ソフトウェア工学 90－2
（1993. 2. 4）

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

伊東栄典† 川口豊† 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、プログラムの信頼性向上手法としてのテストが重要になっている。逐次処理プログラムに比べ並行処理プログラムは動作が複雑であるため、新しいテスト法が必要である。

現在一般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。このC言語には元来並行処理を記述する能力はない。このため、C言語で書かれたプログラムはオペレーティングシステムとの連携により、並行処理を実現している。C言語で書かれたプログラムにおいて、オペレーティングシステムとの連携は、ライブラリとして用意されているシステムコールの呼び出しにより行なわれる。したがって、オペレーティングシステムと連携したプログラムに対しては、言語だけでなく、ライブラリに対応したテスト法と支援システムが必要になる。

本稿では、C言語、UNIXオペレーティングシステムで記述された並行処理プログラムに対するテスト充分性評価の可能性を明らかにするために、UNIXに備えられているプロセス間通信機能の中で、SYSTEM V系のオペレーティングシステムに標準的に備わっているセマフォ、および4.3BSD系に備わっているソケットに対するテスト充分性評価法を考察する。

Testing Criteria for Interprocess Communication in C Concurrent Programs

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

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

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

Testing of concurrent programs is important to increase reliablity of programs. Because the behavior of a concurrent program is more complex than that of a sequencial program, new testing criteria must be introduced for concurrent programs.

Recentry, many programs are written in C languege. It is impossible to discribe concurrent processes in C language alone. Therefore, concurrent processes are realized in cooperation with the operating system. This cooperation is implemented through system calls library. Thus, testing criteria and testing support system should depend not only on the C program but also on the library.

To evaluate testing reliablity of C programs and UNIX operating system, we discuss testing criteria for a semaphore supported in UNIX SYSTEM V and a socket supported in UNIX 4.3BSD.

1. はじめに

並行処理プログラムが普及するに伴い、プログラムの信頼性向上手法としてのテストが重要になっている。逐次処理プログラムに対しては、様々なテスト方法が考案され実用化されている。逐次処理プログラムに比べ並行処理プログラムは動作が複雑である。並行処理プログラムでは、同じ入力データに対し、出力が毎回同じであるとは限らない。このような、非決定的な振舞いが並行処理プログラムの複雑さの原因である。動作が複雑であるので逐次処理プログラムのテスト法を並行処理プログラムに適用しても、それだけでテストが充分であるとは言えない。並行処理とは、複数の逐次処理を行なうプロセス(またはタスク)が並行に実行されながら互いに通信、同期を行ない処理を実現することである。通信、同期も考慮した新しいテスト法が必要である。

そのために並行処理プログラムに対しては、Ada 並行処理プログラムを対象に、遠隔手続き呼び出しと、共有変数とにそれぞれに基づくテスト充分性を評価する方法の検討が既に行なわれている[4, 5]。現在広く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。そこで、C 言語で書かれた並行処理プログラムについてのテスト法を考察することにした。C 言語には元来並行処理を記述する能力はない。このため、C 言語で書かれたプログラムはオペレーティングシステムとの連携により、並行処理を実現している。オペレーティングシステムとの連携は、ライブラリとして用意されているシステムコールを呼び出すことで行なわれる。そのため、オペレーティングシステムと連携したプログラムに対しては、言語だけでなく、ライブラリに対応したテスト法とその支援システムが必要になる。

本稿では、C 言語プログラムと UNIX オペレーティングシステムで記述された並行処理プログラムに対するテスト充分性評価の可能性を明らかにするために、UNIX に備えられているプロセス間通信機能の中で、SYSTEM V 系のオペレーティングシステムに標準的に備わっているセマフォ、および 4.3BSD 系に備わっているソケットに対するテスト充分性を評価する方法について考察する。

