# 一种随机TBFL方法

王蓁蓁[1,2,3] 徐宝文[1,2] 周毓明[1,2] 陈 林[1,2]
(南京大学软件新技术国家重点实验室 南京 210093)[1] (南京大学计算机科学与技术系 南京 210093)[2]
(金陵科技学院信息技术学院 南京 211169)[3]

**摘 要** 许多学者研究了运用测试集对程序错误语句定位的问题,并提出了许多行之有效的方法,这些方法统称为TBFL(testing based fault localization)方法。后来人们发现,测试集里如果出现冗余,则这些冗余测试用例会伤害这些定位方法的功效。为了解决这个问题,Hao等人提出了SAFL(similarity aware fault localization)方法。实际上完全避免冗余是不可能的,因此从另一个角度构造了一个新的TBFL方法,称为随机TBFL方法。该方法的基本思想是:测试前对程序的语句错误概率进行先验分布,并把测试集看成随机变量,用测试用例反映的程序语句有关信息对程序语句的概率作一些调整,调整后的概率称为后验校正概率,最后根据这个后验概率对错误语句进行定位。将传统的TBFL方法如Dicing方法、TARANTULA方法、SAFL方法纳入随机信息分析并通过几个实例进行分析和比较,结果表明,随机TBFL方法不仅能够正确定位错误语句,而且冗余对该方法的功效伤害不大。

**关键词** 错误定位,测试为基础的错误定位,随机错误定位方法
**中图法分类号** TP311 **文献标识码** A

## New Random Testing-based Fault Localization Approach

WANG Zhen-zhen[1,2,3] XU Bao-wen[1,2] ZHOU Yu-ming[1,2] CHEN Lin[1,2]
(State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210093, China)[1]
(Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)[2]
(School of Information Technology, Jinling Institute of Technology, Nanjing 211169, China)[3]

**Abstract** Fixing faults in software are an essential task in software development, and many approaches have been presented to automate fault localization. Among them, testing-based approaches are most promising. These approaches use the information of test cases to localize the faults, and they are called collectively as TBFL approach. But these TBFL approaches have ignored the similarity of the test cases, which may harm the effectiveness of these approaches. In fact it is impossible to completely avoid redundancy. Therefore this paper presented a new TBFL approach named random TBFL approach from a new view. The basic idea is that: the program is viewed as a random variable, and before testing, a prior distribution about the error probability of statements of the program is given, then some adjustments to the error probability of statements are made based on the execution information of the test suite, and the re-adjusted probability is called posterior probability, finally this posterior probability is used to localize the faults. This paper integrated the traditional TBFL approaches into the random framework, and compared and analyzed them on several instances. The analysis demonstrates that the random TBFL approach can correctly locate the faults, and redundancy has little influence on the effectiveness of the random TBFL approach.

**Keywords** Fault localization, Testing based fault localization, Random testing based fault localization

## 1 引言

寻找软件错误位置是一件困难和费时的工作,但它关系到软件质量的改进,所以许多年来,软件界从不同的角度开发了许多方法,以有效地解决错误定位问题。其中运用从软件测试里获得的信息自动确定错误位置是很重要的一种技术手段,这些方法可以统称为TBFL(testing-based fault localization)方法[1-12]。TBFL方法基于许多测试用例从软件测试的结果中挑选出一些值得怀疑的语句,即它们可能导致软件错误,并按怀疑程度进行排序,从而帮助开发人员缩小搜索范

到稿日期:2012-03-21 返修日期:2012-06-23 本文受国家自然科学基金重大研究计划重点项目(90818027),国家高技术研究发展计划(863技术),专题项目(2008AA01Z143,2009AA01Z147),国家自然科学基金面上项目(60773104,60803007),金陵科技学院科研基金(jit-b-201207)资助。

王蓁蓁(1975—),女,博士后,主要研究领域为软件测试、人工智能,E-mail:wangzhenzhen@seu.edu.cn;徐宝文(1961—),男,博士,教授,博士生导师,主要研究领域为程序设计语言、软件工程等;周毓明(1974—),男,博士,教授,主要研究领域为软件度量;陈 林(1979—),男,博士,主要研究领域为程序分析。

围，尽可能快地找到错误根源，提高开发质量。

文献[3]讨论了6种涉及动态定位的TBFL方法：Dicing方法(Agrawl等1995)、TARANTULA方法(Jones等2002；Jones和Harrold 2005)、Nearest Neighbor Queries方法(Renieris和Reiss 2003)、CT(Cleve和Zeller 2005)、SoBER(Liu等2005)和Liblit05(liblit等2005)，发现它们都忽略了实施相似性的测试用例可能产生的问题。所谓测试用例之间的相似性，是指它们都覆盖若干个相同语句，其极端情况便是冗余，即测试用例有完全相同的语句覆盖。文献[3,6]指出，相似性的测试用例可能会损害TBFL方法的功效。为了解决该问题，文献[3]提出了SAFL(similarity-aware fault localization)方法，它把每一个测试用例都看作是一个Fuzzy集合，并用模糊集理论和概率理论计算语句的怀疑程度，借此处理测试用例之间的相似性问题，有效避免了相似性对错误定位的副作用。文献[3]通过两类实验比较SAFL方法、Dicing方法、TARANTULA方法的功效，得出以下结论：SAFL方法不仅能够有效地处理包含了许多冗余测试用例的测试集，而且在没有多少冗余测试用例的测试用例集上也能有效地执行。

在测试实践方面，测试用例之间的相似性总是难以避免的，所以讨论类似SAFL方法是有价值的。另外，TBFL方法(SAFL方法也不例外)也存在这样的问题，就是并没有充分利用原程序和测试用例本身所包含的信息去辅助测试结果寻找错误根源，致使TBFL方法有时对程序员产生误导，甚至对程序里隐蔽错误的揭露"无能为力"。本文的目的就是解决上述在TBFL方法(也包括SAFL方法)中存在的问题，为此提出一个基于随机理论的新的TBFL方法。其基本思想就是：将整个待测程序看成是一个随机变量，这样在测试前对程序的语句错误就有了一个先验概率分布；然后将这个测试集也看成是个随机变量，考察每个测试用例对语句错误的捕捉能力，即充分挖掘测试用例的信息，用测试用例反映的程序语句有关信息对程序语句的概率作一些调整，调整后的概率称为后验校正概率；最后就根据这个后验校正概率对错误语句进行排序。可以说我们是以一种原则性方法开发一个随机TBFL方法类型，传统的TBFL方法都可以纳入这个框架中。因此本文也进而用此模型中的基本概念对几个主要的TBFL方法、SAFL方法在一个实例上进行分析和比较，以说明基于随机信息之上的方法的优越性，尤其是该方法在冗余情况下受到的伤害极小，这样就符合直观观点：多测试对定位错误有帮助。

为了方便，今后如无特殊声明，也把相似性用例称为冗余用例，但从上下文可以看出指的是哪一种情况。同样，今后如无特殊声明，也把文献[3]提出的SAFL方法归到TBFL方法一类中去，即当说到TBFL方法时，也认为它包括了SAFL方法。

## 2 随机TBFL方法算法模型

现在提出一个新类型TBFL方法，称之为随机TBFL算法模型，这里只给出一个参考模型。

### 2.1 程序随机变量 $X$

用 $X=\{x_1,x_2,\cdots,x_m\}=\{x \mid x$ 是程序语句$\}$表示程序，它是语句的集合，其中语句 $x_i$ 的下标 $i$ 可按程序的书写方式编码。假定程序是"精心"编码的，即是由有责任心且有熟练技能的开发人员编写的。然而由于各种各样不可控制的因素的影响，程序发生错误仍然是难免的。把程序的错误归咎到语句层次上，既然程序是精心构造的，那么它的每个语句出错便是偶然现象。因此，视程序 $X$ 为随机变量，它是离散的，用 $r_k=P(X=x_k)$ 表示语句 $x_k$ 出错的概率。$r_k(k=1,2,\cdots,m)$ 是程序 $X$ 这个随机变量的先验概率质量函数。可以根据开发人员的经验、历史资料分析各种语句类型通常犯错误的可能性，从而确定 $r_k$ 之值。如果没有这方面的资料，可以按照统计学上"同等无知"原则，令 $r_k=\frac{1}{m},k=1,2,\cdots,m$，其中 $m$ 是程序 $X$ 的语句总数。

### 2.2 测试随机变量 $T$

用 $T=\{t_1,t_2,\cdots,t_n\}$ 表示测试用例集，其中 $t_j,j=1,2,\cdots,n$ 表示测试用例，它们用来对程序 $X$ 进行测试。把 $T$ 分为两类：$T_p$ 和 $T_f$。$T_p$ 是这样的测试用例组成的子集合，即该测试用例测试程序时没有发现错误，也就是说它的测试结果和预想结果是一致的。同理，$T_f$ 是测试程序时发现错误(即实际结果与预想结果不一致)的测试用例组成的子集合。

设计测试用例的目的是想捕获程序错误，它们捕获错误的可能性也是随机现象，可以根据测试人员的经验和软件测试理论确定每个用例捕获错误的概率，用 $p_t=p(t)$ 表示 $t$ 用例能检测出错误的概率。若没有这方面的历史资料，不妨假设 $p_t=\frac{1}{n}$，$n$ 是 $T$ 中测试用例的总数，这也是统计学中"同等无知"原则的应用。

