软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(10):4705−4723 [doi: 10.13328/j.cnki.jos.006674]

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

面向代码审查的细粒度代码变更溯源方法*

王 敏[1,2], 潘兴禄[1,2], 邹艳珍[1,2], 谢 冰[1,2]

[1](高可信软件技术教育部重点实验室 (北京大学), 北京 100871)

[2](北京大学 计算机学院, 北京 100871)

通信作者: 邹艳珍, E-mail: zouyz@pku.edu.cn

摘 要: 代码审查是现代软件分布式并行开发过程中的重要机制. 在代码评审时, 帮助代码评审者快速查看某一段源代码的演化过程, 可以让评审者快速理解此段代码变更的原因和必要性, 从而有效提升代码评审的效率与质量. 现有工作虽然提供了一些类似的代码提交历史回溯方法及对应工具, 但缺乏从历史数据中进一步提取辅助代码评审相关辅助信息的能力. 为此, 提出一个面向代码评审的细粒度代码变更溯源方法 C2Tracker. 给定一段方法 (函数) 级别的细粒度代码变更, C2Tracker 能够自动追溯到历史开发过程中修改该段代码相关的代码提交, 并在此基础上进一步挖掘其中与该段代码频繁共现修改的代码元素以及相关的变更片段, 辅助代码评审者对当前代码变更的理解与决策. 在 10 个著名开源项目的数据集下进行实验验证. 实验结果表明, C2Tracker 在追溯历史提交的准确率上达到 97%, 在挖掘频繁共现代码元素任务上的准确率达到 95%, 在追溯相关代码变更片段任务上的准确率达到 97%; 相比现有评审方式, C2Tracker 在具体案例的代码评审效率和质量上均有较大提升, 在绝大多数的代码评审案例中被评审者认为能提供“明显帮助”或“很大帮助”.

关键词: 代码评审; 历史溯源; 频繁项集挖掘; 相关代码变更

中图法分类号: TP311

中文引用格式: 王敏, 潘兴禄, 邹艳珍, 谢冰. 面向代码审查的细粒度代码变更溯源方法. 软件学报, 2023, 34(10): 4705–4723. http://www.jos.org.cn/1000-9825/6674.htm

英文引用格式: Wang M, Pan XL, Zou YZ, Xie B. Fine-grained Code Changes Tracking Approach for Code Review. Ruan Jian Xue Bao/Journal of Software, 2023, 34(10): 4705–4723 (in Chinese). http://www.jos.org.cn/1000-9825/6674.htm

Fine-grained Code Changes Tracking Approach for Code Review

WANG Min[1,2], PAN Xing-Lu[1,2], ZOU Yan-Zhen[1,2], XIE Bing[1,2]

[1](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

[2](School of Computer Science, Peking University, Beijing 100871, China)

Abstract: Code review is an important mechanism in the distributed development of modern software. In code review, providing the context information of the current changes can help code reviewers understand the evolution of a certain source code quickly, thereby enhancing the efficiency and quality of code review. Existing studies have provided some commit history tracking methods and corresponding tools, but these methods cannot further extract auxiliary information relevant to code review from historical data. Therefore, this study proposes a novel code change tracking approach for code review named C2Tracker. Given a fine-grained code change at the method (function) level, C2Tracker can automatically track the history commits which are related to the code changes. Furthermore, the frequent co-occurrence changed code elements and relevant code changes are mined to help reviewers understand the current code changes and make decisions. Experimental verification is conducted on ten well-known open-source projects. The results show that the accuracy of C2Tracker in tracking historical commits, mining frequent co-occurrence code elements, and tracking related code change fragments are

* 基金项目: 国家自然科学基金 (61972006)
收稿时间: 2021-09-28; 修改时间: 2022-01-28, 2022-03-05; 采用时间: 2022-03-11; jos 在线出版时间: 2023-04-04
CNKI 网络首发时间: 2023-04-06

97%, 95%, and 97%, respectively. Compared with existing review methods, C2Tracker greatly improves its code review efficiency and quality in specific cases. Additionally, reviewers acknowledge that it can play a significant role in helping improve the efficiency and quality of most review cases.

Key words: code review; history tracking; frequent items mining; related code change

现代软件开发中, 代码处于不断变更、持续演化的状态. 代码审查是开发过程中提高代码质量的重要手段和途径[1]. 在代码评审过程中, 评审者常常需要同时面对数十个待评审的代码提交 (commit), 且一个代码提交中可能包含数量众多的细粒度代码变更. 因此, 如何有效利用代码演化历史信息辅助评审者进行快速高效的代码评审成为至关重要的问题.

版本控制系统中积累的大量代码演化历史数据蕴含有丰富的语义信息, 包括软件演化过程中的代码提交记录以及对应的自然语言描述信息等[2]. 同时, 版本控制系统提供了追溯历史提交的方法及对应工具, 例如, 以 Git 为内核的版本控制系统提供了查看提交历史的命令接口, 利用 git log 及对应的参数配置命令, 可以快速查看历史的代码提交记录及对应的修改信息; 利用 git blame 及对应的参数配置命令, 可以快速查看某文件的代码修改历史. 然而, Git 的历史回溯工具在追溯历史提交上, 过于粗粒度, 无法精准回溯到细粒度的历史提交. 研究界提出了更加精准的历史回溯工具, 例如, Higo 等人[3]提出了追溯方法级别的代码历史提交的工具 FinerGit, 以及 Grund 等人[4]提出了更加完善精准的方法级别的代码历史追溯工具 CodeShovel, 该方法在回溯方法级别代码片段的历史提交任务上, 取得了目前最佳效果.

然而, 现有工作仅是从代码演化历史中回溯出与当前代码变更相关的代码提交, 而缺乏从历史数据中进一步提取辅助代码评审的代码演化相关信息的能力, 这使得在代码评审的场景下, 现有工具提供的辅助效果非常有限. 譬如, 在代码评审时, 评审者常需要快速了解一段代码变更的必要性和正确性, 这不仅需要回溯这段代码变更的演化历史, 还需要进一步挖掘代码提交历史中相似或相关变更信息, 从而通过类比信息查看和比较, 有效提升代码评审的效率与质量. 类似的, 在代码评审的场景下, 如果评审者能够关注到频繁共现修改的代码上下文信息, 则有助于评审者对当前变更的决策进行影响判断, 从而预防潜在的代码缺陷或代码坏味的发生[5].

为了解决上述问题, 本文提出了一种面向代码评审的细粒度代码变更的溯源方法及工具 C2Tracker. 对于给定的一段细粒度代码变更, 该方法能够自动地从软件演化历史中抽取出修改过该段代码变更的历史代码提交. 进一步地, 该方法利用基于 FPGrowth 的频繁项集挖掘算法挖掘出历史提交中与该段代码变更频繁共现修改的代码元素, 并且利用基于树的代码变更表示方法追溯出历史提交中与该段代码变更最为相关的代码变更信息, 最后将这些信息返回给评审者, 从而提升代码评审的效率与决策质量.

为了验证本文方法的有效性和实际应用有效性, 本文在 10 个著名的开源项目的数据集进行了实验验证. 实验结果表明, 本文方法在回溯历史提交任务上的准确率达到 97%, 在挖掘频繁共现修改的代码元素任务上的准确率达到 95%, 在追溯相关代码变更信息任务上的准确率达到 97%; 更进一步地, 本文通过人工模拟评审的对比实验, 实验结果表明, C2Tracker 在具体案例的代码评审效率和质量上均有较大提升, 在绝大多数的案例中被评审者认为能提供“明显帮助”或“很大帮助”.

总体来说, 本文的主要贡献包括以下 3 方面.

(1) 提出一种从软件历史提交中自动挖掘频繁共现修改代码元素的方法. 该方法在回溯细粒度代码提交历史的基础上, 基于 FPGrowth 的频繁项集挖掘算法进一步地挖掘共现修改代码信息, 辅助评审者对当前变更的决策进行影响判断.

(2) 提出一种从软件历史提交中自动追溯相关代码变更的方法. 该方法在回溯细粒度代码提交历史的基础上, 进一步地利用基于树结构的代码变更表示追溯出最为相关的代码变更信息, 辅助评审者对当前代码变更进行理解和决策.

(3) 在 10 个开源项目的公开数据集上验证本文方法的有效性. 采用人工分析的方式, 在具体案例上实验分析方法抽取的历史演化信息在代码评审任务上的应用有效性.

本文第 1 节介绍本文的动机和主要挑战. 第 2 节具体介绍本文所提出的细粒度的历史变更溯源方法. 第 3 节

进行实验验证并讨论. 第 4 节介绍相关工作. 第 5 节总结全文.

1 动机与挑战

本节以著名的开源项目 Checkstyle 为例, 阐述从历史提交中回溯细粒度代码评审辅助信息的动机和挑战.

软件演化历史的代码提交中蕴含的一种值得评审关注的信息是频繁共现修改的代码元素信息. 例如, 在项目 Checkstyle 中的 Checker.java 文件中, 其 process 方法在演化历史中发生多次修改, 涉及 40 次的代码提交. 在这些代码提交中, TreeWalker.java 文件中的部分代码元素与 process 方法共同发生变更的次数超过 20 次. 此外, PropertyCacheFile.java 文件与 DetailAST.java 文件的部分代码元素同样与该方法频繁共现修改. 这个实例表明, 如果 Checker.java 文件中的 process 方法发生变更时, TreeWalker.java、PropertyCacheFile.java 以及 DetailAST.java 的代码元素也大概率会发生修改. 在代码评审过程中, 如果能将这样的频繁共现修改的代码元素信息反馈给评审者, 那么评审者就可以提前关注到修改 Checker.java 文件中的 process 方法带来的关联影响, 从而预防可能产生的程序缺陷或代码坏味.

