软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2019,30(2):266−281 [doi: 10.13328/j.cnki.jos.005677]

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

# 增强上下文的错误定位技术*

张 卓[1], 谭庆平[1], 毛晓光[1], 雷 晏[2,3], 常 曦[4], 薛建新[1,4]

[1](国防科技大学 计算机学院,湖南 长沙 410072)

[2](重庆大学 大数据与软件学院,重庆 400044)

[3](信息物理社会可信服务计算教育部重点实验室(重庆大学),重庆 400044)

[4](上海第二工业大学 计算机与信息工程学院,上海 200127)

通讯作者: 雷晏, E-mail: yanlei.cs@outlook.com

**摘 要**: 错误定位就是寻找程序错误的位置.现有的错误定位方法大多利用测试用例的覆盖信息,以标识一组导致程序失效的可疑语句,却忽视了这些语句相互作用导致失效的上下文.因此,提出一种增强上下文的错误定位方法Context-FL,以构建上下文的方式来优化错误定位性能.Context-FL 利用动态切片技术构建数据与控制相关性的错误传播上下文,显示了导致失效的语句之间传播依赖关系;然后,基于可疑值度量来区分上下文片段中不同语句的可疑度;最后,Context-FL 以标记可疑值的上下文作为定位结果.实验结果表明,Context-FL 优于 8 种典型错误定位方法.

**关键词**: 错误定位;上下文;动态切片;SFL;可疑值

**中图法分类号**: TP311

中文引用格式: 张卓,谭庆平,毛晓光,雷晏,常曦,薛建新.增强上下文的错误定位技术.软件学报,2019,30(2):266−281. http://www.jos.org.cn/1000-9825/5677.htm

英文引用格式: Zhang Z, Tan QP, Mao XG, Lei Y, Chang X, Xue JX. Effective fault localization approach based on enhanced contexts. Ruan Jian Xue Bao/Journal of Software, 2019,30(2):266−281 (in Chinese). http://www.jos.org.cn/1000-9825/5677.htm

## Effective Fault Localization Approach Based on Enhanced Contexts

ZHANG Zhuo[1], TAN Qing-Ping[1], MAO Xiao-Guang[1], LEI Yan[2,3], CHANG Xi[4], XUE Jian-Xin[1,4]

[1](College of Computer, National University of Defense Technology, Changsha 410072, China)

[2](School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China)

[3](Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, Chongqing 400044, China)

[4](College of Computer and Information Engineering, Shanghai Polytechnic University, Shanghai 200127, China)

**Abstract**: Fault localization is a process to determine the root causes of abnormal behavior of a faulty program. Most existing fault localization approaches usually utilize coverage information of test cases to identify a set of isolated statements responsible for a failure, but do not show how these statements act on each other to cause the failure. Thus, this study proposes Context-FL: An approach enhancing contexts for these existing localization approaches by constructing contexts for fault localization optimization. Specifically, Context-FL uses dynamic slicing technology to construct a context showing how data/control dependence propagates to cause the faulty output. Then, it adopts suspiciousness evaluation to distinguish the elements of the context in terms of the suspiciousness being faulty. Finally, Context-FL outputs the context with suspiciousness as the localization result. The empirical results show that the proposed approach significantly outperforms 8 state-of-the-art fault localization techniques.

**Key words**: fault localization; context; dynamic slice; SFL; suspiciousness

* 基金项目: 国家自然科学基金(61602504, 61672529, 61379054, 61502296)
Foundation item: National Natural Science Foundation of China (61602504, 61672529, 61379054, 61502296)
收稿时间: 2017-11-06; 修改时间: 2017-03-26, 2018-06-05, 2018-08-11; 采用时间: 2018-10-08

软件调试是软件开发过程中非常耗时的活动.为了提升软件调试的性能,研究人员提出了许多错误定位方法(例如文献[1−6]).其中,基于频谱的错误定位方法(spectrum-based fault localization,简称 SFL)[2]是被广泛研究和使用的错误定位方法,展示了其在减少检测到错误代码的百分比方面的有效性[1,5,7,8].SFL 尝试通过度量每个错误语句的可疑值来识别错误语句.首先,SFL 利用插桩工具来对程序进行插桩,自动收集语句在测试用例执行中的覆盖数据,以此构建程序谱;然后,基于程序谱,SFL 利用不同的度量公式来评估语句为错误的可疑性;最后,SFL 输出以可疑性为排名的语句列表.

然而,SFL 关注语句选择和排名,却忽略调试工程师所需要关心的上下文信息和可疑语句之间的传播关系,这些关系可以方便其理解和分析失效原因.研究[9,10]表明,缺乏上下文信息可能会降低错误定位性能.因此,有必要构建上下文信息来优化错误定位.

为了构建上下文信息,本文发现,程序切片技术[11,12]可能成为解决这个问题的候选者.程序切片技术提取程序语句的数据与控制依赖关系,以选出影响程序输出的语句子集.它将这个语句子集命名为切片.可以发现,切片可以显示切片中语句之间的数据与控制依赖关系,以及如何传播到输出的过程.也就是说,切片显示影响程序输出的相关语句相互作用的上下文信息.这意味着错误定位可以利用切片来提供上下文信息.现有的程序切片技术大致可分为静态切片和动态切片两大类:静态切片根据数据和控制依赖关系分析程序,而不运行程序;动态切片沿着执行路径的依赖关系进行分析.在实际应用中,静态切片具有更高的时间复杂度,切片体积较大,可能会包含过多与具体错误行为无关的语句;动态切片则具有较高的空间复杂度,切片体积较小,可能会出现切片丢失错误语句的情况[3,13].由于动态切片体积较小且与具体执行相关联,与静态切片相比,动态切片使用更为广泛.因此,本文将动态切片纳入研究.

虽然程序切片提供了上下文信息,但是它对切片内的语句赋予相同的可疑度.换言之,即使与原始程序大小相比,在切片体积明显变小的情况下,它也不能区分切片中的哪个语句更可疑.然而,切片仍然包含许多语句,特别是当程序规模较大时,需要进行大量的人工查找.因此,衡量一个片段中的语句的可疑性至关重要.

基于上述分析,本文试图将程序切片融入错误定位中,自动获取调试工程师所需的上下文信息,并进一步将可疑性度量引入其中.因此,在前期研究[10]的基础上,本文提出了 Context-FL,通过使用动态切片来构建上下文信息,以及基于 SFL 的可疑性度量来评估上下文中语句的可疑性.本文在 14 个开源程序的大量错误版本上展开了实验,与 8 种个典型错误定位方法进行了对比.实验结果表明,与 8 种典型错误方法相比,Context-FL 的检查代码的平均成本降低最多可达 52.79%.

本文第 1 节介绍基于频谱的错误定位和动态切片.第 2 节具体描述本文方法 Context-FL.第 3 节给出 Context-FL 的实现.第 4 节描述实验以及结果分析.第 5 节介绍相关工作.第 6 节总结本文.

## 1 相关背景

### 1.1 基于频谱的错误定位

基于频谱的错误定位技术(spectrum-based fault localization,简称 SFL)意在构建程序谱,并使用可疑评估公式来衡量每个语句的可疑值.程序谱是数据的集合,包括语句执行覆盖信息以及测试用例是否通过的信息,是程序动态行为的一种具体视图[5].

SFL 基于程序谱定义了 4 个参数,并以此构建可疑评估公式.4 个参数 $a_{np}$、$a_{nf}$、$a_{ep}$、$a_{ef}$ 与具体语句绑定,分别表示执行或者未执行该语句的通过或者失败测试用例数.下标的第 1 部分表示该语句是否被测试用例所执行(e)或者未被执行(n),而第 2 部分则表示测试用例结果是通过(p)或失败(f).

表 1 给出了 5 个测试用例$\{T_1,T_2,...,T_5\}$的示例,以表明如何计算每个语句的 4 个参数.如表 1 所示,前 3 行表示 3 个程序语句,最后一行为结果(0 表示通过,1 表示失败).左 5 列表示 5 个测试用例,每个单元格表示语句在对应测试用例的执行情况(1 为执行,0 为未执行).例如,语句 1 中的 $T_1$ 值为 1 表示测试用例 $T_1$ 运行时执行了 Statement 1.对于语句 1,$a_{ep}$ 的值为 2 表示执行了 Statement 1 的通过测试用例有 2 个,$a_{ef}$ 的值为 1 表示执行了 Statement 1 的失败测试用例有 1 个,$a_{np}$ 的值为 0 表示结果通过但未执行 Statement 1 的测试用例不存在.

