基于影响分析的回归测试优先级错误定位方法

张 慧
(东南大学计算机科学与工程学院 南京 211189)

摘 要 基于程序行为特征的错误定位方法由于只孤立地看待每个程序实体,使其错误定位的效率受到影响,而回归测试错误定位又由于需要执行全部测试用例将大大增加开发和测试成本。针对以上问题,提出一种基于影响分析的回归测试优先级错误定位方法,该方法将联合依赖图、基于程序行为特征的错误定位方法和回归测试优先级进行有机结合。实验结果表明,与 Ochiai,Tarantula,PPDG,CP 和 Naish 等经典方法相比,该方法可更加有效地定位软件错误。

关键词 错误定位,测试用例,回归测试优先级,联合依赖图

中图法分类号 TP311.5 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.10.034

Regression Testing Prioritization Fault Localization Method Based on Influence Analysis

ZHANG Hui
(School of Computer Science and Engineering, Southeast University, Nanjing 211189, China)

Abstract Since the fault localization method based on programs' behavior characteristics sees every program entity isolated, the efficiency of fault localization is influenced. And since the regression test fault localization needs to execute all test cases, the developing and testing costs increase largely. In view of the above problems, this paper put up a regression testing prioritization fault localization method based on influence analysis, which organically integrates the joint dependency graph, the fault localization method based on programs' behavior characteristics and the regression testing prioritization. The experimental results show that compared with classical methods such as Ochiai, Tarantula, PPDG, CP and Naish, this method can more efficiently position software errors.

Keywords Fault localization, Test case, Regression testing prioritization, Joint dependency graph

随着社会的进步,计算机在世界各个领域不断发展,越来越庞大复杂的软件规模使得保证软件质量工作变得日益艰难,为了解决这个问题,人们越来越重视软件测试。软件测试阶段主要分为单元测试、集成测试、系统测试和验收测试,这些测试阶段都需要先设计测试用例,然后运行测试用例,最后根据得到的实际结果与预期结果对比判断软件是否存在错误。测试到错误以后,开发人员还需要通过调试定位错误和修复错误,软件错误的不稳定性等特点使得定位错误成为最困难的一步。

目前的调试工具主要是集成开发环境中的调试器,如 Eclipse、Visual Studio 等,这些调试器通过设置断点,执行程序查看中间结果,再逐步分析程序运行结果来定位程序错误,这种调试方法需要开发人员对程序的逻辑、结构、功能、语义等特点都非常了解,并且有着丰富的经验直觉去设置断点。由于调试程序需要从程序的入口开始逐条检查每一行执行语句,将所有语句错误的可能性看作是一样的,而不是按照一定的优先级考虑语句检查顺序,因此需要花费大量的时间和精力。

为了解决手工操作调试器的问题,近几十年来,人们提出了自动化错误定位方法。通过自动化错误定位方法缩小搜寻错误的范围,然后再通过手工方式找到错误根源。目前其主要分为3个类型:基于程序依赖的错误定位方法[1-4]、基于程序状态修改的错误定位方法[5-8]和基于程序行为特征的错误定位方法[9-11,14,15]。以上3种技术中,基于程序行为特征的错误定位方法由于不对程序内部构造进行分析,只关心执行信息,计算方式也相对简单,因此被越来越多的研究者关注,成为错误定位方法的一个热点。但是这种方法是将每个程序实体孤立地看待,忽视了错误可能在程序实体间进行传播。从错误根源到错误传播再到程序失效,这期间可能是很远的距离,因为一个缺陷引发一个故障需要3个阶段:缺陷被执行,并感染程序状态;感染通过数据流或者控制流进行传播;感染最终引起故障。所以仅仅利用基于程序行为特征的方法往往会定位到与真正错误相距较远的语句。因此,我们需要对目前的基于程序行为特征的错误定位方法进行改进,增加分析程序的内部构造,即在基于程序行为特征的错误定位方法的基础上,通过影响分析方法得到最终可疑度排行来提高基于程序行为特征的错误定位方法的效率。

当程序中的错误修复完成以后,需要进行回归测试,回归测试结束再进行调试(即错误定位和修复),回归测试与调试是循环进行的,直到所有测试通过为止。不论对于回归测试还是回归测试错误定位,最好的方法都是将所有的测试用例执行一遍,保证测试和定位的充分性,但是这将大大增加开发和测试成本。为了解决这个问题,我们选择测试用例优化的方法。在本文中,回归测试只考虑对前面修复错误的检测(包

到稿日期:2015-09-10 返修日期:2016-05-31

张 慧(1982—),女,博士生,工程师,主要研究方向为软件测试、错误定位,E-mail:794014804@qq.com。

括由于修复引入新错误),不考虑增加新模块等版本升级。将上面基于影响分析的错误定位方法的最终可疑度排行作为回归测试用例优先级排序[16]的基础,根据排序后的测试用例进行回归测试错误定位。

本文的主要贡献有:通过引入联合依赖图分析程序内部构造之间的关联;然后将这种影响分析得到的可疑度排行对Ochiai[11]的语句可疑度排行进行重新排序,得到新的语句可疑度排行,同时计算已有测试用例覆盖语句的新的语句可疑度排行总和,数值越大的测试用例的优先级越靠前;最后利用新的测试用例优先级对已经修复过错误的程序进行回归测试错误定位。

本文第 1 节使用完整的例子说明了基于影响分析的错误定位方法;第 2 节继续第 1 节的例子说明了基于影响分析的回归测试优先级错误定位;第 3 节给出验证本文方法有效性的比较实验及其结果;第 4 节给出了相关工作;最后总结全文并展望下一步工作。

1 基于影响分析的错误定位方法

在程序中,有 if,while 等谓词判断逻辑的真假以及变量的定义与引用,这分别属于控制依赖和数据依赖。控制依赖的判断影响着程序的走向,数据依赖的变量影响着变量的值的变化。控制依赖和数据依赖对程序的影响可以统称为影响分析。因此在本文中,影响分析覆盖了控制依赖和数据依赖,并且提出了与影响分析相关联的联合依赖图,即通过联合依赖图找出程序中的实体间的影响分析。