2. 被覆率

逐次処理プログラムではテスト充分性の評価に被覆率を用いる。並行処理プログラムでもこれを用いる。まず、被覆率 C(Coverage) を次のように定義する。

[定義 1] 被覆率 C

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

ただし、W はプログラム実行時に発生する事象の集合、M はプログラムに含まれる測定事象の集合、$|\cdot|$ は集合の要素数を表す。

逐次処理プログラムの構造テストでは、プログラム内の全ての文を測定事象の集合 M とする C_0(文) 被覆率や、プログラムを制御フローグラフで表わした場合における全ての枝を M 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。しかしながら、これらの被覆率は、テスト充分性を定量的に表わす指標の 1 つとして用いられている。並行処理プログラムにおいても逐次処理プログラムと同様にテスト充分性を評価するためのテスト基準が必要である。テスト基準は、定量的に評価が可能で、しかも利用が容易でなければならない。

定義 1 の被覆率は、測定事象の定義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を 100% にすることを考える。

並行処理プログラムにおける被覆率を用いたテスト充分性評価についてはいくつかの試みが行なわれている。Taylor[7] は、プログラムのプロセスの状態の組合せ(並行状態)を節点とし、並行状態を変更する事象を枝とする並行状態グラフを定義した。テストを実施したときに実現された並行状態グラフの節点や枝の被覆率によってテスト充分性を評価する。しかしながら並行状態グラフには次の 2 つの問題点がある。

(1) 並行状態グラフの節点や枝の数がプロセス数に対して組合せ論的に増大するために、被覆率を 100% にすることが困難である。

(2) 並行状態グラフを定義するためにはプロセスの数が静的に決まっていなければならない。

C プログラムが並行処理を行なう場合、複数のプロセスが別々に起動され互いに通信を行ないながら処理をする場合がある。さらに 1 つのプロセスがシステムコール *fork* を用いて自分のコピーを複数個作り、それらが並行に処理を行なう場合も多い。つまり動的にプロセスが生成される場合も多く並行状態グラフを用いたテスト基準は実用的とはいえない。

以下に述べるセマフォテスト基準やソケットテスト基準はソースコードを基にしたテスト基準である。そのために、測定対象の事象は、静的に決定できるので、Taylor の並行状態グラフにおける問題点の (2) はない。

3. セマフォ

セマフォは Dijkstra により導入された機構である[3]。セマフォは実現しやすく十分に強力であるため、並行処理プログラミングの問題に対して、エレガントな解を与えることができる。

セマフォはセマフォ変数 **semid** とその **semid** に対する 2 つの操作、P 命令と V 命令からなる。これらの操作の定義は以下の通りである。

[定義 2] P 命令、V 命令

- P(semid)
  if(semid > 0)
  then　semid の値を 1 つ減らす
  else　プロセスの実行を一時停止
- V(semid)
  if(待機中のプロセスのキューが空でない)
  then　キューの最初のプロセスを再開
  else　semid の値を 1 つ増やす

これらの命令は SYSYTEM V では、システムコール *semop* を用いて実現される [1]。システムコール *semop* は、パラメータの値に応じて、P 命令または V 命令の働きをする。

4. セマフォテスト基準

前節で述べたように、UNIX SYSTEM V では、システムコール *semop* によってセマフォに対する操作が実現されている。まず、セマフォに対する操作そのものを測定対象とする被覆率を定義し、その 100% を Sem_0 テスト基準とする。ただし、$\{\cdot\}$ は集合を表わす。

[定義 3]

$$Sem_0 : M = \bigcup_{\mathtt{semid}} \{semop(\mathtt{semid})\}$$

Sem_0 テスト基準は、セマフォに対する操作が少なくとも 1 回は実行されることを必要とする。テスト基準の複雑さを測定対象の数で表すと、Sem_0 テスト基準の複雑さは *semop* の数である。すなわち、*semop* の総数を n とすれば複雑さは $O(n)$ となる。

セマフォに対する操作 *semop* は、セマフォ変数の値、あるいはプロセスキューの長さに応じて、動作が異なる。従って、テスト充分性を評価するためには、*semop* の実行だけでなく、異なる動作を全て実行する必要がある。そこで、次のようなセマフォ命令 semcom を定義し、それを用いて Sem_1 テスト基準を定義する。