不同于影响集分析[6]中的代码共现元素, 本文所指的频繁共现修改代码元素是指: 在同一个方法级别代码的历史提交中, 与该方法频繁共同修改的其他具备语义信息的代码元素, 如类、方法以及变量等. 即我们关注历史数据中蕴含的频繁共现性, 而不仅代码本身的程序依赖性. 一种发现这种共现修改代码元素的思路是采取静态程序分析的技术建立修改代码元素的数据流关系. 然而, 基于程序分析的这种方式可能会造成代码变更元素关联到大量的细粒度的代码元素, 比如, Checker.java 文件中的 process 方法可能与大量的代码元素产生程序依赖, 但这些代码元素不一定会产生共同修改, 即, 采取程序分析的方法可能会造成大量挖掘结果的假阳性 (即并非共现修改的代码元素也会被挖掘出来), 从而造成评审者的评审负担.

综上, 我们提出本文的第 1 个研究动机和挑战: 基于细粒度的代码历史回溯, 从演化历史的代码提交数据中挖掘共同修改的代码元素信息. 其采取基于代码演化历史的频繁共现修改代码元素挖掘的思路, 需要解决的核心挑战是: 如何在回溯出的历史代码提交中提取出具备语义信息的细粒度代码变更元素, 以及如何构建频繁项集进行高效准确的共现修改代码元素挖掘.

软件演化历史的代码提交中蕴含的另外一种值得评审关注的信息是相关的代码变更信息. 例如, 在项目 Checkstyle 的 JavadocMethodCheck.java 文件中, 图 1 展示了 2 次关于 checkThrowsTags 方法的代码提交. 在 2008 年 4 月 22 日的一次提交中, 开发者更改了 HashSet 的实例化方式, 而在 2016 年 9 月 21 日的一次提交中, 另一名开发者又变更了该 HashSet 的实例化方式. 在代码评审的场景下, 如果评审者在评审 2016 年的这次代码提交时, 能够了解历史上 2008 年的相关代码变更, 那么对于当前评审则会获得更多的参考信息, 从而提升了代码评审的效率和质量.

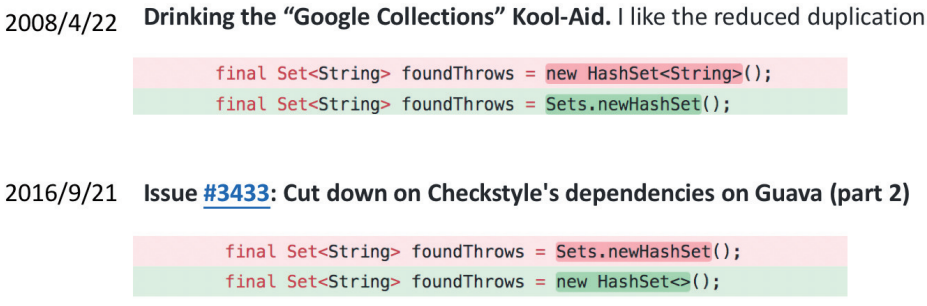


图 1 checkThrowsTags 方法中的两次细粒度变更实例

不同于相似代码变更, 本文所定义的相关代码变更是指: 在同一个方法 (函数) 的相关历史提交中, 开发者围绕同一目的而进行的具备紧密关联的代码变更操作, 即我们关注的是代码变更在语义上的相关性, 而不仅是代码变更内容上的相似性. 图 1 展示了历史代码提交中出现的一对相关的代码变更实例, 该示例中, 开发者同样是围绕 HashSet 的实例化方式进行的代码变更操作, 我们称历史提交中这样的代码变更为相关代码变更.

综上, 我们提出本文的第 2 个研究动机和挑战: 基于细粒度的代码历史回溯, 追溯历史提交中的相关代码变更信息辅助评审者进行当前代码变更的理解. 区别于方法/文件级别的代码追溯, 方法体内的细粒度代码变更更加细琐繁杂, 并且一般不存在明显的文本相似特征. 例如: 变量名更改, for 循环和 foreach 循环转化, 或者方法体内的代码行重构等; 因此本文方法需要关注细粒度代码变更在代码语法结构上的相似性特征, 泛化文本上的相似; 需要关注细粒度代码变更的上下文的相关度特征, 即语法树上父节点的相关度. 其需要解决的核心挑战是: 如何对细粒度的代码变更进行表示以获取代码元素中的各类特征信息, 以及如何设置启发式规则判定两段代码变更之间的语义相关性.

2 细粒度的代码变更溯源方法

本文研究提出了一种面向代码审查的细粒度代码变更溯源方法 C2Tracker. 如图 2 所示, 该方法的输入为一次代码提交中的一段代码变更 (本文定义为方法级别的代码变更) 与代码仓库, 输出为与该段代码变更共现修改的代码元素和历史上与该段代码变更最为相关的代码变更. C2Tracker 可分为历史代码提交提取、细粒度代码变更表示、频繁共现元素挖掘以及相关代码变更追溯 4 个主要部分.

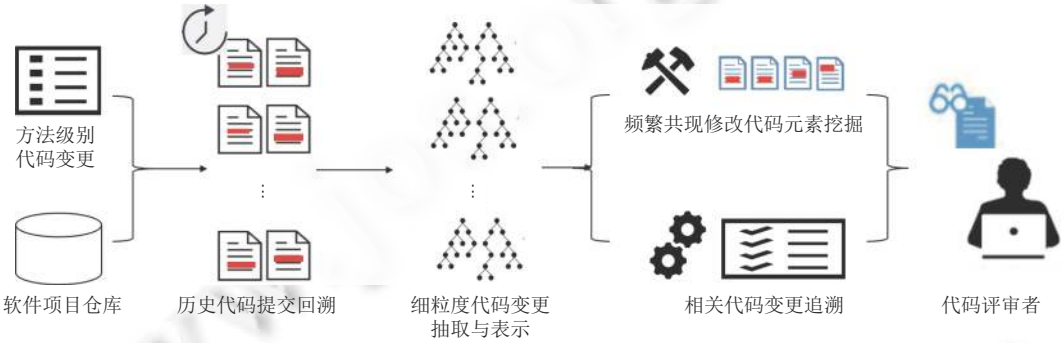


图 2 C2Tracker 的方法流程概览图

● 历史代码提交抽取: 以版本控制系统上主分支的代码提交记录作为数据源, 给定一段方法级别的代码变更 (包含方法体内细粒度的语句级别变更), 抽取出历史记录中修改过该方法的代码提交, 得到关于该方法历史上的代码提交集合.

● 细粒度代码变更表示: 对上述步骤得到的代码提交集合中的每一次代码提交, 分别对前后版本代码进行抽象语法树解析, 进行细粒度的代码变更抽取, 得到每次代码提交中的代码变更元素集合.

● 频繁共现元素挖掘: 基于上述步骤构建好的代码变更元素集合, 以代码元素作为频繁项集中的频繁项构建数据集, 利用频繁项集挖掘算法进行频繁共现修改的代码元素挖掘, 得到频繁共现代码元素集合.

● 相关代码变更追溯: 基于树形结构的代码变更表示, 进一步追溯历史代码提交集合中与当前代码变更最为相关的代码变更片段, 得到相关代码变更集合.

2.1 历史代码提交抽取

现有研究工作提出了很多针对方法级别的历史提交回溯方法并进行了工具开源, 本文采用当前最佳工具 CodeShovel (https://github.com/ataraxie/codeshovel)[4]进行历史代码提交抽取. 该工具输入为一个起始代码提交, 一个指定文件下的特定方法, 这里用符号 m 进行形式化表示, 输出为历史上修改过 m 方法的代码提交记录, 形式化定义为 $H(m)=\{C_1,C_2,\ldots,C_n\}$. 其中, $H(m)$ 表示历史代码提交集合, C_i 表示一次修改过 m 方法的代码提交.

2.2 细粒度代码变更表示

对上述步骤得到的历史代码提交集合 $H(m)$ 中的每一次代码提交 C_i, 分析其前后版本的代码, 抽取出细粒度的代码变更. 现有研究工作提出了很多细粒度的代码差异分析方法及对应开源工具. 本文采用 GumTree 工具

(https://github.com/GumTreeDiff/gumtree)[7]进行细粒度代码的抽取与表示, 该工具输入一次代码提交, 通过抽取前后版本代码的抽象语法树并进行树节点的映射, 对代码差异进行细粒度的分析与表示, 最后输出细粒度的代码差异.

本文保留 GumTree 工具解析代码差异之后的两类信息, 一种是代码语法类型信息, 另一种是代码差异内容信息. 形式上, 输入每一次代码提交 C_i, 得到细粒度的代码变更元素集合 $D=\{e_1,e_2,\ldots,e_n\}$. 其中, D 表示代码变更的元素集合, e_i 表示具体的一处变更的代码元素, 包括该元素的语法类型信息和变更内容信息等.

2.3 频繁共现元素挖掘

本文基于上述步骤得到的代码差异元素集合进行历史提交中频繁共现修改代码元素的挖掘. 其具体策略是: 给定 m 方法的历史提交 $H(m)=\{C_1,C_2,\ldots,C_n\}$, 通过细粒度的代码变更表示得到历史上 n 次代码提交的代码差异元素集合 $HD=\{D_1,D_2,\ldots,D_n\}$. 其中, $D_i=\{e_1,e_2,\ldots,e_n\}$. 最后通过 FPGrowth 的频繁项集挖掘算法[8]进行频繁共现代码元素挖掘.

具体地, FPGrowth 算法分为两个阶段: 第 1 阶段是构建 FP 树; 第 2 阶段是从 FP 树中挖掘频繁项集. 图 3 展示了一个简单的数据集示例及其对应的构建好的 FP-Tree 结构示意图.

