软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(10):4642−4661 [doi: 10.13328/j.cnki.jos.006980]
©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

基于历史缺陷信息检索的语句级软件缺陷定位方法*

岳　雷[1], 崔展齐[1], 陈　翔[2], 王荣存[3], 李　莉[1]

[1](北京信息科技大学 计算机学院, 北京 100101)
[2](南通大学 信息科学技术学院, 江苏 南通 226019)
[3](中国矿业大学 计算机科学与技术学院, 江苏 徐州 221116)
通信作者: 崔展齐, E-mail: czq@bistu.edu.cn

摘　要: 软件在开发和维护过程中会产生大量缺陷报告, 可为开发人员定位缺陷提供帮助. 基于信息检索的缺陷定位方法通过分析缺陷报告内容和源码文件的相似度来定位缺陷所在位置, 已在文件、函数等粗粒度级别上取得了较为精确的定位效果, 但由于其定位粒度较粗, 仍需要耗费大量人力和时间成本来检查可疑文件和函数片段. 为此, 提出一种基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator, 首先检索出与被测程序缺陷报告相似度较高的历史缺陷报告, 并提取其中的历史缺陷语句; 然后根据被测程序源码文件与缺陷报告的文本相似度检索可疑文件, 并提取其中的可疑语句; 最后计算可疑语句与历史缺陷语句的相似度, 并进行降序排列, 以定位缺陷语句. 为评估 STMTLocator 的缺陷定位性能, 使用 Top@*N*、*MRR* 等评价指标在基于 Defects4J 和 JIRA 构建的数据集上进行对比实验. 实验结果表明, 相比静态缺陷定位方法 BugLocator, STMTLocator 在 *MRR* 指标上提升近 4 倍, 在 Top@1 指标上多定位到 7 条缺陷语句; 相比动态缺陷定位方法 Metallaxis 和 DStar, STMTLocator 完成一个版本缺陷定位平均消耗的时间减少 98.37% 和 63.41%, 且具有不需要设计和执行测试用例的显著优势.

关键词: 软件调试; 缺陷定位; 信息检索; 缺陷报告
中图法分类号: TP311

中文引用格式: 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. http://www.jos.org.cn/1000-9825/6980.htm
英文引用格式: 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). http://www.jos.org.cn/1000-9825/6980.htm

Statement Level Software Bug Localization Based on Historical Bug Information Retrieval

YUE Lei[1], CUI Zhan-Qi[1], CHEN Xiang[2], WANG Rong-Cun[3], LI Li[1]
[1](School of Computer Science, Beijing Information Science and Technology University, Beijing 100101, China)
[2](School of Information Science and Technology, Nantong University, Nantong 226019, China)
[3](School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China)

Abstract: A large number of bug reports are generated during software development and maintenance, which can help developers to locate bugs. Information retrieval based bug localization (IRBL) analyzes the similarity of bug reports and source code files to locate bugs, achieving high accuracy at the file and function levels. However, a lot of labor and time costs are consumed to find bugs from suspicious files and function fragments due to the coarse location granularity of IRBL. This study proposes a statement level software bug localization method based on historical bug information retrieval, STMTLocator. Firstly, it retrieves historical bug reports which are similar to the bug report of the program under test and extracts the bug statements from the historical bug reports. Then, it retrieves the

* 基金项目: 江苏省前沿引领技术基础研究专项 (BK202002001); 国家自然科学基金 (61702041); 北京信息科技大学“勤信人才”培育计划 (QXTCP C201906)
收稿时间: 2023-02-06; 修改时间: 2023-03-28; 采用时间: 2023-06-05; jos 在线出版时间: 2023-09-06
CNKI 网络首发时间: 2023-09-08

suspicious files according to the text similarity between the source code files and the bug report of the program under test, and extracts the suspicious statements from the suspicious files. Finally, it calculates the similarity between the suspicious statements and the historical bug statements, and arranges them in descending order to localize bug statements. To evaluate the bug localization performance of STMTLocator, comparative experiments are conducted on the Defects4J and JIRA dataset with Top@*N*, *MRR*, and other evaluation metrics. The experimental results show that STMTLocator is nearly four times than the static bug localization method BugLocator in terms of *MRR* and locates 7 more bug statements for Top@1. The average time used by STMTLocator to locate a bug version is reduced by 98.37% and 63.41% than dynamic bug localization methods Metallaxis and DStar, and STMTLocator has a significant advantage of not requiring the construction and execution of test cases.

Key words: software debugging; bug localization; information retrieval; bug report

软件规模和复杂程度的不断提高, 为软件质量保障带来了严峻挑战. 软件调试是保障软件质量和可靠性的重要手段, 主要包括缺陷定位和缺陷修复两个步骤. 其中缺陷定位旨在辅助开发人员快速确认缺陷所在位置, 是软件调试过程中最为关键的环节[1−3]. 根据是否需要执行源程序, 现有的软件缺陷定位方法可分为动态缺陷定位[4]和静态缺陷定位[5]. 其中, 动态缺陷定位通过收集和分析测试用例的执行行为和运行结果, 确定缺陷所在位置, 其定位粒度较细、精度较高, 但需要耗费大量的人力和时间开销来设计和执行测试用例; 静态缺陷定位主要通过分析程序结构、缺陷报告等程序静态信息来识别出可疑度较高的模块, 相较于动态缺陷定位具有成本低、时间开销小的优势, 但定位粒度较粗[6].

常用的静态缺陷定位方法主要有基于信息检索的缺陷定位[7−9]和基于历史的缺陷定位[10]等. 其中, 基于信息检索的缺陷定位 (information retrieval-based bug localization, IRBL) 因为其计算成本低、能够充分利用缺陷报告提供的信息而被广泛使用[6]. 根据缺陷定位粒度, IRBL 可分为文件级和函数级的缺陷定位. 其中, 文件级的 IRBL 虽然具有较高的准确率[11], 但开发人员仍需要耗费大量时间按照文件的可疑度和文件中代码语句的默认顺序确认缺陷语句所在位置. 相比于文件级的缺陷定位, 因为函数中包含的代码文本和词汇较少, 导致函数级别的缺陷定位效果有所下降. 另外, 即使将存在缺陷的函数排在第一, 开发人员同样需要逐行检查函数中的语句. 研究表明, 开发人员在检查给定列表中的少量语句未发现缺陷后便会失去耐心[12], 文件级和函数级的粗粒度 IRBL 并不能满足软件缺陷定位的实际需求. 若直接将 IRBL 应用于语句级缺陷定位, 则会由于单条语句中包含的文本和词汇过少, 语句间的语义可区分性差, 导致语义映射结果误报率增大, 缺陷定位准确率将大幅下降.

为此, 本文提出了基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator. 首先, 检索出与被测程序缺陷报告相似度较高的历史缺陷报告, 并提取出其中的历史缺陷语句; 然后, 使用信息检索技术从被测程序的源码文件中检索出可疑度较高的文件, 将其中代码作为可疑语句; 最后, 计算可疑语句与历史缺陷语句之间的相似度, 以进行语句级缺陷定位. 我们实现了 STMTLocator 的原型工具, 并在基于 Defects4J 缺陷代码库 (https://github.com/rjust/defects4j) 和 JIRA (https://issues.apache.org/jira/secure/Dashboard.jspa) 中的历史缺陷报告所构建的数据集上进行了实验. 实验结果表明, 与静态缺陷定位方法 BugLocator 相比, STMTLocator 具有更好的缺陷定位性能. 对于 Top@*N* 指标, 当 $N = 1$ 时, STMTLocator 比 BugLocator 多定位到 7 条缺陷语句; 对于 *MRR* 指标, STMTLocator 比 BugLocator 提升了近 4 倍. 此外, 与动态缺陷定位方法相比, STMTLocator 相比 Metallaxis 和 DStar 完成一个版本缺陷定位所需时间平均减少了 98.37% 和 63.41%, 且不需要构建和执行测试用例集.

本文的主要贡献如下.

- 提出了基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator, 融合缺陷报告、代码语句的文本及结构特征进行语句级缺陷定位, 细化了基于信息检索的静态缺陷定位技术的定位粒度, 提升了定位精度.
- 为评估 STMTLocator 方法的有效性, 实现了原型工具, 并在 Defects4J 数据集上与静态缺陷定位方法 BugLocator 以及动态缺陷定位方法 Metallaxis 和 DStar 进行了对比实验.

本文第 1 节介绍相关软件缺陷定位方法. 第 2 节描述本文的研究动机. 第 3 节描述基于历史缺陷信息检索的语句级软件缺陷定位方法. 第 4 节介绍实验设计及相关评价指标, 并对实验结果进行分析. 第 5 节对 STMTLocator 的参数取值、局限性和有效性进行讨论并分析缺陷类型对 STMTLocator 方法的影响. 最后在第 6 节对本文的工作进行了总结并展望未来工作.

© 中国科学院软件研究所 http://www.jos.org.cn

1 相关工作

根据是否需要执行源程序, 软件缺陷定位方法可以分为静态缺陷定位和动态缺陷定位. 静态缺陷定位不需要执行程序, 仅通过对程序代码、结构以及相关文档进行静态分析来确定缺陷所在位置, 所需时间开销小, 但是缺陷定位精度较低. 动态缺陷定位则需要使用测试用例驱动程序运行, 收集程序运行信息并用于缺陷定位, 与静态缺陷定位方法相比所需时间开销大, 但定位精度更高. 在本节中将对静态缺陷定位和动态缺陷定位相关工作进行介绍.

1.1 静态缺陷定位方法

静态缺陷定位多采用基于信息检索的方法, 通过分析代码和缺陷报告的特征, 计算缺陷报告与代码的相关性, 并根据相关性进行缺陷定位[13].

基于信息检索的缺陷定位工作关注使用不同的信息检索模型来提升缺陷定位精度. Lukins 等人[14]在 2008 年首次提出基于信息检索的缺陷定位方法, 即使用 LDA (latent Dirichlet allocation) 模型来自动化定位缺陷. 该方法首先基于源代码文件构建的语料库生成 LDA 模型, 其中 LDA 模型的输出是词主题概率分布和主题-文档分布, 然后使用缺陷报告作为查询来计算其与所生成 LDA 模型之间的相似性. 接着, Rao 等人[15]将 VSM 模型 (vector space model)、SUM 模型 (smoothed unigram model)、一元模型 (unigram model) 和 LDA 模型等一系列通用的文本信息检索方法应用在缺陷定位问题上, 结果表明 SUM、VSM 等模型具有更好的效果. 考虑到经典 VSM 对于不同长度的文本的评价并不公平, 即较短的文本更容易具有较高的相似度, Zhou 等人[16]提出了 rVSM 模型 (revised vector space model), 根据文本的长度来优化 VSM 模型, 以进行更精确的缺陷定位. 张文等人[17]提出了一种方法级的细粒度软件缺陷定位方法 MethodLocator, 首先对缺陷报告和源代码文件中的方法利用词向量 (Word2Vec) 和 TF-IDF (term frequency and inverse document frequency) 结合的方法进行向量表示, 然后根据方法之间的相似度对方法进行扩充, 最后计算扩充后的方法与缺陷报告间的余弦相似度并进行排序, 排名越高的方法存在缺陷的可能性越大.

