北京大学

硕士研究生学位论文

题目：测试文档

姓　　名：某某

学　　号：0123456789

院　　系：某某学院

专　　业：某某专业

研究方向：某某方向

导　　师：某某教授

某年某月

版权声明

摘要

关键词：其一，其二

Test Document

Test (Some Major)

Directed by Prof. Somebody

ABSTRACT

Test of the English abstract.

KEYWORDS: First, Second

目录

第一章　引言

本章主要介绍了自动缺陷定位的研究背景及其重要意义，然后阐述了本文的研究内容、主要贡献和论文结构。

1.1　研究背景

随着软件的发展，生活中越来越多的方面都与软件有着紧密的关系。小到人们的日常出行、购物、餐饮等，大到航空航天、医药等领域，软件在人们的生活中扮演着重要的角色。随着软件的应用领域的扩大，软件的复杂性上升，提升了软件缺陷的可能性。软件缺陷可能会导致巨大的损失。一个著名的被广泛引用的例子是海湾战争时，一颗导弹由于导航软件的精度缺陷而偏离了目标，导致28人死亡和100人受伤[56]。2002年，美国国家标准与技术研究院(NIST)发表的一篇报告[37]显示，软件缺陷每年会导致约595亿美元的经济损失。{TODO:每年这么多国内公司漏洞事件} 发现并修复软件缺陷，保障软件的高质量成为一项重要的任务。

1.2　研究意义

在发现软件缺陷之后，开发人员为了解决这个缺陷往往需要三步[35]。第一步，缺陷定位，需要找到程序中和这个缺陷有关的语句。第二步，理解缺陷，明白为什么会发生缺陷。第三步，修复缺陷，修改代码以让缺陷消失。这三个步骤合起来就是调试的过程。缺陷定位作为调试的第一步，其完成速度和准确性对后面的步骤有着很大的影响。在传统的开发环境当中，人们可以手动调试来定位缺陷，比如插入断点、打印日志信息等等。在1989年Collofello等人就指出尝试去减少软件中的错误会花费50%到80%的开发和维护的精力[10]。随着软件的复杂性的上升，手动地定位软件缺陷将会耗费更多开发者的时间和精力。为了提高定位缺陷的速度，研究人员对自动化的缺陷定位展开了研究，并取得了巨大的进展{TODO:cite}。然而在2011年，Partin和Osro的一篇调查[35]通过研究缺陷定位技术在实际应用场景下的效果，发现以往的评价指标并不能准确的反映缺陷定位技术在实际应用中的效果。以往的缺陷定位技术是基于一系列关于开发人员会如何调试的假设，而这些假设在实际场景的某些情况下会失效。自动化缺陷定位技术还有很大的发展空间。

1.3 本文研究内容

1.4 本文主要贡献

1.5 论文结构

第二章　相关工作

本章讲介绍自动缺陷定位、机器学习的相关工作和缺陷定位所使用的数据集以帮助更好理解本文的工作。

2.1　自动缺陷定位相关工作

程序切片[41, 42]是自动调试最早的技术之一，但是程序切片之后可能出错的语句数量仍然比较庞大。为了解决程序切片调试方法的短板，一种通过观察错误程序的执行特征和正确程序的执行特征的调试技术被提出。这些技术通过收集程序执行信息，观察不同的某种特征，来定位缺陷。比如使用路径概要[38]，反例[6, 14]，语句覆盖[19]和谓词值[26, 28]等等。

本文根据北京大学熊英飞研究员对缺陷定位的分类[49]，将缺陷定位分为以下几类。

- 基于切片的缺陷定位
- 基于频谱的缺陷定位
- 基于状态覆盖的缺陷定位
- 基于变异的缺陷定位
- 基于构造正确执行状态的缺陷定位
- 基于算法式调试的缺陷定位
- 基于差异化调试的缺陷定位

本文的研究内容主要根据基于频谱的缺陷定位和基于状态覆盖的缺陷定位。

2.1.1　基于切片的缺陷定位

Weiser在1981年提出的程序切片[41, 42]是自动调试（特别是缺陷定位）最早的技术之——。给定一个程序P和一个在P的语句s中使用的变量v，程序切片会找到P中所有可能会影响s中v的值的语句。如果s中v的值是错误的，那么导致这个错误的错误语句一定在这个切片当中。也就是说，不在这个切片当中语句可以在调试过程中被忽略。尽管程序切片已经减少了可能出错的语句的数量，但是切片中的语句的数量仍然比较大。为了解决这个问题，Korel和Laski在1988年提出了动态程序切片[24]。动态程序切片计算某一个特定执行的切片。后来又有很多的动态程序切片的变种被提出[12, 15, 54, 55]，用于解决调试问题，并且产生了大量研究工作[4, 5, 20, 23, 29, 30, 47]。

2.1.2 基于频谱的缺陷定位

基于频谱的缺陷定位是使用最广泛的自动化缺陷定位方法[49]。程序频谱(Program Spectrum)最早由Reps等人于1997年提出[38]，用于解决千年虫问题。Harrold等人在2002年[16]提出使用测试覆盖信息作为频谱信息的调试方法。 Renieris等人在2003年提出使用通过的测试用例和失败的测试用例进行缺陷定位[36]，奠定了此后基于频谱的缺陷定位的基础。

考虑一种极端的情况。比如当某一个语句s被执行的之后，测试用例就会失败。而通过的测试用例都不会执行语句s。那么语句s很有可能就是导致缺陷的语句。找出所有这样的语句s就可以大幅减少需要排查错误的语句。但是，在实际的代码中这种极端的情况很少出现。对于一个出错的语句s，它很可能既被失败的测试用例执行，也被通过的测试用例执行。因为一个语句在其不同的上下文作用下会产生不同的效果。简单地计算通过的测试用例覆盖的语句和失败的测试用例覆盖的语句的差集是无法准确找出错误语句的。利用通过的测试用例覆盖的语句的交集和并集，与失败的测试用例覆盖的语句取差集，是最早的一种基于频谱的缺陷定位方法[36]。这种方法也隐含着基于频谱的缺陷定位的假设：被失败的测试用例执行的语句，更有可能有错误。而被通过的测试用例执行的语句，更有可能是正确的。

Jones等人提出的Tarantula[19]，直观地展示给开发者展示了每个语句在通过的测试用例和失败的测试用例下的参与情况。参与情况也被称为怀疑度。怀疑度更高的语句会在怀疑列表更靠前的位置。相比于交集并集差集的方法，Tarantula在Siemens数据集上可以将错误的语句放在怀疑列表更前面的位置[18]。

Tarantula之后，又有很多计算怀疑度的公式被提出。效果比较好的Ochiai由Abreu等人提出[1]。 Ochiai由[31]提出用于计算基因的相似度。 Abreu等人将其引入用于计算怀疑度，并与Jaccard[9]，Tarantula，AMPLE[11]比较，发现Ochiai计算的怀疑度使得定位效果更好[1, 2]。

Xie等人在理论上证明了不存在单一最佳公式[48]。

除了直接提出用于计算的公式之外，研究人员也开始使用机器学习的方法去学习怀疑度的公式。 Wong等人提出使用反向传播神经网络来定位缺陷[46]。使用的输入数据是频谱信息（语句覆盖信息）和对应的测试用例是通过还是失败。输入数据每一行对应一个测试用例。第i列为1表示的是该测试用例覆盖了第i个语句，为0则表示没有覆盖。预测的标签为1表示该测试用例失败了，为0表示通过了。为了减少需要分析的可能出错的语句的个数（每一行输入数据的维度），优先使用所有失败的测试用例覆盖的语句。此后Wong又提出了使用径向基核函数的神经网络来定位缺陷[44]。