ID	元素项集	排序后的频繁项集
001	b, c, k, p, s	c, b, p
002	b, w, f, o, j,	f, b
003	a, c, d, g, i, m, p	f, c, a, m, p
004	a, c, f, e, l, p, m, n	f, c, a, m, p
005	b, f, l, o, m, c, a	f, c, a, b, m

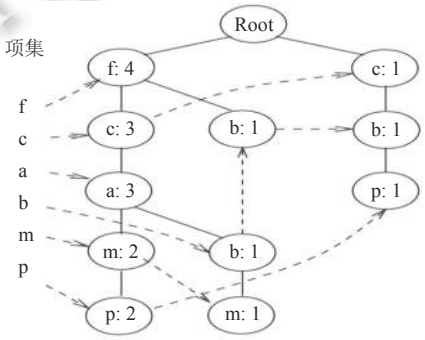


图 3 一个简单的数据集示例及其对应的 FP-Tree 树结构

算法 1 展示了构建 FP 树的主要流程. 其中 FP-Tree 是使得该挖掘算法相比 Aprioris 等算法较为高效的关键数据结构, FP-Tree 将数据集中的所有事务 (在本文中指代一次 commit 中共现的细粒度的代码变更元素) 高度压缩成树的路径, 所有的频繁项 (在本文中指代频繁共现代码变更元素) 都成为树的一个节点, 每个节点都拥有相应的计数, 代表该频繁项在数据集中出现的次数, 其中叶子节点的计数等于前向遍历路径中的 *FIs* (frequent items) 出现在数据集中的次数. 第 2 阶段在第 1 阶段构建好的 FP-Tree 中挖掘频繁项集, 算法 2 展示了 FPGrowth 算法的主要流程.

基于上述算法, 本文从构造好的 HD 数据集中挖掘出历史提交中关于 m 方法的频繁共现修改的代码元素, 形式化表示为 $FIs=\{FI_1,FI_2,\ldots,FI_n\}$. 其中, FI_i 表示频繁共现修改的代码元素.

算法 1. 构建 FP-Tree.

输入: 历史 commit 代码差异元素数据集 *HD*, 最小支持度阈值 min_sup;
输出: FP-Tree.

1. 扫描数据集 *HD* 一次, 获得频繁项的集合 *F* 和其中每个频繁项的支持度. 对 *F* 中的所有频繁项按其支持度进行降序排序, 结果为频繁项表 *L*;
2. 创建一个 FP-Tree 的根节点 *T*, 标记为“null”;
3. **for** *HD* 中每个 *D* **do**
4. 对 *D* 中的所有频繁项按照 *L* 中的次序排序;
5. 对排序后的频繁项表以 [*e*|*E*] 格式表示, 其中 *e* 是第 1 个元素, 而 *E* 是频繁项表中除去 *e* 后剩余元素组成的项表;

6. 调用函数 insert_tree([*e*|*E*], *T*);
7. **end for**

insert_tree([*e*|*E*], root)

1. **if** root 存在孩子节点 *N* and *N*.item-name=e.item-name **then**
2. *N*.count++;
3. **else**
4. 创建新节点 *N*;
5. *N*.item-name=*e*.item-name;
6. *N*.count++;
7. *e*.parent=root;
8. 将 *N*.node-link 指向树中与它同项目名的节点;
9. **end if**
10. **if** *E* 非空 **then**
11. 把 *E* 的第 1 项目赋值给 *e*, 并把它从 *E* 中删除;
12. 递归调用 insert_tree([*e*|*E*], *N*);
13. **end if**

算法 2. FP-Growth 算法流程.

输入: 已经构造好的 FP-Tree, 项集 α (初值为空), 最小支持度 *min_sup*;
输出: 数据集 *HD* 中的频繁项集 *L*.

1. *L* 初值为空
2. **if** Tree 只包含单个路径 *P* **then**
3. **for** 路径 *P* 中节点的每个组合 (记为 β) **do**
4. 产生项目集 $\alpha \cup \beta$, 其支持度 support 等于 β 中节点的最小支持度数;
5. **return** $L=L \cup$ 支持度数大 min_sup 的项目集 $\beta \cup \alpha$
6. **else**
7. **for** Tree 的头表中的每个频繁项 αf **do**
8. 产生一个项目集 $\beta=\alpha f \cup \alpha$, 其支持度等于 αf 的支持度;
9. 构造 β 的条件模式基 B, 并根据该条件模式基 B 构造 β 的条件 FP- 树 *Tree*β;
10. **if** *Tree*$\beta \neq \varnothing$ **then**
11. 递归调用 FP-Growth(*Tree*β, β);
12. **end if**
13. **end for**
14. **end if**

2.4 相关代码变更追溯

本文在上述步骤得到的历史提交集合 $H(m)=\{C_1, C_2, \ldots, C_n\}$ 中追溯最为相关的代码变更片段. 其基本策略是: 对集合 H 中的每一次代码提交中的 m 方法进行细粒度代码变更表示, 计算每一次代码提交 C_i 与其他代码提交中 m 方法代码变更的相关度, 从而得到历史上不同代码提交中关于 m 方法当前变更的相关代码变更. 主要分为代码变更细粒度表示与细粒度代码变更相关度计算两部分.

具体地, 我们首先利用第 2 步细粒度代码变更表示中的 GumTree 工具得到 m 方法中的细粒度的代码变更内

容, 其中包含: 代码变更的操作类型信息, 即插入 (INS)、删除 (DEL)、移动 (MOV) 或更新 (UPD); 代码变更节点的语法类型信息; 代码变更的内容信息, 即代码变更的具体文本差异. 在得到这些细粒度的代码差异信息之后, 我们对代码变更语句进行向量化转化.

插入 (INS)、删除 (DEL) 以及移动 (MOV) 操作导致的代码语句变更的具体语法类型及变更语句示例见表 1.

表 1 INS、DEL、MOV 操作类型的代码变更语句表示及示例

序号	语法类型	表示向量	变更语句示例	示例语句特征值
1	AssertStatement	[SType, OP, Parent, Expresion]	+ assert(m.size() == 1);	[AssertStatement, INS, MethodDeclaration, “m.size() == 1”]
2	Assignment	[SType, OP, Parent, Operator, VarName, Expression]	+ level = new Level();	[Assignment, INS, MethodInvocation, “+”, “level”, “new Leval()”]
3	BreakStatement	[SType, OP, Parent, Identifier]	+ break;	[BreakStatement , INS, MethodInvocation, “”]
4	CatchClause	[SType, OP, Parent, ExceptionType]	+ catch (IOException e)	[CatchClause, INS, MethodInvocation, “IOException”]
5	ContinueStatement	[SType, OP, Parent]	+ continue;	[ContinueStatement, INS, MethodDeclaration]
6	DoStatement	[SType, OP, Parent, Condition]	+ do {} while(condition);	[DoStatement, INS, MethodDeclaration, “condition”]
7	EnhacedForStatement	[SType, OP, Parent, VarType, Collector]	+ for (Item item: items)	[EnhacedForStatement, INS, MethodDeclaration, “Item”, “items”]
8	ForStatement	[SType, OP, Parent, Condition]	+ for (int index = 0; index < size; index ++)	[ForStatement, INS, MethodDeclaration, “index<size”]
9	IfStatement	[SType, OP, Parent, Condition]	+ if (condition)	[IfStatement, INS, MethodDeclaration, “condition”]
10	LabelStatement	[SType, OP, Parent, Identifier]	+ label:	[LabelStatement, INS, MethodDeclaration, “label”]
11	MethodInvocation	[SType, OP, Parent, VarType, Method, ParamCount]	+ id.getValue();	[LabelStatement, INS, MethodDeclaration, “id”, “getValue”, 0]
12	NewClassInstance	[SType, OP, Parent, Type]	+ new Initializer();	[NewClassInstance, INS, MethodDeclaration, “Initializer”]
13	ReturnStatement	[SType, OP, Parent, ReturnType, ReturnValue]	+ return 1;	[ReturnStatement, INS, MethodDeclaration, “Integer”, “1”]
14	SwithCase	[SType, OP, Parent, Expression]	+ case CASE:	[SwithCase, INS, MethodDeclaration, “CASE”]
15	SwithStatement	[SType, OP, Parent, Expression]	+ switch (cases)	[SwithStatement, INS, MethodDeclaration, “cases”]
16	SynchronizeStatement	[SType, OP, Parent, VarName]	+ synchronized (lock)	[SynchronizeStatement, INS, MethodDeclaration, “lock”]
17	ThrowStatement	[SType, OP, Parent, ExceptionType]	+ throw new IOException();	[ThrowStatement, INS, MethodDeclaration, “IOException”]
18	TryStatement	[SType, OP, Parent, Resources]	+ try { }	[TryStatement, INS, MethodDeclaration, “”]
19	PrefixExpression	[SType, OP, Parent, Operator, VarName]	+ – –count;	[PrefixExpression, INS, MethodDeclaration, “– –”, “count”]
20	PostfixExpression	[SType, OP, Parent, Operator, VarName]	+ count– –	[PostfixExpression, INS, MethodDeclaration, “– –”, “count”]
21	VariableDeclaration	[SType, OP, Parent, VarType, VarName, Initializer]	+ String str = “S2Miner”;	[VariableDeclaration, INS, MethodDeclaration, “String”, “Str”, “S2Miner”]
22	WhileStatement	[SType, OP, Parent, Condition]	+ while (condition)	[WhileStatement, INS, MethodDeclaration, “condition”]

表 1 中, 表示向量的特征维度为:

(1) 语法类型 (SType): 发生变更代码语句的语法类型.

(2) 操作类型 (OP): INS、DEL 和 MOV 中的一种.

(3) 父节点语法类型 (Parent): 距离变更语句 (对应一个操作节点) 最近的祖先 Block 节点的父节点, 此信息来保留代码变更的上下文信息.

(4) 变更语句特定内容信息: 即不同类型的语句具体变更的代码差异内容.

不同于上述 3 种类型的操作变化, UPD 语句的表示注重刻画语句的更新方式, 即关注于前后版本的内容变化. 基于 GumTree 的抽取结果, 我们对 UPD 操作的信息进行抽取、表示, 并同样表示为向量的形式: [OP, Node, Parent, Position, Literal]. 其中, OP 表示操作类型; Node 表示操作节点的语法类型; Parent 表示更新节点的父节点语法类型; Position 表示节点在父节点上的位置信息; Literal 表示代码节点对应的内容文本.

经过上述阶段, m 方法里的代码变更被表示为一组有序的细粒度的变更语句, 每个变更语句按照上述特征向量进行对应表示. 在衡量代码变更间的相关度时, 我们首先确定代码变更间匹配的变更语句, 然后计算变更语句序列的相似度, 并以此衡量代码变更间的相关度. 具体过程如下.