[定義 4] セマフォ命令 semcom は次の 2 つ組である。

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

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

[定義 5]

$$Sem_1 : M = \bigcup_{\mathtt{semid}} \{\mathtt{semcom(semid)}\}$$

Sem_1 テスト基準は、セマフォに対する操作の動作が少なくとも 1 回実行されることを必要とする。この場合、測定対象の数はセマフォ命令の数、すなわち、セマフォに対する操作の 2 倍であるので、テスト基準の複雑さは $O(n)$ である。

次に、2 つのセマフォ命令が連続して実行された際の動作に基づく Sem_2 テスト基準を定義する。

[定義 6]

$$Sem_2 : M = \bigcup_{\mathtt{semid}} \{< \mathtt{semcom(semid)}_1, \mathtt{semcom(semid)}_2 >\}$$

ただし、$< \cdot >$ は順序対を表す。

$< \mathtt{semcom(semid)}_1, \mathtt{semcom(semid)}_2 >$ をセマフォ順序対と呼ぶ。セマフォ順序対は、任意のセマフォ命令の対としているので、実際には、実行できない順序対が存在する可能性がある。

Sem_2 テスト基準の複雑さは $O(n^2)$ である。

[定義 7]

$$Sem_\infty : M = \bigcup_{\mathtt{semid}} \{< \mathtt{semcom(semid)}_1, \mathtt{semcom(semid)}_2, \cdots >\}$$

プログラムに繰り返しが存在する場合、セマフォの実行列は、一般に無限個になるので、Sem_∞ テスト基準を満足する (被覆率を 100% にする) ことは、不可能である。しかしながら、セマフォ命令の全ての実行列を含んでいるので、プロセス間の同期がセマフォのみで行なわれている場合には、このテスト基準が満たされると、デッドロックやライブロックなどの同期誤りを発見できる。

以下に簡単な例を示す。図 1は親プロセスから生成された 2 つの子プロセスが、きわどい領域においてプリント文を実行するという testsem プログラムである。

```
/* semaphore example header file */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
extern int errno;
#define SEMPERM 0600
#define TRUE    1
#define FALSE   0

/* initialize semaphore */
initsem(semkey)
      key_t semkey;
{
  int status = 0,semid;
  if((semid =
semget(semkey,1,SEMPERM|IPC_CREAT|IPC_EXCL)) ==
-1){
     if(errno == EEXIST)
       semid = semget(semkey, 1, 0);
  }else /*if created...*/
     status = semctl(semid, 0, SETVAL, 1);
  if(semid == -1 || status == -1){
     perror("initsem failed");
     return (-1);
  }else
     return semid; /*all okay*/
}
/*** testsem -- test semaphore routines ***/
main()
{
```

```
    key_t semkey = 0x200;
    if(fork() == 0)
      handlesem1(semkey);
    if(fork() == 0)
      handlesem2(semkey);
}

handlesem1(skey)
key_t skey;
{
    int semid, pid = getpid();
    struct sembuf p_buf,v_buf;
     p_buf.sem_num = 0;
     p_buf.sem_op  = -1;
     p_buf.sem_flg = SEM_UNDO;
     v_buf.sem_num = 0;
     v_buf.sem_op  = 1;
     v_buf.sem_flg = SEM_UNDO;
    if((semid = initsem(skey))<0)
      exit(1);
    printf(" process %d before critical section\n",pid);
    /* P(semid) */
    semop(semid, &p_buf, 1);
    printf(" process %d in critical section\n",pid);
    /*in real life do something interesting*/
    sleep(5);
    printf(" process %d leaving\n",pid);
    /* V(semid) */
    semop(semid, &v_buf, 1);
    printf(" process %d exiting\n",pid);
    exit(0);
}

handlesem2(skey)
key_t skey;
{
    int semid, pid = getpid();
    struct sembuf p_buf,v_buf;
     p_buf.sem_num = 0;
     p_buf.sem_op  = -1;
     p_buf.sem_flg = SEM_UNDO;
     v_buf.sem_num = 0;
     v_buf.sem_op  = 1;
     v_buf.sem_flg = SEM_UNDO;
    if((semid = initsem(skey))<0)
      exit(1);
    printf(" process %d before critical section\n",pid);
    /* P(semid) */
    semop(semid, &p_buf, 1);
    printf(" process %d in critical section\n",pid);
    /*in real life do something interesting*/
    sleep(5);
    printf(" process %d leaving\n",pid);
    /* V(semid) */
    semop(semid, &v_buf, 1);
    printf(" process %d exiting\n",pid);
    exit(0);
}
```

