软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007583]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于异常检查点植入的软件缺陷定位方法 [1]

宋壹, 谢晓园, 张熙灏, 辛奇, 邢宸亮

(武汉大学 计算机学院,湖北 武汉 430072)

通讯作者: 谢晓园, E-mail: xxie@whu.edu.cn

摘 要: 软件缺陷定位任务以程序表层的执行错误为入口,通过分析程序运行过程中异常的内部状态,定位到底层代码级的缺陷根因.目前主流的基于频谱和基于突变的缺陷定位,以及当前最先进的 SmartFL 技术分别采用覆盖信息、突变信息及以程序语义为代表的信息,作为观测程序内部状态的窗口,所用信息过于宽泛、针对性不足,且三类(项)技术分别受限于语句风险值捆绑、突变成本过高及信息规模庞大等瓶颈.为此,最近一项新的基于异常触发信息的缺陷定位技术 EXPECT 提出通过程序自带的异常处理语句(Try-catch 块)对程序错误的执行状态进行监控,以更低的成本实现了超越上述主流技术的缺陷定位效果,该方法的运行前提是错误程序中必须含有足量的异常处理语句.然而,在真实的开源环境中,许多软件程序并没有设置良好的异常处理机制,导致其代码仅包含十分稀疏甚至不包含异常处理语句,这直接影响 EXPECT 技术赖以运行的基础.为此,提出基于异常检查点植入的软件缺陷定位方法 INSPECT.通过为错误程序自动植入临时的异常处理语句作为对运行时内部状态的检查点、设计更加先进的程序语句风险值计算方法,将 EXPECT 方法的适用范围拓展到不包含异常处理语句的更普遍程序,有效将异常触发信息这一缺陷定位高效源数据的应用场景进行了拓展,实现了泛化性的提升.实验结果表明,INSPECT 比现有最先进技术实现了更好的缺陷定位效果:在最好、平均、最差 EXAM 指标上的提升幅度分别为 95.25%、55.92%和 16.65%(模拟缺陷),以及 93.39%、57.54%和 13.92%(真实缺陷);在 MRR 指标上的提升为 311.47%(模拟缺陷)和 283.31%(真实缺陷).

关键词: 软件缺陷定位;异常处理机制;检查点植入;软件质量保障

中图法分类号: TP311

中文引用格式: 宋壹,谢晓园,张熙灏,辛奇,邢宸亮.基于异常检查点植入的软件缺陷定位方法.软件学报. http://www.jos.org.cn/1000-9825/7583.htm

英文引用格式: Song Y, Xie XY, Zhang XH, Xin Q, Xing CL. Exception Trigger Stream-based Fault Localization with Automated Try Block Injection. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7583.htm

Exception Trigger Stream-based Fault Localization with Automated Try Block Injection

SONG Yi, XIE Xiao-Yuan, ZHANG Xi-Hao, XIN Qi, XING Chen-Liang

(School of Computer Science, Wuhan University, Wuhan 430072, China)

Abstract: Software fault localization tasks begin with the failure of program execution, and locate the root cause of the failure at the code level by analyzing the abnormal internal state during the program execution. The current mainstream spectrum-based and mutation-based fault localization techniques, as well as the most advanced technique SmartFL, utilize coverage information, mutation information, and information represented by program semantics, respectively, as windows to observe the internal state of the program while running. These three types of information are too broad and not targeted enough, and are limited by bottlenecks such as the tie of statement risk values, high mutation cost, and large information scale, respectively. EXPECT, a fault localization technique using

[1] 基金项目: 国家重点研发计划课题(2024YFF0908003); 国家自然科学基金面上项目(62472326); CCF-智谱大模型创新基金(CCF-Zhipu202408)

收稿时间: 2025-09-05; 修改时间: 2025-10-20; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-26

exception trigger stream (a new source of information), was recently proposed to monitor the abnormal internal state of the program through the exception handling statements (Try-catch blocks), and achieved promising effectiveness that surpasses the aforementioned mainstream methods. The premise of EXPECT is that the faulty program must contain enough exception-handling statements. However, in the real open-source community, many software programs do not have a good exception handling mechanism, resulting in their codes containing only very sparse or even no exception handling statements, which directly affects the basis on which EXPECT runs. To this end, a software fault localization method based on exception handling statements injection, INSPECT, is proposed in this paper. By automatically injecting temporary exception handling statements into the faulty program as checkpoints for the internal state, designing a more sophisticated algorithm of the calculation for program statements' risk value, the running scope of EXPECT is expanded to more general programs that do not contain exception handling statements. In other words, INSPECT effectively expands the application scenario of the exception trigger information, which is an efficient source of data for fault localization, and thus delivers higher generalization. Experimental results show that INSPECT obtains better fault localization effectiveness than the state-of-the-art technique with improvements of 95.25%, 55.92%, and 16.65%(simulated faults) as well as 93.39%, 57.54%, and 13.92%(real-world faults) in the best, average, and worst EXAM metrics, respectively, and 311.47%(simulated faults) and 283.31%(real-world faults) in the MRR metric.
Key words: software fault localization; exception handing mechanism; checkpoint injection; software quality assurance

软件作为一种人类制品,不可避免地会含有缺陷,在软件系统越多越多地作为基础设施融入到社会发展各个领域的背景下,软件缺陷造成的危害往往十分严重.例如,网络安全公司 CrowdStrike 于 2024 年 7 月发布的一次软件更新存在缺陷,导致全球范围内 Windows 系统出现中断,银行、机场、医院等大量要害部门的业务因此受到影响[1];此外,权威机构 Synopsys 所发布的一份报告显示,2022 年全年美国经济因软件质量问题损失了至少 2.41 万亿美元,超过了全球 170 个国家的 GDP,而这一数字在 2020 年还是 1.31 万亿美元[2].当软件发生故障时,对导致故障的底层代码缺陷根因进行及时、准确地定位,对于后续对其的进一步处理十分重要[3-5].

近几十年来,软件缺陷定位技术作为软件质量保障特别是软件调试任务中的重要一环,受到了国内外学者的广泛关注[6-10].软件缺陷定位的基本流程是,以软件系统表层的执行失效(症状)为入口,通过深入分析程序运行时的内部状态,找到应该为表层失效负责的底层代码级缺陷根因(症结).当前,基于频谱的缺陷定位(Spectrum-based Fault Localization, SBFL)和基于突变的缺陷定位(Mutation-based Fault Localization, MBFL)是两种最为主流的技术.SBFL 首先根据测试用例实际输出与预期输出是否相符,将所有测试用例分为通过和失败两种,然后记录测试用例执行过程中对程序的覆盖信息,为那些被更多失败测试用例、更少通过测试用例覆盖的程序实体(文件、函数、语句等)赋予更高的含缺陷风险值[11-13];MBFL 对程序语句进行随机突变,并比对突变前后的测试结果,认为在含有缺陷的语句上进行突变更可能使失败测试用例变为通过,为那些被更多从失败转为通过测试用例覆盖的突变语句赋予更高的风险值[14-16]. 近年,一项基于程序语义的缺陷定位技术 SmartFL 被提出[36],其核心是对程序语义信息、静态分析信息与动态执行跟踪信息进行综合建模,已被证实为当前最先进的缺陷定位技术.尽管 SBFL 和 MBFL 技术已被证实取得了良好的缺陷定位效果,它们均将覆盖信息作为对程序内部状态进行分析的源数据,具有信息过于宽泛、针对性不强的弱点.此外,这两项技术还具有各自分别的短板.具体而言,SBFL 技术仅仅依靠覆盖信息对程序实体进行表达,使得真正含有缺陷的程序实体和与其具有相同覆盖信息的其他程序实体将被赋予同样的风险值(即“风险值捆绑”问题,亦称为“Tie”问题),该问题十分常见,已被先前许多研究指出在很大程度上威胁着 SBFL 技术的有效性[17-19].MBFL 技术通过突变对程序源码进行修改,在修改后需要重复进行编译和测试执行过程,具有很高的时间和计算成本.而当大量程序语句均被覆盖时,SmartFL 仍可能面临语义信息规模过大的问题.为解决上述挑战,于前期提出了基于异常触发流的缺陷定位技术 EXPECT[20],将程序中自带的异常处理语句(Try-catch 块)作为监测程序运行内部状态的“检查点”,分别在失败执行和通过执行下记录 Try-catch 块上的异常触发情况,将该对信息中暴露出分歧(如其中一次执行在该检查点捕获异常而另一次未捕获)的首个检查点作为“分歧点”,为分歧点和其前序最近一个检查点之间的程序语句进行风险值投票,最终定位到缺陷语句.

尽管 EXPECT 方法的缺陷定位有效性大幅超过了现有最优缺陷定位技术 SmartFL,其高度依赖于错误程序自身所带的异常处理语句.然而,在真实的开源环境中,大量软件程序因为工期紧张、仅重视功能实现等原因,并没有遵循软件开发最佳实践设置完备的异常处理机制[21],导致源代码中仅含有少量甚至不含异常处理

语句的情况十分常见[22,23].例如,对 Java 社区中流行的数学计算项目 Math 核心包中的 12,363 行代码进行扫描,未找到开发者设置的异常处理代码.由此可见,EXPECT 方法在真实开源社区的软件系统中应用能力有限,其较高的缺陷定位能力会因稀疏或缺失的异常处理语句而严重受制,使得异常触发信息这一高效的缺陷定位源数据难以在广泛真实的调试环境中发挥作用.

为此,论文提出一种基于异常检查点植入的软件缺陷定位方法 INSPECT(**INS**erted Check**P**oint-bas**E**d Fault Lo**C**aliza**T**ion),将 EXPECT 应用范围扩展至不含异常处理语句的错误程序.具体而言,首先向待测错误程序中以特定的规则、密度和粒度,自动植入 Try-catch 块作为临时检查点,在其上记录程序运行过程中的异常触发信息,作为后续缺陷定位的主要信息源;然后基于程序历史版本、不同开发分支或突变策略等得到失败测试用例的通过执行,与失败执行的异常触发信息进行比对以找到程序执行状态的分歧点;接下来基于对分歧点的分析设计一种新的程序语句风险值计算方法,得到每条语句的风险值,并进一步应用行级缺陷定位技术对上述风险值进行精化,缓解因多条语句共享同一风险值而造成的 Tie 问题;最后将程序语句按精化后的风险值进行排序,作为 INSPECT 方法的输出.在 Defects4J 数据集的 540 个错误程序版本上对方法进行验证,结果表明,相较于当前该领域的最优方法,INSPECT 在 EXAM_Best、EXAM_Average 和 EXAM_Worst 指标上的提升幅度分别为 95.25%、55.92%和 16.65%(模拟缺陷),以及 93.39%、57.54%和 13.92%(真实缺陷),MRR 指标的提升幅度为 311.47%(模拟缺陷)和 283.31%(真实缺陷).

论文工作的数据及代码在 GitHub 上开源,访问地址为: https://github.com/yisongy/INSPECT_Repo.

1 背景知识及 EXPECT 方法介绍

本文所提 INSPECT 方法利用程序异常触发信息作为主要数据源,实现更加准确的代码行级缺陷定位.下面就相关概念、基本知识,以及前期所提 EXPECT 方法予以介绍.

1.1 软件缺陷定位任务及所用信息源

软件质量保障一般分为软件测试和软件调试两个阶段[24,25].在测试中,动态测试因其较高的准确性和较低的误报率而被广泛采用(如无特殊说明,本文后续的软件质量保障均在动态环境下讨论),其执行测试用例并通过比对程序实际输出和预期输出是否相同,将测试用例划分为通过测试用例和失败测试用例,失败测试用例揭示了程序中缺陷的存在[26,27].调试过程则负责对缺陷进行底层代码层面的定位(软件缺陷定位任务),进而开展修复[28].可见,缺陷定位在整个软件质量保障过程中处于承下启下的位置,直接影响软件正常功能的恢复.根据软件测试调试领域 PIE(Propagation, Infection, Execution)模型[29],软件底层的缺陷只有感染到中间的运行状态,才能外在表达为表层可观测的程序失效(即失败测试用例).根据此模型可以得到,软件缺陷定位任务实际上以测试阶段的失败测试用例为入口,通过挖掘并分析程序运行的中间状态,找到底层代码中导致上述表层失效的根因.

