软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(9):2665−2690 [doi: 10.13328/j.cnki.jos.006274]

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

软件缺陷自动修复技术综述*

姜佳君[1], 陈俊洁[1], 熊英飞[2,3]

[1](天津大学 智能与计算学部,天津 300350)

[2](北京大学 信息科学技术学院 计算机科学技术系 软件研究所,北京 100871)

[3](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 陈俊洁, E-mail: junjiechen@tju.edu.cn

摘 要: 软件缺陷是软件开发和维护过程中不可避免的.随着现代软件规模的不断变大,软件缺陷的数量以及修复难度随之增加,为企业带来了巨大的经济损失.修复软件缺陷,成为了开发人员维护软件质量的重大负担.软件缺陷自动修复技术有希望将开发者从繁重的调试中解脱出来,近年来成为热门的研究领域之一.搜集了94篇该领域最新的高水平论文,进行了详细的分析和总结.基于缺陷修复技术在补丁生成阶段所使用的技术手段不同,系统性地将软件自动修复技术分为4大类,分别是基于启发式搜索、基于人工模板、基于语义约束和基于统计分析的修复技术.特殊地,根据对近几年最新研究的总结,首次提出了基于统计分析的技术分类,对已有分类进行了补充和完善.随后,基于对已有研究的分析,总结了该领域研究所面临的关键挑战及对未来研究的启示.最后,对缺陷修复领域常用的基准数据集和开源工具进行了总结.

关键词: 软件维护;软件质量保障;软件缺陷修复;程序调试;软件自动化

中图法分类号: TP311

中文引用格式: 姜佳君,陈俊洁,熊英飞.软件缺陷自动修复技术综述.软件学报,2021,32(9):2665−2690. http://www.jos.org.cn/1000-9825/6274.htm

英文引用格式: Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021,32(9):2665−2690 (in Chinese). http://www.jos.org.cn/1000-9825/6274.htm

Survey of Automatic Program Repair Techniques

JIANG Jia-Jun[1], CHEN Jun-Jie[1], XIONG Ying-Fei[2,3]

[1](College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

[2](Software Engineering Institute, Department of Computer Science and Technology, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

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

Abstract: Program defects are inevitable during the development and maintenance processes. With the rapid increase of software scales, the number and repair complexity of program defects increase as well, which has caused huge economic loss to enterprises, and becomes the big burden for developers during maintaining. Automatic program repair (APR) techniques have the potential to release developers from heavy debugging tasks, and become a popular research topic recently. This study collected the most recent 94 high-quality publications in this research field. According to analyzing the approaches used for patch generation, APRs are systematically classified into four categories, i.e., search-based, template-based, constraint-based, and statistical-analysis-based APRs. Especially, this study

* 基金项目: 国家自然科学基金(62002256, 61922003); 天津市智能制造专项资金项目(20193155)

Foundation item: National Natural Science Foundation of China (62002256, 61922003); Intelligent Manufacturing Special Fund of Tianjin (20193155)

本文由“面向领域的软件系统构造与质量保障”专题特约编辑潘敏学教授、魏峻研究员、崔展齐教授推荐.

收稿时间: 2020-09-13; 修改时间: 2020-10-26; 采用时间: 2020-12-19; jos 在线出版时间: 2021-01-15

proposed the category of statistical-analysis-based APR for the first time based on the most recent publications, which complements and improves existing taxonomy. Based on existing techniques, the key challenges and insights are summarized for future research. Finally, benchmarks and open-source APR tools are briefly summarized for reference.

Key words: software maintenance; software quality assurance; program repair; program debugging; software automation

软件缺陷(software defects)在软件的开发过程中是不可避免的,特别是随着现代信息技术的迅速发展,软件规模在不断增加,软件缺陷的数量也在随之增加.软件缺陷会破坏程序的正常执行,使得程序在某种程度上不能满足其既有的功能要求.严重的软件缺陷不仅会造成企业的重大经济损失,甚至会对人们的生命安全造成重大威胁.因此,及时修复程序中的缺陷十分重要,已经成为软件维护中的一项重要任务.对 Linux 开发者的一项调查研究表明,在程序的开发过程中大约一半时间是用在了缺陷修复上[1].然而,修复程序中的缺陷不仅耗时,且容易出错.研究表明:开发者在修复软件缺陷时,有可能会引入新的程序缺陷[2],使得软件缺陷的修复变得更加困难.

软件缺陷自动修复(automatic software repair)技术有希望将开发人员从繁重的修复任务中解脱出来.从 2009 年开始,软件缺陷自动修复技术成为了一个热门的研究方向,吸引了来自软件工程(software engineering)、程序语言(programming language)、人工智能(artificial intelligence)、形式化验证(formal verification)等多个社区的大量研究人员.已有研究提出了一系列的软件自动修复技术,综合使用了软件分析(software analysis)、启发式搜索(heuristic search)、程序综合(program synthesis)以及机器学习(machine learning)等多种技术手段.缺陷自动修复技术根据程序中的测试或者通过静态分析技术等获取程序的规约(specification)信息,并基于此定位程序中出错的代码位置,最后采用不同的技术手段尝试生成修复代码使程序满足规约要求.

在过去的 10 多年中,大量的缺陷自动修复技术被提出.为了对该研究问题的进展进行系统地归纳总结和分析比较,本文搜集了最近 10 多年(2009~2020)发表的关于缺陷自动修复的相关论文并进行了梳理.我们采用谷歌学术搜索、ACM Digital Library、IEEE Explore、Springer、Elsevier 以及 CNKI 等搜索引擎和数据库,并且使用“program/software/fault/bug/defect repair/fix”“软件修复”和“程序修复”等关键字进行检索,并要求论文发表时间自 2009 年至今.同时,我们结合缺陷自动修复的共享主页(http://program-repair.org)对搜索结果进行了补充.该主页记录了大部分与软件缺陷自动修复技术相关的已发表论文,涉及新的修复技术以及相关的实证研究(empirical study)等.最后,对于每一篇论文,我们通过人工筛选过滤掉无关论文以及少于 5 页的短文.最终,我们一共搜集到了 186 篇程序缺陷自动修复相关的论文.

图 1 展示了从 2009 年~2020 年每年的论文发表数量统计,其中包括会议论文 149 篇,期刊论文 37 篇.从图中数据可以发现,该研究领域论文发表数量呈逐年增加的趋势.仅在 2018 年,就有 36 篇相关的论文发表.该数据表明,程序缺陷自动修复领域的研究热度在不断增加.我们又进一步对论文发表的会议和期刊进行了统计.图 2 列出了在过去 10 年里,不同会议和期刊上所发表的相关论文数量,其中,“others”分类包含了所有仅包含一篇论文的会议和期刊,JoS 和 SCIS 分别代表期刊《软件学报》和英文版《计算机科学》.从图中数据可以发现:相关论文主要发表在软件工程领域的高质量会议和期刊上,如 ICSE 和 TSE 等.此外,在形式化验证(如 CAV)以及编程语言(如 PLDI 和 POPL)等高质量会议上也有相关论文发表.

事实上,早在 2016 年,武汉大学的玄跻峰教授等人[3]就已经对软件自动修复方法的研究进展进行了分析和总结.并在此之后,先后有多篇综述论文发表[1,2,4−6].但由于上述论文发表时间比较早,它们主要针对 2017 年之前的相关研究进行了详细介绍,且大部分论文发表于 2016 年及之前.根据图 1 中的统计数据分析可以发现,大概一半的论文在 2017 年~2020 年间发表.在此期间,研究人员提出了一些新的方法和思路.特别地,随着最近几年人工智能领域的迅速发展,一些智能化的新技术也被应用到缺陷修复领域中[7−10].因此,本文旨在对目前已有的缺陷自动修复技术进行一个更加全面的总结和分析,并对已有技术进行系统性分类.此外,基于目前该领域的研究现状,本文提出该领域研究的可行思路以及面临的挑战,为未来的研究提供指导.

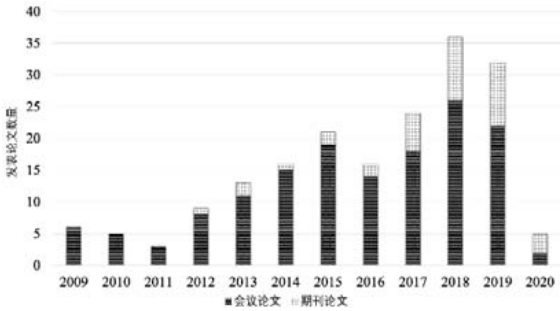


Fig.1 Distribution of publications over years

图 1 不同年份论文发表分布

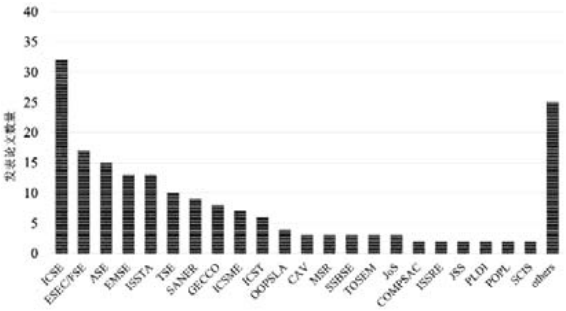


Fig.2 Distribution of publications in venues

图 2 不同会议或期刊论文发表分布

鉴于已有综述论文对部分相关文献已经有了相关介绍,本文在介绍不同类型的缺陷修复技术时将更加侧重对 2016 年之后发表论文的介绍.但是,由于个别类型的自动修复方法主要集中在早期所发表的工作中(2016 年或之前),为了使得本文内容更加完整,以及更好地反映相关研究的发展脉络,本文中所介绍论文与已有综述论文可能会存在少部分重叠.最终,本论文选取了 94 篇软件缺陷自动修复相关论文进行详细介绍,其中绝大多数论文是所涉及领域的高质量会议和期刊,例如 ICSE 会议(21 篇)、ESEC/FSE 会议(8 篇)、POPL 会议(1 篇)、CAV 会议(1 篇)、PLDI(1 篇)、OOPSLA 会议(1 篇)、ISSTA 会议(7 篇)、ASE 会议(8 篇)、TSE 期刊(10 篇)、TOSEM 期刊(1 篇)、《软件学报》(3 篇).

综上所述,本文的主要贡献总结如下.

(1) 对国内外软件缺陷自动修复技术的最新研究进展进行了系统的分析和介绍;

(2) 提出了基于统计和分析的修复技术分类,对已有的技术分类进行了补充;

(3) 对当前自动修复技术所面临的挑战进行了系统的梳理,并总结了未来可能的研究方向;

(4) 总结了该领域常用的缺陷数据集以及开源工具,促进未来的研究.

本文第 1 节介绍缺陷修复的研究框架并概述缺陷修复的组成模块、修复流程以及修复工具的评价指标.第 2 节对已有的基于测试或静态分析的缺陷修复技术进行详细介绍,并梳理和分析自动修复方法的演进方向.第 3 节根据研究现状的介绍总结该领域研究所面临的挑战及对未来研究的启示.第 4 节对自动修复常用验证数据集以及开源修复工具进行总结.最后,第 5 节对全文进行总结.

1 研究框架

(1) 缺陷修复过程

软件缺陷自动修复技术根据其修复的过程大致可以分为 3 个模块:缺陷定位(fault localization)、补丁生成(patch generation)、补丁排序过滤以及验证.图 3 是目前主流缺陷修复技术的一般流程.

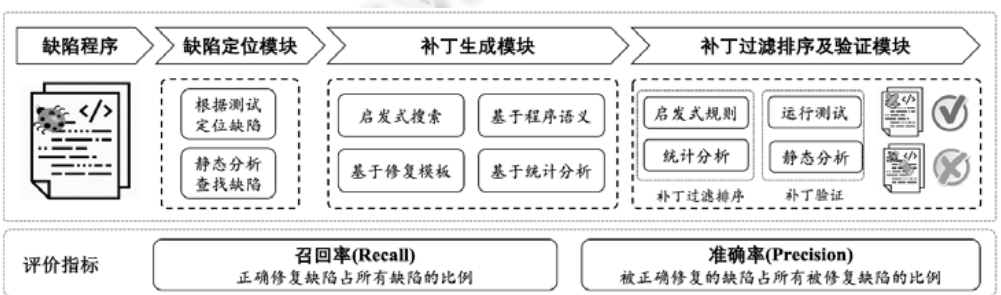


Fig.3 Automatic software defect repair research framework

图 3 软件缺陷自动修复技术研究框架

当给定一个缺陷程序,修复工具:(1) 首先通过静态分析、动态测试等技术手段确定程序中可能的缺陷代码位置;(2) 然后,每次选择一个候选出错位置,使用补丁生成技术尝试生成修复补丁;(3) 最后验证生成的修复补丁的正确性.基于静态分析检测的修复方法由分析工具预定义的规则检查补丁是否正确,而基于动态测试的修复方法要求修复补丁可以通过所有的测试.然而,修复工具可能产生多个补丁修复同一个缺陷,为了提升修复的效率和补丁的质量,通常在验证补丁之前或之后应用启发式规则或统计分析的方法对候选的补丁进行过滤和排序.修复工具针对候选出错位置依次迭代上述修复过程,直到找到正确的修复补丁或者达到终止条件,如运行超时等.

在上述修复过程中,缺陷定位的准确率会影响缺陷自动修复工具的修复结果.Liu 等人[11]通过实验表明:相比于使用自动化缺陷定位方法,提供准确的代码出错位置使得自动修复工具 TBar 多修复了 31 个程序缺陷(提升了 72.1%).因此,提升缺陷定位的准确率可以有效提升已有修复工具正确修复缺陷的数量.有很多相关的研究致力于提升定位的准确率,提出了各种不同的缺陷定位技术.目前,在缺陷自动修复领域使用比较多的缺陷定位技术存在两类:基于动态测试执行[12−15]以及静态程序分析的方法[16−19].基于动态测试执行的定位技术根据程序中的未通过测试以及通过测试的动态运行时特征对代码候选出错位置进行排序,比如基于程序频谱的定位技术.基于静态分析的方法通过预定义的规则描述程序的规约,然后通过程序分析技术判断程序是否满足给定规约.缺陷定位是一个独立的研究领域,本文对其不进行详细介绍,更多关于定位的相关技术介绍请参考最新的相关综述论文[20].

补丁生成模块负责生成缺陷修复代码,是自动修复方法的核心.根据上文的介绍,定位准确率会影响修复结果.事实上,缺陷定位对修复的整体影响依然十分有限,即使给定正确的代码出错位置,已有的自动修复方法也仅能正确修复小部分(<20%)程序缺陷[11].因此,目前的自动修复方法所面临的核心挑战依然是如何正确高效地生成修复补丁.最新的研究也是主要针对该模块进行优化和创新.因此,本文接下来的综述部分将根据修复工具所使用的补丁生成策略进行分类.如图 3 所示,本文将缺陷修复分为 4 大类,具体内容将在第 2 节进行详细介绍.

