ソフトウェア工学 96−14
（1994. 1. 21）

順序列テスト基準を用いたテスト充分性評価システムの試作

川口豊† 伊東栄典† 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

並行処理プログラムが普及するに伴い、信頼性向上手法としてのテストが重要となってきている。並行処理プログラムは逐次処理プログラムに対して動作が複雑であるので、新しいテスト法が必要である。

現在一般的に使用されているプログラミング言語としてC言語がある。C言語で並行処理を実現するには、オペレーティングシステムで準備されたシステムコールを利用する。そこで、我々はC並行処理プログラムのテスト基準として、オペレーティングシステムで準備されている通信、同期方法に対して順序対や順序列に着目したテスト基準を提案してきた。また、我々はそのテスト基準に基づくテスト充分性評価システムを試作している。

本稿では、我々が試作したテスト充分性評価システムについて考察する。このシステムの入力はCのソースコードであり、出力はテスト充分性評価の指標である被覆率である。このシステムでは、順序列の実行を検出するためにモニタプロセスを使用する。このシステムでは、テストすべき事象として実行不可能な事象を取り込むことがある。

On constructing testing reliability evaluation system with ordered sequence testing criteria

Yutaka Kawaguchi†, Eisuke Itou†, Zengo Furukawa‡
and Kazuo Ushijima†

†Department of Computer Science and Communication Engineering,
‡Educational Center for Information Processing,
Kyushu University.

Testing concurrent programs is important to increase reliability of the program. Because the behavior of concurrent programs is more complex than that of sequential programs, new testing criteria are necessary for concurrent programs.

Recently, many programs are written in C language. We use system calls of an operating system to do concurrent processing in C language. We have proposed the criteria based on ordered pairs or ordered sequences of communication commands prepared in the operating system as testing criteria for C concurrent programs. And we have implemented a prototype of testing reliability evaluation system with the criteria.

In this paper, we describe the system. The input of this system is C source code, and the output of this system is the testing coverage. This system detects the execution of ordered pairs using "monitor process". This system may hold an unexecutable ordered pair as an object which should be tested.

1. はじめに

近年、プロセッサの高性能化、低価格化により、マルチプロセッサシステムや、計算機の分散環境が普及してきた。それに伴い並行処理プログラムの普及が進んだ。並行処理プログラムの普及に伴い、その信頼性向上が重要な課題となってきている。

従来、逐次処理プログラムの信頼性向上手法としてソフトウェアテストが研究されてきた。並行処理プログラムは逐次処理プログラムに比べ動作が複雑である。そのため逐次処理プログラムのテスト法は、並行処理プログラムのテストに用いるには不十分である。並行処理プログラムに対しては、その特性を考慮したテスト法が必要である。

並行処理プログラムが逐次処理プログラムに比べ動作が複雑であるのは、並行処理プログラムがプロセス間通信、同期によって異なる動作を行なうためである。並行処理プログラムに対するテスト法には、プロセス間通信、同期の実行検出が必要となる。

ソフトウェアのテストにおいて、テスト充分性評価はテストケースやテストデータの信頼性の評価やテスト終了判定に用いることができる。しかし、テスト充分性評価を人手で行なうと、テストされた事象の見落としが生じて、評価の信頼性が低くなる。さらに、プロセス間通信・同期命令の実行を人手で検出するのは困難である。そのため、並行処理プログラムのテスト充分性評価を行なうためには、テスト充分性評価システムが必要となる。

我々は、C 言語で記述された並行処理プログラムのテストのために、プロセス間通信・同期命令の順序列に着目したテスト基準を提案し[1]、そのテスト基準に基づくテスト充分性評価システムを試作した[2][3]。本稿では、そのテスト充分性評価システムについて議論する。

2. テスト基準

2.1 被覆率

テスト充分性評価は、テスト基準に基づいて、プログラムから実行すべき測定事象の集合を作り、その測定事象がどの程度テストで実行されたかを調べることである。

テスト充分性評価を行なうに当たって、その指標となる値が必要となる。テスト充分性評価の指標として**被覆率**という値を用いる。被覆率 C の定義は以下の通りである。

[定義 1] 被覆率

$$C = \frac{|W|}{|M|}$$

M : プログラムに含まれる測定事象集合
W : 実行された事象の集合
$|\cdot|$: 集合の要素数

逐次処理プログラムのテストにおいては、プログラム内の全ての文を測定事象とする C_0 被覆率や、プログラム内の全ての制御の移行を測定事象とする C_1(分岐) 被覆率を用いたテスト充分性評価が実用化されている。C_0 や C_1 被覆率を 100% にするというテスト基準は、被テストプログラムの正しさを保証するものではない。しかし、これらの被覆率は、テスト充分性を定量的に表す指標の一つとして用いられている。