1.1 程序依赖图转换为联合依赖图

联合依赖图是在程序依赖图的基础上建立的,而程序依赖图是在控制依赖图和数据依赖图的基础上建立的,下面将介绍程序依赖图和联合依赖图。

定义 1(程序依赖图[17]) 程序依赖图 PDG(Program Dependence Graph)是一个有向图。图中的节点表示程序中的语句或者是谓词表达式(操作符或者是操作数),图中与节点相连的边分别表示节点操作依赖的数据值(数据依赖)或者操作执行所依赖的控制条件(控制依赖)。由于程序依赖图是把程序中的每一个操作符都用确切的控制依赖和数据依赖表示,因此该图是对一个程序依赖关系的描述。

表 1 is_num_constant()方法实例及测试用例覆盖信息及测试结果

	出错语句 s5 应为 while(*(str+i)! ='\0'){	测试用例覆盖信息及测试结果 t_1	t_2	t_3	t_4	t_5	t_6	t_7
1	int is_num_constant(token str){							
2	char ch;							
3	int i=1;	●	●	●	●	●	●	●
4	if(isdigit(*str)){//if	●	●	●	●	●	●	●
5	while(*(str+i+1)!='\0'){	●	●	●		●	●	●
6	ch=*(str+i);	●	●	●				●
7	if(isdigit(ch))	●	●	●				●
8	i++;	●		●				●
9	else							
10	return(FALSE);		●					●
11	}/*end WHILE*/							
12	return(TRUE);	●		●		●	●	
13	}//end if							
14	else							
15	return(FALSE);				●			
16	}	F	P	P	P	P	F	P

为了更好地说明控制依赖、数据依赖、程序依赖图之间的关系,下面使用 Siemens 套件中 print_tokens2 程序的is_num_constant()方法举实例说明(如表 1 所列),is_num_constant()方法主要判断一个字符串是否全由数字组成,并返回相应的真假逻辑值。由于基于影响分析的错误定位方法需要最初的语句可疑度,即利用 Ochiai 计算每条语句的可疑度,因此在程序实例旁附上测试用例覆盖信息及测试结果。

is_num_constant()方法实例的程序依赖图如图 1 所示,图中实线表示控制依赖,虚线表示数据依赖。

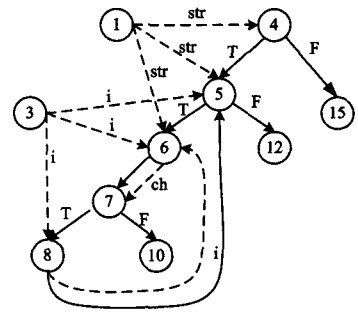

图 1 is_num_constant()方法实例的程序依赖图

图 2 中,出现一个节点的多个控制依赖边的前驱或后继和多个数据依赖边的前驱或后继,并且谓词节点同时出现谓词的结果和一组动态数据依赖,这使得每个节点不能都有各自的状态集或者变量集。为了后续的分析和计算,将一些节点(同时被控制依赖边前驱和后继以及数据依赖边前驱和后继连接)进行分解,即将同时含有状态集和变量集的节点进行分解,使得每个节点有各自的状态集或者变量集。例如图 2 中,节点 5 被分成 D5 和 5 两个节点,节点 D5 连接数据依赖的前驱和后继,节点 5 连接控制依赖的前驱和后继,其中一些连接数据依赖的节点需要连接控制依赖节点,例如 D5 和 5 两个节点或者其他节点连接节点 5 需要经历 D5,所以在连接数据依赖的节点上出现了实线,最后得到的联合依赖图如图 2 所示。

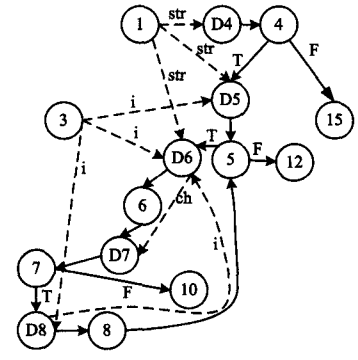

图 2 is_num_constant()方法实例的联合依赖图

定义 2(联合依赖图) 联合依赖图是一个二元组(S,E),其中$S=(N,DN,SV)$,N为联合依赖图的节点集合,DN为联合依赖图的数据依赖节点的集合,SV为联合依赖图的节点状态集和变量集的集合。其中有数据依赖入边的节点$n'\in N$,都有$dn\in DN$。对于任意一个联合依赖图中的节点$n\in N$,都有其对应的状态集和变量集$sv(n)\in SV$。E为含有权值的控制依赖边和数据依赖边构成的有向边集,其中控制依赖边的权值为 T(True)或 F(False),数据依赖边的权值为其变量。

1.2 基于联合依赖的可疑度排行的计算

1.2.1 初始可疑度计算

在基于程序行为特征的错误定位方法中，经过实验数据的对比，Ochiai 的错误定位效率强于 Tarantula 等技术，所以本文在计算节点的初始语句可疑度时利用 Ochiai 公式（见式(1)）来计算。

$$Ochiai(s)=\frac{failed(s)}{\sqrt{totalfailed(s)\times(passed(s)+failed(s))}} \tag{1}$$

其中，$passed(s)$和 $failed(s)$分别表示程序实体 s 成功执行和失败执行的次数；$totalfailed(s)$表示失败运行的测试用例数。

根据式(1)计算表 1 各条语句的可疑度，结果如表 2 所列。

表 2 is_num_constant()方法实例的 Ochiai 可疑度

语句	Ochiai 可疑度
3	0.54
4	0.54
5	0.58
6	0.35
7	0.35
8	0.41
10	0
12	0.71
15	0

在 is_num_constant()方法中，s5 是真正的错误语句，但是表 2 中 s12 得到最高的可疑度 0.71，出现该情况的原因可能是错误通过控制依赖或数据依赖影响了其他的语句，使得错误在执行到错误语句时没有表现错误，而在与错误语句相互影响的语句中出现失效，并且在表 1 的测试用例覆盖及测试结果中发现，测试用例 t_1 和 t_3，t_5 和 t_6 分别覆盖的信息相同，但是得到的测试结果不同，可以断定出现了偶然性正确的测试用例，所以在计算语句的可疑度时仅仅计算测试用例的覆盖信息和测试结果是远远不够的，因此提出了基于影响分析的方法来解决这些问题。

