计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
50(5): 1030-1043, 2013

基于条件执行切片谱的多错误定位

文万志[1] 李必信[1] 孙小兵[2] 齐珊珊[1]
[1](东南大学计算机科学与工程学院 南京 211189)
[2](扬州大学信息工程学院 江苏扬州 225127)
(wenwanzhi@126.com)

A Technique of Multiple Fault Localization Based on Conditioned Execution Slicing Spectrum

Wen Wanzhi[1], Li Bixin[1], Sun Xiaobing[2], and Qi Shanshan[1]
[1](School of Computer Science and Engineering, Southeast University, Nanjing 211189)
[2](College of Information Engineering, Yangzhou University, Yangzhou, Jiangsu 225127)

Abstract Program spectrum-based software fault localization (PS-SFL) has become one of the hottest research directions due to its high efficiency of localizing faults. It usually localizes program faults by computing the suspiciousness of program statements according to the test coverage information. However, the efficiency of this technology will decrease with the increase of the number of faults in the program. One reason is that the suspiciousness of program statement is computed by counting the number of failed tests and passed tests, but the failed tests that covered the program statement may be caused by different faults. This paper proposes a multiple fault localization (CESS-MFL) technique, which is based on conditioned execution slicing spectrum to alleviate the above problem and improve the efficiency of localizing multiple faults. The CESS-MFL technique firstly constructs a spectrum matrix of fault-related conditioned execution slice according to the predicates of input variables, then computes the suspiciousness of elements (statements or blocks of statements) in fault-related conditioned execution slice and generates a suspiciousness report. The experiment shows that the CESS-MFL technique has higher efficiency than the popular program spectrum-based Tarantula technique, program slicing-based Intersection technique and Union technique in the multiple-fault program, and can be implemented in effective space and time complexity.

Key words multi-fault location; program slicing spectrum; program spectrum; conditioned execution slice; software debugging

摘 要 基于程序谱的错误定位技术由于其较高的定位效率已成为当前软件调试领域研究热点之一.这种技术通常根据测试覆盖信息计算程序语句发生错误的可疑度来进行错误定位.然而,这种技术会随着程序中错误数目的增多效率不断下降.鉴于此,提出了一种基于条件执行切片谱的多错误定位技术(conditioned execution slicing spectrum-based multiple fault localization, CESS-MFL),以提高多错误定位的效率.CESS-MFL技术首先根据输入变量的谓词条件构建错误相关条件执行切片的谱矩阵,然后依次计算错

收稿日期:2012-09-28;修回日期:2012-12-24
基金项目:国家自然科学基金项目(60973149);高等学校博士学科点专项科研基金项目(20100092110022);中国科学院计算机科学国家重点实验室开放基金项目(SYSKF1110);国家自然科学基金青年科学基金项目(61202006)
通信作者:李必信(bx.li@seu.edu.cn)

误相关条件执行切片中的元素(语句或语句块)的可疑度,并生成可疑度报告.实验验证了 CESS-MFL 技术比当前流行的基于程序谱的 Tarantula 技术、基于程序切片的 Intersection 技术、Union 技术有更高的多错误定位效率,并且可在有效的时间和空间复杂度内完成.

关键词 多错误定位;程序切片谱;程序谱;条件执行切片;软件调试

中图法分类号 TP311

在软件开发和维护过程中,程序员需要不断地调试程序,以发现程序中可能的错误并予以纠正.错误定位是软件调试活动中最困难、最耗时的任务之一.特别是当前软件越来越复杂多样,迫切需要自动化或半自动化技术以提高错误定位的效率.自动化或半自动化的错误定位技术也正成为当前的研究热点之一.

经典的 Delta 调试技术[1-2]通过不断的迭代运行程序,交换成功运行的内存状态和失效运行的内存状态来缩小错误查找的范围以提高错误定位的效率,不足之处是迭代运行和内存交换的代价太大.类似的基于值替换的错误定位技术[3-4]通过改变语句中变量值,以使失效运行的结果输出正确,从而确定错误的位置.这种技术虽然仅考虑在给定时间的特定语句值的改变,但仍然需要相当多的计算时间.传统的程序切片技术[5]通过分析程序的数据依赖和控制依赖,提取与错误相关的切片来缩小搜索域,主要有静态切片技术[6]和动态切片技术[7].然而,传统切片构造的时间和空间复杂度都很大,而且切片相对保守,错误定位的效率不高.统计分类可疑谓词进行错误定位的技术[8-9]通过顺序检查可疑谓词来进行错误定位,这种方法依赖于谓词样本.基于模型诊断的方法[10]通过将程序语句视为组件从而使用模型诊断的方法来进行错误的推理分析和定位,但模型诊断的时间复杂度一般过高.基于贝叶斯统计推理的方法[11]通过构建贝叶斯网络推理计算程序模块包含的概率进行错误定位.

基于程序谱的错误定位技术根据测试覆盖信息计算程序语句发生错误的可疑度来进行错误定位[12-18].通常情况下,对程序谱信息的收集比较简单,容易实现,而且便于存储,定位效率非常高.研究表明,基于程序谱的错误定位方法的平均查错精度不足整个程序的 20%[13-14].基于程序谱的错误定位技术的不足之处在于,随着单个程序中错误数的增加错误定位的效率会不断降低.其中一个重要的原因是基于程序谱的错误定位技术会对通过程序语句的失效测试进行统计计算,而在多错误定位中随着程序中错误数的增多,不同错误引起的失效测试会对同一条语句的统计结果有叠加影响,从而影响最终的错误定位效率.

在前期的工作[16,18]中,我们提出了一种基于程序切片谱的错误定位技术.这种技术考虑到传统的基于程序谱的错误定位技术信息缺乏依赖分析,因此结合了程序切片技术以进一步提高错误定位的效率.但该技术没有考虑随着程序中错误数的增多而导致错误定位效率降低的问题.

针对传统的基于程序谱的错误定位技术的优点和不足,本文提出了一种基于条件执行切片谱的多错误定位技术(conditioned execution slicing spectrum-based multiple fault localization, CESS-MFL),以降低多错误定位中失效测试数目的叠加影响,提高多错误定位的效率,其主要贡献如下:

1) 提出了使用条件执行切片谱进行多错误定位的方法;

2) 提出了一种基于条件执行切片谱的可疑度度量方法;

3) 实验验证了 CESS-MFL 技术比当前流行的基于程序谱的 Tarantula 技术、基于程序切片 Intersection 技术和 Union 技术在分别包含 2~5 个错误的多错误程序中错误定位效率更高,并且分析了 CESS-MFL 技术的时间和空间复杂度的有效性.

1 基于程序谱的软件错误定位技术

通常,基于程序谱的错误定位[12-18]过程包括 3 步:

1) 通过运行测试用例收集程序中的语句覆盖信息,构建程序谱矩阵;

2) 根据程序谱计算出覆盖语句的出错可疑度;

3) 根据可疑度大小顺序逐句排查来进行错误定位.

1.1 基本概念

定义 1. 失效测试与成功测试. 已知程序 P 及其测试集合 $T=\{T_1,\cdots,T_m\}$,其中,T_i 的预期结果为 δ_i,实际输出结果为 θ_i. 如果 $\delta_i=\theta_i$,则测试 T_i 为

成功测试,否则,如果 $\delta_i \neq \theta_i$,则测试 T_i 为失效测试.

定义 2. 程序谱. 已知程序有 n 条语句,测试执行集合 $T=T_F \cup T_P$,其中,$T_F=\{T_1,\cdots,T_s\}$表示失效测试集合,$T_P=\{T_{s+1},\cdots,T_m\}$表示成功测试集合,则程序谱是一个二维矩阵 $\boldsymbol{M}_{n,m}$,矩阵元素 $b_{i,j}$为

$$b_{i,j}=\begin{cases}\text{True}, & T_j \text{ 的测试执行路径覆盖语句 } S_i;\\ \text{False}, & \text{否则};\end{cases} \tag{1}$$

其中,$1\leqslant i\leqslant n$,$1\leqslant j\leqslant m$.

