软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(6):2690−2707 [doi: 10.13328/j.cnki.jos.006498]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向众包软件开发的错误定位方法*

李乐平, 张宇霞, 刘 辉

(北京理工大学 计算机学院, 北京 100081)
通信作者: 张宇霞, E-mail: yuxiazh@bit.edu.cn

摘 要: 在软件开发中, 错误定位是修复软件缺陷的必要前提. 为此, 研究者们提出了一系列自动化的错误定位方法. 这些方法利用了测试用例运行时的覆盖路径和运行结果等信息, 大幅减少了定位错误代码的难度. 在竞争性众包软件开发中, 往往存在多个竞争性实现 (解决方案), 提出一种专门面向众包软件工程的错误定位方法. 主要思想是, 在定位错误语句时, 将其多个竞争性实现作为参考程序. 针对程序中的各个语句, 在参考程序中搜索参考语句, 并利用参考语句计算其错误概率. 给定一个错误程序和相应的测试用例, 首先运行测试用例并使用广泛流行的基于频谱的错误定位方法计算其初始错误概率. 然后, 根据此语句与其参考语句的相似性调整错误概率. 在 118 个真实的错误程序上进行实验, 结果表明所提方法相比基于频谱的方法, 定位错误的成本降低了 25% 以上.
关键词: 错误定位; 众包; 可疑度; 竞争性实现; 测试用例
中图法分类号: TP311

中文引用格式: 李乐平, 张宇霞, 刘辉. 面向众包软件开发的错误定位方法. 软件学报, 2023, 34(6): 2690–2707. http://www.jos.org.cn/1000-9825/6498.htm
英文引用格式: Li LP, Zhang YX, Liu H. Crowdsourcing Software Development Oriented Fault Localization. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2690–2707 (in Chinese). http://www.jos.org.cn/1000-9825/6498.htm

Crowdsourcing Software Development Oriented Fault Localization

LI Le-Ping, ZHANG Yu-Xia, LIU Hui
(School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

Abstract: Fault localization is an essential precondition for repairing in software development. To this end, researchers have proposed automated fault localization (AFL) methods to facilitate the task. Such approaches have taken full advantage of information such as the execution tracks and execution results of given test cases and receive significant effectiveness in reducing the difficulty of fault localization. In competitive crowdsourcing software development, one task could receive multiple competitive implementations (solutions). This study proposes a novel approach for AFL in crowdsourcing software engineering. The key insight of the proposed approach is that when locating faulty statements in a program, it regards competitive implementations as reference programs. By searching for reference statements in reference programs for each statement in buggy program, it calculates the suspicious score of the statement by leveraging its references. Given a set of test cases and a buggy program, the test scenario is run and the initial suspicious score for each statement in the buggy program is calculated by wildly used SBFL approach. After that, suspicious score of each statement is adapted according to its similarity with statements in competitive implementations. The proposed approach is evaluated on 118 real word buggy programs that are accompanied with competitive implementations. The evaluation results suggest that compared with SBFL approaches, the cost of fault localization is reduced by more than 25%.
Key words: fault localization; crowdsourcing; suspicious score; competitive implementation; test case

错误定位 (fault localization) 是指在错误的程序中鉴别出错的语句[1]. 在软件开发过程中, 这是一项极其消耗时间和人力成本的工作[2]. 通过定位错误的语句, 程序员能够在不需要人工审查的情况下排除大部分正确的语句,

* 基金项目: 国家自然科学基金 (61690205, 61772071)
收稿时间: 2020-12-07; 修改时间: 2021-03-25; 采用时间: 2021-09-18; jos 在线出版时间: 2022-11-24
CNKI 网络首发时间: 2022-11-25

© 中国科学院软件研究所 http://www.jos.org.cn

进而更好地进行错误代码修复[1]. 因而, 错误定位的准确度对降低修复错误的成本非常重要[3]. 研究者们提出了一系列自动化或半自动化的定位技术[4–7]. 这些技术的分类包括: 基于切片 (code slicing based)[8,9]、基于频谱 (spectrum based)[10–13]、基于统计 (statistics based)[14]、基于程序状态 (program state based)[15]、基于机器学习 (machine learning based)[16]、基于数据挖掘 (data mining based)[17]、以及基于模式 (model based)[18]等. 这些错误定位技术大大降低了发现和修复错误的成本[1].

众包软件工程 (crowdsourced software engineering)[19] 是软件工程中的新兴领域, 是指在公开的平台上发布软件任务[20]. 与传统的软件开发工作由企业自身或外包团队完成不同, 众包软件开发中任务都是由大众成员完成的[20]. 具体而言, 各自独立的开发人员接到企业发出的任务后独自完成软件开发工作, 最终提交的各个解决方案相互竞争, 胜出者得到奖励[21]. 目前, 众包软件开发的模式已在实践中采用[22,23]. 在某些情况下, 胜出的解决方案也可能需要改进. 一方面, 即使在竞争中被认为最优秀的实现依然可能在使用后被发现有错误. 另一方面, 某一个实现虽然有微小的错误, 但是在性能上 (响应时间、内存占用) 表现优秀, 这样的实现也可能会胜出. 竞争性众包软件开发中, 错误程序往往伴随着竞争性解决方案 (solution), 即竞争性实现 (competitive implementation). 竞争性实现包含大量有价值的信息, 但现有自动化或半自动化的缺陷定位方法并没有利用这些信息. 对同一需求的竞争性实现 (解决方案) 大部分都被丢弃了, 浪费了人力资源[24].

为了提升竞争性众包情景中错误定位的准确性, 本文提出利用竞争性实现的错误定位方法 (competitive implementation based fault localization, CBFL). 假设需求方在众包平台上面向大众公开提出了某个需求, 例如新的函数、方法或接口, 并规定了运行框架和编程语言, 最终收到若干相似或差异的实现. CBFL 能够在定位单个错误程序中的错误语句时, 充分利用来自其他实现的信息, 从而提升定位准确性. CBFL 的核心思路是, 一个语句与其参考程序 (竞争性实现) 中对应语句差异越大, 则越有可能错误; 反之, 如果与对应语句一致性越大, 则越有可能正确. 对一个错误程序及其相应的测试用例, 我们首先分别运行错误程序和参考程序, 并用基于频谱的错误定位方法计算错误程序中各语句的可疑度, 然后用两个步骤来调整可疑度: (1) 对错误程序中的每个条件语句, 根据语句在成功的测试用例运行时行为 (取值) 的一致性从参考程序中搜索其参考语句. 然后对比错误程序中条件语句与其参考语句在失败的测试用例运行时取值是否仍然一致, 若不一致, 则调高此条件语句的可疑度. (2) 对其他的语句, 通过文本相似性在参考程序中搜索其参考语句, 搜索到的相似参考语句越多, 则调整后此语句可疑度越低.

我们在包含 118 个伴随竞争性实现的真实错误程序的数据集上进行实验. 实验结果表明, 当竞争性实现存在时, 相比常用的基于频谱的错误定位方法 (SBFL), CBFL 能够减少 25% 的平均定位成本. 本文的主要贡献包括: (1) 针对众包软件开发情景提出了一个新的错误定位方法 CBFL; (2) 在真实错误程序组成的数据集上对 CBFL 进行初步验证, 结果表明, CBFL 能够明显提升错误定位的准确性.

1 方法原理

1.1 方法原理总览

本文提出的错误定位方法 CBFL 的流程图如图 1 所示. 整体看, CBFL 包含 3 个步骤.

• 首先, 用 SBFL 方法计算各语句初始可疑度.

• 其次, 基于语义相似性调整错误程序中条件语句的可疑度.

• 最后, 基于文本相似性调整错误程序中其他语句的可疑度.

设 p 是一个错误的程序, 对应地, 有若干竞争性实现, 记作 $CP=\{cp_1, cp_2, \ldots, cp_n\}$; 相应的测试用例记作 $TC=TC_{\text{passed}}\cup TC_{\text{failed}}$, 其中 TC_{passed} 和 TC_{failed} 分别表示在程序 p 上通过的和失败的测试用例.

CBFL 算法按照如下步骤计算各个语句的可疑度.

• 第 1 步, 在错误程序及其各个竞争性实现上运行测试用例并记录运行信息, 包括每个测试用例运行结果及覆盖语句情况.

• 第 2 步, 用基于频谱的错误定位方法 (SBFL)[10]来计算错误程序中各个语句的可疑度.

© 中国科学院软件研究所 http://www.jos.org.cn

• 第 3 步, 针对错误程序中的每个条件语句, 首先根据在通过的测试用例执行时取值的一致性, 从竞争性实现 (亦称作参考程序) 中搜索其相关语句 (亦称作参考语句). 然后根据二者在失败的测试用例执行时的差异性, 修改错误语句的可疑度.