当前,基于频谱的缺陷定位技术 SBFL 和基于突变的缺陷定位技术 MBFL 应用最为普遍.SBFL 技术将程序运行覆盖信息作为程序运行中间状态的代表数据,将测试用例运行过程中对程序语句的覆盖情况记录为与语句数等长的二进制向量(1 代表覆盖,0 代表未覆盖),然后基于“被越多失败测试用例覆盖、越少通过测试用例覆盖的程序语句应具有越高的含缺陷风险”思想,将该思想数学化表达为程序语句风险值计算公式,进而得到程序语句的风险值排序列表作为方法输出[30-32].MBFL 技术同样以通过和失败测试用例作为分析对象,其对程序语句进行随机突变,然后基于“在含有缺陷程序语句上的突变更有可能使失败测试用例变为通过,在不含有缺陷语句上的突变更有可能使通过测试用例变为失败”思想,设计能够体现这一思想的数学化公式来为程序语句计算风险值,最后返回风险值排序[33-35].尽管上述两类技术均在缺陷定位领域十分经典,其分别具有不同的短板.具体而言,SBFL 技术收集的程序执行覆盖信息往往过于庞大,使得该方法具有较高的运行成本;此外,覆盖信息只能从浅表的角度反映程序运行时的中间状态,使得许多程序语句可能具有完全相同的覆盖信息,它们将被赋予相同的风险值而无法被区分开(具有相同风险值的语句绑定形成“Tie”).特别是当真正含

有错误的语句处于 Tie 中时,缺陷定位的有效性可能受到严重威胁.MBFL 技术依靠程序突变得到大量变体,对于生成的变体均需重新编译执行,时间和空间成本均十分可观,且其有效性同样受到 Tie 问题的影响.近期,一种基于语义的缺陷定位技术 SmartFL 被提出[36],其对程序语义信息、静态分析信息与动态执行跟踪信息进行综合建模,采用基于概率的方法为程序语句预测风险值以实现缺陷定位,该技术已被证实效果超过了 SBFL 和 MBFL 技术,展现了语义信息用于分析程序运行中间状态、进而定位缺陷的潜力.尽管如此,该方法所采集语义信息的规模仍与程序执行跟踪信息的规模高度相关,当大量程序语句均被覆盖时,SmartFL 所用语义信息的大小可能和传统覆盖信息接近.

综上所述,现有主流缺陷定位技术均面临着所用信息源庞大、难以精确对缺陷位置进行分析的不足.因此,一项重要问题是如何找到一种能够有效在表层执行失效和底层缺陷根因之间建立起链接,但又具有相对较小规模的程序运行中间状态信息源,以开展更加高效的缺陷定位.

1.2 程序异常处理机制

异常是一项主要用于处理程序运行时事件的机制[37].在软件程序运行过程中,可以在程序执行至特定代码位置且符合特定条件时抛出异常,并由开发者设置异常处理代码对该异常进行捕获,进而分析导致异常抛出的原因,调整程序后续的执行策略.异常处理机制设计的主要初衷之一,是能够准确地记录并报告程序运行时的中间状态,为后续可能的软件质量保障活动提供信息,以帮助提高软件程序的可靠性.异常所表现的抛出和未抛出行为是观察程序执行中间状态的有效信息,其具有简单、轻量级等特点.因此,结合前文所述 PIE 模型的定义,异常触发信息具备作为观测程序运行中间状态的信息源,帮助更有效地建立“表层执行失效→中间运行状态→底层缺陷根因”链接的潜力.

以 Java 语言为例对异常处理机制进行介绍.在 Java 语言中,异常是 Thorwable 类的实例,一般包括要求开发者使用异常处理语句显式处理的 Checked Exception、允许开发者不做处理的 Unchecked Exception 和通常会导致所在线程终止运行的 Error.其中:Checked Exception 指受检查的异常(如 IOException 和 FileNotFoundException 等),除非在方法声明后使用 throws 关键字标明对应方法中可能抛出的异常类型,否则必须使用 Try 代码块捕获并处理;Unchecked Exception 指不受检查的异常(如 ArithmeticException 和 ClassCastException 等),在 Java 语言中,除通过 throw 关键字抛出指定类型的异常以外,部分语句在执行时也可能按照编译器的规则抛出异常,如除零异常和空指针访问异常等;Error 指错误(如 OutOfMemoryError 和 StackOverFlowError 等),由于此类异常通常代表程序运行时出现较为严重的错误,开发者往往不对其进行捕获和处理[38,39].当异常被抛出时,其可以被最近的异常处理代码(即 Java 中的 Try 代码块)捕获,并根据紧跟在 Try 代码块后的 catch 代码块中指定的异常类型来匹配由开发者设计的异常处理逻辑,然后依照对应逻辑进行如打印异常相关日志、执行特定业务逻辑或终止线程运行等异常处理过程.

1.3 前期所提出基于异常信息的缺陷定位方法EXPECT

为了验证程序异常信息用于缺陷定位任务的潜力,于前期进行了一项探索性实验.以 GitHub 开源社区中广泛使用的 Gson 项目[40]为分析对象,将该程序自带的异常处理语句作为观测程序运行中间状态的窗口(称为“检查点”),记录在检查点上显现的异常触发与否信息(触发/未触发)和执行跟踪(测试执行的所有程序语句),将以上两项信息合称为“异常触发流”.基于程序突变技术生成 Gson 的 40 个缺陷版本,在缺陷版本和原始无缺陷版本上分别运行单个失败测试用例,分别得到该失败测试用例在失败执行和通过执行中的异常触发流,进行比对以找到“分歧点”(即最先显现出失败执行和通过执行异常触发流差异的检查点).结果发现,在 80%的缺陷版本中,缺陷语句都位于分歧点和上一个最近的检查点之间.这一现象表明了程序异常触发信息在定位缺陷方面具有的良好潜力.基于该探索性实验的结果,提出了一种基于异常触发信息的缺陷定位方法 EXPECT.该方法以待测错误程序和其对应的失败测试用例为输入,首先为错误程序生成替代版本使得失败测试用例在其上能够变为通过,然后在错误程序和其替代版本上分别执行失败测试用例,以获得该失败测试用例的失败执行和通过执行;接下来,基于程序中原先自带的异常处理语句,分别在上述失败执行和通过执行中记录得到

异常触发流,并通过比对找到两组信息开始体现差异的分歧点,以该分歧点的位置为依托确定错误语句的潜在范围;最后引入行级缺陷定位技术,为范围内语句进一步赋予更细粒度的风险值.该工作的实验结果表明,EXPECT 方法能够明显超过现有最优的 SmartFL 技术.有关该工作的具体细节,请查阅参考文献[20].

尽管 EXPECT 实现了良好的缺陷定位效果,该方法强烈依赖于待测程序中原本自带的异常处理语句,当待测程序中不含或仅含少量异常处理语句时,其有效性将受到严重威胁.具体而言,EXPECT 方法的关键是根据收集到的异常触发流确定分歧点,然后才能根据分歧点开展后续的语句风险值计算.而收集有效异常触发信息的前提是程序中含有一定数量的异常处理语句,否则收集到的异常触发信息将十分稀疏甚至为空.尽管软件开发最佳实践往往推荐开发者在程序中设置有效的异常处理语句,但该建议在许多情况下并不被遵循,导致真实开源程序中仅含少许甚至不含异常处理语句的情况十分常见.论文对 GitHub 社区中的开源项目进行了调研,进一步证实了这一现象.例如,对 Java 社区流行的数学计算项目 Math 核心包[41]的 12,363 行代码进行扫描,未发现开发者书写的异常处理代码.这表明,EXPECT 方法虽然实现了较好的缺陷定位效果,但其对异常处理语句的依赖限制了其泛化性.因此,如何在异常处理语句缺失这一十分常见的情况下,依然能够利用程序异常信息在缺陷定位任务中的潜力,开展高效的缺陷定位,是软件调试领域需要解决的一项重要问题.

2 基于检查点植入的软件缺陷定位方法 INSPECT

为解决上述基于异常信息的缺陷定位技术在真实开源环境泛化性受限的挑战,论文提出一种基于检查点植入的软件缺陷定位方法 INSPECT(**INS**erted check**P**oint-bas**E**d fault lo**C**aliza**T**ion),通过自动植入检查点的方式使得在待测程序没有异常处理代码的情况下,依然能够采集到异常触发信息并将其作为观测程序运行中间状态的窗口,进而收集得到完整的异常触发流,以开展缺陷定位.INSPECT 方法首先在待测程序上运行测试用例集,根据实际输出和预期输出是否相同,将所有测试用例划分为通过测试用例和失败测试用例,失败测试用例将作为输入,驱动检查点植入、伪正确版本生成、异常触发流收集及分歧点确定、程序语句风险值计算这四个步骤的开展,如图 1 所示.下面对这四个步骤分别进行详细描述.

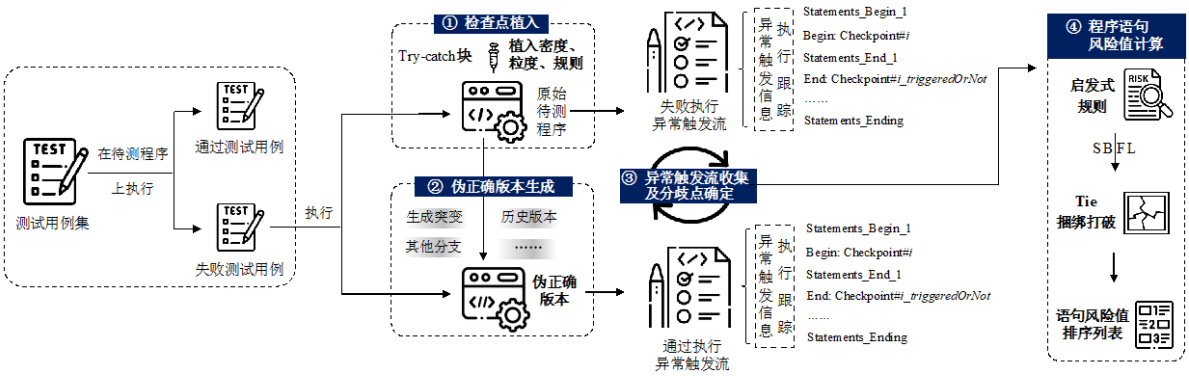


图 1 基于检查点植入的软件缺陷定位技术 INSPECT 流程图

2.1 检查点植入

如前文所述,程序异常处理代码在缺陷定位任务中的作用是检查相应位置的代码是否触发异常,因此植入的异常处理代码也称为“检查点”.从植入粒度、植入规则、和植入密度三个维度确定检查点植入的方法.下面分别予以详细介绍.

(1) 植入粒度.检查点植入粒度是指每个检查点检查程序执行状态的范围.为了确定所检查的范围内是否有异常被触发,并记录异常触发信息,采用 Try 代码块的形式植入检查点.因此,检查点植入的粒度实际上是每个 Try 代码块所包裹和检查的程序语句范围.程序开发是一个较为复杂的过程,不同项目

往往有各自不同的代码逻辑与结构设计,为了尽可能使得植入的 Try 代码块可以有效捕获到程序执行过程中触发的异常,以更全面地反映程序执行状态,方法以函数为粒度植入 Try 代码块,即每个被植入的检查点唯一对应一个程序函数,其 Try 代码块包裹该函数中的所有程序语句.选择函数作为检查点植入的粒度有两个主要原因.一方面,假设采用更细的粒度进行检查点植入(如采用语句级粒度,每个检查点的 Try 代码块仅包裹一条或多条程序语句),则植入 Try 代码块的过程需要考虑不同项目的代码逻辑和结构差异,导致检查点的植入受到具体项目的限制,这与 INSPECT 技术提高缺陷定位泛化能力的目标不符;另一方面,仅当 Try 代码块所包裹的程序语句触发异常时,检查点才能记录该次被触发的信息,若缩小 Try 代码块包裹的程序语句范围,可能增加程序执行时所触发异常被遗漏的概率,以函数为粒度的 Try 代码块可以捕获所有当前函数中触发的异常,因此能够更好地体现当前的程序执行状态.

