基于统计分析的弱变异测试可执行路径生成

党向盈[1),2)] 巩敦卫[1),4)] 姚香娟[3)]

[1)](中国矿业大学信息与电气工程学院 江苏 徐州 221116)

[2)](徐州工程学院信电工程学院 江苏 徐州 221000)

[3)](中国矿业大学理学院 江苏 徐州 221116)

[4)](兰州理工大学电气工程与信息工程学院 兰州 730050)

摘 要 变异测试是一种面向缺陷的软件测试技术,然而高昂的测试代价,影响了其在实际程序测试的应用. Papadakis 等人将某一程序的弱变异测试问题,转化为另一程序的变异语句真分支覆盖问题,以期采用已有的分支覆盖方法,生成变异测试数据.但是,上述方法使得转化后程序包含大量的变异分支,增加了分支覆盖测试数据生成的难度.如果采用合适的方法,约简转化后程序中包含的变异分支,并依所属的路径,对约简之后的变异分支分组,那么,将能够利用已有的路径覆盖测试方法,生成高质量的变异测试数据,从而提高弱变异测试的效率.但是,如何基于某一程序和变异体,生成可执行路径,至今缺乏有效的方法.鉴于此,文中通过考察变异语句真分支之间的相关性,提出了一种用于弱变异测试的可执行路径生成方法,使得覆盖这些路径的测试数据,能够杀死所有的变异体.该方法首先考察变异语句真分支之间的占优关系,约简被占优的变异分支,从而减少变异分支的数量;然后,将非被占优的变异分支插入到该程序,转化为另一被测程序,并基于转化之后的程序,考察同一语句形成的多个变异分支的相关性;通过组合相关变异分支,形成新的变异语句真分支;接着,利用被测语句与新变异语句真分支的相关性,生成包含新变异语句真分支和被测语句的可执行子路径;最后,采用统计分析,基于子路径之间的执行关系,构建并约简相关矩阵,将相关的子路径组合,生成一条或多条可执行路径.将所提方法应用于 9 个基准和工业程序测试中,并与传统方法进行了比较.实验结果表明,所提方法生成了为数较少的可执行路径,且运行时间短;更重要的是,这些可执行路径能够覆盖所有的变异分支.此外,所提方法涉及的样本容量,对生成的可执行路径数有一定的影响,但对程序的运行时间影响较小.

关键词 变异测试;弱变异测试;变异分支;路径覆盖;可执行路径

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.02355

Feasible Path Generation of Weak Mutation Testing Based on Statistical Analysis

DANG Xiang-Ying[1),2)] GONG Dun-Wei[1),4)] YAO Xiang-Juan[3)]

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

[2)](*School of Information and Electrical Engineering, Xuzhou Institute of Technology, Xuzhou, Jiangsu* 221000)

[3)](*School of Science, China University of Mining and Technology, Xuzhou, Jiangsu* 221116)

[4)](*School of Electrical and Information Engineering, Lanzhou University of Technology, Lanzhou* 730050)

Abstract Mutation testing is a fault-based testing technique. The high cost, however, limits its widespread applications in practical testing. Papadakis et al. transformed the problem of weak mutation testing of a program into that of covering the true branches of mutant statements of another program, with the purpose of generating mutation test data by using previous methods of branch coverage. The converted program contains, however, a great number of mutant branches by using the above approach, thus having a difficulty in generating test data that cover these branches. If appropriate methods are employed to reduce the mutant branches in the converted

收稿日期:2015-05-19;在线出版日期:2016-02-26.本课题得到国家"九七三"重点基础研究发展规划项目基金(2014CB046306-2)、国家自然科学基金(61375067,61203304,61573362)资助.**党向盈**,女,1978 年生,博士研究生,副教授,主要研究方向为软件测试. E-mail: dangpaper@163.com. **巩敦卫**(通信作者),男,1970 年生,博士,教授,博士生导师,主要研究领域为基于搜索的软件工程、智能优化与控制. E-mail: dwgong@vip.163.com. **姚香娟**,女,1975 年生,博士,教授,主要研究领域为基于搜索的软件工程.

program, and the reduced mutant branches are grouped according to the paths to which they belong, mutation test data with high quality will be generated by using previous methods of path coverage, hence improving the efficiency of weak mutation testing. Effective methods for generating feasible paths based on a program and its mutants are, however, of absence up to date. In view of this, this paper proposes an approach to generate feasible paths for weak mutation testing by considering the correlation of the true branches of mutant statements, with the purpose of killing all mutants by test data that cover all these feasible paths. To fulfill this task, the dominance relation of the true branches of mutant statements is first determined and employed to reduce the dominated true branches. Following that the non-dominated branches of mutant statements are instrumented into the program to form another program. The true branches of mutant statements that are generated by mutating the same statement are transformed into a new one based on their correlation. Then feasible sub-paths that contain these new branches and the original statement are produced based on the correlation among the original statement and the new branches. Finally, a correlation matrix is generated and reduced based on the execution of these sub-paths using statistical analysis, and one or more feasible paths that contain all these feasible sub-paths are obtained based the correlation matrix. The proposed method is applied to nine benchmarks or industrial programs, and compared with traditional algorithms. Our experimental results demonstrate that the proposed method can generate a small number of paths which cover all the mutant branches, with less time in executing the program. In addition, the sampling size involved in the proposed method has an influence on the number of feasible paths to some degree, whereas slightly impacts time in executing the program.

Keywords mutation testing; weak mutation testing; mutant branch; path coverage; feasible path

1 引 言

软件测试是保证软件质量的重要手段,通过测试,不但能够检测软件可能存在的缺陷,而且能够提高软件的可靠度.在诸多测试技术中,变异测试直接面向程序缺陷,生成测试数据,是提高测试数据充分性的一种有效方法[1].

为了进行变异测试(mutation testing),首先,通过使用变异算子(mutation operator),对程序的某一语句做合乎语法的微小变动,以产生一个新的程序,称该程序为一个变异体(mutant)[2];然后,以某一测试数据,分别执行变异体和原程序,如果两者的输出不同,那么,称测试数据杀死了该变异体,这种变异测试准则称为强变异测试(strong mutation testing)[3];如果两者在变异点执行后,变量状态出现不一致,即可认为该变异体被杀死,那么,这种变异测试准则称为弱变异测试(weak mutation testing)[4].

基于弱变异测试准则,判定某变异体是否被杀死时,不需要完整的执行原程序和变异体,因此,提高了变异测试的效率.Horgan 和 Mathur[5] 的理论分析表明,在特定条件下,基于弱变异测试得到的变异得分(mutation score),与基于强变异测试得到的变异得分基本保持一致.此外,Offutt 和 Lee[6] 的实验结果表明,在多数情况下,弱变异测试可以替代强变异测试.一个程序往往存在很多变异体,为了杀死这些变异体,也需要大量的测试数据;而且,这些测试数据需要同时执行原程序和变异体,因此,变异测试的效率通常很低.为了克服上述不足,Papadakis 等人[7] 通过将变异语句转化为变异分支(mutant branch),并将所有转化后的变异分支,插入到原程序后,形成新的被测程序,这样一来,将原程序的弱变异测试问题,转化为新程序的分支覆盖(branch coverage)问题,以采用已有的分支覆盖方法,解决变异测试问题,从而提高了弱变异测试的效率.

但是,采用上述方法进行弱变异测试转化,将导致新程序包含大量的变异分支,使得分支覆盖测试数据生成问题非常复杂.如果采用合适的方法,约简(reduction)新程序中的变异分支,并对约简之后的变异分支,依所属的路径进行分组,那么,能够将分

支覆盖问题，转化为路径覆盖(path coverage)问题，这样一来，能够减少生成的测试数据，进一步提高弱变异测试的效率.

鉴于此，本文提出一种基于统计分析的弱变异测试可执行路径生成方法. 首先，对同一被测语句进行变异，形成相应的变异分支，并根据这些分支的执行关系，形成新的变异分支；然后，基于新的变异分支与被测语句的相关性，形成可执行子路径；最后，基于统计分析的方法，生成一条或多条包含这些子路径的可执行路径.

本文的贡献主要体现在：(1) 给出了基于同一被测语句变异，形成新变异分支的方法，减少了变异分支的条数；(2) 给出了可执行子路径形成的策略，使得形成的子路径，包含所有的新变异分支；(3) 给出了可执行路径的生成方法，使得生成的可执行路径，包含所有的子路径.

通过上述工作，将弱变异测试问题转化为路径覆盖问题，生成了包含所有变异分支的可执行路径，那么，覆盖这些路径的测试数据，能够杀死相应的变异体，从而提高了变异测试的效率.

本文第 2 节综述相关的工作；第 3 节阐述本文提出的方法，包括基于同一语句变异形成新变异分支、可执行子路径的生成、可执行路径的生成以及实例分析等；第 4 节通过对比实验，评价所提方法的性能；最后，第 5 节总结全文所做的工作，并提出后续的研究问题.

2　相关工作

本节从如下 5 个主要方面，总结已有的研究工作，主要包括变异测试、变异体约简、变异测试数据生成、路径覆盖测试以及弱变异测试转化. 在此基础上，引出本文研究的动机.

2.1　变异测试

变异测试是由 DeMillo 等人[2]提出的一种面向缺陷的测试技术. 进行变异测试时，不仅可以选择注入缺陷的位置和类型，而且可以根据不同层次的测试，选择变异算子，因此，变异测试具有很好的灵活性和针对性[8-10].

进行变异测试时，对原程序的语句，按照句法规则进行一些小的改动，称为变异[2,8]；改动语法的规则，称为变异算子；改动之后的语句，称为变异语句；在变异体中，如果变异语句只有一个，那么，称为一阶变异体[11]；相应的变异测试，称为一阶变异测试. 本文仅研究一阶变异测试，简称变异测试.

在强变异测试准则下，判断一个变异体能否被杀死，需要满足如下 3 个条件[2,12]：(1) 可达性. 测试数据能够执行到变异体中的变异语句；(2) 必要性. 测试数据执行变异语句之后，产生与原程序不同的状态；(3) 充分性. 上述不同的状态，能够导致原程序和变异体的输出不同.

如果对于任何测试数据，执行原程序和变异体后，输出均相同，那么，称该变异体为等价变异体(equivalent mutant)[8].

除了强变异测试准则之外，还有弱变异测试准则，该准则只需要满足可达性和必要性条件，即可认为变异体被杀死. 已有研究表明，满足可达性和必要性条件的测试数据，也能够在很大程度上满足充分性条件[5-6]，因此，弱变异测试是强变异测试的有效替代[13-15].