首先, 我们提出不同操作类型的变更语句的匹配条件. 如公式 (1) 所示, S_1、S_2 分别代表待比较的变更语句. 当语法类型、变更类型和父节点语法类型相同时, 方法才会继续判断其他语法特征是否相同; 否则两者被判定为不同, 并终止后续信息的比较. 在满足上述条件后, 语句将按照表 2 中的匹配条件进行后续信息比较.

$$S_1.\text{SType} = S_2.\text{SType} \wedge S_1.\text{OP} = S_2.\text{OP} \wedge S_1.\text{Parent} = S_2.\text{Parent} \tag{1}$$

表 2 INS、DEL 及 MOV 语句中的匹配条件

序号	语法类型	匹配条件
1	AssertStatement	$Expression_1 = Expression_2$
2	Assignment	$Operator_1 = Operator_2 \wedge (VarName_1 = VarName_2 \vee Expression_1 = Expression_2)$
3	BreakStatement	$Identifier_1 = Identifier_2$
4	CatchClause	$ExceptionType_1 = ExceptionType_2$
5	ContinueStatement	—
6	DoStatement	$Condition_1 = Condition_2$
7	EnhacedForStatement	$VarType_1 = VarType_2 \wedge Collector_1 = Collector_2$
8	ForStatement	$Condition_1 = Condition_2$
9	IfStatement	$Condition_1 = Condition_2$
10	LabelStatement	$Identifier_1 = Identifier_2$
11	MethodInvocation	$VarType_1 = VarType_2 \wedge Name_1 = Name_2 \wedge ParamCount_1 = ParamCount_2$
12	NewClassInstance	$Type_1 = Type_2$
13	ReturnStatement	$ReturnType_1 = ReturnType_2 \vee ReturnValue_1 = ReturnValue_2$
14	SwithCase	$Expression_1 = Expression_2$
15	SwithStatement	$Expression_1 = Expression_2$
16	SynchronizeStatement	$Expression_1 = Expression_2$
17	ThrowStatement	$ExceptionType_1 = ExceptionType_2$
18	TryStatement	$Resources_1 = Resources_2$
19	PrefixExpression	$Operator_1 = Operator_2 \wedge VarName_1 = VarName_2$
20	PostfixExpression	$Operator_1 = Operator_2 \wedge VarName_1 = VarName_2$
21	VariableDeclaration	$VarType_1 = VarType_2 \wedge (VarName_1 = VarName_2 \vee Initializer_1 = Initializer_1)$
22	WhileStatement	$Condition_1 = Condition_2$

给定 UPD 语句 $S_1 = [e_{1,1}, e_{1,2}, \ldots, e_{1,|s_1|}]$ 和 $S_2 = [e_{2,1}, e_{2,2}, \ldots, e_{2,|s_2|}]$, 其中 $e_{1,i}$ 和 $e_{2,j}$ 均为 UPD 语句中的变更操作. 当 $e_{1,i}$ 和 $e_{2,j}$ 的 5 个维度的特征值完全一致, 表示为相同的操作. 我们利用 Myers 算法[9] (一种计算文本行差异算

法), 获取S_1与S_2之间最长公共操作序列, 并记为 CE. 基于此, 定义S_1与S_2的匹配条件如公式 (2) 所示. 该公式所代表的含义是, 当S_1与S_2间相同的变更操作的占比超过阈值γ时, S_1与S_2为匹配的 UPD 语句. 本文通过对比实验, 设置γ取值为 0.6, 达到最佳实验效果.

$$\frac{|CE|}{|S_1|}\geqslant\gamma\wedge\frac{|CE|}{|S_2|}\geqslant\gamma \tag{2}$$

接下来计算两段代码变更的相关度. 给定代码变更$C_1=[S_{1,1},S_{1,2},\ldots,S_{1,|C_1|}]$和$C_2=[S_{2,1},S_{2,2},\ldots,S_{2,|C_2|}]$. 其中, $S_{1,i}$和$S_{2,j}$分别为代码变更C_1和C_2中的变更语句. 在获得上述步骤匹配好的匹配语句后, 考虑到 m 方法内部的代码变更语句顺序, 我们利用 Myers 算法[9]获取C_1与C_2间的最长公共语句序列, 并记为 SS. 最终衡量两段代码变更相关度的计算如公式 (3) 所示. 该公式的含义是: 对于两次提交中 m 方法的细粒度代码变更, $|SS|$相似的匹配语句数量与两次变更中最大修改语句数量的比值. 我们以这种公共修改语句重叠率的方式衡量两次代码变更的相关度, 最后得到相关的代码变更. 本文通过对比实验设定相关度大于 0.8 时, 则认为是最为相关的代码变更, 达到最佳实验效果.

$$Rel(C_1,C_2)=\frac{|SS|}{\max(|C_1|,|C_2|)} \tag{3}$$

3 实验分析

基于上述方法, 本文设计并实现了基于代码演化历史的细粒度代码变更溯源工具 C2Tracker. 相比于当前最新的细粒度代码变更回溯方法 CodeShovel[4], 其不仅实现了历史代码提交回溯, 还实现了频繁共现修改代码元素挖掘和相关代码变更片段的溯源. 为了验证本文具体工作的有效性和实际应用有效性, 本文在 10 个开源项目的数据集下进行了实验验证.

3.1 研究问题

在下述实验中, 我们将重点分析以下两方面的研究问题.

RQ1: 相比于现有代码回溯方法 CodeShovel, 本文提出的 C2Tracker 在挖掘细粒度代码变更的历史演化信息任务上的效果如何? 主要验证本文方法在历史代码提交回溯、频繁共现修改代码元素挖掘以及相关代码变更溯源 3 个子任务上的整体情况及效果.

RQ2: 在代码评审中, C2Tracker 挖掘到的代码提交中的频繁共现修改代码元素与相关变更片段对评审者有多少辅助作用? 具体地, 采用数据分析以及人工模拟评审的方式, 探究频繁共现修改代码元素与相关代码变更片段对代码评审效率和质量的参考作用.

3.2 实验设置

3.2.1 实验数据

基于 CodeShovel 开源的数据集, 本文选取了其中 10 个著名开源项目的相关数据进行实验分析. 表 3 展示了 10 个项目的基本数据情况. 本文在选取项目时考虑选取不同规模、不同领域的著名开源项目. 具体地, 本文在选取项目时, 综合考虑了该项目在 GitHub 上的 Star 数量以及项目本身的 commits 数量. 最终所选出的项目均为开源社区中非常著名的软件项目, 我们认为这些数据在一定程度上具有代表性.

CodeShovel 的研究人员在 10 个项目中选取了 100 个方法, 并人工分析提取了这 100 个方法在各自项目历史上的代码提交, 从而构造了一个方法级别的代码历史回溯数据的标准集. 本文在此标准集的基础上进一步进行了频繁共现代码元素挖掘和相关代码变更追溯, 并邀请 4 位具备 5–8 年 Java 开发经验的研究人员对这些结果进行人工查看和检验. 具体地, 4 位研究人员被分为两组, 分别对上述项目中的 100 个方法的历史演化信息进行人工判定, 由于具备 CodeShovel 标准集的数据基础, 因此大大降低了判定难度, 从而保证人工标注的准确性. 该过程持续 3 周, 花费近 70 个小时. 两组的判定结果基本一致, 未出现明显差异.

在具体实验中, 本文选取上述 10 个项目中的 100 个方法作为研究对象. 具体操作为, 选取最近提交的一次代

码提交作为起始代码提交, 即假设为待评审代码提交, 定位到对应的方法级别的代码变更, 验证 C2Tracker 在历史代码提交回溯、频繁共现修改代码元素挖掘以及相关代码变更溯源上的效果.

表 3 开源项目数据集数据统计概况

项目仓库	Commits数量	Methods数量	Stars数量	选取Methods数	方法相关Commits数
commons-io	2 123	996	488	10	96
elasticsearch	40 353	18 261	33 640	10	95
Hadoop	19 805	32 888	7 801	10	83
hibernate-search	6 172	5 069	283	10	73
intellij-community	226 106	36 387	6 335	10	72
jetty	15 991	11 522	2 139	10	81
lucene-solr	30 500	29 888	1 840	10	105
mockito	4 811	1 366	7 358	10	79
pmd	13 360	2 567	1 738	10	87
spring-boot	17 818	2 451	27 527	10	88

3.2.2 度量标准

本文在衡量 C2Tracker 在回溯历史代码提交的效果上, 采取精准率与召回率的指标. 其中, 精准率与召回率的计算公式分别如下所示. 其中, N_{detect} 表示 C2Tracker 回溯到的历史上的所有代码提交数量, N_{hit} 表示所有回溯到的代码提交中正确的代码提交数量, 即代码提交确实是关于该方法的代码变更. N_{oracle} 表示标准数据集中关于方法的历史代码提交的数量.

$$precision = \frac{N_{\text{hit}}}{N_{\text{detect}}},\ recall = \frac{N_{\text{hit}}}{N_{\text{oracle}}} \tag{4}$$

本文在衡量 C2Tracker 在挖掘频繁共现修改的代码元素的效果上, 采用准确率的指标. 在挖掘频繁共现修改元素的任务上, 我们更关注挖掘结果的真阳性样例, 即保证挖掘结果的精准率, 因此我们在牺牲一定召回率的前提下, 确保返回给评审者的共现信息是准确的, 而并不追求挖掘出来的代码元素的数量, 从而不会造成评审者额外的负担. 其计算公式如公式 (5) 所示. 其中, N_{true} 为 C2Tracker 挖掘到的正确的历史提交中的频繁共现修改代码元素数量, 这里正确的共现代码元素指的是人工判定的确实是频繁共现的有语义的代码元素. N_{total} 表示 C2Tracker 挖掘的共现修改代码元素数量.

$$Accuracy = \frac{N_{\text{true}}}{N_{\text{total}}} \tag{5}$$

同理, 本文在衡量 C2Tracker 在追溯相关代码变更的效果上, 采用准确率的指标. 我们确保推荐给评审者的相关代码变更是准确的, 从而不造成评审者更多的评审负担. 此时, N_{true} 为 C2Tracker 挖掘到的历史提交中的正确的相关代码变更数量, 这里的正确的相关代码变更指代的是人工判定的确实是最为相关的代码变更. N_{total} 表示所有挖掘的相关代码变更的数量.

