並行処理プログラムにおけるテストケースの定義と生成ツールの試作

片山徹郎[†] 菰田敏行[†]
古川善吾[††] 牛島和夫[†]

ソフトウェアのテストを行う際，テストデータについての条件を記述したテストケースは，ソフトウェアの信頼性向上に重要な役割を果たす．逐次処理プログラムのテストケース作成技法については，さまざま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，およびその生成ツール(TCgen)の試作と利用経験について述べる．プログラム単位ごとに事象グラフを作り，事象グラフ間で同期する節点を同期関係で結んだ事象同期グラフ(ESG)によって，並行処理プログラムをモデル化する．事象同期グラフ上の協調路(Copath)を，並行処理プログラムのテストケースと定義する．テストケース生成ツールTCgenは，プログラミング言語Adaで書かれた並行処理プログラムを入力とし，協調路を出力とするツールである．ツールTCgenによって作成された協調路は，テストケースの漏れや重複を少なくすると期待される．しかしながら，実際のテストデータを作成する段階で，実行可能性についての問題が残る．

Definition of the Test-Case for Concurrent Programs and Prototype of Test-Case Generation System

Tetsuro Katayama,[†] Toshiyuki Komoda,[†] Zengo Furukawa[††] and Kazuo Ushijima[†]

In testing of software, test-cases, which specify test-data, play an important role for improving the reliability of software. For sequential programs, we have practical methods for generating test-cases. Test-cases for concurrent programs have been seldom studied. 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 use Event-Synchs-Graph (ESG) which consists of Event Graph, which represents abstract control flows of a program unit, and of Synchs, which represent a simultaneous execution among Event Graphs. We define that test-cases of concurrent programs are Copaths on an ESG. We input an Ada concurrent program to TCgen, and then we get Copaths of the program. The TCgen makes sufficient test-cases and no overlapping. However, it is necessary to validate feasibility of them for deciding test-data.

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，実際に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証できるので，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，テストを行う際には，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．また，テストケースに重複があると，テスト作業に余分な時間がかかり，ソフトウェア開発にかかるコストが増大する．テストケース生成技法は，逐次処理プログラムの場合，ソースコードや仕様に基づいた研究がなされている[2]．一方，並行処理プログラムの場

† 九州大学工学部情報工学科
Department of Computer Science and Communication Engineering, Kyushu University
†† 九州大学情報処理教育センター
Educational Center for Information Processing, Kyushu University

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[15),16)]．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，これまで並行処理プログラムに対して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いいがたい．

本稿では，並行処理プログラムのテストケースの定義，およびテストケース生成ツール TCgen 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動作を把握するために，事象同期グラフ（ESG：Event-Synchs-Graph）を用いる[8)~10)]．2章では，事象同期グラフを，事象グラフ（EG：Event Graph）と同期関係（Synchs）を用いて簡単に紹介する．3章では，この事象同期グラフを用いて，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）について述べる．6章では，議論および評価を行う．

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同期グラフ[8)~10)]について簡単に述べる．事象同期グラフは，事象グラフと同期関係とからなる．

図1は，言語 Ada で書かれた，「5人の哲学者の食事問題」[16)] を解くプログラムのソースリストである．各文の前の番号は，対応する事象グラフ（後述）の節点番号を表す．このプログラムは，哲学者（汎用体パッケージ philosopher），フォーク（タスク型 fork），main（手続き dining philosopher）の3つの部分からなる．main の本体では，何の仕事も行っていない．しかしながら，タスクやパッケージの実体の宣言によって，タスクやパッケージの実行を開始する．タスク型 fork は，フォークの取り上げと解放に対応するエントリ up と down を持っている．汎用体パッケージ philosopher には，食事と瞑想の時間を表す変数 eat と think および哲学者の動作を表すタスク型 p がある．哲学者は，自分の左側のフォークを取り上げ，次に右側のフォークを取り上げる．隣の哲学者がフォークを使用中であれば待たされる．食事が終わると，左右のフォークを解放し，think の時間瞑想する．これら一連の動作を繰り返す．

