软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(2):244−265 [doi: 10.13328/j.cnki.jos.005657]

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

程序自动修复:关键问题及技术*

李 斌[1,2], 贺也平[1,2,3], 马恒太[2]

[1](中国科学院大学,北京 100049)

[2](基础软件国家工程研究中心(中国科学院 软件研究所),北京 100190)

[3](计算机科学国家重点实验室(中国科学院 软件研究所),北京 100190)

通讯作者: 李斌, E-mail: libin@iscas.ac.cn; 贺也平, E-mail:yeping@iscas.ac.cn

摘 要: 程序自动修复技术能够有效地降低软件维护成本,是近年来学术研究的热点问题.待修复程序规约的刻画,对自动修复过程具有至关重要的作用.从规约的角度对程序自动修复问题和技术进行了分析梳理.从待修复程序是否具有完整的程序规约,将现有修复问题分为不完全规约、完全规约和半完全规约这 3 大类待修复问题.以 3 类抽象问题为线索,梳理了不同前提假设下修复技术面临的核心问题、问题之间的联系和技术体系中的逻辑关系.分析了不完全规约程序修复问题中高精度补丁生成、规约补全和补丁择优等问题,梳理了完全规约程序修复问题中内存泄漏、资源泄露、并发错误中的数据竞争、原子性违背、顺序违背和死锁,配置错误以及特定性能错误等具体问题及研究进展,整理了半完全规约程序修复问题中多种形式的修复具体问题及研究进展.最后分析了程序自动修复面临的机遇和挑战.

关键词: 程序自动修复;静态分析;程序规约;补丁生成;测试集

中图法分类号: TP311

中文引用格式: 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.htm

英文引用格式: 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). http://www.jos.org.cn/1000-9825/5657.htm

Automatic Program Repair: Key Problems and Technologies

LI Bin[1,2], HE Ye-Ping[1,2,3], MA Heng-Tai[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)

[2](National Engineering Center of Fundamental Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: Automatic program repair technology can effectively reduce the cost of software maintenance, which is a hot topic of academic research in recent years. Specifications description of to be fixed program plays a vital role in the automatic program repair process, this article analyses the problems and technologies of automatic program repair from specifications point of view. According to whether the specifications to be repaired program is complete, the existing repair problems are divided into three kinds of problems to be repaired, such as incomplete specifications, complete specifications, and semi-complete specifications problems to be repaired. It is analyzed that the core problems, the relationship between problems and the logical relationship of technology under different assumptions based on three kinds of abstract problems. Also analyzed issues are high-precision patch generation, specifications completion and patch selection in incomplete specifications repair, and the specific problems and progress in memory leak, resource leak, concurrency errors include data competition, atomic violation, sequence violation and deadlock, configuration error and specific performance error in

* 基金项目: 核高基国家科技重大专项(2014ZX01029101-002)
Foundation item: CHB National Science and Technology Major Project of China (2014ZX01029101-002)
收稿时间: 2018-05-10; 修改时间: 2018-06-13; 采用时间: 2018-09-02

complete specifications repair. The various specific repair problems and research progress of the semi-complete specifications repair problem are collated. Finally, the opportunities and challenges faced by automatic program repair are analyzed.

Key words: automatic program repair; static analysis; program specification; patch generation; test suite

由于现代程序规模和复杂度(scale and complexity)的上升,程序错误(program error)不可避免地存在且数量逐年上升.传统维护面临成本和维护能力不足、不可控条件下无法维护等诸多问题,这使得程序错误修复这一问题上升到新的高度,程序自动修复技术(automatic program repair)应运而生并成为研究热点.

(1) 传统维护面临成本和维护能力不足问题

统计表明,软件维护占用了软件开发成本的 50%~75%[1],其中,成本消耗最大的就是程序错误定位和修复.2006 年,Mozilla 公司的维护人员发现:每天有大约 300 个软件错误发生,其规模远远大于 Mozilla 公司的处理能力[2].现在软件发布越来越快,软件维护周期越来越短.面对越来越多的软件错误,许多公司开始寻求外部开发者的帮助,甚至通过悬赏的方式寻求缓解日益增长的软件错误问题.例如,著名 IT 公司 Mozilla[3]、谷歌[4]为较高安全等级的软件错误修复设立了专门的奖励基金,微软也建立了赏金计划[5].

(2) 不可控条件下无法维护问题

由于软件逻辑设计复杂性和运行环境的限制,当软件发生错误时,维护人员可能无法远程修复该类错误.在航天领域中应用的软件系统,当卫星、航天飞船等空间飞行器与地面的通信联络中断时存在不可控的情况.例如,2005 年,NASA 发射的“深度撞击”号探测器由于重置探测器计算机遇到一个软件通讯的小故障,使得该探测器处于失控状态.地面控制人员无法发送指令修复程序错误,最终,美国航天局宣布探测器电池耗尽已经死亡[6].

程序自动修复是一个飞速发展的研究领域,为了方便研究人员进入该领域并充分了解研究现状,国内外学者已形成多篇研究综述.其中,在 2016 年,已有国内学者玄跻峰等人对程序自动修复方法及实证研究基础进行了总结[7],但是该综述文章将研究对象限制在基于测试集的程序自动修复方法范畴,仅仅分析和梳理了 2015 年 8 月之前基于测试集的程序自动修复方法和修复的实证基础研究进展.2017 年,国内学者王赞等人[8]在综述性文献[7]的基础上进一步对修复方法进行分类总结,新增的 38 篇研究文献主要集中在 2016 年底之前的研究成果.该综述梳理了缺陷定位、补丁生成和补丁评价这 3 个阶段的程序自动修复研究成果,同时总结了该领域已有的 benchmark 缺陷库、修复工具和活跃的研究团队.国外学者 Claire 等人[9]早在 2013 年就发表了回顾型论文(review),主要分析了该团队的核心研究工作 GenProg 方法的创新和不足,以及当时程序自动修复研究领域面临的挑战.2018 年,国外学者 Martin 等人[10]主要对 2016 年底之前(除了 1 篇 AAAI 2017 对编译错误进行修复的研究文献)程序自动修复和程序动态容错两个领域的研究成果进行梳理和总结,将研究对象上升到软件自动修复(automatic software repair)的范畴.

与该领域国内[7,8]、国外[9,10]已有的研究综述相比,我们的工作有如下不同.

首先,我们从一个新的角度对程序自动修复技术领域进行了分析,以问题为导向,分析技术逻辑关系更加自然.现有的综述更倾向于对已有的程序自动修复技术在方法层面进行梳理和分类总结,阐述的是具体方法和具体问题以及相互的异同.比如,玄跻峰等人[7]将基于测试集的程序自动修复方法划分为基于搜索、基于代码穷举和基于约束求解这 3 个方面进行阐述,王赞等人[8]将补丁生成阶段划分为基于搜索、基于语义和其他这 3 类对应方法进行阐述.以上综述有利于理清具体问题的差异和每种技术体系下各类方法的发展趋势,但不利于对程序自动修复领域各类具体问题之间的联系和技术体系中逻辑关系的理解.例如,基于搜索和基于语义的修复方法其实面向的是同一类不完全规约的修复问题,即具有共同的潜在假设,认为补丁通过全部测试用例则是正确的[11].我们的工作尝试抽象出该领域面对的几类问题和不同的前提假设,然后以抽象问题作为脉络,分析各类技术体系和典型的方法.

其次,在上述综述之后,又有一些新的典型问题和方法出现,我们增加了新的有代表性的工作.例如,Le 等人[11]发现,基于语义的修复方法也同样存在过拟合问题,但某些情况下,过拟合现象和基于搜索的方法所表现的形式不同,这是对不完全规约修复问题中过拟合问题的重要说明.例如,Mechtaev 等人[12]引入了参考程序的思想缓解过拟合问题.该方法完全不依赖测试集,从而区别开了已有基于测试集的修复方法和补丁择优的方法,为不

完全规约修复问题中缓解过拟合这一关键问题提供了新的思路.我们的梳理工作也吸收了上述典型问题及其代表性研究成果,总体上新增前文综述未覆盖的研究文献 22 篇.

我们将视角扩展到整个程序自动修复的范畴,为了方便分析和说明规约刻画对程序自动修复的重要性,提出了一种基于规约的程序自动修复描述.基于该描述所面对的不同修复问题,将现有修复技术所面向的修复对象抽象为完全规约、不完全规约和半完全规约这 3 大类问题.以上述问题分类为线索,并结合基于规约的程序自动修复描述,梳理了各类技术体系的发展现状和需要研究的核心问题.本文的主要工作内容如下.

(1) 提出了一种基于规约的程序自动修复描述,说明了程序规约的刻画在修复过程中的重要性.从一个新的角度对程序自动修复技术领域进行分析,根据描述中对待修复程序刻画的程序规约 *S*(specification)是否完整,将现有修复技术面向的问题抽象为完全规约、不完全规约和半完全规约这 3 大类.由于能否完整地描述和刻画待修复对象的程序规约的不同,各类程序自动修复技术的应用场景、方法关注点和困难点存在很大区别.

(2) 在梳理了不完全规约修复问题和方法,即基于测试集的程序自动修复方法最新进展的基础上,重点分析了该类方法面临的高精度补丁生成、补丁判定中的规约补全和补丁择优等核心问题和已有的解决方案.

(3) 梳理了完全规约和半完全规约的修复问题和方法,对其中完全规约的程序自动修复热点问题(内存泄漏、并发错误中的数据竞争、原子性违背、顺序违背和死锁、资源泄露、配置错误)以及特定性能错误等具体问题类型和研究进展进行了梳理.

本文第 1 节给出一种基于规约的程序自动修复描述,并基于描述给出问题分类.第 2 节介绍不完全规约的程序自动修复问题及方法.第 3 节介绍完全规约的程序自动修复问题及方法.第 4 节介绍半完全规约的程序自动修复问题及方法.最后在第 5 节进行总结和展望.

1 程序自动修复描述

1.1 相关概念

程序自动修复技术针对各类不同的错误(error)修复场景,将整个修复过程自动化.一个程序错误(program error)是指程序的预期行为和实际执行时所发生情况之间存在的一种偏差(deviation)[13].该定义引入了一个概念:预期行为,其实,一系列预期行为的集合就是程序规约(program specification),程序规约可以是自然语言书写的文本、形式化的逻辑公式,或者测试集(test suite)等.有些文献中提到一个和程序规约非常相近的概念:测试预言(test oracle),测试预言可以确定一个程序执行结果是否正确,通过测试用例的预期结果与实际执行结果的对比机制来判断测试用例的结果是否通过[14].但测试预言和程序规约之间存在一定的区别:测试预言只与程序的输出相关,而程序规约还包括程序的输入区间、程序的内部逻辑要求和非功能属性等规范.因此,测试预言是程序规约的一个子集.例如,一个程序规约要求遍历一个输入的字符串中是否包含字符 *A*.若该程序的某个程序变体(program variant)不是遍历输入的字符串,而是直接判断该字符串的某一位(例如第 1 位)是否为 *A*.如果给定的输入集字符串恰好都是以 *A* 开头,则该程序变体和原程序在给定的输入集对应的测试预言相同,实则两者逻辑语义截然不同.

1.2 程序规约与程序自动修复

程序自动修复是一种将不符合程序规约的错误语义自动转换为符合程序规约的技术.一般的程序自动修复方法包括错误定位、补丁生成和补丁判定等步骤.例如,输入一个带有 bug 的程序(buggy program)和测试用例集(至少 1 个测试用例执行不通过,执行未通过的用例检测出了待修复的 bug,这里将测试集作为程序规约),程序自动修复工具通过错误定位技术确定可能的出错位置,然后利用补丁生成技术产生一系列候选补丁,最后利用程序规约构造判定条件输出 0 个、1 个或多个正确补丁,最终输出的补丁使得整个修复后的程序执行行为符合规约.