• 第 4 步, 对于非条件语句, 根据文本相似性从竞争性实现中搜索与其语法相似的参考语句, 并根据搜索到含参考语句的程序数量修改其可疑度.

• 最后, 根据各个语句最终的可疑度对其重新排序.

在后面的小节中将详细叙述各步骤的细节.

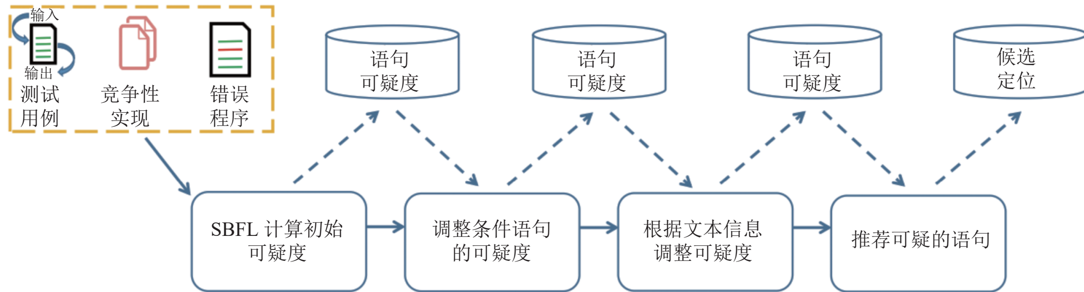

图 1　方法原理总览

1.2　语句初始可疑度的计算

本文选用广泛应用的 SBFL (基于频谱的错误定位)[10]方法计算待测试程序 p 中各语句的初始可疑度. 具体过程如下.

• 首先在程序 p 上运行测试用例并记录执行时的路径覆盖信息与运行结果 (成功或失败).

• 其次, 根据测试用例执行的覆盖路径, 对每个语句 st_i, 计算其被成功测试用例覆盖的次数 ($N_{cp}(st_i)$), 被失败测试用例覆盖的次数 ($N_{cf}(st_i)$), 测试用例执行成功且没有覆盖 st_i 的次数 ($N_{up}(st_i)$), 测试用例执行失败且没有覆盖 st_i 的次数 ($N_{uf}(st_i)$), 如表 1 所示.

表 1　程序测试的覆盖信息

是否覆盖语句	测试用例是否通过	
	是	否
是	N_{cp}	N_{cf}
否	N_{up}	N_{uf}

• 最后, 根据 SBFL 的计算公式来计算各个语句的可疑度:

$$Score(s)=\frac{N_{cf}(st_i)}{\sqrt{(N_{uf}(st_i)+N_{cf}(st_i))(N_{cp}(st_i)+N_{cf}(st_i))}}.$$

1.3　条件语句可疑度的调整

文献 [25,26] 指出条件语句是程序中相对较容易出错的位置. 为此, 本文设计了一个鉴别可疑条件语句的算法, 此过程分两部分: 第 1 部分, 针对错误程序中的某个条件语句, 从参考程序中搜索其参考语句 (如算法 1 所示). 第 2 部分, 通过对比该条件语句与参考语句的运行信息调整其可疑度 (如算法 2 所示).

第 1 部分的具体过程如算法 1 所示, 由于参考程序中的条件语句与错误程序中的条件语句可能数量不同, 或没有一一对应关系, 对此, 我们根据条件语句在成功的测试用例执行时的行为 (取值) 在参考程序中搜索相应的参考语句. 首先, 获取参考程序中所有条件语句 (算法 1 第 3–5 行), 并将它们与错误程序中的条件语句比较 (算法 1 第 7 行). 假设 cs_i 是程序 p 中的条件语句, cs_j 是参考程序 cp_k 中的一个条件语句. CBFL 通过如下过程 (算法 1 第 13–30 行) 判定 cs_j 是否为 cs_i 的参考语句.

© 中国科学院软件研究所 http://www.jos.org.cn

• 首先在所有测试用例中选取那些同时在错误程序 p 和参考程序 cp_k 运行成功的测试用例, 记作 $TS(cp_k$, passed, p, passed).

• 然后, 对 $TS(cp_k$, passed, p, passed) 中的每个测试用例 t, 如果错误程序中的条件语句 cs_i 与参考程序中的条件语句 cs_j 在测试用例 t 运行时取值为 true/false 的次数是相同的 (第 17–21 行), 我们就令参数 *Consistent* 加 1 (第 22 行), *Consistent* 的初值被设定为 0 (第 15 行).

• 最后, CBFL 判定 cs_j 为 cs_i 的一个参考语句当且仅当 *Consistent* > $|TS(cp_k$, passed, p, passed)$|\times\beta$ 成立 (第 25–28 行). 本文实验中, 参数 β 代表语句 cs_i 与 cs_j 具有对应关系的置信度, 我们设定显著性水平为 0.1, 因而置信度 β=0.9 (1–0.1), 即当观测到 cs_i 与 cs_j 在 90% 的情况下取值一致时, cs_j 被视为 cs_i 的参考语句.

算法 1. 搜索参考语句.

```
Input: TC; //测试用例
       cs_i; //条件语句
       p; //错误程序
       CP; //竞争性实现体
Output: RS. //cs_i 的参考语句

1. RS ← ∅
2. RunProgram(p, TC)
3. for each cp_k in CP do
4.    RunProgram(cp_k, TC)
5.    ConList_k ← GetConditionsStatements(cp_k)
6.    for each cs_j in ConList_k do
7.       if IsReference(p, cp_k, cs_i, cs_j, TC) then
8.          RS.add(cs_j)
9.       end if
10.   end for
11. end for
12. return RS
13. function IsReference(p, cp_k , cs_i, cs_j, TC)
14.   TS(cp_k, passed, p, passed) ← TC
15.   Consistent = 0
16.   for each ts in TS(cp, passed, p, passed) do
17.      true_csi ← Retrieval(cs_j, ts, true)
18.      true_csj ← Retrieval(cs_j, ts, true)
19.      false_csi ← Retrieval(cs_i, ts, false)
20.      false_csj ← Retrieval(cs_j, ts, false)
21.      if true_csi=true_csj && false_csi=false_csj then
22.         Consistent + +
23.      end if
24.   end for
25.   if Consistent > |TS(cp_k, passed, p, passed)| × β then
26.      return true
```

27. else
28. return false
29. end if
30. end function

第 2 部分的具体过程如算法 2 所示, CBFL 根据错误语句的执行情况来修改语句的可疑度. 假设 cs_i 是错误程序 p 中的条件语句, $RS=\{rs_1, rs_2, ..., rs_m\}$ 是上述第 1 部分中搜索到的一系列参考语句. CBFL 按照如下步骤调整 cs_i 的可疑度.

• 首先, 对每个参考语句 $rs_j=\langle cs_j, cp_j\rangle\in RS$, 我们从所有的测试用例中选取那些在程序 p 上运行不通过而在参考程序 cp_j 运行成功的测试用例, 记作 $TS(p, \text{failed}, cp_j, \text{passed})$(算法 2 第 8 行). 对 $TS(p, \text{failed}, cp_j, \text{passed})$ 中的每个测试用例 t, 如果错误程序中的条件语句 cs_i 与参考程序中的条件语句 cs_j 在运行 t 时取值为 true/false 的次数是相同的 (第 10–14 行), 我们就令 $Consistent(cs_i, rs_j)$ 加 1 (第 15 行), $Consistent(cs_i, rs_j)$ 的初值被设定为 0 (第 7 行). 如果 $Consistent(cs_i, rs_j)<|TS(p, \text{failed}, cp_j, \text{passed})|\times\alpha$ 成立且 $TakeEffectP(cp_j)=0$ (在本文中, α 默认设置为 $1-\beta=0.1$, 即 cs_i 与 cs_j 一致的置信度为 0.1, 矛盾的置信度为 $\beta=0.9$), 我们就令 $Conflict(cs_i)$ 和 $TakeEffectP(cp_j)$ 各自加 1 (第 19–20 行). 其中 $Conflict(cs_i)$ 和 $TakeEffectP(cp_j)$ 初值被设定为 0 (第 2, 4 行). 我们采用变量 $TakeEffectP(cp_j)$ 的目的是避免同一个竞争性实现具有多个参考语句而过度影响 cs_i 的可疑度.

• 我们用如下公式来调整 cs_i 的可疑度:

$$Score(cs_i)_{\text{new}}=Score(cs_i)+(1-Score(cs_i))\times\frac{Conflict(cs_i)}{n},$$

其中, $Score(cs_i)$ 是条件语句 cs_i 的初始可疑度, n 是参考程序 (竞争性实现) 的数量.

