软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(8):1993−2007 [doi: 10.13328/j.cnki.jos.004858]

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

基于二次定位策略的软件故障定位*

宗芳芳, 黄鸿云, 丁佐华

(浙江理工大学 信息学院,浙江 杭州 310018)

通讯作者: 丁佐华, E-mail: zouhuading@hotmail.com

摘 要: 故障定位是软件调试过程中耗力和耗时的活动之一,尤其是对规模大和复杂性高的软件.目前的一些定位技术可分为两类:基于组件和基于语句.前者太粗,不能准确地定位到地方;后者太细,运算复杂度过大.提出一种新技术,称为二次定位策略(double-times-locating,简称DTL),来定位故障:第1次定位,从程序中抽象出函数调用图,再从函数调用轨迹中建立程序谱,最后用基于模型的诊断(model-based diagnosis,简称MBD)对可能含有故障的函数进行排序;第2次定位,利用DStar定位函数中故障的代码行.实验结果表明,该技术比目前基于统计的方法更有效.

关键词: 故障定位;函数调用图;程序频谱;模型诊断

中图法分类号: TP311

中文引用格式: 宗芳芳,黄鸿云,丁佐华.基于二次定位策略的软件故障定位.软件学报,2016,27(8):1993−2007. http://www.jos.org.cn/1000-9825/4858.htm

英文引用格式: Zong FF, Huang HY, Ding ZH. Software fault location based on double-times-locating strategy. Ruan Jian Xue Bao/Journal of Software, 2016,27(8):1993−2007 (in Chinese). http://www.jos.org.cn/1000-9825/4858.htm

Software Fault Location Based on Double-Times-Locating Strategy

ZONG Fang-Fang, HUANG Hong-Yun, DING Zuo-Hua

(School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou 310018, China)

Abstract: Fault localization is a physical and time-consuming activity in the debugging process, especially for the software with large size and high complexity. Existing techniques to locate faults can be classified into two categories: component based and statement based. The former is too coarse to locate the accurate place, while the latter is too fine to contain the computation complexity. This paper proposes a new technique, called double-times-locating (DTL) strategy, to locate software faults. For the first time locating, it abstracts function call graph from the code, builds program spectrum to abstract function traces, and then uses model-based diagnosis (MBD) to sort with probability possible functions candidates that have faults. For the second time locating, it uses DStar to locate faults in the functions. Experimental results show that the proposed technique is more effective than the existing statistics based methods.

Key words: fault localization; function call graph; program spectrum; model diagnosis

调试不仅仅是费时的、繁琐的、昂贵的,而且也是极易出错的[1−4].在各种调试活动中,软件故障定位(简称为故障定位)已被确定为最昂贵的活动之一[5].能够快速地定位故障的根源,可以减少测试阶段所花费的时间.故障定位技术[6−19]的发展为未来的程序修复铺平了道路.

常用的故障定位技术的基本思想可解释如下:首先定出最有可能的故障部分(例如程序语句和谓词),并对它们进行排序;然后,根据排列的顺序进行逐一检查,直到找到故障的部分为止.一种好的定位技术应当是把有故障的部分排到更靠前,这样,故障就能被更快地检查出来.理想情况下,有故障的部分排在第一位,不需要程序员的进一步分析,故障完全可以自动定位.

* 基金项目: 国家自然科学基金(61210004, 61170015)
Foundation item: National Natural Science Foundation of China (61210004, 61170015)
收稿时间: 2015-03-19; 修改时间: 2015-03-28, 2015-04-04; 采用时间: 2015-05-19

目前,主要的故障定位方法有:

- 基于频谱的故障的定位(spectrum-based fault localization)[20].该方法通过抽象程序的轨迹,使得软件的组件与程序的故障关联起来.
- 相似程序谱的故障定位(nearest neighbor queries)[13].该方法假设存在一个失败的程序谱和很多成功的程序谱,然后根据距离准则挑选出一个与失败运行最相似的成功运行的程序谱,进而比较两个程序谱的不同之处,以分离软件故障.虽然从复杂性的角度来看,统计方法非常有吸引力,但是不能推理解释多错误导致的失败.但实际上,大多数的程序是有多错误的.
- SOBER 的故障定位方法[21].该方法对谓词在成功执行和失败执行中的取值模式进行建模,然后基于统计学中假设检验的原理,量化每个谓词的故障相关性,按照谓词怀疑度(the suspicion degree)的大小审查源代码,发现故障的位置.然而,该方法需要足够多的测试用例.
- 基于模型的诊断(model-based diagnosis,简称 MBD)[16,17,20,22−28].该方法通过逻辑推理和构建行为命题模式来推导软件故障.该推理方法的主要缺点是:(1) 模型生成通常是静态的分析,因为无法捕捉的动态数据依赖或者条件控制流;(2) 生成诊断候选集的代价是呈指数增长的,通常禁止使用超过几百行程序[24].

然而,基于频谱的故障定位较为常用,可分为两类:基于语句和基于组件.基于语句的故障定位工作有文献[11,20,22,23,29,30],基于组件的故障定位工作有文献[16,17].

基于语句的故障定位需要定位到代码行,就意味着需要大量的工作去掉与故障的代码行不相关的语句,其粒度过于精细.基于组件的故障定位组件范围很大(如服务器),程序员不能准确地定位具体的位置,其粒度又显得过于粗糙.为了解决这一问题,在前人工作的基础上,我们提出一种新的故障定位方法,即二次定位(double-times-locating,简称 DTL).第 1 次函数定位,第 2 次语句定位.下面给出我们的方法框架图.图 1 包含两个部分:第 1 部分叫做函数定位,我们把程序中的函数作为组件生成候选集,并根据算法计算出候选集的怀疑度进行排序,并将怀疑度较大的候选集放到故障定位的集合中,准备进行语句定位;第 2 部分叫做语句定位,我们把候选集中的函数所对应的语句作为组件,根据算法计算出语句的怀疑度进行排序,逐个检查怀疑度较大的语句,直到定位故障语句为止.

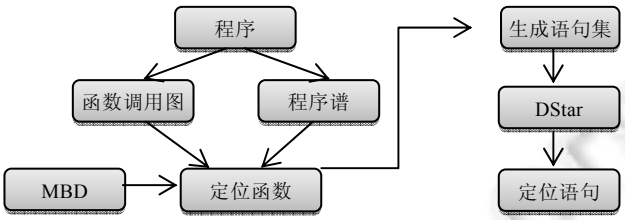

Fig.1 Frame of our method

图 1 我们的方法框架图

本文第 1 节描述函数定位的基本原理.第 2 节描述语句定位的基本原理.第 3 节用我们的方法对一个例子进行故障定位并分析.第 4 节给出实验结果及分析.第 5 节讨论和分析相关工作.第 6 节是全文的总结.

1 一次定位——函数定位

我们利用工具从程序中生成函数调用图,再从函数调用轨迹中建立程序谱,将软件的组件与程序的故障关联起来,再利用 MBD 的逻辑推理和构建行为命题模式来对函数进行初步定位.我们分别详细地介绍函数调用图的生成以及函数定位.

