高技术通讯 2012 年 第22 卷 第8 期:797 ~ 803

doi:10.3772/j.issn.1002-0470.2012.08.004

一种基于运行差异统计的软件故障定位方法①

陈 荣② 洪丽娜

(大连海事大学信息科学技术学院 大连 116026)

摘 要 针对典型的软件故障定位方法需要将大量失效运行和成功运行相比较来定位有问题的语句,从而不可避免地存在大量运行路径冗余,导致定位效率和准确性降低的问题进行了研究,提出了一种基于运行差异统计的故障定位方法。该方法首先对所有的成功运行和失效运行聚类,消除运行路径冗余,然后对剩余数据通过对失效运行与成功运行的差异进行统计的方法得出故障报告,并对故障报告中的语句进行重要性排序。实验结果表明,这种方法比 Wang 的传统方法在定位效率和准确性方面都有很大提高。

关键词 故障定位,路径冗余,聚类,统计方法

0 引 言

软件故障(又称程序错误)对系统可靠性的影响是不可小视的。很多程序员仍然在使用传统的源程序级的调试工具,反复思考哪些程序语句出了问题,直至改正错误。这种传统的软件调试是一项复杂又耗时的过程。近年来出现的故障定位(fault localization)方法正在改变这种现状,它们能够在计算机的辅助下自动找到故障的位置,即定位有问题的语句。软件工程领域和人工智能领域已经出现了各种诊断程序错误的技术,有代表性的程序错误定位方法包括基于模型软件调试[1]、程序切片技术、错误解释[2]、轻量式规约辅助的错误定位[3]、Delta 调试[4]以及程序运行统计方法。对比程序的成功运行和失败运行是程序运行统计方法的一种典型思路[4-8]。对比是为了发现错误运行中的哪些点偏离了成功运行,而这些偏离可能就是导致软件故障的原因。根据比较方法的不同,这类故障定位方法又可分为基于距离度量的方法和基于特征统计的方法。基于距离度量的方法就是通过一定的距离度量技术寻找与失效运行最相近的一个成功运行,然后计算它与失效运行的差异进行故障定位。例如,Wang 等[9]提出使用控制流信息对齐两条执行路径,定性计算两者差异,把差异点视为故障点。使用定性差异,Wang 等在文献[10]中提出了一种自动生成与一条失效路径最相近的成功运行的方法。Renieris 等[7]提出了“近邻模型”(nearest neighbor),其结论是,与失效路径最相似的成功路径往往比随机选择的成功路径更加有助于定位故障。相比之下,基于特征统计的方法是通过对动态程序行为的统计信息进行分析来定位故障,即通过比较在两类运行里的程序元素的统计信息,如语句[6,11]或者谓词[12,13]统计信息,来找到与错误相关的语句(或者就是错误语句)。这两种方法都需要大量的失效和成功运行,而这些运行中可能有很多执行特征是相同的。

为了更好地定位程序错误,本文提出一种基于差异统计的故障定位方法,即首先对所有成功运行和失效运行分别聚类,消除路径冗余,然后计算每一类失效运行和每一类成功运行的差异以及每一类失效运行之间的差异(失效运行和成功运行的差异表明故障出现在那些位置的几率很大),通过对这些差异统计分析得到可疑语句,并对其进行可疑性排序,返回故障报告。实验对比分析表明,Wang 的算法花费大量时间用于计算差异,在数据中存在大量运行路径冗余的情况下,时间花费呈指数级增长,而在此情况下,本文算法时间花费在聚类上,故障定位方面的开销有所增加,定位效率和准确性方面都有很大提高。此外,本文算法不但能够诊断每一个失

① 国家自然科学基金(61175056)和中央高校基本科研业务费专项资金(2011QN033, 2009JC29)资助。

② 男,1969 年生,博士,教授;研究方向:软件工程,智能诊断;联系人,E-mail: rchen@dl.cn

(收稿日期:2011-07-01)

