基于谓词执行序列的软件缺陷定位算法

李 伟 郑 征 郝 鹏 高乙超 饶培峰 宫 成
(北京航空航天大学自动化科学与电气工程学院 北京 100191)

摘 要 谓词执行信息收集和利用的程度会直接影响基于谓词的统计学缺陷定位方法(PBSD)的定位效果.文中主要围绕两个问题进行研究:(1)是否可以通过增加谓词的执行信息量来提高算法的定位精度?(2)执行信息量与算法定位精度有什么关系?在此基础上,设计了一种基于谓词执行序列的软件缺陷定位算法,通过引入谓词执行序列增大算法使用的谓词执行信息量.实验表明,增大谓词执行信息量确实可以提高缺陷定位精度,且当程序中谓词执行信息量充足时,定位精度会随信息量的增加不断提高.

关键词 软件缺陷定位;软件调试;统计学调试;谓词;执行序列
中图法分类号 TP311 **DOI号** 10.3724/SP.J.1016.2013.02406

Predicate Execution-Sequence Based Fault Localization Algorithm

LI Wei ZHENG Zheng HAO Peng GAO Yi-Chao RAO Pei-Feng GONG Cheng
(*School of Automation Science and Electrical Engineering, Beihang University, Beijing* 100191)

Abstract For predicates based statistical debugging, the strength of gathering and utilizing execution information of predicates has a strong influence on the accuracy of software fault localization. This paper mainly focuses on two questions: (1) Whether the accuracy of fault localization can be improved by enriching the predicate execution information? (2) How to evaluate the relationship between the accuracy of fault localization and the augment of predicate execution information? Based on the researches related with the two questions, a new statistical method, called Pesla, is proposed, which introduces the predicate execution-sequences to enrich the predicate execution information. Experimental results clearly demonstrate that more predicate execution information can indeed improve the accuracy of the fault localization. Furthermore, if the predicate execution information of target program meets the algorithm's demand, the higher accuracy of the fault localization will be achieved as the more predicate execution information is used.

Keywords software fault localization; software debugging; statistical debugging; predicate; execution-sequence

1 引 言

软件缺陷定位是指当软件发生失效时,程序员通过分析程序源代码,找到引发该失效的缺陷代码的行为[1].传统的软件缺陷定位多是利用交互式的调试工具人工完成,调试效率低,很难满足和适应大型复杂软件的调试需求,因此,相关研究人员一直致力于研究自动化的软件缺陷定位方法.近年来,一类基于谓词的统计学缺陷定位方法[2-7](Predicates

收稿日期:2011-08-31;最终修改稿收到日期:2012-10-22.本课题得到国家自然科学基金(60904066,91018001)资助.李 伟,男,1986年生,硕士,主要研究方向为软件测试和缺陷定位.E-mail:liwei20048004021@yahoo.cn.郑 征(通信作者),男,1980年生,博士,副教授,主要研究方向为智能决策、软件可靠性与测试.E-mail:zhengz@buaa.edu.cn.郝 鹏,男,1986年生,硕士研究生,主要研究方向为软件测试和缺陷定位.高乙超,男,1989年生,硕士研究生,主要研究方向为软件测试和缺陷定位.饶培峰,男,1987年生,硕士研究生,主要研究方向为软件测试和缺陷定位.宫 成,男,1987年生,硕士研究生,主要研究方向为软件测试和缺陷定位.

Based Statistical Debugging，PBSD)相继被提出，该类方法通过收集和分析程序运行过程中谓词的执行信息来确定程序中谓词与缺陷的相关程度. 具体过程为：(1)通过对谓词进行插桩，得到程序运行过程中谓词的执行信息. 虽然程序任何一次单独运行所输出的谓词执行信息不一定能反映完整的程序执行特征，但是通过多次运行可以收集到谓词的多组执行信息，将这些信息进行统计学分析可以得到谓词的一个较完整执行特征. (2)将谓词在程序运行成功与运行失败时所得到的执行特征进行一致性比较，获得谓词与缺陷的关联度.

上述过程表明，作为从程序中抽取的特征，谓词执行信息的类型和信息量大小是这类方法的核心和基础. 因此，本文针对现有 PBSD 方法所利用的执行信息进行分析，发现关于谓词执行信息收集存在的不足：现有方法忽略了对于谓词判断顺序这一重要信息的收集，导致其对某些缺陷的定位精度不够准确. 据此，本文围绕以下两个问题展开研究：

问题 1：是否可以通过增加谓词的执行信息量来提高算法的定位精度？

问题 2：谓词执行信息量的增加与算法的定位精度之间有什么关系？

我们首先考虑增加算法中谓词的执行信息量，将谓词判断顺序这一有用信息加入到算法的设计中，提出了一种基于谓词执行序列的软件缺陷定位算法(Predicate Execution-Sequence Based Fault Localization Algorithm，简称 Pesla 算法)，并通过进一步增大谓词执行信息量对算法进行扩展，得到 Nbit-Pesla 算法. 实验表明，通过增加谓词的执行信息量的确可以有效提高算法的定位精度，但同时也发现，单纯增加算法中使用的谓词执行信息量并不一定会提高算法的定位精度，因为算法的定位精度还与待测程序有关. 当待测程序中所蕴含的信息可以满足算法的需求时，算法的定位精度会随着执行信息量的增加不断提高，相反则会下降. 基于该思想，本文最后讨论了如何针对不同程序的执行信息为 Nbit-Pesla 算法选取合适 N 值的问题.

本文第 2 节简单介绍缺陷定位领域的研究现状；第 3 节给出 Pesla 算法的研究动机；第 4 节给出 Pesla 算法的基本思想和详细步骤，并将其扩展为 Nbit-Pesla 算法；第 5 节中将 Pesla 算法与 Nbit-Pesla 算法分别在 Siemens、Space 软件包中进行实验，将实验结果与 CBI 算法[2-3]、SOBER 算法[4]、Mann-Whitney 算法、Wilcoxon 算法以及 F-test、t-test 算法[5]进行比较分析，得出结论；第 6 节对本文进行总结和展望.

2　相关研究

软件缺陷定位的研究自 20 世纪 70 年代起，至今已形成几类比较有代表性的方法. 本文主要介绍其中的两类.

基于程序切片的方法[8]是最早提出的一类方法，该类方法将可能影响同一变量值的相关语句所组成的集合定义为一个切片，通过对比程序运行成功与失败时，同一变量所对应切片包含语句的异同，推测缺陷可能的位置. 根据所使用的不同程序切片，基于切片的方法还可分为基于静态切片的方法[9]、基于动态切片的方法[8,10-12]以及基于执行切片[13]的方法.

另外一类重要的缺陷定位方法是基于程序谱的方法，该类方法最早由 Collofello 和 Cousins 提出[14]，通过插桩，收集程序运行过程中输出的执行信息来定位程序中的缺陷，但是早期的方法仅考虑了程序运行失败时的情况，直到 Tarantula 算法[15]的提出，才加入了程序运行成功时的执行信息. 而根据所使用的程序谱的不同，该类方法可以分为基于可执行语句的方法和基于谓词的方法. 作为一种典型的基于可执行语句的方法，Tarantula 算法利用程序中可执行语句在失败运行时出现的频率来估计其含有缺陷的可能性，在此基础上，学者们又相继提出了另外 33 种基于可执行语句的方法，Naish 等人[16]已对这 33 种方法进行了总结和系统的介绍.

本文所提算法属于基于谓词的方法. CBI 和 SOBER 作为两种代表性算法，在第 3 节论述本文研究动机时会有介绍. Hu 和 Zhang 等人[6]指出 SOBER 所使用的假设检验方法有一个基本前提，即谓词的“Evaluation Bias”[4]符合正态分布，但是实验表明，相当一部分谓词的“Evaluation Bias”并不符合正态分布，因此，文献[6]将非参数假设检验的方法 Mann-Whitney 检验、Wilcoxon 检验分别应用于缺陷定位[5]，实验表明，非参数假设检验方法具有更好的定位效果. 此外，在 CBI 和 SOBER 的基础上，Zhang 等人[7]还提出了一种基于谓词“evaluation sequences”的改良方法，同样提高了对部分缺陷的定位精度.

近年来，将软件缺陷定位技术与机器学习、数据挖掘等领域相关理论融合逐渐成为该方向研究的新热点，一系列算法被相继提出：Wong 等人[17]提出了

基于 BP 神经网络的缺陷定位算法；而 Briand 等人[18]则提出使用 C4.5 决策树算法生成规则对测试用例进行分类进而达到缺陷定位的目的；另外还有一些基于机器学习的算法在文献[19]中有较详细介绍.

3 研究动机

本节对 CBI 算法[2-3]和 SOBER 算法[4]中谓词执行信息的收集和使用进行分析比较，以一个实例说明本文的研究动机.

Liblit 等人[2-3]首次在缺陷定位方法中引入针对谓词的插桩，使用谓词执行信息构建 CBI 算法. 该算法通过比较程序在运行成功与运行失败时谓词是否被判断为“真”来计算各谓词与缺陷的关联程度，实验结果表明程序中的缺陷确实与谓词的执行有关，这也证明了基于谓词进行缺陷定位这种思想的正确性和可行性.

