软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2151−2166 [doi: 10.13328/j.cnki.jos.007510] [CSTR: 32375.14.jos.007510]

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

基于代码变更语义分析的缺陷隔离方法*

刘书宁[1,2], 吴毅坚[1,2], 宋学志[1,2], 陈碧欢[1,2], 彭 鑫[1,2], 赵文耘[1,2]

[1](复旦大学 计算与智能创新学院, 上海 200438)

[2](上海市数据科学重点实验室 (复旦大学), 上海 200438)

通信作者: 吴毅坚, E-mail: wuyijian@fudan.edu.cn

摘 要: 在现代软件开发中, 频繁的代码提交和更新已成为常态, 虽然加速了功能实现, 但更可能会引入新的缺陷, 进而威胁软件的稳定性和可靠性. 一旦缺陷导致程序错误或故障, 开发团队必须迅速采取行动隔离缺陷以确保系统持续正常运行. 缺陷隔离是快速定位问题并恢复系统稳定性的关键技术手段, 但传统的增量调试 (delta debugging, DD) 方法依赖大量测试尝试, 导致在变更集合较大时性能瓶颈明显, 且缺乏对代码变更语义的有效利用, 无法精准定位与缺陷相关的代码变更. 提出了一种基于代码变更语义分析的缺陷隔离方法——DISAC. 该方法通过将缺陷引入的复合提交拆解为具有单一功能语义的原子提交, 并通过建模提交之间的顺序依赖关系, 确保隔离过程中不破坏变更间的前置依赖. 与传统的 DD 方法相比, DISAC 不仅能够返回最小的功能语义变更, 还能保留必要的上下文和依赖信息, 从而为开发人员提供更完整、精确的缺陷修复支持. 实验结果表明, 与 DD 方法相比, DISAC 在缺陷隔离效率和精度上均有显著提升. 具体而言, DISAC 在 Defects4J 数据集上的隔离效率提高了 633.65%, 在回归缺陷集上的效率提升了 733.75%. 此外, 当 DISAC 与 DD 结合使用时, 约减率分别提高了 2.36% 和 8.66%, 显著提高了隔离效果. 用户实验显示, DISAC 能提高根因确定效率约 59.90%, 准确率提升 12%. 这些结果表明, DISAC 在提高缺陷隔离精度的同时减少了不必要的变更组合尝试, 从而在复杂代码提交的缺陷隔离任务中表现出更高的效率和稳定性.

关键词: 缺陷隔离; 代码变更; 语义分析; 大语言模型

中图法分类号: TP311

中文引用格式: 刘书宁, 吴毅坚, 宋学志, 陈碧欢, 彭鑫, 赵文耘. 基于代码变更语义分析的缺陷隔离方法. 软件学报, 2026, 37(5): 2151–2166. http://www.jos.org.cn/1000-9825/7510.htm

英文引用格式: Liu SN, Wu YJ, Song XZ, Chen BH, Peng X, Zhao WY. Defect Isolation Method Based on Semantic Analysis of Code Changes. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2151–2166 (in Chinese). http://www.jos.org.cn/1000-9825/7510.htm

Defect Isolation Method Based on Semantic Analysis of Code Changes

LIU Shu-Ning[1,2], WU Yi-Jian[1,2], SONG Xue-Zhi[1,2], CHEN Bi-Huan[1,2], PENG Xin[1,2], ZHAO Wen-Yun[1,2]

[1](College of Computer Science and Artificial Intelligence, Fudan University, Shanghai 200438, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200438, China)

Abstract: In modern software development, frequent code commits and updates have become the norm, which accelerates feature implementation but may introduce new defects, thus threatening software stability and reliability. Once a defect causes program errors or failures, the development team should take quick action to isolate the defect to ensure the continuous operation of the system. Defect isolation is a key technique for rapidly locating the problem and restoring system stability. However, the traditional delta debugging (DD) methods rely on numerous testing attempts, resulting in significant performance bottlenecks under a large change set. Additionally, they fail to effectively utilize the semantics of code changes, making it difficult to accurately locate defect-related code changes. This study

* 基金项目: 国家自然科学基金 (62172099)
收稿时间: 2024-12-19; 修改时间: 2025-05-12; 采用时间: 2025-08-12; jos 在线出版时间: 2025-11-05
CNKI 网络首发时间: 2025-11-06

proposes a defect isolation method based on code change semantic decomposition—DISAC. The method decomposes composite commits introduced by the defect into atomic commits with single functional semantics. It then models the sequential dependency between commits to ensure that the dependency chain will not be broken during the isolation process. Compared to the traditional DD methods, DISAC not only returns the smallest functional semantic changes but also preserves necessary context and dependency information, thereby providing developers with more complete and accurate support for defect repair. Experimental results show that compared to the DD method, DISAC significantly improves defect isolation efficiency and accuracy. Specifically, the isolation efficiency is increased by 633.65% on the Defects4J dataset and by 733.75% on the regression defect set. Additionally, when DISAC is combined with DD, the isolation reduction rate improves by 2.36% and 8.66% respectively, significantly enhancing isolation effectiveness. User experiments show that DISAC increases root cause determination efficiency by approximately 59.90% and improves accuracy by 12%. These results demonstrate that DISAC not only improves defect isolation accuracy but also reduces unnecessary change combination attempts, thus showing higher efficiency and stability in defect isolation tasks committed by complex codes.

Key words: defect isolation; code change; semantic analysis; large language model (LLM)

在现代软件开发中, 为了快速响应不断变化的需求, 频繁的代码提交和更新已成为常态. 这种高频率的变更虽然加速了功能实现, 但也增加了缺陷被引入的风险[1−3]. 尽管持续集成 (continuous integration, CI) 等工具能够及时检测缺陷提交, 但其本身并不具备识别引发缺陷的具体变更的能力, 难以在不影响提交中正常功能的前提下恢复系统的运行, 仍需开发人员迅速定位缺陷问题, 并将缺陷相关的变更剥离出来, 但这一过程耗时耗力, 严重增加软件维护的成本.

因此自动化缺陷隔离方法成为解决这一问题的关键手段. 该方法可在检测出缺陷提交后, 进一步精准定位引入缺陷的代码变更, 并将其从其他代码中剥离. 通过回滚这些变更, 系统可以快速恢复到缺陷出现前的稳定状态, 从而避免缺陷对整体功能的影响. 这一方法不仅能够有效应对因频繁变更而引发的质量风险, 还为后续精确修复提供了时间, 是保障系统稳定性和开发效率的重要工具[4−7].

缺陷隔离是一项复杂且具有挑战性的任务[8,9]. 在实际开发中, 由于不规范的开发流程, 单次代码提交往往包含多个不同的开发活动, 这种“复合提交”现象使得缺陷隔离难度进一步加大[10−12]. 例如, 开发人员可能在一个提交中同时完成对多个功能的增强、代码重构或性能优化等变更. 当系统因缺陷出现故障时, 开发人员需要从这些交织的变更中精准定位引入缺陷的部分, 并对其进行回滚, 同时避免对其他功能或开发活动的代码产生影响. 复合提交的复杂性要求开发人员对提交内容进行详细的分析, 以明确每个变更的作用及其关联性. 在此基础上, 开发人员才能有效隔离缺陷相关的代码, 从而快速消除问题并保持其他代码的完整性和稳定性.

增量调试 (delta debugging, DD)[13]是软件工程领域经典的缺陷隔离方法. 其核心思想是通过逐步回滚代码变更组合, 实现对缺陷的隔离. 对于代码变更集合 X, 定义函数 $f(X)=\{T,F\}$, 其中 f 表示回滚指定代码变更集并进行编译测试, 返回值 T 表示缺陷未发生, F 表示缺陷被触发. DD 的目标是找到最小的变更子集 $X^*\subseteq X$ 使得 $f(X^*)=T$, 即通过回滚后缺陷消失的最小代码变更组合. 由于寻找 X^* 是一个非确定性多项式 (non-deterministic polynomial, NP) 困难问题, DD 采用分治思想折半尝试变更组合, 其过程由测试反馈驱动, 逐步缩小缺陷范围, 最终准确隔离引入缺陷的代码变更. 理论上, DD 以 $O(n^2)$ 的时间复杂度完成缺陷引入变更的查找[7].

DD 技术能够将缺陷隔离到最小变更单元 (如行级别)[14]在软件实践中展现了重要价值, 但其效率受限于大量测试的需求. 尽管研究人员们对其性能进行了系列优化[15−18], 但最新研究表明, DD 仍存在显著局限性[19]: (1) 随着变更集合规模增大, DD 性能显著下降. 其递归缩小测试范围进行隔离的方式, 导致测试次数和时间开销呈指数级增长. 在资源有限或测试成本较高的场景中, DD 可能因耗时过长而难以给出结果. 这限制了 DD 在大规模代码库和复杂提交情境中的适用性. (2) DD 缺乏对代码变更语义信息的利用能力. 其隔离过程仅依赖测试结果的变化, 而未充分考虑变更背后的功能关联和开发意图, 可能导致隔离结果中包含与缺陷无关的代码变更 (如功能增强或代码重构). 这些多余变更的回滚不仅无助于缺陷修复, 反而可能破坏系统功能或干扰其他开发活动. (3) 即使 DD 能够隔离出最小缺陷变更, 其结果往往缺乏必要的上下文信息. 孤立的变更片段可能无法准确呈现缺陷的完整背景, 开发人员仍需回溯原始变更集以理解缺陷的根源或评估影响范围, 增加了工作负担并延长了修复周期.