### 2.3 $(X,T)$联合概率计算

$X,T$ 两个随机变量如前所述，用 $p(x,t)$ 表示程序 $X$ 和测试 $T$ 的联合概率质量函数。根据概率论，$p(x,t)=p(t)\cdot p(x|t)$，其中 $p(t)$ 是用例 $t$ 捕获错误的先验概率，$p(x|t)$ 是已知 $t$ 时语句 $x$ 发生错误的条件概率，或者说是根据 $t$ 用例测试结果对语句 $x$ 发生错误的概率的后验校准，建议用下述方式进行校准：

· 如果 $t\in T_p$，

$$p(x|t)=\begin{cases}(\beta-1)r_x, & x\in t\\ \beta r_x, & x\notin t\end{cases}(\beta>1) \tag{1}$$

式中，$r_x$ 是语句 $x$ 出错的先验概率；$x\in t(x\notin t)$ 表示运用测试用例 $t$ 时，语句 $x$ 被执行(语句 $x$ 未被执行)；$\beta$ 是归一化因子，它由下式决定：

$$\sum_x p(x|t)=\sum_{x\in t}(\beta-1)r_x+\sum_{x\notin t}\beta r_x=\beta-\sum_{x\in t}r_x=1$$

所以

$$\beta=1+\sum_{x\in t}r_x \tag{2}$$

· 如果 $t\in T_f$，

$$p(x|t)=\begin{cases}\beta r_x, & x\in t\\ (\beta-1)r_x, & x\notin t\end{cases}(\beta>1) \tag{3}$$

同样 $\beta$ 由下式决定：

$$\sum_x p(x|t)=\sum_{x\in t}\beta r_x+\sum_{x\notin t}(\beta-1)r_x=\beta-\sum_{x\notin t}r_x=1$$

所以

$$\beta=1+\sum_{x\notin t}r_x \tag{4}$$

式(1)和式(3)都源自一个自然想法，即如果 $t\in T_p$，则 $t$ 所覆盖的语句出错的怀疑程度降低，而程序的错误在它未覆盖的语句应负更大一些责任。同样，对于 $t\in T_f$，$t$ 覆盖的语句应对程序出错负责，而 $t$ 未覆盖的语句其怀疑程度降低。

当然也可以用其他方法调整语句的后验出错概率，这里给出的 $\beta$ 因子计算法则式(2)和式(4)只不过是一个参考参数，即 $\beta$ 计算法则未必是最好的一种参数。

### 2.4 边缘分布

前面已经叙述$(X,T)$为二元随机向量，它的联合概率质量函数 $p(x,t)=p(t)p(x|t)$。由 $p(x,t)$的形式很容易求出$(X,T)$两个边缘分布。作为$(X,T)$中"成份"$T$ 的边缘分布即为 $p(t)$，而作为$(X,T)$联合二元随机变量中"成份"$X$ 的边缘分布可以用式子$\sum_t p(t)\cdot p(x|t)$计算。

为了避免混淆，用 $X_T$ 表示$(X,T)$中成份 $X$ 的随机变量。$X_T$ 可以形象地看成观测到 $T$ 后产生的新随机变量，一般来说，它关于程序中语句 $x$ 的出错概率与原程序变量 $X$ 在语句 $x$ 的出错概率有所不同，即

$$P(X_T=x)=\sum_t p(t)\cdot p(x|t) \tag{5}$$

注意，$X$ 和 $X_T$ 都是关于程序的随机变量，$X$ 的分布是程序的先验分布，$X_T$ 分布是程序的后验分布。然后根据 $X_T$ 的分布来决定怎样帮助开发人员寻找错误语句。直观上，$P(X_T=x)$越大的语句，$x$ 越值得怀疑。

这样就给出了用随机变量构造 TBFL 方法的一般框架。总结如下：

1. 视程序为随机变量 $X$，它取语句为值，即 $X=\{x_1,x_2,\cdots,x_m\}$，其中 $x_i(i=1,2,\cdots,m)$为程序的第 $i$ 条语句，$m$ 为程序语句总数。根据理论、经验、历史资料，甚或根据"同等无知"统计原则，确定 $X$ 的先验分布，即令

$r_k=P(X=x_k),k=1,2,\cdots,m$

满足条件：$0\leqslant r_k\leqslant 1,k=1,2,\cdots,m;\sum_k r_k=1$。

2. 视测试集为随机变量 $T$，它取测试用例为值，即 $T=\{t_1,t_2,\cdots,t_n\}$，其中 $t_j(j=1,2,\cdots,n)$为测试集中第 $j$ 个测试用例，$n$ 为用例总数。根据理论、经验、历史资料，甚或根据"同等无知"统计原则，确定 $T$ 的分布，即令

$p_{t_i}=P(T=t_i),i=1,2,\cdots,n$

满足条件：$0\leqslant p_{t_i}\leqslant 1,i=1,2,\cdots,n;\sum_{t_i} p_{t_i}=1$。

3. 用测试集 $T$ 对程序 $X$ 进行测试，根据测试结果，把 $T$ 分成两组：$T_p$ 和 $T_f$。它们分别表示失败用例集和通过用例集。

4. 求 $p(x|t)$：用每个用例 $t$ 的测试结果调整每条语句 $x$ 的出错概率。

若 $t\in T_p$，则

$$\forall x,p(x|t)=\begin{cases}(\beta-1)r_x, & x\in t\\ \beta r_x, & x\notin t\end{cases}(\beta>1)$$

式中，$\beta=1+\sum_{x\in t} r_x$。

若 $t\in T_f$，则

$$\forall x,p(x|t)=\begin{cases}\beta r_x, & x\in t\\ (\beta-1)r_x, & x\notin t\end{cases}(\beta>1)$$

式中，$\beta=1+\sum_{x\notin t} r_x$。

5. 求程序后验分布 $X_T$。

$\forall x,P(X_T=x)=\sum_t p(t)\cdot p(x|t)$

6. 按 $X_T$ 的分布对程序语句出错的可能性排序，概率越大，其就越值得怀疑，因此排位越靠前。

## 3 实例分析

我们用文献[3]中给出的程序和测试用例(见图 1)应用上节所推荐的随机 TBFL 模型给出语句怀疑度排列，并对 Dicing、TARANTULA、SAFL 3 个方法进行随机分析和比较。有关图 1 详细情况请查阅该文献，该程序的错误语句是 $x_2$($x_2$ 应为 $m=z$)和 $x_7$($x_7$ 应为 $m=x$)。图 2 是该程序的流程图(旁边标注语句编号，底部($k$)标注路径 $k$)。

| | | Test suite 1 | | | | Test suite 2 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
| Mid(){ int x,y,z,m; | statements | 2,1,3 | 1,3,2 | 5,5,3 | 1,2,3 | 3,3,5 | 1,1,2 | 2,2,3 | 5,5,2 |
| read ("Enter 3 numbers:",x,y,z); | $x_1$ | ● | ● | ● | ● | ● | ● | ● | ● |
| m=x; | $x_2$ | ● | ● | ● | ● | ● | ● | ● | ● |
| if (y<z) | $x_3$ | ● | ● | ● | ● | ● | ● | ● | ● |
| if (x<y) | $x_4$ | ● | | | ● | ● | ● | ● | |
| m=y; | $x_5$ | | | | ● | | | | |
| else if (x<z) | $x_6$ | ● | | | | ● | ● | ● | |
| m=y; | $x_7$ | ● | | | | ● | ● | ● | |
| else | $x_8$ | | ● | ● | | | | | ● |
| if (x>y) | $x_9$ | | ● | ● | | | | | ● |
| m=y; | $x_{10}$ | | | | | | | | |
| else if (x>z) | $x_{11}$ | | ● | ● | | | | | ● |
| m=x; | $x_{12}$ | | | ● | | | | | ● |
| printf ("Middle number is :",m); | $x_{13}$ | ● | ● | ● | ● | ● | ● | ● | ● |
| } | | F | F | P | P | P | P | P | P |

图 1 A faulty program and its execution traces

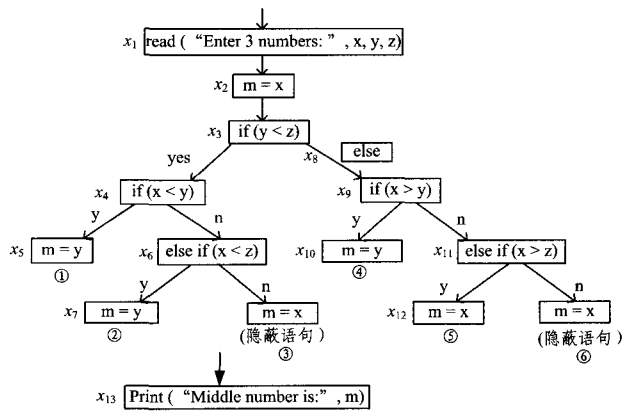


图 2 图 1 中程序流程图

### 3.1 随机 TBFL 方法

首先采用上述程序和测试集，以及前面介绍的随机 TBFL 方法。

现在程序 $X=\{x_1,x_2,\cdots,x_{13}\}$，例如 $x_1$ 表示编号为 1 的语句 read("Enter 3 numbers:",x,y,z)，根据编码经验，语句 $x_2,x_5,x_7,x_{10},x_{12}$ 容易出错，特别是语句 $x_2$，它牵涉到编码"技巧"，更容易出错。令每个语句的出错概率为 $r_k,k=1,2,\cdots,13$。由图 2 可以看出，$x_5,x_7,x_{10},x_{12}$ 以及由 $x_2$ 语句所决定的两个"隐蔽语句"是程序用来断定输入的 3 个数 $x,y,z$ 的中位数，因此 $x_2$ 犯错误的可能性应是 $x_5$ 等语句犯错误的可能性的 2 倍。至于 $x_5,x_7,x_{10},x_{12}$，主要由输入 3 个数 $x,y,z$ 的排序关系决定，例如当 $x<y<z$ 时，由路径①得到中位数 $m=y$，所以 $x_5$ 犯错误的可能性应是判定语句例如"if $x<y$"犯错误的可能性的 2 倍。由于对称性，不妨认为 $x_5,x_7,x_{10},x_{12}$ 犯错误可能性的大小一样。当把其他语句(即除了 $x_2,x_5,x_7,x_{10},x_{12}$)犯错误的可能性视为一样时，便得到程序 $X$ 的语句犯错误概率 $r_k,k=1,2,\cdots,13$：$r_2=2r_5=2r_7=2r_{10}=2r_{12}=4r_k(k\neq 2,5,7,10,12)$。由 $\sum r_k=1$，可得：