补丁过滤排序及验证模块是缺陷修复过程中的最后一步,也是保证补丁质量的关键一步.由于修复中使用的程序规约通常是不完备的,例如非精确的静态分析结果或不完备的测试集(弱测试集问题[21])等.因此,即使不正确的修复补丁也有可能会通过验证(在第 3 节,我们会更详细地讨论补丁的质量问题),对候选的修复补丁进行过滤[22,23]和排序[24],成为提升补丁质量的一个有效手段.但由于目前修复工具主要面临的挑战是难以生成正确的修复补丁,所以过滤和排序策略仅是对自动修复方法效果的进一步优化,并不能从根本上解决生成正确补丁难的问题.

综上,本文主要针对软件缺陷自动修复技术中的补丁生成模块进行详细的综述,对于缺陷定位以及补丁过滤排序及验证模块不做展开讨论(本文的局限性).读者可以参考相关的综述论文[1−6,20]了解更详细的内容.

(2) 修复效果评价指标

对于自动修复工具生成的修复补丁,我们将其分为以下 3 种类型:错误补丁(incorrect patches)、似真补丁(plausible patches)和正确补丁(incorrect patches).其中:错误补丁指不能通过验证模块验证的补丁;似真补丁可以通过验证,但由于改变了程序的原始语义而并非正确;正确补丁即语义正确的修复补丁.在度量缺陷自动修复工具的修复效果时,目前广泛使用的评价指标是召回率(recall)和准确率(precision).其定义如下:

$$召回率=\frac{|正确补丁修复的缺陷|}{|所有缺陷|},准确率=\frac{|正确补丁修复的缺陷|}{|正确补丁或似真补丁修复的缺陷|}.$$

召回率越高,反映自动修复工具的补丁生成能力越强;准确率越高,反映修复工具生成补丁的质量越好.在实际应用中,上述两个指标在某种程度上相互制约.修复的数量增加,代表修复工具有能力生成更多种类的补丁,从而导致生成似真补丁的可能性增大而降低准确率.因此,自动修复技术通常需要在两者中进行权衡.例如,已有的自动修复方法通过限定修复缺陷的类型[25],降低生成似真补丁的概率,提升修复准确率.这样的修复技术在原理上就限制了其补丁生成能力.本文在第 2 节中针对具体修复技术进行相关介绍.除上述两个通用评价指标,一些论文中也会采用上述指标的变体,比如针对语法等价的补丁的召回率和准确率(即要求正确补丁与目标

补丁语法等价)、Top-*N*(正确补丁排在候选补丁中的前 *N* 位置)召回率等.

2 程序缺陷自动修复技术综述

本节基于第 1 节中介绍的缺陷自动修复研究框架,根据补丁生成阶段所采用的技术,对已有的修复方法进行分类.本文将已有的自动修复技术分为以下 4 大类:基于启发式搜索(例如 GenProg[26−28]和 SimFix[29])、基于人工修复模板(例如 PAR[30]和 SketchFix[31])、基于语义约束(例如 Nopol[32]和 Angelix[33])以及基于统计分析(例如 DeepFix[8]和 SequenceR[9])的自动修复技术.需要说明的是,单一的缺陷修复方法可能同时应用多种技术.比如,SimFix 在项目中搜索相似代码的同时也利用了从历史修复补丁中提取的常用修改操作对补丁进行过滤.在这种情况下,我们将根据方法的主要创新对其进行分类.

2.1 基于启发式搜索

(1) 利用变异算子

基于启发式搜索的自动修复技术通过人工定义启发式规则,指导修复补丁的生成过程.2009 年被提出来的缺陷自动修复技术 GenProg[26−28]是典型的基于启发式搜索的技术,其基本思想是代码具有重复性,因此通过复用项目中的代码,有希望产生正确的修复补丁.在搜索过程中,GenProg 采用遗传算法,通过定义代码片段的交叉和变异操作实现已有代码片段的重新组合,增大补丁的搜索空间.该方法将生成补丁所通过的测试数量作为优化目标,对候选补丁迭代应用交叉和变异操作,直至产生可以通过所有测试的补丁.通过实验证明,该方法可以修复程序中的缺陷.该工作使得自动修复技术进入了一个新的时期,吸引了大量的科研人员开始对自动修复技术进行探索.

Qi 等人[34]在 2014 年提出了 RSRepair,该方法采用与 GenProg 方法相同的修复框架,只是将其中的遗传算法替换成了随机搜索.实验对比分析表明:GenProg 中的遗传算法并没有发挥较大的作用,使用随机搜索算法不仅可以正确修复同样的缺陷,且在 23 个缺陷上,随机搜索相比遗传算法实现了更高的效率.

在此之后,先后有一系列针对 GenProg 进一步优化的方法被提出.

2018 年,Oliveira 等人[35]提出了一种新的代码修改表示方法来对遗传算法进一步优化.原始的 GenProg 方法中所采用的代码修改表示将代码与修改作为两个独立的部分,互不干扰.GenProg 每次首先从所有定义的变异算子中选择一个,例如交叉或变异,然后根据此再去选择候选的修复补丁或代码片段,最后应用对应的变异算子产生新的候选补丁.Oloveira 等人提出的代码表示方法将变异算子和算子对应的代码元素编码在同一条“染色体”上.该表示方法的好处在于,变异阶段可以复用之前的代码操作.此外,通过变异“染色体”上的变异算子部分,该方法可以实现对同一段代码的不同变异操作.因此,相比于传统的遗传算法,其灵活性更强,可以提升修复工具的补丁生成能力.

2018 年,Yuan 等人[36]提出了基于多目标遗传算法的自动修复方法 ARJA,该方法同样是对遗传算法中的表示方法进行了优化.ARJA 将代码补丁表示为三元组$\langle b,u,v\rangle$,分别用来编码代码上的修改位置、修改操作类型和复用的代码元素.该方法由于记录了代码的修改过程,因此具有更强的表达能力,通过变异操作可以获得更大的补丁空间.除此之外,ARJA 将补丁生成过程转化为多目标优化问题,其两个优化目标分别是补丁对代码改动的大小以及应用补丁后程序通过测试的数量.因此,该方法倾向于修改较小的修复补丁.最后,ARJA 在补丁验证阶段通过移除与当前补丁无关的测试以及通过编译器分析提前排除不合法的补丁等方式提升补丁验证的效率.最终实验结果表明,ARJA 正确修复缺陷的数量接近 GenProg 的 4 倍.

与 ARJA 类似,2018 年,Mehne 等人[37]同样通过筛选程序中的测试样例对修复的过程进行加速.其具体方法为:对于给定的修复补丁,如果一个测试在运行的过程中没有覆盖补丁所修改的代码位置,那么该测试在验证该补丁时可以被过滤,即节省了部分测试运行的时间开销.除此之外,Mehne 等人还通过过滤候选出错位置实现修复加速.在程序运行时,分别搜集通过测试和未通过测试在候选出错位置处的变量取值情况:如果变量在未通过测试运行中的取值是其在通过测试运行中取值的子集,则对应代码位置出错的概率会比较小,在修复的时候可以被过滤掉.其依据为,同样的变量取值有很大的概率触发相同的程序错误.

2018 年,Sun 等人[38]通过对遗传算法的种群初始化以及变异过程进行优化来改善原有的 GenProg 修复方法.GenProg 方法在初始化最初的种群时,根据缺陷定位的结果从高到低采用贪心策略逐一尝试.然而,由于实际中缺陷定位的结果并不一定准确,真正出错的代码可能排在比较靠后的位置或者是候选位置相关的代码.在真实的工业场景下,定位的不准确会导致修复工具生成更多不正确的修复补丁,从而降低修复工具的实用性.针对上述问题,Sun 等人提出弱化定位结果的影响,在初始化种群时,同时考虑多处候选出错位置,并同时生成候选补丁,以此来缓解定位不准确带来的影响.除此之外,为了更高效地探索补丁空间,该文中方法采用了两种交叉操作,将候选补丁根据适应度分数分为高、低适应度两个集合,然后对其分别进行交叉操作,在提升高质量补丁收敛速度的同时,可以有效探索更大的补丁空间.类似方法还有 Dantas 等人[39]在 2019 年提出的使用语言模型中的 Doc2Vec 和 LSTM 模型优化 GenProg 的适应度函数,以及 2020 年 Villanueva 等人[40]提出的使用新颖性搜索(novelty search)算法对 GenProg 的搜索进行优化,以避免陷入局部最优解.本文不再逐一详细介绍.

上述方法主要针对 GenProg 方法在算法上的一些优化,实际上对其补丁生成空间影响不大,依然是复用项目中已有代码片段.2019 年,Yuan 等人[41,42]针对 GenProg 的补丁空间进行了扩充,将 GenProg 和 PAR[30](基于人工模板的补丁生成方法,将在第 2.2 节详细介绍)方法结合提出了 ARJA-e.根据上面的介绍我们知道,GenProg 是直接复用项目中已有的代码来生成补丁.在此基础之上,ARJA-e 借用 PAR 方法的修复补丁模板再额外产生一些新的代码片段加入到遗传算法的候选补丁集合中,与其原有的补丁一起进行变异和遗传.除此之外,ARJA-e 对 GenProg 的适应度函数(fitness function)进行了优化.根据上面的介绍,GenProg 将通过测试的个数作为补丁的适应度值.然而实际上,单个测试函数中可能存在多条断言(assertion),任何一条断言的失败都会导致整个测试函数的失败.ARJA-e 针对该情况进行了细化处理,将以测试为单位修改为以断言为单位,根据补丁通过断言的数量评价补丁的适应度.在 Defects4J 数据集[43]上的实验验证表明:ARJA-e 相比 GenProg 正确修复数量有了明显提升,大概是后者的 4 倍.

2019 年,Ghanbari 等人[44]提出了基于 JVM 字节码(bytecode)的缺陷修复技术 PraPR.在此之前的自动修复技术主要针对源代码.上述所介绍的基于变异的技术均通过定义源码上的变异算子生成修复补丁.与字节码相比,源码级别的修复存在结构复杂、反复编译耗时等缺点.字节码级别的修复可以有效克服上述缺点.类似遗传算法,PraPR 根据预先定义好的字节码上的遗传算子对缺陷代码进行修改.由于字节码中的结构种类有限且简单,因此变异算子所能覆盖到的补丁空间会更大.但是由于对字节码修改不需要二次编译,可以直接在 JVM 上运行,所以其效率会比较高.实际验证效果也表明,该方法相比已有的技术具有更高的效率.该方法因为不涉及到源代码的结构,也因此具有更强的通用性,可以修复任何中间代码为 JVM 字节码的编程语言程序,例如 Java, Kotlin,Scala 等.然而 PraPR 也存在其固有的缺点,字节码简单的同时也伴随着部分程序语义信息的丢失,例如变量名称等,使得基于源码字面语义的启发式方法不适用于对 PraPR 的进一步优化.

(2) 利用历史修复补丁

与上述基于遗传算法的缺陷修复不同,Le 等人[45]认为:在不同的应用软件中,软件缺陷会重复出现,之前的缺陷修复历史可以为修复补丁提供有效的指导.因此,引入了第三方数据源(即历史修复)对 GenProg 进行优化.基于该思想,作者在 2016 年提出一个新的缺陷修复技术 HistoricalFix.该方法同样采用了遗传算法来生成修复补丁,不同的是,HistoricalFix 仅仅采用变异(mutation)而不使用交叉(crossover)操作.在每次变异之后,计算候选补丁所涉及的修改操作,根据其他项目历史修复中常用的代码修改对补丁进行筛选.该方法相比之前的遗传算法引入了第三方的缺陷修复历史数据,对补丁的生成起到了更好的指导作用.最终的实验证明,该方法相比原始的 GenProg 方法有了非常大的效果提升.在 Defects4J 数据集中的 90 个缺陷上进行验证,GenProg 只能正确修复 1 个缺陷,而 HistoricalFix 可以正确修复 23 个.该实验表明,历史修复数据可以为补丁生成提供有效指导.

与 HistoricalFix 类似,2018 年,Wen 等人[46]同样应用从历史修复中挖掘的代码修改操作指导补丁生成,其不同点在于考虑了代码修改的上下文信息,从而可以提供更加准确的指导,有效约束补丁的搜索空间.基于该方法,作者实现了一个自动修复工具 CapGen.在历史修复提取阶段,Wen 等人通过记录被修改代码节点的父节点类型信息作为修改应用的前置条件,例如:“insert EXPR_STMT under METH_DECL”表明,插入的“EXPR_

STMT”应该是“METH_DECL”的子节点.因此,根据该规则,CapGen 不能在循环语句“FOR_STMT”中插入“EXPR_STMT”.类似的代码修改操作限定了特定修改操作的应用范围.除此之外,CapGen 在复用项目中已有代码生成补丁时,会统计不同代码表达式在项目中出现的频繁程度并进行排序,被频繁使用的表达式具有更高的优先级.上述的补丁空间约束方法和代码复用策略有效地提升了补丁的质量.在 Defects4J 数据集上的实验表明,该方法可以达到 84%的补丁准确率.但由于其对补丁空间的约束,导致其修复的数量并不多.

HistoricalFix 和 CapGen 方法应用历史修改为补丁搜索提供指导.2019 年,Kim 等人[47]提出直接应用历史中人工修复的模板,并开发了自动修复工具 ConFix.该工具从人工修复中直接提取代码修改操作.ConFix 在代码的抽象语法树(abstract syntax tree)上,根据修改代码的位置提取相关的上下文信息作为应用对应人工修复的条件,即仅当出错代码正确匹配上下文才可以应用对应的代码修改.ConFix 根据语法树的结构提取语法树中被修改节点的相邻节点作为上下文,例如父节点、左右邻节点等.实验结果表明,使用上下文信息平均可以减少 48%的不必要代码修改.

(3) 利用相似代码

上述技术主要针对补丁的搜索策略和代码的修改进行优化,还有一部分最新研究关注于优化补丁生成过程中所使用的代码元素.上述介绍的方法在复用项目中已有的代码时,仅考虑代码出现的频繁程度(例如 CapGen),甚至完全不考虑代码的特征,并且均没有考虑缺陷代码与复用代码之间的联系.针对于此,2016 年,Ji 等人[48]提出通过复用项目中与缺陷位置相似的代码片段生成补丁,并实现了缺陷修复工具 SCRepair.作者认为,与缺陷代码过于相似的代码可能存在同样的错误.因此,SCRepair 在复用相似代码作为修复参考时考虑了代码之间需要具有差异性.SCRepair 通过对比缺陷代码与参考代码之间的语法树结构的一致性衡量代码相似性,使用 ChangeDistiller[49]提取缺陷代码到参考代码的修改操作衡量代码之前的差异性.根据预设的相似性和差异性阈值对复用的参考代码进行过滤.SCRepair 与 RSRepair 相似,差别在于 SCRepair 在复用代码时考虑上述过滤条件,而 RSRepair 采用无过滤的随机搜索.

