足する。

今後の課題として、以下の点が挙げられる。

i) 様々なプログラムへのツール *tcgen* の適用
今回、*tcgen* を 3 種類の Ada 並行処理プログラムに適用した。しかしながら、3 種類とも statement 数が 100 行に満たない、いわば小さなプログラムである。今後は、もっと大規模なプログラムへの適用を検討する。

ii) 例外処理対応
現在の *tcgen* では、例外処理を含むプログラムに対処できないことが分かっている。今後は、そのようなプログラムにも対応できる、拡張版 *tcgen* について検討する。

iii) テストケースの実行可能性
今回述べてきた方法は、テストケースを機械的に作成するものである。そのために、プログラムの実行可能性についての保証はない。このことは早急に解決する必要がある。

iv) テストケースの強制実行
今回、ツール *tcgen* によって、並行処理プログラムからテストケース(協調路)を取り出すことができた。この協調路通りに、被テストプログラムを強制的に実行させることが必要である。これが解決すれば、上記のテストケースの実行可能性についても、何らかの解答が得られる可能性がある。

謝辞

並行処理プログラムについて御教授下さった九州大学工学部情報工学科の荒木啓二郎助教授、程京徳助教授に感謝致します。また、テスト法について議論して頂いた、九州大学工学部情報工学科の、三浦好弘君、伊東栄典君に感謝します。

参考文献

[1] Denney,R.: "Test-Case Generation from Prolog-Based Specifications," IEEE Softw. Vol.8, No.2, pp.49-57, 1991.

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

[3] Taylor,R.N., Levine,D.L. and Kelly,C.D.: "Structural testing of Concurrent Programs," IEEE Trans. on Softw. Engin., Vol.18, No.3, pp.206-215, Mar. 1992.

[4] Chang,C.K. *et al.*: "INTEGRAL - An Integrated Framework for Distributed Software Validation and Verification," Proc. Workshop on the Future Trend of Distributed Computing Systems in the 1990s, pp.301-310, 1988.

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

[6] Unaited States Department of Defence Reference Manual for the Ada Programming Language, 1983.

[7] 片山徹郎, 古川善吾, 牛島和夫: "並行処理プログラムにおけるテストケースについて," 情報処理学会第 43 回全国大会, Vol.5, pp.321-322, 1991 年 10 月.

[8] Katayama,T., Furukawa,Z. and Ushijima,K.: "Event-Constraint Model of a Concurrent Program for Test-Case Generation," Proc. of JCSE'92, pp.285-292, Mar. 1992.

[9] 古川善吾, 牛島和夫: "並行処理プログラムのテスト法に関する一考察," 日本ソフトウェア科学会第 6 回大会論文集, pp.185-188, 1989 年.

[10] 古川善吾, 牛島和夫: "事象グラフを用いた Ada プログラムのモデル化," 日本ソフトウェア科学会第 7 回大会論文集, pp.149-152, 1990 年.

[11] 古川善吾, 牛島和夫: "ランデブー通路を用いた Ada 並行処理プログラムのテスト十分性評価," 電子情報通信学会論文誌 D-I, Vol.J75-D-I, No.5, pp.288-296, 1992 年 5 月.

[12] 有村耕治, 古川善吾, 牛島和夫:"並行プログラムにおける広域データフロー基準の拡張," 情報処理学会第 42 回全国大会, 1991 年 3 月.

[13] YACC - Yet Another Compiler Compiler : ULTRIX-32 Supplementary Documents, Vol. II:Programmer.

[14] LEX - Lexical Analyzer Generator : ULTRIX-32 Supplementary Documents, Vol. II:Programmer.

[15] Fisher,G.: Philippe Charles A LALR grammar for ANSI Ada Adapted for YACC (UNIX) Inputs, 1984.