效运行的原因,而且通过对所有失效运行故障报告的统计分析,也可得到整个程序的故障报告。

1 基于运行差异统计的软件故障定位算法

本文软件故障定位算法的基本工作过程如下:

(1) 对路径聚类,消除运行路径冗余。

(2) 差异计算。

(3) 统计差异,生成故障报告。

算法输入为一个成功运行路径和失效运行路径的集合。经过第(1)步消除冗余后,成功和失效运行路径被分成若干类,第(2)步计算每一类失效运行与每一类成功运行的差异,第(3)步使用差异统计信息定位故障。

1.1 消除运行路径冗余

定义 1 事件(event)。一个事件 e_i 是给定程序中某条语句 e 的第 i 次执行。约定用语句行号代表与这个事件语句行,相关的程序语句记作 $\mathrm{stmt}(e_i)$。

定义 2 运行(run)。给定一个程序 P,一条运行路径就是一个事件序列 $<e_0, e_1, e_2, \cdots, e_{n-1}>$,其中 e_i 表示执行过程中的第 i 个事件。一个运行 π 的第 i 事件记作 e_i^{π}。

以图 1 中的小程序为例,运行不同的测试用例,得到成功和失效的运行汇总如表 1 所示。

```
1  package function;
2  public class ThreeNum{
3  /*功能:计算三个数最大值 */
4      public int max(int x, int y, int z)
5      {
6          int max;
7          if (x > y)
8          {
9              max = y;
10         }
11         else
12         {
13             max = y;
14         }
15         if (z > max)
16         {
17             max = z;
18         }
19         return max;
20     }
21 }
```

图 1 一个小程序例子

表 1 图 1 中程序对应的测试用例和运行路径

result	succ0	succ1	succ2	succ3	fail0	fail1	succ4	succ5	succ6	succ7	fail2	fail3
input	2,3,4	2,4,3	3,2,4	3,4,2	4,2,3	4,3,2	5,7,8	5,8,7	7,5,8	7,8,5	8,5,7	8,7,5
	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1	6_1
	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2	8_2
	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3	9_3
			11_4		11_4	11_4			11_4		11_4	11_4
	15_4	15_4		15_4			15_4	15_4		15_4		
	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5	17_5
	19_6		19_6		19_6		19_6		19_6		19_6	
	21_7	21_6	21_7	21_6	21_7	21_6	21_7	21_6	21_7	21_6	21_7	21_6

表 1 中第 2 行为输入的测试用例,每一个测试用例对应一条运行路径,如当输入 x,y,z 的值分别为 2,3,4 时,运行路径为 $<6_1, 8_2, 9_3, 15_4, 17_5, 19_6, 21_7>$,其中每一个元素是一个事件,例如事件“$8_2$”表示程序的第 8 行语句被执行一次,且是执行过程中的第 2 个事件。运行结果在第 1 行中给出,succ 表示运行成功,fail 表示运行失败。

定义 3 动态控制依赖(dynamic control dependence)。给定一个运行 π,事件 e_i^{π} 动态控制依赖于另一个事件 e_j^{π},如果 e_j^{π} 是 π 中 e_i^{π} 之前的最后一个满足 $\mathrm{stmt}(e_i^{\pi})$ 静态控制依赖于 $\mathrm{stmt}(e_j^{\pi})$ 的事件。用 $\mathrm{dep}(e_i^{\pi}, \pi)$ 表示事件 e_i^{π} 动态控制依赖的事件。

定义 4 对齐(alignment)。对 π 中任意一个事件 e 和 π' 中任意一个事件 e',$\mathrm{align}(e,e') = \mathrm{true}$,如果①$\mathrm{stmt}(e) = \mathrm{stmt}(e')$。②或者 e,e' 是 π,π' 中出现的第一个事件,或者满足 $\mathrm{align}(\mathrm{dep}(e,\pi), \mathrm{dep}(e',\pi')) = \mathrm{true}$。