定義 1 の被覆率は、測定事象の作成と実行時に発生した事象の記録ができれば定量化が可能であるので、並行処理プログラムのテスト充分性評価にも利用できる。逐次処理プログラムにおいてと同様、テスト基準としては被覆率 100% にすることを考える。

被覆率を用いた並行処理プログラムのテスト充分性評価技法として、Taylor らによる並行状態グラフを用いたもの[4] や、Tai らによる同期列を用いたもの[5] がこれまでに提案されている。しかし、これらの技法には、(1) テストの測定事象がプロセス数に対して組合せ論的に増大する、(2) 並行状態グラフや同期列を定義するためにはプロセス数が静的に決定していなければならない、という問題点がある。

2.2 順序列テスト基準

[1]

並行処理プログラムでは、通信や同期によってプログラムは異なる動作を行なう。プロセス間の通信・同期は、二つの通信・同期命令の間で実行される。したがって、並行処理プログラムのテスト基準として、プログラム中の通信・同期命令の順序対を全て実行する、という基準が考えられる。

[定義 2] 順序対テスト基準

$$\mathrm{OSC}_2 = \{< s_1, s_2 >\}$$

s ・・・ 通信・同期命令
$< a, b >$ ・・・ 順序対
$\{\cdot\}$ ・・・ 集合

また、順序対テスト基準を一般化して、通信・同期命令の長さ k ($k > 1$) の順序列を全て実行する、というテスト基準が考えられる。

[定義 3] 順序列テスト基準

$$\mathrm{OSC}_k = \{< s_1,\ s_2,\ \cdots, s_k >\}$$

s ・・・ 通信・同期命令
$< \cdot >$ ・・・ 順序列
$\{\cdot\}$ ・・・ 集合

テストが OSC_k テスト基準を満たす場合の発見能力について伊東らが分析している[1]。それによると、順序列の長さ k を十分大きくとることによって、通信、同期による誤りを多く検出することができる。しかし、被覆率計算に要する計算の複雑さは、プログラム中の通信・同期命令数を m とすると、$O\left(m^k\right)$ となる。そのため長さ k 順序列テスト基準は、k が大きくなると、実用的ではなくなる。

我々が試作したシステムでは、長さ 2 の順序列 (すなわち順序対) テスト基準を採用した。順序対テスト基準に基づくテストでは、任意のテストデータに対して必ず誤りを生じるという**完全通信誤り**を全て検出することができる。

2.3 C 言語に関するテスト基準

C 言語は、言語仕様として並行処理を記述する能力を持たない。C 言語を用いて並行処理を実現する際には、オペレーティングシステムのシステムコールを用いる。プロセス間の通信・同期はシステムコール呼び出しの形で実装する。したがって、C 並行処理プログラムに対するテスト充分性評価システムを実装する際には、オペレーティングシステムで準備されている通信・同期方法ごとに対応させていかねばならない。

UNIX SYSTEM V上に実装されているセマフォやUNIX 4.3BSD上に実装されているソケットに対して、順序列テスト基準が具体的にどのように定義されるか、以下の小節で述べる。

2.3.1 セマフォに対する順序対テスト基準

セマフォに対しては、P命令、V命令という2つの操作が許されている。P命令はセマフォの値が正であるかどうかで、V命令は待機中のプロセスが存在するかどうかで動作が異なる。テストを行なう場合には、この動作の違いを区別しなければならない。そこで、セマフォに対するテストのためにセマフォ命令を以下のように定義する。定義中、*semop* はUNIX SYSTEM VでP命令、V命令を実現するシステムコール、**semid** はどのセマフォに対する操作を行なうかを示すセマフォ識別子である。

[定義 4] セマフォ命令 **semcom** は次の2つ組である。

$$\mathbf{semcom(semid)} = (semop(\mathbf{semid}), b)$$

$$b = \begin{cases} 1 & \cdots \begin{cases} \text{セマフォ } \mathbf{semid} = 0, \text{または} \\ \text{プロセスキューが空でない} \end{cases} \\ 0 & \cdots \begin{cases} \text{セマフォ } \mathbf{semid} > 0, \text{または} \\ \text{プロセスキューが空} \end{cases} \end{cases}$$

