软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(8):2008−2024 [doi: 10.13328/j.cnki.jos.004844]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

消息传递并行程序的弱变异测试及其转化*

巩敦卫[1], 陈永伟[1], 田 甜[2]

[1](中国矿业大学 信息与电气工程学院,江苏 徐州 221116)

[2](山东建筑大学 计算机科学与技术学院,山东 济南 250101)

通讯作者: 巩敦卫, E-mail: dwgong@vip.163.com

摘 要: 并行程序执行的不确定性,增加了测试的复杂性和难度.研究消息传递并行程序的变异测试,提出其弱变异测试转化方法,以提高该程序变异测试的效率.首先,根据消息传递并行程序包含语句的类型和语句变异之后导致的变化构建相应的变异条件语句;然后,将构建好的所有变异条件语句插入到原程序中,形成新的被测程序,从而将原程序的弱变异测试问题转化为新程序的分支覆盖问题.这样做的好处是,能够利用已有的分支覆盖方法解决变异测试问题.将该方法应用于 8 个典型的消息传递并行程序测试中,实验结果表明,该方法不但是可行的,也是必要的.

关键词: 消息传递;并行程序;变异测试;弱变异测试;转化;变异条件语句

中图法分类号: TP311

中文引用格式: 巩敦卫,陈永伟,田甜.消息传递并行程序的弱变异测试及其转化.软件学报,2016,27(8):2008−2024. http://www.jos.org.cn/1000-9825/4844.htm

英文引用格式: Gong DW, Chen YW, Tian T. Weak mutation testing and its transformation for message passing parallel programs. Ruan Jian Xue Bao/Journal of Software, 2016,27(8):2008−2024 (in Chinese). http://www.jos.org.cn/1000-9825/4844.htm

Weak Mutation Testing and Its Transformation for Message Passing Parallel Programs

GONG Dun-Wei[1], CHEN Yong-Wei[1], TIAN Tian[2]

[1](School of Information and Electrical Engineering, China University of Mining and Technology, Xuzhou 221116, China)

[2](School of Computer Science and Technology, Shandong Jianzhu University, Ji'nan 250101, China)

Abstract: A parallel program can yield nondeterministic execution, which increases the complexity and the difficulty in program testing. The mutation testing of a message passing parallel program is investigated, and an approach to transforming the weak mutation testing for the program is presented in this study with the purpose of improving the efficiency of the mutation testing. First, the mutation condition statements are built based on the type of statements and the changes resulted from mutating these statements. Then, a new program is formed by inserting all these mutation condition statements into the original program. As a result, the problem of the weak mutation testing of the original program can be transformed into that of covering the branches of the new program, therefore providing advantages of solving the problem of mutation testing by using previous methods of branch coverage. The proposed approach is applied to test eight benchmark message passing parallel programs, and the empirical results demonstrate that this new approach is not only feasible but also necessary.

Key words: message passing; parallel program; mutation testing; weak mutation testing; transformation; mutation condition statement

测试是提高软件质量的重要途径[1].为了对软件进行测试,通常需要准备一定数量的测试数据,称为测试数

* 基金项目: 国家自然科学基金(61375067); 国家重点基础研究发展计划(973)(2014CB046306-2); 江苏省自然科学基金(BK2012566)

Foundation item: National Natural Science Foundation of China (61375067); National Program on Key Basic Research Project of China (973) (2014CB046306-2); Natural Science Foundation of Jiangsu Province of China (BK2012566)

收稿时间: 2014-12-26; 修改时间: 2015-03-23; 采用时间: 2015-04-09

据集.测试数据集的充分性对于提高软件测试的可信度是非常重要的.评价某测试数据集的充分性,往往通过对被测软件实施变异测试[2].实施变异测试时,首先采用变异算子对原程序的某个语句进行符合语法的微小变动,得到的新程序称为变异体;然后采用某一相同的输入,分别执行原程序和变异体,如果这两个程序的输出或者某中间状态不同,则称该输入杀死了变异体;如果对于任何输入,这两个程序的输出或者中间状态都是相同的,则称该变异体是等价变异体[3].

变异测试的有效性一般通过变异得分衡量,即被杀死的变异体个数与全部非等价变异体个数的比.

在变异测试中,变化语句所在的位置称为变异点,变化之后的语句称为变异语句.对于某变异体而言,对原程序的变化得到的变异语句可能有一个,也可能有多个.如果变异语句只有一个,那么称该变异体为一阶变异体;否则,称为高阶变异体[4].相应的变异测试分别称为一阶和高阶变异测试.考虑到一阶变异测试是高阶变异测试的基础,本文仅研究一阶变异测试.变异测试是一种直接面向缺陷的测试方法,具有很强的检错能力,但是对于某程序而言,变异体数量的增多,增大了变异测试代价,降低了变异测试效率.

如前所述,可以采用不同的准则判定某变异体是否被杀死.如果基于原程序和变异体的输出,那么该变异测试准则称为强变异测试[5];如果基于原程序和变异体变异点的状态,那么相应的变异测试准则称为弱变异测试[6].已有研究表明:弱变异测试不但能够保证变异测试的有效性,而且能够大幅度降低变异测试代价[7].鉴于此,本文考虑弱变异测试.

所谓并行程序,是指含有两个或者以上并行执行进程(线程)的程序,已经广泛应用于科学研究和生产生活中,如天气预报、邮件过滤以及病毒分析等[8].在诸多并行程序中,消息传递并行程序的应用非常广泛,该程序不但具有传统串行程序的很多语句类型,而且包含若干并行环境控制和通信语句[9].与串行程序相比,消息传递并行程序不但变异体数量多,而且变异语句类型繁多.此外,该程序的执行还具有不确定性,这使得消息传递并行程序的变异测试非常具有挑战性.这也说明,研究该程序的变异测试理论与方法是十分有意义的.

本文研究消息传递并行程序的一阶弱变异测试,借鉴已有串行程序弱变异测试转化的方法,提出该程序的弱变异测试转化方法.该方法将消息传递并行程序的弱变异测试问题转化为另一新程序的分支覆盖问题,为采用已有的分支覆盖方法解决变异测试问题奠定了基础,从而提高了变异测试效率.为此,首先根据消息传递并行程序包含语句的类型和语句变异之后导致的变化,构建相应的变异条件语句;然后,将构建好的所有变异条件语句插入原程序中,形成新的被测程序.可以看出,消息传递并行程序变异条件语句的构建和新的被测程序的形成是本文的关键.

本文的创新之处主要体现在:(1) 给出了变异点语句为进程之间通信语句的变异条件语句构建方法;(2) 给出了变异点语句为进程之间通信语句的消息传递并行程序弱变异测试转化方法.

本文第 1 节介绍与本文研究相关的工作.本文提出的方法在第 2 节详细阐述,包括进程内语句和进程之间通信语句的变异条件语句构建以及并行程序弱变异测试转化等.第 3 节是本文方法在典型并行程序测试中的应用以及对比实验.第 4 节总结全文,并指出需要进一步研究的问题.

1 相关工作

鉴于本文研究消息传递并行程序的一阶弱变异测试,因此,与本文研究相关的工作主要包括如下 3 个方面:变异测试、串行程序弱变异测试转化以及并行程序测试.

1.1 变异测试

记被测程序为 S,变异点所在的语句为 s_j,对该语句实施某一变异算子之后,得到的变异语句记为 s_j'.如果用语句 s_j' 替代 s_j,并保持 S 的其他语句不变,那么可以得到 S 的一个变异体,记为 S'.

记 S 的输入向量为 $\bar{x}=\{x_1,x_2,...,x_i,...,x_m\}$,其中,$x_i(i=1,2,\ldots,m)$为第 i 个输入分量,其取值范围为 D_i,那么 $\bar{x}$ 的取值范围可以表示为 $D=D_1\times D_2\times...\times D_m$.

以 $\bar{x}$ 的某一取值分别作为 S 和 S'的输入(测试数据),如果这两个程序的输出或者中间状态不同,那么称该测试数据能杀死变异体 S'.特别地,如果二者输出不同,那么称该测试数据在强变异测试准则下杀死变异体 S'.

强变异测试最早由DeMillo等人提出[5].基于强变异测试准则判定某变异体是否被杀死,必须同时满足以下3个条件:(1) 可达性,测试数据能够到达变异点;(2) 必要性,测试数据到达变异点之后,能够使变异点的状态发生改变;(3) 充分性,变异点状态的改变,能够传播到程序的输出,且使得变异体和原程序的输出不同[10].容易看出,强变异测试需要执行整个被测程序,因此测试代价非常高.

