软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(11):5282−5311 [doi: 10.13328/j.cnki.jos.006746]
©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

软件库依赖图谱的复杂性度量方法及其潜在应用*

于 海[1], 王 莹[1,3], 徐美秋[1], 杨 博[1], 许 畅[2,3], 朱志良[1]

[1](东北大学 软件学院, 辽宁 沈阳 110169)
[2](南京大学 计算机科学与技术系, 江苏 南京 210046)
[3](计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210046)
通信作者: 王莹, E-mail: wangying@swc.neu.edu.cn

摘 要: 在软件开发过程中, 软件库可以减少开发时间和节约成本而被广泛使用, 因此现代软件项目包含多种不同来源的代码而使得系统具有更高的复杂性和多样性. 软件库在使用的过程中常常伴随着各种风险, 如低质量或安全漏洞, 从而严重影响软件项目的质量. 通过分析与软件库的耦合强度, 来量化由软件库的依赖关系而引入客户代码的复杂性和多样性. 首先, 根据客户代码与软件库之间方法的调用关系建立软件边界图模型, 区分开客户代码和软件库的代码边界; 进而基于此提出一套软件库依赖图谱的复杂性度量指标 RMS, 用以量化不同来源软件之间的耦合强度. 在实验过程中, 挖掘 Apache 开源社区中 10 个流行软件所有历史版本数据, 最终收集到 7857 个真实项目间依赖缺陷问题. 在上述真实数据基础上, 结合所提出的复杂性度量指标 RMS, 利用假设验证方法开展实证调查研究来探讨: H1: 风险因子更高的边界节点是否更容易引入更多数量的项目间依赖缺陷; H2: 风险因子更高的边界节点会是否更容易引入严重等级高的项目间依赖缺陷; H3: RMS 度量指标数值多大程度地影响了引入项目间依赖缺陷数量和严重等级. 实验结果表明, 根据 RMS 度量指标评估, 与软件库耦合度更高的边界节点容易引入更多数量且严重等级高的项目间依赖缺陷. 与传统复杂性度量指标对比, RMS 度量指标较大程度地影响了引入项目间依赖缺陷的数量和严重等级.

关键词: 经验软件工程; 第三方库; 软件度量指标; 假设验证
中图法分类号: TP311

中文引用格式: 于海, 王莹, 徐美秋, 杨博, 许畅, 朱志良. 软件库依赖图谱的复杂性度量方法及其潜在应用. 软件学报, 2023, 34(11): 5282–5311. http://www.jos.org.cn/1000-9825/6746.htm

英文引用格式: Yu H, Wang Y, Xu MQ, Yang B, Xu C, Zhu ZL. Measurement Method for Complexity of Software Library Dependency Graph and Its Potential Applications. Ruan Jian Xue Bao/Journal of Software, 2023, 34(11): 5282–5311 (in Chinese). http://www.jos.org.cn/1000-9825/6746.htm

Measurement Method for Complexity of Software Library Dependency Graph and Its Potential Applications

YU Hai[1], WANG Ying[1,3], XU Mei-Qiu[1], YANG Bo[1], XU Chang[2,3], ZHU Zhi-Liang[1]

[1](Software College, Northeastern University, Shenyang 110169, China)
[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210046, China)
[3](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210046, China)

Abstract: In the process of software development, software libraries are widely used as they can reduce development time and costs. Consequently, modern software projects contain code from different sources, which makes the systems highly complex and diversified. In

* 基金项目: 国家自然科学基金 (62141210, 61932021, 61902056, 61802164, 61977014); 沈阳市中青年科技创新人才计划 (ZX20200272); 中央高校基本科研业务费 (N2217005); 南京大学软件新技术国家重点实验室开放基金 (KFKT2021B01)
收稿时间: 2021-09-18; 修改时间: 2022-03-11, 2022-05-21; 采用时间: 2022-07-16; jos 在线出版时间: 2023-06-16
CNKI 网络首发时间: 2023-06-19

© 中国科学院软件研究所 http://www.jos.org.cn

addition, various risks come along with the usage of software libraries, such as low quality or security vulnerabilities, seriously affecting the quality of software projects. By analyzing the intensity of the coupling with software libraries, this study quantifies the complexity and diversity introduced by the dependence on the software libraries to the client code. For this purpose, a software boundary graph (SBG) model is constructed according to the method invocation relationships of the client code with the software libraries to distinguish their code boundaries. Then, a metric suite RMS for the complexity of the software library dependency graph is proposed on the basis of the SBG model to quantify the intensity of the coupling with the software from different sources. In the experiment, this study mines the data on all the historical versions of 10 popular software in the Apache open-source community and finally collects 7857 dependency defects among real-world projects. With the above-mentioned real-world data, empirical investigation based on hypothesis testing is conducted according to the proposed complexity metric suite RMS to discuss the following issues: H1: whether boundary nodes with higher risk factors are more likely to introduce more inter-project dependency defects; H2: whether boundary nodes with higher risk factors are more likely to introduce serious inter-project dependency defects; H3: what is the extent to which the value of the metric suite RMS affects the number and severity of introduced inter-project dependency defects. Experimental results show that according to the evaluation with the RMS, the boundary nodes exhibiting higher coupling degrees with the software libraries are more likely to introduce more inter-project dependency defects with higher severity. Moreover, compared with traditional complexity metrics, RMS greatly influences the number and severity of introduced inter-project dependency defects.
Key words: empirical software engineering; third-party libraries; software metrics; hypothesis testing

开源软件社区 (open-source software community) 允许用户自由学习、修改、拷贝和发布软件信息, 以此方式得到了广泛流行和认可. 为了提高开发效率及节约工作成本, 开发者不可避免地借助于第三方开源软件库 (简称为软件库) 来实现预期的功能[1−3]. 在开源时代, 软件不仅结构复杂、规模庞大, 还包含大量不同来源, 结构各异的第三方代码及构件, 呈现出一种“混源”形态. 中国信息通信研究院于 2020 年发布的《开源生态白皮书》[1]显示, 全球最大的开源代码托管平台 GitHub 已包含软件 1.4 亿, 预计 2026 年将突破 3 亿. 目前已经有 90% 以上的企业全面拥抱开源, 平均每个项目依赖开源组件 14 个, 大规模软件项目引入组件的数量可高达 200–1000 个. 大量软件库的引入伴随开放式、自动化的构建方式, 为开源时代软件开发带来便利的同时, 也为软件项目自身及其供应链的可靠性、可信性及可持续维护性带来了隐患.

若软件 *A* 在开发过程中依赖了软件 *B* 来实现部分功能, 则我们称 *A* 为 *B* 的客户代码 (client code), *B* 为 *A* 的软件库 (library). 如果软件在运行过程中出现无法满足正确规约的状态, 则本文将这种破坏正常运行能力的错误定义为缺陷. 软件库在为开发者带来便利的同时也会带来如下弊端[2]: (1) 如果开源软件社区中多数用户对某些软件库兴趣减弱, 则上述软件库便会渐渐地不再被维护或扩展功能, 这将严重影响客户代码的可维护性; (2) 隐含在软件库中的安全漏洞和缺陷极易通过依赖关系对客户软件系统进行攻击; (3) 客户代码与软件库在各自的生命周期中没有受到统一的开发和测试质量标准的制约. 如果软件库缺乏清晰的文档和良好的代码结构, 或者软件库也同时依赖了多个低质量的软件库, 这些情况也极易通过依赖关系引入到客户代码中. 通过与软件库之间的依赖关系而引入客户代码中的代码缺陷我们称为项目间依赖缺陷.

传统的面向对象软件复杂性的度量方法 (如 CK 度量套件和 MOOD 度量方法等) 和结构化程序复杂性的度量方法 (代码行和 McCabe 圈度量方法等) 已被广泛应用于实际的软件开发过程中, 用于量化软件的质量属性, 发现代码的缺陷, 从而确保软件产品的质量. 遗憾的是, 现如今由于现代软件具备如下特性而使得传统的面向对象软件的复杂性度量方法对其不再适用: (1) 复杂异质异构. 软件项目所集成的各个软件库由具备不同背景和能力的人员开发, 因此代码的结构和质量差异较大; (2) 程序状态与行为空间爆炸. 集成多个软件库之后, 软件项目的复杂性和缺陷检测代价随着其规模的增长呈指数级增加; (3) 多源异步演化. 不同来源的软件库产品生命周期演化过程互不同步, 因此如何获取异源软件之间复杂交互行为的变化是一个需要解决的问题. 然而, 迄今没有新的度量模型能够针对现代软件项目的上述特性, 准确地量化由软件库的依赖强度而引入客户软件项目中的复杂性和多样性.

鉴于此, 本文从软件项目的拓扑结构出发, 针对多种质量要素, 提出一种软件库依赖图谱的复杂性度量套件 RMS 来刻画项目间依赖缺陷. 首先, 根据不同来源的软件之间的复杂异质结构, 将客户代码中直接连接自身与软件库的方法视为软件的边界节点, 建立软件边界图模型 (software boundary graph, SBG). 然后, 在规模庞大的混源

© 中国科学院软件研究所 http://www.jos.org.cn

形态下, 量化软件边界节点与软件库之间的边缘性、可达性、多样性、穿透性、交互性和枢纽性等. 同时, 实现了一个自动化的软件库依赖图谱的复杂性度量工具 Certifies 来获取生命周期中任意时刻的各个质量要素, 以评估客户代码质量. 实验中, 以开源软件项目作为对象, 分析 RMS 度量指标套件与代码缺陷的相关性以及其数值对项目间依赖缺陷和严重等级的影响程度. 实验结果表明根据 RMS 度量指标评估, 与软件库耦合程度更高的节点容易引入更多数量且严重等级高的缺陷, 与传统复杂性度量指标对比, RMS 度量指标较大程度地影响了引入项目间依赖缺陷的数量和严重等级.

本文第 1 节介绍相关研究工作. 第 2 节介绍本文研究动机, 通过数据收集定位到项目间依赖缺陷, 并对其普遍性、严重性信息进行统计. 第 3 节建立软件边界图模型并结合该模型提出 RMS 度量指标套件, 同时介绍软件库依赖图谱的复杂性度量工具 Certifies 的主要功能特征. 第 4 节可视化真实开源软件的 RMS 度量指标数据, 并对结果进行分析讨论. 第 5 节利用假设验证法来探究 RMS 度量指标与项目间依赖缺陷的相关性. 第 6 节介绍了实验过程的效度威胁性. 最后总结全文并对未来研究工作进行了展望.

1 相关研究

1.1 软件库依赖分析技术

随着开源软件社区的迅速发展, 软件库是重要的可复用资源, 在软件开发中扮演着不可或缺的角色, 越来越多的研究人员对软件库进行了相关研究.

广泛地使用软件库来扩展功能及提高性能是现如今移动应用程序主流的开发模式. 由于目前大部分研究检测软件库不完善且不准确, 王浩宇等人[4]提出一种自动检测和分类软件库的方法, 利用多级聚类技术准确地识别软件库, 同时结合机器学习理论对软件库的功能进行准确分类. Android 系统只提供一种粗粒度的权限控制机制, 软件库会与其所在的主应用程序具有相同的权限, 导致软件库权限过大, 对用户隐私造成严重威胁, 湛家伟等人[5]通过对恶意软件库隔离技术的相关背景和已有方案进行综述, 分析研究工作中存在的问题. 针对 Android 系统面临的隐私数据泄露、关键配置信息泄露等安全问题, 路晔绵等人[6]对部分 Android 应用和软件库设计了数据劫持攻击和拒绝服务攻击方案, 并分析软件库与客户应用的特征, 从而实现了漏洞静态检测工具. 软件产品开发的成败很大程度上取决于是否为预期的软件产品选择了合适的软件库. 由于理论与实践存在差距, 在工业实践中很少采用理论的方法来选择合适的软件库, Ayala 等人[7]调查了工业实践中软件库的选取方案, 以提供初步的经验基础, 使得研究与工业实践协调统一. Java 虚拟机开发实现的过程异常繁琐, 涉及不同领域的知识. 为解决这个难题, Geoffray 等人[8]通过使用软件库来实现 Java 虚拟机, 最终得到的产品在功能和性能上均与工业顶级的开源 Java 虚拟机相当. Haddox 等人[9]基于软件包的概念, 提出了一种通过理解和测试软件组件的方式来降低软件风险的技术, 以便更好地理解软件库集成进客户代码中的行为方式. Raemaekers 等人[2]通过调查软件库的使用频次, 计算其“共性”和“隔离”等级, 作为软件库使用风险的指标, 从而实现自动化地评估软件的使用给客户代码的可靠性带来的风险.

尽管已经涌现很多软件库依赖分析相关的技术研究, 然而很少有文献从软件项目的结构特征出发来度量其质量属性, 为软件库的使用和客户代码的维护过程提供指导. 本文从现代软件复杂异质异构的视角探究项目间依赖缺陷问题.

1.2 面向对象的软件网络模型

复杂网络的研究强调从整体上把握系统, 研究复杂系统在宏观上涌现出的新特性, 该技术可以为理解软件系统提供有价值的视角和分析维度[10].

Valverde 等人[11]首先将复杂网络理论应用到软件系统拓扑结构分析中, 通过逆向工程的方法从程序代码中得到类图, 将此面向对象软件系统的类图作为研究对象, 用无向网络表示软件系统, 即模型中的节点表示类, 边表示类之间的继承、关联等交互关系. 随后, de Moura 等人[12]、Wheeldon 等人[13]、Valverde 等人[14]和 Myers 等人[15]使用有向网络描述软件系统的结构, 将模块或函数抽象为节点, 它们之间的调用关系抽象为有向网络中的边, 将这种描述软件结构的网络定义为软件协作图或者软件依赖网络, 并认为软件系统实质上代表了一种人工复杂网络.

© 中国科学院软件研究所 http://www.jos.org.cn

软件系统的复杂网络研究可以用形式化的方法来描述软件系统的性质, 具有良好的数据基础, 更容易洞察系统整体的特征和规律[10]. Lopez-Fernandez 等人[16]研究模块网络显示出模块间交互关系. Zimmermann 等人[17]在 Windows Server2003 中根据二进制文件之间的数据依赖关系和调用依赖关系, 将子系统建模为依赖图来分析内部间交互关系. Musco 等人[18]基于该理论提出一套软件依赖有向图生成模型更好理解软件系统演化的过程. Bhattacharya 等人[19]应用该理论建立软件系统中静态函数调用和模块协作有向网络图以及开发者协作有向网络图来分析软件系统演化过程. Wang 等人[20−22]将软件系统抽象成多粒度、多类型依赖关系的网络模型, 在拓扑结构分析的基础上, 结合故障树分析、聚类分析及多目标优化算法等提出了一系列软件质量保证技术, 包括软件可靠性风险分析算法、测试用例优先级排序技术、集成测试序列生成算法及体系结构级别自动化重构技术, 以提高软件的可维护性.

然而, 上述软件网络模型均未考虑客户代码与软件库依赖交互的视角. 由于软件库的频繁使用, 软件项目常会出现多种质量风险问题, 为此我们结合软件的拓扑结构特征建立软件边界图模型进而分析其多维质量属性.

1.3 软件复杂性度量方法

软件度量学目的是利用度量学方法科学地评价软件质量并对软件缺陷进行预测, 更有效地对软件开发过程进行控制和管理, 合理地组织和分配资源, 制定切实可行的软件开发计划, 以低成本获取高质量的软件[23].

Vasa 等人[24]根据软件网络的边数和节点数之间的关系来研究系统结构的变化, 从而预测软件的规模和构造软件系统所需的代价. Girolamo 等人[25]根据介数等指标定义了一套类层、网络层和设计层的度量指标, 在不同层次识别和检测软件结构的缺陷及有问题的类, 从而评估软件设计的质量. Zimmermann 等人[17]借助图论定义了一系列与拓扑结构相关的复杂性度量指标, 并基于社交网络分析中的网络中心度指标识别出处于系统中心位置的程序模块, 从而预测模块部署后的缺陷数目. 随着多媒体软件的广泛应用, 如何度量与提高软件的质量成为一个重要的问题, 为了计算软件质量的复杂性和模糊性, Yi 等人[26]采用模糊物元分析法, 提出了一种基于模糊物元的软件质量评估模型. 张文等人[27]将体系结构分为动态复杂性与结构复杂性, 提出了基于熵的动态复杂性和结构复杂性评估方法. 为了提高软件的可靠性, 可将软件系统分解为较小的子系统, Lew 等人[28]提出了模块内、外部的复杂性度量方法, 为系统的分解提供指导. 为了度量组件化软件系统的总体复杂性, Tiwari 等人[29]提出了一种基于组件的循环复杂度计算方法, 即分析各个组件的复杂性及组件间交互的复杂性.

现有的度量指标均未从不同来源的软件之间的依赖关系角度去审视软件系统的复杂性. 而本文的研究目的是通过分析客户代码与软件库的耦合强度来量化软件项目的多维度质量属性.

1.4 软件缺陷预测研究

软件缺陷预测是当前软件仓库挖掘领域的研究热点, 目的是在项目开发的初期, 识别出项目中的疑似包含缺陷的代码模块. 在此基础上, 对这种代码模块分配足够的测试资源来保证进行充分的代码审查或单元测试, 以提高软件的质量[30].

缺陷预测可分为同项目缺陷预测和跨项目缺陷预测. 在实际应用中需要预测项目的历史数据在有些情况下较为稀缺, 因此跨项目缺陷预测成为软件缺陷预测领域内的研究热点. 该项研究的挑战在于源项目与目标项目数据集间存在的分布差异性以及数据集内存在的类不平衡问题. 类不平衡是缺陷预测数据集中普遍存在的问题, 因缺陷在软件模块中的分布大致符合帕累托原则, 即大部分 (约 80%) 的缺陷集中位于小部分 (约 20%) 的程序模块内. 在收集的数据集中, 非缺陷模块 (多数类) 的数量要远远超过缺陷模块 (少数类) 的数量, 导致模型在预测时偏向多数类, 而对少数类的预测精准度较低. 为了克服这一难题, 何吉元等人[31]提出了一种基于搜索的半监督式跨项目软件缺陷预测方法. 陈翔等人[32]提出了一种软件缺陷预测的研究框架, 并对所提出框架的 3 个重要影响因素 (度量元的设定、缺陷预测模型的构建方法及缺陷预测数据集的相关问题) 进行了深入分析. 在缺陷预测数据集的收集过程中, 由于多种不同的特征会造成维数灾难问题, 陈翔等人[33]将软件缺陷预测特征选择问题转变为多目标优化问题, 提出了 MOFES 方法来平衡两个目标: 特征子集的规模和缺陷预测模型的预测效果.