根据我们的了解,目前的相关研究文献针对程序自动修复大多是具体问题的步骤描述和示例说明,并没有给出统一的描述和刻画.为了说明程序规约的刻画在自动修复过程中的重要性,以及更加方便地说明程序自动修复中出现的各类问题及方法特点,下面我们给出一种基于规约的程序自动修复描述.假设最简单的一种情况,待修复的程序 *BP*(buggy program)只包含 1 个错误,且修复该错误所需的补丁只有 1 条语句.修复之前程序 *BP* 不满足程序规约 *S*,修复过程中产生的补丁语句集合表示为 *P*=*patch*(*L*,*OP*,*C*),其中 *L*(location)表示 bug 程序 *BP* 的出错位置,也是修复时打补丁的位置;*OP*(operater)表示补丁生成的操作符,包括增加、删除和修改某条语句等变换的操作,也可以理解为修复模板;*C*(content)表示补丁语句所包含的内容,是操作符 *OP* 的操作对象,也可以理解为修复模板的参数.则程序自动修复可描述为:

1) 假设给定程序 *BP* 和对应的程序规约 *S*,且 *BP* 不满足 *S*(即已经发现程序 *BP* 包含一个错误).
2) 修复过程是求解函数 *P*=*patch*(*L*,*OP*,*C*),即通过错误自动定位、程序分析等技术计算出错位置 *L*,利用程序规约 *S* 或归纳方法总结适合的修复操作或修复模板 *OP*,基于规约 *S* 指导的程序分析技术或其他方法获取修复模板的参数或者补丁内容 *C*,然后求解补丁生成函数产生候选补丁集合 *P*.
3) 使得程序 *BP* 应用上述补丁集合 *P* 中的特定补丁后满足程序规约 *S*,即判定打补丁后的程序语义和程序规约 *S* 等价,并输出该符合判定条件的补丁.

通过以上描述可以看出,修复的前提假设需要根据程序规约 *S* 判定程序存在错误.需要特别指明的是,上述步骤 2)的补丁求解在很多情况下其实是隐含程序规约指导的,最后也是通过程序规约 *S* 判定补丁的质量.下面以内存泄露错误的修复[15]为例,使用如图 1 所示的示例程序 *BP* 进行说明.

```
1  #include ⟨stdio.h⟩                    9        *b=10;
2  #include ⟨stdlib.h⟩                  10       a=20;
3  void f(int a){                       11       }
4        int *b=(int*)malloc(sizeof(int));  12  }
5        if (a==0){                      13  main(){
6              free(b);                  14        int a=10;
7        }                               15        f(a);
8        else{                           16  }
```

Fig.1 An example of memory leaks

图 1 一个内存泄露错误的示例

1) 给定程序 *BP*,通过先验知识,我们可以完全明确地描述,不发生内存泄露错误的正确程序规约是“对于任意执行路径和任意插入的释放语句(*free*),要保证释放前已分配、无双重释放、无释放后使用这三者同时满足”,则程序 *BP* 的完全规约 *S* 描述为和原程序语义等价并不发生内存泄露.

BP 程序中,第 4 行申请了内存空间;紧接着,第 5 行的条件语句之后第 1 个分支对该空间进行了释放,而第 2 条分支对该空间使用后并未释放.从而确定 *BP* 程序包含一个内存泄露错误(具体可用测试技术或程序分析技术发现该错误),即已知前提是 *BP* 不满足规约 *S*.

2) 修复过程中,对函数 *P*=*patch*(*L*,*OP*,*C*)进行求解.首先确定出错位置 *L*,这里,根据规约 *S* 指导反复使用数据流分析技术获得.具体通过过程间分析技术,使用前向数据流分析检查释放前已分配,后向数据流分析检查释放后未使用,前后向数据流分析检查无双重释放的位置,求解出错位置 *L* 为第 9 行之后和第 11 行之前的区间.根据规约 *S* 确定修复该类错误主要是在合适的位置插入内存释放语句,即修复模板 *OP* 为 *free*(*C*).在数据流分析过程中,通过处理各种复杂的计算(如循环、全局变量、多重分配、空指针判断等问题)求解补丁内容 *C* 是指针 *b*,即修复模板的参数为指针 *b*.求解函数 *P*=*patch*(*L*,*OP*,*C*)获得候选补丁集合 *P*,即“在第 9 行之后和第 11 行之前的区间插入语句 *free*(*b*);”.

3) 基于在安全插入区间尽早释放内存的原则,最终判定补丁“在第 10 行之前插入语句 *free*(*b*);”符合程序规约 *S*,修复完毕.事实上,由于该类内存泄露错误的程序规约是完全规约,因此采用比较精确的分析技术和完全规约能够求解出高精度补丁,另外一个候选补丁“在第 10 行之后插入语句 *free*(*b*);”同样符合程序规约 *S*.

针对补丁包含多条语句的情况,可以在单条补丁语句表示 *patch*(*L*,*OP*,*C*)的基础上进行扩展.在多个单条补

丁语句之间增加先后顺序描述,从而组合出一个包含多语句的补丁程序块.

综上所述,程序自动修复中,对程序规约 S 的刻画是非常重要的问题,直接影响到程序自动修复关注的核心过程:错误定位、补丁生成和补丁判定这 3 个方面.错误自动定位技术作为一个独立的研究领域已经有 30 多年的研究历史,Wong 等人、虞凯等人、陈翔等人[16−18]给出了研究综述,很多程序自动修复方法使用基于程序频谱的错误定位技术[19]获得可能出错位置排序,这里不再赘述.我们从程序规约的角度对程序自动修复研究内容进行梳理,具有问题和技术逻辑关系清晰的优势.本文梳理文献范围涵盖软件领域的顶级会议(OSDI、POPL、ICSE、FSE、ASE、PLDI、ISSTA、CAV、AAAI 等)和顶级期刊(IEEE Trans. on Software Engineering,简称 TSE)等,以该领域的问题为导向,从程序规约的角度重点梳理了补丁自动生成和补丁判定技术典型的相关研究成果和存在的核心问题.

1.3 程序自动修复问题分类

程序自动修复的目的是对软件开发和维护过程中出现的程序错误,在源代码级别进行自动修复,图 2 给出了问题分类框架.

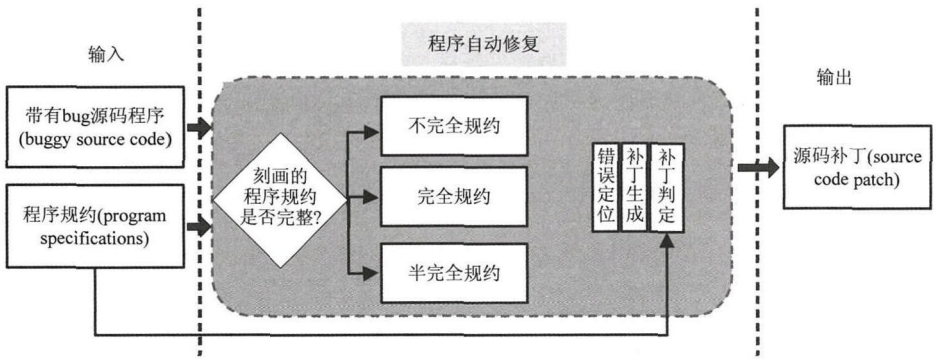

Fig.2 Framework of automatic program repair diagram

图 2 程序自动修复框架示意图

由于在实际场景下针对待修复程序能够获取到的程序规约各不相同,从而程序自动修复技术面向的研究问题、大的前提假设、具体方法关注点和困难点也有很大区别.鉴于程序规约 S 对自动修复的至关重要性,本文根据是否能够完整地刻画待修复程序的规约 S,将程序自动修复技术面向的问题分为不完全规约、完全规约和半完全规约的程序修复问题 3 大类.

(1) 不完全规约的程序自动修复问题是目前研究成果最多的一类问题.

该类问题就是基于测试集的修复技术面向的问题,其将测试集或者通过测试集提取的条件约束作为最终判定自动生成补丁质量的程序规约 S.一方面,现实世界中不易获取充足的测试用例;另一方面,更关键的问题是测试集并不能完整地表示程序规约[20,21].基于测试集的修复技术可进一步分为基于搜索和基于语义的修复技术两种类型,两者具有共同的潜在假设,即如果生成的补丁通过全部测试用例,则都认为该补丁正确[11].

不完全规约的修复问题中,基于搜索的经典修复技术直接将测试集作为程序规约 S 构造搜索算法来判定候选补丁搜索空间中的补丁质量,目标是输出通过全部测试用例的程序补丁,其标准结构符合生成-检测类型(generate-and-validate).该类修复技术要求输入一个带有 bug 的程序及其测试用例集,包含至少 1 个测试用例因为待修复的 bug 而无法通过测试,输出 0 个、1 个或多个通过测试集的补丁.打过补丁的程序通过整个测试用例集中原来未通过的用例表示修复了该 bug,同时,通过剩余的原来已经通过测试的用例表示该补丁未引入新的错误.输出 0 个补丁表示未能成功修复;1 个表示该修复方法可生成唯一针对待修复 bug 的补丁;输出多个补丁表示针对当前用例集存在多种修复补丁的情况.由于测试集本质上不能表示完整的规约,通过全部测试用例的补丁往往包含疑似正确补丁(plausible patch)[22],即候选补丁通过给定的测试集并不能保证测试集之外的功能

也符合原程序期望的语义.因此,该类修复技术目前的主要问题是输出的补丁正确率较低,存在过拟合问题(overfitting problem)[23],即生成的疑似正确补丁只是拟合了测试用例集而并非完整的原程序期望语义.

不完全规约的修复问题中,基于语义的修复技术通过符号执行技术和测试集提取条件约束,然后转化为约束求解问题,并结合程序合成(program synthesis)技术生成补丁.该类修复技术将基于测试用例提取的全部语义约束作为程序规约 S,对比基于搜索的修复技术需要处理庞大的候选补丁搜索空间,其更有针对性地提取约束条件并采用比较精确的分析技术生成补丁.由于基于搜索和基于语义的修复技术具有共同的潜在假设,基于语义的修复技术也不可幸免地存在过拟合问题[11].

不完全规约的修复问题特性,使得不断提高输出补丁的精度成为基于测试集的修复技术面临的核心问题.在补丁生成阶段,已有很多细粒度的补丁生成方法致力于提高生成补丁的精度.在补丁判定阶段,存在疑似正确补丁的现象[22]被描述为过拟合问题[23],是该阶段需要克服的主要困难.即在生成的多个补丁均通过全部测试用例或者基于测试用例提取的全部约束条件的情况下,人工分析后发现依然包含错误的补丁.本质上是测试用例集或者约束条件刻画的程序规约 S 的区分度不够,无法进一步区分输出的潜在错误补丁.

缓解过拟合问题可以分为规约补全问题和补丁择优问题两类.两类问题的研究目的都是为了提高输出补丁的正确率.规约补全问题是研究程序规约本身,如何通过额外的程序期望语义信息直接加强规约本身来增强基于已有测试集构造的补丁判定条件.补丁择优的研究内容与增强规约本身无关,在已刻画的程序规约 S 无法识别潜在错误补丁的情况下,如何通过其他规约之外的信息进一步识别潜在错误补丁,同时尽可能地减少对正确补丁的误报,从而进一步提高输出补丁的正确率是其研究的核心内容.目前,解决补丁择优问题的主要技术包括补丁最小化、利用启发式规则、基于概率模型的分类和排序思想等.文献[24]指出,选择最小化的补丁作为最终采用的补丁,其基本假设为最小化的补丁引入新错误的可能性最小;文献[25]指出,选择和原程序相似度最高的补丁作为最终采用的补丁,其基本假设为原 BUG 程序和修复后的正确程序之间语义高度相似等.

