软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(10):2504−2520 [doi: 10.13328/j.cnki.jos.004749]

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

基于统计占优分析的变异测试*

张功杰[1,4], 巩敦卫[2], 姚香娟[3]

[1](中国矿业大学 计算机科学与技术学院,江苏 徐州 221116)

[2](中国矿业大学 信息与电气工程学院,江苏 徐州 221116)

[3](中国矿业大学 理学院,江苏 徐州 221116)

[4](江苏师范大学 计算机科学与技术学院,江苏 徐州 221116)

通讯作者: 巩敦卫, E-mail: dwgong@vip.163.com

摘 要: 为数众多的变异体产生的高昂测试代价严重影响了变异测试技术在实际程序中的应用.为了大幅度减少弱变异测试中变异体的数量,提出基于统计占优分析的变异体约简方法.该方法首先利用变异前后的语句构造变异分支,并将所有变异分支集成到原程序中,形成新的被测程序;然后,通过统计测试用例对各个变异分支的覆盖信息,确定变异分支之间的占优关系;最后得到非被占优分支集,其对应的变异体就是约简后的变异体.将该方法用于 8 个程序的测试,结果表明:该方法能够约简平均 90%的变异体,从而显著提高了变异测试的效率.

关键词: 软件测试;变异测试;变异体约简;占优关系;统计占优分析

中图法分类号: TP311

中文引用格式: 张功杰,巩敦卫,姚香娟.基于统计占优分析的变异测试.软件学报,2015,26(10):2504−2520. http://www.jos.org.cn/1000-9825/4749.htm

英文引用格式: Zhang GJ, Gong DW, Yao XJ. Mutation testing based on statistical dominance analysis. Ruan Jian Xue Bao/Journal of Software, 2015,26(10):2504−2520 (in Chinese). http://www.jos.org.cn/1000-9825/4749.htm

Mutation Testing Based on Statistical Dominance Analysis

ZHANG Gong-Jie[1,4], GONG Dun-Wei[2], YAO Xiang-Juan[3]

[1](School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China)

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

[3](College of Science, China University of Mining and Technology, Xuzhou 221116, China)

[4](School of Computer Science and Technology, Jiangsu Normal University, Xuzhou 221116, China)

Abstract: The high cost resulting from a large number of mutants hinders mutation testing in practical application. In order to greatly reduce mutants under weak mutation testing, a new approach to reducing mutants based on statistical dominance analysis is presented. In the proposed approach, mutant branches are first constructed by combining statements before and after mutation, and a new program is formed by integrating all mutant branches into the original program. Furthermore, the dominance relations among mutant branches in the new program are determined by statistical information of coverage after executing test cases. Finally, the non-dominated mutant branches are obtained corresponding to the mutants after reduction. The proposed approach is applied to test eight programs, and the experimental results demonstrate that it can reduce 90% mutants on average, therefore greatly improve the efficiency of mutation testing.

Key words: software testing; mutation testing; mutant reduction; dominance relation; statistical dominance analysis

作为软件开发周期中的重要环节,软件测试能够发现程序中存在的缺陷,从而提高被测软件的质量.变异测

* 基金项目: 国家自然科学基金(61375067, 61203304); 江苏省自然科学基金(BK2012566); 中央高校基本科研业务专项基金(2012QNA41)

收稿时间: 2013-05-15; 修改时间: 2014-07-01; 定稿时间: 2014-11-03

试是一种直接面向缺陷的测试技术,包括 3 个主要环节:(1) 对原程序进行合乎语法的微小变动,生成被称为变异体的原程序副本,相应的语法变动规则称为变异算子;(2) 基于给定的测试用例集,分别执行原程序和变异体,如果某变异体执行结果不同于原程序,则称该变异体被杀死;(3) 计算变异得分(mutation score,被杀死变异体占所有可杀死变异体的百分比,不可杀死的变异体称为等价变异体),并对测试结果进行评价.

变异测试以人为的方式注入缺陷[1],可以选择注入缺陷的种类和发生的位置,具有很强的针对性[2].研究表明,这种变异缺陷很大程度上能够反映软件中的真实缺陷[3,4].变异测试通过发现这些变异缺陷,评价和改善测试用例集的完备性.与其他结构化测试准则相比,满足变异测试准则的测试用例集具有更高的缺陷检测能力[5].研究结果表明:变异测试能够反映测试用例集的实际缺陷检测能力[6−10],而传统覆盖测试(如语句覆盖、分支覆盖、all-uses 等)与测试用例的缺陷检测能力没有很强的联系[9−13].

变异测试的可行性基于如下两个基本假设:胜任的程序员假设(competent programmer hypothesis,简称 CPH)[14]和组合效应假设(coupling effect hypothesis,简称 CEH)[14,15].CPH 认为:胜任的程序员开发的程序“接近”正确版本[16],因此程序中的缺陷是有限的、可描述的,并可以通过简单的语法转化更正.CEH 认为:复杂缺陷由简单缺陷组合而成[16],因此能够检测到所有简单缺陷的测试用例集,也能够检测到绝大多数的复杂缺陷.CPH 和 CEH 在变异测试中被广泛认可,CEH 更是得到充分的理论证明和实验验证[15,17,18].作为一种有效的测试方法,变异测试已取得丰硕的研究成果,相关工具也已应用于实际的软件测试中[19].

实际的软件通常包含大量的代码、复杂的语句以及各种变量[20],这些显著增加了可变异的位置和可实施的变异算子类型,从而导致变异体数量急剧增加.为了杀死这些变异体,所需的测试用例也随之增加.这说明,庞大的变异体数量显著增加了变异测试的成本,从而降低了变异测试的效率.因此,采用合适的策略减少变异体的数量,将有助于提高变异测试的效率.尽管已经有一些约简变异体的方法[21],但是现有方法约简变异体的效率仍有待于进一步提高.

本文根据弱变异测试准则,提出基于统计占优分析的变异体约简新方法.该方法首先基于弱变异测试转化思想[22]构造反映杀死变异体必要性条件的变异分支;然后,利用统计方法分析变异真分支之间的占优关系,那些被占优变异分支对应的变异体就是需要约简的变异体.相应地,非被占优分支对应的变异体则是约简后的变异体;最后,依据本文所提方法建立变异体自动约简原型系统.8 个典型程序的变异测试结果表明:所提方法能够大幅度约简变异体,从而提高变异测试效率.

本文第 1 节综述相关的研究工作和本文的研究动机.所提出的方法在第 2 节详细加以阐述,包括弱变异测试转化思想、统计占优分析模型以及变异体约简的方法.第 3 节建立基于本文方法的原型系统,并进行实验和分析.最后,在第 4 节中总结全文,并指出后续的研究内容.

1 相关工作

降低变异测试代价一直是研究的热点,常用方法有:减少变异体数量和通过弱变异测试降低执行代价.此外,语句的相关性分析也有助于降低变异测试代价.本节从以上 3 个方面回顾已有的研究工作,并引入本文的研究问题.

1.1 变异体约简

为数众多的变异体是影响变异测试效率的主要原因,因此,如何有效地减少变异体数量,成为相关研究的主要动机之一.Acree 和 Budd 通过随机选择一定比例的变异体执行变异测试,首次提出抽样变异测试思想[23,24].Mathur 等人在 10%~40%的抽样范围内,以 5%的步长考察抽样比例对变异测试充分度的影响,结果发现:随着抽样比例的减小,变异得分明显下降[25].

Mathur 对各类变异算子进行分析,发现变异算子 ASR(array reference for scalar variable replacement)和 SVR(scalar variable replacement)产生 30%~40%的变异体,因此建议舍弃这两类变异算子,并提出选择变异测试方法[26].Offutt 等人的实验结果表明:随着被舍弃变异算子的增多,变异得分呈现明显的下降趋势[27].Offutt 等人通过 28 个测试程序考察各类变异体的分布情况,发现仅保留 ABS(absolute value insertion)等 5 类变异算子,能够

维持较高的变异测试充分度[28].