解决上述问题面临着两大挑战: 一方面是如何对复杂变更任务进行合理拆解; 另一方面是如何在代码变更之

间建立语义关联. 单独的 DD 技术并不能有效应对这些挑战. 当前软件工程领域内的代码变更拆解技术具有解决以上问题的能力, 这些方法通过构建变更之间代码层面的依赖关系, 将代码变更拆分为多个依赖完整的独立提交. 但是该类方法忽略了开发活动层面的功能及语义信息, 导致具有依赖关系的代码变更可能隶属于两个不同的开发活动, 甚至是两个独立的功能, 其拆解结果的粒度难以直接满足缺陷隔离任务需求. 此外, 即使能够拆解为较小粒度, 拆解后依然面临着组合搜索问题, 组合搜索的结果可能涉及多个独立提交, 但提交之间的关系往往难以确定. 开发人员无法判断缺陷是由这些提交的共同作用引起, 还是源于某个单一提交. 这种不确定性使得组合搜索的结果难以直接指导修复工作.

因此, 本文提出了一种基于代码变更语义分析的缺陷隔离方法——DISAC. 该方法由变更拆解和缺陷隔离两个核心模块组成. 在变更拆解模块中, 方法使用结合了程序分析技术和大语言模型的智能体对缺陷引入提交 (defect inducing commit, DIC) 进行分解, 得到一系列的具有单一功能语义的原子提交, 例如将 DIC 中存在对某一功能的重构和增强拆成两个独立原子提交, 为后续缺陷隔离提供更高的精度. 在缺陷隔离模块中, DISAC 对一系列原子提交进行序列建模, 构建提交之间的顺序依赖关系, 确保在隔离过程中不会破坏变更间的前置依赖. 随后, 利用二分查找精准定位引发缺陷的原子提交及其相关依赖并进行隔离. 与传统的 DD 方法相比, DISAC 不仅返回了最小的功能语义变更, 还保留了必要的上下文和依赖信息, 为开发人员后续的缺陷修复提供了更完整且高效的支持.

为了验证方法的有效性, 本文在 Defects4J 数据集[20]和回归缺陷集[21]中进行了对比实验和消融实验, 以全面评估 DISAC 的性能和各组件的贡献. 对比实验结果表明, 与最先进的 DD 技术相比, DISAC 的隔离效率提升分别为 633.65% 和 733.75%; 隔离效果在 Defects4J 数据集上差别不大, 但在代码变更规模较大的回归缺陷集上, 成功隔离缺陷的数量提高 12.42%. 此外, 当 DISAC 与 DD 结合使用时, 隔离效率分别提升 19.47% 和 30.30%, 隔离约减率分别提高了 2.36% 和 8.66%. 消融实验结果显示, 语义拆解部分各模块均对 DISAC 方法的隔离效果有较大贡献. 最后, 为了评估 DISAC 对开发人员理解缺陷根因的支持能力, 本文设计了用户实验, 结果表明, 与 DD 方法相比, DISAC 提高了用户根因确定效率约 59.90%, 根因确定准确率提升约 12%.

综上所述, 本文的主要贡献如下.

- 提出了一种基于代码变更语义分析的缺陷隔离方法. 能够有效地对引发故障的缺陷变更进行及时隔离, 提升了缺陷隔离的精度和效率, 在保证其他相关功能正常运转的同时, 也为开发人员提供了有价值的缺陷变更信息.
- 方法研制了缺陷隔离的工具, 并在 Defects4J 数据集和回归缺陷数据集上构建了充分的实验, 验证了相比传统的 DD 方法, DISAC 在缺陷隔离任务上的效果和性能均有显著的提升. 另外, 通过用户实验证实了工具的实用性.

本文第 1 节介绍基于代码变更缺陷隔离研究相关工作. 第 2 节阐述本文提出的方法思路及具体细节. 第 3 节介绍实验的设计, 包括实验数据、基准方法、评价指标、实验流程以及关键参数设置. 第 4 节对实验结果进行评估和详细分析. 第 5 节总结全文, 并对未来的研究工作进行展望.

1 相关工作

本节介绍了基于代码变更语义分析的缺陷隔离的相关工作, 即增量调试、变更集分解等相关工作.

1.1 增量调试

增量调试是一个在保留特定属性的同时查找元素子集的自动化过程, 元素是可以按块、行或任何指定单位进行计算的源代码变更, 在缺陷隔离研究中, 增量调试算法通过最小化代码变更来定位引入缺陷的代码. 传统的增量调试算法是 Zeller 等人[7]提出的 ddmin 算法, 它基于对元素的二分查找来实现, 用于确定两个软件版本之间的最小引起缺陷的变更. 此后, ddmin 被应用于各种特定领域, Misherghi 等人[15]引入了 HDD, 用树结构来提高增量调试效率; Heo 等人[17]提出了 CHISEL 方法进行代码精简, 使用强化学习选择 ddmin 中更有可能满足目标属性, 减小增量调试的搜索范围, 从而增强 ddmin 的有效性. 但是 ddmin 的核心算法并没有改变. 尽管 ddmin 能够发现关键变化, 但是这种算法本身的计算开销是巨大的, 因为必须测试变化元素的大量可能的组合来确定该组合是否包含关键变化. Wang 等人[18]提出的 ProbDD 是一种不同于 ddmin 的算法, 在概率模型的指导下, 它根据测试的历史

过程来估计代码变更元素是关键变更的概率, 并用贪心算法选择元素. ProbDD 效率很高, 但它受到代码更改元素彼此独立的假设的影响, 这个假设在实际代码提交中经常被打破. 而缺少依赖变更元素会导致编译错误 (compile error, CE), 增量调试过程中为了通过测试编译, 会引入完全与缺陷无关的变更. Hashimoto 等人[16]提出了 DDJ 算法, 通过对源代码变更进行预处理, 对存在依赖关系的代码变更进行分组, 一定程度上解决代码变更元素之间具有依赖关系的问题. 然而, 具有依赖关系的变更代码不一定都是引入缺陷的代码, 同时这种算法依然依赖 ddmin. Song 等人[19]提出的 C2D2 算法是目前最新的 DD 算法, 使用基于矩阵的搜索机制, 解决缺少依赖关系导致的编译错误问题, 但是在大规模数据上依然有局限性.

增量调试技术能够用于隔离引入缺陷的关键更改, 但由于真实代码变更中具有复杂的依赖关系, 这些依赖关系导致增量调试中存在大量处理变更代码间信息传播的过程, 不仅测试过程花费大量开销, 而且最终定位到的缺陷范围也可能较大, 缺陷隔离的效率和准确性受到严重限制. 与增量调试的过程不同, 本文基于代码变更的语义, 通过利用大语言模型对代码变更的分析, 在减少开销的同时, 解决了依赖关系对缺陷隔离的影响, 从而实现了更精确地隔离引入缺陷的代码变更.

1.2 变更集分解

目前已经有了多种用于解决变更代码分解问题的方法, Herzig 等人[10]使用了一种基于启发式的算法来处理解开纠结变化, 他们认为它们之间的关系是信任投票者. 类似地, Wang 等人[11]提出了 CoRA, 这是一种将提交分解为不同部分并为审查者生成简洁描述的自动方法, 它采用了一种通过代码依赖分析和基于树的相似代码检测的启发式方法来分解复合提交以进行代码审查. Barnett 等人[22]提出了 CLUSTERCHANGES, 它依赖于方法的定义和引用等连接来聚类相同的变化. Guo 等人[23]提出了 CHGCUTTER 技术, 利用程序依赖关系构建相关的变更子集, 同时筛选相关的测试用例, 减少了进行回归测试的时间. Muylaert 等人[24]基于程序依赖图和一次提交中抽象语法树上的变更, 使用程序切片技术来分解复杂的变更集. Dias 等人[25]开发了 EpiceaUntangler, 基于代码结构预测两个变更属于同一个集合的概率, 分解复杂的变更为原子提交, 并通过一些工作表明了开发人员参与分解过程的必要性. Taeumel 等人[26]提出了一种名为 Thresher 的交互式图形工具, 采用可适应的脚本支持开发人员对变更进行分组和提交, 特别是对于细粒度的更改跟踪. Yamashita 等人[27]提出了 ChangeBeadsThreader (CBT), 这是一种用于拆分和合并变更集的交互式环境, 该环境基于 4 个度量来分解变更集, 并允许用户完全手工分割/合并变更集. Shen 等人[28]提出了 SmartCommit 方法, 提出了一种可扩展的图表示方法, 它提出了变更代码之间的多种启发式连接, 并设计了基于图划分的变更集分解算法.

本文的研究与这些工作的重要不同之处在于, 这些工作仅考虑了变更代码之间的依赖和调用关系, 本文的变更集拆解方法将一次代码提交中的变更代码分解为多个具有单一活动的子提交, 需要同时考虑变更代码的语义信息, 因此, 本文提出的程序分析和大语言模型的提交拆解方法, 保证了每个拆解后的变更具有独立性.

2 基于代码变更语义分析的缺陷隔离方法

2.1 总体方法框架

DISAC 方法主要包含 3 个主要阶段: 依赖关系构建、代码变更拆解和缺陷变更隔离, 如图 1 所示.

在依赖关系构建阶段, 首先检测两个版本代码间的所有代码变更, 并将其结构化为变更代码块; 在变更过滤中检测和排除与缺陷开发无关的变更活动, 如测试代码变更、非代码变更、代码风格调整等, 缩小检测规模, 降低后续处理的复杂性和分析成本. 之后进行依赖分析, 基于程序分析技术构建变更代码块之间的依赖关系图, 识别变更代码块之间的关联, 为后续变更集拆解提供依据.