鉴于 CBI 算法的良好定位效果，Liu 等人[4]提出了 SOBER 算法，该算法继承了 CBI 算法的基本思想，同时指出 CBI 算法中谓词执行信息的收集具有局限性：程序运行一次，一个谓词只执行一次，这种情况与绝大多数程序中谓词的执行情况是不相符的，实际上，大多数谓词在程序一次运行的过程中被多次执行. 据此，SOBER 算法在 CBI 算法的基础上细化了对谓词执行信息的收集，用程序一次运行中谓词被判断为“真”和为“假”的次数代替是否被执行为“真”的判断，并提出“Evaluation Bias”的定义(如式(1)所示，该定义用来衡量谓词 P 在程序一次运行中被判断为“真”的概率，其中，$\Phi(P)$ 表示“Evaluation Bias”；$n_t(P)$ 表示程序运行一次，谓词 P 被判断为“真”的次数；$n_f(P)$ 表示程序运行一次，谓词 P 被判断为“假”的次数). 根据该定义，Liu 利用假设检验的方法构建出计算谓词与缺陷相关度的公式(如式(2)所示，其中，$S(P)$ 表示谓词与缺陷的相关程度；$f_s(P)$ 表示在所有成功测试用例中，P 被判断为“真”的概率；$f_f(P)$ 表示在所有失败测试用例中，P 被判断为“真”的概率；Sim 表示 $f_s(P)$ 与 $f_f(P)$ 的相似程度，$f_s(P)$、$f_f(P)$ 和 Sim 的具体计算过程详见文献[4]).

$$\Phi(P)=n_t(P)/(n_t(P)+n_f(P)) \quad (1)$$

$$S(P)=-\log(Sim(f_s(P),f_f(P))) \quad (2)$$

SOBER 算法首先使用式(1)分别计算出程序在执行一个测试用例，运行成功或失败时，谓词 P 被判断为“真”的概率 $\Phi_s(P)$ 或 $\Phi_f(P)$，运行全部测试用例后统计得到 $f_s(P)$ 和 $f_f(P)$；最后利用式(2)计算出 P 与缺陷的相关程度 $S(P)$.

实验表明，由于 SOBER 算法加入了谓词执行次数的比较，其定位效果在一些程序上的表现比 CBI 更好.

然而，通过分析可以发现 SOBER 算法依然存在不足. 图 1 是 Replace 程序中第 26 个版本的部分代码. 图中第 8 行为缺陷代码，第 9 行为其正确版本. 可以看出，该缺陷属于逻辑错误，即使经验丰富的程序员也需要一步步调试才可能定位. 我们首先尝试用 SOBER 算法对其定位：加载该程序两个测试用例，一个运行成功，一个运行失败，缺陷所在谓词 P 的执行情况如表 1 所示，P 判断为“真”记为“1”，判断为“假”记为“0”.

```
1 if(X45_handle_cond_expr(96, (lin[*i]!=10)))
2 (
3    advance = 1;
4
5 )
6    break;
7    case '$':
8 if(X45_handle_cond_expr(98, (lin[*i]<=10)))
9 /*if(X45_handle_cond_expr(98, (lin[*i] == 10)))*/
10 (
11   advance = 0;
12
13   )
14   break;
15   case '[':
```

图 1 Replace 程序包中 v26 版本的缺陷代码

表 1 P 在两个测试用例中的执行情况

谓词 P 的执行情况	程序运行结果
10011100	成功
10001101	失败

将这两次运行所输出的谓词执行信息代入式(1)，得到如下结果：

$$\Phi_s(P)=\Phi_f(P)=4/8=0.5.$$

在这两次运行中，P 被判断为“真”的概率是相同的，而这并不能发现 P 上的缺陷. 为确认这一结果，我们加载该程序所有测试用例，将执行信息分别代入式(1)后进行统计处理得到：

$$f_s(P)=0.4733,\ f_f(P)=0.4762.$$

两者仅仅表现出微小差别，进一步将其代入式(2)得到该谓词可疑度仅为 -0.0013，排在所有谓词第 47 位，考虑到该程序共有 65 个谓词，这意味着需要检测超过 72% 的谓词才有可能发现该缺陷.

通过分析发现，SOBER 算法对该缺陷定位精度低下的原因在于：该算法仅考虑了谓词的执行次数，而忽略了谓词的判断顺序. 在上述两次运行中，谓词判断为“真”和判断为“假”的次数是相同的，但其判断顺序不同，分别为“真假假真真真假假”和“真假假

假真真假真”. SOBER 算法虽然比较了这两次执行中 P 被判断为“真”和“假”的次数，但是忽略了其判断顺序，从而无法找到其中的差别. 对于该程序的其它测试用例，这种情况同样存在，SOBER 算法无法捕捉谓词判断顺序上的差异，从而影响其定位精度.

为解决这一问题，一个很自然的想法是加入谓词的判断顺序. 由此引出本文研究的两个核心问题：

(1) 是否可以通过增加谓词的执行信息量来提高算法的定位精度？

(2) 谓词执行信息量的增加与算法的定位精度之间有什么关系？

为回答这两个问题，在下一节中，本文将引入谓词执行序列的概念，通过定义谓词执行序列向量来表征谓词的判断顺序，从而在计算谓词缺陷关联度时引入谓词的判断顺序.

根据本文 4.1 节描述的定义 2，将表 3 所示信息代入，可得到成功时的 2 位执行序列向量为 $\{2,1,2,2\}$，失败时的 2 位执行序列向量为 $\{2,2,2,1\}$，可以看出这两个 2 位执行序列向量有明显区别，进一步将所有测试用例代入，使用本文 4.2 节的算法 1，可得该谓词的缺陷关联度在所有谓词当中排在了第 2 位，相比 SOBER 算法提高了 45 位.

4　基于谓词执行序列的软件缺陷定位算法

4.1　基本定义

定义 1(谓词执行序列).　设 P_i 是程序中第 i 个桩点处的谓词，程序在第 k 次运行中，P_i 执行 m 次，则其执行序列定义为

$$ES_{P_i}^k=(es_{P_i}^1 es_{P_i}^2\cdots es_{P_i}^m),$$

其中，$es_{P_i}^j$ 表示程序第 k 次运行中第 j 次经过 P_i 时，P_i 的判断结果，若判断为“真”则 $es_{P_i}^j=1$，否则 $es_{P_i}^j=0$.

谓词执行序列是通过在程序中插桩所获得的谓词实际执行信息，表示了程序运行过程中谓词的实际执行情况，包括谓词执行的次数、每次的判断结果以及判断为“真”和“假”的先后顺序.

定义 2(N 位执行序列向量).　设 P_i 是程序中第 i 个桩点处的谓词，程序第 k 次运行，P_i 的 N 位执行序列向量定义为

$$\boldsymbol{E}_{P_i}^{N_k}=(\overbrace{\underbrace{n_{00\cdots00}}_{N},\underbrace{n_{00\cdots01}}_{N},\cdots,\underbrace{n_{11\cdots10}}_{N},\underbrace{n_{11\cdots11}}_{N}}^{2^N}),$$

其中，$n_{a_1a_2\cdots a_N}$ 为执行序列 $ES_{P_i}^k$ 包含子序列 $a_1a_2\cdots a_N$ 的个数，$a_l=0$ 或 $1,l=1,2,\cdots,N$.

由此可得程序第 k 次运行，P_i 的 2 位执行序列向量为 $\boldsymbol{e}_{P_i}^k=(n_{00},n_{01},n_{10},n_{11})$，其中 $n_{a_1a_2}$ 为执行序列 $ES_{P_i}^k$ 包含子序列 a_1a_2 的个数，$a_l=0$ 或 $1,l=1,2$.

N 位执行序列向量是从谓词执行序列中提取出的谓词执行信息，是算法直接操作和使用的谓词执行信息.

定义 3(缺陷关联度).　谓词 P_i 的缺陷关联度定义为程序运行成功时 P_i 的 N 位执行序列向量 $\boldsymbol{E}_{P_{i+}}^N$ 与程序运行失败时 P_i 的 N 位执行序列向量 $\boldsymbol{E}_{P_{i-}}^N$ 的距离，用 2-范数表示：

$$R_{P_i}=\|\boldsymbol{E}_{P_{i+}}^N-\boldsymbol{E}_{P_{i-}}^N\| \tag{3}$$

当 $N=2$ 时该表达式也可表示为

$$R_{P_i}=\|\boldsymbol{e}_{P_{i+}}-\boldsymbol{e}_{P_{i-}}\| \tag{4}$$

其中，$\boldsymbol{e}_{P_{i+}}$ 是程序运行成功时，P_i 的两位执行序列向量；$\boldsymbol{e}_{P_{i-}}$ 是程序运行失败时，P_i 的两位执行序列向量.