このプログラムを例にとり，以後説明する．また，

```
   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．

Fig. 1 A dining_philosopher program written in Ada language.
The numbers in head of statements are node numbers. '0' denotes start node. '−1' denotes final node.

Ada の用語は断りなしに使用する（詳しくは文献17)を参照）．

2.1 事象グラフ

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）ごとに，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．

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

N はグラフ EG の節点集合，E はグラフ EG の枝集合で，$e=\langle u, v\rangle \in E$ ならば，$u, v \in N$ である．ここで節点 u を枝 $e=\langle u, v\rangle$ における枝元節点，v を枝 $e=\langle u, v\rangle$ における枝先節点と呼ぶ．s と f は，そ

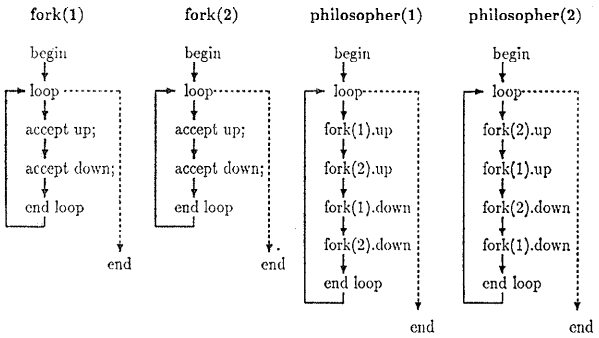


図 2 哲学者が2人の場合の事象グラフ
実線は枝，点線は loop からの仮想出口を表す．
Fig. 2 The Event Graphs for two philosophers.
A solid arrow denotes an edge. A dotted arrow denotes a virtual exit from an infinite loop.

れぞれ以下の式を満たし，それぞれグラフの開始節点および終了節点と呼ぶ．

$s \in N \wedge \forall u[u \in N \rightarrow \langle u, s\rangle \notin E]$,

$f \in N \wedge \forall u[u \in N \rightarrow \langle f, u\rangle \notin E]$.

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すべての事象グラフの集合を EGs で表す．

$EGs = \{EG \mid EG = \langle N, E, s, f\rangle\}$.

図 2 は，図1のプログラムにおける哲学者が2人の場合の事象グラフである．実際のプログラムには無限ループが存在するが，テストを有限時間で終了させるために，無限ループには仮想出口を考える．図中の点線は，ループからの仮想出口を表す枝である．

2.2 同期関係

2つのタスク T_A, T_B 間で通信を行う際に，それぞれの事象グラフ EG_A, EG_B の，節点集合 N_A と N_B の要素の対の集合 Syn を考える．集合 Syn の要素に含まれる節点の対 $\langle a, b\rangle$（同期対と呼ぶ）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．

$Syn(EG_A, EG_B) = \{\langle a, b\rangle \mid a \in N_A, b \in N_B\}$,

ただし，$\langle a, b\rangle$ 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，そのプログラム中のすべての同期の組を要素とする集合を，同期関係 $Synchs$ と呼ぶ．

$Synchs = \{\langle a, b\rangle \mid \exists A, \exists B[\langle a, b\rangle \in Syn(A, B) \wedge A, B \in EGs]\}$.

例えば，言語 Ada において同期対は，エントリの呼び出し文とエントリに対して同じ名前の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の開始文との対である．

事象グラフと同期関係とを用いることにより，並行処理プログラムを事象同期グラフ ESG によって，モデル化する．これは，以下のように表せる．

$ESG = \langle EGs, Synchs\rangle$.