算法 2. 调整条件语句的可疑度.

Input: TC; //测试用例
cs_i; //条件语句
P; //错误程序
CP; //竞争性实现体
Output: $Score(cs_i)$. // cs: 调整后的可疑度

1. $Score(cs_i)\leftarrow$ SBFL(TC, p)
2. $Conflict(cs_i)\leftarrow 0$
3. for each cp_k in CP do
4. $TakeEffectP(cp_k)\leftarrow 0$
5. end for
6. for each rs_j in RS do
7. $Consistent(cs_j, rs_j)\leftarrow 0$
8. $TS(p, \text{failed}, cp_j, \text{passed})\leftarrow TC$
9. for each ts in $TS(p, \text{failed}, cp_j, \text{passed})$ do
10. $\text{true}_{csi}\leftarrow Retrieval(cs_i, ts, \text{true})$
11. $\text{true}_{rsj}\leftarrow Retrieval(rs_j, ts, \text{true})$
12. $\text{false}_{csi}\leftarrow Retrieval(cs_i, ts, \text{false})$
13. $\text{false}_{rsj}\leftarrow Retrieval(rs_j, ts, \text{false})$
14. if $\text{true}_{csi}=\text{true}_{rsj}$ && $\text{false}_{csi}=\text{false}_{rsj}$ then
15. $Consistent(cs_i, rs_j)++$

www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn

16. end if
17. end for
18. if $Consistent(cs_i, rs_j) < |TS(p, \text{failed}, cp_j, \text{passed})| \times \alpha$ && $TakeEffectP(cp_j) = 0$ then
19. $Conflict(cs_i)$ + +
20. $TakeEffectP(cp_j)$ + +
21. end if
22. end for
23. $Score(cs_i)_{new} \leftarrow Score(cs_i) + ((1 - Score(cs_i)) \times Conflict(cs_i))/n$

为展示算法的具体过程, 我们在代码片段 1 中展示错误程序 p 的片段以及 p 的一个参考程序 cp_k. 错误程序和参考程序均来自任务 59A (http://codeforces.com/problemset/problem/59/A). 代码片段 1 中, 错误代码段中的 if 语句是错误的, 参考程序中的 if 语句也是错误的. 但本文方法能够利用参考程序中的 if 语句信息识别错误程序中 if 语句的异常性, 从而判定其有误.

在这个任务中, 输入一个字符串, 如果小写字母数量 num_l 大于或等于大写字母数量 num_u, 则要求程序员输出字符串的小写形式, 否则要求输出大写形式. 错误程序 p 首先计算了 num_l 和 num_u, 对应变量 $cnt1$ 和 $cnt2$. 但是, 程序 p 使用了错误的条件 $cnt1>cnt2$, 而不是预期的条件 $cnt1 \geqslant cnt2$. 换言之, 并没有考虑 $cnt1==cnt2$ 的情形. 而参考程序亦包含错误的语句. 该程序员首先计算 num_l, 对应变量 $count$, 并给出了条件 $count \geqslant s.length()/2$. 此程序员考虑了输入字符串大小写字母数相同的情形, 因而他用了"≥"而不是">". 但是他没有考虑在 Java 程序中, 当除法运算出现除不开 (有余数) 的情况时, 返回的数值会自动取整, 而不是返回小数值. 此时, 当 $num_l=num_u-1< s.length()/2$ 时, 条件 $count \geqslant s.length()/2$ 的布尔值为 true, 但在现实中, 该条件为 false. 综合上述分析, 我们可以将全部的测试用例 TS 根据是否在 p 或 cp_k 上执行通过而分成 4 个不相交的子集, 如表 2 所示.

代码片段 1. 调整条件语句可疑度示例.

```
//错误代码段
…
if (cnt1 > cnt2)
//正确条件: cnt1 ≥ cnt2
        s=str.toLowerCase() ;
else
        s=str.toUpperCase() ;
system.out.println(s) ;
//竞争性实现体中的代码段
…
if (count ≥ s.length() /2){
//正确的条件:
//count×2 ≥ s.length()
        system.out.println (s.toLowerCase()) ;
}
else {
        system.out.println (s.toUpperCase()) ;
}
```

© 中国科学院软件研究所 http://www.jos.org.cn

表 2 错误程序与参考程序的运行结果

	程序p运行通过	程序p运行不通过
程序cp_k运行通过	TS(p, passed, cp_k, passed): 此时$num_l \neq num_u$ 且$num_l \neq num_u -1$	TS(p, failed, cp_k, passed): 此时$num_l = num_u$
程序cp_k运行不通过	TS(p, passed, cp_k, failed): 此时$num_l = num_u -1$	TS(p, failed, cp_k, failed): 这种情形不存在

我们将错误程序 p 中的条件 $cnt1 \geqslant cnt2$ 记作条件语句 cs_i, 将参考程序 cp_k 中的条件 $count \geqslant s.length()/2$ 记作条件语句 cs_j, 它们在运行测试用例时的布尔值如表 3 所示.

表 3 错误语句与参考语句的运行信息

	程序p运行通过	程序p运行不通过
程序cp_k运行通过	TS(p, passed, cp_k, passed): 此时若$num_l < num_u$ 则cs_i=cs_j=false ;若$num_l > num_u-1$则cs_i=cs_j=true	TS(p, failed, cp_k, passed): 此时cs_i = false且cs_j = true
程序cp_k运行不通过	TS(p, passed, cp_k, failed): 此时cs_i = false 且cs_j = true	TS(p, failed, cp_k, failed): 这种情形不存在

在本例中, cs_i 和 cs_j 都不在循环结构中, 因此最多被执行一次, 即, 它们在每个测试用例运行时取值为 true/false 的次数只能是 0/1 或者 1/0. 由于在 TS(p, passed, cp_k, passed) 中的测试用例 (表 3 左上) 执行时, 必有 $cs_i = cs_j$, 我们可以得出结论: $Consistent = |TS(cp_k, \text{passed}, p, \text{passed})| > |TS(cp_k, \text{passed}, p, \text{passed})| \times \beta$, 其中 $Consistent$ 和 β 在算法 1 中被定义. 从而根据前文的叙述, cs_j 是 cs_i 的参考语句. 接下来, 我们来判别它们是否在 p 执行不通过而 cp_k 执行成功的测试用例运行时具有差异性. 根据算法 2, 两个条件语句 cs_i 和 cs_j (在算法 2 中记作 rs_j) 定义为不一致当且仅当 $Consistent(cs_i, cs_j) < |TS(p, \text{failed}, cp_j, \text{passed})| \times \alpha$. 在此例中, 由于此时 false = $cs_i \neq cs_j$ = true, 必有 $Consistent(cs_i, cs_j)=0$ (参照表 3 右上). 于是, cs_j 被认为是 cs_i 的反例, 因而条件语句 cs_i 被认为是更加可疑的, 其可疑度被调高 (根据算法 2 中的公式).

注意, 在上述例子中, 第 1 个代码片段 p 是错误程序, 而第 2 个代码片段 cp_k 是参考程序. 然而, 当我们尝试对第 2 个代码片段进行错误定位时, p 与 cp_k 、cs_i 与 cs_j 就会互换. 根据表 3 中的执行信息, 我们亦能够鉴别出第 1 段代码中的语句 if ($cnt1 > cnt2$) 是第 2 段代码中条件 if ($count \geqslant s.length()/2$) 的一个反例. 因此, 这个例子揭示了本文方法 CBFL 在错误定位上的优越性: 针对同一软件需求的众包软件可以相互作为错误定位的有价值信息进行参考, 即使我们不能预先保证参考程序的正确性.

1.4 基于文本信息调整语句可疑度

除了条件语句, CBFL 也利用参考程序的文本信息来调整其他语句的可疑度. 这样做的假设是, 如果一个语句经常出现 (在错误程序和多个参考程序中出现), 那么这样的语句是不可疑的.

基于文本调整可疑度的具体过程如算法 3 所示. 对于错误程序 p 及其参考程序 $cp_1, cp_2, ..., cp_n$, 按照如下步骤调整语句的可疑度.