4.2　算法描述

本节将详细介绍基于谓词执行序列的缺陷定位算法 Pesla 的基本思想和实现步骤.

Pesla 算法的基本思想是：对于桩点处的谓词 P，比较 P 在程序运行成功与失败时的执行序列，根据定义 3 算出所有桩点处谓词的缺陷关联度，将谓词按照缺陷关联度进行排序. P 的缺陷关联度越大，则认为 P 以及与 P 有数据或控制关联的程序代码含有缺陷的概率越大.

Pesla 算法的实现分为数据收集和数据统计两部分：

(1) 数据收集部分. 传统的数据收集方式是通过在待测程序中插入桩点，然后输入程序的测试用例来实现的. 为公平比较算法的定位效果，本文选择与 SOBER 相同的插桩方式，对 if/while/for 这 3 类分支语句以及函数返回值(return 语句)[2] 中的谓词进行插桩；

(2) 数据统计部分. 对收集到的谓词执行信息进行统计处理是 Pesla 算法的核心. 收集到的谓词执行信息分为两类，一类是程序运行成功时的谓词执行序列集 $INFO_s$；另一类是程序运行失败时的谓词执行序列集 $INFO_f$(s 和 f 分别表示成功测试用例和失败测试用例的个数). 分别从 $INFO_s$ 和 $INFO_f$ 中统计出谓词 P_i 的 2 位执行序列向量 $\boldsymbol{e}_{P_{i+}}$ 和 $\boldsymbol{e}_{P_{i-}}$，代入式(4)即可得到 P_i 的缺陷关联度. 计算出所有桩点处谓词的缺陷关联度后，将谓词按照缺陷关联度大小排序即可得到输出 $Report$. 详细描述

见 Pesla 算法.

算法 1. Pesla 算法.

输入：$INFO_s$，$INFO_f$

输出：$Report$：最终的谓词排序结果

符号：$Info_k$：程序第 k 次运行，待测程序中所有桩点处的谓词执行序列的集合；

$ES^k_{P_i}=(es^1_{P_i}es^2_{P_i}\cdots es^m_{P_i})$：程序第 k 次运行，P_i 的执行序列；m 表示 P_i 共执行的次数；

$stat(ES^k_{P_i})$：用于统计得到 P_i 的 2 位执行序列向量的函数；

$Compositor()$：根据缺陷关联度大小对谓词进行排序的函数

1. For all $P_i\in P$ do
2. For all $Infok\in INFOs$ do
3. If $ES^k_{P_i}\in Info_k$ then
4. $\boldsymbol{e}^k_{P_i}\leftarrow stat(ES^k_P)$
5. Endif
6. Endfor
7. For all $Info_k\in INFO_f$ do
8. If $ES^k_{P_i}\in Info_k$ then
9. $\boldsymbol{e}^k_{P_i}\leftarrow stat(ES^k_{P_i})$
10. Endif
11. Endfor
12. $\boldsymbol{e}_{P_{i+}}\leftarrow\sum_{k=1}^{s}\boldsymbol{e}^k_{P_i}/s$
13. $\boldsymbol{e}_{P_{i-}}\leftarrow\sum_{k=1}^{f}\boldsymbol{e}^k_{P_i}/f$
14. $R_{P_i}\leftarrow\|\boldsymbol{e}_{P_{i+}}-\boldsymbol{e}_{P_{i-}}\|$
15. Endfor
16. $Compositor()$

值得注意的是对于 $stat(ES^k_{P_i})$ 函数，有 3 种情形需要讨论：

情形 1. $ES^k_{P_i}=\varnothing$，即 P_i 在程序第 k 次运行过程中没有被执行，则 $\boldsymbol{e}^k_{P_i}=(0,0,0,0)$.

情形 2. $ES^k_{P_i}=(es^1_{P_i})$，即 P_i 在程序第 k 次运行过程中仅执行了一次，若 $es^1_{P_i}=1$，则 $\boldsymbol{e}^k_{P_i}=(0,0,0.5,0.5)$；否则 $\boldsymbol{e}^k_{P_i}=(0.5,0.5,0,0)$.

情形 3. $ES^k_{P_i}=(es^1_{P_i}es^2_{P_i}\cdots es^m_{P_i})$，其中 $m\geq 2$，即 P_i 在程序第 k 次运行过程中执行 m 次，则分别统计 n_{00}、n_{01}、n_{10}、n_{11} 的次数分别为 $\boldsymbol{e}^k_{P_i}$ 的元素赋值，最后 $\boldsymbol{e}^k_{P_i}\leftarrow\boldsymbol{e}^k_{P_i}/(m-1)$.

对于情形 2，本文采用等概率处理的方法：若谓词在程序一次运行中只执行了一次，则假设第 2 次执行判断为“真”和为“假”的概率各为 1/2，然后再对 $\boldsymbol{e}^k_{P_i}$ 中的元素赋值.

4.3 算法扩展

Pesla 算法是基于谓词的 2 位执行序列向量，这种算法同样可以扩展到 3 位、4 位以及更高位的执行序列向量，以增加更多执行信息的比较，但是这种执行信息的增加可能带来两种新的情况：一种是由于信息更加丰富，使算法定位精度提高；另一种是由于信息存在冗余，使算法定位精度下降. 这就引出了前面提到的第 2 个问题，即谓词执行信息量的增加与算法的定位精度之间存在什么关系？

为回答该问题，本节首先将对 Pesla 算法进行扩展，使其基于更大的谓词信息量进行计算，并且对扩展后的算法进行时间复杂度分析. 后面的实验部分对该问题进行了回答.

扩展后的算法命名为 Nbit-Pesla 算法（Nbit 表示算法使用 N 位执行序列向量），其基本思想与 Pesla 算法基本一致. 两者的不同说明如下.

(1) 对于函数 $stat(ES^k_{P_i})$，原算法因为只用到了 2 位执行序列向量，所以只需分为 3 种情形进行讨论；在扩展后，由于所需向量位数增加，谓词执行次数小于向量位数的情形也相应增加，对这种情况，我们一律采用 4.2 节用到的等概率补齐的方法对 N 位执行序列向量的元素赋值，详细描述见 Nbit-Pesla 子算法.

算法 2. Nbit-Pesla 子算法.

$stat(ES^k_{P_i})$：Nbit-Pesla 算法中，用于统计得到 P_i 的 N 位执行序列向量的函数

输入：$ES^k_{P_i}=(es^1_{P_i},es^2_{P_i},\cdots,es^m_{P_i})$：程序第 k 次运行，P_i 的执行序列；m 表示 P_i 共执行的次数

输出：$\boldsymbol{E}^{N_k}_{P_i}$：$N$ 位执行序列向量

符号：$a_1a_2\cdots a_N$：同定义 2 中的 $a_1a_2\cdots a_N$

1. If $m=0$ then /* P_i 没有被执行 */
2. $\boldsymbol{E}^{N_k}_{P_i}=(0,0,\cdots,0)$
3. Else if $1\leq m<N$ /* P_i 执行次数小于 N 次 */
4. $n_{es^1_{P_i}es^2_{P_i}\cdots es^m_{P_i}b^1b^2\cdots b^{N-m}}\leftarrow 1/2^{N-m}$ /* $b^l=0$ 或 1，$l=1,2,\cdots,N-m$ */
5. $\boldsymbol{E}^{N_k}_{P_i}$ 中其它元素赋值为 0
6. Else if $m\geq N$ /* P_i 执行次数大于等于 N 次 */
7. For all $j\in[1,m-N+1]$ do
8. If $es^j_{P_i}es^{j+1}_{P_i}\cdots es^{j+N-1}_{P_i}=a_1a_2\cdots a_N$
9. $n_{a_1a_2\cdots a_N}\leftarrow n_{a_1a_2\cdots a_N}+1$
10. Endif
11. Endfor
12. $\boldsymbol{E}^{N_k}_{P_i}\leftarrow\boldsymbol{E}^{N_k}_{P_i}/(m-N-1)$
13. Endif

(2) 计算缺陷关联度的函数由式(4)改为了式(3).

Nbit-Pesla 算法的基本思想表明，若 N 增大，算法所需处理的谓词执行信息量就会增加，并导致算法的计算复杂度提升. 设待测程序包含 t 个桩点，s 个运

行成功的测试用例，f 个运行失败的测试用例，那么对于 N bit-Pesla 算法，记录所有执行序列向量需要 $t\times(s+f)\times 2^N$ 个存储单元，其空间复杂度为 $O(t\times(s+f)\times 2^N)$. 计算谓词缺陷关联度的过程需要做 $(2^N\times(s-1)+2^N\times(f-1)+1)\times t$ 次加法，2×2^N 次除法和乘法运算，则 N bit-Pesla 算法的时间复杂度为 $O(t\times(s+f)\times 2^N)$，对于同一待测程序，若 t、s、f 都不变，则算法的空间和时间复杂度会随 N 的增长呈级数增长. 若待测程序规模很大且测试用例较多，这种增长速度是不可接受的. 我们在选择算法时，不仅要考虑算法的定位精度，还需要权衡所付出的代价. 因此，有必要研究算法定位精度与谓词执行信息量之间的规律，只有探明这其中的规律，才能合理选择算法以达到定位精度与计算代价之间的平衡.

