基于“生成-检验”框架的软件代码错误自动修复技术研究

（申请清华大学工学博士学位论文）

培 养 单 位：软 件 学 院

学　　科：软 件 工 程

研 究 生：郭 心 睿

指 导 教 师：孙 家 广 教 授

副指导教师：顾 明 教 授

联合导师：宋 晓 宇 教 授

二〇一七年六月

Automated Debugging Based on "Generate-and-Validate" Systems

Dissertation Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in

Software Engineering

by

Guo Xinrui

Dissertation Supervisor : Professor Sun Jiaguang

Associate Supervisor : Professor Gu Ming

February, 2017

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘　要

论文的摘要是对论文研究内容和成果的高度概括。摘要应对论文所研究的问题及其研究目的进行描述，对研究方法和过程进行简单介绍，对研究成果和所得结论进行概括。摘要应具有独立性和自明性，其内容应包含与论文全文同等量的主要信息。使读者即使不阅读全文，通过摘要就能了解论文的总体内容和主要成果。

论文摘要的书写应力求精确、简明。切忌写成对论文书写内容进行提要的形式，尤其要避免“第 1 章……；第 2 章……；……”这种或类似的陈述方式。

本文介绍清华大学论文模板 ThuThesis 的使用方法。本模板符合学校的本科、硕士、博士论文格式要求。

本文的创新点主要有：

- 用例子来解释模板的使用方法；
- 用废话来填充无关紧要的部分；
- 一边学习摸索一边编写新代码。

关键词是为了文献标引工作、用以表示全文主要内容信息的单词或术语。关键词不超过 5 个，每个关键词中间用分号分隔。（模板作者注：关键词分隔符不用考虑，模板会自动处理。英文关键词同理。）

关键词：TEX；LATEX；CJK；模板；论文

Abstract

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summarization of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words “the first chapter”, “the second chapter” and the like should be avoided in the abstract.

Key words are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 key words, with semi-colons used in between to separate one another.

Key words: TEX; LATEX; CJK; template; thesis

目 录

主要符号对照表

HPC	高性能计算 (High Performance Computing)
cluster	集群
Itanium	安腾
SMP	对称多处理
API	应用程序编程接口
PI	聚酰亚胺
MPI	聚酰亚胺模型化合物，N-苯基邻苯酰亚胺
PBI	聚苯并咪唑
MPBI	聚苯并咪唑模型化合物，N-苯基苯并咪唑
PY	聚吡咙
PMDA-BDA	均苯四酸二酐与联苯四胺合成的聚吡咙薄膜
ΔG	活化自由能 (Activation Free Energy)
χ	传输系数 (Transmission Coefficient)
E	能量
m	质量
c	光速
P	概率
T	时间
v	速度
劝学	君子曰：学不可以已。青，取之于蓝，而青于蓝；冰，水为之，而寒于水。木直中绳。輮以为轮，其曲中规。虽有槁暴，不复挺者，輮使之然也。故木受绳则直，金就砺则利，君子博学而日参省乎己，则知明而行无过矣。吾尝终日而思矣，不如须臾之所学也；吾尝跂而望矣，不如登高之博见也。登高而招，臂非加长也，而见者远；顺风而呼，声非加疾也，而闻者彰。假舆马者，非利足也，而致千里；假舟楫者，非能水也，而绝江河，君子生非异也，善假于物也。积土成山，风雨兴焉；积水成渊，蛟龙生焉；积善成德，而神明自得，圣心备焉。故不积跬步，无以至千里；不积小流，无以成江海。骐骥一跃，不能十步；驽马十驾，功在不舍。锲而舍之，朽木不折；锲而不舍，金石可镂。蚓无爪牙之利，筋骨之强，上食埃土，下饮黄泉，用心一也。蟹

六跪而二螯，非蛇鳝之穴无可寄托者，用心躁也。——荀况

第 1 章　绪论

1.1　研究背景

在软件开发过程中，代码错误的发现与修正贯穿始终。近年来大量错误检测工具（如静态分析工具、动态分析工具、代码验证工具、测试管理工具等）在工业界得到了较好的应用，这些工具大大提高了开发人员发现代码错误的效率，然而这些错误仍然需要人工分析和修复。研究表明，错误的定位错误的定位与修复最高可占用软件开发过程中 70% 的时间如何在错误修复这一环节提供工具支持并提高效率已成为软件工程研究的重要内容。

基于“生成-检验”框架的错误自动修复技术是众多错误修复技术中的一个分支。该技术从程序源代码出发，以源码自带的测试集是否通过为判别程序正确与否的标准，试图生成针对源码的修改建议，使其修改后能够通过测试集，从而达到修复错误的目的。

图**??**展示了“生成-检验”框架的一般结构和工作流程。如图所示，“生成-检验”系统的输入包括程序源代码、测试代码和测试结果（如通过、不通过、错误路径等），内部共包含“错误定位器”，“搜索引擎”和“检验器”三个主要模块，分别对应“错误定位”、“变体生成”和“修复检验”三个主要工作步骤，最终系统输出一系列可能的修改建议，使得源代码经修改后可以通过测试代码中的测试。具体而言，首先，错误定位器将分析程序的运行轨迹，找出源代码中与测试错误相关的部分，称为“可修改位置”，并将各个位置按照出错的可能性由高到底排序，形成“可修改位置”列表。接着，搜索引擎按照这一列表由高到低的顺序针对各个位置生成可能的代码修改方案。最终，检验器将这些修改方案逐一应用到源代码中，重新编译并执行测试代码。当测试代码通过时，被检验的修改方案输出给开发人员，通过人工判断决定采用哪一种修改方案。

Listing 1.1　“生成-检验”框架应用示例

```
1 public class Calculator {
2 public int add (int a, int b) {
3 return a + b;
4 }
5 public int minus (int a, int b) {
6 return a + b; // Error: should be (a - b)
```

```
7 }
8 }
```

Listing 1.2　“生成-检验”框架应用示例

```
1  public class CalculatorTest {
2  @Test
3  public void testAdd () {
4  Calculator cal = new Calculator();
5  int a = 1;
6  int b = 2;
7  Assert.assertEquals(3, cal.add(a, b));
8  }
9  @Test
10 public void testMinus (int a, int b) {
11 Calculator cal = new Calculator();
12 int a = 1;
13 int b = 2;
14 Assert.assertEquals(-1, cal.minus(a, b));
15 }
16 }
```