为了降低变异测试代价,Howden 提出了弱变异测试准则[11].该准则基于变异点的状态是否与原程序相同,判定某变异体是否被杀死.可以看出,只需满足强变异测试中的可达性和必要性条件,即可依据弱变异测试准则判定某变异体被杀死.由于弱变异测试只需执行部分被测程序,因此,弱变异测试所需的代价明显低于强变异测试,这通过Girgis等人[12]和Marick[7]的实验得到了验证.但是变异点状态的改变,不一定能够传播到程序的输出,因此,弱变异测试的有效性一般比强变异测试低.不过,Horgan 等人从理论上证明,在特定条件下,弱变异和强变异测试能够产生同样有效的测试数据[13].之后,Offutt等人通过实验验证了上述论断的正确性[14].

除了弱变异测试之外,还可以通过约简变异体降低变异测试代价.所谓约简变异体,是在保证变异测试有效性的前提下,通过分析变异体之间的相关性减少需要杀死的变异体[15,16].目前,已有的变异体约简方法主要包括变异体抽样、变异体聚类、选择性变异以及高阶变异等[2].

1.2 串行程序弱变异测试转化

与变异测试相比,结构覆盖测试也是一种重要的软件测试方法.该方法要求生成测试数据以覆盖程序的某种结构,如语句、分支以及路径等,相应的结构覆盖准则分别称为语句覆盖[17]、分支覆盖[18]以及路径覆盖[19]等.将结构覆盖问题转化为优化问题,并采用进化优化方法生成期望的测试数据,成为目前软件测试界的重要研究方向之一.Lin 等人采用遗传算法生成覆盖路径的测试数据[20],之后,Ahmed[21],Watkins 和 Hufnagel[22]等人也提出了用于路径覆盖的测试数据生成方法.Pargas 利用控制依赖图和遗传算法,分别生成覆盖语句和分支的测试数据[23].Wegener等人针对嵌入式软件,提出了基于进化算法的分支覆盖测试数据生成方法[24].此外,我们针对多路径覆盖问题,采用遗传算法生成了期望的测试数据[25,26].

对于前面提及的弱变异测试,如果将杀死变异体的条件转化为分支覆盖条件,就能将生成杀死变异体的测试数据问题转化为覆盖分支的测试数据生成问题,而后者可以利用已有的方法求解.这样一来,就可以利用结构覆盖测试的成果解决变异测试问题.

鉴于此,Papadakis提出了弱变异测试转化方法,基于杀死变异体的必要条件生成条件语句的谓词表达式,从而将杀死变异体的问题转化为条件语句真分支的覆盖问题,并利用符号执行、混合执行以及基于搜索的方法生成变异测试数据[27].进一步地,Papadakis通过选择合适的路径生成测试数据,提高了变异测试的实用性[28].

1.3 并行程序测试

如前所述,并行程序包含多个进程,其执行具有不确定性.这使得对于相同的程序输入,不同的调度序列可能会产生不同的输出.这样一来,对于一个并行程序的测试不但要考虑该程序的输入,而且要考虑不同进程的调度序列.因此,与串行程序相比,并行程序测试的复杂度大为提高,难度大为增加.

对于并行程序的结构覆盖测试,目前已有了一些研究成果.Carver 等人对并行程序的通信序列进行了测试[29].Hilbrich等人通过追踪等待状态,检测一个并行程序可能发生的死锁[30].Takahashi等人针对并发程序提出了相应的覆盖准则,以发现该程序的特定缺陷[31].Souza 等人提出了并行程序的多种结构覆盖准则,为测试数据的生成提供了指导[8,9].此外,我们定义了等价路径,并利用遗传算法生成了覆盖目标路径或其等价路径的测试数据[32].

鉴于变异测试具有很强的缺陷检测能力,因此对并行程序实施变异测试,对于及时发现该程序存在的缺陷是非常必要的.但是并行程序存在的大量变异体,使得该程序的变异测试代价非常高.此外,并行程序生成的变异体还可能导致程序死锁,这进一步增加了变异测试的难度.

尽管如此,目前已有了诸多并行程序变异测试的方法.Carver 提出了一种执行测试和变异测试相结合的并行程序变异测试方法[33],之后,Bradbury等人针对java语言定义了一系列与线程和同步运算相关的变异算子[34].

考虑并发程序可能产生多个调度序列,Sen 等人提出了一种新的覆盖准则,使得生成的测试数据集更加充分[35].Gligoric 等人利用程序的执行信息,减少了多线程代码的变异测试代价[36],之后,Gligoric 提出了一种选择变异测试方法,减少了并发代码生成的变异体数量[37].针对消息传递并行程序,Sliva 等人设计了相关的变异算子,为该程序的变异测试奠定了基础[38].但是,已有的研究成果要么适用于并发程序的变异测试,要么针对消息传递并行程序,仅设计了变异算子,还没有提出针对消息传递并行程序变异测试效率提高的方法,特别是还没有研究该程序的弱变异测试方法.

对于串行程序的弱变异测试,已有将变异测试转化为分支覆盖测试的方法.但是对于消息传递并行程序的弱变异测试,如果期望利用上述思想转化为分支覆盖问题,那么需要采用合适的方法,生成条件语句的谓词表达式.考虑到并行程序包含的语句不但有进程内部的语句,还有不同进程之间的通信语句,而基于不同的语句产生的变异体对程序执行的影响不同,因此需要采用有针对性的方法,对于不同类型的语句构建相应的条件语句.

2 提出的方法

本文研究消息传递并行程序的弱变异测试问题,提出了该程序的弱变异测试转化方法,从而将该程序的弱变异测试问题转化为另一新程序的分支覆盖问题,以期采用已有的方法解决变异测试问题,从而提高变异测试的效率.为此,首先根据消息传递并行程序包含语句的类型和语句变异之后导致的变化构建相应的变异条件语句,然后将构建好的所有变异条件语句插入原程序中,形成新的被测程序.

为了清楚地阐述本文提出的方法,首先引出一些常用的概念.

2.1 基本概念

在不引起混淆的前提下,记被测的并行程序为 S,该程序由 $n(n>1)$个进程构成,分别记为 $S^0,S^1,\ldots,S^{n-1}$.这 n 个进程并行执行,共同完成任务,那么 S 可以表示为 $S=\{S^0,S^1,\ldots,S^{n-1}\}$.

对于程序 S 的第 i 个进程 S^i,其基本执行单元称为一个节点.对于程序 S 的某一输入向量,一个节点中的语句要么全部执行,要么全部不执行.一个节点可以是一条分支语句的判断条件,也可以是一条循环语句的控制条件;可以是一或多条连续执行的语句,也可以是一条消息发送或接收语句.通常,记进程 S^i 的第 j 个节点为 n_j^i.如果节点 n_j^i 是一条消息通信语句,那么称该节点为一个通信节点.

考虑进程 S^i,其控制流图是一个有向图,记为 $G(S^i)=\{N(S^i),E(S^i),s(S^i),e(S^i)\}$,其中,

- $N(S^i)$是 $G(S^i)$的顶点集,对于 $G(S^i)$的任何一个顶点,进程 S^i 中都有一个节点与之对应;对于进程 S^i 的任意 2 个节点 n_k 和 n_l,如果在某一输入下,n_k 执行之后 n_l 立即执行,那么这 2 个节点形成了 $G(S^i)$的一条边,记为$\langle n_k,n_l\rangle$;
- $G(S^i)$的所有边形成的集合为 $E(S^i)$;
- 此外,$s(S^i),e(S^i)$分别为 S^i 的入口和出口节点.

基于进程的控制流图,可以构建一个并行程序的控制流图.如果记 S 的控制流图为 $G(S)$,那么其顶点集记为 $N(S)$,可以表示为 $N(S)=\{N(S^0),N(S^1),\ldots,N(S^{n-1})\}$;其边集记为 $E(S)$,可以表示为

$$E(S)=\{E(S^0),E(S^1),\ldots,E(S^{n-1}),E(S^0,S^1),\ldots,E(S^0,S^{n-1}),E(S^1,S^0),\ldots,E(S^1,S^{n-1}),\ldots,E(S^{n-1},S^0),\ldots,E(S^{n-1},S^{n-2})\},$$

其中,$E(S^i,S^j)(i,j=0,1,\ldots,n-1)$为从进程 S^i 发送到 S^j 的通信边集.需要注意的是,由于进程 S^i 到 S^j 的通信是有向的,因此一般情况下,$E(S^i,S^j)\neq E(S^j,S^i)$;此外,对于 S 而言,由于该程序的多个进程可以同时开始执行,因此,并行程序 $G(S)$的入口和出口节点均不再是唯一的,且这些节点的个数与该程序包含的进程数密切相关.