1.1 函数调用图

Doxygen 是一种开源跨平台的,可以从一套归档源文件开始,生成 HTML 格式的在线类浏览器,或离线的

LATEX,RTF参考手册.Graph Visualization Software(Graphviz)是一个由AT&T实验室启动的开源工具包,用于绘制DOT语言脚本描述的图形,它也提供了供其他软件使用的函数库.我们使用文档生成工具Doxygen和图像生成工具Graphviz生成函数调用图,记为 G.

生成的函数调用图是有向无回路的拓扑结构,我们将函数之间的相互关系用拓扑结构进行描述.图 2 是最常见的函数调用图.我们知道,函数之间常用的调用关系包括顺序、循环、选择方式.这里我们假设,如果函数是错误的,那么经过它的测试用例中必然是有错的.现在我们分析一下顺序调用和循环调用的情况,比如,函数 $F3$ 调用函数 $F8$ 时,假如函数 $F8$ 有错,函数 $F3$ 的测试用例中,如果经过函数 $F8$ 的测试用例是有错的,我们就可以把函数 $F8$ 放进函数的候选集中.选择调用的情况,比如,函数 $F1$ 调用函数 $F4$ 和函数 $F5$ 时,如果只有函数 $F4$ 有错,那么函数 $F1$ 的错误测试用例的个数不小于函数 $F4$ 的错误测试用例的个数,而且经过函数 $F4$ 的测试用例是有错的.此时,我们就可以把函数 $F4$ 放进函数的候选集中.我们把具体实现的算法放在第 1 节的算法 1 中实现.

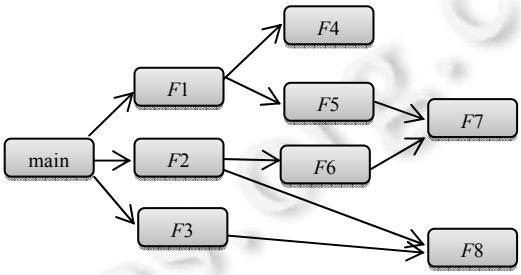

Fig.2 Common function call graph

图 2 常见的函数调用图

1.2 程序谱

假定一个软件系统由 M 个组件 C_j 组成,其中 $j\in\{1,...,M\}$,并且存在多个故障,记为 C.诊断报告 $D=\langle\ldots,d_k,\ldots\rangle$ 是一个由诊断候选集 d_k 构成的有序集合.程序谱是由参与了系统的特定动态行为的组件构成的集合,它统计了系统执行的 N 个测试用例的结果(成功/失败).我们的行为模型是通过一组涉及计算的组件构成,并且不需要详细地描述行为的具体信息.我们把程序成功/失败的信息输入到程序谱.通过实例化程序中每一个组件,形成矩阵 A.在 A 中的元素 a_{ij} 表示组件 C_j 是否参与了第 i 个测试用例的执行:若是,则 a_{ij}=1;若不是,则 a_{ij}=0.测试用例的结果(成功/失败)用向量 e 表示.向量 e 中的元素 e_i 表示第 i 个测试用例的执行结果:若成功,则 e_i=0;若失败,则 e_i=1.因此,矩阵(A,e)就是我们需要的程序谱.

1.3 定位函数

基于模型的推理方法得到的诊断报告描述了对多个故障候选集 d_k 的排序,例如 $D=\{\langle 2,3\rangle,\langle 1\rangle\}$,这意味着组件 C_2 和 C_3 有故障,或组件 C_1 有故障.正如上述描述的基于模型的推理方法,我们将这一方法应用在我们具体的程序模型中.在前人工作的基础上[17,20,23],本文提出以函数作为组件建立程序模型.本文在逻辑命题方面将每个组件 C_j 定义为 $h_j\rightarrow(ok_{inp_j}\rightarrow ok_{out_j})$,其中,$h_j$,$ok_{inp_j}$,$ok_{inp_j}$ 是布尔类型,分别代表组件的正确值、组件的输入变量和输出变量.这个模型表达了特定的行为:假如组件正确并且输入变量的值是正确的,那么其输出也是正确的.为了更清晰地说明这类特定行为,该模型没有指定故障的具体行为.即使组件是错误的和/或输入值不正确,还是可能导致正确的输出.因此,一个程序通过并不意味着经过的组件的正确性.在函数定位中,我们首先把程序中的函数作为组件生成候选集,在算法 1 中,把函数调用图 G 和程序谱(A,e)作为输入,由第 4 行~第 22 行生成函数候选集 d_k;接下来,在函数定位过程中,计算出候选集的怀疑度进行排序.现在假定候选集是相互独立的,我们利用贝叶斯定理计算候选集 D 的条件概率:

$$P(d_k\mid e)=\frac{P(e\mid d_k)P(d_k)}{P(e)}.$$

其中,$P(e)$是一个标准化常数,由所有的 d_k 定义,不需要直接计算. $P(d_k)=p^{|d_k|}\times(1-p)^{M-|d_k|}$,其中,$M$ 是组件的个数,$|d_k|$是候选集 d_k 包含的组件个数.在本文中,我们假定 p=0.01,即程序本身大部分是正确的.

因为每次执行是独立的,因此,

$$P(e\mid d_k)=\prod_{i=1}^{N}P(e_i\mid d_k).$$

其中,$P(e_i|d_k)$的定义如下:

$$P(e_i\mid d_k)=\begin{cases}1, & \text{if } d_k\wedge e_i \text{ are inconsistent}\\ 0, & \text{if } d_k \text{ is unique to } e_i\\ \varepsilon, & \text{otherwise}\end{cases}.$$

关于ε的定义有很多种,本文的定义为

$$\varepsilon=\begin{cases}\prod\limits_{j\varepsilon d_k\cap a_{ij}=1}h_j, & e_i=0\\ 1-\prod\limits_{j\varepsilon d_k\cap a_{ij}=1}h_j, & e_i=1\end{cases}.$$

其中,$h_j\in[0,1]$表示组件 j 正常工作的概率.

我们可以得出 $P(e|d_k)$关于 h_j 的表达式,通过最大似然估计法计算 $H=\max\limits_{H}(P(e\mid d_k))$, 其中,$H=\{h_j\in[0,1]|j\in d_k\}$.最后,将得到的 $P(e|d_k)$,$P(d_k)$带入公式,计算出 $P(d_k|e)$进行排序,将排序较高的候选集进行语句定位.

算法 1 中的描述包括两个主要部分:

- 第 1 部分,我们把程序中的函数作为组件生成的函数,调用图 G 和程序谱(A,e)作为输入,生成函数候选集 $D=\langle\ldots,d_k,\ldots\rangle$:第 1 步,我们计算出$|L|=\{A_{k^*}\mid e_k=1\}$,并且把那些函数满足没有调用其他函数、相关系数最大且 $n(v_j)=|L|$直接放入候选集 d_k 中,并且更新函数集 R 和函数调用图 G;第 2 步,对于不满足上述条件的函数,存在函数 $n_{cs}(v_j)/n_{cf}(v_j)=0$.我们重复第 1 步,直到 R 中没有满足条件的函数为止.
- 第 2 部分,我们通过计算第 1 部分生成的函数候选集 $D=\langle\ldots,d_k,\ldots\rangle$中每个 d_k 的 $P(d_k|e)$值,生成有序的诊断报告 D'.

