计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200200116

面向 Java 的 Randoop 自动化单元测试生成工具性能分析

刘 芳[1] 洪 玫[2] 王 潇[1] 郭 丹[1] 杨正卉[1] 黄小丹[1]
1 四川大学计算机学院 成都 610065
2 四川大学软件学院 成都 610065
(756493201@qq.com)

摘 要 自动化单元测试是现代软件开发研究的热点。Randoop 自动化单元测试用例生成工具针对 Java 和.NET 代码,基于反馈指导随机生成测试用例,在业界应用广泛。为了有效使用 Randoop 进行自动化测试,采用经验软件工程的方法,通过实验分析 Randoop 的性能特点;实验选取 4 个有代表性的 Java 开源项目,分析 Randoop 生成测试用例的代码覆盖率和变异体检测能力,以及它们与时间成本和被测类源代码的代码结构之间的关系。实验发现,Randoop 可以在短时间内生成有效的测试用例,生成测试用例的性能随时间增加而上升,并在测试用例生成时间为 120s 时趋于稳定,其平均变异体覆盖率达 55.59%,且平均变异体杀死率为 28.15%。Randoop 生成的测试用例的性能与被测类源代码的代码结构和复杂度存在一定的关系。该研究为软件测试人员有效使用 Randoop 工具提供了有价值的参考。

关键词:Randoop;面向对象自动化单元测试;测试用例自动生成;代码覆盖率;变异分析
中图法分类号 TP311.5

Performance Analysis of Randoop Automated Unit Test Generation Tool for Java

LIU Fang[1],HONG Mei[2],WANG Xiao[1],GUO Dan[1],YANG Zheng-hui[1] and HUANG Xiao-dan[1]
1 School of Computer Science,Sichuan University,Chengdu 610065,China
2 School of Software Engineering,Sichuan University,Chengdu 610065,China

Abstract Automated unit testing is a hotspot in modern software development research. Randoop,an automated unit test cases generation tool,is designed for Java and .NET code,and generates test cases based on feedback guidance. It is widely used in the industry. In order to effectively use Randoop for automated testing,this paper uses empirical software engineering methods to analyze the performance characteristics of Randoop through experiments. Four representative Java open source projects are selected to analyze the code coverage of Randoop-generated test cases and the ability to detect mutants,and the relationship of the effectiveness of randoop with the time cost and the source code structure. The experiment find that Randoop can generate valid test cases in a short time. With the increase of generation time,the performance of Randoop generation test cases increases,and tends to be stable when the tests generation time is 120s,with an average mutants coverage of 55.59% and an average mutants kill rate of 28.15%. The performance of the test cases generated by Randoop is related to the structure and complexity of the source code of the tested classes. This paper provides a valuable reference for software testers to effectively use the Randoop tool.

Keywords Randoop,Object-oriented automation unit testing,Automatic generation of test cases,Code coverage,Mutation analysis

1 引言

Randoop[1]是 MIT 的 Carlos Pacheco 博士针对 Java 和.NET单元测试开发的自动生成随机测试用例的开源工具,它以随机但包含反馈指导的方式生成被测试类的方法调用序列,再使用这种序列生成测试用例。Randoop 是随机测试用例生成工具中最稳定和最受欢迎的代表。目前,国内外针对自动化单元测试的研究较多,也倾向于分析并比较流行的单元测试生成工具的性能[2-5]。衡量工具生成的测试用例有效性的标准有检错率、覆盖率和变异分数等。

Shamshiri[6]团队研究了自动生成的测试用例是否能检测到真实缺陷。研究表明,EvoSuite,Randoop 以及 Agitar 这 3 个工具生成的测试用例的检测率均未高于 40.6%。最终,Shamshiri 团队分析了这 3 个具有代表性的工具的局限性,并提出了改进建议。Kochhar 等[7]研究了测试用例的覆盖率和检错率之间的关系,利用 Randoop 工具为两个大型软件中的真实缺陷生成覆盖率不同的测试套件。研究表明:代码覆盖率和检错率间存在显著的统计学相关关系。Jalali 等[8]选取 Defects4j 数据集上的 5 个 Java 开源项目和 Randoop,EvoSuite,JCrasher 3 个测试用例生成工具,得出实验结论:检测出

到稿日期:2020-02-26 返修日期:2020-06-14 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
通信作者:洪玫(hongmei@scu.edu.cn)

更多变异体的测试套件具有更高的缺陷检错率，即测试套件的变异分数可以与缺陷检错率等效，用于衡量测试用例的有效性，以帮助测试人员补充测试用例集。

现有研究[6-8]以工具生成的测试用例的缺陷检测率、覆盖率或变异分数三者之一为衡量标准，分析 Randoop 及其他工具生成的测试用例的有效性。本文与已有研究的不同处在于：1）以 Randoop 工具为主，结合代码覆盖率和变异分数，从多角度评估测试用例的性能；2）调整时间参数，以分析时间对 Randoop 工具性能的影响，并给出使用建议；3）提取被测代码的结构特征，采用 Pearson 相关系数分析测试用例有效性与被测代码各项结构特征的相关关系，以讨论工具的适用场景。