図 1: セマフォの利用例 : testsem プログラム

testsem プログラムの動作の概要を図 2に示す。P, V 命令をセマフォ命令 semcom の定義に従って、図 2に示すように、P 命令は P' と P''、V 命令は V' と V'' で表し区別する。

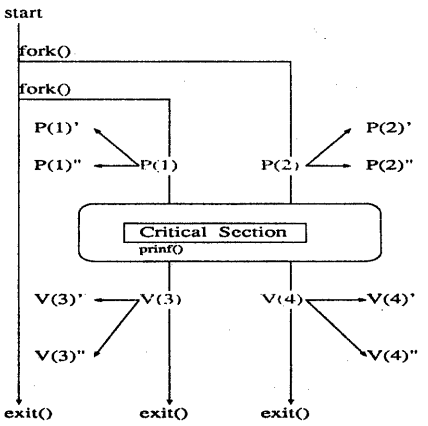

図 2: testsem プログラムの動作の概要

このプログラムの、Sem_0 テスト基準の測定事象は以下の 4 個である。

$$M = \{P(1), P(2), V(3), V(4)\}$$

Sem_1 テスト基準の測定事象は、以下の 8 個である。

$$M = \{P(1)', P(1)'', P(2)', P(2)'', V(3)', V(3)'', V(4)', V(4)''\}$$

Sem_2 テスト基準の測定事象は以下の 64 個である。

$$M = \left\{ \begin{array}{ll} < P(1)', P(1)' >, & < P(1)', P(1)'' >, \\ < P(1)', P(2)' >, & < P(1)', P(2)'' >, \\ < P(1)', V(3)' >, & < P(1)', V(3)'' >, \\ < P(1)', V(4)' >, & < P(1)', V(4)'' >, \\ \multicolumn{2}{c}{\vdots} \\ < P(1)'', P(1)' >, & < P(1)'', P(1)'' >, \\ < P(1)'', P(2)' >, & < P(1)'', P(2)'' >, \\ < P(1)'', V(3)' >, & < P(1)'', V(3)'' >, \\ < P(1)'', V(4)' >, & < P(1)'', V(4)'' >, \end{array} \right\}$$

5. ソケット

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 2 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等の要因により、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。ソケットは通信のための端点を提供する UNIX のファイルアクセス機構の一般化と考えることができる。

その使用方法は以下の通りである。

i) ソケットの生成
システムコール *socket* を用いてソケット s を次のように生成する。

$$s = socket(\texttt{domain}, \texttt{type}, \texttt{protocol});$$

domain は通信を行なう領域、type は通信データの型、protocol はソケットを操作するための通信プロトコルをそれぞれ指定する。また、*socketpair* というシステムコールもある。これはアドレスを割り当てることなしに2つの結合されたソケットを生成するシステムコールである。パイプとほとんど同じあるが、両方向にデータが流れることが出来るという点で異なる。

ii) 局所アドレスの決定

$$bind(\texttt{s}, \texttt{localaddr}, \texttt{addrlen})$$

ソケット s に、局所アドレス localaddr を与えるシステムコールである。addrlen にはアドレスの長さを指定する。

iii) ソケットの結合 (connect)
システムコール *connect* を用いてソケット s を通信相手と結合する。

$$connect(\texttt{s}, \texttt{destaddr}, \texttt{addlren})$$

destaddr は通信を行なう相手のアドレス、addrlen はアドレスの長さを指定する。

iv) データの送信
write, *writev*, *send*, *sendto*, *sendmsg* という5つのシステムコールを用いてデータを送信することが出来る。5つのうち、*write*, *writev*, *send* はソケットが結合されている場合にのみ使用できる。残りのシステムコール *sendto*, *sendmsg* は、結合されていないソケットでも送信可能である。

v) データの受信
受信は送信と対をなしている。*read*, *readv*, *recv*, *recvfrom*, *recvmsg* という5つのシステムコールを用いてデータを受信することができる。5つのうち、*read*, *readv*, *recv* はソケットが結合されている場合にのみ使用できる。残りのシステムコール *recvfrom*, *recvmsg* は、結合されていないソケットでも受信可能である。

vi) ソケットの終了

$$close(\texttt{s})$$

ファイルに対するアクセスと同様に、システムコール *close* でソケットは終了する。これとは別に、ソケットを生成したプロセスが終了すれば、システムは自動的にソケットを終了させる。