- 第 1 步, 收集错误程序中的所有语句, 记作序列 L_p (算法 3 第 1 行).
- 第 2 步, 收集各个参考程序 cp_i 的所有语句, 记作序列 L_i.
- 第 3 步, 对 L_p 和 L_i 中的每个语句, 把语句中的变量名抽象为变量的类型 (算法 3 第 2, 5 行). 例如, 语句“int sum = 0”被抽象为“int int = 0”. 进行抽象化的目的是为了排除不同程序中对应的变量可能会使用不同变量名的干扰. 此过程中, 同一程序中的不同语句抽象化后可能会相同, 此时只记录一次, 以避免重复.
- 第 4 步, 对错误程序中的每个语句 st_i, 计算包含相同语句的参考程序数量 (算法 3 第 9–12 行). 如果至少 1/3 的参考程序含有该语句, 我们就根据如下公式降低该语句的可疑度 (算法 3 第 14–16 行):

$$Score(st_i)_{\text{new}} = Score(st_i) \times \left(1 - \frac{m}{n}\right),$$

其中, m 是包含与 st_i 相同语句的参考程序数量, n 是参考程序总数. 此公式的意义在于, 我们根据含有与 st_i 相同语句的参考程序数量来调整语句 st_i 的可疑度, 避免在极端情况下个别参考程序中的错误语句会误导定位的情况.

© 中国科学院软件研究所 http://www.jos.org.cn

算法 3. 基于文本相似性调整可疑度.

Input: p; //错误程序
  CP; //竞争性实现体
Output: L_p. //可疑语句

1. $L_p \leftarrow ParseProgram(p)$
2. $L_p \leftarrow ReplaceVariable(L_p)$
3. for each cp_i in CP do
4. $L_i \leftarrow ParseProgram(cp_i)$
5. $L_i \leftarrow ReplaceVariable(L_i)$
6. end for
7. for each st_i in L_p do
8. $m \leftarrow 0$
9.   for each L_i in $\{L_i\}$ do
10.   if $L_i.contain(st_i)$ then
11. m++
12.   end if
13.   end for
14.   if $m \geqslant n/3$ then
15. $Score(st_i)_{new} \leftarrow Score(st_i) \times (1 - m/n)$
16.   end if
17. end for
18. Sort L_p according to $Score(st_i)$

2 实验结果评估

2.1 研究问题

为了评估 CBFL 方法的在错误定位中的性能, 本文探究如下问题.

- 问题 1: CBFL 方法是否优于被广泛使用的 SBFL 方法?
- 问题 2: 对条件语句可疑度的调整是否有效果?
- 问题 3: 基于文本的可疑度调整是否有效果?
- 问题 4: 用不同的 SBFL 方法计算初始可疑度, 是否会影响 CBFL 的效果?
- 问题 5: CBFL 方法是否有效定位不同类型的错误语句?

问题 1 在于比较 CBFL 与 SBFL 的性能. 为此, 我们选择最常用的 3 个 SBFL 方法[25]: *Jaccard*[11]、*Ochiai*[10]以及 *Tarantula*[12]. 这 3 个 SBFL 方法都是根据测试用例执行时的语句覆盖情况计算各语句可疑度, 计算公式如下:

$$Jaccard(s) = \frac{N_{cf}}{N_{cf}+N_{cp}+N_{uf}},$$

$$Ochiai(s) = \frac{N_{cf}}{\sqrt{(N_{uf}+N_{cf})(N_{cp}+N_{cf})}},$$

$$Tarantula(s) = \frac{\dfrac{N_{cf}}{N_{uf}+N_{cf}}}{\dfrac{N_{cf}}{N_{uf}+N_{cf}} + \dfrac{N_{cp}}{N_{up}+N_{cp}}}.$$

© 中国科学院软件研究所 http://www.jos.org.cn

此外, 本文还将 DStar[3]纳入对比, 因为 DStar 是最新的 SBFL 工具, 并且被证明比其他 SBFL 更有效. 它的计算公式为:

$$D_n(s)=\frac{N_{cf}^n}{N_{uf}+N_{cp}}.$$

要注意的是, 本文并不是要证明本文的方法比 SBFL 优异. 由于 SBFL 并非针对众包软件设计, 因而没有利用竞争性实现的信息. 我们要证明的是在存在竞争性实现可以利用的情况下, CBFL 法能够明显提升错误定位的准确性. 问题 2 和问题 3 探究的是 CBFL 两个部分各自的效果. 通过分别单独使用 CBFL 的两个部分进行实验, 证明两个部分都是有价值的. 问题 4 探究的是 CBFL 的可推广性. 为了得到错误程序中各个语句初始的可疑度, CBFL 利用了目前自动修复工具所常用的 SBFL 方法. 为了证明在选择任意 SBFL 计算公式时 CBFL 都有效果, 本文选择不同的 SBFL 公式计算初始可疑度并进行实验. 问题 5 探究的是 CBFL 方法在定位不同类型错误时的适用性. 为此, 我们对实验结果进行了具体的深入分析.

2.2 实验设计

为了衡量 CBFL 的效果, 我们从 Codeforces (http://codeforces.com/) 网站上下载错误程序. Codeforces 网站是世界知名的编程平台. 每一个公开的软件需求都能获得大量的竞争性实现. 从成千上万的软件需求中, 本文实验选择了最热门的 30 个. 对每个软件需求, 选择满足如下要求的 10 个最新提交 (同一用户的重复提交只取一个).

- 程序用 Java 语言编写.
- 至少在一个测试用例上运行失败 (给出错误的输出).
- 在该程序员的下一个提交版本中, 通过修改或删除原错误程序中的语句, 使程序被成功修复.

最终收集到来自 30 个软件需求的 300 个程序, 其中 118 个错误程序因明确改动了若干语句而被修复, 在实验中被用作错误程序进行定位, 同时针对该软件需求的其他程序在进行错误定位时被用作参考程序. 其余 182 个程序因被程序员重写而修复, 在本实验中不作为错误程序进行研究, 而作为其他程序错误定位时的参考程序. 数据集详细信息如表 4 所示. 第 1 列表示编程任务序号, 详细内容在 Codeforces 官网均可查阅. 第 2–4 列表示每个任务中程序的总行数、总语句数、总测试用例数. 第 5 列表示每个任务的 10 个提交中用于进行定位实验的错误程序数.

2.3 实验过程

对软件需求 $Task_i$ 所获得的每个错误程序 (提交) $P_{i,j}$, 本文的实验按如下过程进行.

- 第 1 步, 用 $P_{i,j}$ 作为错误程序, 用需求 $Task_i$ 所获得的其余所有程序 (提交) 作为竞争性实现 (参考程序) 进行实验.
- 第 2 步, 用 SBFL 方法在程序 $P_{i,j}$ 上进行实验, 作为对照组.
- 第 3 步, 在参考程序的帮助下用 CBFL 在程序 $P_{i,j}$ 上进行实验并比较其与 SBFL 的效果.

我们的实验环境是: Lenovo ThinkCenter, Windows 10, Intel(R) Xeon(R) E5 2640 v4 (10 cores), 16 GB RAM.

2.4 实验度量

为了衡量 CBFL 的效果, 文本主要使用两个在错误定位研究中常用的度量指标: 累计审查语句 (cumulative number of statements examined, CNSE)[3]以及相对改进 (relative improvement, RImp)[27]. CNSE 是指揭示所有错误版本中错误语句所需要审查的语句数量. CNSE 越小, 表示错误定位越准确. 由于不同语句可能会有相同的可疑度, 本文使用 3 个情形下的 CNSE 度量: 最好情形, 平均 (均值) 情形, 以及最差情形. 最好情形指的是错误语句在与其具有相同可疑度的语句中第 1 个被程序员审查的情形. 最差情形指的是错误语句在与其具有相同可疑度的语句中最后一个被程序员审查的情形. 平均情形是这两者的算数平均数, 表示 CNSE 的数学期望值. RImp 是为了比较两个不同的错误定位方法在揭示错误前所需要审查语句数量上的差距. RImp 越低表示新方法的提升越显著. 我们的实验按照文献 [3] 的方法, 用累计审查语句的数量来计算 CNSE 和 RImp. 由于一共有 30 组程序, 限于篇幅空间, 我们在实验结果中仅展示审查语句总数及其对应的其他数据.

© 中国科学院软件研究所 http://www.jos.org.cn

此外, 为了度量 CBFL 在具体每个程序上的定位效果分布, 我们还使用两个额外的常用度量指标: Top-*K* 和 MFR (mean first rank). Top-*K* 表示实验中某种定位方法能够将错误语句排序到前 *K* 个语句的程序数量. MFR 表示某种定位方法在所有程序中各自检查出第 1 处错误所需要的审查语句数 (即成本) 的中位数.