D’Ambros 等人[34]将版本演化过程中, 频繁地同时进行变更的一组模块视为具有变更耦合性, 通过度量软件

© 中国科学院软件研究所 http://www.jos.org.cn

系统演化过程中的历史版本变更信息, 分析模块之间的变更耦合性与软件缺陷的相关性, 进而对软件缺陷进行预测来提高软件质量. Schröter 等人[35]对 52 个 Eclipse 插件进行经验调查, 并基于软件组件间的依赖关系构建模型, 以此预测容易出错的软件组件. Nagappan 等人[36]根据软件组件依赖关系构建网络关系图, 分析其软件依赖性、代码改动和缺陷三者之间的关系, 结果表明软件依赖性和代码改动可以有效地预测缺陷. Holmes 等人[37]提出一种识别程序调用依赖关系的方法, 进而根据静态和动态调用图有效地预测函数的修改情况. Bhattacharya 等人[19]定义了一套度量指标并将之应用到静态函数调用、模块协作有向网络图及开发者协作有向网络图中, 以此来预测缺陷的严重等级、软件系统的维护工作量及缺陷数量.

现有的缺陷预测方法大多是针对传统软件、开源软件或者面向某个特殊领域而提出的. Ma 等人[38]对开源社区中的程序员们展开了关于项目间依赖缺陷的在线问卷调查, 其调查结果显示, 在开发人员遇到的缺陷中, 项目间依赖缺陷平均占据了 17.28%. 相比于项目内部及独立的缺陷, 超过一半的采访者认为项目间依赖缺陷更难修复, 且 40.74% 的采访者认为项目间依赖缺陷会造成更加严重的影响. 因此, 本文通过建立软件项目的边界图模型, 提出一套度量指标分析不同来源软件之间的耦合强度, 进而量化耦合强度与代码缺陷的相关性以分析软件项目中由于不良的代码结构对引入项目间依赖缺陷数量和严重等级造成多大程度的影响.

2 研究动机

为调查项目间依赖缺陷在软件项目中发生的普遍性和严重性, 我们做了如下数据收集和分析工作. 首先, 收集开源社区中依赖了较多软件库且历史维护信息丰富的 Java 项目. 进而在它们相应的缺陷跟踪系统中挖掘并分析由于与软件库的直接或传递性依赖关系而引入客户代码中的项目间依赖缺陷.

2.1 数据收集

本文在 Apache 开源社区选取 Java 语言编写的软件作为实验对象, 选取 Apache 软件的原因如下: (1) Apache 是最受欢迎的开源社区之一, 由于其跨平台和安全性而被广泛使用, 且包含多种不同类型的软件. 目前很多经验软件工程的调查研究[39–46]都开展于 Apache 开源社区的软件系统之上, 因此选择它作为实验对象具有一定代表性. (2) Apache 社区具有丰富的软件维护历史信息. 具体来说使用缺陷跟踪系统 (如 Bugzilla 和 JIRA 系统) 来跟踪缺陷; Apache 软件基金会为所有项目提供 Git 镜像的官方支持; 缺陷报告和代码存储库向公众开放, 因此这些条件为我们研究目标问题提供了极大的便利.

同时, 我们选择的实验对象满足如下条件: (1) 具有超过 5 年的开发历史以及代码提交记录平均不少于 5000 次. 这些条件可以充分说明软件系统的代码历史记录完整, 对应的缺陷跟踪系统的描述信息丰富, 有助于对其分析理解; (2) 至少有 30 个发布版本依赖软件库的个数大于 30. 所使用的软件库越多, 那么出现项目间依赖缺陷问题的可能性就越大. 上述条件对于项目间依赖缺陷问题的分析和研究是十分必要的. 我们最终随机选取满足上述条件的 10 个 Java 软件项目, 收集其对应的依赖软件库个数大于 30 的所有发布版本作为实验对象. 表 1 为实验对象的统计信息, 属性包括: 软件发布版本的起始时间、代码提交次数和依赖软件库的个数和版本数.

表 1 实验对象

实验对象	起始时间	#Commits	#Library (Avg)	#Version
Maven	2009/07	10562	41	53
Jmeter	2011/11	16405	63	30
Groovy	2003/12	16207	40	150
Cassandra	2010/03	24720	53	182
Jena	2012/05	8014	37	56
Tomcat	2010/04	21330	31	175
Camel	2009/01	39531	294	120
Xerces-J	1999/11	5510	30	49
Batik	2000/10	5000	35	30
Fop	1999/11	8307	30	20

© 中国科学院软件研究所 http://www.jos.org.cn

2.2 定位项目间依赖缺陷

Bugzilla 和 JIRA 均为缺陷跟踪系统, 可以管理软件开发中缺陷的提交、修复和关闭等整个生命周期. 每个实验对象所关注的版本发布时间范围内, 根据缺陷跟踪系统所记录的软件缺陷的描述信息和开发人员关于缺陷的讨论内容, 如图 1 所示, 我们通过以下 4 个步骤准确定位到项目间依赖缺陷报告数据集.

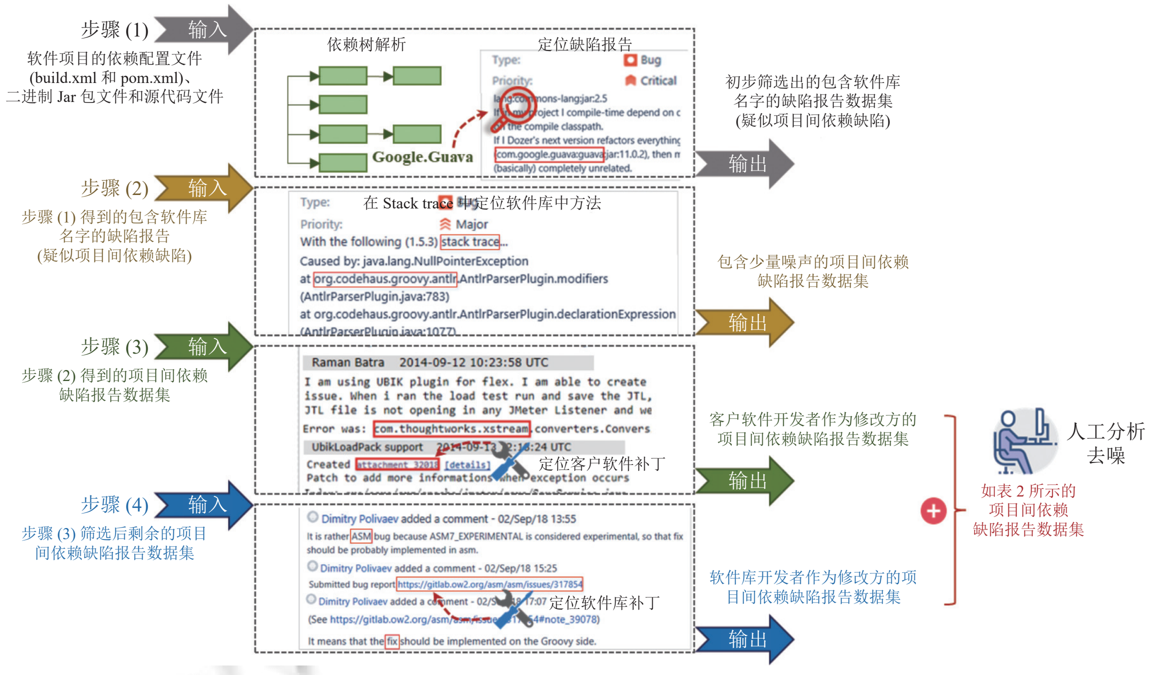


图 1　定位项目间缺陷的方法

(1) 首先通过解析软件项目的依赖配置文件 (build.xml 和 pom.xml 等) 获取实验对象的所有版本中软件库依赖树信息. 进而将项目依赖的软件库名字作为关键词, 以定位到问题的相关描述及开发人员的评论中出现相应软件库名字的缺陷报告. 将其视为疑似由于使用这些软件库而引起的项目间依赖缺陷问题. 同时为了保证报告描述的问题是开发者承认的缺陷, 我们过滤掉 open, duplicated, invalid, won't fix 等状态的缺陷报告, 只保留标记为 fixed 或 workaround (缺陷报告被标记为 workaround, 是指问题已经被开发者确认, 但是没有找到官方的修复方案, 只是用其他办法暂时解决[47]) 已经被确认修复的缺陷报告.

在技术实现上, 我们需要软件项目的依赖配置文件 (build.xml 或 pom.xml)、二进制 Jar 包文件和源代码文件作为输入信息. 首先, 利用于依赖配置文件能够解析软件库的依赖树结构, 针对支持 Ant 构建工具的依赖配置文件 build.xml, 基于 ApacheIvy 插件来解析软件库依赖树; 而针对支持主流的 Maven 构建工具的依赖配置文件 pom.xml, 基于 Dependency 插件来解析软件库依赖树. 第二, 扫描客户软件的源码文件可以得知客户代码与软件库的界限. 第三, 借助 Soot 静态分析工具 (https://www.sable.mcgill.ca/soot/index.html) 逐一扫描每个软件库可获取每个软件库方法节点集合, 用以区别软件库之间的边界.

(2) 在第 (1) 步骤得到的缺陷报告数据集中, 为了准确地定位到项目间依赖缺陷问题, 我们只关注包含堆栈信息, 且堆栈信息中涉及软件库中方法的缺陷报告. 原因是堆栈信息记录了在程序发生异常时刻方法的执行轨迹, 如果执行轨迹最后一帧为软件库中的方法, 那么将该问题视为项目间依赖缺陷. 具体而言, 若在缺陷报告的描述和相关讨论内容中出现堆栈信息 (带有“stack trace”关键字), 且堆栈信息的最后一帧方法属于该项目所依赖的软件库, 则将满足该条件的缺陷报告收集起来.

(3) 经步骤 (2) 筛选过滤后的剩余缺陷报告数据集中, 为了保证报告描述的问题是开发者所承认的项目间依赖

© 中国科学院软件研究所 http://www.jos.org.cn

缺陷. 将同时满足下述两个条件的缺陷报告视为项目间依赖缺陷 (客户软件项目开发者作为修改方): ① 该缺陷信息中包含相关的修复补丁; ② 该补丁对直接或间接调用到的软件库中的问题方法进行了修改.

(4) 在步骤 (3) 筛选后剩余的缺陷报告数据集中, 所收集的项目间依赖缺陷报告 (软件库开发者作为修改方) 应同时满足: ① 缺陷报告描述中关联了该项目所依赖的软件库的缺陷链接; ② 关联的软件库缺陷报告中包含相关的修复补丁, 且补丁对该项目直接或间接调用到的问题方法进行了修改.

经上述步骤 (3)–(4) 收集的项目间缺陷报告数据依然可能存在噪声, 因此我们经过人工分析的方式进一步筛选. 由 3 名具有软件工程专业背景和开发经验的研究生独立地对每个缺陷信息进行分析理解, 过滤掉那些缺陷的描述信息中只是偶然地出现软件库的名字或方法名而并无具体语义的缺陷数据 (数据集下载链接: https://github.com/NEUSoftwareEngineering/inter-project-dependency-defects). 当 3 名数据收集者对缺陷分类的理解产生分歧时, 我们最终通过讨论的方式以达成结果统一. 上述去噪过程大约花费 3 名数据收集者的 45 个工作日.

图 2 列举了收集的项目间依赖缺陷实例[48–51]. 其中图 2(a) 为根据步骤 (1) 所收集到的噪声数据, 显然软件库名字 com.google.guava 只是偶然出现在开发人员关于缺陷的讨论内容中, 而并非真正意义上的项目间依赖缺陷, 因此在人工分析的过程中会被过滤掉. 图 2(b)–图 2(d) 分别对应步骤 (2)–(4) 筛选出的项目间依赖缺陷. 如图所示, 经步骤 (2) 得到的缺陷#Groovy-2604[49]对应的堆栈信息中, 最后一帧为该项目引入的软件库 org.codehaus.groovy.antlr 中的方法, 程序从客户代码的方法入口根据依赖关系执行到软件库中的方法 AntlrParserPlugin.modifiers() 时, 抛出 NullPointerException 异常, 我们将该问题视为项目间依赖缺陷. #Jmeter-56975[50]为经步骤 (3) 获取的项目间依赖缺陷实例, 其相应的补丁对方法 SaveService.showDebuggingInfo() 进行了修改 (该方法直接调用了软件库内部的方法), 且缺陷信息中提及并讨论了调用该软件库出现的问题. 经步骤 (4) 获取的缺陷实例如#Groovy-8727[51]所示, 该缺陷信息中关联了项目 Groovy 所依赖的软件库 ASM 的缺陷链接, 同时关联的缺陷链接对应的补丁对直接或间接被 Groovy 依赖的方法进行了修改, 则我们视该缺陷为请求上游软件库修复的项目间依赖缺陷.

(a) 经步骤 (1) 所收集的噪声数据#Maven-6739[48]

(b) 经步骤 (2) 筛选的项目间依赖缺陷#Groovy-2604[49]

(c) 经步骤 (3) 筛选的项目间依赖缺陷#Jmeter-56975[50]

(d) 经步骤 (4) 筛选的项目间依赖缺陷#Groovy-8727[51]

图 2　项目间依赖缺陷实例

2.3 普遍性统计信息

在表 1 所示的 10 个开源软件项目所对应的 865 个版本中, 根据上述方法收集的项目间依赖缺陷统计数据如后文表 2 所示. 选取的软件项目平均引入 65 个软件库, 项目间依赖缺陷平均占据缺陷总数的 21.34%. 其中, 项目间依赖缺陷数量最多的 Groovy, 其项目间依赖缺陷数量占它所有类型缺陷数据总数的 31.56%. 这说明由于软件库的依赖关系而导致的项目间依赖缺陷具有一定的普遍性, 应该引起开发人员的高度重视.

© 中国科学院软件研究所 http://www.jos.org.cn

2.4 严重性统计信息

为验证项目间依赖缺陷的严重性, 本文统计了收集到的所有缺陷对应在缺陷追踪系统的严重等级 (Blocker: 阻塞的、Critical: 关键的、Major: 严重的、Normal: 正常的、Minor: 次要的、Trivial: 不重要的以及 Enhancement: 增强的), 如表 3 所示. 从数据中可以看出, 7100 个缺陷 (90. 37%) 均分布在 Blocker、Critical、Major 和 Normal 这 4 个等级, 只有 757 (9. 63%) 的缺陷具有较低的严重等级 Minor、Trivial 及 Enhancement. 这说明项目间缺陷对软件项目的可维护性带来的影响是极大的.

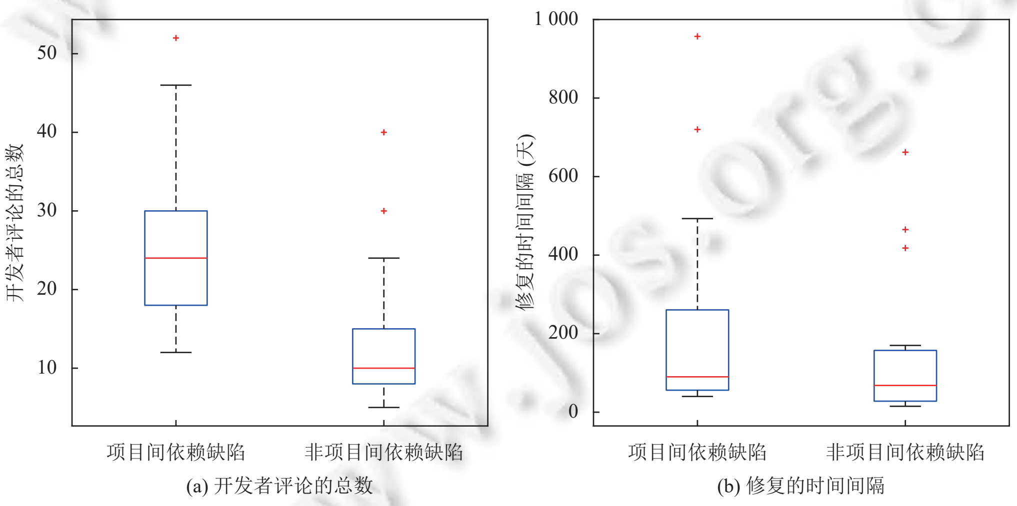
表 2 实验对象项目间依赖缺陷信息统计

实验对象	软件库数量	缺陷报告总数	项目间依赖缺陷报告数量 (所占比例 (%))
Maven	41	3097	815 (26.32)
Jmeter	63	4110	909 (22.12)
Groovy	40	5831	1840 (31.56)
Cassandra	53	8074	1615 (20.00)
Jena	37	915	235 (25.68)
Tomcat	31	3105	688 (22.16)
Camel	294	4663	1015 (21.77)
Xerces-J	30	1487	177 (11.90)
Batik	35	1164	213 (18.30)
Fop	30	2569	350 (13.62)

表 3 项目间依赖缺陷数据集的严重等级分布

严重等级	缺陷数
Blocker	933
Critical	406
Major	3658
Normal	2103
Minor	288
Trivial	44
Enhancement	425
缺陷总数	7857

此外, 我们随机选取了 7857 个非项目间依赖缺陷 (同等数目), 对比其与项目间依赖缺陷的修复复杂度的差异. 为了对比公平, 所选取的非项目间依赖缺陷的严重等级与表 3 对应的缺陷数据集的严重等级分布相同. 受文献 [3] 启发, 我们使用每个缺陷信息中开发者评论的总数以及修复的时间间隔 (从缺陷被报告当日到被修复当日的间隔天数) 来表征对应缺陷被修复的难易程度. 最终得到两种不同类型的缺陷修复情况如图 3 所示. 显然, 由于软件库的功能不透明、疏于维护等问题使得项目间依赖缺陷的修复难度更大, 耗费时间更长, 需要开发人员投入更多的测试和维护成本, 在修复周期上平均比非项目间依赖缺陷多花费 65 天. 由此可以证明该类问题的严重性.