(2) 完全规约的程序自动修复问题.

该类问题的特点是能够完全明确地描述待修复程序的完整规约,即刻画的程序规约 S 和原程序语义等价且修复后不引入原程序语义变化.针对该类问题,主要枚举了几种明确的特定错误类型,能够完整地刻画程序规约,其规约 S 和原程序语义等价且修复特定错误不引入原程序语义变化,其中,不发生特定错误能够进行明确和完整的描述.该类问题的前提假设是开发人员事先已经确定错误类型,由于面向具体问题的不同而方法各不相同,具体方法修复能力只针对特定错误类型有效.目前已有的完全规约程序自动修复问题包括内存泄漏、并发错误中的数据竞争、原子性违背、顺序违背和死锁、资源泄露、配置错误以及特定性能错误的修复问题.例如,针对内存泄露[15]的特定类型错误,其完整的程序规约 S 和原程序语义等价并且修复内存泄露错误不引入原程序语义变化,无内存泄露的完整描述前文已述.若修复后的程序满足以上规约,则不存在内存泄露的同时满足原程序的功能要求.针对并发错误中常见的子类型,包括死锁、数据竞争、原子性违反和顺序违背这 4 类错误[26],不发生以上 4 类错误的原因也有明确的定义并可以完整地刻画程序规约,其程序规约 S 和原程序串行语义等价并且不发生以上并发错误.

(3) 半完全规约的程序自动修复问题.

除了明确的不完全规约和完全规约的程序自动修复问题以外,剩余的问题都属于半完全规约的程序自动修复问题,即修复中刻画的程序规约 S 是否完整不确定.针对该类问题的修复,往往先假设具有完整的程序规约 S,通过判定的修复补丁还需要事后进一步的人工确认或者正确性证明.一些文献中基于契约(contract)[27]进行程序自动修复,假设这些契约是完整的程序规约,则用于判定修复的正确性之后,再人工地确认或进行正确性证明.契约是指编程元素(例如类或者函数)之间存在的某种固有约定,具体表示形式为前置条件(precondition)、后置条件(postcondition)和类不变式(class invariant)等程序局部要保持的性质,其完整性不确定.还有一些文献需要使用特定语言手工编写程序规约,其手工编写的完整性不确定.甚至有一些文献中使用的规约来自于神经网络模型,神经网络模型通过计算机能够理解的特征来刻画程序的语义特征,这种学习模型的质量取决于特征向量的选择和训练集的质量,该模型表示的程序规约 S 的完整性不确定.

2 不完全规约的程序修复问题及方法

正如第 1.3 节所述,不完全规约的修复问题主要是基于测试集的程序自动修复技术面向的问题.基于测试集的程序自动修复技术包括基于搜索和基于语义两类修复方法,其具有共同的潜在假设,将通过全部测试用例的补丁认为是正确的[11].由于测试用例集表示的不完整规约导致最终产生疑似正确补丁,这些补丁只满足了测试集而不能保证同时满足测试集之外的程序期望语义,该问题被称为过拟合问题[22].

2.1 过拟合问题分析

2015 年,Qi 等人[22]发现,基于测试集的修复技术中基于搜索的修复方法修复正确率并不高,通过测试集中全部测试用例的补丁并不是必然正确,通过全部测试用例却依然错误的补丁被称为疑似正确补丁.疑似正确补丁虽然通过了选定的测试用例集,但并不能保证符合测试集之外的其他程序规约.产生大量的疑似正确补丁不但不能达到程序自动修复的目的,反而增加了大量人工确认输出补丁正确性的工作量.自此之后形成了一个明显的转折点,如何提高该类修复方法输出补丁的精度成为研究的关键问题.Smith 等人[23]进一步分析认为,补丁的质量和修复中所使用的测试用例集的覆盖率成正比.

2016 年,Long 等人[28]解释了基于搜索的自动修复方法产生的补丁质量较低的原因.在整个候选补丁搜索空间中,正确的补丁是稀疏的,而疑似正确的补丁相对来说更加丰富.在他们的实验中,针对同一个错误经常有成百上千个疑似正确补丁,其中只有一两个是正确的补丁.因此,一方面,修复工具很难从大量的搜索空间中找出正确的补丁;另一方面,虽然更大和更丰富的搜索空间包含更多的绝对数量正确补丁,但被准确识别出的正确补丁却更少.因为更多的候选补丁增加了判定时间,占比更多的疑似正确补丁的验证阻碍了正确补丁的发现.

2018 年,Le 等人[11]发现,基于语义的修复方法也同样存在过拟合问题,但某些情况下过拟合现象和基于搜索的方法的表现形式不同.他们对比了基于搜索的修复方法中存在的过拟合问题,通过 IntroClass[29]和 Codeflaws[30]两个 benchmarks,以及 2016 年 Mechtaev 等人[31]提出的基于语义修复方法 Angelix 分析过拟合问题.他们假设测试集中测试用例的总量和来源、未执行通过的测试用例数量、基于语义的修复方法设计方式等和对应的过拟合问题相关.实验结果表明:一些情况下,基于搜索和基于语义的修复方法过拟合结果一致;另一些情况下,两者过拟合结果不同.他们发现,使用多种程序合成引擎(program synthesis engine)是基于语义的修复方法提高修复能力的一种可能措施.这一结论与 2017 年 Le 等人[32]得出的结论一致.他们通过分析多个过拟合的实例和实验现象进一步指出,基于语义的修复方法存在过拟合问题的一种可能原因是底层程序合成引擎采用过于保守的策略,其生成补丁时直接使用第 1 种求解合成的方案,而没有其他可替代的备用方案.

2.2 基于测试集的程序修复新进展

下面分别从补丁生成和补丁判定两个阶段介绍不完全规约程序修复问题的研究进展.由于不完全规约的修复问题特性使得修复产生的补丁精度不高,在补丁生成阶段,如何生成高精度补丁是核心问题;在补丁判定阶段,规约补全和补丁择优则是需要研究的关键问题.

2.2.1 补丁生成

1) 基于搜索的补丁生成方法

2016 年,Le 等人[33]从人工修复的历史信息中挖掘模板来指导高质量候选补丁的生成,通过计算挖掘模板的频度,赋予修复模板对应的权重.权重信息高效地指导候选补丁生成,同时也有助于对输出补丁排序,从而提升修复能力和修复效率.具体使用 AST 级别的 commit 比较获取修复历史的变化情况,同时将挖掘对象限制在单代码行的改变,这有利于过滤掉 bug 修复操作中包含增加新特性或功能等代码块的影响.利用突变测试技术(mutation testing)生成候选补丁,并根据匹配挖掘的 5 类规则的频度信息对输出补丁质量进行排序.基于第 1.2 节我们提出的补丁语句表示 $patch(L,OP,C)$,之前代表性的方法 GenProg[24]相当于将 *OP* 模板和 *C* 补丁语句所包含的内容在语句级别通过变异和交叉操作进行粗粒度的考虑,而 PAR[34]方法将 *OP* 模板单独提出并细化为 10 种类型,该方法在前文基础上进一步赋予 *OP* 模板权重.因此,该方法最终获得比 GenProg 和 PAR 两种工具更好的修复效果.

2017 年,Suzuki 等人[35]提出了 REFAZER,从正确修复的实例程序(example)中学习细粒度的修复模板(称作程序转换(program transformation))来指导包含同类错误问题程序的修复.例如,接口误用问题在修复时需要在多处调用位置使用新接口替换.该类问题具有相同的修复方式(替换接口),但在不同程序实现中具体使用的函数名称、变量名称或表达式不同.REFAZER 具体利用领域专用语言(domain-specific language,简称 DSL)描述程序语法转换,形成一系列在 AST 级别的变换序列,事实上,相当于刻画补丁语句表示 *patch*(*L*,*OP*,*C*)中的修复模板 *OP*.为了减少漏报(修复模板太抽象)和误报(修复模板太具体),在抽取修复模板时需要在具体和抽象中间取得一个平衡,REFAZER 借鉴了归纳编程(inductive programming,简称 IP)和样例编程(programming-by-example,简称 PBE)的思想.REFAZER 利用 720 个学生参加的 4 个编程任务所产生的数据集进行实验,可以帮助学生修复 87%的同类错误.

2017 年,熊英飞等人[36]针对不完全规约程序修复问题,聚焦在条件错误,提出了 ACS,以专门解决高精度补丁生成和缓解程序修复过拟合问题.鉴于前述 Prophet、Qlose 和 Angelix 等方法利用候选补丁正确率排序的粒度过大问题,他们将修复对象聚焦在单变量条件类错误问题,利用启发式方法合成待修复条件语句,并按正确性排序.具体基于变量使用的局部性原理,利用变量间的依赖关系排序确定条件表达式中应该使用的候选变量,并利用文本分析技术对相关的 API 文档信息分析进一步过滤候选变量.利用挖掘技术对其他项目中相似上下文代码片段所使用的谓词按频率排序确定候选谓词,最后,通过正反测试用例获取测试预言(test oracle),合成待修复的条件语句.结合补丁语句表示 *patch*(*L*,*OP*,*C*)来说明,出错位置 *L* 利用传统程序频谱的方法[19]并结合已有的基于谓词切换的错误定位技术[37],兼顾程序规模和定位精度,高效地对条件类错误进行定位.条件类错误的修复模板 *OP* 是相对固定的.该方法对合成条件补丁的素材 *C*,包括变量(或表达式)、谓词等进行了更细粒度的重点研究,因此获得了更高的修复准确率.在 Defects4J benchmark 上的实验结果显示其修复准确率在 78.3%,显著高于前述方法(一般修复准确率低于 40%).

2017 年,Ripon 等人[38]提出了专门修复面向对象程序错误的自动修复方法 ELIXIR,其主要关注函数调用和表达式错误.ELIXIR 利用基于频谱的定位技术 Ochiai 确定可能的出错位置 *L*,通过收集到的对象和变量等基础素材 *C*,具体使用 8 种修复模板 *OP* 进行变换产生候选补丁(具体修复模板包括改变变量类型、改变表达式返回值状态、空指针检查等).该方法的主要贡献是利用机器学习模型对候选补丁进行排序,从而提高用于补丁判定的候选补丁质量.特征向量的选择主要关注出错语句的上下文,包括标识符在上下文中的使用频率、标识符最近使用位置和出错语句位置之间的距离、修复补丁中是否包含与上下文命名相似的元素、修复使用的标识符是否在错误报告中引用这 4 种特征.在 Defects4J benchmark 上的实验结果显示其修复准确率在 85%,在自己提供的 Bugs.jar 实验数据集上正确修复率在 57%.

2017 年,Chen 等人[39]提出从 JAVA 代码中自动提取细粒度状态抽象信息(类似于 Eiffel 语言中手工编写的契约信息),从而指导生成高质量候选补丁,形成的方法 JAID 属于基于搜索的修复方法.该方法通过函数纯度分析(purity analysis)等静态分析方法提取状态抽象的断言信息,基于状态抽象信息形成的断言和测试确定出错的可疑位置 *L*.补丁生成主要是基于状态抽象断言信息对出错区域做变换,设计了 5 个修复模板 *OP*,具体类似增加一个布尔条件对 action 语句和已有旧语句进行 if-else 代码块组合.补丁内容 *C* 在该方法中其实是 action 语句,action 语句的求解通过对已有语句进行语法和语义的修改,使其符合相应状态抽象断言,具体修改操作包括修改状态、修改表达式、修改一条语句和修改控制流这 4 种.JAID 在 Defects4J benchmark 上进行实验,产生通过测试集的修复补丁对应 31 个 bug,并基于启发式规则和出错定位可疑值信息对输出补丁进行排序,通过人工进一步确认排名靠前的输出补丁,其中 25 个修复补丁是正确的.