另一部分工作则通过融合更多的程序静态信息来提升缺陷定位性能. Wu 等人[18]提出了一种使用缺陷报告中的崩溃堆栈信息来定位缺陷的方法 CrashLocator, 通过静态调用图中相关的函数来扩展崩溃堆栈, 以推断出可能导致崩溃的执行轨迹, 然后在可能导致崩溃的执行轨迹中计算每个函数的可疑度, 根据函数的可疑度进行排名, 向开发人员提供进一步的筛选建议. Wang 等人[19]将程序的历史版本、相似缺陷报告以及结构信息进行结合, 提出了基于信息检索的文件级缺陷定位方法 AmaLgam. Chen 等人[20]提出了使用日志中的信息来重新构造执行路径以改进基于信息检索的缺陷定位方法 Pathidea, 使用静态分析创建文件级别的调用图, 并根据缺陷报告中的堆栈和日志信息来重新构建执行路径, 对路径上的文件赋予路径分数, 同时将路径分数和文本相似度以及日志分数进行加权组合, 最终得到每个文件的可疑度分数, 可疑度越大的文件存在缺陷的可能性越大. Takahashi 等人[21]研究了代码异味对软件缺陷定位的影响, 他们认为具有代码异味的模块更易于变化且更可能存在缺陷, 因此将代码异味的严重等级与信息检索中的文本相似度相结合, 不仅能够识别出与缺陷报告相似的模块, 还能识别出更有可能包含缺陷的模块.

在基于信息检索的缺陷定位基础上, 基于历史的缺陷定位使用程序的历史版本信息辅助进行缺陷定位. Sisman 等人[22]分析源码文件在过去版本中的修改频率计算出文件存在缺陷的先验概率, 通过结合基于信息检索的缺陷定位方法, 提升文件级缺陷定位性能. Tantithamthavorn 等人[23]认为一个缺陷文件被修复时, 与其共同被更改的文件也与缺陷相关, 据此提出了一种使用共同更改历史记录来提升缺陷定位性能的方法, 在现有文件级缺陷定位结果基础上, 分析缺陷定位结果中源码文件之间的共同变化历史, 计算源码文件的共同变化得分, 以更新源码文件存在缺陷的可能性. Wang 等人[24]通过收集代码的相似历史缺陷、代码修复历史等信息, 使用多维度卷积神经网络模型提取对应特征, 然后将提取出的特征作为卷积神经网络的输入, 以学习特征和缺陷位置之间的非线性关系, 最后模型根据源码文件的历史信息等特征判断文件存在缺陷的可能性.

上述静态缺陷定位工作在文件和函数级取得了较好的缺陷定位性能, 但粗粒度的缺陷定位结果并不能满足开发人员的实际需求, 仍需要根据粗粒度的缺陷定位结果按照默认顺序检查大量语句. 此外, 基于历史的缺陷定位方法所使用历史缺陷信息仅为当前项目的文件级历史版本信息, 未使用跨项目以及语句级的历史缺陷信息. 与基于信息检索的缺陷定位方法相同的是, 本方法通过计算缺陷报告、历史缺陷报告等信息与源代码的相关性定位缺

© 中国科学院软件研究所 http://www.jos.org.cn

陷, 不同的是使用了相似缺陷报告对应的历史缺陷语句实现更细粒度的语句级缺陷定位. 与基于历史的缺陷定位方法相同的是, 本方法在基于信息检索的缺陷方法基础上使用了历史缺陷信息辅助进行缺陷定位, 不同的是使用来自不同项目的历史缺陷报告以及对应缺陷语句进行语句级的缺陷定位.

1.2 动态缺陷定位方法

与静态缺陷定位方法不同, 动态缺陷定位方法多采用分析程序执行信息来定位程序中的缺陷, 其中基于频谱和基于变异分析的缺陷定位是两类应用最广泛的动态缺陷定位方法.

基于频谱的缺陷定位方法通过运行测试用例来构建频谱, 然后使用频谱计算每个代码实体的可疑分数, 其主要思想是被更多失败测试用例和更少通过测试用例所覆盖的代码实体更有可能存在缺陷. 在此基础上, Abreu 等人[25]提出了 Ochiai 可疑度计算公式, Wong 等人[26]根据 Kulczynski 系数提出了 DStar 可疑度计算公式, 两者均取得了有较好的缺陷定位效果. 由于基于频谱的缺陷定位方法通常存在排名并列问题, 且频谱并不能直接反映导致缺陷的根本原因, Wen 等人[27]提出了基于历史谱的缺陷定位方法. 该方法首先从版本历史中识别出导致缺陷的提交, 用于构建历史谱, 最后根据历史谱和频谱计算出语句的可疑度并进行排名. Xie 等人[28]考虑到原始的程序频谱存在严重的类不平衡问题, 提出了一种通用的数据增强方法 Aeneas. 该方法首先应用修正的主成分分析 (principal component analysis) 生成缩减的特征空间, 以更简洁地表示原始程序频谱, 然后通过条件式变分自编码机 (conditional variational autoencoder) 从缩减的特征空间合成失败测试用例来缓解不平衡数据问题, 最后将增强后的频谱应用于基于频谱的缺陷定位方法, 以提升缺陷定位性能. 张卓等人[29]为解决语句覆盖信息无法体现语句在具体执行中的重要程度、可能会降低缺陷定位有效性的问题, 提出了基于 TF-IDF 的缺陷定位方法. 该方法采用词频-逆文件频率识别单个测试用例中语句的影响程度, 构建出可识别语句重要程度的信息模型, 并用来计算语句的可疑度值.

基于变异分析的缺陷定位则使用变异分析信息, 而不仅是程序执行信息来进行缺陷定位. Papadakis 等人[30]首次将变异测试应用于缺陷定位, 提出了基于变异分析的缺陷定位方法 Metallaxis, 该方法依赖变异体之间的相似性来检测未知缺陷, 其总体思路是对被测程序进行变异生成变异体, 然后执行测试用例, 将执行结果与源程序测试结果构建特征比较, 通过计算变异体的可疑度反映变异体所在程序语句存在缺陷的可能性, 以定位缺陷语句. 受软件自动修复技术的启发, Moon 等人[31]认为在对程序变异后, 过去失败的测试用例更可能通过, 过去通过的测试用例更可能失败, 据此提出了基于变异分析的缺陷定位技术 MUSE. MUSE 使用变异体来模拟不同的程序行为, 并设计了反映变异体间相似程度的度量公式, 据此计算语句的可疑度. 虽然 MUSE 与 Metallaixs 都是利用程序变异来改变程序行为, 但他们用于计算可疑度的度量公式不同, 因此其定位效果也不同. 贺韬等人[32]利用变异分析修正缺陷定位结果, 降低了偶然成功测试用例对缺陷定位的影响, 提高了缺陷定位的召回率. Zhang 等人[33]首次在回归测试中应用变异测试来进行缺陷定位, 提出了基于缺陷注入的缺陷定位方法 FIFL (fault injection for fault localization), 对于新旧两个版本, 首先对旧版本进行变异测试并记录变异体执行结果, 利用频谱信息计算编辑语句 (即新版本中修改过的语句) 的可疑度; 然后根据特定推理规则推断编辑语句与变异体之间的映射关系, 计算每个变异体的可疑度, 如果一个变异体在编辑后对程序依然有相似的影响, 则赋予这个变异体更高的可疑度; 最后使用变异体的可疑度来调整编辑语句的可疑度, 以定位编辑语句中的缺陷.

上述动态缺陷定位工作虽然在语句级取得了较好效果, 但是通常都需要使用大量测试用例驱动程序运行, 并对收集到测试用例的覆盖信息和执行结果进行分析, 以定位缺陷所在位置. 在此过程中, 构建和执行测试用例集都需要耗费大量人力和时间开销, 也增大了开发人员定位缺陷的难度. 与上述方法不同的是, 本文提出了基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator, 使用程序的静态信息辅助进行缺陷定位, 不需要构建和执行测试用例, 在取得了较好的缺陷定位效果的前提下有效节省了时间开销.

2 动机示例

在此前的研究中, 由于程序的单条代码语句所能提供的静态信息较少, IRBL 主要应用于文件和函数级别. 即使粗粒度 IRBL 具有较好的缺陷定位性能, 开发人员仍需要按照代码语句在文件和函数中的默认排序依次检查语句以确定缺陷所在位置. 例如, 图 1 为 Defects4J 中 Lang-26 (项目 Lang 的版本 26) 中缺陷语句所在文件 FastDate-

© 中国科学院软件研究所 http://www.jos.org.cn

Format.java 的代码片段, 其中第 445 行为缺陷所在语句. 即使 IRBL 可以精准地将 FastDateFormat.java 文件排名至第一, 开发人员仍需检查 445 行代码才能定位到缺陷语句, 这将极大地增加缺陷定位的人力和时间开销. 因此粗粒度的 IRBL 难以满足开发人员的实际需求.

开源社区或平台中包含了大量开发人员遇到过的编码问题及解决方案. 开发人员在编码调试过程中遇到无法解决的问题时, 常会将遇到的问题信息总结后在开源社区或平台检索相同或相似问题, 以参考相同或相似的方案来解决问题[34,35]. 与此相似的是, 缺陷库中包含了大量历史缺陷报告, 其中已解决的缺陷报告中一般包含开发人员针对缺陷代码的修改情况, 根据被测程序的缺陷报告可在缺陷库中检索到相似缺陷报告, 其中通常会包含相似缺陷, 与历史缺陷相似度更高的代码语句存在缺陷的可能性更大[16,19,34]. 通过这种方式可进行语句级的缺陷定位.

例如, 图 2 为使用 Lang-26 的缺陷报告检索出到相似缺陷报告 CKL-170 中的缺陷代码片段, 其中第 3 行为缺陷语句, 与图 1 中 445 行 Lang-26 的缺陷语句具有很高的相似度. 这表明使用相似缺陷报告中的历史缺陷语句可有效识别出程序中的缺陷语句, 辅助进行语句级缺陷定位. 此外, 由于 IRBL 在文件级别具有较好的定位性能, 因此在 IRBL 文件级定位的基础上进行语句级缺陷定位可以排除更多无关代码语句, 提高缺陷定位精准度. 为此, 本文提出了基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator, 将在第 3 节中进行介绍.

行号	源代码
444	public String format(Date date) {
445	Calendar c = new GregorianCalendar (mTimeZone);
446	c.setTime(date);
447	return applyRules(c, new StringBuffer (mMaxLengthEstimate)).toString();
448	}

图 1 FastDateFormat.java 文件代码片段

行号	源代码
1	buffer.append(textField);
2	if (!isReadonly() && !isDisabled()) {
3	Calendar calendar = new GregorianCalendar();
4	buffer.append("<img align=\"top\" ");
5	buffer.append("style=\"cursor:hand\" src=\"");

图 2 CKL-170 缺陷代码片段

3 基于历史缺陷信息检索的语句级软件缺陷定位方法

图 3 为基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator 框架图. STMTLocator 将被测程序的源码文件和缺陷报告作为输入, 通过使用信息检索技术输出排序后的可疑语句列表以进行缺陷定位, 主要分为 3 个步骤. 首先进行历史缺陷语句提取, 基于缺陷库构建历史缺陷报告库, 检索出与被测程序缺陷报告相似的历史缺陷报告, 提取其中的历史缺陷语句 (在第 3.1 节中详细介绍); 然后进行可疑语句提取, 通过计算被测程序的缺陷报告与源码文件之间的文本相似度检索可疑文件, 并提取其中的可疑语句 (在第 3.2 节中详细介绍); 最后, 进行缺陷语句定位, 计算历史缺陷语句与可疑语句的相似度并降序排列, 生成排序后的可疑语句列表, 列表中排名越高的语句存在缺陷的概率越高 (在第 3.3 节中详细介绍).