下面我们对算法 1 的时间和空间复杂度进行计算.算法 1 第 1 部分的时间复杂度就是对每个函数(M)相似度的计算,即(A,e)的每一行,复杂度为 $O(N\cdot M)$;第 2 部分诊断报告的时间复杂度为 $O(M\cdot\log M)$.所以算法 1 的时间复杂度为 $O(M\cdot(N+\log M))$.算法 1 的空间复杂度就是我们的方法需要保存函数的候选集 C,空间复杂度为 $O(2^{|C|}\cdot M)$.

算法 1. 诊断算法(一次定位算法).

输入:矩阵(A,e),函数调用图 G.

输出:诊断报告 D.

1. $\gamma\leftarrow\varepsilon$
2. $L\leftarrow\{A_{k^*}\mid e_k=1\}$
3. $D\leftarrow\varnothing$
4. $G\leftarrow TOPOLOGICAL\text{-}SORT(G)$
5. $R\leftarrow SORT(H,A,e)$
6. while $j<G.length$ do
7. if $P(v_j)=\text{Max}(P(R))\wedge n_{cf}(v_j)=|L|$ then
8. $Push(D,j)$
9. $A\leftarrow EXTRACT(A,j)$
10. $G\leftarrow EXTRACT(G,v_j)$
11. $R\leftarrow R\backslash\{j\}$

12. end if
13. end while
14. while $R\neq\varnothing\wedge P(v_j)\neq 0$
15. $j'\leftarrow Pop(R)$
16. while $n_{cs}(v_j)/n_{cf}(v_j)=0$
17. $Push(D,j')$
18. $A\leftarrow EXTRACT(A,j')$
19. $G\leftarrow EXTRACT(G,v_{j'})$
20. $R\leftarrow R\backslash\{j\}$
21. end while
22. end while
23. for all $d_k\in D$
24. $R\leftarrow SORT(A,e,d_k)$
25. $i\leftarrow 0$
26. $P(d_k)^i\leftarrow 0$
27. while $|P(d_k)^i-P(d_k)^{i-1}|<\varepsilon$
28. for all $j\in d_k$
29. $h_j\leftarrow h_j+\gamma*\nabla R(h_j)$
30. end for
31. $P(d_k)^i\leftarrow FUNCTION(R,\forall_{j\in d_k}h_j)$
32. $i\leftarrow i+1$
33. end while
34. end for
35. Return 诊断报告 D'

2 二次定位——语句定位

我们利用 Wong 等人[30]提出的一种能够自动引导开发人员定位程序故障的方法,即 DStar.该方法通过跨越 21 个不同的项目进行评估,DStar 显示比同时代的其他 16 种故障定位技术更有效(12 similarity coefficient-based, Tarantula,Ochiai,H3b and H3c techniques).DStar 的优越性并不限于定位只包含一个故障的程序,可以延伸到多个故障的程序,此外,无需任何对程序结构或语义先验信息.本文语句定位采用了当前基于语句效率较高的方法 DStar.根据经验[11,20,23],我们知道:

1) 一个语句的怀疑度正比于它本身覆盖失败的测试用例的次数.那么,这个语句被失败的测试执行次数越多,越值得怀疑.
2) 一个语句的怀疑度反比于它本身覆盖成功的测试用例的次数.那么,这个语句被成功的测试执行次数越少,越值得怀疑.
3) 一个语句的怀疑度反比于它本身未覆盖失败的测试用例的次数.那么,测试失败的情况下不涉及这个特定的语句,则该语句怀疑度小.

其中,情形 1)是最健全的,并应该有一个更高的权重.我们重视失败的测试覆盖语句,而不是成功的测试用例覆盖的语句和失败的测试用例未覆盖的语句.Wong 等人的实验结果表明,D^3 找到第 1 个故障的效果显著地高于 Tarantula,D^2,Mountford,Ochiai.因此,本文用到的 DStar 的公式:

$$P(s)=\frac{[N_{cf}(s)]^3}{N_{uf}(s)+N_{cs}(s)}.$$

符号含义如下:

- $N_{cf}(s)$:通过第 s 行的结果为 fail 的用例总数.
- $N_{uf}(s)$:不通过第 s 行的结果为 fail 的用例总数.
- $N_{cs}(s)$:通过第 s 行的结果为 pass 的用例总数.

算法 2 中的描述包括 3 个主要部分:第 1 部分将一次定位的函数所在的行用数组 B 进行存储;第 2 部分利用 DStar 方法将 B 数组的所有行求出怀疑度;第 3 部分将怀疑度进行排序,从而定位错误的语句行.

算法 2. 二次定位算法.

输入:诊断报告 D.

输出:语句怀疑度排序.

1. For all $d_k \in D$ do
2. $B \leftarrow Pop(R)$
3. end for
4. For all $b_j \in B$ do
5. $P(b_j) \leftarrow DStar(b_j)$
6. end for
7. Return $SORT(P(b_j),B)$

算法 2 的时间复杂度是对候选集 C 中函数所在的语句 $\sum C_j$ 的怀疑度进行排序 $O\left(\sum C_j \cdot N\right)$. 因此,算法 1 和算法 2 的时间复杂度总和为 $O\left(N \cdot \left(M + \sum C_j\right) + M \cdot \log M\right)$. .统计技术(Tarantula 方法[11]、Ochiai 方法[23]和 DStar 方法[30])每个语句 $\sum M_j$ 的相似度计算的复杂度为 $O\left(N \cdot \sum M_j\right)$, 诊断报告的复杂度 $O\left(\sum M_j \cdot \log \sum M_j\right)$. 因此,统计技术时间复杂度总和为 $O\left(N \cdot \sum M_j + \sum M_j \cdot \log \sum M_j\right)$. 由时间复杂度可知,程序的语句行与函数行的数量级差别越大,我们的方法定位的速度就越快.因此,我们的方法在时间复杂度上更优于统计技术.

对于空间复杂度,统计方法需要对所有的语句 $\sum M_j$ 相似度计算进行保存(N_{cf},N_{uf},N_{cs},N_{us}).因此,空间复杂度为 $O\left(\sum M_j\right)$. 我们的方法需要的空间复杂度为 $O(2^{|C|} \cdot M)$.从数量级别表明,我们的方法的空间复杂度呈指数增长,这就需要我们更准确地定位函数的候选集,那么空间复杂度就会更小.

3 一个例子

现在我们通过一个小例子来描述我们的方法.表 1 中有 4 个测试用例(x=–5,1,3,0),其中,黑点表示这一行被对应的测试用例运行时所覆盖,空格则表示这一行在测试用例运行时未覆盖.现在给出 4 个测试用例的测试结果.两个故障语句如下:正确程序 $f2(n)\{\ldots,n\%i==0,\ldots\}$,$f5(n)\{\ldots,i<n+1,\ldots\}$,故障程序 $f2(n)\{\ldots,n\%i!=0,\ldots\}$,$f5(n)\{\ldots,i<n,\ldots\}$.通过图 3 描述的小例子的函数调用图,我们知道 $f1$~$f5$ 之间不存在函数之间的调用.