一个形象化的基于程序谱的错误定位模型如表 1 所示,其中行 $S_1,\cdots,S_n$ 表示程序中对应的语句,列 $T_1,\cdots,T_m$ 表示 m 个测试执行,包括 s 个失效测试和 m−s 个成功测试,$b_{i,j}(1\leqslant i\leqslant n,1\leqslant j\leqslant m)$是程序谱矩阵元素,其值为 *True* 或 *False*,值为 *True* 表示测试 T_j 经过语句 S_i,值为 *False* 表示测试 T_j 不经过语句 S_i,最后一列表示相应行语句包含错误的可疑度(suspiciousness),可疑度通常是基于这样的假设:经过某条程序语句 S_i 的失效测试数目越多,而成功测试数目越少,该语句发生错误的可能性越大,即在表 1 中,$\sum_{j=1}^{s} b_{i,j}$ 越大,$\sum_{j=s+1}^{m} b_{i,j}$ 越小,suspiciousness(S_i)越大.

Table 1 Program Spectrum-Based Software Fault Localization

表 1 基于程序谱的错误定位

Program	Test Cases						Suspiciousness
	Failed			Passed			
	T_1	⋯	T_s	T_{s+1}	⋯	T_m	
S_1	$b_{1,1}$	⋯	$b_{1,s}$	$b_{1,s+1}$	⋯	$b_{1,m}$	p_1
S_2	$b_{2,1}$	⋯	$b_{2,s}$	$b_{2,s+1}$	⋯	$b_{2,m}$	p_2
⋮	⋮		⋮	⋮		⋮	⋮
S_n	$b_{n,1}$	⋯	$b_{n,s}$	$b_{n,s+1}$	⋯	$b_{n,m}$	p_n

基于程序谱的度量方法主要有 *Tarantula*[12-13],*Jaccard*[14,19],*Ochiai*[14,19],它们的可疑度计算方法如下:

$$suspiciousness_{\text{T}}(S_i)=\frac{failed(S_i)/TotalFailed}{failed(S_i)/TotalFailed+passed(S_i)/TotalPassed}; \tag{2}$$

$$suspiciousness_{\text{J}}(S_i)=\frac{failed(S_i)}{TotalFailed+passed(S_i)}; \tag{3}$$

$$suspiciousness_{\text{O}}(S_i)=\frac{failed(S_i)}{\sqrt{failed(S_i)(failed(S_i)+passed(S_i))}}; \tag{4}$$

其中,failed(S_i)表示经过语句 S_i 的失效测试的数目,passed(S_i)表示经过语句 S_i 的成功测试的数目,TotalFailed 表示失效测试总数目,TotalPassed 表示成功测试总数目. 显然,failed(S_i)=0 时 S_i 可疑度为 0,式(2)~(4)给出了当 failed(S_i)≠0 时可疑度的计算方法. 不同的研究人员对于可疑度(*suspiciousness*)有着不同的定义方法,总的目标希望根据语句 S_i 的可疑度大小顺序,尽可能快地定位出错误语句位置.

1.2 基于程序谱的软件错误定位实例

本节通过一个简单的程序实例来介绍基于程序谱的单错误定位过程和多错误定位效率降低问题. 图 1 给出了计算 4 个数中最大数的一种程序实例. 其中,语句 S_6 和 S_{11} 中包含了错误. 已知程序的输入(a,b,c,d)分别为(6,6,7,5),(6,7,9,3),(8,6,9,7),(5,4,6,8),(7,6,6,4),(4,5,6,7),(5,7,6,3),(7,6,8,8).

```
S1   int max,a,b,c,d;
S2   Read(a,b,c,d);
S3   if(a>b)
S4   {max=a;
S5   if(max<c)
S6       max=a; /* fault, should be S'6; */
/* S'6 max=c; */
S7   if(max<d)
S8       max=d;
     }
     else
S9   {max=b;
S10  if(max<c)
S11  max=b; /* fault, should be S'11; */
/* S'11 max=c; */
S12      if(max<d)
S13      max=c;
         }
S14  print(max);
```

Fig. 1 Example program including two faults.

图 1 包含两个错误的实例程序

1.2.1 单错误定位

表 2 给出了程序仅包括错误语句 S_{11} 的程序谱以及基于上述 3 种方法计算出来的语句可疑度. 表 2 中,√ 表示 *True*,× 表示 *False*. 在仅包括错误语句 S_{11} 时,输入(6,6,7,5),(6,7,9,3),实际输出的最大值分别为 6,7,这与预期的结果 7,9 不等,所以输入为(6,6,7,5),(6,7,9,3)的测试为失效的测试. 其

他的输入测试结果与预期结果相符,为成功的测试.根据可疑度从大到小依次确定错误位置,第 1 次就成功定位到错误语句 S_{11}.基于程序谱的定位技术在单错误程序中有较好的定位效率.

Table 2 Example of Program Spectrum-Based Single Fault Localization

表 2 基于程序谱的单错误定位实例

Program	Test Case								Suspiciousness		
	Failed		Passed								
	6,6,7,5	6,7,9,3	8,6,9,7	5,4,6,8	7,6,6,4	4,5,6,7	5,7,6,3	7,6,8,8	TAR	JAC	OCH
S_1	√	√	√	√	√	√	√	√	0.50	0.25	0.63
S_2	√	√	√	√	√	√	√	√	0.50	0.25	0.63
S_3	√	√	√	√	√	√	√	√	0.50	0.25	0.63
S_4	×	×	√	√	√	×	×	√	0.00	0.00	0.00
S_5	×	×	√	√	√	×	×	√	0.00	0.00	0.00
S_6'	×	×	√	√	×	×	×	√	0.00	0.00	0.00
S_7	×	×	√	√	√	×	×	√	0.00	0.00	0.00
S_8	×	×	×	√	×	×	×	×	0.00	0.00	0.00
S_9	√	√	×	×	×	×	√	√	0.75	0.50	0.82
S_{10}	√	√	×	×	×	×	√	√	0.75	0.50	0.82
S_{11}	√	√	×	×	×	×	√	×	0.86	0.67	0.89
S_{12}	√	√	×	×	×	×	√	√	0.75	0.50	0.82
S_{13}	×	×	×	×	×	×	√	×	0.00	0.00	0.00
S_{14}	√	√	√	√	√	√	√	√	0.50	0.25	0.63

1.2.2 多错误定位

当程序中包含错误语句 S_6 和 S_{11} 时,输入(6,6,7,5),(6,7,9,3),(8,6,9,7)的实际输出结果为 6,7,8,这和预期的结果 7,9,9 不符,所以为失效的测试,其他的输入测试结果与预期结果相符,为成功的测试,建立的程序谱以及可疑度值如表 3 所示:

Table 3 Example of Program Spectrum-Based Multi-Fault Localization

表 3 基于程序谱的多错误定位实例

Program	Test Case								Suspiciousness		
	Failed		Passed								
	6,6,7,5	6,7,9,3	8,6,9,7	5,4,6,8	7,6,6,4	4,5,6,7	5,7,6,3	7,6,8,8	TAR	JAC	OCH
S_1	√	√	√	√	√	√	√	√	0.50	0.38	0.80
S_2	√	√	√	√	√	√	√	√	0.50	0.38	0.80
S_3	√	√	√	√	√	√	√	√	0.50	0.38	0.80
S_4	×	×	√	√	√	×	×	√	0.36	0.17	0.50
S_5	×	×	√	√	√	×	×	√	0.36	0.17	0.50
S_6	×	×	√	√	×	×	×	√	0.45	0.20	0.58
S_7	×	×	√	√	√	×	×	√	0.36	0.17	0.50
S_8	×	×	×	√	×	×	×	√	0.00	0.00	0.00
S_9	√	√	×	×	×	√	√	×	0.63	0.40	0.82
S_{10}	√	√	×	×	×	√	√	×	0.63	0.40	0.82
S_{11}	√	√	×	×	×	√	×	×	0.77	0.50	0.89
S_{12}	√	√	×	×	×	√	√	×	0.63	0.40	0.82
S_{13}	×	×	×	×	×	√	×	×	0.00	0.00	0.00
S_{14}	√	√	√	√	√	√	√	√	0.50	0.38	0.80

按照可疑度大小顺序,我们能很快定位到错误语句 S_{11},而语句 S_6 的可疑度相对较小,错误定位的效率就相对较低. 其中,S_1,S_2,S_3,S_{14} 语句可疑度值相对 S_6 较大,主要是由于由 S_{11} 和 S_6 引起的失效测试都经过 S_1,S_2,S_3,S_{14},从而导致不同失效测试数目叠加统计 S_9,S_{10},S_{11},S_{12} 语句可疑度偏大主要是由于由 S_{11} 失效测试通过了这些语句.

