自动化软件错误定位技术研究进展

虞 凯[1),2)] 林梦香[1),3)]

[1)](北京航空航天大学软件开发环境国家重点实验室 北京 100191)

[2)](北京航空航天大学计算机学院 北京 100191)

[3)](北京航空航天大学机械工程及自动化学院 北京 100191)

摘 要 调试过程中代价最昂贵和最耗时的活动之一就是定位错误.为了辅助开发人员进行程序错误的定位和修正,自动化错误定位技术通过对源程序、测试结果以及各种程序行为特征信息的计算分析,给出造成故障的软件缺陷在源代码中的可能位置.文中对现有错误定位技术进行了分类,介绍了各种代表性技术的原理以及建模方法,并给出了常用的评测基准集和评价标准,最后还指出了若干值得进一步研究的方向.

关键词 错误定位;自动化调试;程序分析;自适应测试

中图法分类号 TP306 **DOI 号**: 10.3724/SP.J.1016.2011.01411

Advances in Automatic Fault Localization Techniques

YU Kai[1),2)] LIN Meng-Xiang[1),3)]

[1)](*State Key Laboratory of Software Development Environment, Beihang University, Beijing* 100191)

[2)](*School of Computer Science and Engineering, Beihang University, Beijing* 100191)

[3)](*School of Mechanical Engineering and Automation, Beihang University, Beijing* 100191)

Abstract One of the most expensive and time-consuming activities of the debugging process is locating the faults. To guide the developers to locate and correct program errors, automatic fault localization techniques identify possible locations of faults by analyzing the source code, test outcomes and various program spectra. This work classifies current techniques and introduces principles and models of representative ones. Some widely-used benchmarks and evaluation metrics are provided. Finally, some on-going research issues are discussed.

Keywords fault localization; automated debugging; program analysis; adaptive testing

1 引 言

为了保证软件开发的质量,工业界在软件测试阶段投入了大量的人力物力.据统计,软件测试约占软件开发和维护成本的 50%~75%[1],其中最耗时、代价最昂贵的任务之一就是调试过程[2],这是指对程序错误进行定位和修正的过程.而错误定位又是软件调试中最耗时和困难的一步:调试过程需要理解程序的功能、实现、结构、语义以及相关的失败执行的特点.通常调试任务只能由程序的开发人员来完成,其它人员难以胜任,错误定位过程中的任何改进都可以大大降低调试成本.

用于检测程序缺陷的技术很多,本文专门讨论基于实际执行的动态定位技术.这类技术通过对源程序、测试结果以及各种程序行为特征信息的计算分析,给出造成故障的软件缺陷在源代码中的可能位置,辅助开发人员进行程序错误的定位和修正.本

收稿日期:2010-10-26;最终修改稿收到日期:2011-04-27.本课题得到软件开发环境国家重点实验室自主课题(SKLSDE-2011ZX-07)资助.**虞 凯**,男,1985 年生,博士研究生,主要研究方向为程序分析和错误定位.**林梦香**(通信作者),女,1968 年生,博士,讲师,主要研究方向为模型检查技术和自动测试数据生成. E-mail: mxlin@nlsde.buaa.edu.cn.

文调研了自动化错误定位技术的研究进展:第 2 节概述自动化错误定位技术的研究;第 3 节根据错误定位技术的类别分类陈述最新的学术进展;第 4 节讨论错误定位技术中进行有效性度量的标准;第 5 节为总结并展望未来的研究方向.

2 自动化软件错误定位技术研究概述

2.1 问题陈述

令 $T=\{\tau_1,\tau_2,\cdots,\tau_n\}$ 表示一个出错程序 P 的测试组件,其中 $\tau_k=(i_k,o_k)(1\leq k\leq n)$ 是以 i_k 为输入以 o_k 为期望输出(即测试谕言(test oracle)为 o_k)的测试用例. P 执行 τ_k 的实际输出结果为 $\hat{o}_k=P(i_k)$. 称 τ_k 是 P 的一个通过或成功(passed/successful)测试用例当且仅当 $\hat{o}_k=o_k$,否则称 τ_k 是 P 的一个失败(failed)测试用例. 根据测试用例的执行结果,可以将测试用例集 T 划分成不相交的两个集合 T_p 和 T_f, T_p 是成功测试用例的集合,T_f 是失败测试用例的集合. P 执行 τ_k 的运行时信息(比如覆盖信息或调用图等)用 $\Gamma(\tau_k)$ 表示,则 P 执行 T 时的全部执行信息表示为 $\Gamma=\Gamma_p\cup\Gamma_f$,其中 $\Gamma_p=\{\Gamma(\tau_k)\mid\tau_k\in T_p\}$, $\Gamma_f=\{\Gamma(\tau_k)\mid\tau_k\in T_f\}$.

自动化错误定位技术的任务是在给定 (P,T,Γ) 的条件下(其中 Γ 可以取空),定位 P 中值得怀疑的出错程序实体的集合 S.

2.2 研究历史

Weiser 最早提出程序切片[3]可以用于程序理解和软件调试. 之后,研究人员发现动态切片[4-7]更适用于错误定位与理解. 此外,早期的研究者试图创造两个相似的程序输入,一个可以导致程序成功执行而另一个会导致执行失败. 他们假设相似的输入会导致相似的执行过程,进而开发人员可以通过比较执行的不同来定位错误. 由于通信费用的日益降低和计算能力的飞速发展,使得搜集程序执行信息和操纵程序执行状态成为可能. 自 2002 年以来,自动化错误定位技术逐渐成为软件工程界的研究热点,表 1 按照时间顺序,列出了部分典型的错误定位技术以及相关的研究机构和论文发表的会议或期刊,简要地描述了错误定位技术研究的历史.

表 1 自动化错误定位技术研究历史

时间	方法	研究机构	发表会议/期刊
1999	Delta Debugging	Saarland University	ESEC/FSE
2002	Tarantula	Georgia Institute of Technology	ICSE
2003	Nearest Neighbor Queries	Brown University	ASE
2005	Cooperative Bug Isolation	University of Wisconsin-Madison	PLDI
2006	SOBER	UIUC	TSE
2006	Predicate Switching	University of Arizona	ICSE
2006	SAFL	Peking University	JASE
2007	Implicit Dependence	Purdue University	PLDI
2008	Value Replacement	University of California at Riverside	ISSTA
2008	TWT	IBM T. J. Watson Research Center	ICSE
2009	CP	University of Hong Kong	ESEC/FSE
2009	HOLMES	Microsoft Research	ICSE
2010	DES	University of Hong Kong	JSS
2010	PPDG	Georgia Institute of Technology	TSE
2011	Optimal Ranking Metric	University of Melbourne	TOSEM

表 1 中列出的这些技术,大多通过(半)自动化开发人员日常调试时采用的策略,最终给出错误可能存在的位置. 有的是通过类似二分查找的方式,缩小引发错误的条件[8-9];有的寻找和给定的失败执行最相似的成功执行,然后比较找出不同[10];有的认为经常出现在失败执行中的程序实体更值得怀疑[11];有的观察程序谓词在成功执行和失败执行中的取值模式的异常[12];还有的修改程序运行时状态来寻找对测试执行结果有影响的谓词或语句等等[13-14]. 按照使用和操纵的程序执行信息的不同,目前的错误定位方法可以划分为 3 类:基于行为特征对比的方法、基于程序状态修改的方法和基于程序依赖关系的方法.