3.2.3 挖掘结果对比

在衡量 C2Tracker 挖掘出的历史演化信息的有效性时, 我们在 CodeShovel 公开的 10 个项目的数据集上进行了挖掘结果对比.

(1) 在历史代码提交回溯任务上, 本文对 10 个项目的 100 个方法级别的代码进行了历史代码提交的回溯, 并验证了其精准率和召回率, 以提供后续历史演化信息挖掘的数据源基础.

(2) 在频繁共现修改代码元素挖掘任务上, 本文主要验证 C2Tracker 的准确率, 即我们关注 FPGrowth 算法所挖掘出的代码元素是否真正共现, 注意到这里的代码元素指代的是代码提交中具备特定语义信息与结构特征属性的代码实体, 例如, 类实体, 方法实体或变量实体等, 其他类似 public、int 等 Java 保留字等信息不在频繁项集范围之内. FPGrowth 算法的一个重要超参数是最小支持度的阈值设定, 本文启发式地设定该阈值的计算方式如下所示. 这里 N_{commit} 为检测到的代码提交数量, k 为启发式参数.

$$min\_sup = N_{\text{commit}}/k \tag{6}$$

该计算公式的含义是, 频繁项集中的频繁项至少要在历史代码提交的数据集中出现 $1/k$ 以上, 否则不能纳入频繁共现的考虑范围之内. 我们选取 k=2, 3, 5 的情况予以对比分析.

此外, 为了验证挖掘频繁共现修改代码元素的正确性, 我们通过人工标注的方式, 通过判断该共现元素的频繁共现次数, 以及该代码元素是否具备代码语义来标注挖掘代码元素的正确性.

(3) 在相关代码变更的溯源任务上, 我们主要验证 C2Tracker 在该任务上的准确率以及能够追溯到的相关代码变更的数量. 具体实验方法为, 我们选取最近一次的代码提交中的方法级别的代码变更作为源头, 利用 C2Tracker 对其进行相关代码变更的溯源, 最后判断溯源的相关代码变更数量及准确性.

此外, 为了验证追溯相关代码变更的正确性, 我们通过人工分析的方式, 通过判断该段相关代码变更的自然语言描述信息 (例如, commit message 或代码行间注释) 等来确定追溯出的相关代码变更的正确性.

3.2.4 人工评审对比

在衡量 C2Tracker 挖掘出的历史演化信息对代码评审的应用有效性时, 我们首先分析了挖掘出的历史演化信息在 10 个项目的数据集下的数据分布情况, 即探究存在多少方法级别的代码变更评审, C2Tracker 能够挖掘出的频繁共现修改代码元素的数据规模如何, 以及能够追溯出的相关代码变更的数据分布情况. 我们通过这种数据分析的方式证明 C2Tracker 在代码评审场景下追溯历史演化信息的适用性.

在衡量上述历史演化信息对代码评审实际场景的有用性时, 我们采用人工模拟评审代码变更进行定性分析的方式, 探究 C2Tracker 挖掘出的历史演化信息对于代码评审及代码理解上的辅助作用. 具体地, 本文将每个项目中的 10 个方法划分为 A、B 两组. 其中, 审查 A 组的代码提交时, 采用 C2Tracker 的工具进行历史信息辅助, 审查 B 组的代码提交时, 不采用工具辅助. 本文选取了 3 位参与人员全程模拟了代码变更的评审工作, 且每位参与者至少具有 4 年以上的 Java 项目的开发经验. 我们选取 10 个项目中所有参与者都比较熟悉的 4 个项目进行对照实验, 分别是 Hadoop、intellij-community、lucene-solr 以及 spring-boot. 我们采用人工评分 (5-points Likert Scale) 的方式[10]对历史演化信息的应用有效性进行对照评分. 其中, 1 代表没有任何帮助, 甚至出现负面作用; 2 代表没有帮助; 3 代表有轻微辅助作用; 4 代表明显帮助; 5 代表很大帮助.

3.3 RQ1 实验结果

3.3.1 历史代码提交回溯

表 4 展示了 C2Tracker 在面向代码审查场景下的细粒度代码变更的历史演化信息挖掘任务上的总体有效性结果及数据分布情况. 其中, 10 个项目中分别抽取了 10 个方法级别的代码, 总计包含 100 个方法, 涉及的历史上的代码提交数量总计 859 个.

表 4 C2Tracker 的有效性度量结果

项目	历史代码提交回溯				频繁共现修改代码元素挖掘						相关代码变更溯源	
					频繁代码元素数量			正确率				
	方法数	相关代码提交	精准率	召回率	k=2	k=3	k=5	k=2	k=3	k=5	相关代码变更数量	正确率
commons-io	10	96	1.00	1.00	56	81	130	0.91	0.63	0.43	18	1.00
elasticsearch	10	95	0.94	0.93	24	36	64	0.92	0.61	0.38	31	0.94
hadoop	10	83	1.00	1.00	36	79	79	0.92	0.44	0.44	11	1.00
hibernate-search	10	73	1.00	0.98	14	49	70	1.00	0.29	0.18	14	0.93
intellij-community	10	72	0.98	0.83	26	41	41	1.00	0.63	0.63	6	1.00
jetty	10	81	0.99	0.95	31	58	72	1.00	0.53	0.43	18	1.00
lucene-solr	10	105	0.94	0.99	4	47	47	1.00	0.08	0.08	13	1.00
mockito	10	79	0.99	0.96	78	121	180	0.97	0.63	0.43	17	0.92
pmd	10	87	0.94	0.98	20	44	54	0.90	0.46	0.37	9	1.00
spring-boot	10	88	0.99	0.94	13	37	37	1.00	0.35	0.35	6	1.00
总数	100	859	0.97	0.96	302	593	774	0.95	0.51	0.39	143	0.97

如表 4 所示, 历史代码提交回溯一列可视为使用 CodeShovel 的挖掘结果. 可以看到, CodeShovel 仅能够挖掘函数的历史代码提交, 对代码评审者来说不够充分; 而使用本文方法能够更进一步地挖掘出频繁共现代码和相关代码变更, 从而能够帮助代码评审者进一步提高代码评审效率.

在代码历史提交回溯任务上, C2Tracker 在 10 个项目 100 个方法级别所涉及的 859 个历史代码提交回溯上的总体精准率达到 97%, 召回率达到 96%. 其中, C2Tracker 在 commons-io 项目下的精准率和召回率均达到 100%; 在所有项目的历史提交回溯精准率上, C2Tracker 均达到了 94% 以上的效果. 其中, 存在极少数的历史提交识别错误; 在所有项目的历史提交回溯召回率上, C2Tracker 也达到了 93% 以上的效果, 其中, 在 intellij-community 项目上召回率达到最低的 83%, 通过人工分析发现, 这是由于 intellij-community 项目的历史代码提交规模庞大, 且经历了数次大型的项目重构操作, 从而导致回溯历史记录时遗漏了部分代码提交. 总体而言, C2Tracker 在历史代码提交回溯任务上取得了很好的效果, 证明了该方法在代码提交回溯上的可行性.

3.3.2 频繁共现修改代码元素追溯

在频繁共现修改代码元素挖掘任务上, 本文利用第 3.2.3 节提出的启发式计算方法设置了不同的最小支持度阈值进行频繁项集挖掘算法的对比实验. 在 RQ1 中我们主要验证 C2Tracker 在不同最小支持度阈值下挖掘频繁共现修改代码元素任务上的有效性, 即分析频繁元素的数量及正确率. 表 4 结果表明, 当 k=2 时, 即 C2Tracker 挖掘出的代码元素至少在该方法的历史提交中出现一半以上的共现修改时, 在 859 个历史提交的数据集下, 可以挖掘 302 个与对应方法级别代码的共现修改代码元素, 且准确率可以达到 95%. 当 k=3 时, C2Tracker 挖掘出的频繁共现代码元素数量显著提升, 但其中存在大量的假阳性样例, 即并非具备实际语义的非相关代码元素, 因此挖掘的准确率也显著下降; 当 k=5 时, C2Tracker 挖掘出的频繁共现代码元素数量进一步提升, 但准确率急剧下降, 基本上无法达到实际的推荐效果. 为了确保 C2Tracker 推荐给评审者的频繁共现修改的代码元素是具备实际的评审意义以及确实需要关注的频繁共现信息, 我们设置 k=2, 即提升代码元素的频繁性条件约束的力度, 以保证不会给评审者造成额外的评审负担. 注意到表 4 的频繁代码元素数量是针对 10 个方法的总体挖掘数量, 实际过程中, 大量方法并不存在频繁共现修改的代码元素, 我们在 RQ2 中会进行进一步讨论与分析. 总体而言, 当 k=2 时, C2Tracker 在频繁共现修改代码元素挖掘任务上准确率达到 95%, 实验充分证明了该方法在代码评审场景下, 向评审者推荐频繁共现代码元素信息的有效性.

3.3.3 相关变更片段追溯

在追溯相关代码变更的任务上, 本文首先对方法级别代码中细粒度的代码变更语句进行了基于语法树的解析, 抽取了其中的操作类型、语法特征等结构信息, 通过向量化的表示方式进行了语句匹配, 最后根据语句匹配的重叠比例进行代码变更相关度的判断. 在 RQ1 中我们主要验证 C2Tracker 在追溯相关代码变更任务上的有效性, 即分析能够追溯到的相关代码变更的数量及正确率. 表 4 结果表明, C2Tracker 在 10 个项目的 100 个方法级别的代码中, 能够追溯到相关代码变更数量是 143 个, 准确率达到 97%. 人工抽样分析发现, C2Tracker 的准确率很高的原因主要是: (1) 在第 1 步回溯了历史代码提交的基础上, 追溯相关的代码变更相较于其他相似变更挖掘的工作而言, 搜索范围变小; (2) C2Tracker 在追溯相关代码变更时, 通过基于树的代码变更表示充分挖掘了代码变更的结构化信息, 并通过语句匹配重叠比例的启发式方法判断代码变更相关度, 具备很高的可行性. 注意到 143 个相关代码变更的数量指的是 100 个方法的相关代码变更的总体数量, 存在大量的方法在其历史提交中并不能追溯到任何相关的代码变更, 我们会在 RQ2 中进行进一步讨论与分析. 总体而言, C2Tracker 在追溯相关代码变更的任务上准确率达到 97%, 实验充分证明了, 该方法在代码评审的场景下, 向评审者推荐相关的代码变更信息的有效性.