(a) 开发者评论的总数 (b) 修复的时间间隔

图 3 缺陷修复的难易程度

综上所述, 有必要从客户代码与软件库依赖交互的视角, 来分析由于软件库的频繁使用而引入软件项目中的结构缺陷问题, 进而剖析不同来源软件之间依赖关系的多维质量属性.

© 中国科学院软件研究所 http://www.jos.org.cn

3 软件库依赖图谱的复杂性度量方法

3.1 软件边界图模型

本文从软件项目的拓扑结构出发, 构建软件边界图模型来刻画项目中不同来源软件代码的复杂异质结构. 设 $S = \{S_{\text{Client}}, Lib_1, Lib_2,\dots, Lib_{NB}\}$为任意软件项目, 其中 S_{Client} 为客户代码, Lib_i 为客户代码依赖的任意直接或间接依赖的软件库, NB 表示 S_{Client} 所依赖的软件库的总数. 令 CM_i, LM_{kt} 分别表示客户代码 S_{Client} 和软件库 Lib_k 中的任意方法, 则可以将软件项目表示为不同来源的方法集合, 即 $S = \{CM_i \mid i \in [1, NH]\} \cup \{LM_{kt} \mid k \in [1, NB], t \in [1, NL_k]\}$. 式中, NH 表示客户代码中的方法总数, NL_k 表示软件库 Lib_k 中的方法总数.

为刻画不同来源软件之间的依赖关系特性, 文献 [18] 将软件项目方法之间的调用关系同样划分出了两种类型: 客户代码中方法之间的调用以及客户软件方法和软件库方法之间的调用. 而本文重点关注于客户代码和软件库之间的方法调用关系, 以达到分析项目间依赖缺陷的目的. 为方便描述, 在此基础上我们引入了软件边界节点和软件边界图模型的概念. 将客户代码中直接调用软件库函数的方法节点视为软件项目中将不同来源的软件集成组装在一起工作的边界节点. 将与任意软件边界节点存在直接或间接调用关系的方法构建成与该边界节点相对应的软件边界图模型, 进而剖析项目之间的依赖结构. 上述两个定义的形式化描述如下.

定义 1. 软件边界节点. 设 CM_i 和 LM_{kt} 分别为客户代码 S_{Client} 和软件库 Lib_k 中的任意方法, 若软件项目中存在依赖关系$<CM_i, LM_{kt}>$, 即方法 CM_i 调用了方法 LM_{kt}, 则称方法 CM_i 为客户代码中直接与软件库相连接的软件边界节点.

定义 2. 软件边界图 (software boundary graph, SBG). 设方法 CM_i 为软件边界节点, VD_i 为软件库中直接被 CM_i 调用的方法集合, VH_i 为客户代码 S_{Client} 中直接或间接与 CM_i 存在调用关系的方法集合, VL_i 为软件库中与 CM_i 及 $M_t \in VD_i$ 存在直接或间接调用关系的方法集合, 若 $V_i = \{M_t | M_t \in \{VH_i \cup VL_i\}\}$, $E_i = \{<M_p, M_q> | M_p, M_q \in V_i, p \neq q\}$, 则我们定义有向依赖图 $G_i = (V_i, E_i)$ 为与软件边界节点 CM_i 相对应的软件边界图模型.

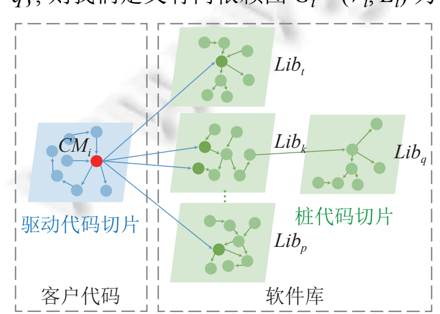


图 4 与软件边界节点 CM_i 对应的软件边界图模型

根据定义, 针对客户代码中的每一个边界节点, 我们都能构建出一个与之对应的软件边界图模型, 如图 4 所示, 该模型由客户代码中驱动该边界节点的代码切片 (driver) 以及软件库中对应于该边界节点的桩代码切片 (stub) 构成. 软件边界图模型可以认为是连通不同来源软件的媒介, 抽象出了客户代码与软件库交互的拓扑结构剖面. 借助于该模型可以进一步分析客户代码与软件库方法间依赖关系的复杂机制, 以及由于软件库的频繁使用而引发的多种质量风险问题.

3.2 软件库依赖图谱的复杂性度量指标

软件结构的复杂性本质上就是软件元素间连接/交互关系的复杂性, 且不同类型的软件系统对应着不同的典型连接模式[52]. Milo 等人[53]将这种元素互连的子图喻为构造系统网络的基本“砖块”, 是组成系统的基本单元. 研究表明, 系统子图的可靠性体现了软件微观结构的可靠性, 影响了软件的可维护性. Zimmermann 等人[17]更是通过构建“ego”网络子图来描述系统中任意一个方法节点与其邻居节点的连接关系. 他们认为“ego”网络子图能够反映出任意一个节点相比于其邻居节点的局部重要性, 进而针对该网络子图模型提出一系列度量指标 (包括“ego”网络子图的规模、节点对、中心性等) 对软件缺陷进行预测. 然而, 现有软件复杂性度量指标均未从不同来源的软件之间的依赖关系角度探讨其与客户代码与软件库结构缺陷的相关性[54].

受上述研究工作的启发, 基于软件边界图 (SBG) 模型, 我们提出一套风险度量指标以量化软件项目边界节点与软件库之间的接触面积、可达性、多源性、多源传递性、穿透性和枢纽性等多维质量属性, 利用其刻画不同来源的软件之间的耦合强度.

3.2.1　规模 (size)

软件边界图模型 G_i 的规模为模型中包含的方法节点总数. 该度量指标通过量化客户代码中边界节点 CM_i 对直接或间接依赖的软件库中方法的调用范围, 刻画出它在软件体系结构中所处的逻辑位置的复杂性.

3.2.2　接触面积 (touching-area)

为了避免软件开发过程中遗留技术债务, 在版本演化过程中软件库的版本会持续更新升级, 或者从一个软件库迁移到功能相似的其他软件库[55]. 而在软件库版本更新和迁移的过程中, 开发人员应当关注客户代码所直接调用的软件库中 API (application programming interface) 的兼容性和 API 使用正确性等问题[56]. 为量化软件库更新和迁移所带来的风险, 软件边界图模型 G_i 中, 我们将边界节点 CM_i 直接调用软件库中方法的总数定义为 CM_i 与软件库的接触面积. 边界图模型中客户代码与软件库的接触面积越大, 则引发上述问题的概率便会增大.

3.2.3　可达性 (reachability)

利用模块的扇入、扇出属性来计算结构的复杂度是软件工程的经典度量方法[57−59]. 然而, 节点的连接度并不能反映该节点对其他非直接关联节点的影响力及整个系统可靠性的贡献作用. 因此 Ma 等人[60]引入了节点可达范围的概念, 将软件模块直接或间接调用的所有节点视为其可达集. 由于可达集中的任意一个节点出现问题都会波及它, 经实验证明, 可达集的数值与对应模块的潜在出错概率具有较高的相关性.

本文利用上述思想来审视软件项目之间节点的可达性问题. 软件边界图模型 G_i 中, 我们将边界节点 CM_i 直接或间接调用软件库中方法的总数定义为边界节点 CM_i 的可达性. 若边界节点直接或间接依赖了越多的软件库中的方法, 那么软件库中的质量风险通过这种调用关系的“涟漪效应”引入到客户代码中的可能性便会随之剧增, 导致边界节点出现错误的概率也会急剧增加[2,3].

3.2.4　多源性 (multi-source)

不同软件库的质量不同, 维护活跃程度也存在差异. 一旦软件库存在安全漏洞、可靠性低或停止维护, 那么它们将会以较高的概率被替换掉[61,62]. 将边界节点 CM_i 直接或间接调用的软件库的个数记为该边界节点 CM_i 的多源性. 若边界节点直接或间接依赖越多来源不同的软件库, 则它的可靠性便会受越多不确定因素制约, 导致其可维护性越低、发生变更的可能性越大.

3.2.5　多源传递性 (multi-source transitivity)

在软件开发之初, 开发者通过边界节点直接调用软件库中的 API 来实现特定的功能, 然而直接依赖的软件库可能又借助了其他的软件库来协同工作[63]. 在这种情况下, 传递性依赖的软件库中的方法便会以“被动”的方式引入软件项目之中[64]. 由于开发者对传递性依赖的软件库的功能特征、质量属性及维护活动等信息并不熟知, 因而更容易被疏于维护而导致质量缺陷问题或是被注入恶意攻击程序而引发安全性问题[65].

若将软件库中与其他软件库存在直接调用关系的节点定义为多源传递节点, 在软件边界图模型 G_i 中, 将边界节点 HM_i 直接或间接调用多源传递节点的频次定义为 CM_i 的多源传递性, 那么多源传递性更高的边界节点引入可靠性风险的概率越大, 应该在软件生命周期中给予更高的维护关注度和有侧重性地投入测试成本.

3.2.6　穿透性 (penetrability)

在软件项目 S 中, 设 $\mathcal{L} = \{Lib_1, Lib_2, \ldots, Lib_{|\mathcal{L}|}\}$为客户代码 S_{Client} 直接依赖的软件库集合, 若 $Lib_i \in S$, 且 $Lib_i \notin \mathcal{L}$, 则我们称 Lib_i 为传递性依赖的软件库. 在软件边界图模型 G_i 中, 将边界节点 CM_i“穿透”集合 $\mathcal{L}$ 中的依赖关系而间接地调用传递性依赖的软件库中的方法总数, 称之为边界节点 CM_i 的穿透性. 穿透性指标与多源传递性指标分别从调用传递性依赖的软件库中方法的频次和规模互补地评估传递性依赖的软件库通过调用关系对软件质量造成的影响. 显然地, 穿透性越强的边界节点, 在维护过程中, 越应该引起开发人员的高度重视.

3.2.7　枢纽性 (hub degree)

近年来, 复杂网络理论中 K-核分解的度量方法[66,67]也被用来表征大规模软件系统的层次结构和模块的重要程度, 进而分析软件系统中存在的缺陷[68,69]. 本文利用 K-核分解指标分别度量出边界节点在客户代码中的枢纽程度以及被该边界节点直接调用的节点在软件库中的枢纽程度, 即在软件系统中所处拓扑位置的核心程度. 由于功

© 中国科学院软件研究所 http://www.jos.org.cn

能核心节点具有较大的波及范围, 并且同时需要与很多组件共同协作来实现功能[31]. 若枢纽程度高的边界节点与同样具有较高枢纽程度的软件库节点具有依赖关系, 那么软件库在版本迭代和质量维护过程中发生的变更, 便极容易对软件项目造成灾难性的影响.

设软件边界图模型 $G_i = GH_i \cup GL_i$, 其中 GH_i 为客户代码中与边界节点 CM_i 存在依赖关系的软件子图, GL_i 为软件库中与节点集合 VD_i 存在依赖关系的软件子图, VD_i 表示被边界节点 CM_i 直接调用的软件库方法集合. 为了度量软件边界节点在不同来源的软件之间的枢纽程度, 我们借助 K-核度量指标[70]引入如下定义.

定义 3. 边界 K-核. 边界图模型中, 客户代码子图 GH_i 的 K-核是指在 GH_i 的图形基础上反复去掉度值小于或等于 K 的节点及与其相连的边之后所剩余的子图, 而节点的核数表示包含该节点最深的核子图, 即节点存在于 K-核中, 但是它在 $(K+1)$-核中被移除, 则该节点的核数为 K. 同理可得软件库子图 GL_i 的边界 K-核.

根据上述定义, 设软件边界节点 CM_i 在客户代码子图 GH_i 中的核数为 LH_i, 软件库中被 CM_i 直接调用的任意方法 $M_t \in VD_i$ 在软件库子图 GL_i 中的核数为 LB_t, 则核数 LH_i 和 LB_t 分别代表边界节点 CM_i 和方法 $M_t \in VD_i$ 在各自软件系统中的枢纽程度. 利用公式 (1) 进一步对其进行归一化处理可得:

$$hub(HM_i) = LH_i / TL_h,\ \ hub(M_t) = LB_t / TL_l \tag{1}$$

其中, TL_h, TL_l 分别表示客户代码和软件库中节点的最大核数. 由于边界节点可能会直接调用多个软件库的节点, 本文定义边界节点 CM_i 在不同来源软件之间的枢纽程度为其自身在客户代码中的枢纽程度与它所直接调用的节点在软件库中的枢纽程度最大值之和:

$$HubDegree = hub(CM_i) + \max\{hub(M_t)\} \tag{2}$$

软件边界节点 CM_i 的枢纽性指标越高, 则说明源自不同软件的越靠近功能核心的节点之间存在直接的信息交互行为. 枢纽性高的边界节点在与枢纽性高的软件库节点发生协作和数据交互时, 承担着更多的工作职责. 与此同时, 由于软件库的演化而引入的不兼容性变更或缺陷问题对软件项目造成的影响也会更大. 因此, 枢纽性指标可以表征软件边界节点的脆弱性, 在维护过程中, 保证枢纽性高的节点正常工作, 并提高其鲁棒性应当作为软件质量保障的重要手段之一.

3.3 软件边界图模型风险度量指标的计算

以图 5 所示的代码片段为例, 说明其软件边界图模型的构建方式及本文定义的各项风险度量指标的计算方法. 图 5 的代码构成了一个软件项目, 它由客户代码中的类 *Client* 和分别来源于 3 个软件库的类 Lib_1、Lib_2 和 Lib_3 所组成. 客户代码中的方法 *Client.a*(*x*, *y*) 直接调用软件库中方法 $Lib_1.b(x, y)$、$Lib_2.b(x, y)$ 和 $Lib_2.c(x, y)$, 则根据本文定义, *Client.a*(*x*, *y*) 为客户代码中的边界节点. 如图 6 所示, 我们给出与边界节点 *Client.a*(*x*, *y*) 对应的软件边界图模型的构建方式, 图中的虚线圆圈表示定义 3 所描述的边界 K-核, 即方法所处的 K-核分解结构.

```
1: publicclassHost {
2:     publicinta(intx, inty) {
3:         Lib1 lib1= newLib1();
4:         Lib2 lib2= newLib2();
5:         intm= lib2.b(x, y) -lib2.c(x, y);
6:         returnlib1.b(x, y) + m;
7:     }
8:     publicintb(intx, inty) {
9:         intm= a(x, y);
10:        returnm* m;
11:    }
12     publicvoidc(intx) {
13:        intm= a(x, x) + b(x, x);
14:        System.out.println(m);
15:    }
16:    publicbooleand(intx, inty) {
17:        if(b(x, x) % b(y, y) == 0)
18:            return true;
19:        else
20:            return false;
21:    }
22: }
```

```
1: publicclassLib1 {
2:     privateinta(intx, inty) {
3:         if(x>= y)
4:           returnx;
5:         else
6:           returny;
7:     }
8:     publicintb(intx, inty) {
9:         intm= a(x, y);
10:        returnm* m;
11:    }
12     publicintc(intx, inty) {
13:        intm= b(x, y);
14:        if(m% y== 0)
15:            returny;
16:        else
17:            returnm% y;
18:    }
19:    publicvoidd(intx, inty) {
20:        System.out.println(b(x, y) +c(x,y));
21:    }
22: }
```

```
1: publicclassLib2 {
2:     publicintq;
3:     privateinta(intx, inty) {
4:         Lib3lib3= newLib3();
5:         returnlib3.c(x, y);
6:     }
7:     publicintb(intx, inty) {
8:         intz= a(x, y);
9:         returnz;
10:    }
11:    publicintc(intx, inty) {
12         Lib1 lib1= newLib1();
13:        intm= lib1.c(x, x);
14:        intn= b(x, y) + d(x);
15:        returnm+ n;
16:    }
17:    privateintd(intx) {
18:        returnx* q;
19:    }
20: }
```

```
1: publicclassLib3{
2:     privateinta(intx) {
3:         intm= x* x;
4:         returnm;
5:     }
6:     publicintb(intx, inty) {
7:         intz= d(x, y);
8:         returnz;
9:     }
10:    publicintc(intx, inty) {
11:        intm= a(x) + b(x, y);
12         returnm;
13:    }
14:    privateintd(intx, inty) {
15:        returne(x)+e(x * y);
16:    }
17:    public int e(inty) {
18:        returny* y;
19:    }
20: }
```