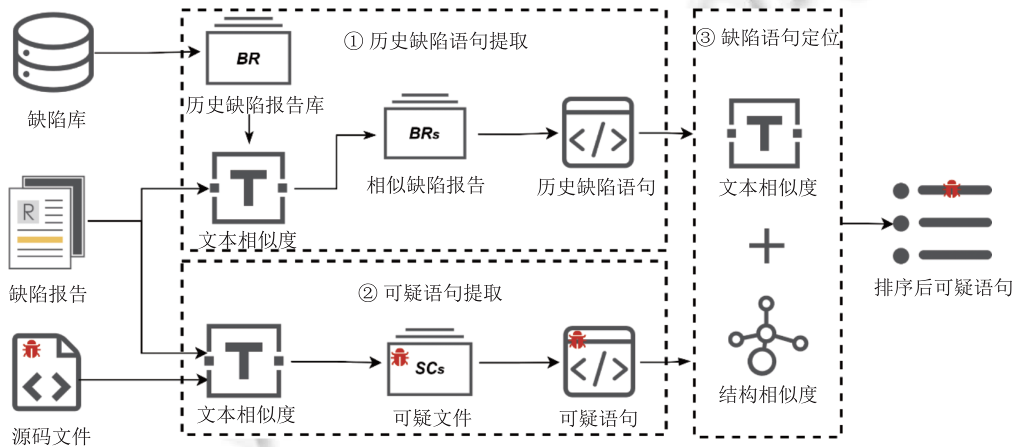

图 3 基于历史缺陷信息检索的语句级软件缺陷定位方法框架图

© 中国科学院软件研究所 http://www.jos.org.cn

3.1 历史缺陷语句提取

历史缺陷语句能够为语句级缺陷定位提供辅助信息. 为获取历史缺陷语句, 我们在构建的历史缺陷报告库中检索相似缺陷报告, 并提取其中包含缺陷的语句作为历史缺陷语句.

首先, 构建历史缺陷报告库. 通过获取缺陷库中管理的历史缺陷报告, 可构建大规模历史缺陷报告库 *BR*. 为了确保所获取的历史缺陷报告可用于相似缺陷报告检索和历史缺陷语句提取, 并辅助缺陷语句定位, 在构建过程中, 通过以下 3 个条件筛选有效的历史缺陷报告.

条件 1: 缺陷库中包含多种如 Improvement、Test 等非缺陷报告, 由于本方法获取的是缺陷报告, 因此需限制所获取的报告类型为 Bug 类型.

条件 2: 缺陷库中的缺陷报告包含已解决和未解决两种状态, 本方法需获取历史缺陷报告及对应缺陷语句, 而未解决的缺陷报告未提供对应缺陷语句, 因此需限制缺陷报告状态为 Closed 或 Resolved 状态.

条件 3: 本方法通过分析历史缺陷报告中上传的修复附件提取对应缺陷语句, 而未包含修复附件的缺陷报告将无法获取对应缺陷语句, 因此需限制所获取的缺陷报告包含修复附件.

然后, 从历史缺陷报告库中检索相似缺陷报告. 如图 4 所示为 Lang-26 缺陷报告的主要信息, 包括缺陷编号、版本、摘要和描述, 其中摘要和描述包含了与缺陷相关的信息. 我们将历史缺陷报告和被测程序缺陷报告 *br* 的摘要和描述视为文本, 使用文本相似度来度量 *br* 和 *BR* 中每个历史缺陷报告的相似度, 以检索相似缺陷报告.

缺陷编号：645
版本：Lang-26
摘要：FastDateFormat.format() outputs incorrect week of year because locale isn't respected
描述：
FastDateFormat apparently doesn't respect the locale it was sent on creation when outputting week in year (e.g. "ww") in format(). It seems to use the settings of the system locale for firstDayOfWeek and minimalDaysInFirstWeek, which (depending on the year) may result in the incorrect week number being output.
Here is a simple test program to demonstrate the problem by comparing with SimpleDateFormat, which gets the week number right:
If sv/SE is passed to Locale.setDefault() instead of en/US, both FastDateFormat and SimpleDateFormat output the correct week number.

图 4　Lang-26 缺陷报告

对于两段文本 T_1 和 T_2, 分别进行分词预处理, 将 T_1 和 T_2 分别转化为单词列表 X 和 Y. 然后将 X 和 Y 合并去重后构建由 n 个单词组成的词袋 $D=[w_1,w_2,\ldots,w_n]$, 根据所构建的词袋 D 以及两段文本的单词列表 X 和 Y, 分别计算单词 $w_i\in D$ 在 X 和 Y 中出现的频率 x_i 和 y_i (即词频), 以构建两段文本的词频向量 $\vec{x}=[x_1,x_2,\ldots,x_n]$ 和 $\vec{y}=[y_1,y_2,\ldots,y_n]$.

最后计算两个词频向量的相似度以度量词频向量对应文本之间的文本相似度. 公式 (1)–公式 (3) 为常用的文本相似度计算公式[36,37], 分别用于计算向量 $\vec{x}$ 和 $\vec{y}$ 的余弦相似度、皮尔逊相关系数和欧几里得距离. 余弦相似度和皮尔逊相关系数越大、欧几里得距离越小表示两段文本越相似. 为了将相似度计算公式计算结果与文本相似度统一为正相关 (即公式计算结果越大两段文本越相似), 本方法将欧几里得距离的相反数作为文本相似度. 基于上述文本相似度计算方法, 计算 *br* 与 *BR* 中每个历史缺陷报告之间的文本相似度 *textSimilarity*, 以检索出最相似的 r 个历史缺陷报告, 构成由相似缺陷报告组成的集合 $BRs=\{br_1, br_2,\ldots, br_r\}$, 其中 $BRs\subseteq BR$.

$$\cos\theta=\frac{\vec{x}\cdot\vec{y}}{\|x\|\cdot\|y\|}=\frac{\sum_1^n(x_i\times y_i)}{\sqrt{\sum_1^n x_i^2}\times\sqrt{\sum_1^n y_i^2}} \tag{1}$$

$$\rho=\frac{cov(\vec{x},\vec{y})}{\sigma\vec{x}\sigma\vec{y}} \tag{2}$$

$$dist = -\sqrt{\sum_{1}^{n}(x_i - y_i)^2} \tag{3}$$

最后提取历史缺陷语句. 经分析, 所提交的历史缺陷报告附件中通常会包含两次相邻提交 (commit) 间的代码差异, 如使用“–”标记针对缺陷进行修改后删除的代码语句, 使用“+”标记针对缺陷进行修改后新增的代码语句. 我们提取将 BRs 中每个相似缺陷报告中的前 k 条被删除的代码语句为作为历史缺陷语句, 最终构成由 $r\times k$ 条历史缺陷语句组成的历史缺陷语句集合 $BS=\{bs_1, bs_2,\ldots, bs_{r\times k}\}$, 用于进行缺陷语句定位.

如图 5 为 Lang-26 缺陷报告 br 的相似缺陷报告 CKL-170 中提取的信息, 其中“+”标记针对缺陷进行修改后新增的代码语句 (即图 5 中绿色代码语句), “–”标记针对缺陷进行修改后删除的代码语句 (即图 5 中红色代码语句). 其中由“–”标记的两条代码语句将被提取作为历史缺陷语句.

```
import java.util.Arrays;
import java.util.Calendar;
import java.util.Date;
−import java.util.GregorianCalendar;
import java.util.Locale;
import javax.servlet.ServletContext;
@@ −534,7 +533,7 @@
            buffer.append(textField);
            if (!isReadonly() &amp;&amp; !isDisabled()) {
−                Calendar calendar = new GregorianCalendar();
+                Calendar calendar = Calendar.getInstance(getLocale());
                buffer.append("&lt;img align=\"top\" ");
                buffer.append("style=\"cursor:hand\" src=\"");
@@ −565,7 +564,7 @@
                buffer.append(" align :           'cr', \n");
                buffer.append(" singleClick : true, \n");
                buffer.append(" firstDay :      ");
−                buffer.append(calendar.getFirstDayOfWeek());
+                buffer.append(calendar.getFirstDayOfWeek() −1);
                buffer.append("\n});\n");
                buffer.append("&lt;/script&gt; \n");
            }
```

图 5 缺陷报告 CKL-170 附件信息

3.2 可疑语句提取

IRBL 在文件级具有较高的定位精度, 在此基础上进行语句级的定位将有助提升缺陷定位精度. 我们使用文件级 IRBL 检索出可疑文件, 并提取可疑文件中的可疑语句, 后续将在可疑语句上进行语句级缺陷定位.

缺陷报告中包含的缺陷摘要、描述一般包含与缺陷文件相关的代码片段或关键字, 相比于其他源码文件, 引发缺陷的源码文件与缺陷报告具有更高的相似度. 因此, 我们使用文本相似度来度量被测程序的源码文件与缺陷报告的相似度, 以检索出可疑度较大的源码文件, 其中包含的代码语句更有可能存在缺陷.

首先进行可疑文件检索. 对于被测程序的缺陷报告 br 和源码文件 $SC=\{sc_1, sc_2,\ldots, sc_m\}$, 依次计算每个源码文件 $sc_i \in SC$ 与 br 的相似度, 以进行可疑文件检索. 在计算过程中, 先对文本进行分词预处理. 需要注意的是, 由于文

© 中国科学院软件研究所 http://www.jos.org.cn

本中包含了代码语句, 因此在分词过程中还需对程序通用的关键词进行去除 (如 int、double、float 等), 以减少对后续计算文本相似度的影响. 接下来, 根据单词列表构建词袋, 并统计单词列表中每个单词的词频以构建文本的词频向量, 计算词频向量间的相似度以衡量 br 与 sc_i 之间的文本相似度. 最后, 根据文本相似度对 SC 中所有源码文件进行降序排序, 检索出排名前 f 的源码文件以获取可疑文件组成的集合 $SC_s=\{file_1, file_2, \ldots, file_f\}$.

如图 6 所示为 Lang-26 的缺陷报告以及包含缺陷的源码文件. 其中缺陷报告内包含了一些关键的文本信息, 例如 FastDateFormat、format 等. 首先, 分别计算 Lang-26 的缺陷报告与每个源码文件的文本相似度以筛选可疑文件, 表 1 所示为 Lang-26 中排名前五的可疑文件, 其中包含缺陷的源码文件 FastDateFormat.java 排名第一, 这是因为该文件中包含了大量与目标缺陷报告中相同的关键词, 与缺陷报告具有更高的文本相似度, 使用文本相似度可将包含缺陷的源码文件排在更靠前的位置. 然后, 进行可疑语句提取. 对 SC_s 中每个可疑文件进行分析, 获取可疑文件中所有代码语句, 删除注释、空行、括号等非代码语句, 并记录可疑语句所在文件以及行号, 以构成由可疑语句组成的集合 $ST=\{st_1, st_2, \ldots, st_n\}$.