3.3.4 方法运行效率分析

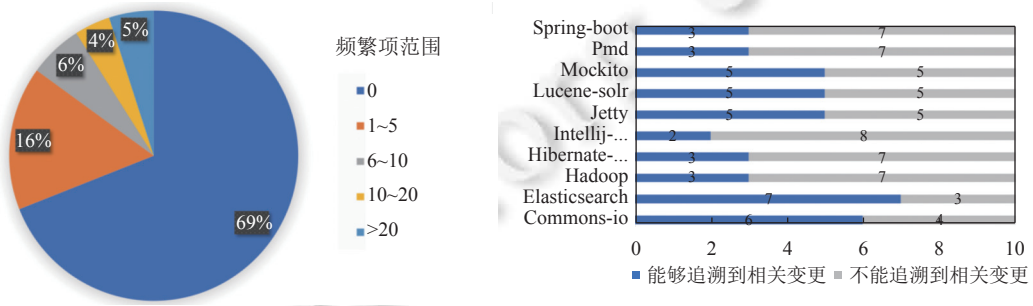
本文选取了 10 个不同规模、不同领域的开源项目的 100 个方法级别的代码作为研究对象进行了实验. 如表 3 所示, 这些开源项目中包含不同数量的代码提交和方法级别代码. 例如, 在开源项目 intellij-communit 就包含 226 106 个 commits 以及 36 387 个 methods, 属于较大规模的软件项目. 本文方法 C2Tracker 在上述开源项目的历史演化信息 (频繁共现元素以及相关代码变更) 挖掘实验中, 平均运行时间为 15 s. 其中, 在较大规模开源项目的追溯过程中, 最长耗时为 47 s. 实验结果表明, C2Tracker 在不同规模的软件项目上, 均可以在评审人员可接受的时间

内, 高效地获取历史演化信息结果.

3.4 RQ2 实验结果

RQ2 主要验证 C2Tracker 在代码评审场景下, 追溯历史演化信息对评审者的辅助作用. 我们首先对 C2Tracker 在 10 个项目所抽取的 100 个方法上的追溯结果的分布情况进行分析.

图 4 展示了 C2Tracker 在 10 个项目 100 个方法级别代码的实验样本下, 追溯历史演化信息的数据分布情况. 由于本文设置了严格的启发式阈值来确保挖掘结果的正确性, 因此从整体的实验结果看, 绝大多数的频繁共现修改代码元素分布在 1–20 范围内. 其中, C2Tracker 能够挖掘到的频繁共现修改代码元素的最小值是 1, 最大值是 32, 中位数是 5, 平均数是 6.7; 相关代码变更片段也集中在 1–3 个范围内. 其中, C2Tracker 能够追溯到的相关代码变更片段的最小值是 1, 最大值是 5, 中位数是 2, 平均数是 2.3. 从上述数据来看, 本文挖掘的历史演化信息的规模并不庞大, 从而不会给代码评审者造成额外的负担.



(a) 100 个样本下频繁代码元素挖掘数量分布情况 (b) 100 个样本下相关代码变更追溯情况

图 4 C2Tracker 在 100 个样本下追溯结果分布情况分析

3.4.1 具体案例分析

在 100 个方法的频繁共现修改代码元素挖掘任务上, C2Tracker 能够为 16 个方法挖掘出 1–5 个频繁共现的代码元素; 为 6 个方法挖掘出 6–10 个频繁共现的代码元素; 为 4 个方法挖掘 10–20 个频繁共现的代码元素; 该实验结果表明, C2Tracker 在近 30% 的方法级别代码的历史提交记录中, 能够高效地挖掘出 1–20 个频繁共现的代码元素. 此外, 人工分析上述挖掘结果, 我们发现, 1–20 个频繁共现代码元素的数量规模符合正常的方法级别代码的变更共现模式. 但注意到存在 5 个方法级别的代码变更的历史提交记录中, C2Tracker 能够挖掘出超过 20 个频繁共现的代码元素, 通过人工分析发现, 这是因为上述 5 个方法在各自项目上都处于活跃修改的核心方法, 在软件演化过程中, 发生过频繁的代码提交. 例如, 在 apache/hadoop 项目的 FifoScheduler.java 文件中, allocate 方法发生过近 40 次的代码变更. 同时, 存在近 70% 的方法, C2Tracker 无法挖掘出任何频繁共现修改的代码元素, 人工分析其原因是: (1) C2Tracker 在挖掘频繁共现元素上设置了很严格的频繁约束条件, 其目的是确保挖掘结果能够正确地反馈给评审者, 不会造成额外的评审负担; (2) 在一个相对成熟的项目中, 方法级别的频繁共现修改并非高频出现, 因为这会破坏项目的“开闭”原则, 即低耦合高内聚的程序设计原则. 总体而言, 频繁共现修改代码元素挖掘能够在大部分的方法级别的代码变更评审中适用.

在相关代码变更的追溯任务上, C2Tracker 在 58 个方法级别的代码变更中, 无法追溯到相关的代码变更, 在 42 个方法级别的代码变更中, 能够追溯到相关的代码变更. 通过分析无法追溯到相关代码变更的方法样本, 我们发现无法追溯到相关变更的原因是: 一些方法级别的代码片段过于简单, 在开发历史中, 并没有出现反复修改, 即每一次的代码变更都是出于不同目的, 独立不相关的. 实验结果表明, 在近 45% 的方法级别的代码变更中, 可以追溯到以往的历史代码提交中的相关代码变更. 人工分析发现, 开发人员通常会对这 42 个方法级别的代码片段围绕同一目的进行反复修改, 因此造成了某些代码片段的相关变更记录在版本控制系统的若干代码提交中, 而在下一次进行代码变更的评审时, C2Tracker 可以追溯到历史版本记录中修改过该段代码的相关代码变更, 以此来获取代

码演化的信息, 以辅助评审者对当前变更的决策. 例如, 在 elastic/elasticsearch 项目的 TransportClusterInfoAction.java 中, commit “7548b2e”和 commit “f4bf0d5”对 masterOperation 方法的代码修改都是围绕 concreteIndices 方法的调用参数进行的相似修改. 在之后的代码评审场景下, 涉及该代码片段的代码提交, C2Tracker 可以追溯到上述两个 commits 的相关代码变更, 为代码评审提供更多的参考信息, 辅助理解当前代码变更. 总体而言, 追溯相关代码变更能够在大多数的方法级别的代码变更评审中适用.

3.4.2 人工评分实验

为了进一步探究本文方法提取出的历史演化信息 (代码频繁共现修改信息以及相似代码变更信息) 对代码审查的辅助作用, 本文采用了人工模拟代码审查的方式进行了定性分析. 为了确保人工评估结果的客观公正, 在模拟代码评审的过程中, 3 名参与人员单独进行评估测试, 并且所有参与人员事先并不知道这些代码变更的最终评审结果. 模拟过程结束后, 我们发现参与者的评审结果和社区原来的评审结果较为一致. 例如, 在评审开源项目 spring-boot 的代码提交“00430ac”时, C2Tracker 辅助评审者回溯到历史代码提交“35a51e4”中的相关代码变更片段, 2 次代码提交同样是围绕该代码文件中 getBindHandler 方法中 BindHandler 类的实例化对象 handler 的初始化方式而进行的代码修改, 当评审者评审后提交的“00430ac”时, 可以快速分析 handler 的代码元素的演化过程和原因, 从而对当前变更作出了合并决策.

我们采取人工评分的方式 (5-points Likert Scale) 进行方法实际应用的有效性验证, 实验结果如图 5 所示. 在 4 个项目的 40 个方法的评审中, 没有任何一位参与者认为 C2Tracker 方法在评审过程中出现负面作用, 充分说明该方法在追溯到的历史演化信息的准确性. P1 参与者认为有 25 个方法级别的代码变更评审中这些信息能够具备辅助作用, 甚至非常有用; P2 参与者与 P3 参与者则分别认为有 29 个和 15 个方法级别的代码变更评审中这些信息能够起作用. 该结果表明, 3 位参与者普遍认为 C2Tracker 追溯出的历史演化信息对于评审能够起到辅助作用. 注意到 3 位参与者仍然认为存在 11–25 个方法的代码变更的评审中, 这些历史演化信息不具备辅助作用, 或作用不明显. 这是因为, 存在部分方法级别的代码变更无法追溯到相关代码变更信息或者频繁共现修改的代码元素信息, 也就不具备任何作用.

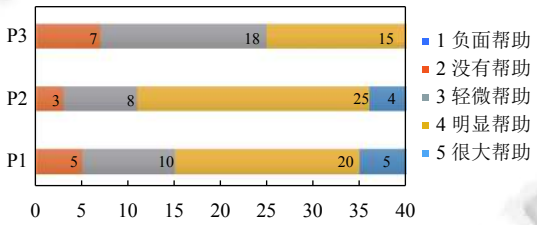


图 5 40 个方法级别的代码变更评审下的人工评分结果

总体而言, 在具备代码频繁共现修改信息的辅助下, 评审者可以更好地理解代码变更的上下文, 从而预防当前代码变更对整体项目带来的影响; 在具备相关代码变更信息的辅助下, 评审者可以参考相关代码变更进行当前代码提交的理解与对比决策, 从而加快对代码变更的评审.