程序中存在错误,执行众多测试用例就会得到大量成功运行和大量失效运行。可以看出,表中存

在大量相同的路径,如 fail0 和 fail2,succ0 和 succ4 等。既然这些运行的路径很多都是相同的,我们没有必要去比较每一个失效运行和每一个成功运行,而是希望在比较之前对运行路径进行聚类,以此消除冗余路径的影响。

这里介绍一种根据对齐对运行聚类以消除冗余的方法。首先我们给出相关的定义。

定义 5　一个运行 π'相对于另一个运行 π 的 0_1 向量。将 π'与 π 对齐,对 π 中每一个事件 e,如果 π'中存在与之对齐的事件 e',在相应位置标记“1”,反之,如果不存在与之对齐的事件,在相应位置标记“0”;并且对 π'中每一个事件,如果 π 中存在与之对齐的事件 e,在相应位置标记“1”,反之,如果不存在与之对齐的事件,在相应位置标记“0”。同时满足以上两个条件的唯一一个完全由“0”和“1”组成的向量。记为 Vector0_1(π,π')。

推论 1　给定一个运行 π 相对于另一个运行 π'的 0_1 向量,于是 Vector0_1(π,π') = Vector0_1(π',π)。

定义 6　一个运行集相对于某个运行 π 的 0_1 矩阵。对运行集中每一个运行,计算其相对于运行 π 的 0_1 向量,由这些向量组成的矩阵,向量维数不够的在末尾补“0”。

对一个运行集,我们首先任取其中的一个运行,计算运行集中所有运行相对此运行的 0_1 向量,进而构造一个 0_1 矩阵,这样就将复杂的运行路径转化为只是由“0”和“1”组成的序列,最后对此矩阵操作,运用聚类方法,实现将运行集中的运行按照运行路径是否相同分成若干类,消除了路径冗余。按照这个思路,关键的工作是构造 0_1 向量,图 2 给出了根据对齐构造 0_1 向量的算法。

```
算法：createVector0_1(run1,run2)
功能：根据对齐构造0_1向量
输入：run1,run2，两条运行
输出：vector，run2相对run1的一个0_1向量
1.  temp2=1,j=0;//临时存储run2中事件下标
2.  vector0_1={};//结果向量
3.  outer:
4.  for(int i=1; i<=run1中事件总数; i++){
5.      flag=alignExist(run1, i, run2);   //flag记录run2中是否存在与run1中的e_i相对齐的事件
6.      if(flag==true){ // 如果run2中存在与run1中的e_i相对齐的事件
7.          j=getAlignIndex(run1,i,run2); //取得 run2 中与 run1 中的 e_i 相对齐的事件的下标
8.          k=j-temp2-1;    //run2 中在此前面必有连续 k 个不能对齐
9.          for(n=0;n<k;n++){
10.             vector0_1.add(0);
11.         }
12.         vector0_1.add(1);  //加入一个“1”
13.     }
14.     else{ //如果 run2 中不存在与 run1 中的 e_i 相对齐的事件，加入一个“0”
15.         vector0_1.add(0);
16.     }
17.     temp2=j; // 暂时保存 run2 中能对齐的事件下标
18.     continue outer;
19. }
20. return vector0_1;
```

图 2　根据对齐构造 0_1 向量的算法

使用算法 createVector0_1(run1,run2)生成 0_1 向量,构造 0_1 矩阵,对矩阵按列聚类,可将表 1 中 4 个失效路径分为 2 类:FⅠ:fail0(4,2,3)、fail2(8,5,7),和 FⅡ:fail1(4,3,2)、fail3(8,7,5)。同理 8 个成功路径被分为 3 类:SⅠ: succ0(2,3,4)、succ4(5,7,8),SⅡ: succ1(2,4,3)、succ3(3,4,2)、succ5(5,8,7)、succ7(7,8,5)和 SⅢ: succ2(3,2,4)、succ6(7,5,8)。