图 1(a)~图 1(c)为某一并行程序的源程序,该程序的功能是求取 3 个整数的最大公约数,包含 4 个进程.其中,图 1(a)为进程 S^0 的源程序,图 1(b)为进程 S^1 和 S^2 的源程序,图 1(c)为进程 S^3 的源程序.对于每一进程,采用上面的方法可以得到该进程的控制流图.考虑不同进程的通信,可以得到基于不同进程的节点形成的通信边.与上述控制流图一起,可以构成整个并行程序的控制流图,如图 1(d)所示.

```
#include"stdio.h"
#include"mpi.h"
int main(int argc,char **argv){
1   int myid,num,x,y,z, buf[2];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&num);
    buf[0]=x;buf[1]=y;
2   MPI_Send(buf,2,MPI_INT,1,1,MPI_COMM_WORLD);
3   buf[0]=y; buf[1]=z;
4   MPI_Send(buf,2,MPI_INT,2,2,MPI_COMM_WORLD);
5   MPI_Recv(buf,1,MPI_INT,MPI_ANY_SOURCE,2,
    MPI_COMM_WORLD,&status);
6   x=buf[0];
7   MPI_Recv(buf,1,MPI_INT,MPI_ANY_SOURCE,2,
    MPI_COMM_WORLD,&status);
8   y=buf[0];
    if(x>1&&y>1)
9      { buf[0]=x; buf[1]=y;
10  MPI_Send(buf,2,MPI_INT,3,1,MPI_COMM_WORLD);
11  MPI_Recv(buf,1,MPI_INT,3,0,MPI_COMM_WORLD,&status);
12   z=buf[0]; }
13   else{
     buf[0]=-1;buf[1]=-1;
14   MPI_Send(buf,2,MPI_INT,3,1,MPI_COMM_WORLD);
15   MPI_Recv(buf,2,MPI_INT,3,1,MPI_COMM_WORLD,&status);
16    z=1;}
17   printf("result=%d\n",z);
     MPI_finalize();
     return  0;}
```

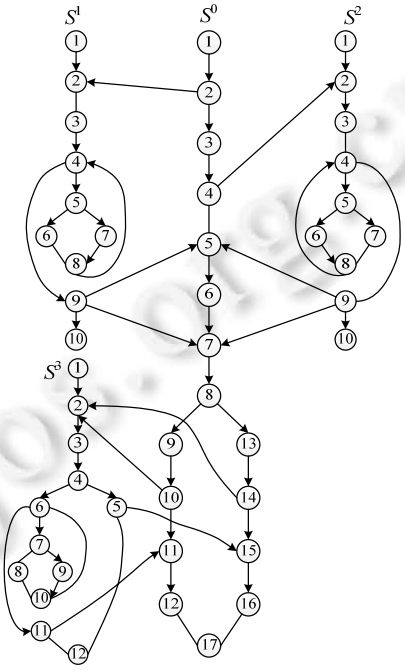
(a) 进程 S^0

```
#include"stdio.h"
#include"mpi.h"
int main(int argc,char **argv)
{
1   int myid,num,a,b,buf[2];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,
    &myid);
    MPI_Comm_size(MPI_COMM_WORLD,
    &num);
2   MPI_Recv(buf,2,MPI_INT,0,
    MPI_ANY_TAG,MPI_COMM_WORLD,
    &status);
3   a=buf[0]; b=buf[1];
4   while(a!=b)
5   {
    if(a<b)
6      b=b-a;
    else
7      a=a-b;
8   }
9   MPI_Send(&a,1,MPI_INT,0,2,
    MPI_COMM_WORLD);
10  MPI_Finalize();
    return  0;
}
```

(b) 进程 S^1 和 S^2

```
#include"stdio.h"
#include"mpi.h"
int main(int argc,char **argv)
{
1   int myid,num, a,b,buf[2];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&num);
2   MPI_Recv(buf,2,MPI_INT,0,1,MPI_COMM_WORLD,
    &status);
3   a=buf[0]; b=buf[1];
4   if(a==b)
5   MPI_Send(buf,2,MPI_INT,0,1,MPI_COMM_WORLD);
       else {
6   while(a!=b) {
7      if(a<b)
8        b=b-a;
9      else
         a=a-b;
10              }
11   MPI_Send(&a,1,MPI_INT,0,0,MPI_COMM_WORLD);
        }
12   MPI_Finalize();
     return  0;
}
```

(c) 进程 S^3



(d) 控制流图

Fig.1 Sample parallel program and its control flow graph

图 1 示例并行程序及其控制流图

2.2 变异条件语句构建

容易知道,对并行程序的不同进程、同一进程的不同变异点实施变异算子以及同一变异点实施不同的变异算子,都可以得到该程序的不同变异体.这样一来,并行程序的变异体是非常多的.因此,有必要采用合适的方法提高变异测试的效率.

为了提高变异测试效率,Papadakis等人考虑弱变异测试准则,以杀死变异体的必要条件为基础,将变异体杀死问题转化为某一条件语句真分支的覆盖问题[27].这样做的好处是,能够利用已有的方法解决变异测试问题.这样一来,与变异体对应条件语句的构建,简称为变异条件语句,显得十分重要.Papadakis 等人给出的方法如下:记由某变异点语句形成的表达式为 e,变异后语句形成的表达式为 e',那么杀死该变异体的必要条件可以表示为"$e!=e'$".以此作为某条件语句的谓词表达式,以反映该变异体被杀死的标志语句,作为该条件语句的真分支,即可得到与该变异体对应的条件语句.

可以看出,构建与某变异体对应的条件语句的关键在于形成该条件语句的谓词表达式.对于串行程序而言,上述条件语句的构建比较容易.但是,由于并行程序除了包含各进程的语句之外还包含反映不同进程通信的语句,且对于后类语句的变异有可能引起通信环境的改变和程序的死锁,这使得已有的变异条件语句构建方法很难适用于并行程序.这说明,有必要采用新的方法构建针对并行程序的变异条件语句.

对于进程内的语句,实施变异操作之后形成的变异体,可以采用文献[27]的方法构建其变异条件语句.但是对于进程之间的通信语句,实施变异操作形成的变异体,构建相应的变异条件语句之前需要重现原程序的通信环境.构建这类语句的变异条件语句时,不仅要考虑杀死变异体的必要条件,还要避免程序死锁,这些问题都是已有方法所不能解决的.下面详细阐述这两类语句的变异条件语句构建方法.

2.2.1 进程内语句的变异条件语句构建

考虑并行程序 S 的进程 S^i,记变异点所在的语句为 s_j^i,采用某一算子对 s_j^i 变异之后,得到的变异语句记为 $s_j'^i$,相应的变异体记为 S'^i.此外,记语句 s_j^i 和 $s_j'^i$ 的谓词表达式分别为 e_j^i 和 $e_j'^i$,那么根据文献[27]的方法,相应变异条件语句的谓词表达式为"$e_j^i != e_j'^i$",真分支为包含该分支覆盖标志的语句.这样一来,即可得到与该变异体对应的变异条件语句.

考虑图1示例程序的进程 S^3,变异点所在的语句为 s_7^3:"if ($a<b$)",该语句的谓词表达式为"$a<b$".采用算术运算变异算子对上述语句实施变异操作之后,得到的变异语句为 $s_7'^3$:"if ($a\leqslant b$)",其谓词表达式为"$a\leqslant b$".根据上面的方法,构建的变异条件语句的谓词表达式为"($a<b$)!=($a\leqslant b$)",真分支为"sgn=1",其中,sgn 为反映分支覆盖的标志变量.这样一来,变异条件语句可以表示为

```
if ((a<b)!=(a≤b))
    sgn=1;
```

2.2.2 进程间通信语句的变异条件语句构建

对于一个并行程序,对其变异之后,可能引起程序死锁:如果引起死锁,则说明该变异体是可以被杀死的;如果不引起死锁,则需要采用合适的方法构建相应的变异条件语句.表1列出了一些常用的MPI变异算子[38].下面根据程序变异后是否引起程序死锁,对这些变异算子进行分析.

对于表1中的变异算子"*ReplProbe*",将其施加于消息接收函数"*MPI_Recv*(&*recvbuf*,*count*,*dtype*,*source*,*tag*,*comm*,&*status*);"之后,得到变异语句为一个消息探测函数"*MPI_Probe*(*source*,*tag*,*comm*,&*status*);".考虑到消息探测函数没有改变原来接收函数的消息标签,因此可以直接执行.这样一来,该变异算子对程序中某语句的变异将不会引起变异体的死锁.类似的变异算子还有 *ReplRoot*,*ReplReduce*,*ReplGather*,*DelBarrier*, *MoveCollectiveUp Down*,*ReplModelSend*,*ReplModeSend*,*ReplSendRecv* 以及 *ReplModelRecv*.