本文工作结合以上研究的实验方法和结论，对随机单元测试用例生成工具 Randoop 进行性能分析，在 Defects4j 数据集上进行实验，与 Shamshiri 等以回归测试为研究背景不同，本文以单元测试为研究背景；在 Jalali 等研究的基础上，分析测试用例的有效性，并用 Kochhar 等的研究成果分析测试用例的有效性和被测代码的结构之间的关系。本文拟回答以下 3 个问题。

（1）Randoop 生成的测试用例有效吗？

本文以测试用例的行覆盖率、分支覆盖率、变异覆盖以及变异分数为衡量标准，从多角度评估测试用例的质量，评价 Randoop 生成的测试用例的有效性。

（2）测试用例有效性与测试用例生成成本有关吗？

在每年基于搜索的软件测试国际研讨会（Search-Based Software Testing，SBST）举办的 Java 单元测试自动生成工具的竞赛中，时间成本是工具竞赛得分的一个关键项。竞赛中虽有考虑时间成本而设置不同的时间来生成测试用例，但最终只是将其用于得分计算公式中。本文通过分析不同时间成本下生成的测试用例的覆盖率和变异分数，研究测试用例的有效性与时间成本之间的关系，以指导测试人员在使用 Randoop 工具时进行成本/效益的权衡。

（3）测试用例的有效性与被测源代码的结构有关吗？

在 Shamshiri 和 Kochhar 等团队的研究中，Randoop 生成的测试用例在不同项目上有不同的表现，但均通过计算所有项目的平均覆盖率和平均变异分数来分析测试用例的有效性。本文通过提取被测类源代码的行数、分支数、方法数、圈复杂度和函数深度等代码结构特征，分析测试用例的有效性与被测源代码的结构之间的关系，并研究 Randoop 工具的适用场景。

2　实验设计

为回答本文提出的研究问题，文中设计了 4 个实验。基本思路是：首先选择一组实验对象，设置实验条件，使用工具获得实验数据，包括测试用例生成时间、语句覆盖率、分支覆盖率、变异体覆盖率和变异分数等；然后对实验数据进行分析，并基于分析结果回答研究问题。实验对象选取 Defects4j 上 4 个开源项目的 224 个 fixed 版本，为其生成 6 组不同时间参数的测试套件，时间分别设置为 30 s，60 s，90 s，120 s，180 s 和 240 s，测量每组测试套件的语句覆盖率、分支覆盖率、变异体覆盖率和变异分数，回答研究问题（1）和（2）；通过工具获取被测类源代码的代码结构特征，分析被测类源代码的结构与测试用例的有效性之间的关系。

2.1　实验数据集

Defects4j[9]数据集是 Ernst 教授等于 2014 年发布的真实缺陷数据库和可扩展框架，它的初始版本提供了来自 5 个 Java 开源项目的 357 个真实缺陷，每个缺陷对应一个 fixed 版本和 buggy 版本，buggy 版本用于对 Java 程序进行受控测试研究。Defects4j 提供框架访问程序的缺陷版本、修复版本和人工编写的测试用例集；还提供测试执行框架，包含生成测试、执行测试、代码覆盖以及变异分析等。Defects4j 数据集提供了 5 个开源项目，其源代码结构特征如表 1 所列。由于 Randoop 不针对私有类和私有方法生成测试用例，而 Closure 项目的被测类中私有方法占比高达 70%，因此本文实验仅选用其余 4 个项目作为被测类源代码。

Shamshiri 团队在主题为“自动化单元测试生成工具是否能发现真正的缺陷”的研究中，选取 Defects4j 数据集的 357 个真实缺陷，证实了 Randoop 工具的缺陷检测率为 25%。由于不能确定这 357 个缺陷是否为自动化单元测试可以发现的缺陷，因此不能得出 Randoop 工具检错率低的结论。本文引入变异测试，通过变异分数进一步研究该工具的检错能力。

表 1　Defect4j 数据集上的开源项目

Table 1　Open source projects on Defect4j

项目名	文件数	行数	语句数	分支占比/%	类个数	方法数/类	语句数/方法	最大圈复杂度	最大函数深度	平均函数深度	平均圈复杂度	被测类源代码数量
Lang	218	87537	35557	10.7	368	12.36	6.27	98	9+	2.06	2.01	65
Math	976	218988	78666	12.2	1335	6.98	6.34	144	9+	2.2	2.17	106
Chart	1060	327644	110376	11.8	1133	10.95	6.55	57	9+	1.96	2.17	26
Time	317	141616	66366	7.6	526	17.99	5.51	54	9+	1.99	1.58	27
Closure	990	377083	164996	20.5	2054	11.36	5.49	430	9+	2.58	2.51	133