2017 年,Wang 等人[50]同样基于复用相似代码的基本思想,提出了修复工具 CRSearcher.与 SCRepair 不同的是,CRSearcher 将代码的搜索范围扩展到了其他的项目,应用基于标识(token-based)的代码相似度衡量方法,并且不要求相似代码与缺陷代码一定具有差异性.CRSearcher 并不算一个完整的自动修复工具,它不包含补丁验证模块,而是针对 FindBugs 检测出的缺陷为开发者推荐修复,需要人工确认补丁的正确性.

与上述方法类似,Xin 等人[51]在 2017 年提出的修复方法 ssFix 同样从代码库中搜索与缺陷代码相似的代码作为修复补丁的原材料.ssFix 采用阿帕奇公司(Apache)的 Lucene 作为代码搜索引擎.在生成补丁阶段,ssFix 并不是直接复用相似代码,而是通过 ChangeDistiller 提取缺陷代码与相似代码之间的差异,根据其中的不同点逐一应用修改操作.相比上述复用技术,ssFix 的代码复用粒度更细,有效地提升了其补丁生成能力.在 Defects4J 数据集上的实验表明,ssFix 可以正确修复 20 个缺陷.但是由于其细粒度的代码修改、补丁空间增大导致补丁的准确率比较低,产生了两倍数量的似真补丁.

Jiang 等人[29]认为,同一个项目中的代码具有更好的参考价值.同一个项目中的代码通常由同一个开发者或同一开发团队所编写,对于相似功能的代码模块,代码风格(例如程序结构和变量名等)具有较高的相似性.基于该思想,在 2018 年,他们提出了自动修复工具 SimFix,通过代码结构特征以及代码语义特征(变量和函数命名)搜索项目中的相似代码.在应用相似代码阶段,SimFix 同样采用基于差异的代码修改策略,即通过对比缺陷代码与相似代码之间的差异提取细粒度的代码修改操作.在复用代码阶段,会替换相似代码中的变量使其在缺陷代码位置语法正确.与 ssFix 不同,SimFix 所提取出的代码修改操作可以进行组合应用,并且同时使用历史修复中的常用修改操作对候选补丁进行过滤优化.相似代码和历史修改对 SimFix 的补丁空间提供了很好的约束,使其在最终的实验验证中不仅正确修复了更多数量的缺陷,其修复的准确率相比 ssFix 也提升了将近一倍.

基于类似的思想,2019 年,Hu 等人[52]提出了根据正确的代码为学生提交的作业代码推荐修复补丁,并实现了修复工具 Refactory.该方法首先将学生提交的正确代码作为代码库,通过预定义变换规则对其进行重构使代码结构一致.当学生提交的代码不能通过测试验证时,Refactory 通过将其与代码库中的正确代码进行匹配定位

缺陷代码可能出错位置,然后依赖测试在正确代码上的运行结果推断代码规约,最后基于该规约为缺陷代码生成修复代码骨架(sketch),并通过枚举所有的可能填充代码骨架产生修复代码.该方法与上述方法的不同在于其中包含了代码重构部分,使得语义相近的代码尽可能在结构上相似,方便之后的代码匹配.但其应用的场景是针对学生提交的作业,要求相同功能的代码存在不同的实现.在工业开发环境中,相同功能代码并不一定总是存在,使得其应用范围受到了一定限制.

(4) 本节小结

基于启发式搜索的自动修复方法是到目前为止研究最为广泛的一类方法.该类方法的优点是具有较强的通用性,适用于不同种类的程序缺陷.然而,其缺点是容易产生似真补丁而降低补丁的准确率.因此,其核心是如何定义合适的补丁搜索空间以提升补丁的质量.如前所述,修复方法的召回率和准确率是两个相互制约的指标,在保证准确率达到一定指标要求的情况下提升自动方法的修复数量,是搜索空间定义的重大挑战.

2.2 基于人工修复模板

(1) 利用模板生成补丁

基于人工模板的补丁生成,指根据开发者或研究人员的经验预定义一些补丁模板或者补丁生成策略用于指导修复的过程,2013 年被提出来的 PAR[30]是其中的代表性研究.通过分析人工修复补丁的特征,Kim 等人为 PAR 人工定义了 10 个修复模板,涵盖 6 大类的代码修改,例如替换函数的参数、替换变量初始化等.PAR 中包含了每个修复模板的实现逻辑,当给定一个缺陷代码位置,PAR 按照一定的顺序逐一实例化预定义的修复模板产生修复补丁.由于补丁生成的模板由人工定义和编写,生成补丁的质量相比随机产生会有较大提升.对开发者的调查问卷表明,PAR 产生的补丁具有较好的可读性.在此之后,2019 年,Koyuncu 等人[53]将 PAR 定义的修复模板用来修复缺陷报告中的缺陷,并提出了修复工具 iFixR.然而,该方法的缺点也是比较明显的.人工定义模板负担重且难以覆盖所有类型的程序缺陷,适合于分布较广泛的特定类型缺陷.比如,2019 年,Marginean 等人[54]提出的修复工具 SapFix 主要针对面向对象语言中的空指针缺陷等.

2017 年,Tian 等人[55]提出了一种针对 C 语言错误的自动修复方法 ErrDoc,该方法主要针对静态分析工具检测出来的不正确条件检查(error check)和资源释放(resource release)以及值的错误传播(error propagation)和输出错误(error output)这 4 种常见错误进行修复.类似地,ErrDoc 针对每种特定的缺陷定义了对应的修复模板.当检测到对应的程序缺陷时,即应用对应的修复模板.例如,函数的返回值出错的一种修复模板,是直接将返回值替换成为对应的期望值.相比于 PAR 以及 SapFix,该方法的主要不同点在于:通过静态分析在程序的控制流图上检测出错路径和未出错路径,通过对比差异为修复提供指导.

类似 ErrDoc,很多基于模板的自动修复技术主要是针对一些特定类型的缺陷.原因是缺陷的修复方式相对比较固定,数量有限,方便人工定义.比如:早在 2016 年,Gao 等人[56]通过定义 3 种修复模板来修复缓冲区溢出(buffer overflow)缺陷,实现了修复工具 BovInspector;Gao 等人[57]和 Yan 等人[58]分别在 2013 年和 2016 年提出了 LeakFix 和 AutoFix,通过预定义的修复模板(插入 free 语句),借助静态分析技术修复内存泄漏缺陷;再比如:2019 年,Xu 等人[59]通过添加空指针检查等模板,修复静态分析技术检测出来的空指针异常错误.

2018 年,Liu 等人[60]首先使用代码修改提取工具 GumTree[61]从 Stack Overflow 上的缺陷代码修复中提取代码修改操作,然后由人工总结成修复模板.该方法相比上述定义模板的好处在于:一些常用的变量或常量在问答网站上可能经常出现,这些数据可以包含到修复模板中,在生成修复补丁时可以起辅助作用.与此类似,Tan 等人[62]通过分析安卓(Android)应用软件崩溃的常见原因,总结了 8 个修复模板并用于自动化修复安卓缺陷.

2018 年,Hua 等人[31]提出了基于模板的自动修复技术 SketchFix,该方法主要关注的是缺陷修复技术中的效率问题.根据之前的介绍,通常情况下补丁生成是一个不断迭代过程,每次生成新的修复补丁都需要对程序进行重新编译.因此,修复的一部分时间开销在于重复编译.SketchFix 使用辅助函数替换待修复的代码,辅助函数内部的具体实现根据预定义的补丁模板生成.这样做的好处是:程序只需要编译一次即可,修复过程每次只修改辅助函数内的代码,因此只有非常有限的代码需要每次更新编译,从而避免了由于编译整个项目所带来的时间开销,可以有效提升修复的效率.与 SketchFix 类似,2017 年,Durieux 等人[63]提出了 NPEfix,通过元编程避免不同补

丁代码的重复编译.与此不同,2018 年,Mechtaev 等人[64]从测试的角度出发来提升修复的效率,并实现了修复工具 F1X.该工具同样适用预定义模板生成修复补丁,其主要的贡献在于:通过将修复补丁根据语义等价归类来避免具有相同语义的修复补丁被重复验证,最终实现提升修复效率的目的.

2019 年,Saha 等人[65]将基于模板的补丁生成技术扩展到了多位置修复,并实现了修复工具 HERCULES.其基本思想是:具有相似特征(如代码结构、变量命名等)的代码片段,以及在代码的历史修改中经常同时演化的代码片段有较大的概率需要协同的修复.因此,HERCULES 通过分析当前代码中与候选出错代码相似的代码片段以及与候选出错代码存在共同演化的代码片段扩展缺陷修复的位置.在该过程中,HERCULES 根据可达定义分析(reaching-definition analysis)对不同代码之间的关联性进一步分析,从而过滤掉不相关的代码.最后,在多个位置同时应用预定义的修复模板生成补丁.该方法在一定程度上扩展了之前方法仅针对单个代码位置的修复,但由于其对修复的多处位置具有上述限制,因此其实际上并不能修复所有类型的多位置缺陷.

基于模板的修复技术,由于其模板通常由人工定义,产生的补丁质量相比随机搜索的方法有所提升.但正因如此,该类方法的修复能力和应用范围由其定义的模板数量直接决定.2019 年,Liu 等人[11]通过分析已有的基于模板的修复技术,实现了一个新的修复工具 TBar.TBar 集成了已有方法中大部分的补丁模板,在实验验证中实现了更多正确的修复.然而,通过人工定义的方法毕竟受模板数量限制,难以大规模使用.因此,研究如何自动化地提取修复模板,是使自动修复技术走向实用化的一个有效途径.我们在第 2.4 节将介绍相关研究.

(2) 利用补丁模板优化候选补丁

与 PAR 相反,2016 年,Tan 等人[66]通过分析修复工具 GenProg 和 SPR[67]所产生的修复补丁发现,自动修复工具产生的大部分不正确补丁具有公共的特点:删除了程序中的部分代码语句.基于上述发现,作者提出了反模式(anti-pattern)方法.该方法在已有的自动修复工具基础之上定义了一系列规则对工具产生的补丁进行过滤,比如不能删除返回语句(return)、不能删除条件语句(if)等.作者基于 GenProg 和 SPR 自动修复工具实验,对其增加了上述补丁过滤条件.实验结果表明,上述过滤规则可以有效避免不正确的修复补丁.此外,该方法由于删除了部分候选补丁,补丁验证时间减少,修复的效率平均提升了 1.4 倍.但相比原始方法,过滤规则并没有使修复工具产生新的修复补丁,修复工具的召回率依然由其自身所决定,反模式只是在一定程度上提升了补丁的准确率.

2018 年,Soto 等人[68]通过分析已有的修复补丁,提取比较常用的修复模板,然后将其与 PAR 定义的模板一起用于优化 GenProg 方法的补丁生成过程.在 GenProg 迭代生成候选补丁过程中,使用上述定义的修复模板对补丁进行排序.与上述的反模式类似,由于该方法中定义的补丁作为辅助而非直接用于生成补丁,因此最后实验效果提升并不明显.

(3) 本节小结

基于人工修复模板的方法较大可能地应用了开发者或研究者的领域知识.修复模板因具有较强的目标性,即修复特定类型的缺陷,因此其优点是生成的补丁质量较高,不会产生晦涩难懂的代码修改.然而,其缺点也很明显.由于依赖人工定义修复模板,模板的种类受开发者经验限制,人工成本大.所以,该类方法主要针对一些比较常见的、特征比较明显的缺陷类型,比如空指针缺陷等.

2.3 基于语义约束

(1) 基于组件的补丁综合

基于语义约束的自动修复方法通过某种手段推断程序的正确规约,作为约束指导补丁的生成过程或对补丁的正确性进行验证.比较有代表性的研究工作是新加坡国立大学 Roychoudhury 教授团队在 2013 年和 2015 年先后提出来的自动修复工具 SemFix[69]和 DirectFix[70],这两种方法均通过符号执行技术在缺陷代码位置搜集使测试通过的程序语义约束信息,然后使用基于组件(component-based)的程序合成方法生成修复补丁.该过程需要依赖约束求解器(SMT solver)求解.DirectFix 相比于前者对补丁生成过程进行了优化:DirectFix 不严格按照定位的结果,而是优先选择比较简单的候选语句进行修复;其次,DirectFix 将补丁生成问题转换为部分最大可满足问题(partial MaxSAT),通过将补丁的复杂度(即对原始代码的修改大小)作为软约束(soft condition)来控制 pMaxSMT 求解器,生成更简单的修复补丁.

2016 年,D'Antoni 等人[71]认为,已有方法仅仅通过代码的语法变化来衡量补丁的复杂度来对补丁进行排序并不是有效的手段,进而提出一种基于代码结构和语义特征的补丁排序方法 Qlose.与 DirectFix 类似,在语法层面,Qlose 通过对比修改前后代码的语法结构差异作为补丁的语法复杂度,例如修改表达式的个数等.修改数量越少,表示补丁越简单.在语义层面,Qlose 针对给定的测试集计算语义差异.其具体做法是:对比通过的测试在修复前后程序上执行的路径,并计算路径的汉明距离(Hamming distance)来衡量补丁的复杂度.汉明距离越小,表示补丁越简单.最后,将上述两方面约束作为约束求解器的软约束求解可行修复补丁.该方法在一定程度上提升了补丁的质量.

Mechtaev 等人[33]在 2016 年提出了针对 SemFix 和 DirectFix 的优化方法 Angelix,该方法同样采用基于组件的程序合成方法,通过使用受控的符号执行技术使其具有更好的延展性(scalability),可以用于修复较大规模项目中的缺陷.此外,Angelix 采用增量的方式生成补丁,即首先根据部分测试生成修复补丁,然后做回归测试,将不能通过的测试再添加到用于生成补丁的测试集中,如此递归生成补丁.这样的方法可以有效简化程序的约束,将约束求解器的部分负担转移到运行测试验证,在一定程度上可以提升修复效率.Angelix 相比之前的工作可以修复程序中的多处错误.

2016 年,Rothenberg 等人[72]针对符号执行难以处理程序中的循环问题(搜索空间爆炸),提出了 AllRepair 方法.该方法通过指定循环展开次数和递归深度将代码转换为无循环代码,这样可以将代码转换为静态单赋值(static single assignment)形式,进而可以直接转换为 SMT 的约束.AllRepair 为了产生简单的修复补丁,当发现一个修改序列可以通过测试,会将其所有的超集排除.该方法相比之前的方法主要的贡献是使用循环展开近似原程序,其优点是可以处理循环,缺点是得到的修复补丁并不能保证通过测试,需要执行测试进一步验证.