3 自动化错误定位技术分类

3.1 基于行为特征对比的方法

程序行为特征,也被称为程序光谱,是程序执行特征的统计信息. Reps 等人[15]在调试解决千年虫的问题时首次提出程序光谱的概念. 为了揭示程序

行为特征与程序错误之间的关系，Harrold 等人[16]对此做了大量的实验.实验结果表明，程序出现异常的行为特征未必意味着代码中存在缺陷，但是错误的程序运行往往会表现出异常的行为特征.

基于行为特征对比的方法，通常假设失败的测试执行会表现出异常的程序行为特征，所以成功执行和失败执行中的行为特征的差异可以用于指导错误定位，其工作流程如下：首先，根据收集信息类型的需要，对源代码进行插桩并执行程序，收集执行信息.然后，判断每个测试用例的执行结果.接着，解析执行信息，得到执行行为特征.然后，根据给定的模型，建模程序实体的怀疑度，即可能出错的程度[11].最后，以程序实体排名的方式给出定位结果，将各程序实体按照怀疑度大小从大到小排列，供开发人员查看.按照使用的行为特征信息的种类和策略，大体可以分为五类：基于语句或基本块的、基于谓词的、基于方法的、基于定义使用对或信息流的以及行为特征信息精炼的方法.

3.1.1　基于语句或基本块

2003 年，Renieres 和 Reiss 提出使用相似的程序光谱来进行错误定位[10].NNQ(Nearest Neighbor Queries)法假设存在一个失败的执行和很多成功的执行，然后根据距离准则挑选出一个程序光谱和失败运行最相似的成功运行(即失败执行的最近邻居)，进而比较它们光谱的不同之处以分离软件错误.

NNQ 法认为那些出现在成功执行中的程序实体不应该被怀疑，与此不同，Jones 和 Harrold 提出的 Tarantula 法[11,17]认为只要是主要被失败用例执行的程序实体就值得怀疑，同时它也容忍出错的程序实体偶尔地被成功用例执行.他们使用常用的信息来辅助错误定位，包括每个测试用例的执行结果、程序实体(语句、分支或函数等)被每个测试用例覆盖的信息以及程序的源代码.对于程序实体 e，它的怀疑度计算公式如下：

$$Tarantula(e)=\frac{\dfrac{failed(e)}{|T_f|}}{\dfrac{passed(e)}{|T_p|}+\dfrac{failed(e)}{|T_f|}},$$

其中，$failed(e)$ 和 $passed(e)$ 分别表示失败用例和通过用例执行程序实体 e 的个数，$|T_f|$ 和 $|T_p|$ 表示测试组件中所有失败用例和通过用例的个数.e 的怀疑度取值范围从 0 到 1，数值越大，出错的可能性越大，开发人员可以按照怀疑度从大到小的顺序审查源代码.

不同于 NNQ 和 Tarantula 这些基于直观或启发式的计算方法，Wong 等人[18]提出了一个定义良好的统计方法 Crosstab.它利用覆盖信息和测试结果为每条可执行语句 ω 构建一个交叉表，进而计算卡方统计量(Chi-square statistic)$\chi^2(\omega)$ 和列联相关系数 $M(\omega)$.语句 ω 的怀疑度定义如下：

$$Crosstab(\omega)=\begin{cases}M(\omega), & \varphi(\omega)>1\\ 0, & \varphi(\omega)=1\\ -M(\omega), & \varphi(\omega)<1\end{cases},$$

其中 $\varphi(\omega)=(failed(\omega)/|T_f|)/(passed(\omega)/|T_p|)$.

与上述方法不同，Hao 等人[19]提出应该考虑测试用例的相似性，并消除相似的测试用例对于定位结果的影响.他们提出一种名为 SAFL(Similarity-Aware Fault Localization)的方法.覆盖信息和测试结果使用执行矩阵 $\boldsymbol{E}=(e_{ij})$ 来表示.其中，e_{ij} 表示第 j 条语句被第 i 个测试用例覆盖的信息，$e_{i(m+1)}$ 表示第 i 个测试用例的状态信息.SAFL 方法认为每个语句块对于测试用例状态的贡献是相同的.根据执行矩阵，可得到量化矩阵 $\boldsymbol{F}=(f_{ij})$，其中，f_{ij} 表示第 j 条语句对第 i 个测试用例的状态的贡献度.根据模糊集理论，第 j 条语句的怀疑度通过其在失败用例和所有用例中的贡献度的比值确定，即

$$SAFL(j)=\frac{\sum_{k=1}^{m}\max(f_{ik}\,|\,e_{ij}>0\wedge e_{i(m+1)}=0\wedge 1\le i\le n)}{\sum_{k=1}^{m}\max(f_{ik}\,|\,e_{ij}>0\wedge 1\le i\le n)}.$$

本质上，SAFL 方法计算的是每个语句块属于失败用例集合的隶属度和属于所有用例集合的隶属度的比值，比值越大，认为其出错的可能性就越大.

Naish 等人[20]总结了 30 多种基于语句的定位方法.他们首先构造了一段名为 ITE2(If-Then-Else-2)的程序，然后在这个程序中讨论各种情形.这个程序虽然简单，但可以用于刻画错误定位中的两种重要场景：存在“噪声”(存在与错误行为强相关的正常行为)和信号“微弱”(错误行为很难被监测到).基于在 ITE2 代码模型上的分析，他们提出了两种 Optimal metric，语句 s 的怀疑度定义如下：

$$O(s)=\begin{cases}-1, & |T_f|>failed(s)\\ |T_p|-passed(s), & \text{其它}\end{cases},$$

$$O^p(s)=failed(s)-\frac{passed(s)}{|T_p|+1}.$$

直观上，对于只包含一个错误的程序，公式

$O(s)$认为对于出错语句，必定有$|T_f|=failed(s)$. 所以任何$|T_f|>failed(s)$的语句怀疑度都最低. 此外，由于认为执行出错语句的成功用例数会较少，未执行出错语句但通过的用例数就会较大，所以使用$|T_p|-passed(s)$来确定怀疑度. 对于公式$O^p(s)$的解释在定位多错误程序时更合理些. 如果有多个错误，则对于出错语句，$|T_f|=failed(s)$并不总是成立. 因此，$O^p(s)$首先考虑了失败用例执行语句的个数，然后才是通过用例执行语句的个数.

