基于关联挖掘的软件错误定位方法

赵 磊[1),2)] 王丽娜[1),2)] 高东明[1),2)] 张震宇[3)] 熊作婷[1),2)]

[1)](武汉大学空天信息安全与可信计算教育部重点实验室 武汉 430072)

[2)](武汉大学计算机学院 武汉 430072)

[3)](中国科学院软件研究所计算机科学国家重点实验室 北京 100190)

摘 要 基于覆盖率的错误定位(Coverage Based Fault Localization,CBFL)方法旨在通过分析程序执行的结果预测错误信息,是一种行之有效的错误定位方法.然而,CBFL方法中代码覆盖率的独立统计忽略了程序内存在的复杂控制依赖和数据依赖,从而忽视了语句间的语义关系,影响错误定位的准确性.该文借助实例重点分析了基于代码覆盖率所得到的错误可疑度与错误代码的表现关系,指出现有CBFL方法的不足是片面地将基于覆盖率的错误可疑度直接作为错误代码判定的依据;提出程序失效规则及基于覆盖向量的覆盖信息分析模型,并在此模型基础之上,指出高可疑代码与错误代码在执行路径上的覆盖一致性,进而提出用以挖掘与高可疑代码相关联的错误代码的频繁集求解方法.以SIR基准程序为实验对象建立的受控实验结果表明,相比之前的研究,文中方法在一定程度上能够改进错误定位结果.

关键词 软件调试;错误定位;关联挖掘;覆盖向量;频繁集

中图法分类号 TP311 **DOI号**: 10.3724/SP.J.1016.2012.02528

Mining Associations to Improve the Effectiveness of Fault Localization

ZHAO Lei[1),2)] WANG Li-Na[1),2)] GAO Dong-Ming[1),2)] ZHANG Zhen-Yu[3)] XIONG Zuo-Ting[1),2)]

[1)](*Key Laboratory of Aerospace Information Security and Trust Computing*, *Ministry of Education*, *Wuhan* 430072)

[2)](*School of Computer*, *Wuhan University*, *Wuhan* 430072)

[3)](*State Key Laboratory of Computer Science*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)

Abstract Coverage-based fault localization (CBFL) techniques find the fault-related positions in programs by comparing the execution statistics of passed executions and failed executions have been proven to be efficient by several empirical studies. However, these techniques assess the suspiciousness of program entities individually, whereas the individual coverage information cannot reflect the complicated control- and data-dependency relationships, and thus oversimplify the execution spectra. In this paper, we first use motivating examples to show the impact of the cause-effect relationship on the effectiveness of CBFL. Second, we propose the rules of program failures and design the execution analysis model based on the coverage of different program execution spectrum. By computing the frequency items for statements with high suspiciousness, we also bring out the coverage vector to mine fault-prone statements. The controlled experiments based on the SIR benchmarks indicate that our technique is promising.

Keywords software debugging; fault localization; association mining; coverage vector; frequency items

---

收稿日期:2010-10-19;最终修改稿收到日期:2011-07-20.本课题得到国家自然科学基金(90718006,60970114,61003027,61073006)、教育部博士研究生学术新人项目资助.**赵 磊**,男,1985年生,博士研究生,主要研究兴趣为软件调试技术、安全测试.E-mail:zhaolei.whu@gmail.com.**王丽娜**,女,1964年生,博士,教授,博士生导师,主要研究领域为网络安全、可信计算.**高东明**,男,1990年生,硕士研究生,主要研究兴趣为软件错误定位.**张震宇**,男,1979年生,博士,副研究员,研究方向为软件测试、软件调试.**熊作婷**,女,1988年生,硕士研究生,主要研究兴趣为软件调试.

1 引　言

近年来，软件发展日新月异，覆盖了日常生活、工作等社会各个层次.然而，软件缺陷的存在却经常导致信息系统的失效和崩溃，给系统的可信运行带来挑战[1].缺陷的产生，很大一部分是由编码阶段注入的软件错误导致的."软件调试"是定位并排除软件错误的常用手段，也是软件开发过程中繁琐且易出错的过程，具有很高的自动化需求."错误定位"指的是在已知程序失效的情况下定位错误代码的过程[2]，是软件调试的首要步骤.精确的错误定位方法研究，一方面有利于缺陷的检测、诊断及修复，另一方面则可以降低软件测试及维护成本，对软件质量及可靠性的提高具有重要意义[3].

基于覆盖率的错误定位(Coverage Based Fault Localization，CBFL)以多次程序执行轨迹之间的统计与比较作为其主要实践方法，是一种行之有效的软件错误定位技术[1,3-5].然而，现有 CBFL 方法只是统计代码语句或代码基本块的覆盖率，并基于代码覆盖率计算代码的错误可疑度.这种计算方法对程序的执行流程进行了高度的简化，忽略了软件中存在着的复杂控制依赖和数据依赖关系[5].由于程序内错误的触发及失效在很大程度上取决于程序执行时的上下文环境[6-7]，因此现有 CBFL 方法在很多情况下的错误定位并不准确[5]，如何提高错误定位方法的准确性和有效性是 CBFL 方法研究中的重点问题，也是近几年的研究热点.

本文首先分析了 CBFL 方法在某些情况下准确性缺失的原因，指出基于覆盖率计算所得到的错误可疑度与错误代码之间并不存在直接的因果关系，进而不能将错误可疑度直接作为判定代码出错与否的依据.考虑程序的失效过程，具有较高错误可疑度的高可疑代码是程序失效的重要特征，是错误定位的重要线索，同时高可疑代码与错误代码之间存在强关联性，这种关联性是提高错误定位的重要因素.为了挖掘这种关联性，我们基于覆盖向量建立执行轨迹分析及频繁集求解模型.该模型以高可疑代码为目标，通过在失效执行轨迹上的频繁集求解来挖掘该高可疑代码的关联代码，并以挖掘出的关联代码作为代码检查的对象.以 SIR 基准程序的实验表明，基于覆盖向量的频繁集挖掘方法能在一定程度上提高错误定位的准确性.

本文的贡献主要有以下几点：(1)首次提出以挖掘错误代码与高可疑度代码之间的关联关系为手段的错误定位方法；(2)以 SIR 基准程序为对象的验证实验表明，本文方法较现有方法能够在一定程度上提高错误定位准确度.

本文第 2 节介绍和总结目前的研究进展和相关工作，并以实例论述研究动机；第 3 节给出一些基本定义及规则；第 4 节论述错误定位分析模型的建立以及在此模型基础上的关联挖掘算法；第 5 节以 SIR 基准程序为对象建立实验环境、分析实验结果；第 6 节讨论本文方法的有效性及可能的改进方式；最后总结全文.

2 相关工作及研究动机

本节主要介绍以分析比较测试结果为主要手段的错误定位方法的研究现状，并着重分析 CBFL 的相关研究，以实例来说明本文的研究动机.

2.1 错误定位方法研究现状

错误定位的研究按照其方法大致分为 3 类：基于程序切片的错误定位方法、差分式错误定位方法及基于覆盖率的错误定位方法.

基于程序切片的错误定位利用程序执行的代码语句对程序输出的数据传递关系，动态切分程序执行代码，得到失效路径[8].该类方法具有很好的实用性，但是在处理复杂软件时依然面临切片代码庞大的问题[9].

借助于执行轨迹之间的比较，Agrawal 等人[2]提出通过计算成功与失效执行轨迹的差集以隔离代码的方法.在此基础之上，Renieris 和 Reiss[6]提出基于最小距离邻近路径的定位方法，较之前的方法具有更好效果.Cleve 和 Zeller[10]以二分法构造测试输入，并且以细粒度的程序执行状态对比为手段，提出了差分式调试方法(Delta Debugging).