3.5 实验有效性分析

本文实验验证的内部有效性威胁来自 C2Tracker 在方法上的启发式参数设置对实验结果的影响. C2Tracker 在频繁共现修改元素挖掘的任务上, 启发式的确定了频繁项的最小支持度阈值, 该阈值的设定有可能影响频繁元素挖掘的效果, 从而降低评审辅助的作用. 因此, 本文设置了对比实验, 验证了不同最小支持度下的频繁挖掘结果; C2Tracker 在相关代码变更追溯的任务上, 设置了代码变更相关度计算阈值, 本文通过启发式的经验, 得到了最佳的相关度阈值并进行了实验验证, 最大程度上保证了追溯相关代码变更的正确率. 此外, C2Trakcer 的参数设置都可以人工进行设定, 开发者在使用 C2Tracker 时, 面对不同的评审场景, 可以灵活设置以满足不同的历史信息需求, 从而最大程度保证了 C2Tracker 辅助代码评审的有效性.

本文方法在进行实验验证时, 主要的外部有效性威胁来自所选取的实验数据是否具备泛化性, 即在不同项目

下的数据下, 方法的表现性如何. 本文选取了分别从 10 个项目中选取了总计 100 个方法级别的代码作为研究对象. 在所选取的项目规模、领域上不具备任何特殊性, 最大程度上确保了实验数据集的客观性以及真实性, 且本文方法在 10 个项目中的实验效果趋于稳定, 未出现明显差异, 充分证实了本文方法的泛化性. 本文实验的另一个有效性威胁来自人工分析中, 分组差异带来的实验结论的不稳定性, 即不同项目不同方法级别的代码提交的评审难度各异, 因此, 在人工分析阶段, 我们采取了交叉对照的方式, 最大程度上保证了模拟代码评审场景的真实性与客观性.

4 相关工作

本文提出并研究了基于代码演化历史的细粒度代码变更溯源方法, 主要解决了细粒度代码变更在代码评审时缺乏历史相关的辅助信息问题. 本节分别介绍代码历史提交溯源、频繁代码元素挖掘以及代码变更挖掘这 3 方面的相关工作.

4.1 历史代码提交溯源

软件的版本控制系统中积累了大量的历史代码提交数据. 历史代码提交溯源指的是给定一段代码, 追溯软件开发历史过程中修改过该段代码的代码提交记录, 也被称之为历史切片. 以 Git 为内核的版本控制系统中具备 git log、git blame 等查看历史记录的命令接口, 但受限于较粗粒度的追溯以及较低的准确率, 因此, Git 系统中自带的历史追溯工具往往无法满足历史代码提交溯源的效果. Hata 等人[11]提出了一个名为 Historage 的工具, 它可以提供 Java 中细粒度实体的完整历史, 例如方法、构造函数、字段等. Historage 的一个特点是能够回溯实体历史, 包括重命名更改等. Higo 等人[3]在 Historage 的基础上, 进一步地提升了 Java 编程语言中方法实体的历史追溯方法的效率, 并实现了工具 FinerGit. Maruyama 等人[12]提出了一种历史切片机制, 可以从过去的代码更改的整个历史中仅提取构建 Java 程序的特定类成员所需的代码变更. Servant 等人[13]在 2012 年提出了历史切片的概念并提出了历史切片方法 Chronos, 它可以在任意数量的历史代码提交记录中自动识别最小数量的关于特定代码变更的代码提交记录, 用于细粒度的任意源代码段的回溯. Servant 等人[14]在 2017 年进一步地提出了利用模糊理论进行代码行粒度的历史追溯工作. Li 等人[15,16]提出了一种语义层次的代码历史切片方法 CSlicer, 该方法利用程序分析与测试执行分析等技术能够追溯出实现相同语义的代码片段在演化历史上的代码提交. Li 等人[17]2016 年在 CSlicer 的基础上进一步结合 Dynamic Delta Refinement 的技术手段改进了 CSlicer 的历史提交追溯效果. Grund 等人[4]在 2021 年提出了一种追溯方法级别的历史代码提交记录的方法及工具 CodeShovel, 该方法能够回溯 Java 编程语言上方法级别的代码提交记录, 包括文件重命名、方法重命名以及方法移动等特殊情况的回溯处理, 在回溯效果和效率上超过了以往的所有方法.

本文在回溯历史代码提交的任务上, 直接利用了 CodeShovel 的主要算法, 并重构在 C2Tracker 的第 1 步的代码变更历史提交回溯的部分, 在实验部分, 验证了本文方法在历史代码提交回溯任务上的有效性.

4.2 共现代码元素挖掘

版本控制系统中的历史代码提交数据中蕴涵了大量的代码变更数据, 共现修改代码元素挖掘指的是挖掘历史提交中频繁共同出现修改的代码元素, 代码元素一般指代类、方法及变量等具备一定语义与结构的代码实体. Zimmermann 等人[18]在 2004 年提出一种名为 ROSE 的方法, 其思路是利用挖掘版本历史共现元素去指导代码变更, 该方法使用 Apriori 算法[19]去挖掘版本历史中频繁共现修改的代码元素, 取得了一定的效果. Palomba 等人[5]提出一种基于历史数据挖掘的方法去检测代码坏味的方法 HIST, 其中, 该方法在针对 5 种特定的代码坏味的检测时, 采用类似 ROSE 的频繁项集挖掘算法去检测特定共现代码元素的隐含关联关系, 从而发现这些特定代码元素模式下的代码坏味. van Rysselberghe 等人[20]受频繁共现演化的思路启发, 提出了一种基于命名实体的检测方法, 用行匹配和标识符匹配的方式去检测重构操作的共现演化性并进行了验证. Zou 等人[21]提出从任务交互历史中检测“交互的共现耦合” (即, IDE 观察到的工件在任务中何时被使用或修改的记录), 并使用此信息来挖掘共现模式以帮助理解维护活动. D’Ambros 等人[22]提出了一种可视化方法, 该方法集成了不同抽象级别的逻辑耦合共现代

码元素信息. 这有助于对逻辑耦合共现进行深入分析, 同时, 根据逻辑耦合对系统模块进行表征. 所提出的方法支持对软件系统和维护活动 (如重构和重构文档) 的回顾性分析. Nasir Ali 等人[23]提出了一种方法 (CoChaIR), 作为现有基于信息检索技术的需求可追溯性链接恢复方法的补充. CoChaIR 利用文件的历史共现变更信息来提高链接恢复方法的准确性. Mondal 等人[24]提出了一种独特的方法级别代码共现变化模式的实证研究, 该模式有可能查明软件系统中的设计缺陷, 并通过使用对方法级别代码变更共现的关联规则进行合理约束检查来自动识别这种模式. Mondal 等人[25]在 2014 年提出利用频繁项集挖掘等算法挖掘出的代码元素共现关联关系只存在统计上的频繁性, 这种方式很可能造成结果的不准确或忽略那些不频繁但真正相关的共现变更, 因此, 他们提出变更对应性的概念 (change correspondence), 借鉴了在代码库中混合概念位置的想法, 确定对共现变更的实体的更改是否对应, 从而确定它们是否真正相关. Lozano 等人[26]提出了一种自动化方法来推导出共现变更隐含的原因. 他们将共现变更的原因定义为共现变更的元素中共有的一组属性, 并考虑两种属性: 指示显式依赖关系的结构属性和揭示隐式依赖关系的语义属性. Mo 等人[27]提出了历史耦合依赖的概念, 在此基础上, 提出了一个新的模型, 历史共现耦合空间 (HCSpace), 以链接共同更改的文件并表示文件在历史上如何连接为一个组. Mondal 等人[28]在 2020 年研究了利用克隆检测的技术来增强最先进的关联规则挖掘技术在发现变更影响集方面的效果. Mondal 等人[29]提出了一种轻量级共现变更推荐技术, 该技术可以使用 WA-DiSC 为目标代码片段自动推荐片段级别的类似共同变更候选.

挖掘共现变更代码元素的历史演化信息经常被用于不同的软件开发任务上, 包括但不限于上述相关工作中提及的代码影响集分析、重构模式检测、代码坏味检测以及代码智能化变更等. 本文利用代码共现修改信息来辅助代码评审的代码理解, 可以很好地预防代码变更带来的缺陷以及代码坏味.

4.3 代码变更挖掘

本文所定义的相关代码变更指的是在一个方法级别的代码片段中, 当包含相似的代码语句变更操作达到一定比例时, 代码变更是相关的. 相关代码变更追溯指的是在同一个项目的历史代码提交中, 在涉及某个特定方法级别的代码片段的修改的代码提交集合中, 追溯出相关的代码变更. 很多研究工作[30–34]关注于相似代码的检测与挖掘及代码克隆技术的研究. 与代码克隆等相似代码挖掘任务不同的是, 相似代码变更的挖掘更关注于代码语句在变更上的相似性, 即不仅是局限于代码本身内容的相似性, 因此, 相似代码变更所关注的相似特征及计算方法也有所不同, 现有关于相似代码变更挖掘的研究工作相对较少. Nguyen 等人[35]对软件演化中这种代码变更的重复性进行了大规模数据上的实证研究, 研究表明, 代码变更的重复性在变更规模较小时可能高达 70%–100%, 并且随着规模的增加呈指数下降. 其次, 在跨项目中的代码变更重复性比在项目内的比例更高. Nadi 等人[36]提出了基于相似代码变更集来推荐代码更改, 即每个新更改集可能与之前更改集的实例相似. Mondal 等人[37]同样对基于相似代码变更进行代码片段推荐任务进行了实证性研究, 并表明这种相似性对于推荐效果的提升显著. Kreutzer 等人[38]提出了基于聚类算法的相似代码变更挖掘的工具 C3, 并将挖掘出的相似变更数据集作为输入用于缺陷修复的工具 LASE[39]上, 取得了一定效果. Nguyen 等人[40]提出了工具 CPatMiner, 该方法利用静态程序分析技术构建了细粒度的代码变更依赖图, 并基于图异构的算法在大规模的数据集上进行了相似代码变更模式的挖掘, 并取得了当前研究在相似代码变更挖掘上的最佳效果.

与以往基于聚类或基于图的相似代码变更检测等方法不同的是, 本文采取一种融合代码结构特征的细粒度代码变更向量化表示的方法去追溯相关代码变更, 首先通过解析代码抽象语法树进行代码变更结构信息的抽取, 再进行向量化表示, 并提出启发式的相关度计算方法. 此外, 通过人工验证分析, 本文追溯出的相关代码变更对于评审者理解当前的代码变更起到了显著的参考作用, 为评审者评审决策当前代码提交提供了重要参考信息.