$$r_2=\frac{4}{20},r_5=r_7=r_{10}=r_{12}=\frac{2}{20},$$

$$r_k=\frac{1}{20},k\neq 2,5,7,10,12 \tag{6}$$

3.1.1 测试集中有冗余用例情况

(1) 测试随机变量 $T$

选择 Test suite 1+Test suite 2 作为整个测试集 $T$。$T$ 有 8 个用例，按图 1 顶格右边给出的次序分别把它们编号为 $t_1,t_2,\cdots,t_8$。例如 $t_1$ 是用例：$x=2,y=1,z=3$。在 8 个用例中，$t_1,t_2$ 是失败用例，其余 6 个是通过用例。用 $T_f,T_p$ 分别表示失败用例集合和通过用例集合，即

$$T=\{t_1,t_2,\cdots,t_8\},T_f=\{t_1,t_2\},T_p=\{t_3,\cdots,t_8\} \tag{7}$$

根据测试经验，$t_1,t_2,t_4$ 比 $t_3,t_5,t_6,t_7,t_8$ 更容易捕获错误，这是由于程序中的判别式都是严格不等式，当输入的 3 个数中有 2 个值相同时，程序的错误容易蒙混过去。例如 $t_1=\{2,1,3\}$，它代表输入 $x=2,y=1,z=3$。显然 $t_1$ 执行时，经过图 2 的路径②是失败路径。而 $t_5=\{3,3,5\}$，它表示 $x=3,y=3,z=5$。它也经过第②条路径，但由于 $x=y=3$，因此语句 $x_7(m=y)$虽然有错，但是这一用例却“蒙混”过关，通过测试。总之，由于图 1 中程序里的判定式都是严格不等式，因此图 2 流程图里 6 条路径表示小、中、大 3 个输入数的不同排列，而输入 3 个数中若有 2 个数相同，则它可以看作是“两个排列”。因此我们认为 $t_1,t_2,t_4$ 3 个用例捕获错误的可能性是其余 5 个用例(它们输入的 3 个数中都有两个是相同的)捕获错误的可能性的 2 倍是合理的，故不妨认为它们捕获错误的概率如下：

$$\begin{aligned}&p(t_1)=p(t_2)=p(t_4)=\frac{2}{11}\\&p(t_3)=p(t_5)=p(t_6)=p(t_7)=p(t_8)=\frac{1}{11}\end{aligned} \tag{8}$$

(2) $(X,T)$的联合概率质量函数

由于 $t_1$ 是失败用例，因此根据式(3)计算 $p(x|t_1)$，其中根据式(4)和式(6)计算 $\beta$ 值为：$\beta=\frac{29}{20}$。于是把 $\beta$ 代入 $p(x|t_1)$表达式中，最后得到

$$p(x|t_1)=\begin{cases}\frac{29}{20}\times\frac{1}{20}, & x=x_1,x_3,x_4,x_6,x_{13}\\ \frac{29}{20}\times\frac{4}{20}, & x=x_2\\ \frac{29}{20}\times\frac{2}{20}, & x=x_7\\ \frac{9}{20}\times\frac{2}{20}, & x=x_5,x_{10},x_{12}\\ \frac{9}{20}\times\frac{1}{20}, & x=x_8,x_9,x_{11}\end{cases} \tag{9}$$

类似地，可以求出 $p(x|t_2)$、$p(x|t_3)$、$p(x|t_4)$、$p(x|t_5)$、$p(x|t_6)$、$p(x|t_7)$、$p(x|t_8)$，限于篇幅，计算省略。

根据 $p(t,x)=p(t)\cdot p(x|t)$可以求出$(X,T)$的联合概率质量函数。但本文对 $p(x,t)$并不关心，故无需计算。

(3) $(X,T)$的边缘分布——$X_T$ 的分布

有了 $T$ 分布式(8)以及 $p(x|t_k),k=1,2,\cdots,8$ 等条件分布，根据式(5)可以计算$(X,T)$二元随机变量的边缘随机变量 $X_T$ 的分布如下：

$$\begin{aligned}&P(X_T=x_1)=\frac{195}{4400};P(X_T=x_2)=\frac{780}{4400};\\&P(X_T=x_3)=\frac{195}{4400};P(X_T=x_4)=\frac{195}{4400};\\&P(X_T=x_5)=\frac{430}{4400};P(X_T=x_6)=\frac{235}{4400};\\&P(X_T=x_7)=\frac{470}{4400};P(X_T=x_8)=\frac{255}{4400};\\&P(X_T=x_9)=\frac{255}{4400};P(X_T=x_{10})=\frac{510}{4400};\\&P(X_T=x_{11})=\frac{255}{4400};P(X_T=x_{12})=\frac{430}{4400};\\&P(X_T=x_{13})=\frac{195}{4400};\end{aligned} \tag{10}$$

(4) 语句出错可能性排序

按 $X_T$ 的概率分布对语句出错可能性排序，概率越大，排序越靠前，说明该语句更可能有错误，排序见表 1，其中序号是语句排序号。

表 1 有冗余测试时随机 TBFL 方法中 $X_T$ 的分布及语句排序

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ |
|---|---|---|---|---|---|---|---|
| 排序 | $x_2$ | $x_{10}$ | $x_7$ | $x_5$ $x_{12}$ | $x_8$ $x_9$ $x_{11}$ | $x_6$ | $x_1$ $x_3$ $x_4$ $x_{13}$ |
| 概率 | 780/4400 | 510/4400 | 470/4400 | 430/4400 | 255/4400 | 235/4400 | 195/4400 |

3.1.2 测试集中无冗余用例情况

(1) 测试随机变量 $T^*$

选择图 1 中 Test suite 1 作为无冗余测试集 $T^*$，$T^*$ 只有 4 个用例，它们的编号分别为 $t_1,t_2,t_3,t_4$，与前段 1 中的前 4 个用例一样。即：

$$T^*=\{t_1,t_2,t_3,t_4\} \tag{11}$$

式中，$t_1,t_2\in T_f^*,t_3,t_4\in T_p^*$。

显然，式(7)中 $T$ 集合是由 $T^*$ 通过增添测试用例 $t_5,t_6,t_7,t_8$ 等 4 个冗余用例(都是通过测试)而形成的。

和前段 1 中理由一样，不妨认为 $t_1,t_2,t_4$ 捕获错误的概率为 $t_3$ 的 2 倍，于是

$$p(t_1)=p(t_2)=p(t_4)=\frac{2}{7},p(t_3)=\frac{1}{7} \tag{12}$$

(2) $(X,T^*)$的联合分布

$p(x|t_1),p(x|t_2),p(x|t_3),p(x|t_4)$ 4 个条件分布分别和前段在 $T$ 测试下的条件分布相同，这是因为它们的计算只涉及到 $X$ 的先验分布和用例 $t$ 是否为失败(或通过)以及用例 $t$ 覆盖语句的情况。有了条件分布以及用例捕获错误的概率(即式(12)数据)，便能计算$(X,T^*)$的联合分布。同理，这里并不需要$(X,T^*)$的分布。

(3) $(X,T^*)$的边缘分布——$X_{T^*}$ 的分布

有了 $T^*$ 分布式(12)以及诸 $p(x|t_k),k=1,2,3,4$，根据式(5)，可以计算$(X,T^*)$二元随机变量之边缘随机变量 $X_{T^*}$ 的分布如下：

$$\begin{aligned}&P(X_{T^*}=x_1)=\frac{150}{2800};P(X_{T^*}=x_2)=\frac{600}{2800};\\&P(X_{T^*}=x_3)=\frac{150}{2800};P(X_{T^*}=x_4)=\frac{130}{2800};\\&P(X_{T^*}=x_5)=\frac{180}{2800};P(X_{T^*}=x_6)=\frac{170}{2800};\\&P(X_{T^*}=x_7)=\frac{340}{2800};P(X_{T^*}=x_8)=\frac{150}{2800};\\&P(X_{T^*}=x_9)=\frac{150}{2800};P(X_{T^*}=x_{10})=\frac{260}{2800};\\&P(X_{T^*}=x_{11})=\frac{150}{2800};P(X_{T^*}=x_{12})=\frac{220}{2800};\\&P(X_{T^*}=x_{13})=\frac{150}{2800}\end{aligned} \tag{13}$$