結合に基づく通信の概略を図3,4に示す。

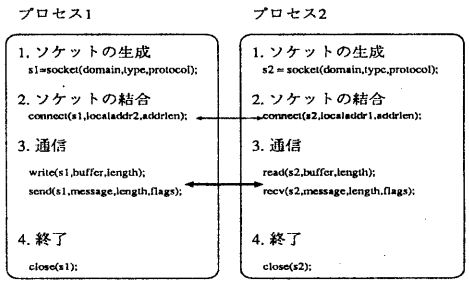

図 3: 1対1通信

図3は、1対1通信の例であり、プロセス1からプロセス2にメッセージが送られる。

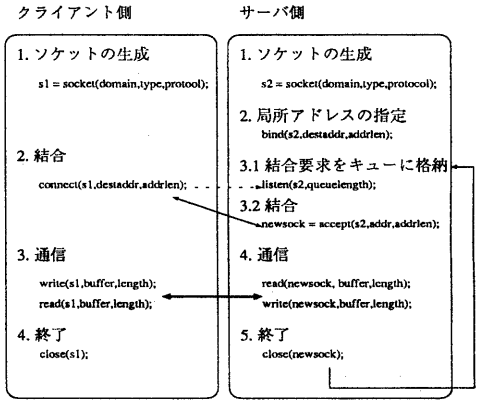

図 4: クライアント - サーバモデル

図4は、クライアントサーバモデルにおける通信の概略図である。サーバは1つであるが、クライアントは複数個存在する可能性がある。クライアント側から結合要求を出し、サーバ側が結合要求を受けつけた後に通信を行なう。

6. ソケットテスト基準

前章で述べたように、ソケットを用いるプロセス間通信は、プロセスが相手と結合 (conect) するか、あるいは相手を指定し、2つのプロセスが対をなして行なう。

最も簡単なテスト基準として、Sem_0 テスト基準と同様にソケットに関する全てのシステムコールを少なくとも1回実行する、という基準を考えることができる。しかし、この基準は逐次処理プログラムにおける C_0(文) テスト基準が満たされると必ず満たされる。また並行処理プログラム特有の複雑さに対するテストを保証するものではない。セマフォテスト基準や以下に述べるソケットテスト基準の間の関係は従来のテスト基準との関係については、7.2節で検討する。

そこでセマフォテスト基準における Sem_2 テスト基準と同様のテスト基準を考える。ソケットを用いたプロセス間通信では、データの送信と受信のためのシステムコールにより通信を行なう。そこで、データの送信と受信のためのシステムコールを通信命令と考えて、通信命令の順序対を少なくとも1回実行するとしてソケットテスト基準 $Sock_2$ を形式的に定義する。

[定義 8] 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 9] 通信命令集合 (COMS)
プログラムに含まれる、全通信命令の集合

[定義 10] ソケットテスト基準 $Sock_2$

$$Sock_2 : M = \{< e_1, e_2 > | e_1, e_2 \in \mathsf{COMS}\}$$

ここで、$< e_1, e_2 >$ を通信順序対と呼ぶ。

テスト基準の複雑さは、プログラム内の全通信文数を n とすれば、$O(n^2)$ である。

セマフォテスト基準と同様に、通信命令の列のすべてを測定対象とするソケットテスト基準 $Sock_\infty$ を定義することができる。

[定義 11]

$$Sock_\infty : M = \{< e_1, e_2, \cdots > | e_1, e_2, \cdots \in \mathsf{COMS}\}$$

$Sock_2$ テスト基準は任意の2つの通信命令を対にとるので、実行不可能な通信順序対が測定対象になる可能性がある。

7. 議論

7.1 測定可能性

前章までに C プログラムの2つのプロセス間通信、セマフォおよびソケットに基づくテスト基準、セマフォテスト基準、ソケットテスト基準を定義した。これらのテスト基準について、測定可能性、包含関係および信頼性について議論する。