2.1.3　基于状态覆盖的缺陷定位

在缺陷定位的时候，定位的程序元素的大小也会影响结果。程序元素可以是一条语句，一个方法，一个文件。程序元素的粒度越细，对测试信息的利用越精确。然而单个元素上覆盖的测试数量越少，统计显著性越低。如果把程序的每个执行状态作为程序元素，那么这会是一个比语句更加精细的粒度。定位结果也将更加精细，对测试的利用也会更加充分。但是，几乎不会有两个测试覆盖完全相同的状态，因为一个状态所包含的上下文信息往往十分复杂，很难完全一致。于是使用抽象状态代替具体状态。使用谓词将具体状态划分为抽象状态。谓词是形如`a > 0`这样的条件式。

Liblit等人最早提出了预定义谓词来划分状态[26]，并提出了统计性调试。通过预定义在哪些代码结构中插入哪些谓词，统计性调试能够收集到许多抽象状态的覆盖情况。

Liu等人改进了计算公式，提出了SOBER[27]。虽然Liblit的方法可以有效定位一些错误，但是Liblit的方法只考虑了一个谓词是否在一次执行中为真，而没有考虑为真的次数。 SOBER提出新的计算公式，从概率分布的角度来计算怀疑度。

除了预定义谓词以外，研究人员还提出各种从程序中获取谓词的方法。 Le等人提出Savant[25]，使用程序中的不变式的变化来划分状态。程序中的不变式使用Daikon[13]挖掘。 Savant使用Learning-to-rank方法，通过分析经典的怀疑度分数和在通过的测试用例和失败的测试用例上观察到的不变式，来定位错误的方法。 Savant基于三个出发点。一，在失败的测试用例和通过的测试用例中表现出不同的不变式的程序元素，被怀疑是有错误的。二，如果这些程序元素拥有很高的经典的怀疑度分数，那么它们更有可能是错误的。三，有一些不变式比其他不变式更加可疑，比如 `x == null`。而Savant的工作并没有引用Liblit[26]和Liu[27]，很可能是在不知道统计性调试的情况下完成的。

2.1.4　基于变异的缺陷定位

变异是对程序的任意随机修改，由变异算子得到。变异分析是测试领域的一个概念，被用于衡量一个测试集的好坏。变异分析在程序中插入变异，得到很多变异体，然后使用一组测试去执行变异体。如果一个测试集中任意测试在一个变异体上得到不同的结果，那么这个变异体被这个测试杀死。能杀死越多变异体的测试集越好。

变异被引入缺陷定位，用于定位缺陷。 Papadakis等人提出Metallaxis[34]，一个基于变异的缺陷定位。 Metallaxis基于两个假设：

- 当变异和错误在一个程序的同一条语句上时，失败的测试用例输出发生变化的概率大于通过的测试用例输出发生变化的概率。
- 当变异和错误不在同一条语句上时，通过测试用例输出发生变化的概率大于失

m	变异体
m_f	变异m导致输出发生变化的失败的测试用例个数
m_p	变异m导致输出发生变化的通过的测试用例个数
m_{f2p}	变异m导致失败的测试用例变成通过的测试用例的个数
m_{p2f}	变异m导致通过的测试用例变成失败的测试用例的个数
F	失败的测试用例的个数

表 2.1　基于变异的缺陷定位的数学符号及其意义

败的测试用例输出发生变化的概率。

基于表2.1，Metallaxis的怀疑度计算公式为

$$\text{Metallaxis}(m) = \frac{m_f}{\sqrt{F \times (m_f + m_p)}}$$

与Ochiai类似。

Moon等人提出另一个基于变异的缺陷定位技术MUSE[32]。 MUSE利用变异分析去捕捉单个语句和观察到的缺陷之间的关系。 MUSE基于的两个假设是：

- 一个失败的测试用例，比起在变异了正确语句的变异体上，在变异了错误语句的变异体上更容易变成通过的。
- 一个通过的测试用例，比如在变异了失败语句的变异体上，在变异了正确语句的变异体上更容易变成失败的。

基于表2.1，MUSE的怀疑度计算公式为

$$\text{MUSE}(m) = m_{f2p} - m_{p2f} \times \frac{\sum_m m_{f2p}}{\sum_m m_{p2f}}$$

2.1.5　基于构造正确执行状态的缺陷定位

MUSE通过变异体，可以把失败的测试用例变成通过，通过的测试用例变成失败的。假如有一个变异体，它可以把失败的测试用例变成通过的，且不会影响通过的测试用例，那么这个变异体很可能就是缺陷的补丁。但是直接分析出这样的变异体是很困难的。

Zhang提出的谓词翻转[53]巧妙地避免了直接分析出正确的补丁，而是使用改变程序状态来达到相同的目的。假如出错的是一个布尔表达式，改变程序中一个布尔表达式的取值（把真变成假，或者把假变成真），强制改变执行的分支。假如谓词翻转后，失败的测试用例变成通过的，那么对应的布尔表达式很可能有错误。

谓词翻转是局限在布尔表达式，天使调试[8]则试图解决任意表达式的错误。天使

调试要求同时具有天使性和灵活性。天使性是指，存在常量c（天使值）把表达式的求值结果替换成c，失败的测试变得通过。灵活性是指，对于所有通过的测试中的每一次表达式求值，都可以把求值结果换成一个不同的值，并且测试仍然通过。利用符号执行约束求解计算得到天使值。也由于符号执行的开销，天使调试无法应用到大型程序上。

2.1.6　基于算法式调试的缺陷定位

Shapiro提出的算法式调试[39]，通过对子问题询问“是”或“否”来定位缺陷。算法式调试把复杂的计算步骤拆为小的子问题。算法是调试的一个问题是，子问题的正确结果可能是不知道的。如果是让人进行交互式地判断，那么人需要花费时间计算判断子问题的结果。

2.1.7　基于差异化调试的缺陷定位

差异化调试由Zeller等人提出[51, 52]。不同于以往的使用动态分析或静态分析的方法去关注源代码，差异化调试关注程序状态，特别地，差异化调试关注当程序没有出错时的程序状态和程序出错时的程序状态。差异化调试尝试找到一个最小的修改集合，当把这个集合应用到没有出错时的程序状态后，程序出错了。

2.2　机器学习相关工作

2.3　缺陷定位的数据集

要研究缺陷定位，需要一个包含缺陷的数据集。这个数据集一般来说，需要有多个缺陷。对每一个缺陷，会有对应的测试用例，和对应的一个正确的版本。这些测试用例中既有通过的，也必定有失败的。失败的这个测试用例就是由缺陷导致。

Siemens数据集[17]是一个很早的数据集，用于测试充分性的实验。它由七个C程序组成，大小在141行到512行之间。这七个C程序衍生出132个有缺陷的C程序。每一个错误版本会恰好有一个缺陷。这个缺陷可能涉及多行甚至多个文件。但是这些程序的缺陷是由作者手动插入的，根据作者的描述其实和一个简单的变异操作非常相似。Siemens数据集的输入被构造用于实现完全的代码覆盖。尽管它一开始并不是被用于缺陷定位，但是很多缺陷定位技术都使用它来验证效果，比如最早的基于频谱的缺陷定位方法[36]，Ochiai[1, 2]， SOBER[27]，BPNN[46]等等。

Defects4j数据集[21]是一个真实、独立、可重现缺陷的数据集。它由六个Java开源项目的395个缺陷组成（2018年4月时，该数据集仍在更新当中）。每一个错误版本会恰

好有一个缺陷。这个缺陷可能涉及多行甚至多个文件。与Siemens数据集相比，Defects4j数据集的缺陷和测试用例都更接近实际开发情况。 Savant[25]就是在Defects4j数据集上验证的。