(2) 植入规则.检查点植入的规则包括植入 Try 代码块的具体形式、其判断异常是否触发的规则,以及对被测程序执行逻辑的影响.INSPECT 在函数上植入检查点的模板如图 2 所示.其中,第 1 行的“func”代表函数声明,第 5 行的“Statements”代表函数体中的程序语句,这两部分是待植入函数的原有内容.第 4 ~ 6 行的 Try-catch 代码块包裹了目标函数中的所有程序语句,因此,当函数中的任意语句触发异常时,都可以被植入的 Try-catch 代码块捕获.第 6 行的 catch 关键字指明了 Try 代码块将捕获的异常类型为 Exception,意味着不限制具体的异常类型,任意 Exception 的子类都将在被函数中的语句触发后由第 6 行的 catch 代码块捕获.为了记录目标函数的异常触发信息,在第 2 行以 *triggeredOrNot* 变量标识异常的触发情况,该变量的初始值为 0(代表函数中没有异常被触发).在执行函数的过程中,若某条语句触发了异常,该异常将导致函数执行跳转至第 6 行的 catch 代码块,从而使得第 7 行语句将 *triggeredOrNot* 变量的值修改为 1(代表函数中有异常被触发).检查点上具体的异常触发信息记录由第 3 行语句(检查点开始)和第 10 行语句(检查点结束)完成,其中的“Checkpoint#*i*”是在不同函数上植入的以“*i*”为唯一标识的检查点.第 2.3.1 小节的图 4 给出了一个检查点植入的具体示例.

```
1.      func{
2.          String triggeredOrNot = "0";
3.          System.out.println("Begin:Checkpoint#i");
4.          try{
5.              Statements;
6.          } catch (Exception triggered){
7.              triggeredOrNot = "1";
8.              throw triggered;
9.          } finally {
10.             System.out.println("End:Checkpoint#i_" + triggeredOrNot); }}
```

图 2 检查点植入模板

值得注意的是,INSPECT 方法植入检查点是为了缓解现有基于异常信息的缺陷定位技术对程序自带异常处理语句的依赖,其仅仅是对程序原有异常处理语句的临时性替代,并非对程序异常处理语句设置的推荐或修改.因此,植入的检查点不应对程序原有的执行逻辑产生影响.在图 2 所示的检查点植入模板中,若仅在原始函数上植入 Try 代码块并设置捕获所有 Exception 子类的 catch 代码块,可能导致函数中原本会向上层调用栈抛出的异常被植入的检查点隐藏,从而扰乱程序原有执行逻辑.以 Java 中流行度较高的 Spring 框架[42]对异常处理机制的应用为例,Spring 的关系型数据库事务管理往往会在捕获到异常时进行事务的回滚,如果该异常被植入的 Try 代码块捕获并隐藏,可能导致原本需要回滚的事务被正常提交,使得数据库操作出错.为了避免这种情况,方法在所植入检查点的 catch 代码块中向上抛出捕获的异常,如图 2 中的第 8 行所示.这一处理使得植入的检查点不会影响程序原有的执行逻辑,且第 8 行的 throw 语句不会阻止第 10 行中检查点结

束信息的记录,因为在异常处理相关的语法规则中,一旦程序执行进入 Try 代码块,任何 return 和 throw 语句的结果都将被暂时保存,并在第 9 行的 finally 代码块执行后返回.

(3) 植入密度.检查点植入的密度是在被测程序中植入检查点的数量.由于 INSPECT 方法以函数为粒度植入 Try 代码块,检查点植入的密度实际上是待测程序中被选择植入 Try 代码块的函数数量占所有函数的比例.开发人员在程序中设置的异常处理代码反映了其对程序的理解,这些异常处理代码往往存在于程序中更可能出现错误执行状态的位置,因而能够以较低的密度有效检查程序执行状态.INSPECT 方法采用自动化、轻量级的手段植入临时检查点作为对程序原有异常处理代码的替代,因而难以完全覆盖可能出现错误执行状态的位置.为此,以相对于一般程序中异常处理语句(即开发者设置的 Try 代码块)更高的密度植入检查点.具体而言,顺序扫描待测程序中的所有函数,以固定间隔 4 为函数植入检查点,最终为被测程序中 20%的函数植入检查点(检查点植入具体密度值的选择将在第 4.3 节中进一步分析).

需要说明的是,INSPECT 方法为待测程序植入的检查点仅会临时存在于缺陷定位任务中,其并不会被真实地引入到程序本身,因此植入检查点这一操作不会增加程序的运行开销.

2.2 伪正确版本生成

与 SBFL、MBFL 技术采用覆盖信息、突变信息开展缺陷定位一样,采用异常信息开展缺陷定位同样通过对失败执行和通过执行进行比对,得到指向底层缺陷根因位置的线索.根据 PIE 模型,揭露了待测程序中缺陷语句存在的失败测试用例,其实际输出与预期输出不符的原因在于这些测试用例执行通过了缺陷语句,导致执行状态受到感染.相对地,通过测试用例的实际输出与预期输出一致,其执行状态在一定程度上体现了程序的“标准行为”,因而具备与失败执行进行比对的潜力.原始无缺陷程序在真实调试环境下并不存在,通过执行因而难以获得,前期 EXPECT 方法通过生成伪正确版本,来获得一条失败测试用例对应的通过执行,以进行比对.具体而言,伪正确版本是指与待测错误程序不同,能够将待测错误程序上一条失败测试用例转变为通过的程序版本(因为得到该版本的目的仅是获得失败测试用例对应的通过执行,以对二者进行比对,而非对错误程序进行修复,因此该版本称为“伪正确版本”).EXPECT 方法已经证实,在程序自身含有足量异常处理语句的情况下,通过伪正确版本得到的通过测试用例具有体现程序“标准行为”的能力,能够有效和失败执行进行比对并定位缺陷.在程序自带异常处理语句缺失、INSPECT 方法自动植入检查点的情况下,通过测试用例是否仍具有这一能力有待检验.

论文以 Math 项目[41]为例,通过探索性实验验证了这一推测.例如,现有 Math 项目的一个错误版本(ArithmeticUtils.java 文件第 479 行中存在一处缺陷: “==”运算符被误写为“ !=”),如图 3 所示.由于该缺陷的存在,测试用例“testLcm”的实际输出与预期输出不一致,被判为失败测试用例. 采用基于突变的策略生成不同于待测错误程序的版本,尝试获取能够使得测试用例“testLcm”执行通过的伪正确版本.结果发现,将源文件 ArithmeticUtils.java 中第 509 行代码的“==”运算符变为“!=”,可将失败测试用例“testLcm”变为通过.按照与待测程序相同的方法在该伪正确版本上植入检查点,并收集异常触发信息,将所收集的信息与在原始无缺陷 Math 项目上收集的信息进行对比.结果显示,虽然伪正确版本中仍存在缺陷语句,但测试用例“testLcm”在伪正确版本上执行和在原始无缺陷版本上执行收集到的异常触发信息保持了一致.这表明在 INSPECT 的工作环境下,伪正确版本可以有效代替原始无缺陷版本来获取通过执行.该例子中植入的检查点、失败测试用例、失败和通过执行下的异常触发流以及程序源代码,在论文开源仓库中予以提供.

```
475.  public static int lcm(int a, int b) throws MathArithmeticException{
476.      if (a == 0 || b == 0){
477.          return 0;}
478.      int lcm = FastMath.abs(ArithmeticUtils.mulAndCheck(a / gcd(a, b), b));
479.      if (lcm != Integer.MIN_VALUE) {       //缺陷语句: “!=” 应为 “==”
480.          throw new MathArithmeticException(LocalizedFormats.LCM_OVERFLOW_32_BITS, a, b);}
```

```
481.        return lcm;}
```

图 3 Math 项目一错误版本

为进一步验证这一现象,基于突变策略为上述 Math 项目生成 40 个含有缺陷的错误版本,尝试为每个错误版本的失败测试用例生成能够提供其通过执行的伪正确版本,并比对测试用例在伪正确版本和原始无缺陷程序上收集到的异常触发信息.结果显示,对于 100%的失败测试用例,都能够容易地获取到其伪正确版本,且测试用例在伪正确版本上基于植入的检查点收集到的异常触发信息与在原始程序上收集到的一致.该结果提示,在基于植入检查点收集异常触发信息的情况下,伪正确版本仍可以作为原始无缺陷程序的有效替代,用以获取通过执行.上述探索性研究中 40 个错误版本的失败测试用例数量,以及其失败测试用例在伪正确版本、原始无缺陷版本上收集到的异常触发信息和比对结果,在论文开源仓库中予以提供.

以上介绍了伪正确版本用于 INSPECT 工作流的可行性,下面具体介绍伪正确版本的获取方法.伪正确版本的获取有多种方案,包括回滚历史版本、挖掘开发分支、实施突变策略等.历史版本是在程序开发过程中保留的过往版本,其典型用途是在程序的较新版本出现严重故障时进行回滚,因此,历史版本可能与被测程序相近并有可能将错误程序上的失败测试用例转变为通过,进而提供失败测试用例的通过执行.开发分支通常在开发者尝试为程序开发新功能或修复已有故障,且不希望对现有开发活动造成干扰时被创建.对于与待测错误程序不同的其他开发分支,同样有可能将错误程序上的失败测试用例转变为通过,因此可以作为伪正确版本.通过近期历史版本或不同开发分支获取伪正确版本不会引入额外开销,且上述两类渠道在实际程序开发中往往较容易得到(例如,开源项目 Gson 具有超过 2,000 个历史版本,开源项目 FastJson 具有超过 4,000 个历史版本和 10 个活跃的开发分支).当通过历史版本或不同开发分支无法获取失败测试用例对应的伪正确版本时,实施突变策略为待测错误程序生成若干变体,将第一个把失败测试用例的执行结果转为通过的变体作为伪正确版本.

值得注意的是,尽管伪正确版本在收集异常触发信息方面可以有效替代原始无缺陷版本,根据观察,在其上收集的传统信息(覆盖信息等)却可能和在原始无缺陷版本上收集的存在显著差异.因此,伪正确版本生成后难以用于基于传统信息的缺陷定位技术.此外,INSPECT 方法作为一种轻量级的缺陷定位技术,其各步骤的设计均尽量避免不必要的运行开销.在方法执行过程中,一处可能的较高开销点在于如上所述的采用基于突变的方法生成伪正确版本.但是,这是在通过历史版本或不同开发分支均无法获得伪正确版本时的兜底方案,而实际上,历史版本和开发分支是软件演化与团队协作过程的产物,在实际面向真实开源工程的缺陷定位任务中,该类资源可以较为容易地获取,因此,INSPECT 的执行开销与传统基于覆盖信息的缺陷定位技术基本持平.即便需要采用基于突变的方法来生成伪正确版本,INSPECT 的整体开销也仅会上升至与传统 MBFL 技术相似的水平.

2.3 异常触发流收集及分歧点确定

2.3.1 异常触发流收集

为了更加全面地反映程序运行的中间状态,INSEPCT 按测试用例在执行时的实际顺序收集异常触发信息和执行跟踪两类数据,将二者合称为“异常触发流”,作为后续开展缺陷定位的源信息(即,异常触发信息是异常触发流的一个子部分).异常触发信息包括对应检查点的位置以及在检查点上是否触发异常,执行跟踪信息包括测试用例执行经过的每一条程序语句.异常触发流的结构如表 1 所示,其中,蓝色字体表示异常触发信息,黑色部分表示程序执行跟踪.下面分别予以详细介绍.

表 1 异常触发流结构

独立受检	异常触发流
-	Statements_Begin_1
1	Begin: Checkpoint#*i*
-	Statements_End_1

1	End: Checkpoint#*i*_*triggeredOrNot*
-	Statements_Begin_2
2	Begin: Checkpoint#*i*
-	Statements_End_2
2	End: Checkpoint#*i*_ *triggeredOrNot*
	……
-	Statements_Ending

在异常触发信息中,“Begin: Checkpoint#*i*”表示程序执行进入所植入的检查点 Checkpoint#*i*(对应图 2 中的第 3 行),以“End: Checkpoint#*i*”为开头的行表示程序执行退出检查点 Checkpoint#*i*(对应图 2 中的第 10 行),其中 *i* 是每个所植入检查点的唯一标识.考虑到单个检查点(Try-catch 块)可能在一次程序执行中被多次进入,采用 “独立受检”区分程序执行过程中每次进入的检查点,例如,在表 1 中,“独立受检 1”、 “独立受检 2”分别表示程序执行进入的第 1、2 个检查点,依此类推,每个独立受检对应的 Checkpoint#*i* 可能相同或不同.在程序退出检查点时,会在“End: Checkpoint#*i*_ *triggeredOrNot*”行中以变量 *triggeredOrNot* (对应图 2 中的第 2、7 行)的形式记录对应独立受检中是否触发异常,当该值为 1 时,表示检查点 Checkpoint#*i* 中捕获到了异常,为 0 表示未捕获.

在程序执行跟踪中,以“Statements_Begin”开头的行表示程序在进入对应独立受检前执行的语句,以“Statements_End”开头的行表示程序在进入独立受检后、退出独立受检前执行的语句,上述两组语句间不存在交集.例如,在表 1 中,“Statements_Begin_1”代表程序在第一次进入检查点前执行的所有语句,“Statements_End_1”代表程序在第一次进入检查点后、退出该检查点前执行的所有语句.特别地,“Statements_Ending”表示程序执行退出最后一个独立受检后执行的所有语句.