図 3 は，図1のプログラムにおいて哲学者が2人の場合の事象同期グラフを図式化したものである．

3. テストケースの定義

事象同期グラフ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．

3.1 事象グラフのテストケース

事象グラフは，プログラム単位ごとに作成する．このため，事象グラフ自体は，逐次処理プログラムを表している．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*）」を定義した後に，路を定義する．

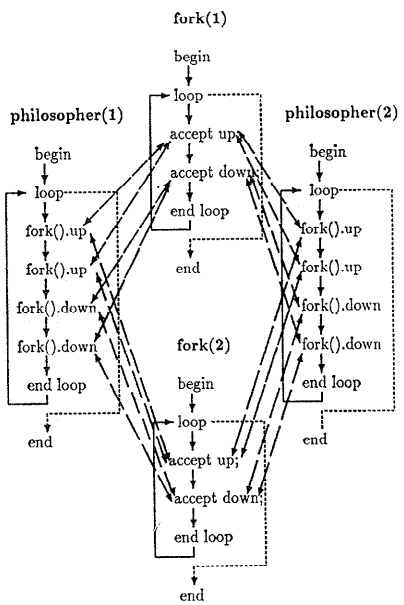


図 3 哲学者が2人の場合の事象同期グラフ
実線は枝，破線は同期対を表す．
Fig. 3 The ESG for two philosophers.
A solid arrow denotes an edge. A dash arrow denotes a Synch.

【定義1】 部分路 (*Subpath*)—事象グラフ $EG=\langle N, E, s, f\rangle$ 上の節点の列 α で，隣合う節点の対が枝の集合 E に含まれる．すなわち，以下のように表せる．

$$Subpath(EG)=\{\alpha \mid \alpha\in Seq(N)\wedge Arc(\alpha, EG)\},$$
$$Arc(\alpha, EG)=\forall i[1\leq i<|\alpha|\rightarrow\langle\alpha(i), \alpha(i+1)\rangle\in E],$$

ただし，$Seq(N)$ は節点の列，$|\alpha|$ は列 α の長さ，$\alpha(i)$ は列 α の i 番目の要素を表す．

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*)—路 α は，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 である部分路である．すなわち，以下のように表せる．

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

路作成のアルゴリズムを以下のように構築した．

【路作成アルゴリズム】

Step 1. 開始節点から出発して終了節点に至る路を，深さ優先探索によって1つ見つけ出す．

Step 2. 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）を終点とする別の部分路を見つけ出す．発見できなければ終了．

Step 3. 路の分岐点から合流点までの部分路を，Step 2. で見つけ出した部分路で置き換える．置き換えてできた部分路は路である．

Step 4. Step 2. へ戻る．

3.2 事象同期グラフのテストケース

事象グラフの路を用い，同期関係に基づいて，事象同期グラフ *ESG* 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*)—A, B を事象グラフとし，α, β をそれぞれ A, B 上の路とするとき ($\alpha\in Path(A)$, $\beta\in Path(B)$), $Syn(A, B)$ の要素 $\langle a_j, b_k\rangle$ に対して，α と β 内の a_j と b_k の個数が同じでかつ出現順序が等しい路の対である．すなわち，協調路は以下の式を満足する．

$$Copath(A, B)=\{\langle\alpha, \beta\rangle \mid \alpha\in Path(A)\wedge\beta\in Path(B)\wedge Suc(\langle\alpha, \beta\rangle, Syn(A, B))\},$$
$$Suc(\langle\alpha, \beta\rangle, Syn(A, B))=\forall\langle a, b\rangle[\langle a, b\rangle\in Syn(A, B)\rightarrow[Num(\alpha, a)=Num(\beta, b)]\wedge\exists\langle c, d\rangle\in Syn(A, B)\rightarrow\wedge\forall i[1\leq i\leq Num(\alpha, a)\rightarrow[\alpha(j_i)=a]\wedge Num(\langle\alpha(j_{i-1})\cdots\alpha(j_i-1)\rangle, c)=0\wedge[\beta(k_i)=b]\wedge Num(\langle\beta(k_{i-1})\cdots\beta(k_i-1)\rangle, d)=0],$$