第三章　问题分析

本章将基于频谱的缺陷定位和基于状态覆盖的缺陷定位应用在数据集Defects4j的某些缺陷上，分析这些技术在实际缺陷上的优点与不足。

3.1　使用基于频谱的缺陷定位

基于频谱的缺陷定位对代码的内容没有假设，所使用的信息只有语句的覆盖情况。考虑Defects4j中math项目的第五个缺陷，其代码如下：

```
1  public Complex reciprocal() {
2      if (isNaN) {
3          return NaN;
4      }
5
6      if (real == 0.0 && imaginary == 0.0) {
7          return NaN; // Faulty code
8                      // Should be "return INF;"
9      }
10     ...
11 }
```

为了使用基于频谱的缺陷定位，我们运行测试用例，并且收集语句的覆盖情况。针对第2，3，6，7行语句，得到语句的覆盖情况如表3.1所示。共有一个失败的测试用例。

可以发现，第3行肯定不是缺陷语句，因为它没有被失败的测试用例覆盖过。第2，6，7行都有可能是缺陷语句。这三行都是被一个失败测试用例覆盖。根据它们被覆盖的通过测试用例的个数，可以知道第7行最有可能出错，其次是第6行，最后是第2行。这是根据了基于频谱的缺陷定位的假设，即被失败的测试用例执行的语句，更有可能

语句	被覆盖的失败测试用例个数	被覆盖的通过测试用例个数
2	1	5
3	0	1
6	1	4
7	1	0

表 3.1　Defects4j中Math的第五个缺陷的测试用例覆盖语句的情况

a_{ef}	一个语句被失败的测试用例覆盖的次数
a_{nf}	一个语句未被失败的测试用例覆盖的次数
a_{ep}	一个语句被通过的测试用例覆盖的次数
a_{np}	一个语句未被通过的测试用例覆盖的次数
a_f	失败的测试用例的个数
a_p	通过的测试用例执行的次数

表 3.2　基于频谱的错误定位的数学符号及其意义

公式名称	公式
Ochiai[1]	$\text{Susp}(s) = \frac{a_{ef}}{\sqrt{a_f \times (a_{ef}+a_{ep})}}$
Tarantula[19]	$\text{Susp}(s) = \frac{\frac{a_{ep}}{a_p}}{\frac{a_{ep}}{a_p}+\frac{a_{ef}}{a_f}}$
Barinel[3]	$\text{Susp}(s) = 1 - \frac{a_{ep}}{a_{ep}+a_{ef}}$
DStar[43]	$\text{Susp}(s) = \frac{2}{a_{ep}+(a_f-a_{ef})}$
Op2[33]	$\text{Susp}(s) = a_{ef} - \frac{a_{ep}}{a_p+1}$

表 3.3　部分基于频谱的缺陷定位怀疑度公式

有错误。而被通过的测试用例执行的语句，更有可能是正确的。

为方便此后的表述，引入一些数学符号，见表3.2。表中的统计量就是程序频谱。

表3.3中列出了部分经典的怀疑度公式。这些公式都遵循被失败的测试用例执行的语句，更有可能有错误。而被通过的测试用例执行的语句，更有可能是正确的。

利用表3.3中的公式，计算第2，6，7行的怀疑度，得到表3.4。可以发现，这五个怀疑度公式都满足

$$\text{Susp}(7) > \text{Susp}(6) > \text{Susp}(2)$$

这五个怀疑度公式都认为第7行是最有可能出错的语句，而第7行也确实是出错的语句。这些公式之间的差距不同。比如Tarantula和Op2认为这三行的怀疑度是非常接近的。 Ochiai，Barinel和DStar认为第2行和第6行的怀疑度接近，而第7行的怀疑度明显高于第2行和第6行的怀疑度。

这个例子体现了基于频谱的错误定位准确定位错误的能力。但是实际上效果往往没有这么好。其实在该缺陷中，还存在一个正确的语句，它被一个失败的测试用例覆盖过，且从没有被正确的测试用例覆盖过。这个正确的语句的频谱信息和错误语句的频谱信息完全一致，所以它们的分数会相同。而这个正确的语句将会干扰开发者对缺陷的分析。

公式	第2行	第6行	第7行
Ochiai	0.4082	0.4472	1.0000
Tarantula	0.9988	0.9990	1.0
Barinel	0.1667	0.2000	1.0
DStar	0.2000	0.2500	Infinity
Op2	0.9988	0.9990	1.0

表 3.4　Defects4j中Math的第五个缺陷的经典公式怀疑度

3.2　使用基于状态覆盖的缺陷定位

考虑Defects4j数据集中Math的第二个缺陷，其代码如下：

```
1 public double getNumericalMean() {
2     return (double) (getSampleSize() * getNumberOfSuccesses()) / (
          double) getPopulationSize(); // Faulty code
3     // Should be "return getSampleSize() * (getNumberOfSuccesses() / (
          double) getPopulationSize())"
4 }
```

使用Ochiai方法的话，该错误语句被排到第11位。并列的分数将取其平均排名（期望排名），这是很多研究方法所使用的评估方式[22, 40, 45, 50]。事实上该语句的分数位列第2位，但是一共有17个语句和该语句分数并列，导致最终排名为第11位。基于频谱的缺陷定位方法在这里失效了。其实这个缺陷更加适合使用基于状态覆盖的缺陷定位技术。

状态覆盖就是使用谓词把具体状态划分为抽象状态。比如，对于如下代码C代码，

```
1 a = abs(a);
2 if (update_b) {
3     b = sqrt(a);
4 }
```

当a和b的类型都为int时，如果a的值为最小的int时（a = -2147483648），则代码会在第3行出错（b的值为NAN）。这是因为当a = -2147483648时，第1行的a会被赋值为一个负数，于是在第3行进行sqrt操作的时候，就被出错。在第1行的时候，考虑两个抽象的状态a ≥ 0和a < 0。发现通过的测试只有a ≥ 0这个状态，而失败的测试只有a < 0这个状态。所以可以认为a < 0是缺陷状态，引入这个状态的第1行的语句很可能就是缺陷语句。通过谓词a ≥ 0和a < 0把程序的具体状态划分成了两个抽象状态，从而定

t_f	一个谓词被观测为真的失败的测试用例的个数
t_p	一个谓词被观测为真的通过的测试用例的个数
a_f	一个谓词被观测的失败的测试用例的个数（谓词不一定为真）
a_p	一个谓词被观测的通过的测试用例的个数（谓词不一定为真）
F	失败的测试用例的个数
P	通过的测试用例执行的次数

表 3.5　统计性调试的数学符号及其意义

位了第3行的缺陷。

3.2.1　统计性调试

Liblit[26]提出的统计性调试使用预定义谓词。预定义谓词分为三类

- **分支**：对每一个条件语句，观察这个条件语句为真的谓词和为假的谓词。这个条件语句包括`if`条件这样的，也包括各种隐式的条件比如循环。
- **返回**：在C程序中，一个函数的返回值往往会被用于表达成功或者失败。对于每一个数值的返回值，观察六种谓词$< 0, \leq 0, > 0, \geq 0, = 0, \neq 0$。
- **数值对**：对于每一个数值赋值语句`x = ...`，找到所有和`x`同类型的、在作用域内的变量`y`和常量表达式`c`。对于每个`y`和`c`，观察六种谓词$<, \leq, >, \geq, =, \neq$。

为方便此后的表述，引入数学符号表3.5。

通过预定义谓词被测试用例的覆盖情况，计算得到每个谓词对应的怀疑度：

$$\text{StatisticalDebugging}(s) = \frac{2}{\frac{1}{\frac{t_f}{t_f+t_p}-\frac{a_f}{a_f+a_p}} + \frac{log(F)}{log(t_f)}}$$