3.1.2 基于谓词

与 Tarantula 面向内部测试(in-house testing)不同，Liblit 和他的同事提出了 CBI(Cooperative Bug Isolation)技术[21]用于定位已部署软件(deployed software)中的错误. 他们的基本想法是搜集用户在使用软件时产生的执行信息，进而通过分析这些数据将软件缺陷分离出来. 与现有的系统通常只搜集崩溃时的报告不同，程序执行信息更好地刻画了软件实际使用时的场景. 然而搜集执行信息肯定会对用户使用的软件性能有一定的影响. 为了解决这个问题，Liblit 等人通过在源代码上的变换，使用稀疏的随机抽样，较好地控制了客户端的性能并返回执行时的摘要信息.

他们的抽样方法如下：程序中任何语句的集合都可以被认为是一个插桩点，进而被设计为可供采样的而不是无条件运行的. 即每次程序执行到插桩点时，随机决定是否要执行插桩. 为了捕获可以辅助缺陷定位的程序行为，Liblit 等人对于 C 程序采用了分支、返回值和标量对 3 种插桩方案.

为了识别和程序故障相关的谓词，对于一个谓词 P，令 $F_T(P)$和 $S_T(P)$分别表示 P 被观察到的在失败和成功执行中取值为真的次数，$F_O(P)$和 $S_O(P)$分别表示 P 被观察到的在失败和成功执行中出现的次数，则 P 的怀疑度被计算如下：

$$CBI(P)=\frac{2}{\dfrac{1}{\dfrac{F_T(P)}{F_T(P)+S_T(P)}}-\dfrac{F_O(P)}{F_O(P)+S_O(P)}}+\frac{1}{\dfrac{\log(F_O(P))}{\log(|T_f|)}}.$$

这个计算公式本质是求一个调和平均数，用以有效地平衡两方面的因素：谓词的特异性(specificity)和谓词的灵敏性(sensitivity). 和信息抽取中查准率(precision)及查全率(recall)的概念类似，特异性是指在成功执行中谓词没有错误地预测存在故障，而灵敏性是指在失败执行中谓词被观察到的比例.

尽管 CBI 技术成功地从某些广泛使用的系统中识别出了一些错误，但是它只考虑了那些在失败执行中取值为真的谓词. 而对于一个总是取值为真的谓词，CBI 技术就丧失了它的判别能力. 这也说明，模型还有进一步改进的地方. Liu 等人[22]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，量化每个谓词的错误相关性，建立了 SOBER 方法. 特别地，一个谓词 P 的取值偏差是指它在一次执行中取真值的概率. 如果 P 的取值偏差在成功和失败执行中的差异越大，则谓词 P 出错的可能性越大. 令 Y 表示谓词 P 的取值偏差在所有失败执行中的均值. SOBER 使用如下公式计算 P 的怀疑度：

$$SOBER(P)=\log\left(\frac{\sigma_p}{\sqrt{\dfrac{|T_f|}{2\pi}}\cdot e^{-\frac{Z^2}{2}}}\right).$$

其中，σ_p为谓词 P 在所有成功执行中谓词的取值偏差的方差，Z 是 Y 的标准化随机变量. 开发者可以按照谓词怀疑度的大小审查源代码，发现错误的位置. 与 Crosstab 方法使用假设检验来提供测试结果与每条语句的覆盖信息之间“依赖性/独立性”参考不同，SOBER 方法使用假设检验来量化谓词的取值偏差在成功和失败执行中的差异不同.

复合谓词在取值时会由于短路求值造成不同执行时表达式中需要计算的原子条件表达式可能不同. Zhang 等人[23]研究了短路求值和求值序列对于错误定位技术的影响. 具体地，他们提出了一种基于谓词的定位方法的改进策略：DES(Debugging through Evaluation Sequences)策略，将每个谓词排名最高的取值序列作为该谓词的排名. 实验结果表明，DES 策略可以提高基于谓词的错误定位技术的有效性而同时仅产生较小的额外的性能开销.

3.1.3 基于方法

对于面向对象的语言，Dallmeier 等人[24]认为在失败执行中调用了与通过执行中不同的方法的类更值得怀疑. 他们提出了基于方法调用序列的 Ample 技术，认为只出现在通过执行或失败执行中的方法调用子序列应该被怀疑. 与在通过和失败执行中都出现的子序列相比，这些子序列被分配较大的怀疑度值. 遗失或增添的子序列都值得怀疑，因为它们都可能引发程序故障.

给定一个类 C 以及相关的一个失败执行 c_f和一个通过执行 c_p. 对于给定长度 k，两个执行产生的子序列集分别为 $S(c_f)$和 $S(c_p)$. 对于没有同时出现在两个执行中的子序列 s，其怀疑度 $w(s)$赋值为 1，

否则赋值为 0. 则类 C 的怀疑度可以用它包含的子序列的怀疑度的平均值来计算，即

$$Ample(C)=\frac{1}{|S_C|}\sum_{s\in S_C}w(s),$$

其中 $S_C=S(c_f)\bigcup S(c_p)$.

Ample 技术也可以使用多组通过执行来得到子序列集 $S(c_p)$ 并用于计算怀疑度值. 进一步的研究表明，子序列长度 k 的取值会影响定位结果的灵敏性，k 通常取值在 5～10 之间较好.

与 Ample 技术不同，Yilmaz 等人[25]提出使用时间光谱作为程序执行特征的抽象. 时间光谱是指程序实体(比如方法、函数)运行的时间特征信息，通常用于程序性能的评价和优化. 他们提出一种叫做 TWT(Time Will Tell)的方法，首先收集成功执行和失败执行的时间光谱，接着基于成功执行的时间光谱建立程序行为模型，然后使用这个模型来识别失败执行和成功执行的偏离程度. 比如，一种方法如果在失败执行中花费了比通过执行值得怀疑的较长或较短的时间，那么这种方法可能就和程序错误相关. TWT 技术选取的是方法的执行时间作为时间光谱，因为方法提供了合适的粒度水平和功能的界限. 对每个方法，基于通过执行的时间光谱创建一个高斯混合模型. 这是个多维概率密度模型，首先识别出数据聚成的类，然后对每个类使用高斯分布建模. 对于一个给定的失败执行，以它和聚类中心点的偏离程度作为怀疑度的得分，距离越远，怀疑度越大.