2.2　变异体约简

为了提高变异测试效率，一种有效的方法是尽可能减少需要杀死的变异体，此即变异体约简[16-17]. 通过变异体约简，减少了需要运行程序(变异体)的次数，提高了变异测试的效率. Just 等人[16]通过约简同类变异算子生成的变异体，减少需要杀死的变异体，从而提高变异测试的效率. Offutt 和 Lee[17]通过采用部分变异算子代替所有的变异算子，生成变异体，以减少需要杀死的变异体. 徐拾义[18]通过与变异体对应条件语句之间的占优关系，约简变异体. 此外，还有学者采用遗传算法[19]，约简变异体.

2.3　变异测试数据生成

基于变异测试准则，生成测试数据，以杀死相应的变异体. Offutt[20]提出了基于约束的测试数据生成方法，生成的测试数据能够检测 98% 的变异体. 但是，该方法受到符号执行技术本身的限制；为了克服上述不足，Offutt 等人[21]、刘新忠等人[13]以及单锦辉等人[22]从不同方面提出了动态域约简方法，提高了测试数据生成的效率. 但是，上述研究工作仅局限于采用传统的约束求解方法，生成变异测试数据.

基于搜索的方法[23]，如遗传算法[24]，也可以用于生成测试数据. 张功杰等人[25]以测试数据集作为决策变量，构造目标函数，并采用集合进化，生成变异测试数据，但是，该方法没有对变异体约简，使得生成变异测试数据的代价比较高.

2.4　路径覆盖的测试

与变异测试相比，结构覆盖测试近年来取得了丰硕的研究成果，该测试方法以覆盖程序的某种结构，如语句、分支，或者路径，为测试目标，生成测试

数据，相应的覆盖准则分别称为语句覆盖、分支覆盖以及路径覆盖.

在这些结构覆盖准则中，路径覆盖最为常用.其中，采用基于搜索的方法，生成测试数据[26-27]，成为近年来软件测试研究的热点.该方法首先将路径覆盖问题，转化为一个数值函数优化问题；然后，采用某一搜索方法，如进化优化方法，生成期望的测试数据.

Bueno等人[28]、Lin等人[29]以及Watkins等人[30]分别利用遗传算法，生成覆盖路径的测试数据.但是，上述方法存在的共同不足是，一次运行遗传算法，仅能生成覆盖一条路径的测试数据.为了克服上述不足，Ahmed等人[31]将多路径覆盖测试数据生成问题，转化为多目标优化问题，使得一次运行遗传算法，能够生成覆盖多条路径的测试数据.但是，当需要覆盖的路径很多时，基于该方法建立的模型，将包含很多目标函数，提高了模型求解的复杂度.后来，文献[32-33]针对很多路径覆盖问题，通过路径分组，简化了优化模型，从而降低了问题求解的难度.

但是，路径覆盖测试面临的问题是，如何生成需要覆盖的目标路径；此外，还需要判断这些路径是否可执行的，这是因为，如果这些路径是不可执行的，将花费很多时间，用于生成覆盖这些路径的测试数据，从而降低了软件测试的效率.Zhang等人[34]基于程序的控制流图，采用深度优先搜索方法，自动生成基本路径集.虽然这些路径能够覆盖控制流图的所有节点和边，但是，有些路径是不可执行的.为了避免生成不可执行的路径，Yan等人[35]根据控制流图，采用广度优先搜索方法，生成可执行的基本路径集.但是，对于大型程序，构造其控制流图的代价是相当大的；此外，在软件测试时，往往只需测试程序的部分代码及其形成的路径，此时，构造完整的控制流图通常是没有必要的.

鉴于此，本文基于选择的被测语句，通过弱变异测试转化，并采用合适的策略，生成包含这些被测语句及其转化后语句的可执行路径集，以期提高软件测试的效率.

2.5 弱变异测试转化

鉴于结构覆盖测试已经取得的成果，Papadakis等人[7]将基于弱变异测试准则杀死变异体的问题，转化为变异条件语句真分支的覆盖问题.为此，首先，对于变异前后的语句 s 和 s'，基于弱变异测试的必要条件，构建变异条件语句“if $s!=s'$”，其真分支为一个标志语句；然后，把这些变异条件语句，插入到原程序的相应位置，形成新的被测程序.那么，能够覆盖新程序变异条件语句真分支的测试数据，一定能够杀死该变异条件语句对应的变异体.这样做的好处是，能够利用已有的分支覆盖测试数据生成方法，生成变异测试数据.

进一步，Papadakis等人[36]提出一种路径的选择策略，以覆盖所有的变异分支.为此，从候选路径集中，选择合适的路径，使得覆盖这些路径的测试数据，也能够覆盖变异分支，从而通过已有的路径覆盖测试数据生成方法，生成杀死变异体的测试数据.但是，由于该方法没有约简变异体，使得新程序包含大量的变异分支，从而增加了程序的复杂度，降低了变异测试数据生成的效率.

3 提出的方法

对于一个实际的被测程序，尽管该程序的代码可能很多，但是，程序的缺陷往往仅集中于某些代码范围内，如果针对所有的语句均实施测试，那么，将会花费很大的测试代价.鉴于此，可以选择出现缺陷概率比较大的语句[10]进行测试，对于提高测试效率，是非常有帮助的.

现在我们考虑上述语句的变异测试问题.采用Papadakis等人[7]思想，将弱变异测试转化为分支覆盖问题，我们研究发现，在新的被测程序中，被插入的这些变异分支之间存在一定的关联，体现为多个变异分支分布在同一条路径上.因此，我们考虑基于变异分支所在的路径，对上述变异分支分组，那么，将弱变异测试问题，转化为路径覆盖问题，以期提高变异测试的效率.

3.1 思 想

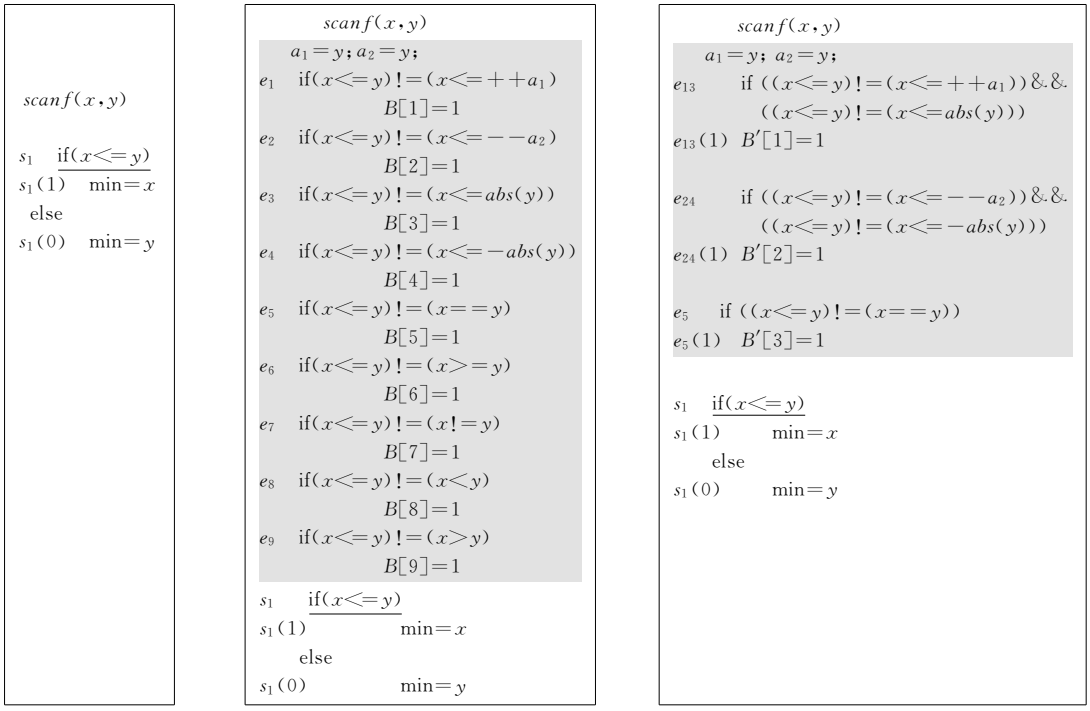
首先，基于对同一被测语句变异，形成的多个变异分支，采用静态分析，根据这些分支的执行关系，形成新的变异分支；然后，基于新的变异分支与被测语句的相关性，形成一条或多条可执行的子路径；最后，基于统计分析方法，根据这些子路径的执行关系，自动生成一条或多条可执行路径.

3.2 基于同一语句变异形成新变异分支

本节考虑针对同一被测语句变异，形成新的变异分支.首先，对同一语句进行变异，基于变异分支之间的执行关系[37]，进行分组；然后，基于每组变异分支所属条件语句的语义关系，形成新的变异分支.在 s_i 之前插装变异分支后形成新程序，插装的变异分支分别记为 $e_1,e_2,\cdots,e_p,\cdots$，这些变异分支的真分支分别记为 $e_1(1),e_2(1),\cdots,e_p(1),\cdots$.记被测语句

为 s_i，如果该语句是条件语句，记其真分支为 $s_i(1)$，假分支为 $s_i(0)$.

下面通过一个例子，说明变异分支分组策略和新的变异分支形成过程. 如图 1(a)为被测程序.



(a) 被测程序　　(b) 插装变异分支后的新程序　　(c) 插装新变异分支后的新程序

图 1　示例程序

对被测语句“$x<=y$”实施关系运算符替换(Relation Operator Replacement，ROR)、算术运算符替换(Arithmetic Operator Replacement，AOR)和单目运算符替换(Unary Operator Replacement，UOI)等 3 种变异算子[17]，其中，ROR 包含的变异算子为$\{(u,v)\mid u,v\in\{>,>=,<,<=,==,!=\}\wedge u\neq v\}$，AOR 包含的变异算子为$\{(u,v)\mid u,v\in\{+,-,*,/,\%\}\wedge u\neq v\}$，UOI 包含的变异算子为$\{(v,--v),(v,++v),(v,v--),(v,v++)\}$. 实施这 3 种变异算子之后，被测语句“$x<=y$”共生成 11 个变异语句.

首先，对 9 个非等价变异体对应的变异语句转化为相应的变异分支，并插装到原程序的被测语句“$x<=y$”之前；然后，分析新程序中 9 个变异分支之间的占优关系，由文献[37]可知，当一个变异分支执行时，其他一或多个变异分支一定执行，记该变异分支占优其他变异分支，当程序的某一输入，使得变异分支 e_1、e_3 或 e_5 执行时，变异分支 e_6 一定执行，那么，e_6 是被占优分支. 类似的，变异分支 e_7、e_8 和 e_9 也是被占优分支. 约简这些被占优分支之后，5 个非被占优变异分支为 $e_1,e_2,\cdots,e_5$. 如图 1(c)的灰色部分所示.