在代码变更拆解阶段, 以第 1 阶段得到的代码变更依赖关系图作为输入, 首先对变更代码块之间存在的强依赖关系进行预合并, 再采用基于大语言模型的智能体进行语义拆解, 直到完全分解为多个具有单一功能语义的变更子集, 将分解得到的变更子集及其标签作为结果输出.

在缺陷变更隔离阶段, 将分解得到的每个变更子集视为一次提交, 基于变更子集间的依赖关系图进行拓扑排

序生成提交序列, 之后使用二分查找的方法, 找到引入缺陷的变更子集, 即隔离到的具有单一功能的缺陷引入. 该方法将缺陷引入变更的组合搜索任务转化为在提交序列中的线性查找任务, 利用单一开发语义的原子提交特性及其顺序依赖关系, 有效简化了缺陷定位过程.

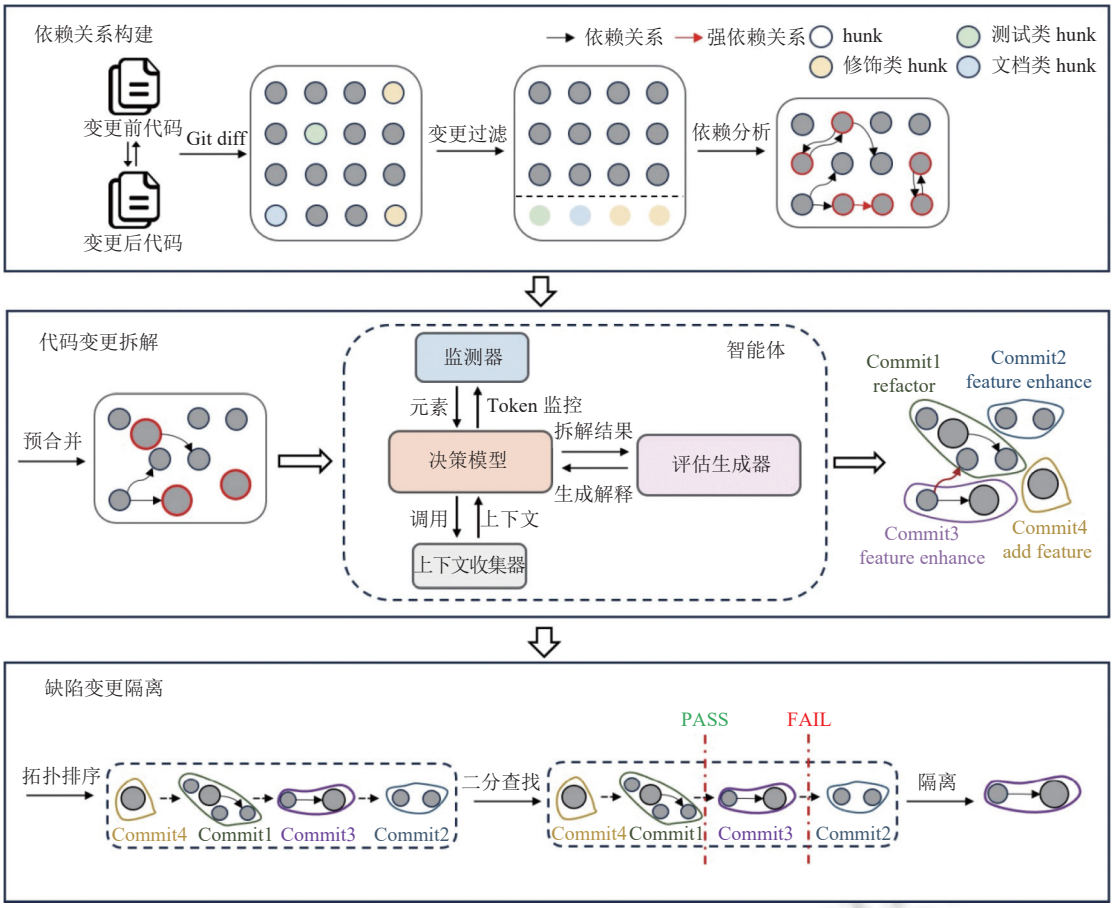

图 1 基于代码变更语义分析的缺陷隔离方法框架图

2.2 依赖关系构建

2.2.1 变更代码块构建

变更代码块的构建过程通过使用 Git 工具来检索两个版本之间的所有代码更改. 具体来说, Git 工具对比两个版本的代码, 并将它们之间的差异以一种结构化的方式呈现. 每个差异块被称为变更代码块 (diff hunk, 在下文中简称 hunk), 它通常由一组相邻的具体增删代码行组成. 在本文的研究中, hunk 被视为构成代码变更的最小单元, 并作为基本的分析单位.

2.2.2 变更过滤

针对给定 DIC 中的变更代码集合 $H=\{h_1,h_2,\ldots,h_n\}$, 变更过滤的目的是得到 $H^*=H\backslash H_{\text{irrel}}$, 其中 H_{irrel} 表示确定与缺陷无关变更代码块的集合. 定义 $h_i\in H_{\text{doc}}\cup H_{\text{test}}\cup H_{\text{style}}$, 则 $h_i\in H_{\text{irrel}}$, 其中 H_{style} 表示代码风格变更, H_{doc} 表示文档相关变更, H_{test} 表示测试相关变更. 变更过滤通过缩小后续拆解范围, 降低缺陷分析的复杂性, 提高缺陷隔离的效率.

具体来说, DISAC 通过变更代码所在的文件路径和文件名使用正则检测判断 h_i 是否属于测试相关变更; 通过分析变更代码所在的文件名后缀以及代码中的引用情况判断 h_i 是否属于无关文档变更; 通过语法解析工具解析代

码为抽象语法树 (abstract syntax tree, AST), 判断 h_i 所包含的变更代码是否只包含空格、缩进、换行的变更或仅涉及标识符 (变量名) 的变化. 若 $\Delta_{\mathrm{AST}}(h_i)=0$, 则 $h_i \in H_{\mathrm{style}}$; 或 $\Delta_{\mathrm{AST}}(h_i)\neq 0$ 且 $\Delta_{\mathrm{NonID}}(h_i)=0$, 则 $h_i \in H_{\mathrm{style}}$, 其中 $\Delta_{\mathrm{AST}}(h_i)$ 表示 h_i 变更前后 AST 的差异, $\Delta_{\mathrm{NonID}}(h_i)$ 非标识符节点的变更差异.

2.2.3 变更依赖分析

给定过滤后的变更集 H^*, DISAC 沿用变更代码分解中构建依赖关系的方法使用 JavaParser (https://javaparser.org/) 进行静态程序分析, 将前一个程序和当前程序分别转换为相应的 AST, 通过比较之前的 AST 和当前的 AST 之间的差异来提取细粒度的代码变更, 识别这些细粒度代码变更之间的依赖关系. 本文使用一个三元组 $D<H,E,T>$ 表示 hunk 之间的依赖关系. 其中 H 表示变更代码集, 其中每个元素代表一个具有唯一 ID 的变更代码块. E 表示 H 中存在的依赖关系的集合, $E=\{(s,t,\tau)|(s,t)\in H\times H, s\neq t, \tau\in T\}$ 表示 E 中的每一条边连接两个不同的代码变更块, 并通过依赖属性进行描述. T 表示依赖边属性的集合, $T=\{T_{\mathrm{dep}}, T_{\mathrm{sim}}\}$, 共包含了两种依赖属性: 代码依赖 T_{dep} 和相似依赖 T_{sim}. 与传统的变更代码拆解工作不同, DISAC 方法将依赖关系作为拆解的依据之一, 结合语义分析进一步优化拆解过程. 因此仅使用依赖于代码的结构性依赖和相似性依赖关系, 避免了因过度细化依赖关系而导致的过度合并风险.

T_{dep} 是代码分解工作中常用的属性, 表示代码之间的继承、调用以及数据流等关系, DISAC 复用这种传统的依赖分析关系, 检测代码之间的依赖关系, 并将其映射到相应的变更子集上, 帮助识别变更之间的基础依赖结构. 此外, 本文使用 T_{sim} 作为依赖关系的一种补充, 表示两个变更之间存在相似关系. 本文提出了一种结合文本相似性和语法相似性的计算方法, 其计算公式如下:

$$T_{\mathrm{sim}}(h_i,h_j)=S_{\mathrm{txt}}(h_i,h_j)+S_{\mathrm{ast}}(h_i,h_j).$$

文本相似性 S_{txt} 基于字符级别的重合程度计算, 具体公式为:

$$S_{\mathrm{txt}}(h_i,h_j)=\frac{|C(h_i)\cap C(h_j)|}{\max(|C(h_i)|,|C(h_j)|)},$$

其中, $C(h_i)$ 表示 h_i 的字符集, $|C(h_i)|$ 为该集合的大小, $\left|C(h_i)\cap C(h_j)\right|$ 为两个变更块中共有的字符数量. 语法相似性 S_{ast} 则通过比较 AST 节点的相似性来度量, 其公式如下:

$$S_{\mathrm{ast}}(h_i,h_j)=\frac{|N(h_i)\cap N(h_j)|}{\max(|N(h_i)|,|N(h_j)|)}.$$

类似地, $N(h_i)$ 表示 h_i 的 AST 节点集合, $|N(h_i)|$ 为节点集合的大小, $\left|N(h_i)\cap N(h_j)\right|$ 为两个变更块中共有的节点数量. 为了判断 h_i 和 h_j 是否具有相似关系, 引入了相似度阈值 θ (在本文的研究中设置 θ 为 0.7), 当 $T_{\mathrm{sim}}(h_i,h_j)\geqslant\theta$ 时, 则认为 h_i 和 h_j 之间存在相似性依赖关系 T_{sim}.