与上述变异体约简方法不同的是,Jia 等人利用元启发式搜索方法,将多个传统变异体组合成一个高阶变异体(high order mutant,简称 HOM)[29].高阶变异体包含多个缺陷,而仅包含一个缺陷的变异体,称为一阶变异体(first order mutant,简称 FOM).高阶变异体不仅能反映实际软件中的复杂缺陷,还能显著减少变异体的数量[30].Polo 等人通过生成二阶变异体,降低了约 50%的变异测试成本,并维持较高的变异测试充分度[31].

抽样和选择方法能够有效地减少变异体数量,并维持较高的变异测试充分度.但是,随着约简率的提高,变异测试充分度受到明显的影响.因此对于实际的应用程序,抽样和选择很难在维持很高变异得分的同时取得较高变异体约简率.高阶变异测试无疑是一种有效、实用的测试技术,但是随着变异体阶次的提高,测试成本也急剧攀升.

1.2 弱变异测试

要杀死变异体,首先,测试用例必须能够执行到变异语句;然后,执行变异语句后,程序的状态发生改变;最后,状态的改变必须影响到程序的输出结果.DeMillo 等人将上述 3 个条件分别定义为可达性、必要性和充分性[32].强变异测试准则以满足充分性条件作为杀死变异体的判定依据.Howden 认为,在强变异测试中,反复执行变异语句的后续语句是不必要和低效的,并提出弱变异测试准则[33].该准则考察执行变异语句后的程序状态,如果与原程序不同,则认为变异体被杀死,即,满足杀死变异体的必要性条件.

可达性、必要性和充分性这 3 个条件逐次增强,因此,满足充分性条件更加困难.而满足必要性条件并不能保证执行结果一定不同.Horgan 等人的分析表明:在特定情况下,弱变异测试和强变异测试同样有效[34].DeMillo 等人的实验结果也表明:满足必要性条件的测试数据,能够在很大程度上满足充分性条件[35].Offutt 等人通过实验分析得出:在绝大多数的情况下,弱变异测试是强变异测试的有效替代,且弱变异测试能够节省 50%的测试成本[16].

为了进一步降低弱变异测试成本,Papadakis 等人根据杀死变异体的必要性条件,将变异前后的语句组合成变异分支,并把所有变异分支集成到原程序中,将变异测试问题转化为分支覆盖问题进行求解,进一步提高了变异测试效率[22].Papadakis 等人还通过分析变异分支所在的路径,选择适当的路径,高效地生成覆盖相应路径的测试用例[36].但是,由于没有对变异体实施约简,变异体转化为等量的变异分支,集成到新程序中,极大地增加了转化后程序的复杂度,降低了测试用例的生成效率.

1.3 相关性分析

弱变异测试方法能够显著降低变异测试代价;基于弱变异测试准则,将变异测试问题转化为分支覆盖问题,或路径选择问题进行求解,可以进一步提高变异测试效率.然而,弱变异测试或基于弱变异准则的测试转化方法并没有减少所需杀死的变异体数量,这说明,这类方法的测试效率仍有很大的提升空间.因此,采用合适的策略大幅度约简变异体,能够显著提高弱变异测试效率.研究表明:程序语句之间存在复杂的相关性,设计合适的测试策略,充分利用某种相关性,能够有效降低测试代价.

Ghiduk 等人在研究语句覆盖测试问题时,根据目标语句之间的相关性,通过仅覆盖非被占优语句,有效减少了所需覆盖的目标语句数量[37].Gong 等人利用分支之间的占优关系判定不可行路径,从而避免了试图覆盖不可行路径产生的不必要测试成本[38].对语句进行相关性分析,同样有益于提高变异测试效率.单锦辉等人分析同一位置产生的多个变异体,并利用杀死这些变异体条件的相似性将它们组合成复合变异体,从而显著减少了变异体数量[39].徐拾义根据变异语句之间的控制关系,得到非受控变异语句对应的变异体,即为约简后变异体,明显减少了变异体数量[40].

本文基于弱变异测试准则,研究变异体约简方法,以提高变异测试用例的生成效率.本文利用 Papadakis 等人[22]的弱变异测试转化思想,将变异测试问题转化为分支覆盖测试问题进行求解.利用统计分析方法,检测变异分支之间的占优关系,得到一个非被占优的分支集,该集合对应的变异体,即为约简后的变异体.覆盖非被占优分支集的测试用例,也能够以弱变异测试准则杀死所有变异体.

2 本文所提方法

本节详细阐述基于统计占优分析进行变异体约简的方法,该方法首先根据弱变异测试转化思想,将变异体转化为变异分支,并集成到原程序中,形成新的被测程序;然后建立统计占优分析模型,自动检测新的被测程序中变异分支之间的占优关系;最后,约简被占优分支对应的变异体.本节主要从以上 3 个方面进行阐述,并给出相应的案例分析.

2.1 弱变异测试转化

对于被测程序 P,以语句为单位,记某变异点为 s,对 s 实施某变异算子后,产生的变异语句记为 s',对应的变异体为 m.根据弱变异测试准则,如果测试数据 t 能够杀死 m,那么 t 必须能够执行到 s',并且执行 s'后状态发生改变,即 $s!=s'$.如果将 $s!=s'$作为条件构建分支语句 b,那么覆盖 b 真分支的测试数据能够以弱变异测试准则杀死变异体 m.这样,就将杀死 m 的变异测试问题转化为覆盖 b 的真分支的分支覆盖测试问题.

所构造的分支语句 b 仅反映杀死变异体的必要性条件,并且与变异体 m 一一对应,称 b 为变异分支.变异分支在程序 P 中,不执行任何实质性操作,因此不会对该程序的执行产生任何影响,而且 b 与变异前语句 s 具有相同的可达性.此外,b 是由变异前后语句 s 和 s'通过"!="组合而成的,因此容易构建.

对程序 P 实施所有变异算子,产生变异体集 M,包含变异体 $m_1,m_2,\ldots,m_{|M|}$,其中,$|M|$为 M 的元素个数,即变异体个数.相应地,所构造的变异分支集为 B,包含变异分支 $b_1,b_2,\ldots,b_{|B|}$,与变异体一一对应,$|B|=|M|$.将这些变异分支依次插入到程序 P 的相应位置,得到新的转化后程序 P'.至此,所有变异体都以变异分支的形式集成到程序 P'中.

图 1 为三角形分类程序 Triangle[30]的部分代码,利用 MuClipse 变异测试工具[41],对第 5 行~第 13 行所对应的代码段实施全部的 15 类方法级(traditional-level)变异算子[42],共生成 78 个变异体,见表 1.其中,代码第 5 行~第 7 行产生 26 个变异体,组合后得到 26 个变异分支,其分支条件见表 2.图 2 为融入变异分支后,新程序的控制流程图.

表 2 中,实施 AOIS 变异算子后得到的分支条件,其执行对原程序构成影响.例如,对 if (a==b)的变量 a 实施 AOIS 变异算子,得到++a,−−a,a++和 a−−(对应表 2 中的分支条件 1、条件 2、条件 5 和条件 6),将++a 和−−a 替换为(a+1)和(a−1).但是,a++和 a−−的影响具有滞后性,即,对后续引用变量 a 的语句产生影响,因此,如果确定后续存在对该变量引用的语句,可以通过变量替换的方式,在后续语句中植入变异分支,或者直接将 a++和 a−−转化为(a+1)和(a−1),本文采用后者.