Table 1 An example

表 1 例子

function(){double *y*, *s*;		Test case 1	Test case 2	Test case 3	Test case 4	*T*	*O*	*D*
1	*read*("*Enter x*=",*x*);							
2	if (*x*>0){	•	•	•	•	0.5	0.707 106 78	4
3	*y*=*f*1(*x*);		•	•		0.5	0.5	0.5
4	if (*y*–1>0)		•	•		0.5	0.5	0.5
5	*s*=*f*2(*y*); //debug			•		1	0.707 106 78	1
6	else							
7	*s*=*f*3(*y*);}		•			0	0	0
8	else							
9	*y*=*f*4(*x*);	•			•	0.5	0.5	0.5
10	if (*y*>0)	•			•	0.5	0.5	0.5
11	*s*=*f*5(*y*); //debug	•				1	0.707 106 78	1
12	else							

Table 1 An example (Contiuned)

表 1 例子(续)

	function(){double *y*, *s*;	Test case 1	Test case 2	Test case 3	Test case 4	*T*	*O*	*D*
13	*s*=*f*6(*y*);				●	0	0	0
14	*Print*("*s*:"+ *s*)}	●	●	●	●	0.5	0.707 106 78	4
15	*f*1(*n*){							
16	if (*n*==1)		●	●		0.5	0.5	0.5
17	return 1;		●	●		0.5	0.5	0.5
18	return *n*+*f*1(*n*−1);}		●	●		0.5	0.5	0.5
19	*f*2(*n*){							
20	for (int *i*=2; *i*<=*sqrt*(*n*−1); *i*++){			●		1	0.707 106 78	1
21	if (*n*%*i*!=0) //debug *n*%*i*!=0→*n*%*i*==0			●		1	0.707 106 78	1
22	return 0;}}			●		1	0.707 106 78	1
23	return 1;}							
24	*f*3(*n*){							
25	if (*n*==1){		●			0	0	0
26	return 1;}		●			0	0	0
27	return *n***f*3(*n*−1);}		●			0	0	0
28	*f*4(*x*){							
29	return *x***x*;}	●			●	0.5	0.5	0.5
30	*f*5(*n*){							
31	int *sum*=0;	●				1	0.707 106 78	1
32	if (*n*>0){	●				1	0.707 106 78	1
33	for (int *i*=1; *i*<*n*; *i*++){ //debug *n*→*n*+1	●				1	0.707 106 78	1
34	*sum*+=*i*;}}	●				1	0.707 106 78	1
35	return *sum*;}	●				1	0.707 106 78	1
36	*f*6(*x*){							
37	return 6**x*;}				●	0	0	0
	Pass/Fail Status	F	P	F	P			

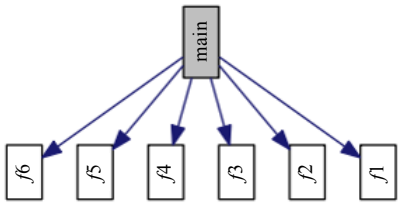

Fig.3 Function call graph of the example

图 3 例子的函数调用图

根据我们的算法 1 得到表 2 函数与测试用例关系表.

Table 2 Relational tables of functions and test cases

表 2 函数与测试用例关系表

	*F*1	*F*2	*F*3	*F*4	*F*5	*F*6
测试用例 1	0	0	0	1	1	0
测试用例 2	1	0	1	0	0	0
测试用例 3	1	1	0	0	0	0
测试用例 4	0	0	0	1	0	1
相关系数 *T*	0.5	1	0	0.5	1	0
相关系数 *O*	0.5	0.707	0	0.5	0.707	0
相关系数 *D*	0.5	1	0	0.5	1	0

计算出函数的候选集 d_1=⟨1,4⟩,d_2=⟨2,5⟩;然后最大化 $H=\max_H(P(e\mid d_k))$,得到 h_1=h_4=0.5,h_2=h_5=0;最后,由贝叶斯定理计算:

$$P(d_1|e)=h_1\times h_4\times(1-h_1)\times(1-h_4),$$
$$P(d_2|e)=(1-h_2)\times(1-h_5).$$

$P(d_1|e)$=0.9412,$P(d_2|e)$=0.0588,因此,函数定位的函数是 $f2$ 和 $f5$.我们利用算法 2,首先将一次定位的函数 $f2$ 和 $f5$ 中所有代码行对应的行号 b_j=20~23,31~35 用数组 B 进行存储;然后,利用 DStar 方法求出 $P(b_j=23)=0$, $P(b_j\neq23)$=0.70710678;最后,我们成功地找到第 1 个错误行.最好的情况是检查 1 行,最坏的情况是检查 8 行.为了与实验统一,我们平均检查的行数是 4.5,小例子可执行的总代码行为 27,那么成功地找到错误行需要检查总代码行的比值为 $P(Our\_Method)$=4.5/27=0.167.我们计算得到 Tarantula,Ochiai,DStar 方法成功地找到错误行需要检查总代码行的比值分别为:$P(Tarantula)$=5.5/27=0.204,$P(Ochiai)$=6.5/27=0.241,$P(DStar)$=6.5/27=0.241.

4 实验分析

4.1 实验建立

本文使用了经典的 Siemens 数据集进行实验,以证明我们方法的有效性,见表 3.第 1 列是实验对象的名称,第 2 列是每个实验对象的错误版本个数,第 3 列是每个实验对象主体的行数,第 4 列是测试用例的数量,第 5 列是实验对象的版本说明.其中,Siemens 数据集可以从 SIR(http://sir.unl.edu/content/sir.html)网站上免费获得.为了证明该方法的有效性,我们选择与一些经典的基于频谱的故障定位方法进行比较.

Table 3 Details of the Siemens data set

表 3 Siemens 数据集详细信息

Program	Faulty version	LOC	Test cases	Description
print_tokens	7	472	405 6	Lexical analyzer(词法分析器)
print_tokens2	10	399	407 1	Lexical analyzer(词法分析器)
Replace	32	512	554 2	Pattern replacement(模式更换)
Schedule	9	292	265 0	Priority scheduler(优先级调度)
Schedule2	10	301	268 0	Priority scheduler(优先级调度)
Tcas	41	141	157 8	Altitude separation(高度分离)
tot_info	23	440	105 4	Information measure(信息测度)

4.2 经典的基于频谱的故障定位方法介绍

我们现在总结一下基于频谱的故障定位方法.Jones 等人提出了一种基于代码覆盖的错误定位方法[11,31],这种方法首先在程序运行时,在成功次数和失败次数的基础上计算语句怀疑度;其次,根据怀疑度的高低对语句进行排序;然后,调试人员根据排序表依次检查,直到他们找到错误为止.随后,一些研究人员在 Tarantula 公式的基础上,对怀疑度公式进行了改进.Abreu 提出了 Ochiai 错误定位[23],定义语句的怀疑度.而 Ochiai 是来自于分子生物学中的相似系数.在一个程序中,Kulczynski 系数可以定位单个错误或多个错误.在此基础上,Wong 等人提出了 DStar(D^*)[30].Abreu 等人提出的 Barinel[20]以语句为粒度,应用模型诊断的方法及贝叶斯推理的方法进行错误定位.为了证明我们方法的有效性,本文使用 Tarantula 方法、Ochiai 方法、DStar 方法和 Barinel 方法与我们提出的方法进行了比较,现在我们分别给出 Tarantula 方法、Ochiai 方法、DStar 方法的定义.