鉴于此,本文提出了一种基于条件执行切片谱的多错误定位方案.

2 基于条件执行切片谱的多错误定位技术及其实现

本节将介绍基于条件执行切片谱的多错误定位技术原理及其实现算法.

2.1 条件执行切片

条件切片由 Canfora 等人[20]提出,它是根据程序的输入状态提取的一个程序语句子集.

定义 3. 条件切片准则[20]. 令 V_{in} 是程序 P 的输入变量的子集,$F(V_{in})$ 是关于 V_{in} 的一阶逻辑公式. 程序 P 的条件切片准则是一个三元组 $C=(F(V_{in}),p,V)$,其中,p 是 P 中的一条语句,V 是 P 中变量的一个子集.

定义 3 的切片准则中,p 通常是程序 P 中的一条输出语句,V 通常是输出语句 p 中的变量集合,V_{in} 通常是切片条件所使用的输入变量,在确定输入变量时,$F(V_{in})$ 通常用谓词条件公式表示. 例如,在图 1 的实例程序中,p 通常为输出语句 S_{14},V 通常是输出语句 S_{14} 中的变量集合{max},输入变量集合为{a,b,c,d},在切片条件为 a>b 时,子集 V_{in} 为{a,b},谓词条件公式为(a>b). 根据这个准则,在切片条件为 a>b 时得出的条件切片语句集合为{S_1,S_2,S_3,S_4,S_5,S_6,S_7,S_8,S_{14}}. 具体地,要获得条件切片,通常的做法是根据输入条件对程序进行简化,即通过符号执行器来产生简化的条件化程序,然后在条件化程序上基于程序依赖图产生切片. 然而,依赖图生成和切片提取的时间和空间代价都是很昂贵的. 而根据测试历史,提取执行切片只需要常量时间,所以本文结合条件切片原理和执行切片提取的高效率,提出条件执行切片来进行错误定位,具体的定义如下.

定义 4. 执行切片[21]. 给定一个测试 T,执行切片是被 T 执行的代码块 E(T).

定义 5. 满足集. 令 V_{in} 是程序 P 的输入变量的集合,$F(V_{in})$ 是关于 V_{in} 的一阶逻辑公式,$\{T_1,\cdots,T_n\}$ 是程序 P 的 n 个测试. $F(V_{in})$ 的满足集 $S(F(V_{in}))$ 是 $E(T_i)$ 构成的集合,其中,测试 T_i 的输入满足了公式 $F(V_{in})$,即

$$S_{(F(V_{in}))}=\{E(T_i)\mid 1\leqslant i\leqslant n,\ \text{且 } E(T_i)\text{ 的输入满足公式 } F_{(V_{in})}\}. \tag{5}$$

定义 6. 条件执行切片准则. 令 V_{in} 是程序 P 的输入变量的集合,$F(V_{in})$ 是关于 V_{in} 的一阶逻辑公式,$\{T_1,\cdots,T_n\}$ 是程序 P 的 n 个测试. 程序 P 条件执行切片准则是一个二元组$\langle F(V_{in}),\{T_1,\cdots,T_n\}\rangle$.

定义 7. 条件执行切片. 程序 P 关于条件执行切片准则$\langle F(V_{in}),\{T_1,\cdots,T_n\}\rangle$的条件执行切片 ConditionedSlice($\langle F(V_{in}),\{T_1,\cdots,T_n\}\rangle$)为

$$ConditionedSlice(\langle F(V_{in}),\{T_1,\cdots,T_n\}\rangle)=\bigcup_{1\leqslant i\leqslant n,E(T_i)\in S(F(V_{in}))}E(T_i). \tag{6}$$

2.2 条件执行切片谱的构造

本文的 CESS-MFL 技术主要分为 4 步:

1) 计算错误相关条件执行切片,缩小错误搜索域;

2) 构造条件执行切片谱矩阵;

3) 根据条件执行切片谱,计算错误相关条件执行切片内每个元素(语句或语句块)的可疑度;

4) 根据可疑度大小依次定位程序中的错误.

本节主要介绍基于错误相关条件执行切片构造条件执行切片谱的过程.

定义 8. 错误相关条件执行切片. 已知程序 P 及其测试执行集合 $T=T_F\cup T_P$,其中 $T_F=\{T_1,\cdots,T_s\}$ 表示失效测试集合,$T_P=\{T_{s+1},\cdots,T_m\}$ 表示成功测试集合. 令 $F(V_{in})$ 是关于输入变量集合 V_{in} 的一阶逻辑公式,则错误相关的条件执行切片 Related Slice 为 ConditionedSlice($\langle F(V_{in}),\{T_1,\cdots,T_s\}\rangle$),即

$$RelatedSlice=\bigcup_{1\leqslant i\leqslant s,E(T_i)\in S(F(V_{in}))}E(T_i). \tag{7}$$

错误相关的条件执行切片根据已有的失效测试历史去除了错误无关部分,从而缩小了错误定位的搜索域. 算法 1 给出了错误相关条件执行切片算法.

算法 1. 错误相关条件执行切片构造算法.

Function:GetRelatedSlice(P,T,$F(V_{in})$)

输入:P——程序 P;

T——测试用例集合 $\{T_1,T_2,\cdots,T_m\}$;

$F(V_{in})$——输入变量 V_{in} 的一阶逻辑公式函数;

输出:RelatedSlice——错误相关条件执行切片.

```
begin
  RelatedSlice=∅;
  for each test case Tj in T do
    if Tj is failed
      if E(Tj)∈S(F(Vin))
        RelatedSlice =RelatedSlice∪E(Tj);
      end if
    end if
  end for
  return RelatedSlice;
end begin
```

定义 9. 条件执行切片谱. 已知程序 P 的输入变量集合 V_{in}、关于 V_{in} 的一阶逻辑公式 $F(V_{in})$ 及测试执行集合 $T=T_F\cup T_P$. 其中 $T_F=\{T_1,\cdots,T_s\}$ 表示失效测试集合,$T_P=\{T_{s+1},\cdots,T_m\}$ 表示成功测试集合. 令 $T'_F=\{T_i\mid 1\leqslant i\leqslant s, E(T_i)\in S(F(V_{in}))\}$, $T'=T'_F\cup T_P=\{T'_1,\cdots,T'_l\}$, 基于切片准则 $\langle F(V_{in}),T\rangle$ 的错误相关条件执行切片 RelatedSlice 中元素个数为 n(即 n=|RelatedSlice|),则条件执行切片谱 CESS 表示一个二维矩阵 $\boldsymbol{M}_{n,l}$,矩阵元素 $f_{i,j}$ $(1\leqslant i\leqslant n, 1\leqslant j\leqslant l)$ 为

$$f_{i,j}=\begin{cases}k, \text{测试 } T_j \text{ 经过 } RelatedSlice \text{ 中元素 } e_i \text{ 的次数};\\ 0, \text{测试 } T_j \text{ 未经过 } RelatedSlice \text{ 中元素 } e_i.\end{cases} \tag{8}$$

条件执行切片谱记录了所有的成功测试和满足 $F(V_{in})$ 条件的失效测试中元素的执行频率,它分离了与条件 $F(V_{in})$ 无关的失效测试. 算法 2 给出了基于条件 $F(V_{in})$ 的条件执行切片谱的构造算法.

