# 基于执行轨迹的软件缺陷定位方法研究

王新平 顾 庆 陈 翔 张 鑫 陈道蓄
(南京大学计算机软件新技术国家重点实验室 南京 210093)

**摘 要** 软件中隐含的缺陷数目与可靠性直接相关,软件缺陷定位是移除软件缺陷的关键,缺陷定位的及时性和有效性直接影响软件的可用性。基于执行轨迹的软件缺陷定位能够很好地与自动化测试相结合,有较强的现实意义。讨论了基于执行轨迹的软件缺陷定位方法通用框架FLOC,详细介绍了该框架的各个阶段,包括执行轨迹的组织、执行轨迹的选择、怀疑率的计算、定位报告的评价。分析了现有的基于执行轨迹的软件缺陷定位方法,并按照框架的结构比较了这些方法的特点,提出了改进的思路。最后对缺陷定位的发展提出展望。

**关键词** 软件调试,缺陷定位,执行轨迹,相似度,怀疑率
**中图法分类号** TP511 **文献标识码** A

## Research on Software Fault Localization Based on Execution Trace

WANG Xin-ping GU Qing CHEN Xiang ZHANG Xin CHEN Dao-xu
(National Laboratory of Novel Software Technology, Nanjing University, Nanjing 210093, China)

**Abstract** Software reliability is directly relevant to the count of faults in software. Fault localization is the key to detect and eliminate the faults. Execution trace-based fault localization is of great significance because it can be integrated well with automatic software testing. Proposed the framework of execution trace-based fault localization FLOC, which can be divided into four components: organization of execution trace, selection of execution trace, computation of suspiciousness, and evaluation of the output. The typical current execution trace-based approaches were described and compared in FLOC. Finally some improvements were proposed according to FLOC. The purpose of this paper is to compare the advantages and disadvantages of those localization approaches in a unified framework, and provide some improvements on those approaches.

**Keywords** Software debugging, Fault localization, Execution trace, Similarity degree, Suspiciousness ratio

## 1 引言

随着计算机软件的日趋复杂和网络的迅速发展,软件可信的要求越来越高,可信要求软件具有高可靠性和高可用性。软件中隐含的缺陷数目与可靠性直接相关,缺陷定位是移除软件缺陷的关键步骤,缺陷定位的及时性和有效性直接影响软件的可用性。

从缺陷引入的不同角度,导致了不同的缺陷定位方法。文献[4,8]认为缺陷是由于程序员对程序做修改而引入的,该方法把程序版本间的差异分解成原子修改集合(atom changes),找到影响测试用例执行结果的影响修改集(affecting changes),然后对该集合中的原子修改进行搜索,定位到导致程序失效的最小原子修改集。文献[10,11]认为程序的失效可以追溯到某些关键变量,该方法捕获某一时刻通过的和未通过的测试用例运行时的内存状态,通过二分查找方法定位导致失效的最小变量集合。文献[1-3,6,7,9,14]认为未通过的测试用例执行轨迹有不同于通过的测试用例执行轨迹的特征,在程序上运行大量的测试用例,通过比对通过的和未通过的测试用例的执行轨迹来定位导致失效的最小语句块的集合。其中对测试用例执行轨迹进行分析来定位缺陷,把这类方法称为基于执行轨迹的缺陷定位方法。由于该类方法不需要关于程序的预先知识,而且能和自动化测试紧密结合,本文主要分析基于执行轨迹的缺陷定位方法。

本文主要从框架结构的角度对基于执行轨迹的缺陷定位方法进行了综合分析,对现有的方法进行了介绍和比较。本文第2节给出基于执行轨迹的软件缺陷定位相关概念的定义;第3节详细介绍基于执行轨迹的软件缺陷定位方法的框架;第4节对现有的基于执行轨迹的软件缺陷定位方法进行简单介绍;第5节按照框架对现有方法进行比较并提出改进思路;最后是对本文的总结和展望。

## 2 问题描述

程序 $P=\{s_1,s_2,\cdots,s_m\}$,其中 $s_j$ 代表第 $j$ 个语句块,语句块是在任何输入下,执行情况(是否被执行)都相同的代码的

到稿日期:2008-11-19 返修日期:2009-07-28 本文受国家863项目(2006AA01Z177),国家自然科学基金项目NSFC(60873027),江苏省自然科学基金基础研究项目(BK2006115)资助。