5　实验及结果分析

本文实验由 3 部分组成：(1) Pesla 算法与现有算法的比较，包括与 CBI、SOBER、Mann-Whitney、Wilcoxon、F-test、t-test 各种基于谓词算法的比较（其中 SOBER、Mann-Whitney、Wilcoxon、F-test、t-test 算法都是基于“Evaluation Bias”所提出的方法）；(2) 当 N 取不同值时，各 N bit-Pesla 算法之间定位效果的比较；(3) 根据程序执行信息选择 N 的经验公式及其效果验证. 为了更好地描述实验，在 5.1～5.3 节中我们首先对目标程序、评价方法和实验设置进行介绍.

5.1　目标程序

参照之前的研究，本文选取 Siemens 软件包中的全部 7 段程序，和一个应用软件 Space 作为目标程序进行实验. Siemens 软件包中的每个程序分别包含 7～41 个缺陷版本，每个缺陷版本中有一个人工植入的缺陷. 表 2 中列出了目标程序的主要信息. 例如，对于 tcas 程序来说，共有 41 个缺陷版本，每个版本中包含 133～137 行可执行语句，其中有 11 个桩点谓词，1608 个测试用例，平均有 23 个失败的测试用例和 1585 个成功的测试用例.

表 2　目标程序的主要信息

程序	缺陷版本数	可执行语句数	谓词数	测试用例数	失败测试用例/成功测试用例
print-tokens(2 programs)	17	341～345	83	4130	38/4082
schedule(2 programs)	19	261～294	40	2710	32/2678
replace	32	505～518	65	5542	83/5499
tot-info	23	272～274	47	1052	71/981
tcas	41	133～137	11	1608	23/1585
Space	38	6218	914	13496	1997/11499

Space 软件是欧洲航天局开发的一个矩阵描述语言(ADL)解释器，它可以从文件中读取 ADL 描述语句并检查文件内容是否合乎语法规则，如果 ADL 文件正确，Space 将输出一个矩阵数据文件，包含矩阵元素信息、位置、激励，否则将输出错误提示，Space 软件包含 6218 行可执行语句，914 个桩点谓词，13 496 个测试用例，38 个缺陷版本，每个版本中包含一个真正的缺陷，Space 软件是软件缺陷定位研究领域公认的典型测试对象.

5.2　P-score 评价方法

本文选用 P-score[20] 作为比较各算法定位精度的主要评价方法. P-score 由 Zhang 等人[20] 提出，相较 T-score[21]，不需要假设“有一个完美的程序调试员”，且使用更为方便、直接. 下面介绍 P-score 的使用方法.

首先给出 P-score 的计算公式：

$$P\text{-score}=\frac{indexof\underline{P}}{L}\times 100\% \qquad (5)$$

其中，L 为目标程序中的谓词数量，$\underline{P}$ 表示程序中与缺陷最相关的谓词，$indexof\underline{P}$ 是谓词 P 在输出 $Report$ 中的排序序号. 为方便理解，举例如下，假设一个缺陷程序中含有 10 个谓词 $\{P_1,P_2,P_3,\cdots,P_{10}\}$，与缺陷最相关的谓词为 P_8，利用算法得到的 $Report$ 中谓词的排序为 $\{P_3,P_7,P_{10},P_8,\cdots,P_2\}$，$P_8$ 排在第 4 位，则 P-score$=4/10\times 100\%=40\%$.

P-score 可以解释为所需检查的谓词数量越少，则算法对缺陷的定位精度越高.

5.3　实验设置

为公平比较各算法的优劣，本文需要对 4 个影响实验结果的主要因素进行设置：缺陷版本的选择、缺陷最相关谓词的确定、桩点位置的确定和测试用例的选择.

首先，根据 5.1 节中的介绍，Siemens 软件包一共包含 132 个缺陷版本，Space 软件中包含 38 个缺陷版本，但本文的实验并没有使用全部的缺陷版本，而是根据以下两条原则对缺陷版本进行筛选：

(1)对于一个程序的缺陷版本,其测试用例集中既要包含成功的测试用例,又要包含失败的测试用例;

(2)缺陷程序中与缺陷最相关的谓词要能够明确地确定,这是因为程序中与缺陷最相关谓词的确定会对最终的实验结果产生较大影响,若无法清晰地确定该谓词的位置,则将此缺陷版本从实验中剔除.

根据这两条原则,在 Siemens 软件包中共排除掉 21 个缺陷版本,在 Space 软件中排除掉 10 个缺陷版本①.

本文选择文献[5]中使用的最相关谓词选择策略:

第 1 步,确定缺陷位置.若缺陷在可执行语句上,标记该可执行语句,若缺陷为可执行语句的缺失,则标记距离缺失语句最近的下一条可执行语句;

第 2 步,根据第 1 步标记的可执行语句,选择距离该可执行语句最近的谓词作为程序中与缺陷最相关的谓词,如有两个或更多谓词与该可执行语句距离一样近,无法确定唯一的谓词,将此缺陷版本剔除(在实验所选用的 139 个缺陷版本中,缺陷最相关谓词所在语句与标记的可执行语句之间都没有超过 3 行代码).

与以往的研究[2-7]相同,由于本文研究不涉及缺陷定位的效果对测试用例的依赖,因此,在实验中同样是加载程序的全部测试用例,再将所得结果进行比较.在选择桩点位置时,本文选择与 SOBER 相同的设置,仅对程序代码中的分支语句(if、for、while)以及返回值(return)中的谓词[2]进行插桩.

本实验所用计算机的 CPU 是两颗 Xeon-E5630,主频 2.53GHz,内存 12GB.

本文所选用的 Siemens 软件包与 Space 软件全部在 Software-artifact Infrastructure Repository (SIR)网站下载获得,而 Mann-Whitney、Wilcoxon、*F*-test、*t*-test 假设检验算法也是在 ALGLIB 网站下载获得,CBI、SOBER 算法则根据文献[2,4]中的描述实现.

5.4 Pesla 算法与各算法的比较

本节,我们将针对各算法的整体定位效果、不同程序的定位效果以及单个缺陷版本的定位效果 3 个方面分别进行比较,最后用假设检验来证明 Pesla 算法在定位精度上的优越性.

首先,本文给出将各种算法应用在 Siemens 软件包的各缺陷版本后,所得到的定位效果对比,如图 2 所示(其中 Wilcoxon、Mann-Whitney、SOBER、CBI、*F*-test、*t*-test 的数据来源于文献[5]).图中横坐标表示所需检查的谓词占全部谓词的百分比,纵坐标为所发现的缺陷占全部缺陷的百分比.从图 2 中可以看出,在 10 个统计节点处,相对于其它算法,Pesla 算法都表现出了更好的定位效果,尤其是当检查的谓词数量相对较少时,这种优势更为明显,例如,当检查 10%的谓词时,Pesla 可以定位 33.33%的缺陷,其它算法中表现最好的 Wilcoxon 也仅能定位 17.11%,Pesla 定位到的缺陷数量接近 Wilcoxon 的 2 倍;Mann-Whitney 的 5 倍,是其它算法的 7~30 倍,而当检查 30%的谓词时,这种优势虽然有所下降,但 Pesla 仍旧表现出更好的定位效果.

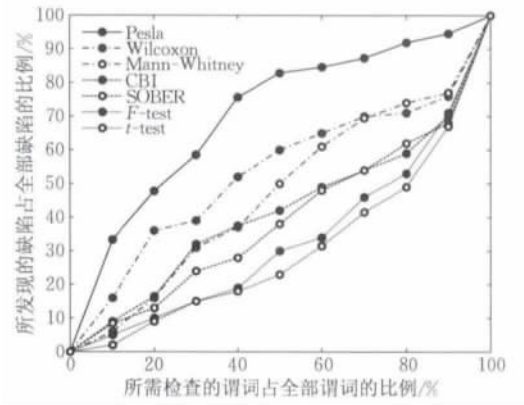

图 2 各算法在 Siemens 软件包上的定位效果对比

图 3 给出了各种算法在各个程序上的定位效果对比,可以看出,在大多数程序上,相比于其它算法,Pesla 都表现出了更高的定位精度,只有在 tcas 程序上,Wilcoxon 比 Pesla 表现得更好.

为了进一步验证 Pesla 算法在定位单个缺陷时同样具有更高的定位精度,本文又对各种算法在定位同一个缺陷时的表现做了比较,如表 3、表 4 所示.

在表 3 中,"Min"表示使用该算法定位一个缺陷所得到的最小 *P*-score;"Max"表示最大的 *P*-score;"Median"表示 *P*-score 的中位数;"Mean"表示 *P*-score 的均值;"Stdev"表示 *P*-score 的标准差.这 5 个指标中,"Median"和"Stdev"体现了算法在定位不同缺陷时所表现出的稳定性,其值越小,表明该算法的稳定性越好.从表 3 中可以看出,在这两个指标中,Pesla 算法都是表现最好的那一个,即在使用 Pesla 算法定位不同缺陷时具有更高的概率,能够获得更高的定位精度.