2.2　实验工具

2.2.1　自动化单元测试生成工具 Randoop

Randoop 的主要思想是“随机生成”“系统化修剪”，Pacheco 博士称其为有导向的随机测试。如图 1 所示，Randoop 以先前生成的测试用例的执行情况作为反馈信息，来指导后续测试用例的生成，尽可能地避免或减少后续用例出现冗余或不合法的情况。Randoop 增量式地生成随机方法调用序列，即后面的每个用例包含的随机方法调用序列比前面的多[10]。使用 Randoop 作为随机测试用例生成工具，工具的输入为被测类源代码、生成时间和契约文件（被测代码需要遵从

的规范)[11],输出为可在 Junit 上运行的测试用例。这些测试用例可用于两类测试:一种是用于测试违反契约的揭错测试用例;另一种是用于回归测试的测试用例,可以捕获不希望的行为改变。

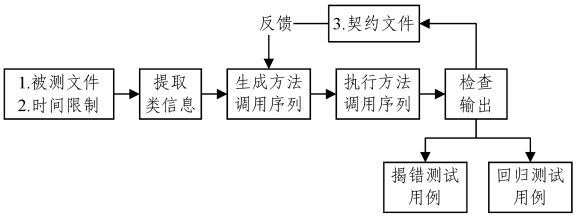

图 1 Randoop 设计框架

Fig. 1 Design framework of Randoop

2.2.2 测试用例覆盖率分析和变异分析工具

本文实验基于 Defects4j 数据集提供的测试执行框架(该框架集成了覆盖率分析工具 Cobertura[12] 以及变异分析工具 Major[13]),获取语句覆盖率、分支覆盖率、变异体覆盖率以及变异分数。

Cobertura 是开源的 Java 代码覆盖率分析工具,在 Defects4j 数据集上,通过命令行在指定的项目版本上运行测试用例,最终生成覆盖率分析结果以及相应的日志文件。分析结果包含了被测类总行数、总分支数、语句覆盖数以及分支覆盖数。

Major[14] 是一个针对 Java 语言的变异分析工具,用于缺陷传播和变异分析。Major 提供了一组常用的变异运算符(常量替换、运算符替换、修改分支条件和删除语句),对原版本进行变异,以测试用例覆盖变异体、检测变异体的能力评估现有测试用例的质量。

2.3 实验环境设置

本文实验对象为 Defects4j 上的 4 个开源项目,且覆盖率测量工具以及变异分析工具也集成在该数据集上。正确使用 Defects4j 的运行环境需如表 2 所列。

表 2 实验环境

Table 2 Experimental environment

操作系统	实验平台	实验工具
Linux, Ubuntu-16.4	Defects4j[15]	Randoop:自动化单元测试工具 Cobertura:代码覆盖率测量工具 Major:变异分析工具
Windows7/8/10		Source Monitor:代码特征提取工具 R:数据分析工具

2.4 实验步骤设计

实验操作流程如图 2 所示。首先,针对 4 个项目的 224 个版本,使用 Randoop 工具为被测类生成 1 344 个测试用例集。Randoop 生成的测试用例中可能包含不可编译或不稳定的测试用例,需要去除这些测试用例。其次,使用 Junit 工具,在被测类上运行测试用例;使用 Cobertura 工具,进行覆盖率分析;使用 Major 工具,进行变异分析。然后,为了回答研究问题(3),使用 SourceMonitor 工具来提取被测类的程序结构特征数据。最后,对实验获取的数据进行综合分析,发现其中的内在联系和规律。

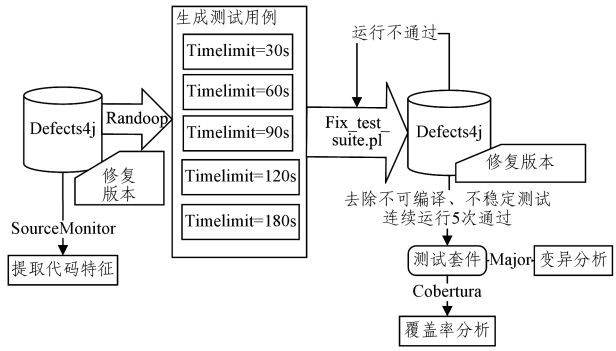

图 2 实验步骤

Fig. 2 Overview of experimental steps

2.4.1 测试用例生成和筛选

(1)Randoop 生成随机测试用例集

在 Defects4j 数据集上运行 Randoop 脚本,为 4 个开源项目的 224 个版本各生成 6 组测试用例集,生成时间参数分别设置为 30 s,60 s,90 s,120 s,180 s 和 240 s。除--null-ratio 参数以外,Randoop 的其余参数均采用默认值;--null-ratio 表示将 null 作为方法调用的参数的频率,其默认值为 0.05,本实验设置为 0.1,这提升了 Randoop 在实验中的有效性[6]。最终,Randoop 为该数据集生成了 1 344 个测试用例集,每个测试用例集包含 50～1 000 不等的测试用例。