1.2.2 联合依赖的可疑度排行的计算

联合依赖的计算过程如图 3 所示。

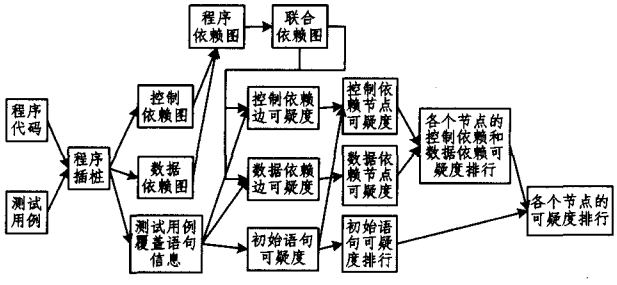

图 3 联合依赖的可疑度排行的计算过程

(1)控制依赖边和数据依赖边的计算

令 $T=\{t_1,t_2,\cdots,t_n\}$表示出错程序 P 的一个测试组件，其中每个测试用例用 $t_k(1\leqslant k\leqslant n)$表示。$T$ 可以被分为两个互不相交的测试集。$T_\surd$ 和 $T_\times$ 分别表示通过和失败测试集。

给定一条动态依赖关系，$\varepsilon=(s_1,s_2)$，ε 包含了两条语句：s_1 和 s_2。称 s_1 支配 s_2 或者 s_2 依赖于 s_1。进一步使用 ε_c 来表示一条动态控制依赖关系，而使用 ε_d 来表示一条动态数据依赖关系。同时，本文使用 $\varepsilon_c(s,i)$和 $\varepsilon_c(s,j)$来分别强调有谓词 s 支配的两条分支。记号 $\varepsilon_d(s,\cdot)$和 $\varepsilon_d(\cdot,s)$用来相应地表示语句 s 的数据依赖语句集和 s 的数据依赖支配语句集。记号 $\varepsilon_c(s,\cdot)$和 $\varepsilon_c(\cdot,s)$也可以被相应地解释。

与控制依赖相关的错误指选择、循环的谓词错误导致程序进入了错误的控制流路径，从而产生的软件错误。

与数据依赖相关的错误指在程序执行过程中，变量定义、使用时产生的错误。

本文利用测试用例覆盖语句信息及控制依赖图和数据依赖图，分别根据控制依赖和数据依赖的走向得到控制依赖边和数据依赖边的测试用例覆盖语句信息，进而利用式(1)得到控制依赖边可疑度和数据依赖边可疑度。

以表 1 is_num_constant()方法为例，首先根据表 1 的测试用例覆盖信息和图 2 的联合依赖图，分别得到控制依赖覆盖的 Ochiai 可疑度以及数据依赖覆盖的 Ochiai 可疑度，如表 3 和表 4 所列。

表 3 is_num_constant()方法实例的控制依赖覆盖

控制依赖	t_1	t_2	t_3	t_4	t_5	t_6	t_7	Ochiai 可疑度
(4,5)T	●	●	●		●	●	●	0.58
(4,15)F				●				0
(5,6)T	●	●	●				●	0.35
(5,12)F	●		●		●	●		0.71
(6,7,8)T	●		●				●	0.41
(6,7,10)F		●					●	0
(8,5)	●		●				●	0.41
测试用例结果	F	P	P	P	P	F	P	

表 4 is_num_constant()方法实例的数据依赖覆盖

控制依赖	t_1	t_2	t_3	t_4	t_5	t_6	t_7	Ochiai 可疑度
(1,4)str	●	●	●	●	●	●	●	0.54
(1,5)str	●	●	●		●	●	●	0.58
(1,6)str	●	●	●				●	0.35
(3,5)i	●	●	●		●	●	●	0.58
(3,6)i	●	●	●				●	0.35
(3,8)i	●		●				●	0.41
(6,7)ch	●	●	●				●	0.35
(8,6)i	●		●				●	0.41
测试用例结果	F	P	P	P	P	F	P	

表 3 中由于语句 6,7 属于一个基本块，因此出现(6,7,8)T 和(6,7,10)F 控制依赖分支，之所以没有利用基本块概念，是因为本文的影响分析需同时考虑控制依赖和数据依赖，如果利用基本块将会造成数据依赖分析的不便。

(2)控制依赖节点可疑度和数据依赖节点可疑度的计算

在控制依赖节点可疑度计算中，考虑程序中的初始控制依赖节点的可疑度为式(1)计算的可疑度，即 $s_0=Ochiai(s_0)$，s_0 控制依赖的分支分别为 $\varepsilon_c(s_0,s_i)$和 $\varepsilon_c(s_0,s_j)$，$s_i=Ochiai(s_0)+\varepsilon_c(s_0,s_i)$，存在控制依赖关系 $\varepsilon_c(s_i,s_k)$，$s_k=s_i+\varepsilon_c(s_i,s_k)$。由此可以计算得到控制依赖节点可疑度：

$$s_c=\begin{cases}s_0=Ochiai(s_0)\\ s_i=s_0+\sum_0^i\varepsilon_c(s_j,s_k),0\leqslant j\leqslant i,1\leqslant k\leqslant i\end{cases} \tag{2}$$

其中，s_c 表示控制依赖节点 s 的可疑度。以表 1 中的 is_num_constant()方法为例，根据图 2 和表 3 可以得到控制依赖的 5 条路经：{4F,15}；{4T,5F,12}；{4T,5T,6,7F,10}；{4T,5T,6,7T,8,5F,12}；{4T,5T,6,7T,8,5F,6,7F,10}。其中，5T,6,7T,8,5T,6,7T,8…为循环路径，本文的联合依赖计算公式需要计算语句执行次数，所以假设循环语句只执行一次，防止