缺陷编号：64

版本：Lang-26

摘要：FastDateFormat .format() outputs incorrect week of year because locale isn’t respected

描述：

FastDateFormat apparently doesn’t respect the locale it was sent on creation when outputting week in year (e.g. "ww") in format(). It seems to use the settings of the system locale for firstDayOfWeek and minimalDaysInFirstWeek, which (depending on the year) may result in the incorrect week number being output.

Here is a simple test program to demonstrate the problem by comparing with SimpleDateFormat, which gets the week number right:If sv/SE is passed to Locale.setDefault() instead of en/US, both FastDateFormat and SimpleDateFormat output the correct week number.

源码文件：FastDateFormat .java

```
public class FastDateFormat extends Format {
…
    private static final Map< FastDateFormat , FastDateFormat > cInstanceCache = new HashMap<FastDateFormat,
FastDateFormat >(7);
private static final Map<Object, FastDateFormat > cDateInstanceCache = new HashMap<Object, FastDateFormat >(7);
private static final Map<Object, FastDateFormat > cTimeInstanceCache = new HashMap<Object, FastDateFormat >(7);
private static final Map<Object, FastDateFormat > cDateTimeInstanceCache = new HashMap<Object, FastDateFormat >(7);
    private static final Map<Object, String> cTimeZoneDisplayCache = new HashMap<Object, String>(7);
…
}
```

图 6 Lang-26 缺陷报告和缺陷源码文件

表 1 Lang-26 可疑文件列表片段

排名	可疑源码文件
1	**org.apache.commons.lang3.time.FastDateFormat.java**
2	org.apache.commons.lang3.time.DateUtils.java
3	org.apache.commons.lang3.LocaleUtils.java
4	org.apache.commons.lang3.text.FormatFactory.java
5	org.apache.commons.lang3.time.DurationFormatUtils.java

© 中国科学院软件研究所 http://www.jos.org.cn

3.3 缺陷语句定位

与历史缺陷语句越相似的语句存在缺陷的可能性越高, 本文通过计算历史缺陷语句与可疑语句的文本相似度以及结构相似度来度量语句的可疑度, 并根据相似度对可疑语句降序排列, 排名越高的可疑语句存在缺陷的可能性越高.

首先计算可疑语句与历史缺陷语句文本相似度分数. 具有相似缺陷的代码语句一般包含相同的单词, 例如对于 Lang-26 的缺陷语句“Calendar c = new GregorianCalendar(mTimeZone);”, 以及提取出的历史缺陷语句“Calendar calendar = new GregorianCalendar();”, 两者都包含“Calendar”“new”和“Gregorian”等单词, 具有较高的文本相似度. 公式 (4) 可用于计算可疑语句和历史缺陷语句集的文本相似度分数 *textScore*. 其中归一化函数 N 使用如公式 (5) 所示的最大-最小归一化方法将和的文本相似度 *textSimilarity* 归一化至 [0, 1], *max* 和 *min* 分别表示中可疑语句的最大和最小文本相似度, x 为待归一化的文本相似度. 分别计算与每条历史缺陷语句的文本相似度后, 将最大的文本相似度归一化后作为的文本相似度分数, 越大表示可疑语句与历史缺陷语句在文本上越相似.

$$textScore(st_i) = N(\max\{\cos\theta(st_i, bs_j), bs_j \in BS\}) \tag{4}$$

$$N(x) = \frac{x - min}{max - min} \tag{5}$$

其次计算可疑语句与历史缺陷语句结构相似度分数. 除文本信息外, 代码的结构信息也体现了代码的特征, 因此与历史缺陷语句具有相似结构信息的可疑语句存在缺陷的可能性同样较大. 抽象语法树 (abstract syntax tree, AST) 可直观地描述代码结构信息, 本文通过构建 AST 来提取代码的结构信息, 以度量代码之间的结构相似度. 需要注意的是, 对于一条代码语句, 由于无法直接为其构建对应的 AST, 我们首先将单条代码语句扩充为只包含所需分析的代码语句的类, 然后构建类的 AST, 此时 AST 中将包含所需分析代码语句的 AST 信息. 接下来对代码语句的 AST 信息进行分析, 提取代码语句 AST 中各节点类型信息以组成结构信息序列, 以获取可疑语句和历史缺陷语句的结构信息序列. 图 7 为构建代码语句结构信息序列示例, 对于语句“Calendar calendar = new Gregorian-Calendar();”, 首先将语句扩充为类, 然后构建类的 AST, 通过分析类的 AST 可以获取到与此语句相关的 AST 片段, 最后依次提取 AST 片段中的节点类型信息, 组成语句的结构信息序列“VariableDeclarationStatement SimpType SimpleName VariableDeclarationFragment SimpleName ClassInstanceCreation SimpleType SimpleName”. 由于结构信息序列中节点类型信息的顺序同样体现了代码的结构特征, 为度量可疑语句和历史缺陷语句的结构相似度, 本文使用最短编辑距离 (edit distance) 计算两个结构信息序列间的相似程度, 距离越小则表示两个序列越相似, 即两段序列对应的代码也越相似. 公式 (6) 为可疑语句 $st_i \in ST$ 结构相似度 *structureScore* 的计算方法, 由于最短编辑距离越小表示两段序列越相似, 在分别计算 st_i 与每条历史缺陷语句的最短编辑距离后, 选择其中的最小值, 取其相反数并归一化至 [0, 1], 作为 st_i 的结构相似度分数 $structureScore(st_i)$, 结构相似度分数越大则表示可疑语句与历史缺陷语句在结构上越相似.

$$structureScore(st_i) = N(-\min\{EditDistance(st_i, bs_j), bs_j \in BS\}) \tag{6}$$

例如对于 Math-56, STMTLocator 缺陷定位过程中提取到的历史缺陷语句为“CoderResult result = encoder.encode(chars, bytes, true);”, 可疑语句 1: “indices[last] = idx;”和可疑语句 2: “count – = offset;”与历史语句的文本相似度 *textScore* 均为 0, 其中可疑语句 1 为缺陷语句, 而可疑语句 2 不是缺陷语句. 若仅使用文本相似度, 无法准确度量缺陷语句与历史缺陷语句的相似性, 导致可疑语句 1 可疑度排名靠后, 也不能将其与可疑语句 2 区分. 而使用结构相似度分数度量可疑语句时, 历史缺陷语句将被表示为“VariableDeclarationStatement SimpleType SimpleName VariableDeclarationFragment SimpleName MethodInvocation SimpleName SimpleName SimpleName SimpleName BooleanLiteral”, 两条可疑语句将分别被表示为“VariableDeclarationStatement SimpType SimpleName Variable-DeclarationFragment SimpleName ClassInstanceCreation SimpleType SimpleName SimpleName”和“Expression-Statement Assignment SimpleName SimpleName”. 根据公式 (6) 计算出两条可疑语句与历史缺陷语句结构信息序列的结构相似度分数 *structureScore* 分别为 0.92 和 0.86, 此时缺陷语句与历史缺陷语句相似度极高, 可疑度排名靠

前, 且能够有效地区分两条可疑语句. 可以看出, 使用结构相似度度量代码相似度也有助于进行语句级的缺陷定位.

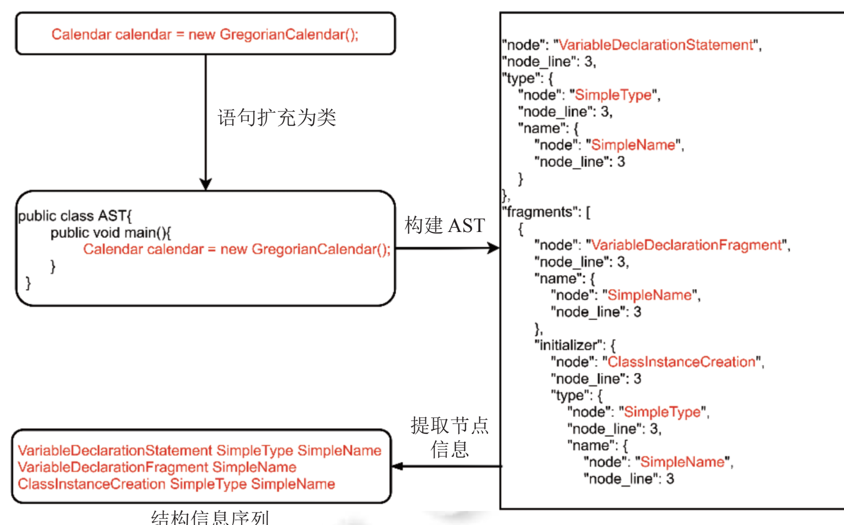

图 7 代码语句的结构信息序列获取示例

最后计算可疑语句综合相似度分数. 可疑语句 st_i 的文本相似度 $textScore(st_i)$ 和结构相似度分数 $structureScore(st_i)$ 从不同角度度量了可疑语句与历史缺陷语句的相似性, 我们使用公式 (7) 对两个相似度分数进行结合, 即将 $textScore(st_i)$ 和 $structureScore(st_i)$ 相加, 并对相加后的结果归一化, 作为 st_i 的综合相似度分数 $overallScore(st_i)$. 在计算 ST 中所有可疑语句的综合相似度分数后, 根据综合相似度分数对 ST 中的可疑语句进行降序排列生成可疑语句列表, 列表中排名越高的可疑语句存在缺陷的可能性越大.

$$overallScore(st_i) = N(textScore(st_i) + structureScore(st_i)) \tag{7}$$

4 实验设计与评估

4.1 研究问题

为了评估 STMTLocator 方法的有效性, 本文提出 3 个研究问题, 在 Defects4J 数据集上进行实验, 并使用 Top@N、MRR 指标评估缺陷定位性能.

RQ1: 相较于静态缺陷定位方法, STMTLocator 的语句级缺陷定位性能如何?

STMTLocator 使用缺陷报告及缺陷语句等静态信息, 基于信息检索技术进行语句级缺陷定位, 属于静态缺陷定位方法. Zhou 等人[16]提出的 BugLocator 是比较典型和广为接受的文件级静态缺陷定位方法, STMTLocator 基于 BugLocator 进行语句级的缺陷定位. 因此我们将 STMTLocator 与 BugLocator 的缺陷定位性能进行对比.

RQ2: 相较于动态缺陷定位方法, STMTLocator 的语句级缺陷定位性能如何?

语句级缺陷定位常使用动态缺陷定位方法, 基于频谱的缺陷定位方法 (spectrum-based fault localization, SBFL) 和基于变异分析的缺陷定位方法 (mutation-based fault localization, MBFL) 是常用的动态缺陷定位方法. 实证研究表明[38], DStar[26]和 Metallaxis[30]分别是定位性能最好的 SBFL 和 MBFL 方法. 因此我们将 STMTLocator 与 DStar 和 Metallaxis 的缺陷定位性能进行对比.

RQ3: 代码的文本相似度和结构相似度如何影响 STMTLocator 的缺陷定位性能?