例如，上图展示了一个简单 Java 程序的源代码。图 1 是主程序，它包含一个类 Calculator，并提供两个公有方法 add 和 minus，分别处理加、减两种操作。图 2 是类 Calculator 对应的 JUnit 单元测试类 CalculatorTest。测试类中包含两个测试方法，分别测试加、减两种操作的正确性。读代码可知，图 1 第 6 行中返回值表达式有一处错误：减法操作本应返回两个表达式相减，而实际返回了两个表达式相加。实际运行测试代码时也可看到，测试方法 testMinus 将在图 2 第 16 行触发 AssertionFailureException，表示期望返回值（-1）与实际返回值（3）不符。

为修正这一错误，“生成-检验”系统将首先定位与异常发生相关的程序语句，此处即为主程序第 6 行。接着，系统将对第 6 行做合理的程序变换，例如将表达式 a + b 变为 a - b, a + 1, 1 + b, -a, -b... 最后，系统将这些变换代回源代码，重新编译并执行测试，此时可发现若是用 a - b 或-a 替换 a + b，则两个测试用例均能够通过，因此两种修改方式均会提交给开发人员做人工判断。不难看出，将 a + b 替换为-a 使测试集通过仅仅是巧合，正确的修改应当是替换为 a - b。至此，程序错误

被修复。

评价一个“生成-检验”系统的优劣应从两个维度出发。一是“正确率”，即在相同代码错误集合上能够成功修复的代码错误比例，二是“效率”，即修复同一代码错误所消耗的时间。从工作流程上看，如果不限时间，“生成-检验”系统的修复能力，即其所能够成功修复的错误范围完全由其搜索引擎中所包含的程序变体生成模板集合决定。模板集合越大，所能覆盖的程序错误范围越广，能够生成出正确修改方案的可能性就越高，系统正确率也越高。这使得“生成-检验”系统在理论上可以修正任何由测试集定义的代码错误而不局限于特定的错误类型。然而，由于程序变体数量巨大，实际的计算过程中难以穷尽，如何在保证一定的正确率前提下尽可能提高系统的效率成为了实际系统设计与实现中的重要问题，也是本文的中心内容。

错误定位算法、搜索引擎及检验器对系统效率具有如下影响：

1. 错误定位的结果决定了源代码中发生错误的代码位置被搜索到的顺序，因此它也直接决定了系统将在生成一个正确的修改方案前所花费的时间。错误定位的结果越准确，耗费时间越少，效率越高。
2. 搜索引擎中包含了一系列预定义的程序变体生成模板。对所有可能的代码修复位置，搜索引擎均会根据这些生成模板生成一系列的程序变体。由于所有修复方案均需要输入到检验器做测试，模板所覆盖的变体生成方案越多，检验过程消耗的时间也越长，效率也就随之降低。
3. 在检验器检验一个备选修复方案是否能够使得源程序测试集通过时，源程序将被修改、重新编译并运行，这一过程将在编译、运行环节上耗费大量时间。因此检验器的设计和实现将直接影响系统的整体效率。

基于以上分析，本文将从提高错误定位算法准确度、压缩搜索引擎的搜索空间、提高检验器检验修复方案等角度提高系统效率，优化系统设计，使“生成-检验”系统向实际应用更进一步。

1.2　研究现状

现有研究工作的研究内容可分为两大类。一是“生成-检验”框架中的某个具体工作步骤，即错误定位算法，修复建议生成算法及修复建议验证算法的研究。二是对“生成-检验”框架的整体改进。本节本文将对这几个方面的研究现状分别阐述。

1.2.1　错误定位算法

错误定位是“生成-检验”框架中的第一个计算步骤。事实上，错误定位算法成为一个独立的研究领域远早于“生成-检验”框架的提出。在本节我们将分为两个方面介绍错误定位算法的研究现状。

算法设计：错误定位算法的根本目的是根据程序的出错信息找到程序源代码中的错误位置，方便开发人员发现错误。根据算法的基本思想，现有的错误定位算法可以分为以下两大类。第一类是基于程序数据流、控制流分析的错误定位算法，如[]

另一类是基于统计的错误定位算法（Spectrum-based Fault Localization），如。。。。这一类算法根据程序测试集中测试用例的覆盖（Coverage）信息，依据“被越多的通过测试用例执行次数越多的语句越可能是正确的代码，被越多的不通过测试用例执行次数越多的语句越可能是错误的代码”这一经验规律，设计出一系列输入为执行次数，输出为程序语句错误的概率估计值的计算公式，最终根据该公式的计算结果按可疑值（Suspiciousness）由高到低排序，给出程序中最可能导致程序错误的语句列表。目前比较公认的计算公式是[?]，很多实验[?] 都表明该公式给出的排序准确度能够稳定的超过其他公式，达到...???, 即。。。SFL 方法的优点是算法简单，速度快，并且由于很多语言已经提供了一定的插桩支持工具（如针对 C 语言的 gcov，针对 Java 的 JProfiler 等）工程实现也比较容易。然而缺点也十分明显，由于经验公式给出的结果具有一定的随机性，这类算法并不能够保证在所有程序错误上都获得较准确的计算结果，一些研究工作表示，如果计算结果不够准确，这类算法实际上会加长开发人员查错改错的时间，降低开发效率。

算法分析：错误定位算法的理论分析工作主要集中在对 SFL 的理论分析上。[?] 以一个简单分支结构程序对 30 余个 SFL 计算公式进行了概率分析，其结论是共有？个计算公式在理论上超过其他现有公式。然而[] 等人的实验结果与上述理论分析矛盾，其原因可能是实际程序中包含循环、递归等无法完全用顺序与分支结构来描述的逻辑结构。针对实际程序的 SFL 理论分析仍有待进一步研究。

1.2.2　修复建议生成

修复建议生成算法是“生成-检验”框架的核心算法。算法设计需要解决的核心矛盾搜索空间规模与搜索时间之间的冲突。从算法设计的角度，现有的研究工作可以在定位于以下坐标系：