以图 3 的错误函数为例对异常触发流进行演示.用图 2 所示的规则为其植入检查点,植入后的效果如图 4 所示(其中下划线语句为植入的异常处理语句,无下划线语句为函数自身语句),其中第 477 行、第 486 行的语句分别对应表 1 中一条以“Begin”开头的语句和一条以“End”开头的语句(即异常触发信息).用以“Begin”开头和以“End”开头的语句作为分界,可形成表 1 中以“Statements_Begin”、 “Statements_End”开头的行所代表的程序语句块和“Statements_Ending”语句块(即程序执行跟踪).

```
475.  public static int lcm(int a, int b) throws MathArithmeticException{
476.      String triggeredOrNot = "0";
477.      System.out.println("Begin:ArithmeticUtils.java 475");
478.      try{
479.          if (a == 0 || b == 0){
480.              return 0;}
481.          int lcm = FastMath.abs(ArithmeticUtils.mulAndCheck(a / gcd(a, b), b));
482.          if (lcm != Integer.MIN_VALUE) {       //缺陷语句:“!=” 应为 “==”
483.              throw new MathArithmeticException(LocalizedFormats.LCM_OVERFLOW_32_BITS, a, b);}
484.          return lcm;}
485.      catch (Exception triggered) { triggeredOrNot = "1"; throw triggered; }
486.      finally { System.out.println("End:ArithmeticUtils.java 475 " + triggeredOrNot); }}
```

图 4 检查点植入及异常触发流示例

为便于后续的形式化描述,在具有 n 个失败测试用例 f_i (i = 1, 2, …, n)的待测程序中,将为 f_i 生成的伪正确版本记为 pcv_i,在待测错误程序上的失败执行和在 pcv_i 上的通过执行分别记为 e_i 和 pe_i,在 e_i 和 pe_i 上收集到的异常触发流分别记为 ets_fail_i 和 ets_pass_i.

INSPECT 将收集到的异常触发流作为缺陷定位任务的源信息,在所瞄准缺陷的触发对异常捕获与异常分析敏感的情况下,将收到更好的效果.如果缺陷的执行过程不涉及异常产生及传播,则在异常触发流收集环节可能难以得到富含缺陷指示意义的数据.我们认为,程序异常信息与其他各种缺陷定位源信息一样,难以完全覆盖所有缺陷场景.例如,当前得到最广泛使用之一的程序执行覆盖信息可能面临偶然性正确问题,因为根

据 PIE 模型,即便真实的缺陷语句被覆盖,错误也可能不被传导到中间的执行状态,导致覆盖信息在定位缺陷方面存在本质不足.为了更全面地度量 INSPECT 在真实环境下面向不同种类缺陷的定位能力,在实验中基于 Defects4J 这一缺陷定位领域最流行数据集之一中常用的 Chart, Time, Math 和 Lang 工程,生成了 480 个突变缺陷版本,并选取了 60 个真实缺陷版本,以开展更加全面的实验(详见第 3 章和第 4 章).

2.3.2 分歧点确定

根据 PIE 模型,底层代码中含有的缺陷传导到中间状态,才能外在体现为表层的执行失效.论文将程序异常触发流作为观测程序运行中间状态的窗口,因此,需要比对在失败执行 e_i 和通过执行 pe_i 上分别收集到的异常触发流 ets_fail_i 和 ets_pass_i,将二者首次显现执行状态差异的位置记为分歧点 bp_i,以此作为计算程序语句风险值、进而定位底层代码缺陷的依据.

对比 ets_fail_i 和 ets_pass_i 时,首先将两套异常触发流中以“End”开头的信息对齐,以便成对比较.设 ets_fail_i 中有 $count\_fail_i$ 条以“End”开头的信息,ets_pass_i 中有 $count\_pass_i$ 条以“End”开头的信息.按收集顺序依次检查 ets_fail_i 和 ets_pass_i 中以“End”开头的信息,ets_fail_i 中第 k (k = 1, 2, …, min($count\_fail_i$, $count\_pass_i$))条信息将与 ets_pass_i 中第 k 条信息对比.在比较一对以“End”开头的信息时,其体现的执行状态包括两部分,即 *Checkpoint#i* 和 *triggeredOrNot*.具体而言,若一对信息的 *Checkpoint#i* 不同,代表其所对应的独立受检是在不同的关键检查点上完成的,在此时 e_i 已与 pe_i 具有不同的执行路径,因此被视为存在不同的执行状态;若一对信息的 *triggeredOrNot* 不同,代表 e_i 和 pe_i 在对应的独立受检中分别触发和未触发异常,同样被视为存在不同的执行状态.换句话说,当且仅当一对信息的 *Checkpoint#i* 和 *triggeredOrNot* 均相同时,e_i 和 pe_i 被认为在对应的独立受检中具有相同的执行状态.按照顺序依次对 ets_fail_i 和 ets_pass_i 中的前 k 条信息进行比对,根据是否发现不同的执行状态,可以得到**情况-1** (在独立受检中发现不同执行状态)和**情况-2** (在独立受检中未发现不同执行状态).

在情况-1 中,发现了不同的执行状态的信息将被记为 bp_i.若多对以“End”开始的信息均体现了不同的执行状态,则最早被收集的信息将被记为 bp_i.这是因为异常触发流是按照程序的实际执行顺序收集的,执行状态首次出现不同时,缺陷语句可能已被执行,该策略可减少缺陷语句的排查空间.

若对比 ets_fail_i 和 ets_pass_i 中前 k 条以“End”开头的信息未能发现不同的执行状态(即情况-2),则进一步尝试通过对比 $count\_fail_i$ 和 $count\_pass_i$ 的值来确定分歧点 bp_i.具体可分为以下五种子情况:

- **子情况-2.1:** $count\_fail_i = count\_pass_i$.若 ets_fail_i 和 ets_pass_i 中的独立受检数量相等,且按顺序对比独立受检无法发现不同的执行状态,则在 f_i 上基于检查点收集的异常触发信息不足以用于捕获执行状态分歧点.在这一子情况中,bp_i 无法确定,f_i 将在后续的缺陷定位环节被忽略;
- **子情况-2.2:** $0 < count\_fail_i < count\_pass_i$.若 ets_fail_i 中存在更少的独立受检,则第 $count\_fail_i$ 条以“End”开始的信息将成为 bp_i.因为 e_i 在该点后直接终止了执行,而 pe_i 却在该点后继续进入更多的独立受检,所以尽管该点并未显式地通过 *Checkpoint#i* 或 *triggeredOrNot* 表现出 e_i、pe_i 之间的执行差异,其仍可被视为二者之间执行状态的分歧点;
- **子情况-2.3:** $0 < count\_pass_i < count\_fail_i$.若 ets_fail_i 中存在更多的独立受检,则第 $count\_pass_i$ 条以“End”开始的信息将成为 bp_i.因为 pe_i 在该点后直接终止了执行,而 e_i 却在该点后继续进入更多的独立受检,所以尽管该点并未显式地通过 *Checkpoint#i* 或 *triggeredOrNot* 表现出 e_i、pe_i 之间的执行差异,其仍可被视为二者之间执行状态的分歧点;
- **子情况-2.4:** $0 = count\_fail_i < count\_pass_i$.若仅 pe_i 进入了独立受检,则无法将任何一条以“End”开始的信息确定为 bp_i,该种情况下的语句风险值计算方法将在第 2.4 节中讨论;
- **子情况-2.5:** $0 = count\_pass_i < count\_fail_i$.若仅 e_i 进入了独立受检,由于 ets_fail_i 中以“End”开始的信息均未参与比对,故不将任何一条以“End”开始的信息确定为 bp_i,该种情况下的语句风险值计算方法将在第 2.4 节中讨论.

2.4 程序语句风险值计算

前期 EXPECT 方法已经证实,在待测程序原本自带异常处理语句的情况下,收集得到的异常触发流及基于其得到的分歧点可以帮助实现有效的缺陷定位.在论文所面向的异常处理语句缺失情况下,异常触发流的收集及分歧点的确定基于的是 INSPECT 方法自动植入的检查点,此时它们是否仍具备良好的缺陷定位能力尚不明确.为此,利用第 2.2 节探索性实验中随机生成的 40 个待测错误版本继续进行验证.具体而言,在这些待测错误版本及其对应的伪正确版本上,收集所有失败测试用例的失败执行和通过执行,进而分别获取两种执行状态下的异常触发流并进行比对.结果显示,在 100%的待测错误版本中,均可以容易地通过比对来确定分歧点,且真实缺陷语句均存在于分歧点之前的执行跟踪信息中.上述探索性研究中 40 个错误版本真实缺陷语句与分歧点的位置关系比对结果,在论文开源仓库中予以提供.这一结论说明,在论文面向的异常处理语句缺失的情况下,上述异常触发流收集方法及衍生的分歧点定位方法,具有为缺陷定位任务提供有效支撑的潜力.但是,前期 EXPECT 方法的实施基于的是待测错误程序自带的异常处理语句,其本身反映了开发人员对程序中缺陷可能所在位置的理解,因此,在其结论中,绝大多数缺陷都处于“分歧点和其之前最近的检查点”之间.而在论文所提 INSPECT 方法中,异常处理语句均为自动化植入,因此缺陷位置难以被精确地限定在同一范围,而只能如上所述地被推测位于分歧点之前.因此,如何在更加模糊的范围内设计一套有效的程序语句风险值计算方法,是 INSPECT 方法相较于前期 EXPECT 方法需要解决的一项新的挑战.

为此,提出如下方案,解决检查点自动植入情境下的程序语句风险值计算问题.对于待测错误程序中的一条程序语句 s,其风险值 $Suspiciousness_s$ 由公式(1)计算:

$$suspiciousness_s = \sum_{i=1}^{n} sus_s^i + Assist\_sus_s \tag{1}$$

其中,sus_s^i是基于第 i 条失败测试用例计算得到的语句 s 的风险值,其通过对比第 i 条失败测试用例的失败执行 e_i 上的 ets_fail_i 和在相应伪正确版本上通过执行 pe_i 的 ets_pass_i,并分析由此确定的分歧点 bp_i 计算得到;$\sum_{i=1}^{n} sus_s^i$是在 n 个失败测试用例上计算的语句 s 的风险值之和,其将作为 INSPECT 为程序语句 s 所计算风险值的主要部分(将在第 2.4.1 小节中具体介绍).$Assist_sus_s$是通过在待测错误程序上应用行级缺陷定位方法得到的语句 s 的风险值,作为 INSPECT 为程序语句 s 所计算风险值的辅助部分,以缓解多条语句可能具有相同风险值造成的 Tie 问题(将在第 2.4.2 小节中具体介绍).

2.4.1 基于分歧点分析的语句风险值计算

运行第 i 条失败测试用例时,程序语句 s 在一对失败执行 e_i 和通过执行 pe_i 上的风险值sus_s^i由公式(2)计算:

$$sus_s^i = \begin{cases} 1 & s \in Suspicious\_Group_i \\ 0 & s \notin Suspicious\_Group_i \end{cases} \tag{2}$$

其中,$Suspicious_Group_i$由在 e_i 和 pe_i 上被判定为可能含有缺陷的语句组成,$Suspicious_Group_i$中的语句以表 1 中以“Statements_Begin”、“Statements_End”开头的行和 Statements_Ending 为单位, 代表 ets_fail_i 在异常触发流中以独立受检的进入和退出为分界的一组被执行程序语句(记为$statements_x^i$).判断$statements_x^i$是否被包含在 $Suspicious_Group_i$中的规则如公式(3)所示:

$$statements_x^i \in Suspicious\_Group_i, \text{ if } \begin{cases} End\_diff_x^i > 0 \ \textbf{or} \\ End\_diff_x^i = 0 \text{ and } (Sit_i = 2.2 \text{ or } 2.3 \text{ or } 2.4) \ \textbf{or} \\ End\_diff_x^i = 0 \text{ and } Begin\_diff_x^i = 0 \text{ and } Sit_i = 2.5 \end{cases} \tag{3}$$

具体而言,根据 ets_fail_i,设 e_i 在$statements_x^i$前进入了$Begin_Count_x^i$个独立受检,退出了$End_Count_x^i$个独立受检(以