然后，根据这些非被占优变异分支之间的执行关系，对 5 个变异分支分组. 如图 1(b)所示，根据执行关系，考虑变异分支 e_1 和 e_3 的条件谓词表达式，这 2 个条件谓词表达式并不矛盾，因此，可以将这 2 个变异分支分为一组；类似的，将变异分支 e_2 和 e_4 分为一组；变异分支 e_5 单独为一组.

最后，考虑上述 3 组变异分支，在各自组内形成新的变异分支. 对于 e_1 和 e_3 所在的组，通过“逻辑与”的方式，把其条件谓词表达式连接起来，新形成的变异分支为 e_{13}：

$$\begin{aligned}&\text{if }((x<=y)!=(x<=++a_1))\&\&\\&\quad((x<=y)!=(x<=abs(y)))\\&\quad B'[1]=1.\end{aligned}$$

类似的，基于变异分支 e_2 和 e_4，新形成的变异分支为 e_{24}；新形成的变异分支如图 1(c)所示.

以这种方式分组变异分支和形成新变异分支，有利于生成数量较少的可执行路径. 但是，通过“逻辑与”的连接方式，可能增加了包含这些新变异分支路径覆盖的难度. 解决该问题的途径之一是，利用比较成熟的路径覆盖技术，生成测试数据，以降低测试数据生成的代价，关于该内容的研究，已经超出了本文的范围.

3.3 可执行子路径的生成

不引起混淆的情况下，被测语句 s_i 之前插装的新变异分支仍记为 $e_1,e_2,\cdots,e_p,\cdots$，它们的真分支分别记为 $e_1(1),e_2(1),\cdots,e_p(1),\cdots$. 现在由这些新变异分支与被测语句，生成可执行子路径. 如果条件语句的谓词表达式，那么，考察上述变异分支 e_p 的条件语句与该条件语句 s_i 之间的相关性. 如果它们是真真相关的[38]，那么，基于变异真分支和被测语句的真分支，形成一条子路径，记为 $(e_p(1),s_i(1))$；如果它们是真假相关的[38]，那么，基于变异真分支和被测语句的假分支，形成一条子路径，记为 $(e_p(1),s_i(0))$. 如果被测语句不是条件语句的谓词表达式，那么，直接基于变异真分支与该被测语句，形成一条子路径，记为 $(e_p(1),s_i)$.

通过上述方法，对于每一新变异分支，与被测语句，能够形成一条子路径. 进一步，根据变异分支的可执行性，判定子路径是否是可执行的，从而得到可执行的子路径.

类似方法，对于所有变异分支 $e_1,e_2,\cdots,e_p,\cdots$，与被测语句 s_i，形成的可执行子路径集合为

$$\{a_{i1},a_{i2},\cdots,a_{ip},\cdots\}=\{(e_1(1),s_i(b)),(e_2(1),s_i(b)),\cdots,(e_p(1),s_i(b)),\cdots\},\ b\in\{*,0,1\},$$

取 $*$、1 或 0，其中“$*$”表示为被测语句既没有真分支也没有假分支. 因此，$s_i(b)$ 分别表示为被测语句，被测语句的真分支、或被测语句的假分支.

对于图 1 的示例程序，由于新变异分支 e_{13} 的条件谓词表达式“$((x<=y)!=(x<=++a_1))\&\&((x<=y)!=(x<=abs(y)))$”与被测语句 s_1 的条件谓词表达式“$(x<=y)$”在语义上是矛盾的，因此，e_{13} 和 s_1 真假相关，形成的子路径为 $a_{11}=(e_{13}(1),s_1(0))$；类似的，$e_{24}$ 和 s_1 真真相关，形成的子路径为 $a_{12}=(e_{24}(1),s_1(1))$；$e_5$ 和 s_1 真真相关，形成的子路径为 $a_{13}=(e_5(1),s_1(1))$. 由于 e_{13}、e_{24} 和 e_5 均为可执行变异分支，因此，它们所属的子路径 $a_{11}=(e_{13}(1),s_1(0))$、$a_{12}=(e_{24}(1),s_1(1))$ 和 $a_{13}=(e_5(1),s_1(1))$ 也是可执行的，从而生成的可执行的子路径集合为 $\{a_{11},a_{12},a_{13}\}=\{(e_{13}(1),s_1(0)),(e_{24}(1),s_1(1)),(e_5(1),s_1(1))\}$.

3.4 可执行路径的生成

假设原程序有 n 条被测语句，其第 i 条被测语句为 $s_i(i=1,2,\cdots,n)$，采用第 3.2 和 3.3 节的方法，得到的可执行子路径集合为 $A_i=\{a_{i1},a_{i2},\cdots,a_{i|A_i|}\}$. 对于所有的被测语句，可执行子路径的集合为 $A=\{A_1,A_2,\cdots,A_n\}$，容易知道，A 包含的子路径条数为 $|A|=\sum_{i=1}^{n}|A_i|$. 由于 A_i 的子路径之间不能同时执行，因此，需要将 A_i 与 $A_j(j=1,2,\cdots,n,j\neq i)$ 的子路径进行组合，生成可执行路径.

下面考虑插入新变异分支之后的新程序. 为了使生成的可执行路径比较少，必须充分考虑子路径之间的执行关系. 首先，基于子路径之间的执行关系，构建一个相关矩阵；然后，根据矩阵中元素的取值，约简该矩阵，同时，使得可执行子路径集合包含的元素最少；最后，根据与某子路径相关的子路径条数，依一定的顺序，将相关的子路径组合起来，生成一或多条可执行路径.

(1) 相关矩阵的构建

为了构建相关矩阵，考虑 A 的 2 个子路径 a_{ip}，a_{jk}，$i\neq j$. 对于程序的某一输入 x，a_{ip} 可能被穿越，也可能不被穿越，a_{jk} 也是如此. 为了反映这 2 个子路径被穿越的可能性，定义如下 2 个随机变量：

$$\mu_{ip}(x)=\begin{cases}1, & x\text{ 穿越子路径 }a_{ip}\\ 0, & x\text{ 没有穿越子路径 }a_{ip}\end{cases},$$

$$\mu_{jk}(x)=\begin{cases}1, & x\text{ 穿越子路径 }a_{jk}\\ 0, & x\text{ 没有穿越子路径 }a_{jk}\end{cases}.$$

显然，变量 μ_{ip} 和 μ_{jk} 服从(0,1)分布. 对于变量 μ_{ip} 和 μ_{jk}，若给定 μ_{ip}，且 $P(\mu_{ip}=i)>0$ 时，μ_{jk} 的条件分布律为

$$P(\mu_{jk}=j\,|\,\mu_{ip}=i)=\frac{P(\mu_{jk}=j,\mu_{ip}=i)}{P(\mu_{ip}=i)},\ i,j\in\{0,1\}\tag{1}$$

式(1)反映了变量 $\mu_{ip}=i$ 发生的条件下 $\mu_{jk}=j$ 发生的概率.

如果随机变量 μ_{ip} 和 μ_{jk} 的值存在一定的联系；那么，子路径 a_{ip} 和 a_{jk} 的执行也具有一定的相关，反之亦然. 因此，可以利用式(1)的 μ_{ip} 和 μ_{jk} 的条件分布率，考察 a_{ip}，a_{jk} 执行的相关程度.

在程序的输入域中采样 R 次，采样值分别为 $x_1,x_2,\cdots,x_R$，对于每一采样值，根据子路径 a_{ip} 和 a_{jk} 是否被穿越，计算上述随机变量的 μ_{ip} 和 μ_{jk} 值. 子路径 a_{ip} 和 a_{jk} 之间的相关度，记为 $\alpha_{ip,jk}$，可以定义如下：

$$\alpha_{ip,jk}=\frac{\sum_{x\in\{x_1,x_2,\cdots,x_R\}|\mu_{ip}(x)=1}\mu_{jk}(x)}{\sum_{x\in\{x_1,x_2,\cdots,x_R\}}\mu_{ip}(x)}\tag{2}$$

由式(2)可以看出，两个子路径的相关度的取值范围是 $0\leq\alpha_{ip,jk}\leq1$；当 $\alpha_{ip,jk}=1$ 时，表示同一或两个不同的子路径最相关；当 $\alpha_{ip,jk}=0$ 时，表示这两个

不同的子路径最不相关.

由式(2),类似的,可以得到所有子路径之间的相关度,对于可执行子路径集合 $A_i=\{a_{i1},a_{i2},\cdots,a_{i|A_i|}\}$,由这些相关度的值,够构建如下的相关矩阵:

$$\boldsymbol{\Lambda}=\begin{array}{c} \\ a_{11} \\ a_{12} \\ \\ a_{ip} \\ \\ a_{n|A_n|} \end{array}\begin{array}{c} \begin{array}{ccccccc} a_{11} & & a_{12} & & a_{jk} & & a_{n|A_n|} \end{array} \\ \begin{bmatrix} \alpha_{11,11} & \alpha_{11,12} & \cdots & \alpha_{11,jk} & \cdots & \alpha_{11,n|A_n|} \\ \alpha_{12,11} & \alpha_{12,12} & \cdots & \alpha_{12,jk} & \cdots & \alpha_{12,n|A_n|} \\ & \cdots & & \cdots & & \cdots \\ \alpha_{ip,11} & \alpha_{ip,12} & \cdots & \alpha_{ip,jk} & \cdots & \alpha_{ip,n|A_n|} \\ & \cdots & & \cdots & & \cdots \\ \alpha_{n|A_n|,11} & \alpha_{n|A_n|,12} & \cdots & \alpha_{n|A_n|,jk} & \cdots & \alpha_{n|A_n|,n|A_n|} \end{bmatrix} \end{array}.$$

由矩阵 $\boldsymbol{\Lambda}$ 可以看出,(1)矩阵 $\boldsymbol{\Lambda}$ 的对角线元素均为 1,即子路径与自身的相关度为 1,表明子路径与自身最相关;(2)$\alpha_{ip,jk}$ 的值越大,那么,子路径 a_{ip}、a_{jk} 的相关度越高;特别的,$\alpha_{ip,jk}=1$ 时,表示两个不同的子路径是最相关的,即子路径 a_{ip} 的执行,必然导致 a_{jk} 也执行;(3)$\alpha_{ip,jk}$ 的值越小,那么,子路径 a_{ip}、a_{jk} 的相关度越低;特别的,$\alpha_{ip,jk}=0$ 时,表示两个不同的子路径是最不相关的,即子路径 a_{ip} 的执行,必然导致 a_{jk} 不执行.