对于变异算子"*ReplSource*",如果将其施加于消息接收函数"*MPI_Recv*(&*recvbuf*,*count*,*dtype*,*source*,*tag*,*comm*,&*status*);",那么,可以得到变异语句为"*MPI_Recv*(&*recvbuf*,*count*,*dtype*,*othersource*,*tag*,*comm*,&*status*);".可以看出,该变异算子将接收函数中的参数"*onesource*"变为"*othersource*".这样一来,变异后的接收语句将不能接

收到来自进程“*onesource*”的消息,可能会一直等待,使得变异体发生死锁.对于表 1 中的其他变异算子,如 *DelSend*,*ReplTag*,*ReplArg*,*ChanArg*,*ReplCall*,*DelCall*,*ReplStart*,*ReplWait* 以及 *InsUnaArg*,对相应的语句实施变异操作之后,也可能使得变异体发生死锁.

Table 1 Commonly used MPI mutation operators

表 1 常用的 MPI 变异算子

函数分类	变异算子	变异对象	变异方式
集合通信函数	*ReplRoot*	*MPI_Reduce*	改变参数 *root* 的值
	ReplReduce	*MPI_Reduce*	变异为 *MPI_Allreduce*
	ReplGather	*MPI_Gather*	变异为 *MPI_Allgather*
	DelBarrier	*MPI_Barrier*	移除该函数
	MoveCollectiveUpDown	集合通信函数	上移或下移若干行
点到点通信函数	*ReplModelSend*	*MPI_ISend*	变异为*_Send*
	ReplModeSend	*MPI_Send*	变异为*_Ssend*
	DelSend	*MPI_Send*	移除该函数
	ReplSource	*MPI_Recv*	改变参数 *Source* 的值
	ReplTag	*MPI_Recv*	改变参数 *Tag* 的值
	ReplProbe	*MPI_Recv*	变异为 *MPI_Prob*
	ReplModelRecv	*MPI_Recv*	变异为 *MPI_Irecv*
	DelRecv	*MPI_Recv*	移除该函数
	ReplFin	*MPI_Finalize*	变异为 *MPI_Abort*
	ReplSendRecv	*MPI_SendRecv*	变异为 *MPI_SendRecv_replace*
	ReplStart	*MPI_Startall*	变异为 *MPI_Start*
	ReplWait	*MPI_Wait*	变异为 *MPI_Waitall*
	DelFinTask	*MPI_Abort*	移除该函数
	DelDerivDType	*MPI_Type_contiguous*	移除该函数
	DelDetach	*MPI_Buffer_detach*	移除该函数
全部 MPI 函数	*ReplArg*	全部 MPI 函数	改变函数内部参数的值
	ChanArg	全部 MPI 函数	函数内部参数相互替换
	ReplComm	全部 MPI 函数	改变通信器
	ReplCall	全部 MPI 函数	改变函数
	DelCall	全部 MPI 函数	移除函数
	InsUnaArg	全部 MPI 函数	插入一元操作符

对于不会引起死锁的并行程序,其通信语句执行之后,可能引起该程序通信环境的改变.在构建变异条件语句之前,为了保证通信环境的一致性,需要重现并行程序原来的通信环境,使得变异前后在同一通信环境下具有可比性;另外,还需要在变异语句或(和)与该语句匹配的语句之后,添加原语句或(和)原语句的匹配语句,以便于对比.这样一来,添加一些语句,用于记录变异语句或其匹配语句执行之后的参量值,根据记录的参量值,便可形成变异条件语句的谓词表达式,再将能够反映变异体被杀死的标志语句作为该条件语句的真分支,即可得到变异条件语句.

容易知道,某并行程序的不同通信语句对应不同的通信环境.这需要采用有针对性的方法重现相应的通信环境.对于点到点通信函数,由于它们只涉及两个相关的进程,因此只需在变异语句和原语句之间插入相应的赋值语句,即可重现相应的通信环境.但是集合通信函数往往涉及多个进程,在重现相应的通信环境时,不仅要在变异语句和原语句之间插入相应的赋值语句,还需要对其他相关进程进行必要的处理.

现在通过一个例子说明上述构建变异条件语句的方法.

考虑图 1 示例程序的进程 S^1,变异点所在的语句 s_2^1 为

“*MPI_Recv*(*buf*,2,*MPI_INT*,0,*MPI_ANY_TAG*,*MPI_COMM_WORLD*,&*status*);”.

对该语句施加变异算子“*ReplModelRecv*”之后,得到的变异语句 $s_2'^1$ 为

“*MPI_Irecv*(*buf*,2,*MPI_INT*,0,*MPI_ANY_TAG*,*MPI_COMM_WORLD*,&*status*);”.

由于该变异算子不会引起变异体死锁,因此在构建变异条件语句之前,首先保留变异语句 $s_2'^1$ 及其匹配语句

s_2^0;然后,把语句 s_2^1 及其匹配语句 $s_2'^0$ "*MPI_Send*(*buf*,2,*MPI_INT*,1,1,*MPI_COMM_WORLD*);"分别添加到 $s_2'^1$ 和 s_2^0 之后;最后,在语句 $s_2'^1$ 和 s_2^1 之间插入赋值语句"*buf*[0]=*m*[0];*buf*[1]=*m*[1];",以重现相应的通信环境.构建变异条件语句时,首先在语句 $s_2'^1$ 和 s_2^1 之间插入赋值语句"*n*[0]=*buf*[0];*n*[1]=*buf*[1];",以记录变异语句的参量值;然后,采用"(*buf*[0]!=*n*[0])||(*buf*[1]!=*n*[1])"作为变异条件语句的谓词表达式,判断变异前后接收缓冲区 *buf* 中数据是否发生变化;最后,将"*sgn*=1"作为该条件语句的真分支,以反映该分支的覆盖情况.

综上所述,在转化后的新程序中,与对 s_2^1 实施变异算子"*ReplModelRecv*"对应的代码如图 2 所示.

```
2    MPI_Send(buf,2,MPI_INT,1,1,
     MPI_COMM_WORLD); //对应语句 s_2^0
2'   MPI_Send(buf,2,MPI_INT,1,1,
     MPI_COMM_WORLD); //对应语句 s_2'^0
```

(a) 进程 S^0 中对应的代码

```
     int m[2],n[2];
     m[0]=buf[0];  //记录通信环境
     m[1]=buf[1];
2'   MPI_Irecv(buf,2,MPI_INT,0,MPI_ANY_TAG,
     MPI_COMM_WORLD,&request); //对应语句 s_2'^1
     n[0]=buf[0];  //记录变异语句执行后的参量值
     n[1]=buf[1];
     buf[0]=m[0];  //重现相应的通信环境
     buf[1]=m[1];
2    MPI_Recv(buf,2,MPI_INT,0,MPI_ANY_TAG,
     MPI_COMM_WORLD,&status);   //对应语句 s_2^1
     if( (buf[0]!=n[0])||(buf[1]!=n[1]) )
        sgn=1;  //根据变异前后的参量值构建变异条件语句
```

(b) 进程 S^1 中对应的代码

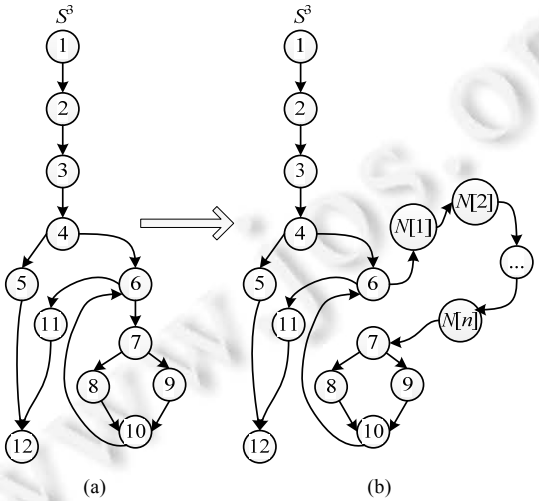
Fig.2 Building the mutation condition statement of communication statements

图 2 通信语句的变异条件语句构建

2.3 弱变异测试转化

对于每一变异语句,按照第 2.2 节的方法构建变异条件语句之后,将这些语句插入到原程序的合适位置,能够形成一个新的被测程序,简称新程序.然而,由不同类型的语句构建的变异条件语句,其插入原程序的方式不同,下面根据不同类型的变异条件语句给出其插入原程序的不同方法.

如果变异点所在的语句属于某进程内的语句,那么将这些变异条件语句依次插入到变异点所在语句之前.如图 3 所示,其中,图 3(a)为图 1 示例并行程序进程 S^3 的控制流图,图 3(b)为插入节点 7 的变异条件语句之后形成新程序的控制流图.该图中,节点 *N*[1],*N*[2],…,*N*[*n*]分别对应不同的变异条件语句.



(a) (b)

Fig.3 Mutation statement is the statement in-process

图 3 变异语句为进程内语句

如果变异点所在的语句是不同进程之间的通信语句,那么将该语句插入到变异语句之后,并将这些变异条件语句依次插入到原语句或与之匹配的接收语句之后.如图 4 所示,其中,图 4(a)为示例并行程序进程 S^0 和 S^1 的部分控制流图.由于变异点所在的语句为 S^0 的节点 2,变异语句为 S^0 的节点 2′,与节点 2 匹配的接收语句为 S^1 的节点 2′,因此,将变异条件语句插入到 S^1 的节点 2′之后,如图 4(b)所示.