**Table 1** An example illustrating program spectra

**表 1** 程序谱示例

| | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $a_{np}$ | $a_{nf}$ | $a_{ep}$ | $a_{ef}$ |
|---|---|---|---|---|---|---|---|---|---|
| Statement 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 2 | 1 |
| Statement 2 | 0 | 1 | 0 | 0 | 1 | 1 | 2 | 2 | 0 |
| Statement 3 | 1 | 0 | 1 | 1 | 0 | 2 | 0 | 1 | 2 |
| Test result | 1 | 0 | 0 | 1 | 0 | – | – | – | – |

假设程序 $P=\{s_1,s_2,...,s_n\}$,这表明 $P$ 包含 $n$ 个语句.该程序被测试套件 $T=\{T_1,T_2,...,T_m\}$执行,$T$ 包含 $m$ 个不同测试用例且至少包含 1 个失败测试用例(参见图 1).如图 1 所示,测试用例运行完毕后,语句覆盖情况与测试结果组成 $M\times(N+1)$的矩阵.左边 $M\times N$ 矩阵表示语句执行情况.如果语句 $s_j$ 由测试用例 $t_i$ 执行,则 $x_{ij}$ 为 1;否则为 0.矩阵最右侧的结果向量 $\boldsymbol{e}$ 表示测试结果.如果 $t_i$ 是失败测试用例,则 $\boldsymbol{e}_i$ 为 1;否则为 0.

$$\begin{matrix} & N\text{ statements} & & & \text{errors} \end{matrix}$$
$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} \begin{bmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \\ \vdots \\ \boldsymbol{e}_M \end{bmatrix}$$

Fig.1 Coverage of $M$ test cases after executed

图 1 $M$ 个测试用例执行后的覆盖信息

基于图 1 矩阵,SFL 计算出每个语句的 4 个参数 $a_{np}$、$a_{nf}$、$a_{ep}$、$a_{ef}$,然后使用可疑性度量公式来计算每个语句的可疑值.SFL 构建可疑性度量公式的基本思想:希望错误语句具有相对较高的 $a_{ef}$ 值、较低的 $a_{ep}$ 值.其理想情况是错误语句的 $a_{ef}$ 值为最大,而 $a_{ep}$ 值为最小.这意味着错误语句在所有失败测试用例中被执行得最多,而通过的测试用例中被执行得最少.可疑性度量公式会赋予错误语句最大可疑值.一般来说,不同度量公式赋予通过和失败测试用例的不同权值,从而产生不同的排名.研究人员对他们提出的方法进行了实证调查,其中,Xie 等人[2,14]理论分析了大量 SFL 度量公式,总结出了 5 种较优公式.除了这 5 种公式,Ochiai、Barinel 和 D*是最新错误定位评价研究[4]中排名前 3 的错误定位方法.表 2 给出了这 8 种方法的可疑性度量公式.需要说明的是,D*公式中的*通常被赋值为 2[4],这也是本文实验所采取的赋值.

**Table 2** Maximal formulas of SFL

**表 2** 最优 SFL 公式

| Name | | Formulas | Name | Formulas |
|---|---|---|---|---|
| ER1′ | Naish1 | $\begin{cases}-1, & \text{if } a_{ne}>0\\ a_{np}, & \text{if } a_{ne}\leqslant 0\end{cases}$ | GP02 | $2(a_{ef}+\sqrt{a_{np}})+\sqrt{a_{ep}}$ |
| | Naish2 | $a_{ef}-\dfrac{a_{ep}}{a_{ep}+a_{np}+1}$ | GP03 | $\sqrt{\lvert a_{ef}^2-\sqrt{a_{ep}}\rvert}$ |
| | GP13 | $a_{ef}\left(1+\dfrac{a_{ep}}{2a_{ep}+a_{ef}}\right)$ | GP19 | $a_{ef}\sqrt{\lvert a_{ep}-a_{ef}+a_{nf}-a_{np}\rvert}$ |
| ER5 | Wong1 | $a_{ef}$ | Dstar(D*) | $\dfrac{a_{ef}^*}{a_{nf}+a_{ep}}$ |
| | Russel&Rao | $\dfrac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$ | Barinel | $1-\dfrac{a_{ep}}{a_{ef}+a_{ep}}$ |
| | Binary | $\begin{cases}0, & \text{if } a_{ne}>0\\ 1, & \text{if } a_{ne}\leqslant 0\end{cases}$ | Ochiai | $\dfrac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$ |

SFL 信息收集和工作原理比较简单,并能准确定位错误语句,因而得到了广泛的研究和应用.然而它并没有

给出执行程序的上下文信息,也没有考虑到语句之间的依赖关系.其产生的语句排名列表没有显示语句之间依赖关系,有不少可疑性较高的语句与程序错误输出并没有关联.这种语句孤立与割裂,加重了开发人员理解和搜索问题的难度.当程序规模大时,不相关的语句将迅速增加,这可能对软件调试有很大的干扰.因此,需要构建与错误输出相关联的可疑上下文,以便更好地了解错误问题,并进一步减少手动查找范围.

### 1.2 动态切片

程序切片技术通过提取程序语句的数据和控制依赖关系,选择影响错误输出的语句的子集来构建上下文信息.程序切片技术首先由 Weiser 引入[11],已经在程序测试、程序理解和软件维护方面得到广泛应用.给定一个程序 $P$ 和输入 $I$,当 $P$ 根据 $I$ 执行时(例如文献[15−17]),切片技术切出直接或间接地影响一些变量出现的值的语句集合.静态程序切片由程序 $P$ 中可能会影响变量 $v$ 在某个点 $p$[11,18]的值的所有语句组成,但它涉及所有可能的程序执行.然而,调试常处理特定的错误执行,以此查找导致该执行不正确的原因.因此,本文使用保留特定程序输入的程序行为切片,而不是针对所有输入的集合的静态切片.这种切片被称为动态切片[12].动态切片沿着执行路径收集运行时的信息,与静态切片相比可以显著地减少切片的大小[19,20].静态切片基于静态数据相关性计算,其切片中的语句数量很多,且通常根据对象在程序中的存在和指针进行保护.相对而言,动态切片针对具体执行捕获动态数据和控制依赖关系,更有针对性且精度较高,有助于调试人员理解和缩小其搜索范围.因此,本文采用动态切片技术来构建上下文.

动态切片标准可以表示为元组 $C=(S_k,V,I)$,其中,$S_k$ 表示语句变量为 $V$ 的输入 $I$ 上的语句 $S$ 的 $k$ 次出现.如图 2 所示,标准($n$=$9_1$,$v$,2),其中,$9_1$ 表示第 1 次出现的语句 9.由于输入 $n$=2,这表明该循环执行了 2 次,即 $v$=5 和 $v$=6 分别执行 1 次.因为通过在第 2 个循环中分配 5 到 $v$ 而使第 1 个循环中的变量 $v$ 的赋值发生位移,所以可以从动态切片中省略 if 语句的 else 分支.而标准(9,$v$)下的程序的静态切片包括了整个程序.显然,动态切片更有利于缩小查找范围,且与具体执行相关,更具有针对性.

| Program $P$ | Dynamic slice | Static slice |
|---|---|---|
| 1. *read*($n$);<br>2. $i=1$;<br>3. while ($i<=n$){<br>4. if ($i$%2=0)<br>5. $v=5$;<br>6. else<br>7. $v=6$;<br>8. $i=i+1$}<br>9. *output*($v$); | Input: $n=2$;<br>Output: $v=5$; | Input: $n=2$;<br>Output: $v=5$; |
| | Slicing criterion:<br>($9_1$,$v$,2) | Slicing criterion:<br>(9,$v$) |
| | Slice result:<br>{1,2,3,4,5,8,9} | Slice result:<br>{1,2,3,4,5,6,7,8,9} |

Fig.2 Examples of dynamic slice and static slice

图 2 动态切片与静态切片的例子

## 2 CONTEXT-FL 算法设计

### 2.1 算法设计

Context-FL 的基本思想是:将动态切片技术应用于 SFL,构建可疑上下文及其元素.Context-FL 首先构建上下文,显示语句在程序执行过程中如何影响和被影响;然后,利用 SFL 评估上下文中元素的可疑值.

算法 1 和算法 2 描述了 Context-FL 方法的关键部分.

**算法 1**. Get suspicious context.

1: **Function**: *GetSuspiciousContext*($V$,$E$,$T$)

2: **Inputs**: ***E***: The location of the output statement that outputs the error result

3: ***V***: One of the variables or a collection of variables

4: ***T***: Test *cases*{$T_1$,$T_2$,...,$T_n$}.

5: **Outputs**: *Context*: The suspicious context that contains a collection of Statements $\{s_1,s_2,...,s_n\}$ that are dynamically related to the error output.
6: Begin
7: *Context*=;
8: For Every test case $T_i$ in the test cases do
9: if $T_i$ is failed
10: *Context*=*Context GetContext*($V$,$E$,$T_i$)
11: End if
12: End for
13: Return *Context*
14: End begin

**算法 2**. Get statements suspiciousness in the context.

1: **Function**: *GetSuspiciousness*($S$,$M$)
2: **Inputs**: ***S***: *Context*
3: ***M***: An input matrix that can be used to calculate the SFL formulas.
4: **Outputs**: *EnhancedContext*.
5: Begin
6: *EnhancedContext*=;
7: For Every statement $s_i$ in $S$ do
8: *ElementInfo*($s_i$)=*SearchElementInfo*($s_i$,$M$)
9: *GetSuspiciousness*($s_i$,*ElementInfo*($s_i$))
10: *EnhancedContext*=*EnhancedContext GetSuspiciousness*($s_i$)
11: *Order*(*EnhancedContext*)
12: End for
13: Return *EnhancedContext*
14: End begin

首先对算法 1 和算法 2 进行解释.本节采用第 1 节定义的程序 $P$ 和测试用例套件 $T$.即程序 $P$ 由一组表示为$\{s_1,s_2,...,s_m\}$的语句组成,相应的测试套件 $T=\{T_1,T_2,...,T_n\}$.*Context*$\{s_1,s_2,...,s_n\}$表示与错误输出相关的语句的集合.*EnhancedContext*$\{s_1,s_2,...,s_n\}$表示带可疑值标记的上下文语句的集合.接下来,将讨论 Context-FL 算法中的每个步骤.