形成新的被测程序 P'后,杀死 P 所有变异体的变异测试问题就转化为分支测试问题,即,以 B 为目标集的分支覆盖测试问题.毋庸置疑,该转化方法能够进一步降低弱变异测试代价.但是为数众多的变异体,导致转化后新程序 P'中融入大量的额外分支,显著增加了转化后被测程序的复杂度,从而降低了转化后问题的求解效率.

```
…
1     if (a≤0||b≤0||c≤0) {
2           return 4;
3     }
4     int trian=0;
5     if (a==b) {
6           trian=trian+1;
7     }
8     if (a==c) {
9           trian=trian+1;
10    }
11    if (b==c) {
12          trian=trian+3;
13    }
…
```

Fig.1 Part of Triangle program

图 1 Triangle 的部分代码

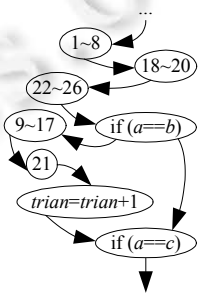

Fig.2 Control flow graph of the new program

图 2 新程序的控制流程图

Table 1 Implemented mutation operators and generated mutants

表 1 实施的变异算子及产生的变异体

变异算子	说明	变异体数量
AORB	基本算术运算符替换	12
AOIU	插入基本一元算术运算符	3
AODU	删除基本一元算术运算符	0
ROR	关系运算符替换	15
COD	删除条件运算符	0
SOR	移位运算符替换	0
LOI	插入逻辑运算符	9
ASRS	赋值运算符替换	0
AORS	简洁算术运算符替换	0
AOIS	插入简洁算术运算符	36
AODS	删除简洁算术运算符	0
COR	条件运算符替换	0
COI	插入条件运算符	3
LOR	逻辑运算符替换	0
LOD	删除逻辑运算符	0
	合计	78

Table 2 Predictions of constructed mutant branches

表 2 构建的变异分支条件

变异算子	变异分支的条件
AOIS	1. if ((*a*==*b*)!=(++*a*==*b*)); 2. if ((*a*==*b*)!=(−−*a*==*b*)); 3. if ((*a*==*b*)!=(*a*==++*b*)); 4. if ((*a*==*b*)!=(*a*==−−*b*)); 5. if ((*a*==*b*)!=(*a*++==*b*)); 6. if ((*a*==*b*)!=(*a*−−==*b*)); 7. if ((*a*==*b*)!=(*a*==*b*++)); 8. if ((*a*==*b*)!=(*a*==*b*−−)); 9. if ((*trian*+1)!=(++*trian*+1)); 10. if ((*trian*+1)!=(−−*trian*+1)); 11. if ((*trian*+1)!=(*trian*+++1)); 12. if ((*trian*+1)!=(*trian*−−+1)).
AOIU	13. if ((*trian*+1)!=(−*trian*+1)).
AORB	14. if ((*trian*+1)!=(*train*−1)); 15. if ((*trian*+1)!=(*trian**1)); 16. if ((*tiran*+1)!=(*train*/1)); 17. if ((*trian*+1)!=(*trian*%1)).
COI	18. if ((*a*==*b*)!=(!(*a*==*b*))).
LOI	19. if ((*a*==*b*)!=(~*a*==*b*)); 20. if ((*a*==*b*)!=(*a*==~*b*)); 21. if ((*trian*+1)!=(~*trian*+1)).
ROR	22. if ((*a*==*b*)!=(*a*<*b*)); 23. if ((*a*==*b*)!=(*a*≤*b*)); 24. if ((*a*==*b*)!=(*a*!=*b*)); 25. if ((*a*==*b*)!=(*a*≥*b*)); 26. if ((*a*==*b*)!=(*a*>*b*)).

图 1 中实施变异的代码段,其有效行数为 9,分支语句个数为 3,而新程序 *P*′中将增加 78 个变异分支,如果每个变异分支包含 3 行有效代码,那么,*P*′的代码行将增加 78×3÷9=26 倍.这说明采用适当的方法减少所需覆盖的变异分支数量,从而显著简化转化后的新程序,非常有助于进一步提高变异测试效率.

2.2 统计占优分析

P′融入大量的变异分支,显著增加了新程序的复杂度.但是这些变异分支之间并非完全独立,它们之间存在某种相关性.例如表 2 中的分支条件 19“if ((*a*==*b*)!=(~*a*==*b*))”,其真分支执行,分支条件 5、分支条件 6、分支条件 14、分支条件 15 等对应的真分支必然执行;再如,分支条件 23 为“真”,分支条件 22 和分支条件 24 必然为“真”.

这说明,P'的变异分支之间存在某种必然联系.

如果能够判定进而充分利用这些相关性,将有助于减少所需覆盖的变异分支,从而降低所需杀死的变异体数量,达到约简变异体的目的.上述变异分支之间的关系称为占优关系,下面给出变异分支占优关系的定义.

定义 1(变异分支之间的占优关系). P'为转化后的新程序,$b_i,b_j\in B$ 为 P'的两个变异分支,以任一测试数据执行 P',如果 b_i 的真分支被执行,b_j 的真分支也必然被执行,那么称 b_i 占优 b_j,记为 $b_i\succ b_j$;称 b_i 为占优分支,b_j 为被占优分支.

对于转化后的新程序 P',占优关系描述变异分支之间的一种必然相关性,这种相关性可以通过统计分析的方法获得.为此,定义随机变量,并根据变异分支的覆盖情况计算这些随机变量之间的条件概率.对于被测程序 P',其取值域为 D,$b_i,b_j\in B$ 是 P'的 2 个变异分支,$\omega\in D$ 为测试数据,定义随机变量 X 和 Y.

$$X=\begin{cases}1, & covered(\omega,b_i)\\ 0, & \neg covered(\omega,b_i)\end{cases} \tag{1}$$

$$Y=\begin{cases}1, & covered(\omega,b_j)\\ 0, & \neg covered(\omega,b_j)\end{cases} \tag{2}$$

其中,$covered(\omega,b_i)$表示测试数据ω能够覆盖 b_i 的真分支,$\neg covered(\omega,b_i)$表示ω不能够覆盖 b_i 的真分支.变量 X 和 Y 是两个服从(0,1)分布的随机变量,如果 b_i 和 b_j 之间存在占优关系,那么,X 和 Y 的取值存在必然的制约关系.因此,通过 X 和 Y 的条件概率描述 b_i 和 b_j 之间的相关性.当 X=0,1 时,Y 的分布率为

$$\mathrm{Pro}\{Y=y\mid X=0\}=\begin{cases}1-p, & Y=0\\ p, & Y=1\end{cases} \tag{3}$$

$$\mathrm{Pro}\{Y=y\mid X=1\}=\begin{cases}1-q, & Y=0\\ q, & Y=1\end{cases} \tag{4}$$

公式(4)表示当 X=1 时,Y=1 或 Y=0 的概率,即:表示变异分支 b_i 执行,b_j 执行或者不执行的概率.

定理 1. q=1 时,$b_i\succ b_j$.

