doi: 10.12194/j.ntu.20201123001
引文格式: 鞠小林, 钱洁, 赵春雨, 等. 一种基于反馈优化的启发式软件调试框架[J]. 南通大学学报(自然科学版), 2022, 21(1):25-36.

一种基于反馈优化的启发式软件调试框架

鞠小林[1,2], 钱 洁[1], 赵春雨[1], 陈志华[1]
(1. 南通大学 信息科学技术学院, 江苏 南通 226019;2. 南京大学 计算机科学与技术系, 江苏 南京 210023)

摘要:基于频谱的缺陷定位方法通过生成缺陷定位报告来引导程序员识别软件缺陷。由于缺少足够的上下文信息,基于该报告识别缺陷的过程非常耗时费力。此外,在连续搜索软件缺陷的过程中,程序员先前的判断对提高缺陷定位报告的效率没有任何帮助。针对上述问题,提出一种基于反馈优化的启发式软件调试框架。程序员可以在该框架上使用可视化的启发式图迭代地识别缺陷。首先,基于静态程序切片技术分析缺陷相关信息,并通过执行跟踪技术记录实时运行信息,以 DOT 格式组织并可视化这些信息,构造出有助于程序员识别缺陷的启发信息;然后,提出一种优化缺陷报告的算法,该算法可以根据程序员的反馈使用正向和反向切片技术来动态调整缺陷定位报告;最后,还提出了一种增强的算法,该算法可以基于程序员的多个反馈更高效地调整缺陷定位报告。实验结果表明,提出的启发式软件调试框架可以有效提高程序调试的效率。

关键词:反馈优化;交互调试;缺陷定位;启发信息建模;软件测试
中图分类号: TP311.5 **文献标志码**: A **文章编号**: 1673-2340(2022)01-0025-12

An interactive debugging framework with heuristic graph based on the feedback optimization

JU Xiaolin[1,2], QIAN Jie[1], ZHAO Chunyu[1], CHEN Zhihua[1]
(1. School of Information Science and Technology, Nantong University, Nantong 226019, China;
2. Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: The spectrum-based fault localization approaches guide programmers to identify defects with the defect localization report. Due to lacking sufficient context, the process of identifying defects based on the fault localization report is very time-consuming and laborious. Furthermore, during the identification for software defects, the programmer′s previous work is not utilized to optimize fault localization reports. This paper proposes a two-stage heuristic interactive debugging framework on which programmers can iteratively identify defects with visual heuristic graphs. The first stage is to construct heuristic graphs that help programmers identify defects. It first analyzes the defect-related information based on static program slicing technology, and records the run-time information by execution tracking, then organizes and visualizes them in DOT format. The second stage is to optimize the defect localization based on the programmer′s feedback. An algorithm, which dynamically adjusts defect localization reports based on the programmer′s feedback using forward and backward slicing techniques, is proposed for optimizing defect reports. At last an enhanced algorithm is proposed that can adjust the defect localization report with the multiple feedbacks from the programmers. The results show that the proposed approach can effectively improve the debugging efficiency for

收稿日期: 2020-11-23
基金项目: 国家自然科学基金面上项目(61673384);国家自然科学基金青年科学基金项目(61502497, 61602267)
第一作者简介: 鞠小林(1976—), 男, 副教授, 博士, 主要研究方向为软件分析与测试、程序调试和智能软件工程。
E-mail:ju.xl@ntu.edu.cn

programmers.

Key words: feedback optimization; interactive debugging; fault localization; heuristic information modeling; software testing

软件缺陷定位的终极目标是帮助调试人员快速准确地识别并修复缺陷。已有的缺陷定位技术大多数是给定出错可疑度高的候选程序实体集合,或者这些候选程序实体依据可疑度降序排列的排序列表[1-2]。然而,最新的一份研究表明,这些缺陷定位技术在软件开发实践中并没有得到广泛的应用[3]。主要原因在于这些缺陷定位技术给出的定位报告将程序实体视为独立个体而忽略这些实体之间的关系,包含的程序实体与程序执行顺序或源代码中出现的顺序并不一致,从而不利于程序员调试时分析和理解程序[4]。例如,基于程序语句的缺陷定位技术通常直接给出一个按语句怀疑度降序排列的检查列表[5-6],并不提供列表中语句之间的关系,而大多数情况下,由于检查列表中的语句在源程序中并不连续,程序员在识别语句是否有错时,通常需要自己分析该语句的前后相关语句,人工厘清语句之间的相关关系,并凭借经验判断当前审查语句是否出错。这种方式增加了程序员的工作量,并增加了人为出错的可能性。在没有上下文信息的前提下,调试人员在审查缺陷定位报告时,只阅读一条程序语句,往往很难判断其是否存在缺陷[7]。此外,在实践中,程序缺陷往往不仅仅是程序中的某一行出错,有些时候会跨越几个行代码甚至跨越不同类、文件等[8]。为此,我们需要一些启发式的信息辅助程序员进行判断。例如,提供当前审查的程序语句在执行时的上下文,以及与当前语句存在依赖关系的那些语句等信息。程序员基于这些启发信息,可以更高效地甄别当前审查的语句是否包含软件缺陷。此外,以往的缺陷定位技术往往是一次性提供定位信息,不能依据调试人员已有工作动态地调整缺陷定位报告,这种方式会影响到软件调试的精度和效率。研究发现许多软件缺陷在定位报告排序列表中往往处于列表的后端[9],这种现象更是削弱了现有缺陷定位技术在实践中的效用。为此,我们在程序调试过程中引入反馈机制,即当程序员完成对当前语句判断之后,依据判断结论对其余语句的怀疑度做动态调整,进一步优化缺陷定位报告,从而得到更优的语句审查顺序。

基于上述考虑,本文提出了一种基于程序员调试反馈的启发式软件调试框架。该调试框架借助程序分析技术,自动将准确的相关语句信息等以直观的图形展示给程序员,以帮助程序员进行缺陷定位的判断;此外,根据程序员的反馈,动态优化前期所得的缺陷定位报告为后续剩余缺陷定位服务,从而提高缺陷定位报告的效率。

本文的主要贡献如下:

1)提出一种收集并建模缺陷相关信息方法。首先定义了该上下文信息表示的数据结构,接着借助程序静态分析技术和动态执行跟踪技术,跟踪并记录软件失效时执行上下文信息。

2)给出精化的缺陷相关上下文信息的可视化方法。通过直观给出缺陷相关信息的依赖和传递关系,并只保留必要的缺陷关联信息,帮助调试人员高效识别和修复软件缺陷。

3)基于程序员在调试过程中的反馈信息,提出动态调整缺陷定位报告的策略,并给出基于单个反馈和成组反馈的缺陷定位优化算法。

4)设计并实现了一个基于上述策略和算法的原型系统(称为 HDebug)。通过邀请一批学生使用,实验验证了本文提出方法的有效性。

1 预备知识

1.1 基于频谱的缺陷定位

基于频谱的缺陷定位方法首先运行一组测试用例,借助程序跟踪技术记录程序实体(如程序语句、谓词、基本块等)的覆盖信息及程序执行结果(如图 1 所示);接着,利用一组缺陷怀疑度计算公式(如 Tarantula[10]、Naish[11]、DStar[12]等)计算程序实体的缺陷可疑程度;最后,将程序实体按照可疑度降序排列,形成缺陷定位报告。

这类方法的可疑度计算大部分是基于一个共同的假设:程序实体的可疑度与失败测试用例执行

覆盖次数正相关，与成功测试执行覆盖测试负相关[10]。图 1 中左边矩阵（N spectra）表示包含 M 个程序实体的程序经过 N 个测试用例运行的覆盖情况（a_{ij} 表示第 j 个程序实体被第 i 个测试用例覆盖信息），右边的矩阵则记录了这 N 个测试用例的执行结果（成功或失败）。基于频谱的缺陷定位先根据上述假设设计怀疑度计算公式，然后基于图 1 中的两个矩阵数据计算每个程序实体的可疑度，程序员根据怀疑度的降序依次甄别程序实体是否包含缺陷。

$$\begin{array}{c} \\ N\text{ spectra} \end{array} \begin{array}{c} \overset{M\text{ components}}{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix}} \end{array} \quad \overset{\text{error detection}}{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}}$$

图 1　基于频谱的缺陷定位矩阵

Fig. 1　Matrix for spectrum-based fault localization

1.2　信息收集与建模

为了辅助程序员调试时识别软件缺陷，需要提供与程序出错语句密切相关的程序信息。这些信息可以初步分为静态信息和动态信息两类。其中程序员关注的动态信息主要为程序执行路径，以及与出错语句相关的变量取值变化等；而静态信息则包括出错语句控制依赖的其他语句信息等。下面我们分析这两类信息的搜集与建模。

对于程序静态信息的搜集，我们的主要工作是获取目标程序的控制流程图。控制流图构造技术已经十分成熟，具体的技术不在文中一一赘述。得到的控制流图将被用于后续阶段模型分析及可视化展示。

程序的动态信息主要是为了帮助调试人员了解软件行为。在本文的方法中将记录如下信息：1）程序执行的轨迹。由于程序的任意基本块中的所有语句执行轨迹一致，因此我们选用基本块作为执行轨迹的一部分；此外，由于谓词决定了程序下一步执行哪个基本块，并且从实际的调试反馈可知，谓词是最容易发生缺陷的程序实体之一[13]，因此执行轨迹中也记录执行的谓词信息。2）谓词或基本块对应的变量取值。对于程序执行轨迹中的基本块，调试人员通常关心基本块中涉及的变量在进入基本块前的取值情况，以及当控制流进入并执行完基本块后涉及变量的取值变化。

由于在软件调试过程中，调试人员通常只关心与程序产生失效输出的位置紧密相关的其他语句，而非程序中所有语句，因此，我们并不需要不加筛选地提供程序所有的信息，相反，只需要提供少量的与可疑缺陷语句密切相关的语句和谓词。在方法实现时，我们借助队列机制，滚动记载程序最近的执行动态信息，用于交互调试阶段的可视化程序信息展示。图 2 给出了本文方法中描述程序执行队列及队列元素所使用的变量表的数据结构。

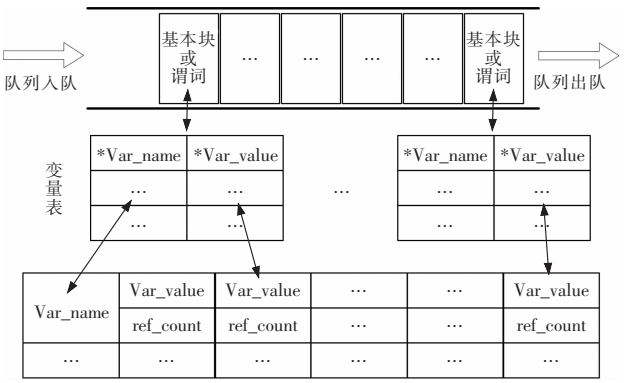

图 2　程序执行动态信息跟踪数据模型

Fig. 2　Data traced during program running

图 2 所示的数据结构设计借鉴了动态语言 Python 编译器对变量（对象）的处理方式。其中，队列用于保存一定长度的执行序列（以基本块和谓词为单位）。不保存所有的跟踪执行序列原因有两点：一是由于存储空间的制约无法全部保存，因此，程序执行过程可能会产生无限长的执行序列；二是没有必要全部保存这些信息。保存这些信息只是用于辅助程序员调试时判断决策，太多的信息对程序员的判断不但没有帮助，反而会降低判断效率，甚至会干扰程序员作出正确的判断。变量表用于存放队列中基本块或谓词所使用的变量（为提高处理速度并节省存储空间，采用指针形式存放链接到变量实际存储位置的地址）。变量表的构造可通过

$$\begin{cases} Out(b_i) = In(b_i) + Gen(b_i) - Killed(b_i) \\ In(b_i) = Out(b_{i-1}) \end{cases} \tag{1}$$

迭代计算得到。

此外,通过实时跟踪记录变量的值。为了能追踪到变量历史的取值信息,我们保存一定时期内的变量取值变化。不同基本块可能使用到相同的变量,但是在使用时其值可能变化,也有可能不变,因此,我们保存了变量值(Var_value)及变量的引用计数器(ref_count)。当变量值发生变化时,如果新的变量值在表中不存在,则新建一项并将引用计数器设置为 1;如果新的变量值在表中已经存在,则将对应项的引用计数器增加 1。在实现上述操作时,我们采用哈希函数映射的方法,可以快速找到并更新表中对应的项。

