一种基于程序变异的软件错误定位技术

贺 韬[1),2)] 王欣明[3)] 周晓聪[1)] 李文军[3)] 张震宇[4)] 张成志[2)]

[1)](中山大学信息科学与技术学院 广州 510275)

[2)](香港科技大学计算机科学及工程学系 香港)

[3)](中山大学软件学院 广州 510275)

[4)](中国科学院软件研究所 北京 100190)

摘 要 发现软件不能正常运行后,如何定位错误代码在程序中的位置是软件开发一个众所周知的难点.最近许多软件自动调试技术通过分析成功和失败测试用例的覆盖信息辅助程序员定位错误代码,但这些技术的准确率会受到偶然性成功测试用例的影响.偶然性成功测试用例执行了错误代码,但却没有引发失败的测试结果.研究表明这种测试用例在实际测试中广泛存在,而它们的存在会显著降低错误定位的准确率.针对此问题,文中提出一种称为 Muffler 的技术.Muffler 使用程序变异分析来修正错误代码定位结果,以提高定位的准确率.文中利用 8 个在错误代码定位研究领域广泛使用的基准程序验证了 Muffler 的有效性.实验结果表明,与传统错误代码定位技术相比,Muffler 能减少程序员 50.26%的错误定位代价.

关键词 软件调试;错误定位;程序变异分析;软件工程

中图法分类号 TP312 **DOI 号** 10.3724/SP.J.1016.2013.02236

A Software Fault Localization Technique Based on Program Mutations

HE Tao[1),2)] WANG Xin-Ming[3)] ZHOU Xiao-Cong[1)] LI Wen-Jun[3)]

ZHANG Zhen-Yu[4)] CHEUNG Shing-Chi[2)]

[1)](*School of Information Science and Technology, Sun Yat-Sen University, Guangzhou* 510275)

[2)](*Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China*)

[3)](*School of Software, Sun Yat-Sen University, Guangzhou* 510275)

[4)](*Institute of Software, Chinese Academy of Sciences, Beijing* 100190)

Abstract Recent fault localization techniques leverage program coverage of both passed test runs and failed test runs to reduce the high cost of debugging. The effectiveness of such techniques can be adversely affected by coincidental correctness, which occurs in a passed test run when a fault has been executed but no failure is detected. Studies have shown that coincidental correctness is a common phenomenon and its occurrence can significantly reduce the effectiveness of fault localization. In this paper, a fault localization technique named Muffler is proposed, which uses mutation analysis to address this problem and improve fault localization. Muffler systematically mutates statements in a faulty program and estimates their likelihood of being faulty based on both coverage and how mutation affects the outcome of passed test cases. Experiments on eight benchmark programs widely used in fault localization are conducted to evaluate our method. Results indicate that Muffler can help programmers locate faults effectively with a reduction of 50.26% in code examination effort.

Keywords software debugging; fault localization; program mutation analysis; software engineering

收稿日期:2012-12-03;最终修改稿收到日期:2013-08-20.本课题得到国家自然科学基金(6103027)、中山大学中央高校基本科研业务费专项资金(10LGZD05,1LGPY39)、香港研究资助局项目(61210)和国家科技重大专项经费(2012ZX01039-004)资助.**贺 韬**,男,1987 年生,硕士研究生,主要研究方向为软件错误定位. E-mail: elfinhe@gmail.com. **王欣明**,男,1980 年生,博士,主要研究方向为软件测试. **周晓聪**(通信作者),男,1971 年生,博士,副教授,主要研究方向为软件测试. E-mail: isszxc@mail.sysu.edu.cn. **李文军**,男,1966 年生,博士,教授,主要研究领域为软件工程. **张震宇**,男,1979 年生,博士,副研究员,主要研究方向为软件测试. **张成志**,男,1962 年生,博士,教授,主要研究领域为软件工程.

1 引　言

最近许多软件自动调试技术通过分析成功和失败测试用例的覆盖信息辅助程序员定位错误代码，例如 Jaccard[1]、Tarantula[2]、Ochiai[3] 和χDebug[4] 等. 这些技术统称为基于覆盖信息的错误代码定位(Coverage-Based Fault Localization, CBFL[5])技术. CBFL 技术通常从成功和失败测试用例的运行中收集程序代码的覆盖信息(分析代码的粒度可有不同选择，如语句[1-4,6-7]、分支[8-9]或函数等)，然后根据这些覆盖信息，为每段代码计算错误疑似度，也即程序代码的覆盖信息与失败运行结果的关联度. 直观地说，一段代码如果经常被失败用例覆盖，但却很少被成功用例覆盖，则这段代码的错误疑似度就很大. 程序员可按照错误疑似度由高至低的顺序检查程序代码以找到错误位置.

尽管 CBFL 技术取得了初步成功[7,10-11]，但其准确率仍受到许多负面因素影响. 已有研究[10,12]指出，偶然性成功测试用例(Coincidental Correctness)是最重要的负面因素之一. 偶然性成功测试用例指的是这样的成功测试用例：它覆盖了程序的错误代码，但却未引发失败的运行结果[13]. 偶然性成功测试用例的出现会显著影响 CBFL 技术的准确率，因为它们的出现降低了错误代码和失败运行结果的关联度. 同时研究[5,14-15]指出，偶然性成功在实际测试中广泛存在. 因此降低偶然性成功测试用例对提高 CBFL 技术的准确率很有意义.