3.1.4　基于定义使用对、分支或信息流

Santelices 等人[26]发现覆盖信息类型的选择可以极大地影响到错误定位技术的有效性：有些缺陷最好使用语句覆盖信息来定位，而有些最好使用分支或定义使用对的覆盖信息. 他们首先计算一个程序实体的怀疑度，然后对于分支或定义使用对，根据 3 条规则，将每条语句的可疑度定义为和它关联的分支或定义使用对可疑度的最高值. 在完成映射之后，采用 max-SBD，avg-SBD 和 avg-BD 3 种策略来获得一个语句的可疑度得分. 即一个语句的可疑度得分可以是和它关联的 3 种程序实体怀疑度的最大值、平均值或只是定义为和它关联的分支和定义使用对怀疑度的平均值. 实验表明，一般来说，max-SBD 并不比使用单一覆盖类型的定位技术有效. 相反，avg-SBD 和 avg-BD 都要比单一覆盖类型的定位技术有效，并且这个优势是统计显著的. 这说明综合使用多种覆盖类型的信息，确实可以提高错误定位技术的有效性.

与 Santelices 等人使用单一模型建模程序实体不同，Yu 和他的同事[27]进一步提出了一个使用多个模型来捕捉不同类型的错误的方法 LOUPE. 他们假设，对于任一类型的错误，存在一种适合模型，它能够较好识别出错误语句. 由于错误类型事先未知，LOUPE 方法建立了多个模型来捕捉语句的异常行为，并试图选出相应的适合模型. 具体的，分别使用 CDBug 模型和 DDBug 模型来建模控制流和数据流上错误的异常行为. 从实验结果中发现，出错语句在其适合模型下往往有较高的怀疑度. 对于每条语句，LOUPE 方法以给出较高怀疑度的模型作为适合模型，并且这条语句的怀疑度得分就是这个模型给出的怀疑度值. 对于语句 s，它的怀疑度计算公式如下：

$$Loupe(s)=\max(susp_{\mathrm{CDBug}}(s),susp_{\mathrm{DDBug}}(s)).$$

动态信息流分析是一种更加重量级的方法，它考虑 5 种类型的依赖关系：动态直接控制依赖、动态直接数据依赖和 3 种过程间的动态依赖关系. 由于动态信息流分析能够识别运行时程序对象之间的信息流，所以也可以用来建模程序元素之间复杂的交互作用. Masri 提出了一种基于动态信息流分析的错误定位方法[28]. 对于信息流 f，怀疑度计算公式为

$$Difa(f)=\left(\frac{\frac{failed(f)}{|T_f|}}{\frac{passed(f)}{|T_p|}+\frac{failed(f)}{|T_f|}}+\max\left(\frac{passed(f)}{|T_p|},\frac{failed(f)}{|T_f|}\right)\right)\Big/2.$$

最后，每条可执行语句被赋予流经它的怀疑度最大的信息流的值. 而且，值得怀疑的信息流的源语句应该先于这条信息流上的其它语句被检查.

3.1.5　行为特征信息的精炼

尽管研究人员已经尝试了不同的怀疑度计算方法，使用了不同类型的行为特征，并研发了不同的原型工具，但是由于受到很多因素的影响，在实践中基于行为特征对比的方法的有效性仍然会受到限制. Masri 等人[29]实验研究了这类方法的有效性下降的 4 个场景，分别是：错误条件满足，但程序故障并未出现(作者称为巧合正确性)；错误语句被执行，但程序故障并未出现(作者称为弱巧合正确性)；程序故障和不止一个的不同类型的程序元素的组合有关；以及很多程序元素不在通过执行中出现，却出现在

所有的失败执行中.前 3 个场景可能会影响错误定位技术的准确性,后一个会影响精确性.实验结论是弱巧合正确性对定位技术的有效性有相当大的影响,尤其是基于语句的.此外,使用多种程序执行特征的组合很可能会提高定位的有效性.为了提高定位技术的有效性,研究人员提出了不同的精炼的方法,来减少各种因素(主要是弱巧合正确性)的影响.

针对弱巧合正确性,Wang 等人[30]受后向动态切片的启发,将与程序输出结果无关的实体的覆盖信息剔除,以精炼程序执行特征.然而,后向动态切片不能捕捉诸如遗漏语句这样的错误.观察表明当错误语句被执行并且引发程序故障时,执行前后的动态控制流和数据流往往与特定模式匹配,称为上下文模式.事实上,后向切片仅使用了其中的一个模式,即失败执行输出之间存在的动态依赖关系.然而,这个模式对于其它类型的错误是无效的,比如遗漏语句的情形.作者推断使用一些常用的上下文模式可以用于精炼行为特征信息.具体地,他们使用事件表达式和扩展的有限状态机描述了 12 个上下文模式.这些模式可以用于匹配 13 类常见的错误类型.然后,程序执行信息被精炼,使用常用的怀疑度计算公式(如 Tarantula 公式)计算程序实体的可疑度,并得到错误定位报告.

之前的基于行为特征对比的错误定位技术通常只关注于评估单个的程序实体的怀疑度,而忽略了被感染的程序状态在它们之间的传播.Zhang 等人[31]提出使用控制流边的信息来表示成功执行和失败执行,然后比较它们的差异来建模每个基本块对于程序故障的贡献程度.具体来说,首先计算一条控制流边 e 的怀疑度 $\theta(e)$.然后,对于每个基本块 b_j,假设 b_k 是 b_j 的一个后继块,使用 $W(b_j,b_k)$ 来表示控制流边 (b_j,b_k) 的怀疑度 $\theta(b_j,b_k)$ 占 b_k 的所有入边怀疑度的比例,则可以得到描述基本块 b_j 和它的后继块之间关系的等式:

$$BR(b_j)=\sum_{\forall edg(b_j,b_k)}[BR(b_k)\cdot W(b_j,b_k)].$$

假设共有 n 个基本块,则可以得到 n 个含 n 元变量的方程组.这种情况下,方程组满足可解的必要条件,并可以用已有的有效算法求解(比如高斯消去法).在求出每个 $BR(b_j)$ 的怀疑度后,每条语句的怀疑度即为其所在基本块 b_j 的怀疑度.

3.2 基于程序状态修改的方法

基于程序状态修改的方法通常在程序执行时,获得并修改程序的状态,然后观察修改后的测试结果(成功/失败),进而找出对测试结果有影响的关键谓词或语句.由于程序运行时的状态有多种可能,这类方法往往需要采用一些简化策略,提高定位的效率.