① 本文中的设置与文献[5]中完全相同,因此所排除的缺陷版本也与其相同.Siemens 软件包中排除的缺陷版本为 schedule2-v9; print_tokens-v4, v5, v6; replace-v12, v27; tcas-v7, v8, v13, v14, v16, v17, v18, v19, v33, v36, v38; tot_info-v6, v10, v19, v21. Space 软件排除的缺陷版本为 v1, v2, v25, v26, v30, v32, v34, v35, v36, v38.

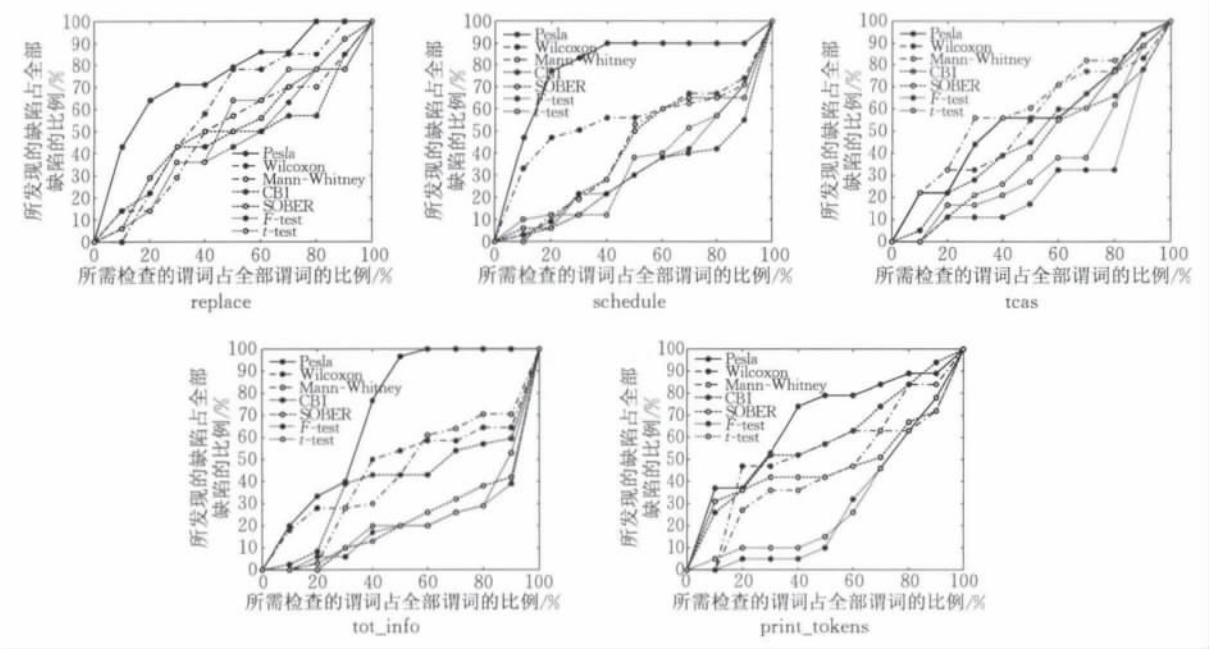

图 3　各种算法在单个程序上的定位效果对比

表 3　各种算法在 Siemens 软件包上定位效果的详细对比

算法	Min/%	Max/%	Median/%	Mean/%	Stdev/%
Pesla	1.59	100.00	24.56	30.45	26.85
Wilcoxon	0.89	100.00	39.82	46.91	35.71
Mann-Whitney	3.45	100.00	50.00	53.38	30.56
SOBER	3.70	100.00	63.06	60.64	32.06
CBI	0.91	100.00	63.64	58.58	34.34
F-test	4.44	100.00	75.86	67.71	29.09
t-test	4.50	100.00	80.30	70.74	28.23

表 4　Pesla 算法对单个缺陷的定位效果对比

比较范围/%	缺陷个数					
	$P\text{-score}_{Pesla}-P\text{-score}_{Wilcoxon}$	$P\text{-score}_{Pesla}-P\text{-score}_{Mann\text{-}Whitney}$	$P\text{-score}_{Pesla}-P\text{-score}_{SOBER}$	$P\text{-score}_{Pesla}-P\text{-score}_{CBI}$	$P\text{-score}_{Pesla}-P\text{-score}_{F\text{-}test}$	$P\text{-score}_{Pesla}-P\text{-score}_{t\text{-}test}$
<-1	61	75	74	85	92	89
$-1\sim1$	4	2	3	4	1	3
>1	46	34	34	22	18	19
<-5	57	71	74	83	91	87
$-5\sim5$	12	12	8	10	3	5
>5	42	28	29	18	17	19
<-10	51	67	71	77	89	82
$-10\sim10$	24	17	12	17	6	11
>10	34	27	28	16	16	18
<-20	46	58	63	68	81	76
$-20\sim20$	21	34	28	32	18	25
>20	42	19	20	11	12	10

表 4 是 Pesla 分别与其它算法进行比较的结果，为方便理解该表，用一个例子来说明：对于第 1 列“$P\text{-score}_{Pesla}-P\text{-score}_{Wilcoxon}$”的“$<-1\%$”，它表示，在 111 个缺陷中，有 61 个缺陷，其 $P\text{-score}_{Pesla}-P\text{-score}_{Wilcoxon}<-1\%$，也可以理解为，Pesla 算法在定位 61 个缺陷时，定位精度比 Wilcoxon 算法高 1%以上. “$>1\%$”则表示，在 111 个缺陷中，有 46 个缺陷，其 $P\text{-score}_{Pesla}-P\text{-score}_{Wilcoxon}>1\%$，可理解为 Wilcoxon 算法在定位 46 个缺陷时，定位精度比 Pesla 算法高 1%以上. 从表 4 中可以看出，相比于其它算法，Pesla 算法在定位绝大多数缺陷时的定位精度更高.

下面我们给出各种算法应用在 Space 软件上的定位效果对比，如表 5、表 6 所示.

表 5 各算法在 Space 上的定位效果对比

谓词检查量/%	缺陷个数						
	Pesla	Wilcoxon	Mann-Whitney	SOBER	CBI	*F*-test	*t*-test
1	15	16	18	10	5	1	1
2	21	21	20	11	5	1	1
5	24	23	22	18	6	5	5
10	26	24	23	18	7	9	9
20	26	26	25	21	8	12	13
50	28	28	28	25	8	25	25
100	28	28	28	28	28	28	28

表 6 各算法在 Space 上的定位效果详细对比

算法	Min/%	Max/%	Median/%	Mean/%	Stdev/%
Pesla	0.11	44.97	0.77	4.70	11.55
Wilcoxon	0.11	46.78	1.55	5.66	15.26
Mann-Whitney	0.11	48.46	1.37	5.57	15.79
SOBER	0.11	95.68	3.11	17.23	31.18
CBI	0.11	96.74	46.00	35.44	45.37
F-test	0.77	95.40	21.17	26.40	26.96
t-test	0.77	94.97	20.68	25.81	26.95

由于 Space 软件较大，其中包含的谓词数量较多，本文侧重比较当检查少量谓词时，各算法所能定位到的缺陷数量. 由表 5 可以看出，只有检验少于 1% 的谓词时，Pesla 定位到的缺陷数量稍少于 Wilcoxon 和 Mann-Whitney，而在其它的统计节点处，Pesla 都是定位到最多缺陷的算法.

我们进一步对各算法在定位 Space 软件单个缺陷时的情况进行比较，从表 6 可以看出，相比于 SOBER、CBI、*F*-test、*t*-test 这 4 种算法，Pesla 算法无论是在定位精度还是定位缺陷的稳定性上都有很大的优势；而相比于 Wilcoxon 和 Mann-Whitney，Pesla 则在定位缺陷的稳定性上表现更好.

综合以上分析，我们可以得到以下结论：Pesla 算法相对于其它算法具有更高的定位精度以及更好的定位稳定性.

为检验以上结论在统计上的显著性，我们使用 *t* 检验来证实该结论，设：

H_0：利用 *P*-score 进行比较时，Pesla 算法与 X 算法的定位精度没有显著差异.

H_1：利用 *P*-score 进行比较时，Pesla 算法与 X 算法的定位精度有显著差异.

表 7 为当 X 为不同算法时，得到的 *t* 检验的检验统计量 *P*-value，如 Pesla 与 Wilcoxon 比较，使用 *t* 检验可以得到 *P*-value<0.00005. 可以看到，所有的 *P*-value 均小于 0.05. 若给定显著性水平 $\alpha=0.05$，则由表 8 中的结果可以得到：在显著性水平 $\alpha=0.05$ 下，拒绝原假设 H_0，认为 Pesla 算法与 X 算法具有显著差异，即使选择显著性水平 $\alpha=0.01$，以上结论仍然成立.