当把统计性调试应用到Defects4j数据集中Math的第二个缺陷时，会在出错的代码处增加谓词，因为出错的代码处刚好是一个返回。虽然在Java程序中，函数返回值不会像C程序那样经常用于表达成功或失败，但是这些返回值有时也表达出程序执行的一些信息。比如在Math的第二个缺陷中，会发现该错误语句处的六个谓词会有表3.6中的覆盖情况。另外还有$a_f = 1$，$a_p = 6$。然而谓词1、2、5这些真分支被失败的测试用例覆盖的谓词的怀疑度都为0，谓词3、4、6的怀疑度都为负数。因为谓词1，2，5的$t_f = 1$，导致$log(t_f) = 0$，然后$\frac{log(F)}{log(t_f)} = INF$，于是最终计算得到的怀疑度为0。而谓词3、4、6由于$t_f = 0$，导致$log(t_f) = -INF$，致使分母中第二项为0。虽然谓词1、2、5的分数比3、4、6的分数高，但是0分并没有让这个出错的语句在整个代码的执行语句中排到前面。事实上对于所有真分支被失败用例覆盖的语句，由于其$t_f = 1$，最终其怀疑度都为0。由于每个谓词都存在和它取值相反的另一个谓词（比如`x > y`和`x <=`

	谓词	谓词为真的失败的测试用例个数t_f	谓词为真的通过的测试用例个数t_p
1	`retValue < 0`	1	0
2	`retValue <= 0`	1	1
3	`retValue > 0`	0	5
4	`retValue >= 0`	0	6
5	`retValue != 0`	1	5
6	`retValue == 0`	0	1

表 3.6　返回值谓词的覆盖情况，其中
`retValue = (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()`

y)，所以$t_f = 1$总是存在的。

在这个例子中，其实统计性调试得到了具有划分缺陷状态和非缺陷状态的谓词。但是由于统计性调试的需要多个失败的测试用例覆盖出错的语句，而在Defects4j这个数据集中多数缺陷都只有一个测试用例覆盖到，导致统计性调试的效果在Defects4j数据集上效果不好。在Liblit[26]的实验中，对每一个研究对象生成32000个随机输入。于是一个错误语句往往能被多个失败的测试用例覆盖。可见统计性调试的方法在实际数据集里往往只有一个失败的测试用例的情况下并不适用。

3.2.2　SOBER

SOBER[28]也是基于状态覆盖的错误定位，改进了统计性调试的计算方法。SOBER的公式计算的是对一个谓词，在失败的测试用例下这个谓词为真的概率分布，和在通过的测试用例下这个谓词为真的概率分布是否相似。如果概率分布无论是在失败的测试用例中还是通过的测试用例中都一样，那么这个谓词对应的变量等和缺陷的关系就越小。如果两个概率分布相差很大，说明这个谓词对应的抽象状态很有可能就有缺陷状态。引入这个缺陷状态的语句很可能就是出错的语句。

SOBER的计算公式为

$$\text{Sober}(s) = -log(\text{Sim}(f(X|\theta_p), f(X|\theta_f)))$$

其中$f(X|\theta_p)$表示通过的测试用例下这个谓词为真的概率分布， $f(X|\theta_f)$表示失败的测试用例下这个谓词为真的概率分布， Sim函数则计算这两个概率分布的相似度。

为了计算相似度，首先提出零假设

$$\mathcal{H}_0 : f(X|\theta_p) = f(X|\theta_f)$$

即两个概率分布没有区别。然后使用总平均μ和方差σ^2来刻画概率分布，所以零假设为 $\mu_p = \mu_f$并且$\sigma_p^2 = \sigma_f^2$。假设一共有m个失败的测试用例，令$\mathbf{X} = (X_1, X_2, ..., X_m)$是一个从$f(X|\theta_f)$得到的独立同分布随机样本。在零假设下，根据中心极限定理，统计量

$$Y = \frac{\sum_{i=1}^{m} X_i}{m}$$

渐近于$N(\mu_p, \frac{\sigma_p^2}{m})$，一个均值为$\mu_p$方差为$\frac{\sigma_p^2}{m}$的正态分布。令$f(Y|\theta_p)$为$N(\mu_p, \frac{\sigma_p^2}{m})$的概率密度函数。使用似然函数$L(\theta_p|Y)$作为相似度计算的函数，有

$$\text{Sim}(f(X|\theta_p), f(X|\theta_f)) = L(\theta_p|Y) = f(Y|\theta_p)$$

根据正态分布的性质，统计量

$$Z = \frac{Y - \mu_p}{\sigma_p / \sqrt{m}}$$

渐近于$N(0, 1)$，而且

$$f(Y|\theta_p) = \frac{\sqrt{m}}{\sigma_p}\varphi(Z)$$

其中$\varphi(Z)$是$N(0, 1)$的概率密度函数。最后得到怀疑度计算公式：

$$\text{Sober}(s) = log\left(\frac{\sigma_p}{\sqrt{m}\varphi(Z)}\right)$$

从SOBER的公式可以看出，SOBER仍然是建立在有大量测试用例的基础上。少量的测试用例会让概率分布不能准确反映出谓词真假分支的取值分布。SOBER的验证实验也是在人造的Siemens数据集上完成的。

在Defects4j的Math的第二个缺陷这个例子中，该错误语句的六个谓词中，`((double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()) < 0`得分最高，覆盖情况见表3.7，怀疑度为360.85。在怀疑度列表中排名第10，效果并不理想。

3.3 分析结论

在以上的例子和分析中我们发现，现有缺陷定位存在不足。

对于基于频谱的缺陷定位来说，它仅仅依赖频谱信息去区分正确语句和错误语句，会导致很多正确语句也具有很高的怀疑度。特别地，如果一个正确语句只被失败的测试用例覆盖，那么它将拥有非常高的怀疑度。这是由于频谱信息的信息量太少，

测试用例编号	覆盖真分支的次数	覆盖假分支的次数	当前测试用例状态
1	0	2	通过
2	0	2	通过
3	0	2	通过
4	0	2	通过
5	1	0	失败
6	0	10	通过
7	0	1000	通过

表 3.7　SOBER方法下，Defects4j的Math第二个缺陷的错误语句里，分数最高的谓词的覆盖情况

基于频谱的缺陷定位忽略了程序状态等被基于状态覆盖的缺陷定位关注的信息。

而基于状态覆盖的缺陷定位，虽然能够获得比频谱信息更多的信息，但是现有的方法都依赖于大量的测试用例。在测试用例不足的时候，基于状态覆盖的缺陷定位无法给出具有区分度的怀疑度。

第四章　设计

4.1　结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位

既然基于频谱的缺陷定位和基于状态覆盖的缺陷定位各有优劣，那么是否可以结合这两种缺陷定位的方法呢？事实上已经有研究[25, 50]，结合了多种缺陷定位方法，并且获得了比较好的结果。{TODO:展开}。但是这些研究的结合方式都是在比较高的层次，比如使用机器学习方法对不同缺陷定位得到的结果进行组合。这样的结合方式会有两个缺点。一是他们难以解释为什么他们的方法会起作用。二是他们没有深入理解缺陷定位方法起作用的原因，仅仅是把各个方法的结果合在一起。

所以，本文试图提出一个能够结合多种缺陷定位（比如基于频谱的缺陷定位和基于状态覆盖的缺陷定位）的方法去改进缺陷定位技术，同时本文试图解释这个结合为什么起作用的原因。