表 4 实验程序信息

编程任务序号	代码行数	可执行语句数	测试用例数	错误版本数
1A	169	102	20	5
4A	132	61	20	5
41A	239	122	40	5
50A	181	98	35	5
58A	250	143	40	1
59A	241	122	30	6
96A	245	142	44	3
110A	260	130	34	3
122A	202	116	53	1
118A	204	113	42	3
122A	261	138	36	6
131A	355	185	56	1
133A	209	110	85	5
148A	295	90	40	7
236A	230	127	85	5
263A	316	168	25	4
266A	236	136	34	2
266B	310	167	44	2
271A	236	142	27	5
281A	152	75	10	2
318A	266	141	25	7
339A	352	221	21	7
443A	218	117	22	4
451A	192	85	28	2
472A	198	103	33	3
479A	259	157	29	2
486A	154	69	39	4
546A	196	114	15	3
617A	133	62	34	6
734A	225	140	15	4

2.5 问题 1: 对 SBFL 的提升

为了回答问题 1, 本文对比 CBFL 与 4 个常用 SBFL 方法在 118 个众包程序上定位错误的效果. 表 5 和图 2 展示了累计成本的实验结果. 表 5 中第 2–6 列分别表示 *Ochiai*, *Tarantula*, *Jaccard*, DStar 和 CBFL 的实验数据. 第 2–4 行分别表示在最好、平均、最差情形下的 CNSE 指数. 图 2 展示了本文方法与不同 SBFL 方法相比的 RImp 指标. 例如, 根据图 2 中 *Ochiai* 的实验结果, 在最好、平均、最差的情形下, CBFL 与 *Ochiai* 相比仅需 90%, 73%, 68% 的成本就能揭示错误. 从表 5 和图 2 中分析可得出如下结论.

表 5 各错误定位技术的实验结果 (CNSE)

情形	*Ochiai*	*Tarantula*	*Jaccard*	DStar	CBFL
最好	255	214	267	247	230
均值	555.5	545.5	566	585.5	408
最差	856	877	865	924	586

© 中国科学院软件研究所 http://www.jos.org.cn

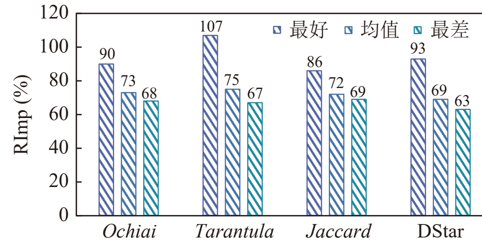

图 2　CBFL 对各错误定位工具的 RImp

(1) 本文方法 (CBFL) 所需要审查的语句明显比 SBFL 方法少. 根据表 5, CBFL 的 CNSE 指标在最好、平均、最差的情形下分别是 230, 408, 586. 而 SBFL 方法最优秀的结果则分别是 214、545.5、856. 按照平均值 (数学期望) 计算, CBFL 方法比最优的 SBFL 方法减少了 25.2% = (545.5−408)/545.5 的成本.

(2) CBFL 在最差的情形下效果更为优异. 亦即, CBFL 更擅长通过排除正确的语句来减少人工审查的成本. 根据图 2, CBFL 在最差的情形下 RImp 从 63% 至 69% 不等.

(3) CBFL 在最好的情形下效果相对一般. 亦即, CBFL 在鉴别潜在错误的语句上相对效果有限. 根据表 5, CBFL 的 CNSE 大于 *Tarantula*, 小于 *Ochiai*, *Jaccard* 以及 DStar. 这个结果表明在最好的情形下 CBFL 的不一定能超越所有 SBFL 方法. 然而, 在与错误语句具有相同可疑度的语句中, 错误语句恰好最先被审查的概率在实际中发生的概率低. 在大多数情况下, 程序员需要人工检查的语句数量是介于最好和最坏情形之间的.

(4) 4 个不同 SBFL 方法的 CNSE 较为接近. 根据表 5, 其差距在 10% 以下. 因此可以认为, 在错误定位上, 仅靠改变 SBFL 的公式是难以获得较大的效果提升的. 反之, 利用来自竞争性实现 (参考程序) 的文本或执行信息对提升定位准确性非常有帮助.

为了全面考察 CBFL 的效果, 我们统计了在各个程序上 CBFL 与 SBFL 的定位效果, 并统计 Top-*K* (本文中取 Top-1 和 Top-5) 和 MFR. 表 6 展示了“均值”情况下的结果. 从表 6 中, 我们可以得出如下结论.

表 6　各错误定位技术的实验结果 (Top-*K* 和 MFR)

参数	*Ochiai*	*Tarantula*	*Jaccard*	DStar	CBFL
Top-1	17	13	17	19	35
Top-5	75	76	75	69	91
MFR	4	4	4.5	4.5	2

(1) CBFL 方法在 Top-1 和 Top-5 指标上领先其他 SBFL 方法 84.2%=(35–19)/19 和 19.7%=(91–76)/76, 表明了 CBFL 更容易精准地定位错误语句.

(2) CBFL 方法在 MFR 指标上比其他 SBFL 方法明显降低, 而其他 SBFL 方法的 MFR 指标较为接近, 这证明了 CBFL 通过利用参考程序的信息, 能够在 SBFL 的基础上进一步提升定位的准确性, 降低定位成本.

2.6　问题 2: 调整条件语句可疑度的效果

为了回答问题 2, 我们仅使用 CBFL 中调整条件语句可疑度的部分并重新进行实验. 表 7 展现了实验结果. 结果表明, 针对条件语句调整可疑度的部分明显减小了错误定位的成本. 按照均值 (数学期望) 计算, 减少的幅度占完整 CBFL 方法减少数量的 21% = 1−(555.5−439)/(555.5−408). 注意到去掉调整条件语句可疑度的步骤后, 最差情况下成本依然增加, 而在最好的情况下, 定位成本进一步减少 32%=(255−222)/(255−230)−1, 这是由去掉该步骤后, CBFL 方法区分同一程序块内各语句错误概率的能力下降导致的, 最终表现为均值成本的上升 (408 上升至 439). 这证明了调整条件语句可疑度的步骤对从同可疑度的语句中鉴别错误语句的重要性.

2.7　问题 3: 基于文本信息调整可疑度的效果

为了探究问题 3, 我们仅使用 CBFL 方法中基于文本相似性调整可疑度的部分并重新进行实验. 表 7 展现了

© 中国科学院软件研究所 http://www.jos.org.cn

实验结果. 结果表明, 基于本文相似性调整可疑度的效果是显著的. 按照均值情形衡量, 其减少考察语句数量的比例达到整体方法的 63% = 1−(555.5−501.5)/(555.5−408).

表 7 使用单个调整可疑度方法时 CBFL 的实验结果 (CNSE)

情形	SBFL	取消调整条件语句可疑度的CBFL	取消基于文本信息调整语句可疑度的CBFL	完整的CBFL
最好	255	222	260	230
均值	555.5	439	501.5	408
最差	856	656	743	586

为了探究基于本文相似性调整可疑度如何提升错误定位的准确性, 我们人工检查了实验数据集, 发现有些正确语句的初始可疑度大于错误语句的初始可疑度, 但是由于被 CBFL 方法鉴定为高频出现的语句, 因而其可疑度被降低了.

代码片段 2. 基于文本信息调整可疑度示例.

```
//code snippet in another
// competitive implementation
…
if (sml > big){
        out = ss.toLowerCase () ;
}
else {
        out =ss.toUpperCase () ;
}
…
```

本文提出基于文本相似性调整可疑度的依据是, 如果一个语句在若干不同程序员各自提交的程序中都出现, 那么该语句是高频语句, 出错的可能性较低. 此处仍以代码片段 1 中的第 1 段代码为例. 如表 3 所示, 只要程序 p 没有通过测试用例, 那么条件语句 cs_i (即 if ($cnt1$>$cnt2$)) 的布尔值必然是 false, 因而语句 s = str.toUpperCase() 一定会被执行. 此时相比于真正错误的语句 (即 if ($cnt1$>$cnt2$)), s = str.toUpperCase() 被更少的执行成功的测试用例覆盖且被相同数量的执行不通过的测试用例覆盖. 于是, SBFL 方法倾向于给语句 s = str.toUpperCase() 赋予比真正错误语句更高的可疑度. 然而, CBFL 方法通过分析文本相似性, 能够判定高频的语句. 代码片段 2 展示了一个参考程序 cp_i 中的代码片段. 程序 cp_i 中的语句 out = ss.toUpperCase() 在文本上等价于程序 p 中的语句 s = str.toUpper-Case(). 虽然它们包含不同的变量名, 但是 CBFL 方法能够通过将语句抽象化为 String = String.toUpperCase() 消除此影响. 如果有多个参考程序中都包含 String = String.toUpperCase() 这样的语句, 那么程序 p 中的语句 s 就会被认为是高频语句, 其可疑度将被降低.

从效果上看, 相比于针对条件语句可疑度的调整, 基于文本相似性的可疑度调整更重要. 这是因为后者能够处理多种类型的语句, 而前者只能处理条件语句, 因而发挥作用的空间相对有限.