- Tarantula:

$$s(j)=\frac{\dfrac{N_{cf}(j)}{N_{cf}(j)+N_{uf}(j)}}{\dfrac{N_{cs}(j)}{N_{cs}(j)+N_{us}(j)}+\dfrac{N_{cf}(j)}{N_{cf}(j)+N_{uf}(j)}}.$$

- Ochiai:

$$s(j)=\frac{N_{cf}(j)}{sqrt((N_{cf}(j)+N_{uf}(j))\times(N_{cf}(j)+N_{cs}(j)))}.$$

- DStar:

$$s(j)=\frac{[N_{cf}(j)]^3}{N_{uf}(j)+N_{cs}(j)}.$$

符号的含义:

- $N_{cf}(j)$:通过第 j 行的结果为 *fail* 的用例总数.
- $N_{uf}(j)$:不通过第 j 行的结果为 *fail* 的用例总数.
- $N_{cs}(j)$:通过第 j 行的结果为 *pass* 的用例总数.
- $N_{us}(j)$:不通过第 j 行的结果为 *pass* 的用例总数.

4.3 实验结果

我们通过对 Siemens 数据集的实验,得到一次定位的函数集以及在二次定位中发生故障时需要检查代码行占总行数的比值,作为故障定位的有效性值[1].在表 4 中,

- 一次定位的“无”代表故障没有出现在函数中.
- 二次定位的“0”代表 5 种情况:第 1 种是故障出现在头文件中;第 2 种是故障的原因是缺少代码;第 3 种是没有故障的测试用例;第 4 种是故障出现在函数声明上;第 5 种是故障的代码行是程序未编译.

Table 4 Function sets of the first time fault locating and the effectiveness values of the second time fault locating

表 4 一次故障定位的函数集及二次定位的有效性值

print_tokens	一次定位	二次定位	print_tokens	一次定位	二次定位
V1	get_token()	0.043 59	V5	get_token()	0.020 512 821
V2	get_token()	0.020 513	V6	无	0
V3	get_token()	0.005 154 639	V7	numeric_case()	0.025 641
V4	无	0			
print_tokens2	一次定位	二次定位	print_tokens2	一次定位	二次定位
V1	get_tokens()	0	V6	is_num_contant	0.337 5
V2	get_tokens()	0.455	V7	is_token_end	0.62
V3	get_tokens()	0	V8	is_token_end	0.03
V4	get_tokens()	0.417 5	V9	is_token_end	0.02
V5	is_str_contant	0.397 5	V10	is_str_contant	0
replace	一次定位	二次定位	replace	一次定位	二次定位
V1	dodash	0.014 344	V17	esc	0.004 098
V2	dodash	0.010 373	V18	omatch	0.008 197
V3	subline	0.131 148	V19	get_line	0
V4	subline	0.100 41	V20	esc	0.004 098
V5	dodash	0.006 148	V21	get_line,addstr、makepat	0.192 623
V6	locate	0.024 59	V22	getccl	0.069 672
V7	in_set2	0.006 148	V23	esc	0.014 344
V8	in_set2	0.034 836	V24	omatch	0.006 148
V9	dodash	0.036 885	V25	omatch	0.012 295
V10	dodash	0.049 18	V26	omatch	0.004 098
V11	dodash	0.036 885	V27	in_set2	0.502 049 18
V12	无	0	V28	in_set2	0.006 148
V13	subline	0.145 492	V29	in_set2	0.006 148
V14	omatch	0.008 197	V30	in_set2	0.006 148
V15	makepat	0.151 639	V31	omatch	0.008 197
V16	in_set2	0.026 639	V32	dodash	0.502 049 18
schedule	一次定位	二次定位	schedule	一次定位	二次定位
V1	find_nth	0.503 289 474	V6	find_nth	0.503 289 474
V2	unlock_process	0.503 289 474	V7	upgrade_process	0.503 289 474
V3	upgrade	0.013 158	V8	upgrade_process	0
V4	upgrade	0.016 447	V9	main	0.503 289 474
V5	upgrade_process	0.503 311 258			
schedule2	一次定位	二次定位	schedule2	一次定位	二次定位
V1	new_job	0	V6	flash	0.624 031
V2	get_process	0	V7	get_process	0
V3	get_process	0	V8	put_end	0
V4	get_command	0	V9	finish	0
V5	put_end	0.453 846	V10	get_command	0.193 798

Table 4 Function sets of the 1st time fault locating and the effectiveness values of the 2nd time fault locating (Contiuned)

表 4 一次故障定位的函数集及二次定位的有效性值(续)

print_tokens	一次定位	二次定位	print_tokens	一次定位	二次定位
tcas	一次定位	二次定位	tcas	一次定位	二次定位
V1	Non_Crossing_Biased_Climb	0.023 077	V22	Non_Crossing_Biased_Climb	0.2
V2	Inhibit_Biased_Climb	0.061 538	V23	Non_Crossing_Biased_Descend	0.207 692
V3	alt_sep_test	0.546 154	V24	Non_Crossing_Biased_Descend	0.2
V4	Non_Crossing_Biased_Climb	0.015 385	V25	Non_Crossing_Biased_Descend	0.015 385
V5	alt_sep_test	0.5	V26	alt_sep_test	0.530 769
V6	Own_Below_Threat	0.084 615	V27	alt_sep_test	0.5
V7	initialize	0.346 154	V28	Inhibit_Biased_Climb	0.084 615
V8	initialize	0.438 462	V29	Inhibit_Biased_Climb	0.130 769
V9	Non_Crossing_Biased_Descend	0	V30	Inhibit_Biased_Climb	0.130 769
V10	Own_Below_Threat, Own_Above_Threat	0.053 846	V31	Non_Crossing_Biased_Climb	0
V11	Own_Below_Threat, Own_Above_Threat	0.053 846	V32	Non_Crossing_Biased_Descend	0
V12	alt_sep_test	0.515 385	V33	initialize	0.392 308
V13	无	0	V34	alt_sep_test	0.515 385
V14	无	0	V35	Inhibit_Biased_Climb	0.084 615
V15	无	0	V36	无	0
V16	initialize	0.3	V37	ALIM	0.023 077
V17	initialize	0.269 231	V38	无	0
V18	initialize	0.238 462	V39	Non_Crossing_Biased_Descend	0.023 077
V19	initialize	0.269 231	V40	Non_Crossing_Biased_Climb	0.023 077
V20	Non_Crossing_Biased_Climb	0.2	V41	Non_Crossing_Biased_Climb	0.023 077
V21	Non_Crossing_Biased_Climb	0.2			
tot_info	一次定位	二次定位	tot_info	一次定位	二次定位
V1	InfoTbl	0	V13	InfoTbl	0.097 561
V2	main	0.195 122	V14	main	0.056 911
V3	main	0.040 65	V15	gser	0.052 846
V4	gct	0.032 52	V16	main	0.040 65
V5	main	0.121 951	V17	gct	0.077 236
V6	无	0	V18	InfoTbl	0.105 691
V7	InfoTbl	0.081 301	V19	无	0
V8	gser	0.024 39	V20	InfoTbl	0.130 081
V9	main	0.117 886	V21	无	0
V10	无	0	V22	InfoTbl	0.032 52
V11	gser	0.052 846	V23	gct	0.073 171
V12	Lgamma	0.093 496			

为了更清晰地表示故障定位的有效性值,我们给出一次定位函数集有效值的曲线图.在图 4 中,我们把每个版本定位出的故障的函数占该版本总函数的百分比作为一次定位的有效值,并计算不同区间内能够检查出的故障函数集.X 轴表示检测的函数的百分比;Y 轴上给出定位故障函数的百分比;图上的节点表示在检查不同的函数区间能够定位出故障的百分比.在一次定位实验中,我们的方法检查 60%的函数就可以定位大约 86%的故障函数.