- 步骤 1:构建可疑片段和他们的元素.

如算法 1 所示,该步骤迭代地构造可疑上下文,直到遍历完所有失败测试用例为止.第 2 行~第 4 行表示算法 1 所需要的输入变量,$E$ 表示输出错误结果的输出语句,$V$ 是语句中变量之一或变量集合,$T$ 表示测试用例套件.第 5 行是算法 1 的输出,即上下文,由一组至少影响 1 个错误输出的语句集合组成.由于上下文由导致程序错误输出的执行轨迹构成,它包含了导致失败的测试用例的错误语句.在第 7 行中,上下文初始化为空集.第 8 行和第 9 行表示循环依次遍历完每个失败测试用例.第 10 行获取可疑上下文片段,即每个失败测试用例错误输出的切片并集.最后,算法 1 返回可疑上下文片段 Context.

- 步骤 2:计算可疑片段中每个元素的可疑值.

如算法 2 所示,第 2 行和第 3 行显示输入变量,$S$($S=\{s_1,s_2,...,s_n\}$)表示由算法 1 生成的可疑上下文片段.$M$ 是可用于 SFL 公式计算的输入矩阵,结构如图 1 所示.第 4 行是输出,并且包含由标记着可疑值的增强型可疑上下文.在第 6 行中,增强的上下文初始化为空.第 7 行搜索上下文 $S$ 中的每个元素.在第 8 行和第 9 行中,函数 *SearchElementInfo*($s_i$,$M$)分析测试用例 $T$ 的语句覆盖信息和测试结果,函数 *GetSuspiciousness*($s_i$,*ElementInfo*($s_i$))

采用 SFL 公式计算 $S$ 中每个语句的可疑值.语句覆盖信息的格式和测试用例的测试结果是一个矩阵,如图 1 所示.由于一些语句的可疑值可能相同,所以 *Order*(*EnhancedContext*)函数按照它们的降序对语句进行排序.在本研究中,SFL 以源代码中的行号降序排列具有相同可疑值的语句.

- 步骤 3:输出错误定位结果.

此步骤将定制本地化结果输出.该定位包含一个增强的上下文,其语句以 SFL 给出的可疑值降序排列.定位结果可以帮助开发人员了解和定位错误,将一些信息附加到每个语句上,如可疑值和行号.

### 2.2 例子演示

如图 3 所示,本节通过包含错误语句 $s_3$ 的程序 $P$,以此说明如何应用 Context-FL.这个例子选择 GP02 来计算 16 个语句的可疑值.每个语句之下的单元格表示该语句是否被测试用例执行(1 表示执行,0 表示未执行),而最右侧的单元格表示测试用例的执行是否失败(1 表示失败,0 为通过).由 GP02 根据覆盖信息和测试用例结果得出排序:{$s_7$,$s_8$,$s_9$,$s_{12}$,$s_{14}$,$s_{10}$,$s_{11}$,$s_2$,$s_3$,$s_1$,$s_{13}$,$s_4$,$s_6$,$s_5$,$s_{15}$,$s_{16}$}.这个集合不能表示可疑语句之间的关系,也不能区分影响错误输出的语句.例如,$s_3$ 和 $s_8$ 都被全部失败测试用例所执行,$s_3$ 是错误的语句而 $s_8$ 不是.然而与 $s_8$ 相比,$s_3$ 被更多地通过测试用例执行.因此,$s_3$ 的可疑值要比 $s_8$ 低.但事实是,$s_8$ 并不在导致错误输出的执行序列当中.这就解释为什么 $s_3$ 排名第 9,并且可疑值比一些与错误输出无关语句的可疑值低.

<table>
<tr><td colspan="11">Program P</td><td colspan="8">Dynamic slice(DS)</td></tr>
<tr><td colspan="11">S1:Read(v1,v2,v3)<br>S2:d1=0,d2=0,d3=0;<br>S3:if(v2＜0){<br>S4:d1 = v2;<br>S5:d2 = v3;<br>S6:d3 = v1;}<br>S7:else {d1 =v2+1;<br>S8: d2 = v3+1; S15:else {output(d2);<br>S9:if(v1＜0){ S16:output(d3);}<br>S10:v1 = v1+v3;}<br>S11: else v1 = v1+v2;<br>S12: d3 = v1+1;}<br>S13:if(v3>0){<br>S14:output(d1);}<br>S3 is faulty.<br>Correct form:<br>If(v2<6){</td><td colspan="8">Input: v1=-1,v2=5,v3=3<br>Ouput:d1=6<br>Slicing criterion:<br>(t1,14,d1)<br>Slice result:<br>{S1,S3,S7,S14}</td></tr>
<tr><td>test</td><td>V1,v2,v3</td><td>S1</td><td>S2</td><td>S3</td><td>S4</td><td>S5</td><td>S6</td><td>S7</td><td>S8</td><td>S9</td><td>S10</td><td>S11</td><td>S12</td><td>S13</td><td>S14</td><td>S15</td><td>S16</td><td>result</td></tr>
<tr><td>t1</td><td>-1,5,3</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>t2</td><td>-2,-7,5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>t3</td><td>5,-6,-8</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>t4</td><td>-5,8,-8</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>t5</td><td>4,7,11</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>t6</td><td>4,2,1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td colspan="2">suspiciousness</td><td>6.00</td><td>6.00</td><td>6.00</td><td>4.24</td><td>4.24</td><td>4.24</td><td>8.24</td><td>8.24</td><td>8.24</td><td>6.46</td><td>6.46</td><td>8.24</td><td>6.00</td><td>8.24</td><td>4.24</td><td>4.24</td><td rowspan="2">Computed by GP02</td></tr>
<tr><td colspan="2">Rank by sfl</td><td>10</td><td>8</td><td>9</td><td>12</td><td>14</td><td>13</td><td>1</td><td>2</td><td>3</td><td>6</td><td>7</td><td>4</td><td>11</td><td>5</td><td>15</td><td>16</td></tr>
<tr><td colspan="2">Rank by DS</td><td>4</td><td></td><td>3</td><td></td><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td>2</td><td></td><td></td><td></td></tr>
</table>

Fig.3 An example illustrating Context-FL

图 3 Context-FL 例子说明

为了解决这个问题,Context-FL 通过使用切片标准($t_1$,$s_{14}$,$d_1$)来构建错误输出的动态切片.该切片包括与失败测试用例 $t_1$ 的输出有直接关系的语句{$s_1$,$s_3$,$s_7$,$s_{14}$}.最后,GP02 生成新的排名列表:{$s_7$,$s_{14}$,$s_3$,$s_1$},其中,$s_3$ 被排在第 3 且{$s_8$,$s_9$,$s_{12}$,$s_{10}$,$s_{11}$,$s_2$,$s_{13}$,$s_4$,$s_6$,$s_5$,$s_{15}$,$s_{16}$}不包括在上下文中,因为它们与失败测试用例 $t_1$ 和 $t_6$ 的错误输出没有关系.因此,Context-FL 比原来的 GP02 更有效.

## 3 CONTEXT-FL 实现框架

### 3.1 实现框架

本节将详细介绍错误定位方法 Context-FL 的实现框架.如图 4 所示,Context-FL 主要架构由 3 个主要功能模块组成,分别是程序信息提取模块、可疑上下文生成模块、可疑计算模块.

程序信息提取模块的主要功能是在执行测试用例后提取程序覆盖信息.第 1 个不可或缺的步骤是程序插桩,其目的是捕获程序的运行信息.插桩点的选择由错误定位方法确定.因为收集足够的和非冗余的信息非常重要,所以需要采用适当的插桩技术,以此控制插桩的代码量,并且不会产生大量的冗余信息引起干扰.第 2 步是收

集数据.因为在插桩之后,程序会产生大量的运行信息,必须以适当的方式收集数据进行抽象和恢复,以便于下一步的分析和计算工作.由于数据量相对较大,需要对数据进行简化.接下来是对每个测试用例生成的数据进行分类,使得所需数据以统一的方式存储.最后生成SFL计算的矩阵信息文件.由于软件的复杂性和规模日益增加,程序的运行数据显著增加.虽然采取了各种措施来压缩或减少在收集阶段收集的数据量,但需要分析的数据量仍然很大.因此,有效的数据分析方法是绝对必要的.程序切片信息、状态信息和统计信息是分析的 3 个主要数据表现形式.Context-FL 使用统计信息来查找与错误相关联的位置,这种类型的信息是通过分析程序的统计数据和提取特定模型来生成的.该方法通常使用语句、分支、预测信息、基本块和功能等程序覆盖信息,比较失败测试用例和成功测试用例并分析其差异性.最后,在不同要素之间的联系的基础上,产生了可疑值.