2.8 问题 4: CBFL 的稳定性

本文的 CBFL 方法采用了 SBFL 方法中的 *Ochiai* 公式计算各个语句的初始可疑度. 由于任意的 SBFL 公式均可替换 *Ochiai* 公式, 问题 4 旨在探究本文方法的稳定性. 为此我们用上述实验中另外 3 个 SBFL 公式作为计算初始可疑度的方法, 然后重复实验. 实验结果如表 8 和图 3 所示. 表 8 展现了在使用不同的 SBFL 公式计算语句的初始可疑度时, 本文方法定位错误所需要考察的语句总数量. 图 3 展现了 CBFL 方法基于不同 SBFL 公式的 RImp 值. 从表 8 和图 3 中, 我们能得到如下结论.

© 中国科学院软件研究所 http://www.jos.org.cn

表 8　用不同工具计算初始可疑度时 CBFL 的实验结果 (CNSE)

情形	$CBFL_{Och}$	$CBFL_{Tar}$	$CBFL_{Jac}$	$CBFL_{DSt}$
最好	230	236	241	239
均值	408	440.5	414.5	450.5
最差	586	645	588	662

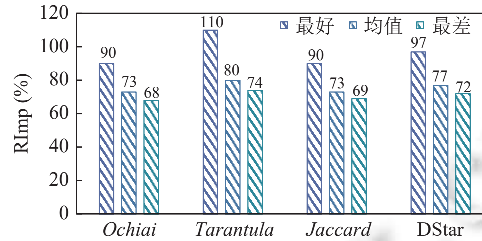

图 3　用不同工具计算初始可疑度时 CBFL 对各错误定位工具的提升 (以 RImp 计)

(1) 在选择 4 个不同的 SBFL 方法计算初始可疑度时, CBFL 方法都能减少发现错误所需要审察的语句数量. 按照均值情况下的 RImp 衡量, 本文的方法相对于 SBFL 只需要检查 73%–80% 的语句.

(2) 在最坏情形下的 RImp, 使用不同 SBFL 计算初始可疑度时对定位效果的提升相差不大, 从 68% 至 74% 不等.

(3) 在最好的情形下, 本文方法不如 *Tarantula* 有效. 这是因为 *Tarantula* 倾向于对语句赋予相同的可疑度, 导致其在最好情形的定位效果准确, 而最差情形的效果不准确. 然而, 最好的情况只是理论上存在的极端情况, 在实际中出现的概率低[1].

综上, 我们可以得出结论: CBFL 方法是具有稳定性的. 在使用不同的 SBFL 方法计算初始可疑度时, CBFL 方法都显示出优越性, 尤其是在最差情形中.

2.9　问题 5: CBFL 的可扩展性

为了探究 CBFL 能否有效地定位不同类型的错误语句, 我们检查了实验中的 118 个程序, 统计了在定位不同错误语句时 CBFL 与 SBFL 比较的结果, 统计结果如表 9 所示.

表 9　在定位不同类型错误时 CBFL 与 SBFL 占优数对比

情形	if语句条件	for语句条件	输出语句	其他语句
CBFL占优	32	13	16	7
二者性能相等	12	3	17	17
SBFL占优	0	0	1	0
CBFL占优比例 (%)	73	81	47	41

从表 9 的数据中, 我们能得到如下结论.

(1) CBFL 能够在定位不同类型的错误语句时, 都有一定数量优于 SBFL 方法, 提升定位准确性.

(2) 在绝大多数情况下, CBFL 的性能优于或等于 SBFL, 只有 1 个程序上 CBFL 性能劣于 SBFL. 这证明了 CBFL 的实用性.

(3) CBFL 在定位错误的条件语句时效果更佳明显. 这是因为相比 SBFL, CBFL 考虑到了条件语句的语义信息, 进而更加精准地识别潜在错误语句.

(4) CBFL 在定位输出语句和其他常规性语句 (赋值语句等) 性能稍弱, 这是因为某些错误语句的语法结构并没有错误, 即 AST 树结构是正确的, 而本文的 CBFL 方法在基于文本相似性调整可疑度时, 考虑的是语法结构相似性, 因而难以识别出部分错误语句.

© 中国科学院软件研究所 http://www.jos.org.cn

综上, 我们可以得出结论: CBFL 方法能够有效地定位各种类型的错误语句, 并且在定位错误的条件语句时效果显著.

3 有效性讨论

本文实验有效性的一个外部威胁 (external threat to validity) 是, 使用的度量是否能够很好的衡量错误定位的效果. 本文使用的度量是计算发现错误之前所需要审查的语句数量. 首先, 为了能够全面地展现错误定位技术的准确性, 在实验度量中考察最好、平均、最差这 3 个情形下的实验结果. 相关研究指出[28–30], 当程序中存在若干可疑度与错误语句相等的其他语句时, 错误语句被第一个或者最后一个审查的概率较低的, 因而单独使用最好情形或最差情形下的度量指标均不能完全揭示错误定位的效果. 同时, 本文也使用了 Top-N 和 MFR 来度量定位方法在实验数据集中的有效性分布情况, 能够揭示定位方法的实用性和稳定性.

另外, 审查语句数量的度量基于一个假设[31]: 用户是理想的用户. 换言之, 只要一个错误语句被用户检查到时, 用户必定能够发现其错误. 在实践中, 程序员可能需要参考上下文才能确定一个语句是否含有错误. 此外, 程序员也有可能犯错误, 将正确的代码误判为有错的. 因而, 发现错误实际上需要审查更多的代码. 文献 [1] 指出, 发现错误代码的资源不仅是人力成本, 也包括工具、计算资源、时间、存储空间成本等等. 然而, 这个威胁是针对所有自动化错误定位技术的, 不仅仅是本文提出的 CBFL.

另一个对有效性的外部威胁是, 在本文的实验中, 每个程序都有 9 个竞争性实现. 如果在现实中某个众包软件需求并没有获得相应数量的实现, 则本文的错误定位方法有可能会出现性能下降的问题. 为了考察本文的方法在竞争性实现数量少的情况下定位效果, 我们将探索了 CBFL 在具有 4–9 个参考程序情况下的在 118 个程序上定位的实验效果, 结果表明, 当参考程序数上升时, CBFL 方法检测错误语句的成本中位数 (MFR) 是相等的 (均为 2), 这证明了 CBFL 在具有 4 个以上参考程序时均具备有效性. 另一方面, 在具体的每个程序上, 我们观测到当参考程序数量上升能时, CBFL 在更多的程序上定位成本呈稳定或下降趋势, 因为 CBFL 的稳定性会随着参考程序数量提升. 这是因为参考程序越多, CBFL 受到个别极端情况的参考程序误导的概率越低, 效果越稳定. 所以 CBFL 应当选择使用更多的竞争性实现. 在实际的众包软件开发中, 以 TopCoder 网站[22]为例, 需求发布者为了收获更多高质量的实现体, 会为分数最高的若干个合格的实现体按顺序支付奖金, 并且总得分的评定综合考虑了提交的先后与软件质量, 因而一个项目能够充分吸引多个在线开发者提交竞争性实现体.

此外, 与普通程序相比, 本文实验中利用的数据集是 Codeforces 网站上的开源程序与编程任务, 程序规模相对较小, 因而我们的数据集能否代表真实的众包软件及其缺陷, 可能会对本文的有效性构成威胁. 本文利用 Codeforces 网站程序的原因有两方面. 其一, 在众包软件情景中, 需求发布者往往会将复杂的需求拆分成若干简单需求后发布, 因而众包软件的规模亦小于普通程序. 据 TopCoder 网站的官方统计, 80% 以上的众包软件限定工期不超过 3 周, 平均报酬不超过 4 000 美元, 这些数据证明众包软件情景中的程序是较为简单的. 另一方面, 现实中的众包软件并没有被其平台公布. 同时, 我们能够从 Codeforces 网站上下载不同程序员提交的实现体, 这模仿了现实中在网络平台上提交众包软件的不同水平的程序员, 证明了本文实验中数据集的多样性和 CBFL 的实用性.

第 4 个外部威胁是实验是否充分, 以及实验结果是否有说服力. 为了避免这个外部威胁, 本文在来自 30 个不同类型软件需求的 118 个错误程序. 并且这些程序中的错误都是真实的而不是人工植入的, 这保证了实验数据集具有一定代表性.