然而降低偶然性成功测试用例的负面影响是一个很困难的问题，因为很难准确区分哪些成功测试用例是偶然性成功用例，除非已经知道程序错误代码的位置，但这正是调试的目标. 目前仅有少量工作尝试解决此问题. Wang 等人[15]提出一种使用上下文模式来提炼覆盖信息. 他们假设程序员预先知道程序的错误类型并通过这些错误类型的模式来排除偶然性成功用例引入的错误代码覆盖. 但在实际开发中，该假设不一定成立. Masri 等人[16]提出一种区分偶然性成功测试用例的启发式方法. 但该方法并不十分准确，实验结果表明该方法会产生超过 41.3%的错判率，以致对提高错误代码定位技术准确率没有明显的帮助.

针对已有工作的不足，本文提出一种新的思路，不需要程序员预先判断程序的错误类型，也不需要区分哪些成功测试用例是偶然性成功测试用例. 我们的核心想法是利用程序变异分析(Mutation Analysis)降低偶然性成功测试用例的影响. 该想法基于这样的观察：对于一个成功测试用例，假如变异程序的错误代码后再执行它(即把原来的代码替换为另一段不同的代码)，那么不管它是否是偶然性成功测试用例，测试结果趋向于不变，因为这只是将一个错误改为另一个错误，并没有增加程序的错误数；然而，若变异程序的正确代码，那么这个用例的测试结果很可能就从成功变为失败，因为这时程序多了一个额外错误，也就增加了出错的概率. 根据这种观察，我们认为不同代码段变异后引发的由成功变为失败的测试用例数提供了新的错误定位线索. 相比于代码覆盖，这种线索较少受偶然性成功测试用例的影响，用它修正代码覆盖所推导出的结果，可减少偶然性成功测试用例的负面影响，从而更准确地评估代码的错误疑似度，提高错误定位的准确率.

我们开发了一套称为 Muffler 的软件实现了上述想法，应用在错误定位研究中广泛使用的 Siemens 基准程序集[14]的 7 个程序共 123 个错误程序版本上，并与多个 CBFL 技术进行了实验比较. 实验结果表明，与目前最好的 CBFL 技术相比，Muffler 把平均代码检查率从 19.34%降到 9.62%，减少比例为 50.26%(=100%−(9.62%/19.34%)). 我们还使用了一个实际应用程序 space 及其真实错误版本，展示 Muffler 在实际应用中的有效性.

本文的主要贡献有两方面：(1) 提出一种利用程序变异分析提高错误定位准确率的方法；(2) 报告了一组实验及其结果，说明本文的方法比现有 CBFL 技术能更准确地定位程序的错误代码.

本文第 2 节简述现有错误定位技术和程序变异技术；第 3 节在使用完整例子说明本文方法基本思想的基础上，给出 Muffler 系统的总体工作流程和在变异分析方面的设计要点；第 4 节给出验证 Muffler 系统有效性的比较实验及其结果；第 5 节总结全文并展望下一步工作.

2 研究背景

近年来出现了许多CBFL技术，例如Tarantula[2]、Ochiai[3]、χDebug[4] 和 Naish[17] 等. CBFL 技术的核心部分是根据程序代码的覆盖信息和测试结果计算程序代码错误疑似度的公式. 代码覆盖信息的分析粒度可以是语句、分支或函数等，但多数 CBFL 技术针对程序语句进行分析. Naish 等最近的研究[17]表

明，他们提出的公式比 30 多种现有公式能更准确地定位程序错误. 其公式的定义如下：

$$Susp_{\text{Naish}}(S_i)=$$
$$Failed(S_i)\times(TotalPassed+1)-Passed(S_i) \quad (1)$$

其中，$Susp_{\text{Naish}}(S_i)$是语句S_i的错误疑似度，$Failed(S_i)$和 $Passed(S_i)$分别是覆盖语句 S_i的失败和成功测试用例数，$TotalPassed$ 是成功测试用例总数. 由于大部分 CBFL 技术仅在疑似度计算公式上不同，本文以4 种知名的 CBFL 技术，即 Tarantula、Ochiai、χDebug 和 Naish，作为代表进行分析比较，更多技术可参阅文献[17].

程序变异测试与分析(Program Mutation Testing and Analysis)是由 Hamlet[18] 和 DeMillo 等人[19]提出的一种基于错误植入的软件测试技术. 这种技术主要用于衡量测试用例集发现错误的有效性. 变异测试通过在程序中逐个引入符合语法的变化，把原始程序变异成若干变异程序，以检验测试用例集的错误检测能力. 本文主要利用变异测试技术的错误植入能力，在指定程序语句上产生符合语法的程序错误，并利用随之带来的测试结果变化预测该程序语句存在错误的可能性.

程序变异测试与分析在原始程序上应用变异算子(Mutation Operator)以植入错误. 根据变异对象的类型，变异算子可分为语句、运算符、变量、常量等类型[20]；根据变异的行为，可分为替换、插入、删除 3 种类型. 若原始程序和变异程序之间只应用了一次变异算子，则称为一阶变异；若应用了多次则称为高阶变异. 为设计简明与更清晰的分析，本文仅使用一阶变异.

