软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(2):532−580 [doi: 10.13328/j.cnki.jos.006953]

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

测试集有效性评价: 问题、进展与挑战*

路则雨[1,2], 张 鹏[1,2], 王 洋[1,2], 郭肇强[1,2], 杨已彪[1,2], 周毓明[1,2]

[1](计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 计算机科学与技术系, 江苏 南京 210023)

通信作者: 杨已彪, E-mail: yangyibiao@nju.edu.cn; 周毓明, E-mail: zhouyuming@nju.edu.cn

摘 要: 测试用例集的缺陷检测有效性指测试集能够在多大程度上检测出软件中存在的缺陷. 如何评价测试集的缺陷检测有效性是一个重要问题. 覆盖率和变异得分是两个最重要和最广泛使用的测试集有效性度量. 为量化测试集的缺陷检测能力, 研究人员对测试集有效性评价进行了大量研究并且取得了较大的进展. 与此同时, 现有研究存在不一致的结论, 该领域依然存在一些亟待解决的挑战. 对多年来国内外学者在测试集有效性评价领域的研究成果进行系统性的梳理和总结. 首先, 阐述测试集有效性评价研究中的问题. 然后, 介绍并分析基于覆盖率和基于变异得分的测试集有效性的评价以及介绍测试集有效性评价在测试集优化中的应用. 最后, 指出测试集有效性评价研究中面临的挑战并给出建议的研究方向.

关键词: 覆盖率; 变异得分; 缺陷检测; 测试集有效性

中图法分类号: TP311

中文引用格式: 路则雨, 张鹏, 王洋, 郭肇强, 杨已彪, 周毓明. 测试集有效性评价: 问题、进展与挑战. 软件学报, 2024, 35(2): 532–580. http://www.jos.org.cn/1000-9825/6953.htm

英文引用格式: Lu ZY, Zhang P, Wang Y, Guo ZQ, Yang YB, Zhou YM. Evaluation of Test Suite Effectiveness: Problem, Progress, and Challenges. Ruan Jian Xue Bao/Journal of Software, 2024, 35(2): 532–580 (in Chinese). http://www.jos.org.cn/1000-9825/6953.htm

Evaluation of Test Suite Effectiveness: Problem, Progress, and Challenges

LU Ze-Yu[1,2], ZHANG Peng[1,2], WANG Yang[1,2], GUO Zhao-Qiang[1,2], YANG Yi-Biao[1,2], ZHOU Yu-Ming[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: The effectiveness of a test suite in defect detection refers to the extent to which the test suite could detect the defects hidden in the software. How to evaluate this performance of a test suite is an important issue. Coverage and mutation score are two of the most important and widely used metrics for test suite effectiveness. To quantify the defect detection capability of a test suite, researchers have devoted a large amount of research effort to this issue and have made significant progress. However, inconsistent conclusions can be observed among the existing studies, and some challenges still call for prompt resolution in the area. This study systematically summarizes the research results achieved by scholars both in China and abroad in the field of the evaluation of test suite effectiveness over the years. To start with, it expounds the problems in the research on the evaluation of test suite effectiveness. Then, it outlines and analyzes the evaluation of test suite effectiveness based on coverage and mutation score and presents the application of the evaluation of test suite effectiveness in test suite optimization. Finally, the study points out the challenges faced by this line of research and suggests the directions of future research.

Key words: coverage; mutation score; defect detection; test suite effectiveness

* 基金项目: 国家自然科学基金 (62172205, 62072194)
收稿时间: 2022-10-26; 修改时间: 2023-02-23; 采用时间: 2023-04-09; jos 在线出版时间: 2023-08-30
CNKI 网络首发时间: 2023-09-01

软件测试是一种重要的软件质量保证技术, 其目的是检测软件产品中尽可能多的缺陷. 在测试过程中, 测试用例被应用于被测程序, 以验证执行结果是否与预期结果一致. 当执行结果与预期结果不一致时, 表明测试用例检测到了被测程序中的缺陷. 一个测试集发现的缺陷越多, 其缺陷检测能力越高.

软件测试需要花费大量的时间和精力. 据估计, 软件开发人员花费接近一半的时间用于测试他们编写的代码[1]. 如果测试不够充分, 会对被测软件的质量产生负面影响; 如果进行过多的无效测试, 则会导致测试成本过高从而降低效益. 因此, 需对测试集的缺陷检测有效性进行评价, 以便在软件测试成本与软件质量之间进行权衡.

如何评价测试集的有效性是一个重要的研究问题. 测试集的有效性是指测试集在多大程度上能够检测出被测程序中存在的缺陷. 检测到被测程序中更多比例缺陷的测试集的有效性更高. 然而, 由于被测软件中的缺陷数量是未知的, 因此无法直接量化测试集的缺陷检测有效性. 为此, 人们通常使用测试集的特征作为测试集缺陷检测有效性的代理, 通过覆盖率或变异得分等评价测试集的有效性. 目前, 测试集有效性的两个最重要和最广泛使用的度量是代码覆盖率和变异得分. 覆盖率是指测试集执行时覆盖到的被测程序的元素 (例如语句、分支) 的比例, 变异得分是指测试集能够杀死的被测程序变异体的比例.

覆盖率和变异得分应用广泛. 覆盖率广泛用于各种相关领域, 例如测试用例自动化生成[2]、灰盒模糊测试[3,4]、变异测试[5]、回归测试[6]、增量程序测试[7]、变更影响分析[8]、缺陷预测[9]和缺陷定位[10]等. 变异得分被应用于各种任务, 例如测试数据生成[11]、测试集约简[12]、测试用例优先排序[13,14]、缺陷预测[15]和深度学习模型测试[16]. Zhu 等人[17]对变异测试的应用进行了系统的文献综述. 因此, 对测试集有效性评价的研究具有重要意义.

在过去的几十年间, 测试集有效性评价领域的研究工作十分活跃. 到目前为止, 许多工作研究了覆盖率和缺陷检测有效性之间的关系[18−22], 也有许多工作研究了变异得分与缺陷检测有效性之间的关系[23−30]. 然而, 由于研究方法等存在差异, 现有研究对覆盖率和变异得分与缺陷检测有效性的关系存在一些不一致的结论, 测试集有效性评价的研究仍然面临重要的挑战.

(1) 覆盖率与缺陷检测有效性的关系不一致. 一些研究表明覆盖率与缺陷检测有效性相关性强, 而另一些研究表明覆盖率与缺陷检测有效性相关性弱.

(2) 变异得分与缺陷检测有效性的关系不一致. 一些研究表明变异得分与缺陷检测有效性之间的相关性强, 而另一些研究表明变异得分与缺陷检测有效性相关性弱.

(3) 测试集有效性评价研究的研究方法不统一. 实验设置中的测试用例来源、测试集构造方式、测试充分性准则、统计方法、缺陷类型等不尽相同.

值得注意的是, 尽管存在上述诸多挑战, 但却很少有文献对测试集有效性评价领域的研究进行总结. 据我们所知, 尽管有许多实证研究探讨并简单回顾了测试集有效性评价的研究工作, 目前没有关于测试集有效性评价的总结性的综述工作 (截至 2022 年). 由于对测试集有效性评价研究的论文体量已经较为庞大, 为了弥补这一空白, 同时为了方便后续相关研究工作的进行, 本文对 1991–2022 年期间的相关研究进行了全面地回顾, 对测试集有效性评价的研究进展进行分类介绍和总结, 并在此基础上指出测试集有效性评价领域所面临的挑战并展望未来的研究方向.

本文第 1 节概述测试集有效性评价领域的主要的研究问题. 第 2 节说明本文的文献检索与筛选方式和文献汇总信息. 第 3 节详细介绍基于覆盖率和基于变异得分的评价, 并进一步介绍面向领域软件的测试集有效性评价. 第 4 节介绍测试集有效性的影响因素. 第 5 节从实验设置的不同层面介绍测试集有效性评价分析. 第 6 节基于测试集优化介绍测试集有效性评价的应用. 第 7 节分析测试集有效性评价的现有挑战和未来的研究方向. 最后第 8 节总结本文的内容.

1 研究问题

1.1 测试集有效性评价概要介绍

一个测试用例 t 由两部分组成: 一个用于执行被测程序的测试输入 (test input) 和一个用于检查测试执行正确性的测试预言 (test oracle). 图 1 展示了基本的数据结构栈的代码, 示例代码来源于 EvoSuite[31]. 图 2 中 test0 和 test1 是使用 EvoSuite 生成的两个测试用例. 测试预言通常采用可执行断言的形式, 例如图 2 测试用例 test0 中的 assertTrue(stack0.isEmpty()) 和 assertFalse(stack0.isEmpty()) 语句.

```java
package tutorial;

import java.util.EmptyStackException;

public class Stack<T> {
    private int capacity = 10;
    private int pointer  = 0;
    private T[] objects = (T[]) new Object[capacity];

    public void push(T o) {
      if(pointer >= capacity)
          throw new RuntimeException("Stack exceeded capacity!");
      objects[pointer++] = o;
    }
    public T pop() {
      if(pointer <= 0)
          throw new EmptyStackException();
      return objects[--pointer];
    }
    public boolean isEmpty() {
      return pointer <= 0;
    }
}
```

图 1　栈的代码

```java
@Test(timeout = 4000)
public void test0()  throws Throwable  {
    Stack<String> stack0 = new Stack<String>();
    assertTrue(stack0.isEmpty());

    stack0.push((String) null);
    stack0.push("{y{j!vqtKrk^P~");
    stack0.pop();
    assertFalse(stack0.isEmpty());
}
/*                                                  */
@Test(timeout = 4000)
public void test1()  throws Throwable  {
    Stack<Object> stack0 = new Stack<Object>();
    stack0.push((Object) null);
    assertFalse(stack0.isEmpty());

    stack0.pop();
    assertTrue(stack0.isEmpty());
}
```

图 2　栈的代码的测试用例

测试集 T 是被测程序的测试用例 t 的集合, $t \in T$. 测试集的有效性即测试集的缺陷检测能力, 可以量化为测试集检测到的缺陷数量与缺陷总数的百分比. 为此, 需要得知被测程序的缺陷总数, 然而在实际中缺陷总数是未知的. 如图 3 所示, 因为被测软件中的缺陷数量是未知的, 无法直接量化测试集的缺陷检测有效性, 所以测试集的覆盖率或变异得分通常用作测试集缺陷检测能力的代理来评价测试集的有效性.

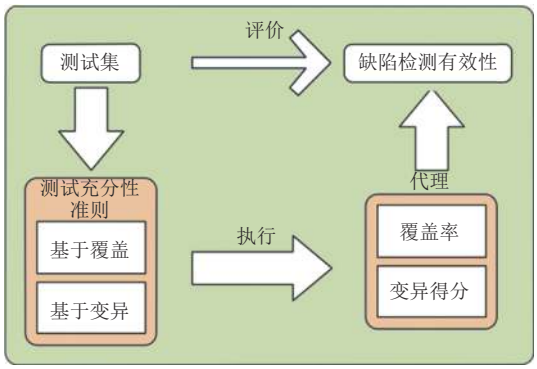


图 3　测试集有效性评价

代码覆盖率或变异得分越高, 测试集的缺陷检测有效性越高. 代码覆盖率和变异得分是根据测试充分性准则在被测程序上执行测试集得到的. 测试充分性准则是基于何时确定停止测试过程[32]提出的. 测试充分性准则定义了测试过程的要求, 并以 3 种不同的方式指导测试人员[33]: 通过指出在设计测试时应该执行的元素、通过提供终止测试的标准以及通过量化测试集的充分性. 测试充分性准则包括基于覆盖的准则和基于变异的准则.

(1) 基于覆盖的准则

覆盖准则有两种基本类型: 基于输入的准则和基于预言的准则. 基于输入的准则量化由测试输入运行程序执行的源代码的百分比. 这样的例子包括语句覆盖、分支覆盖、路径覆盖和边界值覆盖等. 基于预言的准则考虑了测试预言信息, 量化由测试预言检查的代码的百分比. 断言覆盖即断言的动态后向切片, 也称为检查覆盖, 属于这一类. 覆盖准则作为一种评估测试集的方法已被广泛接受, 以至于在某些领域, 例如航空电子设备, 强制要求使用修正条件/判定覆盖[34].

(2) 基于变异的准则

变异测试由 DeMillo 等人[35]和 Hamlet[36]提出. 变异测试可以通过对被测程序进行一个或多个语法变更模拟软件产品的真实缺陷.

如图 4 所示, 变异测试首先根据变异算子 (mutation operators) 即对原始程序 p 的语法的更改规则创建一组变异体 M (mutants). 变异体 $p' \in M$ 是原始程序 p 的错误版本. 然后测试集 T 应用于原始程序 p. 为了评估给定测试

集 T 的有效性, 对一组变异体 M 执行测试集 T. 如果变异体 p' 的执行得到与原始程序 p 不同的输出, 则称该变异体 p' 被测试用例“杀死”. 对于给定的一组变异体 M, 测试集 T 杀死的 M 中变异体的比例称为 T 的变异得分. 通常, 变异得分越高则认为测试集 T 的缺陷检测能力越强.

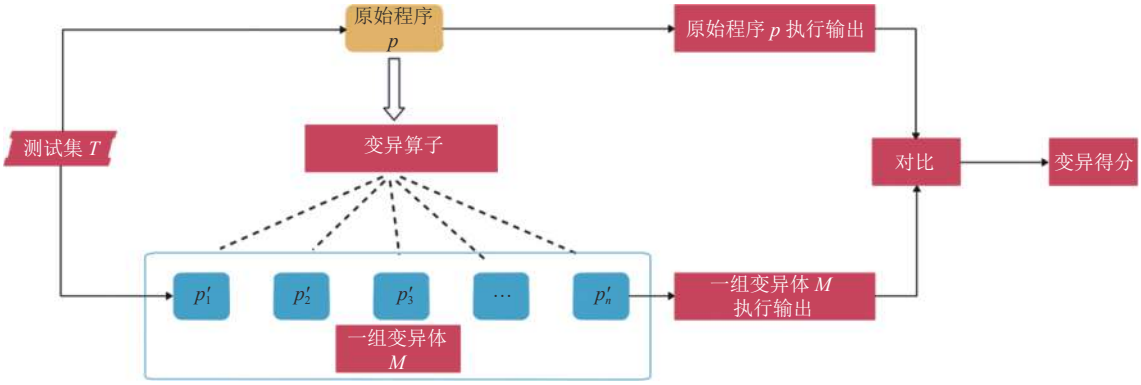


图 4 变异分析的过程

1.2 测试集有效性评价研究内容

目前, 测试集有效性评价领域的研究重点是覆盖率和变异得分与缺陷检测有效性的关系. 因此, 本文首先分析现有研究得出的覆盖率和变异得分与缺陷检测有效性的关系, 总结现有研究的主要发现.

1.2.1 基于覆盖率和基于变异得分的评价

人们并不清楚覆盖率和变异得分能否作为可靠的测试集缺陷检测有效性的评价准则, 因此对覆盖率和变异得分作为缺陷检测有效性评价准则是否可靠的研究是测试集有效性评价领域的研究热点. 研究人员探讨了:

(1) 覆盖率与缺陷检测有效性的关系.

(2) 变异得分与缺陷检测有效性的关系.

具体内容将在第 3 节详细介绍.

1.2.2 测试集有效性评价的研究方法

目前, 研究人员得出的覆盖率或变异得分与缺陷检测有效性之间的关系的结论不一致. 为了分析不一致的结论产生的原因, 本文从测试集有效性评价研究的研究方法的不同层面, 包括测试用例来源、测试集构造方式、测试充分性准则、影响因素、统计方法、缺陷类型, 分析测试集有效性评价的现有研究. 具体内容将在第 4 节和第 5 节详细介绍.

1.2.3 测试集有效性评价的应用

目前, 覆盖率和变异得分广泛用于各种相关领域. 本文关注覆盖率和变异得分在测试集维护即回归测试测试集优化中的应用. 当前的测试集约简、回归测试选择和测试用例优先排序等测试集优化方法大多建立在代码覆盖率或变异得分的基础上. 本文探讨:

(1) 基于覆盖和基于变异的测试集优化方法.

(2) 测试集优化与缺陷检测能力的关系.

具体内容将在第 6 节详细介绍.

2 文献检索

在本节中, 首先描述文献的检索与筛选方法, 然后从年代分布、出版物分布和时间进展这 3 个方面展示文献的筛选结果.

2.1 文献检索与筛选

软件测试是开发高质量软件的关键活动, 对测试集有效性评价的研究一直以来是热点研究问题. 本文的参考

文献力图覆盖 1991–2022 年与测试集有效性评价相关的主要研究工作. 具体来说, 本文使用以下的步骤进行相关文献的检索与筛选.

(1) 检索目标: 谷歌学术搜索引擎 (Google Scholar)、DBLP computer science bibliography、ACM Digital Library、IEEE Xplore Digital Library 和 SpringerLink Digital Library 论文数据库. 检索的文献主要来自软件工程方向一流的国际会议 ICSE、FSE/ESEC、ASE 和 ISSTA 等和一流的期刊 TSE 和 TOSEM 等以及其他重要的会议和期刊如 ISSRE、Journal of Systems and Software 等.

(2) 检索关键词: 搜索的关键词为“test/testing/test case/test suite effectiveness”“test/testing/test case/test suite fault/bug/defect detection”“test size effectiveness”“coverage effectiveness/fault detection”“mutation effectiveness/fault detection”“mutation testing effectiveness/fault detection”“test faults”“test quality”.

本文在数据库中对文献的题目和内容使用上述关键词进行搜索.

(3) 检索起止时间: 检索 1991–2022 年 8 月共 32 年间的文献.

(4) 文献审查: 对检索的文献首先进行初步筛选, 为了保证文献质量, 本文不选择会议的 workshop 文献. 通过阅读去掉与研究主题不相关或低相关的文献, 得到了 101 篇文献. 在阅读后, 本文增加了 2 篇重要的 workshop 文献, 最终得到了 103 篇文献.

2.2 历史文献的汇总

经过文献检索与筛选, 本文收集了测试集有效性评价领域中 1991–2022 年间的学术论文共 103 篇. 这些研究论文将会分别按照基于覆盖率[18−22,37−54]和基于变异得分[23−30,55−60]的评价以及面向领域软件的测试集有效性评价[61−84]、测试集有效性的影响因素[20,24,29,44,48,85−101]、测试集有效性评价分析[22,23,29,44,102−107]、测试集有效性评价的应用[12−14,108−123]分别在第 3–6 节进行重点详细介绍.

图 5 展示了 1991 年以来测试集有效性评价相关的研究论文发表的数量分布情况. 从图 5 中可以看出, 该领域发表的论文数量整体上呈现出上升的趋势, 尤其是 2011–2021 年的这 11 年间发表的论文数量较多, 这说明测试集有效性这个领域多年来一直受到研究人员的重视. 图 6 展示了测试集有效性评价领域的研究论文在出版物上的发表分布情况. 其中, 会议论文有 68 篇, 包括 ICSE 论文 13 篇[18,29,40,47,59,61,62,72,82,89,97,102,108]、ESEC/FSE 论文 9 篇[12,23,28,41,63,67,73,75,95]、ISSTA 论文6 篇[38,48,60,88,113,121]、ASE 论文 2 篇[71,91]、SOSP 论文 1 篇[69]、ISSRE 论文 6 篇[13,21,70,87,100,110]、SANER 论文 3 篇[74,90,106]、ICSME 论文 1 篇[85], 还有其他的会议论文 27 篇[20,30,37,43,46,49−52,56,64−66,76,77,79,84,92,94,96,98,101,107,109,111,114,115]; 期刊论文有 35 篇, 包括 TSE 论文 10 篇[22,27,39,44,55,86,104,116,120,123]、TOSEM 论文 4 篇[19,99,117,118]、STVR 论文 4 篇[14,57,78,112]、JSS 论文 2 篇[26,93]、ESE 论文 1 篇[122]、SCP 论文 1 篇[54], 还有其他的期刊论文 13 篇[24,25,42,45,53,58,68,80,81,83,103,105,119]. 从图中可以看出, 对测试集有效性评价的研究是软件工程领域重要的研究方向.

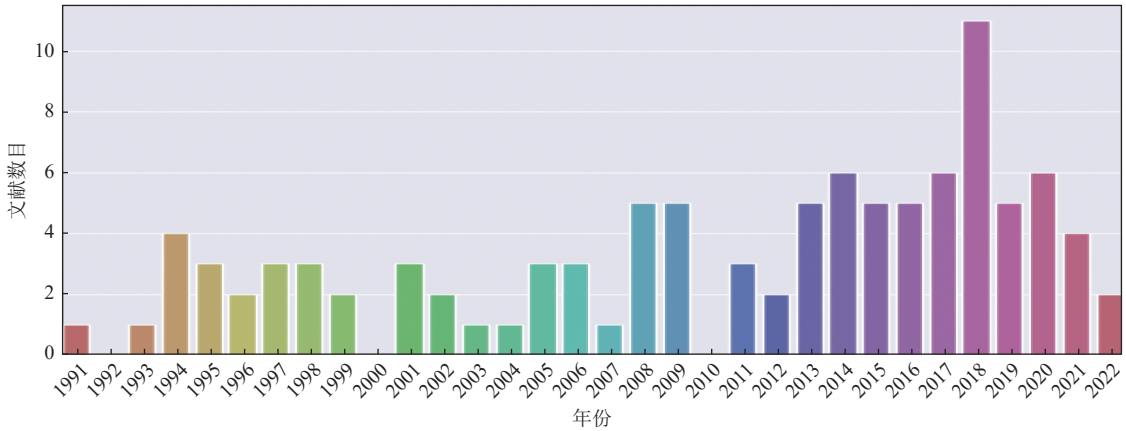


图 5 测试集有效性评价相关研究文献的年代分布

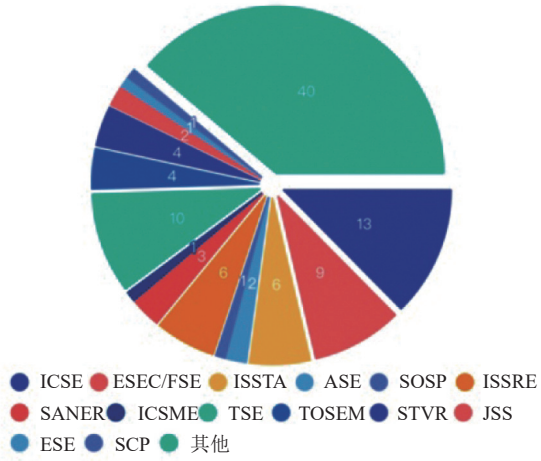


图 6　测试集有效性评价相关研究文献的出版物分布

图 7 展示了对测试集有效性评价研究[18−22,28−30,48,85−87,92−95,97,101,111,112,124,125]的进展时间轴. 时间线上面展示了测试集有效性的影响因素. 这些因素包括测试集规模[87]、缺陷种类[92−94]、测试预言[95]、程序结构[97,98]和每个测试覆盖的方法数量[101]等. 现有的研究得出了许多相互矛盾的结论, 时间线下面展示了得出与大多数研究结论相悖结论的研究, 例如变异得分与缺陷检测能力相关性弱的研究[28−30]. 从图 7 中可以看出, 代表性的研究集中在 2011 年及以后, 这段时间的文献数量有 60 篇, 占筛选后的论文的 58.25%. 而 2000–2010 年的这段时间, 经过文献筛选后得到的论文较少, 只有 24 篇, 代表性的研究也较少.

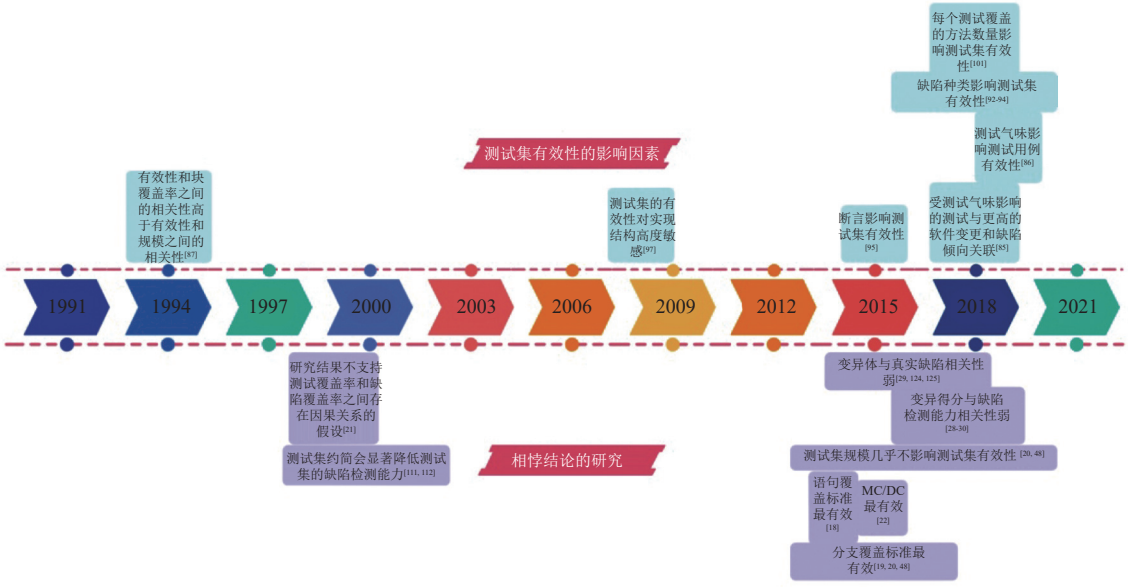


图 7　测试集有效性评价相关研究文献的进展时间轴

3　基于覆盖率和基于变异得分的评价

本节旨在了解代码覆盖率和变异得分与缺陷检测有效性的关系, 即覆盖率和变异得分是否是测试集有效性的可靠指标. 本节分别在第 3.1 节和第 3.2 节探讨覆盖率和变异得分与缺陷检测有效性的关系. 本节在第 3.3 节探讨

面向领域软件的测试集有效性评价.

3.1 基于覆盖率的评价

3.1.1 覆盖准则

表 1 展示了调研的除面向领域软件外的 80 篇文献中使用的基于覆盖的充分性准则的情况. 调研文献中使用的其他的基于覆盖的充分性准则还有修正条件覆盖 (modified condition coverage)[89]、判定/条件覆盖 (decision/condition coverage)[56]、多条件覆盖 (multiple-condition coverage)[56]、异常覆盖 (exception coverage)[20]和输出覆盖 (output coverage)[20]等. 下面对其中的部分覆盖准则进行了详细介绍.