虽然在直觉上我们认为基于频谱的缺陷定位和基于状态覆盖的缺陷定位是完全不一样的。因为基于频谱的缺陷定位依靠的是程序元素的覆盖情况，而基于状态覆盖的缺陷定位依靠的是用谓词来划分状态。但是事实上这两种缺陷定位技术有相似的地方。基于频谱的缺陷定位的频谱信息，其实相当于是对每一个语句都关联了一个`true`这样的谓词。这样看来基于频谱的缺陷定位相当于基于状态覆盖的缺陷定位的一个特殊情况。而基于状态覆盖的缺陷定位收集的谓词的覆盖信息也可以看做是程序频谱信息的一种，所以基于状态覆盖的缺陷定位也可以看做基于频谱的缺陷定位的一个特殊情况。

考虑3.2章中基于状态覆盖的缺陷定位的例子。统计性调试和SOBER都无法给出很好的定位结果。但是当观察统计性调试的覆盖情况3.6，我们却可以“猜测”出当前语句很可能是错误语句。这是因为我们带入了基于频谱的缺陷定位的假设：被失败的测试用例执行的语句，更有可能有错误。而被通过的测试用例执行的语句，更有可能是正确的。根据这个假设，表3.6中的谓词3、4、6都不太可能是能够划分出缺陷状态的谓词，因为它们都没有被失败的测试用例覆盖过。谓词1最有可能是能够划分出缺陷状态的谓词，其次是谓词2，最后是谓词5。这是因为谓词1、2、5都被一个失败的测试用例覆盖过，而谓词1没有被通过的测试用例覆盖过。这种情况下被越少的通过的测试用例覆盖，越有可能就是能够划分出缺陷状态的谓词。怎样去具体地表示这个怀疑度呢？这其实是基于频谱的缺陷定位解决的问题了，那就是使用怀疑度公式。使用Ochiai怀疑度公式去计算表3.6中谓词的怀疑度，得到表4.1。可见谓词1以1.0000的分

	谓词	Ochiai分数
1	`retValue < 0`	1.0000
2	`retValue <= 0`	0.7071
3	`retValue > 0`	0.0000
4	`retValue >= 0`	0.0000
5	`retValue != 0`	0.4082
6	`retValue == 0`	0.0000

表 4.1　使用Ochiai计算谓词怀疑度，其中
`retValue = (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()`

数远远高于其他谓词，成为怀疑度很大的谓词。使用Ochiai怀疑度公式，计算Math的第二个缺陷的各个谓词怀疑度，错误语句排名第3（第1到4名并列），相比于基于频谱的状态覆盖第11位、统计性调试全部为0和SOBER第10的结果，有显著提升。

4.2　预定义谓词和预测谓词

在统计性调试和SOBER中，都使用的是预定义的谓词。一个谓词的好坏决定了能否划分出缺陷状态。

考虑Defects4j中Math的第四个缺陷，其代码如下：

```
1  public Vector3D intersection(final SubLine subLine, final boolean
       includeEndPoints) {
2      // compute the intersection on infinite line
3      Vector3D v1D = line.intersection(subLine.line);
4  +   if (v1D == null) {
5  +       return null;
6  +   }
7
8      // check location of point with respect to first sub-line
9      Location loc1 = remainingRegion.checkPoint(line.toSubSpace(v1D));
10     ...
11 }
```

第4，5，6行是修复缺陷的代码。这个的缺陷是缺少了对变量`v1D`是否是空的判断。考虑谓词`v1D == null`，会发现通过的测试用例都不会覆盖这个谓词，只有失败的测试用例覆盖这个谓词。因为一旦这个谓词为真，那么后续某些使用这个`v1D`变量的操作就会造成空指针的错误。

然而这个谓词并不能由预定义的谓词得到。但是事实上代码中其他的地方存在var

== null这样的判断。于是本文提出了一种基于机器学习的预测谓词的方法，来更准确地找出划分缺陷状态的谓词。

或许我们不使用机器学习方法，而是把var == null这样经典的判断加入预定义谓词呢？但问题是这样的谓词永远是加不完的。而且对于每个程序，它们可能还拥有跟自己上下文有关的独特的谓词。所以从它们自己的代码中来学习出谓词是更有效的方法。

4.3　缺陷定位框架

本文的缺陷定位框架主要包括以下几步

- **收集特征** 当使用预定义谓词时不需要这一步。假如缺陷代码是版本v，则从版本v的源代码中提取特征。特征分为两种，一种是变量预测模型的特征，一种是谓词预测模型的特征。
- **训练模型** 当使用预定义谓词时不需要这一步。把得到的特征放入机器学习模型中训练。两种特征分别单独训练，得到一个预测变量出现在谓词中概率的模型（简称VAR模型），和一个根据一个变量预测可能出现哪些谓词的模型（简称EXPR模型）。
- **收集失败测试用例覆盖的语句** 只有被失败的测试用例覆盖的语句才有可能是错误的语句。因为只需要对失败测试用例覆盖的语句收集其谓词的覆盖情况就可以了。
- **获取谓词** 如果使用预定义谓词，则根据预定义谓词的规则分析代码，得到谓词。如果使用预测谓词模型，则提取需要插入谓词的相关变量和语句的特征，放入已经训练好的模型中，预测出当前位置最有可能出现的N个谓词。
- **收集谓词覆盖情况** 把谓词插入代码中，并执行测试用例，收集谓词的覆盖情况。不同怀疑度公式收集的覆盖情况可能会有不同。覆盖情况包括：谓词的真（或假）分支被多少个失败（或通过）的测试用例覆盖，谓词（无论是真分支还是假分支）被多少个失败（或通过）的测试用例覆盖，谓词的真（或假）分支在一个失败（或通过）的测试用例中被覆盖的次数等等。
- **计算怀疑度** 根据上一步收集的谓词覆盖情况，带入基于频谱的缺陷定位和基于状态覆盖的缺陷定位的怀疑度公式计算怀疑度。

4.4　基于机器学习的预测谓词模型

对谓词的预测分为三步：

- 第一步，使用VAR模型预测语句中的某个变量出现在谓词中的概率P_{var}。
- 第二步，使用EXPR模型预测语句中的某个变量会出现在谓词$pred_i$中的概率P_{pred_i}。
- 第三步，将谓词按照$P_{var} \times P_{pred_i}$排序。

第一步中的变量是赋值表达式的左值。也就是说我们只会对失败的测试用例覆盖的赋值语句进行表达式的预测。

4.4.1 机器学习特征

对于一个有缺陷的代码，我们从当前这个版本的所有源代码中提取特征。{TODO:现在提取的特征的说明}

4.4.2 机器学习特征编码

编码

对于数值型的变量，直接使用其值作为特征。而对于分类变量（categorical variable），虽然其值可能表现为0、1、2…… 这样的数字，但是其实并不存在$0 < 1 < 2$这样的大小关系。直接使用其值会让模型以为这个变量存在大小关系。所以对于分类变量，本文采取独热（one-hot）编码。比如对于一个值为0、1、2的分类变量v，使用新的变量v_0、v_1、v_2代替v。其中$v_i = 1$表示$v = i$，而$v_i = 0$表示$v \neq i$。

字符串特征编码

对于字符串类型的特征，比如文件名、函数名和变量名，本文也会使用独热编码。但是直接使用独热编码会有两个问题：

1. 特征维度过大。比如对Defects4j中Math项目的第一个缺陷来说，仅不同的方法就有795个。这意味着如果把方法改成独热编码，将会把一个1维的特征变成795维的特征。这样可能造成维度灾难[7]。虽然一开始随着特征数的上升，机器学习模型的预测效果也会上升，但是当维度过高的时候，实际性能是下降的。但这还不是最关键的因素。
2. 丢失了字符串内部的特征。字符串的变量和其它的变量不同，它们之间还有内部的特征。比如变量名len和length之间的相似度比len和domain之间的相似度要高，文件名SubLine.java和Line.java之间的相似度比SubLine.java和BracketFinder.java之间的相似度高。简单地把不同字符串看成完全独立地不同特征会丢失很多有用的信息。