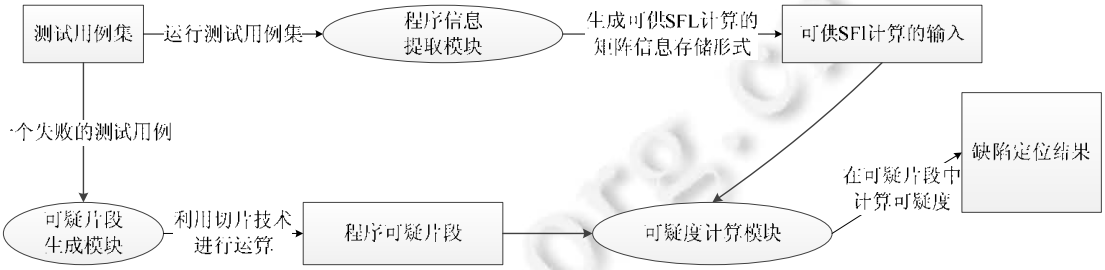



Fig.4 Architecture of Context-FL

图 4 Context-FL 框架结构

可疑上下文生成模块的主要功能是找到与错误语句相关联的一组语句片段,该片段包含了错误语句且又大幅小于源代码量.其目的是通过缩小搜索范围来提高错误定位的效率.程序切片技术是一种分解技术,根据具体计算提取相关语句.它可以分析元素之间的依赖关系并缩小搜索范围.与其他可疑上下文提取技术相比,程序切片提供了一种有效的方法来了解程序执行是否影响输出.根据第 1.2 节切片技术的分析,本框架采用动态切片技术提取可疑片段.虽然切片技术在语句关联分析中具有很强的能力,但是由于程序切片生成的可疑上下文中的语句不能区分,因此这些语句的可疑值是相同的.由于目前软件的规模不断扩大,所以片段的规模将会大为增加.如果只从片段中找出错误的位置,效率会很低,效果也会大为降低.可疑计算模块的主要功能是根据可疑环境,按照可疑值的大小顺序对片段中的语句进行排序.

受SFL技术的启发,当语句的执行影响了失败的测试用例的输出时,语句的可疑值会增加.另一方面,如果程序语句的执行影响成功测试用例的输出,则其可疑值会降低.因此,可疑计算模块定义了新的统计变量.

### 3.2 Context-FL使用的工具

首先介绍 EMMA[21]和 JSlice[22].EMMA 是用于测量和报告 Java 代码覆盖率的开源工具包.它是 100%纯 Java 且没有依赖外部库,可以在任何 Java 2 JVM 中工作.它支持 4 种不同的覆盖类型:类、方法、行和基本块.此外,它可以检测单个源代码行何时被部分覆盖.EMMA 的输出报告类型包括纯文本、HTML 和 XML.特别地,HTML 报告支持源代码链接.本文 Context-FL 主要使用行覆盖类型和 HTML 输出类型,如图 5 所示.

如图 5 所示,88 行的深色表示在执行测试用例时没有执行该行,83 行和 92 行的浅色表示该行只有一部分被执行,其余标深色的行完全被执行.本文过滤覆盖信息并摆脱其他不相关的信息.由于大型程序的文件数量众多,因此需要根据测试用例的顺序对覆盖信息文件进行重新编号、分类和存储.在运行所有测试用例之后,覆盖信息文件以矩阵的形式进行存储,如图 1 所示.

JSlice 是 Java 程序的动态切片工具.给定一个程序 $P$,切片工具执行一个切片标准$(I,L,V)$,其中,$I$ 是一个输入,$L$ 表示在程序 $P$ 执行期间根据输入 $I$ 执行的一些语句的集合,$V$ 是由 $L$ 引用的一组变量[22,23].由切片工具生成切片的目的是:在执行 $P$ 期间找到影响语句 $L$ 中的变量 $V$ 的一组语句的集合,这些语句与 $L$ 及其变量 $V$ 是动态相关的.首先,将 Java 程序的执行路径转换为与执行跟踪相关联的字节流,并以简洁的形式进行压缩.Jslice 使用的

动态切片算法是直接在压缩程序跟踪上运行的算法,可以找到被忽略的错误语句.一般动态切片算法只给出 1 组影响语句 $L$ 中变量执行的语句,而 Jslice 除了发现通用算法发现的语句之外,还可能会发现一些影响语句 $L$ 中的变量 $V$ 的隐藏语句.本文在 Context-FL 中使用的 JSlice 的版本是 JSlice v1.0,支持本文实验的程序都是顺序程序.JSlice 可以使用的是 Fedora Core 3/4 或类似的系统.下一节的实验是在 Fedora Core 3 上运行的.

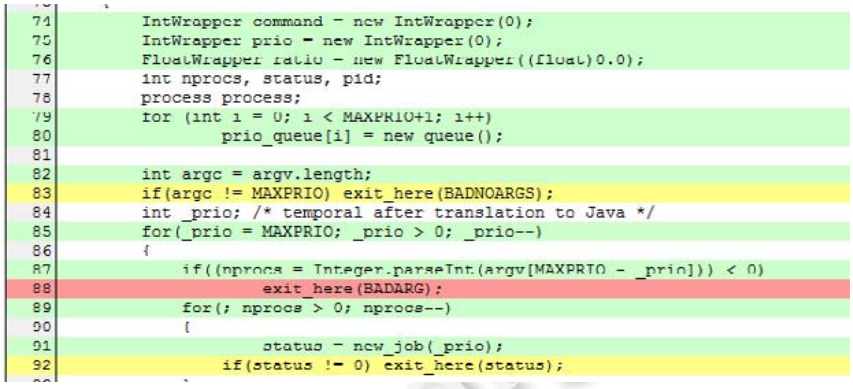



Fig.5 EMMA coverage report

图 5 EMMA 覆盖报告

## 4 实 验

### 4.1 实验设计

实验将表 2 中的 8 种错误定位方法与 Context-FL 对比.对比基准程序集为 14 组 Java 程序.表 3 描述了这些实验程序,依次为简要描述(第 2 列)、实验使用的错误版本数量(第 3 列)、语句行数(第 4 列)和测试用例数(第 5 列).这些从 SIR[24]和 Defect4J[25]获取的程序被广泛使用.

**Table 3** Subject programs

**表 3** 实验对象

| Program | Description | Versions | LOC | Test |
|---|---|---|---|---|
| Print tokens1 | lexical analyzer | 5 | 587 | 4 071 |
| Print tokens2 | lexical analyzer | 10 | 571 | 4 056 |
| Schedule1 | priority scheduler | 8 | 422 | 2 650 |
| Schedule2 | priority scheduler | 4 | 383 | 2 710 |
| Tot info | Information measure | 17 | 411 | 1 052 |
| Jtcas | collision avoidance | 14 | 198 | 1 608 |
| NanoXML_v1 | XML parser | 7 | 5 369 | 206 |
| NanoXML_v2 | XML parser | 7 | 5 650 | 206 |
| NanoXML_v3 | XML parser | 10 | 8 392 | 206 |
| NanoXML_v5 | XML parser | 7 | 8 795 | 206 |
| chart | JFreeChart | 10 | 96 000 | 2 205 |
| math | Apache commons math | 18 | 85 000 | 3 602 |
| mockito | Framework for unit tests | 9 | 6 000 | 1 075 |
| time | Joda-Time | 3 | 53 000 | 4 130 |

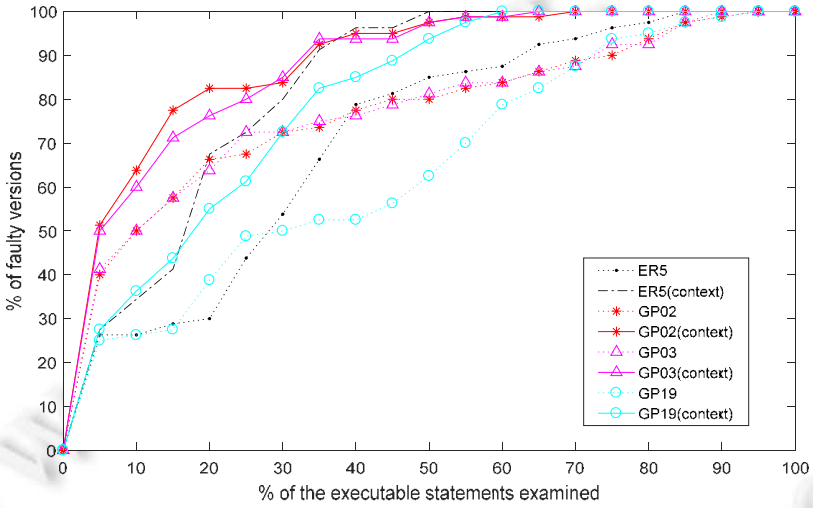
表中前 10 个程序[24]原本包含 102 个错误版本.实验没有采用 schedule 1 的版本 1、schedule 2 的版本 2、版本 4、版本 9、版本 10、版本 12、版本 14 和 Tot_info 的版本 4、版本 10、版本 12、版本 14.因为它们没有失败的测试用例,至少有 1 个失败的测试用例是执行动态切片所必需的.实验还排除了执行动态切片失败的 Tot_ info 的版本 11,以及错误语句在依赖库中的版本 19.对于 Jtcas,实验使用了 14 个代表性的错误.后 3 个程序选自 Defect4J[25],Defect4J 是一个真实错误的数据集.在 Defect4J 中,实验选取了 chart 的版本 1、版本 3、版本 6、版本 7、版本 12、版本 13、版本 17、版本 20、版本 23 和版本 24,mockito 的版本 1、版本 5、版本 7、版本 8、

版本 26、版本 28、版本 29、版本 34 和版本 38,time 的版本 4、版本 16 和版本 19,math 的 35 个单错误版本的 18 个.其他错误版本没有被采用的原因:一是本文研究主要是针对单错误,而其他版本为多错误;二是没有编译运行成功.因此如表 3 所示,实验最终采用了 129 个错误版本.