循环语句的可疑度过高。节点 4 为控制依赖图的起点，在图 2 中其 Ochiai 可疑度为 0.54，即 $s_c(4)=0.54$，在表 3 中有两个分支，4T 的控制依赖可疑度为 0.58，4F 的控制依赖可疑度为 0。根据后续节点受到前驱结点的影响，节点 15 的控制依赖可疑度 $s_c(15)=0.54$。节点 5 的控制依赖可疑度 $s_c'(5)=1.12$。节点 5 有两个分支，5T 的控制依赖可疑度为 0.35，5F 的控制依赖可疑度为 0.71。节点 12 的控制依赖可疑度 $s_c(12)=1.83$。节点 6 的控制依赖可疑度 $s_c(6)=1.47$。节点 6，7 相连没有出现分支，因此节点 7 的控制依赖可疑度仍为 1.47。节点 7 有两个分支，7T 的控制依赖可疑度为 0.41，7F 的控制依赖可疑度为 0。所以节点 10 的控制依赖可疑度 $s_c(10)=1.47$。节点 8 的控制依赖可疑度 $s_c(8)=1.88$。最后，作为循环返回边，节点 8，5 路径的控制依赖可疑度为 0.41，节点 5 的控制依赖可疑度得到更新为 $s_c'(5)+\varepsilon_c(8,5)=1.53$。由于节点 8，5 是循环返回边，不是选择分支，因此节点 8，5 路径的控制依赖可疑度对节点 5 产生作用，不考虑对节点 5 的其他后继节点产生作用，其他后继节点的控制依赖可疑度不变化。

由于控制依赖主要关心分支的选择，因此在计算节点的控制依赖可疑度时需要考虑父节点的控制依赖可疑度和控制依赖边的可疑度，而在数据依赖节点可疑度计算中，考虑变量的定义和引用通过数据依赖边进行变量的传递，每个数据依赖节点都受到数据依赖入边的影响，所以只计算各节点的数据依赖边的入边的可疑度，得到式(3)：

$$s_d=\sum_{s\in DN}\varepsilon_d(\cdot,s) \tag{3}$$

其中，s_d 表示数据依赖节点 s 的可疑度。根据图 2 和表 4 可以得到 8 条数据依赖边，其中节点 1 和节点 3 只有出边没有入边，即没有前驱节点，所以数据依赖可疑度为 0。节点 4 只有节点 1 一个前驱节点，根据表 4 节点 4 的数据依赖可疑度 $s_d(4)=0.54$。节点 5 有两个前驱节点，分别为节点 1 和节点 3，根据表 4 节点 5 的数据依赖可疑度 $s_d(5)=\varepsilon_d(1,5)+\varepsilon_d(3,5)=1.16$。节点 8 只有一个前驱节点 3，根据表 4 节点 8 的数据依赖可疑度 $s_d(8)=0.41$。节点 6 的前驱节点包括节点 1，3 和 8，根据表 4 节点 6 的数据依赖可疑度 $s_d(6)=\varepsilon_d(1,6)+\varepsilon_d(3,6)+\varepsilon_d(8,6)=1.11$。节点 7 只有一个前驱节点 6，根据表 4 节点 7 的数据依赖可疑度 $s_d(7)=\varepsilon_d(6,7)=0.35$。

(3)各个节点的可疑度排行

根据以上得到的各节点的控制依赖可疑度和数据依赖可疑度，在程序中，节点关联的控制依赖边和数据依赖边越多，越容易找到错误的根源，在这里加入了 Ochiai 可疑度，由于可疑度越大，出错的可能性越大，与节点关联越多的控制依赖边和数据依赖边产生双保险。于是将各节点的控制依赖可疑度和数据依赖可疑度相加，得到式(4)：

$$s_{cd}=s_c+s_d \tag{4}$$

其中，s_{cd} 表示各个节点的控制依赖节点可疑度与数据依赖节点可疑度之和。以表 1 中的 is_num_constant()方法为例，节点 4 的可疑度为 1.08；节点 5 的可疑度为 2.69；节点 6 的可疑度为 2.58；节点 7 的可疑度为 1.82；节点 8 的可疑度为 2.29；节点 10 的可疑度为 1.47；节点 12 的可疑度为 1.83；节点 15 的可疑度为 0.54。

为了计算各个节点的最终可疑度排行，本文将 Ochiai 可疑度排行加上控制依赖节点可疑度及数据依赖节点可疑度之和的排行，得到最终可疑度排行，如式(5)所示：

$$sus(s)=susOchiai(s)+sus_{cd}(s) \tag{5}$$

其中，$sus(s)$ 表示语句最终可疑度排行，$susOchiai(s)$ 表示语句 Ochiai 可疑度排行，$sus_{cd}(s)$ 表示控制依赖节点可疑度及数据依赖节点可疑度之和的排行。这样做的目的是改进基于程序行为特征的错误定位方法，将与控制依赖边和数据依赖边相关联越多并且可疑度最多的节点赋以最高的可疑度排行，即最终可疑度排行中加入了程序影响分析的因素。

以表 1 中的 is_num_constant()方法为例，最终的可疑度排行如表 5 所列。

表 5　is_num_constant()方法实例的最终可疑度排行

语句	Ochiai 排行	可疑度排行	最终可疑度排行
3	3	9	12
4	3	7	10
5	2	1	3
6	6	2	8
7	6	5	11
8	5	3	8
10	8	6	14
12	1	4	5
15	8	8	16

表 5 中，第 3 列可疑度排行表示的是控制依赖节点可疑度与数据依赖节点可疑度相加得到的可疑度排行。从表 5 可以看出，原来在 Ochiai 排行中，排行最高的第 12 条语句在最终可疑度排行中降为 5，而在 Ochiai 排行中，排行为 2 的真正错误的第 5 条语句在最终可疑度排行中升为此函数语句中排行最高的语句，因此帮助了调试者快速定位到错误语句。

2　基于影响分析的回归测试优先级错误定位

第 1 节得到了各条语句的最终可疑度排行，当程序中的错误修复完成以后，需进行回归测试，回归测试结束再进行调试(即错误定位和修复)，回归测试与调试是循环进行的，直到测试通过为止。其中需要对修改的程序及与修改程序相关的程序进行检测，避免有新错误引入。所以不论对于回归测试还是回归测试错误定位，最好的方法都是将所有的测试用例都执行一遍，保证测试和定位的充分性，但这将大大增加开发和测试成本，在时间和精力上是难以完成的。为了解决这个问题，本节考虑将第 1 节的最终可疑度排行引入到回归测试的测试用例的优先级排序中，通过测试用例优先级排序使得大部分的错误在执行少部分的测试用例以后就可以检测和定位出来。首先，最终可疑度排行越靠前的语句其控制依赖可疑度与数据依赖可疑度之和越大，即可能关联控制依赖边与数据依赖边越多，语句的嵌套选择分支也越多，可疑度越大，依据“80%的错误出现在 20%的程序中[13]”的原则，这样的语句越靠前执行，得到的错误越多，错误定位的效率也有所提高。为此计算所有测试用例覆盖的语句的最终可疑度排行之和，其中将失败与成功的测试用例分开，由于失败的测试用例含有错误语句的可能性较大，以表 1 中的 is_num_constant()方法为例，根据表 1 中的测试用例的覆盖信息和表 5 的最终可疑度排行得到表 6 中各个测试用例的最终可疑度排行之和。