Papadakis 和 Traon 在文献[21]中，尝试利用程序变异提高错误定位的准确率. 其方法是对于程序中的每一个可执行语句 S 应用不同变异算子生成若干变异程序，然后对每个变异程序运行 CBFL 算出一个 S 的错误疑似度，最后把其中的最大值作为 S 的最终错误疑似度. 该研究在 Siemens 程序集上进行了实验，比较了不同程序以及不同规模测试集上的结果，说明了其方法在不同场景下的有效性. 不过，文章并没有解释为何从不同变异程序中得到的最大疑似度赋给各个语句后会具有可比性，也并未深入探讨其方法起作用的原因. 本文提出的 Muffler 技术根据语句变异前后测试结果的变化，尝试去解决现有 CBFL 技术的主要缺陷——偶然性成功的测试用例，不仅提高了现有 CBFL 技术的准确率，也因此解释了其作用的内部机理.

3 基于变异分析的错误定位技术

这一节首先结合一个例子详细说明基于变异分析的错误定位技术的基本思想，然后给出 Muffler 系统的设计与实现，并重点说明 Muffler 系统针对错误定位的要求对程序变异所做的优化.

3.1 基本思想

图 1 给出了各种错误定位技术(包括本文提出的技术)对 Schedule 程序的 v2 版本中一个程序片段所有语句错误疑似度的计算，以说明 CBFL 技术的应用和本文提出的方法的基本思想. 图 1 第一部分给出了该程序片段，注意语句 S_2 和 S_3 上有一个错误，导致错误的队列下标 n. 但错误的 n 值不总是会触发导致错误的程序输出，产生失败的测试结果. 实际上，执行语句 S_2 和 S_3 的 1592 个测试用例中，只有 210 个失败测试用例，其余 1382 个测试用例均为偶然性成功. 图 1 第一部分也给出了成功和失败测试用例总数及覆盖每条语句的成功和失败测试用例数，例如 2440 个成功测试用例有 1798 个覆盖 S_1，而 210 个失败测试用例都覆盖 S_1.

图 1 第二部分给出了 Tarantula、Ochiai、χDebug 和 Naish 的语句错误疑似度计算结果，其中 *susp* 列是每个语句的错误疑似度，r 列是每个语句按错误疑似度由高至低排序的排名. 错误语句的排名一定程度上度量了程序员定位到这个错误需要的代价[22]. 从图 1 我们可看出，这些 CBFL 技术都需要检查 88% 的语句才能定位到错误语句 S_2 和 S_3，因为若按照错误疑似度从高到低的顺序检查语句，除 S_1 外其它语句都需要检查，也即检查语句的总数占总语句数的 88%(≈7/8). 从此例可看到高偶然性成功测试用例率(此例是 1382/2440 = 56.6%)对 CBFL 技术准确率存在着负面影响.

图 1 第三、四部分给出了本文提出的在程序变异分析基础上的语句错误疑似度的计算. 图 1 第三部分给出该程序片段每条语句的变异示例及其影响. 该程序片段中有 8 条语句 $S_1\sim S_8$，分别变异这 8 条语句，每条语句产生 5 个变异程序，总共产生 5×8=40 个变异程序，对 40 个变异程序重新运行原先的测试用例，得到原本成功而变异后失败的测试用例数，如图 1 第三部分的“$Change_{p\to f}(M_{S_i},j)$”所示. 例如，将语句 S_1 中的变量“*block_queue*”加上逻辑非“!”得到对应该语句的第 1 个变异程序 M_{S_1},1，对应地有 $Change_{p\to f}(M_{S_1},1)=1644$ 个原先成功的

测试用例在 M_{S_1},1 上运行变为失败测试用例. 又例如,$Change_{p\to f}(M_{S_3},2)=1116$ 表示有 1116 个原先成功的测试用例在语句 S_3 的第 2 个变异程序 M_{S_3},2 上运行变为失败测试用例. 注意,图 1 第三部分变异示例仅给出了每条语句的一个变异,其它 4 个变异受篇幅所限没有一一列出.

图 1 第四部分"$Impact$"列统计了变异每条语句后成功测试用例变化的平均数,我们将其定义为对应语句 S_i 的变异影响 $Impact(S_i)$:

$$Impact(S_i)=\sum_{j=1}^{m}Change_{p\to f}(M_{S_i},j) \quad (2)$$

其中 m 是每条语句的变异个数,在这里的例子中 $m=5$. 过去的研究[23]与我们的初步实验显示,$m=5$ 的抽样已经能得到接近全部变异程序的结果. 可以看到错误语句 S_2 和 S_3 的变异影响比其它语句低,这印证了前面所说的基本观察:变异错误的程序语句更趋向于保持成功测试用例的测试结果,而变异正确的程序语句则更趋向于改变成功测试用例的测试结果.

第一部分

	程序片段	TotalPassed 2440 Passed(s)	TotalFailed 210 Failed(s)
S_1	if (*block_queue*){	1798	210
S_2	*count*=*block_queue*->*mem_count*+1; /*fault: insert '+1'*/	1382	210
S_3	*n*=(int)(count**ratio*); /* fault: missing '+1' */	1382	210
S_4	*proc*=find_nth(*block_queue*, *n*);	1382	210
S_5	if (*proc*){	1382	210
S_6	*block_queue*=del_ele(*block_queue*, *proc*);	1358	210
S_7	*prio*=*proc*->*priority*;	1358	210
S_8	*prio_queue*[*prio*]=append_ele(*prio_queue*[*prio*], *proc*);}}	1358	210
	定位S_2和S_3和代码检查花费:		