表 1 调研文献使用的覆盖准则

覆盖准则			文献	文献数量
白盒测试	控制流	边覆盖、all-edges覆盖	[38−40,54,111,112]	6
		行覆盖	[20,53,106,110,113]	5
		语句覆盖	[12−14,18,19,23,27,28,37,48,50,51,56,59,86,89,90,92,93,95,96,105,114,115,119,120]	26
		块覆盖	[18,21,42−45,87,88,108,109,119,121−123]	14
		条件覆盖	[22,45,56]	3
		分支覆盖	[18−20,45−48,50−53,59,61,90,92,93,105,116,117,120]	20
		判定覆盖	[21,22,27,41,43,44,56,88,89]	9
		修正条件/判定覆盖	[22,27,50,52,56,97−100,102]	10
		路径覆盖	[18]	1
		方法覆盖	[20,110,114,122,123]	5
	数据流	c-use覆盖、all-c-uses覆盖	[21,43,44,88,103]	5
		p-use覆盖、all-p-uses覆盖	[21,43,44,88,103]	5
		all-uses覆盖	[25,26,38,39,41,57,58,103,108]	9
		def-use覆盖、all-DUs覆盖	[40,47,49,50]	4
黑盒测试		边界值测试	[88]	1
灰盒测试		断言覆盖	[37,51,95,96,114]	5

(1) 白盒测试

白盒测试又称结构化测试. 程序的控制流图是有向图 $CFG=(N, E, entry, exit)$, 其中 N 是一组节点, E 是一组边, $entry$ 和 $exit$ 分别表示程序的入口和出口节点. 每个节点 $n\in N$ 表示程序中的一条语句, 每条边 $e=(n_i,n_j)$ 表示从节点 n_i 到节点 n_j 的控制迁移.

1) 控制流

对于被测程序, 程序的分支谓词称为判定 D, D 中包含一个或多个条件 C, 即 $D=C_1\oplus C_2\oplus\ldots\oplus C_n$, 其中 $\oplus$ 表示 $\wedge$ 或 $\vee$, 即逻辑与和逻辑或.

定义 1 (语句覆盖). 语句覆盖准则要求对每个可访问的程序语句 n 至少执行一次.

定义 2 (条件覆盖). 条件覆盖准则要求每个条件 $C_i\in D$ 至少被执行出一次真和假.

定义 3 (分支覆盖). 分支覆盖准则需要测试集测试程序控制流图的每个可到达分支 D, 即取真值或取假值都被执行到.

定义 4 (修正条件/判定覆盖). 修正条件/判定覆盖 (modified condition/decision coverage, MC/DC) 准则要求被测程序同时满足以下 4 个条件: a) 程序的每个入口节点 $entry$ 和出口节点 $exit$ 都至少被执行过一次; b) 程序中的每个分支 D 都被执行出所有可能的结果, 即真和假; c) 对于每个条件 $C_i\in D$, 至少被执行出一次真和假; d) 对于每个条件 $C_i\in D$, 必须被证明独立影响分支 D 的结果, 即固定其他所有条件 $C_j\ (j\neq i)$, 要求 $D(C_i=\text{true})\neq D(C_i=\text{false})$.

定义 5 (路径覆盖). 路径覆盖准则要求执行路径 P 包含控制流图中从入口节点 *entry* 到出口节点 *exit* 的所有执行路径.

定义 6 (方法覆盖). 方法覆盖准则用于检查应用程序源代码中的每个方法是否被测试执行或覆盖.

2) 数据流

定义 7 (定义-使用对). 变量 v 的定义是在其中定义变量 v 的节点 n_d. 变量 v 的使用是使用 v 的节点 n_u. 关于变量 v 的无定义路径 (definition-free path) 是其中不存在变量 v 的重新定义的路径. 如果在节点 n_d 与节点 n_u 之间的路径上不存在变量 v 的重新定义的路径, 就形成了一个定义-使用对 (def-use pair), 表示为 $du(n_d, n_u, v)$. 如果使用出现在计算表达式 (computational expression) 中, 那么该对是 c-use. 如果使用出现在谓词中, 那么该对是 p-use. all-uses 包括 c-use 和 p-use.

定义 8 (c-use 覆盖). c-use (computation use) 覆盖准则要求对所有的 c-use 对至少覆盖一次.

定义 9 (p-use 覆盖). p-use (predicate use) 覆盖准则要求对所有的 p-use 对至少覆盖一次.

定义 10 (all def-uses 覆盖). all def-uses 覆盖 (缩写 all-DUs 覆盖) 准则要求对所有的定义-使用对至少覆盖一次.

(2) 黑盒测试

黑盒测试又称功能测试.

定义 11 (边界值测试). 边界值测试是在输入范围的边界进行测试的过程. 一般的边界值测试是在以下位置选择输入变量值: 1) 最小值; 2) 略高于最小值; 3) 中间值; 4) 略低于最大值; 5) 最大值.

(3) 灰盒测试

灰盒测试结合了结构信息和功能信息进行测试.

传统覆盖率指标的一个已知问题是它们不评估预言的质量, 也就是说, 不评估计算结果是否真的与预期相符. 在 2011 年, Schuler 等人[37]介绍了断言覆盖 (检查覆盖) 的概念. 断言覆盖 (assertion coverage) 是测试预言的动态后向切片, 即断言检查所覆盖的语句. 他们对 7 个开源项目的实验表明, 检查覆盖是预言质量的可靠指标.

3.1.2 覆盖准则与缺陷检测有效性

使用测试覆盖准则控制软件测试过程已经成为一种越来越普遍的做法. 这是基于更高的测试覆盖率有助于实现更高的缺陷覆盖率并因此提高软件质量的假设. 有许多实证研究探讨了某些准则是否比其他准则更好, 以及覆盖率在衡量测试集的缺陷检测能力方面是否有效. 表 2 列出了覆盖准则和缺陷检测有效性的研究, 下面对其中的部分研究进行详细介绍.

表 2 覆盖准则和缺陷检测有效性

作者	年份	测试用例来源	覆盖准则	统计方法	语言	缺陷类型	主要发现总结
Frankl等人[38,39]	1991, 1993	随机生成	all-edges, all-uses	逻辑回归	Pascal	真实缺陷	all-uses充分的测试集比类似规模的all-edges充分的测试集更有效
Hutchins等人[40]	1994	自动生成, 手动生成	all-edges, all-DUs	—	C	注入缺陷	覆盖率超过90%的测试集通常比随机选择的相同规模的测试集表现出更好的缺陷检测有效性. 仅100%的代码覆盖率并不是测试集有效性的可靠指标. 分别基于控制流和数据流准则的测试在有效性上经常是互补的
Frankl等人[41]	1998	自动生成	分支覆盖, all-uses覆盖	—	C	真实缺陷	对于all-uses和分支覆盖, 当达到非常高的覆盖率时, 检测到缺陷的可能性急剧增加. 两种准则有相似的测试有效性
Briand等人[21]	2000	—	块覆盖, 分支覆盖, c-use覆盖, p-use覆盖	R平方, 调整后的R平方	C	模拟缺陷	研究结果不支持测试覆盖率和缺陷覆盖率之间存在因果关系的假设
Chen等人[42]	2001	自动生成	块覆盖	—	C	模拟缺陷	覆盖率可用于预测运行中的软件失效

表 2 覆盖准则和缺陷检测有效性 (续)

作者	年份	测试用例来源	覆盖准则	统计方法	语言	缺陷类型	主要发现总结
Cai等人[43]	2005	现有的测试集	块覆盖, 判定覆盖, c-use覆盖, p-use覆盖	R平方	C	变异缺陷	缺陷发现能力与4种不同的结构化覆盖准则之间存在中度的相关性. 代码覆盖率和缺陷覆盖率之间的相关性在异常测试中较强
Andrews等人[44]	2006	现有的测试集	块覆盖, 判定覆盖, c-use覆盖, p-use覆盖	R平方	C	变异缺陷	块覆盖, 分支覆盖, c-use覆盖和p-use覆盖准则与测试有效性相关
Gupta等人[45]	2008	现有的测试集	块覆盖, 分支覆盖, 谓词覆盖	—	Java	变异缺陷	谓词覆盖 (条件覆盖)准则有效性最高但效率最低, 块覆盖准则效率最高但有效性最低
Wei等人[46]	2010	自动生成	分支覆盖	—	Eiffel	真实缺陷	分支覆盖并不是测试集有效性的良好指标
Hassan等人[47]	2013	现有的测试集	分支覆盖, def-use覆盖, multi-point stride coverage (MPSC)	皮尔逊相关, 调整后的R平方	C, C++, Java	变异缺陷	def-use覆盖与测试有效性强相关, 并且具有与分支覆盖几乎相同的预测能力. MPSC提供比分支覆盖更好的有效性预测
Gligoric等人[19,48]	2013, 2015	现有的测试集, 自动生成	语句覆盖, 分支覆盖, DBB, IMP, AIMP, PCT	肯德尔秩相关, R平方;肯德尔秩相关, 斯皮尔曼秩相关, R平方	C, Java	变异缺陷	覆盖率和测试有效性之间存在相关性. 分支覆盖是预测测试集质量的最佳度量
Hong等人[49]	2013	随机生成	blocking, blocked, blocked-pair, follows, sync-pair, LR-Def, PSet, def-use	皮尔逊相关, 调整后的R平方	Java	变异缺陷, 真实缺陷	现有的并发覆盖指标通常是并发测试有效性的中等到强预测指标, 并且通常是测试集生成的合理目标
Gopinath等人[18]	2014	现有的测试集, 自动生成	语句覆盖, 分支覆盖, 块覆盖, 路径覆盖	R平方, 肯德尔秩相关	Java	变异缺陷	覆盖率和测试有效性之间存在相关性. 语句覆盖可以最好地预测测试集的质量
Gay等人[22]	2015	自动生成	判定覆盖, 条件覆盖, MC/DC覆盖, OMC/DC覆盖	曼-惠特尼-威尔科克森秩和检验	Lustre	变异缺陷	仅满足覆盖准则可能无法很好地表明发现缺陷的有效性, 相同规模的随机测试集通常可以发现相似甚至更多的缺陷. 生成满足OMC/DC准则的测试集比相同规模的随机测试集可以实现更高的缺陷检测
Hemmati[50]	2015	现有的测试集	语句覆盖, 分支覆盖, MC/DC覆盖, 循环覆盖, def-use覆盖	—	Java	真实缺陷	基本的数据流覆盖可以检测到控制流覆盖未检测到的许多缺陷. 未检测到的缺陷主要与规范有关
Cheng等人[51]	2016	自动生成	语句覆盖, 分支覆盖, 表达式覆盖, 断言计数,断言覆盖	肯德尔秩相关	Haskell	变异缺陷	大多数关于命令式程序的发现仍然适用于函数式程序. 在特定的覆盖准则上, 结果可能与命令式不同
Gay[20]	2017	自动生成	BC, DBC, LC, EC, MC, MNEC, OC, WMC	对于每个缺陷, 检测到缺陷的测试集与为该缺陷生成的测试集总数的比例	Java	真实缺陷	最有力的缺陷检测有效性指标是高水平的代码覆盖率, EvoSuite的分支覆盖准则是最有效的
Kochhar等人[53]	2017	现有的测试集	行覆盖, 分支覆盖	斯皮尔曼秩相关, 肯德尔秩相关, 负二项式回归	Java	真实缺陷	在项目级别, 项目的覆盖率除以其圈复杂度与缺陷数量呈中度负相关
Durelli等人[54]	2018	随机生成	边覆盖, 边对覆盖, 主路径覆盖	—	C	变异缺陷	主路径覆盖可以检测到更多的缺陷, 但成本更高

(1) 覆盖准则比较

1) 控制流覆盖准则比较

在 2014 年, Gopinath 等人[18]的研究结果表明, 在原始测试集和由 Randoop 工具生成的测试集中, 语句覆盖是测试集质量的最有效预测指标.

研究人员经常通过测量覆盖率来比较测试集. 覆盖准则 C 提供了一组测试要求并衡量给定测试集达到了多少覆盖率. 达到 100% 覆盖率的测试集称为充分的. 在许多现实情况下, 达到充分的测试集是不切实际的, 甚至是不可能的. 在 2015 年, Gligoric 等人[19]介绍了第一个评估测试集不充分的研究. 他们评估了大量的准则, 包括语句和分支覆盖等基本准则, 以及更强的准则, 包括基于程序路径、覆盖语句的等价类和谓词状态的准则. 这些准则是在一组带有手动编写和自动生成的测试集的 Java 和 C 程序上评估的. 基于实验, 有两个准则表现最好: 分支覆盖和过程内非循环路径覆盖.

在 2017 年, Gay[20]评估了 EvoSuite 框架的缺陷检测能力及其对来自 Defects4J 数据库的 353 个真实缺陷的 8 个适应度函数. 分析发现, 最有力的缺陷检测有效性指标是高水平的代码覆盖率, EvoSuite 的分支覆盖准则是最有效的. 在 2018 年, Durelli 等人[54]使用 39 个 C 程序中的 189 个函数比较了 3 种覆盖准则: 边覆盖 (edge coverage)、边对覆盖 (edge-pair coverage) 和主路径覆盖 (prime path coverage), 使用变异体作为缺陷的代理. 结果表明主路径覆盖可以检测到更多的缺陷, 特别是在具有复杂控制流的程序中, 但成本更高.

2) 控制流和数据流覆盖准则比较

Frankl 等人[38,39]分别于 1991 年和 1993 年使用 9 个项目进行了比较 all-uses 和 all-edges 测试充分性准则的有效性的实验. 他们的分析表明, 对于 5 个项目, all-uses 明显比 all-edges 更有效. 在 4 个项目中, all-uses 充分的测试集比类似规模的 all-edges 充分的测试集更有效. 在 1991 年, 他们使用逻辑回归表明, 在一些项目中, 测试集覆盖的定义-使用关联的百分比与其错误检测能力之间存在很强的正相关. 进一步地, 在 1993 年, 他们使用逻辑回归分析调查测试集检测错误的概率是否随着覆盖的定义-使用关联的百分比或覆盖的边的百分比的增加而增加. 结果表明只有 4 个项目显示错误检测能力与覆盖的定义-使用关联的百分比呈强正相关; 错误检测能力与 4 个项目覆盖的边的百分比呈正相关, 但关系较弱.

在 1994 年, Hutchins 等人[40]的研究调查了 all-edges 和 all-DUs (修改的 all-uses) 两种覆盖准则. 结果表明, 覆盖率超过 90% 的测试集通常比随机选择的相同规模的测试集表现出更好的缺陷检测有效性, 此外, 当覆盖率从 90% 增加到 100% 时, 通常会显著提高基于覆盖率的测试的有效性. 他们的结果还表明, 仅 100% 的代码覆盖率并不是测试集有效性的可靠指标. 他们还发现, 分别基于控制流和数据流准则的测试在有效性上经常是互补的.

在 2015 年, Hemmati[50]研究了 Defects4J 数据集中 5 个开源 Java 项目的测试用例, 比较了几种现有的控制流和数据流覆盖准则的有效性. 他们发现: 1) 语句覆盖等基本准则非常弱 (仅检测到 10% 的缺陷); 2) 将多个控制流覆盖准则组合在一起优于单独使用最强的控制流覆盖准则 (MC/DC 检测到 19% 的缺陷, 所有控制流覆盖准则检测到 28% 的缺陷); 3) 基本的数据流覆盖可以检测到控制流覆盖未检测到的许多缺陷 (79% 未检测到的缺陷可以通过 def-use 覆盖检测到); 4) 平均 15% 的缺陷无法由控制流和数据流覆盖准则检测到, 未检测到的缺陷主要与规范有关.

(2) 覆盖率和测试集有效性的相关性

尽管覆盖准则比较结果不一, 代码覆盖率已成为预测测试集有效性的重要指标. 然而有的研究表明覆盖率在衡量测试的缺陷检测能力方面并不有效. 在 2000 年, Briand 等人[21]的研究结果不支持测试覆盖率和缺陷覆盖率之间存在因果关系的假设.

在 2005 年, Cai 等人[43]发现缺陷发现能力与 4 种不同的结构化覆盖准则 (块覆盖、判定覆盖、c-use 覆盖和 p-use 覆盖) 之间存在中度的相关性. 他们还发现代码覆盖率和缺陷覆盖率之间的相关性在异常测试中较强.

在 2013 年, Hong 等人[49]使用 9 个并发程序评估了 8 个并发覆盖准则和缺陷检测有效性之间的关系. 他们的结果表明, 现有的并发覆盖指标通常是并发测试有效性的中等到强预测指标, 并且通常是测试集生成的合理目标. 然而, 并发覆盖指标作为预测器和测试生成目标的有效性因程序而异.

研究人员已经提出了一些结构化覆盖准则来衡量测试工作的充分性. 在航空电子设备和其他关键系统领域, 满足结构化覆盖准则的测试集是由标准规定的. 随着强大的自动化测试生成工具的出现, 很容易生成测试输入来满足这些结构化覆盖准则. 然而, 产生的测试用例在缺陷检测能力方面的有效性尚未得到充分研究. 在 2015 年, Gay 等人[22]评估生成的测试集的有效性 (这项工作是先前工作[52]的扩展), 他们得出 3 个结论: 1) 仅满足覆盖准则可能无法很好地表明发现缺陷的有效性, 相同规模的随机测试集通常可以发现相似甚至更多的缺陷. 2) 使用结构化覆盖准则作为测试用例生成的补充而不是目标, 随机测试集规模减少到满足覆盖准则的子集比专门生成满足覆盖准则的测试集检测到的缺陷多 13.5%. 3) Observable MC/DC (OMC/DC) 准则[102]可以部分解决传统结构化覆盖准则的缺陷, 生成满足 OMC/DC 准则的测试集比相同规模的随机测试集可以实现更高的缺陷检测. 他们的结果强调需要研究覆盖准则、测试生成方法、使用的测试预言和系统结构如何共同影响测试的有效性.

在 2016 年, Cheng 等人[51]对函数式程序的测试覆盖率进行了第 1 个实证研究. 他们发现: 1) 输入覆盖率与归一化测试有效性 (nomalized test effectiveness) 没有强相关性; 2) 输入覆盖率与原始测试有效性 (raw test effectiveness) 有很强的相关性; 3) 语句/分支覆盖率和表达式覆盖率 (expression coverage) 与测试有效性的相关性没有显著差异; 4) 断言计数 (count of assertions) 与测试有效性的相关性非常弱; 5) 检查覆盖率 (checked coverage) 与测试有效性有很强的相关性. 这些结果表明, 大多数关于命令式程序的发现仍然适用于函数式程序; 在特定的覆盖准则上, 结果可能与命令式不同, 需要对函数式程序进行进一步研究.

在 2017 年, Kochhar 等人[53]进行了分析覆盖率对开源软件发布后缺陷的影响的实证研究. 他们分析了 100 个大型开源 Java 项目, 并测量了这些项目附带的测试用例的代码覆盖率. 他们收集软件发布后记录在问题跟踪系统中的真实缺陷, 并在项目和文件级别分析代码覆盖率与发布后缺陷之间的相关性. 他们还分析代码行数和圈复杂度等指标对代码覆盖率和发布后缺陷之间相关性的影响. 他们的结果表明: 1) 在项目级别, 覆盖率/复杂性 (项目的覆盖率除以其圈复杂度) 与缺陷数量呈中度负相关; 2) 在文件级别, 根据负二项式回归 (negative binomial regression, NBR) 模型, 尽管影响很小, 发现缺陷的数量随着覆盖率值的增加而减少.

3.2 基于变异得分的评价

3.2.1 变异测试

变异分析依赖于两个假设[126]: (1) 熟练程序员假设 (competent programmer hypothesis, CPH) 和 (2) 耦合效应假设 (coupling effect).

假设 1 (熟练程序员假设). 熟练程序员假设表明由熟练程序员编写的程序版本接近于程序的最终正确版本.

假设 2 (耦合效应假设). 耦合效应假设声称能够捕获一阶变异体 (first order mutants, FOM) 的测试集也将检测包含这些一阶变异体的更高阶变异体 (higher order mutant, HOM).

研究人员基于熟练程序员假设变异分析产生的缺陷与真实缺陷相似. 变异测试将缺陷注入被测程序以创建变异体. 这些变异体可分为一阶变异体 (FOM) 或高阶变异体 (HOM). FOM 只注入一个缺陷, 而 HOM 至少注入一个缺陷. 耦合效应的证据来自 Wah[127]的理论分析和 Offutt[128]的实证研究. 然而, 在 2010 年, Papadakis 等人[129]阐明了针对一阶和二阶变异测试策略进行的实证研究的结果. 结果表明, 一方面, 一阶变异测试策略通常比二阶变异测试策略更有效; 另一方面, 二阶策略可以大大减少引入的等价变异体的数量.

变异分析中的一个问题是等价变异体 (equivalent mutants) 的普遍性. 这些是在语义上与原始程序相同的变异体, 在一般意义上, 它们的识别是不可判定的[130]. 识别等价变异体通常分为基于预防和基于检测两类. 基于预防的方法通过识别产生少量等价变异体的算子并使用它们减少等价变异体的发生率[131]. 基于检测的方法通过变异体的各种静态和动态特性检测等价变异体[5,132,133].

定义 12 (变异得分). 测试集的变异充分性根据以下公式计算的分数衡量:

$$MS(T,M)=\frac{KM(T,M)}{M-E},$$

其中, T 是测试集, M 是变异体的总数, E 是等价变异体的数量, $KM(T, M)$ 是测试集 T 杀死的 M 中变异体的数量.

如果不去除等价变异体, 则变异得分的计算公式为:

$$MS(T,M)=\frac{KM(T,M)}{M}.$$

无论是否去除等价变异体, 如果变异得分越高, 那么测试集有效性越高.

如果在执行变异语句后, 变异体的状态立即与原始程序的相应状态不同, 则称变异体被弱杀死 (weakly killed)[134]. 如果原始程序和变异体在他们的输出行为中表现出一些可观察到的差异, 则变异体被强杀死 (strongly killed)[134]. 弱变异只需弱杀死变异体, 强变异要求强杀死变异体. 强变异 (strong mutation) 不包含弱变异 (weak mutation)[135], 因为失败的错误传播 (failed error propagation, FEP) 可能导致状态差异被后续计算覆盖[134]. 在 1994 年, Offutt 等人[55]的研究表明, 为弱变异测试设计的测试数据具有与强变异测试几乎相同的缺陷检测能力, 同时节省 50% 的执行成本. 然而, 在 2017 年, Chekam 等人[59]的研究表明弱变异测试与强变异测试之间的缺陷检测能力存在很大差异.

3.2.2 覆盖准则和变异测试比较

较早对变异测试的研究分别从理论分析[56]和实证研究[57]表明它包含不同的控制流和数据流覆盖准则, 包括语句覆盖、分支覆盖、条件覆盖等控制流覆盖和 all-defs、all-uses 数据流覆盖. 表 3 列出了覆盖准则和变异测试的缺陷检测有效性比较的主要研究. 从表 3 中可以观察到, 变异测试比结构化覆盖准则具有更高的缺陷检测相关性.

表 3 覆盖准则和变异测试的有效性比较

作者	年份	充分性准则	统计方法	语言	缺陷类型	主要发现总结
Wong等人[58]	1995	all-uses, 变异测试	—	Fortran, C	变异缺陷	基于变异的两种变体之一的充分性准则 (abs/ror变异)比all-uses准则具有更好的缺陷检测有效性
Offutt等人[25]	1996	all-uses, 变异测试	—	Fortran, C	注入缺陷	all-uses和变异测试都是有效的, 变异充分的测试集检测到更多的缺陷
Frankl等人[26]	1997	all-uses, 变异测试	—	Fortran, Pascal	真实缺陷	对于all-uses和变异测试, 测试有效性在较高的覆盖率水平上提高. 变异测试表现更好
Baker等人[27]	2013	语句覆盖, 分支覆盖, MC/DC覆盖, 变异测试, 人工审查	—	C, Ada	变异缺陷	在传统的结构化覆盖分析和手动同行评审失败的情况下, 变异测试是有效的
Just等人[23]	2014	语句覆盖, 变异测试	点二序列相关, 秩二序列相关	Java	真实缺陷	变异得分和语句覆盖都与真实缺陷检测相关, 变异得分具有更高的相关性
Ahmed等人[28]	2016	语句覆盖, 变异测试	肯德尔秩相关, R平方	Java	真实缺陷	语句覆盖率和变异得分都与错误修复只有微弱的负相关
Chekam等人[59]	2017	语句覆盖, 分支覆盖, 弱变异, 强变异	威尔科克森检验, A_{12}	C	真实缺陷	缺陷检测与语句覆盖, 分支覆盖, 弱变异之间缺乏联系. 对于强变异, 在某个阈值之上, 增加的覆盖率和增加的缺陷检测之间存在很强的联系, 但是, 低于此阈值水平, 测试集实现的覆盖率与其缺陷检测潜力完全无关

在 1995 年, Wong 等人[58]比较了变异测试、两种变异变体和 all-uses 的缺陷检测有效性. 一种变异变体是通过将变异算子限制为 abs 和 ror 获得的, 另一种变异变体是通过从每种变异体类型中随机选择 10% 的变异体获得的. 数据表明, 按照缺陷检测有效性的降序排列, 依次为变异、abs/ror 变异、all-uses 和 10% 变异.

在 1996 年, Offutt 等人[25]的结果表明, all-uses 和变异测试都是有效的, 变异充分的测试集检测到更多的缺陷. 1997 年, Frankl 等人[26]的结果表明, 对于 all-uses 和变异测试, 测试有效性在较高的覆盖率水平上提高. 变异测试表现更好. 他们还探索了固定规模的测试集的覆盖率 (all-uses 覆盖率和变异覆盖率) 和有效性之间的关系, 发现它是非线性的, 在许多情况下是非单调的.

在 2013 年, Baker 等人[27]对变异测试在已经满足认证覆盖率要求的机载软件系统中的应用进行了实证评估. 他们将变异测试应用于使用 C 和 Ada 的高完整性子集开发的安全关键软件, 识别最有效的变异类型, 并分析测试用例失败的根本原因. 结果表明, 在传统的结构化覆盖分析和手动同行评审失败的情况下, 变异测试是有效的. 他们还表明, 一些测试问题的根源超出了测试活动, 需要改进需求定义和编码过程.

在 2014 年, Just 等人[23]的结果表明变异体检测与真实的缺陷检测正相关, 这种相关性强于语句覆盖率和真实缺陷检测之间的相关性.

在 2016 年, Ahmed 等人[28]提出了对语句覆盖率和变异得分的评估. 他们评估的准则是: 如果程序元素在给定时间点经过良好的测试, 那么与测试不足的元素相比, 经过良好测试的程序应该需要更少的未来的错误修复. 他们使用大量具有代表性的来自 GitHub 和 Apache 的大量开源 Java 程序、真实世界测试集和错误修复进行了评估. 结果表明语句覆盖率和变异得分都与错误修复只有微弱的负相关.