テスト充分性を評価するためには、被テストプログラムの実行時における被覆率が測定できなければならない。2つのテスト基準の被覆率の測定方法を以下に述べその測定可能性を検討する。

まずソースコードから測定対象を抽出する必要がある。セマフォテスト基準とソケットテスト基準の両方ともソースコードで使用しているシステムコールを取り出し、その順序対あるいは順序列を作成する。システムコールの名前が異なるだけであり、2つのテスト基準ではほとんど同じ手法を使用することができる。

実行された測定対象を記録するには幾つかの方法がある。例えば実行時に実行状況の記録 (トレース) を保存し、実行終了後に実行した測定対象を抽出する方法がある。しかしながら、この方法は、トレースの量が膨大になるために、テスト充分性評価に用いるにのには適していない。そこで、ソースコードに挿入した探針 (Probe) によって実行した測定事象を記録する方法を用いる。探針とは被テストプログラムに挿入される測定用のプログラム片である。

セマフォテスト基準、ソケットテスト基準は両方とも、プロセス間通信を行なうシステムコールの順序対、あるいは順序列を測定対象としている。そこで被覆率を測定する探針の機能として、並行に処理されるはずのこれらのシステムコールを、何らかの形で逐次的に処理させる必要がある。逐次的に実行されたシステムコールの記録を取れば、測定事象である順序対や順序列が実行されたか知ることが可能になる。

探針挿入後のプログラムでは並行処理を逐次的に処理する。しかしながら、探針挿入後のプログラムの動作は元のプログラムの動作に含まれるものの1つである。そのために探針を挿入したプログラムで発生した誤りは元のプログラムでも発生する。

探針挿入によるその他の影響をとして、記憶領域の不足と実行時間の増大について考える。探針挿入による記憶領域の不足の場合には、プログラムの実行が異常終了するので誤った測定データを作成することはない。探針の挿入によって、元のプログラムに比べ、探針の実行に要する時間だけ実行時間が増大する。被テストプログラムが実時間処理を含んでいる場合、実行時間の増大により元のプログラムと動作が異なる場合がある。ただし、このプログラムの動作の変化は元のプログラムで出現する可能性のなかった動作を新たに導入するものではない

探針の挿入はセマフォテスト基準とソケットテスト基準で同様に行なうことができる。しかしながら、探針そのものは2つのテスト基準で異なっているので、以下にそれぞれのテスト基準に対する被覆率測定用の探針の形態を述べる。

7.1.1 セマフォテスト基準

システムコール *semop* の実行時における順序対、または順序列を計測するので、*semop* の実行は逐次的に実行される必要がある。そのため、プログラム内の *semop* を全て更にセマフォに対する P 命令と V 命令で囲む、その囲まれた領域で記録を採る、という方針で測定を行なう。具体的には図5に示すように、元のセマフォ命令に対して探針を挿入する。図5に書かれている P,V 命令も本来は *semop* で実現される。しかし分かりやすさの点から P,V と書いておく。check は探針に用いる新たなセマフォのための変数である。

```
P(check);
    { 記録を取る };
    semop(semid);
V(check);
```

図 5: セマフォに対する探針

7.1.2 ソケットテスト基準

ソケットテスト基準の被覆率測定の探針として、並行処理プログラムの動作を監視するためのモニタを使用する。モニタは、デバッグのために使用されてきた。プログラムの実行列を記録し、その実行列に従ってプログラムを再実行させれば、並行処理プログラムを決定的に実行できるため、誤りが再現でき、デバッグが容易になる。

例えば、Tai[8] は Ada 並行処理プログラムを例にとり、タスクの同期に関連した事象の列 (同期列) を定義し、与えられた同期列に基づいてプログラムを決定的に再実行する技法を提案している。再実行させる手段としてモニタを使用している。