2017 年,Qi 等人[40]提出利用已有代码(与错误代码语法相关的数据库代码数据)生成补丁的方法 ssFix.针对如何生成高质量补丁的核心问题,一些研究假设生成补丁所需的正确语句或者表达式可以在出错项目本地代码找到[24]或者在其他项目的代码中找到[41],这些正确语句或者表达式可以直接用于构造补丁.另一些研究指出,通过语义搜索获得修复出错语句相关的正确语义代码片段,利用这些语义片段提高生成补丁的质量.CodePhage[42]和 SearchRepair[43]属于该类型的研究工作.鉴于上述基于语义的搜索方法开销太大且不能处理规

模较大的程序,ssFix 通过搜索和出错语句上下文语法相关的代码片段,形成一种轻量级的利用语法相关的代码片段提高生成补丁质量的修复方法.具体 ssFix 使用 GZoltar 确定疑似出错位置 *L*,通过结构相似性和概念相似性搜索与出错语句上下文语法相关的代码片段,具体设定替换、插入和删除这 3 种修复模板 *OP*,对语法相关的程序片段中的元素进行变换生成补丁.ssFix 在 Defects4J benchmark 上进行实验,能够产生通过测试集的修复补丁 20 个,并且输出每个补丁平均需要 11min.

2018 年,Ming 等人[44]提出了 CapGen 形成上下文感知的补丁自动生成技术.基于搜索的程序自动修复技术不能高效地生成正确补丁,主要原因一方面是候选补丁搜索空间可能本身就不包含正确补丁;另一方面是搜索空间太大,使得在限定的时间阈值未找到正确补丁.该方法有效利用了更细粒度的 AST 上下文信息(context)生成补丁,提出了 3 种模型获取更细粒度的补丁修复素材,同时利用上下文感知信息对变异操作进行排序,从而限制搜索空间.结合补丁语句表示 *patch*(*L*,*OP*,*C*)来说明,出错位置 *L* 利用传统程序频谱的方法进行定位[19].该方法的创新是修复模板 *OP* 由上下文信息指导选择变异操作,*C* 补丁语句所包含的内容相当于在表达式级别(与语句级别相比更细粒度)获取更细粒度内容,主要是有效利用 AST 上下文信息形成 3 种模型选择细粒度内容合成补丁.给出的实验结果 CapGen 可达到 84%的精度,同时过滤掉 98.78%疑似正确的补丁.

2018 年,Hua 等人[45]针对基于搜索的修复方法中庞大的候选补丁搜索空间在遍历时需要迭代的对候选程序重复编译和重复执行的低效率问题,提出了在测试执行时按需生成补丁的方法 SketchFix.和迭代地重复编译和重复执行每个完整候选程序的方法不同,SketchFix 在抽象语法树级别将程序错误的修复部分转换成若干类似组件方式的概要(sketch),每个类似组件方式的概要独立编译,并和原程序正确的部分以“拉抽屉”的形式按需组合出新的候选程序.该方法集成的考虑补丁生成和补丁判定阶段精简搜索空间,同时使用细粒度的抽象语法树级别转换技术生成语义更接近原程序的高质量候选补丁.在 Defects4J benchmark 数据集上,SketchFix 在 23min 内成功修复了 357 个错误中的 19 个,在整个搜索空间中找到第 1 个通过判定的补丁平均使用 1.6%的重复编译和 3.0%的重复执行次数.

2) 基于语义的补丁生成方法

2015 年,Mechtaev 等人针对如何生成高精度补丁的核心问题,提出了生成更简化补丁的自动修复方法 DirectFix[46].其基本假设是:相比复杂的修复操作,更简化的补丁对原程序正确语义修改得更少,简单的补丁引入新错误的可能性更小.DirectFix 不再考虑为每个可疑出错位置枚举候选补丁,而是更高效地将错误定位和补丁生成合并在一起,作为部分最大可满足性问题进行求解,直接选择满足约束的最简化补丁作为输出.DirectFix 在 SIR(software-artifact infrastructure repository)数据集[47]和 Coreutils 数据集[48]上实验,能够成功产生 59%的 bug 补丁,其中 56%是正确的补丁;同时,DirectFix 输出的正确补丁大多数比 SemFix[49]更简单.

2016 年,Mechtaev 等人[31]提出了轻量级符号执行技术处理规模更大的程序中错误修复的方法 Angelix.具体利用测试用例集驱动可控的符号执行技术(controlled symbolic execution)收集路径条件,将通过测试用例的可疑出错语句期望输出约束称为天使森林(angelic forest),把基于测试集获取的修复约束(repair constraint)作为程序规约 *S*.与之前的同类修复技术 SemFix[49]和 DirectFix[46]相比,该方法的符号执行更加轻量级,只对可疑出错的表达式进行符号化而不是对程序输入符号化来获取整个程序的语义信息.轻量的修复技术在提高效率的同时,更重要的是能够处理大规模的程序修复问题.结合补丁语句表示 *patch*(*L*,*OP*,*C*)来说明,事实上,该方法中可疑出错语句的位置 *L* 是作为算法输入手工给定,*OP* 和 *C* 通过符号执行的路径语义信息和约束求解获取,程序规约 *S* 由修复约束(repair constraint)表示,但其规约的完整性依赖于提供的测试用例集.同时,在约束求解过程中使用近似值也可能增加修复的不完整性.开发的 Angelix 修复工具能够进行多点程序修复(multiple buggy locations,即 bug 存在于多条语句之中),同时能够自动修复著名的心脏滴血漏洞(heartbleed vulnerability),在 GenProg Benchmark[50]上的实验显示其修复准确率为 35.7%.

2017 年,玄跻峰等人[51]针对不完全规约程序修复问题,聚焦在条件错误,提出了基于语义的修复方法 Nopol.该问题的研究和 Nopol 方法的更早版本在 2014 年由 DeMarco 等人[52]提出.结合补丁语句表示 *patch*(*L*,*OP*,*C*)来说明,其修复对象的程序规约 *S* 是基于测试集提取的条件约束.该方法通过天使修复定位(angelic fix

localization)来确定修复位置 *L*.修复模板 *OP* 是对已有的条件语句进行修改,或者在已有语句块之前添加卫士前置条件(guard precondition).通过测试用例执行时收集的预期条件约束转化为约束求解问题获取补丁内容 *C*,基于组件的方法合成条件补丁.在给定的数据集上修复精度表现良好,可以正确修复 17 个条件错误中的 13 个.

2.2.2 补丁判定

补丁判定阶段,基于搜索和基于语义的修复方法都存在过拟合问题[11,22],缓解过拟合问题可以进一步分为规约补全问题和补丁择优问题两类.两类问题的研究目的都是为了提高输出补丁的正确率.两类问题的区别在第 1.3 节已经介绍.

1) 规约补全问题和方法

2017 年,Qi 等人[53]提出了 DiffTGen,利用测试用例增强方法缓解修复的过拟合问题.首先,通过生成测试输入识别过拟合的补丁,这些测试输入未覆盖原 BUG 程序和打过候选补丁的程序之间的语义差别;然后,测试打过候选补丁的程序中存在语义差别的部分,并生成新的测试用例,其基本假设是增加新的测试用例对基于测试集的自动修复技术输出高精度补丁有益,将这些新测试用例加入测试集中直接加强了用于最终补丁判定的程序规约 *S*.

直接增加测试用例并不必然对输出高精度补丁有益,增加更多数量的已通过测试用例会使得修复方法产生正确补丁更困难[22],而增加合适数量的执行失败测试用例在一些情况下反而更有益[54].测试集如何影响修复过程已有一些研究成果,测试集包含测试用例的数量严重影响修复方法输出的补丁质量:测试用例数量太少,会导致输出的补丁删除测试集未覆盖的功能[22,28];测试用例数量太多,会使整个修复过程变慢[9,54,55].合适数量的测试用例尽可能多地覆盖程序语义并减少冗余,对成功地进行错误修复非常重要[55,56].测试集的覆盖率也可能影响输出补丁的质量.文献[23]认为,高覆盖率的测试集对基于搜索的修复方法输出高质量补丁有益.文献[57]的研究结果指出,更高的条件覆盖率比语句覆盖率在防止输出错误的补丁方面更有效.

2018 年,Mechtaev 等人[12]针对过拟合问题提出了 SemGraft 方法,引入了测试用例集之外的参考实现程序作为程序规约 *S*,即参考实现程序表示功能和 buggy 程序相同但实现算法不同的程序.该方法将补丁好坏的判断转化为语义等价检测问题,利用反例制导的符号执行技术求解生成补丁,若修复后的程序和对应的参考实现程序之间语义等价,则产生的补丁正确.例如,加法程序和乘法程序功能语义等价但实现算法不同,若加法程序出错,产生的补丁修复加法程序后和乘法程序语义等价,则判定为该补丁正确.该方法相当于将抽取的参考程序语义符号摘要作为最后补丁判定的程序规约 *S*,其假设基于参考程序刻画的程序规约比测试集表示的程序规约更倾向于完整.在给定的 Busybox 和 Coreutils 两个项目的实验数据集中,SemGraft(修复了 12 个错误)比基于测试用例集提取符号约束进行补丁判定的 Angelix 方法(仅修复 4 个错误)更有效,因此也说明参考程序语义代表的规约比给定的测试用例集表示的规约更完备.

2) 补丁择优问题和方法

2016 年,针对过拟合这一关键问题,Tan 等人[58]从自动生成的候选补丁中学习一般的和领域无关的反模式(anti-pattern),利用这些反模式排除包含非法修改的候选补丁.相当于在待验证的程序规约 *S* 中枚举了一系列反例,利用过滤反例附加条件来提升修复质量和性能.Antoni 等人[25]引入程序距离(program distance)的概念来解决过拟合问题.当测试用例集无法进一步区分已通过测试的多个补丁正确与否时,该方法假设修复后的程序和原程序相似度越高,则补丁越趋向于正确.将识别补丁的好坏问题转化为补丁质量排序问题,计算各个打补丁程序和原程序的距离进行排序(距离越小,则相似度越高),相当于在测试集代表的程序规约 *S* 无法进一步区分已通过测试的补丁好坏时,根据相似度计算的排序结果进一步补丁择优.具体开发了 Qlose 修复工具,利用形式化方法刻画原程序(bug 程序)和候选补丁程序的 2 种语法距离和 3 种语义距离,找出通过全部测试用例同时语法和语义距离更接近原 BUG 程序的候选补丁作为最终输出的补丁.Fan 等人[59]利用学习人工修复的正确补丁获得应用独立的概率模型,利用该模型对候选补丁进行排序,从而确定补丁质量.假设补丁的正确性包括补丁本身和上下文交互两部分,该方法通过这两个部分特征学习获得最大似然估计的概率模型,相当于在测试集代表的程序规约 *S* 无法进一步区分已通过测试的补丁好坏时,根据该模型对输出补丁排序进一步择优.开发的 Prophet 修

复工具能够面向真实的大规模程序(成百上千万行代码体量)进行自动错误修复.在 GenProg Benchmark[50]上的实验显示,其修复准确率为 38.5%.

2018 年,熊英飞等人[60]提出一种全新的基于测试用例执行行为相似度进行补丁择优的方法.基于测试用例的补丁质量判定方法只关注给定测试的输入、对应的实际输出结果,并与预期结果比对,却对测试用例的执行过程关注不够.该团队将补丁择优问题转化为分类问题,通过测试用例执行行为的相似度,抽象出一个模型进行分类.