表 1 中异常触发流为例,假设其展示了 ets_fail_i 的完整结构,则对于 Statements_End_2,其$Begin_Count_x^i$值为 2、$End_Count_x^i$值为 1).同时,设 e_i 在执行至 bp_i 为止进入了 $Begin\_Count\_bp_i$ 个独立受检,退出了 $End\_Count\_bp_i$ 个独立受检(考虑到 bp_i 自身是一处退出独立受检的位置,计算 $End\_Count\_bp_i$ 时也包含 bp_i 所在的独立受检退出).若对比 ets_fail_i 和 ets_pass_i 时未能确定 bp_i,则 $Begin\_Count\_bp_i$ 和 $End\_Count\_bp_i$ 的值为 0.公式(3)中的$Begin_diff_x^i$和$End_diff_x^i$分别代表 e_i 在 bp_i、$statements_x^i$前进入和退出的独立受检数量的差值,以公式(4)和公式(5)计算:

$$Begin\_diff_x^i = Begin\_Count\_bp_i - Begin\_Count_x^i \tag{4}$$

$$End\_diff_x^i = End\_Count\_bp_i - End\_Count_x^i \tag{5}$$

下面对公式(3)、公式(4)和公式(5)的设计进行详细解释.公式(3)表明,如果满足三类条件中的任意一个,则$statements_x^i$被判定属于$Suspicious_Group_i$.其中,第一类条件$End_diff_x^i > 0$代表$statements_x^i$在 bp_i 前被执行,这部分语句的执行可能导致 e_i 的执行状态受到感染,并在 bp_i 被观察到.因此,满足该条件的$statements_x^i$被包含在$Suspicious_Group_i$中,成为可疑语句.第二类条件中,Sit_i代表分歧点确定阶段(第 2.3.2 小节)中捕获 bp_i 对应的子情况,例如,若 bp_i 在子情况-2.2 下被捕获,则 Sit_i 的值为 2.2,依此类推.当 bp_i 在子情况-2.2、子情况-2.3 或子情况-2.4 下被捕获时,即使$End_diff_x^i$ 为 0,$statements_x^i$仍被认为包含于$Suspicious_Group_i$中,成为可疑语句.具体而言,$End\_diff_x^i$ 为 0 可能意味着$statements_x^i$位于 bp_i 与 bp_i 之后的第一次独立受检退出之间,也可能意味着 ets_fail_i 中不存在任何独立受检.考虑到情况 2 下捕获的 bp_i 是基于对独立受检数量而非异常触发信息的分析得到,在上述两种可能中,缺陷语句同样可能存在于满足$End_diff_x^i = 0$的$statements_x^i$中.第三类条件针对的是子情况 2.5,在该类情况中,由于仅 ets_fail_i 中存在独立受检,方法认为缺陷语句在 e_i 首次进入独立受检前被执行.因此,限定须同时满足$End_diff_x^i = 0$和$Begin_diff_x^i = 0$两个条件,以进一步缩小被包含于$Suspicious_Group_i$中的程序语句规模.

2.4.2 面向 Tie 问题消除的语句风险值计算

在基于分歧点分析的语句风险值计算中,以$statements_x^i$为单位确定了$Suspicious_Group_i$,公式(1)的前半部分由此得以计算.由于$statements_x^i$可能包含若干条被执行语句,需要对这些语句的风险值进行进一步区分.为此,计算程序语句 s 的辅助风险值$Assist_sus_s$ (公式(1)的后半部分).首先应用一项现有的行级缺陷风险值评估技术,为每条程序语句 s 计算风险值 $Origin\_Assist\_Sus_s$.$Assist_sus_s$通过将 $Origin\_Assist\_Sus_s$ 归一化至[0,1)区间得到,具体计算方式如公式(6)所示:

$$Assist\_Sus_s = \frac{Original\_Assist\_Sus_s - Min\_Original\_Assist\_Sus}{Max\_Original\_Assist\_Sus - Min\_Original\_Assist\_Sus + 1} \tag{6}$$

其中,$Max_Original_Assist_Sus$和$Min_Original_Assist_Sus$分别是所有$Original_Assist_Sus_s$的最大值、最小值.在本小节面向 Tie 问题消除的语句风险值计算中,任何工作在语句粒度的缺陷定位技术都可以被采用.因其轻量、经典的特性,INSPECT 采用 SBFL 技术用于该阶段.尽管单独用 SBFL 技术定位缺陷存在局限性,但其在 INSPECT 方法中的主要作用是在分析异常触发信息分歧点得到语句的主要风险值后,辅助性地对具有相同主要风险值的语句作进一步区分.

3 实验设计

本章对所提出的基于异常检查点植入的缺陷定位方法 INSPECT 进行有效性验证,包括研究问题、参数设置、数据集和评估指标几个方面.

3.1 研究问题（Research Questions, RQ）

为了检验 INSPECT 在缺陷定位任务中的表现(包括缺陷定位有效性和相同风险值语句 Tie 消除能力),分析 INSPECT 中不同组件、参数对整体方法的影响,设置如下研究问题:

- **RQ1:** INSPECT 方法是否具有良好的缺陷定位有效性?
  INSPECT 方法的重点在于在程序异常处理语句缺失的情境下,仍然能够挖掘并利用程序异常信息辅助缺陷定位任务的潜力,从而为缺陷语句赋予更高的风险值.将 INSPECT 方法与当前该领域的 SOTA (State-Of-The-Art)方法 SmartFL 进行对比,检验 INSPECT 是否能够将真正含有缺陷的语句置于所输出排序列表的靠前位置.
- **RQ2:** INSPECT 方法能否有效缓解相同风险值语句造成的 Tie 问题?
  软件缺陷定位技术常常为多条程序语句赋予相同的风险值,这些语句将形成一个个 Tie.特别是当真正含有缺陷的语句也位于 Tie 中时,缺陷定位的有效性将受到威胁.此时,若 Tie 的规模(即 Tie 中包含的语句数量)越小,则造成的影响也相应越小.本研究问题关注 INSPECT 方法对 Tie 问题的缓解能力,即能否减少所输出风险值排序列表中与缺陷语句具有相同风险值的其他语句的数量.
- **RQ3:** 检查点植入密度如何影响 INSPECT 方法的效果?
  如第 2.1 节中所述,在 INSPECT 方法中,所植入检查点的密度是决定方法能否有效捕获程序执行状态分歧点的关键因素之一.若检查点植入密度过大,可能导致方法收集到的异常信息规模较大,进而造成难以有效分析所收集的信息,并削弱异常信息相较其他传统缺陷定位信息源的优势;若检查点植入密度过小,可能导致方法难以有效捕获程序执行状态分歧点,进而影响定位有效性.本研究问题在方法默认 20%检查点植入密度之外,研究在更多密度(1%、5%、10%、25%)下方法的表现.
- **RQ4:** 程序语句风险值计算方法如何影响 INSPECT 方法的效果?
  在收集得到异常触发流并进行分析后,INSPECT 设计了具体的算法为程序语句赋予风险值(如第 2.4 节所述).本研究问题分析该算法设计的合理性,为该算法设计不同的变体,对比不同变体与原有设计在缺陷定位任务中的表现.

3.2 参数设置

如第 2.4.2 小节所述,INSPECT 采用 SBFL 技术为每条程序语句 s 计算辅助风险值 $Assist\_Sus_s$.在实验中,选择 Crosstab[43]、Ochiai[44]、DStar[45]和 Naish2[46]四项 SBFL 技术用于这一过程,因为它们具有高度的代表性且被证实在同类技术中具有最好的效果[31,47].将上述四项 SBFL 技术分别应用于 INSPECT 中,分别产生 $INSPECT_C$、$INSPECT_O$、$INSPECT_D$ 和 $INSPECT_N$ 四种变体,以更加全面地进行实验.

3.3 数据集

为了充分评估 INSPECT 方法应用于不同类型错误程序时的缺陷定位能力及泛化能力,选取 Chart、Time、Math 和 Lang 四个开源项目构建实验数据集,这些项目均在缺陷定位领域最流行的数据集之一 Defects4J 中得到了广泛使用[48],且已被大量应用于对软件测试调试技术的评估和度量[49-52]. 其中,Chart 和 Time 程序中具有一定数量的异常处理代码,而 Math 和 Lang 程序中原有的异常处理代码相对较少.为了充分评估 INSPECT 对缺少异常处理语句的待测程序的泛化能力,并排除待测程序原有异常处理语句对实验的干扰,将忽略所选取项目中的异常处理代码,仅基于植入的检查点收集异常触发流信息.在上述四个项目的原始程序中,采用基于突变的方法注入缺陷,形成 480 个待测错误程序版本.此外,还在上述相同开源项目中选取 60 个真实缺陷版本纳入实验数据集,以评估 INSPECT 在真实环境下面向不同种类缺陷的定位能力.之所以采用的真实缺陷版本少于模拟缺陷版本,是因为 Defects4J 中上述开源项目的真实缺陷版本在目前均已过时,在这些基于不同 JDK 版本的缺陷版本中,最新被支持的 JDK 版本是 1.8,而其对应的开源项目到目前为止支持的最旧 JDK 版本是 11.将这些真实版本进行兼容适配需要手工逐一操作,将造成大量开销.

3.4 评估指标

选取三项在缺陷定位领域现有研究中得到广泛应用的实验度量指标:EXAM[53,54]、MRR(Mean Reciprocal Rank)[55,56]和 WSR(Wilcoxon Signed-Rank tests)[45,57].其中,EXAM 指标度量缺陷定位技术生成的语句风险值排序列表中位于缺陷语句之前的程序语句数量,MRR 指标度量缺陷语句在风险值排序列表中位次的倒数,WSR 指标度量 INSPECT 方法的缺陷定位表现超越现有最先进方法的显著性.

关于 EXAM 指标,实验按照通用做法为其度量 EXAM_Best、EXAM_Average 和 EXAM_Worst 三个方面,分别评估方法在最好、平均和最差情况下的缺陷定位效果(即如果缺陷语句在 Tie 中,则分别按照其在 Tie 中第一个、正中间、最后一个被检查进行计算).具体而言,EXAM_Best 仅考虑风险值严格大于缺陷语句的程序语句数量,对于与缺陷语句存在 Tie 关系(风险值相等)的语句,假设其会在缺陷语句之后被检查;与此相反,对于与缺陷语句存在 Tie 关系的语句,EXAM_Worst 假设其会在缺陷语句之前被检查,因而其考虑的是风险值大于等于缺陷语句的程序语句数量;EXAM_Average 代表在平均情况下计算的 EXAM 分数,缺陷语句的 EXAM 分数将被计算为所有与其风险值相等的程序语句的 EXAM 分数的平均值.对于全部三种 EXAM 指标计算方法,更低的值意味着更优的缺陷定位表现.

MRR 指标定义为第一个相关元素被找到的平均倒数排名,如公式(7)所示:

$$\mathrm{MRR}=\frac{1}{K}\sum_{i=1}^{K}\frac{1}{rank_i} \tag{7}$$

其中,K 表示实验中待测错误程序版本的数量,$rank_i$代表第 i 个错误版本中缺陷语句的风险值排名.MRR 中 $rank_i$的取值会受到 Tie 问题的影响,实验仅在最好情况下进行考虑 ,因为其余两种情况下的方法表现已在 EXAM 指标中得以评估.更高的 MRR 分数代表更优的缺陷定位表现.

WSR 指标用于评估实验结果的统计显著性,即:所提出 INSPECT 方法的缺陷定位表现是否显著优于现有最先进方法.实验设计两个单边备择假设:NB(Not Bad,即 INSPECT 的表现不低于基线方法)和 B(Better,即 INSPECT 的表现优于基线方法).对于基线方法生成的缺陷定位结果,NB 代表在定位缺陷语句前需要检查的程序语句数量大于等于 INSPECT 生成的结果,B 代表在定位缺陷语句前需要检查的程序语句数量严格大于 INSPECT 所生成的结果.这一指标同样仅考虑 Tie 问题发生时最好情况下的结果.

4 实验结果分析

4.1 INSPECT方法是否具有良好的缺陷定位有效性

为了评估所提出 INSPECT 方法的能力,本研究问题选取缺陷定位领域现有最先进技术 SmartFL 作为基线方法进行比较.虽然前期所提 EXPECT 方法的缺陷定位有效性已被证实超过 SmartFL,但其主要使用目标是那些原本就具备异常处理代码的待测程序,难以在论文主要面向的异常处理语句缺失场景下充分发挥作用,因此不将其作为基线方法.如第 3.2 节所述,实验创建了 $\mathrm{INSPECT_C}$、$\mathrm{INSPECT_O}$、$\mathrm{INSPECT_D}$ 和 $\mathrm{INSPECT_N}$ 四种变体,以更加充分地检验方法缺陷定位有效性.实验结果如表 2 和表 3 所示,其中第二列到第四列展示了 INSPECT 的四种变体和基线方法 SmartFL 在三个方面的 EXAM 指标值,括号里的百分数是前者相较于后者的提升幅度;第五列展示了各个方法的 MRR 指标值并在括号里给出了提升幅度;第六列和第七列展示了 INSPECT 四种变体在 WSR 指标的 NB 和 B 两项单边备择假设上的 P 值,其反映了拒绝备择假设对应的原假设的统计显著性.