表 6 is_num_constant()方法实例测试用例覆盖信息的最终可疑度排行之和

测试用例	语句可疑度排行之和	通过和失败测试集
t_1	57	T_x
t_6	30	
t_7	66	$T_\surd$
t_2	58	
t_3	57	
t_4	38	
t_5	30	

由表 6 可以看出，失败的测试用例 t_1 应该在 t_6 之前先执行；成功的测试用例按照 t_7，t_2，t_3 顺序先执行，再按照 t_4，t_5 顺序后执行。出现程序丢失时，通过联合依赖可以利用语句之间的上下文将有直接依赖关系的语句或者邻近的一条可执行语句作为程序的出错点定位出来。当出现偶然性正确的测试用例时，比如表 6 中，t_1 和 t_3 及 t_5 和 t_6 分别覆盖的语句相同，但是得到的测试结果不同，有可能存在偶然性正确的测试用例。在失败的测试用例中，t_1 是第一个被执行的，含有错误语句的可能性很大；在成功的测试用例中，t_3 是第三个被执行的，次序还比较靠前，有助于找到错误语句，分辨出偶然性正确的测试用例。在失败的测试用例中，t_6 是最后一个被执行的，含有错误语句的可能性较小；在成功的测试用例中，t_5 也是最后一个被执行的，说明含有错误语句的可能性也较小。所以依据以上顺序执行测试用例，可以快速检测和定位到回归测试中错误语句及与错误语句关联的程序是否都已经修改，并且是否引入新错误，从而快速定位到未修复的错误。

3 实验

本文通过 Ochiai 初始语句可疑度排行加上控制依赖节点可疑度和数据依赖节点可疑度之和排行得到最终可疑度排行；再利用最终可疑度排行得到各个测试用例中覆盖语句的最终可疑度排行之和；最后利用最终可疑度排行之和对测试用例进行回归测试用例优化，将优化后的测试用例再进行回归测试错误定位。故本文的实验主要研究如下两个问题：

(1)在基于程序行为特征的错误定位方法中，分别使用优化前与优化后的回归测试用例用于 Ochiai 方法，比较两组测试用例在错误定位效率上的差异；

(2)在确保利用优化的回归测试用例后，将本文的错误定位方法与 Ochiai[11]，Tarantula[10]，PPDG[4]，CP[12]，Naish[14] 方法的错误定位效率相比较。

3.1 实验数据

本文的实验对象选择 Siemens Suite 和 space，具体实验数据如表 7 所列。表中第 1 列为实验程序的名称，第 2 列为每个实验程序对应的错误版本个数，第 3 列为实验程序可执行代码的行数，第 4 列为每个实验程序提供的测试用例个数，第 5 列为每个实验程序的功能描述。其中 Siemens Suite 包括 7 组不同功能的 C 程序，每组程序通过人工注入错误的方法建立了实验程序的错误版本以模拟实际项目中可能存在的错误。space 是由欧洲航天局开发，与 Siemens Suite 相比较，space 拥有更多的代码行数和测试用例，它包含 6218 行可执行代码和 38 个错误版本，这些错误都是在开发过程中产生的。

表 7 实验数据

Program	No.	Line	Num	Description
print_tokens	7	472	4130	Lexical analyzer
print_tokens2	10	399	4115	Lexical analyzer
replace	32	512	5542	Pattern replacement
schedule	9	292	2650	Priority scheduler
Schedule2	10	301	2710	Priority scheduler
tcas	41	141	1608	Altitude separation
tot_info	23	440	1052	Information measure
space	38	6218	13585	Array definition interpreter

3.2 实验度量方法

本文方法会产生语句的可疑度排序，根据可疑度大小降序排列。从排在第一位的语句开始查找，直到找到错误语句为止。把这期间所花费的精力作为软件错误定位方法的度量方法，更精确地说，就是将错误语句在可疑度排序中的排行除以程序代码行数得到的比值(percentage of code examed)作为评价一个错误定位方法的定位精度好坏的度量指标。它的计算方法如式(6)所示。

$$T=\frac{rank\ of\ faulty\ statement}{number\ of\ executable\ statements}\times 100\% \tag{6}$$

3.3 实验步骤

本实验通过以下几步进行实验数据的收集和分析。

(1)利用插桩工具收集测试用例覆盖信息和控制依赖、数据依赖的信息。

(2)利用上面的信息构造联合依赖图，其中将同时含有控制依赖和数据依赖的节点分为控制依赖节点和数据依赖节点；再利用(1)的信息和联合依赖图分别计算控制依赖边可疑度、数据依赖边可疑度和初始语句可疑度。

(3)根据(2)得到的信息，利用公式得到控制依赖节点可疑度和数据依赖节点可疑度。

(4)将(3)中的控制依赖节点可疑度与数据依赖节点可疑度相加，得到各个节点的控制依赖和数据依赖可疑度排行。

(5)将(2)得到的初始语句可疑度排行与(4)得到的各个节点的控制依赖和数据依赖可疑度排行相加，得到最终可疑度排行。

(6)利用最终可疑度排行定位并修复实验数据中一些程序中的错误。

(7)计算测试用例覆盖语句的最终可疑度之和。

(8)将(7)中的最终可疑度之和的测试用例降序排列，并按照失败测试用例和成功的测试用例进行分类，失败测试用例先执行，成功的测试用例后执行。

(9)按照(8)中测试用例优先级的排序方式对(6)中修复一部分的实验数据的程序进行回归测试错误定位。

(10)分析统计优化前和(8)中优化的测试用例用于 Ochiai 方法的错误定位效率，以及分析统计当前流行的 6 种错误定位方法的效率，并与本文的方法进行比较。

3.4 实验结果

为了验证本文提出的错误定位方法的有效性，利用 Ochiai 方法和优化前的回归测试用例与 Ochiai 方法与优化后

的回归测试用例进行比较，以及本文方法与 Ochiai，Tarantula，PPDG，CP 和 Naish 方法的比较的实验结果如图 4、图 6 所示，X 轴为查找到错误语句需要检查语句的百分比，Y 轴为成功定位到错误版本的比率，记为 *fault_percent*，如式(7)所示：