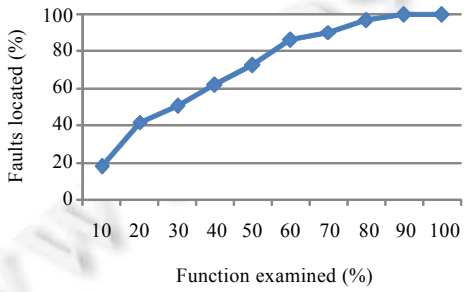

Fig.4 Valid values' graph of function sets of the first time fault locating

图 4 一次故障定位函数集有效值的曲线图

接下来,我们定位的函数以及定位出该故障函数占该版本总函数的比值.在表 5 中,

- 一次定位的“无”代表故障没有出现在函数中.
- 占总函数的比值为“0 代表两种情况:第 1 种是故障没有出现在函数中;第 2 种是该函数没有故障的测试用例,无法定位故障.

Table 5 Function sets of the first time fault locating and its effectiveness values

表 5 一次故障定位的函数集及相应的有效性值

print_tokens	一次定位	占总函数的比值	print_tokens	一次定位	占总函数的比值
V1	get_token()	0.388 889	V5	get_token()	0.333 333
V2	get_token()	0.222 222	V6	无	0
V3	get_token()	0.222 222	V7	numeric_case()	0.055 556
V4	无	0			
print_tokens2	一次定位	占总函数的比值	print_tokens2	一次定位	占总函数的比值
V1	get_tokens()	0.447 368	V6	is_num_contant	0.447 368
V2	get_tokens()	0.447 368	V7	is_token_end	0.842 105
V3	get_tokens()	0.447 368	V8	is_token_end	0.105 263
V4	get_tokens()	0.447 368	V9	is_token_end	0.157 895
V5	is_str_contant	0.447 368	V10	is_str_contant	0
replace	一次定位	占总函数的比值	replace	一次定位	占总函数的比值
V1	dodash	0.071 429	V17	esc	0.523 81
V2	dodash	0.071 429	V18	omatch	0.309 524
V3	subline	0.309 524	V19	get_line	0
V4	subline	0.166 667	V20	esc	0.047 619
V5	dodash	0.071 429	V21	get_line, addstr, makepat	0.309 524
V6	locate	0.047 619	V22	getccl	0.119 048
V7	in_set2	0.047 619	V23	esc	0.380 952
V8	in_set2	0.095 238	V24	omatch	0.047 619
V9	dodash	0.119 048	V25	omatch	0.047 619
V10	dodash	0.119 048	V26	omatch	0.047 619
V11	dodash	0.119 048	V27	in_set2	0.047 619
V12	无	0	V28	in_set2	0.047 619
V13	subline	0.238 095	V29	in_set2	0.047 619
V14	omatch	0.309 524	V30	in_set2	0.047 619
V15	makepat	0.476 19	V31	omatch	0.357 143
V16	in_set2	0.047 619	V32	dodash	0.119 048
schedule	一次定位	占总函数的比值	schedule	一次定位	占总函数的比值
V1	find_nth	0.722 222	V6	find_nth	0.722 222
V2	unlock_process	0.722 222	V7	upgrade_process	0.722 222
V3	upgrade	0.047 619	V8	upgrade_process	0.722 222
V4	upgrade	0.047 619	V9	main	0.722 222
V5	upgrade_process	0.722 222			
schedule2	一次定位	占总函数的比值	schedule2	一次定位	占总函数的比值
V1	new_job	0.444 444	V6	flash	0.444 444
V2	get_process	0.444 444	V7	get_process	0
V3	get_process	0.444 444	V8	put_end	0.416 667
V4	get_command	0.444 444	V9	finish	0
V5	put_end	0.416 667	V10	get_command	0.166 667
tcas	一次定位	占总函数的比值	tcas	一次定位	占总函数的比值
V1	Non_Crossing_Biased_Climb	0.555 556	V22	Non_Crossing_Biased_Climb	0.555 556
V2	Inhibit_Biased_Climb	0.333 333	V23	Non_Crossing_Biased_Descend	0.555 556
V3	alt_sep_test	0.722 222	V24	Non_Crossing_Biased_Descend	0.555 556
V4	Non_Crossing_Biased_Climb	0.555 556	V25	Non_Crossing_Biased_Descend	0.555 556
V5	alt_sep_test	0.611 111	V26	alt_sep_test	0.722 222
V6	Own_Below_Threat	0.222 222	V27	alt_sep_test	0.611 111
V7	initialize	0.5	V28	Inhibit_Biased_Climb	0.222 222
V8	initialize	0.833 333	V29	Inhibit_Biased_Climb	0.333 333
V9	Non_Crossing_Biased_Descend	0	V30	Inhibit_Biased_Climb	0.333 333
V10	Own_Below_Threat, Own_Above_Threat	0.222 222	V31	Non_Crossing_Biased_Climb	0
V11	Own_Below_Threat, Own_Above_Threat	0.222 222	V32	Non_Crossing_Biased_Descend	0
V12	alt_sep_test	0.722 222	V33	initialize	0.833 333

Table 5 Function sets of the first time fault locating and its effectiveness values (contiuned)

表 5 一次故障定位的函数集及相应的有效性值(续)

print_tokens	一次定位	占总函数的比值	print_tokens	一次定位	占总函数的比值
V13	无	0	V34	alt_sep_test	0.722 222
V14	无	0	V35	Inhibit_Biased_Climb	0.222 222
V15	无	0	V36	无	0
V16	initialize	0.666 667	V37	ALIM	0.111 111
V17	initialize	0.611 111	V38	无	0
V18	initialize	0.555 556	V39	Non_Crossing_Biased_Descend	0.555 556
V19	initialize	0.555 556	V40	Non_Crossing_Biased_Climb	0.444 444
V20	Non_Crossing_Biased_Climb	0.555 556	V41	Non_Crossing_Biased_Climb	0.555 556
V21	Non_Crossing_Biased_Climb	0.555 556			
tot_info	一次定位	占总函数的比值	tot_info	一次定位	占总函数的比值
V1	InfoTbl	0.142 857	V13	InfoTbl	0.142 857
V2	main	0.142 857	V14	main	0.142 857
V3	main	0.142 857	V15	gser	0.142 857
V4	gct	0.142 857	V16	main	0.142 857
V5	main	0.714 286	V17	gct	0.142 857
V6	无	0	V18	InfoTbl	0.142 857
V7	InfoTbl	0.142 857	V19	无	0
V8	gser	0.142 857	V20	InfoTbl	0.142 857
V9	main	0.714 286	V21	无	0
V10	无	0	V22	InfoTbl	0.142 857
V11	gser	0.142 857	V23	gct	0.285 714
V12	Lgamma	0.428 571			