- 首先观察到两个准则.
  1) 若两个测试用例具有相似的执行行为,则两者趋向于相同的执行结果(例如触发相同的错误或者均是正确执行行为).
  2) 未打补丁前执行通过的测试用例,在打补丁前后执行行为趋向于相似;未打补丁前执行失败的测试用例,在打补丁前后执行行为趋向于不同.
- 然后,一方面通过生成可达修复区域的定向输入,结合准则 1)获得测试预言(test oracle)形成新的测试用例,从而加强了原来的测试用例集表示的程序规约 *S*;另一方面,利用准则 2)并确定合适的相似度阈值构建分类模型,对通过全部测试用例的多个补丁进一步分类和补丁择优.

该方法相当于在原测试集代表的程序规约基础上,一方面通过新生成的测试用例直接加强测试集表示的程序规约 *S*;另一方面,通过分类模型增加附加条件进行补丁择优.利用 jGenProg、Nopol、Kali 和 ACS 等程序自动修复工具产生的 130 个补丁数据集上进行有效性验证.该方法可识别出 56.3%的错误补丁,并且不会产生误报,即不会将数据集中任何正确的补丁识别为错误的.

2.3 小 结

针对基于测试用例集的程序自动修复方法,一方面由于其修复对象是一类通用问题(如该类方法可修复多种多样的程序错误类型),生成正确率更高的候选补丁受到面向问题的限制很大,往往搜索空间中正确的补丁是稀疏的且大体量的搜索空间严重影响搜索效率;另一方面,测试用例集直接作为程序规约 *S* 用于判定自动修复工具最终输出补丁的正确性,其表示的不完整程序规约必然严重影响输出补丁质量.

针对以上关键问题,一方面从补丁生成的角度,基于第 1.2 节提出的补丁语句表示 *patch*(*L*,*OP*,*C*)进行说明.现有学者首先对修复模板 *OP* 进行深入研究,利用被修复程序代码自身的信息和领域知识作为辅助规约对输出的补丁质量进行排序,从而提高修复的正确率.包括从修复的历史信息中挖掘模板,从而对修复模板赋予不同的权重来更有效地指导候选补丁的生成;通过实例程序(example)学习细粒度的修复模板,并用领域专用语言(domain-specific language,简称 DSL)来刻画这种细粒度的模板用于补丁生成.还有一些学者对合成补丁所需的素材 *C* 进行深入研究,包括 Angelix 利用轻量的符号执行和约束求解技术获取补丁合成素材;ACS 对条件类错误修复所需要的补丁内容 *C* 进行细粒度研究,利用变量使用的局部性原理和谓词频率挖掘等技术获取合成条件类补丁的素材;CapGen 利用抽象语法树 AST 的上下文感知信息,提出 3 种模型来获取更细粒度的补丁修复素材 *C*.以上一系列技术的研究,大幅度提高了候选补丁空间中所包含补丁的正确率.

另一方面,从补丁判定的角度,研究如何加强补丁质量判定条件来缓解过拟合问题.

第 1 类规约补全问题研究规约本身,通过直接加强程序规约 *S*,达到增强补丁质量判定条件的目的.例如,DiffTGen 通过增量生成新的测试用例来加强原有的测试集,新测试用例覆盖原 buggy 程序和打过候选补丁的程序之间存在语义差别的区域.其前提假设是,测试集的覆盖率更高,对输出高精度补丁有益.SemGraft 引入参考程序的思想,将提取的参考程序语义替代测试集来表示待修复程序更完整的程序规约 *S*.

第 2 类补丁择优问题研究在已有规约 *S* 无法进一步区分输出补丁好坏的情况下,利用启发式规则或者概率模型来进一步识别错误的补丁.例如,从自动生成的候选补丁中学习一般的和领域无关的反模式(anti-pattern)的方法,过滤掉错误的补丁.利用测试集信息之外的信息建立概率模型来进一步识别错误补丁,包括:Qlose 利用语法和语义相似度计算程序距离构建排序模型;Prophet 利用补丁本身正确性和上下文交互正确性两部分特征学习人工补丁获得最大似然估计的概率模型;利用测试用例执行行为相似度构建分类模型.

3 完全规约的程序修复问题及方法

如前所述,针对某些特定类型错误的自动修复方法具有完整的程序规约刻画,即刻画的程序规约 S 和原程序语义等价,且修复后不引入原程序语义改变.在长期的开发实践中,一些特定类型错误的发生原因已被开发者分析清楚并能够完整地描述其程序规约.例如内存泄露、资源泄露、特定性能错误、配置错误、并发错误的一些子类型包括死锁、数据竞争、原子性违反和顺序违背等特定类型都各有错误特点.结合第 1.2 节提出的补丁语句表示 $patch(L,OP,C)$来说明该类问题的修复,一般程序出错位置 L 由调用栈或者规约指导的精确分析技术进行推测;修复模板 OP 针对不同错误类型不尽相同,但对特定类型错误一般具有相对固定的修复模式,即结合先验知识可以确定相对固定的修复模板 OP;而补丁所包含的内容 C 是该类方法研究的重点之一,因为具有明确的规约和修复模式,补丁内容可以通过规约指导的精确分析方法求解获得.最终判定补丁质量时,由于程序规约是明确和完全的,因此修复过程产生的补丁准确性较高.每一种特定类型错误都可以代表一类修复场景,需要针对不同问题的特点进行细致研究,对应问题的刻画、程序规约描述和因果分析、方法设计及有效性验证.

3.1 并发错误修复

并发错误(concurrency bug)是由于指令在不符合预期的时机对共享变量进行访问造成的,其完整的程序规约与原程序串行语义等价,且不发生特定并发错误.因此,自动修复的目标是消除并发错误且不引入新的错误,修复后的程序和原程序逻辑语义等价,同时具有较好的并发性能.并发错误中,目前研究最多的包括数据竞争、原子性违背、顺序违背和死锁[26],并发错误并不仅仅包括这几种类型,更多的错误类型随着人类知识的积累会被不断地识别出来.该部分并发错误的修复方法在综述文献[8]中已经进行了梳理,但没有指出该类修复属于完全规约程序修复问题,我们的工作中简单梳理并说明问题.

1) 原子性违背

原子性违背是指对某一为保证正确性必须原子性执行的指令序列,存在一个执行交错,其执行效果不与任何该指令序列原子性执行时的执行交错的执行效果相同[26].该类错误程序的完全规约 S 和原程序串行语义等价,且不发生原子性违背问题.

2011 年,Jin 等人[61]提出了 Afix,能够自动修复一类常见的单变量原子性违反(single-variable atomicity violation)并发错误.该错误的程序规约 S 是可以完全确定的.AFix 首次提出该问题,并通过对错误检测报告(特定并发错误检测工具 CTrigger)进行静态分析,在合适的位置插入新锁来保证不发生原子性违反操作,并在运行时进行补丁正确性验证.然而,AFix 只针对 1 类同步原语(互斥锁),针对特定的检测报告只解决原子性违背这 1 种类型的并发错误,当错误检测器不能给出错误根本原因(root cause)时,则无法修复该类错误.

2012 年,Jin 等人[62]提出了更强大的并发错误修复方法 CFix.该方法可适配更多类型的并发错误检测器(一种集成的检测和修复工具).其中,CFix 使用 CTrigger[63]作为原子性违背错误检测前端时,通过添加新锁对检测到的原子性违背区域进行保护.

2012 年,Liu 等人[64]提出了 Axis 方法,能够进行原子性违背并发错误自动修复,其将原子性违反问题看作约束求解问题,利用 Petri 网(Petri net)模型进行修复,其修复后,程序的安全性和性能比 AFix 更好.换句话说,其修复后的程序不会引人新的死锁,同时并发性能也相对较好.

2014 年,Liu 等人[65]针对前述方法 AFix[61]和 Axis[64]修复原子性违背可能会引入死锁的问题,提出 Grail 更严格的修复原子性违背问题,同时保证安全性,即不会引入新的死锁.主要是利用 Petri 网进行分析从而消除引人新的死锁.该方法只适用于消除 2 个线程的死锁问题.

2016 年,Liu 等人[66]提出了 HFix,首先对收集的 77 个并发错误人工修复补丁进行实证研究,得出人工补丁的 3 类修复策略:通过增加和删除同步(synchronization)原语来改变执行时序、旁过(bypass)一些错误指令或容错(tolerance)处理、共享变量私有化(data private)技术.HFix 针对原子性违背修复策略不同于 CFix,利用程序中原有的同步而不是引入新的同步.该方法由于前期的实证研究基础,能够针对原子性违背错误产生类似于人工编写的高质量补丁.

2) 数据竞争

数据竞争是指对某一共享内存单元,存在来自不同线程的 2 个并发访问,且至少 1 个为写访问[26].该类错误程序的完全规约 S 和原程序串行语义等价,且不发生数据竞争问题.

2012 年,Jin 等人[62]提出了集成并发错误修复方法 CFix.该方法针对数据竞争问题首先确定顺序和互斥关系的组合,然后用静态分析和测试方法来确定插入同步操作的位置 L,修复模板 OP 是顺序化或者添加新的互斥锁.

3) 死锁

死锁是在某线程集合中的每一个线程都在等待另一个线程占有的互斥性资源,由此造成的循环等待即为死锁[26].死锁包括资源死锁和通信死锁[67].死锁错误程序的完全规约 S 和原程序串行语义等价,且不发生线程的循环等待问题.

2016 年,蔡彦等人[68]针对资源死锁问题提出了自动修复方法 DFixer.由于早期的一些并发错误修复技术通过动态插入新锁对共享资源进行加锁,该类方案可能会引入新的死锁.而通过静态强制序列化地执行可能引发死锁的线程来避免死锁,会导致大量的性能损失问题.DFixer 选择一个线程进行修复,通过控制流分析提前获取锁而不引入新锁,消除持有等待必要条件(hold-and-wait necessary condition),并引入唤醒条件保护(context-aware condition)来消除死锁.

4) 顺序违背

顺序违背是指某一指令(组)没有按照设计预期,总是在另一指令(组)之前或者之后执行的问题[26].顺序违背错误程序的完全规约 S 和原程序串行语义等价,且不存在违反预期设计顺序执行的指令.

2012 年,Jin 等人[62]提出了集成并发错误修复方法 CFix,其中使用 ConMem[69]作为顺序违背错误检测前端时,CFix 给出的修复方案是增加线程邻接(thread-join operation)而不是简单地增加信号量和等待.该方法由于前期的实证研究基础,针对顺序违背错误产生的补丁更接近人工编写的质量.

3.2 内存泄露和资源泄露修复

2015 年,高庆等人[15]提出了安全的针对 C 语言的内存泄漏错误修复方法.该方法人工给定了针对内存泄漏的安全修复正确规约,即不发生内存泄露的正确程序规约是“对于任意执行路径和任意插入的释放语句(*free*),要保证释放前已分配、无双重释放、无释放后使用这三者同时满足”,则待修复程序的完整规约 S 和原程序语义等价,并不发生内存泄露.该错误修复模板 OP 是 *free*(C),补丁生成过程是求解 free 语句的安全插入位置和释放空间大小(即修复模板的参数 C)的过程.具体反复使用使用数据流分析和过程间分析技术,通过前向数据流分析检查释放前分配,后向数据流分析检查释放后使用,前后向数据流分析检查双重释放.分析过程中需要解决一系列复杂问题,包括对循环、全局变量、多重分配、空指针判断等问题的分析和处理.在一定程度上,用数据流分析的效率达到了路径敏感分析的效果.开发的 leakFix 修复工具在 SPEC 2000 数据集进行实验,对 15 个项目中检测到的 25 个内存泄露自动生成了 41 个正确修复补丁,平均 1 个内存泄露错误生成 1.6 个补丁.