以下将分类介绍现有搜索算法的基本思想及其实验效果。

基于搜索算法 GenProg[?] 是较早的“生成-检验”系统，它首先提出了基于遗

传算法的修复建议生成算法。该算法基于“代码相似性”假设，即程序中的错误通常可以通过用程序中其他位置的代码修补或替换错误代码得到修正。因此 GenProg 将程序中的代码元素，如表达式、语句等，按照语法规则组合在一起，形成种群，并套用遗传算法框架生成后代。该算法在一套 C 程序测试集上[?]能够修复?? 个错误。在此基础上,[?]发现实现更简单的随机搜索算法能够取得与遗传算法相比类似的修复效果。这一类算法的优点是，搜索空间较广，同时也利用了待修复程序本身的代码特点使得修复更有针对性。然而较大的搜索空间也带来了大量的计算，系统效率较低。

基于模式库： Kim et al[?]首先提出了基于模式库的修复建议生成算法。在[?]中，该团队分析了？个开源代码项目中的代码和错误修改日志，提出了 10？条常见的错误修改模板（见表**??**）。在此基础上，他们实现了系统[?], 并在一套 Java 程序测试集上?? 进行了测试。据论文中报告，在实验中 Par 能够修复？中的？个错误。该算法的优点是，相较于 GenProg，其搜索空间较小，生成出的修改建议也较贴近开发人员的修改习惯，容易读懂。缺点是，模板定义范围并不广，因此系统的正确率较低。几年后发表的研究工作[?]就指出 Par 的实际修复正确率远低于其论文中的实验数据。

基于程序综合： 前两类算法在生成修复建议时，并没有它们对程序语义的影响，因此会生成出大量的无用建议，直到验证阶段才被滤除。基于程序综合的生成算法则较好的避开了这一点。例如,[?]试图修改代码中错误的 if 条件。其基本思想是，在程序执行过程中动态修改某个位置上 if 条件的布尔值使得程序能够通过测试，接着分析在这一位置上所有能够访问到的变量及其所能构造出的布尔表达式的取值情况。一旦找到某个布尔表达式的值能够符合修改后的布尔值，则该表达式可以作为 if 条件表达式的修复方案。这一方案能够在 Defects4J 测试集上修复?? 个错误。[?]将修改范围扩大到一般表达式，它首先利用符号执行技术获得待修改表达式为使程序通过测试用例的所应满足的约束条件，接着以该表达式位置所能使用的变量和函数作为材料综合出符合约束的表达式。[?]更进一步的将错误定位与约束求解融合为一个步骤，使得生成出的表达式尽量简单易读。[?]则。。。基于程序综合技术的优点是，生成出的表达式通常能够具有较好的修复成功率，因此为检验这一计算步骤省去了大量的时间。然而，无论是通过动态修改变量值还是通过符号执行技术获取修改目标表达式所应满足的约束条件都十分耗时。这也解释了为何[?]将修改限定为 if 条件，并且[?]directfix 的实验程序规模都比较小。

基于逻辑公式[?]

多种技术融合由 MIT 实验室开发并陆续发表的 SPR 系统在修复建议生成这一

环节融合了上述提出的多重技术。SPR 是 Staged Program Repair 的缩写，顾名思义它的核心思想是将修复建议生成过程分为几个阶段逐步进行。与 Par 类似，SPR 的搜索空间也由一组修复模板定义。而对这组模板中与修改 If 条件相关的模板，SPR 将修复建议生成过程分为几个阶段。首先，与 Nopol 类似，SPR 为某个 If 条件表达式动态赋值。但与 Nopol 不同的是，在程序每次运行到该表达式时，SPR 会尝试赋不同的值（True/False）。与此同时，SPR 记录下该表达式的哪一个取值序列能够使得程序通过测试。当获得了某个取值序列后，SPR 在一组合法的表达式中过滤出取值与该序列相符的表达式作为 If 条件。实验表明，这一步过滤将去除多达 90% 的备选表达式，从而大大压缩搜索空间，提高系统的整体效率。另外，SPR 的修复模板覆盖了 Par、GenProg 中的变换模式，因此其修复成功率也较高，在 GenProg 测试集上达到了 19/63??。

1.2.3　修复建议验证

所有搜索引擎生成出的修复建议均需逐一应用回程序代码并重新编译测试，而当被测程序规模较大时这一过程将非常耗时。高效的修复建议验证算法对提高系统整体效率具有重要意义。MIT 实验室在 SPR 基础上开发了系统 Prophet，其主要工作是用机器学习方法训练一个计算模型，使得系统能够通过语法结构特征计算各个修复建议候选方案被验证的优先级。实验表明，经过优先级计算排序后，正确的修复方案会被优先验证，这缩短了人工判断最终的修复方案是否正确的时间，从另一角度提高了系统的整体效率。

1.2.4　框架扩展

1.3　研究思路

论文将在“生成-检验”框架内，从多角度解决以上提出的两个问题。

1.4　论文贡献

1) 提高错误定位准确性的技术：Oracle debugging；

2) 本文提出“预过滤”技术优化搜索引擎，压缩搜索空间，提高搜索与验证效率，

3) 本文针对现有修复框架的缺陷提出两种扩展方式：a)“交互式调试”设计模式，即给用户提供方便的图形界面，使其能够依据自己的经验将调试任务分段处理。b) 针对单类别错误的可扩展框架。本文整理了“生成-检验”框架，使其能够

方便的融合针对单类别错误的修复算法，从而扩展修复工具的修复能力，使其能够利用已知类别错误的特异性提高修复准确率和效率。

4) 系统实现 SmartDebug。SmartDebug 以插件形式集成在 Eclipse 集成开发环境中，可供下载使用和二次开发。

1.5　论文结构

第 2 章　基于统计的错误定位

2.1　引言

错误定位是“生成-检验”框架的第一个工作步骤，基于统计的错误定位（Spectrum-based fault localization，简称 SFL) 算法因其实现简单、效果良好被绝大多数现有系统采用。SFL 的基本思想是，利用“被通过的测试用例执行次数越多的语句越可能是正确的，被不通过的测试用例执行次数越少的语句越可能是错误的”这一经验规律，通过统计各个程序语句在各个测试用例中的执行次数，按照某一公式计算各个语句出错的可能性，并将语句按此排序，最终开发人员可以按照此排序逐一检查程序语句。实践表明，通常真正错误的语句会被排在非常靠前的位置，因此开发人员能够省去查看其它正确语句的时间。