(4) 语句出错可能性排序

按 $X_{T^*}$ 概率分布对程序语句出错的可能性进行排序，概率越大，排序越前，表示它越可能有错，见表 2。

表 2　无冗余测试时随机 TBFL 方法中 $X_{T^*}$ 的分布及语句排序

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ |
|---|---|---|---|---|---|---|---|---|
| 排序 | $x_2$ | $x_7$ | $x_{10}$ | $x_{12}$ | $x_5$ | $x_6$ | $x_1$ $x_3$ $x_8$ $x_9$ $x_{11}$ $x_{13}$ | $x_4$ |
| 概率 | $\frac{600}{2800}$ | $\frac{340}{2800}$ | $\frac{260}{2800}$ | $\frac{220}{2800}$ | $\frac{180}{2800}$ | $\frac{170}{2800}$ | $\frac{150}{2800}$ | $\frac{130}{2800}$ |

3.1.3　小结

从上面的分析中可以看出，随机 TBFL 无论是在有冗余情况还是无冗余情况下，都能指出程序的错误语句 $x_2$ 和 $x_7$，即它们的后验校正概率大。由表 1 和表 2 可以看出，在冗余测试里，随机 TBFL 方法把有错语句 $x_7$ 排在无错语句 $x_{10}$ 之后，与无冗余测试相比，还是受到了一点影响，但是在整体上错误语句 $x_2$ 和 $x_7$ 的概率都比其它正确语句要高，说明随机 TBFL 方法并没有受到冗余测试的伤害。这是很重要的结论，因为直观上冗余测试是无法避免的，而且多测试应该有助于开发人员寻找错误。

对其它的 TBFL 方法，也可做类似的随机变量分析。下面就用图 1 中的程序 $X$ 和测试用例集 $T$、$T^*$ 对文献[3]提到的几个 TBFL 方法和 SAFL 方法进行随机理论分析和比较。在分析中用到文献[3]中的结果(见文献[3]资料表 1(results of the motivating example))，为了方便读者，将其总结如下，记为表 3。有关表中的详细情况，请参阅文献[3]。首先分析 Dicing 方法。

表 3　Dicing 方法和 TARANTULA 方法的结果

| The results of test suite 1 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Statement | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
| Dicing | 0 | 0 | 0 | 1 | 0 | 2 | 2 | 1 | 1 | 0 | 1 | 0 | 0 |
| TARANTULA | 0.5 | 0.5 | 0.5 | 0.5 | 0 | 1 | 1 | 0.5 | 0.5 | — | 0.5 | 0 | 0.5 |
| The results of test suite 1+test suite 2 | | | | | | | | | | | | | |
| Statement | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
| Dicing | 0 | 0 | 0 | 2 | 0 | 3 | 3 | 4 | 4 | 0 | 4 | 0 | 0 |
| TARANTULA | 0.5 | 0.5 | 0.5 | 0.43 | 0 | 0.5 | 0.5 | 0.6 | 0.6 | — | 0.6 | 0 | 0.5 |

## 3.2　Dicing 方法随机分析

3.2.1　测试集中包含冗余用例情况

根据测试集 $T$ 的测试结果，采用 Dicing 方法对语句的怀疑程度进行排序，见表 3。利用这个排序规定新随机变量 $X_T$ 的分布，见表 4。

表 4　Dicing 方法的 $X_T$ 分布

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dicing 排序 | 0 | 0 | 0 | 2 | 0 | 3 | 3 | 4 | 4 | 0 | 4 | 0 | 0 |
| $X_T$ 的分布 | 0 | 0 | 0 | 2/20 | 0 | 3/20 | 3/20 | 4/20 | 4/20 | 0 | 4/20 | 0 | 0 |

这里 $X_T$ 表示利用测试结果对语句出错概率重新调整后按新的分布变化的随机变量。根据 Dicing 方法精神，程序 $X=\{x_1,x_2,\cdots,x_{13}\}$ 每个语句的出错概率机会均等，即认为 $r_k=P(X=x_k)=\frac{1}{13}$，$k=1,2,\cdots,13$。测试集 $T$ 中每一个用例捕获错误的概率也视为一样，即认为 $p(t_i)=\frac{1}{8}$，$i=1,2,\cdots,8$。现在利用测试 $T$，将程序的均匀先验概率调整为程序后验概率，即 $X_T$ 的分布。$X_T$ 就相当于二元随机变量$(X,T)$的边缘随机变量 $X_T$。根据 $X_T$ 概率对语句出错的可能性排序，见表 5。

表 5　根据 $X_T$ 对语句出错排序

| | ① | ② | ③ | ④ |
|---|---|---|---|---|
| 排序 | $x_8$ $x_9$ $x_{11}$ | $x_6$ $x_7$ | $x_4$ | 其余语句 |
| 概率 | 4/20 | 3/20 | 2/20 | 0 |

3.2.2　测试集中无冗余用例情况

这里认为 $X=\{x_1,\cdots,x_{13}\}$ 服从均匀分布，即 $r_k=P(X=x_k)=\frac{1}{13}$，$k=1,2,\cdots,13$。同样，对于 $T^*=\{t_1,t_2,t_3,t_4\}$，也认为 $p(t_1)=p(t_2)=p(t_3)=p(t_4)=\frac{1}{4}$。下面讨论 TARANTULA 方法和 SAFL 方法时与讨论 Dicing 方法相同，凡是遇到程序和测试集都认为它们的先验分布皆为均匀分布。今后均作这样的假设，不再赘述。

根据测试集 $T^*$ 的测试结果，采用 Dicing 方法对语句的怀疑程度进行排序，见表 3。利用这个排序规定 $X_{T^*}$ 这个新随机变量的分布，见表 6。

表 6　Dicing 方法的 $X_{T^*}$ 分布

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dicing 排序 | 0 | 0 | 0 | 1 | 0 | 2 | 2 | 1 | 1 | 0 | 1 | 0 | 0 |
| $X_{T^*}$ 的分布 | 0 | 0 | 0 | 1/8 | 0 | 2/8 | 2/8 | 1/8 | 1/8 | 0 | 1/8 | 0 | 0 |

同前面有冗余测试情况一样，$X_{T^*}$ 即为$(X,T^*)$二元随机变量边缘随机变量 $X_{T^*}$，根据 $X_{T^*}$ 的概率对语句出错的可能性排序，见表 7。

表 7　根据 $X_{T^*}$ 对语句出错排序

| | ① | ② | ③ |
|---|---|---|---|
| 排序 | $x_6$ $x_7$ | $x_4$ $x_8$ $x_9$ $x_{11}$ | 其余语句 |
| 概率 | 2/8 | 1/8 | 0 |

3.2.3　小结

向无冗余测试集 $T^*$ 添加冗余用例以后得到有冗余测试集 $T$，在两个测试集上运用 Dicing 方法寻找语句出错可能性的大小。按照随机分析可见，冗余测试对 Dicing 方法的确有伤害。尤其考虑语句出错排序表 5 和表 7，这个伤害更为明显，因为 $T^*$ 把错误语句 $x_7$ 和正确语句 $x_6$ 排在首位，而 $T$ 却把 $x_7$ 排在第 2 个层次，在它之前还有许多正确语句需要检测。除此之外，$T$ 和 $T^*$ 都找不到隐蔽错误语句 $x_2$，而我们认为这个比冗余伤害问题更为严重。

## 3.3　TARANTULA 方法随机分析

3.3.1　测试集中包含冗余用例情况

根据测试集 $T$ 的测试结果，TARANTULA 方法计算语句怀疑程度，见表 3。利用这个结果规定新随机变量 $X_T$ 的概率分布，见表 8(suspiciousness 简写为 sus)。

表 8　TARANTULA 方法的 $X_T$ 分布

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sus | 0.5 | 0.5 | 0.5 | 0.43 | 0 | 0.5 | 0.5 | 0.6 | 0.6 | — | 0.6 | 0 | 0.5 |
| $X_T$ 分布 | 0.5N | 0.5N | 0.5N | 0.43N | 0 | 0.5N | 0.5N | 0.6N | 0.6N | 0 | 0.6N | 0 | 0.5N |

其中，“—”表示语句 $x_{10}$ 未执行，我们也把它作为 0 处理。$N$ 是归一化因子，由 $\sum_k P(X_T=x_k)=1$，可得 $N=\frac{1}{5.23}$，从而可得 $X_T$ 的概率分布，例如当 $X_T=x_6$ 时，$P(X_T=x_6)=0.5N=\frac{0.5}{5.23}=\frac{50}{523}$。

$X_T$ 相当于二元随机变量 $(X,T)$ 的边缘随机变量 $X_T$，即是根据测试结果对程序先验概率的校正，按 $X_T$ 的分布确定语句怀疑程度的次序，见表 9。

表 9　根据 $X_T$ 对语句出错排序

| | ① | ② | ③ | ④ |
|---|---|---|---|---|
| 排序 | $x_8$ $x_9$ $x_{11}$ | $x_1$ $x_2$ $x_3$ $x_6$ $x_7$ $x_{13}$ | $x_4$ | 其余语句 |
| 概率 | 0.6/5.23 | 0.5/5.23 | 0.43/5.23 | 0 |

3.3.2　测试集中无冗余测试用例情况

根据测试集 $T^*$ 的测试结果，采用 TARANTULA 方法计算语句怀疑程度，见表 3。利用这个结果规定新随机变量 $X_{T^*}$ 的概率分布见表 10(suspiciousness 简写为 sus)。

表 10　TARANTULA 方法的 $X_{T^*}$ 分布

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sus | 0.5 | 0.5 | 0.5 | 0.5 | 0 | 1 | 1 | 0.5 | 0.5 | — | 0.5 | 0 | 0.5 |
| $X_{T^*}$ 分布 | 0.5N | 0.5N | 0.5N | 0.5N | 0 | N | N | 0.5N | 0.5N | 0 | 0.5N | 0 | 0.5N |