CBFL 方法引入概率计算，统计和比较语句、分支和谓词等在多个不同的成功及失效的执行过程下的覆盖率，计算错误可疑度，然后再依据可疑度对代码进行隔离和排序，优先检查可疑度高的代码[11].实验研究表明，CBFL 方法不管在可实施性还是有效性方面均较其它方法具有更好的效果[3].

根据代码覆盖信息的体现方式不同，CBFL 方法又可以分为两种，分别是基于代码语句覆盖的错误定位方法和基于谓词覆盖的错误定位方法.

基于代码语句覆盖的方法中的覆盖信息是程序代码在某一次程序执行过程中是否被执行，如

Tarantula[11]、Jaccard[4]、Ochiai[4] 等. Tarantula 认为代码被失效执行覆盖的比例越大，意味着该代码出错的可能性越大. Abreu 等人[4] 引入 Jaccard 和 Ochiai 相似度系数，分别提出 Jaccard 和 Ochiai 方法.

基于谓词覆盖的错误定位方法通过插桩方式在源代码中插入能够表征软件行为（如程序的不同分支、返回值、特定值的比较等）的谓词[12]，然后以谓词覆盖率分析其周围代码的出错概率. 代表性的方法有 CBI[12-13]、SOBER[14] 等，此外，Yu 等人[15] 将 CBI 方法改造成基于代码语句的 SBI 方法. 与基于代码语句覆盖的方法对比，CBI[12] 等方法得到的代码覆盖信息更加丰富. 虽然覆盖信息的体现方式不同，但上述两类 CBFL 方法中代码语句和谓词的错误可疑度计算都是依据其覆盖率的统计与比较[15].

借助于调试反馈信息，Hao 等人[16] 提出了一种交互式错误定位方法；同时考虑到测试用例相似性对 CBFL 方法结果的影响，Hao 等人[17] 又提出了测试用例约减方法以进一步提高 CBFL 方法的有效性.

CBFL 方法中不同代码的覆盖率统计是相互独立的，事实上程序先后执行的代码有很强的依赖关系[5]，在失效执行上则表现为程序感染状态的传递与转换. 所以，独立的代码覆盖率统计没有考虑程序结构中的依赖以及动态执行中程序感染状态的传递等关系，进而将导致定位的不准确，如 CBFL 方法经常可以定位出程序失效时的执行代码，而程序失效时的执行代码很多情况下并不是错误代码[5]. 如何挖掘导致程序失效的错误代码以提高错误定位的有效性是本文的研究动机.

2.2 本文研究动机

图 1 所示是一个实际代码中的错误、程序控制流图及测试情况[5]，代码中框内的部分表示错误所在的位置，测试用例编号为 $t_1 \sim t_7$，每个测试用例对应的执行覆盖信息如图 1 中“●”所示，其中被阴影覆盖的测试用例对应失效执行.

Statements	Control flow graph	Test cases							Coverage	
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	$failed(b)$	$passed(b)$
if (*block_queue*) {	b_1, e_1	●	●	●	●	●	●	●	2	5

count = *block_queue*->*mem_count* + 1;
/*The extra '+1' is the bug*/
n= (int)(count * ratio);
proc = find_nth(*block_queue*, *n*);
if (*proc*) {
	b_2, e_2, e_3, e_4	●		●		●	●		2	2

block_queue = del_ele(*block_queue*, *proc*);
prio = *proc*->priority;
prio_queue[*prio*] =
append_ele(*prio_queue*[*prio*], *proc*);
	b_3, e_5			●					1	0

}}	b_4	●	●	●	●	●	●	●	2	5

图 1 本文研究动机 1

依据执行过程收集到 b_2 和 b_3 的代码覆盖信息为 $\{failed(b_2)=2, passed(b_2)=2\}$ 和 $\{failed(b_3)=1, passed(b_3)=0\}$，即覆盖 b_3 的执行均为失效执行，而覆盖 b_2 的执行中仅一部分为失效执行，按照现有 CBFL 方法如 Tarantula 或 SBI 等可得到 b_2 的错误可疑度小于 b_3 的错误可疑度，而实际上错误代码位于 b_2.

考察程序的执行过程，b_2 执行后产生了错误数据 *count*，同时 b_3 的执行会调用变量 *count*，这使得覆盖 b_3 的执行均由于引用错误数据 *count* 而引发失效，即覆盖 b_3 的执行均是失效执行. 由于覆盖 b_3 的执行中失效执行所占的比例要大于覆盖 b_2 执行中的失效执行所占比例，按照 Tarantula 和 SBI 等的方法可以计算得到 b_3 的错误可疑度大于 b_2.

如本文引言所述，该问题产生的原因是孤立的代码覆盖率统计简化了程序的执行过程. 针对这一问题，文献[5]以状态转换为手段，将程序执行看作感染状态的转换和传递模型，迭代计算错误转换概率，提出了一种基于状态转换的错误定位方法.

然而，文献[5]中按照执行顺序从后往前的感染状态计算方法可能并不适用于所有情况. 文献[5]以连接两个连续基本块的分支路径上的失效覆盖比例来表示感染状态的传递概率，但是被失效执行覆盖的代码基本块并不意味着该段代码在执行时程序状态已感染，在错误被触发之前执行的代码同样会被失效执行覆盖，所以，分支路径上的失效覆盖比例并不能直接作为从程序感染状态在基本块之间的传递概率. 比如，b_1 和 b_2 为两个连续基本块，错误位于 b_2

而 b_1 优先 b_2 执行，错误的触发必须满足条件 C，同时当且仅当 b_1 执行之后才会使得当前的程序状态满足 C 并触发错误. 这种情况下，覆盖 b_1 的失效执行比例很大，但是 b_1 执行时程序并非处于感染状态. 为了更好地说明问题，本文构造了图 2 中的示例代码.

图 2 中代码中错误存在于 b_4，假设代码的测试用例执行情况与图 1 相同，则所有 $b_2 \rightarrow b_3 \rightarrow b_4$ 的路径均为失效路径，$b_2 \rightarrow b_4$ 的路径中一部分为成功路径，一部分为失效路径，可得，覆盖 b_3 的执行中失效执行的比例为 100%，而按照文献[5]的方法计算出错的代码极大可能存在于 b_2.

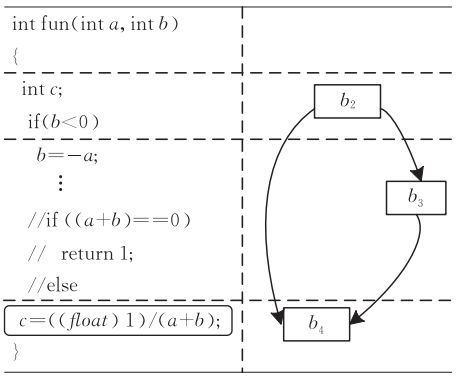

图 2　本文研究动机 2

综合考虑上述几种情形，基于覆盖率计算得到的高错误可疑度的代码主要有如下几种情况：

情形 1.　该高可疑代码本身含有错误，在代码覆盖率上表现为覆盖该代码的执行多为失效执行，而成功执行一般不覆盖该代码.

这是一种较为简单的错误定位，但实际情况中更多的错误触发及程序失效并不该满足简单情形.

情形 2.　该高可疑代码的执行会引用错误触发所产生的错误数据，并将程序的感染状态一直传递至失效.

图 1 中的示例满足情形 2. 这种情况下一般错误位于高可疑代码执行之前.

情形 3.　该高可疑代码的执行会产生满足错误触发的特殊程序状态. 在这种情况下，错误的触发只有在该代码执行后才可能发生.

示例如图 2 中的代码，这种情况下一般错误位于高可疑度代码执行之后.