证明:对于公式(4),当 q=1 时,Pro{Y=1|X=1}=1,这说明在 X=1 的条件下,Y=1 是必然事件,即:当 b_i 的真分支被覆盖时,b_j 的真分支必然被覆盖.因此,$b_i\succ b_j$. □

对于 b_i 和 b_j,只有 q 值确定,才能判定 b_i 是否占优 b_j.对于 q 的取值,既不能依靠经验判断,更不能通过穷举定义域 D 上的所有取值得到,这将严重影响变异测试的自动化程度.因此,本文对 q 的取值进行极大似然估计.在 D 上随机取 N 组测试数据$\omega_1,\omega_2,\ldots,\omega_N$,并以这些测试数据考察 b_i 和 b_j 的覆盖情况,那么得到容量为 N 的样本$(X_1,Y_1),(X_2,Y_2),\ldots,(X_N,Y_N)$,其对应的值为$(x_1,y_1),(x_2,y_2),\ldots,(x_N,y_N)$,概率分布函数为

$$\mathrm{Pro}\{Y=y|X=x\}=q^{xy}(1-q)^{x(1-y)}p^{(1-x)y}(1-p)^{(1-x)(1-y)},x,y=0,1 \tag{5}$$

样本的极大似然函数为

$$L(p,q;x_1,x_2,...,x_N)=\prod_{i=1}^{N}q^{x_iy_i}(1-q)^{x_i(1-y_i)}p^{(1-x_i)y_i}(1-p)^{(1-x_i)(1-y_i)} \tag{6}$$

为了求 q 的极大似然估计值,令:

$$\frac{\partial(\ln(L(p,q;x_1,x_2,...,x_N)))}{\partial q}=0 \tag{7}$$

得到:

$$\hat{q}=\frac{\sum_{i=1}^{N}x_iy_i}{\sum_{i=1}^{N}x_i} \tag{8}$$

利用上述方法,在程序 P 的取值域 D 上随机选取 N 个测试数据,执行并记录每个测试数据对 P'变异分支的覆盖情况.Pro{Y=1|X=1}=q,通过对随机变量 X 和 Y 的取值,计算参数 q 的极大似然估计值 $\hat{q}$. 根据定理 1,当 $\hat{q}=1$

时,b_i 和 b_j 的真分支之间存在占优关系.需要说明的是:有一类特殊的占优关系 $b_i \succ b_j$ 且 $b_j \succ b_i$,对于该类占优关系,本文只取 $b_i \succ b_j$ 或 $b_j \succ b_i$ 的其中之一.

通过上述统计分析方法,能够自动检测变异分支之间的占优关系.根据这些占优关系确定需要约简的变异体,是下一小节要解决的问题.

2.3 变异体自动约简

对于 b_i 和 b_j,如果 $b_i \succ b_j$,那么根据弱变异测试准则,杀死 m_i 的测试数据一定能够杀死 m_j.因此,m_j 可以被约简,即:所有被占优变异分支,其对应的变异体就是要约简的变异体;而非被占优分支对应的变异体,则是约简后的变异体.

定义 2(非被占优变异分支集). 考虑 P'的变异分支集 B,对于变异分支 $b_i \in B$,如果不存在任何 $b_j \in B, b_j \neq b_i$ 使得 $b_j \succ b_i$ 成立,那么称 b_i 为非被占优分支.所有非被占优分支构成的集合称为非被占优分支集,记为 B^{nd}.

定理 2. B^{nd} 所对应的变异体,是约简后的变异体.

证明:采用反证法.假设 B^{nd} 对应的变异体不是约简后的变异体,那么至少存在一个可被约简的变异体,对应的变异分支为 $b \in B^{nd}$,即,b 是被占优分支.这与 B^{nd} 是“非被占优分支集”矛盾. □

以 N 个测试数据 $\omega_1, \omega_2, \ldots, \omega_N$ 作为输入,执行转化后程序 P',并记录各测试数据对变异分支的覆盖情况.生成 P'的执行矩阵 $Exe=e_{ij}$,见表 3.该过程相当于采样,如算法 1 所示.

Table 3 Execution matrix of mutant braches

表 3 变异分支的执行矩阵

ω_i	b_j				
	b_1	b_2	b_3	…	$b_{\|B\|}$
ω_1	0	1	0	…	1
ω_2	1	1	0	…	0
ω_3	0	0	1	…	1
…	…	…	…	…	…
ω_N	1	0	1	…	1

算法 1. 执行矩阵的生成.

Input:测试数据 $\omega_1, \omega_2, \ldots, \omega_N \in D$;

Output:变异分支集 B 的执行矩阵 Exe.

初始化执行矩阵 $Exe=e_{ij}$,设置 $e_{ij}=0, i=1,2,\ldots,N, j=1,2,\ldots,|B|$

函数 $covered(\omega_i, b_j)$,当测试数据 ω_i 能够覆盖变异分支 b_j 时,返回布尔值 true

```
for (i=1; i≤N; i++) {
  for (j=1; j≤|B|; j++) {
    if (covered(ωi,bj)) {
      eij=1
    }
  }
}
```

Exe 记录各变异分支的覆盖情况,$e_{ij}=1$ 表示测试数据 ω_i 能够覆盖 b_j;否则,不能覆盖 b_j.如果所有测试数据都不能覆盖某变异分支,则需要进一步判断该变异分支是否可满足.如果对于任何测试数据都不能执行该变异真分支,如表 2 中的变异分支条件 13,这在语义上等同于等价变异体,因此应排除这类变异分支.

对于执行矩阵 Exe,可以利用统计占优分析方法,检测变异分支之间的占优关系,统计结果用占优矩阵 $Dom=d_{ij}, i,j=1,2,\ldots,|B|$存储.根据公式(8)和定理 1, $\hat{q}=\sum_{k=1}^{N}(e_{ki}\cdot e_{kj})\Big/\sum_{k=1}^{N}e_{ki}$, 如果 $\hat{q}=1$, 那么 $d_{ij}=1$;否则,$d_{ij}=0$.具体过程如算法 2 所示.

算法 2. 生成变异体的占优矩阵.

Input:变异分支集 B 的执行矩阵 Exe;

Output:变异分支集 B 的占优矩阵 Dom.

```
初始化占优矩阵 Dom=d_ij,设置 d_ij=0,i,j=1,2,…,|B|
N 为测试数据数量,也是样本容量
for (i=1; i≤|B|; i++) {
  sumx=0
  for (k=1; k≤N; k++) {
    sumx+=e_ki
  }
  for (j=1; j≤|B|; j++) {
    if (j==i)
      continue
    sumxy=0
    sumy=0
    for (k=1; k≤N; k++) {
      sumxy+=e_ki·e_kj
      sumy+=e_kj
    }
    if (sumxy!=0) {
      if (sumxy==sumx) {
        if (d_ji!=1)
          d_ij=1
      } else {
        if (sumxy==sumy) {
          if (d_ij!=1)
            d_ji=1
        }
      }
    }
  }
}
```

算法 2 执行后,其输出结果为占优矩阵 Dom,见表 4.本质上,Dom 存储的是以 B 的各变异分支为顶点,所包含的占优关系为边构成的有向图 $G(B)=(V(B),E(B))$.$d_{ij}=1$ 表明,存在边$\langle v_i,v_j\rangle\in E(B)$,其中,$v_i,v_j\in V(B)$,与 B 中的变异分支 b_i 和 b_j 相对应.

Table 4 Dominance matrix of mutant branches

表 4 变异分支的占优矩阵

b_i	b_j				
	b_1	b_2	b_3	…	$b_{\|B\|}$
b_1	0	0	0	…	1
b_2	1	0	0	…	1
b_3	0	0	0	…	1
…	…	…	…	…	…
$b_{\|B\|}$	1	0	1	…	0

在 $G(B)$中,顶点 v_i 的出度 $outDegree(v_i)=\sum d_{i*}$ 表示 b_i 占优的变异分支个数,v_i 的入度 $inDegree(v_i)=\sum d_{*i}$ 表示占优 b_i 的变异分支个数.因此,根据表 4 的输出结果,入度为 0 的顶点对应的变异分支是非被占优分支;所有非被占优分支构成的集合就是非被占优集 B^{nd}.

2.4 案例分析

对于图 1 中代码所产生的 78 个变异分支,采用本文的统计占优分析方法随机生成测试数据,对转化后的程序 P'实施分支覆盖测试;且在测试过程中,生成 P'的变异分支执行矩阵 *Exe*.如表 5 所示,采用 11 个测试数据执行转化后的程序 P'(对应的样本容量 N=11),得到对 78 个变异分支的覆盖信息.