2.3 代码变更拆解

2.3.1 预合并

为了尽可能避免大语言模型过细粒度的拆解打破变更子集的完整性, DISAC 根据依赖关系对 DIC 中的变更代码块进行初步的拆解与合并. 与传统方法不同[22−24,28], DISAC 的思想并非将所有具有代码依赖关系的变更都合并为同一提交, 而是关注无法分割的强依赖关系, DISAC 定义了两种基于依赖关系的强依赖关系 $R=\{R_{\mathrm{circ}}, R_{\mathrm{dep}}\}$.

R_{circ} 是具有环状依赖的关系. 当两个或多个变更代码块 h_i 和 h_j 之间存在相互依赖时, 如 $(T(h_i\rightarrow h_j)=T_{\mathrm{sim}}\vee T_{\mathrm{dep}})\wedge(T(h_j\rightarrow h_i)=T_{\mathrm{sim}}\vee T_{\mathrm{dep}})$, 则认为 h_i 和 h_j 之间存在具有环状依赖的强依赖关系, 记为 $R_{\mathrm{circ}}(h_i,h_j)$. 这种依赖通常表示 hunk 之间存在循环引用等紧密耦合结构, 本文采用 Tarjan 算法检测依赖关系图, 将形成环状依赖的变更合并为一个不可拆分的块, 避免在后续拆解过程中破坏依赖闭包.

R_{dep} 是与描述符和标识符变更相关的依赖关系. 标识符是指变量、方法、类、接口、包等的名称, 描述符是字段的数据类型、方法的参数列表 (包括数量、类型以及顺序) 和返回值, 它们在代码中起着重要的链接作用, 当这些标识符或描述符被修改时, 往往会影响其他依赖它们的代码. 若 h_i 更改了代码描述符或标识符, 且另一个代码块 h_j 对它有代码依赖关系, 即 $T(h_j\rightarrow h_i)=T_{\mathrm{dep}}$, 则 h_i 和 h_j 之间存在一种描述符相关的强依赖关系, 记为 $R_{\mathrm{dep}}(h_i,h_j)$.

这种依赖关系通常用于追踪代码中的结构性变更, 例如方法参数的调整影响了方法的调用, 从而指导合并变更代码块.

在 DISAC 的初步分解与合并策略中, 我们合并具有以上两种强依赖关系的变更代码块, 避免在后续过程中打破变更的完整性, 从而保持代码的一致性和可编译性.

2.3.2 基于大语言模型的拆解

给定预合并后的代码变更集合 H 和 H 中的依赖关系 D. DISAC 提出一种基于智能体的变更语义拆解方法, 将 H 进一步拆解为多个仅包含单一开发活动的子提交集 C. DISAC 中的智能体系统基于阿里通义大模型 Qwen-turbo[29], 共包含 4 个角色.

● 决策器. 负责根据输入的 H 和 D 对 H 进一步拆解, 指导大语言模型拆解为子提交集 $C=\{c_1,c_2,\ldots,c_n\}$, $c_i=\{h_m, h_n,\ldots,h_k\}$, 并传入评估消息生成器进一步处理与评估.

● 监测器 (优化器). 监测器的核心功能是负责监控输入令牌 (token) 的长度, 由于大语言模型的输入存在长度限制, 为增强方法的适用性, 在将提示输入到决策模型前, 监测器判断 token 的长度是否超出限制. 如果监测到 token 超出限制, 监测器将选择变更代码块 h_i 生成变更摘要 s_i, 并令 $h_i=s_i$, 即用生成的变更摘要代替原始的变更代码元素, 从而有效减少 token 长度. 为了达成这一目标, DISAC 制定了如下的 hunk 选择机制指导监测器的操作: (1) 整个文件的增加或删除. 在 git diff 中, 整个文件的增加和删除通常会将该文件所有代码的增删作为一个变更块, DISAC 将其总结为路径变更的摘要, 从而压缩数据量. (2) 整个类的添加或者删除. 当整个类被添加或删除时, DISAC 保留类中各方法的签名, 并生成每个方法的描述, 以替代完整的类代码. (3) 整个方法的添加或者删除. 对于长方法的添加或者删除, DISAC 保留方法签名后, 将具体代码实现转换为方法的描述. (4) 相似更改. 对于涉及相同变量名的重构类更改, 这些更改在预合并过程中已合并为一个 h_i 中, DISAC 将对其生成变更摘要.

监测器会逐步应用上述选择机制, 替代和反馈相关代码元素, 直到 token 长度符合限制要求. 若通过以上策略仍无法满足需求, 监测器将采用退步策略——将 H 中所有类或方法添加的 h_i 直接赋予功能添加的语义, 再将这些元素仅根据依赖关系进行拆解. 最后 H 中剩余部分按照交由智能体继续拆分.

● 上下文收集器. 上下文收集器的主要职责是为决策模型提供代码变更的上下文信息, 在代码变更的上下文信息不足时, 补充必要的代码元素信息. 上下文收集器提供了两种可调用的方法: get_deleration_method、get_comment, 分别用于获取特定代码变更所在方法的完整代码, 以及所在类或方法在代码中的注释等描述信息. 这两种方法为决策模型提供了更全面的上下文, 从而帮助其更好地理解代码变更的背景和影响. 上下文收集器的使用由大语言模型根据实际需求自行决策使用, 确保在必要时补充上下文信息.

● 评估消息生成器. 负责对决策模型的拆解结果生成相应描述, 并对这些描述的变化进行评估. 该生成器对 C 中的 c_i 生成提交日志 l_i. 对拆分后的每个提交生成具有特定格式的描述作为提交日志. 之后将所有的提交日志返回给决策模型, 并提示其检查是否存在包含多个开发活动的提交日志需要进一步拆分, 或者是否有相似提交日志 l_i、l_j 对应的 c_i、c_j 可以合并, 决策模型进一步修改拆分的子提交集. 此外, 评估消息生成器通过对生成的描述文本变化的评估, 判断输出是否趋于收敛. 若 $k=3$ 步内输出没有明显变化, 则终止流程以防止无限迭代; 同时, DISAC 设定了最大迭代步骤 $h=10$, 当达到此上限时, 模型迭代也将被强制终止. 这一机制有效地平衡了迭代精度与效率, 确保评估过程在合理时间内完成, 同时避免了过度迭代带来的资源浪费.

为了限制模型的输出以及方便模型理解任务, DISAC 通过构建标准提示 (Prompt)[30], 为决策模型输入设计提示. DISAC 共有 4 个 Prompt 类别, 包括提交变更分解、变更摘要生成、评估消息生成和评估消息反馈, 下面给出用于本任务的 Prompt, 每个提示均包括任务背景、格式规约、输入信息等内容, {{…}}内的文本是占位符, 在运行时将被替换为 JSON 格式的输入[31].

● 提交变更分解. (1) 任务背景: 明确了拆解变更集的任务. (2) 格式规约: 规定了输出为拆解后的多个子提交. (3) 输入信息: 包括具体变更代码和依赖关系. 其 Prompt 模板见图 2.

● 变更摘要生成. (1) 任务背景: 明确了为代码变更生成摘要的任务. (2) 格式规约: 针对不同类型的代码变更, 根据选择机制用不同的方式生成摘要. (3) 输入信息: 为具体变更代码信息. 其 Prompt 模板见图 3.

```
You are an experienced and detail-oriented Java programmer, tasked with decomposing
code changes into distinct development activities. I will give you code change hunks
from a commit and dependencies between these hunks. Your task is to decompose these
hunks into sub-commit sets, where each sub-commit represents a distinct development
activity. Each sub-commit should include Included_Hunks listing the hunks it contains.

Diff Hunks: {{code change hunks}}
Dependencies: {{dependencies between these changes}}
```

图 2 提交变更分解提示

```
You are an experienced and detail-oriented Java programmer, tasked with summarizing
code changes into meaningful description. I will give you a code change hunk , and your
task is to summarize the changes based on the type of modification. The types of
changes can include: path, class, method, or refactor, each type requires a different
summarization strategy:
File path changes: Summarize as a path-level change.
Class additions or removals : Retain the method signatures within the class and
generate a description for each method.
Long method additions: Retain the method signature and convert the code implementation
into a concise description.
Refactor changes: Summarize the refactor with brief descriptions of the changes made.
Format your response as a clear and concise textual description.

Diff Hunk: {{code change hunks}}
```

图 3 变更摘要生成提示

● 评估消息生成. (1) 任务背景: 明确了为子提交生成评估消息的任务. (2) 格式规约: 规定了输出为一段子提交日志. (3) 输入信息: 将之前得到的子提交及其包含的代码变更块作为输入. 其 Prompt 模板见图 4.

```
You are an experienced and detail-oriented Java programmer, tasked with generating
commit message for each sub-commit. I will give you code change hunks from a sub-commit.
Your goal is to provide a structured commit log including the activity type and a brief
description summarizing the sub-commit.

Diff Hunk: {{code change hunks in a sub-commit}}
```