セマフォに対するテストでは、セマフォ命令の全順序対や全順序列が実行されたかどうかをテスト基準として用いる。

[定義 5] セマフォ命令順序対テスト基準 Sem_2

以下に示す集合 M を測定事象集合とするテスト基準をセマフォ命令順序対テスト基準という。

$$M = \bigcup_{i,j,semid} \{< semop_i(semid), semop_j(semid) >\}.$$

ただし、$semid$ はセマフォ識別子、$< a, b >$ は順序対を表す。

2.3.2 ソケットに対するテスト基準

4.2BSDがリリースされるまで、UNIXの標準的な通信機能はパイプだけであった。パイプはフロー制御されたバイトストリームであり、同じ計算機上の二つの関連するプロセス間のみで開設できるものであった。計算機の低価格化、計算機台数の増加、分散環境での使用に対する要求などの要因により、パイプの制約を越えたプロセス間通信機能としてソケット(Socket)が開発された。ソケットは、UNIXのファイルアクセス機構の通信のための端点を提供する一般化と考えることができる。

ソケットの使用方法は、まずシステムコール *socket* でソケットを生成し、システムコール *bind* で局所アドレスを指定する。この後通信を行なう。通信方法には、通信先と結合して通信を行うコネクション型と、通信先と結合しないノンコネクション型がある。コネクション型においてデータを送信するシステムコールは *send*, *write*, *writev*、データを受信するシステムコールは *recv*, *read*, *readv* である。ノンコネクション型でデータを送信するシステムコールは *sendto*、データを受信するシステムコールは *recvfrom* である。データの送受信が終了した際には、システムコール *close* を用いてソケットを終了させる。

ソケットに対しては、プログラムに含まれる**通信命令**という命令の順序対、順序列が実行されたかどうかをテスト基準とする。通信命令の定義を以下に示す。

[定義 6] 通信命令

送信を行なうシステムコール *write,writev,send,sendto,sendmsg*、受信を行なうシステムコール *read,readv,recv,recvfrom,recvmsg* のいずれかを含む文が通信命令である。

[定義 7] ソケットテスト基準

以下に示す集合 M を測定事象集合とするテスト基準をソケットテスト基準とする。

$$M = \bigcup_{i,j} \{< sockop_i, sockop_j >\}.$$

ただし、$sockop_i, sockop_j$ は通信命令、$< a, b >$ は順序対を表す。

3. 被覆率測定系

この節では、我々が試作したテスト充分性評価システム[3]について述べる。我々が試作したテスト充分性評価システムは、セマフォ命令順序対テスト基準、ソケットテスト基準に基づく被覆率を計算する被覆率測定系である。

3.1 概要

測定系の概要を、図1に示す。

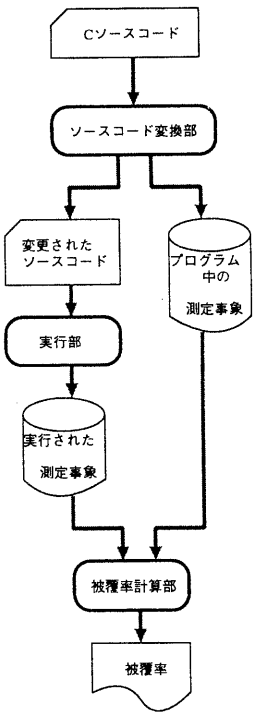

図 1: 被覆率測定系の概要

測定系の入力はCのソースコードであり、出力は被覆率の値である。ソースコードからテストすべき測定対象を検出する。ま

```
semop(exclude,&p_b,1);
*buf = ch;
semop(exclude,&v_b,1);
```

↓

```
comm_with_monitor("producer",13,2,"exclude",
    exclude,p_b.sem_num,p_b.sem_op);
semop(exclude,&p_b,1);  /* line 13 */
*buf = ch;  /* line 14 */
comm_with_monitor("producer",15,2,"exclude",
    exclude,v_b.sem_num,v_b.sem_op);
semop(exclude,&v_b,1);  /* line 15 */
```

図 2: ソースコードの変換

た、テスト中に実際に実行された測定対象を検出するためにソースコードを変換する。

測定系は、ソースコード変換部、実行部、被覆率計算部で構成される。以下で、各部について説明する。

3.1.1 ソースコード変換部

ソースコード変換部は、ソースコードを読み込み、通信・同期命令を抽出、記録する。同時に、ソースコード変換部は、通信・同期命令が実行されたことを検出するため、ソースコードを変換する。ソースコード変換部は、プログラム中の測定対象の集合と、変換されたソースコードを出力する。