Table 5 Execution matrix

表 5 执行矩阵

ω_i	b_j b_1~b_{78}
ω_1	000000000000000000100011100111001000000000000000000001010111111111111111111011111
ω_2	100110010000000001000111001010111111111111111111011111001111011000011000000000000
ω_3	000000000000000001000111001010111111111111111111011111001111000000000000000000000
ω_4	100110010000000001000111001110010000000000000000000011100100000000000000000000000
ω_5	111111111111011111111110101101011111111111111111111111101011111111111111111111111
ω_6	000000000000000001000111000011110110000110000000000000111100000000000000000000000
ω_7	000000000000000001000001110011110000000000000000000000011110110000110000000000000
ω_8	000000000000000001000001110011110000000000000000000000011100110010010010000000000
ω_9	111111111111011111111110101111001100100100100000000000111001100100100100000000000
ω_{10}	011001100000000001000001110011110110000110000000000001010111111111111111111011111
ω_{11}	000000000000000000100000111101011111111111111111101111111100100000000000000000000

表 5 中,分支条件 13(if ((*trian*+1)!=(–*trian*+1)),见表 2)没有被覆盖.进一步分析发现,变量 *trian* 的初始值为 0,因此,分支条件 13 是不可满足分支,对应的变体为等价变异体.对转化后的 77 个可满足分支,利用本文方法进行统计占优分析,得到 B^{nd}={23,25,28,30,47,54,56},对应的变异分支条件为

if ((*a*==*b*)!=(*a*≤*b*));

if ((*a*==*b*)!=(*a*≥*b*));

if ((*a*==*c*)!=(*a*≤*c*));

if ((*a*==*c*)!=(*a*≥*c*));

if ((*trian*+2)!=(–*trian*+2));

if ((*b*==*c*)!=(*b*≤*c*));

if ((*b*==*c*)!=(*b*≥*c*)).

这些变异分支对应的变异体,即为约简后的变异体.将这 7 个变异分支集成到原程序 *P* 中,得到新的约简后测试程序 *P*″.事实上,覆盖上述 7 个变异分支的测试数据可以覆盖 77 个可满足变异分支.

如果仅考虑表 2 中的 26 个分支条件,利用本文方法得到 B^{nd}={9,23,25}(见表 2),对应的分支条件为

if (*trian*+1)!=(*trian*+1+1);

if ((*a*==*b*)!=(*a*≤*b*));

if ((*a*==*b*)!=(*a*≥*b*)).

而覆盖这 3 个分支的测试数据,可以覆盖表 2 中的所有 25 个可满足分支.对图 1 中代码第 5 行~第 13 行和第 5 行~第 7 行的分析结果表明:以原程序的语句段为单位,采用本文方法对转化后的变异分支进行占优分析,结果是有效的.

从以上案例可知,程序整体的分析结果可能不等于各部分结果的并集.这是因为该并集可能存在少量冗余,甚至得到不同的非被占优分支.但是覆盖该并集的测试数据,也能够覆盖原来的所有变异真分支.因此,对于复杂程序,可以进行分组统计分析.

表 5 中的样本值来自容量 N=11 的样本.分析还发现,N 偏大将容易产生不必要的开销.例如对表 2 中的 26

个分支,仅需 7 个测试数据(相对应的,N=7)即可得到 B^{nd}={9,23,25}.测试数据太少(样本容量太小)容易产生误判,即:实际不存在占优关系,误判成占优关系.这种误判一方面导致更多的变异体被约简;另一方面,约简后生成的测试用例集,其缺陷检测能力下降(变异得分减少).例如,对于 77 个可满足变异分支,当采用更少的测试数据进行分析时,尽管得到的 B^{nd} 更小,但是约简后生成的测试用例不能够覆盖所有的变异分支.

上述案例的分析结果初步说明本文方法是可行的,但仍需进一步进行实验,以验证所提方法的有效性.

3 实验与分析

本节通过实验进一步检验所提方法的有效性.首先,给出实验需验证的问题;其次,介绍实验所用的程序;再次,建立实验模型,即,本文的变异体约简原型系统,并描述实验的基本过程;最后,给出实验结果和分析.

3.1 需要解决的问题

本文的目的是通过减少变异体数量,进一步提高测试用例的生成效率.为此,需验证的问题如下:

(1) 本文方法能否显著减少变异体数量?这将通过约简前后的变异体数量进行比较,用被约简的变异体数量占所有非等价变异体数量的百分比,即,约简率进行比较;

(2) 利用本文方法进行变异体约简,是否能够提高变异测试用例的生成效率?这需要对转化后程序 P'以及约简后的新程序 P''分别执行测试用例生成实验,并比较所花费的时间;

(3) 通过本文方法约简后,所生成的测试用例是否有效?这需要对约简前后的测试程序分别生成测试用例,执行分支测试和强变异测试,并比较变异分支覆盖情况(弱变异测试)和强变异得分情况;

(4) 约简后的变异体所生成的测试用例,其缺陷能力有何变化?这需要对约简前后所生成的测试用例分别执行强变异测试,并考察所用的测试用例数量以及杀死的变异体数量;

(5) 样本容量的大小是否影响本文方法的有效性?鉴于样本容量对于本文方法的重要性,考察不同容量的样本值对统计分析结果的影响.

3.2 实验程序

选取 8 个程序,见表 6,进一步验证所提方法的有效性,其中,J1,J2,J3 和 J5 选自文献[22]的实验程序;J4 是文献[30]的实验程序;J6,J7 和 J8 来自 http://commoms.apache.org,为文献[2]的部分变异测试程序.实验所用的 8 个程序均采用 Java 语言编写,其中,J1~J5 为使用频率很高的变异测试基准程序,J6~J8 则是工业应用程序.

Table 6 Information of tested programs

表 6 实验程序信息

ID	测试程序	行数	方法		变异体		程序说明
			方法总数	被测方法	变异体总数	可测试变异体	
J1	Mid	26	1	1	115	115	3 个整数的中间值
J2	FourBalls	28	1	1	213	213	球体的相对重量
J3	TrashAndTakeOut	30	2	2	111	111	未知
J4	Triangle	36	1	1	325	325	三角形判定
J5	Cal	46	2	2	315	314	2 个日期间的天数
J6	Md5Crypt	107	7	2	169	158	Md5 加密算法
J7	WordUtils	173	2	2	262	243	字符串的操作
J8	DurationFormatUtils	365	9	3	722	576	时间格式化
合计		811	25	14	2 232	2 055	–

采用 Metrics 统计各实验程序的代码行数和方法数[43],并利用 MuClipse 生成变异体.如表 6 所列,8 个实验程序共包含 811 行有效代码、25 种方法.其中,被测方法 14 种(绝大部分没有被测试的方法是形如“*getXXX*()”的简单方法).所有实验程序共生成 2 232 个变异体,其中,可测试 2 055 个;而对于 177 个不可测试变异体,在实验中被舍弃.这些不可测试变异体均为错误变异体,如对 *final* 变量实施 AOIS 变异算子所产生的变异体等.

3.3 实验过程

实验基于Microsoft Window XP SP3操作系统和Eclipse 3.4集成开发环境.首先,采用MuClipse生成变异体,并通过对日志文件 *mutation_log* 的解析,自动生成变异分支集 *B*.将所有变异分支插入到原程序 *P* 的相应位置,得到转化后的被测程序 *P′*;对 *P′*的变异分支进行统计占优分析,得到约简后的变异分支集 B^{nd},并插入到原程序 *P* 的相应位置,得到简化后的被测程序 *P″*.实验过程如图 3 所示.

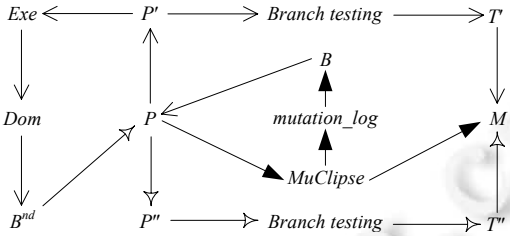

Fig.3 Prototype system of the proposed method

图 3 本文方法原型系统

通过 B^{nd} 统计约简后变异体的数量,计算本文方法的变异体约简率.对约简前后的被测程序 *P′*和 *P″*分别执行分支测试(branch testing),并生成覆盖各自变异分支集的测试用例集 *T′*和 *T″*;同时,统计生成测试用例所需的时间和变异分支覆盖率,并对 *T′*和 *T″*分别执行强变异测试,统计变异得分.实验中生成的单个测试用例,以 JUnit 的“*assertXXX*();”形式给出[44],若干测试用例构成一种测试方法.

