ソフトウエア工学 93−23
（1993．7．9）

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

片山徹郎＊　孤田敏行＊　古川善吾＊＊　牛島和夫＊

＊ 九州大学 工学部 情報工学科
＊＊ 九州大学 情報処理教育センター

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。本論文では、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考察した。タスク型を含んだ並行処理プログラムの同期をとる箇所と、タスク実体の生成個数の決まり方とに着目し、タスク型を含んだ並行処理プログラムを4つに分類した。並行処理プログラムのモデルとして、事象同期グラフ(ESG)を用い、分類したそれぞれに対し、テストケース生成法を考案した。

Testing Method for Concurrent Programs Including Task-Type

Tetsuro Katayama*, Toshiyuki Komoda*, Zengo Furukawa**
and Kazuo Ushijima*

* Department of Computer Science and Communication Engineering,
Kyushu University.
** Educational Center for Information Processing, 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. In this paper, we describe the testing method for concurrent programs including task-type. We classify the programs into four types with the places where tasks in a program synchronize and with how to decide the number of task-substances. We use the Event-Synchs-Graph(ESG) for the model of concurrent programs, and describe the methods for generating test-cases for each of the four types.

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、テストデータを選定する際には、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、人手による場合が多く、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。逐次処理プログラムに対するテストケースの作成技法は、そのソースコードや仕様に基づいた研究がなされている[2]。しかしながら、並行処理プログラムに対しては、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[13][15]。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。特に、タスク型を含む並行処理プログラムに対して、実用的に行なうことのできるテスト法は皆無である。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[1]、信頼性の高い並行処理プログラムを開発するためにテストの質を向上することが重要である。

本論文では、並行処理プログラムのモデルとして、事象同期グラフ(ESG)を用い[6][7][8]、タスク型を含む並行処理プログラムのテストケース生成法について考察する。2章では、これまで提案された並行処理プログラムのモデル化、および、それらのモデルをタスク型を含んだプログラムに適用した際の問題点について述べる。3章では、並行処理プログラムが満たすべきテスト基準について述べる。4章では、タスク型を含んだプログラムの分類を行なう。5章では、4章で分類したそれぞれについて、テストケース生成の方法を与える。6章では、評価および議論を行なう。

2. 並行処理プログラムのモデル化

並行処理プログラムは、実行のタイミングによって、入力データが同じであっても結果が異なる場合がある。並行処理プログラムの動作を明らかにするために、並行処理プログラムのモデル化が必要である。

これまでに、並行処理プログラムのモデルとしていくつか提案されている。

Taylor の並行状態グラフ[15] Taylor らは、並行処理プログラムに対する構造的テスト法の概念を提案した。並行処理プログラムを構成するタスクについて状態を定義し、タスクの状態の組によって並行処理プログラム全体の状態(並行状態)を定義した。各タスクの状態は並行処理に関連した事象である並行事象によって変化するので、1つのタスクの状態の変化に対応して並行状態の遷移を記述した並行状態グラフを作成できる。並行状態グラフの上で被覆率を定義することによってテスト充分性を評価することを提案している。

Tai の同期列[13] Tai は、並行処理プログラムの並行事象の列を並行処理プログラムの同期列として定義した。同期列は、プログラムの動作を並行事象の全順序関係として捉えたものであるので、並行に実行できる並行事象を順序つけてしまう。同期列は、並行処理プログラムの動作を表しているので、同期列によって並行処理プログラムの誤りを定義できる。

事象同期グラフ(ESG)と協調路[6][7][8] 我々は、並行処理プログラムの動作を、各タスク内の並行事象の発生可能性を表す事象グラフと、同期をとる並行事象同士の同期対とによって事象同期グラフとしてモデル化した。このグラフは、基本的には並行処理プログラムのソースコードに基づくものである。さらに、並行処理プログラム全体のテストケースとし