图 4 评估消息生成提示

● 评估消息反馈. (1) 任务背景: 根据生成的消息进行评估和反馈, 判断是否存在包含多个开发任务的提交日志需要进一步拆分, 或是否具有相似的提交日志需要进行合并. (2) 格式规约: 输入评估消息, 输出判定的结果正确或错误, 如果错误, 输出新的子提交拆分结果. (3) 输入信息: 将所有子提交生成的评估消息作为输入. 其 Prompt 模板见图 5.

```
I will give you the commit messages for each sub-commit. Your task is to assess whether
they require further refinement, such as splitting logs that include multiple
development tasks or merging logs with similar purposes. The output is a decision
whether the evaluation is TRUE or FALSE. If the result is FALS, provide a revised set
of sub-commits with messages.

Sub-commit Logs : {{list of sub-commit messages}}
```

图 5 评估消息反馈提示

在代码拆解流程中, DISAC 最终的输出为一个由独立开发活动构成的提交集合 $\mathcal{C}=\{c_1,c_2,\ldots,c_n\}$, 其中每个子提交 $c_i\in\mathcal{C}$ 代表一个独立的开发活动, 由多个代码变更片段构成 $c_i=\{h_m,h_n,\ldots,h_k\}$. 此外, 还得到了子提交之间的依赖关系集合 $\mathcal{D}=\{(c_i,c_j)\big|c_i,c_j\in\mathcal{C}\wedge c_i\mapsto c_j\}$, 如果 $(\exists h_x\in c_i,\exists h_y\in c_j)\wedge h_x\mapsto h_y$, 则 $c_i\mapsto c_j$, 其中 $c_i\mapsto c_j$ 表示 c_i和c_j 之间存在顺序依赖关系, 即 c_i 必须在 c_j 之后提交.

特别地, 为确保后续缺陷隔离的正确进行, 我们对依赖关系中可能存在的环路进行了处理. 如果 $(\exists h_x,h_z\in c_i,\exists h_y,h_w\in c_j)\wedge h_x\mapsto h_y\wedge h_w\mapsto h_z$, 则合并 c_i、 c_j 为 c_{ij}. 具体而言, 若存在两个或以上的子提交, 满足其中的变更块之间存在互为依赖的关系, DISAC 会将它们合并为一个新的子提交.

2.4 缺陷变更隔离

2.4.1 构建提交序列

根据大模型拆解的结果 $\mathcal{C}=\{c_1,c_2,\ldots,c_n\}$ 以及依赖关系集合 $\mathcal{D}=\{(c_i,c_j)\big|c_i,c_j\in\mathcal{C}\wedge c_i\mapsto c_j\}$, 其中 $\forall c_i\mapsto c_j$ 表示

c_i 必须在提交 c_j 之后, 即所有的提交都在其依赖项提交之后进行. 因此, 依赖关系集合 $\mathcal{D}$ 描述了子提交之间的依赖顺序, 构成了一个有向无环图. 在本文中, 选择 Kahn 算法[32]用于对由子提交集合 $\mathcal{C}$ 和依赖关系集合 $\mathcal{D}$ 构成的有向无环图进行拓扑排序, 从而生成一个满足依赖关系的线性序列. Kahn 算法基于贪心思想, 首先, 对于图中的每个节点 (子提交 c_i), 计算其入度 (即依赖于其他节点的数量), 这表示每个子提交被其他子提交依赖的程度, 将所有入度为 0 的节点放入一个队列中, 这些节点表示没有依赖关系的子提交, 可以作为排序的起点. 反复从队列中取出入度为 0 的节点, 将其加入最终的排序结果. 同时, 移除该节点及其出边 (即该节点对其他节点的依赖关系). 对于每条出边所指向的节点 (即依赖当前节点的子提交), 将其入度减 1. 如果某个节点的入度减少至 0, 则将其加入队列. 重复上述过程, 直到队列为空. 通过该算法能够得到满足所有依赖关系的线性有序子提交序列 $\mathcal{C}=\{c_1,c_2,\ldots,c_n\}$, 其中每个子提交 c_i 的提交顺序都满足 $\mathcal{D}$ 中的依赖关系 (c_i,c_j), 即当 $c_i \mapsto c_j$ 时 c_i 出现在 c_j 之后.

该序列确保了每个子提交按照正确的提交顺序进行排列, 避免了依赖冲突, 保证了提交的依赖关系得到严格遵循. 这不仅为后续的缺陷隔离提供了清晰的操作顺序, 还有效减少了由于提交顺序错误导致的依赖问题, 确保后续操作不受先前未处理变更的干扰, 便于在后续流程中更加高效地识别和隔离缺陷.

2.4.2 缺陷变更隔离

给定拆解后的提交集合 $\mathcal{C}$ 中, DISAC 需要找到实际引发缺陷的提交 $c_{\text{bug}} \in \mathcal{C}$, 并进行隔离. 与 DD 不同, DISAC 已经根据集合之间的依赖关系 $\mathcal{D}$ 将提交集合 $\mathcal{C}$ 建模为具有线性关系的提交序列进行搜索. 因此, 查找 c_{bug} 的过程变为在线性区间 $[c_1,c_n]$ 中寻找真实的缺陷引入提交 c_{bug}, 其中 c_1 是最早的提交, c_n 是最新的提交.

DISAC 在线性区间上使用二分查找算法进行搜索. 函数 $F(c_i)$ 用于表示提交 c_i 对代码的测试结果, 返回值是通过 (PASS) 或失败 (FAIL), 目标是找到第 1 个导致测试结果从 PASS 转变为 FAIL 的提交 c_{bug}. 在每一步, 选择当前区间的中点提交 c_m, 即 $m=\left\lfloor \dfrac{i+j}{2} \right\rfloor$, 其中 i 和 j 是当前搜索区间的左右边界索引, 测试该中点提交 c_m 的结果 $F(c_m)$. 如果 $F(c_m)=\text{FAIL} \wedge F(c_{m-1})=\text{PASS}$, 则 c_m 是引入缺陷的提交, 即 $c_{\text{bug}}=c_m$, 停止搜索; 如果 $F(c_m)=\text{PASS}$, 则意味着缺陷出现在 c_m 之后, 因此将搜索区间缩小为 $[c_{m+1},c_j]$; 如果 $F(c_m)=\text{FAIL} \wedge F(c_{m-1})=\text{FAIL}$, 则意味着缺陷出现在 c_m 之前, 将搜索区间缩小为 $[c_i,c_{m-1}]$. 根据上述步骤缩小搜索区间, 直到区间内只有一个提交或找到符合 $F(c_i)=\text{FAIL} \wedge F(c_{i-1})=\text{PASS}$, 则可以确定 $c_{\text{bug}}=c_i$.

由于 DISAC 已经根据提交之间的依赖关系 $\mathcal{D}$ 将 $\mathcal{C}$ 建模为线性区间, 因此可以确保在查找过程中不会违反提交之间的依赖顺序. 例如, 如果 $c_i \mapsto c_j$ (即 c_i 依赖于 c_j), 则在二分查找过程中, 任何包含 c_i 的区间必须在包含 c_j 的区间之前. 这保证了查找过程中的逻辑一致性.

在找到缺陷提交 c_{bug} 后, DISAC 选择子集 $\mathcal{C}^* \subseteq \mathcal{C}$, 其中 $\mathcal{C}^*$ 中包含 c_i 及其前序依赖的提交, 即隔离到的与缺陷相关的所有提交.

3 实验设计

3.1 研究问题

为了评估本文提出的基于代码变更语义分析的缺陷隔离方法 DISAC 的有效性, 我们研究了以下几个问题.

- RQ1: 与增量调试方法相比, DISAC 方法在缺陷隔离任务中的性能如何?
- RQ2: DISAC 方法是否能够增强现有增量调试技术的性能?
- RQ3: 本文提出的 DISAC 方法中不同组成部分如何影响缺陷隔离总体方法的性能?
- RQ4: DISAC 方法与增量调试方法在辅助缺陷根因确定任务中的作用和价值如何?

3.2 实验数据

为了评估 DISAC 方法在缺陷隔离任务上的性能, 我们使用了 Defects4J[20]和回归缺陷[21]两个数据集. Defects4J 是一个广泛使用的经过人工验证的缺陷数据集, 它由 835 个从真实 Java 项目中收集到的缺陷及其修复组成. 对于每个缺陷, 它提供了原始缺陷版本 (V_{obug}) 和缺陷修正版本 (V_{ofix}), 它们之间的代码差异包含了与缺陷无

关的变化[33], 我们使用这些代码差异作为算法的输入. 此外, Defects4J 还提供了缺陷修复的关键更改, 这些更改由人工最小化并进行验证, 可以作为本文 RQ1 和 RQ4 实验中的真实标签 (ground truth). 在之前的研究中[19]发现了 26 条数据的修订版本 V_{obug} 和 V_{ofix} 是不可访问的, 因此最终有 809 个缺陷作为本文的实验数据. RegMiner 构建的回归缺陷数据集包含 1 035 个回归缺陷, 据我们所知, 它是唯一一个包含缺陷引入变更的大规模数据集, 经过人工验证发现[19]其中有 10 条数据是重复的或与不稳定的测试有关, 在本文的实验中使用了筛选后的 1 025 个回归缺陷. 且该数据集在 C2D2[19]中已通过人工验证确定了关键缺陷变更集, 可以作为 RQ1 和 RQ4 实验判断准确率的依据.

本文研究还观察到, 回归缺陷数据集中的代码变更规模 (hunk 的数量) 显著高于 Defects4J, 我们根据代码变更块的数量对数据集进行了离群点检测, 并根据四分位点和离群值划分为 5 个区间, 如表 1 和表 2 所示. 进而, 我们比较本文所提出的 DISAC 缺陷隔离方法在不同规模数据集上进行缺陷隔离时的性能和准确性, 将回归缺陷数据集按照上述 5 个区间进行实验.