(2)测试用例的预处理

Randoop 生成的测试用例中有不可编译的,即不能在后续执行这些测试用例,须将其删除。同时,也可能会生成不稳定测试用例,即这些测试用例在运行时,有时通过有时失败。其原因有两个:一个是全局副作用,即 Randoop 生成的测试用例可能会调用静态字段,创建或删除类的方法;另一个原因是非确定性,即测试用例在不同的运行环境中的表现可能不同,如测试用例包含了引用系统时间的断言。针对上述测试用例,我们需要去除无法编译和无法运行的测试用例,并修复不稳定的测试用例。本文采用两种解决方法:1)通过更改程序(Hash 值、时间函数和随机函数等)使测试用例运行稳定;2)通过使用 Randoop 命令行参数--flaky-test-behavior 自动修复不稳定测试用例。

本实验在 fixed 版本上重复执行测试用例,将不能通过的测试用例删除,并通过 Randoop 自带功能修复不稳定的测试用例,最终在 1344 个测试用例集中删除了 30% 的无法编译和无法运行的测试用例。

2.4.2 计算测试用例的覆盖率

在 Defects4j 数据集上运行 Cobertura 脚本,将收集到的经过处理的测试用例集在对应的源代码版本上执行,得到测试用例覆盖率结果以及日志文件,包括被测类的代码总行数、总分支数、语句覆盖数和分支覆盖数,并计算语句覆盖率和分支覆盖率。

2.4.3 计算测试用例的变异分数

在 Defects4j 数据集上运行 Major 脚本,对被测源代码版本进行变异,得到变异版本,并在变异版本上执行对应的测试用例集,得到测试用例的变异结果和日志文件,计算变异体覆盖率和变异分数。变异体覆盖数是指到达并执行了变异代码的测试用例数,杀死变异体数是指发现变异体

错误的测试用例数[8]。

2.4.4　被测类源代码结构特征的提取

实验使用 SourceMonitor 工具对 224 个被测类源代码进行代码结构特征的提取。该工具针对不同语言会输出不同的度量元，对于 Java 语言，可以反馈总行数、语句数、分支语句比例、注释比例、类个数、总方法数/类、总语句数/方法、最大圈复杂度、最大的函数深度、平均圈复杂度以及平均函数深度，并将 224 个被测类源代码逐一导入，获取其代码结构特征。

3　实验结果分析

3.1　Randoop 生成的测试用例有效性分析

本实验针对 224 个版本生成了 6 组不同时间参数的测试用例集，利用 Cobertura 工具计算其方法覆盖率、语句覆盖率、分支覆盖率、变异体覆盖率以及变异体杀死率。因篇幅有限，表 3 仅展示部分项目的代码覆盖率和变异分析数据，以说明实验可获取的数据形式。

表 3　部分项目版本的覆盖率、分支覆盖率、变异体覆盖率以及变异体杀死率

Table 3　Coverage ratios and mutation scores of generated test suites for part of subject program

项目	版本号	时间	总语句数	语句覆盖数	总分支数	分支覆盖数	变异体生成数	变异体覆盖数	变异体杀死数	语句覆盖率/%	分支覆盖率/%	变异体覆盖率/%	变异体杀死率/%
Chart	1	30	519	82	242	20	444	56	12	15.80	8.26	12.61	2.70
Chart	1	60	519	88	242	20	444	60	13	16.96	8.26	13.51	2.93
Chart	1	90	519	133	242	36	444	98	31	25.63	14.88	22.07	6.98
Chart	1	120	519	147	242	39	444	117	35	28.32	16.12	26.35	7.88
Chart	1	180	519	139	242	37	444	89	23	26.78	15.29	20.05	5.18
Chart	1	240	519	177	242	52	444	126	39	34.10	21.49	28.38	8.78
Math	2	30	66	51	26	18	181	135	70	77.27	69.23	74.59	38.67
Math	2	60	66	65	26	25	181	177	133	98.48	96.15	97.79	73.48
Math	2	90	66	49	26	17	181	135	59	74.24	65.38	74.59	32.60
Math	2	120	66	59	26	22	181	168	110	89.39	4.62	92.82	60.77
Math	2	180	66	64	26	24	181	177	155	96.97	92.31	97.79	85.64
Math	2	240	66	59	26	22	181	168	148	89.39	84.62	92.82	81.77