SFL 计算结果的合理性依赖软件工程中一项称为“测试准则假设（Test Oracle Assumption）”的基本假设，其内容是在测试过程中，总存在一种判断机制，或者称作测试准则（Test Oracle），能够准确的判断被测程序是否正确的执行了一条测试用例[1]。该假设被广泛接受，已经成为许多软件工程问题的分析基础。然而在实际开发过程中，完全正确的测试准则是很难获得的。面对越来越复杂的软件系统，现有的测试准则的自动生成技术远远无法满足需求[2]。事实上，对测试结果进行人工检查仍然是工业界广泛采用的工作方法[3]。换言之，人工判断成为了实际应用中的“测试准则”。

使用人工检查作为测试准则会引发严重的问题。人工检查很容易出错[3]，因此我们所承认和依赖的“测试准则假设”不复存在，测试人员实际上并不能够完全正确的判断被测程序是否真的通过了一个测试用例。这种判断错误在许多情况下都可能出现。例如，对于复杂的软件系统，测试人员很可能无法对程序的所有部分都有清晰准确的理解和认识，因此判断一个具体的测试结果也有一定的难度。另外，当程序的行为比较复杂时，通常需要一个非常大的测试集，由于测试用例基数大，判断失误不可避免。在文章[4] 中作者提到，当测试用例的输入是由一些测试用例生成工具产生时，工具所生成处的测试输入可能比较抽象（如无意义的字符串），测试人员难以直观理解，这也为判断测试结果的正确性带来了难度。以上这些场景在实际开发过程中非常常见，因此开发人员在日常工作中所使用的测试准则往往是有错的。

SFL 计算公式中的输入数据直接来自测试准则判断结果，因此测试准则错误将使 SFL 计算结果精度下降。事实上，由本章第 3 节中的实验数据可见，精度越

高的 SFL 公式对准则错误越敏感。因此，若要使 SFL 在“生成-检验”系统中发挥作用，则需要尽量消除测试准则错误对 SFL 带来的负面影响。

想要消除这一负面影响的一种直接的办法是，抛弃原有的测试准则并重建一套新的测试准则。然而，新的测试准则可能并不比上一套正确率更高，而重建也需要耗费大量的时间和资源。再者，原有的测试准则虽然含有一定的错误，但是其中绝大多数的判断仍是正确的，具有利用价值。因此一个更好的思路是设计一种算法，能够直接改正在原有测试准则上的错误。在本章中，我们提出一种测试准则的自动化纠错技术，它能够发现测试准则做出的错误判断，使得 SFL 计算结果尽量不受影响。

本章提出的测试准则自动化纠错技术是是基于以下观察：通过相似测试路径的测试用例通常具有相同的测试结果（通过、不通过）。这一观察实际来源于基于覆盖率的测试集压缩技术[5][6]。我们试图给出测试用例之间相似度的度量方法，并识别出与其近邻测试结果明显不同的测试用例，将其标为“疑似错误”，并在 SFL 的输入中将其测试结果取反（及将“通过”改为“不通过”，反之亦然）。这一处理过程计算简单，但实验证明效果良好。

为全面评价这一技术，我们采用了软件基础设施库（Software Infrastructure Repository[7]）中的两组实际程序作为测试对象，第一组是西门子测试集（Siemens Test Suites），它被错误定位算法设计研究工作广泛采用。第二组是 `grep` 是 Unix 系统下的实际程序，其代码规模达到 13,000 行。我们首先为西门子测试集中的 7 个测试程序的 132 个变体生成了错误率在 0.01-0.1 之间的测试准则（即测试准则判断测试结果出错的概率在 0.01-0.1 之间）。同时我们也为 4 个版本的 `grep` 程序生成了相同错误率的测试准则。实验表明，经过自动化纠错处理，所识别出的“疑似错误”测试结果有 75% 是真正的测试准则错误。此外，将 SFL 应用于被纠正后的测试结果时，几种 SFL 算法的精确度相较使用未纠错的测试结果均有了较大幅度的提高。

本章的主要贡献包括：

- 在西门子测试集上，以实验证明测试准则的错误将有损 SFL 技术的错误定位准确度。这是目前已知第一项在测试准则出错前提下分析 SFL 技术精度的研究工作。
- 提出一项简单而有效的自动化测试准则纠错算法。实验表明，该算法能够识别出测试结果判断错误中的 75%，而经过纠错的测试结果也使得 SFL 算法精度大幅恢复。
- 定量分析不同类型的测试准则判断错误（如“假通过”和“假失败”）对 SFL

精度的影响。

本章余下内容的组织结构如下：第二节总结相关工作，第三节以实验数据阐明测试准则纠错的必要性，第四节介绍测试准则纠错算法，第五节展示算法的实际效果，并对实验结果进行深入分析，第六节总结本章工作。

2.2 相关工作

本章工作的核心内容是测试准则纠错算法，至今这是第一项探究测试准则错误与 SFL 精确度之间关系的研究工作。与之联系紧密的研究内容可分为如下三类：（1）对各个 SFL 算法精确度的测评，（2）分析与 SFL 算法精度相关的因素及其对 SFL 的影响，（3）测试准则自动生成技术研究。以下将按类别逐一介绍这些研究内容。