$$fault\_percent=\frac{\text{成功定位错误版本数}}{\text{总版本数}}\times 100\% \quad (7)$$

本节针对问题 1 和问题 2 这两项研究收集了相关数据，并分析了本文方法的错误定位效率。

3.4.1 问题 1

从图 4 可以看出两组测试用例，一组是优化前的测试用例，另一组是优化后的测试用例。优化前的测试用例用的是 Siemens Suite 自带的测试用例，优化后的测试用例用的是经过联合依赖图、Ochiai 语句可疑度等公式计算得到的最终可疑度排行，利用最终可疑度排行得到的优化的测试用例。在本实验中，将优化前的测试用例和优化后的测试用例同时用于 Ochiai 基于程序行为特征的错误定位方法中，并在[0%，20%]区间比较两组测试用例的错误定位的效率。图 4 中，在只检查 5%的可执行代码的情况下，优化前测试用例和优化后测试用例分别定位出 8%和 12%的错误；在只检查 10%的可执行代码的情况下，优化前测试用例和优化后测试用例分别定位出 18%和 22%的错误，两者区分不是很明显。然而，在检查 15%的可执行代码的情况下，优化前测试用例和优化后测试用例分别定位出 23%和 41%的错误，差距非常明显，由于在优化的测试用例中主要考虑的是最终可疑度排行，而可疑度排行主要考虑的是语句自身的可疑度、控制依赖传播和数据依赖传播的可疑度，因此越是提前执行的测试用例其关联的控制依赖边和数据依赖边越多，可疑度也越大，越是复杂的程序越是能在关联越多的地方找到越多的错误，故优化的测试用例易于定位到越多的错误。特别在检查 20%的可执行代码的情况下，优化前测试用例和优化后测试用例分别定位出 38%和 62%的错误，两者的差距最大，并且经过优化的测试用例中已经提前执行的测试用例可以定位到一大半的错误。综上所述，优化后测试用例的错误定位效率要明显优于优化前测试用例的错误定位效率。

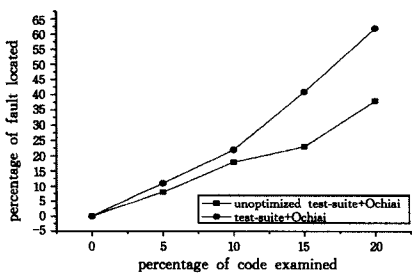

图 4 在[0%，20%]区间，优化前和优化后的回归测试用例用于 Ochiai 错误定位方法的效率比较

3.4.2 问题 2

本节分析统计了本文方法的效率，并与 Ochiai 方法、Tarantula 方法、PPDG 方法、CP 和 Naish 方法进行了比较。Ochiai 方法在前面已有介绍，Tarantula 方法、Naish 方法与 Ochiai 方法相似，其中 Naish 方法分为 Naish1 和 Naish2，4 种方法只是计算公式有异，同样认为被越多的失败的测试用例执行的语句出错的概率越大。PPDG 方法和 CP 方法在后面的相关工作中进行介绍。

在图 5 中，在只检查 10%的可执行代码的情况下，Ochiai 方法和 Tarantula 方法分别定位出 22%和 58%的错误，两者都是基于程序行为特征的错误定位方法，差异如此悬殊可能是两者的计算公式的差异造成可疑度排行不同。但是，在检查多于 20%的如 30%、40%等可执行代码的情况下，Ochiai 表现了强劲的势头，Ochiai 方法定位的错误越来越多于 Tarantula 方法，特别在检查 70%的可执行代码的情况下，Ochiai 方法和 Tarantula 方法分别定位出 100%和 82%的错误。可以断定在大部分情况下，Ochiai 方法的错误定位效率要明显优于 Tarantula 方法的错误定位效率。然而在只检查 10%的可执行代码情况下，Ochiai 方法、Naish1 和 Naish2 方法分别定位 22%，60%和 72%的错误，在检查多于 10%的比如 20%、30%等可执行代码情况下，Naish1 和 Naish2 方法的错误定位效率要强于 Ochiai 方法，这是由于 Naish 方法分别考虑了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)。其中 Naish2 方法首先考虑了失败的测试用例执行语句的个数，然后考虑测试用例执行语句的个数，所以在错误定位效率上强于认为执行出错语句的成功的测试用例数会较少，未执行出错语句但通过的测试用例数就会较大的 Naish1 方法。因为本文方法、PPDG 方法和 CP 方法都是依赖控制依赖和数据依赖，所以在后续比较有依赖关系的错误定位方法与基于程序行为特征的错误定位方法中，基于程序行为特征的错误定位方法采用 Naish2 方法。作为考虑依赖关系的 PPDG 方法和 CP 方法，在只检查 10%的可执行代码情况下，PPDG 方法和 CP 方法分别定位出 43%和 59%的错误。在检查 20%的可执行代码情况下，PPDG 方法和 CP 方法分别定位出 71%和 72%的错误，结果非常接近。但是在检查 30%的可执行代码情况下出现差异，PPDG 方法和 CP 方法分别定位出 76%和 88%的错误。当检查 60%的可执行代码情况下，PPDG 方法和 CP 方法分别定位出 90%和 100%的错误。综上所述可以看出，CP 方法的错误定位效率要强于 PPDG 方法的错误定位效率。出现该情况的原因可能是 PPDG 方法虽然提及了错误理解的思想，然而它的错误理解是简单的并且未考虑到错误的传播性。而 CP 方法至少考虑了控制依赖的错误传播性，对于定位于控制依赖相关的错误要强于 PPDG 方法。将 Naish2 方法与 CP 方法进行比较可以发现，在检查 10%的可执行代码情况下，Naish2 方法和 CP 方法分别定位出 72%和 59%的错误，在检查 20%的可执行代码情况下，Naish2 方法和 CP 方法分别定位出 88%和 71%的错误，但是在检查 30%，40%等可执行代码情况下，Naish2 方法定位到的错误数与 CP 方法定位到的错误数不断接近但是 Naish2 方法的错误定位效率始终高于 CP 方法。并且在检查 60%可执行代码情况下，Naish2 方法和 CP 方法才可以定位出 100%的错误。综上所述，Naish2 方法的错误定位效率要强于 CP 方法的错误定位效率。其原因可能是 CP 主要是面向控制依赖边，没有考虑数据依赖边和每个节点自身的可疑度，而 Naish2 主要考虑了节点自身的可疑度。最后，将本文方法与 Naish2 方法进行比较，在检查 10%的可执行代码情况下，本文方法和 Naish2 方法分别定位出 63%和 72%的错误，两种方法的差距明显，在定位小于 30%的可执行代码情况下，Naish2 方法的错误定位效率强于本文方法，这可能是由于出现的错误主要是节点自身产生的，大多数没有进行错误传播。然而在检查 40%的可执行代码情况下，本文方法和