3.1.1　测试用例的代码覆盖率

图 3、图 4 为不同时间设置下，各项目和所有项目的平均语句覆盖率、平均分支覆盖率的统计图。

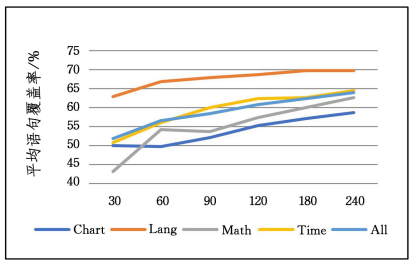

图 3　Chart，Lang，Math，Time 项目的平均语句覆盖率（电子版为彩色）

Fig. 3　Statement coverage rate of generated test suites for all projects

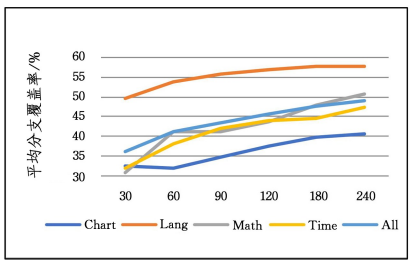

图 4　Chart，Lang，Math，Time 项目的平均分支覆盖率（电子版为彩色）

Fig. 4　Branch coverage rate of generated test suites for all projects

可以看出，当生成时间为 30 s 时，Randoop 为 Math 项目生成平均语句覆盖率为 43.13%的测试用例，为 Lang 项目生成平均语句覆盖率为 62.72%的测试用例；从 4 个项目的 224 个版本来看，Randoop 的平均语句覆盖率为 51.63%。随着生成时间的增加，各项目的平均语句覆盖率和平均分支覆盖率呈上升趋势。当生成时间增加至 240 s 时，Chart，Lang，Math 和 Time 项目的平均语句覆盖率分别为 58.62%，69.71%，62.48%和 64.38%，所有项目的平均语句覆盖率为 63.80%。

3.1.2　测试用例的变异分析

变异测试是一种基于缺陷的软件测试技术，通过对被测类源代码进行符合语法的变更，来评估测试用例的测试充分性。Jalali 等[8]研究证明了在软件测试中变异体可以有效替代真实缺陷，他们得出结论：检测出更多变异体的测试套件具有更高的真实缺陷检测率，即在比较测试套件的表现时，可以使用变异体代替真实缺陷来衡量测试用例的质量。

图 5、图 6 为各项目在不同生成时间下的平均变异体覆盖率和平均变异体杀死率统计图。可以看出，当生成时间为 30 s 时，Randoop 可以为 Math 项目生成平均变异体覆盖率为 39.39%、平均变异分数为 22.09%的测试用例，为 Lang 项目生成平均变异体覆盖率为 59.79%、平均变异分数为 27.69%的测试用例；从 4 个项目的 224 个版本来看，Randoop 工具生成的测试用例的平均变异体覆盖率和平均变异分数分别为 46.53%和 21.14%。当时间增加为 240 s 时，各项目的平均变异体覆盖率和平均变异分数均呈上升趋势。变异体杀死率＝检测到的变异体数/生成的变异体数，由于本实验没有分析未被检测到的变异体中是否存在大量等价变异体，导致变异体杀死率仅为变异体覆盖率的一半。但从变异体覆盖率的角度看，Randoop 生成的测试用例是有效的。

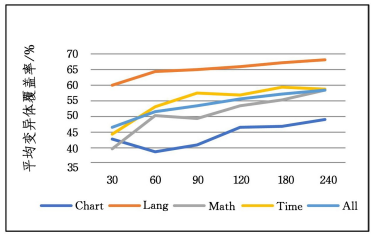

图 5 Chart, Lang, Math, Time 项目的平均变异体覆盖率
(电子版为彩色)

Fig. 5 Mutate coverage rate of generated test suites for all projects

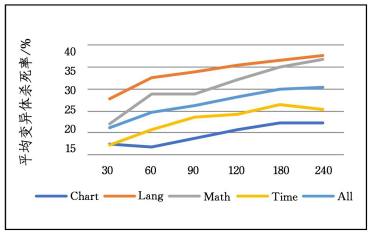

图 6 Chart, Lang, Math, Time 项目的平均变异分数
(电子版为彩色)

Fig. 6 Mutation score of generated test suites for all projects

3.2 生成测试用例时间成本与测试用例有效性的关系

使用 Randoop 时需要设置时间参数来控制测试用例生成过程的结束,本节研究时间成本与测试用例有效性之间的关系。从图 3—图 6 可以看到,时间参数从 30 s 增加至 240 s 时,4 个项目的平均语句覆盖率、平均分支覆盖率、平均变异覆盖率和平均变异分数有不同程度的变化,本文通过绘制各项目的覆盖率和变异分数的箱型图来说明时间与测试用例有效性之间的关系。