表 2 INSPECT 方法缺陷定位有效性 (模拟缺陷)

方 法	EXAM_Best	EXAM_Average	EXAM_Worst	MRR	WSR_NB	WSR_B
SmartFL	**13.27**	**13.27**	**13.27**	**0.2206**	\	\

INSPECT$_C$	**0.63** (95.25%↑)	**5.85** (55.92%↑)	**11.06** (16.65%↑)	**0.9077** (311.47%↑)	**2.40e-102**	**5.49e-93**
INSPECT$_O$	**0.63** (95.25%↑)	**5.86** (55.84%↑)	**11.06** (16.65%↑)	**0.9091** (312.10%↑)	**2.40e-102**	**5.49e-93**
INSPECT$_D$	**0.63** (95.25%↑)	**5.85** (55.92%↑)	**11.07** (16.58%↑)	**0.9090** (312.06%↑)	**2.40e-102**	**5.49e-93**
INSPECT$_N$	**5.88** (55.69%↑)	**11.10** (16.35%↑)	**16.31** (-)	**0.8966** (306.44%↑)	**4.86e-101**	**2.47e-92**

表 3 INSPECT 方法缺陷定位有效性 (真实缺陷)

方 法	EXAM_Best	EXAM_Average	EXAM_Worst	MRR	WSR_NB	WSR_B
SmartFL	**12.86**	**12.86**	**12.86**	**0.2415**	\	\
INSPECT$_C$	**0.85** (93.39%↑)	**5.46** (57.54%↑)	**11.07** (13.92%↑)	**0.9257** (283.31%↑)	**2.18e-14**	**1.28e-12**
INSPECT$_O$	**0.85** (93.39%↑)	**5.46** (57.54%↑)	**11.07** (13.92%↑)	**0.9257** (283.31%↑)	**2.18e-14**	**1.28e-12**
INSPECT$_D$	**0.85** (93.39%↑)	**5.46** (57.54%↑)	**11.07** (13.92%↑)	**0.9257** (283.31%↑)	**2.18e-14**	**1.28e-12**
INSPECT$_N$	**34.58 (-)**	**39.19 (-)**	**43.80 (-)**	**0.9091** (276.44%↑)	**3.62e-14**	**1.28e-12**

以对表 2 的分析为例,可以发现,对于 EXAM 指标,INSPECT 四种变体在最好情况下相较于基线方法均取得了明显提升,提升幅度最高为 95.25%、最低为 55.69%;在平均情况下,INSPECT 四种变体相较于基线方法的提升幅度有一定下降,但最高仍达 55.92%、最低为 16.35%;在最差情况下,除 INSPECT$_N$ 外,其他三种变体的提升幅度均在 16%以上.在 MRR 指标上,INSPECT 四种变体相较于基线方法的提升在 306.44%到 312.10%之间.在 WSR 指标中,根据 NB 和 B 两项单边备择假设的 P 值,可以认为应当拒绝备择假设对应的原假设,从统计学的角度出发,INSPECT 的缺陷定位有效性显著超过了基线方法.在表 3 中,对于 EXAM 指标, INSPECT$_C$, INSPECT$_O$, INSPECT$_D$ 三种变体相较于基线方法均取得了明显提升,在最好、平均和最差情况下的提升幅度分别为 93.39%、57.54%和 13.92%,INSPECT$_N$ 的表现则相对较低. 在 MRR 指标上,INSPECT 四种变体相较于基线方法的提升在 276.44%到 283.31%之间.在 WSR 指标中,根据 NB 和 B 两项单边备择假设的 P 值,可以认为应当拒绝备择假设对应的原假设,从统计学的角度出发,INSPECT 的缺陷定位有效性显著超过了基线方法.综上所述,INSPECT$_C$、INSPECT$_O$ 和 INSPECT$_D$ 在各种指标下的缺陷定位有效性均明显超过现有最优方法.上述结果表明,在不含异常处理语句的待测错误程序上,INSPECT 能够取得相较现有 SOTA 方法更好的缺陷定位效果,将程序异常信息这一有效的缺陷定位信息源泛化到了更加普适的真实环境.

4.2 INSPECT方法能否有效缓解相同风险值语句造成的Tie问题

本研究问题讨论 INSPECT 受 Tie 问题影响的程度,将 INSPECT$_C$、INSPECT$_O$、INSPECT$_D$ 和 INSPECT$_N$ 四种变体与其在程序语句风险值评估任务中分别引入以缓解 Tie 问题的四种 SBFL 技术(即 Crosstab、Ochiai、DStar 和 Naish2)进行对比,以评估 INSPECT 对 Tie 问题的处理能力.实验结果如表 4 和表 5 所示,其中第二行展示了四种 SBFL 技术(为方便表格展示,记为 *M*)在数据集各个待测错误版本上的平均 Tie 规模,第三行展示了分别配置不同 *M* 的 INSPECT 变体的平均 Tie 规模,第四行是 INSPECT$_M$ 相较于 *M* 的提升幅度.

表 4 INSPECT 方法对 Tie 问题的缓解能力 (模拟缺陷)

M	Crosstab	Ochiai	DStar	Naish2
SBFL 技术 *M*	208.86	363.70	363.70	208.86
INSPECT$_M$	10.43	10.44	10.43	10.43
提升幅度	95.01%	97.13%	97.13%	95.01%

表 5 INSPECT 方法对 Tie 问题的缓解能力 (真实缺陷)

M	Crosstab	Ochiai	DStar	Naish2
SBFL 技术 *M*	160.28	284.18	284.22	160.28
INSPECT$_M$	9.22	9.22	9.22	9.22
提升幅度	94.25%	96.76%	96.76%	94.25%

以对表 4 的分析为例,可以看出,四项 SBFL 技术在实验数据集上产生的缺陷定位结果中均存在较严重的 Tie 问题,例如,在 Crosstab 生成的程序语句风险值排序列表中,平均存在 208.86 条与真实缺陷语句具有相同风险值的语句,在 Ochiai、DStar 和 Naish2 的结果中,这一数字分别是 363.70、363.70 和 208.86.在真实的软件缺

陷定位任务中,过大规模的 Tie 会严重影响缺陷定位技术的实用价值,因为调试人员需要手工从 Tie 中逐条找出缺陷语句.与单纯使用 SBFL 技术相比,融合了 SBFL 的 INSPECT 方法能够明显降低 Tie 的规模,削减幅度最高可达 97.13%.从表 5 中可以看出,四项 SBFL 技术在真实缺陷数据集上的缺陷定位同样受制于 Tie 问题,例如,在 Crosstab 生成的程序语句风险值排序列表中,平均存在 160.28 条与真实缺陷语句具有相同风险值的语句.与单纯使用 SBFL 技术相比,融合了 SBFL 的 INSPECT 方法能够明显降低 Tie 规模,削减幅度最高可达 96.76%.

4.3 检查点植入密度如何影响INSPECT方法的效果

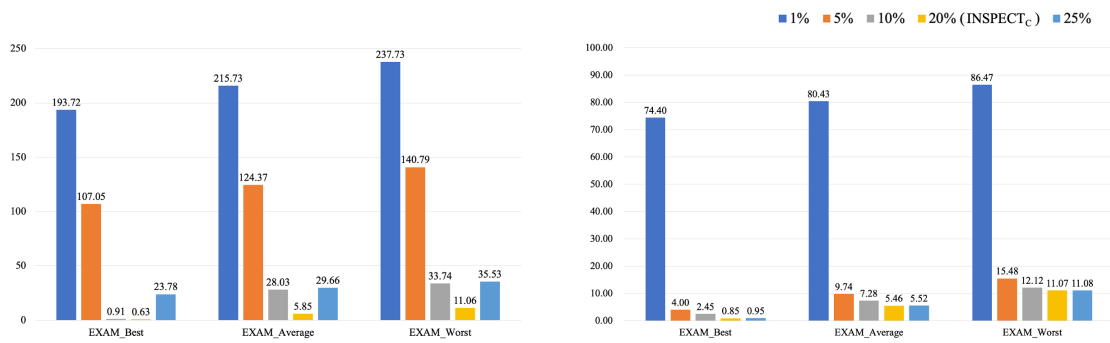
第 2.1 节介绍了植入检查点的具体方法,其中植入密度(即被选为检查点植入目标的程序函数比例)可能对方法的缺陷定位效果有关键影响.方法默认的植入密度为 20%,即顺序扫描程序中的所有函数,以固定间隔 4 为函数植入检查点.本研究问题额外选取 1%、5%、10%和 25%四个不同的密度值,即在检查点植入阶段分别以固定间隔 99、19、9 和 3 为顺序扫描的函数植入检查点.其中,1%、5%和 10%是比原配置 20%更低的密度值,25%则更高.实验不选取 50%、100%等进一步高的密度值,因为在这些密度设置下可能收集到规模过大的异常触发信息,导致异常信息失去相较现有方法广泛采用的传统信息源所具有的规模更小、分析难度更低的优势.本研究问题用最好、平均和最差情况下的 EXAM 指标来进行度量,实验结果如表 6、表 7 和图 5 所示(为控制变量便于比较,本研究问题将打破 Tie 阶段所采用的 SBFL 技术固定为 Crosstab,即对 INSPECT_C 变换不同的密度值).

表 6 不同检查点植入密度下 INSPECT 方法的缺陷定位有效性 (模拟缺陷)

密度值	EXAM_Best	EXAM_Average	EXAM_Worst
1%	193.72	215.73	237.73
5%	107.05	124.37	140.79
10%	0.91	28.03	33.74
20% (INSPECT_C)	0.63	5.85	11.06
25%	23.78	29.66	35.53

表 7 不同检查点植入密度下 INSPECT 方法的缺陷定位有效性 (真实缺陷)

密度值	EXAM_Best	EXAM_Average	EXAM_Worst
1%	74.40	80.43	86.47
5%	4.00	9.74	15.48
10%	2.45	7.28	12.12
20% (INSPECT_C)	0.85	5.46	11.07
25%	0.95	5.52	11.08



(a) 模拟缺陷 (b) 真实缺陷

图 5 不同检查点植入密度下 INSPECT 方法的缺陷定位有效性

以对表 6 的分析为例,可以发现,INSPECT_C 在三种情况下的缺陷定位有效性均随检查点植入密度的降低

而下降.以 EXAM_Average 为例,配置 20%密度值的原始 INSPECT 方法的指标值为 5.85,而在 10%、5%和 1%的密度值下,该值分别为 28.03、124.37 和 215.73,效果呈明显下降趋势.这一现象可能与临时植入检查点的特性有关,具体而言,不同于开发者自己设置的异常处理代码,INSPECT 自动植入的异常处理代码不具备开发者对程序逻辑的深入理解,使得临时植入用于缺陷定位的检查点需要相对更高的密度,以确保其能有效捕获程序错误执行状态,进而准确找到分歧点.此外,相较于原始设置更低的检查点植入密度也可能导致错误的执行状态在产生后更晚被捕获,使得方法更有可能为程序语句计算出一致的风险值,从而影响缺陷定位的有效性.当检查点的植入密度被设置为比原始密度 20%更高的 25%时,可以发现,INSPECT 的缺陷定位效果同样出现了下降(如在 20%密度时的 EXAM_Average 值为 5.85,而在 25%的密度时却上升到 29.66).这一现象的可能原因是,检查点植入密度的增加使得方法收集到的异常触发信息规模一并增加,更大规模的信息有可能引入与反映程序执行状态无关的冗余信息,同时也会增加分析难度.表 7 呈现的趋势与表 6 基本相同.同样以 EXAM_Average 为例,配置 20%密度值的原始 INSPECT 方法的指标值为 5.46, 而在 10%、5%和 1%的密度值下,该值分别为 7.28、9.74 和 80.43,效果呈明显下降趋势. 当检查点的植入密度被设置为比原始密度 20%更高的 25%时,可以发现,INSPECT 的缺陷定位效果同样出现下降,其 EXAM_Average 值上升至 5.52.

综上所述,过高或过低的检查点植入密度均会对 INSPECT 方法的缺陷定位有效性以及 Tie 问题缓解能力产生负面影响.在本研究问题选取的 1%、5%、10%、20%和 25%五种不同的检查点植入密度中,20% (即 INSPECT 原始设置)具有最优的表现.因此,当后续研究者采用本方法时,推荐采用 20%的检查点植入密度,因为更低或更高的密度均会导致效果的下降,同时更高的密度也会增加方法运行的时间和计算成本,与 INSPECT 方法提出时的轻量级初衷不符.