错误定位的不准确表现为错误代码的可疑度较低而错误无关的代码却被计算赋予了更高的可疑度. 为了解决这一问题，有效的可疑度计算方法[5]固然是一个思路；同时，CBFL 方法的准确性缺失表现为“错误无关的代码却计算得到较高的错误可疑度”. 分析这种现象的产生，高可疑代码与程序错误的触发、感染状态传递及程序失效具有直接关系.

仔细考察如上的 3 种情形可以发现，大多数情况下，可疑度高的代码或者是错误代码(情形 1)，或者和错误代码之间存在程序感染状态的传递关系(情形 2)，或者是导致错误触发的诱因(情形 3). 既然错误代码与高可疑度的代码之间存在从感染状态产生到感染状态表现的这种关联性，那么能否建立能够体现关联性的程序执行的分析模型，并在该模型之上挖掘这种关联性以作为错误定位的重要因素？一旦挖掘出与高可疑代码相关的关联代码，则意味着缩小了待检查代码的范围，进而提高错误定位的有效性. 因此，本文的主要研究问题集中在：(1) 如何建立能够突出错误与失效因果关系的执行轨迹分析模型；(2) 如何挖掘与高可疑度相关联的错误代码.

事实上，结合静态分析考虑，程序的执行轨迹包含着丰富的结构化信息，在此基础之上，挖掘错误和程序失效之间的因果关系从而准确定位错误是可行的. 本文将在以下部分中从程序执行轨迹的分析方法和错误与失效的关联挖掘方法两方面进行详细论述.

2.3　同相关工作的关系

在之前的研究中，与本文方法相近的有文献[18]和文献[19].

本文与文献[18]都是通过搜索关联代码来提高错误定位准确度，然而文献[18]通过静态分析获取高可疑代码的数据依赖关系，而我们则是通过执行路径上的频繁集求解来实现，相比较而言，本文中路径分析的代价比数据流分析的代价小.

文献[19]以揭示错误的发生过程为出发点，首先基于元素选择和机器学习方法遴选出错误可疑代码，然后再将错误可疑代码通过控制流联系起来. 然而文献[19]并没有在代码的错误可疑度或错误排查建议的组织方法上改进.

3　基本定义及规则

3.1　基本定义

定义 1.　中间不存在控制跳转的连续代码语句构成一个代码基本块，简称为基本块.

例如，图 1 中 b_2 包含的 4 个语句构成基本块 b_2.

定义 2.　程序代码中的一个函数构成一个控制流图 CFG，定义 $CFG=\{B,E\}$，其中 B 表示基本

块集合，E 为基本块之间的跳转集合.

例如，图 1 中 $B=\{b_1,b_2,b_3,b_4\}$，$E=\{e_1,e_2,e_3,e_4,e_5\}$.

定义 3. 执行路径表示程序在一定输入下所执行代码的路径表示.

例如，图 1 中 t_2 的执行表示为 $b_1\rightarrow b_4$. 根据程序运行结果来区分，成功执行中的路径简称为成功执行路径，失效执行中的路径简称为失效执行路径.

3.2 程序失效执行规则

规则 1. 失效执行只能覆盖控制流图中的可执行路径.

不可执行路径意味着程序在当前测试用例集中的任何测试输入下都不能执行的路径，所以感染状态也不可能沿着不可测试路径传递.

规则 2. 若程序执行表现为失效，则本次执行一定触发了错误，即与本次执行相对应的执行路径会覆盖错误代码.

根据 PIE 模型[20]，程序的失效必须满足如下 3 个条件：(1) 错误触发并产生感染状态；(2) 感染状态随着程序的执行而传递；(3) 程序的输出受感染状态影响. 根据条件(1)可知执行时触发错误. 根据条件(2)和(3)，错误被触发后所产生的感染状态随着执行路径一直传递直至程序失效. 因此，错误代码会出现在执行路径上.

需要说明的是，如果错误为语句缺失型错误(statements omission fault)，比如条件判断分支不充分导致的部分代码被遗漏，则该错误代码不会出现在程序源代码中，失效执行也不会覆盖错误代码. 但是，在定位缺失型错误时，相关的方法也是通过缺失型错误相关的代码来分析[21]. 图 2 中的代码错误遗漏了一个条件判断，定位该错误时仍需要从程序中存在的且与缺失型错误相关的代码入手，如图 2 中的斜体代码. 因此，缺失型错误的定位仍然满足规则 2.

4 面向关联挖掘的频繁集求解

根据定义 3、规则 1 和规则 2 可以发现，执行路径可以很好的来表示错误代码与高可疑度代码之间的关联关系，它们在执行路径的覆盖信息上表现出很好的一致性，这种一致性体现在失效执行路径在覆盖高可疑度代码的同时也会覆盖引起失效的错误代码. 然而，执行轨迹的路径表示是一个很复杂和耗时的工作[22]，为此，我们采用相对轻量级的覆盖向量[15]来近似表示路径的覆盖信息.

4.1 路径的覆盖向量表示

定义 4. 覆盖向量指代码基本块在每次执行中的覆盖信息构成的向量 $\boldsymbol{path}_i=\langle b_1,b_2,\cdots,b_i,\cdots,b_n\rangle$，其中 $\boldsymbol{path}_i$ 表示覆盖向量，b_j 表示程序中的代码基本块，记为该向量的分量，j 记为分量的序号. $b_j=0$ 表示基本块 b_j 没有执行，记为 b_j 没有被 $\boldsymbol{path}_i$ 覆盖；$b_j=1$ 表示基本块 b_j 执行，记为 b_j 被 $\boldsymbol{path}_i$ 覆盖.

在本文的后续章节中(如算法 1)，我们也使用 $\boldsymbol{path}_i(b_k)$ 来表示 b_k 在覆盖向量 $\boldsymbol{path}_i$ 上的分量值. $\boldsymbol{path}_i(b_k)=1$ 表示 b_k 在 $\boldsymbol{path}_i$ 上的分量值为 1，$\boldsymbol{path}_i(b_k)=0$ 表示 b_k 在 $\boldsymbol{path}_i$ 上的分量值为 0.

例如，图 1 中 t_2 执行的覆盖对应的向量为 $\langle 1,0,0,1\rangle$.

虽然覆盖向量并不能直接等价于执行路径[15]，即相同的覆盖向量可能对应于不同的执行路径. 但是，覆盖向量可以保持本文所关注的覆盖信息的一致性，而覆盖向量的获取相对完整的执行路径更简单便捷.

定义 5. 对于程序中的自定义函数和主函数，定义函数在测试集下的执行轨迹的符号化表示为 $EXEM(f_i)=\{B,T,\boldsymbol{PATH}\}$，其中 B 表示函数 f_i 源代码中的基本代码块集合，T 表示测试集中的所有测试用例的集合，$\boldsymbol{PATH}=\{\boldsymbol{path}_0,\boldsymbol{path}_1,\cdots,\boldsymbol{path}_m\}$ 表示全部测试用例对应的覆盖向量集合.

根据定义 4，我们对程序的每一次执行分别计算其覆盖向量，即覆盖向量和测试集中的测试用例一一对应. 设定 n 即为 B 中基本块的个数，m 为 T 中测试用例的个数即程序执行的次数，那么覆盖向量集合 $\boldsymbol{PATH}$ 是由多个多维向量构成的 $n\times m$ 的矩阵.

此外，根据程序执行结果的不同，我们将执行轨迹分为成功执行和失效执行两类，分别标记为 $EXEM_p$ 和 $EXEM_f$.

4.2 频繁集求解

依据高可疑代码与错误代码在覆盖向量上表现出的一致性，假设我们能够用某种“频繁集”刻画出与高可疑代码直接相关的基本块集合，那么，如果高可疑代码没有错误，错误代码则存在于高可疑代码的频繁集中. 在我们的模型中，频繁集按照如下的方法建立. 根据上述分析，基于关联挖掘的错误定位模型可归结为如下描述.