監視用モニタを被覆率測定の探針として利用する手順は以下の通りである。

(1) 被テストプログラムにモニタとなるプロセスを生成させるプログラムを追加する。
(2) 被テストプログラムの各プロセスに各々 1 つづつソケットを追加する。
(3) 各プロセスでは通信命令の実行前に、新たに追加されたソケットを通じてモニタに自分のプロセス番号及び次に実行される通信命令の文番号を知らせ実行許可を求める。
(4) モニタは実行許可要求を記録し、実行許可を出す。

モニタは被覆率を増大させるために用いることも可能である。実行させたい順序列をモニタに予め知らせ、その順序対が実行されるようにモニタが各プロセスの実行順序を調節すれば良い。

7.2 テスト基準の間の関係

セマフォテスト基準 $Sem_i(i \geq 0)$ やソケットテスト基準 $Sock_j(j \geq 1)$、逐次処理プログラムのテスト基準 $C_k(k \geq 0)$ の間の関係について検討する。ただし、逐次処理プログラムのテスト基準 $C_k(k \geq 0)$ は、並行処理プログラムを、逐次的に実行されるプロセスの集合と考え、各プロセスにおける実行文の実行列の和集合を測定対象とする。また、Sem_0 はセマフォを操作するシステムコールの全てを測定対象とする。

まず、テスト基準の間の関係として以下の包含関係を定義する。

[定義 12] 包含関係
Cr_1 テスト基準を満足すれば、必ず Cr_2 テスト基準を満足する場合、Cr_1 が Cr_2 を包含するといい、これを $Cr_2 \supset Cr_1$ で表す。

セマフォテスト基準や、ソケットテスト基準、逐次処理プログラムのテスト基準において、それぞれ以下のような事が成立することは明らかである。

(1) $Sem_{i+1} \supset Sem_i$
(2) $Sock_{i+1} \supset Sock_i$
(3) $C_{i+1} \supset C_i$

また C_0 テスト基準は、全ての実行文を少なくとも 1 回実行するというテスト基準である。全ての実行文が実行されれば、全てのシステムコールも実行されることになるので、定義の途中で述べたように、以下の包含関係が成立する。

(4) $C_0 \supset Sem_0$
(5) $C_0 \supset Sock_1$

しかしながら、$Sem_i(i \geq 1)$ や、$Sock_j(j \geq 2)$ は、並行に実行されるプロセス間での組合せを含んでいるので、$C_k(k \geq 1)$ とは包含関係はない。

次に、並行に実行されるプロセスの実行列をシャフルした実行列を考えてみる。これは、各プロセスでの順序関係を保存し、並行処理プログラムの実行列を全順序化した実行列である。並行処理プログラムの中でプロセスは独立に操作しているのではなく、互いに同期や通信を行なっている。そのために、全順序化した実行列の中で、プロセス間での同期を満足するものだけが、実現できる。並行処理プログラムの実行列を全順序化し、同期を満足する実行列を測定対象として定義することができる。このような実行列を測定対象として被覆率を 100% にするテスト基準を全実行列テスト基準 CC_∞ と呼ぶことにする。

全実行列テスト基準 CC_∞ とセマフォテスト基準やソケットテスト基準、逐次処理プログラムのテスト基準の間に以下の包含関係が成立することは明らかである。

(6) $CC_\infty \supset Sem_\infty$
(7) $CC_\infty \supset Sock_\infty$
(8) $CC_\infty \supset C_\infty$

以上の包含関係を図に表わすと、図 6のようになる。ただし、矢印は包含関係を表わし、矢の根元のテスト基準が矢の先のテスト基準を包含することを意味する。

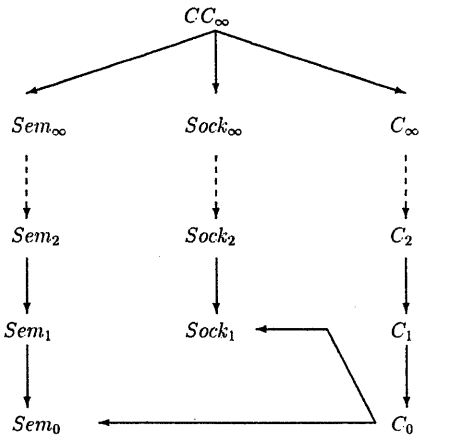

図 6: テスト基準の包含関係

7.3 テスト法の信頼性

セマフォテスト基準やソケットテスト基準を満たすテスト法の信頼性について議論する。

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。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。