2　基于反馈优化的启发式软件调试框架

本文提出一种两阶段的启发式交互调试框架(如图 3 所示)。借助开源软件分析工具 Soot 实现对待测程序的静态分析以获取程序的结构信息,结合缺陷定位报告,计算程序后向切片及控制流程图等。同时,借助JVM 编程接口 JDI 实现对 Java 程序运行时的监控,跟踪并记录程序运行时的堆栈信息。用 dot 格式可视化展示程序运行时与当前审查语句相关的执行上下文。此外,根据调试人员的反馈,动态优化缺陷定位报告,实现交互式的调试,提高定位缺陷的效率。

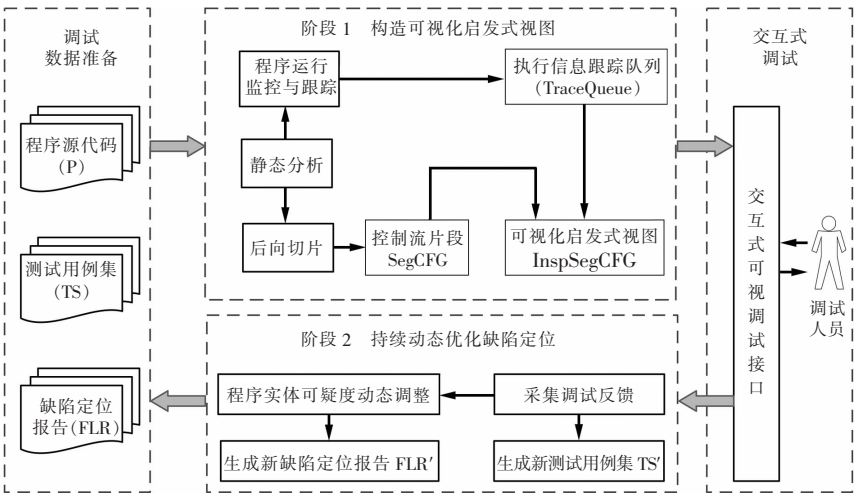

图 3　基于反馈优化的启发式软件调试框架

Fig. 3　Heuristic debugging framework based on the feedback optimization

在图 3 中,程序运行监控和静态分析(后向切片)模块分别用于搜集程序运行时的动态信息和静态切片。在计算静态切片时,根据当前审查语句所在函数的规模,确定需要显示的静态信息,并计算控制流图片段,随后我们在控制流图片段上添加程序执行时的动态信息,并生成 dot 格式文档,通过图中的交互式可视化调试接口展示给调试人员,用于辅助其对当前审查语句判断。此外,调试人员针对当前审查语句给出的判断结论,同样通过调试接口反馈给动态调整模块,根据本文提出的可疑度调整算法动态调整剩余语句的可疑度,从而推荐下一条审查语句。下面从启发信息可视化建模和缺陷报告动态优化两方面详细描述方法实现细节。

2.1　启发信息可视化建模

可视化的目的是为程序调试人员提供针对当前语句的辅助判断信息。过多的信息和过少的信息都不利于调试人员作出正确判断,并且遵循局部化原则,应该尽可能提供与当前语句密切相关的程序内容。一个容易想到的办法是提供当前审查语句所在函数的控制流程图,然后在图上增加其他辅助信息。但是这种方法面临一个困境:当前语句所在函数的规模有可能很大,导致给出的可视化图形模型信息量太大而不便于程序调试人员甄别当前语句是否正确;此外,当前语句所在函数的规模有可能很小(例如只有简单的几行代码),这种情况又导致可提供的信息量太少,同样不利于程序调试人员甄别当前语句。因此,按照当前审查的语句所在函数的规模,本文给出 3 种用于可视化的策略:

1)当所在函数规模适中时(如基本块个数低于某个上界 $\top$,高于某个下界 $\perp$),直接给出函数的控

制流程图，并将动态行为信息（如变量取值等）添加到控制流程图中对应的程序执行路径上加以显示。

2）当所在函数规模过大时（如基本块个数高于某个上界⊤），在该函数的控制流程图上进行删减，删除部分距离程序执行路径较远的分支，并将动态行为信息（如变量取值等）添加到控制流程图中对应的程序执行路径上加以显示。

3）当所在函数规模过小时（如基本块个数低于某个下界⊥），直接给出函数的控制流程图，同时，针对该函数的上一级调用函数进行分析，分析策略即是现在给出的3种策略。最后将动态行为信息（如变量取值等）添加到控制流程图中对应的程序执行路径上加以显示。

上述策略1）到策略3）中的选择依据是函数规模，其中的上界（⊤）和下界（⊥）是两个经验值，由调试人员依据历史调试经验设置，通常可以分别设置为20和8。

接下来我们详细阐述在这3种策略下，可疑语句辅助调试信息可视化的算法实现，下面算法1给出了具体的实现过程。

算法1 可疑语句辅助调试信息可视化算法

输入：

<*P*,*ExaStmt*>; //源代码和当前审查语句

<*TraeInfo*>; //程序调试信息

<⊤,⊥,*W*>; //函数规模的上界、下界和可视化窗口

输出：

DOTFile; //用于可视化显示的DOT文件

1：get *fun where ExaStmt* ∈ *fun*

2：*size* = *sizeof*(*fun*)

3：IF *size* ∈ [⊤,⊥] THEN

4： *SegCFG* = *S*1_*Gen*(*fun*,*ExaStmt*,*TraceInfo*,*W*)

5：END IF

6：IF *size* ∈ (0,⊥) THEN

7： *f* = *fun*

8： *Funlist* = ∅

9：WHILE *sizeof*(*f*) ∈ (0,⊥) ∧ *Funlist.size* < *W* DO

10：*Funlist.add*(*f*)

11：*f* = *getCallerof*(*f*)

12：END WHILE

13：*SegCFG* = *S*2_*Gen*(*FunList*,*ExaStmt*,*TraceInfo*,*W*)

14：END IF

15：IF *size* ∈ (⊥,∞) THEN

16： *SegCFG* = *S*3_*Gen*(*fun*,*ExaStmt*,*TraceInfo*,*W*)

17：END IF

18：*DOT File* = *GenerateDOT*(*SegCFG*)