5 总结与展望

一个成熟的软件项目积累了大量的开发历史, 并且处于不断变化的过程中. 面向新的代码变更的提交评审, 评审者往往需要理解变更代码的来龙去脉. 现有工具在实现了基本的历史代码提交回溯功能后, 往往缺乏进一步地提炼出辅助理解与决策代码变更评审的有用信息. 本文提出了一种基于代码演化历史的细粒度的代码变更溯源方

法, 通过提取细粒度的代码变更在历史上的相关代码提交, 从中回溯出相关的频繁共现修改的代码元素以及相关的代码变更, 并将这些历史演化信息反馈给代码评审者以辅助评审者对当前代码变更的理解与评审决策, 从而加快代码评审的效率以及提升评审质量. 本文在公开的 10 个项目的数据集上验证了 100 个方法级别的细粒度代码变更的溯源效果. 结果表明, 本文方法在回溯历史演化信息任务上具备良好的有效性. 通过人工分析, 结果表明回溯出的历史演化信息可以很好地辅助代码评审者对当前代码变更的理解与评审. 未来工作中, 我们将考虑如何利用更多的历史演化信息综合进行代码变更的决策评审, 进一步提升代码评审的智能化程度.

References:

[1] Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 712−721. [doi: 10.1109/ICSE.2013.6606617]

[2] 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]

[3] Higo Y, Hayashi S, Kusumoto S. On tracking Java methods with Git mechanisms. Journal of Systems and Software, 2020, 165: 110571. [doi: 10.1016/j.jss.2020.110571]

[4] Grund F, Chowdhury SA, Bradley NC, Hall B, Holmes R. CodeShovel: Constructing method-level source code histories. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1510–1522. [doi: 10.1109/ICSE43902.2021.00135]

[5] Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. Detecting bad smells in source code using change history information. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. Silicon Valley: IEEE, 2013. 268–278. [doi: 10.1109/ASE.2013.6693086]

[6] Arnold RS. Software Change Impact Analysis. Washington: IEEE Computer Society Press, 1996.

[7] Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. Vasteras: ACM, 2014. 313–324. [doi: 10.1145/2642937.2642982]

[8] Han JW, Cheng H, Xin D, Yan XF. Frequent pattern mining: Current status and future directions. Data Mining and Knowledge Discovery, 2007, 15(1): 55–86. [doi: 10.1007/s10618-006-0059-1]

[9] Myers EW. An O(*ND*) difference algorithm and its variations. Algorithmica, 1986, 1(1): 251–266. [doi: 10.1007/BF01840446]

[10] Wuensch KL. What is a Likert scale? And how do you pronounce 'Likert?' Technical Report, 2005-10-04/2009-04-30, East Carolina University.

[11] Hata H, Mizuno O, Kikuno T. Historage: Fine-grained version control system for Java. In: Proc. of the 12th Int'l Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution. Szeged: ACM, 2011. 96–100. [doi: 10.1145/2024445.2024463]

[12] Maruyama K, Kitsu E, Omori T, Hayashi S. Slicing and replaying code change history. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 246–249. [doi: 10.1145/2351676.2351713]

[13] Servant F, Jones JA. History slicing: Assisting code-evolution tasks. In: Proc. of the 20th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering. Cary: ACM, 2012. 43. [doi: 10.1145/2393596.2393646]

[14] Servant F, Jones JA. Fuzzy fine-grained code-history analysis. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 746–757. [doi: 10.1109/ICSE.2017.74]

[15] Li Y, Zhu CG, Rubin J, Chechik M. Semantic slicing of software version histories. IEEE Trans. on Software Engineering, 2018, 44(2): 182–201. [doi: 10.1109/TSE.2017.2664824]

[16] Li Y, Rubin J, Chechik M. Semantic slicing of software version histories (T). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lincoln: IEEE, 2015. 686–696. [doi: 10.1109/ASE.2015.47]

[17] Li Y, Zhu CG, Rubin J, Chechik M. Precise semantic history slicing through dynamic delta refinement. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 495–506.

[18] Zimmermann T, Weibgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. In: Proc. of the 26th Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 563–572. [doi: 10.1109/ICSE.2004.1317478]

[19] Agrawal R, Srikant R. Fast algorithms for mining association rules in large databases. In: Proc. of the 20th Int'l Conf. on Very Large Data Bases. Santiago de Chile: Morgan Kaufmann, 1994. 487–499. [doi: 10.5555/645920.672836]

[20] van Rysselberghe F, Rieger M, Demeyer S. Detecting move operations in versioning information. In: Proc. of the 2006 Conf. on Software Maintenance and Reengineering. Bari: IEEE, 2006. 8-278. [doi: 10.1109/CSMR.2006.23]

[21] Zou LJ, Godfrey MW, Hassan AE. Detecting interaction coupling from task interaction histories. In: Proc. of the 15th IEEE Int'l Conf. on Program Comprehension. Banff: IEEE, 2007. 135–144. [doi: 10.1109/ICPC.2007.18]

[22] D'Ambros M, Lanza M, Lungu M. Visualizing co-change information with the evolution radar. IEEE Trans. on Software Engineering, 2009, 35(5): 720–735. [doi: 10.1109/TSE.2009.17]

[23] Ali N, Jaafar F, Hassan AE. Leveraging historical co-change information for requirements traceability. In: Proc. of the 20th Working Conf. on Reverse Engineering. Koblenz: IEEE, 2013. 361–370. [doi: 10.1109/WCRE.2013.6671311]

[24] Mondal M, Roy CK, Schneider KA. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In: Proc. of the 21st Int'l Conf. on Program Comprehension. San Francisco: IEEE, 2013. 103–112. [doi: 10.1109/ICPC.2013.6613838]

[25] Mondal M, Roy CK, Schneider KA. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. In: Proc. of the 2014 Software Evolution Week-IEEE Conf. on Software Maintenance, Reengineering, and Reverse Engineering. Antwerp: IEEE, 2014. 358–362. [doi: 10.1109/CSMR-WCRE.2014.6747194]

[26] Lozano A, Noguera C, Jonckers V. Explaining why methods change together. In: Proc. of the 14th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. Victoria: IEEE, 2014. 185–194. [doi: 10.1109/SCAM.2014.27]

[27] Mo R, Zhan MY. History coupling space: A new model to represent evolutionary relations. In: Proc. of the 26th Asia-Pacific Software Engineering Conf. Putrajaya: IEEE, 2019. 126–133. [doi: 10.1109/APSEC48747.2019.00026]

[28] Mondal M, Roy B, Roy CK, Schneider KA. Associating code clones with association rules for change impact analysis. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. London: IEEE, 2020. 93–103. [doi: 10.1109/SANER48275.2020.9054846]

[29] Mondal M, Roy CK, Roy B, Schneider KA. FLeCCS: A technique for suggesting fragment-level similar co-change candidates. In: Proc. of the 29th IEEE/ACM Int'l Conf. on Program Comprehension. Madrid: IEEE, 2021. 160–171. [doi: 10.1109/ICPC52881.2021.00024]

[30] Johnson JH. Substring matching for clone detection and change tracking. In: Proc. of the 1994 Int'l Conf. on Software Maintenance. Victoria: IEEE, 1994. 120–126. [doi: 10.1109/ICSM.1994.336783]

[31] Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proc. of the 16th IEEE Int'l Conf. on Program Comprehension. Amsterdam: IEEE, 2008. 172–181. [doi: 10.1109/ICPC.2008.41]

[32] Wang PC, Svajlenko J, Wu YZ, Xu Y, Roy CK. CCAligner: A token based large-gap clone detector. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 1066–1077. [doi: 10.1145/3180155.3180179]

[33] Krinke J. Identifying similar code with program dependence graphs. In: Proc. of the 8th Working Conf. on Reverse Engineering. Stuttgart: IEEE, 2001. 301–309. [doi: 10.1109/WCRE.2001.957835]

[34] Perumal A, Kanmani S, Kodhai E. Extracting the similarity in detected software clones using metrics. In: Proc. of the 2010 Int'l Conf. on Computer and Communication Technology. Allahabad: IEEE, 2010. 575–579. [doi: 10.1109/ICCCT.2010.5640465]

[35] Nguyen HA, Nguyen AT, Nguyen TT, Nguyen TN, Rajan H. A study of repetitiveness of code changes in software evolution. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. Silicon Valley: IEEE, 2013. 180–190. [doi: 10.1109/ASE.2013.6693078]

[36] Nadi S, Holt R, Mankovskii S. Does the past say it all? Using history to predict change sets in a CMDB. In: Proc. of the 14th European Conf. on Software Maintenance and Reengineering. Madrid: IEEE, 2010. 97–106. [doi: 10.1109/CSMR.2010.14]

[37] Mondal M, Roy CK, Schneider KA. An empirical study on ranking change recommendations retrieved using code similarity. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. Osaka: IEEE, 2016. 44–50. [doi: 10.1109/SANER.2016.53]

[38] Kreutzer P, Dotzler G, Ring M, Eskofier BM, Philippsen M. Automatic clustering of code changes. In: Proc. of the 13th Int'l Conf. on Mining Software Repositories. Austin: ACM, 2016. 61–72. [doi: 10.1145/2901739.2901749]

[39] Meng N, Kim M, McKinley KS. LASE: Locating and applying systematic edits by learning from examples. In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 502–511. [doi: 10.1109/ICSE.2013.6606596]

[40] Nguyen HA, Nguyen TN, Dig D, Nguyen S, Tran H, Hilton M. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 819–830. [doi: 10.1109/ICSE.2019.00089]

附中文参考文献:

[2] 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]

王敏(1994－), 男, 博士, 主要研究领域为软件工程, 软件复用, 代码审查.

邹艳珍(1976－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件工程, 软件复用, 知识图谱, 智能软件开发.

潘兴禄(1997－), 男, 硕士生, CCF 学生会员, 主要研究领域为软件工程, 软件复用.

谢冰(1970－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 形式化方法, 软件复用, 智能软件开发.