对字符串的编码采取三步。

首先是将字符串转换为向量。对变量名，采取一种类似2-gram的方法。每一个变量名都会被转成一个长度为729(27 × 27)的一维向量。变量名中的大写字母会先被转为小写字符。对转换后的变量名$s_1s_2...s_n$，考虑每两个相邻字符的字符串s_1s_2，s_2s_3 ……$s_{n-1}s_n$。这个向量的第i位为1表示变量名中存在两个相邻字符s_js_{j+1}，满足$f(s_j) \times 27 + f(s_{j+1}) = i$。 f是一个映射函数，将一个字符转换成一个0到26之间的数字。对于f有$f('a') = 0, f('b') = 1, ..., f('z') = 25$，然后a到z以外的字符都被转为26。这样的话len和length的特征向量之间就会有两位相同，而len和domain之间则完全没有相同的。对函数名和文件名，去掉后缀，然后利用Java中使用的驼峰命名法将它们按照驼峰分割开来。比如SubLine.java会被拆为Sub和Line，而Line.java得到Line，BracketFinder.java得到Bracket和Finder。收集所有分割后的词（函数名和文件名分开收集），假设有N_{func}和N_{file}个，则每个函数名（或文件名）就转成一个长度为N_{func}（或N_{file}）的向量。这个向量的第i位为1表示函数名（或文件名）中含有字符串$g_{func}(i)$（$g_{file}(i)$）。 g_{func}和g_{file}是向量下标和字符串的映射。比如$g_{file}(57) =$ "*Line*"的话，Line.java的向量的第57位为1，其他位为0。

然后是对字符串转换后的向量进行聚类。聚类使用的是scikit-learn{TODO:cite}的K-means聚类算法。因为不同的项目所涉及的变量名、函数名、文件名数量差别较大，不适合使用单一的某个常数作为聚类的类别数。所以聚类的类别数为$Unique(names)/20$。其中$names$表示所有变量名或函数名或文件名，$Unique(x)$表示x中不重复的值的数量。

最后是根据聚类给字符串编码。加入聚类结果有N个类，那么就把这N个类编号为1到N。每个类中的每个字符串的编号等于它所处的类的编号。每个字符创的编号就是它的特征值，对这个特征值使用独热编码就得到最终的特征。于是通过重新编码和聚类，字符串特征的内部特征被抽取出来，并且字符串特征的维度也被大幅减小。

预测时的特征新值

在预测的时候，特征里面可能会有在训练的特征里面没有出现过的值。如果这个值是数值型的值，那么直接使用这个值就可以。但是如果是经过编码之后的值，就需要给这个新值一个编码。

为了给出一个和训练时编码一致的编码，需要保存训练时的部分数据。包括变量名、文件名、函数名的聚类模型，文件名、函数名中向量下标和字符串的对应函数g_{func}、g_{file}，训练时使用的独热编码。

对于新的变量名，按照训练时的处理方法将其转为729(27 × 27)的一维向量。然后计算出这个向量与训练时变量名聚类模型中的哪个中心点距离更短，把这个变量名划

入这个中心点的分类。最后使用训练时变量名的独热编码给这个分类编码即可。

对于新的文件名或函数名，有两种情况。第一种情况是，这个文件名虽然没有出现过，但是它按照驼峰拆分之后的字符串都出现过。这种情况下使用g_{func}和g_{file}直接构造特征向量。第二种情况是，这个文件名按照驼峰拆分之后的字符串有没有出现过的。没有出现过的字符串则会被忽略。构造出的特征向量使用训练时的聚类模型划分分类，最后使用训练时的独热编码给改分类编码。

对于除变量名、文件名、函数名以外的分类变量，如果出现了新的值，假设这个分类变量的类别有C个，那么新值转成独热编码后其特征向量为C个0。

4.4.3 机器学习模型

本文在小量数据上尝试了多种机器学习模型，最后选定了两种机器学习模型：全连接神经网络和决策树模型。

神经网络模型

神经网络使用的全连接神经网络，由TensorFlow{TODO:cite}实现。VAR模型和EXPR模型都是六层的全连接神经网络，再加一层softmax层。六层是因为{TODO:cite}。每层有64个节点，激励函数使用{TODO:tf激励函数}。

决策树模型

决策树使用scikit-learn中的决策树DecisionTreeClassfier。

4.5 收集频谱信息

代码插装是一种常用的记录程序执行情况、修改程序的方法。采用的工具是eclipse提供的Java Development Tool①，简称JDT。 JDT不仅可以构造给定Java代码的抽象语法树，还可以新建、修改、插入、删除抽象语法树从而新建、修改、插入、删除Java代码。本章中很多实现都依赖于JDT。

本文使用代码插装技术主要目的是收集频谱信息。比如为了收集失败测试用例覆盖的语句，可以在每个语句后面加上一个打印语句。这个打印语句会打印出当前的文件名和对应语句的行数。用失败的测试用例执行这个插装后的程序，就可以得到失败测试用例覆盖的语句。

对于谓词P和一个测试用例T_i，可能需要收集种信息：

1. P是否被T观察过（无论是真还是假）。

① http://www.eclipse.org/jdt/

2. T_i中P是否至少一次为真。
3. T_i中P为真的次数。
4. T_i中P为假的次数。

对于基于频谱的缺陷定位的公式来说，需要第2个信息。对于统计性调试的公式来说，需要第1、2个信息。对于SOBER的公式来说，需要第3、4个信息。所以整体来说谓词的插装有两种形式。第一种用于基于频谱的缺陷定位公式和统计性调试公式：

```
// predicateSignature is a string that uniquely represents the
   predicate.
SpecLogger.observe(predicateSignature);
if (predicate) {
    SpecLogger.cover(predicateSignature);
}
```

SpecLogger.observe记录这个谓词为观察过（信息1），SpecLogger.cover记录这个谓词为真过（信息2）。第二种用于SOBER公式：

```
// predicateSignature is a string that uniquely represents the
   predicate.
if (predicate) {
    SpecLogger.coverTrueBranch(predicateSignature);
} else {
    SpecLogger.coverFalseBranch(predicateSignature);
}
```

SpecLogger.coverTrueBranch记录这个谓词为真的次数（信息3），SpecLogger.coverFalseBranch记录这个谓词为假的次数（信息4）。

这几个*SpecLogger*的函数可以是把自己内部的某个布尔型的成员变量置为真或假，也可以是把自己内部的某个整型的成员变量加一。这个*SpecLogger*对象在每个测试用例开始的时候重置（使用静态方法变量），并在测试用例结束的时候以某种格式将自己记录的信息输出到文件。

```
private void testSomething() {
        SpecLogger.reset();
    SpecLogger.testStatus = true;

    // test ...
    ...