第二部分

Tarantula		Ochiai		χDebug		Naish	
*susp**	*r***	*susp*	*r*	*susp*	*r*	*susp*	*r*
0.576	8	0.32	8	205.41	8	510812	8
0.638	7	0.36	7	205.83	7	511228	7
0.638	7	0.36	7	205.83	7	511228	7
0.638	7	0.36	7	205.83	7	511228	7
0.638	7	0.36	7	205.83	7	511228	7
0.642	3	**0.37**	3	**205.85**	3	**511252**	3
0.642	3	**0.37**	3	**205.85**	3	**511252**	3
0.642	3	**0.37**	3	**205.85**	3	**511252**	3
88%		88%		88%		88%	

第三部分

	每条语句的变异示例	$Change_{p\to f}(M_{S_i},1)$	$Change_{p\to f}(M_{S_i},2)$	$Change_{p\to f}(M_{S_i},3)$	$Change_{p\to f}(M_{S_i},4)$	$Change_{p\to f}(M_{S_i},5)$
S_1	if (**!***block_queue*){	1644	1798	1101	1101	1644
S_2	*count*=*block_queue*->*mem_count***!=**1;	249	1097	1097	249	1382
S_3	*n*=(int)(*count***<=***ratio*);	249	1116	1101	494	1101
S_4	*proc*=find_nth(*block_queue*, ***ratio***);	1088	638	1136	744	1382
S_5	if (**!***proc*){	1136	1358	1101	1382	1101
S_6	*block_queue*=del_ele(*block_queue*, *proc***-1**);	1123	349	1358	814	1358
S_7	*prio***/=***proc*->*priority*;	1358	1358	1101	1101	1358
S_8	*prio_queue*[*prio*]=append_ele(*prio_queue*[**__MININT__**], *proc*);}}	598	598	1138	1358	1101

第四部分

Muffler		
Impact(S_i)	*susp*	*r*
1457.6	509 354.4	8
814.8	**510 413.2**	2
812.2	**510 415.8**	2
997.6	510 230.4	5
1215.6	510 012.4	6
1000.4	510 251.6	4
1255.2	509 996.8	7
958.6	510 293.4	3
定位S_2和S_3和代码检查花费:	25%	

**susp*: 每条语句的错误疑似度; ***r*: 每条语句根据错误疑似度从高到低的排序

图 1　各种错误定位技术对"schedule"程序的错误版本 v2 的一个程序片段的错误疑似度计算

根据这种观察,我们认为在计算疑似度的时候引入变异影响可提升 CBFL 技术的准确性. 为此,我们在 Naish 错误疑似度计算式(1)的基础上加入变异影响,得到如下计算公式:

$$Susp_{\text{Muffler}}(S_i)=Susp_{\text{Naish}}(S_i)-Impact(S_i) \quad (3)$$

式(3)是我们开发的 Muffler 软件目前采用的公式. 实际上,也可将变异影响用于其它 CBFL 技术的计算公式. 图 1 第四部分的 $Susp$ 列给出了例子程序片段,每条语句用式(3)计算出的错误疑似度,而 r 列给出了对应的排名. 按照该排名,只检查 25% 的语句即可定位到错误语句 S_2 和 S_3,比前述 CBFL 技术 88% 的错误定位代价有显著提高.

3.2　Muffler 系统设计

我们设计并实现了名为 Muffler 的原型系统. 该系统可以自动地生成变异程序、嵌入监控代码、运行测试用例并收集覆盖信息. Muffler 利用变异影响计算每个可执行语句的疑似度,给出疑似度由高至低的语句排序列表.

图 2 给出了 Muffler 原型系统的数据流图. 类似其它错误定位工具,Muffler 接收错误程序和对应的测试用例集为输入,最终输出帮助定位错误的疑似语句排序列表.

Muffler 的工作流程如下:(1) 给定一个错误程序和相应的测试用例集,Muffler 首先运行测试用例,并收集覆盖信息和测试结果;(2) 基于减少变异语句数目、提高运行效率的考虑,Muffler 选出覆盖最多失败用例的语句作为变异候选语句;(3) 对每个变异候选语句,Muffler 逐个应用变异算子以生成所有可能的变异程序,然后从中随机选择固定数量的一部分来衡量该语句的变异影响;(4) Muffler

在测试用例集上运行每个被选择的变异程序，收集测试结果的变化，计算每个语句的变异影响；(5) Muffler 基于变异影响和原始程序的覆盖信息，计算每个语句的错误疑似度.

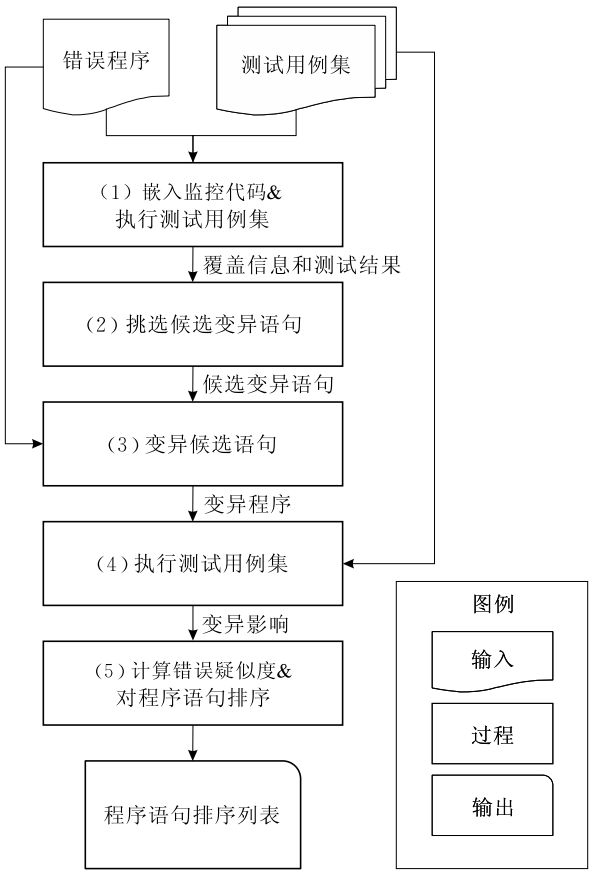


图 2 Muffler 系统的数据流图

3.3 Muffler 系统中变异分析的优化

针对错误定位的需要，Muffler 系统对需要变异的语句数进行了控制，并对初步变异后的程序进行抽样以减少需要运行测试用例的变异程序数从而提高整个系统的效率.