在 2017 年, Chekam 等人[59]的发现是缺陷检测与语句覆盖、分支覆盖、弱变异之间缺乏联系. 强变异与缺陷检测之间的关系表现出一种“阈值行为”形式, 在某个阈值之上, 增加的覆盖率和增加的缺陷检测之间存在很强的联系, 但是, 低于此阈值水平, 测试集实现的覆盖率与其缺陷检测潜力完全无关. 根据他们的结果, 任何试图比较未能达到阈值覆盖率的不充分测试集的尝试都可能受到“噪声效应”的影响: 即使实验者遵循相同的实验程序, 两项低于阈值覆盖率的研究也可能产生不同的结果.

3.2.3 变异得分和测试集有效性的相关性

表 4 列出了变异得分与缺陷检测能力相关性的主要研究.

表 4 变异得分与缺陷检测能力相关性

作者	年份	充分性准则	统计方法	语言	主要发现总结	
Just等人[23]	2014	语句覆盖, 变异测试	点二序列相关, 秩二序列相关	Java	变异得分和语句覆盖都与真实缺陷检测相关, 变异得分具有更高的相关性	变异得分与缺陷检测能力相关性强
Kim等人[24]	2020	变异测试	伪R平方, 点二序列相关	Java	即使在控制规模的情况下, 在细粒度级别上变异得分与缺陷检测能力之间的相关性也很强	
Perretta等人[60]	2022	变异测试	皮尔逊相关	JavaScript, TypeScript, C++	变异得分与手动注入的缺陷检测率之间存在很强的相关性, 变异得分与学生错误的实现检测之间存在中等强度的相关性	
Ahmed等人[28]	2016	语句覆盖, 变异测试	肯德尔秩相关, R平方	Java	语句覆盖率和变异得分都与错误修复只有微弱的负相关	变异得分与缺陷检测能力相关性弱
Papadakis等人[29]	2018	变异测试	伪R平方, 肯德尔秩相关, 皮尔逊相关	C, Java	变异得分与真实缺陷检测之间存在弱相关性. 变异得分有助于测试人员(通过达到相对较高的变异得分)改善测试集. 缺陷检测和变异得分之间的关系是非线性关系	
Hariri等人[30]	2019	变异测试	皮尔逊相关, 肯德尔秩相关	C	SRC和IR这两个级别的变异得分与测试集的实际缺陷检测能力不是很相关	

(1) 变异得分与缺陷检测能力相关性强

在 2014 年, Just 等人[23]使用来自 5 个开源应用程序的 357 个真实缺陷以及开发人员编写和自动生成的测试集, 表明检测变异体的能力与检测真实缺陷显著相关.

在 2020 年, Kim 等人[24]使用 Defects4J 数据集研究了变异体、真实缺陷和测试集规模之间的关系. 他们在类、方法和语句级别定位代码, 生成变异体. 他们发现: ① 粒度 (类、方法、语句) 级别导致了关系的显著差异, 变异体的检测率与缺陷检测能力之间的相关性在细粒度级别上比在粗粒度级别上更强; ② 测试集规模在每个粒度级别对关系的影响不同, 即使在控制规模的情况下, 在细粒度 (方法、语句) 级别上变异得分和缺陷检测能力之间的相关性也很强. 这些发现表明, 当精确定位容易出错的代码时, 变异体的检测率与真实缺陷之间存在很强的相关性, 而与测试集的规模无关.

在 2022 年, Perretta 等人[60]使用包含 2 711 份学生作业提交的数据集, 评估变异得分是否可以很好地代表手动注入的缺陷检测率和学生错误的实现检测率. 结果表明, 变异得分与手动注入的缺陷检测率之间存在很强的相关性, 变异得分和学生错误的实现检测率之间存在中等强度的相关性.

(2) 变异得分与缺陷检测能力相关性弱

在 2016 年, Ahmed 等人[28]表明语句覆盖率和变异得分都与错误修复只有微弱的负相关. 在 2018 年, Papadakis 等人[29]的结果表明, 变异得分有助于测试人员 (通过达到相对较高的变异得分) 改善测试集, 但变异体与缺陷检测的相关性很弱, 使用变异体代替真正的缺陷可能会出现问题. 缺陷检测和变异得分之间的关系是非线性关系.

有多种变异工具可以在不同级别执行变异, 包括源代码 (source code, SRC) 级别的变异测试和编译器中间表示 (intermediate representation, IR) 级别的变异测试. 在 2019 年, Hariri 等人[30]对 SRC 和 IR 级别的变异测试进行了广泛的比较, 具体是在 C 编程语言和 LLVM 编译器 IR 级别. 他们发现 SRC 级别的变异测试更好, SRC 级别产生的变异体少得多, 但两个级别的变异得分都是密切相关. 他们还发现这两个级别的变异得分与测试集的实际缺陷检测能力不是很相关.

3.2.4 基于变异得分的评价的问题

变异测试是一种评估测试集有效性的方法, 其结果被认为比代码覆盖率指标更有意义. 然而变异测试需要大量的计算工作. 在实践中更广泛地采用变异分析的一个障碍是其高计算成本. 对大型程序执行变异分析需要分析更多的变异体, 需要更多次运行测试集. 变异测试成本高的主要原因之一是产生大量的变异体. 现代软件可能有一百万行或更多行代码, 当所有变异体的数量是这样数量级的倍数时, 评估所有变异体是不切实际的. 需要降低变异测试的代价. 已有的许多研究使用更轻量级的方法近似变异测试有效性, 这类研究[136−141]主要包括两类: 预测性变异测试和变异约简.

(1) 预测性变异测试

在 2018 年, Zhang 等人[142]提出了预测性变异测试 (predictive mutation testing, PMT), 这是第 1 种无需执行变异体即可预测变异测试结果的方法. PMT 构建了一个基于执行特征、感染特征和传播特征的总共 14 个特征的分类模型, 并使用该模型预测在不执行变异体的情况下一个变异体是否会被杀死或仍然存活.

在 2020 年, Zhang 等人[143]提出了一种变异测试优化技术, 即覆盖信息驱动的无监督预测性变异测试, 以期实现兼顾效率和效果的变异测试优化. 该方法根据一个变异体在执行过程中被多少测试用例覆盖, 利用概率论的方法估算它存活下来的概率. 在为每一个变异体估算存活概率后, 为所有变异体计算一个存活数目的期望. 进而计算一个预测的变异得分以用于衡量测试用例集的有效性.

(2) 变异约简

Offutt 等人[144]将变异约简的技术归类为 3 类: 1) “do faster”; 2) “do smarter”; 3) “do fewer”. “do faster”方法专注于尽可能快地生成和执行变异体, 例如, 运行编译程序而不是解释程序, 或者使用编译器相关的优化[145]. “do smarter”方法尝试将变异体的执行分布在多台机器上, 或者避免它们的完整执行, 例如, 在变异语句执行后停止变异体的执行[135].

“do fewer”策略尝试生成或执行更少的变异体, 从而减少此类任务所需的计算时间. “do fewer”一般分为选择法 (selective approaches) 和采样法 (sampling approaches). 选择法[138]通过仅使用变异算子的子集生成变异体, 即选择性变异 (selective mutation, SM). 采样法[140]试图随机选择一组有代表性的变异体, 然后用来近似整体的变异得分.

此外, 高阶变异体[146]、变异聚类[147]、变异体包含[148,149]也是降低变异分析成本的方法. 研究人员已经提出了多种降低成本的方法, 在 2019 年, Pizzoleto 等人[150]通过系统的文献回顾总结和分析了当前有关降低变异测试成本的技术和指标.

3.3 面向领域软件的测试集有效性评价

在面向领域的软件的测试中, 一些传统软件的测试方法不再适用, 为此, 研究人员提出了多种面向领域软件的测试充分性准则并评估了测试集有效性. 例如, 针对 Web 应用测试, 在 2012 年, Alshahwan 等人[61]介绍了一种基

于输出唯一性的新型测试集充分性准则. 他们提出了 4 种不同严格程度的输出唯一性定义 (OU-All unique、OU-Struct unique、OU-Seq unique 和 OU-Text unique). 他们对 Web 应用程序测试进行了初步评估, 确认输出唯一性提高了发现缺陷的有效性.

本节回顾 Web 服务、物联网系统、人工智能领域软件和模型驱动测试方法的测试集有效性评价的研究.

3.3.1 Web 服务

Web 服务建立在 Web 协议之上并基于开放的 XML 标准. Web 服务使用 Web 服务描述语言 (Web services description language, WSDL) 进行描述. 面向服务的架构 (service oriented architecture, SOA) 定义了一种通过服务接口使软件组件可重用和互操作的方法.

WS-BPEL (Web services business process execution language) 应用程序是一种面向服务的应用程序. 在这些应用程序中, 各个松耦合的工作流步骤通过基于 XML 的消息交换连接起来, XPath 是查询 XML 文档的方法. 针对未能从 XML 消息中提取正确的数据片段将在此类应用程序中造成的集成错误, 在 2008 年, Mei 等人[62]提出了一种数据结构, 即 XPath 重写图 (XPath rewriting graph, XRG), 以在 WS-BPEL 中对 XPath 进行建模. 他们提出了一系列新的数据流测试充分性准则测试 WS-BPFL 应用程序, 这些准则包括 all-queries、all-query-pu 和 all-query-du. 实验结果表明, 与提出的 3 种准则相比, 随机测试的有效性最差; all-query-du 准则表现出最好的平均缺陷检测能力. 在 2009 年, Mei 等人[63]提出扩展标记迁移系统 (labelled transition system, LTS) 以对编排应用程序 (choreography application) 中服务之间的交互进行建模, 通过合并 XPath 重写图 (XRGs) 扩展 LTS. 他们开发了 XRG 模式的概念捕获不同 XRG 之间的关系. 基于他们的模型和 XRG 模式, 他们提出了一组新的数据流测试准则, 即 all-chore-queries、all-chore-query-uses 和 all-chore-query-pattern-uses. 实验结果表明他们的方法比随机测试更有效的检测缺陷.

在 2005 年, Siblini 等人[64]提出了使用变异分析测试 Web 服务的技术. 他们定义了 9 个 WSDL 文档的变异算子, 并将变异算子应用于 WSDL 文档以生成用于测试 Web 服务的变异 Web 服务接口. 结果表明了该技术的有用性. 在 2009 年, Dominguez-Jimenez 等人[65]介绍了 GAmera, 一种使用遗传算法 (genetic algorithm) 为 WS-BPFL 组合自动生成变异体的系统. 他们使用 26 个变异算子生成变异体. 结果表明, GAmera 可以通过检测潜在的等价变异体帮助改进现有的测试用例集.

在 2009 年, Mei 等人[66]报告了将 Frankl-Weyuker 数据流测试准则应用于面向服务的程序的实证研究. 他们研究了 all-uses 准则, 使用了 8 个开源 WS-BPEL 程序. 实验结果表明, all-uses 准则在检测 BPEL (business process execution language) 代码缺陷方面比检测 WSDL 或 XPath 制品中的缺陷更有效.

3.3.2 物联网系统

物联网 (Internet of things, IoT) 依赖于硬件、软件和架构的组合, 其中日常对象可以配备识别、传感、联网和处理能力, 它们能够通过互联网相互通信以实现某些目标[151]. 在 2022 年, Zhu 等人[152]系统地总结了不同物联网领域的测试方法, 并讨论了现有物联网测试平台的发展现状.

在 2008 年, Lai 等人[67]提出了一个框架测试 nesC 应用程序. 他们引入了上下文间流图 (inter-context flow graphs, ICFG) 表达 nesC 应用程序的行为. 他们进一步提出基于 ICFG 的控制流和数据流测试充分性准则, 即 all-inter-context-edges 准则和 all-inter-context-uses 准则. 他们使用开源的结构健康监测应用程序评估提出的测试准则. 实验结果表明, 平均而言, all-inter-context-uses 具有最高的缺陷检测率, 其次是 all-inter-context-edges; all-inter-context-edges 比 all-branches 的有效性高 21%; all-inter-context-uses 比 all-uses 的有效性高 69%.

在 2018 年, Leotta 等人[68]提出了一种对配备用户界面 (user interface, UI) 的物联网系统进行验收测试的方法. 他们根据状态机对系统行为进行形式化用于指导测试活动, 并应用临时路径覆盖准则, 将循环数限制为一个. 他们在由本地传感器和执行器、基于云的医疗保健系统和 Android 应用程序组成的案例研究中, 使用变异测试对生成的测试集的缺陷检测有效性进行评估. 结果显示了方法的有效性, 可以检测到 93% 的变异体.

3.3.3 人工智能领域软件

(1) 机器学习测试充分性准则

传统软件与基于机器学习的软件之间存在区别. 机器学习模型的决策逻辑不是手动编写的, 而是从训练数据

中学习的[153].

在 2017 年, Pei 等人[69]设计并实现了第一个系统地测试深度学习系统的白盒框架 DeepXplore. 他们引入神经元覆盖 (neuron coverage, NC) 作为深度学习系统的第 1 个白盒测试指标. 他们将神经元覆盖定义为所有测试输入的唯一激活神经元数量与深度神经网络中神经元总数的比率, 神经元被激活即神经元的输出高于阈值. DeepXplore 能够在每个测试的深度神经网络中发现许多错误行为. 随后, 其他的测试深度神经网络的覆盖准则被提出, 在 2020 年, 王赞等人[154]总结了测试覆盖准则.

在 2018 年, Ma 等人[70]提出了深度学习系统的第 1 个变异测试技术 DeepMutation. 他们设计了 8 个直接操作训练数据和训练程序的变异算子, 并进一步设计了 8 个变异算子直接变异深度学习模型. 他们在两个数据集以及 3 个深度学习模型进行了评估, 评估结果表明了提出的变异测试技术的有用性. DeepMutation++[71]合并了 DeepMutation 中引入的用于前馈神经网络模型的 8 个模型级变异算子, 并进一步提出了 9 个用于循环神经网络模型的变异算子.

(2) 深度神经网络覆盖准则与缺陷检测能力

在 2019 年, Li 等人[72]认为神经网络和人类编写的程序之间存在根本差异, 神经网络结构覆盖准则可能会产生误导. 他们表明, 高覆盖率的缺陷检测能力更有可能是由于面向对抗样本的搜索.

在 2020 年, Yan 等人[73]研究了深度神经网络模型覆盖准则与模型质量之间的相关性. 他们使用了 8 个模型和 3 个数据集, 发现深度神经网络覆盖准则与模型质量没有单调关系. 他们发现尽管深度神经网络覆盖准则可用于寻找对抗样本, 但有效的对抗样本可能不会导致更高的覆盖率. 他们还发现大多数现有的深度神经网络覆盖准则彼此相关, 有些具有很强的相关性. 在 2022 年, Yang 等人[74]对 Yan 等人[73]的工作进行了复制研究. 他们的实验结果证实了 Yan 等人的结论. 他们发现模型的质量指标不会随着覆盖率指标的增加而提高, 覆盖率提高样本在改进模型质量方面并没有更有效, 并且基于梯度的方法在发现深度神经网络缺陷方面比覆盖驱动的方法更有效.

在 2020 年, Harel-Canada 等人[75]设计了正则化器自动生成增加神经元覆盖率的测试集, 并评估生成的测试集的缺陷检测能力, 即检测对抗攻击的能力. 他们发现增加覆盖率会更难生成有效的测试集, 更高的神经元覆盖率会导致检测到的缺陷更少. 他们还发现生成的增加神经元覆盖率的测试集的自然输入更少以及模型预测偏向特定类别标签的程度更偏.

在 2020 年, Dong 等人[76]进行了评估深度神经网络的覆盖率、鲁棒性和相关指标之间相关性的实证研究. 他们发现不同的覆盖准则彼此相关, 覆盖准则和鲁棒性指标之间的相关性有限. 他们还发现使用提高覆盖率的新测试用例进行再训练不一定提高模型的鲁棒性.

在 2021 年, Sun 等人[77]提出了独立神经元覆盖 (independence neuron coverage, INC), 与之前的准则相比, INC 的粒度更细. 他们评估了 NC、2-way 覆盖、3-way 覆盖和 INC 这 4 种覆盖准则. 结果表明: 1) 评估的 4 种覆盖准则中的 3 种可以区分不同质量的测试集, 并且更细粒度的覆盖准则表现更好; 2) 数据增强和覆盖准则的结合可以提高深度神经网络的鲁棒性; 3) 覆盖准则的时间成本最好控制在 $O(n^3)$ 以下.

3.3.4 模型驱动测试方法

基于模型的测试 (model-based testing, MBT) 使用模型自动化测试过程, 模型中的信息即系统及其环境的预期行为[155]. 在 2011 年, Utting 等人[156]从 6 个维度, 即模型范围、模型特征、模型范式、测试选择准则、测试生成技术和测试执行, 对不同的 MBT 方法进行分类. 基于模型的测试中常用的模型有多种, 本文关注 UML (unified modeling language) 模型. UML 模型通常用于可视化和理解系统的结构和行为.

UML 类图和交互图. 在 2003 年, Andrews 等人[78]提出了基于 UML 模型元素的测试充分性准则, 该准则可用于定义 UML 设计的测试目标. 他们提出的测试充分性准则基于 UML 类图和交互图. 类图准则用于确定运行测试的对象配置, 包括 AEM (association-end multiplicity) 准则、GN (generalization) 准则和 CA (class attribute) 准则; 交互图准则用于确定应测试的消息序列, 包括 Cond (condition coverage) 准则、FP (full predicate coverage) 准则、EML (each message on link) 准则、AMP (all message paths) 准则和 Coll (collection coverage) 准则. 他们列举了每个

准则检测到的缺陷类型以及结合使用配置和行为相关准则可以发现的缺陷类型.

UML 序列图. 在 2005 年, Rountev 等人[79]定义了几个控制流覆盖准则 (All-RCFG-Paths、All-RCFG-Branches 和 All-Unique-Branches), 用于测试一组协作对象的交互. 这些准则基于从被测代码逆向工程得到的 UML 序列图. 图中的消息序列用于定义准则的覆盖目标. 他们描述了收集每个准则的覆盖率的运行时分析. 他们对不同的准则进行了比较, 研究表明, 要求较低的准则 (基于分支覆盖) 可能是测试对象交互的更实用的选择.

UML 协作图和状态图. 在 2007 年, Ali 等人[80]提出了考虑交互中协作类的所有可能状态增强类的集成测试的技术. 该技术结合了 UML 协作图和状态图, 自动生成中间测试模型 SCOTEM (state collaboration test model). 基于 SCOTEM, 他们定义了多种覆盖准则以从 SCOTEM 模型生成测试路径. 他们开发了工具进行了案例研究, 并使用变异算子生成了 49 个被测系统的缺陷版本. 结果表明, 所提出的方法有效地检测到各种集成缺陷, All-Path 覆盖准则成功检测到所有变异缺陷.

UML 状态图. 在 1999 年, Offutt 等人[81]介绍了用于生成 UML statecharts 测试输入的准则, 包括 Transition Coverage、Full Predicate Coverage、Transition-pair Coverage 和 Complete Sequence. 这些准则允许软件系统级测试从 UML statechart 图中得出. 他们开发了工具可以自动从 UML 规范中生成测试. 实验结果表明可以自动生成高效的测试. 在 2011 年, Zander 等人[157]总结了状态机模型的典型覆盖准则. 在 2004 年, Briand 等人[82]提出了一个精确的仿真和分析流程, 并使用此流程在 3 种不同代表性案例研究上, 研究了最多引用的 statechart 覆盖准则的成本和缺陷检测有效性. 研究的 4 种准则为: AT (all transitions)、ATP (all transition pairs)、TT (all paths in the statechart transition tree) 和 FP (full predicate). 从案例研究的结果中, 他们得出的结论有: 在大多数实际情况下, AT 覆盖不足以确保充分的缺陷检测水平; ATP 几乎检测到所有缺陷, 但成本增加; 在同样的情况下, FP 比 ATP 成本更高, 但有效.

UML 活动图. 在 2009 年, Samuel 等人[83]提出了基于 UML 活动图动态切片的测试用例生成方法. 他们为活动图定义了路径覆盖准则、全谓词覆盖和边界测试准则. 他们使用活动图的流依赖图 (flow dependence graph, FDG) 生成动态切片, 并且针对每个切片自动生成测试用例. 他们的方法自动生成测试数据以实现高路径覆盖. 研究人员已经提出了多种活动图的覆盖准则, 在 2019 年, Ahmad 等人[158]回顾了不同研究中使用的测试覆盖准则, 并将这些覆盖准则分类为与图 (graph)、逻辑 (logic) 和数据 (data) 相关的覆盖准则. 此外, 在 2016 年, Sun 等人[84]提出了一种基于变异活动图的测试生成方法. 他们根据活动图的语法定义, 为活动图定义了 5 类变异算子. 测试用例是通过求解变异的路径约束生成的. 实验结果表明, 该方法可以生成具有更高缺陷检测能力的测试用例.

已经提出了多种源自 UML 模型的覆盖准则. 在 2005 年, McQuillan 等人[159]回顾并分析了存在的各种基于 UML 的覆盖准则, 包括类图准则、顺序图准则、通信图准则、状态机图准则、活动图准则和用例图准则.

3.4 小 结

本节主要介绍了基于覆盖率和基于变异得分的评价. 本节进一步介绍了面向领域软件的测试充分性准则与测试集有效性评价. 现有的研究包含许多相互矛盾的结论, 下面简述主要发现.

(1) 覆盖准则的有效性结论不一致: 有的研究表明语句覆盖准则最有效[18]; 有的研究表明分支覆盖准则最有效[19,20,48]; 还有的研究表明 MC/DC 最有效[22].

(2) 达到较高的代码覆盖率的测试集的缺陷检测能力较高[20,40,41].

(3) 数据流覆盖准则比控制流覆盖准则更有效[38,39].

(4) 一些研究表明覆盖率与缺陷检测有效性相关性强[18,49,51], 而另一些研究表明覆盖率与缺陷检测有效性相关性弱[21,22].

(5) 变异测试比结构化覆盖准则具有更高的缺陷检测相关性, 但是变异分析成本较高. 降低变异测试代价的研究主要包括两类: 预测性变异测试和变异约简.

(6) 一些研究表明变异得分与缺陷检测能力之间的相关性强[23], 而另一些研究表明变异得分与缺陷检测能力相关性弱或不是很相关[28−30].

(7) 面向领域软件的测试充分性准则基于基本的代码覆盖准则提出[62,67,69,79]. 多项研究表明, 神经元覆盖准则与缺陷检测能力既不正相关也不强相关[73−77].

4 测试集有效性的影响因素

一些研究表明, 代码覆盖率与缺陷检测有效性相关性强[18,49,51], 而其他研究[21,22]则相反; 另一些研究表明, 变异得分和缺陷检测能力之间的相关性强[23], 而其他研究[28−30]则相反. 这些不一致的结果表明, 有一些未知的因素会影响满足测试充分性准则的测试集发现缺陷的能力. 为了提高测试集发现缺陷的能力, 需要了解导致这种差异的因素. 这些影响因素如图 8 所示.

图 8 测试集有效性的影响因素

4.1 混杂效应

在数据分析过程中, 除了研究的自变量外, 可能还有其他研究人员无法控制的混杂因素 (confounding factors). 这些变量是混杂变量 (confounding variables), 可能与自变量和/或因变量相关.

如图 9 所示, 自变量 X 和因变量 Y 之间的关系受到其他变量的影响. 因此, 这些变量对研究结果构成威胁, 因为它们可能导致自变量 X 对因变量 Y 的因果效应. 例如, 研究人员如果想要研究面向对象指标与特定类是否出错的概率之间的关系, 那么应该考虑类规模对关联的混杂影响[160,161]. 类规模与缺陷概率相关, 因为较大的类由于其规模而往往具有更多的缺陷.

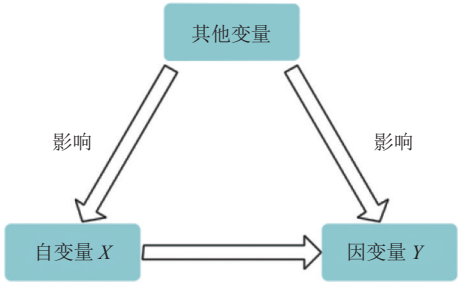


图 9 混杂效应

混杂变量可能会影响实验结果的准确性, 实验需要控制混杂变量. 例如, 在构建缺陷预测模型时, 应该消除类规模的混杂效应[162]. 为了实现有效的缺陷定位, 应减轻程序依赖的混杂效应[163].

4.2 测试用例质量

测试用例本身的质量已被证明对测试有效性有影响[164]. 测试用例遵循良好的设计原则, 这样测试用例更容易理解、维护和用于诊断生产代码中的问题.

测试越来越多地被表示为程序代码[165]. 然而, 以代码表示的测试也可能包含缺陷. 一些测试代码中的缺陷类似于在应用程序代码中发现的缺陷, 而另一些则更微妙, 是由于测试概念和过程的不正确实现造成的. 这些测试代码中的缺陷可能会导致测试在不应该失败的情况下失败, 或者让程序缺陷未被检测到[166]. 与测试代码中的缺陷相关的测试代码模式 (test code patterns) 可以使用轻量级静态分析检测[166]. 与设计模式类似, 单元测试模式促进了单元测试的可维护性质量属性, 例如易于诊断、可修改性和可理解性[167].

4.2.1 测试气味

实现易于维护的测试是测试用例设计中的一个重要目标. 类似于代码坏味 (bad smells in code)[168], 测试气味 (test smells) 表示设计不佳的测试. 在 2001 年, Deursen 等人[169]引入了测试气味的概念, 并定义了 11 种测试气味以及删除测试气味的重构操作. 在 2007 年, Meszaros[170]扩展并定义了 18 种测试气味. 测试气味可能会影响单元测试的有效性和可维护性.

测试气味的例子有: 1) 条件测试逻辑 (conditional test logic): 在测试中包含控制流语句的测试, 测试可以有多个分支点; 2) 饥饿测试 (eager test): 试图验证太多功能的测试, 这可能导致测试代码难以理解; 3) 空测试 (empty test): 测试方法不包含可执行语句; 4) 睡眠测试 (sleepy test): 包含线程挂起调用的测试, 使用此方法调用会给测试执行带来额外的延迟.

许多工作表明了测试气味的普遍性和影响[171,172]. 在 2014 年, Bavota 等人[173]研究了 27 个软件系统中测试气味的扩散及其对程序理解和维护的影响. 他们发现测试气味的扩散相当高, 在分析的 987 个 JUnit 测试中, 86% 的测试至少表现出一种测试气味, 而 49% 的测试至少包含两种测试气味. 他们表明在维护活动期间测试气味对程序理解产生负面影响.

相关文献提供了大量度量检测设计不好的测试, 例如测试气味检测的规则[174,175]. 在 2020 年, Spadini 等人[176]根据测试气味的严重程度研究了测试气味的分类. 他们给测试气味一个严重等级, 从低到非常高. 对于 4 种测试气味 (assertion roulette、eager test、verbose test 和 conditional test logic), 他们定义了严重性阈值.