模型 1. 待考察基本块 $b_o\in B$ 具有一定的错误可疑度，同时已知程序的失效执行轨迹 $EXEM_f$，

则错误定位的对象除了包括基本块 b_o 之外，还包括在 $EXEM_f$ 中在所有覆盖向量上对 b_o 保持覆盖一致性的分量所对应的基本块，这些分量共同构成基本块 b_o 的频繁集，b_o 称之为目标基本块.

需要说明的是，这里所指的覆盖一致性需要针对目标基本块 b_o 而言，与 b_o 保持覆盖一致性的分量仅包括那些在所有 $EXEM_f$ 覆盖向量中当 b_o 分量为 1 时，该分量值也为 1. 此外，覆盖一致性是单向的，举例而言，b_1 对 b_2 保持覆盖一致性是指当 b_2 分量为 1 时，b_1 分量总为 1. 但是却不能推出 b_2 对 b_1 保持覆盖一致性，这是因为当 b_2 分量为 0 时，b_1 分量可以为 1. 本文后文中的覆盖一致性均是此意.

由于频繁集内的基本块具有一定的执行序列关系，所以我们对频繁集也用向量形式表示. 同时为了行文方便，频繁集中的项也称之为频繁集的分量，且分量依据其在频繁集中的位置具有一定的序号.

在错误可疑度的计算方面，本文采用现有成果，并按可疑度从大到小的顺序对各个代码基本块进行排序，降序排列的基本块标记为 OBS.

建立频繁集的算法见算法 1. 算法 1 中 b_k 为目标代码，$\boldsymbol{fg}(b_k)$ 表示对 b_k 保持频繁一致性的分量集，即求解出的以 b_k 为目标代码的频繁集. 首先初始化 $\boldsymbol{fg}(b_k)$ 为单位向量，然后依次遍历 b_k 分量不为 0 的覆盖向量并将该覆盖向量与 $\boldsymbol{fg}(b_k)$ 进行向量的与操作. 最后可得 b_k 的频繁集.

算法 1.　频繁集求解.

输入：OBS，$EXEM_f$

输出：$\boldsymbol{FG}$(频繁集集合)

符号表示：

$\boldsymbol{fg}(b_k)$：以 b_k 为目标代码的频繁集，k 表示 b_k 在 OBS 中的索引

$u \wedge v$：向量与操作

$\boldsymbol{I}$：单位向量，维度为基本块个数

初始化：$\boldsymbol{FG} \leftarrow \varnothing$

```
1.  for each b_k ∈ OBS
2.      fg(b_k) ← I
3.      for each path_i ∈ PATH
4.          if path_i(b_k) > 0 then
5.              fg(b_k) ← fg(b_k) ∧ path_i
6.          end if
7.      end for
8.      FG ← FG ∪ fg(b_k)
9.  end for
```

4.3　代码检查次序组织

通过频繁集挖掘，我们可以获得与待检查代码，尤其对高可疑代码保持覆盖一致性的代码语句序列. 如果高可疑代码不是错误代码，我们可以通过优先检查频繁集中的代码语句的方式来组织检查顺序，以减小代码检查率.

事实上，本文的方法依据程序执行覆盖信息中的向量分析提出了一种用以提高错误定位有效性的待检查的代码语句的次序组织方法. 在 Tarantula 等方法中，每条代码语句具有独立可疑度，而本文方法计算得到的频繁集则是一组代码语句序列，为了能够与之前的方法形成对比，我们设计了如下的待检查代码语句排序方式：

(1) 依据 CBFL 方法计算每一基本块的错误可疑度；并将各基本块按照其可疑度的大小降序排列.

(2) 从排序后的列表中依次检查基本块是否含有错误，如果确认错误存在，转(5)，否则转(3).

(3) 将没有错误的基本块作为目标代码，依据覆盖信息矩阵求解其频繁集.

(4) 将频繁集中的基本块依据其可疑度大小降序排列，依次检查各基本块是否含有错误，如果定位出错误，转(5)，否则转(2).

(5) 统计已检查的基本块数量.

针对相同可疑度的不同基本块之间的检查次序问题，本文在计算代码检查率时采取最差检查策略[23]，即将所有与错误代码具有相同可疑度的代码都认定为定位错误所必须检查的代码.

4.4　复杂度分析

本文方法的复杂度主要体现在频繁集的求解上面. 依据不同类型的错误，复杂度也会不同. 设定程序源代码中的某函数代码的基本块数量为 b_m，测试用例个数为 t_m. 如果程序中仅含有一个错误且该错误不是代码缺失性错误，则通过一次的频繁集求解则可定位出错误，复杂度为 $O(b_m \times t_m)$. 如果程序中含有多个错误，则频繁集求解过程需要进行多次，最坏的情况是对每一行代码语句均求解其频繁集，复杂度为 $O(b_m^2 \times t_m)$.

5　实验及结果分析

在程序没有崩溃而被正常捕获异常的情况下，基本代码块内所有的代码语句是连续执行的，因而具有相同的覆盖率. 本文在实验部分采用代码语句作为错误可疑度计算的最小粒度[11,15].

5.1　实验建立

参照之前 CBFL 的相关研究，本文采用 3 个经

常用到的实际 Unix 程序(flex,gzip,grep)和 SIR[24] 提供的 Siemens 程序集作为实验对象. flex,gzip,grep 的代码行数在 8000～10 000 之间,而 Siemens 程序集中的 7 个程序代码量较小,相比 Siemens 程序集而言,Unix 程序是实际的应用程序且程序结构更加复杂,经常被相关研究当作实验对象[5,23,25].

表 1 所示为本文实验中的程序及相应的测试用例集. 与之前的研究一样[5,8],我们在实验中去掉了那些不能被测试集中的测试用例触发的错误,实际实验中的错误版本数量共计为 222.

表 1 实验样本信息

程序	错误个数	测试用例个数
flex	56	567
grep	21	809
gzip	18	213
print_tokens	7	4130
print_tokens2	10	4115
replace	29	5542
schedule	9	2650
schedule2	9	2650
tcas	40	1578
tot_info	23	1054
合计	222	

我们的实验平台是 ubuntu 10.4,编译器为 gcc-4.4.1,使用 gcc 的组件 gcov 来收集执行轨迹.

5.2 评估指标

在评估错误定位结果的优劣上本文采用定位错误所需检查的代码比例(简称代码检查率)作为评估指标.

按照 CBFL 计算方法,不同的代码语句或谓词可分别计算得到不同的错误可疑度,代码语句的错误可疑度越高则意味着该代码语句越应该被优先检查,然后对程序内的所有代码语句依据其错误可疑度进行降序排列. 所谓代码检查率,则是指从错误可疑度最大的代码语句开始检查,到检查到含有错误的代码语句时的代码检查总量占全部可执行代码的比例. 可执行代码不包括程序注释、空行、函数及变量声明、类型等.

本文分别采用了两种不同实验结果体现方法:(1)在一定代码检查率范围内,不同的定位方法所能够定位出的错误比例;(2)不同方法在定位同一错误时所需的代码检查率.

5.3 实验结果总体分析

本文分别采用 Jaccard[4]、Ochiai[4]、Tarantula[11] 和 SBI[15] 方法计算代码的错误可疑度. Jaccard、Ochiai 和 Tarantula 方法都是基于代码语句的定位方法,具有很强的代表性,且经常被相关研究用作对比分析. CBI[12] 是基于谓词的定位方法的代表. 文献[15]将 CBI 方法修改为基于代码语句的 SBI 方法,因此我们也将 SBI 方法用作对比. 除此之外,基于谓词的代表性方法还包括 Liblit05[13] 和 SOBER[14],但是由于 Liblit05 和 SOBER 方法更多的是依据谓词的真假在不同执行中体现出来的频谱差距,这与本文中代码语句的可疑度计算略有不同,我们没有将 Liblit05 和 SOBER 用作对比.