其中，“—”表示语句 $x_{10}$ 未执行，我们把它作为 0 处理。$N$ 是归一化因子，由 $\sum_k P(X_{T^*}=x_k)=1$，可得 $N=\frac{1}{6}$。同样，$X_{T^*}$ 即为 $(X,T^*)$ 二元随机变量边缘随机变量 $X_{T^*}$。按 $X_{T^*}$ 分布将语句出错的可能性排序，见表 11。

表 11　根据 $X_{T^*}$ 对语句出错排序

| | ① | ② | ③ |
|---|---|---|---|
| 排序 | $x_6$ $x_7$ | $x_1$ $x_2$ $x_3$ $x_4$ $x_8$ $x_9$ $x_{11}$ $x_{13}$ | 其余语句 |
| 概率 | 1/6 | 0.5/6 | 0 |

3.3.3　小结

当无冗余测试集 $T^*$ 增添冗余用例得到测试集 $T$ 后，用 TARANTULA 方法寻找错误语句的能力也确实受到了伤害。如果对语句出错可能性的排序表 9 和表 11 进行比较，更能看出这一点。用 $T^*$ 排序，有错语句 $x_7$ 和正确语句 $x_6$ 被排在首位；而用 $T$ 排序，$x_7$ 降到第 2 层次且和许多无错语句“混”在一起，对开发人员帮助不大。同样，和 Dicing 方法一样，TARANTULA 方法无论是用测试集 $T$ 还是 $T^*$ 都无法明显找到错误语句 $x_2$，这比冗余伤害更为严重。

### 3.4　SAFL 方法随机分析

3.4.1　测试用例集中包含冗余用例情况

根据测试集 $T$，由文献[3]表 2 计算语句怀疑度 $p(j)$ 的结果，规定 $X_T$ 的概率分布，见表 12。

表 12　SAFL 方法的 $X_T$ 分布

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P(j) | 0.78 | 0.78 | 0.78 | 0.78 | 0 | 1 | 1 | 0.89 | 0.89 | 0 | 0.89 | 0 | 0.78 |
| $X_T$ 分布 | 0.78N | 0.78N | 0.78N | 0.78N | 0 | N | N | 0.89N | 0.89N | 0 | 0.89N | 0 | 0.78N |

其中，$N$ 是归一化因子，由 $\sum_j P(X_T=x_j)=1$，得到 $N=\frac{1}{8.57}=\frac{100}{857}$。于是就有了 $X_T$ 的分布，$X_T$ 即为 $(X,T)$ 的边缘分布。按 $X_T$ 的分布把错误语句的可能性排序，见表 13。

表 13　根据 $X_T$ 对语句出错排序

| | ① | ② | ③ | ④ |
|---|---|---|---|---|
| 排序 | $x_6$ $x_7$ | $x_8$ $x_9$ $x_{11}$ | $x_1$ $x_2$ $x_3$ $x_4$ $x_{13}$ | 其余语句 |
| 概率 | 1/8.57 | 0.89/8.57 | 0.78/8.57 | 0 |

3.4.2　测试集中无冗余用例情况

由文献[3]表 2 可以看出，根据测试集 $T^*$ 计算出的 $p(j)$ 和根据测试集 $T$ 计算出的 $p(j)$ 完全一样。所以作为 $(X,T^*)$ 的边缘变量 $X_{T^*}$ 的概率分布与作为 $(X,T)$ 边缘变量 $X_T$ 的概率分布也完全一样，因而排序也同表 13 一样。

3.4.3　小结

与无冗余测试 $T^*$ 相比，冗余测试 $T$ 并没有对 SAFL 方法的功效造成伤害。而无冗余测试下可能有错语句的排序完全和表 13 相同更能说明这一点。这说明 SAFL 确实达到它所设计的目的，即消除了冗余对该方法功效的伤害。但是，同前一样，它也无法指出错误语句 $x_2$，而这个问题比冗余伤害更为严重。

### 3.5　总结：几个 TBFL 方法的比较

我们把第 3.2—3.4 节中的 Dicing 方法、TARANTULA 方法、SAFL 方法都纳入到随机分析框架，这样就可以把它们和我们推荐的随机 TBFL 方法进行比较。

在语句错误可能性排序上，随机 TBFL 方法最优。无论是冗余测试 $T$ 还是非冗余测试 $T^*$，随机 TBFL 方法都能找出原程序真实错误语句 $x_2$ 和 $x_7$，并把它们排在首两位(在 $T^*$ 测试中)，或第一、第三位(在 $T$ 测试中)。而 Dicing 方法、TARANTULA 方法、SAFL 方法都无法“找到”错误语句 $x_2$，即使能指出错误语句 $x_7$，但 $x_7$ 也与其他无错语句(例如和 $x_6$)并列混在一起被同等怀疑，这在有冗余情况下尤其严重，因为在 $x_7$ 的前面还有大量无错语句被怀疑着。因此在帮助开发人员尽快找到错误语句上，随机 TBFL 方法功效更为显著。

随机 TBFL 方法不但避免了功效伤害，而且显示冗余测试对程序开发的功效更有帮助。前面已经提过，这一点尤其重要，直观上多测试应该对发现错误更有利，更何况避免相似性测试是根本无法做到的。因此设计更好的 TBFL 方法，犹如证据繁多并不妨碍有经验的侦破人员办案一样，从重复冗余测试(例如即兴测试)里寻找出错语句的正确位置，是可以做到的，这正是随机 TBFL 方法对我们的启示。

## 4　算法功效进一步说明和小型实验

### 4.1　算法说明

随机 TBFL 算法的实质是把程序 $X$ 里包含的信息(抽象为 $X$ 的先验分布)和测试用例集 $T$ 里包含的信息(抽象为 $T$ 的先验分布)以及具体测试结果(把 $T$ 分为 $T_f$ 和 $T_p$ 两类以及每个用例覆盖语句的情况)综合在一起考虑，这种考虑分为两个层次。首先将每一个测试用例 $t$ 视为通过用例或是失败用例以及覆盖的语句情况，分别对原程序 $X$ 里每一个语句 $x$ 错误的可能性进行调整得到条件分布 $p(x|t)$，这是在微观层次上体现测试工作的实质；然后对每一个语句 $x$ 按照测试集

$T$ 的概率加权，求出 $\sum_t p(t)p(x|t)$，得到程序后验随机变量的后验分布，这是在全局层次上体现测试工作的实质；最后，凭借 $X_T$ 后验分布，把原程序的语句按概率大小排序并把它推荐给开发人员，以便尽快寻找出错语句。上面思想用随机 TBFL 算法流程图总结，如图 3 所示。

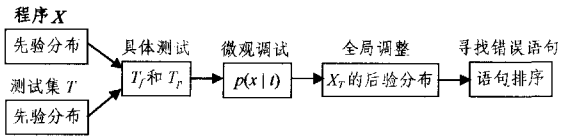


图 3　随机 TBFL 算法流程图

值得强调的是：1）$X$ 的先验分布的确定与原程序里哪个语句真的有错无关，它只是根据程序的“样式”分析得到，例如在上一节我们通过流程图的分析得到诸 $r_x$ 的值。2）$T$ 的先验分布的确定只与测试人员针对程序式样设计测试用例的类型、意图有关，而与它是通过用例或失败用例具体结果无关。总之，这两类分布是客观存在的，独立于具体测试过程。特别是，只要原程序 $X$ 的样式确定，不管它哪个语句真的有错，其 $X$ 的先验分布都是一样的。同样，只要针对 $X$ 的样式设计了测试用例集 $T$，则 $T$ 的先验分布也不依赖于哪个语句真的有错情况。3）$X$ 与 $T$ 的先验分布只是在看到具体实施测试结果以后才发生关联，从而在微观层次上得到 $p(x|t)$，在全局层次上得到 $X_T$ 的后验分布。

上一节用一个具体实例说明了我们的算法及其功效，但是 $X$ 和 $T$ 的先验分布（特别是 $X$ 的先验分布）都有“不精确”之嫌。为了更好地说明问题，本节从以下两个方面给出示例解释。

4.1.1　*原程序变体和存在逻辑错误的程序*

例 1　我们称图 1 中的程序为程序 I，它有两个错误语句 $x_2$（$x_2$ 应为 $m=z$）和 $x_7$（$x_7$ 应为 $m=x$），把程序 I 中的错误语句 $x_2$：“$m=x$” 改为正确语句 $x_2$：“$m=z$”，其余语句保留不变，称这个变体为程序 II，即程序 II 中只有一个错误语句 $x_7$（$m=y$）。现在程序 I 和程序 II 的“式样”完全一样，因此无论是哪个程序，都记为程序变量 $X$，并且 $X$ 的先验分布仍然为式(6)。

为了计算简单，只考虑测试集 1，运用前节同样符号，令 $T^*=\{t_1,t_2,t_3,t_4\}$。由于程序 II 的式样没有改变，因此设计 $T^*$ 的意图和类型也未改变，因此它的先验分布和式(12)中一样。

用测试集 $T^*$ 测试程序 $X$，注意 $t_1,t_2,t_3,t_4$ 覆盖语句情况和用 $T^*$ 测试程序 I 时一样没有改变，除了 $t_2$ 用例现在变为通过用例以外，其它用例结果类型仍和前面用 $T^*$ 测试程序 I 时一样，即这时，$T_f=\{t_1\}$，$T_p=\{t_2,t_3,t_4\}$。因此 $p(x|t_1)$、$p(x|t_3)$、$p(x|t_4)$计算结果不变，只是 $p(x|t_2)$需要重新计算。由此得出 $X_{T^*}$ 的概率分布并按照 $X_{T^*}$ 的大小排列语句，见表 14。