表 1 回归缺陷数据集中分组数量统计

区间	数据量
低位区间 (1–46)	622
次低区间 (47–92)	122
次高区间 (93–138)	47
高位区间 (139–176)	31
离群值区间 (177+)	94

表 2 Defects4J 数据集中分组数量统计

区间	数据量
低位区间 (1–4)	404
次低区间 (5–8)	141
次高区间 (9–12)	65
高位区间 (13–16)	43
离群值区间 (17+)	61

3.3 基准方法

在缺陷隔离任务的比较中, 本文综合已有文献研究和相关开源代码的情况, 我们选取了增量调试方法中传统的 ddmin 算法[7]、ProbDD 算法[18]和最新的 C2D2 算法[19], 在相应的数据集上进行缺陷修复关键变更提取和缺陷引入关键变更提取两个任务上进行比较. 本文实验在所有的增量调试算法中集成 DDJ[16]依赖检测机制以确保实验比较的公平性, 并与本文提出的 DISAC 方法结果进行对比.

在消融实验中, 本文选取最新的变更代码子集拆分算法 SmartCommit[28]——一种仅基于变更代码依赖关系的拆解算法, 代替 DISAC 算法中变更代码子集拆分部分, 进行对比实验.

3.4 评价指标

3.4.1 缺陷隔离准确性评价指标

在用于缺陷隔离的增量调试工作中[16−19], 准确性通常使用以下度量指标.

● 成功隔离的条目数量 (#成功): 在规定时间内能够隔离出引入缺陷的变更代码, 并通过还原变更代码, 项目正常运行并测试通过的条目数量. 本文参考已有 DD 研究[18,19], 将 DD 方法的执行时间上限设定为 2 h. 鉴于实验数据集的项目特性以及 DISAC 方法流程设计的高效性, 本文将 DISAC 实验的时间上限设定为 20 min, 在保障实验顺利完成的同时, 也预留了对潜在异常情况的处理空间.

● 约减率平均值 (约减率 (%)): $\frac{|H|-|H^*|}{|H|}$, 其中 H^* 是通过算法隔离到的缺陷变更代码集合, H 是原始变更代码集合, 计算所有成功数据, 并取平均值.

3.4.2 用户验证实验中相关评价指标

我们在验证 DISAC 结果作用和价值的用户实验中采用以下评价指标.

● 平均完成时间 (min): 用户完成缺陷引入根因确定任务所用时间的平均值.

● 平均正确率 (%): 用户在规定时间内正确描述缺陷根因的数量占比的平均值.

4 实验问题与分析

为了评估本文提出方法的效果, 本节依次详细叙述设计的实验问题以及对应实验结果, 并进行分析.

4.1 RQ1 实验结果与分析

● RQ1: 与增量调试方法相比, DISAC 方法在缺陷隔离任务中的性能如何?

针对回归缺陷数据集, 我们通过隔离并回滚引发缺陷的变更代码子集, 再执行测试进行验证, 如果测试通过, 意味着隔离出的变更代码子集中包含了引发缺陷的变更, 即确定了缺陷的位置. 对于 Defects4J 数据集, 我们隔离和回滚可能修复缺陷的变更代码子集, 再执行测试进行验证, 如果测试失败, 意味着该变更代码子集中包含了修复缺陷的变更, 即确定了缺陷的位置. 我们按照 hunk 数量大小分为 4 组, 实验结果分别如表 3、表 4 所示. 表中各列加粗的数值表示各方法中的最佳值.

表 3 在 Defects4J 数据集上增量调试方法与 DISAC 方法的实验结果 (共 809 条数据)

方法	低位区间 (1–4) 共419条		次低区间 (5–8) 共161条		次高区间 (9–12) 共85条		高位区间 (13–16) 共63条		离群值区间 (17+) 共81条		全部数据	
	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)
ddmin	378	25.44	133	50.02	55	50.88	39	60.01	**53**	64.02	658	38.00
ProbDD	378	24.77	133	44.36	55	43.66	39	41.89	**53**	42.57	658	32.91
C2D2	**404**	25.76	**141**	49.89	**64**	51.83	42	59.71	51	64.47	**702**	38.23
DISAC	391	25.03	139	47.66	60	49.24	**45**	55.62	48	64.23	683	35.36
DISAC+C2D2	391	**26.11**	139	**52.32**	60	**52.33**	**45**	**61.65**	48	**69.19**	683	**39.59**

表 4 在回归缺陷数据集上增量调试方法与 DISAC 方法的实验结果 (共 1 025 条数据)

方法	低位区间 (1–46) 共643条		次低区间 (47–92) 共142条		次高区间 (93–138) 共67条		高位区间 (139–176) 共51条		离群值区间 (177+) 共123条		全部数据	
	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)
ddmin	453	41.78	77	47.19	38	26.96	19	37.75	33	27.08	620	40.03
ProbDD	481	35.32	**81**	35.84	40	22.90	23	11.18	39	3.25	664	30.62
C2D2	485	52.07	70	48.98	31	40.55	10	27.91	9	7.98	605	45.06
DISAC	**490**	48.42	79	41.51	**48**	41.83	**29**	38.74	**51**	29.36	**697**	45.84
DISAC+C2D2	**490**	**56.38**	79	**49.45**	**48**	**49.71**	**29**	**44.15**	**51**	**34.68**	**697**	**53.72**

在 Defects4J 数据集上进行缺陷修复约减的实验结果如表 3 所示, 总体就成功数量而言, DISAC 产生了 683 条成功隔离的数据, 比 ddmin 和 ProbDD 多 3.80%, 比 C2D2 少 2.78%. 就约减率平均值而言, DISAC 的结果分别比 ddmin 和 C2D2 低 2.64 和 2.87 个百分点, 比 ProbDD 高 2.45 个百分点. 由于 Defects4J 数据集中数据 hunk 数量较小, 特别是低位区间里的很多数据, 变更代码全集即为隔离到的缺陷, 而其余各个区间的表现差别不大.

在回归缺陷数据集上进行缺陷引入约减的实验结果如表 4 所示, 就成功数量而言, DISAC 产生了 697 条成功隔离的数据, 比 ddmin 多 12.42%, 比 ProbDD 多 4.97%, 比 C2D2 多 15.21%. 就约减率平均值而言, DISAC 的结果比 ddmin 高 5.81 个百分点, 比 ProbDD 高 15.22 个百分点, 比 C2D2 高 0.78 个百分点. 从各个区间的表现来看, 在低位区间和次低区间上, DISAC 方法的成功案例数和 DD 方法差别不大, 约减率比 C2D2 低 3.65%; 而在次高区间和高区间上, DISAC 方法的成功案例数和约减率均显著高于所有 DD 方法.

由此可见, DISAC 算法在 Defects4J 数据集和较小的回归缺陷数据集上的约减效果略差于 DD, 但是在较大规模的回归缺陷数据集上优势显著, 成功隔离数和约减率均有提高. 具体来看, 在成功隔离数上, DISAC 没有正确隔离的原因主要分为两种情况, 一方面是依赖关系构建不准确, 依然有部分依赖关系因没有被检测到而被拆分, 导致查找过程中存在编译失败. 另一方面是模型拆解的不准确, 没有在缺陷变更隔离的二分查找过程中得到导致测试结果从 PASS 转变为 FAIL 的提交 c_{bug}. 在约减率上, DISAC 隔离到的缺陷包含上下文, 是具有某一开发活动的代码块的集合, 因此在部分数据中比 DD 的结果粒度更大.

另外, 实验结果显示, 在 Defects4J 数据集和回归缺陷数据集中, 最快的 DD 算法 C2D2 方法的平均运行时间分别为 682.66 s 和 987.91 s, 而 DISAC 方法的平均运行时间为 93.05 s 和 118.49 s, 分别提升 633.65% 和 733.75%. DD 算法在过程中需要不断隔离特定的代码变更子集, 执行编译和测试操作, 以逐步缩小缺陷范围. 这种反复的编译和测试操作耗费大量时间, 尤其是在处理大型项目或复杂依赖关系时, 时间开销尤为明显, 也会出现很多超时的情况. 相比之下, 本文提出 DISAC 方法通过结合静态分析技术, 利用大语言模型分析变更代码的语义信息以进行拆解, 拆解后利用二分查找的方法进行, 大大减少了执行繁重的编译测试的次数, 能够以显著更短的时间提供准确的预测结果, 在提升缺陷隔离速度的同时显著降低了计算成本.

上述实验证明了 DISAC 在缺陷隔离中的能力, 为了验证缺陷隔离结果的正确性, 本文进一步将实验结果与 Defects4J 和 C2D2 中构建的真实缺陷变更集进行对比, 在所有成功案例中未发现错误案例.

4.2 RQ2 实验结果与分析

● RQ2: DISAC 方法是否能够增强现有增量调试技术的性能?

DISAC 提供了具有语义的缺陷隔离结果, 但为了应对不同场景的需求, 本文设计了 DISAC 与 C2D2 组合的实验, 以评估 DISAC 方法语义隔离的结果是否能够为增量调试算法提供支持. 具体而言, 在 DISAC 将提交约减到单一功能的子提交后, 进一步应用最新的增量调试算法 C2D2, 以实现更细粒度的缺陷隔离, 期望增强现有增量调试技术的性能.