算法 2. 条件执行切片谱的构造算法.

Function:GetCESS(P,T,$F(V_{in})$)

输入:P——程序 P,描述了程序元素集合 $\{e_1,e_2,\cdots,e_n\}$;

T——测试用例集合 $\{T_1,T_2,\cdots,T_m\}$;

$F(V_{in})$——输入变量 V_{in} 的一阶逻辑公式函数;

输出:CESS——程序谱矩阵,矩阵中每个元素 $CESS[e_i,T'_j]$ 表示错误相关的条件执行切片中程序元素 e_i 被 T'_j 执行的频率 f_{ij}.

```
begin
  RelatedSlice=GetRelatedSlice(P,T,
    F(Vin));
  T'={Ti | Ti is passed or E(Ti)∈
    S(F(Vin))};
  for each test case Tj' in T' do
    for each ei in RelatedSlice do
      CESS[ei,Tj']=the frequency that Tj'
        passed ei;
    end for
  end for
  return CESS;
end begin
```

2.3 基于程序频谱的可疑度计算

比较流行的可疑度计算模型 Tarantula[12-13],Jaccard[14,19],Ochiai[14,19] 是基于程序击谱进行软件的错误定位的,即只考虑程序语句是否被测试覆盖,不考虑语句被执行的频率. 在我们前期的研究工作中,将程序频度和贡献度引入到可疑度的计算中[16,18]. 由于每个测试执行的代码规模大小不一,每条语句被每个测试执行的频率也不一样,所以,语句对测试结果的贡献程度也不一样. 例如,如果某次测试执行经过了 1 条语句,那么这条语句对这个测试结果的贡献度可以达到 100%;而如果这次执行经过了 2 条语句,且每条语句仅被执行一次,那么我们认为这两条语句对测试结果有相同的贡献度,各为 50%. 一般地,基于程序谱的可疑度[12-13] 计算方法定义如下:

定义 10. 可疑度.

$$suspiciousness(S_i)=\frac{failed(S_i)\%}{failed(S_i)\%+passed(S_i)\%}, \tag{9}$$

其中,failed(S_i)% 表示经过语句 S_i 的失效测试数占失效测试总数之比,passed(S_i)% 表示经过语句 S_i 的成功测试数占成功测试总数之比.

结合上述条件执行切片谱定义,本文对 failed(e_i)% 和 passed(e_i)% 的定义如下:

$$failed(e_i)\%=\frac{\sum_{j=1}^{m}(p_{T_j}\times C_{i,j})}{\sum_{j=1}^{m}p_{T_j}},$$

$$passed(e_i)\%=\frac{\sum_{j=1}^{m}[(1-p_{T_j})\times C_{i,j}]}{\sum_{j=1}^{m}(1-p_{T_j})}, \tag{10}$$

其中,

$$C_{i,j}=\frac{f_{i,j}}{\sum_{k=1}^{n}f_{k,j}},$$

$$p_{T_j}=\begin{cases}1, T_j \text{ is failed and } T_j\in S(F(V_{in}));\\ 0, T_j \text{ is passed.}\end{cases} \tag{11}$$

在式(10)～(11)中的 $C_{i,j}$ 表示程序元素 e_i 对测试 T_j 的贡献度,其中,程序切片谱矩阵元素 $f_{i,j}$ 如定义 9 所述,它表示测试 T_j 中元素 e_i 的执行频度; $p_{T_j}=1$ 表示 T_j 是失效测试且满足输入集 V_{in} 的 $F(V_{in})$ 的谓词条件; $p_{T_j}=0$ 表示 T_j 是成功测试; failed(e_i)%和 passed(e_i)%的分母 $\sum_{j=1}^{m} p_{T_j}$ 和 $\sum_{j=1}^{m}(1-p_{T_j})$ 分别表示满足 $F(V_{in})$ 失效的测试数目和成功的测试数目,而分子对经过 e_i 的失效测试的数目和成功的测试数目分别进行加权. 根据上述条件执行切片谱的构造,错误相关条件执行切片谱内元素可疑度的计算算法如算法 3 所示.

算法 3. 可疑度算法.

Function: GetSuspiciousness(P, T, $F(V_{in})$)

输入: P——程序 P,描述了程序元素集合 $\{e_1, e_2, \cdots, e_n\}$;

T——测试用例集合 $\{T_1, T_2, \cdots, T_m\}$;

$F(V_{in})$——输入变量 V_{in} 的一阶逻辑公式函数;

声明: T′——失效测试和满足公式 $F(V_{in})$ 的测试用例集合 $\{T'_1, \cdots, T'_l\}$;

RelatedSlice——错误相关的条件执行切片集合 $\{e_1, e_2, \ldots e_n\}$;

CESS——条件执行切片谱矩阵,矩阵元素 ***CESS***$[e_i, T'_j]$ 记录了 RelatedSlice 中程序元素 e_i 被 T′中 T'_j 执行的频率;

PN——T′中成功测试的数目;

FN——T′中失效测试的数目;

PC_i——e_i 对所有成功测试贡献的累加和;

FC_i——e_i 对所有失效测试贡献的累加和;

PT_i——记录测试 T_i 成功与否;

N_i——T_i 覆盖 RelatedSlice 中所有元素频度的累加和;

输出: Suspiciousness(RelatedSlice)——RelatedSlice 中每个元素 e_i 的可疑度列表,即{suspiciousness(e_1), suspiciousness(e_2), …}.

```
begin
  T′={T_i | T_i is passed or E(T_i)∈S(F(V_in))};
  RelatedSlice=GetRelatedSlice(P,T,F(V_in));
  CESS=GetCESS(P,T,F(V_in));
  PN=0;
  FN=0;
  for each test case in T'_i in T' do
    N_i=0;
    if T'_i is passed
      PT_i=0;
      PN=PN+1;
    else
      PT_i=1;
      FN=FN+1;
    end if
    for each e_j in RelatedSlice
      N_i=N_i+CESS[e_j, T'_i];
    end for
  end for
  for each e_i in RelatedSlice do
    PC_i=0;
    FC_i=0;
    for each test case T'_j in T' do
      PC_i=PC_i+(1-PT_j)×CESS[e_i, T'_j]/N_j;
      FC_i=FC_i+PT_j× CESS[e_i, T'_j]/N_j;
    end for
      suspiciousness(e_i)=FC_i/FN/(FC_i/FN+PC_i/PN);
  end for
  return suspiciousness(RelatedSlice);
end begin
```

2.4 基于条件执行切片谱的多错误定位

为了定位出程序中所有的错误,通常根据输入给出不同的条件执行切片,然后对所有错误相关条件执行切片中的元素进行可疑度度量. 已知程序 P 的输入集为 V_{in},一阶逻辑公式 $F_1, \cdots, F_r$ 给出了关于输入变量 V_{in} 的 r 个输入条件,一个基于条件执行切片谱的多错误定位抽象模型如表 4 中所示. 其中,失效的测试根据 r 个输入条件进行分离,即每个 F_i 列下的测试 T_j 满足 $E(T_j)\in S(F_i(V_{in}))$. 具体地,表 4 中行 $e_1, \cdots, e_n$ 表示所有错误相关切片中的元素,列 $T_1, \cdots, T_m$ 表示 m 个测试执行,包括 s 个失效的运行和 m−s 个成功的运行,其中,s 个失效的运行分别根据 r 个输入条件进行分离,即每个 F_i 列下的测试 T_j 满足 $E(T_j)\in S(F_i(V_{in}))$. $f_{i,j}(1\leqslant i\leqslant n, 1\leqslant j\leqslant m)$ 是条件执行切片谱矩阵元素,其值为 k 或 0,表示测试 T_j 执行元素 e_i 的频度. 最后一列表示相应元素包含错误的可疑度,可疑度初值为 0,本文基于前述算法对 r 类失效测试和成功测试分别计算可疑度,并保存了可疑度的最大值. 具体的生成元素可疑度的错误定位报告算法如算法 4 所示:

Table 4 Conditioned Execution Slicing Spectrum-Based Multi-Fault Localization

表 4 基于条件执行切片谱的多错误定位

Related Slice	Test Cases										Suspiciousness
	Failed							Passed			
	F_1					F_r					
	T_1	…	T_{s1}	…	T_{s-sr+1}	…	T_s	T_{s+1}	…	T_m	
e_1	$f_{1,1}$	…	$f_{1,s1}$	…	$f_{1,s-sr+1}$	…	$f_{1,s}$	$f_{1,s+1}$	…	$f_{1,m}$	p_1
e_2	$f_{2,1}$	…	$f_{2,s1}$	…	$f_{2,s-sr+1}$	…	$f_{2,s}$	$f_{2,s+1}$	…	$f_{2,m}$	p_2
⋮	⋮		⋮		⋮		⋮	⋮		⋮	⋮
e_n	$f_{n,1}$	…	$f_{n,s1}$	…	$f_{n,s-sr+1}$	…	$f_{n,s}$	$f_{n,s+1}$	…	$f_{n,m}$	p_n

算法 4. 错误定位报告生成算法.

Function:GetFaultReport(P,T,F)

输入:P——**源程序**;

T——**测试用例集**{T_1,T_2,…,T_m};

F——**一阶逻辑公式集**{F_1,F_2,…,F_r},**描述了关于输入变量** V_{in} **的** r **个输入条件**;

输出: FaultReport(P)——**一个列表**{FaultReport(e_1), FaultReport(e_2), …},**其中**, FaultReport(e_i)**记录了程序** P **中元素** e_i **的可疑度**.

```
begin
  for each e_i in P do
    FaultReport(e_i)=0;
  end for
  for each F_i in {F_1,F_2,…,F_r} do
    RelatedSlice=GetRelatedSlice(P,T,
      F_i(V_in));
    suspiciousness(RelatedSlice)=
      GetSuspiciousness(P,T,F_i(V_in));
    for each e_j in RelatedSlice do
      if suspiciousness(e_j)>FaultReport(e_j)
        FaultReport(e_j)=suspiciousness(e_j);
      end if
    end for
  end for
  return FaultReport(P);
end begin
```

2.5 实例分析

表 5 **给出了** 2.3 **节多错误情况下的程序频谱及根据** *CESS-MFL* **技术计算出的可疑度. 其中,输入变量**{a,b,c,d}**分成了** a≤b **和** a>b **两种情况. 在**

Table 5 Example of Conditioned Execution Slicing Spectrum-Based Multi-Fault Localization

表 5 基于条件执行切片谱的多错误定位实例

Program	Test Cases								Suspiciousness
	Failed			Passed					
	$a \leqslant b$		$a>b$						
	6,6,7,5	6,7,9,3	8,6,9,7	5,4,6,8	7,6,6,4	4,5,6,7	5,7,6,3	7,6,8,8	
S_1	1	1	1	1	1	1	1	1	0.50
S_2	1	1	1	1	1	1	1	1	0.50
S_3	1	1	1	1	1	1	1	1	0.50
S_4	0	0	1	1	1	0	0	1	0.63
S_5	0	0	1	1	1	0	0	1	0.63
S_6	0	0	1	1	0	0	0	1	0.71
S_7	0	0	1	1	1	0	0	1	0.63
S_9	1	1	0	0	0	1	1	0	0.71
S_{10}	1	1	0	0	0	1	1	0	0.71
S_{11}	1	1	0	0	0	1	0	0	0.83
S_{12}	1	1	0	0	0	1	1	0	0.71
S_{14}	1	1	1	1	1	1	1	1	0.50

$a>b$ 时,错误相关切片集合为$\{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_{14}\}$,根据上述可疑度计算方法,错误 S_6 的可疑度 $suspiciousness(S_6)=(1/8/1)/((1/8/1)+(1/8+1/8)/5)=0.71$,错误相关切片集合内其他语句可疑度由同样的方法计算得到. 同理,在谓词 $a\leqslant b$ 时,错误语句 S_{11} 可疑度 $suspiciousness(S_{11})=((1/8+1/8)/2)/(((1/8+1/8)/2)+(1/8)/5)=0.83$. 根据可疑度大小顺序,$S_{11}$, S_6 都能高效地定位到,比传统的方法定位效率有了明显的提高.

3 实验分析

基于程序谱的错误定位技术给出了程序元素的可疑度报告来度量程序中每个元素(语句或语句块)包含错误概率大小.

本文的实验主要研究以下问题:

1) CESS-MFL 技术与相关的错误定位技术的定位效率比较;

2) CESS-MFL 技术生成的可疑度报告的时间复杂度和空间复杂度.

3.1 实验对象

本文的实验在 Eclipse 平台上实现,我们选取了 3 个面向对象程序作为实验对象,如表 6 所示. 表 6 中的第 1 列是对象名称;第 2 列是多错误程序中植入的错误数;第 3 列是多错误程序的版本数,其中,2 错误程序版本共有 49 个,3 错误程序版本共有 26 个,4 错误程序版本共有 18 个,5 错误程序版本共有 12 个,总共植入了 308 个错误,错误植入的准则是平均覆盖每个输入分类; 第 4 列是程序的代码行数;第 5 列是本实验根据输入特征给出的谓词条件分类;最后一列是所有程序的简单描述,其中,Tetris 是大家熟悉的俄罗斯方块游戏程序①;SimpleJavaApp 是代码覆盖工具 CodeCover 的例子程序②,它主要用来显示、编辑书目列表,并可以将列表以 XML 文件形式保存并加载;JHSA 是我们小组所开发的一个 Java 层次切片工具,该工具用于构造 Java 程序的层次依赖图,计算 Java 程序的层次切片,以及在此基础之上的一些软件维护应用,代码结构图如图 2 所示:

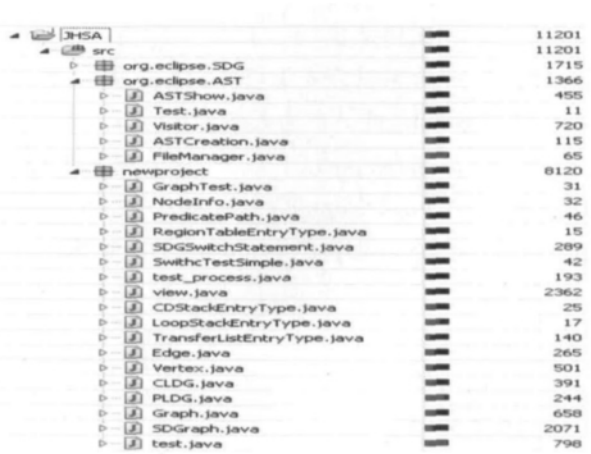

Fig. 2 Experiment subject JHSA.

图 2 实验对象 JHSA

Table 6 Experiment Subject

表 6 实验对象

Subject	Faults	Versions	Lines	Conditions	Descriptions
Tetris	2	12	2 397	MoveLeft ∈ InputSet, RotateClockwise ∈ InputSet, stop and resume∈InputSet,…	Tetris Game
	3	7			
	4	4			
	5	2			
SimpleJavaApp	2	10	1 784	Newbook ∈ InputSet, Open ∈ InputSet, delete∈InputSet,…	A displaying and editing book listing application
	3	7			
	4	5			
	5	3			
JHSA	2	27	11 201	IFProgram, SwithProgram, WhileProgram, BlockProgram, ForProgram	A JAVA hierarchical slicing tool
	3	12			
	4	9			
	5	7			

① 源代码地址: http://www.percederberg.net/games/tetris/tetris-1.2-src.zip.