表 14　程序 II 的 $X_{T^*}$ 的概率分布及语句排序（$N=\frac{1}{2800}$）

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ |
|---|---|---|---|---|---|---|---|---|
| 排序 | $x_2$ | $x_7$ | $x_{10}$ | $x_{12}$ | $x_5$ | $x_6$ | $x_4$ | $x_1\ x_3\ x_8\ x_9\ x_{11}\ x_{13}$ |
| 概率 | 440N | 420N | 340N | 300N | 260N | 210N | 170N | 110N |

虽然程序 II 里正确语句 $x_2$ 排在错误语句 $x_7$ 的前面，但是它们的出错概率相差不大，前者是 440/2800，后者是 420/2800，两者都和第 3 位 $x_{10}$ 的 340/2800 相差较大，可以认为 $x_7$ 与 $x_2$ 属于一个等级，这说明我们的算法即使在小型 4 个测试用例情况下也是“健壮”的。

例 2　将程序 I 的 $x_2$（$m=x$）改为 $x_2$（$m=z$），$x_{11}$（else if（$x>z$））改为 $x_{11}$（else if（$x<z$）），其余语句不动，称这样的变体为程序 III。和例 1 一样，对于程序 III 和测试集 $T^*$，它们的先验概率分别与式(6)、式(12)一样。用测试集 $T^*$ 测试程序 III，显然可以得到 $T_f=\{t_1,t_2,t_3\}$，$T_p=\{t_4\}$。计算 $p(x|t)$ 时，由于只用到 $X$ 的先验分布以及 $t$ 覆盖语句信息和结果类型（失败或通过），现在 $X$ 先验分布未变，$t_1,t_4$ 的结果类型及其覆盖语句和用 $T^*$ 测试程序 I 的情况一致，因此 $p(x|t_1)$、$p(x|t_4)$不变。

对于 $t_2$，虽然它测试程序 I 和 III 的结果都是失败用例，但它测试两个程序覆盖的语句并非完全相同。对于 $t_3$，它不仅改变了结果类型，而且覆盖的语句也不完全相同，所以只需重新计算 $p(x|t_2)$、$p(x|t_3)$，以上分析就很容易验证，故不赘述，计算结果也省略。

由此可以计算关于程序 III 的后验概率，并按此概率的大小排列语句，见表 15。

表 15　程序 III 的 $X_{T^*}$ 的概率分布及语句排序（$N=\frac{1}{2800}$）

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ |
|---|---|---|---|---|---|---|---|
| 排序 | $x_2$ | $x_7\ x_{12}$ | $x_{10}$ | $x_1\ x_3\ x_8\ x_9\ x_{11}\ x_{13}$ | $x_6$ | $x_5$ | $x_4$ |
| 概率 | 656N | 288N | 208N | 164N | 144N | 128N | 104N |

怎样理解上述排序？我们可以分析程序 III 的流程图，程序 III 的流程图可以参照图 2，只是语句 $x_2$ 和 $x_{11}$ 作了变动以及两个隐蔽语句作了相应的调整，故这里省略。我们发现，如果输入 3 个不同的数 $x,y,z$，按它们大小排列，共有 6 种排列，对于每一种排序，程序执行时分别取不同的路径。

1. 按小中大排序，即 $x<y<z$，则程序执行路径①，由于 $x_5$（$m=y$），因此程序执行路径①时是正确的。

2. 按中小大排列，即 $y<x<z$，程序执行路径②，因语句 $x_7$（$m=y$）有错，程序失败，$x_7$ 应改为（$m=x$）。

3. 按大小中排列，即 $y<z<x$，程序执行路径③，这时由 $x_2$（$m=z$）决定的隐蔽语句得到 $m=z$ 是正确的。

4. 按大中小排列，即 $z<y<x$，程序执行路径④，由于 $x_{10}$（$m=y$）得出程序是正确的。

5. 按小大中排列，即 $x<z<y$，程序执行路径⑤，原本在程序 I 里 $x_{12}$（$m=x$）是正确语句，现在发现它成为新的错误语句，应改为 $m=z$。

6. 按中大小排列，即 $z<x<y$，程序执行路径⑥，它由 $x_2$（$m=z$）决定的隐蔽语句程序得到 $m=z$。但这时程序发生错误，它让最小的 $z$ 值当作了中位数，正确的应该是把 $x_2$ 重新写为 $m=x$。

路径③和路径⑥对于语句 $x_2$ 的不同要求，前者要求 $m=z$，后者要求 $m=x$，暴露出由于语句 $x_{11}$ 的错误选择使得 $x_2$ 左右为难，这引起的程序深层次的隐蔽逻辑错误是很严重的。总之，程序 III 有严重逻辑错误，它表现在语句 $x_2$ 的设计上（由于 $x_{11}$ 的条件判断式的选择），还有计算错误 $x_7$（$m=y$）和 $x_{12}$（$m=x$），$x_7$ 是程序 I 固有的，$x_{12}$ 是程序 III 里新产生的。现在我们的算法把这 3 个错误都排在其余语句的前面，特别是 $x_2$ 引起的逻辑错误尤为显目。程序员根据 $x_2$ 及（辅以）$x_{12}$ 的检查，不难查出 $x_{11}$ 设计的错误。这个例子充分说明了原程序及其变体的先验概率分布即使谈不上十分精确，也

是十分合理的，更重要的是它说明我们的算法即使在比较合理的先验分布下也运行得很好，特别对于隐蔽错误（由 $x_2$ 引起）的揭示更为有力。由于无论是失败用例或是通过用例，它们都执行语句 $x_2$，因此关于 $x_2$ 的错误的揭示是十分重要而且困难的。

4.1.2 *极端情况：全是通过用例的程序和先验概率"退化"为均匀分布情况*

例 3（用例全通过程序） 仍然考虑程序 I，它有两个错误语句 $x_2$，$x_7$，其程序变量 $X$ 的先验分布为式(6)。现在用测试集 $T_o=\{o_1,o_2,o_3,o_4\}$ 进行测试，其中 $o_1=\{10,13,15\}$，$o_2=\{8,6,4\}$，$o_3=\{5,9,2\}$，$o_4=\{17,19,21\}$。$T_o$ 的先验分布为均匀分布。

用 $T_o$ 测试程序 I，由图 2 可以看出它们都通过测试，$o_1$，$o_4$ 覆盖路径①诸语句，$o_2$ 覆盖路径④，$o_3$ 覆盖路径⑤，很容易计算诸 $p(x|o_i)$ 条件分布；再由 $T_o$ 的先验分布，不难计算程序的后验 $X_{T_o}$ 的概率分布，并按照后验分布概率得出语句怀疑度排列，见表 16。

表 16 $X_{T_o}$ 的概率分布及语句排序（$N=\frac{1}{1600}$）

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ |
|---|---|---|---|---|---|---|---|---|
| 排序 | $x_7$ | $x_{10}x_{12}$ | $x_2$ | $x_5$ | $x_6$ | $x_{11}$ | $x_4x_8x_9$ | $x_1x_3x_{13}$ |
| 概率 | 246N | 206N | 172N | 166N | 123N | 103N | 83N | 43N |

因为 Dicing 方法、TARANTULA 方法与 SAFL 方法的计算都涉及到失败用例，现在 $T_o$ 测试集全是由通过用例组成的，所以上述 3 个方法无法对程序 I 中的错误给出明确的提示，按照它们方法的思想，现在即使认为程序有错，那也是每个语句处于同等程度的怀疑情况。而我们的算法不仅把有错的 $x_7$ 单独排在第 1 位，而且单独排在第 3 位的有错语句 $x_2$ 前面也只有 2 个正确语句 $x_{10}$，$x_{12}$ 需要鉴定。测试原理告诉我们，测试员只能报告软件缺陷存在，却不能报告软件缺陷不存在。本例看出，Dicing 方法、TARANTULA 方法、SAFL 方法在这种情况下"无话可说"，而我们的算法却能大体上正确给出错误语句的"潜伏"位置。

例 4 先验概率退化情况

我们的算法精神是挖掘程序中和测试集中包含的固有的信息，并把它们掺合进具体测试实施里才能发挥更大的威效。如果没有程序和测试集的资料，只能把程序、测试集的先验分布取为均匀分布。

取原程序 I，先验分布为均匀分布，即

程序 $X=\{x_1,x_2,\cdots,x_{13}\}$ 的先验分布为：$r_k=\frac{1}{13}$，$k=1,2,\cdots,13$。

取测试集 $T^*$，先验分布也为均匀分布，即

$T^*=\{t_1,t_2,t_3,t_4\}$ 且先验分布为：$p(t_1)=p(t_2)=p(t_3)=p(t_4)=\frac{1}{4}$。

按基本算法算出 $p(x|t)$ 和原程序后验 $X_{T^*}$ 的分布，并根据 $X_{T^*}$ 的分布对语句排序，见表 17。

表 17 $X_{T^*}$ 的概率分布及语句排序（$N=\frac{1}{676}$）

| | ① | ② | ③ |
|---|---|---|---|
| 排序 | $x_6x_7$ | $x_1x_2x_3x_4x_8x_9x_{10}x_{11}x_{13}$ | $x_5x_{12}$ |
| 概率 | 65N | 52N | 39N |

如果把排序结果和表 7、表 11、表 13 比较，可以看出在"最坏"（即没有挖掘出程序和测试集信息）的情况下，我们的算法得出的结果也和 Dicing 方法、TARANTULA 方法、SAFL 方法相当，并且我们的算法对 $x_2$ 的重视程度（比 TARANTULA 方法稍强）要比 Dicing 方法、SAFL 方法对 $x_2$ 的重视程度高得多。