ソースコード変換部は、yacc[6] のソースコードとして記述された C の文法規則に通信・同期命令の抽出と探針の挿入を行なうアクションを加えることによって実装した。ソースコード中で通信を行なうシステムコールを呼び出す箇所を見つけた際に、ソースコード変換部はそのシステムコールについての記録をとる。ソースコード変換部は同時にそのシステムコールに対応した探針を埋め込む。

3.1.2 実行部

実行部は、プログラムを実行して、実行された通信・同期命令を検出する。実行部は、テスト中に実際に実行された測定対象(通信・同期命令)の記録を出力する。本システムは、通信・同期命令が実行されたことを検出するために、モニタという別プロセスを用いる。モニタについては、次節で詳説する。

実行部での動作の流れは次の通りである。

- モニタプロセスを起動する
- プログラム本体の実行を開始する
- あるプロセスが通信・同期命令が存在する箇所にさしかかった場合モニタプロセスと通信を行なう
- 許可を受けたプロセスは通信・同期命令を実行する
- (ソケットの場合) 通信命令を実行したプロセスは、モニタプロセスに通信命令の実行が終了したことを知らせる
- モニタプロセスを終了させる
- プログラム終了

3.1.3 被覆率計算部

被覆率計算部は、ソースコード変換部で抽出、記録した通信・同期命令と実行部で検出した通信・同期命令のデータとから、被覆率を計算する。

被覆率計算部は、まずプログラムに含まれる通信・同期命令のデータからプログラムに含まれる通信・同期命令の順序対集合テーブルを作る。その後、実行された通信・同期命令のデータから実際に実行された順序対をチェックし、被覆率を計算する。

3.2 モニタ

3.2.1 モニタの概要

テスト実施の際、システムは通信・同期命令が実行されたことを検出しなければならない。しかし、各通信・同期命令の前に記録を行なう部分を挿入する方法を用いた場合、記録されたデータの順番と実際に実行された通信・同期命令との間の順番とが食い違う可能性がある。

そこで、通信・同期命令が実行されたことを検出するために、本システムではモニタプロセスを使用する。図 3に、モニタの動作の流れを示す。

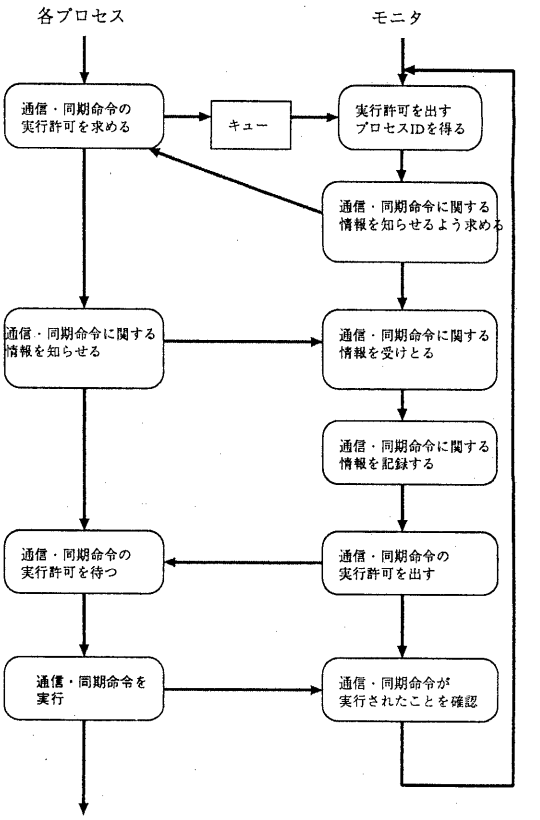

図 3: モニタの動作の流れ

モニタは、ある通信・同期命令の実行中、被テストプログラム

中に存在する他の通信・同期命令は実行されないことを保証する。

3.2.2 モニタの機能

モニタは、各プロセスの通信・同期命令の実行を制御することにより、通信・同期命令の実行を検出し、実行された通信・同期命令についての記録を行なう。

モニタの動作には、次の5つの段階がある。

i) プロセスから通信・同期命令の実行許可を受けとる
ソースコード変換部で、本システムは通信・同期命令の前にモニタと通信を行なうための探針コードを挿入しておく。通信・同期命令を実行しようとするプロセスは、通信・同期命令を実行する前に、モニタに対して通信・同期命令の実行許可を求め、実行許可が出るまで待機する。
各プロセスからの実行許可願いはキューに蓄えられる。モニタは、キューから最も古い実行許可願いを取り出す。