本文方法中的代码语句的错误可疑度计算也分别采用 Jaccard、Ochiai、Tarantula 和 SBI 方法,并分别命名为 AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法.

由于 flex、grep、gzip 等 3 个实际的 Unix 程序和 Siemens 程序集在程序来源和源代码规模上均不同,所以我们分别针对 Unix 程序和 Siemens 程序集建立了实验. Unix 程序实验结果如图 3 所示. 图中横坐标表示代码检查率,纵坐标表示在一定的代码检查率范围内能够定位出的错误个数占错误总数的比例.

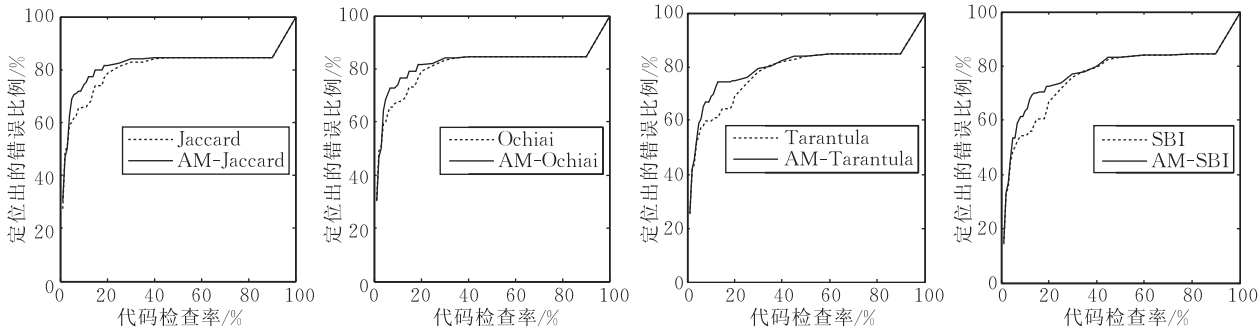

图 3 Unix 实验结果总体对比

图 3 中的实线分别表示基于关联挖掘的 AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法所能够定位出的错误比例随代码检查率的变化趋势,虚线分别表示 Jaccard、Ochiai、Tarantula 和 SBI

方法所能够定位出的错误比例随代码检查率的变化趋势.相同代码检查率下,定位出错误的比例越高说明方法越有效.

从图 3 中可以看出,在 Unix 的 3 个程序上,AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法在定位效果上都要好于原始方法.比如,在 10%的代码检查率范围内,AM-Jaccard 的定位结果为 74.5%,而 Jaccard 的结果为 66.1%;AM-Ochiai 的定位结果为 73.9%,而 Ochiai 的结果为 67.6%;AM-Tarantula 方法能够定位出 69.1%的错误,而 Tarantula 方法能够定位出 59.8%的错误;同样的,AM-SBI 的定位结果为 63.5%,而 SBI 的结果为 54.3%.

对于 flex、grep 和 gzip 3 个 Unix 程序,从图 3 可以看出,在代码检查率很低时(比如 5%),本文的方法和其它方法相比并没有明显优势.这种情况产生的原因是,错误定位所需的代码检查率越低,说明错误代码与错误可疑度之间的表现关系越明显,如 2.2 节中的情形 1.通常情况下,这种错误的触发是很容易被观察并检测到的.然而,对于需要较大代码检查率才能定位出的错误,错误可疑度很高的代码很多情况下并没有错误,而是受到了程序感染状态的传递或者代码执行后产生了能够触发错误的数据状态,如 2.2 节中的情形 2 和情形 3.定位这种错误往往需要沿着程序的执行路径进行审查,会花费更大的代价,本文方法正是在定位此类错误时效果显著.

除此之外,我们还统计了 AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法在不同的程序集样本上相对于 Jaccard、Ochiai、Tarantula 和 SBI 方法的有效性增益.在评估指标下,方法的有效性体现在定位错误所需检查的代码语句的比例,因而,方法有效性增益定义为基于关联挖掘的方法所带来的代码检查率的降低占原方法中代码检查率的比例.以 AM-Jaccard 和 Jaccard 为例进行说明,AM-Jaccard 相对于 Jaccard 的有效性增益为 $\Delta E=\dfrac{\sum_{i=1}^{n}(1-Effectiveness_{\mathrm{AM-J}}(v_i)/Effectiveness_{\mathrm{J}}(v_i))}{n}$,其中 n 为错误版本数量,$Effectiveness_{\mathrm{AM-J}}(v_i)$ 和 $Effectiveness_{\mathrm{J}}(v_i)$ 分别指定位 v_i 版本中的错误所需的代码检查率.按此方法计算得到的分析结果如表 2 所示.

表 2　本文方法在不同 Unix 程序集样本上的有效性增益分析

	AM-Jaccard	AM-Ochiai	AM-Tarantula	AM-SBI
<−0.5%	13	13	12	12
[−0.5%,0.5%]	62	64	59	59
>0.5%	20	18	24	24
<−5%	13	13	12	12
[−5%,5%]	63	64	60	60
>5%	19	18	23	23
<−50%	8	8	10	10
[−50%,50%]	72	74	72	72
>50%	15	13	13	13

从表 2 中可以看出,在 Unix 程序集上,AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法相比 Jaccard、Ochiai、Tarantula 和 SBI 方法能够提高错误定位的准确率,比如,AM-Jaccard 方法相比 Jaccard 能够将其中的 15 个错误的定位有效性提高 50%以上,同时,只对 8 个程序产生了副作用(Jaccard 的有效性相比 AM-Jaccard 高 50%以上).从表 2 中可以观察到,AM-方法在 Tarantula,Ochiai,和 SBI 方法上表现出了类似的效果,这里不再赘述.

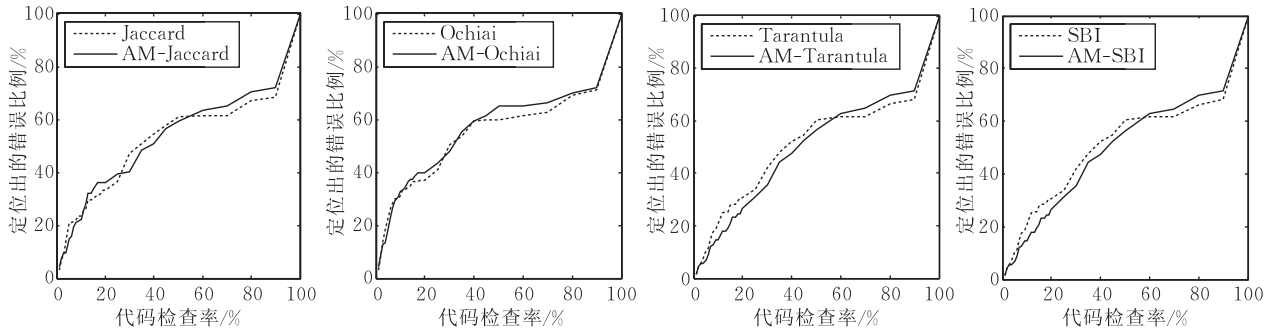

图 4　Siemens 实验结果总体对比

图 4 是 Siemens 程序实验结果,图中横坐标和纵坐标具有和图 3 中横纵坐标相同的含义.图 4 表示,在 Siemens 程序集的实验上,AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法相比其原始方法并没有表现出同样稳定的有效性增益效果.例如,相比 Jaccard 和 AM-Jaccard,后者在 10%～

30%和50%～100%的代码检查率情况下表现出优势，而前者在0%～10%及30%～50%的代码检查率情况下表现出优势.在Ochiai方法上，本文提出的关联挖掘方法(以下简记为AM-方法)的使用表现出同样的效果；而在Tarantula和SBI方法上，AM-方法的使用甚至表现出有效性的退化趋势.