19：RETURN *DOTFile*

在算法1中，输入的是程序源代码*P*、程序员当前审查的语句*ExaStmt*（即缺陷定位技术提供的怀疑度列表中可疑度最高的语句）、程序运行时跟踪的一组动态信息*TraceInfo*（数据结构包含一个队列及其指向的变量存储列表，见图2）、一组系统设置参数<⊤,⊥,*W*>用于表示函数规模的上界、下界及最终显示窗口中节点的最大数量。首先获取当前检查语句*ExaStmt*所在的函数*fun*，统计函数*fun*的规模（基本块数），然后依据系统设定的阈值（⊥,⊤）分别处理策略1）（第3—5行）、策略2）（第6—14行）及策略3）（第15—17行）所面临的情况，最后将处理的结果转化为用DOT语言书写的图形描述文件。算法1中的*S*1_*Gen*(*fun*,*ExaStmt*,*TraceInfo*,*W*)，*S*2_*Gen*(*FunList*,*ExaStmt*,*TraceInfo*,*W*)，以及*S*3_*Gen*(*fun*,*ExaStmt*,*TraceInfo*,*W*)函数以程序的控制流程图为基础，开展数据依赖分析，提取出与当前审查语句相关的基本块和谓词，并将程序执行中获取的动态信息添加到这些相关的基本块和谓词所在的节点。此外，为了避免最终显示过多的辅助信息，通过计算节点与当前执行路径的距离，删除原控制流程图上与当前执行路径关系不大的节点。节点与当前执行路径的距离定义为从节点到当前执行路径所需经过的最少节点数，为

$$\mathrm{dis}(node^*,path)=\min\|path^*\|\ \text{where}\ \forall path^*=\langle node^*,\cdots,node\rangle\wedge\exists node\in path, \tag{2}$$

其中*path**表示一条从*node**节点开始经历一系列节点到达最后节点*node*，并且*path**中只有唯一的节点*node*属于当前执行路径*path*。因此上述距离可以理解为最短路径。

下面依次介绍3个函数的算法实现：

算法2首先构造出从函数入口到指定语句（*ExaStmt*）的一个控制流程图片段*SegCFG*（第1行）；接着以当前语句（*ExaStmt*）为切片准则，在第1步得

到的控制流程图片段 *SegCFG* 上进行后向切片,同时以参数 *W* 限定后向切片的最大规模(第 2 行);随后针对后向切片中的每一依赖语句,计算其引用变量,并从跟踪信息 *TraceInfo* 中将这些引用变量在执行时所对应的值取出添加到控制流程图片段 *SegCFG* 中对应节点的变量位置(第 3—10 行)。需要注意的是:在算法 2 中,用 *node.stmt.v.values* 表示节点 *ndoe* 中有一个语句 *stmt*,并且 *stmt* 有一个引用变量 *v*。类似地,函数 *TraceInfo.getValues*(*node*,*v*)表示对象 *TraceInfo* 有处理函数 *getValues*,其功能是获取对应节点(*node*)中变量(*v*)的值。

算法 2 函数 *S1_Gen*(*fun*,*ExaStmt*,*TraceInfo*,*W*)

```
1: SegCFG = ConsSegCFG(fun,ExaStmt)
2: BWS = BackWardSlice(SegCFG,ExaStmt,W)
3: FOR each stmt ∈ BWS DO
4:     VAR = Ref(stmt)
5:     FOR each node ∈ SegCFG DO
6:         FOR each v ∈ VAR ∧ v is Reference by
           node DO
7:         node.stmt.v.values = TraceInfo.get
           Values(node,v)
8:         END FOR
9:     END FOR
10: END FOR
11: RETURN SegCFG
```

接下来介绍函数 *S2_Gen*(*FunList*,*ExaStmt*,*TraceInfo*,*W*)的算法实现(见算法 3)。

算法 3 函数 *S2_Gen*(*FunList*,*ExaStmt*,*TraceInfo*,*W*)

```
1: SegCFG = ConsSegCFG(FunList[0],ExaStmt)
2: FOR i from 1 to sizeof(Funlist) DO
3:     CallerStmt = FunList[i - 1].getCaller()
4:     SegCFG' = ConsSegCFG(Funlist[i],
       CallerStmt)
5:     SegCFG = SegCFG.Connect(SegCFG')
6: END FOR
7: BWS = BackWardSlice(SegCFG,ExaStmt,W)
8: FOR each stmt ∈ BWS DO
9:     VAR = Ref(stmt)
10:    FOR each node ∈ SegCFG DO
11:        FOR each v ∈ VAR ∧ v is Reference by
           node DO
12:            node.stmt.v.values =
               TraceInfo.getValues(node,v)
13:        END FOR
14:    END FOR
15: END FOR
16: RETURN SegCFG
```

算法 3 首先依次计算函数列表中所有函数的控制流程图片段,并将这些控制流程图片段按照调用和被调用的顺序拼接为一张大的控制流程图片段 *SegCFG*(第 1—6 行),其中函数 *fun.getCaller*()返回的是当前函数调用其他函数 *fun*()的语句集合,由于当前函数在调用其他函数 *fun*()时可能出现多个调用位置,因此函数 *fun.getCaller*()可能返回一组语句;随后的工作(第 7—15 行)类似于 *S1_Gen*()函数的对应部分功能,即计算当前审查语句 *ExaStmt* 的后向切片,并将切片中存在数据依赖关系的变量值通过提取跟踪信息 *TraceInfo* 获得(由函数 *TraceInfo.getValues*(*node*,*v*)实现)。接下来介绍当前审查语句所在函数规模十分庞大、超过限定阈值的情况下,如何显示辅助调试信息的过程(见算法 4)。

算法 4 函数 *S3_Gen*(*Fun*,*ExaStmt*,*TraceInfo*,*v*)

```
1: SegCFG = ConsSegCFG(Fun,ExaStmt)
2: SegPath = TraceInfo.ConsPath(SegCFG,ExaStmt,
   W)
3: FOR each node ∈ SegCFG DO
4:     IF dis(node,SegCFG) > |Min_Distance|
       THEN
5:         SegCFG.remove(node)
6:         SegCFG.delete(aEdges) Where <node,n'>
           ∨ <n',node> ∈ aEdges
7:     END IF
8: END FOR
9: BWS = BackWardSlice(SegCFG,ExaStmt,W)
10: FOR each stmt ∈ BWS DO
11:     VAR = Ref(stmt)
12:     FOR each node ∈ SegCFG DO
13:         FOR each v ∈ VAR ∧ v is Reference
            by node DO
```

14： *node.stmt.v.values = TraceInfo.getValues(nodv,v)*
15： END FOR
16： END FOR
17: END FOR
18: RETURN *SegCFG*