王新平 硕士研究生,研究方向为软件缺陷定位,E-mail:xpwang@dislab.nju.edu.cn;顾 庆 博士,副教授,研究方向为软件测试、软件可靠性;陈 翔 博士研究生,研究方向为软件测试;张 鑫 硕士研究生,研究方向为软件缺陷定位;陈道蓄 博士生导师,研究方向为分布式计算与软件工程。

集合。测试用例集 $T=\{t_1,t_2,\cdots,t_n\}$，其中 $t_i$ 代表第 $i$ 个测试用例。测试用例 $t_i$ 有一个输入 $d_i$ 和预期结果 $o_i$，$t_i=(d_i,o_i)$ $(1\leqslant i\leqslant n)$。$t_i$ 在 $P$ 上的一次运行的实际结果 $o_i'=P(d_i)$，如果 $o_i'=o_i$，则称 $t_i$ 在程序 $P$ 上通过了，$t_i$ 简称为通过的测试用例；反之，如果 $o_i'\neq o_i$，则称 $t_i$ 在程序 $P$ 上未通过，$t_i$ 简称为未通过的测试用例。测试用例 $t_i$ 在程序 $P$ 上的一次运行可以用执行轨迹 $tr_i$ 来表示，$tr_i$ 是被 $t_i$ 执行了的语句块的序列。所有的测试用例执行后，测试用例集 $T$ 对应一个执行轨迹集 $Tr$，且 $T$ 中的元素和 $Tr$ 中的元素一一对应。

按照测试用例是否通过，测试用例集 $T$ 被分成两个互不相交的集合 $T_p$ 和 $T_f$，$T_p$ 是通过的测试用例集合，$T_f$ 是未通过的测试用例集合。执行轨迹集 $Tr$ 被分成两个互不相交的集合 $Tr_p$ 和 $Tr_f$，$Tr_p$ 是通过的测试用例的执行轨迹集合，$Tr_f$ 是未通过的测试用例的执行轨迹集合。

$$T_p=\{t_i\mid P(d_i)=o_i\};\quad T_f=\{t_i\mid P(d_i)\neq o_i\} \tag{1}$$

$$Tr_p=\{tr_i\mid t_i\in T_p\};\quad Tr_f=\{tr_i\mid t_i\in T_f\} \tag{2}$$

程序中的每个语句块都可能隐含缺陷，语句块 $s_j$ 可能隐含缺陷的程度称为该语句块的怀疑度。用区间[0,1]里的实数表示怀疑度，这个实数叫语句块 $s_j$ 的怀疑率 $sus(s_j)$，怀疑率越高表示该语句块被认为隐含缺陷的可能性越大。给定语句块的怀疑率，那么程序 $P$ 中的各语句块就可以按照怀疑率大小排序。缺陷定位报告就是针对程序 $P$ 中语句块的一个子集按照怀疑率大小排序形成的序列 $R=\{s_1',s_2',s_3',\cdots\}$，其中，$sus(s_1')\geqslant sus(s_2')\geqslant sus(s_3')\geqslant\cdots$。

## 3 方法框架

基于执行轨迹的软件缺陷定位首先收集测试用例集 $T$ 在程序 $P$ 上的执行信息，组织执行信息生成执行轨迹 $Tr$。并根据测试用例的执行结果将 $T$ 分成 $T_p$ 和 $T_f$，$Tr$ 分成 $Tr_p$ 和 $Tr_f$。然后分别选择 $Tr_p$ 和 $Tr_f$ 的子集，比对通过的和未通过的测试用例的执行轨迹，计算程序 $P$ 中各语句块的怀疑率 $sus(s_j)$。最后输出一个根据怀疑率排序的语句块序列 $R$。程序员逐一审查 $R$ 中的语句块，直至定位到实际缺陷。

图 1 是基于执行轨迹的软件缺陷定位方法的通用框架 FLOC，该框架可以分成四个阶段：

- 执行轨迹的组织：整理测试阶段收集的信息，按一定的方式组织测试用例的执行轨迹。
- 执行轨迹的选择：选择用于缺陷定位的执行轨迹，排除数据中的噪音，使怀疑率的计算更准确。
- 怀疑率的计算：分析选定的执行轨迹，按一定的计算模型计算各语句块的怀疑率。
- 定位报告的评价：按评价标准对定位报告打分，评价缺陷定位的效果。