STMTLocator 方法融合了代码的文本相似度和结构相似度, 以度量可疑语句与历史缺陷语句的相似度, 代码的文本相似度和结构相似度如何影响 STMTLocator 的缺陷定位性能?

© 中国科学院软件研究所 http://www.jos.org.cn

4.2 实验设置

实验硬件环境为: Inter Core i7-1185G7、16 GB 物理内存; 软件环境为: Windows 10、JDK 1.8.0、Python 3.7.0. STMTLocator 使用 BugLocator (https://code.google.com/archive/p/bugcenter/downloads) 用于检测可疑目标源码文件, EclipseJDT (https://www.eclipse.org/jdt/) 用于构建 Java 代码的 AST. STMTLocator 方法涉及的参数主要包括相似缺陷报告数 r、可疑文件数 f、历史缺陷语句数 k. 我们在实验中使用 $r = 10$、$f = 5$, $k = 1$, 并在第 5.1 节讨论参数设置对 STMTLocator 的影响.

4.2.1 数据集

本文将在缺陷定位工作中被广泛使用的 Defects4J (V1.4.0) 数据集作为实验对象[27,38–40]. 实验对象的基本信息以及实验所用的数据情况如表 2 所示, Defects4J 中共包含来自 6 个项目中的共 395 个缺陷版本. 我们对这 395 个缺陷版本进行了初步分析, 排除了不包含缺陷报告的缺陷版本 22 个, 采用了剩余的 373 个缺陷版本作为实验对象, 并收集了缺陷报告中的缺陷编号、摘要和描述等信息.

表 2 Defects4J (V1.4.0) 数据集基本信息

项目ID	项目名称	缺陷版本数量	采用缺陷版本数量	平均代码行数 (k)
Chart	JFreeChart	26	8	22
Closure	Closure compiler	133	131	150
Lang	Apache commons-lang	65	64	60
Math	Apache commons-math	106	106	160
Mockito	Mockito	38	38	15
Time	Joda-time	27	26	65
总计	—	395	373	472

此外, 我们使用 YAKE (https://github.com/LIAAD/yake) 工具提取缺陷报告的关键词, 用于在 JIRA 中检索相关历史缺陷报告, 对于每个缺陷版本最多收集 20 个相关历史缺陷报告, 并去除重复的历史缺陷报告, 最终共收集到 1 816 个相关历史缺陷报告, 用于后续检索相似缺陷报告. 在收集与 Defects4J 中各缺陷版本相关的历史缺陷报告过程中, 要求报告类型为 Bug、状态为 Resolved 或 Closed, 以确保历史缺陷报告包含缺陷并已得到修复. 收集的历史缺陷报告信息主要包括: 缺陷编号、摘要、描述以及修复附件. 同时, 由于 Defects4J 部分项目的缺陷报告同样管理在 JIRA 中, 因此在收集过程中剔除了与 Defecst4J 中重复的缺陷报告.

4.2.2 评价指标

我们使用 Top@N 和 MRR 指标来评估不同缺陷定位方法的缺陷定位性能, 这两个指标被广泛用于评估缺陷定位方法的性能[1,6,11,27,28].

Top@N: Top@N 衡量了定位缺陷位置的绝对排名, 即对于给定的可疑语句列表, 排名前 N 的语句中包含缺陷的数量, N 越小、Top@N 越大, 则缺陷定位技术效果越好. 参照 Zhou 等人[16]的工作, 本文重点关注 $N = 1$、5、10 时的缺陷定位情况.

MRR (mean reciprocal rank): Reciprocal rank 计算的是在可疑语句列表中发现第 1 个缺陷语句排名的倒数, MRR 是所有可疑语句列表中缺陷语句排名倒数的平均值, 计算方式如公式 (8) 所示. 其中 K 表示缺陷版本数量, $rank_i$ 为第 i 个缺陷版本的可疑语句列表中含有缺陷语句的排名, MRR 值越大, 缺陷定位技术的性能越好.

$$MRR = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{rank_i} \tag{8}$$

4.3 实验结果

4.3.1 RQ1 的实验结果及分析 (STMTLocator 与 BugLocator 缺陷定位性能对比)

为比较 STMTLocator 与 BugLocator 在语句级定位缺陷性能差异, 我们将 BugLocator 的文件级缺陷定位结果映射至语句级, 即对于 BugLocator 生成的可疑文件列表, 统计比缺陷所在文件可疑度排名靠前的所有文件代码总

© 中国科学院软件研究所 http://www.jos.org.cn

行数与缺陷所在文件代码总行数的一半之和作为 BugLocator 的语句级缺陷定位结果. 同时, 为了验证 STMTLocator 使用不同文本相似度计算方法是否会对缺陷定位性能产生影响, 我们在 Defects4J 数据集中的 373 个缺陷版本上使用 STMTLocator 与 BugLocator 进行了对比实验. 实验结果如表 3 所示, 其中 STMTLocator (cos)、STMTLocator (person) 和 STMTLocator (ED) 分别表示使用余弦相似度、皮尔逊相关系数和欧几里得距离计算文本相似度时的 STMTLocator. 对于 *MRR* 指标, STMTLocator 使用 3 种不同文本相似度计算方法时的缺陷定位性能均优于 BugLocator, 其中 STMTLocator (cos) 定位性能最优, 相比 BugLocator 提升了 0.022 18 (399.45%). 对于 Top@N 指标, 当 $N=1$ 和 5 时, BugLocator 均定位不到缺陷语句, 当 $N=10$ 时, BugLocator 可以定位到 2 条缺陷语句, 而 STMTLocator 在 $N=1$、5 和 10 时, 使用不同文本相似度计算方法时均比 BugLocator 定位到更多缺陷语句, 其中 STMTLocator (cos) 可定位到的缺陷语句最多, 相比 BugLocator 多定位到 7、10 和 10 条缺陷语句. 分析发现, BugLocator 虽然具有较好的文件级缺陷定位性能, 但缺陷语句在文件中排名靠前的可能性较低, 按照文件中语句的默认顺序检查难以定位到缺陷语句, 因此 BugLocator 的语句级缺陷定位性能较差. 而 STMTLocator 在检索出的可疑文件基础上, 使用历史缺陷语句进一步筛选出可疑度较大的语句, 排除了更多与缺陷无关的语句, 从而比 BugLocator 具有更好的语句级缺陷定位性能.

表 3　BugLocator 和 STMTLocator 缺陷定位性能对比

指标	BugLocator	STMTLocator (cos)	STMTLocator (person)	STMTLocator (ED)
MRR	0.005 49	0.027 42 (+399.45%)	0.020 70 (+277.05%)	0.017 68 (+222.04%)
Top@1	0	7 (+7)	4 (+4)	4 (+4)
Top@5	0	10 (+10)	8 (+8)	7 (+7)
Top@10	2	12 (+10)	9 (+7)	8 (+6)
Time (s)	3.52	68.21	68.15	68.14

此外, 为了比较 BugLocator 和 STMTLocator 缺陷定位所需时间开销, 实验统计了 BugLocator、STMTLocator (cos)、STMTLocator (person) 和 STMTLocator (ED) 完成一个版本缺陷定位的平均消耗时间, 其中 BugLocator 缺陷定位粒度为文件级别, 因此我们仅统计了其定位缺陷文件所需时间开销. 如表 3 所示, BugLocator 完成一个版本缺陷定位平均需要 3.51 s, 由于其属于静态缺陷定位方法, 无需使用测试用例执行被测程序, 因此需要的时间开销较小. 然而, 在真实场景中, 开发人员还需基于文件级的缺陷定位结果手动检查缺陷所在语句, 这将耗费大量时间. 使用不同文本相似度计算方法时, STMTLocator 完成一个版本缺陷定位平均需要 68.14–68.21 s, 虽然相比 BugLocator 增加了提取历史缺陷语句以及缺陷语句定位所需时间开销, 但由于其缺陷定位粒度为语句级别, 因此可降低根据文件级缺陷定位结果手动检查缺陷语句所需的大量时间开销.

对 RQ1 的回答: 相比静态的缺陷定位方法 BugLocator, STMTLocator 的 *MRR*、Top@N 指标均有所提升, 在语句级的缺陷定位性能明显优于 BugLocator. 虽然 STMTLocator 增加了少量定位时间开销, 但相比 BugLocator 节省了基于文件级缺陷定位结果手动检查缺陷语句所在位置所需的大量时间开销. 此外, 实验表明 STMTLocator 使用余弦相似度方法计算文本相似度时具有更好的缺陷定位性能. 因此, 后续实验中我们将使用余弦相似度的 STMTLocator 作为实验对象.

4.3.2　RQ2 的实验结果及分析 (STMTLocator 与 DStar、Metallaxis 缺陷定位性能对比)

在此前的研究中, 动态缺陷定位方法在语句级取得了较好的定位效果[38], 如基于频谱的缺陷定位方法 DStar 和基于变异分析的缺陷定位方法 Metallaxis 等. STMTLocator 同样关注语句级缺陷定位. 为与动态缺陷定位方法进行对比, 我们将 STMTLocator 与 DStar、Metallaxis 在 Defects4J 数据集上进行实验, 并使用 Top@N 和 *MRR* 指标进行评价. 其中, Metallaxis 需要对变异体的执行信息进行分析以进行缺陷定位, 而在对源程序进行变异分析收集变异体执行信息时, 部分缺陷版本需要执行数十小时, 为了便于统计, 我们将收集变异体执行信息的时间限制为 10 000 s, 当时间超过 10 000 s 时停止收集, 由于此时收集到的执行信息不完整, 因此放弃对相应缺陷版本进行缺陷定位. 根据上述实验设置, 在使用 Metallaxis 收集变异体执行信息时, 共有 137 个缺陷版本在 10 000 s 内完成了变

异体执行信息的收集. 因此, 为了公平比较, DStar 和 STMTLocator 也在这些缺陷版本上进行缺陷定位.

表 4 为 3 种方法的缺陷定位性能对比. 对于 Top@N 指标, 当 $N=1$、5 和 10 时, Metallaxis 分别可以定位 5、9 和 12 条缺陷语句, DStar 分别可以定位到 9、38 和 54 条缺陷语句, STMTLocator 分别可以定位到 3、5 和 7 条缺陷语句, 相比 Metallaxis 分别减少了 2、4 和 5 条, 相比 DStar 分别减少了 6、33 和 47 条. 对于 *MRR* 指标, DStar 具有最好的定位性能, STMTLocator 相比于 Metallaxis 低 0.02123 (36.23%), 相比 DStar 低 0.12751 (77.26%).