测试气味与软件质量关联. 在 2018 年, Spadini 等人[85]研究了测试气味的存在与软件变更倾向 (change-proneness) 和缺陷倾向 (fault-proneness) 之间的关系. 他们的主要发现是受测试气味影响的测试比不受测试气味影响的测试与更高的软件变更倾向和缺陷倾向关联.

测试气味影响测试用例有效性. 在 2019 年, Grano 等人[86]研究了 67 个与生产代码和测试代码相关的因素与测试用例有效性之间的关系. 他们考虑了 5 个维度的 67 个因素, 即动态因素: 代码覆盖率 (语句覆盖率) 和静态因素: 测试气味、生产和测试代码度量、代码气味和可读性. 实证研究结果表明: 1) 对于 41/67 的因素, 有效测试与无效测试在统计上有所不同. 当测试用例具有高语句覆盖率并且不包含测试气味时, 测试用例往往会更有效. 如果生产代码规模大并且复杂度高, 那么测试用例会无法正确地检测错误. 代码气味越少, 测试检测错误的能力就越高. 2) 可以有效地利用估计模型对测试用例的有效性进行分类. 同时依赖动态和静态信息的模型在 F-measure 和 AUC-ROC 方面的性能接近 95%, 而仅使用静态信息的模型性能下降约 9%.

4.3 规模等其他因素与测试集有效性

4.3.1 测试集规模

测试集 T 的规模, 即测试集的基数$|T|$, 是测试集中测试用例的数量. 覆盖率的提高会导致规模增加, 而增加的规模是提高有效性的直接原因. 因此, 现有研究探讨了测试集规模与缺陷检测有效性的关系. 测试集有效性评价的许多研究认为测试集规模是需要控制的混杂因素. 表 5 列出了测试集规模与缺陷检测有效性的主要研究.

(1) 覆盖率

在 1994 年, Wong 等人[87]进行了实证研究, 比较了 (1) 缺陷检测有效性和块覆盖率之间的统计相关性; (2) 缺陷检测有效性和测试集规模之间的统计相关性. 实验结果表明, 有效性和块覆盖率之间的相关性高于有效性和规模之间的相关性. 在 2006 年, Andrews 等人[44]的实验表明, 尽管测试集规模和覆盖率 (块覆盖、判定覆盖、c-use 覆盖和 p-use 覆盖) 具有很强的相关性, 规模和覆盖率在解释缺陷检测方面发挥着互补作用. 在 2009 年, Namin 等

人[88]的实验表明, 当控制规模时, 覆盖率有时与有效性相关, 并且使用规模和覆盖率可以比单独使用规模更准确地预测有效性, 即规模和覆盖率对于测试集的有效性都很重要. 他们的实验还表明, 测试集的规模、覆盖率和有效性这三个变量之间不存在线性关系, 但存在非线性关系.

表 5 测试集规模与缺陷检测有效性

作者	年份	测试用例来源	测试集构造	充分性准则	统计方法	语言	缺陷类型	主要发现总结
Wong等人[87]	1994	自动生成	测试选择	块覆盖	斯皮尔曼秩相关, 肯德尔秩相关, 偏秩相关	C	注入缺陷	有效性和块覆盖率之间的相关性高于有效性和规模之间的相关性
Andrews等人[44]	2006	现有的测试集	贪心选择, 随机选择	块覆盖, 判定覆盖, c-use覆盖, p-use覆盖	R平方	C	变异缺陷	测试集规模和覆盖率具有很强的相关性, 规模和覆盖率在解释缺陷检测方面发挥着互补作用
Namin等人[88]	2009	现有的测试集	随机选择	块覆盖, 判定覆盖, c-use覆盖, p-use覆盖	协方差分析, 主成分分析, 皮尔逊相关, 肯德尔秩相关, 调整后的R平方	C	变异缺陷	当控制规模时, 覆盖率有时与有效性相关. 使用规模和覆盖率可以比单独使用规模更准确地预测有效性. 测试集的规模, 覆盖率和有效性这3个变量之间存在非线性关系
Gligoric等人[48]	2013	现有的测试集, 自动生成	随机选择	语句覆盖, 分支覆盖, IMP, AIMP, PCT	肯德尔秩相关, R平方	C, Java	变异缺陷	使用规模作为附加变量并没有改变覆盖率和杀死的变异体数量的相关性
Inozemtseva等人[89]	2014	现有的测试集	随机选择	语句覆盖, 判定覆盖, 修正条件覆盖	调整后的R平方, 肯德尔秩相关, 皮尔逊相关	Java	变异缺陷	当忽略测试集规模的影响时, 测试集覆盖率和有效性之间存在中度到高度的相关性; 当控制测试集规模时, 覆盖率和有效性之间具有低度到中度的相关性
Kochhar等人[90]	2015	自动生成	随机选择	语句覆盖, 分支覆盖	点二序列相关	Java	真实缺陷	测试集的代码覆盖率与其有效性有中度到高度的相关性. 测试集的规模与测试集的有效性有低度到高度的相关性
Gay[20]	2017	自动生成	直接生成	BC, DBC, LC, EC, MC, MNEC, OC, WMMC	Treatment Learning	Java	真实缺陷	更大规模的测试集不一定能更有效地检测真实缺陷
Papadakis等人[29]	2018	现有的测试集, 自动生成	随机选择	变异测试	伪R平方, 肯德尔秩相关, 皮尔逊相关	C, Java	真实缺陷	当控制测试集规模时, 变异得分和缺陷检测之间的相关性会变弱或适中
Kim等人[24]	2020	现有的测试集, 自动生成	直接生成	变异测试	伪R平方, 点二序列相关	Java	变异缺陷, 真实缺陷	当精确定位容易出错的代码时, 变异体的检测率与真实缺陷之间存在很强的相关性, 而与测试集的规模无关
Chen等人[91]	2020	现有的测试集	测试选择, 测试集约简	基于覆盖的测试, 基于变异的测试, 随机测试	—	Java	真实缺陷	测试集规模是一个不切实际的测试目标, 既不是混杂变量, 也不是应该通过实验操纵的自变量

在 2013 年, Gligoric 等人[48]使用回归分析模拟覆盖率和杀死的变异体数量之间的关系, 他们使用规模作为附加变量并没有改变覆盖率和杀死的变异体数量的相关性.

在 2014 年, Inozemtseva 等人[89]的结果表明, 对于大型 Java 程序, 测试集规模与有效性之间存在中等到非常高的相关性. 当忽略测试集规模的影响时, 测试集覆盖率和有效性之间存在中度到高度的相关性; 当控制测试集规模时, 覆盖率和有效性之间的相关性会下降, 具有低度到中度的相关性. 在 2015 年, Kochhar 等人[90]观察到当多次执行测试集中的一些测试用例时, 测试用例会产生不同的输出. 他们重复运行测试集 50 次检测不确定性, 即测试用例在不同的执行中表现出不同的行为, 并从测试集中过滤不确定性测试用例. 他们的实验发现, 测试集的代码覆盖率与其有效性有中度到高度的相关性. 测试集的规模与测试集的有效性有低度到高度的相关性.

在 2017 年, Gay[20]评估了 EvoSuite 框架的缺陷检测能力发现更大规模的测试集不一定能更有效地检测真实缺陷.

(2) 变异得分

在 2018 年, Papadakis 等人[29]研究了测试集规模和变异得分之间的关系, 结果表明了对数关系. 他们的数据表明, 当控制测试集规模时, 变异得分和缺陷检测之间的相关性会变弱或适中.

在 2020 年, Kim 等人[24]对粒度 (类、方法和语句级别) 级别对变异测试有效性的影响进行了全面的研究, 他们发现变异得分在不同代码粒度级别的缺陷检测能力之间的显著差异, 测试集规模影响变异得分与缺陷检测之间的关系, 其影响随粒度级别的不同而变化.

(3) 覆盖率和变异得分

在 2020 年, Chen 等人[91]解释了测试集规模是一个不切实际的测试目标, 既不是混杂变量, 也不是应该通过实验操纵的自变量. 他们的工作解释了通过随机选择和分层控制测试集规模时出现的概念和统计问题, 并得出结论认为随机选择方法存在缺陷. 此外, 他们提出了: 1) 一种公平的比较测试充分性准则的方法; 2) 概率耦合, 一种估计测试集缺陷检测概率的方法. 使用所提出的方法, 他们得出结论: 基于充分性准则的测试选择优于随机选择, 并且一旦满足基于覆盖的充分性标准再切换到基于变异的测试选择最有效.

4.3.2 缺陷种类

覆盖率和变异得分无法充分区分测试检测到的缺陷. 通过调查测试在检测不同种类缺陷方面的有效性, 以及某些种类的缺陷是否比其他种类的缺陷更能规避测试, 可以向开发人员建议需要改进测试以增加缺陷检测的具体方法.

在 2016 年, Schwartz 等人[92]研究了结构化覆盖和发现缺陷能力之间关系的一种因素: 缺陷种类. 他们调查了 26 种不同种类的面向对象类型缺陷, 缺陷是使用变异分析注入的. 他们发现测试集发现缺陷的能力因缺陷类型而异. 他们还发现, 在所有目标程序中, 某些特定种类的缺陷始终不那么频繁地被发现. 进一步地, 在 2018 年, Schwartz 等人[93]研究了 45 种不同类型的缺陷. 结果表明, 发现的特定种类的缺陷比其他缺陷少. 根据他们的结果和遗漏的缺陷种类, 建议同时关注测试预言的强度和代码覆盖率, 以提高测试集的有效性.

在 2020 年, Petrić等人[94]调查了 7 种缺陷种类, 并分析了在 10 个开源系统中每种缺陷种类未被检测到的频率. 他们在统计上寻找测试集和缺陷之间的任何关系. 结果表明, 单元测试的缺陷检测率相对较低, 通常只发现所有缺陷的一半左右. 此外, 与其他缺陷种类相比, 条件边界缺陷种类和方法调用删除缺陷种类不太容易被测试检测到.

4.3.3 测试预言

测试涉及检查系统的行为是否符合预期以发现潜在的缺陷. 给定系统的输入, 将相应的期望的正确行为与潜在的不正确行为区分开来的挑战称为“测试预言问题”[177−179].

在 2015 年, Zhang 等人[95]的结果表明: 1) 测试集中的断言数量与测试集的有效性密切相关, 并且该因素直接影响测试集规模和有效性之间的关系. 2) 断言覆盖率与有效性密切相关, 在控制了断言覆盖的情况下, 语句覆盖和变异得分之间只有中度到高度的相关性, 测试集有效性对断言覆盖比对语句覆盖更敏感. 3) 不同类型的断言会影响包含它的测试集的有效性.

在 2019 年, Zhang 等人[96]将通过随机构造的项目的原始测试集的子集称为伪测试集. 他们使用伪测试集和原始测试集研究语句覆盖、断言覆盖和变异得分之间的相关性. 他们控制断言的数量和测试集的规模进行相关分

析. 结果表明: 1) 伪测试集的相关性 (覆盖准则和变异得分之间) 远高于原始测试集. 2) 语句覆盖和变异得分的相关性比断言覆盖和变异得分的相关性强.

4.3.4 程序结构

MC/DC 度量的有效性对实现的结构高度敏感, 因此作为测试充分性准则可能存在问题.

在 2008 年, Rajan 等人[97]为了评估 MC/DC 对程序结构的敏感性, 使用了来自民用航空电子领域的 6 个系统, 对于每一个系统, 使用系统的两个实现版本, 即有和没有内联 (inlining). 他们首先生成满足非内联实现的 MC/DC 的测试集, 然后在内联实现上运行生成的测试集, 他们发现, 测试集在非内联实现上比内联实现上 MC/DC 覆盖率平均减少了 29.5%.

在 2008 年, Heimdahl 等人[98]评估 MC/DC 充分的测试集在 5 个工业系统上的缺陷检测能力, 对于每个系统, 创建了两个版本的实现, 即带有和不带有内联的实现. 他们发现, 对于所有 5 个系统, 测试集的有效性对实现结构高度敏感. 内联实现上充分的测试集在 10%–5 940% 的范围内优于非内联实现上充分的测试集, 具有更强的缺陷检测能力.

在 2016 年, Gay 等人[99]扩展了以上两篇文章, 结果表明: 1) 在内联实现上生成的测试集的规模通常比在非内联实现上生成的测试集的规模更大, 多出 125.00% 到 1 566.66% 的测试. 虽然在内联实现上实现 MC/DC 的成本更高, 但缺陷检测改进高达 4 542.47%; 2) 在非内联实现上生成的测试集在应用于内联实现时 MC/DC 覆盖率显著降低, 发现的缺陷比在内联实现上生成和执行的测试少 17.88% 到 98.83%; 3) 使用内联实现生成的测试在非内联实现上达到 100% 的 MC/DC 覆盖率, 并且发现的缺陷比在非内联实现上生成和执行的测试多 5 068.49%.

在 2017 年, Byun 等人[100]提出了对象分支覆盖 (object-branch coverage, OBC) 的严格定义, 称为 Flag-Use OBC, 目的是捕获源代码中条件行为的语义. OBC 是对象代码级别的结构化覆盖准则. OBC 具有独立于编程语言的优势, 但编译器的选择和优化可以显著改变生成代码的结构和用于表示分支的指令. 他们展示了 OBC 的缺陷检测能力对编译器和编译器配置的选择敏感. Flag-Use OBC 扩展了 OBC. 他们根据 5 个来自嵌入式控制系统软件的实例进行了评估. 结果表明, Flag-Use OBC 在缺陷检测和对编译器选择和配置变化的鲁棒性方面比 OBC 更有效.

4.3.5 每个测试覆盖的方法数量

在 2018 年, Petrić等人[101]通过识别一组测试代码度量确定是否存在任何可以区分有效和无效 JUnit 测试的特征. 他们从 7 个开源 Java 系统中的 JUnit 测试中收集这组度量. 他们的研究结果表明: (1) 调查的大多数开源系统都没有得到充分测试. 平均超过 66% 的有缺陷的方法与任何 JUnit 测试无关; (2) JUnit 测试覆盖的方法数量与该测试能够发现缺陷密切相关, 发现缺陷方法的最成功的测试是覆盖多个方法的测试.

4.4 小 结

本节介绍了测试用例质量以及测试集规模等其他因素对测试集有效性的影响. 下面简述主要发现.

(1) 测试气味代表开发人员在开发测试用例时应用的次优设计或实现. 一方面, 测试气味的存在会导致测试代码的可维护性降低[172]; 另一方面, 测试气味可能与测试和生产代码的缺陷倾向[85]等问题有关并影响测试用例有效性[86]. 因此, 测试气味可能会对测试用例在生产代码中发现缺陷的整体能力产生负面影响.

(2) 测试集规模、缺陷种类、测试预言、程序结构和每个测试覆盖的方法数量等都被认为对测试有效性有影响.

(3) 一些研究表明测试集规模影响测试集有效性[29,44,88−90], 而另一些研究表明测试集规模几乎不影响测试集有效性[20,48].

(4) 有的研究表明断言覆盖和变异得分的相关性比语句覆盖和变异得分的相关性强[95], 而有的研究表明语句覆盖和变异得分的相关性比断言覆盖和变异得分的相关性强[96].

5 测试集有效性评价分析

从现有研究的结论来看, 存在许多相互矛盾的结论, 如表 6 所示. 由于面向领域软件的测试充分性准则基于传

统的测试充分性准则, 目前对面向领域软件的测试充分性准则评价的研究较少且结论较为一致, 本节重点分析调研文献中除面向领域软件外的 80 篇文献. 本节对测试集有效性评价现有研究的实验设置进行分析, 主要分析实验设置中的不同层面, 包括测试用例来源、测试集构造方式、测试充分性准则、统计方法和缺陷类型, 这些不同层面的实验设置可能会影响测试集有效性评价的结果和结论.

表 6 主要的矛盾的结论总结

类型	矛盾的结论
覆盖准则的有效性	语句覆盖[18]、分支覆盖[19,20,48]、修正条件/判定覆盖[22]分别被表明是最有效的
覆盖率和缺陷检测有效性	覆盖率和缺陷检测有效性相关性有强[18,49,51]有弱[21,22]
变异得分与缺陷检测有效性	变异得分与缺陷检测能力相关性有强[23]有弱[28−30]
测试集规模与测试集有效性	测试集规模影响[29,44,88−90]或几乎不影响测试集有效性[20,48]
语句覆盖、断言覆盖与缺陷检测有效性	断言覆盖与缺陷检测有效性的相关性或强于[95]或弱于[96]语句覆盖与测试集缺陷检测的相关性

5.1 测试用例来源

测试用例生成是软件测试中的一项关键任务. 测试用例生成可以通过手动和自动的方式. 手动编写单元测试是最耗时的测试活动之一[180]. 因此, 越来越多的工作致力于实现自动生成单元测试集.

5.1.1 测试用例自动化生成

表 7 展示了调研的 80 篇文献中明确表明使用的自动化测试用例生成工具的情况. 从表中可以看出, Randoop 和 EvoSuite 工具使用的频率较多.

表 7 调研文献使用的自动化测试用例生成工具

自动化测试用例生成技术	工具	文献	文献数量
随机测试 (random testing)	Randoop	[18,23,24,29,90,104]	6
	JCrasher	[23]	1
	AutoTest	[46]	1
基于约束的测试 (constraint-based testing)	KLEE	[29,59]	2
	Godzilla	[25,55,115]	3
基于搜索的测试 (search-based testing)	EvoSuite	[20,23,24,29,104]	5
	SWAT	[61]	1
基于需求规范的测试 (requirements-based testing)	Copia	[42]	1
	Irulan	[51]	1
	Kind、JKind	[22,99,102]	3
	NuSMV	[52,97,98]	3

(1) 随机测试. 随机测试是一种用于自动化测试用例生成的简单有效的技术. 在随机测试中, 被测程序仅在随机生成的输入上执行, 因此, 随机测试不可能测试程序的所有可能行为. 然而, 在 2011 年, Ciupa 等人[181]的分析表明, 随机测试是有效的并且具有可预测的性能. 在 2011 年, Arcuri 等人[182]的结果表明, 在某些实际情况下, 随机测试是一种可行的选择. 使用随机测试技术的代表性的工具是 Randoop[183].

(2) 基于约束的测试. 基于约束的测试通过求解符号执行 (symbolic execution)[184−186]产生的约束生成测试数据. 使用符号执行技术的代表性的工具是 KLEE[187]. 符号执行的主要缺点是, 如果沿可行执行路径的符号路径约束包含约束求解器 (constraint solver) 无法求解的约束, 则它无法生成测试输入. 混合了具体执行和符号执行的动态符号执行 (dynamic symbolic execution, DSE) 技术, 也称为 Concolic 测试, 可以缓解这个问题. 使用动态符号执行技术工具有 CUTE[188]、DART[189]. 研究人员基于结构化覆盖准则[2,190]和基于变异测试[191,192]提出了各种测试数据生成方法.

(3) 基于搜索的测试. 基于搜索的软件测试[193,194]是基于搜索的软件工程[195]的一个具体案例. 基于搜索的测试

数据生成[11,196−198]是使用元启发式搜索 (metaheuristic search) 技术, 在目标函数 (objective function) 的指引下, 产生测试输入的过程. 常用的搜索算法有遗传算法 (genetic algorithm)、差分进化算法 (differential evolution algorithm) 等. 基于搜索的测试代表性的工具是 EvoSuite[31], 适用于 Java 语言的自动化测试用例生成.

(4) 基于需求规范的测试. 根据需求规范生成测试用例, 即黑盒测试. 在 2006 年, Rajan[199]根据需求自动生成黑盒测试用例, 即使用形式化为时序逻辑性质的需求, 直接在形式化需求的结构上定义覆盖率度量, 并使用自动测试用例生成工具 (模型检查器) 根据形式化需求生成满足度量准则的测试用例. 在 2011 年, Escalona 等人[200]概述了以非形式化方式描述的功能需求生成测试用例的一组有效的相关方法. 基于规范的测试用例可以与基于结构的测试用例相互补充[201], 即首先生成基于规范的测试用例, 然后用基于结构的测试用例扩充测试集.

5.1.2 手动与自动生成的测试用例比较

自动化生成测试用例的一个常见的假设是这些测试减轻了开发人员的大部分工作. 然而, 尽管这种假设已经持续了几十年, 但迄今为止还没有确凿的证据证实这一假设. 自动生成的测试用例与手动编写的测试用例相比如何是一个开放的研究问题. 在选择目标程序时, 研究人员倾向于选择带有开发人员为目标程序编写的测试集的程序, 而不是选择带有使用自动测试生成工具生成的测试集的程序[18,92,93].

在 2013 年, Fraser 等人[202]进行了一项对照实验, 比较了手动编写的测试用例和借助自动化单元测试生成工具 EvoSuite 生成的测试用例. 他们发现, 一方面, 工具支持可以明显提高代码覆盖率等常用指标, 另一方面, 开发人员实际发现的错误数量并没有明显改善. 在 2015 年, 他们通过另一个更大的对照实验[203]扩展了这项研究, 得到了相同的发现.

在 2019 年, Serra 等人[204]使用 3 种自动测试生成工具 EvoSuite、Randoop 和 JTExpert, 比较了手动和自动创建的测试集的性能, 考虑了 3 个方面: 代码覆盖率、变异得分和缺陷检测能力. 他们的研究结果表明, 1) 当前的自动测试用例生成工具能够比手动编写的测试得到更高的覆盖率和变异得分; 2) 手动编写的测试具有很高的缺陷检测能力, 而自动生成的测试在少数真实缺陷中识别出了缺陷, 即它们检测真实缺陷的能力并不好. 他们观察到自动化工具识别的缺陷通常不同于手动编写的测试识别的缺陷, 开发人员可以采用自动生成和手动编写的互补方式发现生产代码中的更多缺陷.

5.2 测试集构造

可以直接生成满足某种测试充分性准则的测试集. 生成的测试集直接用于分析. 在 2014 年, Gopinath 等人[18]研究了广泛使用的覆盖准则 (语句覆盖、块覆盖、分支覆盖和路径覆盖) 与杀死 Java 程序的变异体的能力之间的相关性. 他们分别使用项目中存在的测试集和 Randoop 生成的测试集通过数百个 Java 项目进行了评估.

开发人员可能随程序提供测试用例. 许多现有的评价测试集缺陷检测有效性的研究通过构造项目的原始测试集的子集形成一组测试集, 然后使用这组测试集分析测试集的缺陷检测能力. 原始测试集是每个项目程序的所有测试用例的集合, 即测试池 (test pool). 一组测试集可以从原始测试集通过测试选择 (test selection) 或测试集约简 (test suite reduction) 构造.

表 8 展示了调研的 80 篇文献的测试集构造方式, 其中测试选择或测试集约简分别不包括与回归测试相关的回归测试选择或测试集约简的文献. 从表中可以看出, 通过测试选择构造测试集的文献数量较多.

表 8 调研文献的测试集构造方式

测试集构造方式	文献	文献数量
直接生成	[18,20,22,23,29,43,46,52,58,60,107]	11
测试选择	[19,24,26,29,30,38−42,44,45,47−49,51,54,59,87−91,93,95,96,104,106,109,111,114,116,117,120]	34
测试集约简	[22,25,49,52,55,91,97−100,102]	11

(1) 测试选择

1) 随机选择

随机选择通过随机选择项目的原始测试集的测试用例构建测试集, 例如每次选择一个测试用例, 以增量地达

到充分性准则的充分性. 执行随机选择是为了消除特定的测试集构造算法可能引入的任何偏差[88]. 在 2014 年, Inozemtseva 等人[89]评估大型 Java 程序的测试集规模、覆盖率和有效性之间的关系, 他们为 5 个系统生成了 31 000 个测试集. 测试集的规模分别为: 3 个方法、10 个方法、30 个方法、100 个方法等, 直到测试集规模少于测试方法的总数, 每种规模生成 1 000 个测试集.

然而, 随机选择可能对测试集充分性和缺陷检测之间的相关性的结论造成威胁. 在 2019 年, Zhang 等人[96]的结果表明随机选择构建的测试集的覆盖率与变异缺陷检测之间的相关性远高于原始测试集. 在 2020 年, Chen 等人[91]表明随机选择方法存在缺陷. 此外, 在 2010 年, Chen 等人[205]综合了与自适应随机测试 (adaptive random testing, ART) 相关的最重要的研究成果. 他们特别注意到多样性在测试用例选择策略中的作用.

2) 贪心选择

对于给定的覆盖准则 C, 从测试池 TP 构建测试程序 P 的覆盖充分的最小测试集 T 的步骤为: 1) 从 TP 中随机选择第一个测试用例, 将该测试用例从 TP 中移除并添加到 T. 2) 更新覆盖信息, 从 TP 中选择一个覆盖新的测试目标的测试用例并将该测试用例添加到 T. 3) 重复步骤 2, 直到不再有测试用例可以添加到 T. 例如, 在 2008 年, Gupta 等人[45]通过贪心策略构建满足覆盖准则的最小测试用例集. 在 2020 年, Chen 等人[91]一次贪心地选择一个测试用例, 逐步达到给定充分性准则的充分性要求.

(2) 测试集约简

测试集约简根据充分性准则对原始测试集进行约简, 例如随机生成测试后根据分支覆盖准则进行约简.

在 2012 年, Staats 等人[52]的结果表明, 为满足分支和 MC/DC 覆盖准则而直接生成的测试集的性能明显低于相同规模的随机测试集; 为了满足分支和 MC/DC 覆盖而约简随机测试集比相同规模的随机生成的测试集更有效. 在 2015 年, Gay 等人[22]的分析表明使用结构化覆盖准则作为测试用例生成的补充而不是目标, 可以实现更有效的缺陷检测.

因此, 在测试集有效性评价的研究中需要考虑测试集约简构建的测试集对研究结果的影响.

5.3 测试充分性准则的优化

(1) 改进的测试充分性准则

改进的测试充分性准则, 即在基于覆盖或基于变异的充分性准则的基础上结合了其他因素的准则, 能够提高该准则的缺陷检测有效性. 改进的测试充分性准则增加的因素有可观察性、多样性和执行语句的频率.

1) 可观察性

在 2013 年, Whalen 等人[102]提出了 Observable MC/DC (OMC/DC), 该覆盖准则将 MC/DC 覆盖指标与可观察性概念结合起来, 有助于确保缺陷结果传播到监控的变量. 他们发现满足 OMC/DC 的测试集明显比满足 MC/DC 的测试集更有效, 与 MC/DC 相比, 最多可以发现 88% 更多的缺陷, 并且对程序结构和监控变量的选择不太敏感. 在 2015 年, Gay 等人[22]的结果表明生成满足 OMC/DC 准则的测试集比相同规模的随机测试集可以实现更高的缺陷检测.