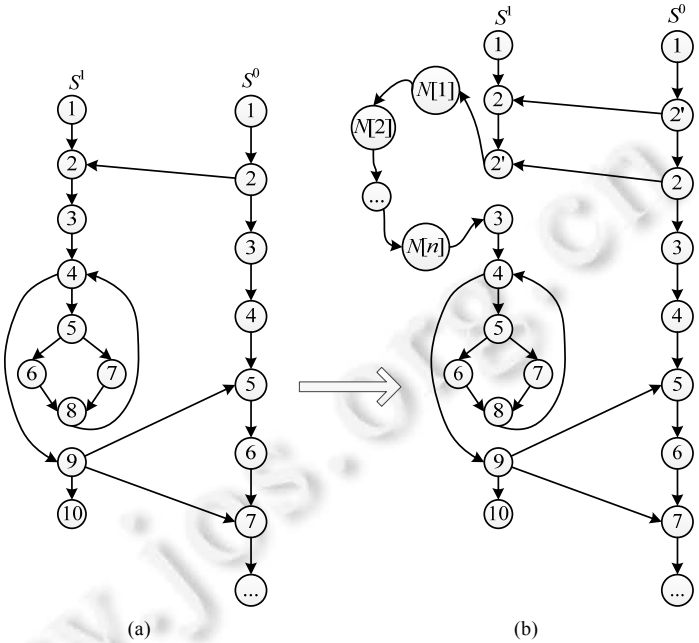


Fig.4 Mutation statement is the communication statement between processes

图 4 变异语句为进程间通信语句

可以看出,形成的新程序包含的节点急剧增多,但是并没有改变原程序的功能.这是因为在相同的调度序列下,对于相同的程序输入,原程序和新程序会执行相同的路径,从而产生相同的程序输出.

由于新程序中的变异条件语句与变异体一一对应,且变异条件语句能够反映杀死变异体的必要性条件,因此,如果某测试数据能够在弱变异测试准则下杀死原程序的某一变异体,那么该测试数据一定能够覆盖新程序对应变异条件语句的真分支.这样一来,就能将杀死原程序变异体的问题转化为新程序的分支覆盖问题,从而实现并行程序弱变异测试的转化.

2.4 实例分析

下面以图 1 中的示例程序为例,说明消息传递并行程序弱变异测试转化的过程和转化的必要性.

对示例程序节点 n_8^0 的表达式“if (*x*>1&&*y*>1)”实施变异操作,得到的 3 个变异后表达式分别为“if (*x*>=1&&*y*>1)”,“if (*x*>1&&*y*=>1)”以及“if (*x*<=1&&*y*>1)”,记它们对应的变异体分别为 *m*1,*m*2 以及 *m*3;对节点 n_2^2 的通信语句“*MPI_Recv*(*buf*,2,*MPI_INT*,0,*MPI_ANY_TAG*,*MPI_COMM_WORLD*,&*status*);”实施变异操作,得到的 2 个变异后语句为“*MPI_Probe*(0,*MPI_ANY_TAG*,*MPI_COMM_WORLD*,&*status*);”和“空语句”,记它们对应的变异体分别为 *m*4 和 *m*5;另外,对节点 n_4^3 的表达式“if (*a*==*b*)”实施变异操作,得到的 2 个变异后表达式分别为“if (*a*<*b*)”和“if (*a*>*b*)”,并记对应的变异体分别为 *m*6 和 *m*7.那么根据原程序和上述生成的变异体,采用第 2.2 节的方法,能够得到与各节点对应的变异条件语句,见表 2;采用第 2.3 节的方法将这些变异条件语句插入原程序中,能够得到弱变异测试转化后新的被测程序,其控制流图如图 5 所示.图中,节点 *si*(*i*=1,2,…,7)对应各变异条件语句,进程 S^0 的节点 4′对应通信语句“*MPI_Send*(*buf*,2,*MPI_INT*,2,2,*MPI_COMM_WORLD*);”,进程 S^2 的节点 2′

对应节点 n_2^2 的变异语句和相关赋值语句.

Table 2 Mutation condition statements n_2^2

表 2 变异条件语句n_2^2

变异所在的节点	变异条件语句	变异条件语句的谓词表达式	变异条件语句的真分支
n_8^0	$s1$	((x>1&&y>1)!=(x>=1&&y>1))	$b1$
	$s2$	((x>1&&y>1)!=(x>1&&y>=1))	$b2$
	$s3$	((x>1&&y>1)!=(x<=1&&y>1))	$b3$
n_2^2	$s4$	(buf[0]!=m[0]||buf[1]!=m[1])	$b4$
	$s5$	(buf[0]!=n[0]||buf[1]!=n[1])	$b5$
n_4^3	$s6$	((a==b)!=(a<b))	$b6$
	$s7$	((a==b)!=(a>b))	$b7$

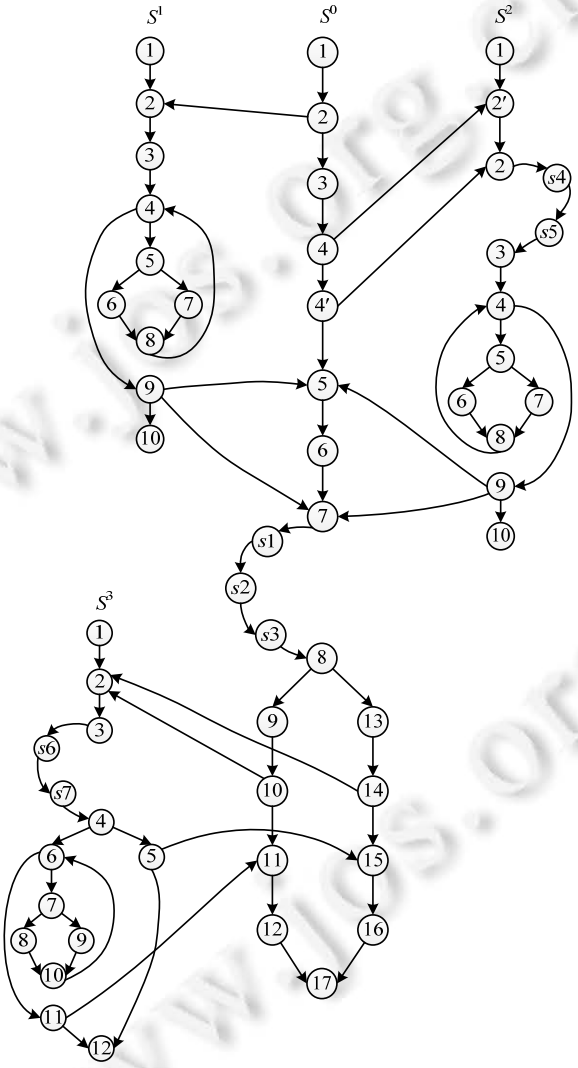


Fig.5 Control flow graph of new program

图 5 新程序的控制流图

考虑到遗传算法是一种高效的测试数据生成方法,因此选择该方法生成覆盖新的被测程序变异条件语句

真分支的测试数据.以覆盖该程序的所有变异条件语句真分支为目标,生成的测试数据集为 T={[46,38,42],[31,58,25],[1,19,91],[32,1,81]}.由于变异条件语句真分支与变异体一一对应,使用 T 的测试数据执行示例程序及其变异体,在弱变异测试准则下,杀死了 M={mi,i=1,2,…,7}的所有变异体.

下一节通过多个被测程序的实验,进一步说明本文方法的可行性和必要性.

3 实 验

本节通过实验来验证本文方法的可行性和必要性.首先给出实验需要验证的问题;然后介绍被测程序的基本信息和实验环境;在介绍实验过程之后,给出实验结果和分析.

3.1 需要验证的问题

通过验证如下两个问题,说明所提方法的可行性和必要性:

(1) 基于本文方法,以分支覆盖为准则生成的测试数据,能否在弱变异测试准则下有效地杀死变异体?

(2) 采用本文方法生成的变异测试数据是否是高效的?

为了回答这两个问题,首先,基于转化后的新程序,采用遗传算法生成覆盖变异条件语句真分支的测试数据,并利用生成的测试数据,在弱变异测试准则下对变异体进行测试,通过考察变异体能否被杀死,反映采用本文方法生成测试数据的可行性;然后,通过对比采用本文方法与传统方法生成测试数据时需要执行程序或变异体的次数和时间,反映采用本文方法生成测试数据的必要性.