4.4 程序语句风险值计算方法如何影响INSPECT方法的效果

INSPECT 方法通过对异常触发流进行分析确定分歧点,并结合现有行级缺陷定位技术,共同为待测错误程序语句计算风险值,具体如第 2.4 节中的公式(1)所示.本研究问题对该风险值计算方法的合理性进行讨论,通过修改算法中sus_s^i的计算规则生成不同的变体,并对它们的缺陷定位效果进行比较.

根据前述介绍,为程序语句计算的风险值 $Suspiciousness_s$ 由$\sum_{i=1}^{n} sus_s^i$和$Assist_sus_s$两项组成,其中前者是主要部分,后者是辅助部分.在计算规则中,sus_s^i由程序语句 s 对可能含有缺陷的语句组$Suspicious_Group_i$的从属关系确定.考虑到自动化植入异常处理语句的特性,为了尽可能保证真正的缺陷语句被包含在$Suspicious_Group_i$中,INSPECT 在其风险值评估算法中不在$End_diff_x^i > 0$的情况下根据$End_diff_x^i$的具体值对程序语句受$Suspicious_Group_i$的包含情况作进一步区分.该设计在提高缺陷语句被排在风险值列表中较前位置概率的同时,也可能导致输出的排序列表中存在更大规模的 Tie.本研究问题对上述sus_s^i的计算规则进行修改,对程序语句组$statements_x^i$在$End_diff_x^i > 0$情况下是否受$Suspicious_Group_i$包含的判断条件作进一步限定,在原始算法之外额外形成四项算法变体,如表 8 所示.相较于原始算法仅要求$End_diff_x^i > 0$(该算法将满足该条件的语句全部纳入范围,因此记为 $Algo_{100\%}$),四种变体在这一条件的基础上继续添加了条件,要求$End_diff_x^i$小于等于End_Count_bpi的 80%、60%、40%和 20% (分别记为 $Algo_{80\%}$、$Algo_{60\%}$、$Algo_{40\%}$和 $Algo_{20\%}$).换句话说,失败测试用例执行至分歧点为止退出的独立受检数量与执行至$statements_x^i$为止退出的独立受检数量的差值$End_diff_x^i$分别不能超过前者的 80%、60%、40%和 20%.以 $Algo_{40\%}$为例,在满足$End_diff_x^i > 0$的基础上,$statements_x^i$还需要满足$End_Diff_x^i \le \max(End_Count_bpi * 0.4, 1)$这一条件,即只有离分歧点最近的 40% 的程序语句组会被判定为从属于$Suspicious_Group_i$.条件中 max(…, 1)这一设置是为了确保当$End_Count_bp_i$的值较小时,仍然能够存在满足条件的程序语句组.设计这四种方法变体的动机在于,假设临时植入的检查点能够及时识别出通过执行和失败执行中程序执行状态的差异,则可以仅将分歧点附近的程序语句组纳入$Suspicious_Group_i$,通过对其规模的缩减,减少缺陷定位结果受 Tie 问题的影响. 与 RQ3 相同,出于控制变量的目的,本研究问题里的所有算法变体均基于 $INSPECT_C$ 运行.因为 $Algo_{100\%}$新增的判断条件$End_Diff_x^i \le \max(End_Count_bp_i * 1, 1)$可以约减为$End_Diff_x^i \le End_Count_bp_i$(因为$End_diff_x^i > 0$时$End_Count_bp_i$的取值总大于等于 1);将上式与公式

(5) 结合可得 $End\_Count\_bp_i - End\_Count_x^i \le End\_Count\_bp_i$, 即 $End\_Count_x^i \ge 0$, 该条件显然总是成立. 因此,$Algo_{100\%}$的实际判断条件仍为$End_diff_x^i > 0$.

表 8 INSPECT 风险值评估算法变体设计

算法变体	判断条件
$Algo_{20\%}$	$0 < End\_Diff_x^i \le \max(End\_Count\_bp_i * 0.2, 1)$
$Algo_{40\%}$	$0 < End\_Diff_x^i \le \max(End\_Count\_bp_i * 0.4, 1)$
$Algo_{60\%}$	$0 < End\_Diff_x^i \le \max(End\_Count\_bp_i * 0.6, 1)$
$Algo_{80\%}$	$0 < End\_Diff_x^i \le \max(End\_Count\_bp_i * 0.8, 1)$
$Algo_{100\%}$ ($INSPECT_C$)	$End\_Diff_x^i > 0$

所有变体的 EXAM_Best、EXAM_Average 和 EXAM_Worst 指标值如表 9 和表 10 所示.

表 9 INSPECT 不同风险值评估算法变体的缺陷定位有效性 (模拟缺陷)

算法变体	EXAM_Best	EXAM_Average	EXAM_Worst
$Algo_{20\%}$	2.70	6.50	10.30
$Algo_{40\%}$	1.90	6.11	10.31
$Algo_{60\%}$	1.52	6.06	10.61
$Algo_{80\%}$	0.71	5.59	10.46
$Algo_{100\%}$ ($INSPECT_C$)	0.63	5.85	11.06

表 10 INSPECT 不同风险值评估算法变体的缺陷定位有效性 (真实缺陷)

算法变体	EXAM_Best	EXAM_Average	EXAM_Worst
$Algo_{20\%}$	0.73	4.26	7.78
$Algo_{40\%}$	0.80	4.60	8.40
$Algo_{60\%}$	0.83	5.08	9.33
$Algo_{80\%}$	0.83	5.27	9.70
$Algo_{100\%}$ ($INSPECT_C$)	0.85	5.46	11.07

由表 9 可见,在 EXAM_Best 指标下,$Algo_{100\%}$到 $Algo_{80\%}$、$Algo_{60\%}$、$Algo_{40\%}$和 $Algo_{20\%}$四项算法变体的指标值总体呈升高趋势(即缺陷定位效果下降),表明缩减$Suspicious_Group_i$的规模与对应算法变体的缺陷定位有效性总体呈负相关关系.在 EXAM_Average 和 EXAM_Worst 指标下,五个对比对象的值基本保持稳定(仅有小幅波动).在表 10 中, $Algo_{100\%}$到 $Algo_{80\%}$、$Algo_{60\%}$、$Algo_{40\%}$和 $Algo_{20\%}$四项算法变体的 EXAM_Best 和 EXAM_Average 指标值基本保持稳定(仅有小幅波动),但 EXAM_Worst 则呈明显的下降趋势,说明缩减$Suspicious_Group_i$的规模能在一定程度上减轻缺陷定位结果受 Tie 问题的影响.考虑到在模拟情况下,$Algo_{100\%}$的 EXAM_Best 指标值最好, EXAM_Average 和 EXAM_Worst 指标值与其他四种变体基本相当;在真实情况下,$Algo_{100\%}$的 EXAM_Best 和 EXAM_Average 指标值与其他四种变体基本相当, EXAM_Worst 的指标值虽然稍差,但先前的研究已经指出现实调试中,在 Worst 情况下才能定位到缺陷的情况比较少见,EXAM_Worst 指标只是为了对方法进行更全面度量的一种参考[7],所以,综合来看,不在$End_diff_x^i > 0$的情况下根据$End_diff_x^i$的具体值对程序语句受$Suspicious_Group_i$的包含情况作进一步区分,即 INSPECT 原始风险值计算方法 $Algo_{100\%}$,是更为合适的设计.

5 相关工作

基于程序频谱的缺陷定位是最具代表性的软件缺陷定位技术之一,此类方法收集失败测试用例和通过测试用例执行时的覆盖信息,构建以 a_{ef} (程序语句被失败测试用例覆盖的次数)、a_{np} (程序语句未被通过测试

用例覆盖的次数)等元素为代表的程序频谱,并设计风险值评估公式分析程序频谱与软件缺陷间的潜在关联,最终得到语句的具体风险值.围绕对此类公式的设计及评估,前期已有许多研究者陆续发表了相关工作,如由 Jones 等人提出的最经典的风险值评估公式之一 Tarantula[58],Wong 等人提出的实践最优的公式 DStar[45]和 Crosstab[43],以及被证明高有效性的 Ochiai[44]和 Naish2[46]等.此外,Yoo 等人还提出了基于遗传算法生成风险值评估公式的方法[59],Xie 等人[31]和 Rui 等人[46]对风险值公式从理论层面分析了优劣关系.

基于突变的缺陷定位与频谱缺陷定位同为最具代表性的缺陷定位技术之一,此类方法在为特定程序语句评估风险值时,首先对目标语句实行突变策略,生成多个程序突变版本,并在每个突变版本上运行测试用例,收集测试用例在突变版本上的执行结果;随后,分析对程序语句突变后测试用例执行结果受到的影响,通过设计数学化公式将此影响量化计算为语句的具体风险值.在现有的突变缺陷定位研究领域,Papadakis 等人提出的 Metallaxis[34]和 Moon 等人提出的 Muse[33]是具有代表性的技术.

动态切片是一种通过删除不相关部分缩减程序规模的技术,已有一些软件缺陷定位方法利用动态切片对待测程序进行简化,以更有效地分析缺陷语句与失败测试用例之间的关联.例如,Wen 等人采用动态切片和统计学方法提取程序元素之间的依赖关系并细化执行历史[60],Mao 等人利用动态切片捕捉程序实体执行对输出的影响,以优化缺陷定位效果[61].此外,亦有研究尝试将动态切片与基于频谱和基于突变的缺陷定位相结合,利用动态切片缩减频谱或突变信息规模,以降低分析难度.例如,Cao 等人利用动态切片生成混合切片谱以优化频谱缺陷定位过程[62],Chaleshtari 等人利用动态切片减少需要变异的语句数量,降低突变缺陷定位技术的执行开销[63].

近年,Zeng 等人提出一项基于语义信息的缺陷定位技术 SmartFL[36],该方法对程序语义信息、静态分析信息和动态执行跟踪信息进行建模,并利用基于概率的方法为程序语句评估风险值以实现缺陷定位.SmartFL 已被证实在有效性方面超过现有频谱及突变缺陷定位技术,因而被选择与本研究所提出的 INSPECT 方法进行比较.

尽管以上所述程序覆盖信息、突变信息和语义信息等信息源均在缺陷定位任务中具有各自的优势,但如第 1.1 节中所指出的,它们亦具有各自不容忽视的短板,如覆盖信息往往体量庞大、冗余、难以精准分析,且其仅从较为浅表的视角观测程序运行,因此许多语句可能具有完全相同的覆盖信息;突变信息的获得需要对突变后的源代码重新编译执行,时间和空间成本十分可观;语义信息的规模与程序执行跟踪信息的规模高度相关,当大量程序语句均被覆盖时其大小可能和传统覆盖信息接近.因此,找到一种更加轻量、更有针对性的缺陷定位信息源,并将其使用场景泛化到真实世界的软件调试场景,对于当前软件质量保障领域的研究十分重要.论文所提 INSPECT 方法通过自动植入异常处理语句, 使得异常触发流这一更轻量和更具针对性的信息源在异常处理语句缺失的情况下仍能用于缺陷定位,有效解决了前期方法对待测程序原本异常处理语句的高度依赖,提升了异常信息在真实世界缺陷定位任务中的泛化性.

6 总 结

针对软件缺陷定位领域面临的信息源庞杂、难以分析问题,以及前期所提基于异常触发流的缺陷定位技术 EXPECT 泛化性不足的挑战,论文提出一种基于异常检查点植入的缺陷定位技术 INSPECT.在待测错误程序缺失异常处理语句的情况下,设计自动化方法在程序中植入 Try-catch 块,使得异常触发信息这一轻量且能有效帮助定位软件缺陷的数据仍能顺利收集;基于伪正确版本的生成对所收集的异常触发信息及程序执行跟踪进行分析,确定失败执行和通过执行的分歧点;基于对分歧点的分析,为所有程序语句计算风险值,并采用行级缺陷定位技术对得到的风险值进行精化,缓解制约缺陷定位任务有效开展的 Tie 问题.实验结果表明,INSPECT 方法能够有效超过现有最优技术:在 Defects4J 的 540 个错误程序上,INSPECT 开展缺陷定位的 EXAM_Best、EXAM_Average 和 EXAM_Worst 指标值的提升幅度分别为 95.25%、55.92%和 16.65%(模拟缺陷),以及 93.39%、57.54%和 13.92%(真实缺陷),MRR 指标值的提升幅度为 311.47%(模拟缺陷)和 283.31%(真实缺陷),基于两个备择假设的显著性检验也证实了 INSPECT 相较于基线技术的提升.