2016 年,Xuan 等人[32]提出一种针对 Java 程序中条件语句错误的修复技术 Nopol,该技术针对出错语句的类型定义了不同修复策略:如果定位出错的代码位置是条件语句,则 Nopol 通常生成的修复补丁为修改原始的条件语句;如果定位出错的代码位置是非条件语句,则通过添加一个新的条件跳过当前语句的执行实现修复.在针对特定类型的缺陷生成修复补丁时,Nopol 首先在出错的代码位置搜集所有变量的取值情况,然后根据期望的条件语句取值情况(true 或 false),将程序语义编码成为 Z3[73]约束求解器的约束进行求解.在 Nopol 方法中,除了程序中的变量可以用于生成修复补丁,同时编码了一些常量(例如 0,–1 等)以及常用函数调用(如 *size*(·),*length*(·)等)用于支持补丁生成.该方法相比上述基于约束求解器的方法,由于仅针对条件语句(只有 true 和 false 两种取值)错误,不需要依赖符号执行搜集程序的约束信息,因此其在大项目上的可延展性会更好.缺点是只修复条件语句缺陷.

2017 年,Chen 等人[74,75]提出一种基于程序契约(contract-based)的缺陷修复技术 JAID.该方法基于程序状态抽象技术,通过监视并记录测试运行过程中不同程序位置的可使用表达式取值情况,同时结合测试执行的结果定位程序中的可疑缺陷位置,最后通过预定义模板生成修复的补丁.在修复过程中,JAID 同时结合了函数的纯度分析(purity analysis)等软件分析技术指导补丁生成.该方法中的核心是,根据程序运行过程中的表达式取值来推断程序的规约以确定出错位置.因此,本文将其归为基于语义约束的修复技术.

实际上,上述介绍的使用约束求解器的方法辅助生成修复补丁的核心是搜集程序中的约束并转化为约束求解问题.2017 年,Nguyen 等人[76]将缺陷修复转换为程序的可达性问题进行求解,其本质和上述技术类似,只是将测试执行的约束映射为程序中的未知条件语句,然后可以使用可达性问题的求解方法进行计算.该方法被证明得到的结果一定是正确的(sound),但不能保证完备性(completeness).

2017 年,Le 等人[77]提出了缺陷自动修复技术 S3,同时考虑程序的语法和语义特征指导补丁的生成过程.与之前介绍的基于约束求解的方法类似,S3 通过符号执行搜集测试的约束信息作为约束求解器的输入;不同的是,S3 为不同的代码修改操作预定义了优先次序,在约束求解的过程中,会根据该排序优化补丁的生成过程.类似地,最后,S3 通过对比修改之后的代码与修改之前代码的语法以及语义的相似程度对补丁进行排序,例如代码修改的大小、修改前后同一表达式的取值情况等,相似程度高的补丁排序优先.

2018 年,Mechtaev 等人[78]在 Angelix 的基础之上进一步优化并提出了 SemGraft 修复工具.SemGraft 在以下

两方面进行了优化:首先,Angelix 依赖测试验证生成补丁的正确性容易产生似真补丁,SemGraft 借助于相似功能的正确代码作为引用来推导缺陷程序的规约,辅助补丁的验证,提升补丁的质量;其次,SemGraft 结合了基于反例的程序综合技术,当程序综合器生成的修复补丁不能满足约束时,将反例返回给综合器用于优化下一次的搜索.该方法通过上述两方面优化可以有效提升补丁的质量,但由于其依赖于存在相同功能的其他代码实现作为参考,其应用场景受到较大的制约.

2018 年,Lee 等人[79]提出了 MemFix,修复 C 语言中的内存释放错误.MemFix 通过类型状态静态分析(typestate static analysis)技术获取程序中每个位置处所有可以被访问的内存对象的状态,该状态可以通过⟨*o*,*must*,*mustNot*,*patch*,*patchNot*⟩来表示,其中,*o* 表示特定内存对象的编号或内存地址,*must* 指在当前位置一定指向内存位置 *o* 的指针集合,*mustNot* 指在当前位置一定不指向内存位置 *o* 的指针集合,*patch* 指在当前位置的安全修复集合,而 *patchNot* 指在当前位置不一定安全的修复集合.*patch* 和 *patchNot* 中的每个修复是一个二元组(*n*,*e*),表示在第 *n* 行插入 *free*(*e*)语句.根据程序的执行流,动态更新程序的每个位置状态信息.最后,在程序的出口处可以得到所有的内存状态信息,将其转化为覆盖问题,即:寻找一个最小覆盖,通过插入 *free*(·)语句,使得所有内存状态最终均得以被释放,且不存在重复释放(double-free).该求解过程属于优化问题,依赖于约束求解器进行求解.

(2) 基于语义的代码搜索和复用

上述介绍的补丁生成方法通过重新组合项目中的已有代码片段生成修复补丁.近期的一些研究通过编码程序的语义信息,搜索可以直接复用的代码片段用于生成补丁代码.与 MemFix 类似,2018 年,van Tonder 和 Le Goues[80]提出了 FootPatch,针对 Java 语言的资源泄漏、内存溢出和空指针引用等缺陷.在该方法中,FootPatch 将程序的语义约束信息用分离逻辑(saparation logic)描述作为程序的规约.在最后的补丁生成阶段,FootPatch 在项目中搜索满足上述规约的代码段,直接复用其作为正确修复补丁.

2017 年,Ke 等人[81]提出了基于代码语义搜索的修复技术 SearchRepair,该方法首先建立了一个代码片段数据库.与之前方法类似,对于缺陷代码,SearchRepair 首先使用符号执行技术搜集满足程序测试的规约,然后在代码库中搜索满足对应程序规约的代码段.在搜索的过程中,SearchRepair 利用约束求解器求解代码段中变量与出错代码中的变量映射关系.当找到满足要求的代码段,则可以根据约束求解结果,使用缺陷代码中的变量替换搜索得到代码段中的变量,然后直接替换缺陷位置代码即可.上述方法一方面涉及到符号执行在大项目上延展性差的问题,另一方面,搜索代码库过程需要频繁的约束求解导致其效率比较低.目前,SearchRepair 将代码段的粒度规定为 1~5 行代码.如果降低代码段的粒度,例如表达式级别,其修复效率将难以接受.

2019 年,Afzal 等人[82]针对 SearchRepair 存在的表达力与效率等问题,提出了改进方法 SOSRepair.该方法采用了一种新的代码编码方式来存储代码,以加速在修复过程中代码的搜索过程.SOSRepair 在代码库的建立阶段记录每个代码段中使用的变量名和变量类型信息,同时通过符号执行搜索程序所有的可能执行路径并编码成为约束.在代码搜索阶段,可以根据变量类型以及代码段的约束验证其是否满足所有测试的输入输出约束.除此之外,SOSRepair 对复杂的代码结构有了更强的兼容性,例如结构体、循环以及库函数等.不仅如此,为了提升代码搜索效率,SOSRepair 会根据不同变量位置指导变量的映射过程(输入变量映射输入变量),避免约束求解枚举所有可能的映射关系.最后,SOSRepair 通过使用反例制导的程序规约推断方法降低了对测试的要求,即使只有未通过测试覆盖修复代码,该方法也可以根据未通过测试推导程序的规约,并在验证补丁失败时动态更新规约指导之后的补丁搜索过程.

(3) 本节小结

基于语义约束的修复技术通过将补丁生成过程转换成约束求解问题,可以充分利用已有的优化求解算法,避免反复执行测试来验证候选补丁的正确性.但是该类方法由于依赖于符号执行和约束求解技术,在比较复杂的大规模程序上难以适用.此外,基于组件的补丁综合方法容易产生过拟合以及可阅读性差的代码片段,从而降低代码可维护性.虽然最新的研究通过复用已有代码可以在一定程度上提升补丁的可读性,但大量的求解过程会导致修复过程的低效,限制了可复用代码片段的粒度,进而约束了修复方法所能修复的缺陷数量.

2.4 基于统计分析

(1) 补丁排序模型

基于统计分析的缺陷修复技术指利用统计的方法或学习的技术指导补丁生成的过程或对候选补丁进行优化.比较早的相关工作是2016年由Long和Rinard[24]提出来的修复方法Prophet,该方法是在SPR[67]基础之上的进一步优化.SPR也是Long等人在2015年提出来的基于人工定义模板的C语言条件语句修复技术.Prophet根据修复前后的代码特征训练了一个排序模型对SPR生成的修复补丁进行排序.Prophet搜集程序的两类特征信息:首先是操作,即修复补丁所使用的代码修改有哪些,例如插入语句或替换语句等;另一类特征是程序的静态特征,例如包含循环语句或检查某个变量的取值等.通过实验表明,该方法可以有效过滤不正确的修复补丁.虽然Prophet使用SPR同样的补丁生成技术,却可以多修复一个缺陷.但该结果也表明,单纯通过补丁排序技术难以实现较大的效果提升.

2017年,Saha等人[83]采用类似Prophet的方法,通过预定义补丁模板指导修复过程;同时,应用机器学习模型对候选的修复补丁进行排序.基于该方法实现了自动修复工具ELIXIR.该作者通过实证研究分析实际项目中的历史修复发现,面向对象语言(Java)程序中存在很多函数调用相关的缺陷,例如替换函数参数、替换同名函数调用等.基于上述分析结果,作者人工定义了一组补丁生成模板(文中称为Repair-Expressions),通过搜集缺陷代码位置可用的程序元素实例化补丁模板生成修复补丁.在此之后,ELIXIR根据程序的上下文特征,例如使用的程序元素在上下文中出现的频率、缺陷报告中出现的关键词等,训练一个补丁排序模型,对上述得到的候选补丁进行排序.该方法的思想与Prophet基本保持一致,差别在于ELIXIR所定义的补丁模板和排序模型以及针对的编程语言不同.

2019年,White等人[84]提出使用神经网络模型对项目中的相似代码进行排序,以供复用生成补丁.基于此,实现了自动修复工具DeepRepair.该修复工具预定义了两种修复模板用于指导生成补丁,分别是插入新语句和替换已有语句.新插入的语句或替换语句来自项目自身代码.DeepRepair通过预训练的神经网络模型对候选语句进行排序.在补丁生成阶段,DeepRepair会使用相似的局部变量替换复用代码中的未定义变量避免编译错误.然而,实验对比传统的基于遗传算法的自动修复技术GenProg表明,DeepRepair并不能产生更多正确修复补丁.其中的一个重要原因是:DeepRepair采用的代码修改模板过于简单,不能很好地覆盖不同的缺陷类型,而神经网络模型并不能增强其补丁生成能力.

(2) 提取补丁模板用于修复

2017年,Xiong等人[25]针对Java语言中的条件语句修复,提出了高精度的条件语句综合技术ACS.该技术应用了3种技术手段优化生成补丁的质量:首先,ACS利用代码的局部性原理,根据变量之间的依赖拓扑关系对修复条件中的变量选取进行排序,在拓扑图中,距离代码修改位置比较近的变量优先使用;此外,ACS通过分析代码中的自然语言注释内容,挖掘程序特征来补充测试所定义的不完整程序规约;最后,ACS通过统计分析相似项目中常用的条件谓词,用来对修复补丁所使用的表达式类型进行筛选和排序.通过分析其他项目中的频繁谓词可以获得领域特定知识,例如变量“hour”通常与常量12进行比较等,从而有效提升补丁的质量.实验结果表明:该方法相比之前方法,实现了大概两倍的修复准确率.

2017年,Long等人[85]针对Java语言的空指针(NPE)、数组越界(OOB)以及类型强转(CC)缺陷,设计并实现了自动修复技术Genesis.该技术针对每类缺陷分别准备数据集,然后应用整数线性规划(ILP)算法为不同的缺陷类型分别推断代码修改模板,并采用二元组$\langle t1,t2\rangle$的形式表示,其中$t1$表示代码修改之前的语法树,$t2$表示修改之后的语法树.在学习模板的过程中,Genesis随机地从同类型的代码修改训练集中选取样本进行抽象,然后将抽象之后的模板应用到训练集上.在尽可能保证修复数量最大化的情况下,控制抽象掉的属性最少(整数规划问题).在修复阶段,首先将缺陷代码与模板中的$t1$进行匹配,如果匹配成功,可以得到表达式的映射关系,然后根据$t2$生成修复之后的代码.在49个缺陷上的实验表明,Genesis可以正确修复22个缺陷.

2017年,Rolim等人[86]提出的Refazer从代码修改样例中学习修改模板.为了方便描述代码的修改操作,作者提出了一个领域特定语言.基于该语言,代码修改被描述为抽象语法树上的一系列代码重写规则.每一条规则

匹配部分语法子树,输出修改之后的语法子树.比如替换一个变量为函数调用,其输入是变量对应的子树(根节点是叶子节点),输出则是函数调用对应的子树(根节点是内部节点).Refazer 在提取模板时(重写规则序列),采用了基于演绎的程序综合方法(deductive program systhesis)生成重写规则.最后,基于人工定义的启发式规则对其生成的模板进行排序.其基本思想是倾向于尽可能复用已有的代码,且被修改的代码不要包含过多的上下文信息,目的是使得到的模板具有更强的鲁棒性.文中通过修复学生作业程序验证 Refazer 的修复效果,实验结果表明,Refazer 可以帮助 87%的学生修复缺陷代码.

2018 年,Zhong 和 Mei[87]提出了一个针对异常(exception)相关缺陷的修复技术 MiMo.与之前介绍的自动修复方法不同的是,MiMo 针对给定的程序缺陷推荐对应的修改操作(repair shape).本文的主要出发点是之前的实证研究关注于挖掘通用类型的代码修改操作,期望为自动化的修复工具提供指导.论文作者认为:当划定缺陷的范围,代码修改推荐效果将有希望进一步提升.为此,作者首先开发了一个根据程序的异常信息对缺陷类型进行自动分类的工具 ExFi,并构建了一个数据集.其中,数据集里的每个修复都被标记为特定类型的缺陷.MiMo 应用 ChangeDistiller,在该数据集中提取缺陷修复的修改操作.对于新的缺陷,MiMo 通过搜索缺陷报告或相关的 Issue 报告中的异常信息来匹配需要使用的代码修改操作.该方法目前只能针对异常相关的缺陷推荐代码修改操作,尚不能直接生成修复补丁.但该论文为以后的研究提供了一个新的思路,即:通过将缺陷分类提取补丁模板,有希望提升现有自动修复方法的准确率.

2018 年,Gulwani 等人[88]提出了 Clara,用于修复学生的作业程序.与 Refazer 不同的是,Clara 不依赖历史修复样例,而是直接聚类相同功能的正确代码.在该过程中,Clara 根据程序的控制流以及变量的使用情况映射功能相同的两段代码.此外,Clara 会考虑程序执行过程中变量的取值情况映射不同变量.最后,被归为一类的代码随机选择一个作为该类的代表.在修复阶段,通过匹配缺陷程序与正确程序中的变量和结构生成对应的修改.