为了证明我们方法的有效性,我们单独统计每一种方法的平均有效性值.在下面的实验中,我们把发生故障时需要检查代码的行数占总行数的百分比作为故障定位的有效值,并计算不同的故障定位方法的有效值.在图 5 中,X 轴表示检测的代码的百分比;Y 轴上给出定位故障的百分比;图上的节点表示在某个检查代码的百分比下,能够定位出故障的百分比.在平均情况下,我们的方法检查 10%的代码行就可以定位大约 57%的故障版本数.在相同的有效值下,Ochiai 和 DStar 能够定位 42%的故障版本数,Tarantula 能够定位 39%的故障版本数, Barinel 能够定位 47%的故障版本数.在统计学中,为了直观地比较二次定位的优越性,现在定义发生故障时需要检查代码的行数占总行数的百分比作为随机变量 X,记录实验观察到 X 发生的概率,计算得到 $E(Tarantula)$=0.302,$E(Ochiai)$=0.296,$E(DStar)$=0.302,$E(Barinel)$=0.275,$E(Our\_Method)$=0.22.也就是说,我们的方法在定位发生故障时需要检查代码的行数占总行数的百分比的数学期望比其他 4 种方法要小.

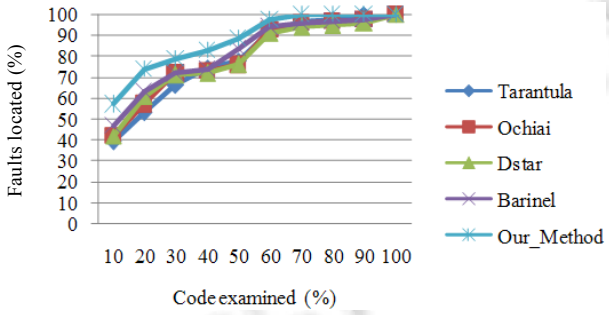

Fig.5 Comparison of different methods of valid values

图 5 不同方法有效值的比较

5 相关工作

本文中已经讨论过错误定位研究方法,现在对所附文献中所用方法进一步详细描述.

- 基于程序依赖关系的方法

Weiser 在文献[9]中提出了程序切片方法,用于描述影响程序某个执行点上特定变量的语句集合.后来的研究者通过考虑不同的依赖关系来解决故障定位中的遇到的各种问题.Baah 等人在文献[10]中扩展了程序依赖图,通过测试用例的执行信息估计节点间的统计依赖,建立了概率程序依赖图 PPDG(probabilistic program dependence graph).它基于概率图模型的框架,首先产生程序依赖图,然后得到标记了子节点和父节点之间条件概率的变换程序依赖图.同时,插桩源程序得到测试用例的执行信息.通过学习执行信息中的数据,最终得到PPDG.在用于故障定位时,首先使用 PPDG 分析一次失败执行的信息,然后对 PPDG 上的节点按照怀疑度进行排序.一个节点的条件概率值被认为是其怀疑度,基于其父节点的值计算得到,条件概率值越低,怀疑度就越高.然而,插桩源代码的开销限制了 PPDG 在大型软件系统和部署软件的应用.

- 基于语句

Renieris 等人在文献[13]中提出了使用相似的程序谱(nearest neighbor queries)来进行故障定位的技术.该方法假设存在一个失败的程序谱和很多成功的程序谱,然后根据距离准则挑选出一个与失败运行最相似的成功运行的程序谱,进而比较两个程序谱的不同之处,以分离软件故障.如果故障在失败程序谱的集合中,那么直接可以定位故障;如果故障不在失败程序谱的集合中,则通过构建程序依赖图,并逐个检查图中的节点,直到定位出故障.由于失败和成功的程序谱的数量较多,该方法还没有为其建立一个完善的过滤系统.

- 基于谓词

Liu 等人在文献[21]中提出了 SOBER 方法.该方法对谓词在成功执行和失败执行中的取值模式进行建模,然后基于统计学中假设检验的原理,量化每个谓词的故障相关性,按照谓词怀疑度的大小审查源代码,发现故障的位置.然而,该方法需要足够多的测试用例,并且其鲁棒性尚不可知.

还有一些研究人员把模型诊断和频谱相结合,提高了故障定位中的排名,如文献[20,22]基于语句的故障定位和文献[16,17]基于组件(架构)的故障定位.然而,基于语句的故障定位需要定位到代码行,这就意味着需要大量的工作去掉与故障的代码行不相关的语句,其粒度过于精细.而基于组件的故障定位组件范围很大(如服务器),程序员不能准确地定位具体的位置,其粒度又显得过于粗糙.

6 结 论

本文提出了基于二次定位策略的故障定位方法.该方法从代码层向上抽象到函数层,首先使用工具生成函数调用图,将函数之间的相互关系用拓扑结构进行描述;再从函数调用轨迹中建立程序谱,将软件的组件与程序的故障关联起来;然后利用 MBD 的逻辑推理和构建行为命题模式来推导函数候选集并计算它们的概率,并根据概率进行怀疑度的排序;最后,利用 DStar 将函数候选集的语句进行故障定位.值得注意的是,我们的方法对以下 5 种情况是不能诊断出故障的:第 1 种是故障出现在头文件中;第 2 种是故障的原因是缺少代码;第 3 种是没有故障的测试用例;第 4 种是故障出现在函数的声明上;第 5 种是故障的代码行是程序未编译.我们知道,C 语言所有程序都含有一个主函数,而主函数是程序处理的起点,在上述情况之外不存在跨函数的情形.因此,我们的方法是可行的.我们的方法在时间复杂度上优于统计技术.基于二次定位策略的故障定位方法,程序的语句行的数量级与函数行的数量级差别越大,我们的方法定位的速度就越快,详细分析见本文第 2 节.

虽然经研究表明,我们的方法有较好的定位效果,然而,我们还需要克服一些缺点并开展进一步的工作:首先,我们假设测试用例大部分是正确的,对程序本身的正确性要求较高,这无疑增加了局限性;其次,如果单个测试用例覆盖所有的函数,那么我们是无法正确诊断出具体的故障函数的,也就是说,我们的故障定位方法对测试用例有很强的依赖性;最后,Siemens 程序包是序列化程序,我们需要对并发程序进行实验,有可能会面临更多的问题.