② 源代码地址: http://codecover.org/documentation/tutorials/SimpleJavaApp.zip.

3.2 实验度量方法

基于程序谱的错误定位技术最终要根据可疑度大小顺序逐条检查语句来确定错误语句,所以评估基于程序谱的错误定位技术有效性可通过根据可疑度大小顺序确定实际错误所需要访问的语句数占程序总规模之比(P_{search})来衡量,即

$$P_{\text{search}}=\frac{s}{|P|}\times 100\%, \tag{12}$$

式(12)中,s 表示根据可疑度报告确定实际错误所需要访问的语句条数;分母表示程序 P 中所包含的语句总条数. P_{search} 越小,定位错误所需要查找的语句数越少,定位效率越高.

3.3 实验过程

本实验通过如下几步进行实验数据的收集与分析:

1) 利用工具 codecover① 收集各个测试覆盖程序元素的信息;

2) 利用条件执行切片谱工具 CESS 统计生成条件执行切片谱(工具的主要算法见 2.2~2.4 节),并计算错误相关条件执行切片内每个元素的可疑度;

3) 根据可疑度报告依序确定并记录每个错误需要查找的语句数,计算 P_{search};

4) 分析统计了当前流行的相关的错误定位技术的错误定位效率,并与 CESS-MFL 技术进行了比较.

3.4 实验结果与分析

3.4.1 CESS-MFL 技术与相关的错误定位技术的定位效率比较

本节分析统计了 CESS-MFL 技术的效率,并与当前流行的基于程序谱的 Tarantula 技术、基于执行切片的集合运算的并集定位技术(Union)和交集定位技术(Intersection)[22]进行了比较. 当前流行的基于程序谱度量的技术主要有 Ochiai, Jaccard, Tarantula,这 3 种度量技术的效率相近,所以本文只选取了 Tarantula 技术进行了比较. 给定一个失效的测试运行 f 和成功的测试运行集合 P,Union 技术通过计算 $f-\bigcup_{p\in P}p$ 来进行错误定位,而 Intersection 技术是通过其补集运算 $\bigcap_{p\in P}p-f$ 来进行错误定位.

实验中,我们采用 3.2 节定义的 P_{search},即成功定位到错误查找的语句数占整个程序规模的百分比来衡量错误定位的效率. 表 7 将 P_{search} 分成 12 段,然后分别统计包含 2,3,4,5 个错误的程序采用 4 种技术在不同的 P_{search} 段成功搜索到的错误数. 由于基于程序谱的定位技术在 P_{search} 的前 10%段定位到的错误相对比较密集,我们将前 10%段分成 3 部分进行了统计,分别是 0~1%,1%~5%,5%~10%. 表 7 中,如果 P_{search} 越小能成功定位到的错误越多,那么错误定位技术的效果越好. 具体地,这 4 种技术通过如下的错误定位顺序对数据进行统计:CESS-MFL 以及 Tarantula 技术根据错误报告中可疑度大小顺序依次定位错误,Union 和 Intersection 技术先顺序查找集合内的程序代码,如果错误不在集合内,根据失效测试的语句执行顺序进行错误地查找和定位. 根据这种方法我们得到了表 7 的统计数据. 总体上,采用 CESS-MFL 和 Tarantula 技术进行错误定位技术的效率要高于 Union 技术和 Intersection 技术. 这主要是因为 Union 和 Intersection 技术仅给出了可能包含错误的集合,而基于程序谱的度量技术对每个程序元素都进行了度量.

Table 7 Distribution of Successfully Locating Faults

表 7 成功定位的错误 *Bugs* 分布表

P_{search} /%	CESS-MFL				Tarantula				Union				Inter			
	2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5
0~1	14	11	9	6	12	9	8	5	2	1	2	2	0	0	0	0
1~5	16	13	8	7	14	11	7	7	5	4	2	1	3	4	2	2
5~10	14	8	10	7	13	10	7	4	9	6	5	5	7	5	4	4
10~20	11	10	9	6	9	5	6	5	8	9	8	7	8	8	7	7
20~30	13	11	10	8	15	12	11	6	10	5	6	6	11	7	6	5
30~40	9	7	5	7	9	7	5	6	9	10	8	5	11	9	9	6
40~50	8	5	7	4	10	8	8	8	14	9	10	9	15	10	10	11
50~60	5	6	7	5	6	7	8	6	12	10	7	8	12	11	10	8
60~70	4	3	2	4	4	4	5	4	9	9	7	5	11	9	7	5
70~80	1	2	3	3	2	2	3	3	8	7	8	7	8	7	8	7
80~90	2	1	0	2	3	2	2	4	5	4	4	3	5	4	4	3
90~100	1	1	2	1	1	1	2	2	7	4	5	2	7	4	5	2

为了更直观地比较,我们根据表 7 得出了图 3 关于 2~5 个错误的多错误程序的错误定位效率比较. 图 3 中,横坐标是 P_{search},纵坐标 P_{Bugs} 是 $Bugs$ 与 $TotalBugs$ 的比值,其中 $Bugs$ 即表 7 中的在 P_{search} 段找到的错误数,$TotalBugs$ 是 $Bugs$ 所在列错误数的和,表示相应错误数的程序中植入的错误总数. 通过这种方式我们以统一的形式对不同个数的多错误程序进行了统计比较. 图 3 中,横坐标值相同时,

① http://www.codecover.org/

纵坐标值越大说明效率越高. 从图 3 可以看出, Intersection 技术的效率最低,在 0～1%段,我们没有搜索到一个错误,这主要是因为交集谱在很多情况下是空集,成功测试的交集一般要么自身是空集,要么是程序的必经结点,也包含在失效测试中. 根据交集思想及前面介绍的本文语句错误定位次序,交集谱进行的错误定位近似从程序开头依次查找语句. Union 技术的效率次之. Union 技术在错误语句只出现在失效测试中而没有出现在成功测试中的时候,错误定位效率很高,否则依次搜索失效测试中的语句定位出错误.

总体上,Union 技术的效率比 Intersection 有所提高. 我们对 Intersection, Union 技术的统计结果优于一般文献中的统计结果,如文献[23]中的统计结果,这主要是因为我们对错误语句在集合外的统计方法不同. 文献[23]对错误的统计方法是如果错误包含在集合中就统计集合大小,否则错误就需要搜索整个程序,而本文精确统计了不在集合中需要搜索的语句数. 基于程序谱度量的 CESS-MFL 技术和 Tarantula 技术明显高于切片集合计算的 Intersection 和 Union 技术. 在多错误程序中, Tarantula 技术由于没有考虑不同错误引起的失效测试在语句上的叠加,从而导致精度下降,效率低于本文的 CESS-MFL 技术.