3.4 实验结果与分析

3.4.1 变异体约简

通过变异体约简率的计算,评价本文方法的有效性.约简率等于被约减的变异体数量占所有非等价变异体数量的百分比.构造变异分支的同时,判定和统计不可满足分支的数量以及等价变异体的数量,并采用本文方法进行变异体约简,结果见表 7.实验中构造的变异分支数量等于变异体数量,相应地,约简后的变异分支数量等于保留的变异体数量.“所用测试数据”是指采用本文方法进行统计占优分析时,为获得相应容量的样本所用的测试数据数量.

Table 7 Reduction rates of the proposed method

表 7 本文方法的约简率

ID	变异体(变异分支)	不可满足变异分支	等价变异体	所用测试数据	约简后变异分支	约简率(%)
J1	115	0	18	30	14	85.57
J2	213	0	34	29	17	90.50
J3	111	0	29	7	5	93.90
J4	325	8	40	39	28	90.18
J5	314	15	43	47	31	88.56
J6	158	10	12	8	8	94.52
J7	243	2	34	11	14	93.30
J8	576	10	65	50	36	92.95
合计	2 055	45	275	221	149	91.19(平均值)

在表 7 所列的 8 个实验程序中,J6 获得了最大的约简率,为 94.52%((158−12−8)/(158−12)),仅保留 8 个变异分支.最小约简率来自 J1,为 85.57%.所有的 8 个测试程序共生成 2 055 个变异分支(45 个不可满足),其中,等价变异体 275 个;通过统计占优分析,得到非被占优变异分支 149 个,所对应的变异体即为约简后的变异体;约简变异体 1 631 个,平均约简率为 91.19%.这说明,本文方法能够约简 90%左右的变异体,因此,本文所提方法能够显著减少变异体数量.

3.4.2 测试用例生成的效率

对约简前后的测试程序分别执行分支测试,生成覆盖各变异分支的测试用例.本文对约简前后测试程序 P' 和 P''采用随机方法生成测试用例.测试用例生成时间以毫秒(ms)为单位,并通过速度比(约简前后所用时间的比值)来评价变异体约简前后测试数据生成效率的变化情况.结果见表 8.

Table 8 Cost of generating test cases before and after mutant reduction

表 8 约简前后测试用例生成成本

ID	约简前		约简后		速度比	
	测试用例	时间(ms)	测试用例	时间(ms)	所有测试用例	单个测试用例
J1	16.5	14.886 3	13.3	1.177 6	12.64	10.19
J2	27.3	44.891 6	15.2	4.110 3	10.92	6.08
J3	6.5	21.618 1	4.3	4.251 5	5.08	3.36
J4	33.7	112.571 9	21.4	7.301 3	15.42	9.79
J5	28.4	9 252.670 5	21.7	1 335.677 0	6.93	5.29
J6	5.9	1 632.170 7	4.3	166.364 2	9.81	7.15
J7	11.9	911.878 5	8.8	55.198 2	16.52	12.22
J8	26.4	42 615.561 9	21.9	4 368.835	9.75	8.09
合计	156.6	54 606.250 0	110.9	5 942.915 1	10.88(平均值)	7.77(平均值)

对于实验的 8 个程序,约简前生成 156.6 个测试用例,用时 54 606.250 0ms;约简后生成测试用例 110.9 个,用时 5 942.915 1ms.表 8 中:约简后的 8 个程序,生成测试用例的速度是约简前的 10.88 倍;而单个测试用例生成速度是约简前的 7.77 倍.其中,J7 速度比最高,达到 16.52;J3 获得速度比最小,为 5.08.所获得速度比的大小与程序本身的复杂程度有关.同样,J3 的单个测试用例生成速度比也最小,J7 的单个测试用例生成速度比则最大.

无论从生成测试用例集还是从生成单个测试用例的角度,实验结果均表明,约简后测试用例生成速度都得到显著提高.这说明,本文方法能够明显提高变异测试用例生成效率.测试用例生成效率的提高,主要得益于所需覆盖的变异分支数量的显著减少.

3.4.3 测试用例的有效性

对约简前后的所有实验程序,分别生成覆盖对应变异分支集的测试用例,并利用这些测试用例执行强变异测试.如表 9 所列,约简前生成的测试用例 158.2 个,覆盖 2 006 个变异分支,平均变异分支覆盖率为 99.79%,其中,J7 的分支覆盖率为 98.34%,其他则实现变异分支全覆盖.约简前生成的测试用例能够杀死 1 747.0 个变异体,得到 98.68%的平均变异得分,其中,J1,J2,J3 和 J6 的变异得分均为 100%;J7 的变异得分最小,为 96.46%.

Table 9 Generated test cases before mutant reduction

表 9 约简前生成测试用例

ID	测试用例	被覆盖变异分支	变异分支覆盖率(%)	杀死变异体	变异得分(%)
J1	16.3	115	100	97	100
J2	27.8	213	100	179	100
J3	6.7	111	100	82	100
J4	33.9	317	100	279.7	98.14
J5	28.7	299	100	264.3	97.53
J6	6.1	148	100	146	100
J7	12.2	237	98.34	201.6	96.46
J8	26.5	566	100	497.4	97.34
合计	158.2	2 006	99.79(平均值)	1747.0	98.68(平均值)

从表 10 可知,约简后的 8 个测试程序共生成 112.1 个测试用例,覆盖 2 004 个变异分支,取得 99.63%的变异分支覆盖率,其中,6 个程序达到全覆盖.约简后,所生成的测试用例能够杀死变异体 1 737.1 个,平均变异得分为 98.20%.其中,J1,J3 和 J6 得到 100%的变异得分;J7 的变异得分最低,为 95.89%.

对比表 9 和表 10,变异体约简前后的各程序,变异分支覆盖率均超过 99%,变异得分则超过 95%.采用本文方法约简后,所得分支覆盖率和变异得分略有下降,分别降低了 0.16%和 0.48%.同时,所生成的测试用例数量减少了 46.1 个(减少 29.14%).因此,本文方法在维持较高的变异充分度的前提下,不仅能够显著减少变异体的数量,还能有效减少生成的变异测试用例数量.

Table 10 Generated test cases after mutant reduction

表 10 约简后生成测试用例

ID	测试用例	被覆盖变异分支	变异分支覆盖率(%)	杀死变异体	变异得分(%)
J1	13.1	115	100	97	100
J2	15.6	213	100	176.8	98.77
J3	4.4	110	99.10	82	100
J4	21.7	317	100	277.4	97.33
J5	22.1	299	100	261.7	96.57
J6	4.5	148	100	146	100
J7	8.9	236	97.93	200.4	95.89
J8	21.8	566	100	495.8	97.03
合计	112.1	2 004	99.63(平均值)	1737.1	98.20(平均值)

3.4.4 缺陷检测能力

为了进一步分析变异体约简前后,测试用例杀死变异体的能力,即缺陷检测能力,分别设置 50%,70%,90%和 Top 共 4 个采样点,考察变异体约简前后,达到采样点对应的变异得分时所需生成的测试用例数量.其中,Top 是指所达到的最高变异得分(约简前的见表 9,约简后的见表 10).结果见表 11.