て協調路を定義した。協調路は、タスクについての事象グラフ上の、同期対を満足する路の組であり、並行事象の半順序関係を保持する。

並行処理プログラムには、タスク型を含むものがある(図 1参照)。これは、雛型となるタスクがソースコード中に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、"Taylor の並行状態グラフ" や "Tai の同期列" では、モデル化を行なう前に、生成されるタスクの個数を知っていなければならないので、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。"事象同期グラフと協調路" は、モデルがソースコードに基づいて作られるので、タスク型で書かれたタスクについては、雛型の1つだけが存在するとしてモデルを構成してしまう。そのため、動的に生成された、タスク実体の間で同期をとる(同期対がある)場合は、被テストプログラムの実際の動作を反映できず、テストケースの生成が行なえない場合があった[9][10]。

3. テスト基準

この章では、並行処理プログラムのモデルとして、前章で述べた事象同期グラフと協調路を用い、並行処理プログラムが満たすべきテスト基準について検討する。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。

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、従来以下のものが考えられてきた[3]。

i) 枝被覆基準 — モデル内の全ての枝を少なくとも1回は通る。
ii) 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。
iii) 相互作用被覆基準 — 並行処理プログラムのタスクの間に存在する相互作用(通信や同期)は、テストを行なう際に、少なくとも1回は実現する。

枝被覆基準は、事象同期グラフにおいて、事象グラフ内の全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、ループが存在する時の基準である。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り、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。

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。事象グラフから、上記の条件を満たす路を作り、同期対を満足するようにそれらの路を組み合わせたものが、協調路である。

相互作用被覆基準については、事象同期グラフの同期対を相互作用の動作の対象とする。しかしながら、タスク型を含んだプログラムを事象同期グラフでモデル化すると、動的に生成されたタスク実体間で同期をとる場合には、被テストプログラムの実際の動作を反映できず、この相互作用被覆基準を満足できなくなる。

タスク型を含む並行処理プログラムに対しては新たなテスト手法を考える必要がある。タスク型を含むプログラムのテスト手法を構築するために、次節では、タスク型を含むプログラムの分類について検討する。

4. タスク型を含んだ並行処理プログラムの分類

先に述べたように、事象同期グラフでは、動的に生成されたタスク実体の間で同期をとる場合は、被テストプログラムの実際の動作を反映できないことが分かっている。これを逆に考えると、動的に生成されたタスク実体の間に同期をとる箇所が存在しないプログラムの場合は、事象同期グラフでモデル化が可能である。

タスク型を含んだ並行処理プログラムについて、タスク実体の間での同期のとり方に着目すると、以下の分類が可能である。

【着目点】 同期のとり方
- タスク実体の間で同期をとるもの
- タスク実体の間では同期をとらないもの

```
with TEXT_IO; use TEXT_IO;
procedure PROGRAM is
    package INT_IO is new
INTEGER_IO(INTEGER);
    use INT_IO;
A_n : constant INTEGER := 10;
N_proc : constant INTEGER := 2;
A_max : constant INTEGER := N_proc*A_n;
A1,A2 :array (INTEGER range 1..A_max) of
INTEGER;
task type T_type is
    entry E1(I:INTEGER);
    entry E2;
end;
type T_access is access T_type;
T : array (1..N_proc) of  T_access;
task body T_type is
    T_ord, N_mod, J : INTEGER;
    begin
        accept E1 (I: INTEGER) do
                T_ord := I;
        end;
        N_mod := T_ord-1 mod N_proc;
        for I in 1..A_n loop
                J := I*N_proc-N_mod;
                A1(J):=A2(J);
        end loop;
        accept E2;
    end T_type;

begin
    for I in 1..A_max loop
        A1(I):= -1;
        A2(I):= I;
    end loop;
    for I in 1..N_proc loop
        T(I) := new T_type;
        T(I).E1(I);
    end loop;
    for I in 1..N_proc loop
        T(I).E2;
    end loop;
    for I in 1..A_max loop
        PUT(A1(I));
    end loop;
    NEW_LINE;
end PROGRAM;
```

図 1: サンプルプログラム

“タスク実体の間で同期をとるもの” には、直接実体の間で同期をとる直接的なものと、手続きなどを間にはさんで同期をとる間接的なものとがある。

次に、タスク実体の個数が、どのように実行時に決定されるかについて考えると、以下の分類が可能である。

【着目点】 タスク実体の生成方法

- タスク実体の生成個数が、与えられた入力データに基づき決まるもの
- タスク実体の生成個数がソースコードに明示されているもの

タスク型を含む並行処理プログラムは、同期のとり方、およびタスク実体の生成方法に着目すると上記の分類が可能である。この 2 種類の分類は直交しているので、タスク型を含む並行処理プログラムは以下の 4 種類に分類が可能である。