```

```
8       SpecLogger.dump();
9   }
```

第三行是根据当前测试用例testSomething是通过的测试用例还是失败的测试用例而插装的。最后SpecLogger.dump把记录的信息输出到文件。其他程序从这个输出文件中可以构造出频谱信息。

4.6 不改变程序状态地插入谓词

统计性调试中自定义的谓词和预测出来的谓词可能有副作用。使用上一章中的插装方法会改变程序的执行状态，所以需要不改变程序状态地插入谓词。

统计性调试中的谓词要么是两个变量构成的二元表达式，要么就是本来就会在程序中执行的条件表达式。前者不会有副作用，而后者即使有副作用，由于其本来就要在原程序中执行一次，因此只要利用执行的这一次的结果就可以。

预测出来的谓词可能含有有副作用的函数，或者含有赋值语句。于是对预测出谓词的静态分析，只留下一定无副作用的谓词。比如除了部分指定函数（如size），含有其他函数的谓词都会被过滤掉。

4.6.1 插入预测的谓词

插入谓词仍然使用JDT。

通过机器学习模型，每个语句可能会关联一组谓词。这些谓词会被机器学习模型赋予出现的概率。最终每个语句我们选择概率最高的5个谓词作为需要插入的谓词。

在插入预测的谓词之前，要先对谓词进行过滤。过滤包括两步：

1. 静态分析过滤掉可能不合法的谓词（比如对一个int类型的变量进行下标访问）。
2. 静态分析过滤掉可能有副作用的谓词。
3. 过滤掉不能编译的谓词。

一个预测出来的谓词分为两部分，一个是谓词$P(x)$，一个是变量v，最终构成谓词$P(v)$。一个谓词会被判定为不合法如果它满足以下至少一点：

- 含有数组访问v[i]且v并不是数组类型。
- 含有变量访问v.a且v是一个基本数据类型（比如int）的变量。 {TODO:bug fix}
- 含有变量访问a.v且v不是a的一个域。
- {TODO:simple name}
- 含有函数调用v.a()且v是一个基本数据类型变量。

- 含有函数调用f()且f不属于预定义的合法函数（如size，length，toString，contains，containsKey，Math.abs，Double.isInfinite，Double.isNaN）。
- 含有域访问v.a且v是一个基本数据类型的变量。
- 含有中缀表达式a op b且a,b的类型和运算符op不匹配（比如对非数字类型进行加法）。

过滤有副作用的谓词主要是过滤掉以下几种：

- 含有前缀表达式，且其中运算符为++或-。
- 含有后缀表达式。
- 含有赋值语句。

最后单独地插入每一条谓词，过滤掉不能顺利编译的。在没有判定谓词是否合法的时候，也可以通过编译来排除不合法的谓词。但是由于谓词数量较多，通过预处理去掉部分肯定不对的谓词可以加速整个流程。

最后对于一个语句s，我们可以得到一组合法无副作用的谓词$P_1, P_2 \ldots$。我们再加入取反的谓词$!P_1, !P_2, \ldots$。

然后就是将收集谓词频谱信息的代码插入到语句前或后。语句前后都插入的有WhileStatement，ForStatement，DoStatement，EnhancedForStatement，插入在语句后面的有Assignment，VariableDeclara ConstructorInvocation，SuperConstructorInvocation，插入在语句前的有IfStatement，SwitchStatement等其他所有语句。

4.6.2　插入预定义的谓词

预定义的谓词也可能有副作用，比如赋值语句a[b++] = c的谓词a[b++] > d，如果使用此前的插装方法，则插入if (a[b++] > d)这样的语句会产生副作用。但是如果使用中间变量，则上述代码可以重写为：

```
1 temp = c;
2 a[b++] = temp;
3 if (temp > d) {
4     ...
5 }
```

所以预定义谓词的三种情况，分支、返回、数值对，都可以使用中间变量这样的方式来插入谓词，从而避免了副作用的情况。

由于分支对应多种情况，使用中间变量会比较复杂。分支的谓词是分支中含有的条件表达式。DoStatement中的条件表达式的值每次循环都会更新，再考虑上continue这样的语句，会让条件表达式的更新逻辑非常复杂。为了简单地插入谓词，使用一个函

数logConditionCoverage替换条件表达式。原代码为：

```
// example.java:
while(!iter.isEmpty()) {
        ...
}
```

插装了收集谓词频谱信息的语句后，代码被修改为：

```
// example.java
while(SpecLogger.logConditionCoverage(!iter.isEmpty(), "!iter.isEmpty()
   ", "!(!iter.isEmpty())") {
        ...
}

// SpecLogger.java
public static boolean logConditionCoverage(boolean condition, String
   trueLogInfo, String falseLogInfo) {
        observe(trueLogInfo);
        observe(falseLogInfo);
        if (condition) {
                cover(trueLogInfo);
        } else {
                cover(falseLogInfo);
        }
        return condition;
}
```

这里同时记录了假分支的覆盖情况。这样做有两个目的：

- 统计性调试中预定义的分支型谓词有条件表达式取反。
- 和预测谓词加入了谓词取反的情况保持一致。

4.7 计算怀疑度

收集频谱信息后，就可以带入计算怀疑度。无论是机器学习得到的谓词，还是预定义的谓词，都使用表3.3中的五种公式，以及统计性调试和SOBER总共七种公式进行计算。

第五章　实现

结论

参考文献

[1] R Abreu, P Zoeteweij and A. J. C Van Gemund. "*An Evaluation of Similarity Coefficients for Software Fault Localization*". In: *Pacific Rim International Symposium on Dependable Computing*, **2006**: 39–46.

[2] R Abreu, P Zoeteweij and A. J. C Van Gemund. "*On the Accuracy of Spectrum-based Fault Localization*". In: *Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation*, **2007**: 89–98.

[3] Rui Abreu, Peter Zoeteweij and Arjan J. C Van Gemund. "*Spectrum-Based Multiple Fault Localization*". In: *Ieee/acm International Conference on Automated Software Engineering*, **2009**: 88–99.

[4] Hiralal Agrawal, Richard A. Demillo and Eugene H. Spafford. *Debugging with dynamic slicing and backtracking*, **1993**: 589–616.

[5] Elton Alves, Milos Gligoric, Vilas Jagannath *et al.* "*Fault-localization using dynamic slicing and change impact analysis*". In: *Ieee/acm International Conference on Automated Software Engineering*, **2011**: 520–523.

[6] Thomas Ball, Mayur Naik and Sriram K Rajamani. "*From symptom to cause: localizing errors in counterexample traces*". *Acm Sigplan Notices*, **2003**, *38*(1): 97–105.

[7] Richard Ernest Bellman. *Dynamic programming*. Princeton University Press, **1957**.

[8] Satish Chandra, Emina Torlak, Shaon Barman *et al.* "*Angelic debugging*". In: *International Conference on Software Engineering*, **2011**: 121–130.

[9] Mike Y. Chen, Emre Kiciman, Eugene Fratkin *et al.* "*Pinpoint: Problem Determination in Large, Dynamic Internet Services*". In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, **2002**: 595–604.

[10] James S. Collofello and Scott N. Woodfield. *Evaluating the effectiveness of reliability-assurance techniques*. Elsevier Science Inc., **1989**: 191–195.

[11] Valentin Dallmeier, Christian Lindig and Andreas Zeller. *Lightweight Defect Localization for Java.* Springer Berlin Heidelberg, **2005**: 528–550.

[12] Richard A. Demillo, Hsin Pan and Eugene H. Spafford. "*Critical slicing for software fault localization*". In: **1996**: 121–134.

[13] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo *et al.* "*The Daikon system for dynamic detection of likely invariants*". *Science of Computer Programming*, **2007**, *69*(1-3): 35–45.

[14] Alex Groce, Daniel Kroening and Flavio Lerda. "*Understanding Counterexamples with explain*". *In Computer-Aided Verification*, **2004**, *3114*: 453–456.

[15] Gyim, Tibor Thy, Besz *et al.* "*An efficient relevant slicing method for debugging*". *Acm Sigsoft Software Engineering Notes*, **1999**, *24*(6): 303–321.

[16] Mary Jean Harrold, Gregg Rothermel, Kent Sayre *et al.* "*An empirical investigation of the relationship between spectra differences and regression faults*". *Software Testing Verification & Reliability*, **2000**, *10*(3): 171–194.

[17] Monica Hutchins, Herb Foster, Tarak Goradia *et al.* "*Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria.*" In: *international Conference on Software Engineering*, **1994**: 191–200.