我们的研究目标是要提升 CBFL 技术的准确率，因此只对那些错误疑似度排在错误语句之上的正确语句感兴趣，而没有必要考虑程序中所有的可执行语句. 例如，对于没有被任何失败用例覆盖的语句，并没有必要做变异分析，因为它们不可能是错误语句. CBFL 技术对这些语句也总是赋予最低的错误疑似度.

目前，Muffler 系统的筛选策略是选出被超过 K 个失败测试用例覆盖的语句进行变异分析；K 可以由程序员根据触发失败用例的错误个数指定. 若程序员明确知道所有失败用例由同一个程序错误触发，例如在本文的实验中，K 就可以取失败用例总数 $TotalFailed$. 此策略可显著减少做变异分析的语句数，从而提高整个 Muffler 系统的效率.

根据一条语句中语法单元的不同，可用的变异算子也不同，产生的变异程序数目也不同. 为统一变异每条语句生成的变异程序数，Muffler 使用由 Acree[6] 和 Budd[24] 提出的变异程序抽样技术. 变异程序抽样技术首先会生成所有可能的变异程序，然后随机选出 $x\%$ 的变异程序运行. Mathur 和 Wong 的研究[23] 指出，测试时使用 10% 的抽样率已经能得到和使用全部变异程序接近的结果. 初步的实验也表明，此结论对于程序调试同样有效；并且基于选择变异技术[25]，仅使用全部变异算子的一个有效子集，实验结果比使用全部变异算子的结果略差，我们发现这是由于对于部分错误语句无法生成对应的变异程序. 因此，Muffler 中对每个可执行语句应用所有变异算子，随机选出 $m=5$ 个(抽样率约等于 10%)变异程序进行实验.

4 Muffler 系统的实验评估

本节报告 Muffler 与 CBFL 技术的 4 个代表(Tarantula[2]、Ochiai[3]、χDebug[4] 和 Naish[17])的实验比较结果. 这里首先给出用于实验的目标程序和实验环境，然后给出实验结果及相应的分析，并讨论了 Muffler 系统的时间效率，最后给出在一个实际应用程序和真实错误上对 Muffler 系统的实用性和准确性所做的进一步验证.

4.1 实验目标程序与实验环境

我们使用 Siemens 程序集的 7 个程序，分别是 tcas，tot_info，schedule，schedule2，print_tokens，print_tokens2 和 replace. 同时还使用真实的程序 space 观察 Muffler 方法在真实错误上的有效性. 之前的错误定位研究[5,7,22,26-27] 也使用这些程序进行实验比较与分析. 所有程序及其错误版本均从软件基础设施库的网站[28] 下载. 表 1 列出了这些程序的基本情况.

表 1 实验目标程序

程序集	版本数	可执行语句数	测试用例数	LOC
tcas	41	63～67	1608	133～137
tot_info	23	122～123	1052	272～273
schedule	9	149～152	2650	290～294
schedule2	10	127～129	2710	261～263
print_tokens	7	189～190	4130	341～343
print_tokens2	10	199～200	4115	350～355
replace	32	240～245	5542	508～515
space	38	3633～3647	13585	5882～5904

Siemens 程序集总共包含 132 个错误程序版本，但我们排除了其中 9 个版本：程序 replace 的版本 v27 和程序 schedule2 的版本 v9 因为没有失败测试用例而被排除；程序 schedule 的版本 v1、v5、v6 和 v9，程序 print_tokens2 的版本 v10，程序 replace 的版本 v19 和 v27 因为失败用例发生了段错误(Segmentation Fault)未能收集到完整的覆盖信息而被排除. 最终我们使用 123 个错误版本进行实验. 对于变异程序运行测试用例时发生的段错误，我们将该测试用例标记为失败.

我们的实验环境是一台配置 2.93 GHz Intel Core i3 CPU 和 4 GB 物理内存的计算机；操作系统是 Ubuntu 10.04.2 LTS，使用 2.6.32-29-generic 版本的 Linux 内核. 我们使用 Python 2.6.5 编写 Muffler 系统的核心代码，调用 gcc 4.4.3 和 gcov 4.4.3 编译程序和收集覆盖信息. 我们利用 Proteum[29] 提供的所有变异算子生成变异程序并使用变异程序抽样技术筛选出 $m=5$ 个变异程序作为代表.

4.2　实验结果与分析

Naish 等人观察到 Naish 公式在现有的 CBFL 疑似度计算公式中最为有效[17]. 为简化本文，我们仅比较 Muffler 和近期引用较多且最为有效的 4 种方法：Tarantula、Ochiai、χDebug 和 Naish.

图 3 给出了 Muffler 和这些 CBFL 技术的全面比较. 图 3 描述了在 123 个错误版本中，有多少比例的错误版本可在检查低于某个比例的代码后即检查到程序错误. 纵坐标是错误版本的比例，横坐标是按错误疑似度从高至低检查语句的比例.