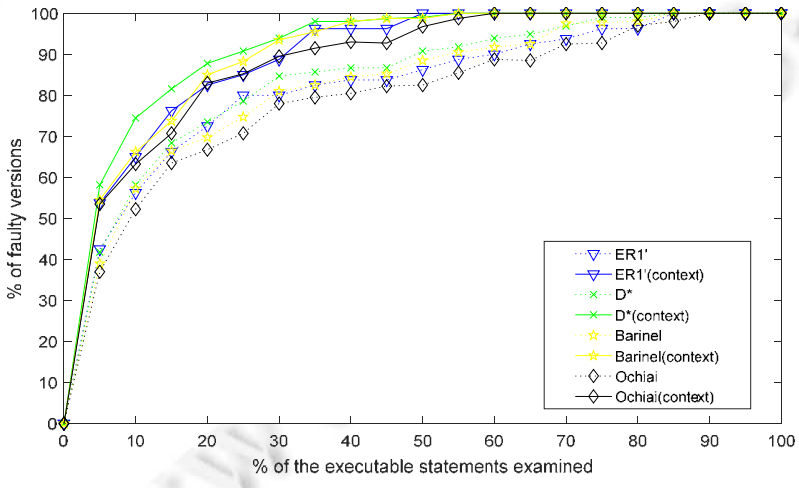
### 4.2 结果分析

为了评估本文提出的错误定位方法的有效性,实验采用错误定位精度(称为 Acc)和相对改进值(称为 RImp)作为评价标准[26].Acc 被定义为在找到真正的错误语句之前要检查的可执行语句的百分比.RImp 是使用 Context-FL 找到所有错误需要检查的语句总数除以使用对比方法需要检查的语句总数.Rimp 的值越低,代表定位效果越好[27].

由于对比方法较多,图 6 将以分组的方式来更好地显示 Context-FL 和对比方法的 Acc 值.对于每个子图——图 6(a)和图 6(b),横坐标表示在所有程序中检查的可执行语句的百分比.纵坐标表示定位方法找出错误的比例.图 6 中的每一个点表示在检查可执行语句的百分比时,定位到的错误的百分比.




(a)



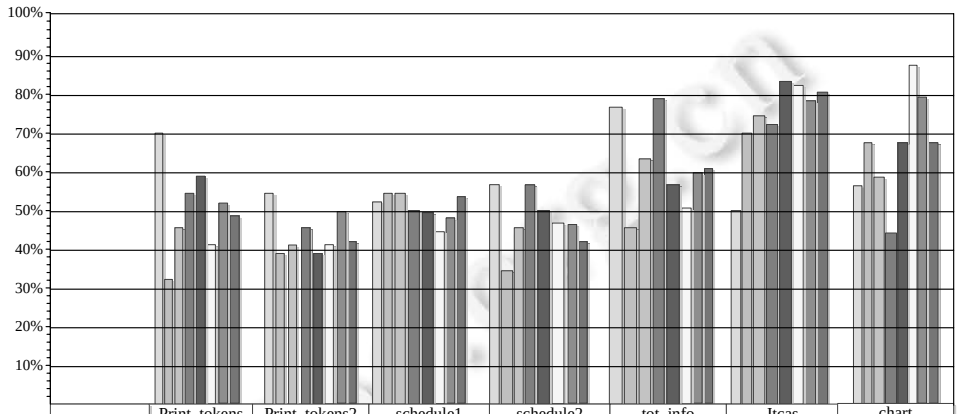

(b)

Fig.6 Acc comparison between SFL and Context-FL

图 6 SFL 和 Context-FL 的 Acc 对比

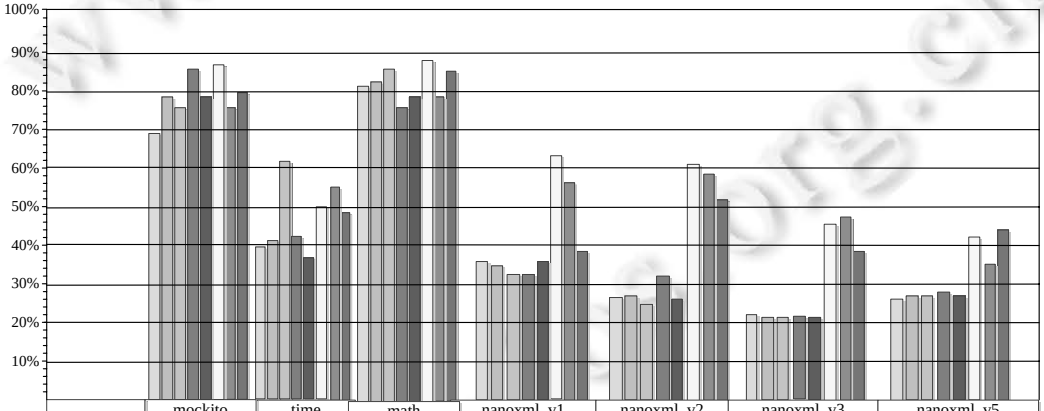
从图 6(a)和图 6(b)可以看出,除了 ER5 和 GP19 之外,检查可执行语句到 5%时则出现了明显的效果提升.即使是 ER5 和 GP19,在横坐标百分比分别为 15%和 10%时也出现了明显的效果提升.因此,Context-FL 方法提高了 8 种定位方法的有效性.

由于对比方法较多,图 7 也分为图 7(a)和图 7(b)两个子图,显示了 Context-FL 在每个程序上的 RImp 值.采用 Context-FL 方法后,需要检查的语句数明显减少,从 Nanoxml_v3 的 GP03 的 20.22%(最优值)到 chart 的 D*的 88.63%(最差值).这意味着 Context-FL 只需要检查 SFL 所需检查的执行语句数的 20.22%~88.63%,就能定位到错误.从图 7(a)和图 7(b)可以发现,Context-FL 更加有效和稳定,特别是当程序规模大时效果则更为显著.




| | Print_tokens | Print_tokens2 | schedule1 | schedule2 | tot_info | Jtcas | chart |
|---|---|---|---|---|---|---|---|
| ER5 | 69.59% | 53.68% | 52.89% | 55.85% | 76.03% | 51.95% | 55.62% |
| GP02 | 33.21% | 37.88% | 54.82% | 32.95% | 47.20% | 70.28% | 68.97% |
| GP03 | 46.40% | 40.35% | 53.89% | 45.28% | 63.07% | 73.84% | 59.68% |
| GP19 | 54.32% | 44.94% | 50.32% | 56.42% | 77.49% | 72.81% | 44.76% |
| ER1' | 58.16% | 40.36% | 50.00% | 48.89% | 57.04% | 83.44% | 67.34% |
| D* | 41.31% | 41.67% | 44.60% | 46.93% | 50.10% | 82.58% | 88.63% |
| Ochiai | 52.36% | 50.21% | 48.15% | 47.25% | 59.01% | 79.34% | 79.26% |
| Barinel | 48.31% | 42.67% | 54.20% | 42.13% | 60.84% | 81.25% | 67.73% |

(a)