2019 年,Bader 等人[7]提出了自动修复技术 GetaFix.其基本方法与 Genesis 类似,在代码的语法树级别提取代码修改操作以及代码修改所涉及到的上下文信息.在应用其修复新的缺陷时,首先将缺陷代码与模板进行匹配,匹配成功,则可以应用对应的修改操作生成补丁.由于不同的代码修改操作涉及的上下文特征不同,为了保证提取到的模板具有一定的通用性(在相似但不相同的上下文中可以应用),GetaFix 使用层次聚类(hierarchical clustering)算法对提取的模板进行归类合并,其目的是抽象掉同一类别模板中的不相同特征,例如使用不同的变量名等.但是,如果对模板中的代码特征过度抽象,会导致模板不能准确地描述其所修改代码的特征,模板的准确性将降低.为了平衡模板的通用性与准确性,GetaFix 在聚类过程中使用 Anti-Unification 算法避免过度抽象.由于该方法依赖相似的代码修改作代码抽象,因此对缺陷的重复性有一定要求,目前仅针对比较常见的一些静态工具检测出的缺陷类型,如空指针异常(NullPointerException)、缺少指定编码(DefaultCharSet)等.

类似的,2019 年,Bavishi 等人[89]同样针对静态分析工具检测出的缺陷进行修复,并实现了修复推荐工具 Phoenix.它同样依赖于聚类算法对从历史中提取的补丁模板进行抽象.为此,作者提出了一个领域特定语言(DSL),用来描述代码的修改模板.Phoenix 同样在代码的抽象语法树上提取修改模板.具体讲,根据静态分析工具报告的缺陷代码位置,Phoenix 从对应的语法树节点出发,使用 DSL 描述代码修改的上下文信息,实际上可以表示为有限状态自动机(finite state automata),自动机中的每个节点对应语法树中的节点,节点间的连边代表节点之间的关系,比如变量声明(decl).在模板的聚类阶段,Phoenix 采用贪心策略将所有兼容(compatible)的模板进行合并,在合并的过程中,会匹配不同模板所对应的自动机并求交集.此处的兼容性根据修改模板中所包含的字符串和涉及到的节点类型等因素决定,最终的目标是聚类之后的模板之间彼此不兼容.在应用其修复缺陷阶段,首先是匹配模板的上下文信息,匹配成功即可应用对应修改.

2020 年,Koyuncu 等人[90]提出了 FixMiner,通过迭代式聚类算法,从相似的代码修改中挖掘可以复用的代码修改模板.本文定义了抽象语法树上的代码编辑脚本语言,用来描述代码修改操作.在迭代式聚类过程中,FixMiner 将编辑脚本拆分成 3 个子部分进行索引,分别是代码结构(shape)、代码修改(action)以及代码文本(token).通过上述拆分,可以实现对代码修改聚类的有效加速.在实验验证部分,作者将 FixMiner 挖掘的修复模板用于修复工具 PAR 中,结果表明,其挖掘得到的模板可以用于修复真实缺陷.与此类似,Liu 等人[91]提出的修复工

具 AVATAR 从静态分析工具报告的缺陷及其对应的修复中学习修复补丁模板,用于生成修复补丁.在 Defects4J 数据集上的验证表明,该方法可以修复之前方法不能修复的缺陷.

上述介绍的方法主要依赖于从重复的代码修改中提取可复用补丁模板.而研究表明[92],实际程序修复中只有不到 20%的缺陷存在重复,大部分类型缺陷无法通过聚类提取补丁模板.针对此,2019 年,Jiang 等人[93]提出了基于单个历史修复的补丁模板提取技术 GenPat.该技术将代码映射成代码图表示形式,通过分析海量的开源代码,为补丁模板的抽象提供指导.具体讲,GenPat 所使用的代码图表示模型中,每个节点包含不同的属性信息,通过分析对应属性在代码库中出现的频繁程度,决定其通用性.频繁出现的代码属性具有较强的通用性,在模板抽象时被保留,否则被抽象掉.该技术通过将代码属性的抽象过程与代码修改分离,克服了对重复修改的依赖,因此可以依据单个修改提取补丁模板.该方法在 Defects4J 数据集上验证有效性,结果表明:GenPat 可以从单个历史修复中学习通用的补丁模板,用于修复真实缺陷.但其依然面临补丁的准确率低的难题.

(3) 端到端补丁生成模型

2017 年,Gupta 等人[8]使用序列到序列(sequence-to-sequence)神经网络(neural network)模型自动修复 C 语言中的编译错误,并基于此实现了自动修复工具 DeepFix.相比已有的自动修复技术,DeepFix 不依赖外部的缺陷定位过程,通过神经网络直接预测程序中的缺陷代码并产生修复补丁推荐.具体讲,DeepFix 采用了一个多层的(multi-layered)编解码(encoder-decoder)模型,其编码器和解码器均为基于 GRU 的循环神经网络(RNN)模型.此外,DeepFix 在模型的解码器端结合了注意力机制(attention mechanism)来提升预测的准确性.在编码程序阶段,DeepFix 将程序转换为符号序列(token sequence)“$(l_1,s_1),\ldots,(l_k,s_k),\langle eos\rangle$”作为网络输入,其中,$(l_i,s_i)$表示位置 l_i 的字符编码是 s_i,而$\langle eos\rangle$代表输入结束.网络预测输出是“(l_i,s_i')”,表示用编码为 s_i' 的符号替换位置 l_i 的原有符号.为了使得预测得到的结果可以还原成源代码,模型中只抽象掉了字符串常量和整数常量,分别用 STR 和 NUM 替代.DeepFix 依赖编译器验证其修复之后的程序是否编译正确.由于其每次预测的输出仅修复单处错误,对于多处位置缺陷,DeepFix 采用迭代式修复策略.最后,实验针对常见的 4 种简单编译错误进行修复,例如缺少分隔符、括号错误等.实验结果表明:DeepFix 可以正确定位 78.7%的错误代码位置,正确修复 27.0%的缺陷,且修复时间仅需要几十毫秒.该方法模型简单,针对简单缺陷比较有效,对于复杂的缺陷效果如何仍需进一步验证.

2018 年,Bhatia 等人[94]针对学生作业 Python 程序提出了自动修复技术.与 DeepFix 不同的是,该工作不仅修复程序的语法错误,同时修复程序的语义错误.其具体方法是:首先,使用神经网络模型修复缺陷代码的语法错误得到语法正确程序;然后,借助基于语义约束的程序修复技术,使程序满足语义规约.在语法修复阶段,作者使用了 RNN 模型,其输入是语法错误程序的符号序列,输出为修改之后的符号序列,即预测输出为程序而非修改.在语义修复阶段,作者通过定义代码重写规则替换程序中的部分变量或者符号,然后将程序的语义转换成逻辑约束,通过参考正确代码的语义,使用约束求解器求解代码可能的修复.该方法针对学生作业进行修复,存在大量重复的相同功能代码,且程序功能相对简单.实验结果表明,该方法平均可以修复 59.8%的语法错误和 23.8%的语义错误.

2019 年,Vasic 等人[10]提出使用联合预测(joint prediction)模型修复程序中的变量使用错误,即应该使用其他变量替换程序中的某些变量使用.论文的基本思想是,代码中出错的变量通常应该使用上下文中存在的某个变量进行替换.因此,作者提出使用多头指针网络模型(multi-head pointer network)对程序代码是否存在缺陷、缺陷的位置以及对应的修复进行联合预测.为此,该方法首先使用 LSTM(long-short term memory)模型对程序进行编码,模型的输出作为指针模型的输入.在该方法中,指针模型包含两个指针,其中一个指针指向错误使用的变量 p 的位置,另一个指针指向期望的正确变量 q 的位置.其对应的修复即使用变量 q 替换变量 p.特殊地,当程序中不存在变量错误使用缺陷时,模型的第一个指针将指向一个预定义的特殊位置.当给定一个数据集,首先对程序中的所有出错位置以及待替换的变量进行标记.在模型训练阶段,分别根据出错位置预测正确数以及修复变量预测正确数建立损失函数(loss function),最后的联合预测则取两者损失函数的求和作为最终的优化目标.最终实验在开源网站上的 15 万条 Python 源代码上进行测试,测试结果表明:基于上述预测模型,出错位置预测准确率为 71.0%,同时,修复的预测准确率为 65.7%.相比之前介绍的基于神经网络模型的修复技术,本方法被应用到大

型的项目代码中,具有更好的延展性;但其缺点是仅针对变量使用错误,应用范围受限.

2019 年,Tufano 等人[95]探索了使用自然语言处理中的 NMT(neural maching translation)技术为程序缺陷预测修复补丁的效果.在该论文中,作者采用了编码器-解码器模型,模型的输入和输出分别是出错的代码片段和正确的代码片段.需要注意的是:为了降低变量名称和常量值等带来的干扰,模型输入的代码首先需要抽象处理,为每个变量和常量等设定唯一的标识符,例如变量 *a* 为 *VAR*_1,常量 0 为 *INT*_1 等.特殊地,为了反映程序的语义特征,在抽象的过程中,作者预定义了一些常用的惯用词(idioms),例如 size,min 等,在抽象过程中被保留.由于上述已经介绍过编码器-解码器模型,基本思路相同,此处不再赘述.为了提升最终预测结果的准确率,作者采用了束搜索(beam search)算法对修复的预测进行调优.最终输出的代码通过还原标识符对应的原始值可得.实验结果表明,使用 NMT 模型对修复补丁的预测准确率在 12.0%左右.相比之前的工作,准确率比较低,其中一个主要原因是其针对的缺陷类型更广、复杂度更高.

2019 年,Chen 等人[9]在 Tufano 等人工作之后提出了针对 Java 语言代码缺陷自动修复技术 SequenceR,同样采用编码器-解码器模型.相比之前的工作,其主要的不同在于:SequenceR 采用了复制机制(copy mechanism)[96],简单描述为,可以将模型的输入端数据直接复制到输出端.这样做的好处是可以实现部分代码的直接复用,缓解模型不准带来的巨大误差.实验证明,SequenceR 的预测准确率高于上述 Tufano 等人提出的 NMT 模型.此外,将 SequenceR 用于修复 Defects4J 数据集中的 75 个单行代码缺陷,正确修复了 14 个.该实验结果表明:尽管相比已有基于搜索的修复方法效果还有差距,但使用机器学习的方法有希望正确修复大型项目中的真实程序缺陷.其优点是不依赖任何人工定义模板以及启发式规则.

(4) 本节小结

基于统计分析的修复技术在最近几年发展迅速,其优点是可以利用海量的开源代码数据指导修复过程,处理各种不同类型的缺陷,具有较强的通用性,并且不依赖人工精心设计启发式搜索方法以及人工定义修复模板,有希望成为未来的实用化修复方法.然而,该类方法依赖于训练数据的质量.如何从海量的开源数据中准确提取代码特征,是提升其修复能力的关键.尽管已有方法已经取得了初步成效(如 GetaFix 和 GenPat 等),但依然具有很大的提升空间.特别是对于端到端的补丁生成模型,已有方法主要适用于分类任务或自然语言处理的应用场景,在自动修复场景中表现并不理想.研究针对代码生成任务的高效方法,依然需要更多探索.基于统计分析的修复技术存在一个固有的能力约束:依赖于代码或代码修改的重复性.而开发者的开发以及修复过程常常伴随较高的创新性.目前的方法从数据中所学习到的领域知识依然十分有限,在缺陷修复中主要起辅助作用,距离实用化尚存较大鸿沟.

3 自动修复技术面临挑战与启示

根据第 2 节最新相关研究的介绍,目前的自动修复工具所能修复的缺陷依然占所有缺陷的小部分.根据我们对已有缺陷修复工具在通用缺陷数据集 Defects4J 上的修复结果统计,所有修复工具可以正确修复的缺陷总数仅有 101 个,还不到完整数据集的 30%(v1.2.0 版本包含 395 个缺陷).不仅如此,自动修复工具所产生的修复补丁准确率依然难以达到工业应用的要求.因此,自动化的修复技术目前依然面临巨大挑战.本节根据之前相关研究的介绍,对目前面临的问题和挑战,以及未来可能研究方向进行总结.

- 低召回率问题

Motwani 等人[97]通过对早期 7 种自动修复工具所修复的缺陷进行分析发现,正确修复的缺陷通常比较简单,例如空指针异常、修改变量等,不涉及大段代码或多文件修改,更不涉及生成复杂的代码结构(例如循环等).影响召回率的因素是多方面的,其中比较重要的原因可以归结为以下两点.

(1) 缺陷定位准确率低.有研究表明[11,98,99],定位的准确率会直接影响缺陷修复的数量,通过提升定位的准确率可以有效提升修复召回率;

(2) 补丁空间受限.已有的自动修复技术主要依赖于人工定义的修复模板或启发式规则约束补丁的空间,导致目前仅仅针对比较常见的简单缺陷进行修复,如 NPEfix 仅修复空指针缺陷.

- 低准确率问题

基于测试的缺陷自动修复技术依赖于测试提供程序规约过滤候选修复补丁.但实际中,测试通常比较弱[22],不能提供完整的程序规约,从而导致即使修复补丁通过测试也可能是不正确的(似真补丁),即修复工具会产生过拟合补丁[100,101].李斌等人[5]已经针对该问题进行了详细的分析和讨论,本文不再赘述.

针对缺陷自动修复存在的上述两点问题,本文将其所面临的的挑战及对未来研究的展望总结如下.