2) 多样性

在 2014 年, Wang 等人[103]定义了上下文多样性的概念, 并且提出了 3 种策略研究上下文多样性如何提高数据流测试准则的有效性. 3 种策略即 CARS-H、CARS-L 和 CARS-E, 分别选择具有更高、更低和更均匀分布的上下文多样性的测试用例, 以构建数据流测试准则充分的测试集. 案例研究表明, CARS-H 和 CARS-E 可以将数据流测试准则的有效性分别提高 10.6%–22.1% 和 0.8%–5.3%, 而 CARS-L 的效果可能会降低 2.0%–22.2%. 实验结果表明, 使用具有更高上下文多样性的测试用例的策略可以显著提高上下文感知软件的现有数据流测试准则的有效性. 在 2018 年, Shin 等人[104]提出了一种多样性感知的变异充分性准则, 即区分变异充分性准则 (distinguishing mutation adequacy criterion), 以及基于该准则的区分变异得分 (distinguishing mutation score). 他们基于该准则的形式化定义, 证明了区分变异充分性准则包含了传统的变异充分性准则. 这种包含关系表明区分变异充分性准则在检测缺陷方面比传统的变异充分性准则更有效. 他们还根据缺陷检测有效性、测试集规模和各种变异得分

水平对该充分性准则进行了实证评估. 他们使用了包含 352 个真实缺陷的真实世界应用程序. 结果表明, 在所有变异得分水平上, 与传统的变异充分性准则相比, 该准则的缺陷检测有效性提高了 74.8%, 然而需要的测试用例多 3.07 倍.

3) 执行语句的频率

在 2021 年, Aghamohammadi 等人[105]提出了语句频率覆盖 (statement frequency coverage) 准则, 将执行语句的频率纳入语句覆盖评估测试集的有效性. 他们利用变异体作为缺陷的代理. 他们通过将语句频率覆盖应用于 22 个真实的 Python 项目进行实证评估, 这些项目有超过 118 000 行源代码和 21 000 个测试用例. 他们的结果表明语句频率覆盖优于语句和分支覆盖准则. 最后, 他们通过包含理论证明语句频率覆盖包含语句覆盖.

(2) 新型测试充分性准则

在 2017 年, Someoliadyi 等人[106]提出了程序状态覆盖 (program state coverage, PSC). PSC 与行覆盖 (line coverage, LC) 几乎相同, PSC 进一步考虑了每一行执行的不同程序状态的数量. 他们对来自 Apache Commons Math 和 Apache Commons Lang 的 120 个测试集进行了实验, 以评估 PSC. 结果表明, 与 LC 相比, PSC 与归一化变异得分 (normalized mutation score) 的相关性更强.

在 2021 年, Ali 等人[107]提出了 3 种基于量子程序 (quantum programs) 输入和输出的覆盖准则, 包括输入覆盖 (input coverage)、输出覆盖 (output coverage) 和输入-输出覆盖 (input-output coverage), 以及两种类型的测试预言. 他们用 5 个量子程序进行了实验, 使用变异分析确定覆盖准则的有效性. 结果表明, 使用成本低的覆盖准则 (即输入覆盖) 可以获得较高的变异得分, 使用成本高的覆盖准则 (即输出覆盖和输入-输出覆盖) 无法大幅提高变异得分.

5.4 统计方法

许多测试集有效性评价的研究通过相关性分析研究测试集与缺陷检测有效性之间的相关性. 相关性越强, 测试集有效性越好. 相关性分析包括不控制任何变量的直接相关分析 (direct correlation analysis) 和控制变量 (如测试集规模) 的偏相关分析 (partial correlation analysis).

(1) 直接相关分析

对于直接相关分析, 变量 X 和 Y 之间的相关性是直接使用这两个变量的值进行的, 而不控制任何其他变量. 然而, 在 X 和 Y 之间观察到的相关可能是由其他的变量引起的, 而不是实际相关.

(2) 偏相关分析

偏相关分析广泛用于控制其他变量. 偏相关测量两个变量之间关系的强度, 同时控制一个或多个其他变量的影响. 偏相关计算两个变量之间的相关性以排除第 3 个变量, 这使得可以找出变量 X 和 Y 之间的相关性 r_{XY} 是否是由变量 Z 生成. 偏相关 $r_{XY.Z}$ 的计算需要各个变量之间的相关性, 即变量 X 和 Y 之间的直接相关性 r_{XY}, 两个变量 X 和 Y 与第 3 个变量 Z 的相关性 r_{XZ} 和 r_{YZ}. 偏相关 $r_{XY.Z}$ 说明变量 X 与变量 Y 的相关性有多强, 计算公式为:

$$r_{XY.Z} = \frac{r_{XY} - r_{XZ} \cdot r_{YZ}}{\sqrt{(1 - r_{XZ}^2) \cdot (1 - r_{YZ}^2)}}.$$

有多种方法可以衡量两个变量之间的相关性, 例如皮尔逊相关性 (Pearson correlation) 可以用于计算 r_{XY}.

表 9 展示了调研的 80 篇文献在相关性分析中使用的不同的相关系数. 从表 9 中可以看出, 在实验中, 现有研究主要使用 4 种相关系数: 1) 皮尔逊相关系数; 2) 肯德尔秩相关系数; 3) 斯皮尔曼秩相关系数; 4) R 平方或调整后的 R 平方. 此外, 点二序列相关系数、秩二序列相关系数和马修斯相关系数使用较少, 这 3 种相关系数的使用占总使用频次的 10.53%.

(1) 皮尔逊相关系数

皮尔逊相关系数, 又称皮尔逊积矩相关系数 (Pearson product-moment correlation coefficient). 皮尔逊相关系数是两个变量 X 和 Y 之间线性相关性的度量. 该相关系数的取值范围是 [−1, 1]. 该相关系数在 0.1–0.3 之间表示低度相关, 在 0.3–0.5 之间表示中等相关, 而在 0.5–1 之间表示高度相关. 当皮尔逊相关系数接近 1 时, 称为完美相关.

表 9 调研文献相关性分析中使用的相关系数

相关系数	文献	文献数量
肯德尔秩相关系数 (Kendall rank correlation coefficient, Kendall's τ)	[18,19,28−30,48,51,53,87−89,95,96,105,106,113]	16
R平方/调整后的R平方 (R-squared/adjusted R-squared)	[12,18,19,21,28,43,44,47−49,88,89,95,96,113]	15
皮尔逊相关系数 (Pearson correlation coefficient, Pearson's r)	[12,14,29,30,47,49,60,88,89,95,96,103,111]	13
斯皮尔曼秩相关系数 (Spearman's rank correlation coefficient, Spearman's ρ)	[12,14,19,53,87,105,114]	7
点二序列相关系数 (point biserial correlation coefficient)	[23,24,90]	3
秩二序列相关系数 (rank biserial correlation coefficient)	[23,104]	2
马修斯相关系数 (Matthews correlation coefficient)	[101]	1

(2) 肯德尔秩相关系数

与皮尔逊相关系数不同, 肯德尔秩相关系数是等级相关性的度量. 肯德尔相关系数根据数据的等级评估统计关联, 使用成对的观测值, 并根据成对之间的一致性和不一致确定关联强度. 设 (x_1,y_1) 和 (x_2,y_2) 是联合随机变量 X 和 Y 的观测值, 如果 (x_2-x_1) 和 (y_2-y_1) 具有相同的符号, 则认为这对观测值是一致 (concordant) 的; 如果 (x_2-x_1) 和 (y_2-y_1) 具有相反的符号, 则认为这对观测值是不一致 (discordant) 的. 然后, 基于一致对和不一致对计算 Kendall's τ 相关系数, 范围从−1 到 1, 1 表示所有数据对一致, 而−1 表示所有数据对不一致.

(3) 斯皮尔曼秩相关系数

斯皮尔曼秩相关系数是秩相关性的非参数度量, 即两个变量之间秩的统计依赖性. 分别对变量 X 和 Y 的数据编秩, 例如将秩 1 分配给列中最大的数字, 设 p_i 表示 X_i 的秩, q_i 表示 Y_i 的秩, 则 $d_i=p_i-q_i$, 斯皮尔曼秩相关系数 r_s 的计算公式为:

$$r_s=1-\frac{6\sum_i d_i^2}{n(n^2-1)},$$

其中, n 是两个变量的数据点数, d_i 是第 i 个元素的秩差.

(4) R 平方或调整后的 R 平方

R 平方是拟合的统计量度, 表示回归模型中的自变量解释了因变量的多少变化. 相关性解释了自变量和因变量之间关系的强度, R 平方解释了一个变量的方差在多大程度上解释了第 2 个变量的方差. 如果模型的 R 平方为 0.5, 则观察到的变化中大约有一半可以由模型的输入解释. 向回归模型添加更多自变量或预测变量往往会增加 R 平方值, 这会诱导添加更多变量, 称为过拟合.

调整后的 R 平方是 R 平方的修改版本, 对模型中的预测变量数量进行了调整, 有助于确定与因变量的相关性在多大程度上是由于添加了这些变量. 调整后的 R 平方尤其适用于多元线性回归.

5.5 软件缺陷

测试技术的评估在软件测试研究中起着重要的作用. 由于提取和复制真实缺陷具有挑战性, 因此软件测试中的研究通常在软件中注入缺陷, 手动注入缺陷或使用一组变异算子注入缺陷 (即变异体). 使用变异算子可以系统地、自动地、可重复地注入大量缺陷, 从而有助于对测试技术的有效性进行分析. 手动注入的缺陷和变异体不同于真实缺陷, 因此可能不适合评估测试技术[124].

尽管如此, 变异缺陷广泛用于评估软件工程领域方法的研究. 在 2012 年, Hao 等人[206]提出了按需约简测试用例集的方法, 该方法将测试集约简问题建模为整数线性规划问题. 他们使用生成的变异体评估测试集约简导致的

缺陷检测能力损失. 在 2018 年, Xue 等人[10]使用变异体评估基于覆盖率的缺陷定位的有效性.

使用真实缺陷可以使软件测试中的实证研究具有可比性、可重复性. Defects4J[207], 是一个数据库和可扩展的框架, 它提供了真实缺陷. Defects4J 的初始版本包含来自 5 个开源程序的 357 个真实缺陷. 每个真实缺陷都伴随着一个全面的可以暴露该缺陷的测试集. 此外, 研究人员提出了基于真实缺陷注入模拟真实缺陷的变异缺陷. 以近年来获得了广泛普及的深度学习为例, 在 2021 年, Humbatova 等人[208]依赖于 3 个关于深度学习系统中真实缺陷的实证研究[209−211], 定义了一组 35 个变异算子, 并且在工具 Deepcrime 中实现了这些变异算子中的 24 个变异算子.

5.5.1 变异缺陷与真实缺陷

一方面, 变异得分可以近似测试集的缺陷检测有效性; 另一方面, 变异缺陷检测可以用来近似真实缺陷检测. 许多测试集有效性评价的研究使用变异缺陷代替真实缺陷评估测试集的缺陷检测有效性. 在没有真实缺陷的情况下使用变异体引起了人们对变异体和真实缺陷是否表现出相似特性的担忧. 表 10 列出了变异缺陷与真实缺陷相关性的主要研究.

表 10 变异体与真实缺陷相关性

作者	年份	统计方法	语言	主要发现总结	
Daran等人[212]	1996	—	C	变异体产生的不正确的内部状态和不正确的输出与真实缺陷产生的相似	变异体与真实缺陷相关性强
Andrews等人[213]	2005	配对t-检验	C	使用变异算子生成的变异体与真实缺陷相似. 变异体与手动注入的缺陷不同, 手动注入的缺陷更难检测	
Andrews等人[44]	2006	R平方	C	被杀死的变异体的比例是真实缺陷检测率的良好预测指标	
Just等人[23]	2014	点二序列相关, 秩二序列相关	Java	73%的真实缺陷与常用变异算子产生的变异体有关. 在控制代码覆盖率时, 平均有2个变异体与1个真实的缺陷耦合. 27%的真实缺陷与常用变异算子产生的变异体无关, 17%的真实缺陷与任何变异体无关	
Petrovic等人[214]	2021	斯皮尔曼秩相关	C++, Java, Go, Python, TypeScript, JavaScript, Dart, SQL, Common List, Kotlin	变异体确实与真实缺陷耦合	
Gopinath等人[124]	2014	R平方	C, Java, Python, Haskell	一个典型的缺陷涉及大约3–4个token, 而且很少等同于任何传统的变异算子. 在一种语言中最优的变异算子可能对其他语言不是最优的	变异体与真实缺陷相关性弱
Papadakis等人[29]	2018	伪R平方, 肯德尔秩相关, 皮尔逊相关	C, Java	变异体与缺陷检测的相关性很弱, 使用变异体代替真实的缺陷可能会出现问题	
Luo等人[125]	2018	肯德尔秩相关	Java	所研究的测试用例优先排序技术在变异体上的相对性能可能与在真实缺陷上的相对性能没有很强的相关性. 相关性因不同变异算子生成的变异体而异	

(1) 变异体与真实缺陷相关性强

在 1996 年, Daran 等人[212]进行了第一个比较由真实缺陷和一阶变异产生的软件错误的实验. 他们使用生成的测试集执行程序, 从程序执行轨迹中记录了总共 3 730 个错误: 1 458 个错误是由真实缺陷产生的, 其他 2 272 个错误是由变异产生的. 由变异导致的 2 272 个错误中有 85% 也是由真实缺陷产生的. 他们观察到变异会产生与真实缺陷一样的错误行为.

在 2005 年, Andrews 等人[213]研究了手动或使用变异算子注入的缺陷. 他们得出的结论是使用变异算子生成的变异体与真实缺陷相似, 然而变异体与手动注入的缺陷不同, 手动注入的缺陷更难检测. 在 2006 年, Andrews 等人[44]基于 4 种常见的控制流和数据流准则 (块覆盖、判定覆盖、c-use 覆盖和 p-use 覆盖) 研究了使用变异算子注入的缺陷, 得到的结果是被杀死的变异体的比例是真实缺陷检测率的良好预测指标.

在 2014 年, Just 等人[23]的结果表明 73% 的真实缺陷与常用变异算子产生的变异体有关. 在控制代码覆盖率

时, 平均有 2 个变异体与一个真实的缺陷耦合. 然而, 27% 的真实缺陷与常用变异算子产生的变异体无关, 17% 的真实缺陷与任何变异体无关, 应该增强常用的变异算子集.

在 2021 年, Petrovic 等人[214]通过涉及 1 502 个错误、近 40 万个变异体和超过 3 300 万个测试目标执行的耦合分析, 发现对于 1 043 个 (70%) 错误, 变异测试会在引入错误的变更中报告缺陷耦合变异, 459 个 (30%) 错误未与任何生成的变异体耦合. 他们确定了没有缺陷耦合的原因: 弱变异算子 (weak mutation operator)、缺失变异算子 (missing mutation operator) 和无变异算子 (no such mutation operator).

(2) 变异体与真实缺陷相关性弱

在 2014 年, Gopinath 等人[124]通过大量随机选择的使用 4 种不同编程语言的项目的分析表明, 一个典型的缺陷涉及大约 3–4 个 token, 并且很少等同于任何传统的变异算子. 他们的分析还表明, 在一种语言中最优的变异算子可能对其他语言不是最优的. 他们建议变异分析需要更好的实证支持来证明变异体检测和在大量实际程序中检测实际程序错误之间的联系.

针对回归测试用例优先排序, 在 2018 年, Luo 等人[125]进行了比较应用真实缺陷和变异缺陷的测试用例优先排序技术的性能的第一个实证研究. 他们在 *APFD* (average percentage of faults detected) 和 *APFDc* (cost-cognizant average percentage of faults detected) 指标方面检查了 8 种不同测试用例优先排序方法的性能, 这些方法应用于来自 Defects4J 数据集的 357 个真实缺陷和一组超过 35k 变异体的数据集. 结果表明, 所研究的测试用例优先排序技术在变异体上的相对性能可能与在真实缺陷上的性能没有很强的相关性. 这表明在某些情况下, 在一组变异体上表现最好的技术在应用于真实缺陷时可能不是最好的技术. 他们还说明, 这些相关性因不同变异算子生成的变异体而异, 这取决于所选变异算子是否反映了项目程序的典型错误.

5.5.2 变异测试工具

由于涉及大量缺陷, 将变异应用于现实世界的程序需要自动化工具. 在这种情况下, 该方法的有效性很大程度上取决于所使用工具的特性. 因此, 在使用自动化工具时, 它们的实施不足可能导致结果不准确. 表 11 展示了调研的 80 篇文献使用的变异测试工具情况. 从表中可以看出, PIT[215]和 Major[216]工具使用较多.

表 11 调研文献使用的变异测试工具

变异测试工具	文献	文献数量
PIT	[12,18,24,28,86,89,94−96,106,113,114]	12
Major	[14,23,24,91,96,104]	6
Mothra	[25,55−58,115]	6
Javalanche	[13,19,37,48]	4
MuJava	[92,93,110]	3
Proteum	[54,88,120]	3
MILU	[27]	1
SRCIROR	[30]	1
MuCheck	[51]	1
MuClipse	[103]	1
Stryker Mutator	[60]	1
Mull	[60]	1
Mutatest	[105]	1

在 2017 年, Kintis 等人[217]评估了 4 种 Java 变异测试工具的缺陷检测能力, 包括 PIT、MuJava、Major 和 PIT 工具的研究版本 PITrv. 基于真实缺陷和变异测试工具引入的变异体, 他们发现工具的有效性之间存在很大差异, 并没有工具能够包含其他工具. 此外, PITrv 工具取得了最好的结果, PITrv 发现的缺陷总数优于其他工具.

5.5.3 使用基于变异的测试评估的有效性威胁

在 2011 年, Namin 等人[218]通过进行的对照实验结果表明, 在测试实验中使用变异体对由变异算子、测试集规模和编程语言等一些影响因素引起的外部有效性威胁高度敏感. 他们建议任何基于变异体的实验结果的解释或

推广都应根据所涉及的影响因素进行论证.

在 2016 年, Papadakis 等人[219]表明被包含的变异体 (subsumed mutants)(也称为冗余变异体) 的存在可以人为地夸大测试技术检测缺陷的能力, 必须在执行基于变异的测试之前移除被包含的变异体, 以避免对有效性造成威胁.

5.6 小 结

本节从测试用例来源、测试集构造、测试充分性准则、统计方法和缺陷类型 5 个层面详细分析了测试集有效性评价现有研究的实验设置. 下面总结主要的发现.

(1) 目前自动化测试用例生成工具使用较多的是 Randoop 和 EvoSuite. 自动生成的测试能够比手动编写的测试得到更高的覆盖率和变异得分, 但是发现的错误数量并没有明显改善[202−204].

(2) 测试集构造可以基于测试充分性准则直接生成[18], 也可以从原始测试集中通过测试选择[89]或测试集约简[22]构造.

(3) 改进的测试充分性准则, 即在基于覆盖率或基于变异的充分性准则的基础上结合了其他因素的准则, 例如可观察性[102]、多样性[103,104]和执行语句的频率[105], 能够提高该准则的缺陷检测有效性.

(4) 相关性分析包括直接相关分析和偏相关分析. 现有测试集有效性评价研究中使用较多的相关系数有: 皮尔逊相关系数、肯德尔秩相关系数、斯皮尔曼秩相关系数以及 R 平方和调整后的 R 平方.

(5) 一些研究表明变异体与真实缺陷耦合[214], 而另一些研究表明使用变异体代替真实的缺陷可能会出现问题[29,124,125]. 在一种语言中最优的变异算子可能对其他语言不是最优的[124]. 变异测试工具的缺陷检测能力之间存在很大差异[217]. 在测试实验中使用变异体对由变异算子、测试集规模和编程语言等一些影响因素引起的外部有效性威胁高度敏感[218].

6 测试集有效性评价的应用

软件回归测试是大多数软件项目开发过程的组成部分. 如图 10 所示[220], 其中 P 是被测程序的原始版本, P' 是 P 经过修改的程序; T 是 P 的测试集, T_{obs} 是过时的 (Obsolete) 测试用例, T_{all} 是 T 删除 T_{obs} 后的测试集. 随着项目变得越来越大并且测试用例数量增加, 使用 T_{all} 中的所有测试用例执行回归测试测试 P'可能变得非常昂贵. 如果软件工程师能够在回归测试期间识别并运行更有可能检测到缺陷的测试, 那么他们可能能够更好地管理回归测试活动. 测试用例集优化技术包括测试用例集约简 (test suite reduction, TSR)、回归测试选择 (regression test selection, RTS) 和测试用例优先排序 (test case prioritization, TCP).

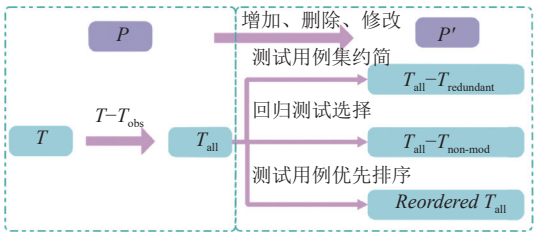


图 10 测试用例集优化技术

回归测试是大多数主要软件项目开发过程中不可或缺的一部分, 该领域研究不断发展, Yoo 等人[221]、Di Nardo 等人[6]、Rosero 等人[222]和 Do[220]通过综述概述了该领域的研究. 本节主要关注测试用例集优化后测试集的缺陷检测有效性, 例如由于测试的时间和资源限制而约简测试集后, 是否保持测试集的缺陷检测能力.

当前的测试用例集约简、回归测试选择和测试用例优先排序等回归测试测试集优化方法大多建立在代码覆盖率或变异得分的基础上. 基于覆盖的技术[6]强调使用覆盖准则, 试图定位已被修改或可能受修改影响的程序组件 (例如语句或定义-使用对), 并从测试集中选择执行这些组件的测试用例. 基于变异的技术[12,13]基于被杀死的变异体进行测试用例集优化.

6.1 测试用例集约简

测试用例集约简与程序版本变更不相关, 其目的是从原始测试用例集中选择能满足所有测试需求 (例如覆盖率或变异得分) 的子集. 测试用例集约简通过去除冗余测试用例 $T_{\text{redundant}}$ 减小 T_{all} 的规模. 表 12 列出了测试集约简与缺陷检测能力的主要研究.

表 12 测试集约简与缺陷检测能力

作者	年份	充分性准则	统计方法	语言	缺陷类型	主要发现总结	
Wong等人[108]	1995	块覆盖, all-uses覆盖	—	C	注入缺陷	即使测试集规模的减小很大, 最小化测试集的缺陷检测能力略有下降或没有下降	测试集约简不会显著降低测试集的缺陷检测能力
Wong等人[109]	1997	块覆盖	—	C	真实缺陷	测试集规模显著减小但具有几乎相同的缺陷检测有效性	
Zhang等人[110]	2011	方法覆盖, 语句覆盖	方差分析	Java	注入缺陷, 变异缺陷	4种传统的测试集约简技术可以有效地约简JUnit测试集, 而不会显著降低测试集的缺陷检测能力	
Rothermel等人[111]	1998	边覆盖	皮尔逊相关	C	注入缺陷	测试集的缺陷检测能力可能会因最小化而受到严重影响	测试集约简会显著降低测试集的缺陷检测能力
Rothermel等人[112]	2002	边覆盖	—	C	注入缺陷, 真实缺陷	测试集的缺陷检测能力可能会因测试集规模的减小而受到严重影响	
Shi等人[113]	2018	行覆盖, 变异测试	肯德尔秩相关, R平方	Java	真实缺陷, 变异缺陷	约简的测试集的FBDL可能很高, 并且可用的FBDL预测器不是很有效, 开发人员在决定使用测试集约简时需要非常谨慎	
Chen等人[114]	2017	语句覆盖, 方法覆盖, 断言覆盖	斯皮尔曼秩相关	Java	变异缺陷	断言覆盖率和断言计数都与基于覆盖率的测试集约简技术的有效性 (包括测试集规模约简和缺陷检测能力损失)显著相关	

6.1.1 基于覆盖的测试集约简

基于覆盖的测试用例集约简的需求由覆盖准则定义, 例如语句覆盖.

在 1995 年, Wong 等人[108]研究了在保持覆盖率不变的情况下减小测试集规模对缺陷检测的影响问题. 对于每个测试集, 他们找到了一个最小子集, 该子集保留了原始测试集的数据流覆盖率. 结果表明, 即使测试集规模的减小很大, 最小化测试集的缺陷检测能力略有下降或没有下降. 在 1997 年, Wong 等人[109]的研究表明, 在测试集规模显著减小但具有几乎相同的缺陷检测有效性方面, 最小化测试集与原始的非最小化测试集相比具有规模/有效性优势. 在 2011 年, Zhang 等人[110]使用真实的 JUnit 测试集研究 Java 程序的测试集约简技术. 他们为 JUnit 测试集实现了 4 种具有代表性的测试集约简技术, 即贪心技术、Harrold 等人的启发式技术[223]、GRE 启发式技术[224]和 ILP[225]技术. 结果表明, 4 种传统的测试集约简技术可以有效地约简 JUnit 测试集, 而不会显著降低测试集的缺陷检测能力.

尽管一些实证证据表明, 与原始测试集相比, 最小化测试集检测缺陷的能力几乎没有损失, 但是在 1998 年, Rothermel 等人[111]的结果表明, 测试集的缺陷检测能力可能会因最小化而受到严重影响. 在 2002 年, Rothermel 等人[112]的研究结果表明, 测试集的缺陷检测能力可能会因测试集规模的减小而受到严重影响.

在 2017 年, Chen 等人[114]第一次调查测试预言对基于覆盖率的测试集约简的影响. 他们对 10 个 GitHub Java 项目进行了 4 种广泛使用的基于覆盖率的测试集约简技术的实证研究. 通过使用断言覆盖率[37]和断言计数[95]衡量断言的有效性, 并使用测试集规模约简和缺陷检测能力损失[112,206]衡量测试集约简技术的有效性. 实验结果表明, 断言覆盖率和断言计数都与基于覆盖率的测试集约简技术的有效性 (包括测试集规模约简和缺陷检测能力损失) 显著相关. 基于研究结果, 他们还提出了一种断言感知测试集约简技术, 该技术在成本效益方面优于传统的测试集约简技术.

6.1.2 基于变异的测试集约简

基于变异的测试用例集约简是在 1995 年由 Offutt 等人[115]提出的, 即约简的测试集达到与原始测试集相同的

变异得分.

在 2014 年, Shi 等人[12]根据被杀死的变异体定义的需求评估测试集约简, 即约简的测试集必须杀死被原始测试集杀死的相同的变异体. 他们还提出了一个更强大的约简需求, 即约简的测试集保留原始测试集的语句覆盖率和杀死的变异体. 结果表明, 基于语句覆盖的约简可以平均减小 62.9% 的测试集规模, 但杀死的变异体减少 20.5%. 相比之下, 基于变异体的约简没有杀死变异体的减少, 与使用语句覆盖约简的测试集相比, 测试集规模增加了 11.9 个百分点 (percentage point). 同时基于语句覆盖和变异体的约简没有杀死变异体的减少, 但与基于变异体的约简相比, 测试集规模增加了 2.7 个百分点. 此外, 他们评估了各种约简技术约简的测试集杀死的变异体如何随着软件的演化而变化. 他们发现杀死的变异体的数量保持稳定, 即使在许多软件版本之后, 杀死的变异体数量的中位数最多下降 0.76 个百分点.