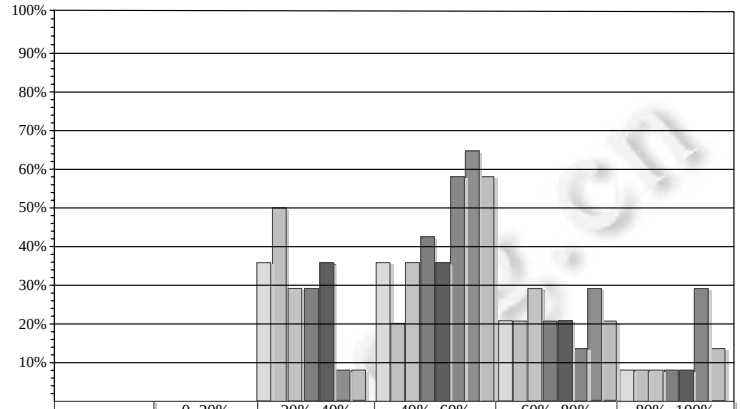

| | mockito | time | math | nanoxml_v1 | nanoxml_v2 | nanoxml_v3 | nanoxml_v5 |
|---|---|---|---|---|---|---|---|
| ER5 | 69.87% | 39.38% | 81.89% | 36.36% | 28.96% | 22.56% | 26.67% |
| GP02 | 78.35% | 42.71% | 82.85% | 35.63% | 28.06% | 20.57% | 27.39% |
| GP03 | 75.58% | 62.89% | 85.43% | 32.93% | 27.97% | 20.22% | 27.39% |
| GP19 | 85.59% | 43.81% | 75.56% | 32.75% | 31.96% | 20.81% | 27.59% |
| ER1' | 68.57% | 36.65% | 78.85% | 36.25% | 27.97% | 20.57% | 27.39% |
| D* | 86.90% | 50.50% | 87.82% | 63.64% | 60.87% | 44.53% | 42.11% |
| Ochiai | 74.43% | 56.79% | 79.37% | 56.75% | 58.37% | 46.87% | 35.67% |
| Barinel | 79.57% | 48.41% | 85.56% | 48.24% | 52.25% | 38.33% | 44.58% |

(b)

Fig.7 RImp of our approach

图 7 本文方法的 RImp

为了对改进效果进行更详细的描述,图 8 显示了 RImp 的分布,分别为 0~20%,20%~40%,40%~60%,60%~80%和 80%~100%.每个间隔代表分布在此区间的 RImp 值的百分比.例如,GP02 在 20%~40%区间的值为 50.00%.这意味着在 50%的错误版本中,使用本文的方法定位到错误,它所需检查的语句数为使用 GP02 所需检查的语句数的 20%~40%.从图 8 可以看出,RImp 值主要分布在 40%和 60%之间,其次是 20%~40%.这表明,本文的方法对于提高定位效果还是非常明显的.

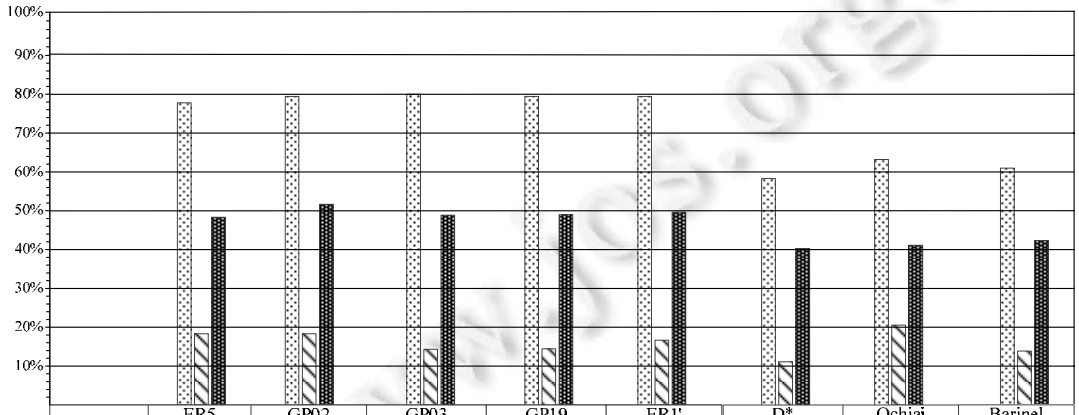



| | 0~20% | 20%~40% | 40%~60% | 60%~80% | 80%~100% |
|---|---|---|---|---|---|
| ER5 | 0 | 35.71% | 35.71% | 21.43% | 7.14% |
| GP02 | 0 | 50.00% | 21.43% | 21.43% | 7.14% |
| GP03 | 0 | 28.57% | 35.71% | 28.57% | 7.14% |
| GP19 | 0 | 28.57% | 42.86% | 21.43% | 7.14% |
| ER1' | 0 | 35.71% | 35.71% | 21.43% | 7.14% |
| D* | 0 | 0 | 57.14% | 14.29% | 28.57% |
| Ochiai | 0 | 7.14% | 64.29% | 28.57% | 0 |
| Barinel | 0 | 7.14% | 57.14% | 21.42% | 14.29% |

Fig.8 RImp distribution of our approach

图 8 本文方法的 RImp 分布

如图 9 所示,最大节约量(*Saving*=100%−*RImp*)为 GP03 的 79.78%,最低节约量为 D*的 11.37%,平均节约范围为 40.56%~52.79%.这意味着在使用 Context-FL 检查语句的数量时,可以减少 11.37%~79.78%.总而言之,与实验中的 8 种定位方法相比,Context-FL 明显减少了语句的检查数量.因此,Context-FL 定位能力更强.




| | ER5 | GP02 | GP03 | GP19 | ER1' | D* | Ochiai | Barinel |
|---|---|---|---|---|---|---|---|---|
| Max Saving | 77.44% | 79.43% | 79.78% | 79.19% | 79.43% | 58.69% | 64.33% | 61.67% |
| Min Saving | 18.11% | 17.15% | 14.57% | 14.41% | 16.56% | 11.37% | 20.63% | 14.44% |
| Avg Saving | 48.48% | 52.79% | 48.93% | 48.64% | 49.89% | 40.56% | 41.15% | 43.28% |

Fig.9 Saving of our approach over SFL

图 9 本文方法和 SFL 的 Saving 对比

Lei 等人[27,28]以静态反向切片和执行切片的交集构建信息矩阵,并基于该矩阵提出重新定义语句怀疑度计算公式的方法(本文简称 Lei-FL).为了进一步说明 Context-FL 的有效性,实验将 Context-FL 和 Lei-FL 进行了比较.Lei-FL 使用的程序集为 C 程序,与实验所使用 JAVA 版本的 Print tokens 1、Print tokens 2、Schedule 1、Schedule 2、Tot info 和 Jtcas 为一组程序,错误版本也相同.因此,实验选取在这些程序上效力较好的 5 种公式(即 ER5, Barinel,GP02,GP19 和 D*),分别应用 Context-FL 和 Lei-FL,并对比它们的效力.如图 10 所示,在所有 5 种公式上,Context-FL 曲线一直高于 Lei-FL 曲线.这说明 Context-FL 的定位效力高于 Lei-FL.

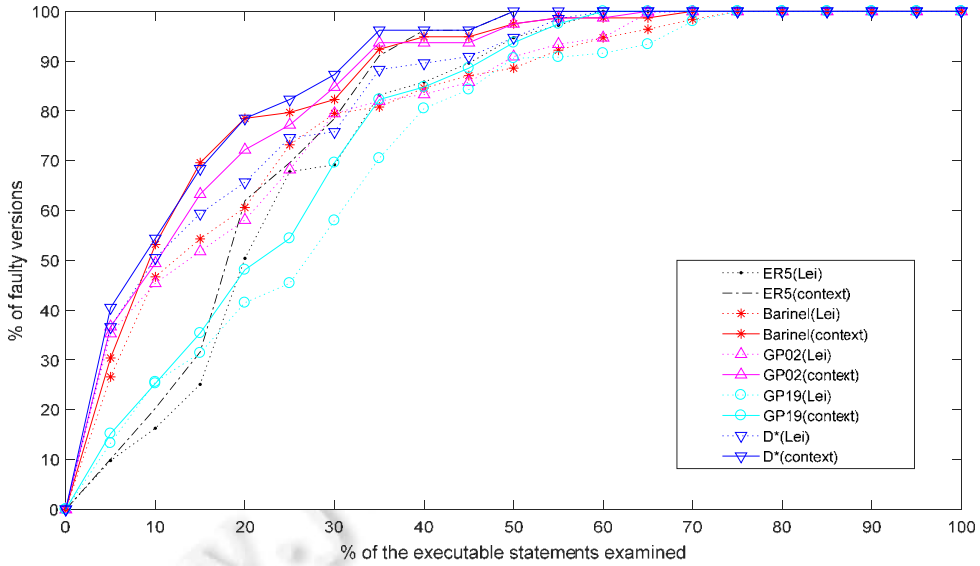



Fig.10 Acc comparison between Context-FL and Lei-FL

图 10 Context-FL 和 Lei-FL 的 Acc 对比

表 4 描述了这些实验程序的时间开销的对比情况.表中数据的分子为使用 Context-FL 时的时间开销情况,分母为未使用 Context-FL 的时间开销情况.可以看出,随着程序规模的扩大,由于采用动态切片的原因,使用 Context-FL 的时间成本会有所增加.