ただし，$j_0=k_0=r(1)=s$, $\alpha(j_i)=a$ は列 α において，先頭から j 番目に現れ，$Syn(A, B)$ の要素として i 番目に現れる a を表す．$Num(r, a)$ は列 r の列の中に含まれる a の要素数である．

協調路作成のアルゴリズムを以下のように構築した．

【協調路作成アルゴリズム】

Step 1. 路集合 $Path(A)$ から，ある路 α を取り出す．

Step 2. 路 α 中に存在する，同期対 $Syn(A, B)$ に含まれる節点を開始節点から順に取り出す．

Step 3. Step 2. で取り出した節点それぞれに対し，その節点と対をなす節点が含まれ，かつその節点の順番が Step 2. で取り出した節点列と等しく現れる路 β を，$Path(B)$ から選び出す．条件を満たす路が $Path(B)$ に存在しなければ終了．

Step 4. 路 α と路 β との対が協調路である．

Step 5. Step 1. に戻る．

ただし，Step 2., Step 3. において，同期をとらない路同士も協調路となる．

並行処理プログラムにおいて，タスクが2つ以上存在するとき，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する．

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

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】 協調路を事象同期グラフ *ESG* のテストケース (*TestCases*) とする．

$$TestCases(ESG)=Copaths(EGs).$$

図4は，図3の事象同期グラフにおける協調路の例である．おのおのの事象グラフから，同期対を満足す

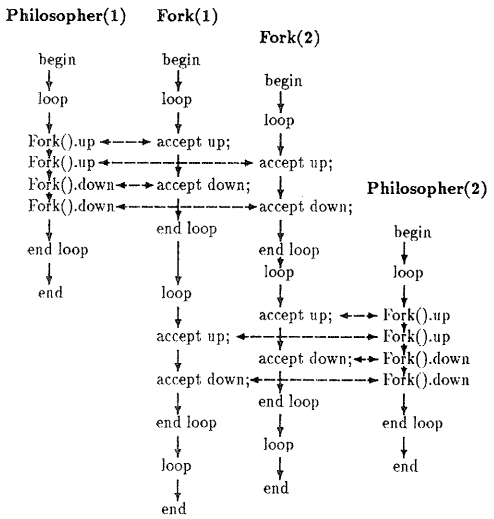


図 4 哲学者が2人の場合のテストケース（協調路）の例
実線は実行順序を表す．破線は同期対を表し，同時に実行されることを意味する．

Fig. 4 A sample test-case for two philosophers. A solid arrow denotes the execution order. A dash arrow denotes a simultaneous execution.

るように，路を取り出したものである．図中における破線は同期対を表しており，協調路に対する理解を助けるために加えた．

4. テスト基準

3章では，事象同期グラフ上での協調路を，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，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について検討する．

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[4]．

1) 枝（節点）被覆基準—モデル内のすべての枝（節点）を少なくとも1回は通る．
2) ループ被覆基準—モデル内にループが存在する場合は0回と1回繰り返す場合を考える．
3) 制約被覆基準—並行処理プログラム内のすべての制約は，テストを行う際に，少なくとも1回は実現される．

枝（節点）被覆基準は，事象同期グラフにおいて，事象グラフ内のすべての枝を少なくとも1回通ることを意味する．また，ループ被覆基準は，事象グラフ内に，ループが存在する場合の基準である．この2つの基準は，逐次処理プログラムにおいて用いられてきたテスト基準である．並行処理プログラムにおいても，タスクは独立かつ逐次的に動作するので，テストケースを生成する際，このテスト基準を満足するようにする．

さらに，ループ被覆基準を用いることにより，ループを特別な対象として認識したテストを保証すると同時に，生成される路の数が，無限になることを防止できる．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．