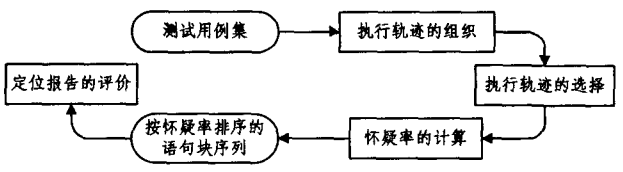


图 1 基于执行轨迹的软件缺陷定位方法的框架

### 3.1 执行轨迹的组织

执行轨迹是测试用例运行过程中所执行的语句块的序列，可以通过程序插装和动态切片等技术收集被测试用例执行了的语句块序列。但原始语句块序列数据量太大（特别是有循环语句的时候），通常按照代码的静态结构来组织执行轨迹。把程序 $P$ 看成是一个向量，向量的每一维对应程序的一个语句块，这样用于缺陷定位的执行轨迹可以按以下两种方式组织。

二值向量：向量值表示语句块是否被测试用例执行，如果语句块被执行，则向量对应维的值是 1，如果没有被执行则为 0，公式如下：

$$tr_i[j]=\begin{cases}1 & \text{if } s_j \text{ is executed by } t_i\\ 0 & \text{if } s_j \text{ is not executed by } t_i\end{cases} \tag{3}$$

多值向量：向量值表示语句块被执行的次数，如果语句块被执行了 $x$ 次，则向量对应维的值为 $x$，如果没有被执行则为 0，公式如下：

$$tr_i[j]=\begin{cases}x & \text{if } s_j \text{ is executed } x \text{ times by } t_i\\ 0 & \text{if } s_j \text{ is not executed by } t_i\end{cases} \tag{4}$$

以下的执行轨迹 $tr_i$ 均是按照向量方式组织的执行轨迹。

### 3.2 执行轨迹的选择

一般认为执行轨迹越多，缺陷定位效果越好[3]。但在自动化测试的情况下，会有大量冗余的测试用例，造成大量冗余信息，冗余信息会降低计算的准确度。而且程序中通常会同时存在多个缺陷，只有针对不同的缺陷选择不同的信息才能保证每个缺陷的计算准确度尽可能的高。可以通过测试用例或其执行轨迹间的相似性来选择执行轨迹[5]。

(1)基于输入相似性

基于输入相似性的方法假设如果输入相似，则其执行也相似。通过测试用例的输入相似性来表示其执行轨迹的相似性。该方法比较直观，但需要知道输入的结构，而且没有考虑测试用例的执行环境，在实际的执行过程中，输入的相似往往并不能保证执行轨迹的相似。

(2)基于执行轨迹距离

可以通过执行轨迹间的距离来表示相似性，距离越小则执行轨迹越相似。以下给出几种计算执行轨迹间距离的方法：

海明距离：对于二值向量可以通过海明距离（Hamming Distance）来计算执行轨迹间的距离。给定两个执行轨迹的向量表示，其距离为两个向量间互不相同的维的个数。

夹角余弦：对于多值向量可以通过向量间的夹角余弦值来计算执行轨迹间的距离。计算极坐标下两个向量间夹角的余弦值，夹角余弦值越大，表示两个执行轨迹间距离越小。

排列距离：对于多值向量可以按照向量各元素值的大小将元素排序，这样可以把执行轨迹看成语句块的排列，通过计算排列间相互转换的步数来计算执行轨迹间的距离。由于选择执行轨迹时只关注各语句块的相对关系，计算执行轨迹间的排列距离采用 Ulam 转换，即元素可以从一个位置移动到任意的位置。例如执行轨迹 $tr_1=(1,1,4,4,1,1)$，$tr_2=(1,1,0,0,1,1)$，对向量各元素排序后，$tr_1=(s_3,s_4,s_1,s_2,s_5,s_6)$，$tr_2=(s_1,s_2,s_5,s_6,s_3,s_4)$，则 $dis(\vec{tr_1},\vec{tr_2})=2$。