Delta Debugging 方法[8-9,32]是由 Zeller 提出的一种能自动缩小程序的成功运行过程和失败运行过程之间区别的技术.在实现层面,它采用分治思想,把软件配置(测试输入、源程序等)变动的集合进行划分,然后分别进行测试,结果可以为通过、失败和无解.然后递归地把导致失败配置的集合并入结果为通过配置的集合.通过逐渐减小两个集合之间的差异,最终确认成功配置和失败配置差别的一个最小子集.

它的首次应用是处理由于 GDB 的代码变动导致的可视化调试工具 DDD 的一个故障.在 2005 年,Delta Debugging 方法被应用到程序状态上,自动化地找出导致程序失败的语句.首先,通过得到程序所有变量以及它们的值构成的集合,产生程序状态图.进而,使用计算公共子图的算法来识别成功与失败程序状态之间的差异.最后,将差异应用到程序状态,找到导致故障的变量,并跟踪这些变量的值,一直到产生错误的语句.

识别对运行时的程序状态做哪些修改可以使得失败执行变成成功执行是一种通用而且有效的自动化调试方法.然而由于程序状态的数目极其巨大,漫无目的搜索所有可能的程序状态的变化是不现实的.Zhang 等人[14]的研究表明,在运行时强制对一个谓词改变取值结果并因此改变程序的控制流,不仅代价较低,而且往往可以使一次失败执行变成成功执行.通过检查被切换的谓词,也就是关键谓词,可以识别故障的根源.由于一个谓词的取值只能有两种可能,因此需要修改的状态数要远小于所有可能的程序状态数.实验结果显示这种方法既实用又有效.进一步的研究表明关于关键谓词的双向的动态切片可以有效地捕捉出错的代码.

改变分支的取值结果是改变运行时变量的值的一种特殊情形.Jeffrey 等人[13]提出了 Value Replacement 方法:首先尝试改变运行时变量的值得到成功用例,进而将这种信息用于错误定位.具体地,首先寻找感兴趣的变量映射对 IVMP(Interesting Value Mapping Pair),然后对包含 IVMP 的语句使用 Tarantula 公式计算怀疑度值并进行排序.IVMP 由一个变量的原始值和替换值组成,并且将变量更改为替换值后可以将一次失败执行变成通过执行.

给定一次失败运行，寻找 IVMP 的方法是直接的：一次只考虑一条在失败运行中执行的语句，然后将使用的值替换成另一个，并观察是否会产生正确结果. 为了有效地获得 IVMP，仅考虑了 4 种类型的值作为替换值. 实验表明，Value Replacement 方法可以有效地定位出错语句或和出错语句存在直接依赖关系的语句.

3.3　基于程序依赖关系的方法

基于程序依赖关系的方法侧重于使用程序的动态依赖关系给出值得怀疑的语句的集合，这个集合除了包含错误语句外，还提供了一个供程序员理解的调试上下文. 但通常这类集合也会包含一些冗余的语句，需要使用一些技术来化简集合.

程序切片[3]最早是由 Weiser 提出，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，以解决错误定位中的遇到的各种问题. Zhang 等人[33]从动态角度提出隐式依赖(implicit dependence)的概念，它只会将已经观测到的发生在谓词和变量使用上的依赖关系加入切片中. Zhang 等人进一步使用一种需求驱动的策略来减小探测隐式依赖的开销. 实验表明，在仅增加一小部分的检查和很少的隐式依赖的情况下，就可以有效地捕捉遗漏语句这种类型的错误. 为了减小动态切片的规模，研究者提出了许多种化简策略. Gupta 等人[34]整合了 Delta Debugging 技术以及前、后向切片的优势用于缩小搜索出错代码的范围. 他们首先使用 Delta Debugging 技术来识别一个最小化的故障相关的输入，然后基于这个输入计算动态前向切片并与以错误输出为准则产生的动态后向切片取交集，作为引发故障的砍片(chop). 实验表明，引发故障的砍片和动态切片比，规模有了极大的减小同时并未显著降低定位错误代码的能力. Zhang 等人[35]观察到失败执行中出现的语句也有可能卷入到成功执行中. 因此他们使用值的概要分析文件(value profile)来计算每条可执行语句的信赖度值，数值越大表明语句产生正确值的概率越大. 进一步的实验表明，给定程序故障的一个失败运行，通过只剪除信赖度值为 1 的语句就可以有效地缩减动态切片的规模，同时将出错语句保留在切片内. 此外，研究发现以信赖度值递增的方式检查语句是一个有效的降低错误定位成本的策略.

Baah 等人[36]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. PPDG 可以有多种应用，包括错误定位和错误理解. 在用于错误定位时，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它的怀疑度，基于其父节点的值计算得到，条件概率值越低，怀疑度越高. 由于 PPDG 直观上显示了失败执行和成功执行的不同，并提供了相关的上下文信息，因此可以用于辅助理解为什么某个特定语句是值得怀疑的.

与 PPDG 学习的是程序中每个语句的依赖关系的分布不同，Feng 和 Gupta 为每种指令类型建立了通用模型[37]. 给定一组程序的执行轨迹，并包括至少一个成功执行和至少一个失败执行，可以基于动态依赖图建立基于贝叶斯网络的错误流图(Error Flow Graph)和通用的概率模型. 然后使用标准的推理算法从叶节点沿着错误流后向追溯寻找出错可能性最大的可执行语句. 实验表明，即使只使用很少的成功执行，错误流图依然可以有效地定位错误.

4　错误定位技术的有效性度量

4.1　错误定位技术的评测标准数据集

4.1.1　Siemens Suite

Siemens Suite 最早是为了研究控制流和数据流的准则对于错误探测能力的影响而创建的[38]. 它包含 7 组实现不同功能的 C 程序，每组程序通过人工注入的方式创建了基本程序的错误版本，这些错误通常通过修改程序中的一行代码来注入，包括语句的增删以及判断条件的修改等，以模拟实际中可能存在的错误. 最终对于每个基本程序产生了 7～41 个不同的错误版本，每个版本中包含有 1 个错误. 同时，研究人员基于对于程序功能的理解和边界分析，创建了相应的测试用例. Siemens Suite 自从 2003 年被引入用于评价 NNQ 法的有效性后，被广泛采用以评估错误定位技术的有效性.