6.1.3 测试集约简相关研究

根据表 12, 即使约简的测试集保持了覆盖率, 这些测试集在某些情况下会显著降低缺陷检测能力[111,112]. 为了限制缺陷检测能力的损失, 一些测试集约简技术有意在约简的子集中保留冗余测试用例[226]. 然而, 这些技术约简的测试集的缺陷检测能力仍然有一定程度的损失. Hao 等人[206]提出了一种允许约简测试集时设置缺陷检测能力损失的上限的方法.

基于其他信息约简测试用例集. 在 2019 年, Philip 等人[227]介绍了一个执行数据驱动测试最小化的系统 FastLane. FastLane 使用基于丰富的测试历史和提交日志构建的轻量级机器学习模型来预测测试结果. 预测了结果的测试不需要显式运行, 从而节省了测试时间和资源.

测试用例集约简的方法有很多, 例如基于覆盖率信息的 tie-breaking[228], 在 2018 年, Khan 等人[229]通过系统的文献综述介绍了测试集约简的方法和方法的评价指标.

6.1.4 测试集约简评价指标

设 T 为测试集, T' 为将测试集 T 约简的子集, $\#Faults()$ 表示测试集检测到的缺陷数, 缺陷检测能力损失 (fault-detection capability loss) 定义为:

$$\frac{\#Faults(T)-\#Faults(T')}{\#Faults(T)}.$$

在 2018 年, Shi 等人[113]提出了第 1 项使用真实测试失败评估测试集约简代价的研究. 他们分析了 32 个在 Travis[230]上运行测试的 GitHub 项目的 1478 个失败构建. 分析表明, FBDL (failed-build detection loss) 即 $(|F|-|F_r|)/|F|\times 100$ (F 是约简点之后所有未来失败构建的集合; F_r 是 F 的子集, 其中约简的测试集检测到原始测试集检测到的所有缺陷), 可高达 52.2%. 他们发现两个传统使用的度量 (规模约简和测试需求损失) 并不是 FBDL 的好的预测度量. 他们的结果表明开发人员在决定使用测试集约简时需要非常谨慎, 因为约简的测试集的 FBDL 可能很高, 并且可用的 FBDL 预测器不是很有效.

6.2 回归测试选择

为了降低成本, 可以使用回归测试选择技术从 T_{all} 中选择测试输入的子集. 回归测试选择通过仅重新运行执行代码变更部分的测试用例降低回归测试成本, 即去除不执行修改代码的测试用例 $T_{\text{non-mod}}$. 当前研究人员已经提出了多种“安全”[116,117]的回归测试选择方法, 即不遗漏所有能检测缺陷的测试用例. 但是, 达到安全的条件并不总是成立[118].

6.2.1 基于覆盖的回归测试选择

在 2011 年, Chen 等人[119]采用集群测试选择 (cluster test selection, CTS) 框架, 原始测试集中的所有测试用例首先在原始程序版本上执行, 可以为每个测试用例生成例如语句覆盖配置文件等多种类型的配置文件, 每个测试用例的执行配置文件由程序切片过滤, 以突出显示受修改影响的软件部分, 然后将具有相似执行配置文件的测试用例放入同一个集群中, 最后从每个集群中随机选择几个测试用例执行, 并且如果一个测试用例失败, 那么同一个集群中的所有其他测试用例都被选择. 由于聚类分析的不确定性和采样策略的随机性, CTS 并不“安全”, 即会错过

一些检测缺陷的测试用例.

6.2.2 回归测试选择相关研究

已经提出了多种回归测试选择算法, 这些算法利用代码覆盖率信息或其他信息选择测试用例, 例如基于重构信息[231]或基于相似度信息[232]. 在 2019 年, Machalica 等人[233]提出了一种预测测试选择策略, 该策略选择一个测试子集, 测试子集执行提交给持续集成系统的每个变更. 该策略是使用基本的机器学习技术从历史测试结果的大型数据集中学习的. 他们的工作表明通过结合许多信息源, 能够构建一个提供良好和可预测性能的测试选择策略. 在 2017 年, Kazmi 等人[234]通过系统的文献综述介绍了有效的回归测试用例选择技术.

6.2.3 回归测试选择评价指标

回归测试选择的一个目标是选择具有较强缺陷检测能力的子集. 设 T 为原始测试集, T' 是选择的 T 的一个子集, T_F 是所有检测缺陷的测试用例的集合, T'_F 表示选择的检测缺陷的测试用例的子集. 那么召回率 (*recall*) 定义为:

$$recall = \frac{|T'_F|}{|T_F|}.$$

“安全”的回归测试选择技术的召回率是 100%. 精度 (*precision*) 定义为:

$$precision = \frac{|T'_F|}{|T'|}.$$

F-measure (*F*) 结合了召回率和精度:

$$F = \frac{2 \times recall \times precision}{recall + precision}.$$

6.3 测试用例优先排序

测试用例优先排序对 T_{all} 中的测试用例重新排序, 提供了一种最早运行具有最高优先级的测试用例的方法. 测试期间提高的缺陷检测速度可以为被测系统提供更快的反馈, 从而更早地修复缺陷.

测试集有效性的提高. 在 2003 年, Harder 等人[235]通过操作抽象变化生成的测试集与具有 100% 分支覆盖率的测试集规模一样小, 检测到的缺陷也一样多, 并且更擅长检测某些常见缺陷. 实证研究表明, 失败错误传播会抑制有效的软件测试[7,236]. 为了提高测试集的有效性, 需要降低测试用例遭受失败错误传播的概率[134]. 由此可见, 研究人员从不同的角度提高测试集的缺陷检测能力, 而测试用例优先排序通过确定应该首先执行哪些测试用例以提高缺陷检测速度.

测试用例集优化可以根据是否丢弃测试用例分类, 如图 11 所示. 测试用例集约简和回归测试选择需要在降低测试代价的同时保持缺陷检测能力. 因为测试用例优先排序不会丢弃测试用例[120], 所以测试用例优先排序可以避免测试集约简和回归测试选择可能出现的缺点. 在可以接受丢弃测试用例的情况下, 可以将测试用例优先排序与测试集约简或回归测试选择结合使用, 对约简后[237]或选择后[238]的测试集中的测试用例进行优先排序, 即测试集约简和优先排序 (test suite reduction and prioritization) 或测试用例选择和优先排序 (test case selection and prioritization).

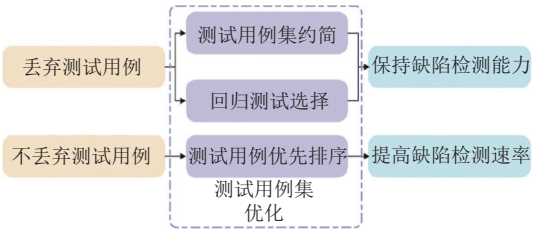


图 11 测试用例集优化与缺陷检测有效性

6.3.1 基于覆盖的测试用例优先排序

基于覆盖率的优先排序方法可以分为两种[120]: 基于总覆盖率 (total coverage) 和基于附加覆盖率 (additional

coverage). 以语句覆盖为例, 总覆盖率方法根据每个测试用例覆盖的语句总数对测试用例进行优先排序. 当多个测试用例覆盖相同数量的语句时, 使用其他规则进行排序. 附加覆盖率方法根据每个测试用例覆盖新的语句数对测试用例进行优先排序. 同样地, 当多个测试用例覆盖相同数量的新的语句时, 使用其他规则进行排序.

在 2002 年, Srivastava 等人[121]构建了 Echelon, 根据对程序所做的变更对应用程序的测试集进行优先排序. Echelon 利用启发式算法计算哪些测试用例将覆盖程序中受影响的基本块, 并排序测试集以最大限度地覆盖受影响的程序, 从而可以快速地发现缺陷.

6.3.2 基于变异的测试用例优先排序

在 2015 年, Lou 等人[13]提出了一种软件演化测试用例优先排序方法. 他们的方法首先确定对原始程序的修改. 其次, 该方法生成变异算子仅发生在修改上的变异体. 最后, 该方法根据缺陷检测能力安排测试用例的执行顺序. 在所提出的方法中, 他们提出了两个模型 (基于统计的模型和基于概率的模型) 计算测试用例的缺陷检测能力. 基于统计的模型根据被杀死的变异体的数量计算测试用例的缺陷检测能力, 基于概率的模型考虑变异体的分布计算测试用例的缺陷检测能力. 实验结果表明, 在测试方法级别对测试用例进行优先排序时, 使用基于统计的模型的方法明显优于使用基于概率的模型的方法.

在 2018 年, Shin 等人[14]提出了结合了基于变异和多样性感知的测试用例优先排序方法. 基于多样性感知变异的技术依赖于变异体区分的概念, 即区分一个变异体与另一个变异体的行为, 而不是与原始程序区分. 他们的研究调查了两种基于变异的优先排序技术 (一种使用传统的变异体杀死, 另一种使用变异体区分) 的成本和有效性, 在单目标 (贪心和混合) 和多目标优先排序方案下研究了这两种不同的基于变异的测试用例优先排序技术. 结果表明, 传统的变异体杀死和变异体区分技术之间没有更占优势的技术.

6.3.3 测试用例优先排序相关研究

在 2006 年, Do 等人[122]的分析表明, 测试用例优先排序可以显著提高 JUnit 测试集的缺陷检测率. 在 2006 年, Do 等人[123]使用变异缺陷评估优先级技术提高测试用例优先排序缺陷检测率的能力. 结果表明, 优先排序是有效的, 但结果会随着变异缺陷的数量和测试集的缺陷检测能力而变化.

基于其他信息确定测试用例的优先级, 例如基于字符串距离的测试用例优先排序[239]. 在 2015 年, Hemmati 等人[240]研究了基于主题覆盖、基于文本多样性和风险驱动的测试用例优先排序方法, 实验表明在快速发布的背景下, 风险驱动的方法比其他方法更有效. 在 2016 年, Henard 等人[241]对几种测试用例优先排序技术进行了全面的实验比较, 包括白盒和黑盒方法. 他们发现黑盒和白盒性能之间几乎没有差异, 即最多 4% 的缺陷检测率差异, 并且黑盒和白盒检测的缺陷之间的重叠很高. 这些对于可能没有可用源代码而不适用白盒技术的回归测试人员来说是积极的发现. 他们还发现有证据表明, 黑盒和白盒测试用例优先排序方法在系统多个版本中都保持鲁棒性 (优先排序的测试集在系统的多个版本上保持性能). 在 2018 年, Khatibsyarbini 等人[242]通过系统的文献综述介绍了回归测试中的测试用例优先排序方法.

6.3.4 测试用例优先排序评价指标

缺陷检测平均百分比 (average percentage of faults detected, *APFD*), 这是评估优先排序测试集有效性的经典指标. *APFD* 计算方法如下[120]:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n},$$

其中, n 为测试用例的数量, m 为测试集检测到的缺陷的数量, TF_i 为检测缺陷 i 的第 1 个测试用例的位置.

为了区分不同测试用例的执行时间和缺陷的严重性, 可以使用成本认知缺陷检测平均百分比 (cost-cognizant average percentage, *APFDc*) 作为评估指标衡量测试用例优先排序技术的有效性.

6.4 小 结

本节介绍了测试集有效性评价在回归测试测试集优化的应用. 本节分析了基于覆盖和基于变异的测试集优化方法以及优化后的测试集的缺陷检测能力. 下面总结主要的发现.

(1) 测试集优化方法包括测试集约简、回归测试选择和测试用例优先排序, 使用覆盖信息、变异信息和测试

执行历史等信息进行测试集优化.

(2) 一些研究表明测试集约简不会显著降低测试集的缺陷检测能力[109,110], 而另一些研究表明测试集的缺陷检测能力可能会因测试集约简而受到严重影响[111−113]. 此外, 断言与基于覆盖率的测试集约简技术的有效性相关[114].

(3) 安全回归测试选择技术的召回率是 100%, 不遗漏所有能检测缺陷的测试用例.

(4) 测试用例优先排序不会丢弃测试用例, 避免了测试集约简或回归测试选择可能带来的缺陷检测能力损失. 测试用例优先排序可以提高缺陷检测速度.

7 现有挑战及研究方向

7.1 测试集有效性评价的现有挑战

从现有研究的结论来看, 存在许多不一致的结论. 因此, 当前对测试集有效性评价的研究还存在着需要解决的重要挑战. 本节从统计方法和实证研究两个方面介绍测试集有效性评价的研究中所面临的挑战.

7.1.1 统计方法中存在的挑战

(1) 高度相关的解释变量的挑战. Chen 等人[91]认为, 偏相关可能会产生虚假结果. 测试集充分性和测试集规模高度相关, 并且两者都与缺陷检测相关. 当控制两个变量即测试集充分性和测试集规模之一时, 观察到的缺陷检测与另一个变量的相关性必然会更弱. 他们认为, 研究测试集充分性和测试集规模的相对重要性的更原则性的数据分析方法是采用多元回归并使用回归系数等效应大小度量. 因此, 不合理的数据分析方法可能导致产生错误的结果.

(2) 解释皮尔逊相关系数的挑战. Chen 等人[91]认为, 皮尔逊相关被广泛用于表示两个变量之间的关系, 但是解释所得相关系数仅适用于皮尔逊相关系数范围从−1 到+1 的情况. 考虑一个连续变量 (测试集充分性) 和一个二分变量 (缺陷检测) 之间的皮尔逊相关性. 皮尔逊相关的这种特殊情况也称为点二序列相关 (point biserial correlation). 这种相关系数最多为 0.8, 其范围可以更小. 因此, 点二序列相关是有界的, 对皮尔逊相关系数的解释可能错误地得出两个完全相关的变量不是强相关的结论.

7.1.2 实证研究中存在的挑战

(1) 测试用例来源的挑战. 在测试集有效性评价的研究中测试用例来源的挑战会影响评估结果. 测试用例可以通过手动编写和自动生成构建. 自动生成的测试用例与手动编写的测试用例相比如何是一个开放的研究问题. 现有的研究[202−204]表明, 与手动编写测试用例相比, 自动生成测试用例可以明显提高代码覆盖率和变异得分, 然而实际发现的缺陷数量并没有明显改善. 因此, 在使用自动生成的测试用例时需要考虑生成的测试用例的质量对研究结果的影响.

(2) 测试集构造的挑战. 测试集可以通过测试选择或测试集约简构造. 1) 测试选择. 通过随机选择项目的原始测试集的测试用例构建的测试集会影响测试集有效性评价研究的结果. Zhang 等人[96]的结果表明随机选择构建的测试集的覆盖率与变异缺陷检测之间的相关性远高于原始测试集. Chen 等人[91]认为随机选择方法存在缺陷, 类不平衡问题导致了先前工作中测试集充分性和缺陷检测之间的相关性的错误结论. 2) 测试集约简. 一些研究表明测试集的缺陷检测能力可能会因测试用例集约简而受到严重影响[111−113]. Chen 等人[91]的结果表明, 大多数基于变异约简的测试集的缺陷检测能力能够保持, 而基于覆盖约简的测试集的缺陷检测能力显著丧失. Staats 等人[52]的结果表明, 直接生成满足分支覆盖和 MC/DC 覆盖准则的测试集的有效性低于随机生成的相同规模的测试集;约简随机生成的测试集以满足分支覆盖和 MC/DC 覆盖比随机生成与约简测试集相同规模的测试集更有效.

(3) 满足测试充分性准则的挑战. 覆盖准则衡量给定测试集达到的覆盖率, 满足 100% 覆盖率的测试集是充分的. 然而, 在许多实际情况下, 例如代码中的不可达路径[243], 充分的测试集是不切实际的, 甚至是不可能的. Gligoric 等人[48]提出了第一项评估不充分测试集的覆盖准则的广泛研究. 此外, 由于问题固有的不可判定性, 生成变异充分即杀死所有可杀死变异体的测试集实际上是不可行的[133,244]. Frankl 等人[245]建议只执行可执行的定义-使用关联, 即排除不可达的路径, 因此在可执行路径已知的情况下, 实现完全覆盖 (覆盖率达到 100%) 成为可能. 测试集不充分是指测试集不会达到 100% 的覆盖率或 100% 的变异得分, 例如 Inozemtseva 等人[89]使用的 5 个项目

Apache POI、Closure、HSQLDB、JFreeChart 和 Joda Time 的测试集的语句覆盖率分别为 67%、76%、27%、54% 和 91%. 使用不充分的测试集可能会影响结果[217,219,246]. 因此, 在测试集有效性评价的研究中需要考虑不充分的测试集对研究结果的影响, 尽可能覆盖可执行路径.

(4) 干净程序假设的挑战. 许多先前的研究都依赖于干净程序假设 (clean program assumption, CPA), 即测试集对缺陷和修复程序版本达到相似的覆盖率. CPA 假设缺陷对程序行为的影响很小, 然而, 因为代码不同, 缺陷程序和干净程序可能有许多不同的程序结构. 测试评估的一种常见的做法是在原始程序上应用工具和测试技术, 然后通过对变异体执行测试检查测试集的缺陷检测能力. 这种做法隐含了 CPA, 即假设原始程序上的覆盖率代表变异体上的覆盖率. 这种做法可能有问题, 因为测试集是在变异体上评估的, 而不是在应用测试集的原始程序上. Chekam 等人[59]证明 CPA 并不总是成立, 从而对先前结果的有效性造成了严重威胁. 该研究还表明, 如果不进行实验控制, CPA 有可能改变实证研究的结果.

(5) 影响因素的挑战. 目前的研究表明有多种因素影响测试集的有效性, 测试集规模、缺陷种类、测试预言、程序结构和每个测试覆盖的方法数量等都被认为对测试集有效性有影响. 在测试集有效性评价的研究中存在影响因素的挑战. 因此, 在研究代码覆盖率和变异得分与缺陷检测有效性的关系时需要考虑混杂变量的混杂效应, 然后移除混杂效应, 例如控制测试集规模等影响测试集有效性的因素.

(6) 缺陷类型的挑战. 在测试集有效性评价的研究中存在注入的缺陷即手动注入的缺陷和变异缺陷的挑战. 现有的研究表明, 注入的缺陷不同于真实缺陷, 因此可能不适合评估测试技术[29,124]. 另一方面, 使用真实缺陷可以使实证研究结果具有可比性、可重复性. 因此, 应该尽可能地使用真实缺陷评估测试集有效性, 或者使变异缺陷接近真实缺陷[208].

(7) 工具的挑战. 在测试集有效性评价的研究中存在所使用工具的挑战. 许多研究应用了自动化工具, 在使用自动化工具时, 工具的缺陷可能导致结果不准确. 例如, Kintis 等人[217]发现 4 种 Java 变异测试工具的缺陷检测能力之间存在很大差异. Zhu 等人[247]检查了 3 个为 Java 语言开发的回归测试选择工具, 并在这 3 个工具中发现了 27 个错误. 因此, 研究人员在使用自动化工具时需要重新评估所使用工具的有效性.

(8) 面向领域软件的测试充分性准则和测试集有效性评价的挑战. 在深度神经网络测试中, 尽管有多种神经网络结构覆盖准则被提出, 但多数研究表明, 这些覆盖准则与缺陷检测能力既不正相关也不强相关, 增加神经元覆盖率并不能提高覆盖准则的缺陷检测能力[73−77]. 在 Web 服务测试、物联网系统测试和模型驱动测试中, 尽管定义了多种特定领域的测试充分性准则, 但是对测试充分性准则有效性评价的研究较少[66,79,82]. 一些研究仅通过简单的实验分析了提出的充分性准则与测试集缺陷检测有效性的相关性[62,68,84], 缺乏大规模实证研究.

7.2 测试集有效性评价的研究方向

针对测试集有效性评价现有研究中的结论以及存在的挑战, 我们总结了以下几个方面的可能需要得到研究人员重点关注的研究方向.

(1) 解决结论中的矛盾. 现有的研究得出了许多相互矛盾的结论, 如表 6 所示. 这是由于方法和实验过程不统一造成的. 因此, 在今后的研究中, 我们推荐研究人员充分考虑理论和实证研究中的挑战, 了解结论相互矛盾的研究的局限, 在此基础上提供统一的方法, 得出一致的结论. 例如 Chen 等人[91]提出了一种公平的比较测试充分性准则的方法.

(2) 研究通过随机选择构建的测试集是否对结论造成威胁. 测试集的构建可能直接影响研究结果. 目前较少的研究探讨了这种威胁的存在以及这种威胁的严重性. Lam 等人[248]的结果表明, 将回归测试技术增强为顺序依赖 (即在以一种顺序运行时通过但在以另一个顺序运行时失败的测试) 可以大大减少不稳定测试 (flaky tests)[249]失败. 随机选择可能破坏原始测试集的顺序依赖. 我们推荐研究人员研究通过随机选择构建的测试集对研究结果的影响, 并研究通过随机选择构建的测试集与缺陷检测有效性的关系. 这将有助于验证现有研究的研究结果.

(3) 研究更强的测试充分性准则. 现有的研究表明一些广泛使用的基于覆盖的测试充分性准则以及基于变异的测试充分性准则与缺陷检测能力的相关性不一致. Whalen 等人[102]提出了 Observable MC/DC (OMC/DC) 覆盖

准则, 将 MC/DC 覆盖与可观察性结合. 他们的结果表明满足 OMC/DC 的测试集比满足 MC/DC 的测试集检测到更多的缺陷. Shin 等人[104]提出了多样性感知的变异充分性准则, 该准则比传统变异充分性准则有效 8.26 倍. Aghamohammadi 等人[105]提出了语句频率覆盖准则, 将执行语句的频率纳入语句覆盖评估测试集的有效性. 结果表明语句频率覆盖优于语句覆盖和分支覆盖准则. 研究更强的测试充分性准则可以提高测试的有效性并且适用于多种测试方法.

(4) 研究其他影响测试集有效性的因素. 目前对测试集有效性的影响因素包括 3 个方面: 一是测试用例自身, 包括测试气味、每个测试覆盖的方法数量; 二是测试集, 包括测试集规模、测试预言; 三是被测软件, 包括缺陷种类、程序结构. 改进的测试充分性准则结合的其他因素, 包括可观察性、多样性和执行语句的频率, 能够提高这些准则的缺陷检测有效性, 这些因素可能会影响有效性. 我们推荐研究人员研究是否存在其他的测试集有效性的影响因素以及如何影响测试集的有效性. 这将有助于明确不同的测试集有效性的影响因素.

(5) 研究代码覆盖率和变异得分与缺陷检测有效性的关系. 现有研究表明了代码覆盖率或变异得分与缺陷检测有效性的关系. 例如 Frankl 等人[26]的结果表明, 固定规模的测试集的覆盖率 (all-uses 覆盖和变异覆盖) 和有效性之间的关系, 发现它是非线性的, 在许多情况下是非单调的. Namin 等人[88]的实验表明, 测试集的规模、覆盖率 (块覆盖、判定覆盖、c-use 覆盖和 p-use 覆盖) 和有效性这 3 个变量之间不存在线性关系, 但存在非线性关系. Papadakis 等人[29]的结果表明, 缺陷检测和变异得分之间的关系是非线性关系. 根据本文的回顾, 包括测试集规模在内的多种因素可以影响测试集的有效性. 我们推荐研究人员在移除其他因素的影响后研究代码覆盖率和变异得分与缺陷检测有效性的关系. 这将有助于了解代码覆盖率和变异得分与缺陷检测有效性之间的真实相关关系.

(6) 预测测试集有效性. 基于代码覆盖率或变异得分可以预测测试集有效性 (代码覆盖率和变异得分为自变量, 缺陷检测有效性为因变量). 目前有多种测试集有效性的影响因素, 根据这些因素与自变量和因变量的关系可以构建测试集有效性的预测模型, 这种方法是轻量的. 例如 Grano 等人[86]研究了 5 个维度的因素 (语句覆盖率、测试气味、生产和测试代码度量、代码气味和可读性) 与测试用例有效性之间的关系. 此外, 在预测性变异测试中, 预测在不执行变异体的情况下变异体是否会被杀死[142,143]; 在测试用例集优化中, 使用测试历史构建的模型预测测试结果[227,233], 关于更可能检测到缺陷的测试用例特征的先验知识对于测试用例优先排序是有帮助的. 预测测试集有效性可以节省测试时间和资源.

(7) 研究测试集有效性的影响因素如何影响测试集优化. 目前许多测试集优化技术建立在代码覆盖率和/或变异得分的基础上. 根据本文的回顾, 有多种因素影响测试集的有效性. 这些因素如何影响基于覆盖或基于变异的测试集优化的研究较少. Chen 等人[114]的实验结果表明, 断言覆盖率和断言计数影响基于覆盖的测试用例集约简. 这类研究将有助于分析和处理测试集优化中的混杂效应.

(8) 研究类不平衡对测试集有效性评价的影响. 根据测试中的帕累托法则[250], 即大约 80% 的错误来自 20% 的代码. Chen 等人[91]认为类不平衡问题导致了通过随机选择构建测试集的工作中测试集充分性和缺陷检测之间相关性的错误结论. 此外, Xue 等人[251]提出了一个机器学习引导的智能合约 (smart contract) 模糊测试框架 xFuzz. xFuzz 首先利用机器学习模型识别最有可能受到攻击的函数, 然后应用模糊测试技术测试识别的容易出错的代码. 他们的结果证实了 xFuzz 在检测跨合约漏洞方面的有效性. Kim 等人[24]建议进一步探索缺陷预测技术, 以增强用于测试的变异体的选择. Xue 等人[251]的研究是在缺陷定位的情况下进行的, 而绝大多数对测试集有效性评价的研究是在没有缺陷定位的情况下进行的. 我们推荐研究人员在缺陷定位以后研究覆盖率和变异得分与缺陷检测有效性的关系, 并分析与在没有缺陷定位的情况下的研究结果的差异. 这将有助于明确类不平衡对测试集有效性评价的影响.

(9) 研究面向领域软件的测试充分性准则和测试集有效性评价. 在深度神经网络测试中, 现有研究表明深度神经网络覆盖准则与缺陷检测能力既不正相关也不强相关[73−77], 因此, 需要设计新的覆盖准则, 并对所提出的覆盖准则进行实证研究, 该覆盖准则能够与缺陷检测能力强正相关. 在 Web 服务测试、物联网系统测试和模型驱动测试中, 目前对测试充分性准则评价的研究有限[66,79,82], 需要进行大规模实证研究, 以比较不同充分性准则的有效性以及分析基于不同准则构建的测试集与其缺陷检测有效性的相关性.