我们进而用表3对Siemens程序的结果进行进一步分析(表3的含义和表2类似).从表3中还可以得到与图4一致的分析结果：在Siemens程序集上，AM-Jaccard、AM-Ochiai、AM-Tarantula和AM-SBI方法相比Jaccard、Ochiai、Tarantula和SBI方法没有明显的有效性增益，通过500%增益的数目统计比较①，我们发现，对于及个别的错误版本(少于5个)，Ochiai程序的有效性甚至能够达到AM-Ochiai程序有效性的500%以上.这样的意外结果显然是造成本文方法在Siemens程序上效果欠佳的原因.

表3 本文方法在不同Siemens程序集样本上的有效性增益分析

	AM-Jaccard	AM-Ochiai	AM-Tarantula	AM-SBI
<−0.05%	61	65	66	66
[−0.05%,0.05%]	37	36	29	29
>0.05%	29	26	32	32
<−5%	56	62	60	60
[−5%,5%]	42	39	35	35
>5%	29	26	32	32
<−500%	1	5	2	2
[−500%,500%]	126	122	125	125
>500%	0	0	0	0

这一原因我们将在第6节中进行分析.

5.4 不同程序的独立结果分析和相互比较

程序结构的不同可能会影响实验结果，为了更好地说明本方法的有效性，我们又分别统计了各种方法应用在flex、grep和gzip等程序上的实验结果②，如图5所示.

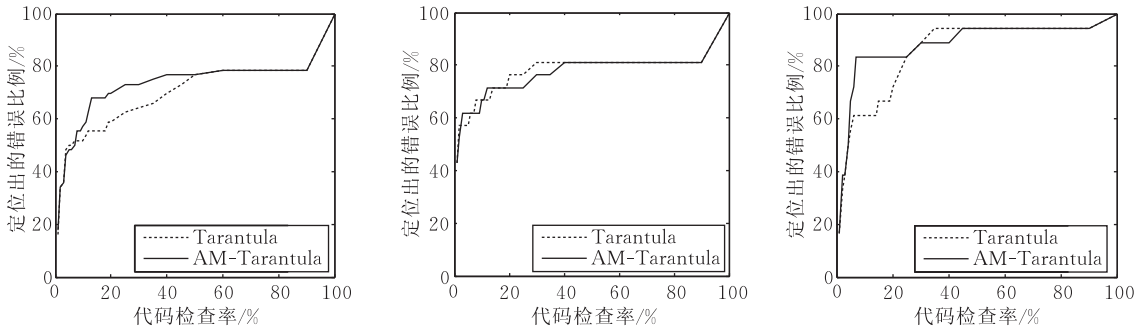

图5 不同程序的实验结果独立对比

从图5可以看出，各个程序的实验结果与总体结果类似，本文方法在flex和gzip程序样本上表现出优势，而在grep程序上尚存有不足.针对这一问题，我们将在本文的第6节通过代码架构、错误类型等展开详细分析.

同时，我们采用t检验[26]对Jaccard和AM-Jaccard、Ochiai和AM-Ochiai、Tarantula和AM-Tarantula以及SBI和AM-SBI在不同程序中的结果进行了假设检验.

如表4所示，在Unix程序上，Jaccard同AM-Jaccard的t-test分析结果为0.0075.这可以直观地解释为，Jaccard方法在Unix程序上的效果和AM-Jaccard方法在Unix程序上的效果来自同一分布的可能性为0.75%.如果我们把常用的5%作为阈值[26]，可以排除“二者没有显著区别”的假设.因为我们在图3，表3和图5中都观察到，AM-Jaccard方法相比Jaccard方法在平均意义上具有一般性优势，因此我们通过假设检验得出结果，在Unix程序上，AM-Jaccard方法相比Jaccard方法具有假设检验含义中的“显著”优势.AM-方法在Ochiai，Tarantula和SBI方法上的效果可以类似地解释.

表4 不同程序的实验结果的假设检验分析

(a) Unix程序

AM-Jaccard同Jaccard比较	AM-Ochiai同Ochiai比较	AM-Tarantula同Tarantula比较	AM-SBI同SBI比较
0.0075	0.0048	0.0080	0.0079

(b) Siemens程序

AM-Jaccard同Jaccard比较	AM-Ochiai同Ochiai比较	AM-Tarantula同Tarantula比较	AM-SBI同SBI比较
0.3578	0.0504	0.0282	0.0282

在表4中，我们同时观察到，如果以5%作为阈值，Jaccard和AM-Jaccard在Siemens程序上，以及

① 由于原始方法和其AM-方法在UNIX程序和Siemens程序上的差别较大，在表2和表3中，我们采用了不同范围的增益值进行比较，以便更好地对结果进行分析和展示.

② 限于文章篇幅(不同技术应用于不同程序共需要40幅图)，同时考虑到Tarantula方法的代表性，实验结果中仅列出了AM-Tarantula和Tarantula方法的对比.

Ochiai 和 AM-Ochiai 在 Siemens 程序上“没有显著区别”的假设不能被排除. 因此，我们认为，Jaccard 和 AM-Jaccard 在 Siemens 程序上以及 Ochiai 和 AM-Ochiai 在 Siemens 程序上的效果没有显著区别. 这和我们对图 4 的分析是一致的. 同时，由于我们在图 4 中观察到 AM-方法在 Tarantula 和 SBI 方法上的使用会带来效果退化的趋势，结合二者的 t-test 结果(均为 0.0282)，我们可以以 5% 为阈值拒绝“二者没有显著区别”的假设，并得出结论——在 Siemens 方法上，AM-方法在 Tarantula 和 SBI 方法上的使用将降低 Tarantula 和 SBI 的差错效果. 这一现象与我们在表 3 和图 4 中的观察结果是一致的. 我们将在第 6 节中对其原因进行分析.

我们进一步分析 AM-方法在不同方法上的应用是否有一致的效果增益. 表 5 用 t-test 检查了 AM-方法在不同方法上使用时是否有显著的差异. 应用同上一节中类似的分析方法，我们可以得出以下的结论. 以 5% 为阈值进行分析，在 Unix 程序上，AM-方法在 Jaccard、Ochiai、Tarantula 和 SBI 上面分别使用时，会带来普遍的效果增益，并无“显著”区别. 同理，在 Siemens 程序上，AM-方法在 Jaccard 上的使用和在其它 3 种方法上的使用有显著的效果区别；在 Ochiai 上的使用和在 SBI、Tarantula 上的使用没有显著的效果区别；在 SBI 和 Tarantula 上使用时效果完全一致. 这样的结果与图 4 的观察结果是一致的.

表 5　不同方法间的效果增益比较的假设检验分析

(a) Unix 程序

	AM-Jaccard的增益	AM-Ochiai的增益	AM-Tarantula的增益
AM-SBI 的增益	0.2901	0.3195	0.2559
AM-Tarantula的增益	0.2901	0.3195	
AM-Ochiai的增益	0.3647		

(b) Siemens 程序

	AM-Jaccard的增益	AM-Ochiai的增益	AM-Tarantula的增益
AM-SBI 的增益	0.0100	0.6095	1.0000
AM-Tarantula的增益	0.0100	0.6095	
AM-Ochiai的增益	0.0389		

结合表 2 和表 3 的数据，通过假设检验，我们得出以下的结论. 在 Unix 程序上，无论 AM-方法在何种方法(Jaccard，Ochiai，Tarantula 或 SBI)上使用，具有普遍的效果增强现象. 在 Siemens 程序上，AM-方法在 Jaccard 和 Ochiai 上效果不明显，也没有达到 t-test 的显著性差异(阈值 0.05)，在 Tarantula 和 SBI 上有副作用(会引起二者的错误定位效果退化).