ii) どの通信・同期命令を実行するかを知らせる
モニタは、共有メモリを使って、どの通信・同期命令を実行するかを各プロセスに知らせる。

iii) 実行する通信・同期命令に関するデータを受けとる
通信・同期命令の実行許可を受けたプロセスは、これから実行しようとしている通信・同期命令を特定するデータを、モニタに送る。セマフォの場合、セマフォ命令が実行されたことを確認するために、システムコール *semop* の第一引数、第二引数の値が必要となる。通信・同期命令の実行許可を受けたプロセスは、これらの値もモニタに送る。
モニタは、データを受けとった後、実行される通信・同期命令に関するデータを記録する。

iv) 通信・同期命令の実行許可を出す

v) 通信・同期命令が実行されたことを確認する
通信・同期命令の実行許可を出した後、モニタがすぐに別の通信・同期命令の実行許可を受け付ける作業に移ると、実行された通信・同期命令の実行順番と記録された順番とが一致しない可能性がある。そのため、通信・同期命令の実行許可を出した後、モニタは実行許可を出した通信・同期命令が実行されたことを確認する必要がある。
ソケットの場合、通信命令が実行されたことを確認するために、モニタに通信命令が実行されたことを知らせるコードを、通信命令の後に挿入しておく。セマフォの場合、ソケットと同じ方法を用いるとデッドロックを起こす可能性が生じる。そこで、モニタは、セマフォの値や待機中のプロセス数の変化からセマフォ命令が実行されたことを確認する。

4. 実行例

4.1 実行結果

本システムで producer-consumer 問題のプログラムを実行した際の被覆率の値を表1に示す。また、producer-consumer 問題プログラムの動作の流れを図4に示す。このプログラムは、3つのセマフォ `pro`, `con`, `exclude` を使用している。このプログラム中でのシステムコール *semop* 呼び出しは、セマフォ `pro`、`con` に対しそれぞれ2箇所、セマフォ `exclude` に対し4箇所の、合計8箇所である。したがって、測定事象であるセマフォ命令順序対数は、セマフォ `pro`,`con` に対しそれぞれ $(2 \times 2)^2 = 16$、セマフォ `exclude` に対し $(4 \times 2)^2 = 64$ の、合計96である。

表 1: producer-consumer 問題のプログラムを実行した際の結果

実行されたセマフォ命令数	被覆率 (%)
15	9.4
39	10.4
215	10.4
2229	14.6
8790	15.6
25176	15.6

表 2: 実行時間の比較

入力データのバイト数	元のプログラムの実行時間	システム上での実行時間	セマフォ命令数
40	0.0†	7.6	322
365	0.3	57.6	2922
1208	0.7	208.6	9666

(時間の単位は10ミリ秒)
†...10ミリ秒以下の実行時間。

実行結果から、実行されたセマフォ命令数が8800から25000に変化しても、被覆率は全く増加していないことがわかる。この間実行されたセマフォ命令順序対は全て、すでに実行されたセマフォ命令順序対である。被覆率が増加しない理由として、次の2つが考えられる。

- 他に実行可能な順序対が存在するけれども決まった順序対しか実行していない
- 測定事象集合の中に、実行不可能な順序対が含まれている

図5に、表2に示す3つのケースに対する実行された順序対を示す。表2に示された3つのケースでは、実行された順序対はどれも同じであった。効率よくテストを行うためには、より多くの種類の順序対が実行されるように、テストデータ生成系、テストケース生成系と連携する必要がある。

しかしながら、測定事象集合に実行不可能な順序対が含まれている場合、被覆率を100%にすることが不可能であるため、テストをいつ打ち切るかの問題が生じる。

4.2 実行時間

表2に、先に出てきた producer-consumer 問題のプログラムをそのまま実行した場合の実行時間と、テスト充分性評価システム上で実行した場合の実行時間を示す。どちらの場合も、与えた入力は同じである。実行は SunSPARCStation10 上で行なった。時間計測は、UNIX の time コマンドで行なった。

この結果を見ると、本システム上で producer-consumer 問題プログラムを実行した場合、このプログラムをそのまま実行した場合よりかなり実行時間が多くかかることがわかる。

本システム上で被テストプログラムを実行する際、各通信・同期命令を実行する前にモニタを呼び出す。モニタは、各通信・同期命令が実行される前にその通信・同期命令に関するデータを記録する。実行時間の差はこのファイルアクセスによるところが大