以上三种距离计算方法中，海明距离只关注语句块是否被执行，往往不能反映两个执行轨迹间真正的相似性，基于夹角余弦的距离考虑了各语句块被执行的次数，但直接利用次数参与计算，受循环语句块的影响较大。排列距离只考虑执行次数的相对关系，能较好地体现执行轨迹间的相似性。

### 3.3 怀疑率的计算

文献[12]应用聚类分析的方法对执行轨迹进行分析，指出未通过的测试用例执行轨迹会偏向于一个较小的聚类，文献[13]通过分析大量实验数据得出未通过的测试用例执行轨迹会有异常的特征。怀疑率的计算根据对通过的和未通过的测试用例执行轨迹进行比对来计算语句块的怀疑率。以下给出几种计算怀疑率的方法。

(1)0-1 法

该方法假设通过的测试用例执行轨迹中不包含缺陷，缺陷只存在于未通过的测试用例执行轨迹中，那么把被通过的测试用例执行了的语句块从未通过的测试用例执行轨迹中移除，剩下的部分就是被怀疑的语句块。执行轨迹表示成二值向量，语句块的怀疑率计算公式如下：

$$sus(s)=\begin{cases}1, & \text{if } s\in tr_f-tr_p\\ 0, & \text{if } s\in tr_f\wedge s\notin tr_f-tr_p\end{cases} \tag{5}$$

(2)简单统计法

0-1 法的假设条件在实际情况下往往不成立，通过的测试用例执行轨迹往往也包括存在缺陷的语句块。如果用 0-1 法，那么这部分存在缺陷的语句块就会在被怀疑代码中忽略。简单统计法统计语句块被测试用例执行的次数，给定语句块 $s_j$，执行了 $s_j$ 的未通过的和通过的测试用例执行轨迹组成的集合 $Tr_f(s_j)$和 $Tr_p(s_j)$，用 $Tr_f(s_j)$和 $Tr_p(s_j)$中执行轨迹的个数的比值来计算 $s_j$ 的怀疑率。显然，如果 $s_j$ 只被通过的测试用例执行，则 $s_j$ 的怀疑率为 0；如果 $s_j$ 只被未通过的测试用例执行，则其怀疑率为 1；执行了 $s_j$ 的未通过的测试用例占的比值越大，$s_j$ 的怀疑率就越大。

(3)模糊集法

简单统计法直接用执行轨迹的个数参与计算，不考虑其间的关系。模糊集法认为每个执行轨迹都是一个模糊集，执行轨迹中每个语句块都有一个属于这个执行轨迹的隶属度，那么 $Tr_f(s_j)$和 $Tr_p(s_j)$都是多个模糊集的并集。

根据模糊集理论可以计算 $Tr_p(s_j)$和 $Trf(s_j)$的模，通过 $Tr_f(s_j)$，$Tr_p(s_j)$的模的比值来计算 $s_j$ 的怀疑率。

0-1 法得到被怀疑的语句块集合，然后用 PDG-ranking 技术(原文中叫 SDG-ranking)[14]来对剩下的节点进行排序。简单统计法和模糊集法通过$|Tr_p(s_j)|$和$|Tr_f(s_j)|$的比值来计算 $s_j$ 的怀疑率，按照怀疑率的大小对语句块进行排序。

### 3.4 定位报告的评价

缺陷定位的意义在于帮助程序员花最小的代价找到程序中的缺陷。对于基于执行轨迹的缺陷定位，评价一个缺陷定位方法的好坏，最直接的方式就是看程序员根据该方法生成的缺陷定位报告定位到实际缺陷必需审查的语句块数 *num*，*num* 越小，程序员找到实际缺陷的代价越小，缺陷定位效果就越好。给定程序 $P$ 的程序依赖图 PDG，每个节点代表一个语句块，边代表节点间的数据依赖或控制依赖，那么缺陷定位报告中的语句块和实际缺陷语句块都是 PDG 上的节点。可以根据两个指标来评价定位报告：1. 查准率，定位报告是否能准确定位到实际缺陷节点；2. 查全率，报告是否能定位到全部的实际缺陷节点。