图 5　示例代码片段

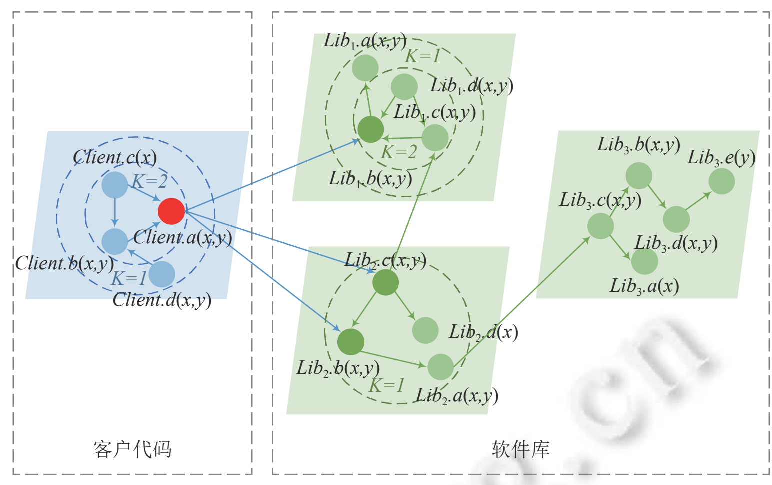
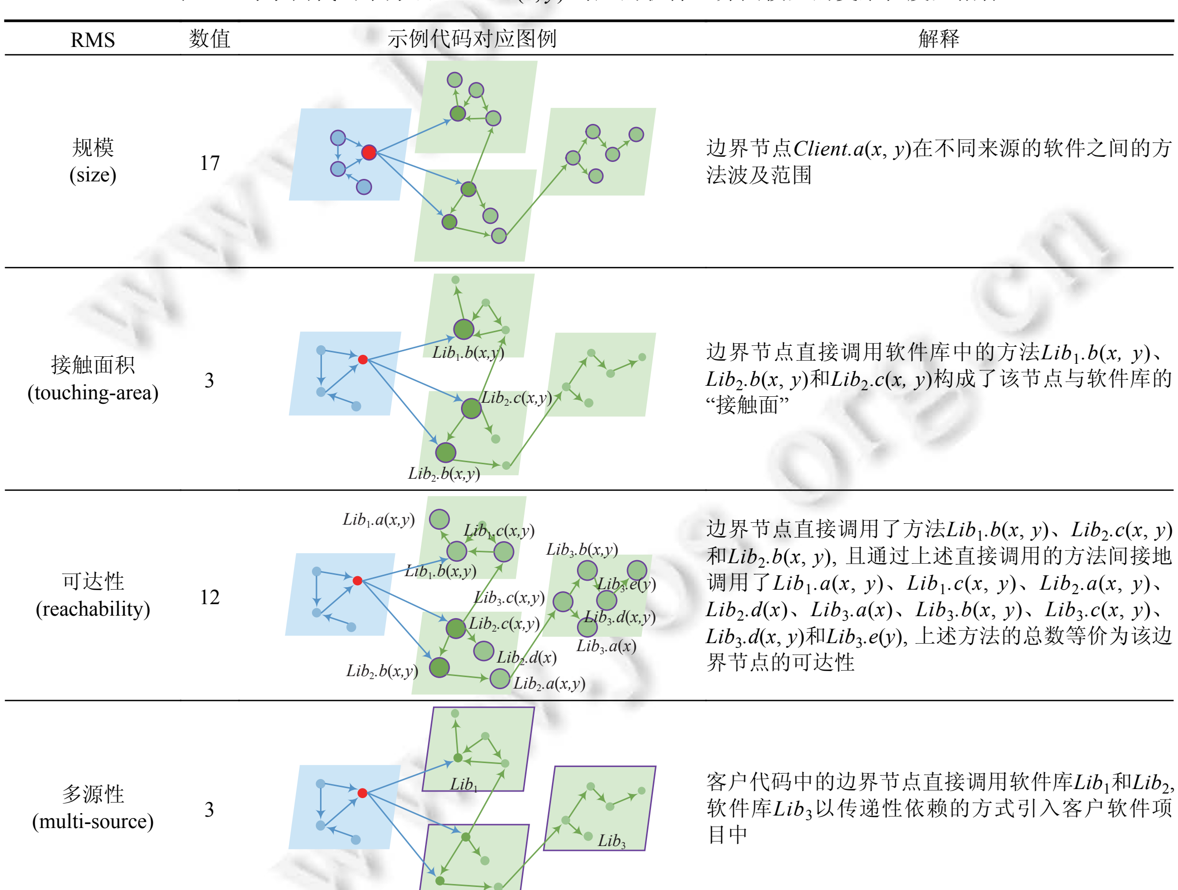


图 6 示例代码中与边界节点对应的软件边界模型

表 4 中的数据和图示解释了上述软件边界图模型耦合强度的各项风险度量指标的计算方式, 这些度量指标以互补的方式, 从不同的角度诠释了软件项目的多维质量属性.

表 4 与示例代码中方法 $Client.a(x, y)$ 对应的软件边界图模型的复杂性度量指标



RMS	数值	示例代码对应图例	解释
规模 (size)	17		边界节点 $Client.a(x, y)$ 在不同来源的软件之间的方法波及范围
接触面积 (touching-area)	3		边界节点直接调用软件库中的方法 $Lib_1.b(x, y)$、$Lib_2.b(x, y)$ 和 $Lib_2.c(x, y)$ 构成了该节点与软件库的“接触面”
可达性 (reachability)	12		边界节点直接调用了方法 $Lib_1.b(x, y)$、$Lib_2.c(x, y)$ 和 $Lib_2.b(x, y)$, 且通过上述直接调用的方法间接地调用了 $Lib_1.a(x, y)$、$Lib_1.c(x, y)$、$Lib_2.a(x, y)$、$Lib_2.d(x)$、$Lib_3.a(x)$、$Lib_3.b(x, y)$、$Lib_3.c(x, y)$、$Lib_3.d(x, y)$ 和 $Lib_3.e(y)$, 上述方法的总数等价为该边界节点的可达性
多源性 (multi-source)	3		客户代码中的边界节点直接调用软件库 Lib_1 和 Lib_2, 软件库 Lib_3 以传递性依赖的方式引入客户软件项目中

© 中国科学院软件研究所 http://www.jos.org.cn

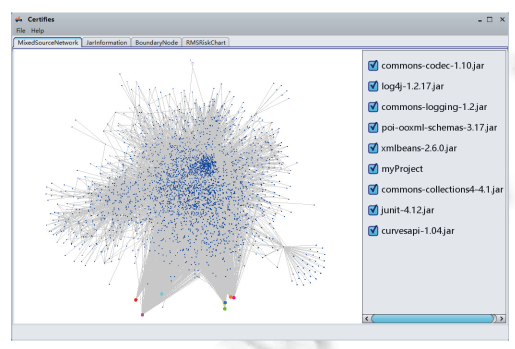
表 4 与示例代码中方法 *Client.a*(*x*, *y*) 对应的软件边界图模型的复杂性度量指标 (续)

RMS	数值	示例代码对应图例	解释
多源传递性 (multi-source transitivity)	2	$Lib_2.c(x,y)$ $Lib_2.a(x,y)$	软件库中的方法$Lib_2.a(x, y)$和$Lib_2.c(x, y)$分别与其他软件库存在直接调用关系, 即为多源传递节点. 边界节点直接或间接调用上述多源传递节点的总数视为它的多源传递性
穿透性 (penetrability)	5	Lib_1 $Lib_3.e(y)$ $Lib_3.b(x,y)$ $Lib_3.c(x,y)$ $Lib_3.d(x,y)$ $Lib_3.a(x)$ Lib_2	Lib_1和Lib_2为客户代码直接依赖的软件库, Lib_3是通过Lib_2传递性依赖的软件库, 故通过“穿透”Lib_2内部的方法之间的调用关系而间接地调用Lib_3中的方法总数, 可以刻画客户代码边界节点的穿透性
枢纽性 (hub degree)	2	K=2 K=2 K=1 K=1	根据边界K-核定义, 软件边界节点在客户代码子图中的核数为2, 软件库中被边界节点直接调用的方法在软件库子图中的核数分别为2、1、1. 由边界节点的枢纽性公式可得$HubDegree$ = 2/2+ max{2/2, 1/1, 1/1} = 2

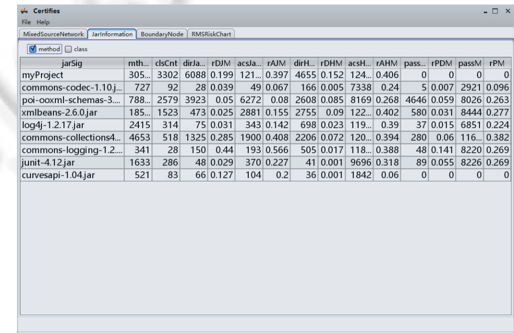
3.4 软件库依赖图谱的复杂性度量工具的实现

基于 SBG 模型所提出的 RMS 度量指标套件已经实现为自动度量风险的开源工具 Certifies (工具下载链接: https://github.com/NEUSoftwareEngineering/Certifies). Certifies 集成了 Soot 工具来解析 Java 工程项目的源代码, 以及 Prefuse 工具包 (http://prefuse.org/) 可将软件项目可视化为网络图的形式帮助开发人员直观地了解软件拓扑结构, 工具主要功能特征描述如下.

(1) Certifies 需要软件项目的依赖配置文件 (build.xml 或 pom.xml)、二进制 Jar 包文件和源代码文件作为输入信息: ① Certifies 借助于依赖配置文件能够解析软件库的依赖树结构, 针对支持 Ant 构建工具的依赖配置文件 build.xml, 基于 ApacheIvy 插件来解析软件库依赖树; 而针对支持主流的 Maven 构建工具的依赖配置文件 pom.xml, 基于 Dependency 插件来解析软件库依赖树. ② 扫描客户软件的源码文件可以得知客户代码与软件库的界限. ③ 利用 Soot 静态分析工具逐一扫描每个软件库可获取每个软件库方法节点集合, 用以区别软件库之间的边界. 工具能够输出如图 7(a) 所示的客户代码与软件库之间的依赖关系图.



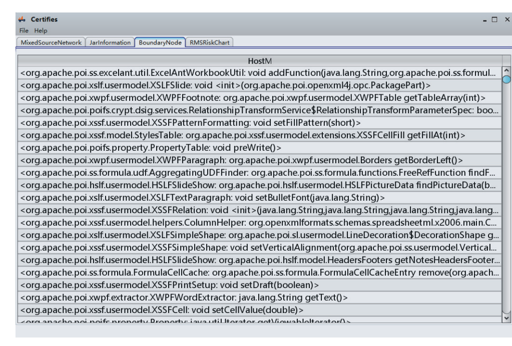
(a) 客户代码与软件库之间的依赖关系图



jarSig	mth...	clsCnt	dirJa...	rDJM	acsJa...	rAJM	dirH...	rDHM	acsH...	rAHM	pass...	rPDM	passM	rPM
myProject	305...	3302	6088	0.199	121...	0.397	4655	0.152	124...	0.406	0	0	0	0
commons-codec-1.10.j...	727	92	28	0.039	49	0.067	166	0.005	7338	0.24	5	0.007	2921	0.096
poi-ooxml-schemas-3....	788...	2579	3923	0.05	6272	0.08	2608	0.085	8169	0.268	4646	0.059	8026	0.263
xmlbeans-2.6.0.jar	185...	1523	473	0.025	2881	0.155	2755	0.09	122...	0.402	580	0.031	8444	0.277
log4j-1.2.17.jar	2415	314	75	0.031	343	0.142	698	0.023	119...	0.39	37	0.015	6851	0.224
commons-collections4...	4653	518	1325	0.285	1900	0.408	2206	0.072	120...	0.394	280	0.06	116...	0.382
commons-logging-1.2....	341	28	150	0.44	193	0.566	505	0.017	118...	0.388	48	0.141	8220	0.269
junit-4.12.jar	1633	286	48	0.029	370	0.227	41	0.001	9696	0.318	89	0.055	8226	0.269
curvesapi-1.04.jar	521	83	66	0.127	104	0.2	36	0.001	1842	0.06	0	0	0	0

(b) 客户代码及软件库调用关系的统计信息

图 7 Certifies 的主要功能



(c) 软件项目的边界节点

(d) RMS 度量指标分布图

图 7　Certifies 的主要功能 (续)

(2) 利用 Soot 静态分析工具对软件项目的二进制 Jar 包文件 (包含客户代码与所有依赖的软件库) 进行扫描分析, 获取软件项目类或方法粒度的调用图谱. 在此基础上, 统计出客户代码及软件库的类级或方法级的直接或间接调用关系信息, 例如客户代码和软件库中类或方法的总数、客户代码直接调用软件库的类或方法个数总和等, 如图 7(b) 所示, 软件项目的错综复杂的耦合关系以数值的形式显示, 供研究人员深入研究.

(3) 获取软件项目的边界节点, 以列表的形式显示详细信息, 如图 7(c) 所示.

(4) 针对软件中每个边界节点, 工具可自动计算其 RMS 度量指标, 图 7(d) 描述了客户代码中边界节点与软件库之间的耦合强度分布情况, 横坐标表示 RMS 度量指标, 纵坐标表示分布于阈值区间的边界节点个数.

4　数据分析

4.1　全局结构特性

边界节点将不同来源的软件集成连接在一起工作. 本文提出的 RMS 度量指标本质上刻画了软件项目的局部结构特性, 旨在剖析边界节点的多维质量属性. 在对真实的软件项目进行 RMS 度量指标可视化及分析之前, 我们首先从整体全局的视角审视了软件项目的复杂依赖机制.

本节对表 2 中的 10 个实验对象的所有版本进行分析, 根据 RMS 度量指标, 计算并统计出其全局结构特性. 如表 5 所示, 针对每个软件项目计算如下 4 种结构特性在各个版本中的平均值: 平均每个客户软件与软件库发生直接交互的类占客户代码中类总数的比例 (proportion of direct class, PDC) 为 93%; 客户代码中与软件库发生直接或间接交互的类占客户软件类总数的比例 (proportion of indirect class, PIC) 为 94.2%. 从方法层面上, 客户代码中直接调用软件库的方法占客户软件方法总数的比例 (proportion of direct method, PDM) 为 14.2%, 客户代码中直接或间接调用软件库的方法占客户软件方法总数的比例 (proportion of indirect method, PIM) 达到 37. 8%. 全局结构特性指标分布情况如图 8 所示.

表 5　全局结构特性

全局结构特性	描述
PDC	客户代码中直接依赖软件库的类占客户软件类总数的比例
PIC	客户代码中直接或间接依赖软件库的类占客户软件类总数的比例
PDM	客户代码中直接调用软件库的方法占客户软件方法总数的比例
PIM	客户代码中直接或间接调用软件库的方法占客户软件方法总数的比例

通过上述数据可以看出, 客户代码中绝大多数的类均与软件库存在直接或间接的依赖关系, 接近一半的方法直接或间接地调用了软件库中的方法来实现特定功能. 所选取的 10 个实验对象的 865 个版本均具备上述相似的

© 中国科学院软件研究所 http://www.jos.org.cn

结构特征, 表明客户代码与软件库之间普遍存在着极高的耦合强度. 如果软件库在版本更新或维护过程中出现问题, 那么客户代码与软件库间的强耦合关系会对软件项目的质量安全造成严重威胁.

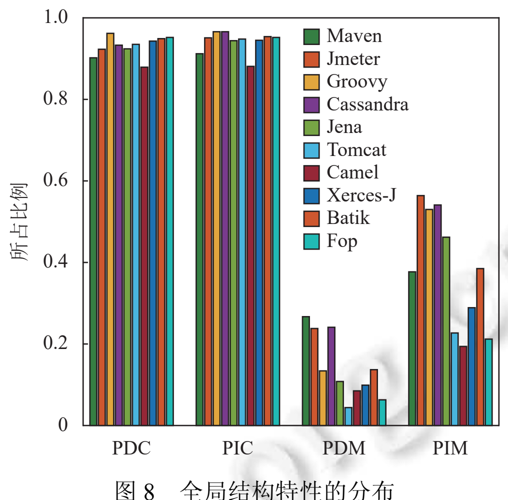


图 8 全局结构特性的分布

4.2 可视化 RMS 度量指标

本节选取表 2 中 10 个软件系统相应的最新版本作为实验对象, 量化边界节点的 RMS 度量指标, 以分析及呈现最新版本内的 RMS 度量指标数值分布情况. 结果表明以上软件项目中边界节点的 RMS 度量指标数值分布呈现一致性趋势. 以 Groovy-2.5.8 为例, 图 9 为其边界节点的 RMS 度量指标分布, X 轴表示边界节点的 RMS 度量指标, Y 轴表示 X 轴相应度量指标的边界节点个数.

实验结果表明, 软件项目中不同边界节点的 RMS 度量指标在数值上差异较大. 如图 9(a) 中软件边界图模型的规模 (size) 最高可达 21 103, 对应的边界节点为类 NodeChildren 中的方法 next(), 而相比之下, 类 XmlParser 中的方法 getEntityResolver() 所在的软件边界图模型的规模仅为 4. 由此可见不同边界节点在软件体系结构中所处的逻辑位置的复杂性差异悬殊. 图 9(b) 中边界节点的最大接触面积 (touching-area) 为 562, 与之对应的是边界节点为类 GrabAnnotationTransformation 中的方法 checkForConvenienceForm(), 该方法直接调用了大量软件库中的方法, 以实现自身复杂的功能. 在软件库版本升级时需考虑 API 的兼容性、正确性等问题, 一旦某些软件库出现安全和质量问题, 便会对客户软件项目造成严重的威胁.

软件项目中多数边界节点的 RMS 度量指标数值较大, 即对软件库存在较强的依赖程度. 从不同边界节点度量指标数值的分布来看, 如图 9 所示, 其相对大小大致服从 Pareto 分布, 即大约有 20% 的边界节点具有明显较高的耦合强度, 与其相比, 其余近 80% 的边界节点耦合强度在数值上存在较大的落差. Groovy-2.5.8 中共包含 3 737 个边界节点, 直接或间接依赖了 41 个软件库. 如图 9(a) 所示, 就规模 (size) 度量指标而言, 排名前 20% 的边界节点所构成的软件边界图模型的规模平均值为 13 475, 而其余 80% 的边界节点规模指标的平均值仅为 4 732. 说明软件项目中少数的高风险边界节点在软件体系结构中所处的逻辑位置的复杂程度相对较高.

类似地, 图 9(b) 接触面积、图 9(c) 可达性、图 9(d) 多源性、图 9(e) 多源传递性、图 9(f) 穿透性与图 9(g) 枢纽性相应边界节点度量指标排名在前 20% 的平均数值分别为 171、3 680、21、1 634、2 541 和 0.84, 而其余 80% 的边界节点度量指标的平均数值分别为 25、1 253、12、739、847 和 0.18. 这些高风险的边界节点直接或间接调用大量软件库中的方法实现其自身功能, 软件库中的风险常常会通过这种强依赖关系引入客户软件项目中. 特别地, 对于边界节点的穿透性指标, 开发人员常常直接调用软件库中的 API 实现某些特定功能属性, 而这些软件库可能又依赖其他软件库, 这些传递性依赖的软件库功能模块的演化及维护过程等信息对开发者是“封装”的, 不被开发人员所熟知. 因此, 经 RMS 度量指标评估, 穿透性强的高风险边界节点更应该引起开发人员的高度重视.