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．今回は，言語を Ada に特定し，制約を *Syn*，すなわち同期対とする．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[5),6)]や「広域データフロー」テスト基準[1),7)]は，制約被覆基準において，言語を 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のデータフローに限ったときのテスト基準である．

協調路を生成するためには，事象グラフから枝（節点）被覆基準とループ被覆基準とを満たす路を作り，同期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，この手順を用いて事象同期グラフのテストケースを作成すると，先に事象グラフの上で路を生成するために，制約被覆基準を満足する協調路が作成できない可能性がある．その際は，制約被覆基準を満足する協調路が作成できるように，上記の枝（節点）被覆基準とループ被覆基準とを緩和して事象グラフ上の路を作り直す．

並行処理プログラムには，タスク型を含むものがある．これは，雛型となるタスクのソースコードが1つだけ存在し，実行時に，動的にタスクの実体が複数個生成される可能性があるものである．このようなタスク型を含んだ並行処理プログラムに対して，従来検討されてきたテスト法では，生成されるタスク実体の個数を前もって知っていなければならず，生成されるタスク実体の個数に応じて，モデルを作り変える必要があった（詳しくは6章で述べる）．

事象同期グラフでは，並行処理プログラム内にタス

ク型が存在する場合，そのタスクが2個生成されるものとしてモデルを構成する[11),12)]．タスク実体が1個だけ生成されるモデルを構成すると，同じタスク型の実体の間で同期をとる（同期対がある）場合，被テストプログラムの実際の動作を反映できず，テストケースを作成することができない．また，3個以上の実体が生成されるとしてモデルを構成すると，モデルが複雑化し，それにともない，テストケース数の増大を招く．

5. テストケース生成ツールの概要

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生成するテストケース生成ツール(TCgen)について述べる．TCgen は，Ada のソースコードから事象同期グラフ上のテストケース（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）．

1) 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，同期関係とを取り出す．入力は，Ada で書かれたソースコードであり，出力は，プログラム単位ごとの制御フローグラフと同期関係である．

2) 事象グラフ作成部
制御フローグラフは，プログラム中のすべての文 (statement) を節点に持つ．事象グラフ作成部は，制御フローグラフから，代入文などの並行処理に直接関係しない文を表す節点を削除し，事象グラフを構成する．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から不要な節点を除いた事象グラフである．

3) 路作成部
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の集合である．

4) 協調路作成部
同期関係に基づき事象グラフ上の路を組み合わせ，協調路を作成する．入力は事象グラフの路の集合と制約関係であり，出力は協調路である．

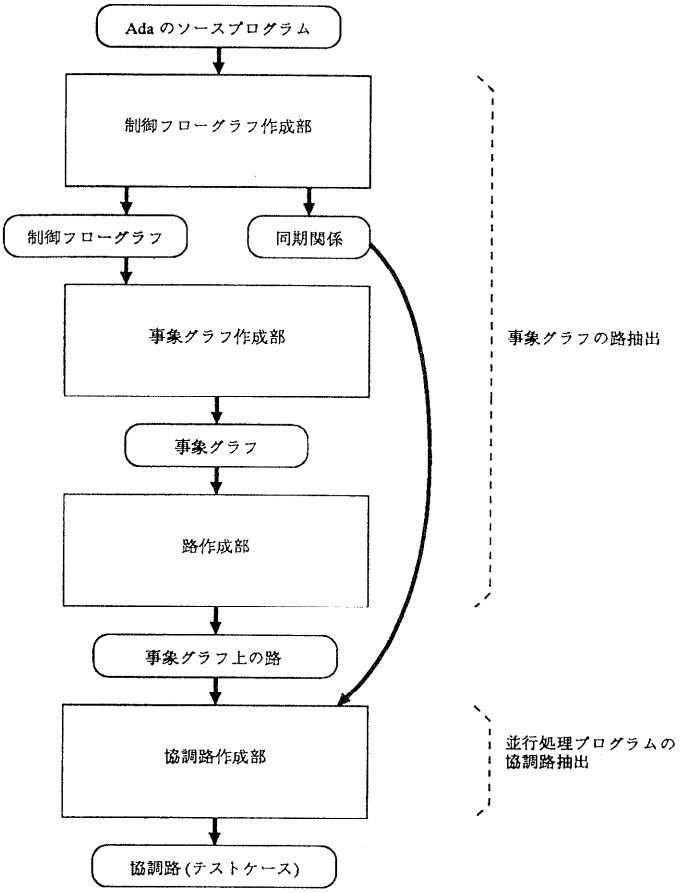


図5 テストケース生成ツール TCgen の概要
Fig. 5 Outline of the test-case generation tool TCgen.

制御フローグラフ作成部は，Yacc[18)]とLex[14)]を用いて作成した．すなわち，Ada の構文規則[3)]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，同期関係とを取り出す．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，Ada で書かれた哲学者の食事問題プログラムを，ツール TCgen に適用した際の出力結果である．[Synchs] 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（前章の枝（節点）被覆基準，およびループ被覆基準を満たす）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，路の組で表される協調路である．