3.2 被测程序和实验环境

选择 8 个并行程序作为被测程序,这些程序的基本信息见表 3,其中,程序 Including 的功能是判断点与多边形之间的位置关系,该程序取自文献[39];Gcd 源于文献[38]中的求取最大公约数程序,并增加了数值对比功能;Matrix 的功能是实现矩阵相乘,并求取结果矩阵中最大的两个元素与预设值之间的关系;此外,程序 Index 用于检索数组中大于某值元素的个数;Max_triangle 用于求取两个数与给定值之间的最大元素,并判断它们能否作为三角形的边,如果能,进一步给出它们构成三角形的类型;Compare 用于对比 2 个数组对应元素之间的关系;Lcm 的功能是求取 3 个数的最小公倍数;Rectangle 用于判断 4 个数能否作为边构成矩形.在上述程序中,Including,Gcd 和 Matrix 是典型的并行程序,而 Index,Max_triangle,Lcm,Rectangle 和 Compare 均为串行基准程序的并行实现.

实验环境配置如下:硬件配置为 Intel G630 CPU、500G 硬盘、2G 内存,软件主要包括 Windows 7 操作系统、Visual C++ 6.0 编译器以及 MPI 应用实现软件 MPICH.

Table 3 Basic information of the programs under test

表 3 被测程序的基本信息

被测程序	程序功能	输入变量个数	进程数	包含的通信语句个数
Including	判断点与多边形之间的位置关系	2	4	24
Gcd	求最大公约数并进行数值对比	3	4	15
Matrix	两矩阵相乘,并判断结果矩阵中最大的两个元素与预设值之间的关系	2	4	12
Index	字符信息检索	10	5	8
Max_triangle	求取最大元素,并判断 3 个数能否作为三角形的边及三角形类型	2	5	32
Compare	对比两个数组对应元素之间的关系	6	3	4
Lcm	求 3 个数的最小公倍数	3	3	8
Rectangle	判断 4 个数能否作为边构成矩形	4	3	8

3.3 实验过程

实验过程如下.

首先,对于每一被测程序采用不同的变异算子,生成需要杀死的变异体.采用的变异算子包括表 1 中不会引起程序死锁的 MPI 变异算子和其他 12 种方法级变异算子[40].利用这些变异算子对每一程序的进程内语句和进

程间通信语句进行变异,各程序的变异体见表 4.由该表可知,进程内语句变异产生的变异体数量明显多于通信语句.需要说明的是:该表中的变异体不包括在弱变异测试准则下的等价变异体,以便于计算变异得分.

Table 4 Mutants of the programs under test

表 4 被测程序的变异体

被测程序	进程间通信语句变异体个数	进程内语句变异体个数	变异体总数
Including	3	110	113
Gcd	12	34	46
Matrix	12	40	52
Index	8	70	78
Max_triangle	32	32	64
Compare	4	39	43
Lcm	8	36	44
Rectangle	6	16	22
合计	85	377	462

然后,基于各程序和它们的变异体,采用本文的方法形成转化后的新程序.进一步地,基于转化后的新程序,采用已有的分支覆盖方法生成覆盖变异条件语句真分支的测试数据,并利用这些测试数据,在弱变异测试准则下对各变异体进行测试,记录变异体的杀死信息,以验证本文方法的可行性.通过对比采用本文方法与传统方法生成的变异测试数据需要执行程序或变异体的次数和时间,评价采用本文方法的必要性.鉴于遗传算法是一种优秀的分支覆盖测试数据生成方法,因此,选择该方法生成覆盖分支的测试数据.

采用遗传算法生成覆盖分支的测试数据时,首先从未覆盖的变异条件语句真分支中选择一个作为目标分支,并把覆盖该分支作为进化目标,在适应度函数的引导下,经过一系列的选择、交叉以及变异操作,生成相应的测试数据.如果该测试数据能够覆盖目标分支或其他未覆盖的分支,那么保留该测试数据,将覆盖的分支标记为已覆盖分支,并从剩下的未覆盖分支中选择一个作为新的目标分支;如果该测试数据不能覆盖目标分支,那么该目标分支仍然保留在未覆盖的变异分支中,直到覆盖所有的目标分支或达到最大进化代数,遗传算法才终止.如果生成的测试数据能够覆盖新程序中全部变异条件语句真分支,那么称生成测试数据成功;否则,称生成测试数据失败.

遗传算法的个体采用二进制编码,适应度函数由分支距离和层接近度构成.其中,分支距离用于衡量变异条件语句谓词表达式为真的满足程度;层接近度用于衡量测试数据穿越的路径偏离变异条件语句所在节点的程度.分支距离和层接近度的计算方法请参见文献[41].遗传操作主要包括轮盘赌选择、单点交叉和单点变异.实验中,种群规模为 10,交叉和变异概率分别为 0.8 和 0.2,最大进化代数为 2 000.值得说明的是,遗传算法的参数取值会对实验结果产生一定的影响,这里选择的参数取值是多次实验结果的最佳值,但不一定是该参数的最优值.

采用传统方法生成测试数据时,以杀死全部变异体为目标,在测试数据的搜索范围内随机生成测试数据,并利用生成的测试数据执行原程序和各变异体,根据原程序和变异体的执行状态判断变异体是否被杀死,并给以标记.这一过程循环进行,直到生成杀死所有变异体的测试数据或达到最大运行次数,传统方法才终止.如果生成的测试数据能够杀死全部变异体,则称生成测试数据成功;否则,称生成测试数据失败.

为了使传统方法在最大运行次数内尽可能地生成期望的测试数据,取最大运行次数为 100 000.对于同一程序,本文方法与传统方法的测试数据搜索范围相同,除了程序 Gcd 的每一输入变量的搜索范围为[0 255]之外,其他程序的搜索范围均为[0 127].另外,由于本文方法需要覆盖的目标分支,与传统方法需要杀死的变异体一一对应,因此,本文方法的目标分支个数与传统方法需要杀死的变异体相同.

最后,分别记录采用本文方法和传统方法生成测试数据的实验结果并进行对比,以验证本文方法的可行性和必要性.为了减少随机因素对实验结果的影响,独立运行每一实验 30 次,并取运行结果的平均值作为实验结果.

3.4 实验结果和分析

3.4.1 本文方法的可行性

为了验证本文方法的可行性,首先,基于本文方法形成转化后的新程序,并采用遗传算法生成覆盖变异条件语句真分支的测试数据;然后利用生成的测试数据,在弱变异测试准则下对各程序的变异体进行测试,并记录杀死变异体的个数和变异得分,结果见表 5.

表 5 中,第 1 列是被测程序;第 2 列为基于分支覆盖生成的测试数据个数;第 3 列为这些测试数据覆盖的变异条件语句真分支个数;第 4 列为在弱变异测试准则下,这些测试数据杀死的变异体个数;第 5 列为这些测试数据的变异得分.

Table 5 Generated test data and mutation score by the proposed approach

表 5 本文方法生成的测试数据及其变异得分

被测程序	测试数据个数	覆盖的变异分支个数	被杀死的变异体个数	变异得分(%)
Including	12.7	113	113	100
Gcd	10.3	46	46	100
Matrix	4.0	52	52	100
Index	18.3	78	78	100
Max_triangle	7.4	64	64	100
Compare	18.0	43	43	100
Lcm	10.1	44	44	100
Rectangle	12.0	22	22	100

由表 5 可知:

(1) 对于不同的被测程序,生成的测试数据个数不同,这些测试数据覆盖的变异分支个数不同,相应地,杀死的变异体个数也不同.生成测试数据最多的程序是 Index,为 18.3,这些测试数据杀死了 78 个变异体;程序 Matrix 生成的测试数据最少,为 4.0,这些测试数据杀死的变异体个数为 52.

(2) 不同测试数据集杀死变异体的能力不同.程序 Matrix 生成的每一测试数据平均杀死 13 个变异体,能力最强;Rectangle 生成的每一测试数据杀死变异体的能力最低,平均仅能杀死 1.8 个变异体.

(3) 每一测试数据集的变异得分相同,均为 100%.

这说明,采用本文方法将变异体杀死问题转化为变异条件语句真分支的覆盖问题之后,采用遗传算法生成的覆盖变异分支的测试数据均能够在弱变异测试准则下杀死所有变异体.即本文提出的转化方法是可行的.

3.4.2 本文方法的必要性

本节通过基于本文方法转化之后,采用遗传算法,能够高效地生成变异测试数据,说明采用本文方法转化的必要性.为此,分别基于本文方法和传统方法生成期望的测试数据,记录生成期望的测试数据时,两种方法生成测试数据的个数、需要执行程序或变异体的次数以及运行时间,并计算传统方法与本文方法的测试数据个数、执行程序或变异体次数以及运行时间的比值,分别记为测试数据个数比、执行次数比以及运行时间比,结果见表 6~表 8.另外,为了确定本文方法与传统方法的上述指标值之间是否具有显著差异,采用 Mann-Whitney U 非参数假设检验,取显著性水平为 0.05.表 9 列出了假设检验结果,其中,0,1 和–1 分别表示针对某一性能指标值,这两种方法没有显著差异、本文方法显著优于传统方法以及传统方法显著优于本文方法.