Renieris 和 Reiss 引入了给报告打分的概念[2]，分数反映的是程序员找到实际缺陷所需要审查的语句块数。如果报告得分 $S=0.95$，则表示程序员只需要审查 5%的语句块就能够找到实际缺陷。给定缺陷定位报告 $R=\{n_1,n_2,\cdots,n_k\}$和程序 $P$ 的程序依赖图 PDG，该评价方法根据两个参数给报告打分：1. 报告中的节点个数；2. 报告中的节点在 PDG 中和实际缺陷节点的距离。

定义 $K_e(n)$表示到节点 $n$ 的距离不超过 $e$ 的节点的集合。对 $R$ 中每个节点 $n$ 定义 $d(n)$表示节点 $n$ 离最近的实际缺陷节点的距离。对报告 $R$ 中的每个节点求最小距离 $m=\min(d(n_i))$，计算节点集 $N=\bigcup K_m(n_i)$。则报告的分数 $S$ 的计算公式为：

$$S(R)=1-\frac{|N|}{|PDG|} \tag{6}$$

程序 $P$ 的程序依赖图如图 2 所示，其中 $n_1$ 是实际缺陷节点。如果缺陷定位报告 $R_1=\{n_2,n_7\}$，则 $N_1=\{n_1,n_2,n_3,n_7,n_8\}$，$S(R_1)=1-5/8=0.375$，$R_1$ 的得分是 0.375。如果 $R_2=\{n_7\}$，则 $S(R_2)=0.5$，如果 $R_3=\{n_3\}$，则 $S(R_3)=0$。$R_1$ 中有 2 个节点而 $R_2$ 中只有一个节点，所以 $R_2$ 的得分比 $R_1$ 高；$R_3$ 中节点离实际缺陷节点 $n_1$ 的距离为 3，大于 $R_1$ 中节点离缺陷节点的最小距离 1，所以 $R_3$ 的得分低于 $R_1$。报告的定位效果 $R_2>R_1>R_3$。

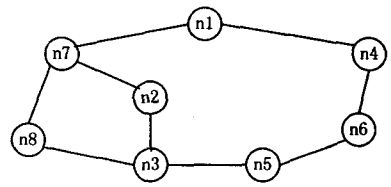


图 2 一个存在缺陷程序的程序依赖图

该方法根据距离报告中节点最近的一个实际缺陷节点来打分，当只有一个实际缺陷节点时，该方法计算的分值能够很好地反映报告的定位效果，但当存在多个实际缺陷节点时，该方法只计算对最近缺陷节点的查准率，不计算查全率，分值并不能反映报告的定位效果。

## 4 现有方法介绍

### 4.1 Dicing

文献[1]中提出利用程序切片(slicing)技术来辅助定位程序中的缺陷，两个切片之间的差称为 dice，该方法把切片表示成二值向量，然后分别选择各通过的和未通过的测试用例的切片，通过 0-1 法计算得到多个 dices，这些 dices 中的语句块组成程序员审查代码的初始集合，当缺陷不在初始集中时，通过 PDG-ranking 技术获得下一步要审查的语句块。

### 4.2 Union 和 Intersection

文献[7]中提出的 union 法和 intersection 法将执行轨迹表示成二值向量，选择一个未通过的测试用例执行轨迹和所有通过的测试用例执行轨迹来定位缺陷。union 法认为，通过的测试用例都没执行的、但未通过的测试用例都执行了的语句块可能隐含缺陷。首先对所有通过的测试用例执行轨迹求并集，然后用未通过的测试用例执行轨迹减去这个并集，从而得到被怀疑的代码块的集合。

$$suspiciousness=tr_f-\bigcup tr_p \tag{7}$$

intersection 法认为，被通过的测试用例都执行了的、但未通过的测试用例都没有执行的语句块可能隐含缺陷。首先把所有通过的测试用例的执行轨迹求交集，然后从中减去未通过的测试用例的执行轨迹得到被怀疑的语句块的集合。

$$suspiciousness = \bigcap tr_p - tr_f \tag{8}$$

这两种方法在实践中的效果并不是很好，因为隐含缺陷的语句块经常会同时被通过的和未通过的测试用例执行，在union方法中只要隐含缺陷的语句块被任何一个通过的测试用例执行了，则该语句块就会被忽略。intersection方法中任何没有执行缺陷语句块的通过的测试用例都会导致该缺陷语句块被忽略。