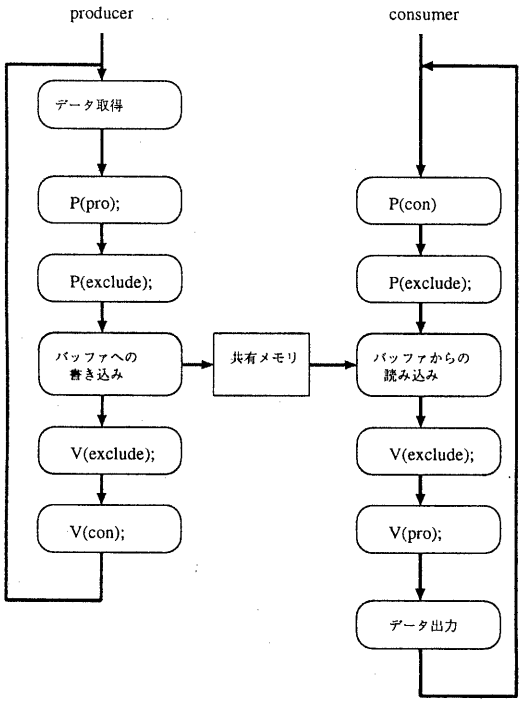

図 4: producer-comsumer 問題のプログラム

$< P'_{consumer}(con), V'_{producer}(com) >$
$< P_{consumer}(exclude), V_{consumer}(exclude) >$
$< V_{consumer}(exclude), P_{producer}(exclude) >$
$< V'_{consumer}(pro), V_{consumer}(pro) >$
$< V'_{consumer}(pro), P'_{producer}(pro) >$
$< P_{producer}(pro), P'_{producer}(pro) >$
$< P'_{producer}(pro), V'_{consumer}(pro) >$
$< P_{producer}(exclude), V_{producer}(exclude) >$
$< V_{producer}(exclude), P_{consumer}(exclude) >$
$< V'_{producer}(con), P'_{consumer}(con) >$

$P(id)$ セマフォ id に対する P 命令 (セマフォ id の値を 1 減らす)
$P'(id)$ セマフォ id に対する P 命令 (セマフォの値が正になるまで待機する)
$V(id)$ セマフォ id に対する V 命令 (セマフォ id の値を 1 増やす)
$V'(id)$ セマフォ id に対する V 命令 (待機中のプロセスの実行を再開させる)

図 5: 実行された順序対

```
P(semid)
int semid;
{
    ...;
    p_buf.sem_op=-1;
    semop(semid,&p_buf,1);
    ...;
}

foo()
{
    ...;
    P(sem);
    ...;
    V(sem);
    ...;
}
```

図 6: P 命令、V 命令を別関数として準備している場合

きいと考えうる。

5. 議論

5.1 システムの実現について

5.1.1 関数で実装されている P,V 関数

プログラムによっては、エラー処理を行なうためや引数に与える値を統一するために、システムコール呼び出しを関数経由で行なうものがある。

例えば、セマフォを使用しているプログラムの中には、引数に与える値の指定が大変繁雑であるため、P 命令、V 命令に相当する関数を作っておいて、実際にセマフォへの操作を行なう際にはそれらの関数を呼び出す形にしているものがある。

本システムは、ソースコード変換部で特定のシステムコール呼び出しを行なっている箇所を探す。本システムは、システムコール呼び出しを行なっている箇所を発見したら、測定事象として記録し、同時にそのシステムコールに応じた探針を挿入する。しかし、プログラムによっては図 6の形で P 関数や V 関数を実現している場合がある。図 6のプログラムは、関数 foo 内でセマフォ sem を直接システムコール *semop* で操作するかわりに、別に準備してある P 関数や V 関数経由でセマフォの操作を行なっている。このようなプログラムに対してテストを行なう場合、プログラム中でその P 関数や V 関数をセマフォ命令として取り込む必要がある。

現在のシステムは、図 6の形のプログラムに対しては、関数 P や関数 V 内で呼び出している *semop* システムコールを抽出し、そのシステムコール呼び出しの前に探針を挿入する。しかし、テストの趣旨からいってこの場合は P 関数や V 関数呼び出しを通信・同期命令として抽出し、P 関数や V 関数の呼び出し前に探針を挿入する方が望ましい。

図 6の形のプログラムで P 関数、V 関数を測定事象として抽出するためには、ソースコード部で抽出すべきシステムコールを記述している配列に P 関数、V 関数の関数名を加える必要があ