Table 6 Number of test data

表 6 测试数据个数

被测程序	本文方法		传统方法		测试数据个数比
	平均值	方差	平均值	方差	
Including	12.7	2.06	12.6	1.37	1.0
Gcd	10.3	2.29	10.9	3.26	1.1
Matrix	4.0	0.43	4.0	0.50	1.0
Index	18.3	2.76	15.7	1.81	0.9
Max_triangle	7.4	0.91	8.2	0.91	1.1
Compare	18.0	0.53	18.0	0.73	1.0
Lcm	10.1	1.33	10.4	1.11	1.0
Rectangle	12	0.87	12	0.47	1.0

Table 7 Times of executing program or mutant

表 7 执行程序或变异体次数

被测程序	本文方法		传统方法		执行次数比
	平均值	方差	平均值	方差	
Including	1 020.0	453 080.00	22 049.8	187 799 614.29	21.6
Gcd	399.3	78 352.89	25 417.5	190 512 668.25	63.7
Matrix	1 158.0	679 729.33	39 243.3	1 137 696 301.53	33.9
Index	673.7	90 163.22	19 935.3	187 802 301.36	29.6
Max_triangle	1 139.0	1 384 782.33	45 807.6	1 322 995 245.77	40.2
Compare	3 021.0	2 990 335.67	63 420.5	1 041 633 590.65	21.0
Lcm	846.0	231 957.33	25 966.3	2 070 354 434.47	30.1
Rectangle	2 205.7	2 111 277.89	48 252.9	383 315 659.66	21.9

Table 8 Run times (μs)

表 8 运行时间 (μs)

被测程序	本文方法		传统方法		运行时间比
	平均值	方差	平均值	方差	
Including	700.31	319 877.90	3 485.13	6 201 893.72	4.98
Gcd	79.28	3 535.21	1 332.37	7 852 44.66	16.81
Matrix	195.90	28 356.37	993.47	1 170 020.10	5.07
Index	300.33	34 670.09	1 013.39	1 303 068.09	3.37
Max_triangle	305.94	90 776.62	3 007.30	6 139 216.20	9.83
Compare	327.38	35 158.70	1 065.89	420 101.87	3.26
Lcm	68.03	1 054.98	989.71	296 930.68	14.55
Rectangle	156.77	8 701.46	781.71	217 133.87	4.99

Table 9 Result of nonparametric hypothesis testing

表 9 非参数假设检验结果

被测程序	测试数据个数	执行次数	运行时间
Including	0	1	1
Gcd	0	1	1
Matrix	0	1	1
Index	–1	1	1
Max_triangle	1	1	1
Compare	0	1	1
Lcm	0	1	1
Rectangle	0	1	1

由表 6 和表 9 可知:

(1) 分别采用本文方法与传统方法,生成期望测试数据的个数不完全相同.有 3 个程序,即 Matrix,Compare 和 Rectangle,生成的期望测试数据个数相同;对于另外 5 个程序,生成的测试数据个数不同.

(2) 对于某些程序,虽然生成的期望测试数据个数不同,但是相差不大.相差最大的是程序 Index,为 2.6 个测试数据;其次是 Max_triangle,仅相差 0.8 个.

(3) 虽然上述测试数据个数相差不大,但是假设检验的结果表明:对于程序 Index 和 Max_triangle 而言,这种差异却是显著的.

由表 7 和表 9 可知:

(1) 本文方法需要执行程序的次数,远小于传统方法需要执行程序或变异体的次数.对于所有程序,执行次数比均超过 20,其中,执行次数比最大的程序是 Gcd,为 63.7;最小的为 21.0,是程序 Compare.

(2) 假设检验的结果表明,本文方法的执行次数显著少于传统方法.这说明采用本文方法能够大幅度减少需要执行程序的次数,从而提高了变异测试的效率.

由表 8 和表 9 可知:

(1) 对于每一被测程序,本文方法的运行时间均少于传统方法.采用本文方法生成变异测试数据,需要运行时间最少的是程序 Lcm,为 68.03μs;最多的是程序 Including,为 700.31μs;而采用传统方法,相同程序

需要的运行时间分别为 989.71μs 和 3 485.13μs.

(2) 对于不同的被测程序,运行时间比不同.运行时间比最大的是程序 Gcd,为 16.81;最小的为 3.26,对应程序 Compare.

(3) 假设检验的结果表明,对于所有被测程序,本文方法的运行时间均少于传统方法.可能的原因在于,本文方法执行程序的次数远少于传统方法.

此外,在执行程序或变异体的次数和运行时间方面,对于每一被测程序,本文方法的方差均小于传统方法;在测试数据个数方面,对于程序 Gcd,Matrix,Max_triangle 和 Compare,本文方法的方差均小于或等于传统方法.这说明,本文方法的稳定性总体优于传统方法.

综合以上实验结果与分析,可以得到如下结论:与传统方法相比,采用本文方法生成变异测试数据,需要执行程序或变异体的次数更少、运行时间更短、生成过程更稳定,从而生成测试数据的效率更高.这说明,采用本文方法进行弱变异测试是非常必要的.

本节实验结果表明,采用本文方法将弱变异测试问题转化为分支覆盖问题,并利用已有的分支覆盖方法高效地生成变异测试数据,对于提高变异测试效率不但是可行的,也是必要的.

4 结束语

本文研究的是消息传递并行程序的变异测试问题,提出了适用于该类程序的弱变异测试转化方法,目的在于提高变异测试的效率.在所提出的转化方法中,通过分析消息传递并行程序不同类型的语句及其变异后引起的状态变化,采用不同的策略构建相应的变异条件语句,并进一步形成新的被测程序.利用提出的转化方法能够将原程序的弱变异测试问题转化为新程序的分支覆盖问题,从而为采用已有的分支覆盖方法解决变异测试问题奠定了基础.

为了验证上述方法的有效性,将所提出的方法应用于 8 个典型的消息传递并行程序测试中,实验结果表明,覆盖新程序变异条件语句真分支的测试数据能够杀死原程序的变异体;与传统的随机方法相比,基于本文方法转化之后,采用遗传算法生成测试数据需要执行程序或变异体的次数更少、运行时间更短,从而提高了变异测试的效率.

值得说明的是,本文仅考虑了原程序与变异体的关系,没有分析不同变异体之间的关系,也即没有分析新程序中不同变异条件语句真分支之间的关系.实际上,这些真分支之间可能存在关联关系.如果通过分析这些变异条件语句真分支之间的关系,采用合适的方法删除冗余的变异分支,那么将会减少变异分支的个数,从而提高变异测试的效率.这将是我们需要进一步研究的课题.

References:

[1] Hamlet D. Software quality, software process, and software testing. Advances in Computers, 1995,41(8):191−229. [doi: 10.1016/S0065-2458(08)60234-X]

[2] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans. on Software Engineering, 2011, 37(5):649−678. [doi: 10.1109/TSE.2010.62]

[3] Madeyski L, Orzeszyna W, Torkar R, Jozala M. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of 2nd order mutation. IEEE Trans. on Software Engineering, 2014,40(1):23−42. [doi: 10.1109/TSE.2013.44]

[4] Jia Y, Harman M. Higher order mutation testing. Information and Software Technology, 2009,51(10):1379−1393. [doi: 10.1016/j.infsof.2009.04.016]

[5] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer, 1978,11(4): 34−41. [doi: 10.1109/C-M.1978.218136]

[6] Offutt AJ, Lee SD. How strong is weak mutation? In: Howden W, ed. Proc. of the Symp. on Testing, Analysis, and Verification. New York: ACM Press, 1991. 200−213. [doi: 10.1145/120807.120826]

[7] Marick B. The weak mutation hypothesis. In: Howden W, ed. Proc. of the Symp. on Testing, Analysis, and Verification. New York: ACM Press, 1991. 190−199. [doi: 10.1145/120807.120825]

[8] Souza SRS, Brito MAS, Silva RA, Souza RSL, Zaluska E. Research in concurrent software testing: A systematic review. In: Lourenço J, Farchi E, eds. Proc. of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. New York: ACM Press, 2011. 1−5. [doi: 10.1145/2002962.2002964]

[9] Souza PSL, Souza SRS, Zaluska E. Structural testing for message-passing concurrent programs: An extended test model. Concurrency and Computation: Practice and Experience, 2014,26(1):21−50. [doi: 10.1002/cpe.2937]

[10] DeMilli RA, Offutt AJ. Constraint-Based automatic test data generation. IEEE Trans. on Software Engineering, 1991,17(9): 900−910. [doi: 10.1109/32.92910]

[11] Howden WE. Weak mutation testing and completeness of test sets. IEEE Trans. on Software Engineering, 1982,8(4):371−379. [doi: 10.1109/TSE.1982.235571]

[12] Girgis MR, Woodward MR. An integrated system for program testing using weak mutation and data flow analysis. In: Lehman MM, Hünke H, Boehm B, eds. Proc. of the 8th Int'l Conf. on Software Engineering. Los Alamitos: IEEE Computer Society Press, 1985. 313−319.