3.2.1 Randoop 生成的测试用例的有效性与时间的关系

图 7、图 8 为 Chart 和 Math 项目的语句覆盖率、分支覆盖率的箱型图。

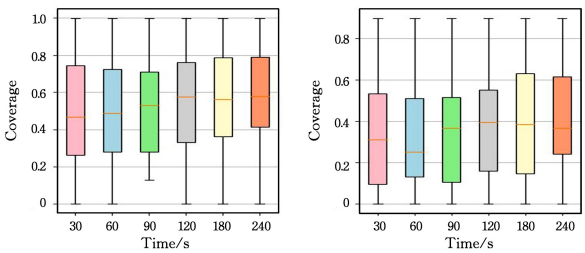

图 7 Chart 项目的语句覆盖率和分支覆盖率的箱型图

Fig. 7 Statement coverage rate and branch coverage rate of Chart

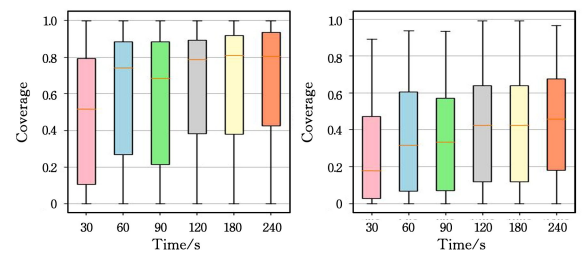

图 8 Math 项目的语句覆盖率和分支覆盖率的箱型图

Fig. 8 Statement coverage rate and branch coverage rate of Math

可以看出,随着时间参数的增加,语句覆盖率和分支覆盖率的分布均呈上升趋势,即 Randoop 工具生成具有低覆盖率的测试用例减少,生成了更多的具有高覆盖率的测试用例。

图 9、图 10 为 Chart 和 Math 项目的变异体检测率和变异分数的箱型图。与代码覆盖情况一样,随着时间的增加,Randoop 工具生成具有低变异体覆盖率、低变异分数的测试用例减少,生成了更多具有高变异体覆盖率和高变异分数的测试用例。

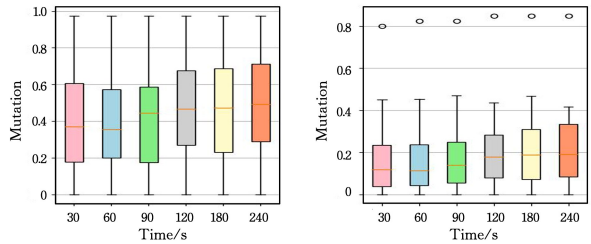

图 9 Chart 项目的变异体覆盖率和变异分数的箱型图

Fig. 9 Mutate coverage rate and mutation score of Chart

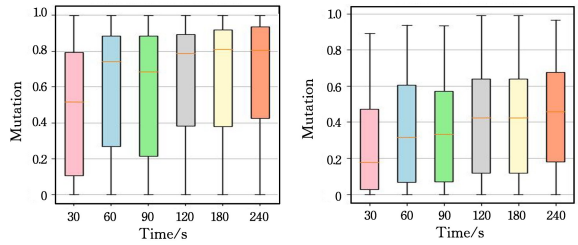

图 10 Math 项目的变异体覆盖率和变异分数的箱型图

Fig. 10 Mutate coverage rate and mutation score of Math

3.2.2 测试用例有效性达到最佳的生成时间

结合上述分析,Randoop 生成的测试用例的有效性随生成时间的增加而提升,但并不是时间设置得越长越好。从 4 个项目看,当生成时间从 30 s 增加至 120 s 时,覆盖率和变异分数的集中分布区间以及中位数有明显上升;而从 120 s 增加至 240 s 时,覆盖率和变异分数的集中分布区间以及中位数相对稳定。

3.3 被测类源代码的结构与测试用例的有效性的关系

本文采用语句数、分支数、方法调用数、平均类方法数(总方法数/类的个数)、平均方法语句数(总语句数/总方法数)、最大圈复杂度、最大函数深度、平均圈复杂度和平均函数深度等特征作为源代码结构特征,这也是衡量代码结构复杂性的主要指标。针对 4 个项目,选用生成时间为 120 s 的测试用例的覆盖率和变异分析数据,通过 SourceMonitor 工具获取了 224 个被测类源代码的代码结构特征。因篇幅有限,表 4 仅展示了 Chart 和 Math 项目的部分被测类源代码的代码结构特征。

为分析被测类源代码的结构与测试用例的有效性的关系,本文通过计算语句覆盖率、分支覆盖率、变异体覆盖率、变异分数与被测类源代码的各项代码结构特征之间的 Pearson 相关系数,来判断被测类源代码的结构与测试用例的有效性之间是否存在相关关系。

$$\rho_{X,Y}=\frac{COV(X,Y)}{\sigma_X\sigma_Y} \tag{1}$$

$$COV(X,Y)=E[(X-E(X))(Y-E(Y))] \tag{2}$$

$$\sigma_X=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i-\mu)^2} \tag{3}$$

$$\sigma_Y=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i-\mu)^2} \tag{4}$$