2018 年,Rijnard 等人[70]提出用静态方法检测和修复指针安全属性违反问题,使用分离逻辑(separation logic)生成补丁,并对输出补丁进行形式验证.形成的 FootPatch 方法可以修复多种错误类型,包括资源泄露、内存泄露和空指针解引用这 3 种.其中,资源泄露、内存泄露这两种错误修复问题属于完全规约修复问题.修复后,程序规约 S 和原程序语义等价,同时不存在资源泄露和内存泄露问题.而空指针解引用问题不属于完全规约修复问题,具体在第 4.2 节手工编写程序规约中介绍.该方法使用分离逻辑生成补丁并防止补丁过拟合,其假设修复代码存在于原程序中,通过静态分析原程序中相关代码片段的语义构造修复补丁.该方法不依赖测试集,程序规约 S 目前需要使用分离逻辑针对错误类型手工编写.其修复模板 OP 和补丁内容 C 都是从原程序中静态分析获取,例如,使用 *sw_free* 和 *swHashMap_node_free*(*hmap*,*root*)等释放语句从待修复程序中已经封装好的语句构造补丁,更符合原程序的编码规范和实现风格.

3.3 性能错误和配置错误修复

2015 年,针对特定性能错误,Adrian 等人[71]提出了 CARAMEL 自动检测和修复特定性能错误.针对循环中当某个条件成立时,剩余的执行都是在浪费计算资源的目标研究问题,提出了针对性的修复方法.通过识别“循环+条件”并分析满足给定的性能错误判定规则,然后在适当的位置插入类似“条件+break”的修复模板 *OP* 来规避性能问题,同时不影响程序原有功能.程序的完全规约 *S* 和原程序语义等价,并消除了上述性能问题.该论文获得了 ICSE 2015 年的最佳论文奖.

2017 年,Weiss 等人[72]针对服务器配置漂移(configuration drift)问题提出了交互式修复方法 Tortoise.配置漂移是指随着时间推移,管理员对一组服务器或者服务做出的配置引起的实际状态,偏离了管理员所期望的配置状态改变.该类问题的完全规约 *S* 和上述特定类型错误有所不同,配置漂移的完全规约 *S* 是由管理员选择的,选择后修复的实际配置状态和管理员的预期状态等价并且消除了配置漂移错误(如配置状态不一致等问题).在变更服务器状态时,管理员会输入一组配置命令序列,Tortoise 同时在后台利用 ptrace 记录由管理员的配置命令引发的服务器中系统调用和文件系统变化信息.然后将配置状态构造成一个模型,其中新的状态变更刻画为硬约束,原来已有的状态配置刻画为软约束.Tortoise 接着将模型表示转化为逻辑公式,利用 SMT 求解器转化为约束求解问题计算状态不一致性并生成修复补丁,利用启发式规约对补丁排序并推荐给管理员选择.在给定的 42 个配置场景中,Tortoise 推荐出的第 1 个修复补丁 76%被管理员采纳.其实,在 2015 年,熊英飞等人[73]提出了 Range Fixes 的概念(即允许配置项的取值在一定范围内变化)交互式的修复软件配置错误.软件配置错误问题的完全规约 *S* 也是由用户选择,选择后修复的实际软件配置状态和用户的预期状态等价并且消除了配置错误.

3.4 小 结

完全规约的程序自动修复问题面向一些特定错误类型.目前出现的该类问题已经枚举了并发错误中的数据竞争、顺序违背、原子性违背和死锁、内存泄露、资源泄露、特定性能问题和配置错误等.以上特定错误都可以进行明确的问题刻画并且完整地描述待修复程序规约 *S*,因此更多地使用精确分析技术以保证较高的修复精度.相对于不完全规约程序的修复问题,该类问题的修复技术不依赖于测试集和庞大的修复空间,能够有效地避免过拟合问题.

虽然完全规约的程序修复问题对应的方法修复精度和产生的补丁质量较高,但定向修复的固有特性使其修复能力受限,只对特定类型错误有效而不能同时修复多种类型的错误.从以上分析可以看出,该类修复技术到目前已枚举了部分特定错误类型,更多的错误类型和相关修复技术有待研究,更多错误类型的实证研究有必要深入挖掘.

4 半完全规约的程序修复问题及方法

如前所述,半完全规约的程序修复问题涵盖了多种类型的程序规约 *S*,包括契约、手工编写的程序规约和神经网络模型等.结合第 1.2 节提出的补丁语句表示 *patch*(*L*,*OP*,*C*)和程序规约 *S* 来说明该类问题的修复,该类问题的主要特征是修复中刻画的程序规约 *S* 是否完整不确定,求解补丁函数 *patch*(*L*,*OP*,*C*)的过程和不完全规约的程序修复问题对应的方法在一定程度上有类似之处.但由于先隐含假设所面向修复问题的程序规约 *S* 是完整的,修复后输出的补丁还需要进一步手工检查或者证明.

4.1 契约作为程序规约

契约作为程序规约,具体表示为前置条件(precondition)、后置条件(postcondition)、类不变式(class invariant)等程序局部要保持的性质.在 Eiffel 语言中具有该类规约,可以类似理解为 Java 语言中 JML(Java modeling language)对其设计规约的描述.该类修复技术中常假设程序中存在契约,并将其作为指导补丁生成和判定补丁正确性的依据,而契约代表的程序规约 *S* 是否完整并不确定.

2010 年,Wei 等人[27]首次提出了 AutoFix-E,利用契约进行程序自动修复.AutoFix-E 将 Eiffel 语言编写的程序中提供的契约作为判断程序正确性的标准,要求在最终修复的程序中,函数在执行前的状态满足前置条件和

类不变式,函数执行后的状态满足后置条件和类不变式,且执行过程中还需要满足过程内部断言(intermediate assertion).这些断言和不变式规范了程序的正确行为,一定程度上可以理解为基于模型的程序自动修复方法.AutoFix-E 在给定的实验数据集上进行实验,可以成功地修复 42 个错误中的 16 个.

2011 年,裴玉等人[74]提出了 AutoFix-E2,在 AutoFix-E 基于模型的修复方法基础上,进一步挖掘程序本身包含的信息来指导修复.他们将其称为基于证据的修复方法,将随机测试、错误定位、动静态分析收集的结果作为证据.具体通过静态分析(表达式依赖和控制依赖)和运行通过/未通过两种测试用例的动态分析,当检测到待修复表达式接受一个可疑值时将其转换为合法的值,从而保持契约的成立.

2014 年,裴玉等人[75]提出了 AutoFix 的最新实现和实验,其部分思想在文献[27,74]中已经介绍.AutoFix 使用 AutoTest 基于契约和随机测试框架自动产生测试用例,由失败的测试用例驱动动态分析进行错误定位,基于契约指导生成补丁并进行判定,最终利用相关性对判定通过的补丁进行排序.在给定的 204 个错误修复实验中,AutoFix 的正确率为 42%.由于 AutoFix 生成补丁的正确性判断依据是给定的契约(如果给定的契约表示了不完整的程序规约,则可能引入误报),在 AutoFix 给出的正确补丁中进行人工检查,其完全正确的补丁占有率为 59%.

2015 年,裴玉等人[76]提出了在 IDE 中进行程序自动修复的思想,将 AutoFix 集成到 EiffelStudio 集成开发环境中.AutoFix 作为一个推荐系统在 IDE 后台自动地检测源码错误并给出建议的修复补丁,这将为开发人员提供强大的自动化功能.

4.2 手工编写程序规约

如下修复问题中刻画的程序规约 S 使用线性时序逻辑(linear-temporal logic)、分离逻辑(separation logic)等手工编写,其针对不同程序和修复问题手工编写,该程序规约 S 是否完整不确定.

2005 年,Jobstmann 等人[77]将程序修复问题刻画为一种游戏,获得一组对程序修改的策略,使得修改后的程序符合程序规约,则认为获得一次胜利.程序规约 S 由线性时序逻辑(linear-temporal logic)给出.该方法假设程序出错范围仅在程序表达式语句或赋值语句的左值,也就是说,其修复能力仅包含程序表达式错误或赋值语句错误的修复.

2013 年,Essen 等人[78]使用线性时序逻辑(linear-temporal logic)给出形式化程序规约 S,提出一种新的修复方法,要求修复后的程序符合该给定的程序规约,同时和原 BUG 程序语义保持相似.该方法最大的特点是在修复过程中强制保持一个原 BUG 程序执行轨迹的子集,从而自动将程序正确部分本身的语义保持下来而不被修改.也就是说,原 BUG 程序符合程序规约的正确部分需要持续保持,同时只需要小范围地修改出错部分的语义,使得整个修复后的程序在保持原正确部分的同时,满足线性时序逻辑刻画的规约 S.

2015 年,Kneuss 等人[79]提出一种修复递归函数数据类型错误(树、链表、整形)的方法.该方法需要开发人员使用特定的语言编写程序和对应的程序规约 S,其假设待修复的错误程序是有限状态的(infinite-state)并且程序规约是完整正确的,最终的修复程序经过 Leon 系统进行形式化验证.

2018 年,Rijnard 等人[70]提出用静态方法检测和修复指针安全属性违反问题,其中,针对空指针解引用问题属于半完全规约的程序修复问题.形成的 FootPatch 方法是一个集成的工具,其中,解决空指针解引用问题的程序规约 S 使用分离逻辑手工编写且是否完整不确定;同时,修复空指针解引用问题后是否改变原程序语义也不确定.该方法对空指针解引用的处理具体包括指针为空添加前置条件检查、空指针引用报告异常处理等方式,暂未考虑实例化一个新的指针进行引用等多样化方式.以上多种修改方案都可能会改变原程序语义,但是因为 FootPatch 采用静态方法进行修复,不会过拟合类似动态测试用例提供的期望语义.在给定的实验中,成功地修复了 24 个空指针解引用错误.

4.3 其他程序规约

1) 测试用例的修复

2010 年,Daniel 等人[80]提出了 ReAssert 自动修复发生错误的测试用例.当一个测试用例执行失败时,可能的情况是被测程序出错或者测试用例出错.ReAssert 对执行出错的变量值和控制流进行动态分析,进而修改错误

赋值和断言,并尽可能地保持原测试用例的逻辑功能.ReAssert 利用以上策略对执行失败的测试用例进行修改,使得原执行失败的测试用例能够执行成功.其潜在假设是程序行为正确,测试用例执行成功则符合程序规约 *S*.因此,ReAssert 无法判定测试用例出错情况以及测试用例正确的执行逻辑.修复后的测试用例以建议的方式给出,其正确性还需要人工进行确认.

2016 年,Gao 等人[81]提出了 SITAR 自动修复 GUI 测试脚本.在回归测试中,由于原 GUI 程序图形组件的变化,使得原测试用例的事件或操作序列等输入不再适应当前的测试脚本,期望测试的 GUI 对象对应的断言和检查点等测试预言(test oracle)也需要适应性地更新.SITAR 通过逆向工程生成的事件流图和用户输入修复原有测试脚本,使其在新的 GUI 程序中可用.

2) 崩溃错误和资源竞争

2015 年,高庆等人[82]提出了基于问答系统的特定修复方法,利用 Q&A 问答系统中的知识修复崩溃错误.发生崩溃错误的原因很多,崩溃时程序行为如何不确定,目前不能确定地给出完整的程序规约 *S*.该方法具体通过抽取崩溃路径信息(crash trace)包含的行号作为候选出错位置,提炼崩溃报的错信息构造一组查询,检索 Stack Overflow 问答系统获得和崩溃相关的问题和回答页面列表.然后,通过模糊程序分析技术(fuzzy program analysis technique)形成修复的样例,并利用结构相似度和文本相似度进一步过滤噪声样例.利用编辑脚本将样例转化为最终的修复补丁.实验中,通过手工确认,表明可以修复实际的崩溃错误.具体在收集了 24 个可复发崩溃错误数据集上,该方法能够修复其中的 10 个崩溃错误,其中 8 个修复结果正确.