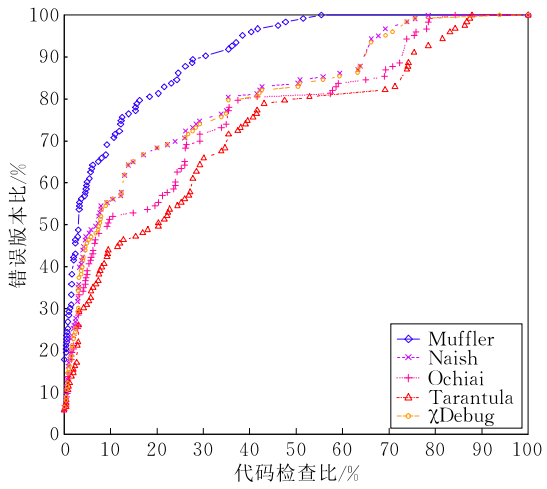


图 3　Muffler 与其它 CBFL 技术的比较

从图 3 可看到，检查 1% 的语句，Naish 技术可找到 17.07% 程序版本的错误，而 Muffler 技术可找到 28.46% 程序版本的错误；当检查的语句数增至 5% 时，Naish 技术可找到 47.15% 程序版本的错误，而 Muffler 技术可找到 60.16% 程序版本的错误. 总览图 3 中全部[0%，100%]的代码检查区域，可看到，Muffler 总比 Naish 能定位到更多错误，并且在超过 60% 语句被检查前，定位到所有错误.

表 2 给出了在某个代码检查比例下，各技术能定位到的错误数. 例如对于 Muffler 系统，检查不多于 1% 的代码，可找到 35 个错误版本的错误.

表 2　各技术在给定代码检查比例时可定位的错误数

代码检查比/%	Tarantula	Ochiai	χDebug	Naish	Muffler
1	14	18	19	21	35
5	38	48	56	58	74
10	54	63	68	68	85
15	57	65	80	80	94
20	60	67	84	84	99
30	79	88	91	92	110
40	92	98	98	99	117
50	98	99	101	102	121
60	99	103	105	106	123
70	101	107	117	119	123
80	114	122	122	123	123
90	123	123	122	123	123
100	123	123	123	123	123

从表 2 和图 3 可看出，Muffler 方法结合了覆盖信息和变异影响两种信息的优势，在每个代码检查量下，都比现有的错误定位技术能更准确和有效地定位错误.

表 3 进一步概括了各技术有效性统计数据. 以 Muffler 为例，在找到错误前，程序员最少需要检查 0% 的程序语句，最多需要检查 55.38% 的程序语句；检查语句比例的中位数和平均值分别为 3.25% 和 9.62%，标准差为 13.22%.

表 3　各技术有效性统计数据

	Tarantula	Ochiai	χDebug	Naish	Muffler
Min	0.00	0.00	0.00	0.00	0.00
Max	87.89	84.25	93.85	78.46	55.38
Median	20.33	9.52	7.69	7.32	3.25
Mean	27.68	23.62	20.04	19.34	9.62
Stdev	28.29	26.36	24.61	23.86	13.22

从表 3 可看到，在 5 种技术中，Muffler 总能在语句检查比例的最小值(Min)、最大值(Max)、中位数(Median)和平均数(Mean)上有最好的结果，且 Muffler 有更低的标准差(Stdev)，这意味着此技术的表现将会更加稳定. 总的来说，Muffler 技术相比现有最好的 Naish 技术平均减少 50.26%(=100%−(9.62%/19.34%))的代码检查量. 注意到 Muffler

技术和 Naish 技术的唯一区别在于 Muffler 技术额外考虑了变异影响这一因素，因此实验结果明确地支持了我们的结论，即使用变异影响修正 CBFL 技术的结果可以提高错误定位的准确率.

4.3 Muffler 系统的时间效率分析

上述实验说明通过结合程序覆盖信息和变异影响，Muffler 技术能比现有 CBFL 技术更有效地定位错误. 但同时 Muffler 也带来额外的时间开销，因为它需要在测试用例集上运行大量变异程序以收集语句的变异影响. 若程序包含 w 个可执行语句，变异每个语句产生 m 个变异程序，对应的测试集包含 n 个测试用例，则总共可能需要运行 $n+w\times m\times n$ 次程序. 我们通过控制被变异语句的数目(减少 w)和变异程序抽样(减少 m)来降低 Muffler 方法的时间开销. 在实际应用中，用户可以调节这两个参数在准确率和时间开销之间做出权衡.

表 4 给出了 Muffler 技术和 CBFL 技术在每个目标程序上的平均时间开销. 其中统计的时间主要包含了嵌入监控代码、执行测试用例和收集覆盖信息的时间开销. 可看到，相对于 CBFL 技术，Muffler 平均需要花费近 62.59 倍的时间.

表 4 CBFL 和 Muffler 在每个目标程序上的时间开销

目标程序	CBFL/s	Muffler/s
tcas	18.00	868.68
tot_info	11.92	573.12
schedule	34.02	2703.01
schedule2	27.76	1773.14
print_tokens	59.11	2530.17
print_tokens2	62.07	5062.87
replace	69.13	4139.19
Average	40.29	2521.46

表 5 给出了 Muffler 在时间开销方面更详细的信息，包括被变异的可执行语句数、总共的可执行语句数、生成的变异程序数和在测试用例集上执行每个变异程序的时间开销. 例如程序 tcas 每个错误版本，平均有 40.15 个可执行语句被 Muffler 变异，平均每个错误版本有 65.10 可执行语句，平均生成了 199.90 个变异程序，在测试用例集上运行每个变异程序需要 4.26 s.