(1) 定位提供的信息.目前的缺陷定位技术在缺陷自动修复框架中所起的作用是返回代码中的疑似语句,指导补丁生成过程,两个过程相对独立而缺少交互.但是在实际开发者人工修复的过程中,并不是严格遵循上述的过程,而是两者相互作用[102]:定位的过程可以为修复提供出错的"原因"指导生成补丁;反过来,修复可以进一步优化定位的结果.DirectFix 通过优先选取简单的语句进行修复,在一定程度上打破了上述的修复范式.但实际上,定位所提供的信息依然有限.最新的研究[103]尝试应用自动修复产生的补丁优化缺陷定位的准确率,但其实验效果相比最新的定位技术并无明显提升.在缺陷定位研究领域,最新的一些研究工作通过概率模型推导程序出错原因.如何将类似信息反馈给补丁生成过程作为有效指导,目前依然缺少探索;

(2) 启发式规则.大部分已有的自动修复技术依赖人工定义的启发式规则作为生成补丁的指导,然而已有的启发式规则仅依赖单一数据特征,如代码的语法特征(DirectFix 将补丁修改的大小作为启发式规则)或代码语义特征(GenProg 将补丁通过测试的数量作为启发式规则).单一的约束会导致补丁搜索陷入局部最优,例如 GenProg 产生很多通过测试的似真补丁.因此,结合代码的多维度特征提出合适的启发式规则用于引导补丁的生成,是提升补丁质量的重要途径;

(3) 补丁模板数量.基于补丁模板的修复方法通常依赖于人工定义好修复的模板,在补丁生成阶段,根据条件逐一尝试.这种修复模式适用于修复比较常见的、开发者比较熟悉且容易定义的缺陷类型,例如插入一个条件语句、替换一个变量等.对于存在复杂逻辑约束的缺陷类型而言并不容易定义,因此在理论上,已有方法只能修复一些比较简单的缺陷.Noda 等人[104]将 ELIXIR 应用于实际的工业场景发现,其有限的模板是限制其修复数量的一个重要原因.虽然最新提出的修复方法尝试自动从历史修复中学习模板(例如 Genesis,GetaFix 等),但由于该类方法依赖大量重复的训练数据,其应用范围受到限制,也是仅仅针对个别类型的缺陷.GenPat 通过从单个代码修改提取补丁模板,可以有效克服上述问题,但目前也仅有少数缺陷可以通过该方式修复.其中的一个关键原因是,自动提取的模板质量低于人工定义模板.所以,研究从少量数据自动学习高质量补丁模板并应用于修复真实缺陷,依然面临重大挑战;

(4) 代码修改大小.已有的自动修复工具只针对修改比较小的缺陷类型,比如单行代码修改或单个函数代码修改等.实际中的代码缺陷可能涉及多函数甚至多文件修改,或者涉及复杂的代码结构(例如循环).尽管最新的工作已经开始探索程序多点修改,如 HERCULES,但其修复能力依然非常有限,仅修复多点代码修改类似或相同的缺陷.如果被修改的多处代码需要完全不同的修改甚至存在相互依赖关系,上述方法将无能为力.例如在程序 A 点添加一个标记变量,在程序 B 点对标记变量进行检查.除此之外,已有的缺陷修复工具通常将代码修改的范围限定在有限的代码区间内(<5 行),这也是限制已有方法召回率的重要原因.但扩大代码修改范围会导致补丁搜索难度增加,降低补丁的准确率.因此,在扩大代码修改范围时,如何根据修改之间的依赖关系约束补丁空间,是其中的重要挑战;

(5) 代码复用粒度.从早期的 GenProg 修复方法开始,利用代码的重复性修复缺陷被证明是有效的.最新的研究通过细粒度的代码复用实现了较大的效果提升,例如 ssFix,SimFix 等.然而,上述修复技术依然依赖于较大段的重复代码(5 行~10 行).在实际的项目中,这样的重复代码并不总是存在.因此,如何在更细粒度搜索并复用已有的代码用于合成新代码,是扩展现有补丁空间的有效手段.虽然基于组件的合成方法(例如 SemFix)在理论上可以实现上述目标,但由于约束求解器并非针对该特定问题所设计,实际求解得到的代码质量低、可读性差.所以,探索通过代码复用合成高质量的修复补丁,是提升修复召

回率的有效手段;

(6) 依赖未通过测试.目前,软件缺陷自动修复方法主要集中于基于测试的修复方法,即,根据程序中未通过的测试定位程序中的错误.然而在真实的开发场景中,未通过的测试并不总是存在的.在缺陷修复领域中常用的验证数据集 Defects4J(v1.2.0)中,96%(=381/395)的程序缺陷在被修复之前无对应的未通过测试[48].该问题会严重影响目前的自动修复技术在真实工业开发场景中的实用性.最新的一些自动修复方法尝试分析缺陷报告[53]、应用软件静态分析[57,59]等技术定位程序中的缺陷或验证补丁正确性,从而不依赖于未通过测试.但其修复效果相比基于测试的修复技术依然存在较大的差距.而且该类方法主要针对一些特定类型的缺陷,如空指针错误[59]、内存溢出[57]等.因此,非基于测试的自动修复技术在未来的研究中依然需要更多的探索;

(7) 不完整规约.测试为缺陷修复工具提供规约,测试不完备是导致似真补丁的一个重要原因.已有研究的实验[105]表明,测试的覆盖率与补丁的质量成正相关.因此,提升测试的覆盖率是提升补丁质量和修复准确率的重要手段.最新的一些研究[22,23,106]通过生成新的测试对候选补丁进行过滤,可以在一定程度上提升补丁质量.然而,由于测试生成依赖测试断言(oracle),导致自动生成的测试对补丁过滤有限.挖掘程序的其他特征对程序规约进行补全,是提升补丁质量的有效途径.已有工作尝试使用类似特征指导补丁生成(例如 ACS),但目前挖掘到的程序规约信息依然比较有限.可以考虑结合更多的数据源,比如项目的演化历史和缺陷报告等,对程序规约进行补全;

(8) 语义理解.程序自动修复技术实际上涉及程序理解的过程.然而,自动修复方法通过启发式搜索或代码复用的方式虽然在一定程度上可以修复程序中的部分代码,但实际上针对其修复结果并不能给出合理的解释,即并没有真正理解程序的语义.因此,补丁的生成过程在一定程度上具有随机性和盲目性.这也是导致对于开发者非常容易修复的一些缺陷,自动修复工具却难以修复的一个重要原因.但程序理解本身就是一件具有重大挑战的研究.随着近年来人工智能的发展,使用智能化的算法(例如深度神经网络)实现程序语义的部分理解,是可探索的方向.

自动修复技术的上述挑战是目前研究被广泛关注的重点问题,也是在未来的研究中亟待解决的难点.部分挑战在之前的综述论文中[1,3]也有被提出过,比如不完整规约导致的补丁过拟合等,此处结合最新研究对其进一步归纳和总结.除此之外,已有综述论文中同时提出了自动修复技术所面临的关于实用性的重要挑战,接下来本文对其进一步总结.

- 修复效率问题

目前,研究比较广泛的基于测试的自动修复方法大部分遵循“生成-验证”的模型[25−28,93],即首先生成修复补丁,然后通过运行测试验证补丁的正确性.该模型由于依赖测试的反复执行,会导致修复效率比较低,一般需要几十分钟到几个小时去修复一个程序缺陷,使得已有的自动修复方法只能离线使用,不能为开发者提供实时的反馈.已有研究从不同角度提升修复效率,如早期的自动修复方法 AE[107],通过分析补丁的等价性来避免重复补丁的验证过程.此外,Prophet[24]通过补丁排序方法使得正确的修复补丁尽可能得到提前验证,在一定程度上可以提升修复的效率.但上述方法依然不能避免验证大量候选补丁所带来的巨大时间开销.最新的研究中,通过补丁隔离[44]和字节码修复[31]避免了补丁验证过程中的代码反复编译开销,大幅度提升了修复的效率.然而,尽管如此,自动修复方法依然难以满足在线与开发者交互的实效要求.因此,修复效率依然是目前自动修复方法所面临的实用性挑战之一.

- 代码可维护性问题

由于自动生成的补丁一般通过组合复用已有的代码片段产生,在缺少领域知识进行有效指导的情况下,补丁代码具有一定的随机性,比如产生类似“if (*a*!=*a*)”和“if (1<3)”等死代码(dead code),甚至会产生重复代码[29]以及晦涩难懂的补丁代码[69]等,严重影响代码的可读性和可维护性.最新的研究通过启发式方法或统计分析尝试从已有代码或补丁数据中学习类似的领域知识,在一定程度上降低了补丁的随机性,但补丁质量依然难以保证.目前,依然缺少关于该问题的针对性研究.

- 工业场景中的实用性问题

根据之前的介绍,现有的通用自动修复方法主要是基于测试的方法.然而在真实的工业生产环境中,依赖的未通过测试用例并不能保证存在.另一方面,由于自动修复方法生成补丁的正确率和质量(可读性和可维护性)依然难以保证,其补丁最终依赖于人工验证.相比于人工修复,使用自动化的修复方法是否会更高效,目前缺少系统性的对比研究.此外,已有的研究主要在实验环境下、少数的数据集上验证方法的有效性.已有研究[104]将自动修复工具应用到工业场景中,实验结果与实验环境相差较大.因此,更加全面地验证自动修复方法的有效性,是评估其实用性的重要方面.近几年,尽管更多丰富的实验数据集被提出(参考第4.1节),但被使用的频率相对较低.其中的一个主要原因是不同的数据集所依赖环境不兼容,适配对应的方法会引入较大的时间开销.因此,设计和开发统一的、易于扩展的、模拟真实工业生产环境的通用缺陷修复平台,有利于验证和对比不同自动修复方法的有效性,促进修复技术的实用化研究进程.

4 缺陷修复数据集和开源修复工具总结

本节总结缺陷自动修复研究领域常用的基准数据集(benchmark)以及重要的开源缺陷自动修复工具.

4.1 常用缺陷数据集

表1中列出了自动修复领域常用的缺陷数据集,表中给出了每个数据集所对应的参考文献、发表时间、缺陷程序涉及语言、数据来源以及下载地址链接,方便之后的研究人员下载使用.

Table 1 Benchmarks for automatic program repair

表 1 缺陷修复验证数据集

名称	参考文献	发表时间	语言	数据来源	下载地址
Defects4J	[43]	2014	Java	开源项目	https://github.com/rjust/defects4j
ManyBugs	[108]	2015	C/C++	开源项目	https://github.com/squaresLab/ManyBugs
IntroClass	[108]	2015	C/C++	学生作业	https://github.com/BugZooOrg/IntroClass
IntroClassJava	[109]	2016	Java	IntroClass 的 Java 版本	https://github.com/Spirals-Team/IntroClassJava
CodeFlaws	[110]	2017	C/C++	Codeforces 编程竞赛	https://codeflaws.github.io/
DroixBench	[62]	2017	Java	开源项目	https://github.com/stan6/droixbench
QuixBugs	[111]	2017	Java/Python	Quixey 公司测试题目	https://github.com/jkoppel/QuixBugs
Bugs.jar	[112]	2018	Java	开源项目	https://github.com/bugs-dot-jar/bugs-dot-jar
Bears	[113]	2019	Java	开源项目	https://github.com/bears-bugs/bears-benchmark
BugSwarm	[114]	2019	不限	开源项目	http://www.bugswarm.org/

目前,使用最为广泛的是2014年Just等人[43]提出的Defects4J数据集.该数据集早期版本(v0.1.0)包含来自5个开源项目的357个真实程序缺陷,该部分缺陷也是目前被广泛使用验证缺陷修复工具效果的对象.该数据集中的每个程序缺陷包含至少一个未通过的测试可以触发程序中的缺陷,其中部分未通过测试来自于原始的程序缺陷报告,部分未通过测试是开发者在修复对应缺陷时新添加的(详见第 3 节中的讨论).近几年,该数据集在不断地扩充,最新的v2.0.0版本已经扩展到17个项目共835个缺陷.

ManyBugs数据集是由LeGoues等人[108]提出来的C/C++语言程序缺陷,该数据集中共包含来自7个开源项目的185个真实程序缺陷.其中,每个缺陷程序都有至少一个未通过的测试触发程序中的缺陷,且未通过测试均由项目的开发者编写.相比表1中其他C/C++语言缺陷数据集,ManyBugs中的项目规模最大.与此不同,LeGoues等人同时提出的IntroClass数据集来自于本科生的课程编程任务,共包含998个学生提交的缺陷程序,涉及6个编程任务.此外,程序代码的规模比较小,一般仅有5行~20行代码.类似地,IntroClass中的每个任务通过多个“输入-输出”样例来描述程序的规约.每个缺陷程序被至少一个测试样例所触发.

IntroClassJava是IntroClass数据集的Java语言版本,由Durieux和Monperrus[109]通过人工定义转换规则,使用脚本自动化将C/C++语言程序转换为Java程序.此外,他们将IntroClass中的“输入-输出”样例转换成了可执行的Junit单元测试.在转换的过程中,由于删除了其中的部分重复程序以及未能正确转换的程序,最终数据集中包含297个缺陷程序.

与 IntroClass 类似,CodeFlaws 数据集来自于在线编程竞赛平台 Codeforces.该数据集中共包含 3 902 个程序缺陷,其中每个缺陷都存在对应的未通过测试.与其他数据集不同,CodeFlaws 根据程序的代码修改将所有的程序缺陷划分为了 39 个缺陷类型,可以用来研究不同修复工具对特定类型缺陷的修复效果.

DroixBench 数据集包含来自 15 个开源 Android 软件的 24 个可复现的运行崩溃缺陷,特别地,该数据集中的程序缺陷均涉及用户界面(UI).

QuixBugs 数据集包含 Python 和 Java 两种语言的缺陷程序,其中 40 个 Python 缺陷程序来自于 Quixey 公司测试题目,程序代码为几行到几十行不等,且每个代码缺陷只涉及一行代码修改.其中的 Java 缺陷程序由人工编写,与上述 Python 缺陷程序一一对应.与 IntroClass 类似,该数据集同样是采用“输入-输出”样例的形式描述程序的规约.

Bugs.jar 是继 Defects4J 之后的一个更大的 Java 语言程序缺陷数据集,该数据集包含 8 个 Apache 开源项目中的 1 158 个程序缺陷.为了提升缺陷程序的多样性,不同的项目涉及不同的应用场景,如数据库、常用库、Web 框架等.此外,除了包含触发程序缺陷的测试,Bugs.jar 中还包含对应缺陷的问题编号(issue id)以及缺陷报告(bug report),可以用来追溯对应缺陷在项目问题追踪系统(issue tracking system)中的记录.

Bears 是由 Madeiral 等人[113]提出来的 Java 缺陷程序数据集.相比于上述的数据集,其特点是易于扩展.它通过检测 GitHub 上开源项目的持续集成(continuous integration)构建结果,自动识别不能通过测试的缺陷代码版本.因此,用户可以根据需要扩展数据集.其初始版本(v1.0)包含 72 个项目中的 251 个程序缺陷.

与 Bears 类似,BugSwarm 采用同样的策略不断扩展数据集的大小.截止其论文发表,BugSwarm 已经搜集了 3 091 个 Java 和 Python 程序缺陷,并执行周期性检测,不断扩展其数据集大小.

根据上面的介绍可以发现,缺陷程序的数据集在向着多语言、多种类、大规模演化.然而,目前使用较广泛的数据集依然比较集中(Defects4J 居多).其原因可能包含以下两点:(1) 使用相同的数据集方便不同方法对比验证;(2) 数据集的易用性.然而,单一的数据集可能会导致自动修复方法面临过拟合问题,而影响其效果的客观评价.因此在未来的研究中,不同的数据集应该被利用起来,同时需要考虑实用化的工业生产环境.