表 4 Metallaxis、DStar 和 STMTLocator 缺陷定位性能及时间开销对比

指标	Metallaxis	DStar	STMTLocator
MRR	0.05884	0.16503	0.03752
Top@1	5	9	3
Top@5	9	38	5
Top@10	12	54	7
Time (s)	3652.65	163.24	59.73

为比较不同缺陷定位方法的时间开销情况, 实验统计了 Metallaxis、DStar、STMTLocator 进行缺陷定位所需时间开销. 其中, 对于动态缺陷定位方法 Metallaxis 和 DStar, 未统计设计测试用例的时间开销. 但需要注意的是, 设计测试用例通常会耗费大量的人力和时间开销. 表 4 统计了 3 种缺陷定位方法在 Defects4J 数据集上进行缺陷定位的平均时间开销, 其中 Metallaxis 完成一个缺陷版本的缺陷定位平均需要 3652.65 s, 时间开销最大, 主要原因在于基于变异分析的缺陷定位方法需要对原始程序语句进行变异, 且对变异后的程序还需要执行测试用例并收集测试用例执行信息, 此过程将消耗大量时间. DStar 完成一个版本缺陷定位平均需要 163.24 s, 相比 Metallaxis 时间开销更小, 主要原因在于基于频谱的缺陷定位方法需要执行测试用例并收集测试用例的执行信息, 但不需要对程序语句进行变异并再次执行测试用例. STMTLocator 完成一个版本缺陷定位平均只需要 59.73 s, 相比 DStar 和 Metallaxis 分别减少了 103.51 s (63.41%) 和 3592.92 s (98.37%). 主要原因在于 STMTLocator 属静态缺陷定位方法, 不需要执行测试用例, 相比动态缺陷定位方法节省了程序执行所需的时间开销. 此外, STMTLocator 的缺陷定位过程不需要构建测试用例集, 将有效减少构建测试用例的人力和时间开销.

对 RQ2 的回答: 虽然 STMTLocator 的缺陷定位性能弱于动态缺陷定位方法 Metallaxis 和 DStar, 但 STMTLocator 完成一个版本缺陷定位所需时间比 DStar 和 Metallaxis 减少了 63.41% 和 98.37%. 更为重要的是, STMTLocator 的缺陷定位过程不需要构建和执行测试用例集, 相比 Metallaxis 和 DStar 减少了构建测试用例的人力和时间开销.

4.3.3 RQ3 的实验结果分析 (针对代码相似度度量组件的消融实验)

为研究代码的文本相似度和结构相似度对 STMTLocator 缺陷定位性能的影响, 实验分别使用文本相似度、结构相似度以及融合两种相似度的综合相似度来度量可疑语句与历史缺陷语句的相似度. 实验结果如表 5 所示, 其中 $\text{STMTLocator}_{\text{Text}}$、$\text{STMTLocator}_{\text{AST}}$ 和 STMTLocator 分别为使用文本相似度、结构相似度和综合相似度的基于历史缺陷信息检索的语句级缺陷定位方法.

表 5 STMTLocator 使用不同代码相似度度量方法的缺陷定位结果

指标	$\text{STMTLocator}_{\text{Text}}$	$\text{STMTLocator}_{\text{AST}}$	STMTLocator
MRR	0.02190	0.02521	0.02742
Top@1	5	5	7
Top@5	8	11	10
Top@10	10	14	12

如表 5 所示, 对于 *MRR* 指标, STMTLocator 比 $\text{STMTLocator}_{\text{Text}}$ 和 $\text{STMTLocator}_{\text{AST}}$ 分别高 0.00552 (25.21%) 和 0.00221 (8.77%), 具有更好的定位效果. 对于 Top@N 指标, 当 $N=1$ 时, STMTLocator 可以定位到 7 条缺陷语

© 中国科学院软件研究所 http://www.jos.org.cn

句, 比 $STMTLocator_{Text}$ 和 $STMTLocator_{AST}$ 分别多定位到 2 条缺陷语句, 同样具有更好的定位效果; 当 $N = 5$ 时, $STMTLocator_{AST}$ 具有最好的定位效果, 可以定位到 11 条缺陷语句, 比 $STMTLocator_{Text}$ 和 STMTLocator 分别多定位到 3 条和 1 条缺陷语句; 当 $N = 10$ 时, $STMTLocator_{AST}$ 具有最好的定位效果, 可以定位到 14 条缺陷语句, 比 $STMTLocator_{Text}$ 和 STMTLocator 分别多定位到 4 条和 2 条缺陷语句. $STMTLocator_{AST}$ 在 $N = 5$ 和 10 时具有更好的缺陷定位性能, 主要原因是部分缺陷版本历史缺陷语句与真实缺陷语句的文本相似度较低, 但其结构相似度较高, 使用结构相似度度量代码相似度时能够将缺陷语句排在更靠前的位置. 需要注意的是, Kochhar 等人[41]的研究发现, 开发人员在逐一检查可疑语句列表中少量语句后, 若未能发现缺陷便会失去耐心, 这表明开发人员更加关注可疑语句列表中排名靠前, 特别是排名第一的语句, 因此本文更关注缺陷定位方法的 Top@1 指标. STMTLocator 的 Top@1 值最高, 同时 *MRR* 也最大, 因此 STMTLocator 具有更好的缺陷定位性能.

对 RQ3 的回答: 融合文本相似度和结构相似度度量代码相似度的 STMTLocator 比单独使用文本相似度或结构相似度具有更好的缺陷定位性能. 其中, 对于 Top@*N* 指标, 当 $N = 1$ 时, STMTLocator 比 $STMTLocator_{Text}$ 和 $STMTLocator_{AST}$ 分别多定位到 2 条缺陷语句; 对于 *MRR* 指标, STMTLocator 比 $STMTLocator_{Text}$ 和 $STMTLocator_{AST}$ 分别高 0.005 52 (25.21%) 和 0.002 21 (8.77%).

5 讨 论

5.1 参数设置对 STMTLocator 方法的影响

STMTLocator 中使用的参数包括相似缺陷报告数 *r*、可疑文件数 *f*、历史缺陷语句数 *k*, 我们对每个参数的取值进行调整, 以评估 STMTLocator 在不同参数取值组合时的有效性. 由于每个参数的取值区间较大, 难以对每个参数取值进行实验, 因此假设 3 个参数的取值都分别为 1、5 和 10. 如果将 3 个参数的 3 种取值进行组合, 共需 27 组实验来分析不同参数取值组合时 STMTLocator 方法的有效性. 为简化实验, 我们选择使用正交实验法来对参数进行组合, 以挑选出具有代表性的参数组合. 表 6 为对参数 *r*、*f*、*k* 设计的正交表, 共计 9 组参数组合, STMTLocator 将在这 9 组参数上进行实验.

表 7 为 STMTLocator 方法使用 9 组不同参数取值组合进行缺陷定位的结果. 对于 Top@*N* 指标, 当 $N = 1$ 时, 即检查排序后可疑语句列表中第 1 条语句, STMTLocator 使用第 6 组实验参数可以定位到 7 条缺陷语句, 相比其他实验参数组合所能定位到缺陷语句数量更多; 当 $N = 5$ 时, 即检查排序后可疑语句列表前 5 条语句, STMTLocator 使用第 2、3、6 组实验参数均可以定位到 10 条缺陷语句, 相比于使用其他实验参数所能定位到的缺陷语句数量更多; 当 $N = 10$ 时, 即检查排序后可疑语句列表前 10 条语句, STMTLocator 使用第 2 组实验参数可以定位到 16 条缺陷语句, 相比其他实验参数所能定位到的缺陷数量更多. 由于在一般情况下开发人员更关注排名第一的缺陷语句数量, 即对于 Top@*N*, 更关注 $N = 1$ 时的缺陷定位结果, 因此我们认为 STMTLocator 使用第 6 组实验参数时具有更好的缺陷定位性能. 同时, 对于 *MRR* 指标, 使用第 6 组实验参数的结果也优于其他 8 组实验参数.

表 6 相似缺陷报告数 *r*、可疑文件数 *f* 和历史缺陷语句数 *k* 参数取值正交表

实验 (组)	*f*	*r*	*k*
1	1	1	1
2	1	5	5
3	1	10	10
4	5	1	5
5	5	5	10
6	5	10	1
7	10	1	5
8	10	5	1
9	10	10	10

表 7 STMTLocator 方法中不同参数取值组合的缺陷定位效果

实验 (组)	Top@1	Top@5	Top@10	*MRR*
1	2	5	9	0.015 38
2	4	10	16	0.024 16
3	3	10	14	0.021 94
4	1	2	6	0.010 14
5	4	9	11	0.021 19
6	7	10	12	0.027 42
7	1	2	4	0.009 26
8	5	8	11	0.023 29
9	3	8	11	0.017 75

© 中国科学院软件研究所 http://www.jos.org.cn

从表 7 中可以看出, STMTLocator 使用第 1、4、7 组实验参数时相比使用其他实验参数缺陷定位效果更差, 分析发现, 此时相似缺陷报告数 r 的取值都为 1, 即当相似缺陷报告数量较少时 (即 r 的取值较小时), 历史缺陷语句数量也会相应减少, 此时历史缺陷语句中包含的缺陷类型较少, 存在与被测程序的缺陷语句相似历史缺陷语句的可能性较低, 因此将导致 STMTLocaotor 缺陷定位性能下降. 此外, 当历史缺陷语句数 k 的取值较大时, 如第 3、5 和 9 组实验参数 k 的取值为 10, STMTLocator 的缺陷定位性能也较差. 分析发现, 当 k 的取值较大时, 从同一相似缺陷报告中提取出的历史缺陷语句较多, 此时将引入更多与被测程序的缺陷语句不相似但与非缺陷语句相似的历史缺陷语句, 这将会增加历史缺陷语句中的噪音, 从而影响 STMTLocator 的缺陷定位性能. 另外, 当可疑文件数 f 取值过小时, 可疑文件中可能不包含缺陷文件, 当 f 取值过大时, 则会引入过多的无关文件, 降低缺陷定位性能.

因此, 在使用不同参数组合时, STMTLocator 可有效进行语句级的缺陷定位, 对于 Top@N 指标 (N = 1、5、10), STMTLocator 能够定位到 1–16 条缺陷语句. 其中, 在 r 值较大、f 值适中和 k 值较小时 (例如 r = 10、f = 5、k = 1), STMTLocator 可分别定位到 7、10、12 条缺陷语句. 我们建议使用较大的 r 值、适中的 f 值以及较小的 k 值作为 STMTLocator 参数设置, 以确保 STMTLocator 具有较好的缺陷定位效果.

5.2 STMTLocator 方法的局限性

为分析 STMTLocator 方法的局限性, 我们随机检查了 50 个定位效果较差的缺陷版本, 分析了定位效果较差的原因, 主要可总结为两类原因.

其中有 18 个缺陷版本定位效果较差的主要原因为文件级 IRBL 不够准确, 获取的可疑文件中不包含缺陷语句. 在可疑文件检索阶段, STMTLocator 使用文件级 IRBL 进行可疑文件检索, 并提取其中的代码语句作为可疑语句, 后续缺陷语句定位阶段将在检索出的可疑语句上进行. 虽然 IRBL 在文件级别已经具有较好的定位效果, 但并不能确保检索出的可疑文件中一定包含缺陷文件. 例如使用 BugLocator 对 Defects4J 中 373 个缺陷版本进行文件级缺陷定位时, 虽然有 245 个缺陷版本能够将缺陷文件排名至前 10 位, 但仍有 128 个版本无法将缺陷文件排名至前 10 位. 当检索出的可疑文件中不包含缺陷文件时, 所提取的可疑语句中也不会包含缺陷语句, 因此后续缺陷语句定位阶段将定位不到缺陷语句, 导致 STMTLocator 定位效果较差.