Table 11 Number of test cases needed for different mutation scores

表 11 达到特定变异得分所需测试用例数量

ID	约简前					约简后				
	50%	70%	90%	Top	单个测试用例缺陷检测能力	50%	70%	90%	Top	单个测试用例缺陷检测能力
J1	3.8	5.5	9.3	16.3	6.0	3.3	5.0	7.8	12.8	7.6
J2	4.4	9.0	18.4	28.0	6.4	3.8	6.2	12.1	15.8	11.2
J3	2.7	3.3	4.3	6.7	12.2	2.0	2.3	3.7	4.6	17.8
J4	11.0	16.3	25.5	33.6	8.3	7.6	11.8	17.4	21.9	12.6
J5	5.6	8.7	21.8	27.6	9.6	4.1	5.5	17.3	22.3	11.7
J6	1.0	1.0	1.0	6.4	22.8	1.0	1.0	1.0	4.6	31.7
J7	3.2	5.7	9.8	12.1	16.7	1.9	3.4	7.0	8.9	22.6
J8	11.7	15.2	19.6	26.8	18.6	10.4	11.9	13.5	22.0	22.5
				157.5(合计)	12.6(平均值)				112.9(合计)	17.2(平均值)

表 11 表明,约简后达到指定变异得分,所需生成测试用例的数量明显小于约简前.例如:J4 在约简前达到 50%的变异得分,需要生成 11.0 个测试用例;达到 70%需要 16.3 个测试用例;达到 90%需要 25.5 个;而达到 Top (98.14%,见表 9)则需生成 33.6 个测试用例.利用本文方法约简变异体后,达到上述的变异得分则分别需要 7.6, 11.8,17.4 和 21.9 个测试用例.8 个实验程序中:J6 在约简前后均只需 1.0 个测试用例就可达到 90%的变异得分; 而达到 Top(100%),则分别需要 6.4 和 4.6 个测试用例.

表 9 和表 10 表明:约简后的测试程序生成的测试用例数量明显减少.表 11 进一步验证,达到指定变异得分时,约简后的测试程序所需测试用例的数量更少.

图 4 为约简后生成的测试用例数量占约简前生成测试用例数量的百分比.可以看出:对于所有测试程序(除 J6 之外),在达到 50%,70%,以及 90%的变异得分时,生成测试数据的数量占约简前的 60%~90%;当变异得分达到 Top 时,生成测试用例仅为约简前的 55%~85%.

图 5 表明:对于每个约简后的测试程序生成的单个测试用例,其缺陷检测能力比约简前提升超过 20%,其中 J2 提升最多,超过 70%;8 个实验程序平均提升超过 35%.

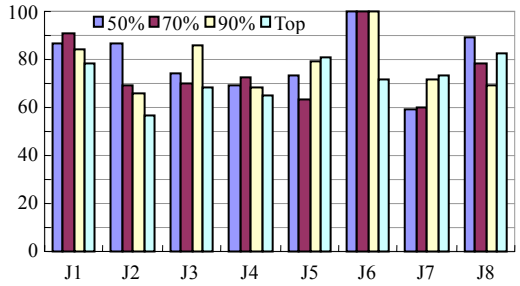

Fig.4 Comparison of number of needed test cases

图 4 所需测试用例数量的比较

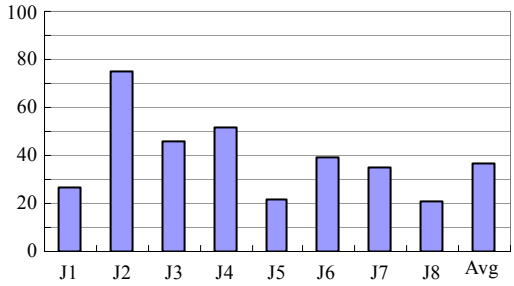

Fig.5 Improved fault detection capability of per test case

图 5 单个测试用例缺陷检测能力的提升

3.4.5 样本容量的影响

样本容量在数值上等于统计分析时所用测试数据的数量,容量的大小影响到变异体约简的有效性.上述实验表明:本文方法能够大幅度减少变异体数量,从而有效提高测试用例生成的效率.鉴于本文变异体约简的方法与样本容量密切相关,因此有必要考察样本容量对本文方法有效性的影响.为此,对不同容量的样本值统计约简后变异分支(变异体)数量以及约简后所生成测试用例的变异分支覆盖率和变异得分.见表 12.

Table 12 Influence of sample size

表 12 样本容量的影响

ID	样本容量	约简后变异分支	变异分支覆盖率(%)	变异得分(%)
J1	16	6	72.17	68.04
	23	10	89.57	91.75
	27	12	93.04	96.91
	30	14	100	100
J2	10	8	79.81	73.18
	16	12	91.08	86.03
	23	15	98.59	96.65
	29	17	100	98.77
J3	2	2	81.95	75.61
	4	4	96.40	92.68
	7	7	99.10	100
J4	10	8	60.88	43.16
	20	14	86.44	68.07
	30	21	94.95	88.77
	39	28	100	97.33
J5	17	13	66.22	53.51
	28	22	81.27	74.17
	38	28	95.32	91.14
	47	31	100	96.57
J6	3	3	90.54	88.36
	5	5	94.19	92.47
	8	8	100	100
J7	5	7	69.29	63.16
	8	11	92.53	89.47
	11	14	97.93	95.89
J8	18	13	58.48	52.45
	27	18	77.39	69.86
	35	26	90.81	84.34
	44	32	96.82	93.35
	50	36	100	97.03

由表 12 可知:对于 8 个被测程序,采用本文方法,当样本容量较小时,约简后的变异体数量更少.相应地,约简后生成的测试用例取得的变异分支覆盖率和变异得分也比较差.这说明,当样本容量不充分时,采用本文方法容易产生误判,即,将不存在的占优关系误判为占优关系.尽管这种误判使得约简率有所提高,但却降低了本文方

法的有效性.

表 12 中:随着样本容量的增加,约简后生成的测试用例,其变异分支覆盖率和变异得分趋于增长,并达到最大值.这说明,本文方法能够有效地判定占优关系,从而得到非被占优变异分支.实验过程中还发现:样本容量达到一定值后(见表 12 中各实验程序,取得最大变异分支覆盖率和最大变异得分时对应的样本容量),即使样本容量继续增加,对应的约简后变异分支数量、变异分支覆盖率和变异得分基本趋于稳定.

通过上述实验结果与分析可以得出如下结论:采用本文方法,能够大幅度约简变异体,维持很高的变异得分,并显著提高变异测试的效率;而且约简后生成测试用例的数量更少,在保持测试用例集整体缺陷检测能力的前提下,单个测试用例的缺陷检测能力更强.此外,尽管样本容量影响本文方法的有效性,但却容易确定保证本文方法有效性的样本容量.

4 总 结

为数众多的变异体是影响变异测试效率的主要因素,弱变异测试或基于弱变异转化的测试方法同样受制于庞大的变异体数量,这表明其测试效率仍有很大的提升空间.因此,采用合适的方法有效减少生成的变异体数量,是提高变异测试效率的重要途径.然而迄今为止,仍缺乏高效的变异体约简方法.

本文基于弱变异测试准则研究变异体约简问题,以进一步提高变异测试效率.本文的主要贡献有:

(1) 提出利用统计占优分析的方法,高效地约简变异体.该方法通过弱变异测试转化,将变异体转化为变异分支,集成到原程序 P 中,得到新程序 P';然后建立统计占优分析模型,分析 P'中变异分支间的占优关系,得到非被占优分支集,并集成到 P 中,得到约简后程序 P'';

(2) 根据本文所提方法建立变异体约简原型系统,并应用于 8 个程序的实验.结果表明:本文方法能够在维持较高变异充分度的前提下大幅度减少变异体数量,提高变异测试效率,增强单个测试用例的缺陷检测能力.

此外,本文还考察了样本容量对本文方法有效性的影响.

本文所提方法是通过对传统变异算子的分析得出的,而实验也同样采用传统变异算子.目前广泛应用的面向对象语言,在产生大量传统变异体的同时也生成很多的类级变异体.因此作为后续工作,本文方法将被用于类级变异体的约简,以进一步检验所提方法的有效性.此外,根据本文思想开发的原型系统仍需进一步加以完善,从而适应更复杂的工业程序,以推进变异测试自动化的程度.