## 4.2 小型实验

用图 1 中的程序及其几个变体作为实验程序。图 2 是图 1 中程序的流程图。图 1 中的程序有两个错误语句 $x_2$，$x_7$，称之为程序 1。把程序 1 中的错误语句 $x_2$："$m=x$"改为正确语句 $x_2$："$m=z$"，其他语句不变，称这个变体为程序 2。若把程序 1 里的错误语句 $x_7$："$m=y$"改为正确语句 $x_7$："$m=x$"，其余不变，称这个变体为程序 3。无论是哪个程序，都记为程序变量 $X$，且先验分布皆为式(6)，这反映测试人员不知哪个语句有错，只是根据程序样式确定先验概率。

根据软件测试经验，应该设计用例集，使得它们能覆盖程序的每一条路径。为此引入新的测试集：

$$T'=\{e_1,e_2,e_3,e_4,e_5,e_6\} \tag{14}$$

式中，$e_1=\{6,7,9\}$，即输入 $x=6$，$y=7$，$z=9$，它覆盖图 2 中路径①诸语句（下面指的路径皆指图 2 中的路径，不再赘述）；$e_2=\{8,7,9\}$，它覆盖路径②诸语句；$e_3=\{10,7,9\}$，覆盖路径③；$e_4=\{11,10,7\}$，覆盖路径④；$e_5=\{9,10,7\}$，覆盖路径⑤；$e_6=\{8,10,9\}$，覆盖路径⑥。若用 $T'$ 无论对程序 1，2，3 哪一个进行测试，皆认为它们捕获错误的可能性相同，即 $p(e_i)=\frac{1}{6}$，$i=1,2,\cdots,6$。我们认为 $T'$ 是无冗余测试集。

假如向 $T'$ 中增添前述式(7) $T$ 里所有用例，设所得的测试集为 $T''$，它共有 14 个用例，用记号

$$T''=\{e_1,e_2,\cdots,e_6,t_1,t_2,\cdots,t_8\} \tag{15}$$

表示，其中前 6 个用例（即 $e_1,\cdots,e_6$）是 $T'$ 中的用例，后 8 个用例（即 $t_1,\cdots,t_8$）是 $T$ 中用例。我们认为 $T''$ 是冗余测试。根据前面的分析，$e_1-e_6$，$t_1$，$t_2$，$t_4$ 这 9 个用例每个捕获错误的可能性应是其余 5 个用例每个捕获错误可能性的 2 倍，即令：

$$p(e_i)=\frac{2}{23},i=1,2,3,4,5,6;p(t_i)=\frac{2}{23},i=1,2,4$$

$$p(t_i)=\frac{1}{23},i=3,5,6,7,8 \tag{16}$$

显然，$\sum_i p(e_i)+\sum_i p(t_i)=1$。式(16)即为 $T''$ 的先验分布。

分别用 $T'$ 和 $T''$ 测试各个程序，将所得的对语句怀疑度排序的实验结果列表于后。把用 Dicing 方法（以下简称 D 方法）、TARANTULA 方法（以下简称 R 方法）、SAFL 方法（简称 S 方法）所做的实验结果也列表于后，并和我们的算法（以下简称 H 方法）进行比较。

表 18 各种方法在程序 1 的两种测试情况下对可能错误语句的排序

| | |
|---|---|
| H 方法 | $T'$ 测试：$x_2$；$x_7$；$x_5x_{10}x_{12}$；$x_6$；$x_4$；$x_1x_3x_{11}x_{13}$；$x_8x_9$ |
| | $T''$ 测试：$x_2$；$x_7$；$x_{10}$；$x_5x_{12}$；$x_6$；$x_{11}$；$x_4$；$x_8x_9$；$x_1x_3x_{13}$ |
| D 方法 | $T'$ 测试：$x_6$；$x_4$；$x_7$；$x_{11}$；$x_8x_9$；其余语句概率为 0 排在最后 |
| | $T''$ 测试：$x_6$；$x_4x_7x_{11}$；$x_8x_9$；其余语句概率为 0 排在最后 |
| R 方法 | $T'$ 测试：$x_6x_7$；$x_4$；$x_1x_2x_3x_{11}x_{13}$；$x_8x_9$；$x_5x_{10}x_{12}$ 概率为 0 |
| | $T''$ 测试：$x_6$；$x_7x_{11}$；$x_4$；$x_1x_2x_3x_{13}$；$x_8x_9$；$x_5x_{10}x_{12}$ 概率为 0 |
| S 方法 | $T'$ 测试：$x_6x_7$；$x_{11}$；$x_4$；$x_8x_9$；$x_1x_2x_3x_{13}$；其余语句概率为 0 |
| | $T''$ 测试：$x_6x_7$；$x_{11}$；$x_4$；$x_8x_9$；$x_1x_2x_3x_{13}$；其余语句概率为 0 |

由表 18 可以看出：无论是 $T'$ 测试或是 $T''$ 测试，对于程序

1，H 方法都把有错语句 $x_2$，$x_7$ 找出，并把它们排在首要前两位，且不与其他语句混杂。就 $x_2$，$x_7$ 的排列位置而言，冗余测试对 H 方法也没有什么伤害。

表 19　各种方法在程序 2 两种测试情况下对可能错误语句的排序

| | |
|---|---|
| H 方法 | $T'$测试：$x_7$；$x_2$；$x_5$ $x_{10}$ $x_{12}$；$x_6$；$x_4$；$x_{11}$；$x_8$ $x_9$；$x_1$ $x_3$ $x_{13}$ |
| | $T''$测试：$x_7$；$x_2$；$x_{10}$；$x_5$ $x_{12}$；$x_6$；$x_4$；$x_{11}$；$x_8$ $x_9$；$x_1$ $x_3$ $x_{13}$ |
| D 方法 | $T'$测试：$x_7$；$x_6$；$x_4$；其余语句概率为 0 |
| | $T''$测试：$x_7$；$x_6$；$x_4$；其余语句概率为 0 |
| R 方法 | $T'$测试：$x_7$；$x_6$；$x_4$；$x_1$ $x_2$ $x_3$ $x_{13}$；其余语句概率为 0 |
| | $T''$测试：$x_7$；$x_6$；$x_4$；$x_1$ $x_2$ $x_3$ $x_{13}$；其余语句概率为 0 |
| S 方法 | $T'$测试：$x_7$；$x_6$；$x_4$；$x_1$ $x_2$ $x_3$ $x_{13}$；其余语句概率为 0 |
| | $T''$测试：$x_7$；$x_6$；$x_4$；$x_1$ $x_2$ $x_3$ $x_{13}$；其余语句概率为 0 |

由表 19 看出：就寻找有错语句 $x_7$ 并把它排在第 1 位而言，上述各种方法无论对哪一个测试功效都是一样的。

表 20　各种方法在程序 3 两种测试情况下对可能错误语句的排序

| | |
|---|---|
| H 方法 | $T'$测试：$x_2$；$x_5$ $x_7$ $x_{10}$ $x_{12}$；$x_6$ $x_{11}$；$x_4$ $x_8$ $x_9$；$x_1$ $x_3$ $x_{13}$ |
| | $T''$测试：$x_2$；$x_{10}$；$x_5$ $x_{12}$；$x_7$；$x_{11}$；$x_8$ $x_9$；$x_6$；$x_4$；$x_1$ $x_3$ $x_{13}$ |
| D 方法 | $T'$测试：$x_6$ $x_{11}$；$x_4$ $x_8$ $x_9$；其余语句概率为 0 |
| | $T''$测试：$x_{11}$；$x_8$ $x_9$；$x_6$；$x_4$；其余语句概率为 0 |
| R 方法 | $T'$测试：$x_6$ $x_{11}$；$x_1$ $x_2$ $x_3$ $x_4$ $x_8$ $x_9$ $x_{13}$；其余语句概率为 0 |
| | $T''$测试：$x_{11}$；$x_8$ $x_9$；$x_1$ $x_2$ $x_3$ $x_{13}$；$x_6$；$x_4$；其余语句概率为 0 |
| S 方法 | $T'$测试：$x_{11}$；$x_6$；$x_8$ $x_9$；$x_4$；$x_1$ $x_2$ $x_3$ $x_{13}$；其余语句概率为 0 |
| | $T''$测试：$x_{11}$；$x_6$；$x_8$ $x_9$；$x_4$；$x_1$ $x_2$ $x_3$ $x_{13}$；其余语句概率为 0 |

由表 20 可知，对于程序 3，无论是用冗余测试 $T''$或是“精心”设计的无冗余测试集 $T'$，D 方法、R 方法、S 方法的效果并不好。D 方法把 $x_2$ 语句的出错概率估计为 0；S 方法虽然把 $x_2$ 排在第 5 位，但却和其它语句混在一起并列为一个等级；R 方法稍微好一点，但也没有给出 $x_2$ 多少明确暗示；随机 H 方法明确指出，$x_2$ 是错误语句的可能性最大，无论是 $T'$或 $T''$测试，都把它孤独地排在第 1 位。

总结：1. 对于有隐蔽错误的程序，或者无论是通过或失败用例都执行的错误语句，用 D 方法、R 方法、S 方法都无法有效地帮助开发人员寻找错误语句，甚至还会引起副作用，把开发人员的精力浪费在无错语句的鉴定上，因为这些无错语句都排位在前。这通过表 18，特别是表 20 可以明确看出。2. 对于无隐蔽错误且错误语句较少的程序，或是先验概率退化为均匀分布时，H 方法和 D 方法、R 方法、S 方法功效相当，前者由表 19 看出，后者由 4.1.2 节看出。