实验结果如表 3 和表 4 最后一行所示, 在 Defects4J 数据集上, DISAC+C2D2 方法约减率比 DISAC 总体提升 4.23 个百分点, 比 C2D2 高 1.36 个百分点; 在回归缺陷数据集上, DISAC+C2D2 方法极大地提高了缺陷隔离任务的约减率, 相比 DISAC 方法提升 7.88 个百分点, 比 C2D2 提升 8.66 个百分点. 同时, 实验结果显示, 在 Defects4J 数据集和回归缺陷数据集中, DISAC+C2D2 方法的平均运行时间分别为 571.42 s 和 758.16 s, 相比 C2D2 分别提升 19.47% 和 30.30%.

在该实验中, DISAC 提供的初步拆分缩小了后续增量调试算法的搜索空间, 显著减少增量调试算法的调试时间和计算开销. 通过先行的 DISAC 拆分步骤, DD 算法可在更小的搜索空间内高效地隔离缺陷, 达到更加精确的结果, 从而在缺陷隔离中表现出较高的整体效率与准确性, 增强了现有增量调试技术的性能.

尽管该组合方法提升了增量调试的性能, 本文仍将 DISAC 设计为可独立使用的模块, DISAC 的结果保留了变更的上下文和语义信息, 更符合实际开发需求; 同时, 该方法也支持根据开发者需求, 与 C2D2 等现有技术灵活集成.

4.3 RQ3 实验结果与分析

● RQ3: 本文提出的 DISAC 方法中不同组成部分如何影响缺陷隔离总体方法的效果?

本文使用 Defects4J 和回归缺陷数据集进行了消融研究. 我们每次独立地禁用了预拆解模块、上下文收集器、评估生成器, 通过消融实验验证算法中各单独部分的有效性及对算法的贡献, 这些版本分别记为 $\text{DISAC}_{\neg\text{preprocess}}$、$\text{DISAC}_{\neg\text{context}}$、$\text{DISAC}_{\neg\text{evaluate}}$. 此外, 我们还分别将代码变更拆解部分替换为仅使用依赖分析的 SmartCommit 算法, 以及仅使用大语言模型进行拆解的方案, 以验证依赖关系建模与大语言模型拆解模块的协同增益效果, 分别记为 DISAC_{SC} 和 $\text{DISAC}_{\text{LLM}}$. 然后我们观察这些消融版本在 Defects4J 和回归缺陷数据集上有效性和效率有何变化, 分别如表 5、表 6 所示.

实验结果表明, 当预拆解模块、上下文收集器或评估生成器被禁用时, DISAC 方法在两个数据集上进行缺陷隔离任务中各指标上的有效性均下降, 其中禁用预拆解模块和评估生成器时下降幅度最大, 这表明这些模块对缺陷隔离任务均具有贡献. 同时, 我们发现 Defects4J 数据集中约减率的下降幅度远小于回归缺陷数据集的下降幅度. 由于回归缺陷数据集的代码变更集平均大小大于 Defects4J 数据集中的代码变更集, 我们推断这些组件对复杂缺陷隔离任务的贡献比对简单缺陷隔离任务的贡献更大.

具体来说, 禁用预拆解模块导致了代码变更中强依赖关系的部分没有被预先合并, 这直接影响了后续的集合拆分过程, 更可能打破变更子集的完整性, 导致生成的子集编译通过率显著下降, 进而影响代码的整体隔离效果,

使得缺陷隔离的准确性降低. 禁用上下文收集器时, 决策模型无法获取完整的代码上下文信息, 导致模型对代码变更的理解缺乏背景支持, 由于上下文信息不足, 某些变更可能被错误合并或拆分, 影响隔离效果. 禁用评估消息生成器时, 缺乏对拆分结果的动态评估和优化过程. 由于提交日志未经过语义分析, 包含多个开发活动的子提交集合无法被进一步拆分或优化, 这导致了生成的子集更为冗余和粗糙, 对缺陷隔离的精确度和子集的粒度都造成不利影响.

表 5 在 Defects4J 数据集上增量调试方法与 DISAC 方法的实验结果 (共 809 条数据)

方法	低位区间 (1–4) 共419条		次低区间 (5–8) 共161条		次高区间 (9–12) 共85条		高位区间 (13–16) 共63条		离群值区间 (17+) 共81条		全部数据	
	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)
$DISAC_{\neg preprocess}$	359	24.19	106	39.52	44	43.22	34	45.16	32	52.87	575	29.88
$DISAC_{\neg context}$	387	24.89	137	47.61	59	49.14	45	55.62	47	64.01	675	35.24
$DISAC_{\neg evaluate}$	367	24.72	119	41.95	45	46.24	37	47.43	31	50.60	599	31.70
$DISAC_{SC}$	389	20.03	117	30.93	40	31.77	29	28.51	30	35.63	605	25.28
$DISAC_{LLM}$	301	21.15	81	33.84	36	39.62	23	41.20	18	47.06	459	26.55

表 6 在回归缺陷数据集上增量调试方法与 DISAC 方法的实验结果 (共 1 025 条数据)

方法	低位区间 (1–46) 共643条		次低区间 (47–92) 共142条		次高区间 (93–138) 共67条		高位区间 (139–176) 共51条		离群值区间 (177+) 共123条		全部数据	
	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)	#成功	约减率 (%)
$DISAC_{\neg preprocess}$	451	37.35	63	38.44	30	27.71	17	22.30	25	14.26	586	33.81
$DISAC_{\neg context}$	485	39.38	76	41.10	43	31.62	24	24.46	38	18.02	666	36.53
$DISAC_{\neg evaluate}$	446	37.15	69	39.14	27	27.20	15	20.11	21	11.41	578	32.65
$DISAC_{SC}$	435	23.25	51	10.13	6	5.12	13	17.01	4	2.57	564	19.02
$DISAC_{LLM}$	384	30.25	48	34.92	19	25.16	7	16.55	6	10.03	464	27.48

当变更拆解部分整体用 SmartCommit 算法替换时, 成功隔离数量在部分区间上差别不大, 但在约减率上有较大下降, 进一步分析发现, SmartCommit 拆解时仅考虑依赖关系, 未能充分考虑变更之间的语义关系, 导致拆解粒度较大, 其生成的子集通常包含更多代码块, 从而更容易通过编译, 但缺陷隔离的精度和拆分子集的粒度会下降, 导致约减率有较大下降, 降低缺陷隔离结果的准确性. 当变更拆解部分只使用大语言模型时, 成功率和约减率上均有所下降, 其中约减率下降幅度相对较小. 这是因为随着变更规模的变大, 大语言模型难以准确建模跨结构或跨模块的依赖关系, 导致相关变更之间的关系被忽略, 影响隔离准确性并降低整体成功率. 但因缺乏依赖关系, 大语言模型拆解的粒度较细, 整体约减率下降幅度较小.

以上消融实验表明, 依赖关系分析与大语言模型在代码拆解中具有互补优势, 其协同使用对提升缺陷隔离的效果具有显著作用.

4.4 RQ4 实验结果与分析

● RQ4: DISAC 方法与增量调试方法在辅助缺陷根因确定任务中的作用和价值如何?

本文设置的用户实验, 旨在验证 DISAC 方法在辅助缺陷根因确定任务上的作用和价值. 实验共招募了 10 名具有 4 年以上 Java 软件开发经验的研究生, 他们被随机分为 A 组和 B 组. 实验数据为 10 个真实缺陷引入提交 (标记为 T1–T10), 覆盖了不同的变更代码块数量区间, 每位参与者需要在已知缺陷隔离结果的基础上, 进一步确定引发缺陷的根本原因并完整描述, 每个任务的完成时间限制为 20 min 内. 为了确保实验公平性, 实验对缺陷引入提交的隔离算法进行了交叉验证, 具体来说, A 组在任务 T1–T5 中使用 C2D2 算法进行缺陷隔离, 而在任务 T6–T10 中使用 DISAC 方法隔离缺陷; B 组则反向安排, 在任务 T1–T5 中使用 DISAC 方法, 在任务 T6–T10 中使

用 C2D2 算法, 两组在实验中使用的方法安排如表 7 所示. 实验过程中记录了参与者的任务完成时间, 使用 C2D2 论文中人工验证的真实缺陷变更集计算定位正确率, 并收集用户体验反馈, 实验结果如表 8 所示.

表 7　实验分组情况

组别	任务T1–T5	任务T6–T10
A组	使用C2D2算法隔离缺陷	使用DISAC方法进行缺陷隔离
B组	使用DISAC方法进行缺陷隔离	使用C2D2算法隔离缺陷

表 8　隔离缺陷引入任务实验结果

组别	任务T1–T5		任务T6–T10	
	平均完成时间 (min)	平均正确率 (%)	平均完成时间 (min)	平均正确率 (%)
A组	16.2	72	**8.7**	**92**
B组	**11.0**	**84**	15.3	80

实验结果显示, 使用 DISAC 算法的结果在任务完成平均时间和定位正确率上均优于 C2D2 算法. 由于 DISAC 方法通过细化的变更拆解和语义理解隔离出具有完整单一活动的代码变更集合, 有助于参与者更快速地聚焦于关键代码改动, 任务完成时间更短. 同时, DISAC 的基于语义的变更隔离方式能够更精确地呈现出引发缺陷的代码范围, 帮助参与者更准确地识别缺陷根本原因. 在对实验参与者的采访中, 大多数参与者表示, DISAC 的隔离结果具有更强的指导性和可操作性, 同时, 很少有不相关代码的干扰. 相比之下, C2D2 在部分数据中给出的结果较为精准, 能够帮助快速确认缺陷, 但是更多结果中包含较多无关代码, 增加了定位根因的难度. 这说明本文提出的方法确保了缺陷的完备性, 隔离出包含有上下文信息的结果, 为开发人员的分析和后续修复提供了有力支持.