1.2　差异计算

定义 7　差异度量(difference metric) 对一个程序的两个运行 π,π', 定义 diff(π,π') = $< e_{i1}^{\pi}, e_{i2}^{\pi}, \cdots, e_{ik}^{\pi} >$ 满足①diff(π,π')中的每一个事件都是出

现在 π 中的分支事件。②diff(π,π')中的事件出现顺序与 π 中相同,即对所有的 $1 \leqslant j < k, i_j < i_{j+1}$(事件 $e_{ij}\pi$ 出现在 e^{π}_{ij+1} 之前)。③对 diff(π,π')中的每一个事件 e,在 π' 中存在另一个分支事件满足 align(e,e') = true,并且 π 中的事件 e 与 π' 中的事件 e' 具有不同的输出。④所有满足①和②的事件都应该被包含在 diff(π,π')中。⑤特殊地,如果 π,π' 具有相同的控制流,规定 diff(π,π') = $< e^{\pi}_{0} >$。

根据定义 7,差异中的语句都是分支语句,且对两个运行,分支语句分别走向不同的分支。故对向量中,如果存在两个相邻值,前面一个为“1”,后面一个为“0”,则“1”位置表示的 run1 的事件加入差异中。使用图 3 中的算法,计算每一类失效运行和成功运行的差异以及每一类失效运行之间的差异,分别如表 2 和表 3 所示。

```
diffMetric(run1,run2)
功能：计算两条运行的差异
输入：run1，run2两条运行
输出：diff，差异的集合，元素为事件下标
1.  diff={};
2.  vector=createVector0_1(run1,run2);
3.  n=0;//记录在“1”之前“0”的总个数
4.  for(int i=0;i<vector.size()-1;i++){
5.      if(vector.get(i)==0){
6.          n++;
7.      }
8.      if(vector.get(i)==1&&vector.get(i-1)==0){//如果第 i 个事件能对齐，而它的下一个事件不能对齐
9.          diff.add(i+1-n+diff.size());//将“1”表示的运行 run1 中事件下标加入差异
10.     }
11.     continue;
12. }
```

图 3 计算两条运行差异算法

表 2 每一类失效运行和成功运行的差异表

	S Ⅰ	S Ⅱ	S Ⅲ
F Ⅰ	9_3	9_3 17_5	null
F Ⅱ	9_3 17_5	9_3	17_5

表 3 每一类失效运行之间的差异

	F Ⅰ	F Ⅱ
F Ⅰ	null	17_5
F Ⅱ	17_5	null

1.3 故障定位

在 Liblit[13] 的方法中,对每一个程序 P 中的谓词 P,它计算了两个条件概率:

Pr1 = Pr(P 失败|P 被观测到)

Pr2 = Pr(P 失败|P 被观测到值为真)

然后它定义 Increase(P) = Pr2 − Pr1 表示谓词 P 为真增加程序失败的可能性。由此我们推出以下结论:失效运行和成功运行的差异表明故障出现在那些位置的几率很大,而失效运行和失效运行的差异则表明故障在那些位置出现的几率很小,也就是谓词为真并不能增加程序失败的可能性。用 $P(e,\pi,S)$ 表示 π 中事件 e 在 π 与成功运行差异中出现频率,$P(e,\pi,F)$ 表示 π 中事件 e 在 π 与失效运行差异中出现频率:

$$P(e,\pi,S) = \frac{e\text{在}\pi\text{成功运行差异中出现的次数}}{\pi\text{与成功运行差异里事件总数}} \quad (1)$$

$$P(e,\pi,F) = \frac{e\text{在}\pi\text{与失效运行差异中出现的次数}}{\pi\text{与失效运行差异里事件总数}} \quad (2)$$