(2)相关矩阵的约简

当不同子路径之间相关度 $\alpha_{ip,jk}=1$ 时,可以约简相关矩阵 $\boldsymbol{\Lambda}$.首先,考察 $\boldsymbol{\Lambda}$ 的第 1 行包含的元素,该行元素反映了子路径 a_{11} 与 a_{jk}($j=1,2,\cdots,n$;$k=1,2,\cdots,|A_n|$)的相关度.如果 $\exists\alpha_{11,jk}$,$j\neq1$;$\ni:\alpha_{11,jk}=1$,此时,子路径 a_{11} 执行,必然导致 a_{jk} 也执行,那么,从 $\boldsymbol{\Lambda}$ 中删除子路径 a_{jk} 对应的列和行,同时从集合 A 中删除 a_{jk};然后,考察约简后相关矩阵的第 2,3,… 行元素,采用类似的方法,继续约简相关矩阵和子路径集合,直到所有行的元素均被考察为止.

在不引起混淆的情况下,仍记约简后的相关矩阵为 $\boldsymbol{\Lambda}$,子路径集合为 A.

(3)可执行路径的生成

为了生成一或多条可执行路径,首先,在约简后的矩阵 $\boldsymbol{\Lambda}$ 中,考察与子路径具有相关的其他子路径的条数,按照一定的顺序,选择基准子路径;然后,针对每一基准子路径,将与该基准子路径相关的子路径结合,生成一条可执行路径;最后,将该可执行路径包含的子路径从集合 A 中删除,直到 A 不包含任何子路径为止.

下面,分别给出基准子路径的选择和可执行路径生成的方法.为了选择基准子路径,首先,考虑 $\boldsymbol{\Lambda}$ 中子路径 a_{ip} 对应的行,记录该行中所有 $0<\alpha_{ip,jk}<1$ 对应的子路径 a_{jk},并统计这些子路径的个数,记为 n_{ip};然后,考虑 $\boldsymbol{\Lambda}$ 的所有行,能够得到集合 $\{n_{ip},\cdots,n_{i'p'},\cdots,n_{n|A_n|}\}$,该集合反映了与每一子路径相关的子路径条数,若 n_{ip} 越小,与子路径 a_{ip} 相关的子路径越少,那么,生成可执行路径时,a_{ip} 可供利用的子路径越少.因此,为了生成比较少的可执行路径,有必要优先选择集合 $\{n_{ip},\cdots,n_{i'p'},\cdots,n_{n|A_n|}\}$ 中最小的元素,设 n_{ip} 为该集合最小元素,那它所对应的子路径 a_{ip} 为基准子路径.

为了生成一条可执行路径,考察与基准子路径 a_{ip} 相关的所有子路径.如果与基准子路径 a_{ip} 相关的子路径有 0 条,那么,生成一条基准子路径自身的可执行路径(a_{ip}).如果与该基准子路径相关的子路径只有 1 条,记与基准子路径相关的子路径为 a_{jk};将子路径 a_{ip} 和 a_{jk} 连接起来,生成一条可行性路径(a_{ip},a_{jk}).如果与该基准子路径相关的子路径多于 1 条,记与基准子路径 a_{ip} 相关的子路径为 a_{jk} 和 a_{lm},如果 $0<\alpha_{jk,lm}<1$ 且 $0<\alpha_{ip,lm}<1$,那么,将(a_{ip},a_{jk})与 a_{lm} 连接起来,生成一条可执行路径(a_{ip},a_{jk},a_{lm}).

类似的,能够生成包含更多子路径的可执行路径.那么,由所有子路径生成的可执行路径集合为 $\{(a_{ip},a_{jk},a_{lm},\cdots),(a_{i'p'},a_{j'k'},a_{l'm'},\cdots),\cdots\}$.

3.5　实例分析

下面通过程序 Triangle 说明采用第 3 节的方法,生成可执行路径的过程.图 2(a)为 Triangle 的源代码.在程序的前、中和后部,分别选取 3 条被测语句,并从传统变异算子[17]中选择 2 种,实施变异操作,得到 21 个变异体,其中 16 个非等价变异体,采用文献[7]方法,生成相应的变异分支,如表 1 所列.

表 1　分组的变异分支与形成的新变异分支

被测语句	变异算子	变异体个数	等价变异体个数	被占优变异分支条数	新变异分支条数	新变异分支
$x>z$	ROR UOI	5 4	3	2	3	$e_{11}(1),e_{12}(1),e_{13}(1)$
$x+y<=z$	AOR ABS	4 2	1	2	2	$e_{21}(1),e_{22}(1)$
$(x==y)\&\&(y==z)$	ROR LCR	5 1	1	2	2	$e_{31}(1),e_{32}(1)$
总计	—	21	5	6	7	—

(a) 源代码 (b) 被测语句转化后的新变异分支 (c) 插入新变异分支的新程序

图 2 Triangle 被测程序

采用第 3.2 节的方法,对这些变异分支分组,并形成 7 个新的变异分支,如图 2(b)所示.将这些新变异分支插入到原程序中,形成新的被测程序,如图 2(c)所示.采用第 3.3 节的方法,形成的可执行子路径集合为

$$H=\{A_1,A_2,A_3\}=\{\{a_{11},a_{12},a_{13}\},\{a_{21},a_{22}\},\{a_{31},a_{32}\}\}$$
$$=\{\{(e_1(1)),(e_2(1)),(e_3(1),1(1))\},\{(e_4(1),2(1)),(e_5(1),2(0))\},\{(e_6(1),3(1)),(e_7(1),3(0))\}.$$

取 $R=3000$ 时,采用第 3.4 节(1)的方法,得到如下相关矩阵 $\boldsymbol{\Lambda}$:

$$\boldsymbol{\Lambda}=\begin{array}{c} \\ a_{11}\\ a_{12}\\ a_{13}\\ a_{21}\\ a_{22}\\ a_{31}\\ a_{32}\end{array}\begin{array}{c}\begin{array}{ccccccc} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{31} & a_{32}\end{array}\\ \begin{bmatrix} 1 & 0 & 0 & 0.34 & 0.41 & 0.05 & 0.36\\ 0 & 1 & 0 & 0.44 & 0.52 & 0 & 0.01\\ 0 & 0 & 1 & 0.47 & 0.51 & 0 & 0\\ 0.02 & 0.64 & 0.02 & 1 & 0 & 0 & 0\\ 0.02 & 0.64 & 0.01 & 0 & 1 & 0.01 & 0.02\\ 1 & 0 & 0 & 0 & 1 & 1 & 0\\ 0.66 & 0.34 & 0 & 0 & 1 & 0 & 1\end{bmatrix}\end{array}.$$

由 $\boldsymbol{\Lambda}$ 可知,子路径 a_{31} 与 a_{11} 的相关度 $\alpha_{31,11}=1$,采用第 3.4 节(2)的方法,从 $\boldsymbol{\Lambda}$ 中删除 a_{11} 对应的行和列,并从集合 A 中删除子路径 a_{11}.类似的,从 $\boldsymbol{\Lambda}$ 中删除 a_{22} 对应的行和列,并从 A 中删除子路径 a_{22}.得到约简后的相关矩阵如下:

$$\boldsymbol{\Lambda}=\begin{array}{c} \\ a_{12}\\ a_{13}\\ a_{21}\\ a_{31}\\ a_{32}\end{array}\begin{array}{c}\begin{array}{ccccc} a_{12} & a_{13} & a_{21} & a_{31} & a_{32}\end{array}\\ \begin{bmatrix} 1 & 0 & 0.44 & 0 & 0.01\\ 0 & 1 & 0.47 & 0 & 0\\ 0.64 & 0.02 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\\ 0.34 & 0 & 0 & 0 & 1\end{bmatrix}\end{array}.$$

约简后的子路径集合为 $A=\{\{a_{12},a_{13}\},\{a_{21}\},\{a_{31},a_{32}\}\}$.

对于约简后的相关矩阵 $\boldsymbol{\Lambda}$ 和子路径集合 A,采用第 3.4 节(3)的方法,分别记录与子路径 a_{12},a_{13},a_{21},a_{31},a_{32} 相关的其他子路径,并统计这些相关子路径的条数,如表 2 所列.

表 2　子路径及与其相关的子路径

子路径	相关子路径	路径条数
a_{12}	a_{21}，a_{32}	2
a_{13}	a_{21}	1
a_{21}	a_{12}，a_{13}	2
a_{31}	0	0
a_{32}	a_{12}	1

首先，选取 a_{31} 作为基准子路径，生成的可执行路径为(a_{31})，并将 a_{31} 从集合 A 中删除；然后，选取 a_{13} 作为基准子路径，生成的可执行路径为(a_{13}，a_{21})，并将 a_{13}，a_{21} 从集合 A 中删除；最后，选取 a_{12} 作为基准子路径，生成的可执行路径为(a_{12}，a_{32})，并从集合 A 中删除 a_{12}，a_{32}. 此时，$A=\varnothing$，生成可执行路径结束. 经过以上步骤，生成的可执行路径集合为 $\{(a_{31}),(a_{12},a_{32}),(a_{13},a_{21})\}$.

4　实　验

本节通过实验，验证所提方法的有效性. 首先，提出实验需要验证的问题；然后，给出实验所用的程序；接着，描述实验过程；最后，给出实验结果和分析.

4.1　需要验证的问题

为了说明本文方法的有效性，需要验证如下问题：

(1) 本文方法生成的路径是否是可执行的？通过采用随机法生成测试数据，考察生成的测试数据能否覆盖这些路径，说明生成路径的可行性.

(2) 本文方法生成的可执行路径，能否覆盖所有的变异分支？通过覆盖可执行路径的测试数据，对变异分支的覆盖率反映.

(3) 本文方法生成的可执行路径是否很少？通过子路径之间的随机结合和排列组合，形成的可执行路径的条数，与本文方法生成可执行路径比较，说明可执行路径的多少.

(4) 计算相关矩阵所需的样本容量，是否会影响本文方法的性能？通过不同样本容量，本文方法生成的可执行路径条数和运行时间，反映样本容量的影响.