References:

[1] 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]

[2] Goel AL. Software reliability models: Assumptions, limitations and applicability. IEEE Trans. on Software Engineering, 1985, 11(12):1411−1423. [doi: 10.1109/TSE.1985.232177]

[3] Liu C, Fei L, Yan X, Han J, 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]

[4] Xie M, Yang B. A study on the effect of imperfect debugging on software development cost. IEEE Trans. on Software Engineering, 2003,29(5):471−473. [doi: 10.1109/TSE.2003.1199075]

[5] Vessey I. Expertise in debugging computer programs. Int'l Journal of Man Machine Studies: Process Analysis, 1985,23(5): 459−494. [doi: 10.1016/S0020-7373(85)80054-7]

[6] Su XH, Gong DD, Wang TT, Ma PJ. Automatic fault localization approach combining test case reduction and joint dependency probabilistic model. Ruan Jian Xue Bao/Journal of Software, 2014,25(7):1492−1504 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4518.html [doi: 10.13328/j.cnki.jos.004518]

[7] Ding H, Chen L, Qian J, Xu L, Xu BW. Fault localization method using information quantity. Ruan Jian Xue Bao/Journal of Software, 2013,24(7):1484−1494 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4294.htm [doi: 10.3724/SP.J.1001.2013.04294]

[8] 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. [doi: 10.1016/j.jss.2009.06.035]

[9] Weiser M. Program slicing. In: Proc. of the 5th Int'l Conf. on Software Engineering. IEEE Press, 1981. 439−449.

[10] Baah GK, Podgurski A, Harrold MJ. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. on Software Engineering, 2010,36(4):528−545. [doi: 10.1109/TSE.2009.87]

[11] Jones JA, Harrold MJ. Empirical evaluation of the Tarantula automatic fault-localization technique. In: Proc. of the 20th IEEE/ACM Conf. on Automated Software Engineering. Long Beach, 2005. 273−282. [doi: 10.1145/1101908.1101949]

[12] 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, 2005. 15−26. [doi: 10.1145/1065010.1065014]

[13] Renieres M, Reiss SP. Fault localization with nearest neighbor queries. In: Proc. of the 18th Int'l Conf. on Automated Software Engineering. Montreal, 2003. 30−39. [doi: 10.1109/ASE.2003.1240292]

[14] 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. [doi: 10.1016/j.jss.2009.09.037]

[15] Zhang Z, Chan WK, Tse TH, Jiang B, Wang X. Capturing propagation of infected program states. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE). Amsterdam, 2009. 43−52. [doi: 10.1145/1595696.1595705]

[16] Casanova P, Schmerl B, Garlan D, Abreu R. Architecture-Based run-time fault diagnosis. In: Crnkovic I, Gruhn V, Book M, eds. Proc. of the 5th European Conf. on Software Architecture (ECSA 2011). LNCS 6903, Berlin, Heidelberg: Springer-Verlag, 2011. 261−277. [doi: 10.1007/978-3-642- 23798-0_29]

[17] Casanova P, Garlan D, Schmerl B, Abreu R. Diagnosing architectural run-time failures. In: Proc. of the 8th Int'l Symp. on Software Engineering for Adaptive and Self-Managing Systems. 2013. 103−112.

[18] Miao Y, Chen ZY, Li SH, Zhao ZH, Zhou YM. A clustering-based strategy to identity coincidental correctness in fault localization. Int'l Journal of Software Engineering and Knowledge Engineering, 2013,23(5):721−741. [doi: 10.1142/S0218194013500186]

[19] He T, Wang XM, Zhou XC, LI WJ, Zhang ZY, Zhang CZ. A software fault localization technique based on program mutations. Chinese Journal of Computers, 2013,36(11):2236−2244 (in Chinese with English abstract).

[20] Abreu R, Zoeteweij P, van Gemund AJC. Spectrum-Based multiple fault localization. ACM Int'l Conf. on Automated Software Engineering. 2009. 88−99. [doi: 10.1109/ASE.2009.25]

[21] Liu C, Yan X, Fei L, Han J, Midkiff SP. Sober: Statistical model-based bug localization. ACM SIGSOFT Software Engineering Notes, 2005,30(5):286−295. [doi: 10.1145/1095430.1081753]

[22] Abreu R, Zoeteweij P, van Gemund AJC. An observation-based model for fault localization. In: Proc. of the Workshop on Dynamic Analysis (WODA 2008). ACM Press, 2008. 64−70. [doi: 10.1145/1401827.1401841]

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

[24] de Kleer J, Williams BC. Diagnosing multiple faults. Artificial Intelligence, 1987,32(1):97−130. [doi: 10.1016/0004-3702(87)90063-4]

[25] Feldman A, Provan G, van Gemund AJC. Computing minimal diagnoses by greedy stochastic search. In: Proc. of the AAAI Conf. on Artificial Intelligence (AAAI 2008). Menlo Park: AAAI Press, 2008. 911−918.

[26] Feldman A, van Gemund AJC. A two-step hierarchical algorithm for model-based diagnosis. In: Proc. of the 21st National Conf. on Artificial Intelligence, Vol.1. Menlo Park: AAAI Press, 2006. 827−833.

[27] Mayer W, Stumptner M. Evaluating models for modelbased debugging. In: Proc. of the Int'l Conf. on Automated Software Engineering (ASE 2008). IEEE Computer Society, 2008. 128−137. [doi: 10.1109/ASE.2008.23]

[28] Pietersma J, van Gemund AJC. Temporal versus spatial observability tradeoffs in model-based diagnosis. In: Proc. of the Int'l Conf. on Systems, Man, and Cybernetics (SMC 2006). IEEE, 2006. 5325−5331. [doi: 10.1109/ICSMC.2006.385155]

[29] Zeller A. Isolating cause-effect chains from computer programs. In: Proc. of the 10th ACM SIGSOFT Symp. on Foundations of Software Engineering. ACM Press, 2002. 1−10. [doi: 10.1145/587051.587053]

[30] Wong WE, Debroy V, Li YH, Gao RZ. Software fault localization using dstar (D*). In: Proc. of the 6th Int'l Conf. on Software Security and Reliability. 2012. 21−30. [doi: 10.1109/SERE.2012.12]

[31] Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proc. of the Int'l Conf. on Software Engineering (ICSE 2002). ACM Press, 2002. 467−477. [doi: 10.1145/581339.581397]

附中文参考文献:

[1] 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]

[6] 苏小红,龚丹丹,王甜甜,马培军.结合用例约简与联合依赖概率建模的错误定位.软件学报,2014,25(7):1492−1504. http://www.jos.org.cn/1000-9825/4518.html [doi: 10.13328/j.cnki.jos.004518]

[7] 丁晖,陈林,钱巨,许蕾,徐宝文.一种基于信息量的缺陷定位方法.软件学报,2013,24(7):1484−1494. http://www.jos.org.cn/1000-9825/4294.htm [doi: 10.3724/SP.J.1001.2013.04294]

[19] 贺韬,王欣明,周晓聪,李文军,张震宇,张成志.一种基于程序变异的软件错误定位技术.计算机学报,2013,36(11):2236−2244.

宗芳芳(1990－),女,安徽淮北人,硕士,主要研究领域为软件建模,软件测试与可靠性.

丁佐华(1964－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件测试与可靠性,软件建模与分析,软件自适应控制系统,智能计算及应用.

黄鸿云(1977－),女,硕士,主要研究领域为软件测试与可靠性.