2.2.1 SFL 算法精度测评

测评 SFL 精度的工作已有很多，包括实验分析和理论分析两类。[8] 将 Tarantula 算法与之前的错误定位算法如 Set union, Set intersection, Nearest neighbour 和 Cause transitions 在测试集上的实验结果进行对比分析。作者得出的结论是 Tarantula 在诸多算法中取得了最好的效果。[9] 对四种 SFL 算法中用到的计算公式进行测试，得出结论是 Ochiai 在所有被测程序中一致的优于 Jaccard 和 Tarantula，Jaccard 不弱于 Tarantula，而 AMPLE 的精度与其他公式相比时优时劣。以上两篇工作都以 Siemens Test Suite 作为目标程序。更进一步，[10] 将 Ochiai, Tarantula，Jaccard 和其他 6 个计算公式在 Siemens Test Suite 和 `space` 程序上做对比测试实验，数据显示 Ochiai 精确度最高。Naish et al.[11] 对 41 种计算公式进行理论分析，最终将其按精确度划分为 6 组，其中每组内公式在单一错误前提，即程序中只有一个错误的前提下理论上都是等价的。Xie et al.[12] 分析了 30 条计算公式并将其按精确度划分为 14 个等价类。该研究进一步分析了在单一错误前提下几个等价类之间的优劣关系，证明了两组公式 ER1 (包括 Naish1, Naish2 两个公式) 和 ER5 (包括 Binary, Russel&Rao, Wong1 三个公式) 是理论上的最优公式。在此结论基础上 [13] 对 Tarantula，Ochiai 以及 ER1 和 ER5 中的公式进行了实际实验。实验结果表明，Ochiai 的实测精确度比 ER1 和 ER5 两组高，而 Tarantula 的精确度平均值也优于 ER1 中的 Naishi1 和 ER5 中的全部算式。这与 [12] 中的理论分析矛盾，作者分析可能的原因是理论分析中假设代码覆盖率为 100%，而实际程序中覆盖率达不到这一数值。本文2.3.2.4节中对几个 SFL 计算公式的测试数据考虑了测试准则出错的可能性，为以上工作做了补充。

2.2.2　影响 SFL 精确度的因素

研究表明，SFL 算法的精度受许多因素的影响，对这些影响进行定性或定量的分析也是 SFL 算法的研究内容之一。在[10]，作者研究了测试用例总数、通过的测试用例数与未通过的测试用例数对 SFL 精度的影响。分析结果表明，想要获得接近最优的精确度，即将真正错误的代码行排在前 20% 仅需有限的测试集规模就可达到。在[14] 中，作者研究了使用 10 种测试集压缩策略缩减测试集对 SFL 精确度的影响。研究表明，SFL 算法的有效性确实会受到测试集压缩策略的影响。[15] 讨论了四种与测试或测试集相关的场景，并首次提出了两个概念：（1）“巧合通过”（Coincidental correctness），意指在一次测试中程序出错条件已经触发，但是实际执行过程中测试却并没有失败的情形，（2）“弱巧合通过”（Weak coincidental correctness），指出错的语句被执行到，但测试结果最终并没有出错的情况。基于以上观察，后来的研究者提出使用测试聚类[16] [17] 和基于上下文模式的覆盖率精化方法[18] 消除这两种情况对 SFL 精度的影响。注意，这里的“（弱）巧合通过”是测试准则的正确判断，与本章工作中关注的测试准则错误并不相同。[19] [20] 注意到软件开发过程中产生的错误报告会被分到错误的类别中，这种分类错误也会对错误定位和预测产生影响。这与本章工作的相同点在于，他们并不假定人工提交的错误报告在人工判断类别时完全正确，但是这两篇工作关注的错误定位粒度在文件级，而非代码级。

2.2.3　测试准则的自动生成

在软件测试过程中，测试准则试判断程序是否运行正确的唯一标准。然而，自动生成可靠的测试准则并不简单。尽管以人工判断作为测试准则仍然常见于工业界，现在已有越来越多的研究工作尝试自动生成可靠的测试准则，节省开发时间。

测试准则的自动生成方法可划分为两类。第一类是生成显式的测试准则。例如，在[21] 中，作者提出了一种自动判断 GUI 操作响应正确性的方法。被测 GUI 程序首先被建模为一组对象的集合，同时每个对象的属性和动作及对动作的反应都被形式化的描述。在执行测试程序时，系统可以自动的判断程序的实际行为是否满足之前的形式化描述。尽管最后的系统判断是完全自动的，系统仍然依赖开始时对 GUI 程序的形式化描述，并且由于 GUI 程序的特殊性，这一技术很难扩展到其他类型的程序中。在[22], 作者研究了将人工神经网络（Artificial Neural Networks，缩写为 ANN）用作三角形分类问题的测试准则的正确率。作者最终得出的结论是，ANN 的判断出错概率约为 19.02%，而这一准确率达到了实用的要求。然而在本章第 5 节的实验数据显示，准确率仅为 80.98% 的测试准则在实际应用中会给程序调

试带来严重的影响。将这一方法推广到其他程序也具有一定的困难。另一种有趣的思路是从软件文档中直接生成测试准则[23]。然而，使用这种方法的前提是软件文档按照文中定义的书写规范精确地定义了软件系统的行为，因此对于已经开发很长时间的遗留系统很难适用。

第二类工作避免了显示地生成一组测试准则，而以其他策略判断被测程序的测试结果。在[24] 中, Davis et al. 首先提出对不容易测试的程序可独立开发满足同一设计规约的多个版本程序作为“假测试准则（pseudo oracles"。作者建议使用两个或多个版本的程序运行同样的测试用例，当测试输出不相同时，可使用某种投票机制决定哪个结果是正确的。显然，与第一类策略相比，使用这种策略将引入大量的编码工作。一个改进的策略发表于[25]，作者提出其他独立版本的程序可以使用代码自动生成技术来开发，而此时引入的额外工作仅仅是对程序行为进行形式化的描述。这显然是上一工作的重要改进，可惜的是并非所有的程序都适合用形式化规约来描述其行为，因此其适用范围也比较有限。在测试准则不易获取的情况下，蜕变测试（Metamorphic testing）也是常用的测试方法。蜕变测试不依赖形式规约，它的基本思想是将程序按某种规则变换后判断其输出结果的变化是否符合预期。在[26] 和[27] 中，作者提出可以将蜕变测试与使用符号输入的基于错误的测试相结合???，但没有阐述具体的系统实现及其实验效果???。接下来,[28] [29] 推出了工具 Amsterdam，它可以自动化蜕变测试过程。遗憾的是，尽管测试过程可以自动化执行，测试人员仍然需要人工描述蜕变测试中程序所应满足的属性，例如输入应如何变形，而输出应具有怎样的变化。在[30] 中，作者声称测试过程已经完全自动化了，其中包括测试准则的自动化生成。而事实上，测试准则仍然存在，只是转化为了使用 Java 建模语言（Java Modeling Language，简称 JML）描述的断言（Assertions）、前置和后置条件（pre- and post-conditions）及类不变量（Class invariants）。严格来讲，这些工作仍然是人工完成的，因此测试过程并不能够称为“完全自动化的”。