**Table 4** Comparison of experimental time cost (s)

**表 4** 实验时间开销对比 (秒)

| Program | ER5 (context)/ ER5 | GP02 (context)/ GP02 | GP03 (context)/ GP03 | GP19 (context)/ GP19 | ER1′ (context)/ ER1’ | D* (context)/ D* | Ochiai (context)/ Ochiai | Barinel (context)/ Barinel |
|---|---|---|---|---|---|---|---|---|
| Print tokens 1 | 0.872/0.201 | 0.889/0.218 | 0.87/0.199 | 0.912/0241 | 0.876/0.205 | 0.819/0.220 | 0.875/0.214 | 0.883/0.197 |
| Print tokens 2 | 0.37/0.216 | 0.367/0.213 | 0.374/0.220 | 0.372/0.218 | 0.388/0.234 | 0.387/0.233 | 0.377/0.215 | 0.397/0.221 |
| Schedule 1 | 0.353/0.180 | 0.266/0.093 | 0.267/0.094 | 0.271/0.098 | 0.285/0.112 | 0.276/0.103 | 0.279/0.091 | 0.259/0.099 |
| Schedule 2 | 0.394/0.191 | 0.302/0.099 | 0.298/0.095 | 0.297/0.094 | 0.299/0.096 | 0.296/0.093 | 0.315/0.112 | 0.305/0.097 |
| Tot info | 0.338/0.106 | 0.28/0.048 | 0.27/0.038 | 0.273/0.041 | 0.282/0.050 | 0.271/0.039 | 0.291/0.045 | 0.287/0.047 |
| Jtcas | 0.223/0.099 | 0.158/0.034 | 0.155/0.031 | 0.161/0.037 | 0.157/0.033 | 0.159/0.035 | 0.159/0.058 | 0.214/0.032 |
| NanoXML_v1 | 8.989/1.531 | 8.755/1.297 | 9.184/1.726 | 8.788/1.33 | 8.848/1.39 | 8.757/1.299 | 8.757/1.167 | 8.622/1.35 |
| NanoXML_v2 | 13.14/1.524 | 12.968/1.352 | 13.8/2.184 | 12.956/1.364 | 12.951/1.335 | 12.936/1.32 | 13.16/1.55 | 12.873/1.79 |
| NanoXML_v3 | 32.79/3.28 | 32.03/2.52 | 32.63/3.12 | 32.335/2.825 | 32.19/2.684 | 32.17/2.66 | 31.89/2.884 | 32.57/2.712 |
| NanoXML_v5 | 56.11/3.216 | 55.661/2.765 | 55.504/2.608 | 55.871/2.975 | 55.81/2.912 | 55.62/2.73 | 55.314/2.798 | 52.31/2.508 |
| Chart | 46.21/3.8 | 45.761/2.854 | 45.504/2.814 | 45.87/3.066 | 45.808/3.04 | 46.217/2.933 | 44.72/2.919 | 46.36/3.18 |
| Math | 21.36/1.79 | 20.78/1.92 | 19.25/1.88 | 18.89/1.85 | 22.27/2.07 | 20.75/2.01 | 19.57/1.99 | 20.33/1.82 |
| Mockito | 25.23/2.123 | 24.134/1.841 | 24.256/2.86 | 23.335/2.468 | 26.85/1.756 | 24.88/2.35 | 24.71/2.276 | 23.19/2.298 |
| Time | 60.89/5.21 | 67.45/4.45 | 62.36/5.26 | 69.89/4.33 | 61.39/5.39 | 61.48/4.59 | 60.87/4.68 | 61.34/5/145 |

### 4.3 有效性威胁

本文实验有如下有效性威胁.

- 实验采用了 JSlice 的动态切片工具.然而,由于切片技术与 JSlice 工具局限性,与系统库相关的依赖无法提取.这时,错误语句很有可能不包含在切片结果中,如 Tot_info 的版本 11.动态切片技术的另一个缺点是对某些类型的错误无法实施切片,例如,遗漏类型的错误语句无法产生执行信息,动态切片无法切到该类型的错误语句.这些缺点是由动态切片技术自身特点所引起的.
- 实验有一个潜在假设:当一个失败的测试用例运行时,错误的语句应该被执行.这个假设通常成立,实验结果也证实了假设的合理性.然而在某些情况下,如程序中出现多重错误时,则假设可能不成立.但是对于单一错误进行研究是必要的,因为它们是对多个错误进行定位的研究基础.这就是为什么大多数现有的研究集中在单一错误情景的原因.
- 实验对象也是有效性威胁之一.实验选择广泛应用于错误定位领域的代表性程序.然而现实调试中存在许多未知因素,说明它们不能覆盖并适用于现实中的所有情况.因此,未来工作使用更多的真实大型程序将会是本文研究的重点.

## 5 相关工作

有许多研究人员根据覆盖信息或切片信息研究错误定位技术.本节简要介绍这些研究.更多的错误定位工作可参考 Wong 等人近期发表的综述[1].

基于覆盖的错误定位技术将程序谱数据从测试执行转换为程序实体的可疑值,并使用诸如基于程序谱的错误定位(SFL)等统计公式对其进行排序.当使用这些技术时,本文不需要知道程序的详细信息,只需运行通过和失败的测试用例.Chen 等人[29]提出了 Jaccard 技术,这是一种统计错误定位算法.Jones 等人[30]提出了 Tarantula 技术,计算每个语句的可疑值,并根据他们的可疑值进行排名.Tarantula 在后续研究中是广泛使用和比较的技术.Abreu 等人[31]应用 Ochiai 定位单一错误.他们指出:他们的技术 Ochiai 一直优于 Tarantula,并且在 EXAM 得分方面平均提高了 5%.Abreu 等人[32]在后来的研究中评估了 Tarantula 的有效性以及其他技术,并指出,Ochiai 对测试用例进行了最佳的评估.Wong 等人[33]利用数据和控制流程,并提出了几个指标,如 Wong1-3,Wong3′.它们都是计算具有失败测试用例的程序实体的可疑值的统计公式.虽然 Wong 等人[8]提出了一种基于代码覆盖的方法,可以通过测试执行自动调整可疑语句的权重,还提供了一些减少搜索域的方法.Wong 等人[34,35]还提出了一种基于交叉表的方法,利用语句的覆盖和执行信息,并提出了一种名为 DStar(D*)的技术.该方法计算了修改语句的可疑值.最近,Xie 等人[14]在理论上研究了 GP 进化公式.

程序切片算法(静态和动态)的应用也被许多研究人员广泛研究用于程序调试.Lei 等人[27,28]采用静态切片对每个测试用例的输出进行切片,以此构建输入矩阵,并基于这个矩阵重新定义可疑值计算方法.本文方法仅对失败测试用例的输出进行动态切片,其针对性强,避免了成功测试用例可能带来的不确定影响.同时,动态切片体积小于静态切片体积,搜索范围更小.本文实验结果也表明,Context-FL 优于 Lei 等人的方法.Zhang 等人[36,37]研究了动态切片在定位错误中的有效性,并开发了一种采用动态切片的策略,通过计算从 0 到 1 的置信度值,来识别可能影响输出以产生不正确值的语句的子集.Zhang 等人[38]还提出了一种类似切片的技术,以便从许多事件中剪除不相关的事件.Jones 等人[30]使用动态切片和执行切片来减少搜索域.Alves 等人[39]将变化影响分析纳入动态切片,以进一步优化动态切片.Wen 等人[40]提出了利用混合切片谱的统计方法,以提高错误定位的有效性.与这些方法不同,本文方法使用的是动态切片,对程序的动态执行轨迹进行了分析,显示了如何捕获动态数据和控制依赖关系.Wong 等人[41]提出了一种基于执行切片和块之间的数据依赖关系,在小范围代码内实现高效准确的错误定位的方法.我们的方法与它不同.我们利用动态切片来构建语句的上下文关系.

偶然正确性(coincidental correctness)问题是错误定位研究的一个热点.当测试用例执行了错误语句,却并没有导致程序失效时,就产生了偶然正确性问题.偶然正确性问题存在于成功测试用例中,对错误定位性能有负效应.为了解决偶然正确性问题,如何识别存在偶然正确性问题的成功测试用例成是关键.Wang 等人[42]提出了上

下文模型(context pattern),通过是否匹配上下文模型来识别存在偶然正确性问题的成功测试用例,并以此优化覆盖矩阵,从而提升错误定位性能.Masri 等人[43]定义了基于缺陷的失效,以此描述具有偶然正确性问题的成功测试用例,从而更有利于缓解偶然正确性问题.随后,Masri 等人[44]对偶然正确性问题进一步研究,采用 Euclidean 标准度量出成功测试用例与失败测试用例之间的相似度,以此移除具有偶然正确性问题的成功测试用例.Miao 等人[45]基于偶然正确性的成功测试用例与失败测试用例有相似行为的思想,采用聚类方法对测试用例进行分类.当成功测试用例被划分到包含失败测试用例的类别时,该成功测试用例存在偶然正确性问题的可能性较大.Bandyopadhyay[46]基于成功测试用例与失败测试用例之间执行语句的相似度来定义权重,并以此预测可能具有偶然正确性问题的成功测试用例,最后,根据预测来优化错误定位性能.Lei 等人[47]系统分析和总结了测试用例集的错误定位效能,从理论和实验证明了偶然正确性问题是成功测试用例对错误定位效能影响最大的因素.他们的研究还明确失败测试用例对错误定位效能具有正效应作用.与成功测试用例的偶然正确性问题相比,失败测试用例的信息更加直接有效.因此,本文方法关注于失败测试用例,通过动态切片提取更精确信息,大幅度缩小错误搜索范围,以此为基础来实现错误定位性能提升,也印证了 Lei 等人[47]对失败测试用例的结论.虽然本文方法不关注优化具有偶然正确性问题的成功测试用例,且采用的动态切片不能根除偶然正确性,但是通过动态切片对失败测试用例的优化,大幅度缩小错误搜索范围,抑制了具有偶然正确性问题的成功测试用例发挥负效应的空间,间接地缓解了偶然正确性问题.