© 中国科学院软件研究所 http://www.jos.org.cn

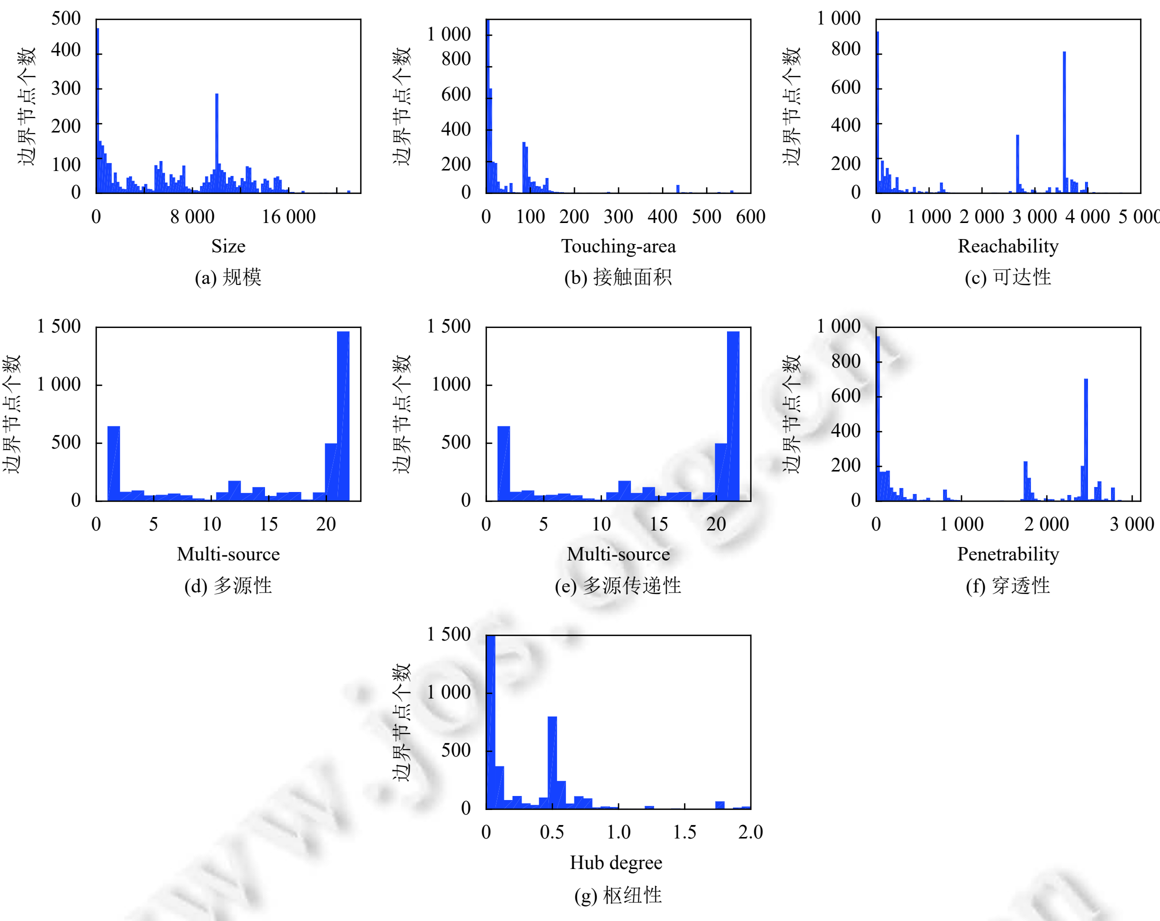


图 9　Groovy-2.5.8 中边界节点 RMS 度量指标的分布图

通过挖掘真实的 bug 案例, 我们发现经 RMS 风险度量识别到的高风险的边界节点, 的确会对客户软件项目的质量维护造成一定的威胁. 例如软件 Groovy-2.5.8 中边界节点 AntlrParserPlugin.declarationExpression() 直接或间接调用了大量软件库中的方法, 其可达性 (reachability) 高达 3 572. 在调查中发现, 由于它直接或间接可达的方法数目过多, 该方法需要频繁应对软件库中的方法在版本迭代中的变更. 经统计, 该边界节点被开发人员累计修改的次数多达 36 次. 在近一年的演化过程中, 由于其依赖的软件库 Transform, Script, AST 和 Antlr 中方法的异常行为而累计 6 次引入项目间依赖缺陷#4228, #3768, #3481, #2845, #2604 和#2007.

类似地, 类 GroovyShell 中的 main() 方法与软件库的接触面积为 138, 对应边界图模型的规模高达 7 327, 可达性为 2 715, 多源性数值为 20, 多源传递性为 1 545, 穿透性是 1 814, 枢纽性更是高达 0.619. main() 方法作为 Java 应用程序的入口, 接触面积即为客户代码与软件库的衔接之处, 其数值过大破坏了 Java 语言的封装特性. 良好的封装设计能够减少模块之间的耦合性, 自由地修改类内部的结构, 同时可以对成员变量进行更精确的控制, 并且能达到隐藏信息、实现细节的目的. 经过统计其累计修改代码的次数为 30 次, 被故障函数波及的频次为 44. 以其自身引入 ID 为#3910 的缺陷为例, 由于错误地使用软件库 Mockito.answer() 方法, 导致抛出 GroovyRuntimeException 异常, 从而无法正常运行脚本或者类, 最终严重影响到客户软件项目的可靠性. 因此在软件库使用及版本升级的过程中, 开发人员应当关注客户代码直接调用软件库 API 的兼容性、正确性和参数配置等问题, 从而确保软件项目的质量.

根据以上对真实软件项目边界节点的 RMS 度量指标数据的可视化分析与讨论, 我们将给定的客户软件项目中度量指标排名前 20% 的边界节点视为该系统中的高风险边界节点. 本文选取表 1 所示的 10 个软件项目所对应

© 中国科学院软件研究所 http://www.jos.org.cn

的 865 个版本作为实验对象, 做进一步分析. 如表 6 所示, 由于高风险的边界节点而引入的项目间依赖缺陷数目的平均分布比例. 表格属性分别为 RMS 度量指标和 W = 平均每个软件系统中引入项目间依赖缺陷的高风险边界节点占系统中引入项目间依赖缺陷的边界节点的百分比. 例如, 表 6 中度量指标为规模时, $W=(W_1+W_2+\ldots+W_{865})/865=(74.4\%+80.4\%+\ldots+72.9\%)/865=76.63\%$, 其中 W_n 表示第 n 个软件项目的上述比例值. 软件中多数 (近 80%) 的项目间依赖缺陷由少数 (近 20%) 的高风险边界节点所引入. 因此, 在产生项目间依赖缺陷之时, 借助于 RMS 度量指标能够帮助开发人员高效地定位和排查高风险的边界节点, 同时通过量化软件模块的风险因子, 可以指导开发人员在软件开发过程中合理地分配维护和测试成本.

表 6 高风险的边界节点引入的项目间依赖缺陷所占比例

度量指标	W (%)
规模	76.63
接触面积	60.55
可达性	80.00
多源性	78.25
多源传递性	84.35
穿透性	78.83
枢纽性	54.61

5 实证调查研究

本文准确地划分出客户代码与软件库的代码边界, 解析出边界节点连接客户代码与软件库的代码切片, 在此基础上通过分析该代码切片的拓扑结构, 利用 RMS 度量指标来量化边界节点的风险因子. 为了评估所提出的 RMS 度量指标的有效性, 本节中我们利用假设验证法[17,19,71]来讨论边界节点的风险因子与项目间依赖缺陷的相关性和影响程度. 首先, 提出如下 3 个假设.

- H1 风险因子更高的边界节点更容易引入更多数量的项目间依赖缺陷.
- H2 风险因子更高的边界节点更容易引入严重等级高的项目间依赖缺陷.
- H3 RMS 度量指标数值对项目间依赖缺陷数量和严重等级具有较强的影响.

基于 SBG 模型的 RMS 度量指标套件由于考虑了客户代码与软件库的依赖关系, 为软件项目耦合强度的度量及项目间依赖缺陷的研究提供了重要依据. 为了评估 RMS 度量指标对软件项目复杂结构的质量度量及项目间依赖缺陷预测的适用性, 我们选取如表 7 和表 8 所示的传统的复杂性度量指标, 在验证 H1, H2 和 H3 的正确性时同 RMS 度量指标进行对比分析. 上述对比的传统复杂性度量指标选取的原则是: (1) 方法级别的复杂性度量指标. 由于 RMS 度量指标套件用于量化方法级软件边界节点的风险因子, 为了便于对比, 我们均选择方法级别的复杂性度量指标. (2) 分别从程序的复杂性度量和软件拓扑结构复杂性度量两个角度进行选择. 由于 RMS 度量指标套件设计的初衷是从软件项目的拓扑结构视角出发, 结合多种质量要素进行度量系统耦合强度的风险性. 为了综合评估 RMS 度量指标的有效性, 我们从两个不同的视角选择进行对比的传统复杂性度量指标: 程序的复杂性和软件拓扑结构的复杂性. (3) 选取的度量指标均已在多个研究文献[17,19,34,71−73]中证明了其与代码缺陷具有一定的相关性, 且具有较强的故障预测能力.

表 7 传统的复杂性度量指标 (程序复杂性)

复杂性度量指标	描述
代码行[17,34,71]	边界节点所在函数内部的代码行数
参数[17,71]	边界节点所在函数的参数个数
圈复杂度[17,19,71]	边界节点所在函数内部的逻辑路径条数
程序长度[71]	边界节点所在函数中出现的所有操作数和操作符的数量之和
程序容量[71]	程序长度*$\log_2$(边界节点所在函数中出现的不同操作符与操作数的数量之和)

此外, 我们选取文献 [19] 中提出的度量指标与 RMS 度量指标进行对比分析. 如表 9 所示, 文献 [19] 同样通过建立软件拓扑图来分析软件系统演化的过程及预测缺陷, 其出发点与本文类似, 但未从客户代码和软件库的依赖关系审视软件系统结构的复杂性.

表 8　传统的复杂性度量指标 (软件拓扑结构复杂性)

拓扑结构度量指标	描述
入度[17,19,34,71]	从其他节点指向边界节点的边的数目
出度[17,19,34,71]	从边界节点指向其他节点的边的数目
聚集系数[19]	边界节点的邻节点之间实际存在的边数与总的可能的边数之比
介数[73]	所有最短路径中经过边界节点的路径数目占最短路径总数的比例
接近数[73]	边界节点到所有节点的距离平均值的倒数

表 9　文献 [19] 复杂性度量指标

度量指标	描述
编辑距离	边界节点所在拓扑图中顶点和边的数量在版本间的变更个数
节点等级	边界节点的所有入度节点所占权重与该节点出度之比的总和
直径	边界节点所在拓扑图中的直径长度
平均度	边界节点所在拓扑图中边总数的2倍占节点总数的比例
模块比例	边界节点所在模块内调用的函数总数与边界节点所在模块间调用的函数总数之比

5.1　研究框架

本文通过挖掘分析软件历史仓库, 从中抽取出引入项目间依赖缺陷的软件边界节点程序模块, 度量其边界节点的质量属性, 在此基础上, 构建缺陷预测模型来预测边界节点引入的项目间依赖缺陷数量及严重等级. 其中软件历史仓库包括项目所处的版本控制系统 (CVS, SVN 或 GitHub 等) 和缺陷跟踪系统 (Bugzilla 或 Jira). 研究框架如图 10 所示.

步骤 1. 数据收集过程. 首先, 基于第 3 节已收集好的如表 2 所示的项目间依赖缺陷数据集, 进一步在缺陷跟踪系统上挖掘出上述数据集中每个缺陷对应的报告信息 (包括缺陷严重等级, 代码修改日志, 修复补丁, 开发人员的相关讨论内容等). 然后, 根据代码修改日志在版本控制系统中抽取相关修改的函数模块, 识别出其中导致该项目间依赖缺陷的客户代码中的边界节点, 或是引入缺陷的软件库中方法节点. 在只识别出引入缺陷的软件库中方法节点的情况下, 在该缺陷软件版本中结合 RMS 的“可达性”度量指标, 进一步定位到通过调用路径可达该软件库中问题方法的客户代码边界节点. 第三, 根据 RMS 度量指标和传统复杂性度量及文献 [19] 所提出的度量指标 (即基准指标) 对上述边界节点的复杂性数值进行量化. 需要说明的是, 我们对表 2 中的 10 个开源软件项目的所有版本进行分析, 计算并统计出每个版本中所有边界节点的 RMS 度量指标及基准指标. 进而求得每个边界节点的 RMS 度量指标及基准指标在各个版本中的平均值, 用于假设验证分析过程.

步骤 2. 假设验证过程. (1) 验证 H1 和 H2: 分析边界节点的 RMS 度量指标 (每个项目所有版本的平均值) 与其引入的项目间依赖缺陷数量及严重等级的相关性, 并与传统复杂性度量及文献 [19] 所提出的度量指标进行对比. 缺陷管理员根据缺陷引起的故障对软件程序的影响程度为其分配严重等级, 表 10 描述了 Bugzilla 缺陷跟踪系统中缺陷严重等级及其对应的排序. 由于边界节点可能在不同软件项目版本中引入多个项目间依赖缺陷的情况, 我们统计每个边界节点在所有版本中引入项目间缺陷的总数, 取其对应缺陷严重等级的平均值作为该边界节点引入缺陷的严重等级. (2) 验证 H3: 以边界节点为单位创建缺陷数据集, 即数据属性为每个边界节点引入的项目间依赖缺陷的数量及严重等级. 使用回归分析技术构建缺陷模型, 分析边界节点的 RMS 度量指标数值多大程度地影响了其引入项目间依赖缺陷的数量及严重等级.

© 中国科学院软件研究所 http://www.jos.org.cn

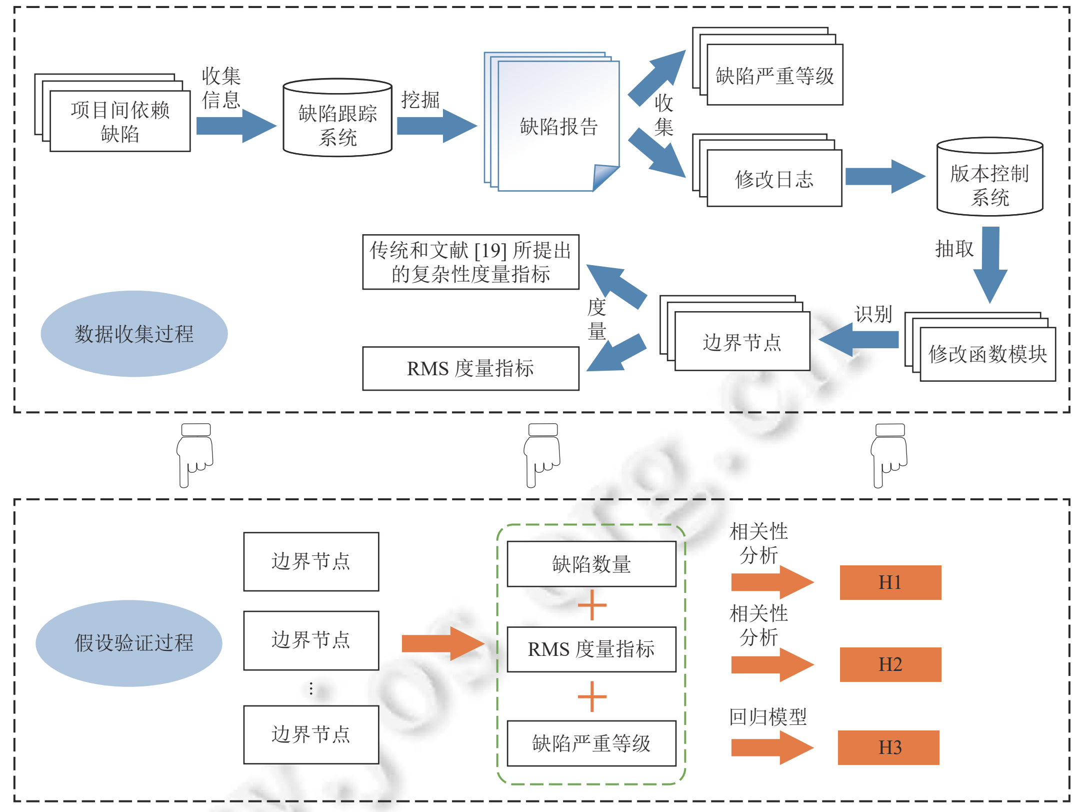


图 10 研究框架图示

表 10 Bugzilla 中缺陷严重等级

缺陷严重等级	描述	排序
Blocker	阻碍开发或测试工作	7
Critical	死机、数据丢失、内存溢出	6
Major	较大的功能缺陷	5
Normal	普通的功能缺陷	4
Minor	较轻的功能缺陷	3
Trivial	产品外观上的问题	2
Enhancement	建议或意见	1

5.2 H1: 风险因子更高的边界节点更容易引入更多数量的项目间依赖缺陷

为了量化边界节点风险因子值与引入项目间依赖缺陷数量的相关性, 本文采用相关系数客观地反映两组向量之间的关联程度. 常用的相关系数有 3 种: (1) Pearson 相关系数[74]适用于有线性关系的数据, 且整体满足正态分布连续变量; (2) Kullback-Leibler 散度[75]是用来度量两个分类变量相关程度的指标, 它能够针对两个有序变量进行分类, 并可以进行非参数的相关度量; (3) Spearman 相关系数[76]对初始数据没有强制性要求, 即数据之间不需要符合任意分布规律, 就能够通过相关系有效地数量化彼此的相关程度. 由于软件项目中边界节点的风险因子与其引入项目间依赖缺陷数量两个维度的数据分布不一定符合正态分布特征, 更不属于两组分类变量, 因此, Spearman 相关系数更适用于进行本文的相关性分析场景. 根据 Spearman 相关系数定义, 相关性 $\rho=\sum(x_i-\bar{x})(y_i-\bar{y})/\left(\sum(x_i-\bar{x})^2\sum(y_i-\bar{y})^2\right)^{1/2}$, 其中我们令 x_i 为边界节点度量量化的风险因子值, y_i 为引入项目间依赖缺陷的数量.