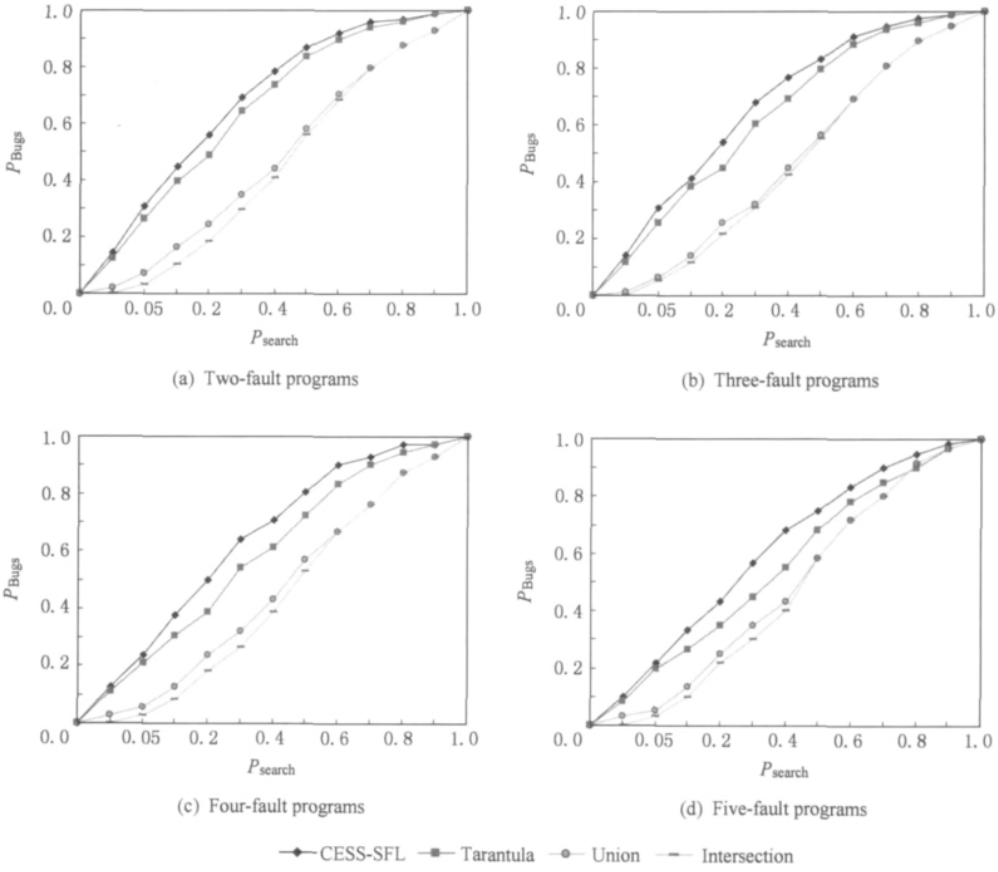

Fig. 3 Comparation of fault localization technique in two to five-fault programs.

图 3 2～5 个错误的多错误程序的错误定位技术比较

3.4.2 时间复杂度和空间复杂度

本节我们分析了 CESS-MFL 技术的时间复杂度和空间复杂度.

实验过程中,我们记录了 CESS-MFL 技术在 3 个实验对象上平均所用时间,如表 8 所示. 实验在 2.60 GHz 的 Intel Pentium® Dual-core CPU 并拥有 2 GB 内存的 PC 上实现. 上述时间通过 JAVA 程序的系统函数 *System.currentTimeMillis*() 和 *System.nanoTime*() 记录. 表 8 中第 1 列是实验对象;第 2 列记录的是读取条件执行切片谱文件,并统计测试数目等计算可疑度相关数据的时间;第 3 列是可疑度计算时间;第 4 列是可疑度排序时间. 从记录的数据可以看出,CESS-MFL 仅需要极少的计算时间和排序时间,在代码量大一些的 JHSA 中也仅用了 1.52 ms 的计算时间和 374 ms 的排序时间,CESS-MFL 的主要时间用在外存条件执行切片谱文件的读取上,即

便如此,读取文件的时间也是有限的.

Table 8 Average Time That CESS-MFL Costs

表 8 CESS-MFL 技术所用的平均时间 s

Subject	I/O	Computation	Sort
Tetris	4.7	0.000 39	0.016
SimpleJavaApp	4.5	0.000 33	0.013
JHSA	18.2	0.001 52	0.374

理论上,根据我们 CESS-MFL 实现的原理,读取条件执行切片谱矩阵的时间复杂度为 $O(|T|\times|P|)$,其中 $|T|$ 为测试规模,$|P|$ 为程序规模. 因为涉及到外存文件的读取,实际所用时间比正常的内存操作时间要长. 可疑度计算的时间复杂度为 $O(r\times|P|)$,其中,r 为根据输入集的条件谓词个数. 排序时间为 $O(|P|\times\log|P|)$.

CESS-MFL 技术的空间复杂度主要在条件执行切片谱的存储. 我们使用 Codecover 工具提取了程序覆盖信息,并最终将条件执行切片谱存储在外存的一个 XML 文件上,文件结构如图 4 所示. 覆盖信息以文本压缩的形式存储,一般的 PC 系统已足以胜任存储条件执行切片谱文件. 而在内存空间使用上,主要的空间复杂度在于可疑度数据的存储,根据上述 CESS-MFL 技术的原理,其空间复杂度约为 $O(|P|)$.

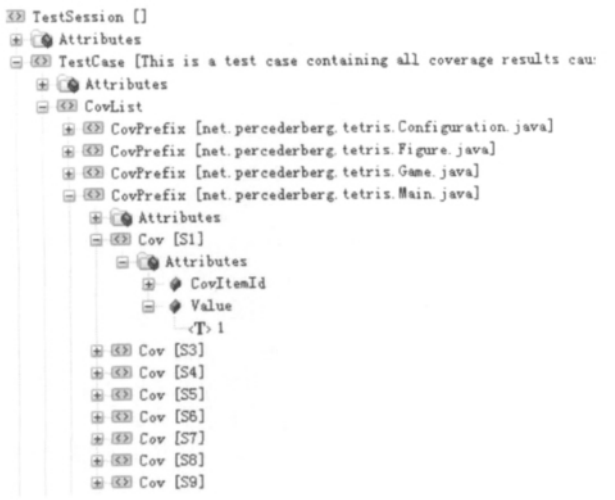

Fig. 4 File tree of conditioned execution slicing spectrum.

图 4 条件执行切片谱文件结构

3.5 讨 论

通过上述实验的结果与分析,CESS-MFL 能够在有效的时间和空间下完成多错误程序可疑度报告的生成,并比 Tarantula 技术、Union 技术和 Intersection 技术的多错误定位效率有了进一步的提高. 实验过程中主要存在的不足如下:

1) 虽然 CESS-MFL 技术比传统的基于程序谱度量技术在多错误定位中效率更高,但随着每个程序中错误数的增多,CESS-MFL 技术依然和传统的程序谱技术一样效率不断下降,如图 3 所示. 这主要是因为在多错误定位中随着程序中错误数的增多,在同一谓词条件下的失效测试由多个不同的错误引起的可能性也会增加从而导致精度下降.

2) 当程序错误由于程序中遇到语句丢失时,通过我们的可疑度度量算法,最终可以将错误定位到丢失语句位置正常执行序列中的上一条语句,从而可以根据这条语句来找到相应的错误. 如果程序错误是由于程序结构不合理引起的,我们错误定位算法可以将错误定位在不合理结构的更高层次上,这两种情况需要配合更多的人工分析.

3) 在并发程序中,错误的输出可能是由于并发程序之间的不当的值传递引起的. 这种情况下,本文的基于条件执行切片谱的技术和其他基于程序谱技术一样,定位效率会降低.

4 相关工作

基于条件执行切片谱的多错误定位技术首先构造条件执行切片谱,然后计算错误相关的条件执行切片内元素的可疑度,并根据可疑度大小最终定位出错误位置. 这主要涉及两方面的错误定位技术:基于程序切片的错误定位技术和基于程序谱度量的错误定位技术.

主流的基于程序切片的技术采用集合运算方式进行错误定位. 削片[23](dice)技术通过计算包含错误的切片与正确的切片之差来初步确定错误的位置. 执行切片[21]的并集错误定位思想是计算包含某条错误的执行切片与所有正确的执行切片的并集之差来确定错误的位置,执行切片的交集错误定位思想是计算所有正确切片的交集与某条包含错误的执行切片之差来进行错误定位. 文献[22]通过计算错误的执行切片和与其距离最近的正确的执行切片之差来进行错误定位. 另外,除了上述主流的基于切片集合运算的错误定位方法外,基于切片的错误定位技术还包括传统的静态切片技术[6]、动态切片技术[7]以及关键切片技术[24]、条件切片技术[20]等等;更进一步地,Burger 等人结合了动态切片、事件切片以及 Delta 调试技术来降低错误定位的平均搜索域[25]. CESS-MFL 与这些技术是不同的,CESS-

MFL 技术首先提取了错误相关的条件执行切片,然后构建了条件执行切片谱,计算了错误相关的条件执行切片内元素的可疑度.