Type1. タスク実体の間で同期をとらず、実体の生成個数が与えられる入力データに依存するもの。

Type2. タスク実体の間で同期をとらず、実体の生成個数がソースコードに明示されているもの。

Type3. タスク実体の間で同期をとり、実体の生成個数が与えられる入力データに依存するもの。

Type4. タスク実体の間で同期をとり、実体の生成個数がソースコードに明示されているもの。

次節では、4 種類に分類したこのそれぞれの Type について、事象同期グラフを用いたテスト手法を検討する。

5. タスク型を含む並行処理プログラムと事象同期グラフ

タスク型を含んだ並行処理プログラムについて、事象同期グラフを用いてモデル化する。従来の方法では、相互作用被覆基準を満足できない場合があった。モデル化により、プログラム中に存在する全ての同期対をグラフ上に書き込むことができれば、相互作用被覆基準を満足するテストケースが作成できるはずである。

Type1. 「タスク実体の間で同期をとらず、実

体の生成個数が与えられる入力データ に依存するもの」については、タスク実体の間で同期をとることがないので、実体が 1 個生成されるとした事象同期グラフを構成すれば、そのグラフ上に全ての同期対を表すことができる。実際の実体生成個数は入力データによって決定するので、実体が 1 個生成されるようなテストデータを選べば、被テストプログラムの動作を、構成した事象同期グラフ上で表すことができる。

Type2. 「タスク実体の間で同期をとらず、実体の生成個数がソースコードに明示されているもの」については、Type1. と同様、実体が 1 個生成されるとした事象同期グラフを構成すれば、そのグラフ上に全ての同期対を表すことができる。実際の実体生成個数はソースコードに明示されているので、その事象同期グラフ上でテスト基準を満足するように、明示された個数分のテストケースを作れば、被テストプログラムの動作をそのグラフ上で表すことができる。

Type3. 「タスク実体の間で同期をとり、実体の生成個数が与えられる入力データに依存するもの」については、タスク実体の間で同期をとるので、実体が 2 個生成されるとした事象同期グラフを構成すれば、そのグラフ上に全ての同期対を表すことができる。実際の実体生成個数は入力データによって決定するので、実体が 2 個生成されるようなテストデータを選べば、被テストプログラムの動作を、構成した事象同期グラフ上で表すことができる。

Type4. 「タスク実体の間で同期をとり、実体の生成個数がソースコードに明示されているもの」については、Type3. と同様、実体が 2 個生成されるとした事象同期グラフを構成すれば、そのグラフ上に全ての同期対を表すことができる。実際の実体生成個数はソースコードに明示されているので、その事象同期グラフ上でテスト基準を満足するように、明示された個数分のテストケースを作れば、被テストプログラムの動作をそのグラフ上で表すことができる。

分類ごとに、テスト法をまとめると、以下のようになる。

Type1. 実体が 1 個生成されるとした事象同期グラフを構成し、実体が 1 個生成されるテストデータを選定する。

Type2. 実体が 1 個生成されるとした事象同期グラフを構成し、その事象同期グラフ上で テスト基準を満足するよう、実際に生成されるタスクの個数分、テストケースを作成する。

Type3. 実体が 2 個生成されるとした事象同期グラフを構成し、実体が 2 個生成されるテストデータを選定する。

Type4. 実体が 2 個生成されるとした事象同期グラフを構成し、その事象同期グラフ上でテスト基準を満足するよう、実際に生成されるタスクの個数分、テストケースを作成する。

例として、図 1のプログラミング言語 Ada (詳しくは[5] を参照) で書かれた、タスク型を含んだ並行処理プログラムについて考えてみる。これは、タスク型 T_type と、main(手続き PROGRAM) の 2 つの部分からなる、配列 A2 の中身を配列 A1 へコピーするプログラムである。main は、タスク型 T_type の実体を N 個生成し、エントリ E1 で、生成した実体に、ID 番号 I を渡す。タスク型 T_type の実体は、渡された ID 番号 I に従って、配列を N 個おきにコピーしていく。コピーが終了した段階で、エントリ E2 により、main と同期がとられ、タスク型 T_type の実体は終了する。全ての、タスク型 T_type の実体が終了した段階で、main はコピー後の配列の中身を表示してプログラムを終了する。