**结束语**　在随机理论的框架上，我们主要做了两项工作：(1)运用随机变量概念开发一个 TBFL 方法新类型，我们推荐的随机 TBFL 方法便是这一类型方法中的一个。通过实例程序 1 及其各种变体和在其测试集(有冗余和无冗余)上进行验证，该方法的效果都很好，尤其表现在它能正确引导开发人员寻找错误语句的位置以及揭示程序的隐蔽错误。(2)该类型方法可以作为一个框架。我们指出怎样把一个具体 TBFL 方法纳入随机分析框架中去，从而在语句出错概率上的排序分析这个 TBFL 方法的功能，进一步得出有关这个方法有意义的结论。

应该说，我们已经达到了引言里设定的目标。今后的工作是研究随机 TBFL 方法能否揭示程序中的更深层次的隐蔽错误，因为传统的 TBFL 方法有时会对程序员产生误导，对程序里隐蔽错误的揭露“无能为力”。我们认为，这个缺陷比相似性引起的问题更为严重。由此，研究如何避免误导以及如何揭露深层程序错误是该模型的一个重要发展方向。另一项工作是寻找一个新的度量标准，虽然本文已指出如何将其它的 TBFL 方法纳入随机分析模型进行比较，但我们希望能够找到更“一般化”的标准，即使它的适用范围更广泛，让它能成为 TBFL 方法功能判定的一个原则性框架。

更进一步，在第 2 节中提出的随机 TBFL 算法框架中，前两点可以分别开发两个软件模块。

关于第 1 点，根据软件测试理论和实践，程序代码在以下诸范畴是容易出错的：数据声明、数据引用、数值计算、数值比较、子程序参数、输入输出、控制和循环、公式和等式等等。可以深入研究这些出错类型以及可能性，在此基础上，开发一个软件，起名为“程序语句出错概率软件”，这样可以使得输入任意一个程序后，该软件输出这个程序各个语句出错概率。

关于第 2 点，根据测试用例编写的理论和实践，测试用例主要表现以下诸范式：测试边界条件、测试次边界条件、路径覆盖、条件覆盖、逻辑流程、功能和互操作影响等等。可以深入研究这些范式及其在这些范式下测试用例各类型捕捉语句错误的可能性，在此基础上，也开发一个软件，起名为“测试用例能力软件”，这样当输入任意一个测试用例集合时，该软件能输出各个测试用例捕捉程序错误的概率。

至于算法第 3 点，它涉及实际测试过程。通过测试，将测试用例集分为通过和失败两类。下面的算法步骤就是根据这动态结果作出的。

最精妙的是第 4 点和第 5 点。关于第 6 点，它把原程序的语句错误概率和实际结果(即该用例是通过或失败以及其覆盖语句情况)结合在一起考察，这是在微观层次上考察每一个语句和每一个用例实施情况的相互作用，用 $p(x|t)$表示该考察的结果。换言之，$p(x|t)$在微观上浓缩了测试实质。紧接着，第 5 点又从全局考察测试后人们对原程序各语句错误的重认识，即 $\forall x, P(X_T=x)=\sum_t p(t)\cdot p(x|t)$。因为各个测试用例的“能力”有差异，所以对任意语句 $x$ 应该按测试用例捕捉错误的能力进行加权，把诸微观结果 $p(x|\cdot)$综合起来形成原程序的后验概率，继而便顺理成章有了第 6 点。

第 6 点，按后验概率 $X_T$ 对程序语句犯错可能性排序，并把此排序作为开发人员修订错误的参考。同样，关于第 4—第 6 点的计算也可以开发一个计算软件来实现自动化。至于第 3 点，现在已经有了测试工作自动化方面的工具。这样，本算法将来可以自动实施。

## 参 考 文 献


[1] Agrawal H, Horgan J, London S, et al. Fault location using execution slices and dataflow tests [C]// Proceedings of sixth International Symposium on IEEE Software Reliability Engineering. 1995: 143-151

[2] Cleve H, Zeller A. Locating causes of program failures [C]// Proceedings of the 27th International Conference on Software Engineering. 2005: 342-351

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


(下转第 18 页)

OL]. http://www.homes.doc.ic.ac.uk/~hfl/phd/papers/to-read/WSFL.pdf

[8] Arkin A. Business Process Modeling Language (BPML) [EB/OL]. Working Draft 0.4, BPMI, March 2001 (cf. http://www.bpmi.org/)

[9] Weske M, Vossen G, Puhlmann F. Workflow and Service Composition Languages[M]// Bernus P, Mertins K, Schmidt G. Handbook on Architectures of Information Systemst. Springer, 2006:369-390

[10] Business Process Model and Notation (BPMN) Version 2.0 [EB/OL]. (visited 5th October 2010). http://www.omg.org/spec/BPMN/2.0/Beta2/PDF

[11] Sun S X, Zhao J L, Nunamaker J F, et al. Formulating the Data Flow Perspective for Business Process Management[J]. Information Systems Research, 2006, 17(4):374-391

[12] Sundari M H, Sen A K, Bagchi A. Detecting Data Flow Errors in Workfows: A Systematic Graph Traversal Approach[C]//17th Annual Workshop on Information Technology & Systems (WITS-2007). 2007

[13] Trčka N, van der Aalst W, Sidorova N. Analyzing Control-Flow and Data-Flow in Workflow Processes in a Unified Way[R]. CS 08/31. Eindhoven University of Technology, 2008

[14] Trčka N, van der Aalst W, Sidorova N. Dataflow anti-patterns: Discovering dataflow errors in workflows[C]// Conference on Advanced Information Systems Engineering. 2009:425-439

[15] Yang Xue-hong, Huang Jun-fei, Gong Yun-zhan. Static Data Flow Analysis and Anomalies Detection for BPEL[C]// International Conference on Test and Measurement. 2009:18-21

[16] Zheng Yong-yan, Zhou Jiong, Krause P. Analysis of BPEL Data Dependencies[C]// Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference. 2007:351-358

[17] Liu D, Law Kincho H, Wiederhold Gio. Analysis of Integration Models of Service Composition[C]// Proceedings of Third International Workshop on Software and Performance. 2002:158-165

[18] Nanda M G, Chandra S, Sarkar V. Decentralizing execution of composite web services[C]// Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. 2004:170-187

[19] Binder W, Constantinescu I, Faltings B. Decentralized Ochestration of Composite Web Services[C]// Proccedings of the International Conference on Web Services, ICWS'06. 2006:869-876

[20] Pandey S, Buyya R. Scheduling and management techniques for data-intensive application workflows[C]// Data Intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management. 2009

[21] Mell P, Grance T. The NIST Definition of Cloud Computing [OL]. http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf

[22] Yuan Dong. Data Management in Scientific Cloud Workflow Systems[C]// The first CS3 PHD Symposium. 2010:58-61

[23] Deelman E, Chervenak A. Data management challenges of data-intensive scientific workflows[C]// IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08). 2008:687-692

[24] Chervenak A, Deelman E, Livny M, et al. Data Placement for Scientific Applications in Distributed Environments[C]// 8th Grid Computing Conference. 2007:267-274

[25] Habich D, Richly S, Grasselt M, et al. BPELDT- data-aware extension of BPEL to support data-intensive service applications [C]// Emerging Web Services Technology. vol. II, 2008:111-128

[26] Habich D, Preissler S, Lehner W, et al. Data-grey-box web services in data centric environments[C]// Proceedings of the 2007 International Conference on Web Services (ICWS 2007). 2007:976-983

[27] Park S M, Humphrey M. Data Throttling for Data-Intensive Workflows[C]// IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008). 2008:14-18

[28] Yuan D, Yang Y, Liu X, et al. A cost-effective strategy for intermediate data storage in scientific cloud workflows[C]// 24th IEEE International Parallel & Distributed Processing Symposium. 2010:1-12

[29] Yuan D, Yang Y, Liu X, et al. A data dependency based strategy for intermediate data storage in scientific cloud workflow systems[J]. Concurrency and Computation: Practice and Experience, 2010

[30] Yuan D, Yang Y, Liu X, et al. A data placement strategy in scientific cloud workflows[J]. Future Generation Computer Systems, 2010, 26(8):1200-1214

(上接第 13 页)

[4] Kyriazis A, Mathioudakis K. Enhance of fault localization using probabilistic fusion with gas path analysis algorithms [J]. Journal of Engineering for Gas Turbines and Power, 2009, 131(5): 51601-51609

[5] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization [C]// Proceeding of the 24th International Conference on Software Engineering. 2002:467-477

[6] 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. 2005:273-282

[7] Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation [C]// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2005:15-16

[8] Schach S R. Object-oriented classical software engineering [M]. Beijing: China Machine Press, 2007:490-193

[9] Liu C, Yan X, Fei L, et al. SOBER: statistical model-based bug localization [C]// Proceedings of the 13the ACM SIGSOFT Symposium on Foundations of Software Engineering. 2005:286-295

[10] Renieris M, Reiss S P. Fault localization with nearest neighbor queries [C]// Proceedings of the 18th International Conference on Automated Software Engineering. 2003:30-39

[11] Zeller A. Isolating cause-effect chains from computer programs [C]// Proceedings of the 10th ACM SIGFOFT Symposium on Foundations of Software Engineering. 2002:1-10

[12] Haykin S. Neural networks-A comprehensive foundation [M]. Beijing: Tsinghua University Press, 2001:484-508