算法 4 首先计算函数 *fun*()的控制流程图片段 *SegCFG*(第 1 行)；接着依据程序的执行跟踪信息，构造一条以当前审查语句 *ExaStmt*，长度不超过 *W* 执行路径片段 *SegPath*(第 2 行)；由于第 1 步中得到的控制流程图片段规模偏大，如果直接显示给调试人员反而不利于调试人员快速获取准确、精炼的信息，需要对初始的控制流图片段进行删减，去除与本次执行关系不大的节点。针对控制流程图片段 *SegCFG* 中的节点，计算其与执行路径片段 *SegPath* 之间的距离，若距离超过系统设置的阈值 *Min_Distance*，则从 *SegCFG* 中移除节点，同时删除与该节点有关的边(第 3—8 行)。随后的工作(第 9—17 行)类似于 *S*1_*Gen*()函数及 *S*2_*Gen*()函数的对应部分功能，即以当前审查语句 *ExaStmt* 为切片准则，在经过删减的控制流程图片段上计算 *W* 规模的后向切片，然后提取切片中变量实际执行时的变量值，将变量取值信息添加到节点中，用于后续可视化显示。

2.2 基于调试反馈的可疑语句推荐

在回答了调试辅助信息识别和显示这两个问题之后，接下来讨论如何依据调试人员的反馈，动态调整缺陷定位的报告。在软件调试过程中，调试人员借助个人经验对当前语句是否是错误语句作出判断，然后继续审查在排序列表中的下一条可疑语句，然而并未利用调试人员的判断。我们认为，调试人员的判断可以用来指导调整后续语句的检查顺序，为此提出了一种动态调整语句怀疑度的算法。该算法基于如下 3 点假设：

假设 1 调试人员对所有审查语句作出的判断都是正确的；

假设 2 如果调试人员认定当前的审查语句 *s* 是正确的，则 *s* 语句的后向切片中的语句可疑度是被高估了的，应该调低后向切片中的语句可疑度；

假设 3 如果调试人员认定当前的审查语句 *s* 是错误的，则 *s* 语句的前向切片中的语句可疑度是被高估了的，应该调低其前向切片中的语句可疑度。

由于现阶段调试人员对审查语句的判断是基于经验的主观认定，人类脑力劳动的复杂性决定了其判断结果有可能出现错误，而且出错概率因人而异。虽然可以基于调试人员出错概率来建立后续动态反馈模型，但那样会大大增加模型的复杂度。为了分析方便，假设调试人员的经验丰富，不会出错(即满足假设 1)。

现有的统计缺陷定位技术通常预先假设程序中的语句均有可能出错。由 PIE 模型[14]可知：软件错误引发程序进入错误的内部状态，经过程序执行路径向程序出口处传播。在传播过程中，从程序开始到该软件错误处，与软件错误相关语句的可疑度值都被高估，因此应该加以修正(即满足假设 3)；与之相对应的，如果一个语句是正确的，则该语句到程序出口的所有相关语句的可疑度也被高估了(即满足假设 2)。

接下来讨论如何进行可疑度动态调整。本文的策略如下：

1)当调试人员认定当前审查语句 *s* 是正确语句时，计算语句 *s* 的后向切片 Ω，计算 Ω 中所有语句的怀疑度总和 *sum*，并将 *s* 语句的怀疑度值 *r* 除以 *sum*，得到一个调整度比值 ρ，最后将 Ω 中语句的怀疑度乘以$(1-\rho)$作为新的怀疑度值。

2)当调试人员认定当前审查语句 *s* 是错误语句时，计算语句 *s* 的前向切片 Ψ，计算 Ψ 中所有语句的怀疑度总和 *sum*，并将 *s* 语句的怀疑度值 *r* 除以 *sum*，得到一个调节度比值 ρ'，最后将 Ψ 中语句的怀疑度乘以$(1-\rho')$作为新怀疑度值。

算法 5 给出调整可疑度动态过程。S_{top} 为排序列表的最顶端的一个语句，其怀疑度值最高。算法 5 的第 2 行 *Feedback*(S_{top})表示调试人员对当前语句 S_{top} 作出判断。当调试人员认定当前语句 S_{top} 是正确语句时，由算法的第 3—10 行实现将当前语句 S_{top} 后向切片中所有语句的可疑度调低；当调试人员认定当前语句 S_{top} 是错误语句时，由算法的第 11—18 行实现将当前语句 S_{top} 的后向切片中所有语句的可疑度调低。即先计算 S_{top} 语句的后向切片集合 Ω，接着计算当前审查语句 S_{top} 的怀疑度值与切片中所有

语句的怀疑度比值ρ,用其作为调节比率,将切片中的所有语句的怀疑度动态更新。类似地,算法5中另一个变量ρ'即用于调节前向切片语句的可疑度。

算法5 基于调试反馈的可疑度动态更新算法

输入:

RankList

输出:

newRankList

1: $< S_{top}, R_{top} >= RankList.pop$

2: $assert = Feedback(S_{top})$

3: IF *assert is Right* THEN

4: $\Omega = BackwardSlice(S_{top})$

5: $o = \frac{R_{top}}{\sum_{S_i \in \Omega^R}}$

6: FOR each $S_i \in \Omega \wedge S_i \in RankList$ DO

7: $R_i = R_i \times (1 - \rho)$

8: $RankList.Update(S_i, R_i)$

9: END FOR

10: END IF

11: IF *assert is Wrong* THEN

12: $\Psi = ForwardSlice(S_{top})$

13: $o' = \frac{R_{top}}{\sum_{S_i \in \Psi^R}}$

14: FOR each $S_i \in \Psi \wedge S_i \in RankList$ DO

15: $R_i = R_i \times (1 - \rho')$

16: $RankList.Update(S_i, R_i)$

17: END FOR

18: END IF

19: $newRankList = RankList.Sort()$

20: RETURN *newRankList*

然而,在算法5中调试人员一次只能确认一条语句,调试的效率较低。事实上,在实际的软件调试过程中,调试人员有可能一次确认一组语句,部分识别为正确语句,部分识别为错误语句,从而可分为两个集合:认定为正确的语句集合(*Good*)和认定为错误的语句集合(*Bad*)。我们需要进一步改进算法5以处理多个语句反馈情况, 改进方法如算法6所示。

算法6 基于调试反馈的可疑度动态更新算法

输入:

RankList

输出:

newRankList

1: $< S_{top}, R_{top} >= RankList.pop$

2: $< Good, Bad >= Feedback(S_{top})$

3: FOR each $S_{good} \in Good \wedge S_{good} \in RankList$ DO

4: $\Omega = BackwardSlice(S_{good})$

5: $o = \frac{R_{good}}{\sum_{S_i \in \Omega^R}}$

6: FOR each $S_i \in \Omega \wedge S_i \in RankList$ DO

7: $R_i = R_i \times (1 - \rho)$

8: $RankList.Update(S_i, R_i)$

9: END FOR

10: END FOR

11: FOR each $S_{bad} \in Bad \wedge S_{bad} \in RankList$ THEN

12: $\Psi = ForwardSlice(S_{bad})$

13: $o' = \frac{R_{bad}}{\sum_{S_i \in \Psi^R}}$

14: FOR each $S_i \in \Psi \wedge S_i \in RankList$ DO

15: $R_i = R_i \times (1 - \rho')$

16: $RankList.Update(S_i, R_i)$

17: END FOR

18: END FOR

19: $RankList = RankList.Remove(Good, Bad)$

20: $newRankList = RankList.Sort()$

21: RETURN *newRankList*

算法6中的第3—10行、第11—18行与算法5的第3—10行、第11—18行对应部分的功能一致:都是先计算语句可疑度的调节率ρ和ρ',然后对原始怀疑度列表中的语句依据调节率调整得到新的可疑度评估值,最后对其进行排序以得到新的排序列表。两者不同之处在于:算法6增加了迭代处理多条语句的功能。当程序调试人员一次反馈给系统多条语句的判断,这些语句判断信息在算法6的第2行被组织成Good和Bad两个集合:算法第3—10行实现迭代调整Good集合中所有语句的后向切片中语句的怀疑度调整;算法第11—18行实现迭代

调整 Bad 集合中所有语句的前向切片中语句的怀疑度调整。此外，算法 6 的第 19 行用于移除可疑列表中已经确认的语句，因为这些已经由调试人员甄别过的语句无需重新确认。

进一步分析算法 5 和算法 6，可以发现，这两个算法中处理调试人员反馈的先后顺序有可能影响到最后生成的可疑语句的排序列表。目前，尚不清楚其具体的影响程度。一个主要的原因在于这两个算法的效果依赖于调试人员给出的判断是否正确，从而导致我们很难从理论上进行分析。下一步对提出的方法进行实验验证。

3 实证研究

3.1 实验基准程序

本文开发了一个用于调试 Java 程序的原型工具，在一组基准程序上开展用户使用研究。选择了 5 个 Java 程序作为实验对象（如表 1 所示），这些实验程序均是采用 Java 编程实现。对于每个实验对象程序，表 1 给出了实验对象名称（第 1 列）、实验对象的可执行代码行数 LOC（第 2 列）、测试用例数（第 3 列）和研究的错误版本数（第 4 列）。其中，前 3 个程序是 Java 版本的西门子程序（Siemens suits），由 Santelices 等[15]从 C 语言版本转换得到的 Java 版本程序。其余程序是从 SIR（subject infrastructure repository）[16]下载得到的。

表 1 实验基准程序特征

Tab. 1 Characteristics of studied subjects

程序名	代码行数	测试用例数	缺陷数
Print token	478	1 200	2
Schedule	290	2 650	2
Tot_info	283	1 052	2
Jtcas	181	1 201	2
Sorting	222	100	2

如表 1 所示，选取的实验对象的可执行代码行数为数百行。除了 Sorting 程序外，其他程序的测试用例均从文献[15–16]中的两个网站下载，仅选择了其中部分的测试用例，然后随机生成 100 组不同长度的数组用于测试 Sorting 程序。表 1 中列出的每个版本实验对象程序被注入单一缺陷，除了 Sorting 程序外，其余程序注入的缺陷均为其他研究人员设计的。针对 Sorting 程序，采用人工设计并注入了 2 个缺陷（包含谓词和赋值缺陷）。对于 Jtcas 程序，从 41 个缺陷版本中随机选择了 2 个缺陷版本以避免实验结果过分受该程序影响。对于其余的实验对象，同样随机选择 2 个缺陷版本。本文一共实证研究 10 个不同缺陷的定位。

3.2 实验设置

首先，选择表 1 中的基准测试程序，依次注入缺陷，并运行测试用例，收集覆盖信息，基于 GZoltar 得到初始的缺陷定位报告[17]。接着，邀请来自南通大学信息科学技术学院 16 名选修软件测试课程的学生参与该研究。为了避免由于编程经验差异而引起的偏差，对于每个程序，将学生随机分为两组，一组使用本文开发的工具 HDebug，另一组使用安装 GZoltar 插件 Eclipse 编程环境。记录下每组实验中修复每个缺陷耗费的时间，并汇总每组修复所有缺陷所需时间。

3.3 实验结果及讨论

分别统计两组学生修复 10 个缺陷耗费的时间（统计精确到分钟）。表 2 中，#1，#2，…，#8 分别为每组学生的编号。可以看出，使用工具 HDebug 的每位学生耗费的时间均小于使用 GZolta 工具所耗费的时间，平均节省了约 20%~35%的调试时间。

表 2 修复全部缺陷所需时间

Tab. 2 Time cost for bug fixing of all defects min

方法	#1	#2	#3	#4	#5	#6	#7	#8
HDebug	43	54	47	55	46	38	41	49
GZoltar	59	74	68	69	71	57	58	64

下面进一步分析分别使用 HDebug 和 GZoltar 在 5 个基准程序上所耗费的时间。为了平衡不同基准程序调试时间的差异性，统一对两组同学使用工具在每个基准程序上调试时间代价进行了变换，统一映射到[0，10]区间内的 1 个值。映射方法为

$$\begin{cases} a_i' = \dfrac{a_i - \mathrm{Min}_k}{\mathrm{Max}_k - \mathrm{Min}_k} \times 10 \\ b_i' = \dfrac{b_i - \mathrm{Min}_k}{\mathrm{Max}_k - \mathrm{Min}_k} \times 10 \end{cases}, \tag{3}$$

其中 Max_k，Min_k 分别为两组学生中调试同一基准程序耗费时间的最大值和最小值。a_i 和 b_i（$i \in [1,8]$）分

别为学生调试某一基准程序缺陷耗费的时间。图 4 给出了两组调试技术所耗费时间的对比。

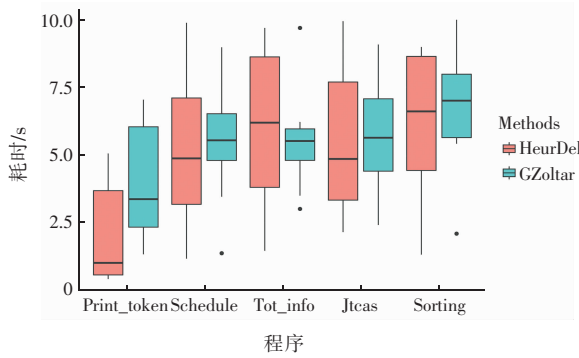

图 4 两种程序调试技术的时间代价对比

Fig. 4 Comparison of time costs between two debug techniques

从图 4 可以看出,除了 Tot_info 基准程序之外的其他 4 个基准程序,采用 HDebug 技术的调试时间代价均低于 GZoltar(中位数均低于对照技术)。但图 4 也表明,基于 HDebug 技术的调试代价的稳定性不高。为了进一步验证两种技术的性能差异,引入以下原假设和备选假设:

H0:两种调试技术之间没有显著性能差异。

H1:两种调试技术之间性能差异非常明显。

采用 Wilcoxon 符号秩检验来评估原假设,如果 p 值 < 0.05,我们将拒绝它。在对比每个基准程序的基础上分别采用 HDebug 和 GZoltar 技术的时间代价,利用 R 计算其对比时的 p 值。观察到在5 个基准程序上计算得到的 p–value 均小于 0.05, 这意味着性能改进是显著的。

4 相关工作

软件调试工具可以分为两大类:手工调试工具和统计调试工具。手工调试工具属于传统工具,基于一个测试用例的单步执行以定位程序缺陷,其主要方式是集成在软件开发环境(integrated development environment,IDE)中;统计调试工具是基于统计缺陷定位的方法,通过统计分析软件的多个测试执行的历史信息以定位程序缺陷。本文工作与第二类工具紧密相关,因此,本节主要讨论统计调试工具的相关研究,这类工具主要表现为两种形态:独立工具形式和集成开发环境形式。

4.1 独立工具形式

独立形式的调试工具内化了缺陷定位算法,通过分析程序静态信息并收集程序运行信息指导程序员定位软件缺陷,有自己一套可视化的图形用户接口。例如,早期的数据呈现调试器(data display debugger,DDD)通过分析一次执行过程中的程序指令序列指导程序员发现软件缺陷,并且支持程序员单步执行以分析软件行为,提供软件局部变量(对象)的状态信息[18]。后来文献[10]提出的 Tarantula 进一步统计程序多次执行(成功和失败)过程中的覆盖信息,借助一个怀疑度计算公式度量程序中每一个语句(程序实体)的可疑度,并用不同颜色在程序代码上标识加以区分。自此大批基于统计缺陷定位的软件调试工具开始涌现。如 Sober[19]、Ochiai[20]、Crosstab[21]、Zoltar[22]、DStar[23]等。Mecca 等[24]通过引入 ACME(演化变体的代码动画),首次建立一个可扩展的代码可视化工具。这些方法大多只是用颜色标记程序实体出错的概率,或者简单给出程序执行语句上下文信息;与他们不同是,我们的方法不仅优化了启发式(上下文)信息并可视化这些信息以辅助程序调试,还能根据程序员的反馈动态调整缺陷定位报告,从而能够进一步提升缺陷定位效率。

4.2 集成开发环境形式

软件开发人员倾向于使用相对更舒适的工具。集成软件开发环境通常会提供许多辅助软件开发的工具,这些工具不仅可以提供有关代码编辑的有用功能(如行号和语法高亮显示),而且还可以提供有关项目组织、代码完成、集成帮助,以及在给定阶段进行断点和查看系统状态的功能,也可以分步执行以分析每一行的系统行为。与 IDE 不同,在程序员编辑源代码的同一环境中,许多工具相互作用在一起,因此自动软件调试工具往往是独立的解决方案,通常以插件的形式被安装到集成开发环境。分析整个系统的一种方法是使用代码覆盖工具,利用这些工具可以查看在系统测试运行期间执行了哪些代码行。通过此信息,可以查看程序是否按预期运行,如果不运行,则查看哪些行与系统故障有关。例如,早期 Hoffmann 等[25]提出的基于集成开发环境 Eclipse 的插件 EclEmma,能依据 IDE 收集的覆盖信息,使用不同的颜色来显示它是否已执行(不计算其失败概率),分别为完全执行(绿色)、部分执行(黄色)和未执行(红色)。后来,Bouillon 等[26]提出的

EZUnit 能够在利用颜色标识代码是否执行的基础上，进一步给出某次测试用例执行时的程序调用图。Lin 等[27]开发了一个 Demo 版的 Eclipse 插件Microbat，其建立在轻量级人类反馈的基础上，并将反馈视为程序的一部分推断程序执行的可疑步骤。最新版本的 SonarSource 提供了 SonarLint IDE 开发插件，基于代码的静态分析技术，支持 20 多种编程语言代码的扫描，并提供详细的问题分析和缺陷修复建议[28]。在预测缺陷方面，曲豫宾等[29]引入基于信息熵的主动学习策略，提升人工标注的准确性，从而提高了预测缺陷的准确度。虽然与开发环境集成增加了调试工具的开发难度，但一方面软件开发人员更习惯使用熟悉的 IDE 定位并修复缺陷，另一方面集成到开发环境有助于从开发环境获取程序的运行信息（如覆盖信息、执行结果等），不需要额外开发跟踪工具，降低了工作量。与这类工具相比，本文的方法增加了基于程序员调试反馈的动态调整过程。

需要指出的是，目前的调试工具提供的可视化信息作用还十分有限，对程序员的调试反馈利用也很少，导致开发人员浪费大量的时间来尝试理解调试工具处理的所有信息。开发人员必须在调试过程中于应用程序之间进行切换（在一个工具中查找软件故障并在另一个工具中对其进行纠正），并且无法获得 IDE 集成的所有好处，例如项目和源代码检测，甚至与代码交互的编辑器，这无疑会增加调试过程中的时间代价。

5 结论与展望

本文提出了一种基于反馈优化的启发式软件调试框架，通过可视化展示缺陷相关的上下文信息，旨在迭代地指导定位缺陷的根本原因，同时能根据程序员当前的反馈，优化已有的缺陷定位报告，从而优化下一轮的缺陷调试。初步的实验表明，本文的方法能够在较大程度上加速软件调试过程。在未来工作中，首先，进一步优化工具的界面，优化启发信息的内容和表现形式，实现启发信息可视化时的友好性；其次，考虑设计成 Eclipse 插件，并和 Junit 集成，以实现程序运行信息收集；然后，在基于程序员反馈的动态缺陷定位优化时，考虑程序员反馈信息的正确概率，引入程序员调试反馈的历史信息，提升动态反馈的准确度，从而得到更优的缺陷定位报告；最后，考虑增加新功能，如动态展示程序出错前后的执行状态转换过程，增加程序员反馈的途径等，增加推荐缺陷修复补丁等，最终提升调试的效率。

参考文献：

[1] WONG W E, GAO R Z, LI Y H, et al. A survey on software fault localization[J]. IEEE Transactions on Software Engineering, 2016, 42(8):707-740.