致谢 感谢各位审稿专家以及本刊编辑的辛勤工作.

References:

[1] Chen JF, Lu YS, Xie XD. Research on software fault injection testing. Ruan Jian Xue Bao/Journal of Software, 2009,20(6): 1425−1443 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3526.htm [doi: 10.3724/SP.J.1001.2009.03526]

[2] Fraser G, Zeller A. Mutation-Driven generation of unit tests and oracles. IEEE Trans. on Software Engineering, 2012,38(2): 278−292. [doi: 10.1109/TSE.2011.93]

[3] Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. ACM Computing Survey, 1997,29(4):366−427. [doi: 10.1145/267580.267590]

[4] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments. In: Roman GC, Griswold WG, Nuseibeh B, eds. Proc. of the 27th Int'l Conf. on Software Engineering. Washington: IEEE, 2005. 402−411. [doi: 10.1109/ICSE.2005.1553583]

[5] Offutt AJ, Pan J, Tewary K, Zhang T. An experimental evaluation of data flow and mutation testing. Software: Practice and Experience, 1996,26(2):165−176. [doi: 10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K]

[6] DeMillo RA, Mathur AP. On the use of software artifacts to evaluate the effectiveness of mutation analysis in detecting errors in production software. Technical Report, SERC-TR-92-P, Lafayette: Purdue University, 1992. 1−32.

[7] Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. In: Cantarella JD, ed. Proc. of the 21st Int'l Conf. on Software Maintenance. Washington: IEEE, 2005. 411−420. [doi: 10.1109/ICSM.2005.9]

[8] Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans. on Software Engineering, 2006,32(9):733−752. [doi: 10.1109/TSE.2006.92]

[9] Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. on Software Engineering, 2006,32(8):608−604. [doi: 10.1109/TSE.2006.83]

[10] Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Jalote P, Briand L, Hoek A, eds. Proc. of the 36th Int'l Conf. on Software Engineering. New York: ACM Press, 2014. 435−445. [doi: 10.1145/2568225.2568271]

[11] Cai X, Lyu MR. The effect of code coverage on fault detection under different testing profiles. ACM SIGSOFT Software Engineering Notes, 2005,30(4):1−7. [doi: 10.1145/1082983.1083288]

[12] Frankl PG, Weiss SN, Hu C. All-Uses vs mutation testing: an experimental comparison of effectiveness. Journal of Systems and Software, 1997,38(3):235−253. [doi: 10.1016/S0164-1212(96)00154-9]

[13] Namin AS, Andrews JH. The influence of size and coverage on test suite effectiveness. In: Rothermel G, Dillon LK, eds. Proc. of the 8th Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2009. 57−68. [doi: 10.1145/1572272.1572280]

[14] 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]

[15] Offutt AJ. Investigations of the software testing coupling effect. ACM Trans. on Software Engineering and Methodology, 1992,1(1): 3−18. [doi: 10.1145/125489.125473]

[16] Offut AJ, Lee SD. An empirical evaluation of weak mutation. IEEE Trans. on Software Engineering, 1994,20(5):377−344. [doi: 10.1109/32.286422]

[17] Kapoor K. Formal analysis of coupling hypothesis for logical faults. Innovations in Systems and Software Engineering, 2006,2(2): 80−87. [doi: 10.1007/s11334-006-0002-z]

[18] Wah KSHT. An analysis of the coupling effect I: Single test data. Science of Computer Programming, 2003,48(2-3):119−161. [doi: 10.1016/S0167-6423(03)00022-4]

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

[20] Lammermann F, Baresel A, Wegener J. Evaluating evolutionary testability for structure-oriented testing with software measurements. Applied Software Computing, 2008,8(2):1018−1028. [doi: 10.1016/j.asoc.2006.06.010]

[21] Domínguez-Jiménez JJ, Estero-Botaro A, García-Domínguez A, Medina-Bulo I. Evolutionary mutation testing. Information and Software Technology, 2011,53(10):1108−1123. [doi: 10.1016/j.infsof.2011.03.008]

[22] 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]

[23] Acree AT. On mutation [Ph.D. Thesis]. Atlanta: Georgia Institute of Technology, 1980.

[24] Budd TA. Mutation analysis of program test data [Ph.D. Thesis]. New Haven: Yale University, 1980.

[25] Mathur AP, Wong WE. An empirical comparison of mutation and data flow based test adequacy criteria. Software Testing, Verification and Reliability, 1994,4(1):9−31. [doi: 10.1002/stvr.4370040104]

[26] Mathur AP. Performance, effectiveness, and reliability issues in software testing. In: Ladd D, ed. Proc. of the 15th Computer Software and Applications Conf. Washington: IEEE, 1991. 604−605. [doi: 10.1109/CMPSAC.1991.170248]

[27] Offutt AJ, Rothermel G, Zapf C. An experimental evaluation of selective mutation. In: Basili VR, DeMillo RA, Katayama T, eds. Proc. of the 15th Int'l Conf. on Software Engineering. Washington: IEEE, 1993. 100−107. [doi: 10.1109/ICSE.1993.346062]

[28] Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. ACM Trans. on Software Engineering and Methodology, 1996,5(2):99−118. [doi: 10.1145/227607.227610]

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

[30] Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. In: Cordy JR, Zhang L, eds. Proc. of the 8th Int'l Working Conf. on Source Code Analysis and Manipulation. Washington: IEEE Computer Society, 2008. 249−258. [doi: 10.1109/ SCAM.2008.36]

[31] Polo M, Piattini M, Garcia-Rodriguez I. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification and Reliability, 2009,19:111−131. [doi: 10.1002/stvr.392]

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

[33] 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]

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

[35] DeMillo RA, Offutt AJ. Experimental results from an automatic test case generator. ACM Trans. on Software Engineering and Methodology, 1993,2(2):109−127. [doi: 10.1145/151257.151258]

[36] Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. Information and Software Technology, 2012,54(9):915−312. [doi: 10.1016/j.infsof.2012.02.004]

[37] Ghiduk AS, Girgis MR. Using genetic algorithms and dominance concepts for generating reduced test data. Informatica, 2010,34(3): 377−385.

[38] Gong D, Yao X. Automatic detection of infeasible paths in software testing. IET Software, 2010,4(5):361−370. [doi: 10.1049/iet-sen.2009.0092]

[39] 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]

[40] Xu SY. A method of simplifying complexity of mutation testing. Chinese 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]

[41] Segura S, Hierons RM, Benavides D, Ruiz-Cortés A. Automated metamorphic testing on the analyses of feature models. Information and Software Technology, 2011,53:245−258. [doi: 10.1016/j.infsof.2010.11.002]

[42] Ma YS, Offutt J. Description of method-level mutation operators for Java. 2005. http://ise.gmu.edu/~ofut/mujava/mutopsMethod.pdf

[43] http://sourceforge.net/projects/metrics/

[44] http://sourceforge.net/projects/junit/

附中文参考文献:

[1] 陈锦富,卢炎生,谢晓东.软件错误注入测试技术研究.软件学报,2009,20(6):1425−1443. http://www.jos.org.cn/1000-9825/3526.htm [doi: 10.3724/SP.J.1001.2009.03526]

[39] 单锦辉,高友峰,刘明浩,刘江红,张路,孙家骕.一种新的变异测试数据生产方法.计算机学报,2008,31(6):1025−1034. [doi: 10.3321/j.issn:0254-4164.2008.06.015]

[40] 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]

张功杰(1979－),男,江苏新沂人,博士生,讲师,主要研究领域为软件测试.

姚香娟(1975－),女,博士,副教授,主要研究领域为基于搜索的软件工程.

巩敦卫(1970－),男,博士,教授,博士生导师,CCF 会员,主要研究领域为软件测试,智能优化算法理论及应用.