对实验有效性的一个内部威胁 (internal threat to validity) 是实验假设是否成立. 在前文中提到, 本文假设是, 如果错误程序中的某个语句与大部分竞争性实现中的参考语句具有一致性, 那么这个语句错误的可能性小; 反之, 如果该语句与参考语句有差异, 则有错误的可能性大. 我们做这样的假设是因为, 多个不同的程序员在实现相同的软件需求时, 在同一位置犯同一错误的可能性较低. 我们在本次实验的 118 个错误程序上验证了该假设. 在 118 个错误程序中共计包含 345 个条件语句和 988 个非条件语句. 在 56 个错误的条件语句中, 有 33%=(19/56) 被发现在运行时的动态取值与参考程序相比是异常的. 相反, 只有 4.8%=(14/289) 的正确条件语句在运行时的动态取值与参考

© 中国科学院软件研究所 http://www.jos.org.cn

程序相比异常. 类似地, 在 54 个错误的非条件语句中, 只有 11%=(6/54) 在参考程序中能搜索到相似的语句, 而对于正确的非条件语句, 这个比例是 21%=(200/934). 我们同时也发现了错误的参考程序可能会误导我们的方法, 使正确的语句被错判为错误语句, 反之亦然. 但是, 实验表明这种情况并不多见, 因而利用参考程序帮助错误定位仍然是有价值的.

对有效性的另一个内部威胁是本文提出的错误定位方法既需要执行错误的程序, 又需要执行参考程序, 并且还需要分析更多的信息. 我们观察运行日志发现, 错误定位的主要时间成本是运行程序的成本, 分析信息并对语句排序的时间成本非常低 (小于 0.1 s). 因而, CBFL 方法所需要的时间成本仅是随着参考程序的数量线性上升, 这个算法复杂度是可接受的.

最后, 参考程序中的错误语句也可能构成有效性威胁, 亦即, 参考程序中也可能包含错误语句, 进而误导 CBFL 调整语句可疑度; 或者参考程序中的正确语句经过抽象化后与错误程序中的错误语句结构相同, 也可能误导 CBFL. 本文的方法设计考虑到了这种威胁, 因而在第 1.4 节的根据文本信息调整可疑度时, 考虑了含有相似语句的参考程序的数量, 当个别参考程序造成误导时, 对调整语句可疑度的幅度影响较小, 从而避免了影响 CBFL 的稳定性. 同时, 本文的方法也设定需要 1/3 的参考程序出现某共同语句才能够用以考虑调整错误程序中的语句可疑度, 亦可避免极端情况误导定位方法.

4 相关研究

本节介绍关于自动化错误定位的研究工作. 文献 [25] 指出, 在各类技术中, 基于频谱的方法 (SBFL) 是最广泛用于代码修复的, 其中, 32% 的代码修复工具采用 GenProg[13], 18% 采用 *Tarantula*[12], 18% 采用 *Ochiai*[10], 5% 采用 *Jaccard*[11]. 在本文的实验采用了 *Tarantula*, *Ochiai* 和 *Jaccard*. GenProg 没有被采用的原因是它给所有潜在错误的语句 (通过和不通过的测试用例运行时都有覆盖) 赋予同样的可疑度, 而不是用公式计算. 此外, 研究[3]表明 DStar 比其他的 SBFL 方法优秀, 因而也被选入本文的实验. 总体而言, 各个 SBFL 方法的思路都是根据测试用例的运行结果和覆盖语句情况计算各个语句的可疑度, 不同点是采用的计算公式不同. 更多细节信息请参考文献 [1]. 本文提出的 CBFL 方法是在 SBFL 方法的基础上, 利用来自参考程序的信息, 进一步精准定位错误代码, 降低软件维护成本.

另一类重要的错误定位技术是基于机器学习的技术. Wong 等人提出了应用反向神经网络模型[32]和 RBF 网络模型[33]实现自动化错误定位. Zhang 等人使用了深度神经网络[34], 并利用动态切片技术加强了此模型[35]. 此外, 他们还尝试了卷积神经网络, 显示出了比其他神经网络优越的性能[16]. 除了神经网络模型, 也有其他的机器学习方法被用于自动化错误定位, 例如 Briand 等人利用了决策树实现定位[36]. 相比于利用大量信息的神经网络模型, 本文的方法只利用程序运行信息和部分语法、语义信息, 具有易用性、可推广性.

除此之外, 还有其他类型的自动化错误定位技术. 基于切片的错误定位技术能够排除与错误无关的代码. Weiser 首先提出利用静态切片[8]减少需要检查的代码量, 从而协助程序员定位错误. 如果某一次运行失败是由错误的变量取值引起的, 静态切片技术将会获取所有能够潜在影响此变量值的代码. 然而, 静态切片技术并没有考虑运行时的信息, 因而有一些本应排除的代码未能被排除. 为了解决此问题, 动态切片的技术被提出[9]. 与静态切片方法不同, 动态切片方法能够在执行某个测试用例时鉴别出可影响某个位置的某个变量值的所有语句. 基于程序状态的技术根据运行时特定位置的变量取值来定位错误. Zeller 等人提出了 delta debugging[15], 能够对比程序的语句状态在测试用例通过与不通过时的差别. 此技术能够用测试用例通过时某个变量在某一点的动态取值替换测试用例不通过时该变量在相同点的取值, 并重新运行测试用例. 如果依然出现相同的报错, 那么认为该变量不是导致错误的原因. Zhang 等人提出了谓词替换方法[37], 能够修改错误的测试用例所运行的路径分支. 如果修改运行路径后测试用例运行通过, 那么对应于该分支的条件谓词是可疑的. Wang 等人也提出了一个类似的方法[38], 不同点在于他们修改的是条件分支执行的结果. 本文方法与切片技术的不同点在于, CBFL 是利用概率模型构建的排序方法, 能够计算出语句间错误概率的差异.

基于统计的技术能够通过比较测试用例执行时动态信息与运行结果来计算代码的错误概率. Liblit 等人提出

© 中国科学院软件研究所 http://www.jos.org.cn

Liblit05[14], 能够通过计算两个比例衡量谓词的可疑度: 某个谓词被覆盖时测试用例不通过的概率以及该谓词取值为 true 时测试用例不通过的概率. 如果后者大于前者, 则此谓词是可疑的. Liu 等人提出了类似的工具 SOBER[39], 此工具计算的是在通过/不通过的测试用例运行时某个谓词被覆盖的比例, 如果两个比例值相差较大, 则该谓词被认为可疑. Wong 等人提出 CBT 工具[40], 除了能够鉴别可疑的谓词, 还能够鉴别潜在错误的语句、函数和方法. 本文的方法与基于统计的技术有相似之处, 都是选择了与程序中的错误潜在相关的一些特征. 基于统计的技术统计了谓词取值等信息, 而本文利用了语句覆盖信息、语法语义信息等.

除了上述方法, 近年来研究者们也提出了整合不同定位方法的思路. 例如, TraPT[41]整合了 SBFL 和 MBFL 方法, 并利用支持向量机学习如何优化整合模型. DeepFL[16]抽取了代码的动态特征 (SBFL 与 MBFL) 和静态特征 (代码复杂性与测试用例的文本), 再利用深度学习模型整合所有特征. ProFL[42]是首个利用错误修复工具进行错误定位的方法. 首先, ProFL 采用基于变异的自动修复工具生成一系列补丁, 通过对各个补丁运行测试用例, 筛选高质量的补丁 (通过更多测试用例). 最后将高质量补丁所修改的位置视为可疑代码, 与 SBFL 方法计算的可疑度进行整合.

更多关于自动化错误定位技术的信息, 请参考文献 [1,43,44]. 文献 [1] 从错误定位的背景、理论假设、技术原理、实证研究、度量、应用等角度进行了全面介绍. 文献 [43] 主要对不同类型的错误定位方法进行概述. 文献 [44] 围绕基于频谱的错误定位方法，对其技术原理、特点、性能等方面进行了详细论述。

5 结束语

本文提出一种面向众包软件工程情景的基于竞争性实现的错误定位方法, 即 CBFL 方法. 该方法利用竞争性实现, 能够更准确地发现错误语句、排除正确语句. 我们从 Codeforces 网站上下载了 118 个错误程序进行实验, 结果显示当竞争性实现存在时, CBFL 方法明显比常用的基于频谱的错误定位方法优秀.

将来我们计划探究其他利用竞争性实现的方法, 来提高错误定位的效果. 并且, 我们也将在更多的模拟众包软件开发背景的数据集中做实验, 从而使实验结果更加可靠. 另一方面, 我们也将尝试把本文方法应用于众包软件测试之外的错误定位场景. 例如在代码复用、代码迁移、代码融合的场景中, 可以将旧代码作为参考程序, 用以测试新代码. 在这些场景中, 充分利用场景的特点进而建立参考程序与被测试程序中代码的对应关系将是一个有价值的研究方向.