### 4.3 NN-Binary 和 NN-Integer

文献[2]中提出的NN(Nearest Neighbor)法取一个未通过的测试用例，然后找到和该未通过的测试用例执行轨迹最相似的通过的测试用例执行轨迹。采用了两种判断执行轨迹间相似性的方法NN-binary和NN-integer，其中NN-binary通过计算执行轨迹向量间的海明距离来判断执行轨迹间的相似性；NN-Integer通过计算执行轨迹向量间的排列距离来判断执行轨迹间的相似性。然后通过0-1法计算语句块怀疑率。

NN法通过比较两个最相似的执行轨迹得到被怀疑代码的集合，该集合比Dicing法获得的集合要小，当缺陷不在初始集中时，同样通过PDG-ranking技术获得下一步要审查的语句块。

### 4.4 Tarantula

文献[3]中提出利用程序可视化技术来辅助缺陷定位。该方法利用所有测试用例的信息，通过色彩和亮度来描述语句块的怀疑率。采用三种颜色red，yellow，green表示语句块的怀疑程度，red表示一定存在缺陷，yellow表示可能存在缺陷，green表示不存在缺陷。计算执行了语句块的通过的和未通过的测试用例的个数的比值，通过下面的公式计算语句块的颜色。

$$color(s_j) = low\ color(red) + \frac{\%passed(s_j)}{\%passed(s_j)+\%failed(s_j)} \times color\ range \tag{9}$$

其中，*low color*是所用色彩区间的最小值，*color range*是所用色彩区间的最大值和最小值的差，$\%passed(s_j)$是执行了$s_j$的通过的测试用例个数和所有通过的测试用例个数的比值，$\%failed(s_j)$是执行$s_j$的未通过的测试用例个数和所有未通过的测试用例个数的比值。

用亮度表示语句块被测试用例执行的比率，如果语句块被所有的测试用例都执行了，则该语句块最亮，如果该语句块只被少数几个测试用例执行，则亮度较暗。如果两个语句块颜色相同且都是红色，则具有较高亮度的语句块应该先被审查。通过下面的公式计算$s_j$的亮度，其中，$\%passed(s_j)$，$\%failed(s_j)$同式(9)。

$$bright(s_j) = \max(\%passed(s_j), \%failed(s_j)) \tag{10}$$

### 4.5 SAFL

文献[6]中提出的SAFL(Similarity-Aware Fault Localization)方法把测试用例是否通过及其执行轨迹组织成一个执行矩阵$E=(e_{ij})$，矩阵的行代表测试用例，列代表语句块，其中最后一列代表测试用例是否通过，1表示通过，0表示未通过。矩阵按如下公式组织，$m$为程序$P$中语句块数。

$$e_{ij} = \begin{cases} 1, & s_j \text{ is executed by } t_i\ (1 \leqslant j \leqslant m) \\ 1, & t_i \text{ is passed}\ (j = m+1) \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

SAFL法认为被某个测试用例执行的每一个语句块对该测试用例执行结果的贡献是相同的，通过执行矩阵可以计算每个语句与测试用例之间的关系，按照如下公式得到语句块与测试用例之间关系的量化矩阵$F=(f_{ij})$，其中，$f_{ij}$是语句$s_j$和测试用例$t_i$间的关系的量化，$tr_{ij}$表示$tr_i$第$j$维的值，$m$同式(11)。

$$F_{ij} = \mu_{t_i}(s_j) = \begin{cases} 1/\sum_{k=1}^{m} tr_{ik}, & \text{if } tr_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

用$Tr_f(s_j)$表示执行了$s_j$的未通过的测试用例执行轨迹的集合，用$Tr(s_j)$表示执行了$s_j$的所有的测试用例执行轨迹的集合，根据模糊集理论，集合$Tr_f(s_j)$和集合$Tr(s_j)$的模的计算公式为：

$$|Tr_f(s_j)| = \sum_{k=1}^{m} \max(\{f_{ik} \mid e_{ij} > 0 \wedge e_{i(m+1)} = 0 \wedge 1 \leqslant i \leqslant n\}) \tag{13}$$

$$|Tr(s_j)| = \sum_{k=1}^{m} \max(\{f_{ik} \mid e_{ij} > 0 \wedge 1 \leqslant i \leqslant n\}) \tag{14}$$