5.5　针对不同错误的结果分析

之前的研究认为，错误定位方法在代码检查率较低(比如 10%[27] 或 20%[5,14])的范围内的定位准确率更能说明方法的有效性. 为此，我们以 10% 的代码检查率为阈值，以 Tarantula 方法为例，从 Unix 的 3 个程序中选择出那些在 Tarantula 或者 AM-Tarantula 方法下代码检查率大于 10% 同时小于 100% 的单个错误，横向比较这些错误在 AM-Tarantula 和 Tarantula 方法下的代码检查率，按照这种选择方法，共在 Unix 的 3 个程序中选择出 26 个错误，其实验结果如图 6.

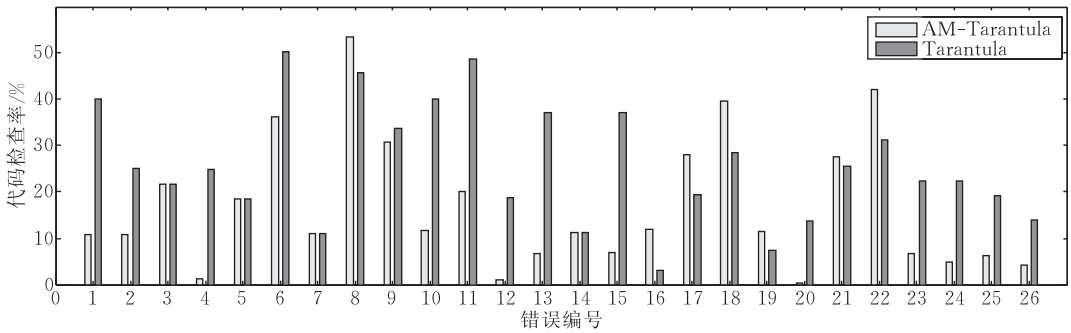

图 6　针对不同错误的代码覆盖率横向比较

如图 6 所示，在选出的 26 个不同错误中，有 15 个错误在 AM-Tarantula 方法下的代码检查率明显低于在 Tarantula 方法下的代码检查率，4 个错误在两种方法下的代码检查率相等，另外有 7 个错误在 AM-Tarantula 方法下的代码检查率高于在 Tarantula 方法下的代码检查率.

在其余的 69 个错误版本中，AM-Tarantula 和 Tarantula 的效果差别均在 10% 以下.

在 Jaccard、Ochiai 和 SBI 方法上，本文方法的应用具有类似的效果；总体来看，本文的方法在

Unix 程序上的大多数情况下具有相比原方法更佳的错误定位效果.

6 讨 论

6.1 本文方法的不足

针对 AM-Jaccard、AM-Ochiai、AM-Tarantula 和 AM-SBI 方法在 Siemens 程序集和 Unix 中的 grep 程序上存在的不足,我们从源代码结构、错误类型、评估指标等多个方面仔细分析了实验结果.得出如下结论:在目前的评估指标下,本文的方法不能很好地处理错误代码和高可疑代码分属不同的函数体的情况.

遵循之前的研究中所采用的执行路径和控制流图等方法[5-19],本文为每一个函数体构建其覆盖向量和覆盖矩阵.一旦高可疑代码与错误代码分属不同的函数体,本文方法则会优先检查高可疑代码所处函数体中的与高可疑代码保持覆盖一致性的代码语句,一定程序上提高了整个函数的检查顺序,相对而言则降低了错误代码的定位准确度.

6.2 案例分析

针对错误代码与高可疑代码分属不同的函数体这一情况,本文有针对性的作了进一步的分析比较实验.针对每一个错误版本,首先,我们分析以 Jaccard、Ochiai、Tarantula 或 SBI 方法得到的具有最高可疑度的代码与实际的错误代码语句是否存在同一函数体,然后统计出高可疑代码语句与错误代码语句不在同一函数体的错误版本数.分析结果如表 6.

表 6 跨函数体的错误版本分析结果

	AM-Tarantula 优于 Tarantula 的结果		Tarantula 优于 AM-Tarantula 的结果	
	faults	diff%	faults	diff%
flex	14	0	3	100
grep	4	0	6	100
gzip	6	0	3	100
print_tokens	1	0	1	100
print_tokens2	2	50	4	100
replace	3	33.3	21	100
schedule	3	33.3	6	100
schedule2	2	100	2	100
tcas	12	8.3	17	100
tot_info	10	0	7	100

在表 6 中,我们以 faults 来表示满足 AM-Tarantula 的结果优于 Tarantula 或者 Tarantula 优于 AM-Tarantula 等条件的错误版本数量,diff%来表示高可疑代码与错误代码分属不同函数体的错误版本占 faults 数量的比例.从表中可以看出,在 Tarantula 优于 AM-Tarantula 的情况下,错误版本的程序中,错误代码与高可疑代码全部分属不同的函数体.

以 grep 为例,grep 共有 21 个错误版本.在其中的 6 个错误版本上,Tarantula 方法要好于 AM-Tarantula 方法,且这 6 个错误版本中的代码语句与高可疑的代码语句都不属于同一函数体.比如,在 grep-v1-fault3 版本中,共有 22 行代码语句的可疑度比错误代码语句高,这 22 行代码语句分属 4 个不同的函数,在这种情况下,本文算法则会优先处理这 4 个函数中的频繁集求解,进而可能导致整体上的效果并不好.

6.3 可改进之处

针对 6.1 节和 6.2 节中的问题,构建不区分函数的全局覆盖向量和覆盖矩阵或许是一种解决高可疑代码与错误代码分属不同函数体的情况,一种较易实现的方法是以全部代码语句的行数作为索引,构建全局覆盖向量,这样可以确保频繁集求解不单独针对某一个函数,从而可能提高错误定位的准确度.

从实际效果上看,本文方法的有效性在如下几个方面需要进一步讨论.

首先,本文的实验首先使用 gcov 来获取代码的覆盖信息,然后再结合静态分析获取执行路径.gcov 获得的覆盖信息不能反映代码的先后执行顺序,因此对于程序结构的执行路径分析存在一定误差.但是,在错误类型不是代码缺失型错误时,覆盖向量对于错误代码与高可疑代码之间的覆盖一致性是没有影响的.

其次,本文的方法更适于解决单个错误的错误定位问题.程序中多个错误同时存在的情况会对本文方法有效性产生不利影响,比如,多个错误的存在可能会使得求解出的频繁集仅包含目标基本块.一个解决该问题的方法是面向错误的程序失效分类[28],如果能够针对不同的错误将程序失效分类,则本文方法可以体现出实验结果中的优势.

程序中总有一部分基本块是某些代码执行时所必须覆盖的,如控制流图中的控制依赖节点[29],如何甄别频繁集中诸如控制依赖基本块对应的分量以进一步缩小待检查代码的范围,是目前困扰我们的主要问题.此外,程序执行中的巧合性正确[25]以及测试用例的相似性[30]一直是困扰错误定位的重要问题,本文的方法仍然不能避免这种影响,我们将在

后续的工作中开展这方面研究.

7 结　论

本文以提高 CBFL 方法的准确性为出发点，首先分析了 CBFL 方法准确性有待提高的原因，随后提出失效执行规则，然后在执行规则的指导下，建立基于覆盖向量的执行轨迹分析模型. 基于此模型，本文提出用于挖掘与高可疑代码相关联的错误代码的频繁集求解算法. 以 SIR 基准程序为对象建立的受控实验证明，相比之前的研究，本文方法在一定程度上可以改进错误定位结果.