4.2 开源缺陷修复工具

工具是验证算法有效性的重要支撑,可以为其他研究人员学习其方法以及对比新技术提供方便.自动修复工具代码开源为后来研究者搭建了好的平台,可以促进该领域的发展.表 2 列出了 2016 年至今的一些开源缺陷修复项目,包含自动修复工具的名称、对应发表论文、发表时间、针对的编程语言、所属的修复技术分类以及下载链接.在该表格中,我们使用 HS(heuristic search)、MT(manual template)、SC(semantic constraint)和 SA(statistical analysis)分别代表基于启发式搜索、人工修复模板、语义约束和统计分析的自动修复技术.特殊地,DiffTGen 通过生成测试对候选的修复补丁进行过滤,该方法本身不能自动生成修复补丁,因此无分类项.此外,一些开源项目(如 Astor)集成了多个自动修复工具,此时,其分类为所有工具分类的并集.根据表中的引用可以检索对应修复工具的具体算法描述,下载地址可以方便阅读者检索和下载.

5 总 结

缺陷自动修复由于有希望将开发者从繁重的程序调试过程中解脱出来而受到越来越多的关注,众多的缺陷修复技术被先后提出.本文针对缺陷自动修复相关的文献进行了系统的整理.对最近 10 年间论文的发表情况进行了详细的分析.特殊地,本文针对 2016 年至今基于测试的缺陷自动修复技术进行了详细的分析和总结.根据缺陷修复技术在补丁生成阶段所使用的不同方法,本文将自动修复技术系统性地分为 4 类.相比已有的综述论文,本文首次提出了基于统计分析的缺陷修复技术类别,对已有的技术分类进行了补充.此外,本文对目前该领域研究所面临的挑战做了分析和总结,对未来的研究具有指导意义.最后,本文对缺陷修复领域常用的数据集和开源修复工具进行了整理和归纳,方便读者引用.

Table 2 Automatic program repair tools

表 2 缺陷自动修复工具

名称	引用	发表时间	支持语言	分类	下载地址
HistoricalFix	[45]	2016	Java	HS	https://github.com/xuanbachle/bugfixes
AllRepair	[72]	2016	C/C++	HS	https://github.com/batchenRothenberg/AllRepair
Angelix	[33]	2016	C/C++	SC	https://github.com/mechtaev/angelix/
Prophet	[24]	2016	C/C++	SA	http://www.cs.toronto.edu/~fanl/program_repair/prophet-rep
Refazer	[86]	2017	C#/Python	SA	http://www.dsc.ufcg.edu.br/~spg/refazer/
NPEfix	[63]	2017	Java	MT	https://github.com/SpoonLabs/npefix
DeepFix	[8]	2017	C/C++	SA	https://bitbucket.org/iiscseal/deepfix
Genesis	[85]	2017	Java	SA	http://www.cs.toronto.edu/~fanl/program_repair/genesis-rep
ErrDoc	[55]	2017	C/C++	MT	https://github.com/yuchi1989/ErrDoc
ACS	[25]	2017	Java	SA	https://github.com/Adobee/ACS
ssFix	[51]	2017	Java	HS	https://github.com/qixin5/ssFix
JAID	[74,75]	2017	Java	SC	https://bitbucket.org/maxpei/jaid/wiki/Home
S3	[77]	2017	Java	SC	https://xuanbachle.github.io/semanticsrepair/
Nopol	[32]	2017	Java	SC	https://github.com/SpoonLabs/nopol/
DiffTGen	[23]	2017	Java	–	https://github.com/qixin5/DiffTGen
CapGen	[46]	2018	Java	HS	https://github.com/justinwm/CapGen
SketchFix	[31]	2018	Java	MT	https://github.com/SketchFix/SketchFix
SimFix	[29]	2018	Java	HS	https://github.com/xgdsmileboy/Simfix
Clara	[88]	2018	Python	SA	https://github.com/iradicek/clara
FootPatch	[80]	2018	Java/C/C++	SC	https://github.com/squaresLab/footpatch
ARJA	[36]	2018	Java	HS	https://github.com/yyxhdy/arja
MemFix	[79]	2018	C/C++	SC	http://prl.korea.ac.kr/MemFix/
Droix	[62]	2018	Java	MT	https://github.com/stan6/droixbench
kGenProg	[115]	2019	Java	HS	https://github.com/kusumotolab/kGenProg
AVATAR	[91]	2019	Java	SA	https://github.com/SerVal-DTF/AVATAR
ConFix	[47]	2019	Java	HS	https://github.com/thwak/ConFix
DeepRepair	[84]	2019	Java	SA	https://github.com/SpoonLabs/astor
PraPR	[44]	2019	JVM bytecode	HS	https://github.com/prapr/prapr
SequenceR	[9]	2019	Java	SA	https://github.com/KTH/chai
TBar	[11]	2019	Java	MT	https://github.com/SerVal-DTF/TBar
Fix2Fit	[106]	2019	C/C++	MT	https://github.com/gaoxiang9430/fix2fit
kPAR	[99]	2019	Java	MT	https://github.com/SerVal-DTF/FL-VS-APR
Astor	[116]	2019	Java	HS/SA	https://github.com/SpoonLabs/astor
iFixR	[53]	2019	Java	MT	https://github.com/SerVal-DTF/iFixR
Refactory	[52]	2019	Python	HS	https://github.com/githubhuyang/refactory
GenPat	[93]	2019	Java	SA	https://github.com/xgdsmileboy/GenPat
SOSRepair	[82]	2019	C/C++	SC	https://github.com/squaresLab/SOSRepair
FixMiner	[90]	2020	Java	SA	https://github.com/SerVal-DTF/fixminer_source
ARJA-e	[41,42]	2020	Java	HS	https://github.com/yyxhdy/arja/tree/arja-e
JaRFly	[100]	2020	Java	HS	https://github.com/squaresLab/genprog4java/

References:

[1] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey. IEEE Trans. on Software Engineering, 2017,45(1):34−67. [doi: 10.1109/TSE.2017.2755013]

[2] Monperrus M. Automatic software repair: A bibliography. ACM Computing Survey, 2018,51(1):1−24. [doi: 10.1145/3105906]

[3] Xuan JF, Ren ZL, Wang ZY, Xie XY, Jang H. Progress on approaches to automatic program repair. Ruan Jian Xue Bao/Journal of Software, 2016,27(4):771−784 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4972.html [doi: 10.13328/j.cnki.jos.004972]

[4] Wang Z, Gao J, Chen X, Fu HJ, Fan XY. Automatic program repair techniques: A survey. Chinese Journal of Computers, 2018, 41(3):588−610 (in Chinese with English abstract).

[5] Li B, He YP, Ma HT. Automatic program repair: Key problems and technologies. Ruan Jian Xue Bao/Journal of Software, 2019, 30(2):244−265 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5657.html [doi: 10.13328/j.cnki.jos.005657]

[6] Liu BB, Dong W, Wang J. Survey on intelligent search and construction methods of program. Ruan Jian Xue Bao/Journal of Software, 2018,29(8):2180−2197 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5529.html [doi: 10.13328/j.cnki.jos.005529]

[7] Bader J, Scott A, Pradel M, Chandra S. Getafix: Learning to fix bugs automatically. In: Proc. of the ACM Programming Language (OOPSLA). ACM, 2019. 1−27. [doi: 10.1145/3360585]

[8] Gupta R, Pal S, Kanade A, Shevade S. Deepfix: Fixing common C language errors by deep learning. In: Proc. of the 31st AAAI Conf. on Artificial Intelligence (AAAI). 2017. 1345−1351.

[9] Chen ZM, Kommrusch SJ, Tufano M, Pouchet LN, Poshyvanyk D, Monperrus M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Trans. on Software Engineering, 2019. [doi: 10.1109/TSE.2019.2940179]

[10] Vasic M, Kanade A, Maniatis P, Bieber D, Singh R. Neural program repair by jointly learning to localize and repair. In: Proc. of the 7th Int'l Conf. on Learning Representations (ICLR). 2019. 1−12

[11] Liu K, Koyuncu A, Kim D, Bissyandé TF. TBar: Revisiting template-based automated program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2019. 31−42. [doi: 10.1145/3293882.3330577]

[12] Abreu R, Zoeteweij P, Van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Proc. of Testing: Academic and Industrial Conf. on Practice and Research Techniques-MUTATION. IEEE, 2007. 89−98. [doi: 10.1109/TAIC.PART.2007.13]

[13] Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B. Evaluating and improving fault localization. In: Proc. of the 39th Int'l Conf. on Software Engineering (ICSE). IEEE, 2017. 609−620. [doi: 10.1109/ICSE.2017.62]

[14] Xuan JF, Monperrus M. Learning to combine multiple ranking metrics for fault localization. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2014. 191−200. [doi: 10.1109/ICSME.2014.41]

[15] Xie X, Chen TY, Kuo FC, Xu B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. on Software Engineering and Methodology, 2013,22(4):1−40. [doi: 10.1145/2522920.2522924]

[16] Bian P, Liang B, Shi WC, Huang JJ, Cai Y. NAR-Miner: Discovering negative association rules from code for bug detection. In: Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2018. 411−422. [doi: 10.1145/3236024.3236032]

[17] Liang B, Bian P, Zhang Y, Shi WC, You W, Cai Y. AntMiner: Mining more bugs by reducing noise interference. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE). IEEE, 2016. 333−344. [doi: 10.1145/2884781.2884870]

[18] Li ZM, Zhou YY. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. of the 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (ESEC/FSE). ACM, 2005. 306−315. [doi: 10.1145/1081706.1081755]

[19] Wang QQ, Parnin C, Orso A. Evaluating the usefulness of IR-based fault localization techniques. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2015. 1−11. [doi: 10.1145/2771783.2771797]

[20] Wong WE, Gao RZ, Li YH, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016,42(8):707−740. [doi: 10.1109/TSE.2016.2521368]

[21] Qi ZC, Long F, Achour S, Rinard M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2015. 24−36. [doi: 10.1145/2771783.2771791]

[22] Xiong YF, Liu XY, Zeng MH, Zhang L, Huang G. Identifying patch correctness in test-based program repair. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). ACM, 2018. 789−799. [doi: 10.1145/3180155.3180182]

[23] Xin Q, Reiss SP. Identifying test-suite-overfitted patches through test case generation. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA), Vol.17. ACM, 2017. 226−236. [doi: 10.1145/3092703.3092718]

[24] Long F, Rinard M. Automatic patch generation by learning correct code. In: Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). ACM, 2016. 298−312. [doi: 10.1145/2837614.2837617]

[25] Xiong YF, Wang J, Yan RF, Zhang JC, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In: Proc. of the 39th Int'l Conf. on Software Engineering (ICSE). IEEE, 2017. 416−426. [doi: 10.1109/ICSE.2017.45]

[26] Forrest S, Nguyen TV, Weimer W, Le Goues C. A genetic programming approach to automated software repair. In: Proc. of the 11th Annual Conf. on Genetic and Evolutionary Computation (GECCO). ACM, 2009. 947−954.

[27] Le Goues C, Dewey-Vogt M, Forrest S, Weimer W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proc. of the 2012 34th Int'l Conf. on Software Engineering (ICSE). IEEE, 2012. 3−13.

[28] Le Goues C, Nguyen TV, Forrest S, Weimer W. Genprog: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2011,38(1):54−72. [doi: 10.1109/TSE.2011.104]

[29] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2018. 298−309.

[30] Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. In: Proc. of the 2013 35th Int'l Conf. on Software Engineering (ICSE). IEEE, 2013. 802−811.

[31] Hua JR, Zhang MS, Wang KY, Khurshid S. Towards practical program repair with on-demand candidate generation. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). ACM, 2018. 12−23. [doi: 10.1145/3180155.3180245]

[32] Xuan JF, Martinez M, Demarco F, Clement M, Marcote SL, Durieux T, Le Berre D, Monperrus M. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. on Software Engineering, 2016,43(1):34−55.

[33] Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE). ACM, 2016. 691−701. [doi: 10.1145/2884781.2884807]

[34] Qi YH, Mao XG, Lei Y, Dai ZY, Wang CS. The strength of random search on automated program repair. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). ACM, 2014. 254−265. [doi: 10.1145/2568225.2568254]

[35] Oliveira VPL, de Souza EF, Le Goues C, Camilo-Junior CG. Improved representation and genetic operators for linear genetic programming for automated program repair. Empirical Software Engineering, 2018,23(5):2980−3006.

[36] Yuan Y, Banzhaf W. ARJA: Automated repair of java programs via multi-objective genetic programming. IEEE Trans. on Software Engineering, 2018,46(10):1040−1067.

[37] Mehne B, Yoshida H, Prasad MR, Sen K, Gopinath D, Khurshid S. Accelerating search-based program repair. In: Proc. of the 2018 IEEE 11th Int'l Conf. on Software Testing, Verification and Validation (ICST). IEEE, 2018. 227−238.

[38] Sun SY, Guo JX, Zhao RL, Li Z. Search-based efficient automated program repair using mutation and fault localization. In: Proc. of the 2018 IEEE 42nd Annual Computer Software and Applications Conf. (COMPSAC). IEEE, 2018. 174−183.

[39] Dantas A, de Souza EF, Souza J, Camilo-Junior CG. Code naturalness to assist search space exploration in search-based program repair methods. In: Proc. of the Int'l Symp. on Search Based Software Engineering (SSBSE). Springer-Verlag, 2019. 164−170.

[40] Villanueva OM, Trujillo L, Hernandez DE. Novelty search for automatic bug repair. In: Proc. of the 2020 Genetic and Evolutionary Computation Conf. (GECCO). ACM, 2020. 1021−1028. [doi: 10.1145/3377930.3389845]

[41] Yuan Y, Banzhaf W. A hybrid evolutionary system for automatic software repair. In: Proc. of the Genetic and Evolutionary Computation Conf. (GECCO). ACM, 2019. 1417−1425. [doi: 10.1145/3321707.3321830]

[42] Yuan Y, Banzhaf W. Toward better evolutionary program repair: An integrated approach. ACM Trans. on Software Engineering and Methodology, 2020,29(1):1−53. [doi: 10.1145/3360004]

[43] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2014. 437−440.

[44] Ghanbari A, Benton S, Zhang LM. Practical program repair via bytecode mutation. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2019. 19−30. [doi: 10.1145/3293882.3330559]

[45] Le XBD, Lo D, Le Goues C. History driven program repair. In: Proc. of the 2016 IEEE 23rd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER), Vol.1. IEEE, 2016. 213−224. [doi: 10.1109/SANER.2016.76]

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

[47] Kim J, Kim S. Automatic patch generation with context-based change application. Empirical Software Engineering, 2019,24(6): 4071−4106.

[48] Ji T, Chen LQ, Mao XG, Yi X. Automated program repair by using similar code containing fix ingredients. In: Proc. of the 2016 IEEE 40th Annual Computer Software and Applications Conf. (COMPSAC). IEEE, 2016. 197−202.

[49] Fluri B, Wursch M, PInzger M, Gall H. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. on Software Engineering, 2007,33(11):725−743. [doi: 10.1109/TSE.2007.70731]

[50] Wang YY, Chen YT, Shen BJ, Zhong H. CRSearcher: Searching code database for repairing bugs. In: Proc. of the 9th Asia-Pacific Symp. on Internetware. ACM, 2017. 1−6. [doi: 10.1145/3131704.3131720]

[51] Xin Q, Reiss SP. Leveraging syntax-related code for automated program repair. In: Proc. of the 2017 32nd Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 660−670.