4.1.2　其它大型的 C/C++程序

由于 Siemens Suite 包含的程序规模都较小，而且错误是人工注入的，它不能代表大型程序中出现的真实错误的特征. 因此，其它的大型的 C/C++程序也常用于错误定位技术的有效性评价中. Space

是一个为欧洲航天局开发的关于数组定义语言的解释器程序[39]. 它包含 6218 行可执行代码,相应的 13 585 个测试用例,并拥有 38 个关联版本,每一个版本都包含一个在程序开发过程中被发现的错误. University of Nebraska 的研究人员[40] 从 GNU Software 中选取了一些工具(包括 flex、grep、gzip、make、sed、bash 和 vim)并注入了一些回归错误用于错误定位的研究. 这些工具及其测试用例以及 Space 程序和 Siemens Suite 都可以从 SIR(http://sir.unl.edu/content/sir.html)网站获得.

在错误定位的研究中,研究者还使用了其它的一些 C/C++程序,包括:Ali 等人[41] 在进行错误定位技术有效性的研究时,创建了 concordance(词汇索引工具,源程序包括 2354 行)的 13 个错误版本和 372 个测试用例. Liblit 等人[21,42] 首先报告了 bc(数值处理计算程序,源程序包括 14 288 行)存在的一个缓冲区溢出错误. 随后,Liu 等人在使用 SOBER 进行错误定位的研究时,随机产生了 4000 个测试用例,并发现了两个未曾报告的程序缺陷. Liblit 等人还研究了 rhythmbox(音乐播放和管理软件,源程序包括 56 484 行). 他们随机产生了 32 000 个随机测试用例,并报告了两个错误. Jiang 和 Su[43] 在进行上下文相关的统计调试的研究中,又发现了两个未知的错误.

4.1.3 JAVA 程序

用于错误定位研究的 JAVA 程序主要包括:NanoXML(小型的 XML 解析器,源程序包含 7646 行),包含 6 个版本以及共 33 个已知的错误. XML-security(XML 数字签名工具,源程序包含 16 800 行)和 Ant(基于 Java 的构建工具,源程序包含 80 500行)以及它们开发时的 JUnit 测试组件. 这 3 个程序都可以从 SIR 网站获得. 此外,常用的基准集还有 AspectJ(JAVA 语言的面向方面的扩展,源程序约 75 000 行),它是由 Dallmeier 和 Zimmermann 开发的 iBUGS 工具[44],从 AspectJ 的项目历史中抽取出的,共有 369 个开发过程中的程序错误,其中 223 个有相关联的测试用例.

4.2 错误定位技术的评价标准

4.2.1 基于程序依赖图的评价标准[10]

Renieres 和 Reiss[22,31] 最早提出了基于程序依赖图的评价标准,也被称为 T-score. 他们假设错误定位报告的使用者会从报告中包含的语句开始,沿着程序依赖关系,对源程序进行广度优先的搜索. 理想的场景是用户看到了出错语句后就会识别出程序错误并停止审查源代码. 错误定位技术的得分即定义为用户找到出错语句时,未被用户审查到的代码占所有代码的百分比. 得分越高,表明错误定位技术越有效,对于用户精确定位错误越有帮助.

Cleve 和 Zeller[8] 指出,T-score 假设程序员能够在任意位置区分错误和正常语句,并且在每次定位中代价都是一样的. 然而,现实中,有些错误比较容易发现,而有些则并非如此,这种差别 T-score 并没有考虑. 同样,程序员也并不总是能很容易地辨别错误和正常语句.

4.2.2 基于语句排名的评价标准[11]

Jones 和 Harrold 在比较 Tarantula 技术和其它错误定位技术的有效性时提出了基于语句排名的评价标准. 由于 Tarantula 产生一个对所有可执行语句怀疑度的排名,开发者被假设从排序列表顶部按怀疑度从大到小逐句地检查直到错误被发现,未被检查到的语句占所有语句的百分比定义为错误定位技术的得分.

显然,基于语句排序评价标准与基于程序依赖图的评价标准假设了不同的代码检查策略. 直觉上,基于程序依赖图的方法可能更接近实际. 但由于大多错误定位技术通常会产生一个可执行语句的怀疑度列表,因此,基于语句排名的评价标准被广泛使用.

4.2.3 基于基本块排名的评价标准[45]

Abreu 等人提出了一种基于基本块排名的评价标准:考虑在找到出错的基本块前,需要检查多少基本块,这和上述基于语句排名的思想类似. 但在具体细节上,他们定义需要检查的基本块的数目,就是所有比出错基本块怀疑度大的和不低于出错基本块怀疑度的基本块数目的和的一半,然后使用未被检查的基本块的百分比来定义错误定位技术的得分.

4.2.4 基于谓词排名的评价标准[46]

Zhang 等人在衡量非参数检验的错误定位技术的有效性时,提出了基于谓词排名的评价标准,也称为 P-score. 对于以谓词可疑度降序排列形式给出的错误报告,首先标记程序中和错误最接近的谓词为出错最相关谓词. 然后定义错误定位技术的得分为检查到出错最相关谓词时,已检查的谓词占列表上所有谓词的百分比. 这个标准适用于产生谓词怀疑度列表的错误定位技术.

4.2.5 基于查全率的评价标准[30]

Wang 等人在研究通过精炼覆盖信息提高基于行为特征对比的错误定位技术的有效性时,提出使

用信息检索中的查全率的概念来衡量可疑语句的集合包含真正出错语句的概率.具体如下：假设程序员最多会检查错误定位报告中前 $k\%$ 的语句，简称为 k-策略.实验研究中的错误版本数为 N，使用 k-策略检查会发现有 N_k 个版本包含错误语句.则错误定位技术的得分即定义为 N_k 与 N 的比例.

5　未来研究趋势

5.1　测试组件的影响

测试组件(test suite)的组成会对错误定位技术的有效性产生影响.当前的错误定位技术通常假设测试用例集满足测试充分性准则，足以满足错误定位的需要.然而，这个假设既没有被实验验证过又缺乏直观的考虑.事实上，减少测试工作量意味着产生一个满足准则的最小测试用例集.然而，错误定位技术又需要尽可能多的测试用例的执行信息.为了解决这个两难问题，Baudry 等人[47]通过分析用于有效错误定位时所需的信息类型，识别出动态基本块对于定位算法准确性的影响，并基于动态基本块，提出了用于诊断的测试(Test-For-Diagnosis，TFD)准则.因此，减少测试用例数和提高错误定位有效性的两难问题，可以通过挑选专门用于定位的测试用例子集得到部分的解决.Yu 等人[48]实验研究了测试组件的缩减对定位有效性的影响.结果表明错误定位技术的有效性取决于所使用的缩减策略.此外，Jiang 等人[49]还研究了测试组件的优先级技术(test case prioritization)对基于行为特征对比的定位技术有效性的影响.