前節で議論した包含関係がある場合、包含されるテスト基準がある種類の不良に対して「信頼できる」ならば、包含するテスト基準も「信頼できる」。

並行処理プログラムの誤りの原因(不良)をプロセス内の処理に関する不良と、プロセス間の処理に関する不良に分けることができる[6]。プロセス内の処理は逐次的であるので、プロセス内の処理に関する不良に対しては路解析のテスト法が「信頼できる」。

セマフォはセマフォの値をセマフォ命令に渡して通信して同期するための機構である。一方、ソケットは通信と同期を行なうことができる。そのため、セマフォテスト基準とソケットテスト基準は、同期と通信不良の両方に関連する。

セマフォ命令や通信命令の全ての実行列を含むセマフォテスト基準 Sem_∞ やソケットテスト基準 $Sock_\infty$ を満たすテスト法は、セマフォやソケットそれぞれに関して発生するデッドロックやライブロックなどの同期不良に対して「信頼できる」。しかしながら、プログラム中にループが存在すると無限個の実行列が生成されることが可能になるので、Sem_∞ や $Sock_\infty$ を満足させることはできない。さらに、前節で述べた全実行列テスト基準は、セマフォやソケットそれぞれだけでなく、全てのプロセス間通信に関して発生する同期不良に関して「信頼できる」けれども、実現は不可能である。

通信不良は、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]。セマフォテスト基準 Sem_2 とソケットテスト基準 $Sock_2$ は、それぞれセマフォ順序対あるいは通信順序対を少なくとも1回は実行することを要求する。そのために、セマフォの値とソケットを通して通信されるデータに関する完全通信不良に対して、それぞれ Sem_2 と $Sock_2$ を満足させるテスト法は「信頼できる」。

8. おわりに

C言語とUNIXオペレーティングシステムを用いた並行処理プログラムについて、UNIX SYSTEM V系のオペレーティングシステムに標準的に備わっているセマフォに対するセマフォテスト基準、およびUNIX 4.3BSD系に備わっているソケットに対するソケットテスト基準を提案した。

また、今回提案した被覆率を基にしたテスト充分性評価が実際に計測可能であることを示した。さらに、テスト基準の間の包含関係についても議論した。

プロセス間通信機能としては4.3BSDにはソケットしかないが、SYSTEM Vにはセマフォ以外にも以下のようなものがある。

- レコードロック
- メッセージ伝達
- 共有メモリ
- FIFO

これらのプロセス間通信を行なう命令に対しても、順序対を定義できればそれをテスト基準に用いたテストが可能である。

本報告では、プロセス間通信におけるテスト基準を提案したけれども、有用性についてはほとんど議論していない。C言語という一般に広く使用されているプログラミング言語に対するテスト基準であるので、定性的な評価だけでなく、実験的あるいは実証的な評価が可能である。現在、被覆率を計測するツールの開発を進めている。

謝辞

本稿のまとめとして、日頃から御指導頂いている九州大学工学部情報工学科荒木啓二朗助教授、程京徳助教授の御助言、御批判に感謝致します。また、一緒に議論しあった、研究室の皆様に謝意を表します。

参考文献

[1] Haviland K., Salama B.,(訳 玉井 浩) : UNIX システムプログラミング, サイエンス社, 1991.

[2] Leffler S. J., Mckusick M. K., Karels M.J., Quarterman J.S., (共訳 中村 明, 相田 仁, 計 宇生, 小池 汎平) : UNIX 4.3BSD の設計と実装, 丸善株式会社, 1991.

[3] Dijkstra E. W., : The structure of the T. H. E. multiprogramming system, *Comm.* ACM, **11**, pp 341-346, 1968.

[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] 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.

[6] 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.

[7] 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.

[8] Tai. K. C., Carver. R. H., and Obaid. E. E., : Debugging Concurrent Ada Programs by Deterministic Execution : IEEE Trans. Softw. Eng, Vol 17, No.1, pp.45-63, 1992.