5 总　结

本文基于代码变更语义分析提出了一种缺陷隔离方法——DISAC, 该方法结合代码间依赖关系和代码的语义信息, 将复合提交中的代码变更分解为多个具有单一功能的变更集, 再利用线性搜索的算法, 定位引入缺陷的变更子集, 提高了缺陷隔离的准确性和有效性.

与传统方法相比, DISAC 具有显著的优势. 首先, 该方法能够将引入缺陷的代码变更单独隔离, 通过回滚的方法, 能够尽快修复缺陷, 从而确保其他功能模块的正常运行不受影响, 极大地降低了缺陷修复对整体系统稳定性的干扰风险. 其次, DISAC 隔离的变更子集不仅具有细粒度, 还保留了完整的上下文语义信息, 使开发人员能够更全面地理解代码变更的背景和影响, 这种上下文信息的丰富性为进一步分析缺陷根因和制定修复策略提供了坚实的基础. 同时, DISAC 方法能够与传统的 DD 方法结合, 从而使所有基于代码变更的缺陷隔离技术能够受益于此, 提升缺陷隔离过程的准确性和效率, 为处理大规模和复杂依赖关系的代码库提供更为高效的解决方案.

在未来的研究中, 可以进一步优化 DISAC 的方法. 例如, 可以调整现有大语言模型, 进一步提高变更子集拆分和缺陷隔离的预测准确率. 此外, 本文方法暂未处理极端情况下可能出现的大量变更提交问题, 此类提交可能引发复杂的依赖链, 超出当前方法的处理范围. 未来工作将进一步探索如何在大规模代码库和复杂项目环境中增强 DISAC 的适用性和效率, 为更多实际场景中的缺陷隔离和修复提供支持.

References

[1] Munson JC, Khoshgoftaar TM. The detection of fault-prone programs. IEEE Trans. on Software Engineering, 1992, 18(5): 423–433. [doi: 10.1109/32.135775]

[2] Pai GJ, Bechta Dugan J. Empirical analysis of software fault content and fault proneness using Bayesian methods. IEEE Trans. on Software Engineering, 2007, 33(10): 675–686. [doi: 10.1109/TSE.2007.70722]

[3] Wright CS, Zia TA. A quantitative analysis into the economics of correcting software bugs. In: Proc. of the 4th Int'l Conf. on

Computational Intelligence in Security for Information Systems. Torremolinos-Málaga: Springer, 2011. 198−205. [doi: 10.1007/978-3-642-21323-6_25]

[4] Agrawal H, Demillo RA, Spafford EH. Debugging with dynamic slicing and backtracking. Software: Practice and Experience, 1993, 23(6): 589–616. [doi: 10.1002/spe.4380230603]

[5] Runeson P, Andrews A. Detection or isolation of defects? An experimental comparison of unit testing and code inspection. In: Proc. of the 14th Int'l Symp. on Software Reliability Engineering. Denver: IEEE, 2003. 3–13. [doi: 10.1109/ISSRE.2003.1251026]

[6] 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]

[7] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans. on Software Engineering, 2002, 28(2): 183–200. [doi: 10.1109/32.988498]

[8] 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]

[9] Zhang Y, Liu JK, Xia X, Wu MH, Yan H. Research progress on software bug localization technology based on information retrieval. Ruan Jian Xue Bao/Journal of Software, 2020, 31(8): 2432–2452 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6081.htm [doi: 10.13328/j.cnki.jos.006081]

[10] Herzig K, Zeller A. The impact of tangled code changes. In: Proc. of the 10th Working Conf. on Mining Software Repositories. San Francisco: IEEE, 2013. 121−130. [doi: 10.1109/MSR.2013.6624018]

[11] Wang M, Lin ZQ, Zou YZ, Xie B. CoRA: Decomposing and describing tangled code changes for reviewer. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 1050–1061. [doi: 10.1109/ASE.2019.00101]

[12] Fan MD, Zhang W, Zhao HY, Liang GT, Jin Z. Detect hidden dependency to untangle commits. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 179–190. [doi: 10.1145/3691620.3694996]

[13] Zeller A. Yesterday, my program worked. Today, it does not. Why? ACM SIGSOFT Software Engineering Notes, 1999, 24(6): 253–267. [doi: 10.1145/318774.318946]

[14] Yuan Z, Yu LL, Liu C. Bug prediction method for fine-grained source code changes. Ruan Jian Xue Bao/Journal of Software, 2014, 25(11): 2499–2517 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4559.htm [doi: 10.13328/j.cnki.jos.004559]

[15] Misherghi G, Su ZD. HDD: Hierarchical delta debugging. In: Proc. of the 28th Int'l Conf. on Software Engineering. Shanghai: ACM, 2006. 142–151. [doi: 10.1145/1134285.1134307]

[16] Hashimoto M, Mori A, Izumida T. Automated patch extraction via syntax- and semantics-aware delta debugging on source code changes. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena: ACM, 2018. 598–609. [doi: 10.1145/3236024.3236047]

[17] Heo K, Lee W, Pashakhanloo P, Naik M. Effective program debloating via reinforcement learning. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 380–394. [doi: 10.1145/3243734.3243838]

[18] Wang GC, Shen RB, Chen JJ, Xiong YF, Zhang L. Probabilistic delta debugging. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 881–892. [doi: 10.1145/3468264.3468625]

[19] Song XZ, Wu YJ, Liu SN, Chen BH, Lin Y, Peng X. C2D2: Extracting critical changes for real-world bugs with dependency-sensitive delta debugging. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 300–312. [doi: 10.1145/3650212.3652129]

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

[21] Song XZ, Lin Y, Ng SH, Wu YJ, Peng X, Dong JS, Mei H. RegMiner: Towards constructing a large regression dataset from code evolution history. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 314–326. [doi: 10.1145/3533767.3534224]

[22] Barnett M, Bird C, Brunet J, Lahiri SK. Helping developers help themselves: Automatic decomposition of code review changesets. In: Proc. of the 37th IEEE/ACM IEEE Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 134–144. [doi: 10.1109/ICSE.2015.35]

[23] Guo B, Song M. Interactively decomposing composite changes to support code review and regression testing. In: Proc. of the 41st IEEE Annual Computer Software and Applications Conf. Turin: IEEE, 2017. 118–127. [doi: 10.1109/COMPSAC.2017.153]

[24] Muylaert W, de Roover C. [Research Paper] Untangling composite commits using program slicing. In: Proc. of the 18th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. Madrid: IEEE, 2018. 193–202. [doi: 10.1109/SCAM.2018.00030]

[25] Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S. Untangling fine-grained code changes. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. Montreal: IEEE, 2015. 341–350. [doi: 10.1109/SANER.2015.7081844]

[26] Taeumel M, Platz S, Steinert B, Hirschfeld R, Masuhara H. Unravel programming sessions with THRESHER: Identifying coherent and complete sets of fine-granular source code changes. Computer Software, 2017, 34(1): 103–118. [doi: 10.11309/jssst.34.1_103]

[27] Yamashita S, Hayashi S, Saeki M. ChangeBeadsThreader: An interactive environment for tailoring automatically untangled changes. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. London: IEEE, 2020. 657–661. [doi: 10.1109/SANER48275.2020.9054861]

[28] Shen B, Zhang W, Kästner C, Zhao HY, Wei Z, Liang GT, Jin Z. SmartCommit: A graph-based interactive assistant for activity-oriented commits. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 379–390. [doi: 10.1145/3468264.3468551]

[29] Bai JZ, Bai S, Chu YF, *et al*. Qwen technical report. arXiv:2309.16609, 2023.

[30] Min BN, Ross H, Sulem E, Ben Veyseh AP, Nguyen TH, Sainz O, Agirre E, Heintz I, Roth D. Recent advances in natural language processing via large pre-trained language models: A survey. ACM Computing Surveys, 2024, 56(2): 30. [doi: 10.1145/3605943]

[31] He J, Rungta M, Koleczek D, Sekhon A, Wang FX, Hasan S. Does prompt formatting have any impact on LLM performance? arXiv: 2411.10541, 2024.

[32] Kahn AB. Topological sorting of large networks. Communications of the ACM, 1962, 5(11): 558–562. [doi: 10.1145/368996.369025]

[33] 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. Madrid: IEEE, 2021. 686–698. [doi: 10.1109/ICSE43902.2021.00069]

附中文参考文献

[8] 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]

[9] 张芸, 刘佳琨, 夏鑫, 吴明晖, 颜晖. 基于信息检索的软件缺陷定位技术研究进展. 软件学报, 2020, 31(8): 2432–2452. http://www.jos.org.cn/1000-9825/6081.htm [doi: 10.13328/j.cnki.jos.006081]

[14] 原子, 于莉莉, 刘超. 面向细粒度源代码变更的缺陷预测方法. 软件学报, 2014, 25(11): 2499–2517. http://www.jos.org.cn/1000-9825/4559.htm [doi: 10.13328/j.cnki.jos.004559]

作者简介

刘书宁, 硕士生, 主要研究领域为软件智能化开发, 软件缺陷挖掘.

吴毅坚, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为软件演化分析, 软件设计, 软件智能化开发.

宋学志, 博士生, 主要研究领域为软件智能化开发与运维, 软件缺陷分析.

陈碧欢, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为软件供应链, 智能网联汽车, AI 系统工程.

彭鑫, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件智能化开发, 云原生与智能化运维, 泛在计算软件系统, 智能网联汽车基础软件.

赵文耘, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件工程.