る。さらに、セマフォに関する関数の場合は、モニタにセマフォに関する情報を送らなければならないので、探針の挿入を行なう部分も変更しなければならない。

5.1.2 条件式内にシステムコール呼び出しがある場合

ソケットの場合、システムコール呼び出しの後にシステムコールの実行が終了したことをモニタに知らせるためのコードを埋め込む必要がある。しかし、C言語の場合、条件式内で値の代入が可能である。例えば、図7のようなコードが存在する。

```
if( (tcounter = read(sock,&buff[counter],cnum))
    == -1) {
      perror("read  ");
      exit(1);
}
```

図 7: 条件式内でシステムコール

まずはif文の条件式内に通信・同期命令が現れた場合について考察する。if文の条件式内で通信・同期命令が現れ、そのif文がelse節を持っている場合は本システムは、then節とelse節両方の最初にコードを埋め込む。もしelse節を持っていない場合、本システムはthen節の最初にコードを埋め込み、さらにelse節を追加してコードを埋め込む。

条件式をもつ文には、if文の他にfor文やwhile文といった繰り返し文がある。繰り返し文の条件式内に通信・同期命令が現れた場合、図8に示すように、繰り返し文の直前、繰り返して実行されるブロックの先頭、ブロックの最後、繰り返し文の直後に探針を挿入すれば良い。

また、return文の戻り値を指定するところに通信・同期命令が現れる可能性もある。この場合、通信・同期命令の実行後に実行すべきコードをreturn文の後ろに挿入しても、そのコードは実行されない。return文で、システムコールの戻り値をそのまま返す場合、通信・同期命令の実行後に実行すべきコードをどこに埋め込むか、今後検討しなければならない。

5.1.3 実行時間の増大

被テストプログラムに対して探針のコードを挿入するため、被テストプログラムをそのまま実行した場合と比べると、本システム上で実行した場合には探針コードの分だけ実行時間が増大する。前節での実行例を見ても、producer-consumer問題プログラムをそのまま実行した場合に比べ、本システム上での実行では多くの時間がかかっている。

```
...;
probe_1(..); /* 通信・同期命令実行前に実行すべき探針 */
while(通信・同期命令が入った条件式) {
  probe_2(..);
    /* .. 通信・同期命令実行後に実行すべき探針 */
  ...;
  probe_1(..);
}
probe_2(..);
...;
```

図 8: 繰り返し文の条件式内に通信・同期命令が現れた場合

```
...;
probe_1(..);
return (通信・同期命令が現れる式);
probe_2(..); /* この文は実行されない */
...;
```

図 9: return 文中に現れる測定事象

本システム上で被テストプログラムを実行する際には、プログラム内の通信・同期命令の実行が遅延されたと考えられる。通常の並行処理プログラムでは通信、同期命令の実行タイミングが決まっているわけではない。モニタが使用するキューが溢れたといったエラーが生じない限り、通信・同期命令はいつか実行される。ただ、実時間処理を行なうプログラムでは、通信・同期命令の実行が遅延されることによるテストへの影響が生じる可能性がある。そこで、実時間処理プログラムにおける実行時間の遅延の影響および対策については今後検討する必要がある。

5.2 テスト充分性評価方法について

第2節で定義したテスト基準にしたがってプログラムから測定事象の集合を作った場合、プログラムの仕様通りの動作において実行不可能な要素を測定事象として取り込んでしまう可能性がある。例えば、前節のproducer-consumer問題プログラムにおいて、セマフォexcludeに着目する。このプログラムが正しく動作する場合、producerプロセスにおいても、consumerプロセスにおいても、セマフォexcludeに対する各々のP命令、V命令が実行された後に、同じP命令、V命令が実行されることはない。しかし、セマフォ命令順序対テスト基準を満たすためには、同じセマフォ命令の順序対が実行されなければならない。したがって、このproducer-consumer問題プログラムでは、セマフォ命令順序対テスト基準のもとで被覆率を100%にすることは不可能である。

測定事象集合から実行不可能な順序対を除くためには、生成される全てのプロセスについて動作の流れを調べ、どの順序対が実行不可能となるか判定しなければならない。プロセス生成は動的に起こり得るので、このようにして実行不可能な順序対を除くのは困難である。