通过$|Tr_f(s_j)|$和$|Tr_r(s_j)|$的比值来计算语句块$s_j$的怀疑率，公式如下：

$$sus(s_j) = \frac{\sum_{k=1}^{m} \max(\{f_{ik} \mid e_{ij} > 0 \wedge e_{i(m+1)} = 0 \wedge 1 \leqslant i \leqslant n\})}{\sum_{k=1}^{m} \max(\{f_{ik} \mid e_{ij} > 0 \wedge 1 \leqslant i \leqslant n\})} \tag{15}$$

SAFL法计算每个语句块属于未通过的测试用例的隶属度和属于整个测试用例集的隶属度的比值，比值越大，则该语句的怀疑率就越高。

## 5 方法比较

现有的基于执行轨迹的软件缺陷定位方法可以看成是FLOC框架下各个子方法的组合，如表1所列，执行轨迹栏表示组织测试用例执行轨迹的方法。选择策略栏表示选择测试用例执行轨迹的方法，其中随机选择表示从$Tr$中随机选择执行轨迹，全部选择表示从$Tr$中选择全部的执行轨迹。执行轨迹数栏通过一个二元组表示执行轨迹的选择结果，$(x, y)$表示选择了$x$个通过的测试用例执行轨迹和$y$个未通过的测试用例执行轨迹。计算模型栏表示各计算怀疑率的模型。定位效果栏根据文献[1-3,6,14]中的数据将定位效果大致分成好、中、差三个级别，按照3.4节的评价方法，认为得分在80以上、缺陷比例大于20%的定位报告的定位效果是好的，得分在20以下、缺陷比例大于80%的定位报告的定位效果是差的，其余的定位效果属中等。

表1 现有基于执行轨迹方法的比较

| 方法 | 执行轨迹 | 选择策略 | 执行轨迹数 | 计算模型 | 定位效果 |
|---|---|---|---|---|---|
| Dicing | binary vector | random | (m, n) | 0-1 | bad |
| Union&Intersection | binary vector | random | (m, 1) | 0-1 | bad |
| NN-Binary | binary vector | Hamming Distance | (1, 1) | 0-1 | medium |
| NN-Integer | integer vector | Permutation Distance | (1, 1) | 0-1 | good |
| SAFL | binary vector | all | (m, n) | fuzzy set | good |
| Tarantula | binary vector | all | (m, n) | naive statistics | good |

(下转第188页)

## 参 考 文 献


[1] 吕腾,顾宁,施伯乐. XML DTD的一种范式[J]. 计算机研究与发展,2004,41(4):615-620

[2] 谈子敬,施伯乐. DTD的规范化[J]. 计算机研究与发展,2004,41(4):594-600

[3] 吴永辉. 消除结构冗余的XML数据库模式规范化设计[J]. 软件学报,2004,41(10):1809-1811

[4] Arenas M, Libkin L. A normal form for XML documents [C]// Proceedings of the 21th ACM SIGA-CT-SIG-MOD-SIGART Symposium on Principles of D-atabase Systems. Madison, Wisconsin, USA: ACM Press, 2002: 85-96

[5] Vincent M W, Liu Jixue. Multivalued dependencies and a 4NF for XML[C]// International Conference on Advance in formation Systems Engineering. Klagenfurt, Austria, 2003

[6] Vincent M W, Liu Jixue, Liu Chengfei. A redundancy Free 4NF for XML[C]// The first International XML Database Symposium. Berlin, Germany, 2003

[7] Vincent M W, Liu Jixue, Liu Chengfei. Strong functional dependencies and their application to normal forms in XML[J]. ACM Transactions on Database System, 2004, 29(3): 445-462

[8] 郝忠孝. 空值环境下数据库导论[M]. 北京:机械工业出版社,1996

[9] 殷丽凤,郝忠孝. XML强函数依赖的推理规则[J]. 计算机科学,2008,35(9):165-167

[10] 殷丽凤,郝忠孝. XML强闭包依赖的研究[J]. 计算机科学,2008,35(11):591-594


(上接第171页)