8 结束语

软件测试在确保系统可靠性方面发挥着重要作用. 评估测试集的有效性是一项具有挑战性的任务. 本文介绍并分析了基于覆盖率和基于变异得分的测试集有效性评价的相关研究, 并介绍了测试集有效性评价的应用. 本文的主要工作总结如下: (1) 总结了基于覆盖率和基于变异得分的评价; (2) 回顾了几类新兴领域软件的测试充分性准则和测试集有效性评价的研究; (3) 总结了包括测试集规模在内的测试集有效性的影响因素; (4) 分析了测试集有效性评价研究不同层面的实验设置, 包括测试用例来源、测试集构造方式、测试充分性准则、统计方法和缺陷类型; (5) 总结了测试集有效性评价在回归测试测试集优化中的应用; (6) 总结了测试集有效性评价研究中得出的各种矛盾的结论; (7) 从统计方法和实证研究的角度总结了当前测试集有效性评价的研究所面临的挑战并从 9 个方面指出了未来需要重点关注的研究方向.

References:

[1] Myers GJ, Sandler C, Badgett T. The Art of Software Testing. 3rd ed., New York: John Wiley & Sons, 2011.

[2] Zhang CY, Yan YC, Zhou HR, Yao YB, Wu K, Su T, Miao WK, Pu GG. Smartunit: Empirical evaluations for automated unit testing of embedded software in industry. In: Proc. of the 40th Int'l Conf. on Software Engineering: Software Engineering in Practice. Gothenburg: ACM, 2018. 296–305. [doi: 10.1145/3183519.3183554]

[3] Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. IEEE Trans. on Software Engineering, 2019, 45(5): 489–506. [doi: 10.1109/TSE.2017.2785841]

[4] Wang YH, Jia XK, Liu YW, Zeng K, Bao T, Wu DH, Su PR. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In: Proc. of the 27th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2020. [doi: 10.14722/ndss.2020.24422]

[5] Schuler D, Zeller A. Covering and uncovering equivalent mutants. Software Testing, Verification and Reliability, 2013, 23(5): 353–374. [doi: 10.1002/stvr.1473]

[6] Di Nardo D, Alshahwan N, Briand L, Labiche Y. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. Software Testing, Verification and Reliability, 2015, 25(4): 371–396. [doi: 10.1002/stvr.1572]

[7] Santelices R, Harrold MJ. Applying aggressive propagation-based strategies for testing changes. In: Proc. of the 4th IEEE Int'l Conf. on Software Testing, Verification and Validation. Berlin: IEEE, 2011. 11–20. [doi: 10.1109/ICST.2011.46]

[8] Ryder BG, Tip F. Change impact analysis for object-oriented programs. In: Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. Snowbird: ACM, 2001. 46–53. [doi: 10.1145/379605.379661]

[9] Al-Ahmad B. Using code coverage metrics for improving software defect prediction. Journal of Software, 2018, 13(12): 654–674. [doi: 10.17706/jsw.13.12.654-674]

[10] Xue XZ, Siami-Namini S, Namin AS. Assessing the effectiveness of coverage-based fault localizations using mutants. Int'l Journal of Software Engineering and Knowledge Engineering, 2018, 28(8): 1091–1119. [doi: 10.1142/S0218194018500316]

[11] Jatana N, Suri B. An improved crow search algorithm for test data generation using search-based mutation testing. Neural Processing Letters, 2020, 52(1): 767–784. [doi: 10.1007/s11063-020-10288-7]

[12] Shi A, Gyori A, Gligoric M, Zaytsev A, Marinov D. Balancing trade-offs in test-suite reduction. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 246–256. [doi: 10.1145/2635868.2635921]

[13] Lou YL, Hao D, Zhang L. Mutation-based test-case prioritization in software evolution. In: Proc. of the 26th IEEE Int'l Symp. on Software Reliability Engineering. Gaithersbury: IEEE, 2015. 46–57. [doi: 10.1109/ISSRE.2015.7381798]

[14] Shin D, Yoo S, Papadakis M, Bae DH. Empirical evaluation of mutation-based test case prioritization techniques. Software Testing, Verification and Reliability, 2019, 29(1–2): e1695. [doi: 10.1002/stvr.1695]

[15] Bowes D, Hall T, Harman M, Jia Y, Sarro F, Wu F. Mutation-aware fault prediction. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 330–341. [doi: 10.1145/2931037.2931039]

[16] Shen WJ, Li YH, Han YL, Chen L, Wu D, Zhou YM, Xu BW. Boundary sampling to boost mutation testing for deep learning models. Information and Software Technology, 2021, 130: 106413. [doi: 10.1016/j.infsof.2020.106413]

[17] Zhu QQ, Panichella A, Zaidman A. A systematic literature review of how mutation testing supports quality assurance processes. Software Testing, Verification and Reliability, 2018, 28(6): e1675. [doi: 10.1002/stvr.1675]

[18] Gopinath R, Jensen C, Groce A. Code coverage for suite evaluation by developers. In: Proc. of the 36th Int'l Conf. on Software

Engineering. Hyderabad: ACM, 2014. 72–82. [doi: 10.1145/2568225.2568278]

[19] Gligoric M, Groce A, Zhang CQ, Sharma R, Alipour MA, Marinov D. Guidelines for coverage-based comparisons of non-adequate test suites. ACM Trans. on Software Engineering and Methodology, 2015, 24(4): 1–33. [doi: 10.1145/2660767]

[20] Gay G. The fitness function for the job: Search-based generation of test suites that detect real faults. In: Proc. of the 2017 IEEE Int'l Conf. on Software Testing, Verification and Validation. Tokyo: IEEE, 2017. 345–355. [doi: 10.1109/ICST.2017.38]

[21] Briand LC, Pfahl D. Using simulation for assessing the real impact of test coverage on defect coverage. In: Proc. of the 10th Int'l Symp. on Software Reliability Engineering. Boca Raton: IEEE, 1999. 148–157. [doi: 10.1109/ISSRE.1999.809319]

[22] Gay G, Staats M, Whalen M, Heimdahl MPE. The risks of coverage-directed test case generation. IEEE Trans. on Software Engineering, 2015, 41(8): 803–819. [doi: 10.1109/TSE.2015.2421011]

[23] Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 654–665. [doi: 10.1145/2635868.2635929]

[24] Kim M, Kim N, In HP. Investigating the relationship between mutants and real faults with respect to mutated code. Int'l Journal of Software Engineering and Knowledge Engineering, 2020, 30(8): 1119–1137. [doi: 10.1142/S021819402050028X]

[25] 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]

[26] 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]

[27] Baker R, Habli I. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Trans. on Software Engineering, 2013, 39(6): 787–805. [doi: 10.1109/TSE.2012.56]

[28] Ahmed I, Gopinath R, Brindescu C, Groce A, Jensen C. Can testedness be effectively measured? In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 547–558. [doi: 10.1145/2950290.2950324]

[29] Papadakis M, Shin D, Yoo S, Bae DH. Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 537–548. [doi: 10.1145/3180155.3180183]

[30] Hariri F, Shi A, Fernando V, Mahmood S, Marinov D. Comparing mutation testing at the levels of source code and compiler intermediate representation. In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification. Xi'an: IEEE, 2019. 114–124. [doi: 10.1109/ICST.2019.00021]

[31] Fraser G, Arcuri A. EvoSuite: Automatic test suite generation for object-oriented software. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 416–419. [doi: 10.1145/2025113.2025179]

[32] Böhme M. STADS: Software testing as species discovery. ACM Trans. on Software Engineering and Methodology, 2018, 27(2): 7. [doi: 10.1145/3210309]

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

[34] Chilenski JJ, Miller SP. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 1994, 9(5): 193–200. [doi: 10.1049/sej.1994.0025]

[35] 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]

[36] Hamlet RG. Testing programs with the aid of a compiler. IEEE Trans. on Software Engineering, 1977, SE-3(4): 279–290. [doi: 10.1109/TSE.1977.231145]

[37] Schuler D, Zeller A. Assessing oracle quality with checked coverage. In: Proc. of the 4th IEEE Int'l Conf. on Software Testing, Verification and Validation. Berlin: IEEE, 2011. 90–99. [doi: 10.1109/ICST.2011.32]

[38] Frankl PG, Weiss SN. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In: Proc. of the 1991 Symp. on Testing, Analysis, and Verification. Victoria British: ACM, 1991. 154–164. [doi: 10.1145/120807.120821]

[39] Frankl PG, Weiss SN. An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Trans. on Software Engineering, 1993, 19(8): 774–787. [doi: 10.1109/32.238581]

[40] Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proc. of the 16th Int'l Conf. on Software Engineering. Sorrento: IEEE, 1994. 191–200. [doi: 10.1109/ICSE.1994.296778]

[41] Frankl PG, Iakounenko O. Further empirical studies of test effectiveness. In: Proc. of the 6th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Lake Buena: ACM, 1998. 153–162. [doi: 10.1145/288195.288298]

[42] Chen MH, Lyu MR, Wong WE. Effect of code coverage on software reliability measurement. IEEE Trans. on Reliability, 2001, 50(2): 165–170. [doi: 10.1109/24.963124]

[43] Cai X, Lyu MR. The effect of code coverage on fault detection under different testing profiles. In: Proc. of the 1st Int'l Workshop on Advances in Model-based Testing. St. Louis: ACM, 2005. 1–7. [doi: 10.1145/1083274.1083288]

[44] 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): 608–624. [doi: 10.1109/TSE.2006.83]

[45] Gupta A, Jalote P. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. Int'l Journal on Software Tools for Technology Transfer, 2008, 10(2): 145–160. [doi: 10.1007/s10009-007-0059-5]

[46] Wei Y, Meyer B, Oriol M. Is branch coverage a good measure of testing effectiveness? In: Proc. of the 2012 LASER Summer School on Software Engineering. Elba Island: Springer, 2012. 194–212. [doi: 10.1007/978-3-642-25231-0_5]

[47] Hassan MM, Andrews JH. Comparing multi-point stride coverage and dataflow coverage. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 172–181. [doi: 10.1109/ICSE.2013.6606563]

[48] Gligoric M, Groce A, Zhang CQ, Sharma R, Alipour MA, Marinov D. Comparing non-adequate test suites using coverage criteria. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. Lugano: ACM, 2013. 302–313. [doi: 10.1145/2483760.2483769]

[49] Hong S, Staats M, Ahn J, Kim M, Rothermel G. The impact of concurrent coverage metrics on testing effectiveness. In: Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation. Luxembourg: IEEE, 2013. 232–241. [doi: 10.1109/ICST.2013.32]

[50] Hemmati H. How effective are code coverage criteria? In: Proc. of the 2015 IEEE Int'l Conf. on Software Quality, Reliability and Security. Vancouver: IEEE, 2015. 151–156. [doi: 10.1109/QRS.2015.30]

[51] Cheng YF, Wang M, Xiong YF, Hao D, Zhang L. Empirical evaluation of test coverage for functional programs. In: Proc. of the 2016 IEEE Int'l Conf. on Software Testing, Verification and Validation. Chicago: IEEE, 2016. 255–265. [doi: 10.1109/ICST.2016.8]

[52] Staats M, Gay G, Whalen M, Heimdahl M. On the danger of coverage directed test case generation. In: Proc. of the 15th Int'l Conf. on Fundamental Approaches to Software Engineering. Tallinn: Springer, 2012. 409–424. [doi: 10.1007/978-3-642-28872-2_28]

[53] Kochhar PS, Lo D, Lawall J, Nagappan N. Code coverage and postrelease defects: A large-scale study on open source projects. IEEE Trans. on Reliability, 2017, 66(4): 1213–1228. [doi: 10.1109/TR.2017.2727062]

[54] Durelli VHS, Delamaro ME, Offutt J. An experimental comparison of edge, edge-pair, and prime path criteria. Science of Computer Programming, 2018, 152: 99–115. [doi: 10.1016/j.scico.2017.10.003]

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

[56] Offutt AJ, Voas JM. Subsumption of condition coverage techniques by mutation testing. Technical Report, ISSE-TR-96-01, Fairfax: George Mason University, 1996.

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

[58] Wong WE, Mathur AP. Fault detection effectiveness of mutation and data flow testing. Software Quality Journal, 1995, 4(1): 69–83. [doi: 10.1007/BF00404650]

[59] Chekam TT, Papadakis M, Le Traon Y, Harman M. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 597–608. [doi: 10.1109/ICSE.2017.61]

[60] Perretta J, DeOrio A, Guha A, Bell J. On the use of mutation analysis for evaluating student test suite quality. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 263–275. [doi: 10.1145/3533767.3534217]

[61] Alshahwan N, Harman M. Augmenting test suites effectiveness by increasing output diversity. In: Proc. of the 34th Int'l Conf. on Software Engineering. Zurich: IEEE, 2012. 1345–1348. [doi: 10.1109/ICSE.2012.6227083]

[62] Mei LJ, Chan WK, Tse TH. Data flow testing of service-oriented workflow applications. In: Proc. of the 30th Int'l Conf. on Software Engineering. Leipzig: IEEE, 2008. 371–380. [doi: 10.1145/1368088.1368139]

[63] Mei LJ, Chan WK, Tse TH. Data flow testing of service choreography. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Amsterdam: ACM, 2009. 151–160. [doi: 10.1145/1595696.1595720]

[64] Siblini R, Mansour N. Testing Web services. In: Proc. of the 3rd ACS/IEEE Int'l Conf. on Computer Systems and Applications. Cairo: IEEE, 2005. 135. [doi: 10.1109/AICCSA.2005.1387124]

[65] Domínguez-Jiménez JJ, Estero-Botaro A, García-Domínguez A, Medina-Bulo I. GAmera: An automatic mutant generation system for WS-BPEL compositions. In: Proc. of the 7th IEEE European Conf. on Web Services. Eindhoven: IEEE, 2009. 97–106. [doi: 10.1109/

ECOWS.2009.18]

[66] Mei LJ, Chan WK, Tse TH, Kuo FC. An empirical study of the use of Frankl-Weyuker data flow testing criteria to test BPEL Web services. In: Proc. of the 33rd Annual IEEE Int'l Computer Software and Applications Conf. Seattle: IEEE, 2009. 81–88. [doi: 10.1109/COMPSAC.2009.21]

[67] Lai ZF, Cheung SC, Chan WK. Inter-context control-flow and data-flow test adequacy criteria for nesC applications. In: Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Atlanta: ACM, 2008. 94–104. [doi: 10.1145/1453101.1453115]

[68] Leotta M, Clerissi D, Olianas D, Ricca F, Ancona D, Delzanno G, Franceschini L, Ribaudo M. An acceptance testing approach for Internet of Things systems. IET Software, 2018, 12(5): 430–436. [doi: 10.1049/iet-sen.2017.0344]

[69] Pei KX, Cao YZ, Yang JF, Jana S. DeepXplore: Automated whitebox testing of deep learning systems. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 1–18. [doi: 10.1145/3132747.3132785]

[70] Ma L, Zhang FY, Sun JY, Xue MH, Li B, Juefei-Xu F, Xie C, Li L, Liu Y, Zhao JJ, Wang YD. DeepMutation: Mutation testing of deep learning systems. In: Proc. of the 29th IEEE Int'l Symp. on Software Reliability Engineering. Memphis: IEEE, 2018. 100–111. [doi: 10.1109/ISSRE.2018.00021]

[71] Hu Q, Ma L, Xie XF, Yu B, Liu Y, Zhao JJ. DeepMutation++: A mutation testing framework for deep learning systems. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 1158–1161. [doi: 10.1109/ASE.2019.00126]

[72] Li ZN, Ma XX, Xu C, Cao C. Structural coverage criteria for neural networks could be misleading. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering: New Ideas and Emerging Results. Montreal: IEEE, 2019. 89–92. [doi: 10.1109/ICSE-NIER.2019.00031]

[73] Yan SA, Tao GH, Liu XW, Zhai J, Ma SQ, Xu L, Zhang XY. Correlations between deep neural network model coverage criteria and model quality. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 775–787. [doi: 10.1145/3368089.3409671]

[74] Yang Z, Shi JK, Asyrofi MH, Lo D. Revisiting neuron coverage metrics and quality of deep neural networks. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Honolulu: IEEE, 2022. 408–419. [doi: 10.1109/SANER53432.2022.00056]

[75] Harel-Canada F, Wang LX, Gulzar MA, Gu QQ, Kim M. Is neuron coverage a meaningful measure for testing deep neural networks? In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 851–862. [doi: 10.5281/zenodo.4021473]

[76] Dong YZ, Zhang PX, Wang JY, Liu S, Sun J, Hao JY, Wang XY, Wang L, Dong JS, Dai T. An empirical study on correlation between coverage and robustness for deep neural networks. In: Proc. of the 25th Int'l Conf. on Engineering of Complex Computer Systems. Singapore: IEEE, 2020. 73–82. [doi: 10.1109/ICECCS51672.2020.00016]

[77] Sun WD, Lu YT, Sun M. Are coverage criteria meaningful metrics for DNNs? In: Proc. of the 2021 Int'l Joint Conf. on Neural Networks. Shenzhen: IEEE, 2021. 1–8. [doi: 10.1109/IJCNN52387.2021.9533987]

[78] Andrews A, France R, Ghosh S, Craig G. Test adequacy criteria for UML design models. Software Testing, Verification and Reliability, 2003, 13(2): 95–127. [doi: 10.1002/stvr.270]

[79] Rountev A, Kagan S, Sawin J. Coverage criteria for testing of object interactions in sequence diagrams. In: Proc. of the 8th Fundamental Approaches to Software Engineering. Edinburgh: Springer. 2005. 289–304. [doi: 10.1007/978-3-540-31984-9_22]

[80] Ali S, Briand LC, Rehman M JU, Asghar H, Iqbal MZZ, Nadeem A. A state-based approach to integration testing based on UML models. Information and Software Technology, 2007, 49(11–12): 1087–1106. [doi: 10.1016/j.infsof.2006.11.002]

[81] Offutt J, Abdurazik A. Generating tests from UML specifications. In: Proc. of the 2nd Int'l Conf. on the Unified Modeling Language. Fort Collins: Springer, 1999. 416–429. [doi: 10.1007/3-540-46852-8_30]

[82] Briand LC, Labiche Y, Wang Y. Using simulation to empirically investigate test coverage criteria based on statechart. In: Proc. of the 26th Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 86–95. [doi: 10.1109/ICSE.2004.1317431]

[83] Samuel P, Mall R. Slicing-based test case generation from UML activity diagrams. ACM SIGSOFT Software Engineering Notes, 2009, 34(6): 1–14. [doi: 10.1145/1640162.1666579]

[84] Sun HY, Chen MS, Zhang M, Liu J, Zhang Y. Improving defect detection ability of derived test cases based on mutated UML activity diagrams. In: Proc. of the 40th IEEE Annual Computer Software and Applications Conf. Atlanta: IEEE, 2016. 275–280. [doi: 10.1109/COMPSAC.2016.136]

[85] Spadini, Palomba F, Zaidman A, Bruntink M, Bacchelli A. On the relation of test smells to software code quality. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution. Madrid: IEEE, 2018. 1–12. [doi: 10.1109/ICSME.2018.00010]

[86] Grano G, Palomba F, Gall HC. Lightweight assessment of test-case effectiveness using source-code-quality indicators. IEEE Trans. on Software Engineering, 2021, 47(4): 758–774. [doi: 10.1109/TSE.2019.2903057]

[87] Wong WE, Horgan JR, London S, Mathur AP. Effect of test set size and block coverage on the fault detection effectiveness. In: Proc. of the 1994 IEEE Int'l Symp. on Software Reliability Engineering. Monterey: IEEE, 1994. 230–238. [doi: 10.1109/ISSRE.1994.341379]

[88] Namin AS, Andrews JH. The influence of size and coverage on test suite effectiveness. In: Proc. of the 18th Int'l Symp. on Software Testing and Analysis. Chicago: ACM, 2009. 57–68. [doi: 10.1145/1572272.1572280]

[89] Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 435–445. [doi: 10.1145/2568225.2568271]

[90] Kochhar PS, Thung F, Lo D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. Montreal: IEEE, 2015. 560–564. [doi: 10.1109/SANER.2015.7081877]

[91] Chen YT, Gopinath R, Tadakamalla A, Ernst MD, Holmes R, Fraser G, Ammann P, Just R. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 237–249. [doi: 10.1145/3324884.3416667]

[92] Schwartz A, Hetzel M. The impact of fault type on the relationship between code coverage and fault detection. In: Proc. of the 11th Int'l Workshop on Automation of Software Test. Austin: ACM, 2016. 29–35. [doi: 10.1145/2896921.2896926]

[93] Schwartz A, Puckett D, Meng Y, Gay G. Investigating faults missed by test suites achieving high code coverage. Journal of Systems and Software, 2018, 144: 106–120. [doi: 10.1016/j.jss.2018.06.024]

[94] Petrić J, Hall T, Bowes D. Which software faults are tests not detecting? In: Proc. of the 2020 Evaluation and Assessment in Software Engineering. Trondheim: ACM, 2020. 160–169. [doi: 10.1145/3383219.3383236]

[95] Zhang YC, Mesbah A. Assertions are strongly correlated with test suite effectiveness. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 214–224. [doi: 10.1145/2786805.2786858]

[96] Zhang JM, Zhang LM, Hao D, Wang M, Zhang L. Do pseudo test suites lead to inflated correlation in measuring test effectiveness? In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification. Xi'an: IEEE, 2019. 252–263. [doi: 10.1109/ICST.2019.00033]

[97] Rajan A, Whalen MW, Heimdahl MPE. The effect of program and model structure on MC/DC test adequacy coverage. In: Proc. of the 30th Int'l Conf. on Software Engineering. Leipzig: IEEE, 2008. 161–170. [doi: 10.1145/1368088.1368111]

[98] Heimdahl MPE, Whalen MW, Rajan A, Staats M. On MC/DC and implementation structure: An empirical study. In: Proc. of the 27th IEEE/AIAA Digital Avionics Systems Conf. St. Paul: IEEE, 2008. 5.B.3-1–5.B.3-13. [doi: 10.1109/DASC.2008.4702848]

[99] Gay G, Rajan A, Staats M, Whalen M, Heimdahl MPE. The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. ACM Trans. on Software Engineering and Methodology, 2016, 25(3): 1–34. [doi: 10.1145/2934672]

[100] Byun T, Sharma V, Rayadurgam S, McCamant S, Heimdahl MPE. Toward rigorous object-code coverage criteria. In: Proc. of the 28th IEEE Int'l Symp. on Software Reliability Engineering. Toulouse: IEEE, 2017. 328–338. [doi: 10.1109/ISSRE.2017.33]

[101] Petrić J, Hall T, Bowes D. How effectively is defective code actually tested?: An analysis of junit tests in seven open source systems. In: Proc. of the 14th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. Oulu: ACM, 2018. 42–51. [doi: 10.1145/3273934.3273939]

[102] Whalen M, Gay G, You DJ, Heimdahl MPE, Staats M. Observable modified condition/decision coverage. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 102–111. [doi: 10.1109/ICSE.2013.6606556]

[103] Wang H, Chan WK, Tse TH. Improving the effectiveness of testing pervasive software via context diversity. ACM Trans. on Autonomous and Adaptive Systems, 2014, 9(2): 9. [doi: 10.1145/2620000]

[104] Shin D, Yoo S, Bae DH. A theoretical and empirical study of diversity-aware mutation adequacy criterion. IEEE Trans. on Software Engineering, 2018, 44(10): 914–931. [doi: 10.1109/TSE.2017.2732347]

[105] Aghamohammadi A, Mirian-Hosseinabadi SH, Jalali S. Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness. Information and Software Technology, 2021, 129: 106426. [doi: 10.1016/j.infsof.2020.106426]

[106] Someoliayi KE, Jalali S, Mahdieh M, Mirian-Hosseinabadi SH. Program state coverage: A test coverage metric based on executed program states. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Hangzhou: IEEE, 2019. 584–588. [doi: 10.1109/SANER.2019.8667968]

[107] Ali S, Arcaini P, Wang XY, Yue T. Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In: Proc. of the 14th IEEE Conf. on Software Testing, Verification and Validation. Porto de Galinhas: IEEE, 2021. 13–23. [doi: 10.1109/ICST49551.2021.00014]

[108] Wong WE, Horgan JR, London S, Mathur AP. Effect of test set minimization on fault detection effectiveness. Software: Practice and Experience, 1998, 28(4): 347–369. [doi: 10.1002/(SICI)1097-024X(19980410)28:4<347::AID-SPE145>3.0.CO;2-L]

[109] Wong WE, Horgan JR, Mathur AP, Pasquini A. Test set size minimization and fault detection effectiveness: A case study in a space application. In: Proc. of the 21st Annual Int'l Computer Software and Applications Conf. Washington: IEEE, 1997. 522–528. [doi: 10.1109/CMPSAC.1997.625062]

[110] Zhang LM, Marinov D, Zhang L, Khurshid S. An empirical study of junit test-suite reduction. In: Proc. of the 22nd IEEE Int'l Symp. on Software Reliability Engineering. Hiroshima: IEEE, 2011. 170–179. [doi: 10.1109/ISSRE.2011.26]

[111] Rothermel G, Harrold MJ, Ostrin J, Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proc. of the 1998 Int'l Conf. on Software Maintenance. Bethesda: IEEE, 1998. 34–43. [doi: 10.1109/ICSM.1998.738487]

[112] Rothermel G, Harrold MJ, von Ronne J, Hong C. Empirical studies of test-suite reduction. Software Testing, Verification and Reliability, 2002, 12(4): 219–249. [doi: 10.1002/stvr.256]

[113] Shi A, Gyori A, Mahmood S, Zhao PY, Marinov D. Evaluating test-suite reduction in real software evolution. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 84–94. [doi: 10.1145/3213846.3213875]

[114] Chen JJ, Bai YW, Hao D, Zhang LM, Zhang L, Xie B. How do assertions impact coverage-based test-suite reduction? In: Proc. of the 2017 IEEE Int'l Conf. on Software Testing, Verification and Validation. Tokyo: IEEE, 2017. 418–423. [doi: 10.1109/ICST.2017.45]

[115] Offutt J, Pan J, Voas JM. Procedures for reducing the size of coverage-based test sets. In: Proc. of the 12th Int'l Conf. on Testing Computer Software. New York: ACM Press, 1995. 111–123.