表 7 H_0 假设检验的结果

	P-value					
	Wil	M-W	*t*-test	*F*-test	SOBER	CBI
Pesla	$<5.0\times10^{-5}$	$<6.0\times10^{-7}$	$<2.0\times10^{-17}$	$<2.0\times10^{-15}$	$<4.0\times10^{-12}$	$<2.0\times10^{-8}$

表 8 不同 *N*bit-Pesla 算法的缺陷定位效果比较

比较范围/%	缺陷个数		
	$P\text{-score}_{3\text{bit-Pesla}}-P\text{-score}_{\text{Pesla}}$	$P\text{-score}_{4\text{bit-Pesla}}-P\text{-score}_{3\text{bit-Pesla}}$	$P\text{-score}_{5\text{bit-Pesla}}-P\text{-score}_{4\text{bit-Pesla}}$
$=0$	22	44	52
<-1	40	19	19
$-1\sim1$	22	45	53
>1	49	47	39
<-5	31	9	9
$-5\sim5$	45	72	82
>5	35	30	20
<-10	8	6	2
$-10\sim10$	61	91	102
>10	24	11	7
<-20	1	3	0
$-20\sim20$	96	107	111
>20	10	1	0

因此，我们对于问题 1 的回答是：通过在算法中增加谓词的执行信息量确实可以有效提高算法的定位精度.

5.5 不同 *N*bit-Pesla 算法的比较

上节的实验结果表明 Pesla 算法相比其它算法具有更高的定位精度，这就引出了问题 2：谓词执行信息量的增加与算法的定位精度之间有什么关系？为此，本文按照递增的方式选取 N 值，设 $N=3、4、5$，将 *N*bit-Pesla 算法应用于 Siemens 软件包与 Space 软件（注：当 N 大于等于 6 时，由于算法时间复杂度太高，对于 Space 而言，需要付出的代价太大，因此

不予考虑).

与上节相同,本节同样针对不同 N bit-Pesla 算法的整体定位效果、不同程序的定位效果以及单个缺陷版本的定位效果进行比较和讨论,最后对问题 2 进行回答.

图 4 和图 5 是 N 取不同值时,N bit-Pesla 算法在 Siemens 软件包上的定位效果对比. 可以看出,当检验的谓词数量不变,而 N 增大时,N bit-Pesla 算法并没有定位出更多的缺陷,相反,定位出的缺陷数量呈现出递减的趋势,尤其是当检验的谓词数量少于 20%时,这种趋势表现的更明显.

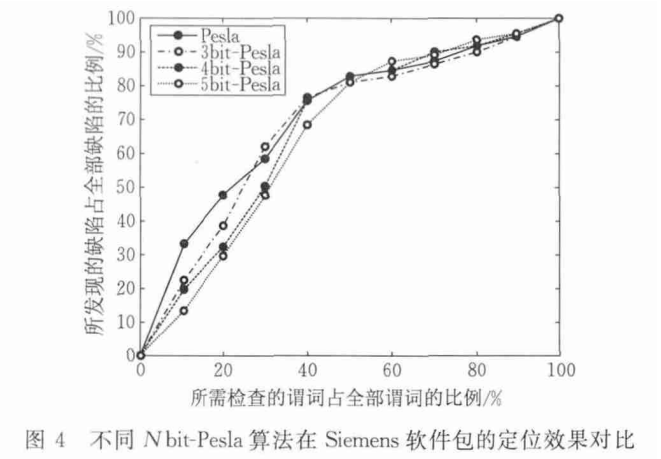

图 4　不同 N bit-Pesla 算法在 Siemens 软件包的定位效果对比

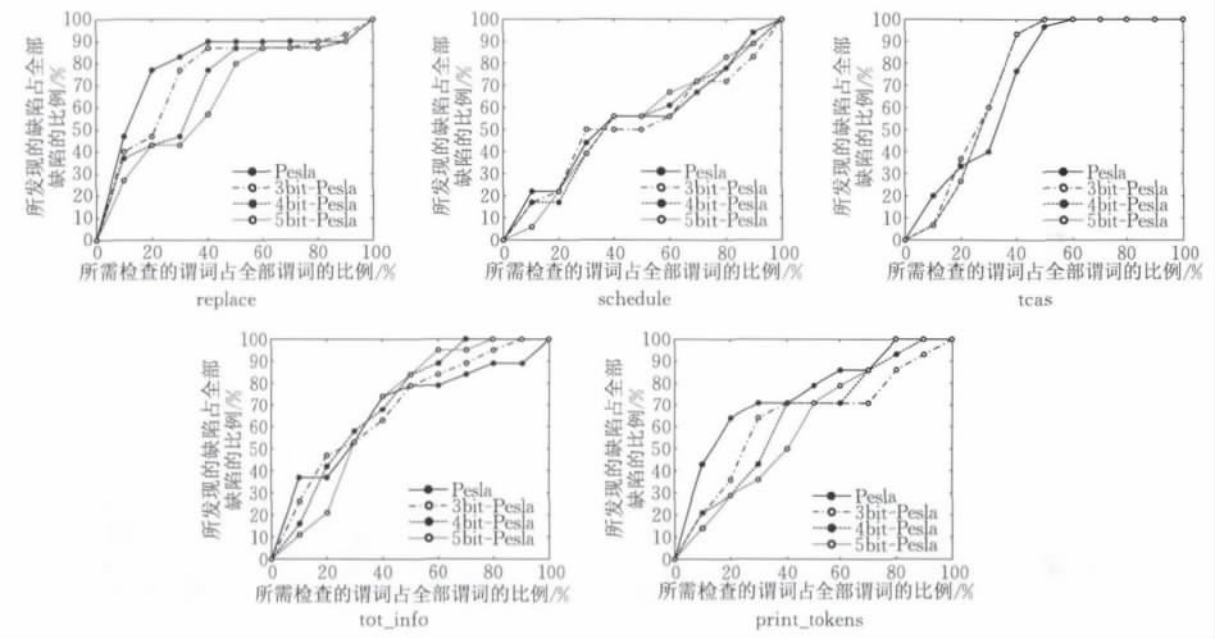

图 5　不同 N bit-Pesla 算法对单个程序的定位效果对比

但是,这并不能证明,随着 N 的增大,N bit-Pesla 对每个缺陷的定位精度都会下降. 因此,我们统计了不同 N bit-Pesla 对单个缺陷的定位结果,如表 8、表 9 所示.

表 9　不同 N bit-Pesla 算法的详细对比

算法	Min/%	Max/%	Median/%	Mean/%	Stdev/%
Pesla	1.59	100.00	24.56	30.45	26.85
3bit-Pesla	1.59	100.00	25.40	32.57	26.16
4bit-Pesla	1.59	100.00	30.00	34.17	24.16
5bit-Pesla	1.59	100.00	31.75	35.53	23.36

从表 9 中可以看出,总体上来说,“Mean”随着N的增大而增大,“Stdev”却越来越小,这表明 N bit-Pesla 算法的平均定位精度有降低的趋势,但针对不同缺陷,定位精度的波动性越来越小.

由表 8,我们可以看出,随着 N 的增大,N bit-Pesla 算法并不是对所有缺陷的定位精度都会降低,还有相当一部分缺陷的定位精度会持续提高,例如,3bit-Pesla 算法对 22 个缺陷的定位精度与 Pesla 算法相同,对 40 个缺陷的定位精度要高于 Pesla 算法,仅对 49 个缺陷的定位精度低于 Pesla 算法;即使 N 增大至 5,仍然有 19 个缺陷的定位精度在提高. 经统计发现,这 19 个缺陷是包含在之前 40 个定位精度提高的缺陷中的,即这 19 个缺陷随着 N 的增大,其定位精度在不断提高.

表 10、表 11 列出了 N bit-Pesla 算法在 Space 软件上对缺陷的定位效果. 与 Siemens 软件包中得出的结论相似,当检验的谓词数量较少,如在 1%、2%时,随着 N 的增大,N bit-Pesla 算法定位到的缺陷数量呈现出了比较明显的下降趋势,且其平均定位精度不断降低.

表 10 不同 Nbit-Pesla 算法在 Space 上定位效果对比

谓词检查量/%	缺陷个数			
	Pesla	3bit-Pesla	4bit-Pesla	5bit-Pesla
1	15	14	11	10
2	21	18	14	13
5	24	23	21	21
10	26	24	25	24
20	26	26	25	25
50	28	28	28	28
100	28	28	28	28

表 11 不同 Nbit-Pesla 算法在 Space 上的详细对比

算法	Min/%	Max/%	Median/%	Mean/%	stdev/%
Pesla	0.11	44.97	0.77	4.70	11.55
3bit-Pesla	0.11	44.75	1.10	5.50	11.34
4bit-Pesla	0.11	44.97	1.70	6.30	11.23
5bit-Pesla	0.11	44.75	2.30	6.60	11.11

但是，与先前观察到的结果一样，并不是对 Space 软件中的所有缺陷，N bit-Pesla 的定位精度都会下降. 我们统计得到，对于 Space 软件 v1、v3、v27 版本中的缺陷，随着 N 增大，其定位精度在不断提高；而对于 v7、v10、v12、v16、v18、v20、v21 版本中的缺陷，其定位精度几乎没有变化.