4.2　被测程序

选取 9 个基准和工业程序作为被测程序，验证本文方法的有效性，这些程序的基本信息如表 3 所列，其中，程序 T1～T5 常用于变异测试，选自文献[39-40]；T6 是一个简单的 UNIX 通用程序，选自文献[41]；此外，为了评价本文方法在工业程序测试中的适用性，选择西门子系统的程序 T7、T8 和 T9 为被测程序，其源代码可以从网站 http://sir.unl.edu/portal/index.php 免费下载；T7 选择的是 Space[42-43] 的 Fixgramp 函数；T8、T9 规模比较大些，包含的分支和函数比较多，且这些函数的类型复杂，既有嵌套调用，又有递归调用，是多个文献[44-46] 的被测程序. 需要说明是，在表 3 第 4 列的输入空间，程序 T9 输入变量采用 ACSII 编码方式，在程序中可以将 ACSII 值和对应的字符进行转化.

表 3　被测程序的基本信息和参数

ID	被测程序	代码行数	输入空间	程序结构	程序功能	被测语句条数	变异体个数	等价变异体个数	变异分支条数	新变异分支条数
T1	Triangle	35	$[1,64]^3$	16 个分支语句，1 个函数	三角形分类	3	21	5	16	7
T2	Mid	26	$[-64,64]^6$	10 个分支语句，1 个函数	求三个数中值	3	24	8	16	8
T3	Profit	24	[1,100]	10 个分支语句，1 个函数	销售人员的提成	3	20	7	13	5
T4	Day	42	$[1,3000]\times[1,12]\times[1,31]$	23 个分支语句，1 个函数	计算某天的次序	3	18	9	9	3
T5	Insert	35	$[-20,60]$	12 个分支语句，1 个函数	插入排序	5	21	8	13	6
T6	Calendar	137	[1000,3000]	29 个分支语句，4 个函数	计算一年的日历	10	68	38	30	8
T7	SpaceFixgramp	91	$[0\sim127]^9$	8 个分支语句，2 个函数	矩阵语言翻译器	6	39	20	19	7
T8	Totinfo_2.0	406	$[0,5]^2[-128,128]^{m\times n}$ $m,n\in\{0,1,\cdots,5\}$	34 个分支语句，7 个函数	统计表中信息，表中数据是整数.	20	141	53	88	23
T9	Replace	564	$[32,126]^m\times[32,126]^n\times[32,126]^p$ $m,n,p\in\{0,1,\cdots,5\}$	66 个分支和循环语句，20 个函数	模式匹配	20	106	45	61	34
总计	—	796	—	—		73	458	193	265	101

4.3　实验过程

实验的硬件条件为：英特尔酷睿双核 3.10 GHz CPU、2 GB 内存；软件采用 Microsoft Windows XP SP3 操作系统和 VC++开发环境.

实验中,根据被测程序的代码行数,分别选择3～20个不同的被测语句进行变异.根据程序的结构和被测语句的类型,选择被测语句的变异算子,实施变异操作,以生成变异体.对非等价变异体采用文献[7]的方法,生成相应的变异分支.对于每个被测程序,首先,将生成的变异分支,插入到原程序中,形成新的被测程序 P';然后,基于第3.2节的方法,对 P' 中的变异分支形成新分支,插入到原程序中,形成另一被测程序 P''.

对于程序 P'',首先,给定某一样本容量,采用第3.3、3.4节的方法,生成一条或多条路径;然后,以所提方法生成的一条或多条路径为目标路径,采用随机测试数据生成方法[45],在被测程序的输入空间内随机采样,直到生成覆盖这些路径的测试数据,或迭代到一定代数,终止程序,并考察这些测试数据覆盖的变异分支;最后,通过子路径之间随机结合和排列组合结合,形成的可执行路径,与本文方法生成的可执行路径比较,说明后者路径条数是否较少.

此外,设置不同的样本容量,考察其对生成的可执行路径数目和运行时间的影响.

4.4 实验结果与分析

为了回答第4.1节提出的问题,设计了4组实验.现在给出实验结果,并进行分析.

(1)生成路径的可执行性

为了验证本文方法生成的路径是可执行的,考虑上述9个被测程序.由表3可知,这些被测程序共产生了458个变异体,其中265个非等价变异体;将这些非等价变异体转化为相应的变异分支,形成101条新的变异分支.

采用第3.4节的方法,生成了39条路径,如表4

表4 随机法生成覆盖可执行路径的测试数据

ID	路径编号	生成可执行路径	迭代次数	测试数据	覆盖率/%
T1	1	(a_{12},a_{32})	106	57,57,60	100
	2	(a_{13},a_{21})	124	22,50,21	100
	3	(a_{31})	2045	23,23,23	100
T2	1	(a_{11},a_{31})	193	−8,−8,−1	100
	2	(a_{21})	254	−2,−3,−3	100
	3	(a_{22})	0	−23,−44,−51	100
	4	(a_{13})	36	4,5,55	100
	5	(a_{23},a_{32})	1357	15,14,15	100
T3	1	(a_{11})	3	1	100
	2	(a_{12})	145	10	100
	3	(a_{21})	91	40	100
	4	(a_{31})	1	68	100
	5	(a_{32})	37	100	100
T4	1	(a_{21})	0	1041,12,11	100
	2	(a_{22})	3	1452,6,29	100
T5	1	(a_{11})	13	28	100
	2	(a_{12})	27	30	100
	3	(a_{13})	50	27	100
	4	(a_{21})	69	−16	100
T6	1	(a_{22})	21	1624	100
	2	(a_{21},a_{41})	0	1041	100
T7	1	(a_{51})	1	2,78,110,41,19,48,57,48,8	100
	2	(a_{61})	0	0,83,4,73,31,26,98,80,105	100
	3	(a_{62})	340	0,0,93,43,127,125,5,25,68	100
T8	1	(a_{11},a_{31},a_{61})	1503	1,4,−59,−67,−97,−115	100
	2	(a_{12})	42	3,3,114,−85,59,118,95,−95,−60,29,−102	100
	3	(a_{41})	62	1,1,0	100
	4	(a_{13})	4	5,2,−11,9,3,−62,69,−39,−124,87,106,55	100
	5	(a_{21})	110	1,1,104	100
	6	(a_{113})	2610	2,0,0	100
	7	(a_{22})	13	1,3,112,61,75	100
T9	1	$(a_{11},a_{22},a_{52},a_{63},a_{101},a_{142})$	989	"□", "*EED*", "*nd*|−*a*"	100
	2	$(a_{12},a_{23},a_{42},a_{91})$	1743	"%∗*Q*", "1*ry*", "3#*gd*"	100
	3	(a_{31},a_{93},a_{113})	2247	"□∗?", "*m*@*mpq*", "6*er*□−"	100
	4	(a_{33},a_{82})	771	"4*Te*", "7@*a*8", "1,/1*A*"	100
	5	(a_{41},a_{71},a_{122})	124	"□∗−", "□&-]", "6*er* □*sd*"	100
	6	(a_{51})	2945	"□∗−", "@*tds*", "*rer* □3"	100
	7	(a_{111},a_{121})	54	"□9∗, "□[%0&-", "1%#1*b*"	100
	8	(a_{133})	536	"□∗?", "*OPE*", "*er*&-∗"	100
总计	39	—	—	—	100

的第 3 列所示. 采用随机法，生成的覆盖路径的测试数据，如表 4 的第 5 列所示，其中 T9 的测试数据为 ACSII 值对应的字符，“□”代表空格. 此外，表 4 的第 4 列还给出了生成这些测试数据的迭代次数；第 6 列统计了生成路径的覆盖率.

由表 4 可以看出，生成路径的覆盖率均为 100%. 这说明，采用本文方法生成的路径，确实是可执行的.

(2) 可执行路径对变异分支的覆盖

为了验证采用本文的方法，生成的可执行路径能否覆盖新程序的所有变异分支. 首先，考察程序 T1. 由表 4 可知，该程序共生成 3 条可执行路径. 为了验证这 3 条路径能否覆盖该程序的 16 条变异分支，针对每一路径，考察该路径覆盖的变异分支，结果如表 5 所列，这 3 条可执行路径覆盖了程序 T1 的所有变异分支. 其中，重复覆盖的变异分支有 3 条. 然后，对于其他被测程序，采用上述方法，能够得到可执行路径覆盖的变异分支，对于重复覆盖的变异分支，只记录 1 次，结果如表 6 所列.

表 5　被测程序 T1 中变异分支的覆盖情况

路径编号	变异分支条数	变异分支

1	9	if (x>z)!=(x<z)…
		if (x>z)!=(x!=z)…
		if ((x+y<=z)!=(x−y<=z))…
		if ((x+y<=z)!=(x/y<=z))…
		if ((x+y<=z)!=(x%y<=z))…
		if (((x==y)&&(y==z))!=
		((x==y)&&(y<=z)))…
		if (((x==y)&&(y==z))!=
		((x==y)&&(y<z)))…
		if (((x==y)&&(y==z))!=
		((x==y)&&(y!=z)))…
		if (((x==y)&&(y==z))!=
		((x==y)‖(y==z)))…

2	3	if ((x>++aa2)!=(x>z))…
		if ((x+y<=z)!=(x*y<=z))…
		if ((x+y<=z)!=
		(x+y<=−abs(z))…

3	7	if ((x>−−aa1)!=(x>z))…
		if (x>z)!=(x>=z)…
		if (x>z)!=(x==z)…
		if ((x+y<=z)!=(x−y<=z))…
		if ((x+y<=z)!=(x/y<=z))…
		if ((x+y<=z)!=(x%y<=z))…
		if ((x==y)&&(y==z))!=
		((x==y)&&(y>z))…

表 6　其他被测程序变异分支的覆盖情况

ID	路径编号	变异分支条数
T2	1	6
	2	5
	3	2
	4	1
	5	2
总计	—	16
T3	1	2
	2	2
	3	5
	4	2
	5	2
总计	—	13
T4	1	7
	2	2
总计	—	9
T5	1	2
	2	2
	3	3
	4	6
总计	—	13
T6	1	23
	2	7
总计	—	30
T7	1	3
	2	5
	3	11
总计	—	19
T8	1	68
	2	6
	3	5
	4	1
	5	3
	6	2
	7	3
总计	—	88
T9	1	10
	2	19
	3	4
	4	4
	5	8
	6	1
	7	14
	8	1
总计	—	61

由表 3、表 5 和表 6 可知，这些程序的可执行路径，能够覆盖它们所有变异分支. 上述实验结果表明，采用本文方法生成的可执行路径，能够覆盖程序的所有变异分支.

(3) 生成的可执行路径条数

为了验证采用本文的方法，生成的可执行路径条数比较少，与另外 2 种路径生成方法比较，一种方法是随机结合法，该方法通过随机结合约简后的相关矩阵中的子路径，生成可执行路径；另一种方法是排列组合法，该方法基于子路径集合，通过排列组合方法，生成可执行路径.