例如表 8 为使用 BugLocator 对 Lang-28 进行可疑文件检索时获取的文件可疑度列表片段, 其中包含缺陷的源码文件仅排在第 20 位. 当 STMTLocator 根据此排名检索出可疑文件时, 由于包含缺陷的源码文件排名较低, 可疑文件中将不会包含此源码文件, 根据可疑文件提取出的可疑语句也不会包含缺陷语句. 在这种情况下, STMTLocator 将无法准确定位缺陷语句.

表 8 文件级 IRBL 定位效果不准确导致 STMTLocator 定位效果差的示例

排名	可疑源码文件
20	**org.apache.commons.lang3.text.translate.NumericEntityUnescaper.java**
21	org.apache.commons.lang3.ClassUtils.java
22	org.apache.commons.lang3.reflect.MethodUtils.java
23	org.apache.commons.lang3.RandomStringUtils.java
24	org.apache.commons.lang3.text.translate.UnicodeEscaper.java

另外有 32 个缺陷版本定位效果较差的主要原因是缺乏有效的历史缺陷信息. STMTLocator 通过检索相似缺陷报告提取历史缺陷语句, 用于辅助进行语句级缺陷定位. 在此过程中, 相似缺陷报告来源于历史缺陷报告库, 在本文实验中构建的历史缺陷报告库共包含 1 816 个历史缺陷报告, 历史缺陷报告库的规模存在一定局限性, 导致部分被测程序的缺陷报告无法检索到与其相似度较高的历史缺陷报告. 在这种情况下, STMTLocator 将难以准确定位缺陷语句. 此外, 虽然部分被测程序的缺陷报告可以找到与其相似度较高的历史缺陷报告, 但历史陷报告中描述的缺陷与被测程序中缺陷的类型不同, 相似度较低, 导致 STMTLocator 难以根据历史缺陷语句准确定位缺陷语句.

例如对于 Chart-9, STMTLocator 检索出的相似缺陷报告同样为 CKL-170, 虽然 CKL-170 与 Chart-9 的缺陷报

© 中国科学院软件研究所 http://www.jos.org.cn

告具有较高的文本相似度, 但是根据 CKL-170 提取出的历史缺陷语句为“Calendar calendar = new Gregorian-Calendar();”, 而 Chart-9 版本中的缺陷语句为“if (endIndex < 0) {”. 两个缺陷类型不同, 语句相似度也较低, 导致 STMTLoctor 难以根据历史缺陷语句准确定位缺陷语句. 表 9 为 STMTLocator 对 Chart-9 进行缺陷语句定位后生成的可疑语句列表片段, 由于历史缺陷报告与缺陷报告描述的缺陷类型不同, 历史缺陷语句与被测程序中缺陷语句之间相似度较低, 缺陷语句仅排至第 542 位, 定位效果较差.

表 9　缺乏有效历史缺陷信息导致 STMTLocator 定位效果差的示例

排名	所在文件	所在行号	源码语句
542	**org.jfree.data.time.TimeSeries.java**	**944**	**if (endIndex < 0) {**
543	org.jfree.data.time.TimeSeries.java	945	emptyRange = true;
544	org.jfree.data.time.TimeSeries.java	947	if (emptyRange) {
545	org.jfree.data.time.TimeSeries.java	948	TimeSeries copy = (TimeSeries) super.clone();
546	org.jfree.data.time.TimeSeries.java	950	return copy;

针对上述问题, STMTLocator 可从以下 3 方面进行改进, 以提升其缺陷定位性能. 首先, 针对 STMTLocator 所使用文件级 IRBL 缺陷定位结果不够准确导致检索出的可疑文件中不包含缺陷语句问题, 可融合更多的程序静态信息, 如堆栈信息、日志信息等, 以及使用更精准的信息检索技术以提升可疑文件检索的准确率; 其次, 针对缺乏有效历史缺陷信息问题, 可基于更多缺陷库及开源项目构建历史缺陷库, 并在构建过程中过滤低质量缺陷报告, 以扩大历史缺陷库规模、提升所获取历史缺陷报告质量; 最后, 针对检索出的相似缺陷报告所描述缺陷与被测程序缺陷不相似问题, 可融合文本相似度和语义相似度进行相似缺陷报告筛选, 以更准确地提取出与被测程序缺陷报告具有相似缺陷描述的历史缺陷报告.

5.3 缺陷类型对 STMTLocator 方法的影响

不同类型的缺陷对软件缺陷定位方法的缺陷定位性能可能有不同的影响[42]. 由于 Defects4J 数据集中的缺陷版本及对应缺陷报告中仅包含部分缺陷的优先级信息, 均未包含缺陷的类型信息. 为分析不同类型的缺陷对 STMTLocator 缺陷定位性能的影响, 我们参照 Catolino 等人[42]总结的 9 种缺陷类型及定义, 人工对实验所采用的 373 个包含缺陷报告的缺陷版本进行了分类, 并统计了 STMTLocator 在不同类型缺陷上的定位结果.

表 10 为使用 STMTLocator 对不同类型缺陷的定位结果, 缺陷数量列为实验所采用的 373 个缺陷版本中各类型缺陷的数量. 其中, 与程序异常相关的缺陷数量最多, 占总缺陷数量的 82.31%, 与配置、GUI、性能、API/权限、安全和测试代码相关的缺陷数量较少, 一共仅占总缺陷数量的 17.69%. 此外, 数据集中不存在与网络和数据库相关的缺陷.

表 10　STMTLocator 对不同类型缺陷的定位结果

缺陷类型	缺陷数量	Top@1	Top@5	Top@10
配置相关	26	1	1	1
网络相关	0	0	0	0
数据库相关	0	0	0	0
GUI相关	3	0	0	0
性能相关	11	0	1	1
API/权限相关	5	0	0	0
安全相关	2	0	0	0
程序异常相关	307	6	8	10
测试代码相关	19	0	0	0
总计	373	7	10	12

从表 10 可以看出, STMTLocator 在定位与配置、性能和程序异常相关的缺陷时效果较好, 分别检查可疑语句列表的前 1、5 和 10 条可疑语句, 分别可成功定位到 7、10 和 12 条缺陷语句, 其中与程序异常相关的缺陷语句数

© 中国科学院软件研究所 http://www.jos.org.cn

量最多, 可成功定位到 6、8 和 10 条缺陷语句. 而成功定位到的与配置和性能相关的缺陷语句数量较少, 分别仅有 1 条. 对于其他类型的缺陷, STMTLocator 无法将缺陷语句排至可疑语句列表的前 10 名. STMTLocator 对与程序异常相关的缺陷定位效果最好, 分析发现, 这是由于与程序异常相关的缺陷主要包括异常、返回值问题以及由于程序逻辑问题而导致的崩溃, 此类缺陷相比其他类型的缺陷出现的频率更高, 且在对应缺陷报告中通常会提供与缺陷相关的代码片段[42], 这将有助于 STMTLocator 检索到与被测程序缺陷报告更相似的历史缺陷报告. 由于 STMTLocator 的缺陷定位性能依赖于是否能够检索到相似的历史缺陷报告, 因此 STMTLocator 在与程序异常相关的缺陷上具有较好的缺陷定位性能. 而对于其他类型的缺陷, 由于其本身出现的频率低, 并且对应缺陷报告中未提供或提供的代码片段较少, 因此 STMTLocator 的缺陷定位性能较差.

5.4 有效性威胁

5.4.1 内部有效性

内部有效性主要关注于实验的正确性. 在本文的实验中, 内部有效性主要体现在不同缺陷定位方法的实现是否正确. 为确保可疑文件检索的准确性, 本文选择使用基于信息检索的文件级开源缺陷定位工具 BugLocator 检索可疑文件, 并将其作为静态缺陷定位基线方法进行对比实验, 并根据文献[16] 将加权系数 α 设置为 0.2. 由于动态缺陷定位基线方法 DStar 和 Metallaxis 未对其源码进行开源, 我们根据文献[26,30] 中提供的算法及公式, 重新实现了 DStar 和 Metallaxis, 并根据文献 [26] 将 DStar 公式中的参数 $*$ 设置为 2. 我们在 Defects4J 数据集上对重现的 DStar 和 Metallaxis 进行了实验, 并将实验结果与 Zou 等人[38]的实证研究结果进行了比较, 以确保其正确性. 此外, 在实验过程中对所编写的代码进行了多次检查和测试, 尽可能减少内部有效性威胁.

5.4.2 外部有效性

外部有效性主要关注实验结果的一般性. 本文使用缺陷定位领域常用的 Defects4J 数据集作为实验对象, 具有一定的代表性. Defects4J 数据集中共包含来自 6 个不同项目的 395 个缺陷版本, 由于 STMTLocator 需要使用缺陷报告, 因此我们排除了 22 个不包含缺陷报告的缺陷版本, 最终在剩余的 373 个缺陷版本上进行实验. 此外, 为了分析不同类型的缺陷对 STMTLocator 缺陷定位性能的影响, 我们参照 Catolino 等人[42]总结的 9 种缺陷类型以及定义, 对 Defects4J 数据集中的缺陷进行人工分类, 为确保分类结果的准确性, 进行了多次核对. 在历史缺陷报告库构建阶段, 需要在现有的缺陷库中收集历史缺陷报告, 因此本文选择使用 JIRA 缺陷库以构建历史缺陷报告库, JIRA 在基于信息检索的缺陷定位工作中常被用于缺陷报告的收集[1,6,43,44], 具有一定的代表性.

5.4.3 构造有效性

构造有效性主要关注实验的评价指标. 本文主要使用 Top@N 和 *MRR* 作为缺陷定位性能的评价指标, Top@N 用于统计缺陷的绝对排名, *MRR* 用于衡量缺陷是否在可疑语句列表中更靠前的位置, 两者常被用于评价缺陷定位的性能[1,6,11,27,28]. 研究发现[41], 开发人员在逐一检查可疑语句列表中少量语句后, 若未能发现缺陷便会失去耐心, 这表明开发人员更加关注可疑语句列表中排名靠前, 特别是排名第一的语句, 因此本文更关注缺陷定位方法的 Top@1 指标.

6 总结与展望

针对基于信息检索的缺陷定位粒度较粗, 难以满足开发人员定位缺陷实际需求的问题, 本文提出了基于历史缺陷信息检索的语句级软件缺陷定位方法 STMTLocator. 该方法使用信息检索技术检索与缺陷报告相似的历史缺陷报告, 提取出其中包含的历史缺陷语句, 通过计算历史缺陷语句与 IRBL 检索出的可疑文件中代码语句的相似度, 以进行缺陷语句定位. 在 Defects4J 数据集上的实验结果表明, STMTLocator 在语句级的缺陷定位性能优于静态缺陷定位方法 BugLocator, 在时间开销上低于动态缺陷定位方法 Metallaxis、DStar. 在下一步工作中, 我们计划构建更大规模的高质量历史缺陷库, 使用 RNN、Transformer 等基于深度学习的文本处理模型以及更全面的程序静态信息辅助进行语句级的缺陷定位, 并探索应用于多缺陷定位场景.