ただ、プログラムの仕様通りの動作において実行不可能である順序対が、プログラムの誤りや、仕様で想定されていない条件の元で実行させた場合に実行できる場合がある。例えば、先ほどのproducer-consumer問題プログラムはproducerプロセスやconsumerプロセスをそれぞれ一つずつ生成する。そのためproducerプロセスに現れるP(exclude)、V(exclude)に対して、順序対< P(exclude),P(exclude) >など同じセマフォ命令の順序対は実行不可能であった。しかし、producerプロセスやconsumerプロセスが同じコードから複数生成されたならば、producerプロセス内のあるコードP(exclude)について、< P(exclude),P(exclude) >という順序対が実行される可能性がある。

そこで、プログラムが仕様通りに動作する場合には実行不可能となる順序対を測定事象から除くことは妥当ではない。その場

合、実行不可能な順序対は、強制的に実行させる。強制実行の方法については今後検討しなければならない。

5.3 テスト方法および支援環境について

本システムは、ソースコードに探針を挿入する方式を用いている。そのため元々の被テストプログラムと、探針を挿入した被テストプログラムとでは、動作が異なる。

本システムは、被テストプログラムの実行中に通信・同期命令が実行されたことを検出するために探針を用いる。探針、モニタ使用により、被テストプログラムの本システム上での動作と、被テストプログラムの元々の動作との間に次のような違いが生じる。

- モニタプロセスの生成
  モニタプロセスが生成されることにより、メモリ使用量、同時に動作しているプロセス数が変化する。モニタ生成の段階でエラーが生じてモニタプロセスが生じなかった場合、計測データが出力されないので、本システムが誤った被覆率の値を出力することはない。
- モニタによる、通信・同期命令の逐次化
  モニタプロセスは、通信・同期命令を実行許可受け付け順に実行する。ある通信・同期命令の実行中、モニタは他の通信・同期命令を待機させるため、通信・同期命令が逐次的に実行される。そのため被テストプログラムの並列度が減少してしまう。
  しかし、本システムは通信・同期命令の順序対の集合を測定事象としている。テスト実施の際、探針、モニタ使用により二つの通信・同期命令の実行開始時間はずれるけれども、この通信・同期命令の実行順序関係のもとでテスト中に誤りが現れるならば、元のプログラムでもこの通信・同期命令の実行順序では誤りを起こす。したがって、探針、モニタ使用による被テストプログラムの動作への影響は、通信、同期命令順序対テスト基準に基づくテストにおいては無視できる。
- 実行時間の増大 5.1.3 節で述べたように、変更前のプログラムに比較し実行時間が増大するので、実時間処理プログラムでは対策が必要である。

6. おわりに

本稿では、順序列テスト基準を用いたテスト充分性評価システムの構成、動作の仕組みについて説明した。さらに、本システムの実装におけるいくつかの問題について議論した。

現在、セマフォを使ったC並行処理プログラムに対しては、本稿のテスト充分性評価システムを使って、セマフォ命令の順序対に基づく被覆率を求めることができる。しかし、これまでに提案されてきたテスト基準では被覆率を100%にすることができない場合がある。

今後の課題として、被覆率の値の変化とテスト充分性判定の関係を、多くのC並行処理プログラムに対する本システムの使用によって調べることがある。通信を行なうCプログラムでは、ソケットを使用しているものが多い。現在行なっている本システムのソケットへの対応が完成すれば、より多くの被テストプログラムを本システムにかけることができる。さらに、本システムをパイプ、共有メモリ、メッセージキューなど他のプロセス間通信、同期方法に対応させていくことも将来の課題としてあげられる。

謝辞

日頃から議論いただいている九州大学工学部情報工学科程京徳助教授の御助言、御批判に感謝致します。

参考文献

[1] 伊東栄典、川口豊、古川善吾、牛島和夫:C並行処理プログラムのプロセス間通信に関するテスト充分性評価について、情報処理学会研究会報告 Vol.93, No.13, 93-SE-90, pp.9-16, 1993.

[2] 川口豊: C並行処理プログラムのテスト充分性評価システムの試作, 九州大学卒業論文, 1993.

[3] 川口豊、伊東栄典、古川善吾、牛島和夫:C並行処理プログラムのテスト充分性評価システムの試作、第47回情報処理学会全国大会論文集, pp.5-177-178, 1993.

[4] Taylor,R.N., Kelly,C.D. : Structural testing of Concurrent Programs,Proc.of Workshop on Software Testing, pp.164-169, 1986.

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

[6] 五月女健治 : yacc/lex プログラムジェネレータ on UNIX, 啓学出版, 1992.

[7] 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, 日本ソフトウエア科学会第6大会論文集,pp185-188,1989.