定义失效运行 π 中事件 e 是其失效原因的排序分数:

$$Score(e,\pi) = \begin{cases} \infty, & P(e,\pi,F) = 0 \\ P(e,\pi,S)/P(e,\pi,F), & P(e,\pi,F) \neq 0 \end{cases} \quad (3)$$

根据表 2、3 和式(1)、(2)、(3),我们得到如表 4所示的失效运行的可疑事件的排序分数。

表 4　失效运行的可疑事件的排序分数

失效运行 F Ⅰ的可疑事件排序分数		
	9_3	17_5
F Ⅰ	∞(2/3)	1/3
失效运行 F Ⅱ的可疑事件排序分数		
	9_3	17_5
F Ⅱ	∞(1/2)	1/2

以失效运行 F Ⅰ为例,因 $P(9_3,F\text{Ⅰ},S)=2/3$, $P(9_3,F\text{Ⅰ},F)=0$, 故 $Score(9_3,F\text{Ⅰ})=\infty$; 而 $P(17_5,F\text{Ⅰ},S)=1/3$, $P(17_5,F\text{Ⅰ},F)=1$, 所以 $Score(17_5,F\text{Ⅰ})=1/3$。注意表中"∞(2/3)"表示"$P(e,\pi,F)=0$, $P(e,\pi,S)=2/3$"。当"$P(e,\pi,F)=0$"时,值都为"∞",可疑性可以通过 $P(e,\pi,S)=2/3$ 的大小来比较。如果两个事件的排序分数相同,根据 Wang[9] 的比较思想,则认为后执行的那个事件可疑性更大,排在前面。因为事件是针对每个失效运行的,程序中的某一个句子可能被多次执行并多次出现在差异中,我们统计每个句子对失效运行原因的排序分数:

$$Score(stmt,\pi)=\sum Score(e,\pi)$$
$$(e\in\pi\text{ 且 }stmt(e)=stmt)\quad(4)$$

根据式(4),我们可以得到如表 5 所示的失效运行的可疑语句排序分数。表格的每一行,代表了每一条语句对某个失效运行原因的排序分数,假设有一条或几条语句的值明显比其它的大,说明这一条或这几条语句的就是引起的这个运行失效的原因。表格的每一列,代表了某一条语句对每个失效运行的排序分数,如果存在一些值明显比其它的大,说明这一条语句是不只一个失效运行的原因。比如语句 9 对 FI、F Ⅱ的影响都很大,表明 9 导致了这两类失效运行的失效。出现在表 5 中的语句都是可疑语句,都可能导致了程序的故障,但哪条语句对整个程序来说更重要呢? 如果程序中某一条语句导致了更多类失效运行的失效,说明这条语句是程序故障的可疑性更大。我们利用以下公式计算每条可疑语句的重要性:

表 5　失效运行的可疑语句排序分数

	9	17
F Ⅰ	∞(2/3)	1/3
F Ⅱ	∞(1/2)	1/2

$$Score_p(stmt)=\sum_{\pi\in F}Score(stmt,\pi)$$
(F 为所有失效运行类的集合)　(5)

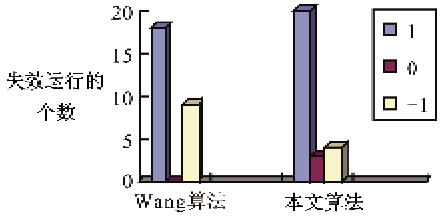
2　实验结果

如表 6 所示,我们选择 4 个带有分支的程序,通过人工向程序中注入不同的错误(有的程序注入一个错误,有的程序注入两个错误),得到 43 个含有错误的程序及 79 个失效运行。