References:

[1] Wong WE, Gao RZ, Li YH, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016,

© 中国科学院软件研究所 http://www.jos.org.cn

42(8): 707–740. [doi: 10.1109/TSE.2016.2521368]
[2] Liu H, Wang WH, Zhang DF. A methodology for mapping and partitioning arbitrary *n*-dimensional nested loops into 2-dimensional VLSI arrays. Journal of Computer Science and Technology, 1993, 8(3): 221–232. [doi: 10.1007/BF02939529]
[3] Li X, Li W, Zhang YQ, Zhang LM. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 169–180. [doi: 10.1145/3293882.3330574]
[4] Renieres M, Reiss SP. Fault localization with nearest neighbor queries. In: Proc. of the 18th IEEE Int'l Conf. on Automated Software Engineering. Montreal: IEEE, 2003. 30–39. [doi: 10.1109/ASE.2003.1240292]
[5] Kochhar PS, Le TDB, Lo D. It's not a bug, it's a feature: Does misclassification affect bug localization? In: Proc. of the 11th Working Conf. on Mining Software Repositories. Hyderabad: ACM, 2014. 296–299. [doi: 10.1145/2597073.2597105]
[6] 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 with English abstract). http://www.jos.org.cn/1000-9825/6130.htm [doi: 10.13328/j.cnki.jos.006130]
[7] Wen M, Wu RX, Cheung SC. Locus: Locating bugs from software changes. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: ACM, 2016. 262–273. [doi: 10.1145/2970276.2970359]
[8] Saha RK, Lease M, Khurshid S, Perry D E. Improving bug localization using structured information retrieval. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Silicon Valley: IEEE, 2013. 345–355. [doi: 10.1109/ASE.2013.6693093]
[9] Zhang J, Wang XY, Hao D, Xie B, Zhang L, Mei H. A survey on bug-report analysis. Science China Information Sciences, 2015, 58(2): 1–24. [doi: 10.1007/s11432-014-5241-2]
[10] Rahman F, Posnett D, Hindle A, Barr E, Devanbu P. BugCache for inspections: Hit or miss? In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 322–331. [doi: 10.1145/2025113.2025157]
[11] Youm KC, Ahn J, Lee E. Improved bug localization based on code change histories and bug reports. Information and Software Technology, 2017, 82: 177–192. [doi: 10.1016/j.infsof.2016.11.002]
[12] Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 199–209. [doi: 10.1145/2001420.2001445]
[13] Chen LG, Liu C. Bug localization method based on Gaussian processes. Ruan Jian Xue Bao/Journal of Software, 2014, 25(6): 1169−1179 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4430.htm [doi: 10.13328/j.cnki.jos.004430]
[14] Lukins SK, Kraft NA, Etzkorn LH. Bug localization using latent dirichlet allocation. Information and Software Technology, 2010, 52(9): 972–990. [doi: 10.1016/j.infsof.2010.04.002]
[15] Rao S, Kak A. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proc. of the 8th Working Conf. on Mining Software Repositories. Honolulu: ACM, 2011. 43–52. [doi: 10.1145/1985441.1985451]
[16] Zhou J, Zhang HY, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 14–24. [doi: 10.1109/ICSE.2012.6227210]
[17] Zhang W, Li ZQ, Du YH, Yang Y. Fine-grained software bug location approach at method level. Ruan Jian Xue Bao/Journal of Software, 2019, 30(2): 195−210 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5565.htm [doi: 10.13328/j.cnki.jos.005565]
[18] Wu RX, Zhang HY, Cheung SC, Kim S. Crashlocator: Locating crashing faults based on crash stacks. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. San Jose: ACM, 2014. 204–214. [doi: 10.1145/2610384.2610386]
[19] Wang SW, Lo D. Version history, similar report, and structure: Putting them together for improved bug localization. In: Proc. of the 22nd Int'l Conf. on Program Comprehension. Hyderabad: ACM, 2014. 53–63. [doi: 10.1145/2597008.2597148]
[20] Chen AR, Chen TH, Wang SW. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. IEEE Trans. on Software Engineering, 2022, 48(8): 2905–2919. [doi: 10.1109/TSE.2021.3071473]
[21] Takahashi A, Sae-Lim N, Hayashi S, Saeki M. A preliminary study on using code smells to improve bug localization. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Program Comprehension. Gothenburg: IEEE, 2018. 324–327.
[22] Sisman B, Kak A C. Incorporating version histories in information retrieval based bug localization. In: Proc. of the 9th IEEE Working Conf. on Mining Software Repositories. Zurich: IEEE, 2012. 50–59. [doi: 10.1109/MSR.2012.6224299]
[23] Tantithamthavorn C, Ihara A, Matsumoto KI. Using co-change histories to improve bug localization performance. In: Proc. of the 14th ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. Honolulu: IEEE, 2013. 543–548. [doi: 10.1109/SNPD.2013.92]
[24] Wang B, Xu L, Yan M, Liu C, Liu L. Multi-dimension convolutional neural network for bug localization. IEEE Trans. on Services Computing, 2022, 15(3): 1649–1663. [doi: 10.1109/tsc.2020.3006214]

© 中国科学院软件研究所 http://www.jos.org.cn

[25] Abreu R, Zoeteweij P, Van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Proc. of the 2007 Academic and Industrial Conf. Practice and Research Techniques-MUTATION. Windsor: IEEE, 2007. 89–98. [doi: 10.1109/TAIC.PART.2007.13]
[26] Wong WE, Debroy V, Gao RZ, Li YH. The DStar method for effective software fault localization. IEEE Trans. on Reliability, 2014, 63(1): 290–308. [doi: 10.1109/TR.2013.2285319]
[27] Wen M, Chen JJ, Tian YQ, Wu RX, Hao D, Han S, Cheung SC. Historical spectrum based fault localization. IEEE Trans. on Software Engineering, 2021, 47(11): 2348–2368. [doi: 10.1109/tse.2019.2948158]
[28] Xie H, Lei Y, Yan M, Yu Y, Xia X, Mao XG. A universal data augmentation approach for fault localization. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 48–60. [doi: 10.1145/3510003.3510136]
[29] Zhang Z, Lei Y, Mao XG, Chang X, Xue JX, Xiong QY. Fault localization approach using term frequency and inverse document frequency. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3448−3460 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6021.htm [doi: 10.13328/j.cnki.jos.006021]
[30] Papadakis M, Le Traon Y. Metallaxis-FL: Mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5–7): 605–628. [doi: 10.1002/stvr.1509]
[31] Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. In: Proc. of the 7th IEEE Int'l Conf. on Software Testing, Verification and Validation. Cleveland: IEEE, 2014. 153–162. [doi: 10.1109/ICST.2014.28]
[32] He T, Wang XM, Zhou XC, Li WJ, Zhang ZY, Cheung SC. A software fault localization technique based on program mutations. Chinese Journal of Computers, 2013, 36(11): 2236–2244 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2013.02236]
[33] Zhang LM, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software. ACM SIGPLAN Notices, 2013, 48(10): 765–784. [doi: 10.1145/2544173.2509551]
[34] Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2665−2690 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]
[35] Yin G, Wang T, Liu BX, Zhou MH, Yu Y, Li ZX, Ouyang JQ, Wang HM. Survey of software data mining for open source ecosystem. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2258−2271 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5524.htm [doi: 10.13328/j.cnki.jos.005524]
[36] Jatnika D, Bijaksana MA, Suryani AA. Word2Vec model analysis for semantic similarities in English words. Procedia Computer Science, 2019, 157: 160–167. [doi: 10.1016/j.procs.2019.08.153]
[37] Wang JP, Dong YH. Measurement of text similarity: A survey. Information, 2020, 11(9): 421. [doi: 10.3390/info11090421]
[38] Zou DM, Liang JJ, Xiong YF, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. IEEE Trans. on Software Engineering, 2021, 47(2): 332–347. [doi: 10.1109/TSE.2019.2892102]
[39] Meng XX, Wang X, Zhang HY, Sun HL, Liu XD. Improving fault localization and program repair with deep semantic features and transferred knowledge. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 1169–1180. [doi: 10.1145/3510003.3510147]
[40] Zeng MH, Wu YQ, Ye ZT, Xiong TF, Zhang X, Zhang L. Fault localization via efficient probabilistic modeling of program semantics. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 958–969. [doi: 10.1145/3510003.3510073]
[41] Kochhar PS, Xia X, Lo D, Li SP. Practitioners' expectations on automated fault localization. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 165–176. [doi: 10.1145/2931037.2931051]
[42] Catolino G, Palomba F, Zaidman A, Ferrucci F. Not all bugs are the same: Understanding, characterizing, and classifying bug types. Journal of Systems and Software, 2019, 152: 165–181. [doi: 10.1016/j.jss.2019.03.002]
[43] Jiang YJ, Liu H, Luo XQ, Zhu ZH, Chi XY, Niu N, Zhang YX, Hu YM, Bian P, Zhang L. BugBuilder: An automated approach to building bug repository. IEEE Trans. on Software Engineering, 2023, 49(4): 1433–1463. [doi: 10.1109/TSE.2022.3177713]
[44] Jiang YJ, Liu H, Niu N, Zhang L, Hu YM. Extracting concise bug-fixing patches from human-written patches in version control systems. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 686–698. [doi: 10.1109/ICSE43902.2021.00069]

附中文参考文献:

[6] 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. http://www.jos.org.cn/1000-9825/6130.htm [doi: 10.13328/j.cnki.jos.006130]
[13] 陈理国, 刘超. 基于高斯过程的缺陷定位方法. 软件学报, 2014, 25(6): 1169–1179. http://www.jos.org.cn/1000-9825/4430.htm [doi: 10.13328/j.cnki.jos.004430]

© 中国科学院软件研究所 http://www.jos.org.cn

[17] 张文, 李自强, 杜宇航, 杨叶. 方法级别的细粒度软件缺陷定位方法. 软件学报, 2019, 30(2): 195–210. http://www.jos.org.cn/1000-9825/5565.htm [doi: 10.13328/j.cnki.jos.005565]

[29] 张卓, 雷晏, 毛晓光, 常曦, 薛建新, 熊庆宇. 基于词频-逆文件频率的错误定位方法. 软件学报, 2020, 31(11): 3348–3460. http://www.jos.org.cn/1000-9825/6021.htm [doi: 10.13328/j.cnki.jos.006021]

[32] 贺韬, 王欣明, 周晓聪, 李文军, 张震宇, 张成志. 一种基于程序变异的软件错误定位技术. 计算机学报, 2013, 36(11): 2236–2244. [doi: 10.3724/SP.J.1016.2013.02236]

[34] 姜佳君, 陈俊洁, 熊英飞. 软件缺陷自动修复技术综述. 软件学报, 2021, 32(9): 2665–2690. http://www.jos.org.cn/1000-9825/6274.htm [doi: 10.13328/j.cnki.jos.006274]

[35] 尹刚, 王涛, 刘冰珣, 周明辉, 余跃, 李志星, 欧阳建权, 王怀民. 面向开源生态的软件数据挖掘技术研究综述. 软件学报, 2018, 29(8): 2258−2271. http://www.jos.org.cn/1000-9825/5524.htm [doi: 10.13328/j.cnki.jos.005524]

岳雷(1998－), 男, 硕士生, 主要研究领域为软件缺陷定位.

王荣存(1970－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件测试, 缺陷定位, 软件维护.

崔展齐(1984－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为软件分析, 软件测试技术.

李莉(1992－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为数据科学与智能系统, 数据融合.

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件缺陷预测, 软件缺陷定位和组合测试.

www.jos.org.cn