このプログラムは、タスク実体の間で同期をとることはなく、また実体の生成個数は 2 個とプログラム中に明記してある。すなわち、この並行処理プログラムは先の分類法によると、Type2. に分類される。よってこのプログラムの事象同期グラフは、図 2 のようになる。事象同期グラフとプログラムとの対応関係を表 1 に示す。図において、円は節点、実線は枝、破線は同期対、節点番号 0 番と −1 番はそれぞれ開始節点と終了節点を表している。また、節点番号が連続でないのは、プログラム中の全ての文に節点番号を付け、並行処理に関連しない文を除いて、事象グラフを作成

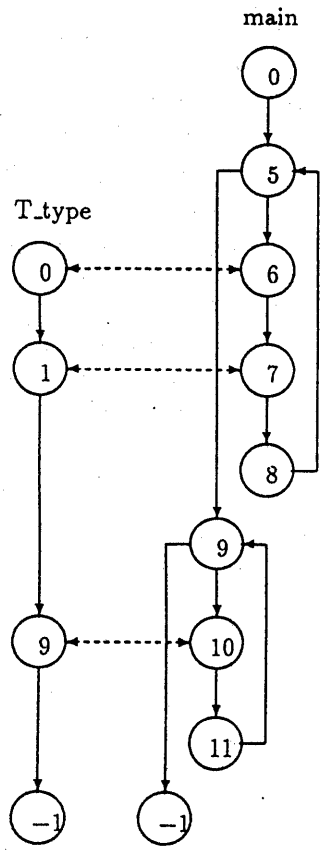

図 2: サンプルプログラムの事象同期グラフ

したためである。

このグラフ上でテスト基準を満足するよう、テストケースを事象グラフ上の路3個1組(タスクT_typeから2個, mainから1個)として作成すればよい。こうして、図3のような協調路ができる。

6. 議論および評価

テスト法が「信頼できる」とは、プログラムに誤りがあるならば、テスト法に従って作成したテストデータによって誤りを必ず発見できることをいう。したがって、「信頼できる」テスト法によって作成したテスト集合(テストデータ)に対し

表 1: サンプルプログラムと事象同期グラフとの対応表

タスク名	節点番号	対応する文(statement)
	0	begin
T_type	1	accept E1 (I) do
	9	accept E2
	−1	end T_type;
	0	begin
	5	for I in 1..N_PROC loop
	6	T(I) := new T_type;
	7	T(I).E1(I);
main	8	end loop;
	9	for I in 1..N_PROC loop
	10	T(I).E2;
	11	end loop;
	−1	end PROGRAM;

てテストの実施結果が全て正当であるならば、プログラムは正当である[14]。しかしながら、任意のプログラムに対して「信頼できる」テスト法は「全数テスト」だけである。

プロセス間の通信に関する不良は、通信における全てのデータに対して誤りを発生する完全通信不良と、一部のデータに対してのみ誤りを発生する部分通信不良に特徴づけることができる[4]。今回述べてきた方法は、プログラムに存在する全ての同期対をグラフに表すことができ、かつ、そのグラフ上で、相互作用被覆基準を満足するよう協調路を作成するので、完全通信不良に対して「信頼できる」と言える。

Taylorらは、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[15]。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、実用的なものとは言いがたい。