[18] James A. Jones and Mary Jean Harrold. "*Empirical evaluation of the tarantula automatic fault-localization technique*". In: *Ieee/acm International Conference on Automated Software Engineering*, **2005**: 273–282.

[19] Jones, A James, Harrold *et al.* "*Visualization of test information to assist fault localization*". *Biochemical Engineering Journal*, **2002**, *24*(2): 115–123.

[20] Xiaolin Ju, Shujuan Jiang, Xiang Chen *et al.* "*HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices*". *Journal of Systems & Software*, **2014**, *90*(1): 3–17.

[21] René Just, Darioush Jalali and Michael D. Ernst. "*Defects4J: a database of existing faults to enable controlled testing studies for Java programs*". In: *International Symposium on Software Testing and Analysis*, **2014**: 437–440.

[22] Benjamin Keller, Benjamin Keller, Benjamin Keller *et al.* "*Evaluating and improving fault localization*". In: *International Conference on Software Engineering*, **2017**: 609–620.

[23] Z. A. Al-Khanjari, M. R. Woodward, Haider Ali Ramadhan *et al.* "*The Efficiency of Critical Slicing in Fault Localization*". *Software Quality Journal*, **2005**, *13*(2): 129–153.

[24] Bogdan Korel and Janusz Laski. "*Dynamic program slicing ☆*". *Information Processing Letters*, **1988**, *29*(3): 155–163.

[25] Tien Duy B. Le, David Lo, Claire Le Goues *et al.* "*A learning-to-rank based fault localization approach using likely invariants*". In: *International Symposium on Software Testing and Analysis*, **2016**: 177–188.

[26] Ben Liblit, Mayur Naik, Alice X. Zheng *et al.* "*Scalable statistical bug isolation*". In: **2005**: 15–26.

[27] Chao Liu, Long Fei, Xifeng Yan *et al.* "*Statistical Debugging: A Hypothesis Testing-Based Approach*". *IEEE Transactions on Software Engineering*, **2006**, *32*(10): 831–848.

[28] Chao Liu, Xifeng Yan, Long Fei *et al.* "*SOBER: statistical model-based bug localization*". In: *European Software Engineering Conference Held Jointly with ACM Sigsoft International Symposium on Foundations of Software Engineering*, **2005**: 286–295.

[29] Chao Liu, Xiangyu Zhang, Jiawei Han *et al.* "*Indexing Noncrashing Failures: A Dynamic Program Slicing-Based Approach*". In: *IEEE International Conference on Software Maintenance*, **2007**: 455–464.

[30] Xiaoguang Mao, Yan Lei, Ziying Dai *et al.* "*Slice-based statistical fault localization ☆*". *Journal of Systems & Software*, **2014**, *89*(1): 51–62.

[31] Meyer, Andréia Da Silvagarcia, Antonio Augusto Francosouza *et al.* "*Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L)*". *Genetics & Molecular Biology*, **2004**, *27*(1): 83–91.

[32] Seokhyeon Moon, Yunho Kim, Moonzoo Kim *et al.* "*Ask the Mutants: Mutating Faulty Programs for Fault Localization*". In: *IEEE Seventh International Conference on Software Testing, Verification and Validation*, **2014**: 153–162.

[33] Lee Naish, Jie Lee Hua and Kotagiri Ramamohanarao. "*A model for spectra-based software diagnosis*". *Acm Transactions on Software Engineering & Methodology*, **2011**, *20*(3): 1–32.

[34] Mike Papadakis and Yves Le Traon. *Metallaxis-FL: mutation-based fault localization*. John Wiley and Sons Ltd., **2015**: 605–628.

[35] Chris Parnin and Alessandro Orso. "*Are automated debugging techniques actually helping programmers?*" In: *International Symposium on Software Testing and Analysis*, **2011**: 199–209.

[36] M Renieres and S. P Reiss. "*Fault localization with nearest neighbor queries*". In: *IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, **2003**: 30–39.

[37] NIST Rep. *The Economic Impacts of Inadequate Infrastructure for Software Testing*, **2002**. `http://www.abeacha.com/NIST_press_release_bugs_cost.htm`, retrieved on 2018-04-18.

[38] Thomas Reps, Thomas Ball, Manuvir Das *et al.* "*The use of program profiling for software maintenance with applications to the year 2000 problem*". *Acm Sigsoft Software Engineering Notes*, **1997**, *22*(6): 432–449.

[39] Ehud Y. Shapiro. "*Algorithmic Program DeBugging*". **1982**.

[40] Friedrich Steimann, Marcus Frenkel and Abreu Rui. "*Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators*". In: *International Symposium on Software Testing and Analysis*, **2013**: 314–324.

[41] Mark Weiser. "*Program Slicing*". *IEEE Transactions on Software Engineering*, **1984**, *SE-10*(4): 352–357.

[42] Mark Weiser. "*Program slicing*". In: *International Conference on Software Engineering*, **1981**: 439–449.

[43] W. Eric Wong, Vidroha Debroy, Ruizhi Gao *et al.* "*The DStar Method for Effective Software Fault Localization*". *IEEE Transactions on Reliability*, **2014**, *63*(1): 290–308.

[44] W. Eric Wong, Vidroha Debroy, Richard Golden *et al.* "*Effective Software Fault Localization Using an RBF Neural Network*". *IEEE Transactions on Reliability*, **2012**, *61*(1): 149–169.

[45] W. Eric Wong, Ruizhi Gao, Yihao Li *et al.* "*A Survey on Software Fault Localization*". *IEEE Transactions on Software Engineering*, **2016**, *42*(8): 707–740.

[46] W. ERIC WONG and YU QI. "*BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION*". *International Journal of Software Engineering & Knowledge Engineering*, **2009**, *19*(04): 573–597.

[47] Franz Wotawa. "*Fault Localization Based on Dynamic Slicing and Hitting-Set Computation.*" In: *International Conference on Quality Software*, **2010**: 161–170.

[48] Xiaoyuan Xie, Tsong Yueh Chen, Fei Ching Kuo *et al.* "*A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization*". *Acm Transactions on Software Engineering & Methodology*, **2013**, *22*(4): 31.

[49] Yingfei Xiong. *Fault Localization*, **2018**. `http://sei.pku.edu.cn/~xiongyf04/SA/2017/18_fault_localization.pdf`, retrieved on 2018-04-07.

[50] Jifeng Xuan and Martin Monperrus. "*Learning to Combine Multiple Ranking Metrics for Fault Localization*". In: *IEEE International Conference on Software Maintenance and Evolution*, **2014**: 191–200.

[51] A. Zeller and R. Hildebrandt. "*Simplifying and Isolating Failure-Inducing Input*". *Software Engineering IEEE Transactions on*, **2002**, *28*(2): 183–200.

[52] Andreas Zeller. "*Isolating cause-effect chains from computer programs*". In: *ACM Sigsoft Symposium on Foundations of Software Engineering*, **2002**: 1–10.

[53] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Locating faults through automated predicate switching*". **2006**, *2006*: 272–281.

[54] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Pruning dynamic slices with confidence*". *Acm Sigplan Notices*, **2006**, *41*(6): 169–180.

[55] Xiangyu Zhang, R Gupta and Youtao Zhang. "*Precise dynamic slicing algorithms*". In: *International Conference on Software Engineering, 2003. Proceedings*, **2003**: 319–329.

[56] Daming Zou, Ran Wang, Yingfei Xiong *et al.* "*A genetic algorithm for detecting significant floating-point inaccuracies*". In: *Ieee/acm IEEE International Conference on Software Engineering*, **2015**: 529–539.

附录 A　附件

致谢

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在□一年/□两年/□三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日