表 5 Muffler 生成变异程序的情况

目标程序	变异语句数	语句总数	变异程序数	运行时间/s
tcas	40.15	65.10	199.90	4.26
tot_info	39.57	122.96	191.87	2.92
schedule	80.60	150.20	351.60	7.59
schedule2	75.33	127.56	327.78	5.32
print_tokens	67.43	189.86	260.29	9.49
print_tokens2	86.67	199.44	398.67	12.54
replace	71.14	242.86	305.93	13.30
Average	56.52	142.79	256.90	7.92

可以看到，即使在采用了变异语句数量控制和变异程序抽样这两个策略后，Muffler 系统仍然需要比较多的时间开销. 作为提高错误定位效率的代价，这些时间开销主要花费在执行变异程序上. 作为一个未来的研究方向，我们会尝试引入一些更进一步的策略提高系统的时间效率，例如忽略变异前未覆盖待变异语句的测试用例以及并行化执行变异程序等.

4.4 Muffler 系统有效性在真实错误上的验证

我们进一步利用真实世界中的 space 程序验证 Muffler 技术的有效性. 表 2 也给出了 space 程序的基本信息. space 程序提供了 1000 个测试用例集，每个测试用例集包含约 150 个测试用例. 我们随机选择了一个包含 153 个测试用例的测试用例集，并选择了 10 个错误程序版本，代表不同的偶然性成功测试用例比例，以模拟真实的程序调试环境.

表 6 给出了不同偶然性成功测试用例比例的错误版本的结果. 此表的“*CC*%”列表示偶然性成功的测试用例占总的成功测试用例的比例. “代码检查行数”表示在遇到错误前需要检查的代码行数，分别列出了 Naish 和 Muffler 两种方法的结果. 例如，版本“v6”的成功测试用例中有 6.92% 的偶然性成功测试用例，在遇到错误语句前，Naish 技术需要检查 40 条语句，而 Muffler 技术仅需检查 7 条语句，代码检查代价减少量为 82.5% = 100% − (7/40).

表 6 Muffler 在 space 程序上的结果

错误版本	*CC*/%	代码检查行数	
		Naish	Muffler
v5	0.90	2	1
v20	1.97	15	5
v21	1.97	15	6
v10	2.74	47	18
v11	6.29	37	14
v6	6.92	40	7
v9	19.05	7	1
v17	30.92	427	244
v28	48.57	268	170
v29	99.32	797	331

从真实错误上的结果可看到，相比 Naish 技术，Muffler 总是检查较少的代码就可以查到程序的错误. 同时也可看到即使 Naish 技术已经取得了不错的效果时，即检查语句数低于 50 时，Muffler 可以把检查语句数目进一步降低到更符合实际调试的级别，即检查少于 10 行或者 20 行语句. 且当偶然性成功比例较高时，检查代码的减少量更加显著，这说明程序变异分析对于降低偶然性成功测试用例的负面

影响确实有显著作用.

5　结论与展望

基于代码覆盖信息的错误定位技术(CBFL)分析成功和失败测试用例的覆盖信息以辅助程序员定位错误代码.在实际环境中常见的偶然性成功测试用例会显著降低 CBFL 技术的准确率.本文提出了一种结合程序变异分析和覆盖信息的错误定位方法来解决这个问题.此方法避开了之前方法的各种局限性,如需要程序员了解程序中的错误类型等,因此具有更广泛的应用.

我们实现了一个原型错误定位系统 Muffler,在 7 个程序的 123 个错误版本上将 Muffler 技术和现有的 4 个代表性 CBFL 技术进行了比较.结果显示,Muffler 技术显著减少了定位到程序错误的平均代码检查量.实际环境中 space 程序的真实错误也进一步验证了 Muffler 的有效性.

在未来工作中,我们计划将本文方法推广到多个程序错误的定位,并考察不同变异算子对于错误定位是否有不同影响,研究如何减少变异语句与变异程序数以进一步提高 Muffler 的效率.

参　考　文　献

[1] Chen M Y, Kiciman E, Fratkin E, et al. Pinpoint: Problem determination in large, dynamic Internet services//Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02). Bethesda, USA, 2002: 595-604

[2] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization//Proceedings of the 24th International Conference on Software Engineering (ICSE'02). Orlando, USA, 2002: 467-477

[3] Abreu R, Zoeteweij P, van Gemund A J C. On the accuracy of spectrum-based fault localization//Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. Cumberland Lodge, UK, 2007: 89-98

[4] Wong W E, Qi Yu, Zhao Lei, Cai Kai-Yuan. Effective fault localization using code coverage//Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07). Beijing, China, 2007, 1: 449-456

[5] Masri W, Abou-Assi R, El-Ghali M, Al-Fatairi N. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization//Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009) (DEFECTS'09). Chicago, USA, 2009: 1-5

[6] Acree A T. On mutation [Ph. D. dissertation]. Georgia Institute of Technology, Atlanta, USA, 1980: 184

[7] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique//Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05). Long Beach, USA, 2005: 273-282

[8] Masri W. Fault localization based on information flow coverage. Software Testing, Verification and Reliability, 2010, 20(2): 121-147