[2] JONES J A. Fault localization using visualization of test information[C]//Proceedings of the 26th International Conference on Software Engineering, May 28, 2004, Edinburgh, UK. New York:IEEE Xplore, 2004:54-56.

[3] PERSCHEID M, SIEGMUND B, TAEUMEL M, et al. Studying the advancement in debugging practice of professional software developers[J]. Software Quality Journal, 2017, 25(1):83-110.

[4] HAO D, ZHANG L, XIE T, et al. Interactive fault localization using test information[J]. Journal of Computer Science and Technology, 2009, 24(5):962-974.

[5] ALI S, ANDREWS J H, DHANDAPANI T, et al. Evaluating the accuracy of fault localization techniques[C]// Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, November 16-20, 2009, Auckland, New Zealand. New York:IEEE Xplore, 2009:76-87.

[6] QI Y H, MAO X G, LEI Y, et al. Empirical effectiveness evaluation of spectra-based fault localization on automated program repair[C]//Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, July 22-26, 2013, Kyoto, Japan. New York:IEEE Xplore, 2013:828-829.

[7] NAGARAJAN V, JEFFREY D, GUPTA R, et al. A system for debugging via online tracing and dynamic slicing [J]. Software:Practice and Experience, 2012, 42(8):995-1014.

[8] WONG W E, GAO R, LI Y, et al. A survey on software fault localization[J]. IEEE Transactions on Software Engineering, 2016, 42(8):707-740.

[9] DIGIUSEPPE N, JONES J A. Fault density, fault types, and spectra-based fault localization[J]. Empirical Software Engineering, 2015, 20(4):928-967.

[10] JONES J A, HARROLD M J. Empirical evaluation of the

tarantula automatic fault-localization technique[C]//Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, November 7-11, 2005, Long Beach, California, USA. New York:ACM, 2005: 273-282.

[11] NAISH L, LEE H J, RAMAMOHANARAO K. A model for spectra-based software diagnosis[J]. ACM Transactions on Software Engineering and Methodology, 2011, 20(3): 1-32.

[12] WONG W E, DEBROY V, GAO R Z, et al. The DStar method for effective software fault localization[J]. IEEE Transactions on Reliability, 2014, 63(1):290-308.

[13] ZHANG Z Y, JIANG B, CHAN W K, et al. Fault localization through evaluation sequences[J]. Journal of Systems and Software, 2010, 83(2):174-187.

[14] VOAS J M. PIE:a dynamic failure-based technique[J]. IEEE Transactions on Software Engineering, 1992, 18(8): 717-727.

[15] SANTELICES R, JONES J A, YU Y B, et al. Lightweight fault-localization using multiple coverage types[C]//Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, May 16-24, 2009, Vancouver, BC, Canada. New York:IEEE Xplore, 2009:56-66.

[16] DO H, ELBAUM S, ROTHERMEL G. Supporting controlled experimentation with testing techniques:an infrastructure and its potential impact[J]. Empirical Software Engineering, 2005, 10(4):405-435.

[17] CAMPOS J, RIBOIRA A, PEREZ A, et al. GZoltar:an eclipse plug-in for testing and debugging[C]//Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, September 3-7, 2012, Essen, Germany. New York:IEEE Xplore, 2012:378-381.

[18] ZELLER A, LÜTKEHAUS D. DDD:a free graphical front-end for UNIX debuggers[J]. ACM SIGPLAN Notices, 1996, 31(1):22-27.

[19] LIU C, YAN X F, FEI L, et al. Sober:statistical model-based bug localization[J]. ACM SIGSOFT Software Engineering Notes, 2005, 30(5):286-295.

[20] ABREU R, ZOETEWEIJ P, VAN GEMUND A J C. On the accuracy of spectrum-based fault localization[C]//Proceedings of the Testing:Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), September 10-14, 2007, Windsor, UK. New York:IEEE Xplore, 2007:89-98.

[21] WONG E, WEI T T, QI Y, et al. A crosstab-based statistical method for effective fault localization[C]//Proceedings of the 2008 1st International Conference on Software Testing, Verification, and Validation, April 9-11, 2008, Lillehammer, Norway. New York:IEEE Xplore, 2008:42-51.

[22] JANSSEN T, ABREU R, VAN GEMUND A J C. Zoltar:a toolset for automatic fault localization[C]//Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, November 16-20, 2009, Auckland, New Zealand. New York:IEEE Xplore, 2009:662-664.

[23] WONG W E, DEBROY V, LI Y H, et al. Software fault localization using DStar (D*)[C]//Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, June 20-22, 2012, Gaithersburg, MD, USA. New York:IEEE Xplore, 2012:21-30.

[24] MECCA G, SANTORO D, SILENO N, et al. Diogene-CT:tools and methodologies for teaching and learning coding[J]. International Journal of Educational Technology in Higher Education, 2021, 18:12.

[25] HOFFMANN M R, BROCK J, MANDRIKOV E. Code coverage analysis for eclipse[R/OL]. (2007-10-10)[2021-10-10]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.233.1138&rep=rep1&type=pdf.

[26] BOUILLON P, KRINKE J, MEYER N, et al. EzUnit:a framework for associating failed unit tests with potential programming errors[M]//Agile Processes in Software Engineering and Extreme Programming, Berlin:Springer, 2007: 101-104.

[27] LIN Y, SUN J, XUE Y X, et al. Feedback-based debugging[C]//Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), May 20-28, 2017, Buenos Aires, Argentina. New York:IEEE Xplore, 2017:393-403.

[28] SonarSource. Code quality and code security[CP/OL]. (2015-10-15)[2021-10-10]. https://www.sonarsource.com/.

[29] 曲豫宾, 陈翔. 基于代价敏感主动学习的软件缺陷预测方法[J]. 南通大学学报(自然科学版), 2019, 18(1):9-15.
QU Y B, CHEN X. Software defect prediction method based on cost-sensitive active learning[J]. Journal of Nantong University (Natural Science Edition), 2019, 18(1): 9-15. (in Chinese)

(责任编辑:仇慧)