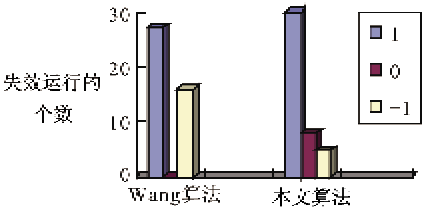
表 6　实验数据描述

程序	最大分支嵌套层次	故障版本			
		存在一个错误	失效运行种类数	存在两个错误	失效运行种类数
P1	0	5	14	3	11
P2	1	4	4	4	7
P3	2	8	9	8	16
P4	3	5	5	7	13

分别用 Wang 的算法和本文算法进行故障定位,将得到的结果进行比较。一条语句如果在故障报告中出现,表示它可能是程序实际的故障位置,即受到怀疑。各语句的排序分数代表了语句受怀疑的程度,排序分数越大表示越受怀疑,即是程序实际故障位置的可能性越大。对本文算法,对每一个失效运行,如果得到的故障报告中受怀疑程度最大的语句恰是程序实际的故障位置,记为"1",如果程序实际的故障位置没有受到怀疑,记为"-1",如果故障报告中最受怀疑的语句不是程序实际的故障位置,但实际的故障位置受到了怀疑,记为"0"。对 Wang 的算法,对每一个失效运行,只要得到的故障报告中存在程序实际的故障位置,就记为"1",不存在记为"-1"。通过实验得到的统计结果如图 4 所示。纵轴为失效运行的个数。其中由两个错误程序导致的全为失效运行,共 8 类,虽然不能求失效与成功运行的差异,但我们给出了失效与失效运行的差异,能够说明程序在那些位置出现故障的几率较小。



(a) 程序中只有一个错误情况



(b) 程序中存在两个错误情况

图 4 两种算法故障报告结果比较

Wang 的算法是将返回最小差异作为故障报告，忽略了故障的位置可能不在此最小差异中，而是出现在其它差异中的情况，而我们利用统计的方法考虑所有的情况，并对故障报告中语句进行了可疑性排序，除非故障不出现在所有差异中，否则必定出现在我们的故障报告中。对于存在两个错误的程序，运用以下标准评估：

$$score = 1 - \frac{|DS_*|}{|PDG|} \tag{6}$$

$$score(\pi_f) = \frac{\sum_{\pi_s \in Closest(\pi_f)} score(\pi_f, \pi_s)}{|Closest(\pi_f)|} \tag{7}$$

$$pgm\_score(P) = \frac{\sum_{\pi_f \in Failing(P)} score(\pi_f)}{|Failing(P)|} \tag{8}$$

公式中 $pgm\_score(P)$ 代表用户为找到程序中的错误可以忽略的代码百分比。由于我们首先对所有失效运行和所有成功运行进行了分类，故对故障定位评估标准中的每个失效运行和每个成功运行，都用每类失效运行和每类成功运行取代。本文算法计算 $score(\pi_f)$ 时，分别计算故障报告中可疑性最大的语句的 *score* 和可疑性排在第二位语句的 *score* 作为 $score(\pi_f)$，并最终计算 $pgm\_score(P)$，得到的结果见表 7。

从表 7 中看出，针对程序中存在两个故障的情况，由本文算法得到的故障报告中可疑性最大的语句比用 Wang 的算法得到的故障报告质量高。而可疑性排在第二位的语句得到的分数也较高，说明故障报告中可疑性排在第二位的语句也很可能正是程序的故障位置，也即是程序中很可能存在两个故障。

表 7 程序中存在两个错误时两算法比较

Score	Wang	本文算法第一位	本文算法第二位
0.8 ~ 1.0	13.3	33.3	46.7
0.7 ~ 0.79	60.0	26.7	6.7
0.6 ~ 0.69	6.7	6.7	20.0
0.5 ~ 0.59	13.3	26.7	6.7
0 ~ 0.49	6.7	6.7	20.0