[9] Santelices R, Jones J A, Yu Y, Harrold M J. Lightweight fault-localization using multiple coverage types//Proceedings of the 31st International Conference on Software Engineering (ICSE'09). Vancouver, Canada, 2009: 56-66

[10] Baudry B, Fleurey F, Traon Y L. Improving test suites for efficient fault localization//Proceedings of the 28th International Conference on Software Engineering (ICSE'06). Shanghai, China, 2006: 82-91

[11] 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'03). San Diego, USA, 2003: 141-154

[12] Jones J A. Fault localization using visualization of test information//Proceedings of the 26th International Conference on Software Engineering (ICSE'04). Edinburgh, UK, 2004: 54-56

[13] Richardson D J, Thompson M C. An analysis of test data selection criteria using the RELAY model of fault detection. IEEE Transactions on Software Engineering, 1993, 19(6): 533-553

[14] Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria//Proceedings of the 16th International Conference on Software Engineering (ICSE'94). Sorrento, Italy, 1994: 191-200

[15] Wang X, Cheung S C, Chan W K, Zhang Z. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization//Proceedings of the 31st International Conference on Software Engineering (ICSE'09). Vancouver, Canada, 2009: 45-55

[16] Masri W, Assi R A. Cleansing test suites from coincidental correctness to enhance fault-localization//Proceedings of the 2010 3rd International Conference on Software Testing, Verification and Validation (ICST'10). Paris, France, 2010: 165-174

[17] Naish L, Lee H J, Ramamohanarao L. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering Methodology, 2011, 20(3): 11

[18] Hamlet R G. Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering, 1977, 3(4): 279-290

[19] DeMillo R A, Lipton R J, Sayward F G. Hints on test data

selection: Help for the practicing programmer. Computer, 1978, 11(4): 34-41

[20] Richard H A, Demillo R A, Hathaway B, et al. Design of mutant operators for the C programming language. Software Engineering Research Center, Purdue University: Technical Report SERC-TR-41-P, 1989

[21] Papadakis M, Traon Y L. Using mutants to locate "Unknown" faults//Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation (ICST'2012). Montreal, Canada, 2012: 691-700

[22] Zhang Z, Chan W K, Tse T H, et al. Non-parametric statistical fault localization. Journal of Systems and Software, 2011, 84(6): 885-905

[23] Mathur A P, Wong W E. An empirical comparison of data flow and mutation-based test adequacy criteria. Software Testing, Verification and Reliability, 1994, 4(1): 9-31

[24] Budd T A. Mutation analysis of program test data [Ph. D. dissertation]. Yale University, New Haven, CT, USA, 1980: 155

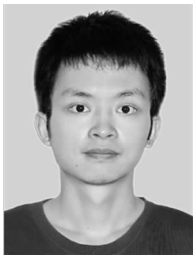
[25] Offutt A J, Lee A, Rothermel G, et al. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology, 1996, 5(2): 99-118

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

[27] Wong W E, 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

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

[29] Delamaro M E, Maldonado J C. Proteum—A tool for the assessment of test adequacy for C programs//Proceedings of the Conference on Performability in Computing Systems (PCS'96). Brunswick, NJ, 1996: 79-95



HE Tao, born in 1987, M. S. candidate. His research interest mainly focuses on software fault localization.

WANG Xin-Ming, born in 1980, Ph. D. His research interest mainly focuses on software testing.

ZHOU Xiao-Cong, born in 1971, Ph. D., associate professor. His research interest mainly focuses on software testing.

LI Wen-Jun, born in 1966, Ph. D., professor. His research field is software engineering.

ZHANG Zhen-Yu, born in 1979, Ph. D., associate professor. His research interest mainly focuses on software testing.

CHEUNG Shing-Chi, born in 1962, Ph. D., professor. His research fields is software engineering.

Background

In software engineering, debugging is an expensive and tedious activity which consists of two steps, fault localization and fault correction. The first step, fault localization, has been recognized as the most effort-consuming step in debugging. Recent advances in fault localization leverage coverage information about the execution of a test suite to help locate faults. These techniques are generally referred to as coverage-based fault localization (CBFL).

Although initial successes of CBFL techniques have been reported, the effectiveness of these techniques can be influenced by many factors. One of such factors pertinent to passed test runs is the presence of coincidental correctness, which refers to the phenomenon that "no failure is detected, even though a fault has been executed". This is because the occurrence of coincidental correctness weakens the correlation between the coverage of faulty program entity and program failures.

In this paper, we propose an idea of using mutation analysis to alleviate the impact of coincidental correctness and improve the accuracy of CBFL. Our key insight comes from an observation that mutating the faulty statement tends to maintain the results of a passed test case. By contrast, mutating a correct statement tends to toggle the results of passed test cases from passed to failed, as this new mutation fault intuitively gives the program an additional chance to fail. Motivated by this observation, we hypothesize that the amount of test cases whose test results are changed from passed to fail after mutating a program entity provides another indicator to the suspiciousness of this entity in additional to its coverage, and using this indicator to refine the ranking result of CBFL can reduce the adverse impact of coincidental correctness. Experiments have shown that our approach is effective in locating faults in the Siemens suite. We have also cross-validated our result using real-world program space with real faults.

This research is supported by the National Natural Science Foundation of China under Grant No. 61003027, the Fundamental Research Funds for the Central Universities of China under Grant No. 10LGZD05 and No. 11LGPY39, the Hong Kong Research Grant Council under Grant No. 612210, and the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No. 2012ZX01039-004)