基于FLOC框架,可以将现有的基于执行轨迹的软件缺陷定位方法分成两类:1. 选择一个未通过的测试用例;2. 选择所有未通过的测试用例。但是现实情况下一个软件缺陷可能导致多个测试用例未通过,多个不同的缺陷也可能只引发一个未通过的测试用例。如果能够通过测试用例选择方法将同一个缺陷引发的多个未通过的测试用例选择出来作为一组,那么所有未通过的测试用例将被分成 $n$ 组,理想情况下 $n$ 就是缺陷的个数。如果按组来定位缺陷,那么定位该缺陷的信息将更充分。

现有基于距离的执行轨迹选择方法大都基于代码级的执行轨迹组织方式,执行轨迹的每一维通常对应程序中某行或某几行代码,对于复杂的测试用例执行轨迹,代码级的显然不能很好地描述执行轨迹间的相似性,例如,某个比较复杂的函数包含大量代码,那么计算相似性时,该函数的影响就会很大。可以从代码层向上抽象到函数层或类层,从而提高抽象的粒度,然后通过在函数层上计算执行轨迹间的距离,以避免偏向于复杂的函数。

目前计算怀疑率的方法都是分别计算每一个语句块的怀疑率,而没有考虑语句块间的交互。实际情况下语句块与语句块间的交互可能更好地体现软件中的缺陷,可以通过数据挖掘的方法挖掘执行轨迹中的这些交互模式(语句块的某种组合),将缺陷定位到某些语句块的组合。

**结束语** 本文给出了基于执行轨迹的软件缺陷定位方法的通用框架—— FLOC,并详细介绍了该框架的各个阶段。把现有基于执行轨迹的缺陷定位方法分解成框架下各个子方法的组合,综合比较了现有方法,并提出了改进思路。

目前软件缺陷定位技术还需要拓展和深化,例如,如何在海量数据中提高缺陷定位的精度;如何在数据量有限的情况下(例如系统运行过程中通过网络返回的失效相关数据)定位缺陷;如何利用已有的数据挖掘技术进行缺陷模式的挖掘等问题都需要进一步研究。

## 参 考 文 献


[1] Agrawal H, Horgan J, London S, et al. Fault localization using execution slices and dataflow tests[C]// Proceedings of IEEE Software Reliability Engineering. Los Alamitos CA: IEEE Computer Society Press, 1995: 143-151

[2] Renieris M, Reiss S P. Fault Localization with Nearest Neighbor Queries[C]// International Conference on Automated Software Engineering. Los Alamitos CA: IEEE Computer Society Press, 2003: 30-39

[3] Jones J A, Harrold M J, Stasko J. Visualization of Test Information to Assist Fault Localization[C]// Proceedings of International Conference in Software Engineering. Los Alamitos CA: IEEE Computer Society Press, 2002: 467-477

[4] Stoerzer M. Finding Failure-Inducing Changes in Java Programs using Change Classification[C]// The 14th ACM Symposium on Foundations of Software Engineering. Oregon, USA, 2006

[5] Groce A D. Error Explanation and Localization with Distance Metrics[D]. Pittsburgh, PA: Carnegie Mellon University, 2005

[6] Hao Dan, Pan Ying. A Similarity-Aware Approach to Testing Based Fault Localization[C]// International Conference on Automated Software Engineering. California, USA, 2005

[7] Chen T Y, Cheung Y Y. On program dicing[J]. Software Maintenance: Research and Experience, 1997, 9(1): 33-46

[8] Ren Xiaoxia, Chesley O C. Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis[J]. IEEE transactions on software engineering, 2006, 32(9): 718-732

[9] Jones J A. Fault Localization using Visualization of Test Information[C]// International Conference in Software Engineering. Edinburgh, UK, 2004

[10] Cleve H. Locating Causes of Program Failures[C]// International Conference in Software Engineering. Missouri, USA, 2005

[11] Zeller A. Isolating cause-effect chains from computer programs [C]// Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering(FSE-10). South Carolina, USA, 2002

[12] Dickinson W, Leon D, Podgurski A. Pursuing failure: The distribution of program failures in a profile space[C]// Proceedings of 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering. 2001: 246-255

[13] Harrold M J, Rothermel G, Sayre K, et al. An empirical investigation of the relationship between spectra differences and regression faults[J]. Software Testing, Verification and Reliability, 2000, 10(3): 171-194

[14] Jones J A. Empirical Evaluation of the Tarantula Automatic Fault Localization Technique[C]// The 20th International Conference on Automated Software Engineering. New York: ACM Press, 2005: 273-282