所提方法在形成新变异分支和生成子路径时，基于同一语句形成的变异分支比较少，程序局部结构比较简单，因此，采用静态分析比较准确；此外，将新变异分支插装到原程序中，将增加测试的代价. 鉴于此，从子路径生成可执行路径这一步，选择随机法和组合法，与所提方法进行比较.

以程序 T1 为例，说明采用随机结合法，生成可执行路径的过程. 由第 3.5 节的约简后矩阵可知，共有 5 条子路径. 通过随机方式结合这 5 条子路径，形成一系列路径，这些路径中包含很多不可执行路径，最终得到了 5 个可执行路径集合，每一个集合包含了若干条可执行路径. 对于其他 8 个被测程序，采用上述方法，也能够得到相应的可执行路径集合，如表 7 所列.

由表 4 和表 7 可知，① 子路径之间的随机结合得到的路径集合，包含的可执行路径不同，这些路径的条数也不尽相同，其中，针对每一被测程序，第 1 个集合包含的可执行路径最多，最后 1 个集合包含的可执行路径最少；② 采用本文方法得到的可执行路径，与最后 1 个集合包含的可执行路径相同.

表 7 采用随机法生成的路径集

ID	路径集编号	可执行路径集	可执行路径条数
T1	1	$\{(a_{12}),(a_{13}),(a_{21}),(a_{31}),(a_{32})\}$	5
	2	$\{(a_{12},a_{21}),(a_{13}),(a_{31}),(a_{32})\}$	4
	3	$\{(a_{12},a_{32}),(a_{21}),(a_{31}),(a_{13})\}$	4
	4	$\{(a_{13},a_{21}),(a_{12}),(a_{31}),(a_{32})\}$	4
	5	$\{(a_{12},a_{32}),(a_{21},a_{13}),(a_{31})\}$	3
T2	1	$\{(a_{11}),(a_{13}),(a_{21}),(a_{22}),(a_{23}),(a_{31}),(a_{32})\}$	7
	2	$\{(a_{13}),(a_{21}),(a_{23}),(a_{22})(a_{11},a_{31}),(a_{32})\}$	6
	3	$\{(a_{13}),(a_{21}),(a_{23}),(a_{31})(a_{11},a_{22}),(a_{32})\}$	6
	4	$\{(a_{13}),(a_{21}),(a_{11}),(a_{31})(a_{23},a_{32}),(a_{22})\}$	6
	5	$\{(a_{13}),(a_{21}),(a_{11},a_{22}),(a_{31}),(a_{23},a_{32})\}$	5
	6	$\{(a_{13}),(a_{21}),(a_{11},a_{31}),(a_{22}),(a_{23},a_{32})\}$	5
T3	1	$\{(a_{11}),(a_{12}),(a_{21}),(a_{31}),(a_{32})\}$	5
T4	1	$\{(a_{11}),(a_{21})\}$	2
T5	1	$\{(a_{11}),(a_{12}),(a_{13}),(a_{21})\}$	4
T6	1	$\{(a_{21}),(a_{22}),(a_{41})\}$	3
	2	$\{(a_{21}),(a_{22},a_{41})\}$	2
	3	$\{(a_{22}),(a_{21},a_{41})\}$	2
T7	1	$\{(a_{51}),(a_{61}),(a_{62})\}$	3
T8	1	$\{(a_{11}),(a_{12}),(a_{13}),(a_{21}),(a_{22}),(a_{31}),(a_{41}),(a_{61}),(a_{113})\}$	9
	2	$\{(a_{11},a_{31}),(a_{12}),(a_{13}),(a_{21}),(a_{22}),(a_{41}),(a_{61}),(a_{113})\}$	8
	3	$\{(a_{31},a_{61}),(a_{11}),(a_{12}),(a_{13}),(a_{21}),(a_{22}),(a_{41}),(a_{113})\}$	8
	4	$\{(a_{11},a_{61}),(a_{12}),(a_{13}),(a_{21}),(a_{22}),(a_{31}),(a_{41}),(a_{113})\}$	8
	5	$\{(a_{11},a_{31},a_{61}),(a_{12}),(a_{13}),(a_{21}),(a_{22}),(a_{41}),(a_{113})\}$	7
T9	1	$\{(a_{11}),(a_{22}),(a_{52}),(a_{63}),(a_{101}),(a_{142}),(a_{12}),(a_{23}),(a_{42}),(a_{91}),(a_{31}),(a_{93}),(a_{113}),(a_{33}),(a_{82}),(a_{41}),(a_{71}),(a_{122}),(a_{51}),(a_{111}),(a_{121}),(a_{133})\}$	22
	2～25	$\{(a_{11},a_{22}),(a_{52}),(a_{63}),(a_{101}),(a_{142}),(a_{12}),(a_{23}),(a_{42}),(a_{91}),(a_{31}),(a_{93}),(a_{113}),(a_{33}),(a_{82}),(a_{41}),(a_{71}),(a_{122}),(a_{51}),(a_{111}),(a_{121}),(a_{133})\}$ ……	21
	26～51	$\{(a_{11},a_{22},a_{52}),(a_{63}),(a_{101}),(a_{142}),(a_{12}),(a_{23}),(a_{42}),(a_{91}),(a_{31}),(a_{93}),(a_{113}),(a_{33}),(a_{82}),(a_{41}),(a_{71}),(a_{122}),(a_{51}),(a_{111}),(a_{121}),(a_{133})\}$ ……	20
	…	……	…
	m	$\{(a_{11},a_{22},a_{52},a_{63},a_{101},a_{142}),(a_{12},a_{23},a_{42},a_{91}),(a_{31},a_{93},a_{113}),(a_{33},a_{82}),(a_{41},a_{71},a_{122})(a_{51}),(a_{111},a_{121})),(a_{133})\}$	8

上述实验结果表明,与随机结合法相比,本文方法得到了较少的可执行路径.

仍以程序 T1 为例,说明采用排列组合法,生成可执行路径的过程. T1 的子路径集合 $A=\{\{a_{11},a_{12},a_{13}\},\{a_{21},a_{22}\},\{a_{31},a_{32}\}\}$,通过静态分析,得知该集合包含子路径最多的集合为 $A_1=\{a_{11},a_{12},a_{13}\}$,共包含 3 条子路径. 因为,这 3 条子路径之间不能相互结合生成可执行路径,所以,可能生成的可执行路径条数≥3.

下面基于 A 包含的子路径,采用排列组合法,生成可执行路径. 首先,从集合 $A_1=\{a_{11},a_{12},a_{13}\}$ 中取出 3 个元素,且每次取出一个不重复的元素,有 $C_3^1C_2^1C_1^1=3!$ 种取法;然后,从集合 $A_2=\{a_{21},a_{22}\}$ 中取出 0 或 1 个元素,且每次取出不重复的元素,有 $C_3^1C_2^1C_1^1=3!$ 种取法;从集合 $A_3=\{a_{31},a_{32}\}$ 中取出 0 或 1 个元素,且每次取出不重复的元素,有 $C_3^1C_2^1C_1^1=3!$ 种取法;最后,删除重复的路径组合,得到不重复的路径集合,共有 $\underline{C_3^1C_2^1C_1^1}\underline{C_3^1C_2^1C_1^1}\underline{C_3^1C_2^1C_1^1}/3!=3!3!3!/3!=36$ 个,如表 8 所列. 在这些路径集合中包含了一些不可执行路径,最终,仅有第 35 和 36 个路径集合包含可执行路径.

表 8 被测程序 T1 采用组合方法形成的 3 条路径

路径集编号	组合路径集	路径集编号	组合路径集
1	$\{(a_{11},a_{21},a_{32}),(a_{12}),(a_{13},a_{22},a_{31})\}$	19	$\{(a_{11},a_{22},a_{31}),(a_{12},a_{21},a_{32}),(a_{13})\}$
2	$\{(a_{11},a_{22},a_{32}),(a_{12}),(a_{13},a_{21},a_{31})\}$	20	$\{(a_{11},a_{21},a_{32}),(a_{12},a_{22},a_{31}),(a_{13})\}$
3	$\{(a_{11}),(a_{12},a_{22},a_{32}),(a_{13},a_{21},a_{31})\}$	21	$\{(a_{11},a_{22},a_{32}),(a_{12},a_{21},a_{31}),(a_{13})\}$
4	$\{(a_{11}),(a_{12},a_{21},a_{32}),(a_{13},a_{22},a_{31})\}$	22	$\{(a_{11},a_{21},a_{31}),(a_{12}),(a_{13},a_{22},a_{32})\}$
5	$\{(a_{11}),(a_{12},a_{22},a_{31}),(a_{13},a_{21},a_{32})\}$	23	$\{(a_{11},a_{22},a_{31}),(a_{12}),(a_{13},a_{21},a_{32})\}$
6	$\{(a_{11}),(a_{12},a_{21},a_{31}),(a_{13},a_{22},a_{32})\}$	24	$\{(a_{11},a_{21}),(a_{13},a_{22},a_{31}),(a_{12},a_{32})\}$
7	$\{(a_{11},a_{21}),(a_{12},a_{22},a_{31}),(a_{13},a_{32})\}$	25	$\{(a_{11},a_{22}),(a_{13},a_{21},a_{31}),(a_{12},a_{32})\}$
8	$\{(a_{11},a_{22}),(a_{12},a_{21},a_{31}),(a_{13},a_{32})\}$	26	$\{(a_{11},a_{21}),(a_{13},a_{22},a_{32}),(a_{12},a_{31})\}$
9	$\{(a_{11},a_{21}),(a_{12},a_{22},a_{32}),(a_{13},a_{31})\}$	27	$\{(a_{11},a_{22}),(a_{13},a_{21},a_{32}),(a_{12},a_{31})\}$
10	$\{(a_{11},a_{22}),(a_{12},a_{21},a_{32}),(a_{13},a_{31})\}$	28	$\{(a_{12},a_{21}),(a_{13},a_{22},a_{31}),(a_{11},a_{32})\}$
11	$\{(a_{12},a_{21}),(a_{11},a_{22},a_{31}),(a_{13},a_{32})\}$	29	$\{(a_{12},a_{22}),(a_{13},a_{21},a_{31}),(a_{11},a_{32})\}$
12	$\{(a_{12},a_{22}),(a_{11},a_{21},a_{31}),(a_{13},a_{32})\}$	30	$\{(a_{12},a_{21}),(a_{13},a_{22},a_{32}),(a_{11},a_{31})\}$
13	$\{(a_{12},a_{21}),(a_{11},a_{22},a_{32}),(a_{13},a_{31})\}$	31	$\{(a_{12},a_{22}),(a_{13},a_{21},a_{32}),(a_{11},a_{31})\}$
14	$\{(a_{12},a_{22}),(a_{11},a_{21},a_{32}),(a_{13},a_{31})\}$	32	$\{(a_{13},a_{22}),(a_{11},a_{21},a_{31}),(a_{12},a_{32})\}$
15	$\{(a_{13},a_{21}),(a_{12},a_{22},a_{31}),(a_{11},a_{32})\}$	33	$\{(a_{13},a_{22}),(a_{11},a_{21},a_{32}),(a_{12},a_{31})\}$
16	$\{(a_{13},a_{22}),(a_{12},a_{21},a_{31}),(a_{11},a_{32})\}$	34	$\{(a_{13},a_{21}),(a_{11},a_{22},a_{32}),(a_{12},a_{31})\}$
17	$\{(a_{13},a_{22}),(a_{12},a_{21},a_{32}),(a_{11},a_{31})\}$	35	$\{(a_{13},a_{21}),(a_{12},a_{22},a_{32}),(a_{11},a_{31})\}$
18	$\{(a_{11},a_{21},a_{31}),(a_{12},a_{22},a_{32}),(a_{13})\}$	36	$\{(a_{13},a_{21}),(a_{11},a_{22},a_{31}),(a_{12},a_{32})\}$