式(1)是 Pearson 相关系数的计算式，X 为语句覆盖率、分支覆盖率、变异体覆盖率和变异分数；Y 为被测类源代码的各项特征值；COV 是协方差，σ_X 是 X 的标准偏差，σ_Y 是 Y 的标准偏差。Pearson 相关系数衡量了两个变量之间的线性相关程度。研究人员提供了解释相关系数的标准，但是这些标准在某些方面都是任意的，相关系数的解释取决于使用背景和目的[16]。本文借鉴 Kochhar 团队分析测试用例的大小与其有效性之间的相关性时采用的相关系数的解释标准[7]，$\rho_{X,Y}\geqslant 0.80$ 表示极强相关，$0.5\leqslant\rho_{X,Y}<0.80$ 表示强相关，$0.25\leqslant\rho_{X,Y}<0.5$ 表示中等程度相关，$0.10\leqslant\rho_{X,Y}<0.25$ 表示弱相关，$0.00\leqslant\rho_{X,Y}<0.10$ 表示极弱相关或不相关。计算完相关系数后需要对它们进行统计显著性检验，这里假设变量之间不相关(即总体的相关系数为 0)，本文限定显著性水平为 0.01，即当显著性检验中的 $P<0.01$ 时，可以拒绝零假设，即变量之间存在相关关系。

表 4　部分源代码的结构特征

Table 4　Structural characteristics of partial source code

项目名	版本号	语句数	分支数	方法调用数	平均类方法数	平均方法语句数	最大圈复杂度	最大函数深度	平均函数深度	平均圈复杂度
Chart	1	519	242	363	66	7.47	22	6	2.28	3.05
Chart	2	808	548	431	52	15.67	41	8	3.31	7.04
Chart	3	371	194	237	54	6.61	13	5	2.33	2.87
Chart	4	1749	966	1128	229	7.01	56	8	2.48	3.19
Chart	5	174	80	107	34	4.79	16	6	2.32	2.5
Math	1	404	200	88	31	4.97	13	4	2.02	2.48
Math	2	66	26	104	17	6.41	6	3	1.89	1.35
Math	3	421	236	94	10.5	9.76	18	7	2.63	3.88
Math	4	204	170	159	44	4.55	9	4	2.03	2.91
Math	5	38	24	28	7	4.86	6	4	1.65	2.71

表 5 列出了被测类源代码的各项代码结构特征与语句覆盖率、分支覆盖率、变异覆盖率和变异分数之间的 Pearson 相关系数。测试用例的覆盖率和变异分数与被测类源代码的平均方法语句数、最大函数深度、平均函数深度及平均函数复杂度都存在着中等程度的负相关性，即被测代码越复杂，所生成的测试用例的覆盖率和检错能力越低。

表 6 列出了显著性水平 $\alpha=0.01$ 时假设变量间不相关的概率。当 $P<0.01$ 时，拒绝零假设，接受原假设，故测试用例的覆盖率和变异分数与被测类源代码的平均方法语句数、最大函数深度、平均函数深度及平均函数复杂度存在中等程度的相关关系。

表 5　被测类源代码的代码结构特征与覆盖率和变异分数之间的 Pearson 相关系数

Table 5　Pearson correlation coefficient between code structure characteristics of source code and coverage and mutation scores

性能指标	结构特征								
	语句数	分支数	方法调用数	平均类方法数	平均方法语句数	最大圈复杂度	最大函数深度	平均函数深度	平均圈复杂度
语句覆盖率	0	0.06	−0.06	0.07	−0.4	−0.2	−0.38	−0.45	−0.43
分支覆盖率	0	0.07	−0.03	0.06	−0.35	−0.16	−0.34	−0.37	−0.35
变异体覆盖率	0.04	0.12	−0.01	0.12	−0.33	−0.2	−0.33	−0.39	−0.36
变异分数	−0.02	0.04	−0.08	0.03	−0.28	−0.21	−0.42	−0.41	−0.32

表 6　相关性的显著性检验

Table 6　Significant test of correlation

性能指标	结构特征								
	语句数	分支数	方法调用数	平均类方法数	平均方法语句数	最大圈复杂度	最大函数深度	平均函数深度	平均圈复杂度
语句覆盖率	0.98	0.49	0.45	0.38	0	0.01	0	0	0
分支覆盖率	0.99	0.39	0.69	0.45	0	0.05	0	0	0
变异体覆盖率	0.64	0.16	0.88	0.14	0	0.02	0	0	0
变异分数	0.85	0.65	0.36	0.73	0	0.01	0	0	0

4　实验局限性

实验仅选取了 4 个开源的 Java 项目，被测类源代码为这 4 个项目中的 224 个版本。考虑到程序的内在属性，我们不确定这些被测类源代码是否具有典型性，以及实验结果是否具有通用性。我们在分析 Randoop 生成的测试用例的有效性时，采用覆盖率以及变异分数为衡量标准。虽然在先前的研究中[7-8,17-18]已证实代码覆盖率和缺陷检测率之间存在显著的统计学相关性，并且变异体可以有效地替代真实缺陷来衡量测试用例的质量，但是衡量测试用例有效性最直接的标准是其缺陷检测率，因此以覆盖率和变异分数作为衡量标准可能会高估或者低估 Randoop 的性能。