2016 年,Wang 等人[83]提出了 ARROW,针对现代浏览器软件的并发执行引起的 Web 应用程序资源竞争问题进行自动修复.例如,客户端 Web 页面包含各种类型脚本(HTML、JS 和样式表等),在并发渲染和异步加载时,这些异步事件和用户输入事件相互竞争资源并以非确定性的顺序执行,可能引起网络延迟、执行异常等问题.Web 页面异步加载和用户请求资源竞争问题存在很多执行交错,也不能完全串行,刻画的程序规约 *S* 是否完全符合预期行为不确定.ARROW 以浏览器渲染各页面元素的前后依赖关系,将 Web 页面静态建模为因果图,通过 Web 页面源码获取开发人员预期的各元素依赖关系.最后,利用求解器对构造的约束进行求解,使得两者不会出现不一致的情况.

3) 缓冲区溢出和整形错误

2016 年,Gao 等人[84]提出了 BovInspector,能够自动检测和修复缓冲区溢出漏洞.为了过滤误报,对静态分析的错误检测结果进行可达性分析,并在可达性指导下进行符号执行收集路径约束条件和指定的溢出约束条件,通过比对两者确定真正的缓冲区溢出漏洞.对确认的缓冲区溢出漏洞采用 3 种修复模板 *OP*:添加边界检查、替换为更安全的 API 和扩展缓冲区空间.以上修复操作都可能改变原程序语义,程序规约 *S* 完整性不确定,修复结果需要人工检查.在给定的实验中,BovInspector 能够以平均 74.9%的准确率检测该类漏洞,并全部产生正确的修复补丁.

2017 年,Cheng 等人[85]通过推测合适的变量和表达式数据类型,自动生成整形错误的补丁,提出交互式的修复方法 IntPTI.该方法通过静态值分析近似表达式的值,并收集其可能的正确类型约束,然后转化为约束求解问题,推测可能的正确数据类型来生成补丁,最后,通过 Web 界面展示供开发人员交互式选择和确认正确的补丁.具体来说,整形错误的程序规约 *S* 由静态分析收集约束并求解获得,其规约完整性不确定;同时,修复后是否引起原程序语义变化也不确定.该方法设计了 3 种修复模板 *OP*:完整性检查(sanity check)、显式类型转换(explicit type casting)和改变申明类型(declared type changing).补丁内容 *C* 即推断出的正确数据类型通过约束求解获得.IntPTI 在收集的 7 个实际项目数据集上实验,能够成功地使 25 个错误中的 23 个补丁被用户确认和采纳.其实,2 个误报是由于采用流不敏感的静态分析技术进行正确的数据类型推断所引起的.

4) 语法错误修复

2017 年,Gupta 等人[86]首次提出了 Deepfix,利用深度学习技术直接生成补丁.这种方法的修复对象是语法错误,将程序抽象为以语句为粒度的序列,利用多层次的神经网络模型来预测出错位置和正确的语句,其修复精度取决于从训练集中学习的神经网络.而神经网络模型的质量取决于特征向量的选择和训练集的质量.该模型表

示的程序规约 S 的完整性不确定.该方法相当于将学习获得的神经网络模型作为程序规约 S,对学生的句法错误修复实验显示,其完全修复的准确率在 27%,在其他更复杂的缺陷上的表现尚不清楚.

5) 搜索语句修复

2014 年,Gopinath 等人[87]提出一种利用面向数据的语言 ABAP 修复 SELECT 语句错误的方法.具体利用数据分布中所隐含的信息,使用半监督的学习方法识别正确行为,修复 SELECT 语句中 WHERE 子句附带的条件错误.该方法学习获得的程序规约 S 完整性不确定,需要人工确认修复结果.

6) 用户体验

2018 年,Sonal 等人[88]提出了 MFix 方法自动修复手机中网页的友好显示问题.由于很多网站不是专门为手机设备设计和开发的,这会导致在手机上显示文本不可读、导航混乱、内容溢出手机设备显示窗体等问题,手机上网页友好显示的问题规约 S 由用户手工确认.MFix 方法主要关注字体大小、点击目标间距、内容缩放调整等问题,通过自动生成层叠样式表(cascading style sheet,简称 CSS)补丁来优化上述问题的友好显示.MFix 首先建立基于图的网页布局模型;然后对这些图进行强制编码以增强显示友好性,同时最小化布局的损坏,从而生成 CSS 补丁.该方法使用访问频度靠前的 38 个网站主页进行评估实验,能够成功解决占比 95%的网站在手机上友好显示的问题.

4.4 小 结

半完全规约的程序修复问题对应的方法扩展了程序规约 S 的刻画方式和使用范畴,所使用的契约、形式规约和学习的行为模型等程序规约有效补充了测试用例不足的问题.该类方法存在的主要问题是:输出的补丁后期需要人工确认或者正确性证明,用于大规模程序错误的修复非常困难.同时,现实世界中自带契约、形式规约等程序规约的待修复问题非常少,很多问题的形式规约需要手工构造.

5 总结和展望

根据上述文献研究结果可以得出:程序自动修复技术虽然研究历史较短,但得到了学术界持续的高热度关注,并取得了大量的研究成果.虽然目前暂时还没有工业界的应用,但一系列的研究成果表明,程序自动修复技术已经在一定程度上具备自动修复实际应用中简单错误的能力.虽然国内对该问题的研究起步较晚,但近年来国内的研究情况令人欣慰,包括北京大学、国防科技大学、南京大学、武汉大学、上海交通大学和中国科学院软件研究所等单位的研究非常活跃,在顶级期刊发表的一系列研究成果得到了国外同行的认可.本文提出一种基于规约的程序自动修复描述,并从程序规约的角度将问题进行分类梳理,程序规约 S 的刻画方式直接影响着补丁生成函数 $P=patch(L,OP,C)$的求解过程和补丁判定条件的构造.从程序自动修复对象是否具有完整的程序规约 S 这个关键问题进行分类,对错误修复的不同场景和技术体系进行分类阐述.梳理了各类修复方法的研究进展,阐述了各类研究问题和方法的异同、研究重点和可能存在的问题.

程序自动修复的研究领域中机遇和挑战并存,有待更多的研究者们取得创新和突破.我们认为,该领域还存在如下值得进一步研究的问题.

(1) 程序自动修复技术给传统的错误定位提供了新的应用场景,传统的错误自动定位目的是辅助人工进行错误修复.辅助人工和辅助机器进行错误修复是不同的问题,他们要求的精度不同.例如,人工修复更倾向于定位到函数级,而机器自动修复更需要语句级甚至表达式级别的精确位置.传统的错误定位更多的研究关注于错误语句的位置排序和可能出错位置的最小集,而对出错语句可疑值本身的精确性和语句内部可疑错误位置研究不足.辅助人和软件进行错误自动修复所需的错误位置精度不同,到目前为止,还没有出现专门针对程序自动修复技术而设计的错误高精度自动定位方法.针对程序自动修复场景,设计更细粒度和更高精度的错误定位技术值得深入研究.

(2) 针对不完全规约的程序自动修复问题,即直接或间接地(基于测试集提取的条件约束)使用测试集作为待修复程序规约 S,高精度修复技术还有待进一步研究.一般的程序错误中,条件错误和函数调用错误占比更高,因此,对表达式条件或更复杂的条件错误、接口错误的修复值得深入研究.由于弱测试用

例问题依然存在,如何利用更细粒度的源代码静态信息、代码执行的动态信息以及其他测试用例之外的信息加强程序规约,从而加强对输出补丁的正确性判定条件和增强的规约指导补丁生成都是重要的研究问题.研究更多的程序规约刻画方法用于补丁质量判定,例如借鉴传统的程序验证和证明技术,结合具体程序修复场景进一步判定测试用例集无法区分的补丁正确性问题.多行错误、互有依赖的多个错误以及跨项目错误的程序自动修复是更复杂的问题,是同样值得研究和探讨的困难问题.

(3) 针对完全规约的程序自动修复问题,即待修复程序规约 S 能够完整刻画的修复问题,需要更多实例基础,更多类型特定错误的发生原因和人工修复方法有待充分的实证研究.基于充分的实证研究基础,更多具有完全规约的程序自动修复问题尚待进一步发掘,由于修复错误而引入的程序语义变化的合理性需要充分讨论.在提高修复精度的同时,修复的补丁质量(例如补丁可读性和人工修复的补丁质量近似等)也是重要的研究问题.另外,将多种特定类型错误修复技术结合,例如与缺陷自动分类技术结合来提升特定错误修复技术的可扩展性和修复能力.

(4) 对于半完全规约的程序自动修复问题,其待修复程序规约 S 是否能够完整刻画不确定.在程序自动修复场景中先假设有完全规约,最终生成的补丁程序正确性校验是核心问题.尤其面对程序规模较大的情况下,如何结合程序验证和证明技术自动进行正确性判定是困难问题.当然,对于完全规约和半完全规约程序修复问题本身也值得进一步研究,充分讨论其内涵和外延有助于进一步扩展程序自动修复技术所面向的问题.

统一的程序自动修复技术评价标准,测试用例和 banchmark 不足依然是面临的客观问题.测试用例自动生成和测试用例修复相关技术为程序自动修复提供有效支撑,更大和更符合错误自然分布的banchmark有待进一步建立.各修复技术的横向对比分析和统一的评价标准有待丰富,从而促进修复技术的持续发展和应用.

总体来讲,程序自动修复技术最终要解决的核心问题是修复真实 bug,针对实际问题的任何改进都值得深入研究.例如,待修复程序的规模,即如何修复规模尽可能大的程序中包含的真实bug;修复能力,即如何修复尽可能多的真实 bug 和涵盖更广泛的错误类型;补丁质量,即在保证生成补丁正确性的前提下,如何使工具自动生成的补丁和人工修复补丁更接近,从而更容易被开发者接受.针对该热点研究,未来的机遇和挑战并存、荣耀和艰辛同在.

References:

[1] Hailpern B, Santhanam P. Software debugging, testing, and verification. IBM Systems Journal, 2002,41(1):4−12.

[2] Anvik J, Hiew L, Murphy GC. Who should fix this bug? In: Proc. of the 28th Int'l Conf. on Software Engineering. 2006. 361−370.

[3] https://www.mozilla.org/en-US/security/client-bug-bounty

[4] https://blog.chromium.org/2010/01/encouraging-more-chromium-security.html

[5] https://technet.microsoft.com/en-us/security/dn425036

[6] https://en.wikipedia.org/wiki/Deep_Impact_(spacecraft)

[7] Xuan JF, Ren ZL, Wang ZY, Xie XY, Jiang 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.htm [doi: 10.13328/j.cnki.jos.004972]

[8] 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).

[9] Goues CL, Forrest S, Weimer W. Current challenges in automatic software repair. Software Quality Journal, 2013,21(3):421−443.

[10] Monperrus M. Automatic software repair: A bibliography. ACM Computing Surveys, 2018,51(1):Article No.17.

[11] Le XBD, Thung F, Lo D, Le Goues C. Overfitting in semantics-based automated program repair. In: Proc. of the Empirical Software Engineering. 2018. 1−27.

[12] 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 2018). 2018. 11−22.

[13] Andersen J, Lawall JL. Generic patch inference. Automated Software Engineering, 2010,17(2):119−148.