```
[Synchs]
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 2 3 4 1 -1
pr:1 - 0 1 2 3 4 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 哲学者の食事問題に TCgen を適用した際の出力結果
Fig. 6 The output for the dining_philosopher program in TCgen.

6. 議論および評価

この章では，テストケース生成ツール TCgen，生成したテストケース（協調路），および，事象同期グラフを用いたテスト法，それぞれについて議論および評価を行う．

6.1 ツール

表 1 は，テストケース生成ツール TCgen を，Ada 並行処理プログラムに適用し，その際にかかった時間を測定した結果である*．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見つけ出して，出力するものである．

Ada 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，標準として定義された例外である．この例外が発生した際の処理を，例外処理として記述できる．Numeric errors の場合は，事象同期グラフにおいて，プログラム中のすべ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，モデル化ができる．しかしながら，事象同期グラフは，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ので，モデル自体が実用的でなくなる可能性がある．

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にも対処することができない．

表 1 テストケース生成ツール TCgen の実行時間
Table 1 Execution time of TCgen.

Program	Tasks	Statements	Time (s)
producer_consumer	3	58	0.15
dining_philosopher	4	59	0.13
sieve	4	78	0.12

* 表 1，2，3 において実行時間の計測には，SUN Sparc Station 2 を使用した．ただし，表 2 は文献16)からの抜粋である．

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．

事象同期グラフ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．それゆえ，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とおりに，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，必ず存在するとはいえない．逐次処理プログラムの場合でも，同じようにテストデータが存在しないテストケースを生成してしまう場合がある．今回の場合，図6の [Test-Case 1] および [Test-Case 2] を実行するためのテストデータは存在しない．

前章で，テストケース生成ツール 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）が取り出せることを示した．次の段階として，この協調路どおりに，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．これを解決すれば，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．

6.3 テスト法

ここでは，事象同期グラフ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[16]．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)タスクの数があらかじめわ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組合せ論的に増加する，という問題がある．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，実用的なものとはいいがたい．

表2は，哲学者の食事問題における，並行状態グラフの生成にかかる時間と，実際の並行状態数である（文献16）より抜粋した）．**表3**は，哲学者の食事問題における，事象同期グラフを生成する時間，および構成要素数である．哲学者の食事問題プログラムは，タスク型を含んで記述されている．4章で述べたように，事象同期グラフは，タスクの実体が2個生成されるとしてモデルを構成する．そのため，哲学者が何人であっても，事象同期グラフは図3で表され，節点数が増えることはなく，テストケースの数も増えることはないので，実用時間でテストを行うことができると考えられる．しかしながら，2つのタスク型の実体を区別していないために不要な同期対が設定され，事象同期グラフを複雑にしている．実体を区別することができれば，事象同期グラフをより簡単にすることができる．

表 2 並行状態グラフの構成にかかる時間
Table 2 Execution time for construction of concurrent state graph.

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 事象同期グラフの構成にかかる時間
Table 3 Execution time for construction of ESG.

Tasks	Nodes	Edges	Synchs	Time (s)
4	28	28	16	0.09

さらに，並行状態グラフは，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，元のプログラムとの対応が困難である．事象同期グラフは，並行処理プログラムのソースコードに基づいて作成される．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のために，同期列を提案した[15]．同期列は，並行に実行可能な文の列を全順序づけられた1つの列で表現する．これに対して，事象同期グラフの協調路は，同期関係を満足する路の組である．このため，おのおのの事象グラフにおいては全順序づけられた列であるが，並行処理プログラム全体に対しては，同期関係で規定された半順序関係を満足する実行系列を表現している．協調路は全順序化しないので，並行処理プログラムのテストケースの数が増大しない．

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，その定義と生成ツールについて述べた．事象

グラフ *EG* と同期関係 *Synchs* を用いた事象同期グラフ *ESG* として並行処理プログラムをモデル化し，その上の協調路としてテストケースを定義し，Ada のソースプログラムから生成ツール TCgen によって自動的にテストケースを生成する．このようにして生成されたテストケースは，与えられたテスト基準を満足するように，系統的に作られるため，漏れや重複の減少が期待できる．事象同期グラフは，プログラムのソースコードに基づいて生成される．そのため，元のプログラムと容易に対応づけることができる．さらに，生成した協調路は，同期関係を満たす路の組であるので，同期関係によって規定された半順序関係を満足する．また，事象同期グラフは，哲学者の食事問題のような，タスク型を含む並行処理プログラムにも，対応できることがわかった．

今後の課題として，以下の点が挙げられる．

1) さまざま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種類とも文数が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行い，検討する必要がある[13]．

2) 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わかっている．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．

3) テストケースの実行可能性

今回述べてきた方法は，プログラムのソースコードからテストケースを機械的に生成するものである．そのために，生成したテストケースの実行可能性を保証することはできない．今後，被テストプログラムを強制的に実行させて，実行可能性を実際のプログラム実行で確認するという対策を検討する必要がある．

4) タスク型を含んだ並行処理プログラムのテスト法についての検討

今回は，哲学者の食事問題を例にとり説明を行った．さらに，各種のプログラムで事例を積み重ねる必要がある．

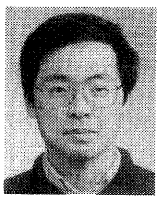
謝辞 並行処理プログラムについてご教授下さった九州大学工学部情報工学科の荒木啓二郎助教授（現在，奈良先端科学技術大学院大学情報科学研究科教授），程京徳助教授に感謝いたします．また，テスト法について一緒に議論した，九州大学大学院情報工学専攻の三浦好弘君，伊東栄典君，川口 豊君に感謝します．