实验中的变异分析选用的是常用的变异算子，并未选用全部变异算子，且没有分析未被杀死的变异体中是否存在等价变异体，这可能会导致变异体杀死率较低。因此，本文的结果分析不仅考虑了变异体杀死率，也会对比分析变异体覆盖

率，以确保结果的可靠性。

结束语 本文选取了4个开源的Java项目的224个版本，使用Randoop为其生成6组不同时间参数的测试用例，以覆盖率和变异分数为衡量标准来评估Randoop的性能。本文的研究回答了前文的3个问题。

(1)Randoop生成的测试用例有效吗？

Randoop可以在短时间内为测试人员生成有效的单元测试用例，当生成时间为30 s时，平均语句覆盖率为51.63%，平均分支覆盖率为36.24%，平均变异体覆盖率为46.53%，平均变异分数为21.14%。

(2)测试用例的有效性与测试用例的生成成本有关系吗？

通过绘制4个项目在不同时间下的测试用例代码覆盖率和变异分数的箱型图可知，随着时间成本的增加，测试用例有效性呈上升趋势，当时间成本为120 s时，其有效性趋于稳定。

(3)测试用例的有效性与被测类源代码的结构有关系吗？

通过Person相关系数分析，Randoop生成的测试用例与被测类源代码的平均方法语句数、最大函数深度、平均函数深度和平均函数复杂度存在中等程度的负相关关系。

在未来的工作中，基于本文的实验，可以加入更多的被测类源代码来分析Randoop适用的场景；同时，通过提高Randoop生成的测试用例的覆盖率来提高测试用例的有效性，为工具的改进提供建议。

参 考 文 献

[1] Randoop官网介绍 [EB/OL]. [2019-06-12]. https://randoop.github.io/ randoop/.

[2] RUEDA U, VOS T E J, PRASETYA I S W B. Unit Testing Tool Competition — Round Three[C]// 2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing (SBST). ACM, 2015.

[3] RUEDA U, JUST R, GALEOTTI J P, et al. Unit Testing Tool Competition — Round Four[C]// 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST). Austin, TX, 2016: 19-28.

[4] PANICHELLA A, MOLINA U R. Java unit testing tool competition— fifth round [C] // 10th International Workshop on Search-Based Software Testing. IEEE P, 2017: 32-38.

[5] RUEDA MOLINA U, KIFETEW F, PANICHELLA A. Java unit testing tool competition— sixth round[C]// 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST). Gothenburg, 2018: 22-29.

[6] SHAMSHIRI S, JUST R, ROJAS J M, et al. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges[C]// International Conference on Automated Software Engineering (ASE). IEEE, 2015: 201-211.

[7] KOCHHAR P S, THUNG F, LO D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems[C]// IEEE International Conference on Software Analysis, Evolution and Reengineering. IEEE, 2015: 560-564.

[8] JALALI D, INOZEMTSEVA L, ERNST M D, et al. Are mutants a valid substitute for real faults in software testing? [C]// ACM Sigsoft International Symposium on Foundations of Software Engineering. ACM, 2014: 654-665.

[9] Defects4j官网介绍 [EB/OL]. [2019-06-12]. https://github.com/rjust/defects4j.

[10] PAN N G, ZENG F P, CAO Q. Automatic Generation and Reduction of Random Test Case[J]. Journal of Chinese Computer Systems, 2011, 32(10): 2035-2040.

[11] PACHECO C, ERNST M D. Randoop: feedback-directed random testing for Java[C]// Companion To the, ACM Sigplan Conference on Object-Oriented Programming Systems and Applications Companion. ACM, 2007: 815-816.

[12] Cobertura官网介绍[EB/OL]. [2019-06-12]. http://cobertura.github.io/cobertura/.

[13] Major官网介绍[EB/OL]. [2019-06-12]. http://mutation-testing.org/.

[14] JUST R, SCHWEIGGERT F, KAPFHAMMER G M. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler[C]// 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). ACM, 2011: 612-615.

[15] JALALI D, ERNST M D. Defects4J: a database of existing faults to enable controlled testing studies for Java programs[C]// International Symposium on Software Testing and Analysis. ACM, 2014: 437-440.

[16] 相关性系数的解释 [EB/OL]. [2019-06-13]. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[17] ALMASI M M, HEMMATI H, FRASER G, et al. An industrial evaluation of unit test generation: finding real faults in a financial application[C]// 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). ACM, 2017: 263-272.

[18] RAMLER R, WINKLER D, SCHMIDT M. Random test case generation and manual unit testing: substitute or complement in retrofitting tests for legacy code? [C]// 2012 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA). IEEE Computer Society, 2012: 286-293.

LIU Fang, born in 1995, postgraduate. Her main research interests include automated unit test generation tools and mutation testing.

HONG Mei, born in 1963, master's degree, professor, is a member of China Computer Federation. Her main research interests include software engineering and software automation testing.