5.2　多个缺陷的定位

错误定位技术通常假设出错程序中只包含一个缺陷，而实际情况并非如此.多个缺陷的存在可能从两方面影响错误定位技术的有效性：一是当程序出错时，缺陷的数目通常是未知的；二是某些缺陷可能会掩盖或混淆其它缺陷.近年来，研究者对如何定位包含多个缺陷的错误程序进行了研究.Jones 等人[50]提出了并行调试的模型，通过自动化的产生面向单个错误的专门的测试组件，为多个开发人员同时调试包含多重缺陷的程序提供了一种途径.与 Jones 等人仅使用程序特征行为的信息不同，Abreu 等人[51]提出了一种包含逻辑推理的混合框架.他们既使用了程序执行的特征信息，又使用贝叶斯推理来推断存在多个缺陷时值得怀疑的程序实体以及它们的可疑度的大小.使用贝叶斯推理的一个特性是它可以很好地解释多个缺陷为何间歇的出现而导致程序出错.

5.3　定位结果的理解

大多数现有的错误定位技术侧重于识别和报告单个可能包含缺陷的语句.然而，即使在程序故障只是由单个语句引起的情况下，一般也很难仅通过查看这条语句来理解错误的产生.故障通常会在一个特定的上下文显露出来，而知道这些上下文对于诊断和修正错误是必要的.Hsu 等人[52]提出了一种识别导致程序故障的语句序列的定位技术：分析失败执行对应的部分执行轨迹并增量地识别其中的公共序列.这可以为开发者提供一个更直接的理解错误的上下文，从而降低调试成本.与他们不同，Cheng 等人[53]通过判别图挖掘来进行错误定位和相关上下文的识别.他们在方法和基本块两个粒度建立程序执行模型，其中节点表示方法或基本块，边表示相应的调用、转移或返回.给定一组成功执行和失败执行，通过比较不同执行中的程序流，从中提取出最有区别力的子图.提取的子图不仅定位了错误，还提供了包含丰富信息的用于理解和修复错误的上下文.Jiang 和 Su[43]构建了程序错误相关的控制流路径用来理解定位结果.他们的方法包括特征选择(准确的选择和故障有关的谓词)、聚类(将相关的谓词分组)和控制流图的遍历 3 个阶段.

5.4　错误修复的建议

能够自动化定位和修正错误的工具将显著地降低软件开发成本.He 和 Gupta[54]使用基于路径的最弱前置条件来自动地产生如何修改一个函数中出错语句的建议.这种方法假设了错误函数的正确规约是可得的，并以前置条件和后置条件的形式书写.与此不同，Jeffrey 等人[55]开发的 BugFix 工具只需要一个出错程序和至少一个失败的测试用例.它自动分析某条语句上的调试情况，并提供相关修复建议的优先级列表，用以指导程序员选取适当的修复方式.BugFix 使用机器学习的方法自动的从过去的修复和新遇到的调试情形中学习.在修复中，BugFix 同时使用了程序的静态结构、语句在成功和失败执行中的使用的动态值以及和语句关联的感兴趣的变量映射对信息.与之不同，Weimer 等人[56]使用自适应的搜索方法系统地使用增加、交换和删除语句的操作，对一个出错的 C 程序进行变异，直到它既能满足需要的功能，又能不再产生之前的错误.对于面向对象的程序，Dallmeier 等人[57]开发了 PACHIKA 工具.PACHIKA 工具首先在失败用例

集和通过用例集上挖掘对象行为模型,然后通过比较模型的差异,产生修复候选集.最后,通过执行整个回归测试用例集并比较运行结果,验证一个修复是否为有效的修复,并将有效修复提供给开发者.

6 结束语

本文从技术基础、评估手段以及研究趋势三方面对现有的自动化错误定位技术进行了总结.自动化错误定位技术在过去十年间取得了长足的进步,但仍有很多问题需要研究,例如缺乏用户研究来理解开发者是如何调试的[58];现有的评测标准数据集不能完全反映现代语言中缺陷的存在和表现情况;错误定位技术不能很好地与开发环境相结合供开发者使用.同时,测试组件的影响、多重缺陷的定位、定位结果的理解以及错误修复的建议将是进行扩展的热点方向.对这些问题的深入研究,将帮助开发者更快地开发高质量的软件.

参 考 文 献

[1] Hailpern B, Santhanam P. Software debugging, testing, and verification. IBM Systems Journal, 2002, 41(1): 4-12

[2] Jones J A. Semi-automatic fault localization[Ph. D. dissertation]. Georgia Institute of Technology, 2008

[3] Weiser M. Program slicing//Proceedings of the 5th International Conference on Software Engineering (ICSE'81). Piscataway, NJ, USA, 1981: 439-449

[4] Korel B, Laski J. Dynamic program slicing. Information Process Letter, 1988, 29(3): 155-163

[5] Agrawal H, Horgan J R. Dynamic program slicing//Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI'90). White Plains, New York, USA, 1990: 246-256

[6] Demillo R A, Pan H, Spafford E H. Critical slicing for software fault localization//Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96). San Diego, CA, USA, 1996: 121-134

[7] Gyim O Thy T, Besz E Des A R A D, Forg A Cs I A N. An efficient relevant slicing method for debugging//Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7). London, UK, 1999: 303-321

[8] Cleve H, Zeller A. Locating causes of program failures//Proceedings of the 27th International Conference on Software Engineering (ICSE'05). St. Louis, MO, USA, 2005: 342-351

[9] Zeller A. Isolating cause-effect chains from computer programs//Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02/FSE-10). Charleston, South Carolina, USA, 2002: 1-10

[10] Renieres M, Reiss S P. Fault localization with nearest neighbor queries//Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering (ASE'03). Montreal, Canada, 2003: 30-39

[11] 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, CA, USA, 2005: 273-282

[12] Liu C, Yan X, Fei L, Han J, Midkiff S P. Sober: Statistical model-based bug localization//Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13). Lisbon, Portugal, 2005: 286-295

[13] Jeffrey D, Gupta N, Gupta R. Fault localization using value replacement//Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08). Seattle, WA, USA, 2008: 167-178

[14] Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching//Proceedings of the 28th International Conference on Software Engineering (ICSE'06). Shanghai, China, 2006: 272-281

[15] Reps T, Ball T, Das M, Larus J. The use of program profiling for software maintenance with applications to the year 2000 problem//Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC'97/FSE-5). Zurich, Switzerland, 1997: 432-449

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

[17] 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, Florida, 2002: 467-477

[18] Wong E, Wei T, Qi Y, Zhao L. A Crosstab-based statistical method for effective fault localization//Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST'08). Lillehammer, Norway, 2008: 42-51