参　考　文　献

[1] Liu Ke, Shan Zhi-Guang, Wang Ji, He Ji-Feng, Zhang Zhao-Tian, Qin Yu-Wen. Overview on major research plan of trustworthy software. Bulletin of National Natural Science Foundation of China, 2008, 22(3): 145-151(in Chinese)
(刘克，单志广，王戟，何积丰，张昭田，秦玉文. “可信软件基础研究”重大研究计划综述. 中国科学基金，2008，22(3)：145-151)

[2] Agrawal H, Horgan J, Lodon S, Wong W. Fault localization using execution slices and dataflow tests//Proceedings of the 6th International Symposium on Software Reliability Engineering. Toulouse, France, 1995: 143-151

[3] Jones J A, Harrold M J. Empirical evaluation of the Tarantula automatic fault-localization technique//Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE 2005). Long Beach, California, USA, 2005: 273-282

[4] Abreu R, Zoeteweij P, Gemund A. On the accuracy of spectrum-based fault localization//Proceedings of Testing: Academic and Industrial Conference, Practice and Research Techniques. Windsor, UK, 2007: 89-98

[5] Zhang Z, Chan W K, Tse T H. Capturing propagation of infected program states//Proceedings of the 17th International Conference on Foundation of Software Engineering (FSE/ESEC 2009). Amsterdam, Nederland, 2009: 43-52

[6] Renieris M, Reiss S. Fault localization with nearest neighbor queries//Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003). Montreal, Canada, 2003: 30-39

[7] Mei Hong, Wang Qian-Xiang, Zhang Lu, Wang Ji. Software analysis: A road map. Chinese Journal of Computers, 2009, 32(9): 1696-1710(in Chinese)
(梅宏，王千祥，张路，王戟. 软件分析技术进展. 计算机学报，2009，32(9)：1696-1710)

[8] Zhang X, Gupta N, Gupta R. Pruning dynamic slices with confidence//Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006). Chicago, USA, 2009: 169-180

[9] Baah G K, Podgurski A, Harrold M J. The probabilistic program dependence graph and its application to fault diagnosis//Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008). Seattle, WA, USA, 2008: 189-200

[10] Cleve H, Zeller A. Locating causes of program failures//Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). Missouri, USA, 2005: 342-351

[11] Jones J A, Harrold M J. Visualization of test information to assist fault localization//Proceedings of the International Conference on Software Engineering (ICSE 2002). Orlando, USA, 2002: 467-477

[12] Liblit B, Aiken A, Zheng A X, Jordan M I. Bug isolation via remote program sampling//Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003). New York: ACM Press, 2003: 141-154

[13] Liblit B, Naik M, Zheng A X, Aiken A, Jordan M I. Scalable statistical bug isolation//Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005). New York: ACM Press, 2005: 15-26

[14] Liu C, Fei L, Yan X F, Han J W, Midkiff S. Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering, 2006, 32(10): 1-17

[15] Yu Y, Jones J A, Harrold M J. An empirical study of the effects of test-suite reduction on fault localization//Proceedings of the 30th International Conference on Software Engineering (ICSE 2008). Leipzig, Germany, 2008: 201-210

[16] Hao D, Zhang L, Xie T, Mei H, Sun J. Interactive fault localization using test information. Journal of Computer Science and Technology, 2009, 24(5): 962-974

[17] Hao D, Zhang L, Xie T, Sun J, Mei H. Test input reduction for result inspection to facilitate fault localization. Journal of Automated Software Engineering, 2010, 17(1): 5-31

[18] Wong E, Qi Y, Effective program debugging based on execution slices and inter-block data dependency. Journal of Systems and Software, 2006, 79(2): 891-903

[19] Jiang L, Su Z. Context-aware statistical debugging: From bug predictors to faulty control flow paths//Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE 2007). Atlanta, USA, 2007: 184-193

[20] Voas J. PIE: A dynamic failure-based technique. IEEE Transactions on Software Engineering, 1992, 18(8): 717-727

[21] Durães A, Madeira H. Emulation of software faults: A field data study and a practical approach. IEEE Transactions on Software Engineering, 2006, 32(11): 849-867

[22] Ball T, Larus J R. Efficient path profiling//Proceedings of

the International Symposium on Microarchitecture. Paris, France, 1996: 46-57

[23] Debroy V, Wong E, Xu X, Choi B. A grouping-based strategy to improve the effectiveness of fault localization techniques//Proceedings of the 10th International Conference on Quality Software (QSIC 2010). Zhangjiajie, China, 2010: 13-22

[24] Do H, Elbaum S G, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005, 10(4): 405-435

[25] Wang X, Cheung S C, Chan W K, Zhang Z. Taming coincidental correctness: Refine code coverage with context pattern to improve fault localization//Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). Vancouver, Canada, 2009: 45-55

[26] Zhang Z, Chan W K, Tse T H, Yu Y T, Hu P. Non-parametric predicate-based fault localization. Journal of Systems and Software, 2011, 84(6): 885-905

[27] Xu J, Chan W K, Zhang Z, Tse T H. A Dynamic Fault Localization Technique with Noise Reduction for Java Programs//Proceedings of the 11th International Conference on Quality Software. Madrid, Spain, 2009: 171-180

[28] Liu C, Zhang X, Han J. A systematic study of failure proximity. IEEE Transactions on Software Engineering, 2008, 44(6): 826-843

[29] Thummalapenta S, Xie T. Alattin: Mining alternative patterns for detecting neglected conditions//Proceedings of the 24th International Conference on Automated Software Engineering (ASE 2009). Auckland, New Zealand, 2009: 283-294

[30] Hao D, Zhang L, Pan Y, Mei H, Sun J. On similarity awareness in testing-based fault localization. Journal of Automated Software Engineering, 2008, 15(2): 207-249

ZHAO Lei, born in 1985, Ph. D. candidate. His research interests include software testing and debugging.

WANG Li-Na, born in 1964, professor, Ph. D. supervisor. Her research interests include network security, trusted computing.

GAO Dong-Ming, born in 1990, M. S. candidate. His research interests include software analysis and debugging.

ZHANG Zhen-Yu, born in 1979, Ph. D., associate researcher. His researches mainly focus on software engineering (software testing and software debugging, in particular).

XIONG Zuo-Ting, born in 1988, M. S. candidate. Her research interests include software testing and debugging.

Background

This paper aims to bring out the technique for efficient and effective fault localization in complex software.

Software debugging is a tedious, challenging and error-prone process in software development. Fault localization is a vital step of debugging. Coverage based fault localization (CBFL) techniques filter statements of programs which are unrelated to bugs by comparing the execution statistics of passed executions and failed executions. The entities that executed during the failed executions more likely contain bugs than entities that executed during passed executions. If the number of failed executions that executed a certain entities is larger, the entities are much more likely containing bugs. This is the main idea of CBFL techniques.

However, the fault suspiciousness is computed individually among such CBFL techniques. In addition, with the impact of random test cases, the coverage of individual program entity cannot reflect the whole complex program executions. Therefore, the individual coverage information to some extent simplifies the program executions, which may lead to the inaccurate results of CBFL techniques. In this paper, we propose the rules of program failures and design the execution analysis model based on the coverage of different executions. By computing the frequent item for statements with high suspiciousness, we also propose the technique of mining fault-prone statements. The controlled experiments based on the SIR benchmarks indicate that our technique is promising.

This work was funded by the Chinese National Natural Science Foundation (Nos. 90718006, 60970114, 61003027, 61073006), and the Research Fund for Excellent Doctoral Student granted by Chinese Ministry of Education.