Naish2 方法分别定位出 100%和 95%的错误，但是 Naish2 方法只有在检查了 60%的可执行代码情况下才可以定位出 100%的错误。综上所述可以看出，大部分情况下，本文的方法要强于 Naish2 方法，其主要原因是本文方法在计算最终可疑度排行时不仅考虑了节点自身的可疑度，还将控制依赖错误传播和数据依赖错误传播考虑进来，可以定位到节点自身的错误、控制依赖错误、数据依赖错误及控制依赖和数据依赖共同产生的错误。所以与前面 6 种方法相比较，所提方法的错误定位效率最好。

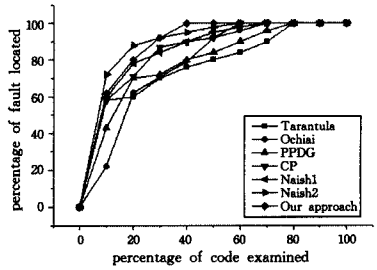

图 5　不同错误定位方法的定位效率比较

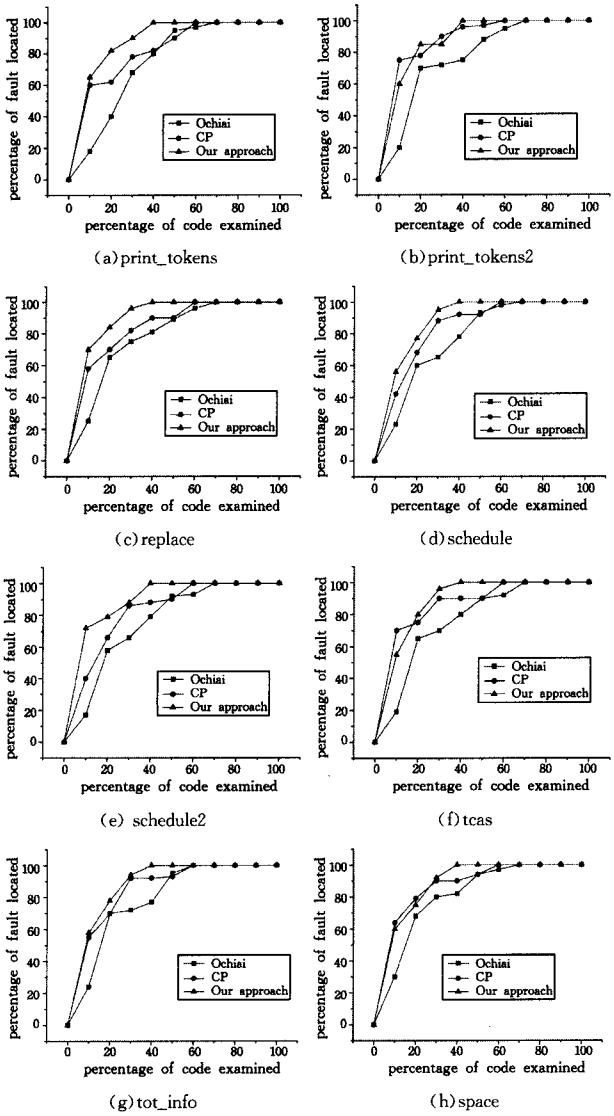

图 6　3 种错误定位方法在单个程序的定位效率比较

图 6 为 3 种错误定位方法在每个实验程序中的错误定位结果。在图 6(a)、(c)、(d)、(e)和(g)中，本文的方法在每个百分比上的错误定位效率都高于 CP 方法和 Tarantula 方法。而在图 6(b)、(f)和(h)中出现局部百分比上，本文的错误定位效率低于 CP 方法的现象，比如图 6(b)中在检查 10%和 30%的代码时错误定位效率低于 CP 方法。但是从每个实验程序中的错误定位结果可以看出，本文的方法通过检查最少的代码百分比就可以定位所有的错误。因此总体上，本文方法的错误定位效率高于 CP 方法和 Tarantula 方法。

3.5　实验讨论

通过上述实验的结果和分析，本文中优化后的测试用例的错误定位效率高于未实现优化的测试用例的错误定位效率。另外，本文的错误定位方法比基于程序行为特征的 Ochiai 方法、Tarantula 和 Naish 方法以及基于程序依赖的 PPDG 方法和 CP 方法的效率有了进一步的提高。但我们的实验过程中还存在着一些问题。

首先，循环语句只考虑一次执行次数，虽然这样可以防止可疑度过高，但是在实际的程序执行中是不可取的，如何既能考虑循环语句的执行次数，又能不让可疑度过高是需要进一步研究的问题。除此以外，对于偶然性正确的测试用例，本文考虑的是分析控制依赖可疑度和数据依赖可疑度，没有考虑成功测试用例与失败测试用例之间的执行路径的差异，需要进一步的研究。

4　相关工作

错误定位由于其困难性越来越受到研究者们的关注，本文在控制依赖和数据依赖的基础上提出了一种新的概念联合依赖，通过联合依赖图改进目前存在的基于程序行为特征的错误定位方法。所以本文涉及的知识点主要是基于程序行为特征的错误定位方法和控制依赖以及数据依赖，其中基于程序行为特征的错误定位方法已经在前文进行了阐述，本节中主要介绍基于控制依赖和数据依赖的错误定位方法。