为了便于分析, 实验中我们将其与传统复杂性度量指标和文献 [19] 所提出的度量指标对比. 需要说明的是, 相关系数越接近−1 或+1, 则两个变量间相关性越高, 其中+1 为正相关, −1 为负相关, 0 表示两个度量是独立的. 同时, 若相关系数分布在 0.7–1.0, 则视为两项指标具有强相关性, 0.3–0.7 为中等相关, 0–0.3 为弱相关[77]. 表 11、表 12 和表 13 中显示了 Spearman 等级相关系数结果, 均在 0.01 级别显著相关.

表 11 RMS 度量指标与缺陷数量的相关性

指标	Maven	Jmeter	Groovy	Cassandra	Jena	Tomcat	Camel	Xerces-J	Batik	Fop
规模	0.319	0.493	0.580	0.530	0.368	0.382	0.509	0.367	0.362	0.370
接触面积	0.371	0.354	0.257	0.318	0.428	0.331	0.321	0.443	0.445	0.429
可达性	0.303	0.497	0.352	0.356	0.459	0.241	0.356	0.451	0.453	0.454
多源性	0.218	0.498	0.252	0.290	0.341	0.294	0.290	0.351	0.354	0.338
多源传递性	0.274	0.439	0.272	0.375	0.435	0.367	0.373	0.436	0.439	0.436
穿透性	0.315	0.497	0.360	0.360	0.470	0.316	0.360	0.463	0.462	0.466
枢纽性	0.238	0.352	0.232	0.306	0.283	0.236	0.311	0.274	0.319	0.272

表 12 传统复杂性度量指标与缺陷数量的相关性

指标	Maven	Jmeter	Groovy	Cassandra	Jena	Tomcat	Camel	Xerces-J	Batik	Fop
代码行	0.082	0.408	0.398	0.274	0.399	0.264	0.270	0.406	0.420	0.393
参数	−0.139	−0.014	−0.084	0.020	0.219	0.055	0.032	0.219	0.234	0.230
圈复杂度	0.094	0.383	0.387	0.089	0.424	0.095	0.072	0.450	0.452	0.439
程序长度	−0.171	0.150	0.003	0.037	0.116	−0.047	0.048	0.120	0.139	0.123
程序容量	−0.107	0.146	−0.012	0.103	0.106	0.007	0.115	0.109	0.133	0.111
入度	−0.053	0.196	−0.080	0.178	0.166	0.125	0.157	0.147	0.160	0.150
出度	0.259	0.058	0.298	0.296	0.448	0.292	0.293	0.462	0.467	0.448
聚集系数	0.031	0.029	0.119	−0.045	0.175	−0.157	−0.035	0.204	0.171	0.182
介数	0.010	0.231	0.059	0.331	0.320	0.229	0.324	0.320	0.325	0.341
接近数	−0.150	0.090	−0.144	−0.128	−0.075	−0.072	−0.123	−0.049	−0.066	−0.060

表 13 文献 [19] 提出的复杂性度量指标与缺陷数量的相关性

指标	Maven	Jmeter	Groovy	Cassandra	Jena	Tomcat	Camel	Xerces-J	Batik	Fop
编辑距离	0.034	−0.089	−0.007	0.012	−0.016	−0.075	−0.081	0.012	−0.051	0.087
节点等级	−0.050	0.081	−0.041	−0.015	−0.145	−0.042	−0.107	−0.010	0.005	−0.014
直径	−0.012	0.031	−0.013	0.012	−0.083	−0.103	−0.017	0.036	0.085	0.003
平均度	−0.093	0.065	0.012	0.026	−0.012	−0.056	0.011	−0.022	0.103	−0.065
模块比例	−0.082	0.114	−0.113	0.011	0.032	−0.059	−0. 078	0. 013	−0.009	−0.010

从整体上分析表 11 中的数据, 与传统复杂性度量指标相比, RMS 度量指标与项目间依赖缺陷数量存在着更高的相关性, 平均每个软件项目中边界节点的 RMS 度量指标与项目间依赖缺陷数量的相关性达到 0.3 至 0.4 水平. 同时表 13 中的数值处于 0.01 级别, 即文献 [19] 提出的复杂性度量指标与项目间依赖缺陷数量不存在相关性. 实验结果表明 RMS 度量指标套件对项目间依赖缺陷研究的适用性与重要性. RMS 度量指标与项目间依赖缺陷数量存在较强的相关性, 其结果证实了我们提出的假设 1 (H1), 即风险因子更高的边界节点会更容易引入更多数量的项目间依赖缺陷.

从表 12 中数据可以得到如下观察结果. 部分传统复杂性度量指标与项目间依赖缺陷数量也存在着较高的相关性 (如代码行、圈复杂度、出度和介数), 即 0.3 至 0.4 左右, 但是这个现象并没有在所有的实验对象上呈现出普遍性. 例如, 边界节点所在函数内部的代码行数越多, 说明函数承担的职责越多, 代码容量越大, 故其调用软件库的

© 中国科学院软件研究所 http://www.jos.org.cn

可能性就越大. 同时, 过大的函数具有较差的可理解性、灵活性和可维护性, 从而使得其代码出现错误的风险性增大. 因此代码行数这个指标在偶然情况下, 也能够间接地反应出与软件库的依赖强度及与项目间依赖缺陷数量的相关性. 例如, 软件 Jmeter、Groovy、Jena、Xerces-J、Batik 和 Fop 中边界节点所在方法内部的代码行数与其引入的项目间依赖缺陷数量的相关性普遍偏高, 达到 0.4 左右. 而 Maven 软件中边界节点所在方法内部的代码行数与引入项目间依赖缺陷数量的相关性仅为 0.082. 然而我们提出的 7 种风险度量指标在不同的软件上均得到相似的规律, 其有效性得到了验证.

5.3 H2: 风险因子更高的边界节点更容易引入严重等级高的项目间依赖缺陷

我们通过计算边界节点的风险因子值与引入项目间依赖缺陷严重等级的 Spearman 相关系数, 对假设 2 (H2) 进行验证. 在实验过程中, 与传统复杂性度量指标和文献 [19] 提出的复杂性度量指标进行对比, 所得相关性数值如表 14、表 15 和表 16 所示, 均在 0. 01 级别显著相关.

表 14 RMS 度量指标与缺陷严重等级的相关性

指标	Maven	Jmeter	Groovy	Cassandra	Jena	Tomcat	Camel	Xerces-J	Batik	Fop
规模	0.475	0.493	0.524	0.521	0.541	0.525	0.497	0.539	0.545	0.566
接触面积	0.325	0.346	0.239	0.193	0.110	0.163	0.205	0.111	0.112	0.128
可达性	0.132	0.113	0.117	0.313	0.101	0.182	0.313	0.104	0.112	0.115
多源性	0.046	0.126	0.091	0.287	0.132	0.152	0.287	0.126	0.131	0.164
多源传递性	0.073	0.008	0.014	0.318	0.148	0.199	0.314	0.150	0.151	0.151
穿透性	0.137	0.122	0.115	0.314	0.129	0.176	0.314	0.131	0.137	0.146
枢纽性	0.463	0.472	0.525	0.542	0.504	0.494	0.547	0.505	0.603	0.519

表 15 传统复杂性度量指标与缺陷严重等级的相关性

指标	Maven	Jmeter	Groovy	Cassandra	Jena	Tomcat	Camel	Xerces-J	Batik	Fop
代码行	0.177	0.203	0.135	0.024	−0.020	0.032	0.019	−0.022	−0.022	−0.009
参数	−0.043	0.222	−0.091	0.251	0.017	0.183	0.262	0.017	0.016	−0.007
圈复杂度	0.143	0.225	0.188	0.054	0.123	0.055	0.033	0.118	0.123	0.114
程序长度	−0.001	−0.099	0.013	−0.087	0.062	−0.009	−0.073	0.063	0.060	0.076
程序容量	−0.032	−0.103	−0.013	−0.104	0.094	−0.027	−0.093	0.094	0.091	0.111
入度	0.051	0.052	0.097	−0.070	−0.144	−0.059	−0.088	−0.109	−0.112	−0.085
出度	0.262	−0.187	0.175	0.165	0.089	0.104	0.166	0.091	0.090	0.109
聚集系数	0.027	−0.141	0.075	−0.148	−0.099	−0.106	−0.139	−0.109	−0.098	−0.089
介数	−0.032	0.053	0.005	0.139	0.076	0.137	0.126	0.073	0.078	0.058
接近数	0.001	0.008	0.069	−0.069	0.037	−0.052	−0.068	0.033	0.035	0.020

表 16 文献 [19] 提出的复杂性度量指标与缺陷严重等级的相关性

指标	Maven	Jmeter	Groovy	Cassandra	Jena	Tomcat	Camel	Xerces-J	Batik	Fop
编辑距离	0. 158	−0. 007	0. 068	0. 045	−0. 013	0. 016	−0. 085	0. 077	0. 004	−0. 001
节点等级	−0. 093	0. 093	−0. 133	−0. 024	0. 060	−0. 050	−0. 126	0. 027	−0. 015	−0. 013
直径	−0. 091	0. 011	−0. 086	0. 026	0. 087	0. 056	−0. 002	−0. 015	−0. 073	0. 045
平均度	−0. 157	−0. 022	−0. 088	0. 018	0. 052	−0. 014	0. 054	−0. 018	0. 007	0. 061
模块比例	−0. 076	0. 016	−0. 058	0. 006	0. 032	0. 153	−0. 039	0. 012	0. 061	0. 012

具体分析表 14 中的数据, 在所有的实验对象中, RMS 度量指标中的规模和枢纽性指标与项目间依赖缺陷严重等级均存在着较强的相关性. 平均每个软件项目中的边界节点所在边界图模型的规模和枢纽性与缺陷严重等级的相关性分别达到了 0.523 和 0.517. 软件边界图模型的规模为边界节点在不同来源软件之间方法调用关系的波

© 中国科学院软件研究所 http://www.jos.org.cn

及范围, 规模数值大说明边界节点在软件项目中所处的逻辑位置较为复杂. 进一步来讲, 若其对应的软件边界图模型中驱动代码切片涉及的方法数越多, 则由该边界节点引发的项目间依赖缺陷对客户代码的影响范围和程度就越大. 而缺陷严重等级是根据缺陷对客户代码的影响程度进行分配的, 因此具备上述特征的边界节点导致的项目间依赖缺陷具备更高的严重等级. 边界节点的枢纽性反映了不同来源的软件中靠近功能核心的节点之间存在直接的信息交互行为. 枢纽性越高, 说明越靠近客户代码功能核心的边界节点直接地与靠近软件库功能核心的节点发生了信息交互. 这种情况下, 该边界节点对应的项目间依赖缺陷对客户代码的影响程度更大, 导致该缺陷具有更高的严重等级. 而 RMS 度量指标中的接触面积、可达性、多源性、多源传递性和穿透性与项目间依赖缺陷严重等级的相关性相对较弱, 即数值在 0.15 上下浮动.

从表 14–表 16 中数据可以看出, 表 14 有 27 个高达 0.3 以上的数值, 而表 15 和表 16 则分别有 65、47 个仅为 0.01 级别的数值. 从整体上分析, RMS 度量指标与项目间依赖缺陷严重等级存在相关性, 而传统、非传统复杂性度量指标与缺陷严重等级不存在相关性. 这充分说明了 RMS 度量指标对项目间依赖缺陷严重等级预测的有效性.

综合上述分析, 我们将软件边界图模型的规模和边界节点的枢纽性定义为项目间依赖缺陷严重等级的强相关风险因子, 其相关性分析结果证实了提出的假设 2 (H2), 对应的软件边界图模型的规模和枢纽值越高的边界节点会更容易引入严重等级高的项目间依赖缺陷.

5.4 H3: RMS 度量指标数值对项目间依赖缺陷数量和严重等级具有较强的影响

根据第 5.2 节、第 5.3 节对 RMS 度量指标与项目间依赖缺陷数量及严重等级相关性的分析, 为验证假设 3 (H3), 本节通过建立多元线性回归模型来探究 RMS 度量指标数值对引入项目间依赖缺陷的数量与严重等级造成“多大程度”的影响.

根据文献 [78,79] 中所描述的统计学中相关分析与回归分析的基本原理, 如果自变量与因变量之间没有相关性, 则没有必要再进行回归分析. 如果自变量与因变量之间存在相关性, 则需要通过回归分析进一步验证其中的关系. 因假设 1 和假设 2 中文献 [19] 提出的复杂性度量指标与项目间依赖缺陷的数量、严重等级均不存在相关性, 即该度量指标对项目间依赖缺陷数量和严重等级没有影响性. 在实验过程中, 我们采用与文献 [17,34] 中相同的实验方法, 即对每个软件项目分别进行 3 组回归分析, 因变量为项目间依赖缺陷数量, 自变量分别为 RMS 度量指标、传统复杂性度量指标、RMS 度量指标与传统复杂性度量指标的结合, 以此来对比 3 种度量指标对项目间依赖缺陷数量和严重等级的影响程度. 其中传统复杂性度量指标选取代码行、圈复杂度、出度和介数, 因为在假设 1 (H1) 中已经得到证实, 以上传统复杂性度量指标与项目间依赖缺陷数量存在着较高的相关性, 因而其对项目间依赖缺陷数量和严重等级影响性的分析有一定研究价值. 在分析项目间依赖缺陷严重等级影响性的研究中, 选取假设 2 (H2) 中得到的项目间依赖缺陷严重等级的强相关风险因子 (即 RMS 度量指标中的规模和枢纽性) 作为回归模型的自变量, 项目间依赖缺陷严重等级为回归模型的因变量.

我们的实验均遵循着 Nagappan 等人[70]提及的方法学, 包括以下过程: 主成分分析[80], 建立回归模型, 评估解释能力和评估影响程度的能力.

5.4.1 主成分分析

主成分分析是一种处理自变量间多重共线性的标准数据统计方法, 通过正交变换将一组可能存在相关性的变量转换为一组不相关的变量, 转换后的这组变量称为主成分[17,34]. 由于自变量间的共线性, 导致因变量在评估过程中出现较大差异, 因此在建立回归模型时不使用实际的自变量 (RMS 度量指标和传统复杂性度量指标), 而选择主成分作为自变量建立回归模型. 在实验中, 选取累计方差贡献率达到 95% 以上的主成分.

5.4.2 建立回归模型

为了保证回归模型的评估性能, 我们采取交叉验证法[81]. 根据文献 [17,34] 的实验方案, 将样本数据进行分组, 随机选取 2/3 的数据作为训练集来构建模型, 剩余 1/3 的数据作为测试集来评价模型. 为了减少交叉验证结果的可变性, 对样本数据集进行 50 次不同的随机分组, 从而实现多次交叉验证, 以确保回归模型的可靠稳定性.

© 中国科学院软件研究所 http://www.jos.org.cn

5.4.3 评估解释能力

我们选取以下指标评估回归模型的解释能力: (1) 决定系数 R^2 反映自变量对因变量变动的解释程度. 决定系数取值范围为 0 到 1, 数值越大意味着自变量对因变量的解释程度越高, 模型有着更好的解释能力. (2) 随着自变量的增多, 决定系数会越来越接近于 1, 这将导致模型的稳定性变差, 即模型在预测训练集之外的数据时, 预测波动将会变得非常大. 为了消除自变量数量的影响, 引入了调整的 R^2, 即校正决定系数, 其数值一般小于决定系数, 可以更好地反映模型的质量. 决定系数和校正决定系数均可以反映模型拟合优度. (3) 对回归模型进行显著性检验, 即 F 检验. 在我们的实验中, 所有回归模型显著性在 99% 的水平上 ($p<0.01$).

5.4.4 评估影响程度

计算项目间依赖缺陷数量及严重等级的评估值与实际值之间的 Spearman 等级相关系数以探究 RMS 指标对项目间依赖缺陷数量及严重等级的影响程度, 相关性接近+1 或–1 是最理想的.

图 11 展示了软件 Groovy 对项目间依赖缺陷数量进行预测的 3 组实验结果, 每组实验实现 50 次交叉验证的决定系数、校正决定系数和 Spearman 等级相关系数分布趋势整体上保持一致, 说明交叉验证对构建稳定可靠模型的必要性. 为了方便对 3 组实验评估结果分析与讨论, 以下我们将 RMS 度量指标为自变量的回归模型简称为 RMS 回归模型, 传统复杂性度量指标为自变量的回归模型简称为传统复杂性回归模型, 自变量为两种度量指标结合的回归模型称之为组合回归模型. 从 3 组实验的整体面积可以看出, RMS 回归模型的各种评估指标数值均高于传统复杂性回归模型, 组合回归模型的各种评估指标数值又相对提升. 这说明与传统复杂性回归模型相比, RMS 回归模型对项目间依赖缺陷数量有着更强的解释能力和影响性.