总之，自动生成测试准则技术尚未成熟，这也是本章工作的主要动机之一。在我们尚不能够完全自动的生成测试准则时，应当尽量充分的利用不完善的测试准则库，使其能够对错误定位和修复产生正面作用。

2.3　测试 Oracle 纠错的必要性

测试准则错误指的是对测试结果的一项错误的判断，例如程序在运行某一测试时实际执行正确，但测试准则却判定其执行错误，测试不通过，或者相反。这种情况会可能会令开发人员非常费解，不可避免的花费许多时间理清其中的问题，

表 2.1　Debugging Process of a Sorting Program

Input: (a, b, c)	(1, 2, 3)	(1, 3, 2)	(2, 1, 3)	(2, 3, 1)	(3, 1, 2)	(3, 2, 1)	Sus_c	Sus_e
1 `if(a>b){`	•	•	•	•	•	•	0.5	0.5
2 `swap(a,b);`			•		•	•	0.5	1.0
3 `if(b>c){`	•	•	•	•	•	•	0.5	0.5
4 `swap(b,c);`	•			•	•	•	1.0	0.6
5 `if(a<b)`	•			•	•	•	1.0	0.6
6 `swap(a,b);`	•				•		1.0	0.3
7 }							–	–
Oracle(correct)	×	✓	✓	×	×	×		
Oracle(error)	✓	✓	×	×	×	×		

例如需要跟踪程序的执行过程，查看、确认是被测代码的问题还是测试代码的错误。如果使用自动错误定位算法，算法的准确度也受到影响，使得开发人员无法充分利用这些技术带来的便利。下一小结的例程对这一情况将有更好的解释。

2.3.1　例程

表2.1展示了一个简单的排序程序及其配套的一组测试用例。程序的目的是将输入的三个数值变量 a，b，c 按照从小到大的顺序排序。开发人员在程序的第 5 行（标红）不小心写错了 if 条件判断表达式，本应是判断（a>b）但实际程序中比较符号方向写反了。这一组测试用例包含六组输入数据，分别覆盖了常见的六种变量间的大小关系。我们将六组测试用例对程序语句的覆盖情况用点号和空格标识，例如第一组数据覆盖了程序的第 1 行和 3 6 行，则在这项行号上标识点号。表格的最后两行给出了两组测试准则对各个测试用例通过与否的判断，打钩表示通过，叉号表示不通过。第一组对测试通过与否判断正确，而第二组对第 1 组和第 3 组测试的判断错误（标红）。

在调试过程中，开发人员可以借助自动错误定位技术的帮助来节省调试时间。自动错误定位技术将代码行按照出错的可能性进行排序，开发人员可依照这个排序列表逐一检查代码行，省去查看不太可能出错的位置花费的时间。基于频谱的错误定位（Spectrum-based Fault Localization，简称 SFL）是一类轻量级的错误定位算法。其输入是测试用例的执行路径（称为频谱）以及测试结果（称为诊断）。据此，SFL 对每一行代码 c 统计以下四项数据，覆盖 c 且通过的测试用例数 a_{ep}，覆盖 c 且未通过的测试用例数 a_{ef}，未覆盖 c 且通过的测试用例数 a_{np} 及未覆盖 c 且未通过的测试用例数 a_{nf}。根据这些统计数据，SFL 使用一条计算公式估计代码行 c 的“可疑程度”，即其是错误代码行的可能性。最终，可疑程度高的代码行被排

在靠前的位置。

例如，知名的 SFL 算法 Tarantula[31] 的计算公式如下：

$$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$$

Tarantula 的基本思想很简单：对于任意一行代码 c，覆盖 c 且失败的测试用例比例越高，c 越有可能是错误的代码行。在表2.1的例子中，我们将 Tarantula 依据正确和错误的测试准则判断计算出的可疑程度值分列在表格的最后两列。当使用正确的测试准则时，Tarantula 指出具有最高可疑值的代码行是第 4，5，6 三行。假设开发人员完全依照这一排序进行检查，那么他应该随机的按任意顺序检查这三行，平均来看他将在检查第二个代码行时检查到第 5 行并发现错误。但是如果使用错误的测试准则，Tarantula 会将第 2 行排在最靠前的位置，于是开发人员在检查第 4 行和第 5 行前需要先检查第 2 行。于是，平均来看他将在检查第 2.5 个代码行时遇到第 5 行并发现错误。

由上例可以看出，Tarantula 的性能由于测试准则的错误受损。尽管多检查 0.5 行看起来并不是非常严重的性能损耗，但是这只是一个 7 行的小程序。由下一小节的实验数据可以看出，由于测试准则出错带来的性能损耗百分比实际上与程序规模关系不大。对较大规模的程序，这些损耗将使得系统的整体效率明显降低。

2.3.2　在西门子测试集上的实验

为了对测试准则带来的定位精度损耗有更清晰的认识，我们在西门子测试集（Siemens Test Suites）上进行了实验。西门子测试集包括 7 个不同的程序以及他们的变体共 132 个。有关该测试集在2.3.2.3小节有更详细的数据介绍。在此测试集上，我们选取了 4 种具有代表性的 SFL 算法，分别统计它们在正确和错误的测试准则下的定位精确度并加以比较，分析其精度受损程度与测试准则错误率之间的关系。

2.3.2.1　定义 SFL 的算法精度

为后续的清晰讨论，本节我们给出 SFL 算法定位精度的详细定义。当被测程序只有一个错误时，定位精度可以被自然地定义为在检查到真正错误的代码行之前所需检查的代码占所有代码行总数的比例。但是实际程序中的错误往往不止一个，在此情况下，我们认为调试过程符合[32] 中提出的“一次一处”模式，即检查到第一处错误时就可将该处错误改正，接着进行下一处错误的改正，因此将 SFL 的精度定义为系统在遇到第一个错误之前所需检查的代码行数比例。

如果有多条代码计算出的可疑值相等，我们假定后续的系统会采取某种策略对排位相同的代码行内部排序。在本章中我们仅考虑测试准则错误带来的影响，因此假定后续系统将采取随机策略选择具体查看哪一行代码，因此平均来讲，真正错误的代码行将在中间位置被查看到。基于此，对 SFL 精度的量化定义如下：