## 6 结 论

本文提出了一种基于上下文的错误定位方法 Context-FL.该方法结合动态切片和可疑性度量方法,构建可疑值标记的上下文.实验结果表明,与 5 种典型定位方法对比,Context-FL 显著优于这些方法,有效缩减了代码检查的范围,大幅度提升了错误定位性能.未来的工作包括方法优化,以提高其准确性.此外,我们还会研究将 Context-FL 扩展到多错误场景下的定位.

**References**:


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

[2] Xie X, Kuo FC, Chen TY, Yoo S, Harman M. Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In: Proc. of the 5th Symp. on Search-based Software Engineering. St. Petersburg: Springer-Verlag, 2013. 224−238.

[3] Acharya M, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems. In: Proc. of the Int'l Conf. on Software Engineering. 2012. 746−765.

[4] Pearson S, Campos J, Just R, *et al*. Evaluating and improving fault localization. In: Proc. of the Int'l Conf. on Software Engineering. Buenos Aires: IEEE, 2017. 609−620.

[5] Naish L, Lee HJ, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans. on Software Engineering and Methodology, 2011,20(3):1−32.

[6] Yoo S. Evolving human competitive spectra-based fault localisation techniques. Int'l Conf. on Search Based Software Engineering, 2012,7515:244−258.

[7] Abreu R, Zoeteweij P, van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Proc. of the Testing: Academic and Industrial Conf. on Practice and Research Techniques (MUTATION). Windsor: IEEE, 2007. 89−98.

[8] Wong WE, Debroy V, Choi B. A family of code coverage-based heuristics for effective fault localization. Journal of Systems and Software, 2010,83(2):188−208.

[9] Parnin C, Orso A. Are automated debugging techniques actually helping programmers. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM Press, 2011. 199−209.

[10] Zhang Z, Mao XG, Lei Y, Zhang P. Enriching contextual information for fault localization. IEICE Trans. on Information and Systems, 2014,E97.D(6):1652−1655.

[11] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984,SE-10(4):352−357.

[12] Korel B, Laski J. Dynamic program slicing. Information Processing Letters, 1988,29(3):155−163.

[13] Zhang X, Gupta R, Zhang Y. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In: Proc. of the 26th Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 502−511.

[14] 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.

[15] Gallagher K, Lyle J. Using program slicing in software maintenance. IEEE Trans. on Software Engineering, 1991,17(17):751−761.

[16] Gupta R, Harrold M, Soffa M. An approach to regression testing using slicing. In: Proc. of the Conf. on Software Maintenance. Orlando: IEEE, 1992. 299−308.

[17] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Sigplan Conf. on Programming Language Design and Implementation, 1990,12(1):26−60.

[18] Weiser M. Programmers use slices when debugging. Communications of the ACM, 1982,25(7):446−452.

[19] Nanda MG. Slicing concurrent programs. In: Proc. of the ISSTA 2000. 1993. 223−240.

[20] Duesterwald E, Gupta R, Soffa ML. Distributed slicing and partial re-execution for distributed programs. In: Proc. of the Int'l Workshop on Languages and Compilers for Parallel Computing. New Haven: Springer-Verlag, 1992. 329−337.

[21] EMMA. http://emma.sourceforge.net/

[22] Jslice. http://jslice.sourceforge.net/

[23] Wang T, Roychoudhury A. Using compressed bytecode traces for slicing Java programs. In: Proc. of the ACM/IEEE Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 512−521.

[24] SIR. http://sir.unl.edu/portal/index.php

[25] defects4j. http://defects4j.org

[26] Debroy V, Wong WE, Xu X, Choi B. A grouping-based strategy to improve the effectiveness of fault localization techniques. In: Proc. of the 10th Int'l Conf. on Quality Software. Zhangjiajie: IEEE Computer Society, 2010. 13−22.

[27] Lei Y, Mao XG, Dai ZY, Wang CS. Effective statistical fault localization using program slices. In: Proc. of the 36th Annual Int'l Computer Software and Applications Conf. Izmir: IEEE Computer Society, 2012. 1−10.

[28] Lei Y, Mao X, Dai Z, Qi Y, Wang C. Slice-based statistical fault localization. Journal of Systems and Software, 2014,89(1):51−56.

[29] Chen M, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic Internet services. In: Proc. of the Int'l Conf. on Dependable Systems and Networks. Bethesda: IEEE, 2002. 595−604.

[30] Jones JA. Fault localization using visualization of test information. In: Proc. of the 26th Int'l Conf. on Software Engineering. Edinburgh: IEEE, 2004. 54−56.

[31] Abreu R, Zoeteweij P, van Gemund AJC. An evaluation of similarity coeffcients for software fault localization. In: Proc. of the 12th Pacific Rim Int'l Symp. on Dependable Computing. Riverside: IEEE Computer Society, 2006. 39−46.

[32] Abreu R, Zoeteweij P, Golsteijn R, van Gemund AJC. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 2009,82(11):1780−1792.

[33] Wong WE, Qi Y, Zhao L, Cai KY. Effective fault localization using code coverage. In: Proc. of the 31st Annual Int'l Computer Software and Applications Conf. Beijing: IEEE Computer Society, 2007. 449−456.

[34] Wong WE, Wei T, Qi Y, Zhao L. A crosstab-based statistical method for effective fault localization. In: Proc. of the 1st Int'l Conf. on Software Testing, Verification and Validation. Lillehammer: IEEE, 2008. 42−51.

[35] Wong WE, Debroy V, Li YH, Gao RZ. Software fault localization using dstar (*D**). In: Proc. of the 6th IEEE Int'l Conf. on Software Security and Reliability. Gaithersburg: IEEE Computer Society, 2012. 21−30.

[36] Zhang X, Tallam S, Gupta N, Gupta R. Towards locating execution omission errors. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2007,42(6):415−424.

[37] Zhang XY, Gupta N, Gupta R. Pruning dynamic slices with confidence. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2006,41(6):169−180.

[38] Zhang X, Tallam S, Gupta R. Dynamic slicing long running programs through execution fast forwarding. In: Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Portland: DBLP, 2006. 81−91.

[39] Alves EC, Gligoric M, Jagannath V, Damorim M. Fault-localization using dynamic slicing and change impact analysis. In: Proc. of the 26th Int'l Conf. on Automated Software Engineering. Lawrence: IEEE, 2011. 520−523.

[40] Wen W, Li B, Sun X, Li J. Program slicing spectrum-based software fault localization. In: Proc. of the 23rd Int'l Conf. on Software Engineering and Knowledge Engineering. Miami Beach: IEEE Press, 2011. 213−218.

[41] Wong WE, Qi Y. An execution slice and inter-block data dependency-based approach for fault localization. In: Proc. of the Asia-Pacific Software Engineering Conf. Busan: IEEE Computer Society, 2004. 366−373.

[42] Wang X, Cheung SC, Chan WK. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In: Proc. of the Int'l Conf. on Software Engineering. Vancouver: IEEE Computer Society, 2009. 45−55.

[43] Masri W, Assi RA. Cleansing test suites from coincidental correctness to enhance fault-localization. In: Proc. of the Int'l Conf. on Software Testing. Washington: IEEE Computer Society, 2010. 165−174.

[44] Masri W, Assi RA. Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. on Software Engineering and Methodology, 2014,23(1):1−28.

[45] Miao Y, Chen ZY, Li SH, Zhao ZH, Zhou YM. A clustering-based strategy to identify coincidental correctness in fault localization. Int'l Journal of Software Engineering and Knowledge Engineering, 2013,23(5):721−741.

[46] Bandyopadhyay A. Mitigating the effect of coincidental correctness in spectrum based fault localization. In: Proc. of the 5th Int'l Conf. on Software Testing, Verification and Validation. Kolkata: IEEE, 2012. 479−482.

[47] Lei Y, Sun CN, Mao XG, Su ZD. How test suites impact fault localisation starting from the size. IET Software, 2018,12(3): 190−205.

**张卓**(1984－),男,山东蓬莱人,博士生,主要研究领域为程序切片,程序测试,深度学习.

**雷晏**(1985－),男,博士,副教授,CCF 专业会员,主要研究领域为软件错误定位,软件自动修复.

**谭庆平**(1965－),男,博士,教授,博士生导师,主要研究领域为软件工程,软件容错技术,系统软件.

**常曦**(1979－),女,博士,副教授,CCF 专业会员,主要研究领域为程序测试,程序分析.

**毛晓光**(1970－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为可信软件,软件维护与演化.

**薛建新**(1980－),男,博士,讲师,CCF 专业会员,主要研究领域为并发理论,程序分析.