Wang 的算法在数据中存在大量的运行路径冗余的情况下，仍需计算每一个失效运行和每一个成功运行的差异，时间主要花在计算差异上。从图 5 中可以看到，随着冗余数据的增加，所用时间增长迅速。我们的算法首先对所有运行路径聚类，消除冗余，然后计算每一类失效运行和每一类成功运行的差异。随着冗余数据的增加，时间的增加主要在聚类算法上，故增加很小。Wang 的算法是对每一个失效运行计算其与成功运行的最小差异得出故障报告，我们的算法不但得出针对每一个失效运行的原因，而且通过对所有失效运行故障报告的统计分析，得出针对整个程序的故障报告，并对此故障报告中的语句进行了重要性排序。

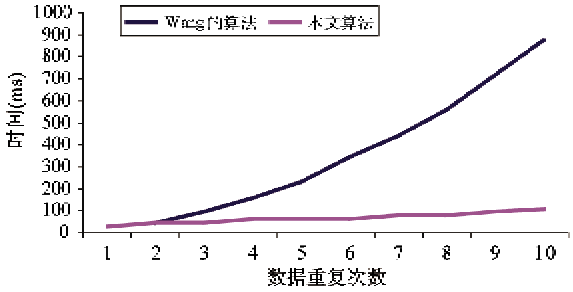


图 5 两算法运行时间比较

3 结 论

本文提出了一种通过聚类消除运行路径冗余的方法，对程序故障定位时不是只返回与失效运行最相近的成功运行的差异作为故障报告，而是通过对所有差异统计得出故障报告，并对故障报告中语句进行了重要性排序，在定位效率和准确性方面都得到很大提高。下一步的工作我们将考虑语句间的数据依赖，以及动态依赖关系，研究抽象路径上的故障定位，以及本文算法对程序失效性预测研究方面的作用等。

参考文献

[1] Wotawa F. Debugging VHDL designs using model-based

reasoning. *Artificial Intelligence in Engineering*, 2000,14(4): 331-351

[2] Groce A. Error explanation with distance metrics. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2988, Springer, 2004. 108-122

[3] Demsky B, Rinard M. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 2003, 38(11): 78-95

[4] James J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, New York, USA, 2002. 467-477

[5] Ball T, Naik M, Rajamani S K. From symptom to cause: Localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIG PLAN-SIGACT Symposium on Principles of Programming Languages, New York, USA, 2003. 97-105

[6] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, Orlando, USA, 2002. 467-477

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

[8] Zeller A, Hildebrandt R. Simplifying and isolatingfailure-inducing input. *IEEE Transactions on Software Engineering*, 2002, 28(2):183-200

[9] Guo L, Roychoudhury A, Wang T. Accurately choosing execution runs for software fault localization. In: Proceedings of the 15th International Conference on Compiler, Berlin, Germany, 2006. 80-95

[10] Wang T, Roychoudhury A. Automated path generation for software fault localization. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, USA, 2005. 347-351

[11] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, USA, 2005. 273-282

[12] Liblit B, Aiken A, Zheng A X, et al. Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, New York, USA, 2003. 141-154

[13] Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, USA, 2005. 15-26

Software fault localization based on statistical difference between runs

Chen Rong, Hong Lina

(School of Information Science and Technology, Dalian Maritime University, Dalian 116026)

Abstract

The authors studied in detail a typical approach to software fault location that pinpoints buggy statements by comparing a large amount of failing program runs with some successful runs, and analyzed the reason of its lower efficiency and accuracy of pinpointing that required execution data inevitably contain a large number of redundant execution paths, and based on this, presented an improved fault localization method by statistical analysis of the difference between reduced program runs. The method can be described below: first, a clustering method is used to eliminate the redundancy in execution paths, next, the statistics of difference between the reduced failing runs and successful runs is calculated, and then, the bug report by ranking the buggy statements is generated. The experimental results show that the proposed algorithm is greatly improved in terms of the efficiency and accuracy compared with the above mentioned Wang's algorithm.

Key words: fault localization, path redundancy, clustering, statistical method