另一种相关技术是基于程序谱的软件错误定位技术.程序谱可以分为程序击谱和程序频谱两种,它们的不同在于程序击谱只考虑测试是否覆盖了语句,而程序频谱统计测试覆盖语句的次数.目前,基于程序谱的错误定位技术主要是基于程序击谱的,这种技术通过统计失效执行和成功执行是否通过某条语句来计算这条语句产生错误的概率,即可疑度.常见的基于程序击谱的错误定位方法主要有 Tarantula[12-13] 技术、Jaccard[14,19] 技术和 Ochiai[14,19] 技术,其度量方法在前面已经有所介绍. Artzi 等人还将这 3 种方法应用到动态 Web 应用程序的错误定位中[17].另外, Wong 等人[26]、谭德贵[27]等人将正确测试的权重因子和错误测试的权重因子引入到可疑度计算中,来提高错误定位的精度.本文的可疑度度量模型与传统的可疑度度量模型是不同的,首先我们采用了程序频谱的方式度量语句的可疑度,其次根据程序输入提取了错误相关的条件执行切片然后对切片内元素进行度量,减少了错误无关语句的度量,并可提高传统谱的多错误定位效率.

5 结论和下一步工作

本文提出了一种基于条件执行切片谱的多错误定位技术——CESS-MFL 技术.该技术首先根据输入条件谓词构建条件执行切片谱,然后分别计算条件执行切片谱中每个元素的可疑度,从而实现多错误定位.本文通过实验验证了 CESS-MFL 技术的多错误定位的效率,并分析了该技术的时间和空间复杂度的有效性.

CESS-MFL 与其他基于程序谱的错误定位技术一样,在并发程序中错误定位的效率很低.未来,我们将着重解决并发程序的基于程序谱的错误定位问题.

参 考 文 献

[1] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input [J]. IEEE Trans on Software Engineering, 2002, 28(2): 183-200

[2] Cleve H, Zeller A. Locating causes of program failures [C] //Proc of Int Conf on Software Engineering. New York: ACM, 2005: 342-351

[3] Jeffrey D, Gupta N, Gupta R. Fault localization using value replacement [C] //Proc of the Int Symp on Software Testing and Analysis. New York: ACM, 2008: 167-177

[4] Jeffrey D, Gupta N, Gupta R. Effective and efficient localization of multiple faults using value replacement [C] // Proc of the IEEE Int Conf on Software Maintenance. Los Alamitos, CA: IEEE Computer Society, 2009: 221-230

[5] Weiser M. Program slicing [J]. IEEE Trans on Software Engineering, 1984, 10(4): 352-357

[6] Weiser M. Programmers use slices when debugging [J]. Communications of the ACM, 1982, 25(7): 446-452

[7] Agrawal H, DeMillo R A, Spafford E H. Debugging with dynamic slicing and backtracking [J]. Software-Practice and Experience, 1993, 23(6): 589-616

[8] Liblit B, Naik M, Zheng A, et al. Scalable statistical bug isolation [C] //Proc of the 2005 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2005: 15-26

[9] Liu Chao, Fei Long, Yan Xifeng, et al. Statistical debugging: A hypothesis testing-based approach [J]. IEEE Trans on Software Engineering, 2006, 32(10): 831-848

[10] Abreu R, Zoeteweij P, van Gemund A J C. Spectrum-based multiple fault localization [C] //Proc of the IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2009: 88-99

[11] Liu Yongpo, Wu Ji, Jin Maozhong, et al. Experimentation study of BBN-based fault localization [J]. Journal of Computer Research and Development, 2010, 47(4): 707-715 (in Chinese)
(柳永坡, 吴际, 金茂忠, 等. 基于贝叶斯统计推理的故障定位实验研究[J]. 计算机研究与发展, 2010, 47(4): 707-715)

[12] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization [C] //Proc of the 24th Int Conf on Software Engineering. New York: ACM, 2002: 467-477

[13] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault localization technique [C] //Proc of the 20th IEEE/ACM Int Confon Automated Software Engineering. New York: ACM, 2005: 273-282

[14] Abreu R, Zoeteweij P, van Gemund A J C. On the accuracy of spectrum-based fault localization [C] //Proc of Testing: Academic and Industrial Conference-Practice and Research Techniques. Piscataway, NJ: IEEE, 2007: 89-98

[15] Abreu R, Gonzlez A. Exploiting count spectra for bayesian fault localization [C] //Proc of the 6th Int Conf on Predictive Models in Software Engineering. New York: ACM, 2010: 1-10

[16] Wen Wanzhi, Li Bixin, Sun Xiaobing, et al. Program slicing spectrum-based software fault localization [C] //Proc of the 23rd Int Conf on Software Engineering and Knowledge Engineering. Skokie, IL: Knowledge Systems Institute, Graduate School, 2011: 213-218

[17] Artzi S, Dolby J, Tip F, et al. Fault localization for dynamic Web applications [J]. IEEE Trans on Software Engineering, 2012, 38(2): 314-335

[18] Wen Wanzhi. Software fault localization based on program slicing spectrum [C] //Proc of the 34th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2012: 1511-1514

[19] Abreu R, Zoeteweij P, van Gemund A J C. An evaluation of similarity coefficients for software fault localization [C] // Proc of the 12th IEEE Pacific Rim Int Symp on Dependable Computing. Piscataway, NJ: IEEE, 2006: 39-46

[20] Canfora G, Cimitile A, Lucia A D. Conditioned program slicing [J]. Information and Software Technology, 1998, 40 (11/12): 595-607

[21] Wong W E, Qi Y. Effective program debugging based on execution slices and inter-block data dependency [J]. Journal of Systems and Software, 2006, 79(7): 891-903

[22] Renieris M, Reiss S P. Fault localization with nearest neighbor queries [C] //Proc of the 18th IEEE Int Conf on Automated Software Engineering. Los Alamitos, CA: IEEE Computer Society, 2003: 30-39

[23] Chen T Y, Cheung Y Y. Dynamic program dicing [C] //Proc of the Conf on Software Maintenance. Los Alamitos, CA: IEEE Computer Society, 1993: 378-385

[24] DeMillo R A, Pan H, Spafford E H. Critical slicing for software fault localization [C] //Proc of the 1996 ACM SIGSOFT Int Symp on Software testing and analysis. New York: ACM, 1996: 121-134

[25] Burger M, Zeller A. Minimizing reproduction of software failures [C] //Proc of Int Symp on Software Testing and Analysis. New York: ACM, 2011: 221-231

[26] Wong W E, Qi Y, Zhao L, et al. Effective fault localization using code coverage [C] //Proceedings of International Computer Software and Applications Conference. Piscataway, NJ: IEEE, 2007: 449-456

[27] Tan Degui, Chen Lin, Wang Ziyuan, et al. Spectra-based fault localization by increasing marginal weight [J]. Chinese Journal of Computers, 2010, 33 (12): 2335-2342 (in Chinese)
(**谭德贵, 陈林, 王子元, 等. 通过增大边际权重提高基于频谱的错误定位效率**[J]. **计算机学报**, 2010, 33(12): 2335-2342)

Wen Wanzhi, born in 1982. PhD candidate in Southeast University. His main research interests include software testing and software fault localization.

Li Bixin, born in 1969. Professor and PhD supervisor in Southeast University. Senior member of China Computer Federation. His main research interests include software modeling, analyzing, testing, verifying and software maintenance techniques.

Sun Xiaobing, born in 1985. Received his PhD from Southeast University. Member of China Computer Federation. His main research interests include software analyzing, understanding and software change impact analysis(xbsun@yzu.edu.cn).

Qi Shanshan, born in 1988. Master candidate in Southeast University. Her main research interests include the trust of Web services and the evolution of composite services(shell0214@seu.edu.cn).