事象同期グラフは、今回述べてきた方法を用いると、被テストプログラムがタスク型を含んだ場

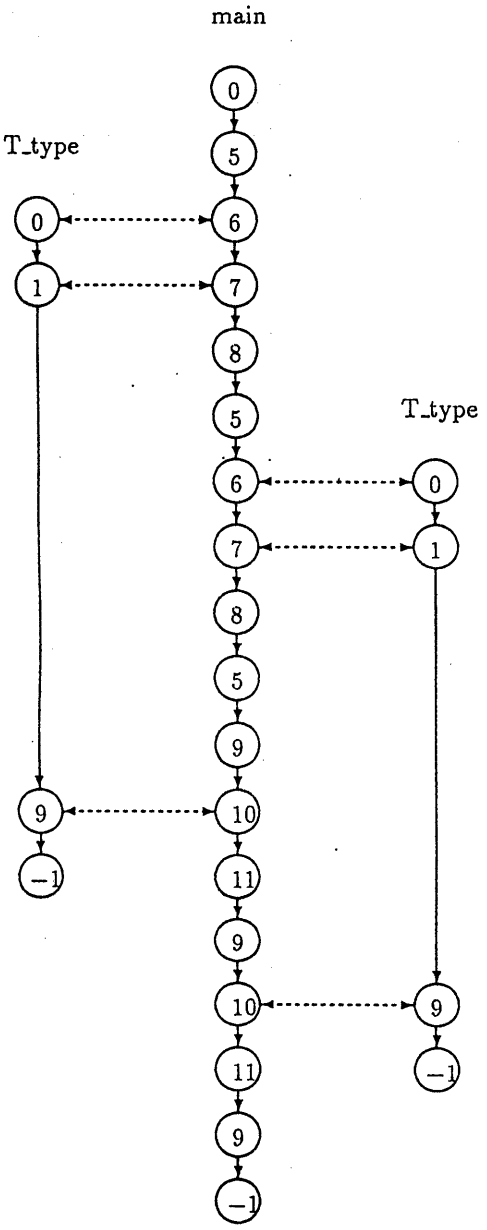

図 3: サンプルプログラムの協調路

合でも、静的にグラフ上での実体個数を決定する。そのため、タスクの実体の個数が増えても、事象同期グラフは図 2で表され、節点数が増えることはないので、実用時間でテストを行なうことができると考えられる。

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[13]。同期列は、並行に実行可能な文の列を全順序づけられた一つの列で表現する。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、各々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は増大しない。

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、プログラムの意味を解釈せず、実行経路から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、今回の方法で得られた協調路の通りに、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存在する保証はない。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。このようなテストケースに対しては、被テストプログラムをテストケース通りに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が考えられる[12]。

7. おわりに

今回は、プログラミング言語 Ada を例にとり、タスク型を含んだ並行処理プログラムのテストケース作成技法について検討した。並行処理プログラムの動作のモデルとして、事象同期グラフ *ESG* を用いた。従来のテスト法は、タスク型を含んだ並行処理プログラムに対して実用的でなかった。今回述べてきた方法で作成されたテスト

ケースは、完全通信不良に対して信頼でき、かつ漏れや重複の減少が期待できる。事象同期グラフは、被テスト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、同期対を満たす路の組であるので、同期対によって規定された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。

- 様々なプログラムへの適用
  今回例として、プログラミング言語 Ada で書かれた並行処理プログラムに適用した。しかしながら、このプログラムは文数が 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ない、検討する必要がある[11]。
- ツール TCgen への適用
  現在我々は、プログラミング言語 Ada で書かれた並行処理プログラムから、協調路を自動生成するツール TCgen の開発を行なっている。しかしながら、現段階の TCgen は、タスク型を含んだプログラムに対して対処できない。TCgen を拡張して、今回述べてきた方法を用い、タスク型を含んだプログラムからも自動的に協調路を取り出す必要がある。
- 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テストケース通りに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が考えられる[12]。

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 (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、伊東栄典君、川口豊君に感謝します。

参考文献

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

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

[3] 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.

[4] 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.288-297, 1992.

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

[6] 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.

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

[8] 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," 情報処理学会研究報告, Vol.92, No.59, pp.9-16 (1992).

[9] 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 9 回大会論文集, pp.49-52 (1992).

[10] 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).

[11] 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," 情報処理学会第 45 回全国大会, Vol.5, pp.253-254 (1992).

[12] 菰田敏行, 片山徹郎, 古川善吾, 牛島和夫: "Ada 並行処理プログラムのテストケース作成とその強制実行に関する一考察," 第 20 回 JAPAN SIGAda 予稿集, pp.9-15 (1993).

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

[14] 玉井哲雄, 三嶋良武, 松田茂広: "ソフトウェアのテスト技法," 共立出版, 1988.

[15] 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, Mar. 1992.