究其原因，我们发现：一个程序，其缺陷谓词在一个测试用例中的执行次数 m 是固定的，当 $m>N$ 时，采用本文算法对该缺陷进行定位，总能获得较好的定位精度，若 N 增大的同时始终满足 $m>N$，则 Nbit-Pesla 算法对该缺陷的定位精度也会持续提高或保持不变，但一旦出现 $m<N$ 的情形，就需使用 4.3 节 Nbit-Pesla 子算法中所介绍的等概率补齐的方法来对该谓词的 N 位执行序列向量赋值. 若 N 持续增大，所需补齐的位数就会增多，这会导致该缺陷谓词在程序运行成功时得到的 N 位执行序列向量与运行失败时得到的 N 位执行序列向量之间的区分度越来越小，最终造成对该缺陷的定位精度下降.

由 4.3 节分析可知，N bit-Pesla 算法的计算复杂度随 N 的增大呈级数增长，图 6、图 7 分别给出了 N 取不同值时，N bit-Pesla 算法定位 Siemens 和 Space 软件包中缺陷时的时间开销变化. 图中纵坐标表示定位一个缺陷，算法的平均运行时间，单位是秒. 可以看出，随着 N 的增大，算法的运行时间越来越大，与之前的分析一致.

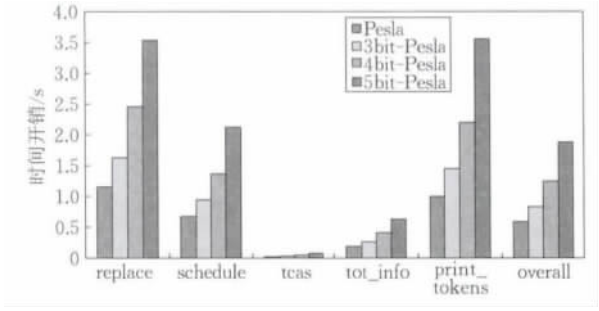

图 6 Nbit-Pesla 算法在 Siemens 上的时间开销

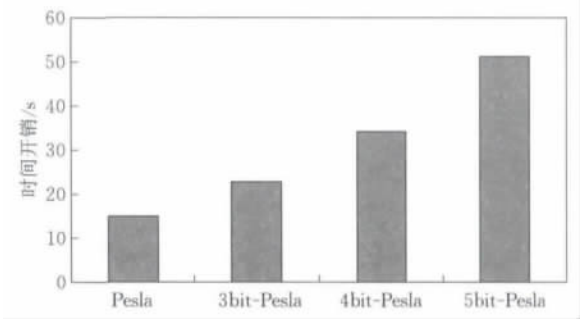

图 7 Nbit-Pesla 算法在 Space 上的时间开销

同样，图 8、图 9 分别给出了 N 取不同值时，Nbit-Pesla 算法定位 Siemens 和 Space 软件包缺陷时的空间开销变化. 图中纵坐标表示定位一个缺陷，算法平均所需占用的内存大小，单位是兆字节. 可以看出，随着 N 的增大，算法所占用的内存空间几乎是严格按照级数增长.

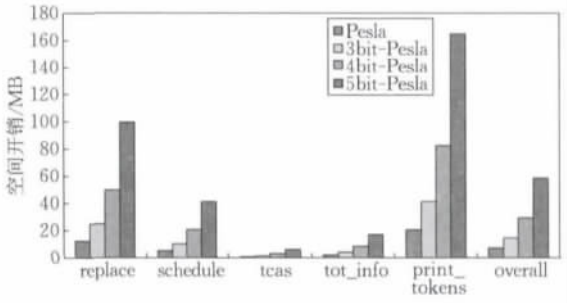

图 8 Nbit-Pesla 算法在 Siemens 上的空间开销

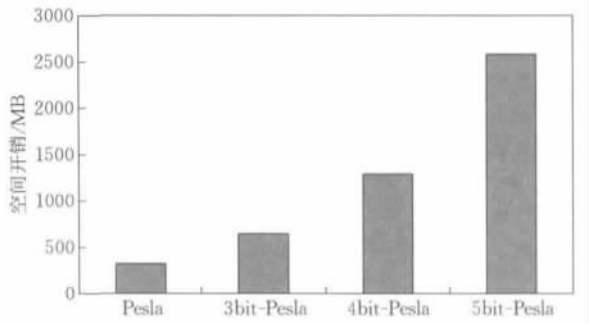

图 9 Nbit-Pesla 算法在 Space 上的空间开销

综合本节实验，可以得到：若待测程序在运行过程中谓词的执行次数较多，所输出的谓词执行信息量可以满足算法需求，增大 N 的取值确实可以有效提高算法的定位精度. 然而，如果无限制地增大 N，不仅计算复杂度呈级数增长，而且对于某些程序，还会因为待测程序信息提供能力有限（由于大部分谓

词的执行次数较少），造成算法定位精度下降. 因此，如何根据待测程序中谓词执行信息的情况来选择适合该程序的 N 是一个重要的问题.

5.6　Nbit-Pesla 算法中 N 的选择

在本节中，通过大量的实验分析和验证，我们给出了根据程序执行信息选择 N 的一种方法，进而对该问题进行初步的探讨.

假设待测程序共有 L 个谓词，P_i 为第 i 个谓词（$1\leq i\leq L$）. 在一次缺陷定位过程中，待测程序共运行了 W 次（即有 W 个测试用例）. $ES^k_{P_i}$ 为程序第 k 次运行中 P_i 的执行序列，$|ES^k_{P_i}|$ 为该执行序列的长度. 令 $mean_{P_i}=\left(\sum_{k=1}^{W}|ES^k_{P_i}|\right)\Big/W$ 表示该程序中谓词 P_i 在所有测试用例中的平均执行次数，并设集合 $M=\{mean_{P_i}\mid i\in[1,L]\}$. 我们用 $Mean_M$ 表示集合 M 中所有元素的均值，即对待测程序中所有谓词的平均执行次数求均值. 进而，我们可以用式（6）对 N 进行设定：

$$N_e=\max(\lfloor\log_2(Mean_M)-1\rfloor,1)\qquad(6)$$

其中 $\lfloor\cdot\rfloor$ 表示向下取整. 该公式等价于

$$N_e=\begin{cases}1, & Mean_M\in[0,8]\\ 2, & Mean_M\in[8,16]\\ \vdots & \vdots\\ n, & Mean_M\in[2^{n+1},2^{n+2}]\\ \vdots & \vdots\end{cases}$$

例如，在 schedule2 的第 1 个版本中，通过运行程序并统计执行信息可以得到 $Mean_M=18$，进而根据式（6）我们可以得到 $N_e=3$. 这与我们在 5.5 节实验中得到的能取得最优定位效果的 N 值是相同的. 该公式的设计来源于以下两个方面的基本思想：

（1）待测程序中谓词执行次数越多则选择的 N 值越大，这与我们在 5.5 节中得到的结论是相呼应的；

（2）待测程序的执行信息能够较好地满足 Nbit-Pesla 算法获取信息的要求，即保证执行序列向量 $\boldsymbol{E}^k_{P_i}$（参见定义 2）中的每一个元素所代表的序列模式都能够有一定数量的子序列与之匹配. 例如，当 $Mean_M\in[2^{n+1},2^{n+2})$，根据式（6）得到 $N_e=n$，则执行序列向量将包含 2^n 个元素. 从而，每个元素所代表的序列模式平均至少有 1 个以上的子序列与之匹配，最多可能存在 $2^{n+2}-n$ 个匹配的子序列；

表 12 显示了在 Siemens 和 Space 软件包中 139 个程序上式（6）对于 N_e 的选择效果. 其中，N_{opt1} 表示能取得最优定位效果的 N 值；N_{opt2} 表示能取得次优定位效果的 N 值；N_{opt3} 表示能取得第 3 位定位效果的 N 值.

表 12　在 Siemens 和 Space 中 N_e 的选择效果

选择效果	版本数	占比/%
$N_e=N_{opt1}$	78	56.1
$N_e=N_{opt2}$	24	17.3
$N_e=N_{opt3}$	10	7.2
其它情况	27	19.4
合计	139	100.0

从表 12 中可以看出，在 139 个程序版本中，N_e 对其中的 78 个准确选择到 N_{opt1}；选择 N_{opt2} 的占所有版本数的 17.3%；选择 N_{opt3} 的占了 7.2%. 超过 80% 的程序版本使用式（6）都可以选择到最好或者较好的 N 值（N_e 的取值位于前 3 位）.

进一步，基于式（6）我们可以通过以下步骤对 Nbit-Pesla 算法进行简单改进，得到的算法简称 $N_e$bit-Pesla 算法：

第 1 步，搜集待测程序中桩点谓词在每个测试用例中的执行信息；

第 2 步，根据执行信息统计得到 $Mean_M$，并使用式（6）计算 N_e；

第 3 步，令 $N=N_e$，然后使用 Nbit-Pesla 算法对该程序进行缺陷定位.