CP(Capture Propagation)[12] 方法认为在程序执行过程中，错误语句会影响程序的执行状态，并且错误会随着程序的执行而传播。该方法首先将程序按照结构划分为基本块，然后在分析程序的控制依赖关系的基础上建立控制流图，并计算各个控制依赖边传播错误的可疑度。最后，根据后续模块的可疑度进行逆向推导，顺次计算前驱模块的可疑度，最终根据各模块的可疑度定位至可疑模块。该方法以控制流边为研究对象，采用的是面向边的技术，适合分析通过控制流边传播的错误。然而，错误并非都是控制依赖传播导致的，还有数据依赖传播以及非传播的错误，面对这些错误此方法的优势就不再明显了，并且该方法没有分析每个节点的具体执行状态。根据后续推导前驱的做法也不符合一般程序运行的逻辑过程。

PPDG(Probabilistic Program Dependence Graph)[4] 方法在分析程序控制依赖和数据依赖的基础上，通过对程序依赖图进行扩展，构建了概率程序依赖图(PPDG)，采用条件推理的方式来获取程序元素间的依赖和独立关系，进而定位与数据依赖相关的软件错误。PPDG 分析了程序语句间的条件依赖，进而计算节点间的条件依赖概率。而计算条件依赖概率的前提条件是假设该语句的父节点一定被执行，但是一般情况下，该父节点可以执行也可以不执行，因此必然会影响其对数据依赖相关的错误定位的精度。并且 PPDG 在错误定位过程中，使用多条成功测试用例和一条失败测试用例，但是大多数实验数据表明，多条成功测试用例和多条失败测试用例的错误定位精度高于多条成功测试用例和一条失败测试用例的。

本文与上述方法的不同之处主要体现在：首先，在程序依赖图的基础上提出了联合依赖，有效地区分控制依赖节点和数据依赖节点的状态集及变量集，并且公式中 Ochiai 的初始语句可疑度以及后面的控制依赖节点可疑度与数据依赖节点可疑度的计算都是建立在多条成功测试用例和多条失败测试用例的基础上。其次，在计算控制依赖节点的可疑度时，考虑的是最前驱的父节点语句可疑度加上从最前驱父节点到目的节点的所有控制依赖边的可疑度，这样的由前驱到后继的推导方法符合逻辑推算，因为只有父节点的触发才会引起子节点的触发。这里既考虑了各个节点自身与节点之间的执行状态及出现错误的可疑度，也考虑了控制依赖边可疑度对错误的控制依赖传播的作用。在计算数据依赖节点可疑度时，考虑的是各个数据依赖节点的前驱数据依赖边的可疑度对数据依赖节点的影响，即数据依赖边对错误的数据依赖传播的作用，同样考虑的是前驱对后继的影响，这里没有考虑像控制依赖节点一样加上父节点的可疑度的原因是：数据依赖主要是对变量的定义和引用，即变量集，不考虑父节点的执行状态即状态集，这样在进行控制依赖节点可疑度与数据依赖节点可疑度相加时避免出现数值冗余，且这样做只需要获取较少的程序信息，减少了各方面的开销。然后，将控制依赖节点可疑度与数据依赖节点可疑度相加得到各个节点可疑度，这个节点可疑度既考虑了控制依赖的错误传播又考虑了数据依赖的错误传播，将节点可疑度排行与 Ochiai 得到的初始语句可疑度排行相加得到最终可疑度排行，这个最终可疑度排行包括了各个节点自身的出错可疑度排行以及节点之间通过影响分析（控制依赖传播和数据依赖传播）得到的可疑度排行。最后，本文利用最终可疑度排行计算各个测试用例语句覆盖的最终可疑度之和，数值越大的测试用例优先级越高。由于测试用例数值越大，说明这个测试用例覆盖含有错误语句的可能性越大，将这样的测试用例提前对于回归测试、错误定位、回归测试错误定位都有很大的帮助。

结束语 本文提出了基于影响分析的回归测试优先级错误定位方法，这种方法首先利用 Ochiai 错误定位方法得到初始语句可疑度及可疑度排行；其次，利用控制依赖图和数据依赖图建立联合依赖图，通过初始语句可疑度和联合依赖图得到各个节点的可疑度和可疑度排行，将初始语句可疑度排行加上各个节点的可疑度排行得到最终可疑度排行；最后，利用最终可疑度排行计算各个测试用例语句覆盖的最终可疑度排行之和，数值越大的测试用例，优先级越高，利用按照优先级排序好的测试用例进行回归测试错误定位。通过实验研究与分析，本文方法能够提高定位程序中定位错误的效率。

在未来的工作中，我们还将致力于对本文提出的方法进行改进，针对失败的测试用例对实验结果的影响较大及成功的测试用例对可疑度也有影响的问题，研究设定一定的权重因子分别给予成功的测试用例和失败的测试用例，达到进一步提高错误定位的效率的目的。

参 考 文 献

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

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

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

[4] Baah G K, Podgurski A, Harrold M J. The probabilistic program dependence graph and its application to fault diagnosis[J]. IEEE Transactions on Software Engineering, 2010, 361: 528-545

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

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

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

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

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

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

[11] Abreu R, Zoeteweij P, Arjan J C, et al. On the Accuracy of Spectrum-Based Fault Localization[C]// In Proceedings of Testing: Academic and Industrial Conference-Practice and Research Techniques. 2007: 89-98

[12] Zhang Z Y, Chan W K, Tse T H. Capturing Propagation of Infected Program States[C]// 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). ACM, 2009: 43-52

[13] 在软件开发中的应用 80：20 原则[EB/OL]. http://www.kuqin.com/shuoit/20131120/336423.html. 2013

[14] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis [J]. ACM Transactions on Software Engineering and Methodology, 2011, 20(3): 1-32

[15] Chen Xiang, Ju Xiao-lin, Wen Wan-zhi, et al. Study on Dynamic Defect Location Method Based on Program Spectrum [J]. Journal of Software, 2015, 26(2): 390-412(in Chinese)
陈翔，鞠小林，文万志，等. 基于程序频谱的动态缺陷定位方法研究[J]. 软件学报，2015，26(2)：390-412

[16] Chen Xiang, Chen Ji-hong, Ju Xiao-lin, et al. Review of Priority Ranking of Test Cases in Regression Test[J]. Journal of Software, 2013, 24(8): 1695-1712(in Chinese)
陈翔，陈继红，鞠小林，等. 回归测试中的测试用例优先排序技术述评[J]. 软件学报，2013，24(8)：1695-1712

[17] 李博. 软件调试中多错误定位的算法研究[D]. 大连：大连海事大学计算机专业，2012