对于一个有 N 行代码的程序，令 $\{l_{f1}, ..., l_{fk}\} \subset \{1, 2, ..., N\}$ 表示具有最高可疑值的错误代码行，令 sus_j 表示行 j 的可疑值，我们定义某一 SFL 算法的“分数”s 为

$$s = \frac{|\{j | sus_j < sus_{l_{f1}}\}|}{N} + \frac{|\{j | sus_j = sus_{l_{f1}}\}|}{k+1} \tag{2-1}$$

同时，我们定义 SFL 算法精度为 $acc = 1 - s$。显然，精度 acc 越高，分数 s 越低，算法的性能就越好。由以上定义可以看出，s 的定义更加直观且与精度具有同等意义，在本章后续内容中我们将围绕 s 的数值进行讨论。

2.3.2.2　SFL 计算公式选择

不同的 SFL 算法主要区别在于其所设计的可疑值计算公式。目前为止研究者已经提出了超过 30 余个不同的计算公式，其定位准确度也各不相同。由于篇幅有限，本章只选取了其中最具代表性的部分公式进行实验研究。

为了保证公式选取的代表性和合理性，我们参考了已有的实验评估报告和理论分析工作。两份实验评估报告[14] 和[10] 均指出 Ochiai 一致的优于 Jaccard 和 Tarantula。在之后发表的理论分析工作[12] 中，作者 Xie et al 也证实了 Ochiai 优于 Jaccard，而 Jaccard 优于 Tarantula。作者们还将 Naish1 和 Naish2 归为一组等价公式 ER1，将 Wong1，Russel&Rao 和 Binary 归为另一组等价公式 ER5，并证明这两组公式理论上优于 Ochiai 等其他公式，因此他们称这两组公式具有“极大精确度”。然而，实验评估报告[13] 指出在实际测试中，Ochiai 的性能好于 ER1，并且

表 2.2　不同 SFL 算法中的计算公式

公式名	公式
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{nf}+a_{ef})(a_{ef}+a_{ep})}}$
Naish2	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
Russel&Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$

Tarantula 的表现也优于 Naish1 和 ER5。作者分析存在这一出入是优于理论分析文章[12] 中假定测试集的覆盖率达到了 100%，而实际程序很难满足这一点。

本文试图在有限篇幅内对现有的 SFL 计算公式给出相对全面的评测和分析，因此选择 SFL 公式的标准如下：

- 被选出的公式应当同时覆盖目前广泛使用的公式以及（在使用正确的测试准则时）性能最优的公式。
- 被选出的公式应能够应用于程序存在单一错误和多个错误的不同场景
- 为便于结果可靠性检查，被选出的公式最好曾被其他研究工作评测过。

对比以上标准，我们从已有研究工作中提出的 SFL 算法中作出如下选择：我们首先在 ER1 中选取 Naish2，在 ER5 中选取 Russel&Rao 两个公式作为理论分析中具有“极大精确度”计算公式的代表。选取这两个公式的原因是，Wong1 与 Russel&Rao 公式是等价的，最终 SFL 的输出结果也一定完全一致。而在证明 Naish1 和 Binary 两个公式的最优性时都假定了程序只有单一错误，因此不能作为 ER5 中的代表。另外，我们选取了 Ochiai 和 Tarantula 作为实验评测中性能较好的公式代表。其中 Tarantula 提出时间最早，相关研究相比于 Jaccard 也较为广泛，因此没有选取 Jaccard。至此，我们选择了 4 种 SFL 计算公式，包括 Tarantula，Ochiai，Naish2 和 Russel&Rao。具体的计算公式参见表2.2.

2.3.2.3　实验设计

为了量化测试准则错误带来的影响，我们需要同时获得被测程序的测试准则的正确和错误版本。然而直接获取实际程序完全正确的测试准则很难，一种迂回的办法是以一个已知正确的程序的运行结果作为测试准则。基于此我们找到了 SIR 提供的西门子测试集。

西门子测试集是错误定位领域用于评价各种算法准确度所广泛采用的一个测试程序库。表2.3列出了测试程序库中的内容。程序库中共包含 7 个不同的真实 C 程序，对每个程序，SIR 提供了正确的程序作为基准，同时也提供了一组改程序的错误变体。程序中的每个变体均人工植入了一个错误，但有些类型的错误实际上会影响程序中的多个位置。例如，C 语言中的宏（#define）定义值错误，在编译阶段宏替换可能会导致多个位置的值错误。另一种情况是语句缺失错误，即有些程序变体中删去了一处语句，如 if 条件判断等。这时，程序中的很多部分都将受到影响。这两种情况都可以模拟“多处错误”的情况。

在以上程序素材基础上，使用 unix 系统工具??? `gcov` 我们可以为任一程序错误变体按序生成以下数据：

表 2.3　西门子测试集简述

程序名	版本数	测试用例数	简介
print_token	7	4130	词法解析器
print_token2	10	4155	词法解析器
replace	32	5542	模式识别程序
scheduler	9	2650	按优先级调度的调度器
scheduler2	10	2710	按优先级调度的的调度器
totinfo	23	1608	高度分离计算程序
tcas	41	1052	信息估计计算程序

1. 通过在正确的程序版本上执行测试集获取正确的测试输出作为测试准则。
2. 通过将正确测试准则的判断以概率 mr 随机改变（将“通过”变换为“不通过”或反之）获取错误率为 $r = mr$ 的测试准则。具体而言，对正确测试准则的每一个判断，使用随机数生成器生成一个 $[0, 1)$ 内的随机浮点数，如果该浮点数值小于 mr，则改变该判断。由这一过程的随机性，最终生成的错误测试准则的判断错误率也应为 r。在本章后面的叙述中我们将不区分 mr 和 r。
3. 在执行测试用例的过程中调用 gcov 获取程序代码行被执行的情况，从而获得被测程序的“频谱”。
4. 基于上述数据，结合正确测试准则的判断，应用 SFL 算法计算上文定义的 s_c，度量此时算法的精确度。
5. 使用错误测试准则的判断，应用 SFL 算法计算上文定义的 s_f，度量此时算法的精确度。
6. 由错误测试准则引起的精确度损失 $s_f - s_c$。

对所有 132 个程序变体，我们令 mr 在范围 0.01 至 0.1 之间以 0.01 为间隔变化，统计和计算四种 SFL 算法的精确度损失。下一节将详细讨论和分析实验结果。