对于被测程序 T2～T9，采用上述方法，也得到相应的可执行路径的集合，如表 9 所列. 表中第 2 列为各被测程序的子路径集合；第 3 列为生成的可执行路径集合；第 4 列为该集合包含的可执行路径条数.

表 9　其他被测程序采用组合方法生成的路径集

ID	子路径集合 A	生成的可执行路径集	路径条数
T2	$\{\{a_{11},a_{12},a_{13}\},\{a_{21},a_{22},a_{23}\},\{a_{31},a_{32}\}\}$	$\{(a_{11},a_{31}),(a_{12},a_{21}),(a_{22}),(a_{13}),(a_{23},a_{32})\}$	5
T3	$\{\{a_{11},a_{12}\},\{a_{21}\},\{a_{31},a_{32}\}\}$	$\{(a_{11}),(a_{12}),(a_{21}),(a_{31}),(a_{32})\}$	5
T4	$\{\{a_{11}\},\{a_{21},a_{22}\}\}$	$\{(a_{11},a_{21}),(a_{22})\}$	2
T5	$\{\{a_{11},a_{12},a_{13}\},\{a_{21},a_{22}\},\{a_{31}\}\}$	$\{(a_{11}),(a_{12}),(a_{13},a_{22}),(a_{21},a_{31})\}$	4
T6	$\{\{a_{11}\},\{a_{21},a_{22}\},\{a_{31}\},\{a_{41}\},\{a_{51}\},\{a_{61}\},\{a_{71}\}\}$	$\{(a_{11},a_{22},a_{31},a_{51},a_{61},a_{71}),(a_{21},a_{41})\}$	2
T7	$\{\{a_{11}\},\{a_{21}\},\{a_{31}\},\{a_{41}\},\{a_{51}\},\{a_{61},a_{62}\}\}$	$\{(a_{11},a_{61}),(a_{51}),(a_{21},a_{31},a_{41},a_{62})\}$	3
T8	$\{\{a_{11},a_{12},a_{13}\},\{a_{21},a_{22}\},\{a_{31}\},\{a_{41}\},\{a_{51}\},\{a_{61}\},\{a_{71}\},\{a_{81}\},\{a_{91}\},\{a_{101}\},\{a_{111},a_{112},a_{113}\},\{a_{121}\},\{a_{131}\},\{a_{141}\},\{a_{151}\},\{a_{161}\},\{a_{171},a_{172}\}\}$	$\{(a_{11},a_{31},a_{51},a_{61},a_{71},a_{81},a_{91},a_{101},a_{111},a_{121},a_{131},a_{141},a_{151},a_{161},a_{172}),(a_{12},a_{112},a_{171}),(a_{41}),(a_{13}),(a_{21}),(a_{22}),(a_{113})\}$	7
T9	$\{\{a_{11},a_{12}\},\{a_{21},a_{22},a_{23}\},\{a_{31},a_{32},a_{33}\},\{a_{41},a_{42}\},\{a_{51},a_{52},a_{53}\},\{a_{61},a_{62},a_{63}\},\{a_{71}\},\{a_{81},a_{82}\},\{a_{91},a_{92},a_{93}\},\{a_{101}\},\{a_{111},a_{112},a_{113}\},\{a_{121},a_{122},a_{123}\},\{a_{131},a_{132},a_{133}\},\{a_{141},a_{142}\}\}$	$\{\{a_{12},a_{23},a_{32},a_{42},a_{62},a_{81},a_{91},a_{112},a_{123},a_{132}\},\{a_{133}\},\{a_{51}\},\{a_{31},a_{93},a_{113}\},\{a_{21},a_{53},a_{61},a_{92},a_{111},a_{121},a_{131},a_{141}\},\{a_{33},a_{82}\},\{a_{41},a_{71},a_{122}\},\{a_{11},a_{22},a_{52},a_{63},a_{101},a_{142}\}\}$	8

由表 4 和表 8、表 9 可知，① 采用组合法生成的路径数，与本文方法生成的可执行路数相同；② 采用组合法生成的路径，通过占优关系约简之后，与本文方法生成的可执行路径相同.

上述实验结果表明，本文方法与采用排列组合方法生成的可执行路径数相同.

本组实验结果表明，采用本文方法生成的可执行路径比随机结合法少；与排列组合法生成的可执行路径相同.

下面从实验消耗成本方面进行分析. 采用随机法生成路径时，一次运行程序，生成一个路径集，如表 7 所列，生成的可执行路径个数不确定. 采用组合法生成的路径集，如表 8 所列，T1 程序仅生成了 3 条路径，有效路径与路径数量的比为 5.5%(2 种可执行路径集合×3/(36 种×3)). 如果生成的路径多于 3 条，那么，有效路径与路径数量的比更小. 对于其他 8 个被测程序，也能得到类似的实验结果.

随着程序规模的增大和变异分支的增多，随机法和组合法形成的路径组合条数将呈爆炸式增长. 而所提方法的有效路径与路径数量的比为 100%. 更重要的是，随机法和排列组合法生成的路径集包含的不可执行路径，检测起来相当困难. 到目前为止，还没有一种有效的方法，自动检测某程序的所有不可执行路径[38]，导致的后果是，需要花费很多的时间，人工检测不可执行路径. 而采用本文方法自动生成的全是可执行路径，且运行时间少.

表 10 给出了 $R=5000$ 时，采用本文方法，由子路径自动生成可执行路径的运行时间. 由该表可以看出，除了 T6 和 T9 运行时间稍多之外，其他程序的运行时间均较少. 因此，与其他 2 种方法比较，本文方法是高效的.

表 10　本文方法生成可执行路径运行时间

ID	时间/ms	ID	时间/ms
T1，T3，T4，T5	0	T7	20
T2	10	T8	109
T6	625	T9	887

(4) 样本容量对本文方法性能的影响

下面，分析样本容量对本文方法生成可执行路径的条数和运行时间等性能指标的影响. 首先，考虑样本容量对本文方法生成可执行路径条数的影响. 如表 11 第 2 和第 8 列为被测程序包含的子路径条数；第 3 和第 9 列为实验中选取的不同样本容量；第 4 和第 10 列为不同样本容量下，两个不同子路径之间相关度 $\alpha\neq 0$ 的个数；第 5 和第 11 列为本文方法生成的相应的可执行路径条数.

由表 11 可以看出，① 随着样本容量的增加，两个不同子路径之间相关度 $\alpha\neq 0$ 的个数增多；② $\alpha\neq 0$ 的个数越多，由子路径生成的可执行路径条数越少；③ 当样本容量很大时，$\alpha\neq 0$ 的个数将保持不变，此时，生成可执行路径的条数将很少.

然后，考察样本容量对程序运行时间的影响. 不同样本容量下，本文方法生成可执行路径运行的时间如表 11 的第 6 和第 12 列所示. 由该表可以看出，除了程序 T6，T9 之外，对于其他被测程序，样本容量对运行时间的影响几乎可以忽略不计.

每个程序样本容量的充分值，需要通过多次运行该程序才能得到. 一般情况下，$\alpha\neq 0$ 个数不再变化时

表 11 样本容量的影响

ID	可执行子路径条数	R	$\alpha\neq0$ 的个数	生成可执行路径的条数	运行时间/ms	ID	可执行子路径条数	R	$\alpha\neq0$ 的个数	生成可执行路径的条数	运行时间/ms
		100	10	5	0			100*	4	4	0
		300	18	4	0			300	4	4	0
T1	7	500	18	4	0	T5	6	500	4	4	0
		1000	18	4	0			1000	4	4	0
		3000*	22	3	0			3000	4	4	0
		100	8	6	0			100	28	3	20
		300	8	6	0			300	28	3	46
T2	8	500	14	6	0	T6	8	500*	34	2	110
		1000*	16	5	10			1000	34	2	150
		3000	16	5	10			3000	34	2	380
		100*	0	5	0			100*	47	3	10
		300	0	5	0			300	47	3	10
T3	5	500	0	5	0	T7	7	500	47	3	15
		1000	0	5	0			1000	47	3	20
		3000	0	5	0			3000	47	3	20
		100*	4	2	0			100	28	18	10
		300	4	2	0			300	95	13	21
T4	3	500	4	2	0	T8	23	500	184	8	40
		1000	4	2	0			1000	210	8	62
		3000	4	2	0			3000*	242	7	78
		100	34	52	35			1000	167	10	278
T9	34	300	80	30	67	T9	—	3000*	208	8	426
		500	97	34	102			—	—	—	—

的样本容量值，即为该程序的样本容量充分值. 如表 11 第 3 和第 9 列所示，R 标注为“*”的值，为该程序的样本容量充分值.

5 总 结

本文研究弱变异测试问题，并将弱变异测试转化为路径覆盖问题，期望利用已有的路径覆盖方法，生成高质量的变异测试数据，以提高变异测试数据生成的效率.

为此，首先，对同一语句变异形成的多个变异分支，基于执行关系，形成新的变异分支；然后，利用该语句与新变异分支的相关性，生成包含新变异分支的可执行子路径；最后，采用统计分析方法，生成一条或多条覆盖所有子路径的可执行路径. 这样做的好处是，覆盖上述可执行路径的测试数据，一定能够杀死相应的变异体，从而为生成高质量的变异测试数据，提供了一条可行的途径.