[52] Hu Y, Ahmed UZ, Mechtaev S, Leong B, Roychoudhury A. Re-factoring based program repair applied to programming assignments. In: Proc. of the 34th Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 388−398.

[53] Koyuncu A, Liu K, Bissyandé TF, Kim D, Monperrus M, Klein J, Le Traon Y. iFixR: Bug report driven program repair. In: Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2019. 314−325. [doi: 10.1145/3338906.3338935]

[54] Marginean A, Bader J, Chandra S, Harman M, Jia Y, Mao K, Mols A, Scott A. Sapfix: Automated end-to-end repair at scale. In: Proc. of the 41st Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2019. 269−278. [doi: 10.1109/ICSE-SEIP.2019.00039]

[55] Tian YC, Ray B. Automatically diagnosing and repairing error handling bugs in C. In: Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 2017. 752−762. [doi: 10.1145/3106237.3106300]

[56] Gao FJ, Wang LZ, Li XD. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In: Proc. of the 31st Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2016. 786−791.

[57] Gao Q, Xiong YF, Mi YQ, Zhang L, Yang WK, Zhou ZP, Xie B, Mei H. Safe memory-leak fixing for c programs. In: Proc. of the 37th IEEE Int'l Conf. on Software Engineering (ICSE). IEEE, 2015. 459−470. [doi: 10.1109/ICSE.2015.64]

[58] Yan H, Sui YL, Chen SP, Xue JL. Automated memory leak fixing on value-flow slices for c programs. In: Proc. of the 31st Annual ACM Symp. on Applied Computing (SAC). ACM, 2016. 1386−1393. [doi: 10.1145/2851613.2851773]

[59] Xu XZ, Sui YL, Yan H, Xue JL. VFix: Value-flow-guided precise program repair for null pointer dereferences. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). IEEE, 2019. 512−523. [doi: 10.1109/ICSE.2019.00063]

[60] Liu XL, Zhong H. Mining stackoverflow for program repair. In: Proc. of the 2018 IEEE 25th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018. 118−129. [doi: 10.1109/SANER.2018.8330202]

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

[62] Tan SH, Dong Z, Gao X, Roychoudhury A. Repairing crashes in android apps. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). IEEE, 2018. 187−198. [doi: 10.1145/3180155.3180243]

[63] Durieux T, Cornu B, Seinturier L, Monperrus M. Dynamic patch generation for null pointer exceptions using metaprogramming. In: Proc. of the 2017 IEEE 24th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2017. 349−358. [doi: 10.1109/SANER.2017.7884635]

[64] Mechtaev S, Gao X, Tan SH, Roychoudhury A. Test-equivalence analysis for automatic patch generation. ACM Trans. on Software Engineering and Methodology, 2018,27(4):1−37. [doi: 10.1145/3241980]

[65] Saha S, Saha RK, Prasad MR. Harnessing evolution for multi-hunk program repair. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). IEEE, 2019. 13−24. [doi: 10.1109/ICSE.2019.00020]

[66] Tan SH, Yoshida H, Prasad MR, Roychoudhury A. Anti-patterns in search-based program repair. In: Proc. of the 2016 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE). ACM, 2016. 727−738. [doi: 10.1145/2950290.2950295]

[67] Long F, Rinard M. Staged program repair with condition synthesis. In: Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 2015. 166−178. [doi: 10.1145/2786805.2786811]

[68] Soto M, Le Goues C. Using a probabilistic model to predict bug fixes. In: Proc. of the 2018 IEEE 25th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018. 221−231. [doi: 10.1109/SANER.2018.8330211]

[69] Nguyen HDT, Qi DW, Roychoudhury A, Chandra S. Semfix: Program repair via semantic analysis. In: Proc. of the 2013 35th Int'l Conf. on Software Engineering (ICSE). IEEE, 2013. 772−781. [doi: 10.1109/ICSE.2013.6606623]

[70] Mechtaev S, Yi J, Roychoudhury A. Directfix: Looking for simple program repairs. In: Proc. of the 37th IEEE Int'l Conf. on Software Engineering (ICSE), Vol.1. IEEE, 2015. 448−458. [doi: 10.1109/ICSE.2015.63]

[71] D'Antoni L, Samanta R, Singh R. Qlose: Program repair with quantitative objectives. In: Proc. of the Int'l Conf. on Computer Aided Verification (CAV). Springer-Verlag, 2016. 383−401.

[72] Rothenberg BC, Grumberg O. Sound and complete mutation-based program repair. In: Proc. of the Int'l Symp. on Formal Methods (FM). Springer-Verlag, 2016. 593−611.

[73] de Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer-Verlag, 2008. 337−340.

[74] Chen LS, Pei Y, Furia CA. Contract-based program repair without the contracts. In: Proc. of the 32nd Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 637−647. [doi: 10.1109/ASE.2017.8115674]

[75] Chen LS, Pei Y, Furia CA. Contract-Based program repair without the contracts: An extended study. IEEE Trans. on Software Engineering, 2020. [doi: 10.1109/TSE.2020.2970009]

[76] Nguyen TV, Weimer W, Kapur D, Forrest S. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer-Verlag, 2017. 301−318.

[77] Le XBD, Chu DH, Lo D, Le Goues C, Visser W. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In: Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 2017. 593−604. [doi: 10.1145/3106237.3106309]

[78] Mechtaev S, Nguyen MD, Noller Y, Grunske L, Roychoudhury A. Semantic program repair using a reference implementation. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). ACM, 2018. 129−139. [doi: 10.1145/3180155.3180247]

[79] Lee J, Hong S, Oh H. Memfix: Static analysis-based repair of memory deallocation errors for C. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2018. 95−106. [doi: 10.1145/3236024.3236079]

[80] van Tonder R, Le Goues C. Static automated program repair for heap properties. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). ACM, 2018. 151−162. [doi: 10.1145/3180155.3180250.]

[81] Ke YL, Stolee KT, Le Goues C, Brun Y. Repairing programs with semantic code search. In: Proc. of the 30th Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2015. 295−306. [doi: 10.1109/ASE.2015.60]

[82] Afzal A, Motwani M, Stolee K, BrunY, Le Goues C. SOSRepair: Expressive semantic search for real-world program repair. IEEE Trans. on Software Engineering, 2019. [doi: 10.1109/TSE.2019.2944914]

[83] Saha RK, Lyu YJ, Yoshida H, Prasad MR. Elixir: Effective object-oriented program repair. In: Proc. of the 32nd Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 648−659. [doi: 10.1109/ASE.2017.8115675]

[84] White M, Tufano M, Martinez M, Monperrus M, Poshyvanyk D. Sorting and transforming program repair ingredients via deep learning code similarities. In: Proc. of the 2019 IEEE 26th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019. 479−490. [doi: 10.1109/SANER.2019.8668043]

[85] Long F, Amidon P, Rinard M. Automatic inference of code transforms for patch generation. In: Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 2017. 727−739. [doi: 10.1145/3106237.3106253]

[86] Rolim R, Soares G, D'Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B. Learning syntactic program transformations from examples. In: Proc. of the 39th Int'l Conf. on Software Engineering (ICSE). IEEE, 2017. 404−415. [doi: 10.1109/ICSE.2017.44]

[87] Zhong H, Mei H. Mining repair model for exception-related bug. Journal of Systems and Software, 2018,141:16−31. [doi: 10.1016/j.jss.2018.03.046]

[88] Gulwani S, Radiček I, Zuleger F. Automated clustering and program repair for introductory programming assignments. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM, 2018. 465−480. [doi: 10.1145/3192366.3192387]

[89] Bavishi R, Yoshida H, Prasad MR. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In: Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. 2019. 613−624.

[90] Koyuncu A, Liu K, Bissyandé TF, Kim D, Klein J, Monperrus M, Le Traon Y. Fixminer: Mining relevant fix patterns for automated program repair. Empirical Software Engineering, 2020, 1−45. [doi: 10.1007/s10664-019-09780-z]

[91] Liu K, Koyuncu A, Kim D, Bissyandé TF. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: Proc. of the 2019 IEEE 26th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019. 1−12. [doi: 10.1109/SANER.2019.8667970]

[92] Yue RR, Meng N, Wang QX. A characterization study of repeated bug fixes. In: Proc. of the 2017 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2017. 422−432. [doi: 10.1109/ICSME.2017.16]

[93] Jiang JJ, Ren LY, Xiong YF, Zhang LM. Inferring program transformations from singular examples via big code. In: Proc. of the 34th Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 255−266. [doi: 10.1109/ASE.2019.00033]

[94] Bhatia S, Kohli P, Singh R. Neuro-symbolic program corrector for introductory programming assignments. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). IEEE, 2018. 60−70. [doi: 10.1145/3180155.3180219]

[95] Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans. on Software Engineering and Methodology, 2019,28(4):1−29.

[96] See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. In: Proc. of the 55th Annual Meering of the Association for Computational Lunguistics (ACL). ACM, 2017. 1073−1083. [doi: 10.18653/v1/P17-1099]

[97] Motwani M, Sankaranarayanan S, Just R, Brun Y. Do automated program repair techniques repair hard and important bugs? Empirical Software Engineering, 2018,23(5):2901−2947.

[98] Yang DH, Qi YH, Mao XG. Evaluating the strategies of statement selection in automated program repair. In: Proc. of the Int'l Conf. on Software Analysis, Testing, and Evolution (SATE). Springer-Verlag, 2018. 33−48.

[99] Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Le Traon Y. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In: Proc. of the 2019 12th IEEE Conf. on Software Testing, Validation and Verification (ICST). IEEE, 2019. 102−113. [doi: 10.1109/ICST.2019.00020]

[100] Motwani M, Soto M, Brun Y, Just R, Le Goues C. Quality of automated program repair on real-world defects. IEEE Trans. on Software Engineering, 2020. [doi: 10.1109/TSE.2020.2998785]

[101] Le XBD, Thung F, Lo D, Le Goues C. Overfitting in semantics-based automated program repair. Empirical Software Engineering, 2018,23(5):3007−3033.

[102] Jiang JJ, Xiong YF, Xia X. A manual inspection of defects4j bugs and its implications for automatic program repair. Science China Information Sciences, 2019,62(10):200102. [doi: 10.1007/s11432-018-1465-6]

[103] Lou YL, Ghanbari A, Li X, Zhang LM, Zhang HT, Hao D, Zhang L. Can automated program repair refine fault localization? A unified debugging approach. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). 2020. 75−87. [doi: 10.1145/3406889]

[104] Noda K, Nemoto Y, Hotta K, Tanida H, Kikuchi S. Experience report: How effective is automated program repair for industrial software? In: Proc. of the 2020 IEEE 27th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020. 612−616. [doi: 10.1109/SANER48275.2020.9054829]

[105] Yi J, Tan SH, Mechtaev S, Bohme M, Roychoudhury A. A correlation study between automated program repair and test-suite metrics. Empirical Software Engineering, 2018,23(5):2948−2979. [doi: 10.1007/s10664-017-9552-y]

[106] Gao X, Mechtaev S, Roychoudhury A. Crash-avoiding program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2019. 8−18. [doi: 10.1145/3293882.3330558]

[107] Weimer W, Fry ZP, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. In: Proc. of the 28th Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2013. 356−366. [doi: 10.1109/ASE.2013.6693094]

[108] Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans. on Software Engineering, 2015,41(12):1236−1256.

[109] Durieux T, Monperrus M. IntroClassJava: A benchmark of 297 small and buggy Java programs. [Research Report] hal-01272126. Universite Lille 1. 2016. https://hal.archives-ouvertes.fr/hal-01272126

[110] Tan SH, Yi J, Mechtaev S, Roychoudhury A. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In: Proc. of the 39th Int'l Conf. on Software Engineering Companion (ICSE-C). IEEE, 2017. 180−182. [doi: 10.1109/ICSE-C.2017.76]

[111] Lin D, Koppel J, Chen A, Solar-Lezama A. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In: Proc. Companion of the 2017 ACM SIGPLAN Int'l Conf. on Systems, Programming, Languages, and Applications: Software for Humanity. ACM, 2017. 55–56. [doi: 10.1145/3135932.3135941]

[112] Saha R, Lyu YJ, Lam W, Yoshida H, Prasad MR. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories (MSR). IEEE, 2018. 10–13.

[113] Madeiral F, Urli S, Maia M, Monperrus M. Bears: An extensible Java bug benchmark for automatic program repair studies. In: Proc. of the 2019 IEEE 26th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019. 468–478. [doi: 10.1109/SANER.2019.8667991]

[114] Tomassi DA, Dmeiri N, Wang YC, Bhowmick A, Liu YC, Devanbu PT, Vasilescu B, Rubio-González C. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). IEEE, 2019. 339–349. [doi: 10.1109/ICSE.2019.00048]

[115] Higo Y, Matsumoto S, Arima R, Tanikado A, Naitou K, Matsumoto J, Tomida Y, Kusumoto S. kGenProg: A high-performance, high-extensibility and high-portability APR system. In: Proc. of the 2018 25th Asia-Pacific Software Engineering Conf. (APSEC). IEEE, 2018. 697–698. [doi: 10.1109/APSEC.2018.00094]

[116] Martinez M, Monperrus M. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. Journal of Systems and Software, 2019,151:65–80. [doi: 10.1016/j.jss.2019.01.069]

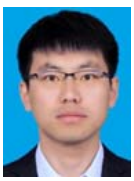
附中文参考文献:

[3] 玄跻峰,任志磊,王子元,谢晓园,江贺.自动程序修复方法研究进展.软件学报,2016,27(4):771–784. http://www.jos.org.cn/1000-9825/4972.html [doi: 10.13328/j.cnki.jos.004972]

[4] 王赞,郜健,陈翔,傅浩杰,樊向宇.自动程序修复方法研究述评.计算机学报,2018,41(3):588–610.

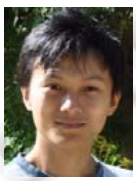
[5] 李斌,贺也平,马恒太.程序自动修复:关键问题及技术.软件学报,2019,30(2):244–265. http://www.jos.org.cn/1000-9825/5657.html [doi: 10.13328/j.cnki.jos.005657]

[6] 刘斌斌,董威,王戟.智能化的程序搜索与构造方法综述.软件学报,2018,29(8):2180–2197. http://www.jos.org.cn/1000-9825/5529.html [doi: 10.13328/j.cnki.jos.005529]



姜佳君(1992－),男,博士,副研究员,CCF 专业会员,主要研究领域为软件工程,代码调试,程序变换.

陈俊洁(1992－),男,博士,副教授,博士生导师,CCF 专业会员,主要研究领域为软件分析与测试.



熊英飞(1982－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为软件工程,程序设计语言.