[14] Staats M, Whalen MW, Heimdahl MPE. Programs, tests, and oracles: The foundations of testing revisited. In: Proc. of the Int'l Conf. on Software Engineering. 2011. 391−400.

[15] Gao Q, Xiong Y, Mi Y, Zhang L, Yang W, Zhou Z, Xie B, Mei H. Safe memory-leak fixing for c programs. In: Proc. of the 37th Int'l Conf. on Software Engineering (ICSE 2015), Vol.1. Piscataway: IEEE Press, 2015. 459−470.

[16] Wong WE, Gao R, Li Y, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016, 42(8):707−740.

[17] Yu K, Lin MX. Advances in automatic fault localization techniques. Chinese Journal of Computers, 2011,34(8):1411−1422 (in Chinese with English abstract).

[18] Chen X, Ju XL, Wen WZ, Gu Q. Review of dynamic fault localization approaches based on program spectrum. Ruan Jian Xue Bao/ Journal of Software, 2015,26(2):390−412 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4708.htm [doi: 10.13328/j.cnki.jos.004708]

[19] Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proc. of the 24th Int'l Conf. on Software Engineering (ICSE 2002). 2002. 467−477.

[20] Tassey G. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, 2002,15(3):125−125.

[21] Dijkstra EW. Notes on structured programming. In: Dahl OJ, Dijkstra EW, Hoare CAR, eds. Proc. of the Structured Programming. Academic Press, 1972. 1−82.

[22] Qi Z, 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. ACM Press, 2015. 24−36.

[23] Smith EK, Barr ET, Goues CL, Brun Y. Is the cure worse than the disease? Overfitting in automated program repair. In: Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM Press, 2015. 532−543.

[24] Le Goues C, Nguyen T, Forrest S, Weimer W. GenProg: A generic method for automatic software repair. IEEE Trans. on Software Engineering, 2012,38(1):54−72.

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

[26] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015,38(11):2215−2233 (in Chinese with English abstract).

[27] Wei Y, Pei Y, Furia CA, Silva LS, Buchholz S, Meyer B, Zeller A. Automated fixing of programs with contracts. In: Proc. of the 19th Int'l Symp. on Software Testing and Analysis. ACM Press, 2010. 61−72.

[28] Long F, Rinard M. An analysis of the search spaces for generates and validates patch generation systems. In: Proc. of the IEEE/ ACM 38th Int'l Conf. on Software Engineering (ICSE). IEEE, 2016. 702−713.

[29] Le XBD, Le TDB, Lo D. Should fixing these failures be delegated to automated program repair? In: Proc. of the IEEE 26th Int'l Symp. on Software Reliability Engineering. IEEE Computer Society, 2015.427−437.

[30] 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. IEEE, 2017. 180−182.

[31] 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. ACM Press, 2016. 691−701.

[32] Le XBD, Lo D, Goues CL. Empirical study on synthesis engines for semantics-based program repair. In: Proc. of the IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2017. 423−427.

[33] Le XBD, Lo D, Goues CL. History driven program repair. In: Proc. of the Int'l Conf. on Software Analysis, Evolution, and Reengineering. IEEE, 2016. 213−224.

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

[35] Suzuki R, Suzuki R, Suzuki R, Polozov O, Gulwani S, Gheyi R, Hartmann B. Learning syntactic program transformations from examples. In: Proc. of the 39th Int'l Conf. on Software Engineering. IEEE Press, 2017. 404−415.

[36] Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In: Proc. of the 39th Int'l Conf. on Software Engineering. IEEE Press, 2017. 416−426.

[37] Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proc. of the 28th Int'l Conf. on Software Engineering. ACM Press, 2006. 272−281.

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

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

[40] Qi X, Reiss SP. Leveraging syntax-related code for automated program repair. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE Press, 2017. 660−670.

[41] Sumi S, Higo Y, Hotta K, Kusumoto S. Toward improving graftability on automated program repair. In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2015. 511−515.

[42] Sidiroglou-Douskos S, Lahtinen E, Long F, Rinard M. Automatic error elimination by horizontal code transfer across multiple applications. ACM SIGPLAN Notices, 2015,50(6):43−54.

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

[44] 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 2018). 2018. 1−11.

[45] 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. 2018. 12−23.

[46] Mechtaev S, Yi J, Roychoudhury A. Directfix: Looking for simple program repairs. In: Proc. of the 37th Int'l Conf. on Software Engineering. Vol.1. IEEE Press, 2015. 448−458.

[47] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005,10(4):405−435.

[48] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the Usenix Conf. on Operating Systems Design and Implementation. USENIX Association, 2009. 209−224.

[49] Nguyen HDT, Qi D, 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.

[50] 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.

[51] 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, 2017,43(1):34−55.

[52] DeMarco F, Xuan JF, Berre DL, Monperrus M. Automatic repair of buggy if conditions and missing preconditions with smt. In: Proc. of the Int'l Workshop on Constraints in Software Testing, Verification, and Analysis. Hyderabad, 2014. 30−39.

[53] Qi X, 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. ACM Press, 2017. 226−236.

[54] Kong X, Zhang L, Wong WE, Li B. Experience report: How do techniques, programs, and tests impact automated program repair? In: Proc. of the IEEE 26th Int'l Symp. on Software Reliability Engineering. IEEE Computer Society, 2015. 194−204.

[55] Qi Y, Mao X, Lei Y, Dai Z, Wang C. The strength of random search on automated program repair. In: Proc. of the 36th Int'l Conf. on Software Engineering. ACM Press, 2014. 254−265.

[56] Ackling T, Alexander B, Grunert I. Evolving patches for software repair. In: Proc. of the 13th Annual Conf. on Genetic and Evolutionary Computation. ACM Press, 2011. 1427−1434.

[57] Assiri FY, Bieman JM. An assessment of the quality of automated program operator repair. In: Proc. of the 2014 IEEE 7th Int'l Conf. on Software Testing, Verification and Validation (ICST). IEEE, 2014. 273−282.

[58] 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. ACM Press, 2016. 727−738.

[59] Fan L, Rinard M. Automatic patch generation by learning correct code. In: Proc. of the ACM Sigplan-Sigact Symp. on Principles of Programming Languages. ACM Press, 2016. 298−312.

[60] 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. ACM Press, 2018. 789−799.

[61] Jin G, Song L, Zhang W, Lu S, Liblit B. Automated atomicity-violation fixing. In: Proc. of the ACM Sigplan Conf. on Programming Language Design and Implementation. 2011. 389−400.

[62] Jin G, Zhang W, Deng D, Liblit B, Lu S. Automated concurrency-bug fixing. In: Proc. of the Usenix Conf. on Operating Systems Design and Implementation. 2012. 221−236.

[63] Park S, Lu S, Zhou Y. CTrigger: Exposing atomicity violation bugs from their hiding places. ACM Sigarch Computer Architecture News, 2009,37(1):25−36.

[64] Liu P, Zhang C. Axis: Automatically fixing atomicity violations through solving control constraints. In: Proc. of the Int'l Conf. on Software Engineering. IEEE, 2012. 299−309.

[65] Liu P, Tripp O, Zhang C. Grail: Context-aware fixing of concurrency bugs. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM Press, 2014. 318−329.

[66] Liu H, Chen Y, Lu S. Understanding and generating high quality patches for concurrency bugs. In: Proc. of the ACM Sigsoft Int'l Symp. on Foundations of Software Engineering. ACM Press, 2016. 715−726.

[67] Joshi P, Naik M, Sen K, Gay D. An effective dynamic analysis for detecting generalized deadlocks. In: Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM Press, 2010. 327−336.

[68] Cai Y, Cao L. Fixing deadlocks via lock pre-acquisitions. In: Proc. of the Int'l Conf. on Software Engineering. IEEE, 2016. 1109−1120.

[69] Zhang W, Sun C, Lu S. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. ACM SIGARCH Computer Architecture News, 2010,38(1):179−192.

[70] 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 2018). 2018. 151−162.

[71] Nistor A, Chang PC. CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering (ICSE). IEEE, 2015. 902−912.

[72] Weiss A, Guha A, Brun Y. Tortoise: Interactive system configuration repair. In: Proc. of the 2017 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2017. 625−636.

[73] Xiong Y, Zhang H, Hubaux A, *et al*. Range fixes: Interactive error resolution for software configuration. IEEE Trans. on Software Engineering, 2015,41(6):603−619.

[74] Pei Y, Wei Y, Furia CA, Nordio M, Meyer B. Code-based automated program fixing. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM Press, 2011. 392−395.

[75] Yu P, Furia CA, Nordio M, Meyer B. Automatic program repair by fixing contracts. In: Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering. Springer-Verlag, 2014. 246−260.

[76] Pei Y, Furia CA, Nordio M, Meyer B. Automated program repair in an integrated development environment. In: Proc. of the 37th Int'l Conf. on Software Engineering, Vol.2. IEEE Press, 2015. 681−684.

[77] Jobstmann B, Griesmayer A, Bloem R. Program repair as a game. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag, 2005. 226−238.

[78] Essen CV, Jobstmann B. Program repair without regret. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer-Verlag, 2013. 896−911.

[79] Kneuss E, Koukoutos M, Kuncak V. Deductive program repair. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer Int'l Publishing, 2015. 217−233.

[80] Daniel B, Jagannath V, Dig D, Marinov D. ReAssert: Suggesting repairs for broken unit tests. In: Proc. of the 24th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2010. 433−444.

[81] Gao Z, Chen Z, Zou Y, Memon AM. SITAR: GUI test script repair. IEEE Trans. on Software Engineering, 2016,42(2):170−186.

[82] Gao Q, Zhang HS, Wang J, Xiong YF. Fixing recurring crash bugs via analyzing Q&A sites. In: Proc. of the 2015 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2015. 307–318.

[83] Wang W, Zheng Y, Liu P, Xu L, Zhang X, Eugster P. ARROW: Automated repair of races on client-side Web pages. In: Proc. of the Int'l Symp. on Software Testing and Analysis. 2016. 201–212.

[84] Gao F, Wang L, Li X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM Press, 2016. 786–791.

[85] Cheng X, Zhou M, Song X, *et al*. IntPTI: Automatic integer error repair with proper-type inference. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2017. 996–1001.

[86] 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.

[87] Gopinath D, Khurshid S, Saha D, Chandra S. Data-guided repair of selection statements. In: Proc. of the 36th Int'l Conf. on Software Engineering. 2014. 243–253.

[88] Mahajan S, Abolhassani N, McMinn P, Halfond GJ. Automated repair of mobile friendly problems in Web pages. In: Proc. of the 40th Int'l Conf. on Software Engineering. ACM Press, 2018. 140–150.

附中文参考文献:

[7] 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.htm [doi: 10.13328/j.cnki.jos.004972]

[8] 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.

[17] 虞凯,林梦香.自动化软件错误定位技术研究进展.计算机学报,2011,34(8):1411–1422.

[18] 陈翔,鞠小林,文万志,顾庆.基于程序频谱的动态缺陷定位方法研究.软件学报,2015,26(2):390–412. http://www.jos.org.cn/1000-9825/4708.htm [doi: 10.13328/j.cnki.jos.004708]

[26] 苏小红,禹振,王甜甜,马培军.并发缺陷暴露、检测与规避研究综述.计算机学报,2015,38(11):2215–2233.

李斌(1985－),男,甘肃天水人,博士生,工程师,主要研究领域为软件分析和缺陷修复,安全操作系统.

马恒太(1970－),男,博士,副研究员,主要研究领域为软件安全分析,操作系统安全.

贺也平(1962－),男,博士,研究员,博士生导师,主要研究领域为系统安全,隐私保护.