将 $N_e$bit-Pesla 算法应用于 Siemens 和 Space 软件包，并与每个缺陷版本上取得最优定位效果的 Nbit-Pesla 算法（即 $N=N_{opt1}$ 时所选择的 Nbit-Pesla 算法，以下简称为最优 Nbit-Pesla）进行比较. 图 10 中横坐标表示分别使用 $N_e$bit-Pesla 和最优 Nbit-Pesla 定位时，P-score 的差值，纵坐标表示版本个数. 例如，图 6 中左边第 1 个柱状图表示：在 119 个缺陷版本上，$N_e$bit-Pesla 与最优 Nbit-Pesla 的定位精度（P-score 计算结果）相差小于 10%. $N_e$bit-Pesla 在 90% 以上的程序都可以取得与最优定位精度接近的效果（差别小于 20%）.

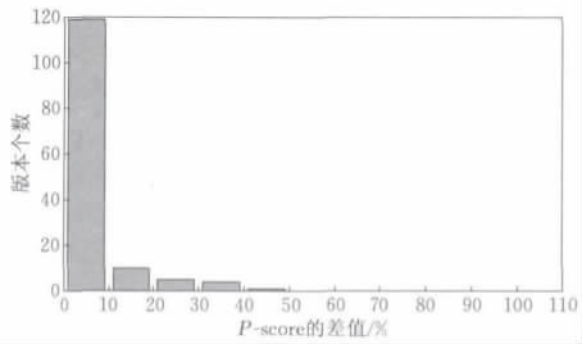

图 10　$N_e$bit-Pesla 相比最优 Nbit-Pesla 的定位效果

通过这两方面的实验，我们可以得到如下结论：使用式（6）计算得到的 N_e 确实可以对 Nbit-Pesla 算

法中的 N 进行最好或较好的选择.

5.7 结论的有效性讨论

影响本文结论有效性的因素主要有以下 3 个:

(1) 各算法的复现是否正确;

(2) P-score 评价方法是否有效;

(3) 目标软件的选取是否合适.

首先,对于 SOBER 算法和 CBI 算法,本文按照文献[2,4]所述步骤进行复现,并用文献[4]中所列举的例子验证了其正确性;对于 Wilcoxon、Mann-Whitney、F-test、t-test 算法,我们直接利用 ALGLIB 数理统计软件进行实现,而该统计软件已被多位学者和研究人员应用于研究和工程实践中,其计算精度和正确性可以保证. 因此可以认为本文对于各算法的复现应该是正确的,所得到的实验结果和数据也是有效的.

当前软件缺陷定位领域使用较多的 3 种评价方法为 T-score、Expense 以及 P-score. 每种评价方法都有不足,但都有一定的适用范围,其中 T-score 由 Renieris 和 Reiss 提出[22],该方法不但适用于基于语句的缺陷定位方法,还适用于基于谓词的缺陷定位方法;Expense 则由 Jones 等人[21]提出,主要适用于基于语句的方法,而 P-score 则由 Zhang 等人[20]于近几年提出,主要适用于基于谓词的方法,因此,选用与本文研究最相关的 P-score 作为评价各算法定位精度的标准也是合理的.

Siemens 软件包一直以来就是研究软件缺陷定位的必选实验对象之一,但是由于其中所包含的缺陷都为人工植入,很难代表真实的软件缺陷,为此,本文增加了 Space 软件作为实验对象,得到了相同的结论,但这依旧不能排除对于其它实验对象,本文的研究会得到不同实验结果的情况. 因此,在今后的研究中,增加更大规模的软件作为实验对象仍旧是一个重要的方面.

6 结束语

软件缺陷定位在软件调试过程中是一项异常费时费力的工作,在现有方法中,基于谓词的统计学缺陷定位方法在定位精度上已有良好表现. 本文在总结前人研究的基础上围绕两个问题展开研究,进一步丰富了算法所需的谓词执行信息,设计出基于谓词执行序列的缺陷定位算法,并对该算法进行了扩展. 最后用实验证明增大谓词执行信息量的确可以有效提高算法的定位精度,并且当待测程序中所蕴含的谓词执行信息可以满足算法的需求时,算法的定位精度会随着执行信息量的增加不断提高. 同时通过本文的工作也发现,若不考虑待测程序运行时对执行信息的提供能力,一味增加算法中处理的谓词执行信息量,反会使算法的定位精度降低. 因此,本文最后讨论了如何针对不同程序的执行信息为 Nbit-Pesla 算法选取合适 N 值的问题. 对 N 值设定方法的进一步分析和改进将是我们下一步的工作重点.

参 考 文 献

[1] Wong W E, Vidroha D. Software fault localization. IEEE Transactions on Reliability, 2010, 59(3): 449-482

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

[3] Liblit B, Aiken A, Zheng A X, Jordan M I. Bug isolation via remote program sampling//Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. San Diego, California, USA, 2003: 141-154

[4] Liu C, Fei L, Yan X F, et al. Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering, 2006, 32(10): 831-848

[5] Zhang Z Y, Chan W K, Tse T H, et al. Non-parametric statistical fault localization. Journal of Systems and Software, 2011, 84(6): 885-905

[6] Hu P F, Zhang Z Y, Chan W K, Tse T H. Fault localization with non-parametric program behavior model//Proceedings of the 2008 the Eighth International Conference on Quality Software. Washington, DC, USA, 2008: 385-395

[7] Zhang Z Y, Jiang B, Chan W K, et al. Fault localization through evaluation sequence. Journal of Systems and Software, 2010, 83(2): 174-187

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

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

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

[11] Sterling C, Olsson R. Automated bug isolation via program chipping//Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging. Monterey, California, USA, 2005: 23-32

[12] Zhang X Y, He H F, Gupta N, Gupta R. Experimental evaluation of using dynamic slices for fault location//Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging. Monterey, California, USA, 2005: 33-42

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

[14] Collofello J S, Cousins L. Towards automatic software fault location through decision-to-decision path analysis//Proceedings of the National Computer Conference. Chicago, USA, 1987: 539-544

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

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

[17] Wong W E, Qi Y. BP neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering, 2009, 19(4): 573-597

[18] Briand L C, Labiche Y, Liu X T. Using machine learning to support debugging with tarantula//Proceedings of the 18th IEEE International Symposium on Software Reliability. Trollhattan, Sweden, 2007: 137-146

[19] Ascari L C, Araki L Y, Pozo A R T, Vergilio S R. Exploring machine learning techniques for fault localization//Proceedings of the 10th Latin American Test Workshop. Buzios, Rio de Janeiro, Brazil, 2009: 1-6

[20] Zhang Z Y, Chan W K, Tse T H, et al. Capturing propagation of infected program states//Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering. Amsterdam, Netherland, 2009: 43-52

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

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

LI Wei, born in 1986, M. S. His research interests include software testing and fault localization.

ZHENG Zheng, born in 1980, Ph. D., associate professor. His research interests include intelligent decision, software reliability and fault localization.

HAO Peng, born in 1986, M. S. candidate. His research interests include software testing and fault localization.

GAO Yi-Chao, born in 1989, M. S. candidate. His research interests include software testing and fault localization.

RAO Pei-Feng, born in 1987, M. S. candidate. His research interests include software testing and fault localization.

GONG Cheng, born in 1987, M. S. candidate. His research interests include software testing and fault localization.

Background

Fault localization is to determine where the faults in a piece of code lie, given information about a failing test case execution. In recent years, much attention has been paid to statistical fault localization techniques (SFLT). The SFLTs obtain the suspiciousness of a statement by collecting and comparing the statement execution information (also be called program spectra in some papers) in both passing and failing test cases execution, and give the programmer a report of potential faulty statements. Studies show that these techniques can be helpful in reducing the fault localization expense by reducing the percentage of the program that must be inspected to find the fault.

SFLTs can be classified as predicate-based or statement-based ones according to the type of information used. In this paper, we focus on the predicate-based SFLTs (called PBSD in this paper), which obtain the correlative relationship between predicate and fault by collecting and comparing the key predicates' execution information in both correct and incorrect runs. CBI and SOBER are two representative techniques. Experiments have been conducted and their results show that the CBI and SOBER methods fail to locate some faults because of the insufficiency of predicate execution information. Therefore, two problems arise. (1) Whether the accuracy of fault localization can be improved by enriching the predicate execution information? (2) How to evaluate the relationship between the accuracy of fault localization and the augment of predicate execution information? To answer the two questions, a new statistical method, called Pesla, is proposed, which introduces the predicate execution-sequences to enrich the predicate execution information. Experimental results clearly demonstrate that more predicate execution information can indeed improve the accuracy of the fault localization. Furthermore, if the predicate execution information of target program satisfies the algorithm's demand, the higher accuracy of the fault localization will be achieved as the more predicate execution information is used.

This work was funded by the National Natural Science Foundation of China (project No. 901018001).