References:

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

[2] Vessey I. Expertise in debugging computer programs: A process analysis. Int'l Journal of Man-machine Studies, 1985, 23(5): 459–494. [doi: 10.1016/S0020-7373(85)80054-7]

[3] Wong WE, Debroy V, Gao RZ, Li YH. The DStar method for effective software fault localization. IEEE Trans. on Reliability, 2014, 63(1): 290–308. [doi: 10.1109/TR.2013.2285319]

[4] Jiang JJ, Xiong YF, Zhang HY, Gao Q, Chen XQ. Shaping program repair space with existing patches and similar code. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 298–309. [doi: 10.1145/3213846.3213871]

[5] Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). San Francisco: IEEE, 2013. 802–811. [doi: 10.1109/ICSE.2013.6606626]

[6] Weimer W, Nguyen T, Goues CL, Forrest S. Automatically finding patches using genetic programming. In: Proc. of the 31st Int'l Conf. on Software Engineering. Vancouver: IEEE, 2009. 364–374. [doi: 10.1109/ICSE.2009.5070536]

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

[8] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984, SE-10(4): 352–357. [doi: 10.1109/TSE.1984.5010248]

[9] Agrawal H, Horgan JR. Dynamic program slicing. ACM SIGPLAN Notices, 1990, 25(6): 246–256. [doi: 10.1145/93548.93576]

[10] Abreu R, Zoeteweij P, van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Proc. of the 2007 Academic and Industrial Conf. Practice and Research Techniques—MUTATION. Windsor: IEEE, 2007. 89–98. [doi: 10.1109/TAIC.PART.2007.13]
[11] Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic internet services. In: Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks. Washington: IEEE, 2002. 595–604. [doi: 10.1109/DSN.2002.1029005]
[12] Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. Long Beach: ACM, 2005. 273–282. [doi: 10.1145/1101908.1101949]
[13] Abreu R, Zoeteweij P, van Gemund AJC. Spectrum-based multiple fault localization. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. Auckland: ACM, 2009. 88–99. [doi: 10.1109/ASE.2009.25]
[14] Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable statistical bug isolation. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Chicago: ACM, 2005. 15–26. [doi: 10.1145/1065010.1065014]
[15] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans. on Software Engineering, 2002, 28(2): 183–200. [doi: 10.1109/32.988498]
[16] Li X, Li W, Zhang YQ, Zhang LM. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 169–180. [doi: 10.1145/3293882.3330574]
[17] Denmat T, Ducassé M, Ridoux O. Data mining and cross-checking of execution traces: A re-interpretation of Jones, Harrold and Stasko test information. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. Long Beach: ACM, 2005. 396–399. [doi: 10.1145/1101908.1101979]
[18] Mayer W, Stumptner M. Model-based debugging using multiple abstract models. arXiv:cs/0309030, 2003.
[19] Zhang XF, Feng Y, Liu D, Chen ZY, Xu BW. Research progress of crowdsourced software testing. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 69–88 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5377.htm [doi: 10.13328/j.cnki.jos.005377]
[20] Estellés-Arolas E, González-Ladrón-De-Guevara F. Towards an integrated crowdsourcing definition. Journal of Information Science, 2012, 38(2): 189–200. [doi: 10.1177/0165551512437638]
[21] Mao K, Capra L, Harman M, Jia Y. A survey of the use of crowdsourcing in software engineering. Journal of Systems and Software, 2017, 126: 57–84. [doi: 10.1016/j.jss.2016.09.015]
[22] TopCoder. 2015. http://www.topcoder.com/
[23] AppStori. 2015. http://www.appstori.com/
[24] Mridha SK, Bhattacharyya M. Network based mechanisms for competitive crowdsourcing. In: Proc. of the 2018 ACM India Joint Int'l Conf. on Data Science and Management of Data. Goa: ACM, 2018. 318–321. [doi: 10.1145/3152494.3167979]
[25] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey. IEEE Trans. on Software Engineering, 2019, 45(1): 34–67. [doi: 10.1109/TSE.2017.2755013]
[26] Xuan JF, Martinez M, DeMarco F, Clément 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. [doi: 10.1109/TSE.2016.2560811]
[27] Debroy V, Wong WE, Xu XF, 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, 2010. 13–22. [doi: 10.1109/QSIC.2010.80]
[28] Alves E, Gligoric M, Jagannath V, d'Amorim M. Fault-localization using dynamic slicing and change impact analysis. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2011). Lawrence: IEEE, 2011. 520–523. [doi: 10.1109/ASE.2011.6100114]
[29] Hofer BG, Wotawa F. Spectrum enhanced dynamic slicing for better fault localization. In: Proc. of the 20th European Conf. on Artificial Intelligence. Montpellier: IOS Press, 2012. 420–425.
[30] Zhang ZY, Chan WK, Tse TH, Hu PF, Wang XM. Is non-parametric hypothesis testing model robust for statistical fault localization? Information and Software Technology, 2009, 51(11): 1573–1585. [doi: 10.1016/j.infsof.2009.06.013]
[31] Renieres M, Reiss SP. Fault localization with nearest neighbor queries. In: Proc. of the 18th IEEE Int'l Conf. on Automated Software Engineering. Montreal: IEEE, 2003. 30–39. [doi: 10.1109/ASE.2003.1240292]
[32] Wong WE, Qi Y. Bp neural network-based effective fault localization. Int'l Journal of Software Engineering and Knowledge Engineering, 2009, 19(4): 573–597. [doi: 10.1142/S021819400900426X]
[33] Wong WE, Debroy V, Golden R, Xu XF, Thuraisingham B. Effective software fault localization using an RBF neural network. IEEE Trans. on Reliability, 2012, 61(1): 149–169. [doi: 10.1109/TR.2011.2172031]
[34] Zheng W, Hu DS, Wang J. Fault localization analysis based on deep neural network. Mathematical Problems in Engineering, 2016, 2016:

© 中国科学院软件研究所 http://www.jos.org.cn

1820454. [doi: 10.1155/2016/1820454]
[35] Zhang Z, Lei Y, Tan QP, Mao XG, Zeng P, Chang X. Deep learning-based fault localization with contextual information. IEICE Trans. on Information and Systems, 2017, E100.D(12): 3027–3031. [doi: 10.1587/transinf.2017EDL8143]
[36] Briand LC, Labiche Y, Liu XT. Using machine learning to support debugging with tarantula. In: Proc. of the 18th IEEE Int'l Symp. on Software Reliability (ISSRE'07). Trollhattan: IEEE, 2007. 137–146 [doi: 10.1109/ISSRE.2007.31]
[37] Zhang XY, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proc. of the 28th Int'l Conf. on Software Engineering. Shanghai: ACM, 2006. 272–281. [doi: 10.1145/1134285.1134324]
[38] Wang T, Roychoudhury A. Automated path generation for software fault localization. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. Long Beach: ACM, 2005. 347–351. [doi: 10.1145/1101908.1101966]
[39] Liu C, Fei L, Yan XF, Han JW, Midkiff SP. Statistical debugging: A hypothesis testing-based approach. IEEE Trans. on Software Engineering, 2006, 32(10): 831–848. [doi: 10.1109/TSE.2006.105]
[40] Wong WE, Debroy V, Xu DX. Towards better fault localization: A crosstab-based statistical approach. IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 2012, 42(3): 378–396. [doi: 10.1109/TSMCC.2011.2118751]
[41] Li X, Zhang LM. Transforming programs and tests in tandem for fault localization. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 92. [doi: 10.1145/3133916]
[42] Lou YL, Ghanbari A, Li X, Zhang LM, Zhang HT, Hao D, Zhang L. Can automated program repair refine fault localization? A unified debugging approach. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2020. 75–87. [doi: 10.1145/3395363.3397351]
[43] Yu K, Lin MX. Advances in automatic fault localization techniques. Chinese Journal of Computers, 2011, 34(8): 1411–1422 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01411]
[44] 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]

附中文参考文献:

[19] 章晓芳, 冯洋, 刘頔, 陈振宇, 徐宝文. 众包软件测试技术研究进展. 软件学报, 2018, 29(1): 69–88. http://www.jos.org.cn/1000-9825/5377.htm [doi: 10.13328/j.cnki.jos.005377]
[43] 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. [doi: 10.3724/SP.J.1016.2011.01411]
[44] 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]

李乐平(1993－), 男, 博士, CCF 学生会员, 主要研究领域为软件缺陷定位与修复.

刘辉(1978－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件重构, 软件演化与维护, 软件测试.

张宇霞(1992－), 女, 博士, 助理教授, CCF 专业会员, 主要研究领域为软件仓库挖掘, 开源软件生态系统.

© 中国科学院软件研究所 http://www.jos.org.cn