[116] Rothermel G, Harrold MJ. Empirical studies of a safe regression test selection technique. IEEE Trans. on Software Engineering, 1998, 24(6): 401–419. [doi: 10.1109/32.689399]

[117] Bible J, Rothermel G, Rosenblum DS. A comparative study of coarse-and fine-grained safe regression test-selection techniques. ACM Trans. on Software Engineering and Methodology, 2001, 10(2): 149–183. [doi: 10.1145/367008.367015]

[118] Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. ACM Trans. on Software Engineering and Methodology, 1997, 6(2): 173–210. [doi: 10.1145/248233.248262]

[119] Chen ZY, Duan YW, Zhao ZH, Xu BW, Qian J. Using program slicing to improve the efficiency and effectiveness of cluster test selection. Int'l Journal of Software Engineering and Knowledge Engineering, 2011, 21(6): 759–777. [doi: 10.1142/S0218194011005487]

[120] Rothermel, Untch RH, Chu CY, Harrold MJ. Prioritizing test cases for regression testing. IEEE Trans. on Software Engineering, 2001, 27(10): 929–948. [doi: 10.1109/32.962562]

[121] Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment. In: Proc. of the 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Roma: ACM, 2002. 97–106. [doi: 10.1145/566172.566187]

[122] Do H, Rothermel G, Kinneer A. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. Empirical Software Engineering, 2006, 11(1): 33–70. [doi: 10.1007/s10664-006-5965-8]

[123] 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]

[124] Gopinath R, Jensen C, Groce A. Mutations: How close are they to real faults? In: Proc. of the 25th IEEE Int'l Symp. on Software Reliability Engineering. Naples: IEEE, 2014. 189–200. [doi: 10.1109/ISSRE.2014.40]

[125] Luo Q, Moran K, Poshyvanyk D, Di Penta M. Assessing test case prioritization on real faults and mutants. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution. Madrid: IEEE, 2018. 240–251. [doi: 10.1109/ICSME.2018.00033]

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

[127] 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]

[128] Offutt AJ. Investigations of the software testing coupling effect. ACM Trans. on Software Engineering and Methodology, 1992, 1(1): 5–20. [doi: 10.1145/125489.125473]

[129] Papadakis M, Malevris N. An empirical evaluation of the first and second order mutation testing strategies. In: Proc. of the 3rd Int'l Conf. on Software Testing, Verification, and Validation Workshop. Paris: IEEE, 2010. 90–99. [doi: 10.1109/ICSTW.2010.50]

[130] Budd TA, Angluin D. Two notions of correctness and their relation to testing. Acta Informatica, 1982, 18(1): 31–45. [doi: 10.1007/BF00625279]

[131] Yao XJ, Harman M, Jia Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 919–930. [doi: 10.1145/2568225.2568265]

[132] Schuler D, Dallmeier V, Zeller A. Efficient mutation testing by checking invariant violations. In: Proc. of the 18th Int'l Symp. on

Software Testing and Analysis. Chicago: ACM, 2009. 69–80. [doi: 10.1145/1572272.1572282]

[133] Papadakis M, Jia Y, Harman M, Le Traon Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: Proc. of the 37th IEEE/ACM IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 936–946. [doi: 10.1109/ICSE.2015.103]

[134] Androutsopoulos K, Clark D, Dan HT, Hierons RM, Harman M. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 573–583. [doi: 10.1145/2568225.2568314]

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

[136] Niedermayr R, Wagner S. Is the stack distance between test case and method correlated with test effectiveness? In: Proc. of the 2019 Evaluation and Assessment on Software Engineering. Copenhagen: ACM, 2019. 189–198. [doi: 10.1145/3319008.3319021]

[137] Guizzo G, Sarro F, Krinke J, Vergilio SR. Sentinel: A hyper-heuristic for the generation of mutant reduction strategies. IEEE Trans. on Software Engineering, 2022, 48(3): 803–818. [doi: 10.1109/TSE.2020.3002496]

[138] Namin AS, Andrews JH, Murdoch DJ. Sufficient mutation operators for measuring test effectiveness. In: Proc. of the 30th Int'l Conf. on Software Engineering. Leipzig: ACM, 2008. 351–360. [doi: 10.1145/1368088.1368136]

[139] Kurtz B, Ammann P, Offutt J, Delamaro ME, Kurtz M, Gökçe N. Analyzing the validity of selective mutation with dominator mutants. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 571–582. [doi: 10.1145/2950290.2950322]

[140] Gopinath R, Alipour A, Ahmed I, Jensen C, Groce A. How hard does mutation analysis have to be, anyway? In: Proc. of 26th IEEE Int'l Symp. on Software Reliability Engineering. Gaithersbury: IEEE, 2015. 216–227. [doi: 10.1109/ISSRE.2015.7381815]

[141] Guizzo G, Sarro F, Harman M. Cost measures matter for mutation testing study validity. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1127–1139. [doi: 10.1145/3368089.3409742]

[142] Zhang J, Zhang LM, Harman M, Hao D, Jia Y, Zhang L. Predictive mutation testing. IEEE Trans. on Software Engineering, 2019, 45(9): 898–918. [doi: 10.1109/TSE.2018.2809496]

[143] Zhang P, Li YH, Ma WWY, Yang YB, Chen L, Lu HM, Zhou YM, Xu BW. CBUA: A probabilistic, predictive, and practical approach for evaluating test suite effectiveness. IEEE Trans. on Software Engineering, 2022, 48(3): 1067–1096. [doi: 10.1109/TSE.2020.3010361]

[144] Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. In: Wong WE, ed. Mutation Testing for the New Century. Boston: Springer, 2000. 34–44. [doi: 10.1007/978-1-4757-5939-6_7]

[145] DeMillo RA, Krauser EW, Mathur AP. Compiler-integrated program mutation. In: Proc. of the 15th Annual Int'l Computer Software & Applications Conf. 1991. 351–352. [doi: 10.1109/CMPSAC.1991.170202]

[146] Harman M, Jia Y, Mateo PR, Polo M. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. Vasteras: ACM, 2014. 397–408. [doi: 10.1145/2642937.2643008]

[147] Hussain. Mutation clustering [MS. Thesis]. London: Kings College London, 2008.

[148] Souza B. Identifying mutation subsumption relations. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM, 2020. 1388–1390. [doi: 10.1145/3324884.3418921]

[149] Ojdanić M, Ma W, Laurent T, Chekam TT, Ventresque A, Papadakis M. On the use of commit-relevant mutants. Empirical Software Engineering, 2022, 27(5): 114. [doi: 10.1007/s10664-022-10138-1]

[150] Pizzoleto AV, Ferrari FC, Offutt J, Fernandes L, Ribeiro M. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. Journal of Systems and Software, 2019, 157: 110388. [doi: 10.1016/j.jss.2019.07.100]

[151] Whitmore A, Agarwal A, Xu LD. The Internet of Things—A survey of topics and trends. Information Systems Frontiers, 2015, 17(2): 261–274. [doi: 10.1007/s10796-014-9489-2]

[152] Zhu SC, Yang SK, Gou XD, Xu Y, Zhang T, Wan YL. Survey of testing methods and testbed development concerning Internet of Things. Wireless Personal Communications, 2022, 123(1): 165–194. [doi: 10.1007/s11277-021-09124-5]

[153] Zhang JM, Harman M, Ma L, Liu Y. Machine learning testing: Survey, landscapes and horizons. IEEE Trans. on Software Engineering, 2022, 48(1): 1–36. [doi: 10.1109/TSE.2019.2962027]

[154] Wang Z, Yan M, Liu S, Chen JJ, Zhang DD, Wu Z, Chen X. Survey on testing of deep neural networks. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1255–1275 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5951.htm [doi: 10.13328/j.cnki.

jos.005951]

[155] Uzun B, Tekinerdogan B. Model-driven architecture based testing: A systematic literature review. Information and Software Technology, 2018, 102: 30–48. [doi: 10.1016/j.infsof.2018.05.004]

[156] Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 2012, 22(5): 297–312. [doi: 10.1002/stvr.456]

[157] Zander J, Schieferdecker I, Mosterman PJ. Model-based Testing for Embedded Systems. Boca Raton: CRC Press, 2017. [doi: 10.1201/b11321]

[158] Ahmad T, Iqbal J, Ashraf A, Truscan D, Porres I. Model-based testing using UML activity diagrams: A systematic mapping study. Computer Science Review, 2019, 33: 98–112. [doi: 10.1016/j.cosrev.2019.07.001]

[159] McQuillan JA, Power JF. A survey of UML-based coverage criteria for software testing, Department of Computer Science. Technical Report NUIM-CS-TR-2005-08, NUI Maynooth, 2005. 1–17.

[160] El Emam K, Benlarbi S, Goel N, Rai SN. The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans. on Software Engineering, 2001, 27(7): 630–650. [doi: 10.1109/32.935855]

[161] Zhou YM, Leung H, Xu BW. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. IEEE Trans. on Software Engineering, 2009, 35(5): 607–623. [doi: 10.1109/TSE.2009.32]

[162] Zhou YM, Xu BW, Leung H, Chen L. An in-depth study of the potentially confounding effect of class size in fault prediction. ACM Trans. on Software Engineering and Methodology, 2014, 23(1): 10. [doi: 10.1145/2556777]

[163] Baah GK, Podgurski A, Harrold MJ. Mitigating the confounding effects of program dependences for effective fault localization. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 146–156. [doi: 10.1145/2025113.2025136]

[164] Athanasiou D, Nugroho A, Visser J, Zaidman A. Test code quality and its relation to issue handling performance. IEEE Trans. on Software Engineering, 2014, 40(11): 1100–1125. [doi: 10.1109/TSE.2014.2342227]

[165] Fraser G, Arcuri A. Whole test suite generation. IEEE Trans. on Software Engineering, 2013, 39(2): 276–291. [doi: 10.1109/TSE.2012.14]

[166] Waterloo M, Person S, Elbaum S. Test analysis: Searching for faults in tests (N). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lincoln: IEEE, 2015. 149–154. [doi: 10.1109/ASE.2015.37]

[167] Gonzalez D, Santos JCS, Popovich A, Mirakhorli M, Nagappan M. A large-scale study on the usage of testing patterns that address maintainability attributes: Patterns for ease of modification, diagnoses, and comprehension. In: Proc. of the 14th IEEE/ACM Int'l Conf. on Mining Software Repositories. Buenos Aires: IEEE, 2017. 391–401. [doi: 10.1109/MSR.2017.8]

[168] Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software, 2007, 80(7): 1120–1128. [doi: 10.1016/j.jss.2006.10.018]

[169] Deursen AV, Moonen L, van den Bergh A, Kok G. Refactoring test code. In: Proc. of the 2nd Int'l Conf. on Extreme Programming and Flexible Processes in Software Engineering. 2001. 92–95.

[170] Meszaros G. xUnit Test Patterns: Refactoring Test Code. Pearson Education, 2007.

[171] Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: Proc. of the 28th IEEE Int'l Conf. on Software Maintenance. Trento: IEEE, 2012. 56–65. [doi: 10.1109/ICSM.2012.6405253]

[172] Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. An empirical investigation into the nature of test smells. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 4–15.

[173] Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. Are test smells really harmful? An empirical study. Empirical Software Engineering, 2015, 20(4): 1052–1094. [doi: 10.1007/s10664-014-9313-0]

[174] Van Rompaey B, Du Bois B, Demeyer S, Rieger M. On the detection of test smells: A metrics-based approach for general fixture and eager test. IEEE Trans. on Software Engineering, 2007, 33(12): 800–817. [doi: 10.1109/TSE.2007.70745]

[175] Greiler M, Zaidman A, van Deursen A, Storey MA. Strategies for avoiding text fixture smells during software evolution. In: Proc. of the 10th Working Conf. on Mining Software Repositories. San Francisco: IEEE, 2013. 387–396. [doi: 10.1109/MSR.2013.6624053]

[176] Spadini D, Schvarcbacher M, Oprescu AM, Bruntink M, Bacchelli A. Investigating severity thresholds for test smells. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. Seoul: ACM, 2020. 311–321. [doi: 10.1145/3379597.3387453]

[177] Staats M, Whalen MW, Heimdahl MPE. Programs, tests, and oracles: The foundations of testing revisited. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: ACM, 2011. 391–400. [doi: 10.1145/1985793.1985847]

[178] Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: A survey. IEEE Trans. on Software

Engineering, 2015, 41(5): 507–525. [doi: 10.1109/TSE.2014.2372785]

[179] Watson C, Tufano M, Moran K, Bavota G, Poshyvanyk D. On learning meaningful assert statements for unit test cases. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1398–1409. [doi: 10.1145/3377811.3380429]

[180] Runeson P. A survey of unit testing practices. IEEE Software, 2006, 23(4): 22–29. [doi: 10.1109/MS.2006.91]

[181] Ciupa I, Pretschner A, Oriol M, Leitner A, Meyer B. On the number and nature of faults found by random testing. Software Testing, Verification and Reliability, 2011, 21(1): 3–28. [doi: 10.1002/stvr.415]

[182] Arcuri A, Iqbal MZ, Briand L. Random testing: Theoretical results and practical implications. IEEE Trans. on Software Engineering, 2012, 38(2): 258–277. [doi: 10.1109/TSE.2011.121]

[183] Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. In: Proc. of the 22nd Companion to the ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion. Montreal: ACM, 2007. 815–816. [doi: 10.1145/1297846.1297902]

[184] Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. ACM Computing Surveys, 2019, 51(3): 50. [doi: 10.1145/3182657]

[185] Cadar C, Sen K. Symbolic execution for software testing: Three decades later. Communications of the ACM, 2013, 56(2): 82–90. [doi: 10.1145/2408776.2408795]

[186] Qi DW, Roychoudhury A, Liang ZK. Test generation to expose changes in evolving programs. In: Proc. of the 2010 IEEE/ACM Int'l Conf. on Automated Software Engineering. Antwerp: IEEE, 2010. 397–406. [doi: 10.1145/1858996.1859083]

[187] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224.

[188] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes, 2005, 30(5): 263–272. [doi: 10.1145/1095430.1081750]

[189] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2005. 213–223. [doi: 10.1145/1065010.1065036]

[190] Gupta N, Mathur AP, Soffa ML. Generating test data for branch coverage. In: Proc. of the 15th IEEE Int'l Conf. on Automated Software Engineering. Grenoble: IEEE, 2000. 219–227. [doi: 10.1109/ASE.2000.873666]

[191] 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]

[192] Liu XZ, Xu GC, Hu L, Fu XD, Dong YS. An approach for constraint-based test data generation in mutation testing. Journal of Computer Research and Development, 2011, 48(4): 617–626 (in Chinese with English abstract).

[193] Gross F, Fraser G, Zeller A. Search-based system testing: High coverage, no false alarms. In: Proc. of the 2012 Int'l Symp. on Software Testing and Analysis. Minneapolis: ACM, 2012. 67–77. [doi: 10.1145/2338965.2336762]

[194] McMinn P. Search-based software testing: Past, present and future. In: Proc. of the 4th IEEE Int'l Conf. on Software Testing, Verification and Validation Workshops. Berlin: IEEE, 2011. 153–163. [doi: 10.1109/ICSTW.2011.100]

[195] Harman M, Jones BF. Search-based software engineering. Information and Software Technology, 2001, 43(14): 833–839. [doi: 10.1016/S0950-5849(01)00189-6]

[196] McMinn P. Search-based software test data generation: A survey. Software Testing, Verification and Reliability, 2004, 14(2): 105–156. [doi: 10.1002/stvr.294]

[197] Riccio V, Humbatova N, Jahangirova G, Tonella P. DeepMetis: Augmenting a deep learning test set to increase its mutation score. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2021. 355–367. [doi: 10.1109/ASE51524.2021.9678764]

[198] Yao XJ, Zhang GJ, Pan F, Gong DW, Wei CQ. Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing. IEEE Trans. on Software Engineering, 2022, 48(4): 1169–1184. [doi: 10.1109/TSE.2020.3014960]

[199] Rajan A. Automated requirements-based test case generation. ACM SIGSOFT Software Engineering Notes, 2006, 31(6): 1–2. [doi: 10.1145/1218776.1218799]

[200] Escalona MJ, Gutierrez JJ, Mejías M, Aragón G, Ramos I, Torres J, Domínguez FJ. An overview on test generation from functional requirements. Journal of Systems and Software, 2011, 84(8): 1379–1393. [doi: 10.1016/j.jss.2011.03.051]

[201] Leung HKN, White L. Insights into regression testing (software testing). In: Proc. of the 1989 Conf. on Software Maintenance. Miami: IEEE, 1989. 60–69. [doi: 10.1109/ICSM.1989.65194]

[202] Fraser G, Staats M, McMinn P, Arcuri A, Padberg F. Does automated white-box test generation really help software testers? In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. Lugano: ACM, 2013. 291–301. [doi: 10.1145/2483760.2483774]

[203] Fraser G, Staats M, McMinn P, Arcuri A, Padberg F. Does automated unit test generation really help software testers? A controlled empirical study. ACM Trans. on Software Engineering and Methodology, 2015, 24(4): 23. [doi: 10.1145/2699688]

[204] Serra D, Grano G, Palomba F, Ferrucci F, Gall HC, Bacchelli A. On the effectiveness of manual and automatic unit test generation: Ten years later. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories. Montreal: IEEE, 2019. 121–125. [doi: 10.1109/MSR.2019.00028]

[205] Chen TY, Kuo FC, Merkel RG, Tse TH. Adaptive random testing: The art of test case diversity. Journal of Systems and Software, 2010, 83(1): 60–66. [doi: 10.1016/j.jss.2009.02.022]

[206] Hao D, Zhang L, Wu XX, Mei H, Rothermel G. On-demand test suite reduction. In: Proc. of the 34th Int'l Conf. on Software Engineering. Zurich: IEEE, 2012. 738–748. [doi: 10.1109/ICSE.2012.6227144]

[207] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 437–440. [doi: 10.1145/2610384.2628055]

[208] Humbatova N, Jahangirova G, Tonella P. DeepCrime: Mutation testing of deep learning systems based on real faults. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2021. 67–78. [doi: 10.1145/3460319.3464825]

[209] Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P. Taxonomy of real faults in deep learning systems. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1110–1121. [doi: 10.1145/3377811.3380395]

[210] Islam J, Nguyen G, Pan R, Rajan H. A comprehensive study on deep learning bug characteristics. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 510–520. [doi: 10.1145/3338906.3338955]

[211] Zhang YH, Chen YF, Cheung SC, Xiong YF, Zhang L. An empirical study on TensorFlow program bugs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 129–140. [doi: 10.1145/3213846.3213866]

[212] Daran M, Thévenod-Fosse P. Software error analysis: A real case study involving real faults and mutations. ACM SIGSOFT Software Engineering Notes, 1996, 21(3): 158–171. [doi: 10.1145/226295.226313]

[213] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? [software testing]. In: Proc. of the 27th Int'l Conf. on Software Engineering. St. Louis: IEEE, 2005. 402–411. [doi: 10.1109/ICSE.2005.1553583]

[214] Petrović G, Ivanković M, Fraser G, Just R. Does mutation testing improve testing practices? In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 910–921. [doi: 10.1109/ICSE43902.2021.00087]

[215] Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. PIT: A practical mutation testing tool for Java (demo). In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 449–452. [doi: 10.1145/2931037.2948707]

[216] Just R. The major mutation framework: Efficient and scalable mutation analysis for Java. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 433–436. [doi: 10.1145/2610384.2628053]

[217] Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N, Le Traon Y. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. Empirical Software Engineering, 2018, 23(4): 2426–2463. [doi: 10.1007/s10664-017-9582-5]

[218] Namin AS, Kakarla S. The use of mutation in testing experiments and its sensitivity to external threats. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 342–352. [doi: 10.1145/2001420.2001461]

[219] Papadakis M, Henard C, Harman M, Jia Y, Le Traon Y. Threats to the validity of mutation-based test assessment. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 354–365. [doi: 10.1145/2931037.2931040]

[220] Do H. Recent advances in regression testing techniques. Advances in Computers, 2016, 103: 53–77. [doi: 10.1016/bs.adcom.2016.04.004]

[221] Yoo S, Harman M. Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability, 2012, 22(2): 67–120. [doi: 10.1002/stvr.430]

[222] Rosero RH, Gómez OS, Rodríguez G. 15 Years of software regression testing techniques—A survey. Int'l Journal of Software Engineering and Knowledge Engineering, 2016, 26(5): 675–689. [doi: 10.1142/S0218194016300013]

[223] Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. ACM Trans. on Software Engineering and Methodology, 1993, 2(3): 270–285. [doi: 10.1145/152388.152391]

[224] Chen TY, Lau MF. A new heuristic for test suite reduction. Information and Software Technology, 1998, 40(5–6): 347–354. [doi: 10.1016/S0950-5849(98)00050-0]

[225] Black J, Melachrinoudis E, Kaeli D. Bi-criteria models for all-uses test suite reduction. In: Proc. of the 26th Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 106–115. [doi: 10.1109/ICSE.2004.1317433]

[226] Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Trans. on

Software Engineering, 2007, 33(2): 108–123. [doi: 10.1109/TSE.2007.18]

[227] Philip AA, Bhagwan R, Kumar R, Maddila CS, Nagppan N. FastLane: Test minimization for rapidly deployed large-scale online services. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 408–418. [doi: 10.1109/ICSE.2019.00054]

[228] Lin JW, Huang CY. Analysis of test suite reduction with enhanced tie-breaking techniques. Information and Software Technology, 2009, 51(4): 679–690. [doi: 10.1016/j.infsof.2008.11.004]

[229] Khan SUR, Lee SP, Javaid N, Abdul W. A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines. IEEE Access, 2018, 6: 11816–11841. [doi: 10.1109/ACCESS.2018.2809600]

[230] Travis CI. 2023. https://www.travis-ci.com/

[231] Wang KY, Zhu CG, Celik A, Kim J, Batory D, Gligoric M. Towards refactoring-aware regression test selection. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 233–244. [doi: 10.1145/3180155.3180254]

[232] Rogstad E, Briand L, Torkar R. Test case selection for black-box regression testing of database applications. Information and Software Technology, 2013, 55(10): 1781–1795. [doi: 10.1016/j.infsof.2013.04.004]

[233] Machalica M, Samylkin A, Porth M, Chandra S. Predictive test selection. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Montreal: IEEE, 2019. 91–100. [doi: 10.1109/ICSE-SEIP.2019.00018]

[234] Kazmi R, Jawawi DNA, Mohamad R, Ghani I. Effective regression test case selection: A systematic literature review. ACM Computing Surveys, 2018, 50(2): 29. [doi: 10.1145/3057269]

[235] Harder M, Mellen J, Ernst MD. Improving test suites via operational abstraction. In: Proc. of the 25th Int'l Conf. on Software Engineering. Portland: IEEE, 2003. 60–71. [doi: 10.1109/ICSE.2003.1201188]

[236] Wang XM, Cheung SC, Chan WK, Zhang ZY. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In: Proc. of the 31st IEEE Int'l Conf. on Software Engineering. Vancouver: IEEE, 2009. 45–55. [doi: 10.1109/ICSE.2009.5070507]

[237] Smith AM, Kapfhammer GM. An empirical study of incorporating cost into test suite reduction and prioritization. In: Proc. of the 2009 ACM Symp. on Applied Computing. Honolulu: ACM, 2009. 461–467. [doi: 10.1145/1529282.1529382]

[238] Mirarab S, Akhlaghi S, Tahvildari L. Size-constrained regression test case selection using multicriteria optimization. IEEE Trans. on Software Engineering, 2012, 38(4): 936–956. [doi: 10.1109/TSE.2011.56]

[239] Ledru Y, Petrenko A, Boroday S, Mandran N. Prioritizing test cases with string distances. Automated Software Engineering, 2012, 19(1): 65–95. [doi: 10.1007/s10515-011-0093-0]

[240] Hemmati H, Fang ZH, Mantyla MV. Prioritizing manual test cases in traditional and rapid release environments. In: Proc. of the 8th IEEE Int'l Conf. on Software Testing, Verification and Validation. Graz: IEEE, 2015. 1–10. [doi: 10.1109/ICST.2015.7102602]

[241] Henard C, Papadakis M, Harman M, Jia Y, Le Traon Y. Comparing white-box and black-box test prioritization. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: IEEE, 2016. 523–534. [doi: 10.1145/2884781.2884791]

[242] Khatibsyarbini M, Isa MA, Jawawi DNA, Tumeng R. Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology, 2018, 93: 74–93. [doi: 10.1016/j.infsof.2017.08.014]

[243] Eichberg M, Hermann B, Mezini M, Glanz L. Hidden truths in dead software paths. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 474–484. [doi: 10.1145/2786805.2786865]

[244] Kintis M, Papadakis M, Jia Y, Malevris N, Le Traon Y, Harman M. Detecting trivial mutant equivalences via compiler optimisations. IEEE Trans. on Software Engineering, 2018, 44(4): 308–333. [doi: 10.1109/TSE.2017.2684805]

[245] Frankl PG, Weyuker EJ. An applicable family of data flow testing criteria. IEEE Trans. on Software Engineering, 1988, 14(10): 1483–1498. [doi: 10.1109/32.6194]

[246] Parsai A, Murgia A, Demeyer S. Evaluating random mutant selection at class-level in projects with non-adequate test suites. In: Proc. of the 20th Int'l Conf. on Evaluation and Assessment in Software Engineering. Limerick: ACM, 2016. 11. [doi: 10.1145/2915970.2915992]

[247] Zhu CG, Legunsen O, Shi A, Gligoric M. A framework for checking regression test selection tools. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 430–441. [doi: 10.1109/ICSE.2019.00056]

[248] Lam W, Shi A, Oei R, Zhang S, Ernst MD, Xie T. Dependent-test-aware regression testing techniques. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 298–311. [doi: 10.1145/3395363.3397364]

[249] Parry O, Kapfhammer GM, Hilton M, McMinn P. A survey of flaky tests. ACM Trans. on Software Engineering and Methodology, 2022, 31(1): 17. [doi: 10.1145/3476105]

[250] Principles of software testing: Defect clustering and Pareto principle. 2023. https://www.softwaretestinghelp.com/7-principles-of-

software-testing/?

[251] Xue YX, Ye JM, Zhang W, Sun J, Ma L, Wang HJ, Zhao JJ. xFuzz: Machine learning guided cross-contract fuzzing. IEEE Trans. on Dependable and Secure Computing, 2022. [doi: 10.1109/TDSC.2022.3182373]

附中文参考文献:

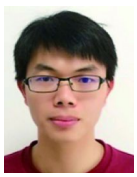
[154] 王赞, 闫明, 刘爽, 陈俊洁, 张栋迪, 吴卓, 陈翔. 深度神经网络测试研究综述. 软件学报, 2020, 31(5): 1255−1275. http://www.jos.org.cn/1000-9825/5951.htm [doi: 10.13328/j.cnki.jos.005951]

[192] 刘新忠, 徐高潮, 胡亮, 付晓东, 董玉双. 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.

路则雨(1996－), 男, 博士生, 主要研究领域为软件分析与测试.

郭肇强(1994－), 男, 博士, 主要研究领域为软件度量与缺陷定位.

张鹏(1996－), 男, 博士, 主要研究领域为软件测试智能化.



杨已彪(1987－), 男, 博士, 特任副研究员, CCF 专业会员, 主要研究领域为软件测试与分析, 缺陷检测与定位.

王洋(1996－), 男, 博士生, 主要研究领域为软件分析与测试.

周毓明(1974－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为软件分析与测试.