为了评价所提方法的性能，将所提方法应用于 9 个基准和工业程序测试，并与随机法和排列组合法比较. 实验结果表明：(1) 采用本文方法生成的路径均为可执行的；(2) 这些可执行路径能够覆盖所有的变异分支；(3) 采用本文方法生成的可执行路径条数少，运行时间短；此外，本文方法涉及的样本容量，对生成的可执行路径数目有一定的影响，而对程序的运行时间基本没有影响.

值得说明的是，本文在生成可执行路径时，仅考虑了生成可执行路径的条数，而没有考虑这些可执行路径覆盖的难易程度. 一种可能的情况是，虽然可执行路径少，但是，如果这些路径难以覆盖，那么，也难于生成覆盖这些路径的测试数据，从而降低了变异测试数据生成效率，鉴于此，在后续的研究中，有必要进一步结合路径覆盖的难易程度，生成条数少且容易覆盖的可执行路径，以提高变异测试数据生成的效率. 此外，对于并行程序的弱变异测试，如何生成相应的可执行路径，也是需要进一步研究的问题.

致 谢 各位审稿专家对本文提出了宝贵的评审意见，这些评审意见对提高论文水平具有很大的帮助. 编辑付出了辛勤工作. 在此一并致谢!

参 考 文 献

[1] Souza S R S, Brito M A S, Silva R A, et al. Research in concurrent software testing: A systematic review//Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. New York, USA, 2011: 1-5

[2] DeMillo R A, Lipton R J, Sayward F G. Hints on test data

selection: Help for the practicing programmer. Computer, 1978, 11(4): 34-41

[3] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 2011, 37(5): 649-678

[4] Howden W E. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering, 1982, SE-8(4): 371-379

[5] Horgan J R, Mathur A P. Weak mutation is probably strong mutation. Purdue University, Lafayette, USA: Technical Report SERC-TR-92-P, 1990

[6] Offutt A J, Lee S D. An empirical evaluation of weak mutation. IEEE Transactions on Software Engineering, 1994, 20(5): 337-344

[7] 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

[8] Chen Xiang, Gu Qing. Mutation testing: Principal, optimization and application. Journal of Frontiers of Computer Science & Technology, 2012, 6(12): 1057-1075(in Chinese)
(陈翔，顾庆. 变异测试：原理，优化和应用. 计算机科学与探索，2012，6(12)：1057-1075)

[9] Fraser G, Arcuri A. Achieving scalable mutation-based generation of whole test suites. Empirical Software Engineering, 2014, 20(3): 783-812

[10] Debroy V, Wong W E. Combining mutation and fault localization for automated program debugging. The Journal of Systems and Software, 2014, 90(1): 45-60

[11] Jia Y, Harman M. Higher order mutation testing. Information and Software Technology, 2009, 51(10): 1379-1393

[12] Chen Jin-Fu, Lu Yan-Sheng, Xie Xiao-Dong. Research on software fault injection testing. Journal of Software, 2009, 20(6): 1425-1443(in Chinese)
(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)

[13] Liu Xin-Zhong, Xu Gao-Chao, Hu Liang, et al. An approach for constraint-based test data generation in mutation testing. Journal of Computer Research and Development, 2011, 48(4): 617-626(in Chinese)
(刘新忠，徐高潮，胡亮. 一种基于约束的变异测试数据生成方法. 计算机研究与发展，2011，48(4)：617-626)

[14] Offutt A J, Lee S D. How strong is weak mutation// Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification. New York, USA, 1991: 200-213

[15] Harman M. Software engineering meets evolutionary computation. Computer, 2011, 44(10): 31-39

[16] Just R, Schweiggert F. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. Software Testing, Verification and Reliability, 2015, 25(5-7): 490-507

[17] Offutt A J, Lee A, Rothermel G, et al. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology, 1996, 5(2): 99-118

[18] Xu Shi-Yi. A method of simplifying complexity of mutation testing. Chinese Journal of Shanghai University (Natural Science Edition), 2007, 13(5): 524-531(in Chinese)
(徐拾义. 降低程序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)

[19] Langdon W B, Harman M, Jia Y. Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software, 2010, 83(12): 2416-2430

[20] Offutt A J. Automatic Test Data Generation [Ph. D. dissertation]. Georgia Institute of Technology, Atlanta, USA, 1988

[21] Offutt A J, Jin Z, Pan J. The dynamic domain reduction procedure for test data generation. Software: Practice and Experience, 1999, 29(2): 167-193

[22] Shan Jin-Hui, Gao You-Feng, Liu Ming-Hao, et al. A new approach to automated test data generation in mutation testing. Chinese Journal of Computers, 2008, 31(6): 1025-1034(in Chinese)
(单锦辉，高友峰，刘明浩等. 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)

[23] McMinn P. Search-based software test data generation: A survey. Software Testing, Verification and Reliability, 2004, 14(2): 105-156

[24] Lin J C, Yeh P L. Automatic test data generation for path testing using GAs. Information Sciences, 2001, 131(1): 47-64

[25] Zhang Gong-Jie, Gong Dun-Wei, Yao Xiang-Juan. Test case generation based on mutation analysis and set evolution. Chinese Journal of Computers, 2015, 38(11): 2318-2331(in Chinese)
(张功杰，巩敦卫，姚香娟. 基于变异分析和集合进化的测试用例生成方法. 计算机学报，2015，38(11)：2318-2331)

[26] Harman M, Mansouri S A, Zhang Y. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, 2012, 45(1): 11

[27] Hermadi I, Lokan C, Sarker R. Dynamic stopping criteria for search-based test data generation for path testing. Information and Software Technology, 2014, 56(4): 395-407

[28] Bueno P M S, Jino M. Automatic test data generation for program paths using genetic algorithms. International Journal of Software Engineering and Knowledge Engineering, 2002, 12(6): 691-709

[29] Lin J C, Yeh P L. Automatic test data generation for path testing using GAs. Information Sciences, 2001, 131(1): 47-64

[30] Watkins A, Hufnagel E M. Evolutionary test data generation: A comparison of fitness functions. Software: Practice and Experience, 2006, 36(1): 95-116

[31] Ahmed M A, Hermadi I. GA-based multiple paths test data generator. Computers & Operations Research, 2008, 35(10): 3107-3124

[32] Gong D, Zhang W, Yao X. Evolutionary generation of test data for many paths coverage based on grouping. Journal of Systems and Software, 2011, 84(12): 2222-2233

[33] Gong D, Tian T, Yao X. Grouping target paths for evolutionary generation of test data in parallel. Journal of Systems and Software, 2012, 85(11): 2531-2540

[34] Zhang G, Chen R, Li X, et al. The automatic generation of basis set of path for path testing//Proceedings of the Test Symposium, Calcutta, India, 2005: 46-51

[35] Yan J, Zhang J. An efficient method to generate feasible paths for basis path testing. Information Processing Letters, 2008, 107(3): 87-92

[36] Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. Information and Software Technology, 2012, 54(9): 915-932

[37] Yao X J, Gong D W, Luo Y J, et al. Test data reduction based on dominance relations of target statements//Proceedings of the Evolutionary Computation (CEC). Brisbane, Australia, 2012: 1-8

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

[39] Offutt A J, Pan J. Detecting equivalent mutants and the feasible path problem//Proceedings of the 11th Annual Conference on Computer Assurance, Systems Integrity, Software Safety, Process Security. Gaithersburg, USA, 1996: 224-236

[40] Offutt A J, Pan J. Automatically detecting equivalent mutants and infeasible paths. Software Testing Verification & Reliability, 1997, 7(3): 165-192

[41] Mouchawrab S, Briand L C, Labiche Y, et al. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. IEEE Transactions on Software Engineering, 2011, 37(2): 161-187

[42] McMinn P, Harman M, Lakhotia K, et al. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. IEEE Transactions on Software Engineering, 2012, 38(2): 453-477

[43] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005, 10(4): 405-435

[44] Zhong H, Zhang L, Mei H. An experimental study of four typical test suite reduction techniques. Information and Software Technology, 2008, 50(6): 534-546

[45] Zhang Yan, Gong Du-Wei. Evolutionary generation of test data for paths coverage based on scarce data capturing. Chinese Journal of Computers, 2013, 36(12): 2429-2440(in Chinese)
(张岩, 巩敦卫. 基于稀有数据扑捉的路径覆盖测试数据进化生成方法. 计算机学报, 2013, 36(12): 2429-2440)

[46] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005, 10(4): 405-435

DANG Xiang-Ying, born in 1978, Ph. D. candidate, associate professor. Her main research interest is software testing.

GONG Dun-Wei, born in 1970, Ph. D., professor, Ph. D. supervisor. His main research interests include search-based software engineering, intelligent optimization and control.

YAO Xiang-Juan, born in 1975, Ph. D., professor. Her research interest is search-based software engineering.

Background

All software testing methods attempt to discover errors in a program by seeking effective test data.

Mutation testing is a fault-based technique for unit level testing. Mutation testing describes a fault as a simple syntactic change to a program, called mutation. A series of mutation operators are employed to create a number of mutants of a program, and test data are generated to kill these mutants. Weak mutation is a way to reduce the complex of mutation testing.

Generally, there exist a great number of mutants based on a program under test, and a large number of test data are required so as to kill these mutants. In addition, these test data must execute the original program and its mutants, reducing the efficiency of mutation testing. To overcome the above drawback, Papadakis et al. ever proposed a method of forming another program by fusing all the true branches of

mutant statements into the program. These branches can be constructed by statements before and after mutation. By this means, we can transform the problem of weak mutation testing to that of covering branches. The above method raises, however, a new problem that the converted program contains a great number of branches, which enhances the complex of generating mutation test data. In view of this, it is necessary to study new approaches to improve the efficiency of mutation testing.

Different from previous work, our approach transforms the problem of covering the true branches of mutant statements to that of covering a small number of feasible paths, with the purpose of solving the problem of weak mutation testing by using previous methods of path coverage. Effective methods for generating feasible target paths based on a program and its mutants are, however, of absence up to date. This paper investigates the problem of generating feasible paths that contain all the true branches of mutant statements, and proposes a method to improve the quality of test data based on weak mutation testing. The experimental results show that a small number of feasible paths generated by the proposed method can cover all the true branches of mutant statements. By this means, we can improve the efficiency of generating test data for mutation testing.

This research is jointly sponsored by the National Basic Research Program (973 Program) of China (2014CB046306-2), and the National Natural Science Foundation of China (61375067, 61203304, and 61573362).