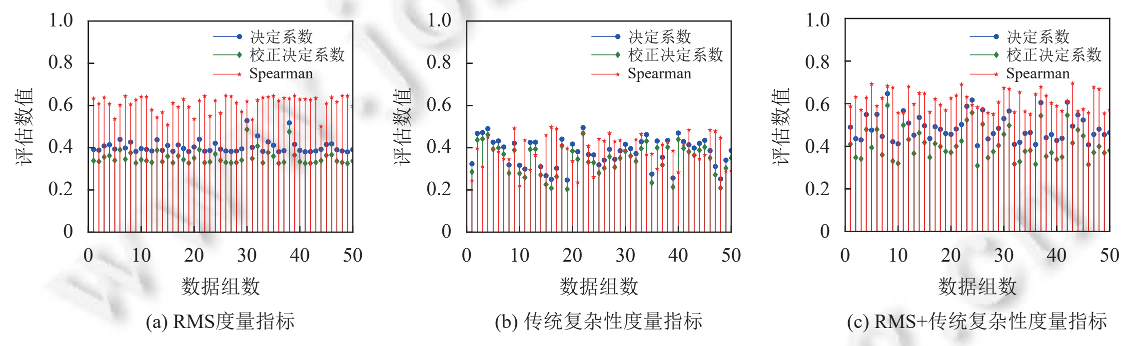


图 11 Groovy 项目间依赖缺陷数量的回归分析结果

针对 10 个实验对象, 每个软件项目取 50 次交叉验证评估指标的均值作为其每组实验的评估结果, 图 12 为 3 组对项目间依赖缺陷数量影响程度的实验评估结果. 从图中可以看出软件项目的 3 组实验中呈现如下趋势: (1) RMS 回归模型的决定系数高于传统复杂性回归模型, 组合回归模型中的决定系数明显提升并达到略高于 RMS 回归模型水平. 说明 RMS 回归模型的解释能力高于传统复杂性回归模型. (2) 校正决定系数由于消除了自变量数量的影响, 数值略低于决定系数. 校正决定系数在 3 种回归模型中的整体趋势与决定系数大致相同, 组合回归模型的校正决定系数与 RMS 回归模型基本持平, 并高于传统复杂性回归模型. RMS 回归模型和组合回归模型均比传统复杂性回归模型有着更好的拟合优度. (3) RMS 回归模型的 Spearman 等级相关系数分布在 0.6 左右, 高于其数值分布在 0.4 左右的传统复杂性回归模型, 可以看出 RMS 回归模型对项目间依赖缺陷数量的影响程度较大, 而传统复杂性回归模型对项目间依赖缺陷数量的影响程度不高, 是因为这些传统复杂性度量指标间接隐含 RMS 度量指标的特性, 即调用第三方软件的可能性, 但并不具备普遍性. 组合回归模型的 Spearman 等级相关系数分布在 0.6 以上, 比 RMS 回归模型略高, 说明组合模型可以对项目间依赖缺陷数量的影响程度更大. 通过以上对项目间依赖缺陷数量预测实验的评估指标结果分析, 我们可以得出结论: 与传统复杂性度量指标相比, RMS 度量指标很大程度地影响着软件中引入项目间依赖缺陷的数量.

© 中国科学院软件研究所 http://www.jos.org.cn

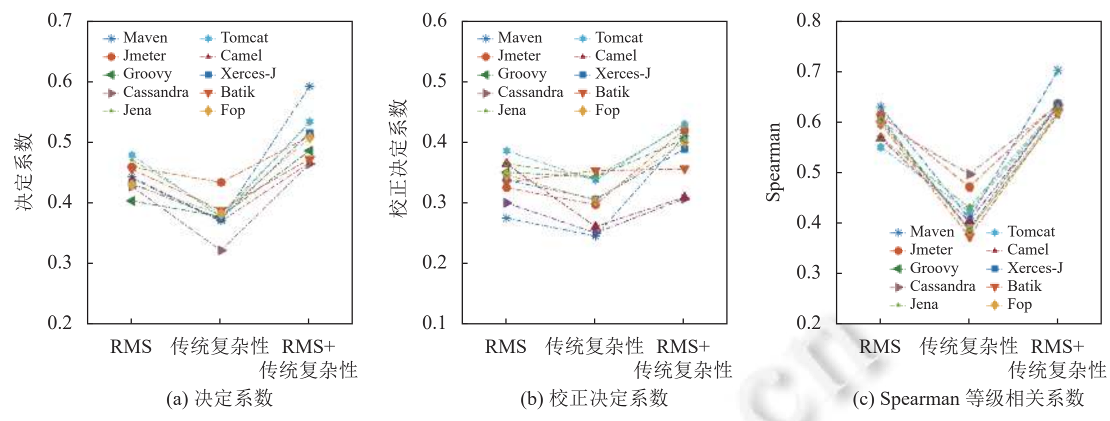


图 12　项目间依赖缺陷数量的回归分析结果

图 13 为软件 Groovy 中 RMS 度量指标对项目间依赖缺陷严重等级影响程度的实验结果, 在 50 次交叉验证中决定系数与校正决定系数分布趋势一致且数值分布在 0.4 左右, 说明 RMS 回归模型有着较好的拟合优度. Spearman 等级相关系数在 50 组数据中也保持一致, 且每次交叉验证对项目间依赖缺陷严重等级的预测值与实际值的相关性分布在 0.6 左右, 同样说明了交叉验证对构建稳定可靠模型的必要性, 亦证明了 RMS 回归模型对项目间依赖缺陷严重等级具有较高的影响性.

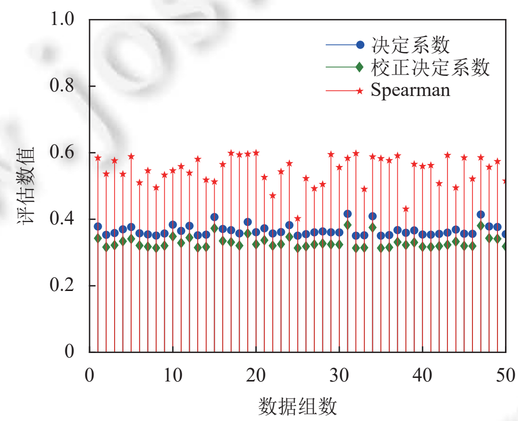


图 13　Groovy 项目间依赖缺陷严重等级的回归分析结果

对每个软件项目同样取 50 次交叉验证评估指标的均值作为其回归分析结果, 图 14 为项目间依赖缺陷严重等级预测实验的评估结果. 平均每个软件项目的决定系数与校正决定系数分别为 0.41 和 0.32, Spearman 等级相关系数达到 0.53, 说明软件项目的 RMS 回归模型有较好的解释能力及预测能力, 即 RMS 度量指标中规模和枢纽性对项目间依赖缺陷严重等级具有较高的影响性.

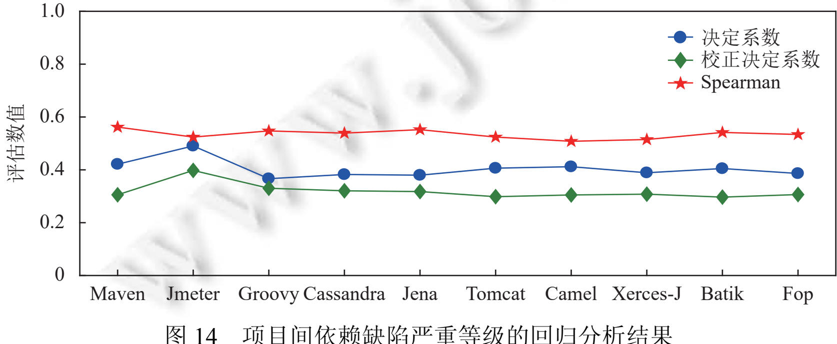


图 14　项目间依赖缺陷严重等级的回归分析结果

利用多元线性回归模型分析 RMS 度量指标对项目间依赖缺陷数量和严重等级的影响程度, 相对于传统复杂性度量指标, RMS 度量指标对项目间依赖缺陷数量具有较高的影响性, 且 RMS 度量指标中规模和枢纽性对项目间依赖缺陷严重等级具有较高的影响性. 同时, 将两种复杂性度量指标相结合, 对项目间依赖缺陷的数量和严重等级的影响程度更高.

6 软件库依赖图谱的复杂性度量方法的潜在应用场景

本文提出的软件库依赖图谱的复杂性度量方法除了在项目间依赖缺陷预测场景具有良好的应用价值外, 在软件生产过程中的如下 5 个应用场景也具备潜在优势.

(1) 软件库版本升级或迁移的风险分析. RMS 度量套件中“规模”和“接触面积”两个指标, 能够量化客户代码直接或间接地调用软件库中方法的规模, 以及边界节点直接调用软件库中 API 的集合. 因此, 这两个度量指标不仅能够刻画出边界节点在软件体系结构中的“枢纽重要性”, 更能够在软件库版本演化过程中提供版本升级或软件库迁移的影响性分析. 开发者需要关注“接触面积”指代 API 集合的兼容性和可靠性, 以避免引入软件运行的崩溃行为.

(2) 客户代码与软件库依赖解耦方案分析. RMS 度量套件中“接触面积”“多源性”“枢纽性”这 3 个指标, 能够量化客户代码与“多少软件库”有“什么程度的耦合关系”, 以及“是否存在功能枢纽的边界节点依赖软件库中功能枢纽方法的情况”. 针对“接触面积”“多源性”“枢纽性”指标数值高的边界节点, 开发者可以考虑采用“封装、隔离”的方式与对应的软件库进行依赖解耦, 以此方式降低软件库版本演化过程中为客户软件项目带来的不确定性影响.

(3) 软件库中安全漏洞 API 的可达性分析. 将 RMS 度量套件中“可达性”“多源传递性”“穿透性”这 3 个指标与公开安全漏洞数据相结合, 可以评估客户代码相对于安全漏洞 API 的可达性和影响范围. 通常一个软件项目可能会引入多个携带安全漏洞的软件库, 而对于边界节点可达的安全漏洞 API 需要引起开发者的关注与及时治理.

(4) 软件项目冗余软件库的检测与分析. 同上, 将 RMS 度量套件中“可达性”“多源传递性”“穿透性”这 3 个指标与精细的静态分析技术相结合 (例如, 具备处理动态绑定与反射机制的能力), 可以检测到未被客户代码真正调用到的软件库, 即软件项目的冗余依赖. 去除冗余软件库可以帮助软件项目“缩减体积”, 减轻维护的负担. 特别是针对安卓应用这类力求“轻量级”的软件, 冗余依赖应该被开发者及时去除.

(5) 测试用例优先级排序技术. 根据 RMS 度量套件中 7 个指标的计算, 可以识别到软件项目中风险因子高的边界节点. 因此, 在测试资源有限的情况下, 可以优先执行覆盖高风险边界节点的测试用例, 以提高项目间依赖缺陷的检测率. 这样的测试用例优先级排序技术能够帮助测试人员更好地分配测试成本, 提高测试效率.

7 效度威胁

本文分别从内在效度、外在效度和结构效度 3 个角度阐述效度威胁性.

● 内在效度. 实验过程中, 在搜集项目间依赖缺陷的数据集时, 使用软件项目所依赖的软件库的名字作为关键字, 以定位到项目间依赖缺陷. 然而, 软件库的名字也极有可能在缺陷的相关描述和讨论中偶然出现, 这样的数据收集方式可能会带来噪声. 为了准确定位到项目间依赖缺陷, 我们经过人工分析的方式去除噪声. 由 3 名具有软件工程专业背景和开发经验的研究生对每个缺陷信息进行分析理解, 挑选出能够从缺陷的描述和相关讨论内容准确地判断病因是与某个软件库相关的缺陷数据; 或是缺陷的堆栈信息中记录下在程序发生异常的时刻软件库函数的执行轨迹, 那么将此类缺陷视为项目间依赖缺陷.

此外, 在假设验证中我们分析边界节点的风险因子与项目间依赖缺陷数量和严重等级的相关性. 实验中, 为了便于对比, 我们分别利用 RMS 度量指标、传统复杂性度量及文献 [19] 所提出的度量指标来量化边界节点的风险因子. 本文在对边界节点风险因子度量时, 针对表 2 中的 10 个开源软件项目的所有版本进行分析, 计算并统计出每个版本中所有边界节点的 RMS 度量指标及基准指标. 进而求得每个边界节点的 RMS 度量指标及基准指标在各个版本中的平均值, 用于假设验证的分析过程. 然而, 在收集项目间依赖缺陷数据过程中, 识别到每个缺陷对应着特定的软件项目版本, 可能会存在边界节点在不同软件项目版本中引入多个项目间依赖缺陷的情况, 这会对相

© 中国科学院软件研究所 http://www.jos.org.cn

关性分析数据统计带来效度威胁. 实际计算中, 为了保证实验分析的可行性, 我们统计每个边界节点在所有版本中引入项目间缺陷的总数, 取其对应缺陷严重等级的平均值作为该边界节点引入缺陷的严重等级. 在此基础上讨论边界节点的风险因子与项目间依赖缺陷数量和严重等级的相关性.

- 外在效度. 本文提出的自动化风险度量技术 Certifies 能够实现对 Java 语言的软件系统进行解析, 提取出客户代码与软件库的依赖关系并量化软件边界节点的 RMS 度量指标, 以评估软件项目的质量. 鉴于此, 实验对象皆选取的是 Java 语言编写的开源软件. 需要澄清的是, 所提出的风险度量套件 RMS 对 C++等其他面向对象语言编写的软件同样适用.
- 构建效度. 本文的结构效度与静态分析技术的局限性相关. 本文的度量工具 Certifies 构建在静态分析工具 Soot 之上, 但是由于 Java 的动态绑定和反射机制, 影响了客户代码和软件库模块之间的依赖关系解析的准确性. 这可能导致计算所得的软件项目耦合强度的风险度量指标结果与真实值存在偏差.

8 总结与展望

本文结合软件项目的拓扑结构特征, 根据客户代码与软件库方法间的交互关系构建软件边界图模型, 并基于软件边界图模型提出一套风险度量指标用以描述多维质量属性, 进而量化软件边界节点对软件库的耦合强度. 通过将本文提出的 RMS 度量指标与传统复杂性度量指标和文献 [19] 所提出的度量指标进行对比分析, 证明 RMS 度量指标与项目间依赖缺陷的数量及严重等级均有较高的相关性, 且 RMS 度量指标对项目间依赖缺陷的数量及严重等级具有较大程度的影响.

结合第 6 节的讨论可以看出, RMS 度量指标在软件生产过程中的多个场景具有应用潜力. 例如, 利用“规模”和“接触面积”两个指标, 针对软件库版本升级或迁移场景进行风险评估; 结合“接触面积”“多源性”“枢纽性”这 3 个指标, 生成客户代码与软件库依赖解耦的方案; 将“可达性”“多源传递性”“穿透性”这 3 个指标与公开安全漏洞数据和精细静态分析技术相结合, 进行软件库中安全漏洞 API 的可达性分析与冗余依赖检测; 以及在测试资源有限的情况下, 设计出针对软件边界代码的自动化测试用例生成和优先级排序技术, 对高风险的边界节点投入较高的测试成本, 以提高测试效率.

References:

[1] China Academy of Information and Communications. Open Source Software White Papers, 2020. (in Chinese) https://www.sgpjbg.com/baogao/21135.html

[2] Raemaekers S, van Deursen A, Visser J. Exploring risks in the usage of third-party libraries. Technical Report, Software Improvement Group, 2011.

[3] Wang Y, Wen M, Liu ZW, Wu RX, Wang R, Yang B, Yu H, Zhu ZL, Cheeung SC. Do the dependency conflicts in my project matter? 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. 319–330. [doi: 10.1145/3236024.3236056]

[4] Wang HY, Guo Y, Mang ZA, Chen XQ. Automated detection and classification of third-party libraries in large scale Android Apps. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1373−1388 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5221.htm [doi: 10.13328/j.cnki.jos.005221]

[5] Zhan JW, Zuo QW, Niu YJ, Wang YW. A survey on isolation technology against malicious behavior of third party libraries of Android application. Computer Applications and Software, 2017, 34(10): 304–309, 315 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-386x.2017.10.054]

[6] Lu YM, Ying LY, Su PR, Feng DG, Jing EX, Gu YC. Security analysis and evaluation for the usage of settings mechanism in Android. Journal of Computer Research and Development, 2016, 53(10): 2248−2261 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2016.20160449]

[7] Ayala C, Hauge Ø, Conradi R, Franch X, Li JY. Selection of third party software in off-the-shelf-based software development—An interview study with industrial practitioners. Journal of Systems and Software, 2011, 84(4): 620–637. [doi: 10.1016/j.jss.2010.10.019]

[8] Geoffray N, Thomas G, Clément C, Folliot B. A lazy developer approach: Building a JVM with third party software. In: Proc. of the 6th Int’l Symp. on Principles and practice of programming in Java. Modena: ACM, 2008. 73–82. [doi: 10.1145/1411732.1411743]

© 中国科学院软件研究所 http://www.jos.org.cn

[9] Haddox JM, Kapfhammer GM. An approach for understanding and testing third party software components. In: Proc. of the 2022 Annual Reliability and Maintainability Symp. Seattle: IEEE, 2002. 293–299. [doi: 10.1109/RAMS.2002.981657]

[10] Li B, Ma YT, Liu J, Ding QW. Advances in the studies on complex networks of software systems. Advances in Mechanics, 2008, 38(6): 805–814 (in Chinese with English abstract). [doi: 10.3321/j.issn:1000-0992.2008.06.013]

[11] Valverde S, Cancho RF, Solé RV. Scale-free networks from optimal design. Europhysics Letters, 2002, 60(4): 512–517. [doi: 10.1209/epl/i2002-00248-2]

[12] de Moura APS, Lai YC, Motter AE. Signatures of small-world and scale free properties in large computer programs. Physical Review E, 2003, 68(1): 017102. [doi: 10.1103/PhysRevE.68.017102]

[13] Wheeldon R, Counsell S. Power law distributions in class relationships. In: Proc. of the 3rd IEEE Int'l Workshop on Source Code Analysis and Manipulation. Amsterdam: IEEE, 2003. 45–54. [doi: 10.1109/SCAM.2003.1238030]

[14] Valverde S, Solé RV. Hierarchical small worlds in software architecture. arXiv:cond-mat/0307278, 2007.

[15] Myers CR. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Physical Review E, 2003, 68(4): 046116. [doi: 10.1103/PhysRevE.68.046116]

[16] Lopez-Fernandez L, Robles G, Gonzalez-Barahona JM. Applying social network analysis to the information in CVS repositories. In: Proc. of the 26th Int'l Conf. on Software Engineering—W17S Workshop Int'l Workshop on Mining Software Repositories (MSR 2004). Edinburgh: IET, 2004. 101–105. [doi: 10.1049/ic:20040485]

[17] Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. In: Proc. of the 30th ACM/IEEE Int'l Conf. on Software Engineering. Leipzig: IEEE, 2008. 531–540. [doi: 10.1145/1368088.1368161]

[18] Musco V, Monperrus M, Preux P. A generative model of software dependency graphs to better understand software evolution. arXiv:1410.7921, 2014.