在本工作中,为了让方法具有较高的泛化性,采用了按固定间隔的方式植入 Try-catch 块,且所植入 Try-catch 块对捕获异常信息的种类没有限制.下一步工作中,考虑引入被测程序逻辑设计和代码结构的先验知识,通过对待测程序进行深入理解,推荐出更有利于缺陷定位任务的检查点植入位置及植入方法,以进一步提升方法效果.

References:

[1] Ty Roush, Brian Bushard. CrowdStrike's Massive Global Tech Outage: Airlines, Hospitals, Banks, 911, Governments Impacted. https://www.forbes.com/sites/tylerroush/2024/07/19/crowdstrikes-massive-global-tech-outage-airlines-banks-911-state-services-impacted.

[2] Krasner H. Cost of poor software quality in the U.S.: a 2022 report. Vol.1: Austin, TX: Consortium for Information & Software Quality, 2022. 1-61.

[3] Yue L, Cui ZQ, Chen X, Wang RC, Li L. Statement Level Software Bug Localization Based on Historical Bug Information Retrieval. Ruan Jian Xue Bao/Journal of Software, 2024, 35(10): 4642–4661 (in Chinese).

[4] Li ZL, Chen X, Jiang ZW, Gu Q. Survey on information retrieval-based software bug localization methods. Ruan Jian Xue Bao/Journal of Software, 2021,32(2):247−276 (in Chinese).

[5] Yang YM, Xia X, Lo D, Bi TT, Grundy J, Yang XH. Predictive models in software engineering: Challenges and opportunities. ACM Transactions on Software Engineering and Methodology, 2022, 31(3): 1-72.

[6] Wu Y, Li Z, Liu Y, et al. Fatoc: bug isolation based multi-fault localization by using optics clustering. Journal of Computer Science and Technology, 2020, 35(5): 979- 998.

[7] Gao RZ, Wong W E. MSeer—An Advanced Technique for Locating Multiple Bugs in Parallel. IEEE Transactions on Software Engineering, 2019, 45(03): 301-318.

[8] Jiang JJ, Wang YM, Chen JJ, Lv DL, Liu MJ. Variable-based fault localization via enhanced decision tree. ACM Transactions on Software Engineering and Methodology, 2023, 33(2): 1-32.

[9] Wen M, Xie Z, Luo K, et al. Effective isolation of fault-correlated variables via statistical and mutation analysis. IEEE Transactions on Software Engineering, 2022, 49(4): 2053-2068.

[10] Li Z, Wu YH, Wang HF, Chen X, Liu Y. Review of Software Multiple Fault Localization Approaches. Chinese Journal of Computers, 2022, 45(2): 256 – 288 (in Chinese).

[11] Widyasari R, Prana G A A, Haryono S A, Wang SW, Lo D. Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects. Empirical Software Engineering, 2022, 27(6): 147.

[12] Song Y, Xie XY, Zhang XH, Liu QM, Gao RZ. Evolving ranking-based failure proximities for better clustering in fault isolation. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2022: 1-13.

[13] Wen M, Chen JJ, Tian YQ, Wu RX, Hao D, Han S, Cheung SC. Historical spectrum based fault localization. IEEE Transactions on Software Engineering, 2019, 47(11): 2348-2368.

[14] Wang HF, Li Z, Liu Y, Chen X, Paul D, Cai YXY, Fan LX. Can higher-order mutants improve the performance of mutation-based fault localization?. IEEE Transactions on Reliability, 2022, 71(2): 1157-1173.

[15] Ghanbari A, Thomas D G, Arshad M A, Rajan H. Mutation-based fault localization of deep neural networks. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering. Luxembourg: IEEE, 2023: 1301-1313.

[16] Du B, Han BL, Liu HY, Chang ZX, Liu Y, Chen X. Neural-MBFL: Improving mutation-based fault localization by neural mutation. In: 2024 IEEE 48th Annual Computers, Software, and Applications Conference. Osaka: IEEE, 2024: 1274-1283.

[17] Xu XF, Debroy V, Eric Wong W, Guo DH. Ties within fault localization rankings: Exposing and addressing the problem. International Journal of Software Engineering and Knowledge Engineering, 2011, 21(06): 803-827.

[18] Xie XY, Xu BW. Essential spectrum-based fault localization. 1st ed., Singapore: Springer, 2021: 1-172.

[19] Sarhan Q I, Vancsics B, Beszédes Á. Method calls frequency-based tie-breaking strategy for software fault localization. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation. Luxembourg: IEEE, 2021: 103-113.

[20] Zhang XH, Song Y, Xie XY, Xin Q, Xing CL. Do not neglect what's on your hands: localizing software faults with exception trigger stream. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2024: 982-994.

[21] Wirfs-Brock R J. Toward exception-handling best practices and patterns. IEEE software, 2006, 23(5): 11-13.

[22] Sena D, Coelho R, Kulesza U, Bonifácio R. Understanding the exception handling strategies of Java libraries: An empirical study. In: Proceedings of the 13th International Conference on Mining Software Repositories. New York: ACM, 2016: 212-222.

[23] Jia XY, Chen SQ, Zhou XQ, Li XT, Yu R, Chen X. Where to handle an exception? Recommending exception handling locations from a global perspective. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension. Madrid: IEEE, 2021: 369-380.

[24] Mishra A, Otaiwi Z. DevOps and software quality: A systematic mapping. Computer Science Review, 2020, 38(100308): 1-14.

[25] Wang B, Lu SR, Jiang JJ, Xiong YF. Survey of Dynamic Analysis Based Program Invariant Synthesis Techniques. Ruan Jian Xue Bao/Journal of Software, 2020, 31(6): 1681-1702 (in Chinese).

[26] Lukasczyk S, Kroiß F, Fraser G. An empirical study of automated unit test generation for Python. Empirical Software Engineering, 2023, 28(2): 1-46.

[27] Tizpaz-Niari S, Monjezi V, Wagner M, Darian S, Reed K, Trivedi A. Metamorphic testing and debugging of tax preparation software. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Society. Melbourne: IEEE, 2023: 138-149.

[28] Hirsch T, Hofer B. A systematic literature review on benchmarks for evaluating debugging approaches. Journal of Systems and Software, 2022, 192(111423): 1-17.

[29] Voas J M. PIE: A dynamic failure-based technique. IEEE Transactions on software Engineering, 1992, 18(8): 717-727.

[30] Song Y, Xie XY, Liu QM, Zhang XH, Wu X. A comprehensive empirical investigation on failure clustering in parallel debugging. Journal of Systems and Software, 2022, 193(111452): 1-17.

[31] Xie XY, Chen T Y, Kuo F C, Xu BW. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on software engineering and methodology, 2013, 22(4): 1-40.

[32] Torkashvan R, Parsa S, Vaziri B. SBFL fault localization considering fault-proneness. Journal of Systems and Software, 2025, 223(112363): 1-14.

[33] Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE 7th International Conference on Software Testing, Verification and Validation. Cleveland: IEEE, 2014: 153-162.

[34] Papadakis M, Le Traon Y. Metallaxis-FL: mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5-7): 605-628.

[35] Liu HY, Li Z, Han BL, Liu YT, Chen X, Liu Y. Delta4Ms: Improving mutation-based fault localization by eliminating mutant bias. Software Testing, Verification and Reliability, 2024, 34(4): 1-32.

[36] Zeng MH, Wu YQ, Ye ZT, Xiong YF, Zhang X, Zhang L. Fault localization via efficient probabilistic modeling of program semantics. In: Proceedings of the 44th International Conference on Software Engineering. New York: ACM, 2022: 958-969.

[37] Cabral B, Marques P. Exception handling: A field study in java and. net. In: Proceedings of the 21st European Conference on Object-Oriented Programming. Heidelberg: Springer, 2007: 151-175.

[38] Gosling J. The Java language specification. 7th ed., Redwood City: Oracle America, 2011: 1-670.

[39] Melo H, Coelho R, Treude C. Unveiling exception handling guidelines adopted by java developers. In: Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering. Hangzhou: IEEE, 2019: 128-139.

[40] GitHub Repository Gson. Available: https://github.com/google/gson, accessed on: March 2024.

[41] GitHub Repository Math. Available: https://github.com/apache/commons-math, accessed on: March 2024.

[42] GitHub Repository Spring. Available: https://github.com/spring-projects, accessed on: March 2024.

[43] Eric Wong W, Debroy V, Xu DX. Towards better fault localization: A crosstab-based statistical approach. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 2011, 42(3): 378-396.

[44] Abreu R, Zoeteweij P, Van Gemund A J C. An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing. Riverside: IEEE, 2006: 39-46.

[45] Eric Wong W, Debroy V, Gao RZ, Li YH. The DStar method for effective software fault localization. IEEE Transactions on Reliability, 2013, 63(1): 290-308.

[46] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering and Methodology, 2011, 20(3): 1-32.

[47] Yoo S, Xie XY, Kuo F C, Chen TY, Harman M. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. ACM Transactions on Software Engineering and Methodology, 2017, 26(1): 1-30.

[48] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York: ACM, 2014: 437–440..

[49] Xu TT, Chen LS, Pei Y, Zhang T, Pan MX, Furia CA. Restore: Retrospective fault localization enhancing automated program repair. IEEE Transactions on Software Engineering, 2020, 48(1): 309-326.

[50] Zhang MS, Li X, Zhang LM, Khurshid S. Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2017: 261-272.

[51] Song Y, Zhang XH, Xie XY, Chen SQ, Liu QM, Gao RZ. SURE: A Visualized Failure Indexing Approach Using Program Memory Spectrum. ACM Transactions on Software Engineering and Methodology, 2024, 33(8): 1-43.

[52] Qian J, Ju XL, Chen X. GNet4FL: effective fault localization via graph convolutional neural network. Automated Software Engineering, 2023, 30(2): 16.

[53] Eric Wong W, Debroy V, Golden R, Xu XF, Thuraisingham B. Effective software fault localization using an RBF neural network. IEEE Transactions on Reliability, 2011, 61(1): 149-169.

[54] Lei Y, Xie H, Zhang T, Yan M, Xu Z, Sun CN. Feature-fl: Feature-based fault localization. IEEE Transactions on Reliability, 2022, 71(1): 264-283.

[55] Chen A R, Chen T H, Chen JJ. How useful is code change information for fault localization in continuous integration?. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, New York: ACM, 2022: 1-12.

[56] Miryeganeh N, Hashtroudi S, Hemmati H. GloBug: Using global data in fault localization. Journal of Systems and Software, 2021, 177(110961): 1-20.

[57] Xie H, Lei Y, Yan M, Yu Y, Xia X, Mao XG. A universal data augmentation approach for fault localization. In: Proceedings of the 44th International Conference on Software Engineering. New York: ACM, 2022: 48-60.

[58] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. New York: ACM, 2005: 273-282.

[59] Yoo S. Evolving human competitive spectra-based fault localisation techniques. In: Proceedings of the 4th International Symposium on Search Based Software Engineering. Berlin: Springer, 2012: 244-258.

[60] Wen WZ. Software fault localization based on program slicing spectrum. In: 2012 34th International Conference on Software Engineering. Zurich: IEEE, 2012: 1511-1514.

[61] Mao XG, Lei Y, Dai ZY, Qi YH, Wang CS. Slice-based statistical fault localization. Journal of Systems and Software, 2014, 89(1): 51-62.

[62] Cao HL, Wang F, Deng ML, Li L. The improved dynamic slicing for spectrum-based fault localization. PeerJ Computer Science, 2022, 8(e1071): 1-26.

[63] Bayati Chaleshtari N, Parsa S. SMBFL: slice-based cost reduction of mutation-based fault localization. Empirical Software Engineering, 2020, 25(5): 4282-4314.

附中文参考文献:

[3] 岳雷, 崔展齐, 陈翔, 王荣存, 李莉. 基于历史缺陷信息检索的语句级软件缺陷定位方法. 软件学报, 2024, 35(10): 4642－4661.

[4] 李政亮, 陈翔, 蒋智威, 顾庆. 基于信息检索的软件缺陷定位方法综述. 软件学报, 2021, 32(2): 247－276.

[10] 李征, 吴永豪, 王海峰, 陈翔, 刘勇. 软件多缺陷定位方法研究综述. 计算机学报, 2022, 45(2): 256 – 288.
[25] 王博, 卢思睿, 姜佳君, 熊英飞. 基于动态分析的软件不变量综合技术. 软件学报, 2020, 31(6): 1681-1702.