参 考 文 献

1) 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，第42回情報処理学会全国大会講演論文集，Vol. 5, pp. 220-221 (1991).
2) Denney, R.: Test-Case Generation from Prolog-Based Specifications, *IEEE Softw.*, Vol. 8, No. 2, pp. 49-57 (1991).
3) Fisher, G. and Charles, P.: A LALR Grammar for ANSI Ada Adapted for YACC (UNIX) Inputs (1984).
4) 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).
5) 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).
6) 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，Vol. J 75-D-I, No. 5, pp. 288-296 (1992).
7) 古川善吾，有村耕治，牛島和夫：並行処理プログラムにおける共有変数のデータフローテスト基準，情報処理学会論文誌，Vol. 33, No. 11, pp. 1394-1401 (1992).
8) 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，第43回情報処理学会全国大会講演論文集，Vol. 5, pp. 321-322 (1991).
9) Katayama, T., Furukawa, Z. and Ushijima, K.: Event-Constraint Model of a Concurrent Program for Test-Case Generation, *Proc. JCSE '92*, pp. 285-292 (1992).
10) 片山徹郎，菰田敏行，古川善吾，牛島和夫：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，情報処理学会ソフトウェア工学研究会研究報告，Vol. 92, No. 59, pp. 9-16 (1992).
11) 片山徹郎，古川善吾，菰田敏行，牛島和夫：並行処理プログラムのテストケース生成におけるタスク型に関する一考察，日本ソフトウェア科学会第9回大会論文集，pp. 49-52 (1992).
12) 片山徹郎，古川善吾，菰田敏行，牛島和夫：並行処理プログラムのテストにおけるタスク型について，平成4年度電気関係学会九州支部連合大会論文集，p. 673 (1992).
13) 菰田敏行，片山徹郎，古川善吾，牛島和夫：並行処理プログラムのテストケース作成ツールについて，第45回情報処理学会全国大会講演論文集，Vol. 5, pp. 253-254 (1992).

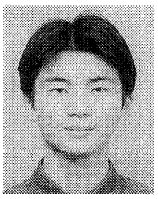
14) LEX—Lexical Analyzer Generator : ULTRIX-32 Supplementary Documents, Vol. II : Programmer.
15) Tai, K. C. : On Testing Concurrent Programs, *Proc. Compsac '85*, pp. 310-317 (1985).
16) Taylor, R. N., Levine, D. L. and Kelly, C. D. : Structural Testing of Concurrent Programs, *IEEE Trans. Softw. Eng.*, Vol. 18, No. 3, pp. 206-215 (1992).
17) United States Department of Defence Reference : Manual for the Ada Programming Language (1983).
18) YACC—Yet Another Compiler Compiler : ULTRIX-32 Supplementary Documents, Vol. II : Programmer.

(平成4年12月28日受付)
(平成5年 9 月 8 日採録)



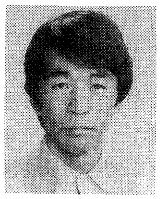
片山 徹郎（正会員）

1969 年生．1991 年九州大学工学部情報工学科卒業．1993 年同大学院情報工学専攻修士課程修了．現在，同大学院博士後期課程に在学し，ソフトウェア工学，特にソフトウェアテスト法に興味を持つ．ソフトウェア科学会会員．



菰田 敏行（学生会員）

1970 年生．1992 年九州大学工学部情報工学科卒業．現在，同大学院情報工学専攻修士課程に在学し，ソフトウェア工学，特にソフトウェアテスト法に興味を持つ．



古川 善吾（正会員）

1952 年生．1975 年九州大学工学部情報工学科卒業．1977 年同大学院情報工学専攻修士課程修了．同年日立製作所システム開発研究所入社．1986 年九州大学工学部勤務．1992 年同大学情報処理教育センター助教授，現在に至る．工学博士．ソフトウェア工学，特にソフトウェアテスト法，日本語文書出力方式に興味を持つ．電子情報通信学会，ソフトウェア科学会，ACM, IEEE CS 各会員．

牛島 和夫（正会員）

1937 年生．1961 年東京大学工学部応用物理学科（数理工学コース）卒業．1963 年同大学院修士課程修了．同年九州大学中央計数施設勤務．1977 年九州大学工学部情報工学科教授（計算機ソフトウェア講座担当），現在に至る．1990 年 4 月から九州大学大型計算機センター長を兼務．1991 年度情報処理学会九州支部長．工学博士．ソフトウェア科学会，電子情報通信学会，ACM 各会員．