[19] Bhattacharya P, Iliofotou M, Neamtiu I, Faloutsos M. Graph-based analysis and prediction for software evolution. In: Proc. of the 34th Int'l Conf. on Software Engineering. Zurich: IEEE, 2012. 419–429. [doi: 10.1109/ICSE.2012.6227173]

[20] Wang Y, Zhu ZL, Yang B, Guo FD, Yu H. Using reliability risk analysis to prioritize test cases. Journal of Systems and Software, 2018, 139: 14–31. [doi: 10.1016/j.jss.2018.01.033]

[21] Wang Y, Zhu ZL, Yu H, Yang B. Risk analysis on multi-granular flow network for software integration testing. IEEE Trans. on Circuits and Systems II: Express Briefs, 2018, 65(8): 1059–1063. [doi: 10.1109/TCSII.2017.2775442]

[22] Wang Y, Yu H, Zhu ZL. A class integration test order method based on the node importance of software. Journal of Computer Research and Development, 2016, 53(3): 517–530 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2016.20148318]

[23] Xu C, Qin Y, Yu P, Cao C, Lv J. Theories and techniques for growing software: Paradigm and beyond. Scientia Sinica Informationis, 2020, 50(11): 1595–1611 (in Chinese with English abstract). [doi: 10.1360/SSI-2020-0079]

[24] Vasa R, Schneider JG, Woodward C, Cain A. Detecting structural changes in object oriented software systems. In: Proc. of the 2005 Int'l Symp. on Empirical Software Engineering. Noosa Heads: IEEE, 2005. 479–486. [doi: 10.1109/ISESE.2005.1541855]

[25] Girolamo A, Newman LI, Rao R. The structure and behavior of class networks in object-oriented software design. 2005. https://github.com/BHOSC/BUAAthesis/issues/115

[26] Yi T, Fang C. A novel method of complexity metric for object-oriented software. Int'l Journal of Digital Multimedia Broadcasting, 2018, 2018: 7624768. [doi: 10.1155/2018/7624768]

[27] Zhang W, Liu G, Zhu YF. Measurement method study on complexity of information system architecture. Application Research of Computers, 2011, 28(11): 4081–4085 (in Chinese with English abstract). [doi: 10.3969/j.issn.1001-3695.2011.11.021]

[28] Lew KS, Dillon TS, Forward KE. Software complexity and its impact on software reliability. IEEE Trans. on Software Engineering, 1988, 14(11): 1645–1655. [doi: 10.1109/32.9052]

[29] Tiwari U, Kumar S. Cyclomatic complexity metric for component based software. ACM SIGSOFT Software Engineering Notes, 2014, 39(1): 1–6. [doi: 10.1145/2557833.2557853]

[30] Chen X, Wang LP, Gu Q, Wang Z, Ni C, Liu WS, Wang QP. A survey on cross-project software defect prediction methods. Chinese Journal of Computers, 2018, 41(1): 254–274 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.00254]

[31] He JY, Meng ZP, Chen X, Fan XY. Semi-supervised ensemble learning approach for cross-project defect prediction. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1455–1473 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5228.htm [doi: 10.13328/j.cnki.jos.005228]

[32] Chen X, Gu Q, Liu WS, Liu SL, Ni C. Survey of static software defect prediction. Ruan Jian Xue Bao/Journal of Software, 2016, 27(1): 1–25 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4923.htm [doi: 10.13328/j.cnki.jos.004923]

[33] Chen X Shen YX, Meng SQ, Cui ZQ, Ju XL, Wang Z. Multi-objective optimization based feature selection method for software defect

© 中国科学院软件研究所 http://www.jos.org.cn

prediction. Journal of Frontiers of Computer Science and Technology, 2018, 12(9): 1420–1433 (in Chinese with English abstract). [doi: 10.3778/j.issn.1673-9418.1707016]

[34] D'Ambros M, Lanza M, Robbes R. On the relationship between change coupling and software defects. In: Proc. of the 16th Working Conf. on Reverse Engineering. Lille: IEEE, 2009. 135–144. [doi: 10.1109/WCRE.2009.19]

[35] Schröter A, Zimmermann T, Zeller A. Predicting component failures at design time. In: Proc. of the 2006 ACM/IEEE Int'l Symp. on Empirical Software Engineering. Rio de Janeiro: ACM, 2006. 18–27. [doi: 10.1145/1159733.1159739]

[36] Nagappan N, Ball T. Using software dependencies and churn metrics to predict field failures: An empirical case study. In: Proc. of the 1st Int'l Symp. on Empirical Software Engineering and Measurement. Madrid: IEEE, 2007. 364–373. [doi: 10.1109/ESEM.2007.13]

[37] Holmes R, Notkin D. Identifying program, test, and environmental changes that affect behaviour. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: IEEE, 2011. 371–380. [doi: 10.1145/1985793.1985844]

[38] Ma WWY, Chen L, Zhang XY, Zhou YM, Xu BW. How do developers fix cross-project correlated bugs? A case study on the GitHub scientific Python ecosystem. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 381–391. [doi: 10.1109/ICSE.2017.42]

[39] Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S. How the Apache community upgrades dependencies: An evolutionary study. Empirical Software Engineering, 2015, 20(5): 1275–1317. [doi: 10.1007/s10664-014-9325-9]

[40] Just S, Premraj R, Zimmermann T. Towards the next generation of bug tracking systems. In: Proc. of the 2008 IEEE Symp. on Visual Languages and Human-centric Computing. Herrsching: IEEE, 2008. 82–85. [doi: 10.1109/VLHCC.2008.4639063]

[41] Ko AJ, Myers BA, Chau DH. A linguistic analysis of how people describe software problems. In: Proc. of the 2006 Visual Languages and Human-Centric Computing. Brighton: IEEE, 2006. 127–134. [doi: 10.1109/VLHCC.2006.3]

[42] Wen M, Chen JJ, Wu RX, Hao D, Cheung SC. Context-aware patch generation for better automated program repair. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 1–11. [doi: 10.1145/3180155.3180233]

[43] Mockus A, Fielding RT, Herbsleb JD. Two case studies of open source software development: Apache and Mozilla. ACM Trans. on Software Engineering and Methodology, 2002, 11(3): 309–346. [doi: 10.1145/567793.567795]

[44] Paulson JW, Succi G, Eberlein A. An empirical study of open-source and closed-source software products. IEEE Trans. on Software Engineering, 2004, 30(4): 246–256. [doi: 10.1109/TSE.2004.1274044]

[45] 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]

[46] Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C. What makes a good bug report? IEEE Trans. on Software Engineering, 2010, 36(5): 618–643. [doi: 10.1109/TSE.2010.63]

[47] Song DH, Zhong H, Jia L. The symptom, cause and repair of workaround. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2020. 1264–1266.

[48] JSR 330 not picking up guice custom bindings. 2021. https://issues.apache.org/jira/browse/MNG-6739

[49] Annotations on a local variable crash GroovyC. 2021. https://issues.apache.org/jira/browse/GROOVY-2604

[50] Error when opening JTL(XML) file (that saves sub samplers results) due to tree marshaller circular reference exception: Recursive reference to parent object. 2014. https://bz.apache.org/bugzilla/show_bug.cgi?id=56975

[51] JDK 11 compilation failure: ClassVisitor.visit NestMemberExperimental throws unsupported operation exception. 2021. https://issues.apache.org/jira/browse/GROOVY-8727

[52] Kashtan N, Itzkovitz S, Milo R, Alon U. Topological generalizations of network motifs. Physical Review E, 2004, 70(3): 031909. [doi: 10.1103/PhysRevE.70.031909]

[53] Milo R, Itzkovitz S, Kashtan N, Levitt R, Shen-Orr S, Ayzenshtat I, Sheffer M, Alon U. Superfamilies of evolved and designed networks. Science, 2004, 303(5663): 1538–1542. [doi: 10.1126/science.1089167]

[54] Valverde S, Solé RV. Network motifs in computational graphs: A case study in software architecture. Physical Review E, 2005, 72(2): 026107. [doi: 10.1103/PhysRevE.72.026107]

[55] Chen CY, Xing ZC, Liu Y, Xiong KOL. Mining likely analogical APIs across third-party libraries via large-scale unsupervised API semantics embedding. IEEE Trans. on Software Engineering, 2021, 47(3): 432–447. [doi: 10.1109/TSE.2019.2896123]

[56] Teyton C, Falleri JR, Blanc X. Mining library migration graphs. In: Proc. of the 19th Working Conf. on Reverse Engineering. Kingston: IEEE, 2012. 289–298. [doi: 10.1109/WCRE.2012.38]

[57] Henry S, Kafura D. Software structure metrics based on information flow. IEEE Trans. on Software Engineering, 1981, SE-7(5): 510–518. [doi: 10.1109/TSE.1981.231113]

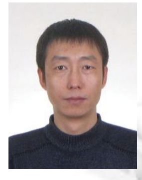
[58] Henry S, Kafura D. The evaluation of software systems' structure using quantitative software metrics. Software: Practice and Experience,

1984, 14(6): 561–573. [doi: 10.1002/spe.4380140606]
[59] Zage WM, Zage DM. Evaluating design metrics on large-scale software. IEEE Software, 1993, 10(4): 75–81. [doi: 10.1109/52.219620]
[60] Ma YT, He KQ, Du DH. A qualitative method for measuring the structural complexity of software systems based on complex networks. In: Proc. of the 12th Asia-Pacific Software Engineering Conf. (APSEC'05). Taipei: IEEE, 2005. 7. [doi: 10.1109/APSEC.2005.14]
[61] Derr E, Bugiel S, Fahl S, Acar Y, Backes M. Keep me updated: An empirical study of third-party library updatability on Android. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2187–2200. [doi: 10.1145/3133956.3134059]
[62] Backes M, Bugiel S, Derr E. Reliable third-party library detection in Android and its security applications. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 356–367. [doi: 10.1145/2976749.2978333]
[63] Kula RG, German DM, Ouni A, Ishio T, Inoue K. Do developers update their library dependencies? Empirical Software Engineering, 2018, 23(1): 384–417. [doi: 10.1007/s10664-017-9521-5]
[64] Mulliner C, Oberheide J, Robertson W, Kirda E. PatchDroid: Scalable third-party security patches for Android devices. In: Proc. of the 29th Annual Computer Security Applications Conf. New Orleans: ACM, 2013. 259–268. [doi: 10.1145/2523649.2523679]
[65] Shewale H, Patil S, Deshmukh V, Singh P. Analysis of Android vulnerabilities and modern exploitation techniques. ICTACT Journal on Communication Technology, 2014, 5(1): 863–867. [doi: 10.21917/ijct.2014.0122]
[66] Alvarez-Hamelin JI, Dall'Asta L, Barrat A, Vespignani A. *k*-core decomposition: A tool for the visualization of large scale networks. arXiv:0504107, 2005.
[67] Dorogovtsev SN, Goltsev AV, Mendes JFF. *k*-core organization of complex networks. Physical Review Letters, 2006, 96(4): 040601. [doi: 10.1103/PhysRevLett.96.040601]
[68] Meyer P, Siy H, Bhowmick S. Identifying important classes of large software systems through *k*-core decomposition. Advances in Complex Systems, 2014, 17(7–8): 1550004. [doi: 10.1142/S0219525915500046]
[69] Qu Y, Zheng QH, Chi JL, Jin YX, He AC, Cui D, Zhang HS, Liu T. Using *k*-core decomposition on class dependency networks to improve bug prediction model's practical performance. IEEE Trans. on Software Engineering, 2021, 47(2): 348–366. [doi: 10.1109/TSE.2019.2892959]
[70] Li H, Zhao H, Xu JQ, Li B, Li P, Wang JL. Research on hierarchy of large-scale software macro-topology base on *k*-core. Acta Electronica Sinica, 2010, 38(11): 2635–2643 (in Chinese with English abstract).
[71] Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. In: Proc. of the 28th Int'l Conf. on Software Engineering. Shanghai: ACM, 2006. 452–461. [doi: 10.1145/1134285.1134349]
[72] Halstead MH. A quantitative connection between computer programs and technical prose. In: Proc. of the 1977 COMPCON. Washington: IEEE, 1977. 332–335. [doi: 10.1109/CMPCON.1977.680855]
[73] He P, Li B, Ma YT, He LL. Using software dependency to bug prediction. Mathematical Problems in Engineering, 2013, 2013: 869356. [doi: 10.1155/2013/869356]
[74] Benesty J, Chen JD, Huang YT, Cohen I. Pearson correlation coefficient. In: Cohen I, Huang YT, Chen JD, Benesty J, eds. Noise Reduction in Speech Processing. Berlin: Springer, 2009. 1–4. [doi: 10.1007/978-3-642-00296-0_5]
[75] Belov DI, Armstrong RD. Distributions of the Kullback-Leibler divergence with applications. British Journal of Mathematical and Statistical Psychology, 2011, 64(2): 291–309. [doi: 10.1348/000711010X522227]
[76] Fenton NE, Pfleeger SL. Software Metrics: A Rigorous and Practical Approach. 2nd ed., Boston: Int'l Thomson Computer Press, 1997.
[77] Ratner B. The correlation coefficient: Its values range between +1/−1, or do they? Journal of Targeting, Measurement and Analysis for Marketing, 2009, 17(2): 139–142. [doi: 10.1057/jt.2009.5]
[78] Wang LJ. Principle of Statistics. Beijing: Tsinghua University Press, 2008. 156–176 (in Chinese).
[79] Wang J. The discussion of correlation analysis and regression analysis in statistics. Modern Economic Information, 2014, (8): 115 (in Chinese with English abstract). [doi: 10.3969/j.issn.1001-828X.2014.08.096]
[80] Jackson JE. A User's Guide to Principal Components. New York: Wiley, 1991. [doi: 10.1002/0471725331]
[81] 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]

附中文参考文献:

[1] 中国信息通信研究院. 开源生态白皮书. 2020. https://www.sgpjbg.com/baogao/21135.html
[4] 王浩宇, 郭耀, 马子昂, 陈向群. 大规模移动应用第三方库自动检测和分类方法. 软件学报, 2017, 28(6): 1373–1388. http://www.jos.

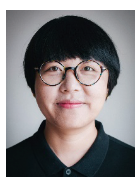
org.cn/1000-9825/5221.htm [doi: 10.13328/j.cnki.jos.005221]
[5] 湛家伟, 左奇伟, 牛莹姣, 王跃武. Android应用程序第三方库的恶意行为隔离技术综述. 计算机应用与软件, 2017, 34(10): 304–309, 315. [doi: 10.3969/j.issn.1000-386x.2017.10.054]
[6] 路晔绵, 应凌云, 苏璞睿, 冯登国, 靖二霞, 谷雅聪. Android Settings机制应用安全性分析与评估. 计算机研究与发展, 2016, 53(10): 2248–2261. [doi: 10.7544/issn1000-1239.2016.20160449]
[10] 李兵, 马于涛, 刘婧, 丁琦伟. 软件系统的复杂网络研究进展. 力学进展, 2008, 38(6): 805–814. [doi: 10.3321/j.issn:1000-0992.2008.06.013]
[22] 王莹, 于海, 朱志良. 基于软件节点重要性的集成测试序列生成方法. 计算机研究与发展, 2016, 53(3): 517–530. [doi: 10.7544/issn1000-1239.2016.20148318]
[23] 许畅, 秦逸, 余萍, 曹春, 吕建. 可成长软件理论方法和实现技术: 从范型到跨越. 中国科学: 信息科学, 2020, 50(11): 1595–1611. [doi: 10.1360/SSI-2020-0079]
[27] 张文, 刘刚, 朱一凡. 信息系统体系结构复杂性度量方法研究. 计算机应用研究, 2011, 28(11): 4081–4085. [doi: 10.3969/j.issn.1001-3695.2011.11.021]
[30] 陈翔, 王莉萍, 顾庆, 王赞, 倪超, 刘望舒, 王秋萍. 跨项目软件缺陷预测方法研究综述. 计算机学报, 2018, 41(1): 254–274. [doi: 10.11897/SP.J.1016.2018.00254]
[31] 何吉元, 孟昭鹏, 陈翔, 王赞, 樊向宇. 一种半监督集成跨项目软件缺陷预测方法. 软件学报, 2017, 28(6): 1455–1473. http://www.jos.org.cn/1000-9825/5228.htm [doi: 10.13328/j.cnki.jos.005228]
[32] 陈翔, 顾庆, 刘望舒, 刘树龙, 倪超. 静态软件缺陷预测方法研究. 软件学报, 2016, 27(1): 1–25. http://www.jos.org.cn/1000-9825/4923.htm [doi: 10.13328/j.cnki.jos.004923]
[33] 陈翔, 沈宇翔, 孟少卿, 崔展齐, 鞠小林, 王赞. 基于多目标优化的软件缺陷预测特征选择方法. 计算机科学与探索, 2018, 12(9): 1420–1433. [doi: 10.3778/j.issn.1673-9418.1707016]
[70] 李辉, 赵海, 徐久强, 李博, 李鹏, 王家亮. 基于k-核的大规模软件宏观拓扑结构层次性研究. 电子学报, 2010, 38(11): 2635–2643.
[78] 王立杰. 统计学原理. 北京: 清华大学出版社, 2008. 156–176.
[79] 王娟. 对统计中相关分析与回归分析的论述. 现代经济信息, 2014, (8): 115. [doi: 10.3969/j.issn.1001-828X.2014.08.096]


于海(1971－), 男, 博士, 副教授, 主要研究领域为软件测试, 软件重构及软件测试技术, 软件体系结构, 复杂网络理论, 混沌加密技术.

杨博(1995－), 女, 硕士, 主要研究领域为智能软件开发, 软件仓库挖掘.

王莹(1987－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件重构技术, 软件测试及分析.

许畅(1977－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据软件工程, 智能软件测试与分析, 自适应和自控软件系统.


徐美秋(1989－), 女, 博士生, 主要研究领域为智能软件开发, 软件测试和分析.

朱志良(1962－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为混沌加密技术, 复杂网络理论, 软件测试技术.

© 中国科学院软件研究所 http://www.jos.org.cn