[13] Horgan JR, Mathur AP. Weak mutation is probably strong mutation. Technical Report, SERC-TR-83-P, West Lafayette: Purdue University, 1990.

[14] Offutt AJ, Lee SD. An empirical evaluation of weak mutation. IEEE Trans. on Software Engineering, 1994,20(5):337−344. [doi: 10.1109/32.286422]

[15] Xu SY. A method of simplifying complexity of mutation testing. Journal of Shanghai University (Natural Science Edition), 2007, 13(5):524−531 (in Chinese with English abstract). [doi: 10.3969/j.issn.1007-2861.2007.05.007]

[16] Shan JH, Gao YF, Liu MH, Liu JH, Zhang L, Sun JS. A new approach to automated test data generation in mutation testing. Chinese Journal of Computers, 2008,31(6):1025−1034 (in Chinese with English abstract). [doi: 10.3321/j.issn:0254-4164.2008.06.015]

[17] Li JJ, Weiss D, Yee H. Code-Coverage guided prioritized test generation. Information and Software Technology, 2006,48(12): 1187−1198. [doi: 10.1016/j.infsof.2006.06.007]

[18] Madeyski L. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. Information and Software Technology, 2010,52(2):169−184. [doi: 10.1016/j.infsof.2009.08.007]

[19] Murrill BW. An empirical, path-oriented approach to software analysis and testing. Journal of Systems and Software, 2008,81(2): 249−261. [doi: 10.1016/j.jss.2007.05.008]

[20] Lin JC, Yeh PL. Automatic test data generation for path testing using GAs. Information Sciences, 2001,131(1-4):47−64. [doi: 10.1016/S0020-0255(00)00093-1]

[21] Ahmed MA, Hermadi I. GA-Based multiple paths test data generator. Computers & Operations Research, 2008,35(10):3107−3124. [doi: 10.1016/j.cor.2007.01.012]

[22] Watkins A, Hufnagel EM. Evolutionary test data generation: A comparison of fitness functions. Software: Practice and Experience, 2006,36(1):95−116. [doi: 10.1002/spe.684]

[23] Pargas RP, Harrold MJ, Peck RR. Test-Data generation using genetic algorithms. Software Testing Verification and Reliability, 1999,9(4):263−282. [doi: 10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y]

[24] Wegener J, Buhr K, Pohlheim H. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In: Langdon WB, Cantú-Paz E, Mathias KE, Roy R, Davis D, Poli R, Balakrishnan K, Honavar VG, Rudolph G, Wegener J, Bull L, Potter MA, Schultz AC, Miller JF, Burke E, Jonoska N, eds. Proc. of the Genetic and Evolutionary Computation Conf. San Francisco: Morgan Kaufmann Publishers, 2002. 1233−1240.

[25] Gong DW, Zhang Y. Novel evolutionary generation approach to test data for multiple paths coverage. Acta Electronica Sinica, 2010,38(6):1299−1304 (in Chinese with English abstract).

[26] Gong DW, Zhang WQ, Yao XJ. Evolutionary generation of test data for many paths coverage based on grouping. Journal of Systems and Software, 2011,84(12):2222−2233. [doi: 10.1016/j.jss.2011.06.028]

[27] Papadakis M, Malevris N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Software Quality Journal, 2011,19(4):691−723. [doi: 10.1007/s11219-011-9142-y]

[28] Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. Information and Software Technology, 2012,54(9):915−932. [doi: 10.1016/j.infsof.2012.02.004]

[29] Carver RH, Lei Y. Distributed reachability testing of concurrent programs. Concurrency and Computation: Practice and Experience, 2010,22(18):2445−2466. [doi: 10.1002/cpe.1573]

[30] Hilbrich T, de Supinski BR, Nagel WE, Protze J, Bsier C, Müller MS. Distributed wait state tracking for runtime MPI deadlock detection. In: Gropp W, Matsuoka S, eds. Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. New York: ACM Press, 2013. 1−12.[doi: 10.1145/2503210.2503237]

[31] Takahashi J, Kojima H, Furukawa Z. Coverage based testing for concurrent software. In: Croll PR, ed. Proc. of the 28th Int'l Conf. on Distributed Computing Systems Workshops. Washington: IEEE Computer Society, 2008. 533−538. [doi: 10.1109/ICDCS.Workshops.2008.76]

[32] Tian T, Gong DW. Model of test data generation for path coverage of message-passing parallel programs and its evolution-based solution. Chinese Journal of Computers, 2013,36(11):2212−2223 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2013.02212]

[33] Carver R. Mutation-Based testing of concurrent programs. In: Proc. of the IEEE Int'l Test Conf. on Designing, Testing, and Diagnostics. Washington: IEEE Computer Society, 1993. 845−853. [doi: 10.1109/TEST.1993.470617]

[34] Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent Java (J2SE 5.0). In: Proc. of the 2nd Workshop on Mutation Analysis. Washington: IEEE Computer Society, 2006. 83−92. [doi: 10.1109/MUTATION.2006.10]

[35] Sen A, Abadir MS. Coverage metrics for verification of concurrent SystemC designs using mutation testing. In: Mishra P, Zilic Z, eds. Proc. of the IEEE Int'l High Level Design Validation and Test Workshop. IEEE, 2010. 75−81. [doi: 10.1109/HLDVT.2010.5496659]

[36] Gligoric M, Jagannath V, Marinov D. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In: Gaudel MC, Cavalli AR, Ghosh S, eds. Proc. of the 3rd Int'l Conf. on Software Testing, Verification and Validation. Washington: IEEE Computer Society, 2010. 55−64. [doi: 10.1109/ICST.2010.33]

[37] Gligoric M, Zhang LM, Pereira C, Pokam G. Selective mutation testing for concurrent code. In: Pezzè M, Harman M, eds. Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2013. 224−234. [doi: 10.1145/2483760. 2483773]

[38] Silva RA, de Souza SRS, de Souza PSL. Mutation operators for concurrent programs in MPI. In: Champac V, Zorian Y, eds. Proc. of the 13th Latin American Test Workshop. Washington: IEEE Computer Society, 2012: 1−6. [doi: 10.1109/LATW.2012.6261240]

[39] Chen GL, An H, Chen L, Zheng QL, Shan JL. Parallel Algorithm Practice. Beijing: Higher Education Press, 2004 (in Chinese).

[40] Ma YS, Offutt J, Kwon YR. MuJava: A mutation system for Java. In: Osterweil LJ, Rombach D, Soffa ML, eds. Proc. of the 28th Int'l Conf. on Software Engineering. New York: ACM Press, 2006. 827−830. [doi: 10.1145/1134285.1134425]

[41] Yao XJ. Theory of evolutionary generation of test data for complex software and applications [Ph.D. Thesis]. Xuzhou: China University of Mining and Technology, 2011 (in Chinese with English abstract).

附中文参考文献:

[15] 徐拾义.降低软件变异测试复杂性的新方法.上海大学学报(自然科学版),2007,13(5):524−531. [doi: 10.3969/j.issn.1007-2861.2007.05.007]

[16] 单锦辉,高友峰,刘明浩,刘江红,张路,孙家骓.一种新的变异测试数据自动生成方法.计算机学报,2008,31(6):1025−1034. [doi: 10.3321/j.issn:0254-4164.2008.06.015]

[25] 巩敦卫,张岩.一种新的多路径覆盖测试数据进化生成方法.电子学报,2010,38(6):1299−1304.

[32] 田甜,巩敦卫.消息传递并行程序路径覆盖测试数据生成问题的模型及其进化求解方法.计算机学报,2013,36(11):2212−2223. [doi: 10.3724/SP.J.1016.2013.02212]

[39] 陈国良,安虹,陈崚,郑启龙,单久龙.并行算法实践.北京:高等教育出版社,2004.

[41] 姚香娟.复杂软件测试数据进化生成理论及应用[博士学位论文].徐州:中国矿业大学,2011.

巩敦卫(1970－),男,江苏铜山人,博士,教授,博士生导师,CCF 专业会员,主要研究领域为基于搜索的软件工程,智能优化与控制.

田甜(1987－),女,博士,讲师,CCF 专业会员,主要研究领域为并行程序测试.

陈永伟(1989－),男,硕士,助理工程师,主要研究领域为基于搜索的软件测试.