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

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

[21] Liblit B, Naik M, Zheng A X, Aiken A, Jordan M I. Scalable statistical bug isolation//Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05). 2005: 15-26

[22] Liu C, Fei L, Yan X, Han J, Midkiff S P. Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering, 2006, 32(10): 831-848

[23] Zhang Z, Jiang B, Chan W K, Tse T H, Wang X. Fault localization through evaluation sequences. Journal of Systems and Software, 2010, 83: 174-187

[24] Dallmeier V, Lindig C, Zeller A. Lightweight defect localization for java//Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05). Glasgow, UK, 2005: 528-550

[25] Yilmaz C, Paradkar A, Williams C. Time will tell: Fault localization using time spectra//Proceedings of the 30th International Conference on Software Engineering (ICSE'08). Leipzig, Germany, 2008: 81-90

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

[27] Yu K, Lin M, Gao Q, Zhang H, Zhang X. Locating faults using multiple spectra-specific models//Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC'11). TaiChung, Taiwan, 2011: 1404-1410

[28] Masri W. Fault localization based on information flow coverage. Software Testing, Verification and Reliability, 2009, 20(12): 121-147

[29] 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 (DEFECTS'09). Chicago, Illinois, USA, 2009: 1-5

[30] 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 2009 IEEE 31st International Conference on Software Engineering (ICSE'09). Vancouver, Canada, 2009: 45-55

[31] Zhang Z, Chan W K, Tse T H, Jiang B, Wang X. Capturing propagation of infected program states//Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09). Amsterdam, The Netherlands, 2009: 43-52

[32] Zeller A. Yesterday, my program worked. Today, it does not. Why? //Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7). Toulouse, France, 1999: 253-267

[33] Zhang X, Tallam S, Gupta N, Gupta R. Towards locating execution omission errors//Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07). San Diego, California, USA, 2007: 415-424

[34] Gupta N, He H, Zhang X, Gupta R. Locating faulty code using failure-inducing chops//Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05). Long Beach, CA, USA, 2005: 263-272

[35] Zhang X, Gupta N, Gupta R. Pruning dynamic slices with confidence//Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06). Ottawa, Ontario, Canada, 2006: 169-180

[36] Baah G K, Podgurski A, Harrold M J. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Transactions on Software Engineering, 2010, 36: 528-545

[37] Feng M, Gupta R. Learning universal probabilistic models for fault localization//Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'10). Toronto, Ontario, Canada, 2010: 81-88

[38] 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). Los Alamitos, CA, USA, 1994: 191-200

[39] Vokolos F I, Frankl P G. Empirical evaluation of the textual differencing regression testing technique//Proceedings of the International Conference on Software Maintenance (ICSM'98). Bethesda, Maryland, 1998: 44-53

[40] 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

[41] Ali S, Andrews J H, Dhandapani T, Wang W. Evaluating the accuracy of fault localization techniques//Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09). Auckland, New Zealand, 2009: 76-87

[42] 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). 2003: 141-154

[43] Jiang L, Su Z. Context-aware statistical debugging: From bug predictors to faulty control flow paths//Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). Atlanta, Georgia, USA, 2007: 184-193

[44] Dallmeier V, Zimmermann T. Extraction of bug localization benchmarks from history//Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). Atlanta, Georgia, USA, 2007: 433-436

[45] 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: 1780-1792

[46] Zhang Z, Chan W K, Tse T H, Hu P, Wang X. Is non-parametric hypothesis testing model robust for statistical fault localization? Information and Software Technology, 2009, 51(11): 1573-1585

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

[48] 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'08). Leipzig, Germany, 2008: 201-210

[49] Jiang B, Zhang Z, Tse T H, Chen T Y. How well do test case prioritization techniques support statistical fault localization//Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09). Washington, DC, USA, 2009: 99-106

[50] Jones J A, Bowring J F, Harrold M J. Debugging in parallel//Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07). London, United Kingdom, 2007: 16-26

[51] Abreu R, Zoeteweij P, Gemund A J C V. Spectrum-based multiple fault localization//Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09). Auckland, New Zealand, 2009: 88-99

[52] Hsu H, Jones J A, Orso A. Rapid: identifying bug signatures to support debugging activities//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08). L'Aquila, Italy, 2008: 439-442

[53] Cheng H, Lo D, Zhou Y, Wang X, Yan X. Identifying bug signatures using discriminative graph mining//Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09). Chicago, IL, USA, 2009: 141-152

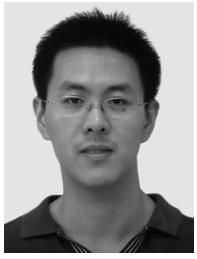
[54] He H, Gupta N. Automated debugging using path-based weakest preconditions//Proceedings of the Fundamental Approaches to Software Engineering (FASE'04). Barcelona, Spain, 2004: 267-280

[55] Jeffrey D, Feng M, Gupta N, Gupta R. Bugfix: A learning-based tool to assist developers in fixing bugs//Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC'09). Vancouver, British Columbia, Canada, 2009: 70-79

[56] Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming//Proceedings of the 31st International Conference on Software Engineering (ICSE'09). Washington, DC, USA, 2009: 364-374

[57] Dallmeier V, Zeller A, Meyer B. Generating fixes from object behavior anomalies//Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09). Washington, DC, USA, 2009: 550-554

[58] Zeller A. Debugging debugging: acm sigsoft impact paper award keynote//Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09). Amsterdam, The Netherlands, 2009: 263-264



YU Kai, born in 1985, Ph. D. candidate. His research interests include program analysis and fault localization.

LIN Meng-Xiang, born in 1968, Ph. D., lecturer. Her research interests include model checking and automated test data generation.

Background

Software today is increasingly complex. Consequently, when programs fail, debugging is also progressively becoming much more difficult and time-consuming. To address the issue, researchers have invested a great deal of effort in developing automated techniques and tools for supporting various debugging tasks. Last decade has seen impressive advances in automated debugging, especially in fault localization. This work classifies current techniques and introduces principles and models of representative ones. Some widely-used benchmarks and evaluation metrics are provided. Finally, some on-going research issues are discussed.

This work is supported by the State Key Laboratory of Software Development Environment SKLSDE-2011ZX-07. The project focuses on execution-based defect detection and fault localization. Specifically, in defect detection, an automatic testing framework including test data generation, test status determination and fault localization will be established and a prototype will be developed. In fault localization, the limitations of current techniques will be studied and the effectiveness of locating faults using multiple spectra-specific models will be evaluated. Besides, a toolset for spectrum-based fault localization will be released publicly to enable rapid prototyping for automatic debugging.