2.3.2.4　实验结果

我们首先在西门子测试集中所有错误的程序变体上执行测试用例，并以正确的测试准则判断测试结果，以此时 4 中 SFL 计算公式的精确度记录下来作为基准。图**??**中展示了这一结果。令 V 表示所有错误变体的集合，令 $v_i \in V$ 表示任意一个错误变体。设 $\Omega = \{Naish2, Ochiai, Russel\&Rao, Tarantula\}$ 表示所有我们关心的 SFL 算法，其中任一算法由 $\omega_i \in \Omega$ 表示。令 s_x 表示“score”轴上的某一具体值，则在“proportion”轴上的相应数值定义为

$$p_{\omega_j}(s_x) = \frac{|\{k | s(v_k, \omega_j) \leq s_x\}|}{|V|}$$

直观的解释，$p_{\omega_j}(s_x)$ 表示了所有程序变体中使用相应 SFL 公式计算得到的定位精度超过 s_x 的程序变体所占的比例。

如图**??**所示，Russel&Rao 表现出了最佳性能，在超过 80% 的错误变体上，错误代码行被排在了第一位。Ochiai 排在第二位，在超过 95% 的程序变体上仅需查找不超过 15% 的代码行即可发现错误。Naish2 比 Tarantula 略好，但是这二者性能都不如 Ochiai。图中数据与[9] 的实验结论“Ochiai 性能一致稳定地优于 Tarantula”吻合，也与[13] 中的结论部分吻合。[13] 得出的结论是 Ochiai 优于 Naish2，但弱于 Russel&Rao。本章工作与这项研究产生不一致是由于二者的评价标准略有差异。[13] 比较了所有程序变体上错误定位精度的平均值，而我们则画出了定位精度达到了 $[0, 1]$ 之间所有精度值的程序变体比例做出二维图并以图像作为依据进行直观的比较。理论分析文章[12] 中的出的结论是 Naish2 应优于 Ochiai，这与本文中的数据以及其他实验测评研究得到的结论均相悖。除了[13] 中提出的“覆盖率 100% 难以达到”这一条之外，我们认为还有两条原因可以解释这一现象。一是[12] 中的理论分析是建立在“单一错误假设”的基础上。尽管西门子测试集中的每个程序变体都仅包含一个错误，但某些错误可能产生的影响类似“多处错误”。例如我们发现有 8 个版本中的错误类别都是 C 语言中宏定义（#define）的常量值发生错误。另一个版本中某个变量在声明处本应定义为“DOUBLE”类型，但实际定义成了“FLOAT”类型。另一个原因是,[12] 也没有考虑到代码缺失这一错误类型，但是测试集中的程序有 13 个版本的错误都属于这一类型，if 语句中的条件缺失就更普遍了。这些偏差使得实测结果与理论分析的结论有出入，但总体来讲这四种算法都能够较好的完成错误定位工作，在超过 80% 的程序上它们能够将错误代码圈定在前 20% 的代码行上。

为了量化测试准则带来的影响，令 $s_0(v_i, \omega_j)$ 代表应用 ω_j 使用正确的测试准则对程序变体 v_i 进行错误定位的准确度，令 $s_r(v_i, \omega_j)$ 表示应用同样的定位算法使用错误率为 r 的测试准则作为输入的定位准确度。我们定义使用 ω_j 在测试准则错误率为 r 时对 v_i 进行定位的绝对精确度损耗为

$$\Delta^{-}_{|\cdot|}(\omega_j, v_i, r) = s_r(v_i, \omega_j) - s_0(v_i, \omega_j)$$

我们以正确的测试准则为基准，通过对测试结果按照比率 r 随机取反得到了错误的测试准则。在 r 从 0.01 向 0.1 变化的过程中，几种 SFL 算法的精确度也有不同程度的损耗。为了方便直观理解，我们将 $r = 0.05$ 时的 score-proportion 切片图列在图**??**的右方。如图所示，使用 Tarantula 计算时，精度 s_x 落在 $[0.15, 0.25]$ 中的程序变体的比例 $p(s_x)$ 有一定的下降。相应的 Ochiai，Naish2 和 Russel&Rao 的曲线也

明显向下凹陷。左图中显示 Ochiai 原本可以对超过 95% 的程序变体达到优于 15% 的定位精度，而现在只有 75% 能够达到这一标准。而对于 Naish2 和 Russel&Rao，这一比率原本是 80%，而现在只有 30%。

为了对定位精度的损耗有更加全面的认识，我们令 $r \in \{0.01, 0.02, \ldots, 0.1\}$，并记录四种 SFL 算法在所有程序变体上的精度损耗 $\Delta^{-}_{|\cdot|}(\omega_j, v_i, r)$。图**??**展示了四种算法的绝对精度损耗随 r 的变化情况。

在图**??**中，r 轴表示测试准则的错误率 r，Δ_x 表示绝对精度损耗数值，p 表示绝对损耗值小于 Δ_x 的程序变体所占的比例，定义如下：

$$
\begin{aligned}
&p_{\omega_j}(\Delta_x, r) \\
&= \frac{|\{k | \Delta^{-}_{|\cdot|}(\omega_j, v_k, r) \le \Delta_x\}|}{|V|}
\end{aligned}
$$

其中 ω_j 表示所采用的 SFL 算法。图**??**采用颜色图谱展现 p 随 Δ_x, r 的变化，并按 0.1 为间隔标出了 p 在 $[0.5, 1)$ 的曲面等高线，将其投影在水平面上。

如图所示，不同 SFL 算法所受的影响也不同。与 $r = 0.05$ 时的曲面切片一致，Tarantula 对测试准则错误的影响最不敏感。对超过 80% 的程序变体，在遇到真正错误的代码行前所需要检查的代码行数仅上升了 2% 左右。Ochiai 比 Tarantula 敏感一些，在大约 20% 程序上精度损失了 15% 左右。Naish2 和 Russel&Rao 的精度损失相当，在超过 50% 的程序的程序上损失超过了 15%。值得注意的是，图中绘制的是绝对精度损失，也就是说这里的 15% 指的是需要多检查被测程序代码总行数的 15%，对于“生成-检验”系统而言这将带来明显的效率降低。

得到这样的实验结果并不是巧合。对于 Russel&Rao，a_{ef} 是影响最终代码行排序结果的唯一数值。当随机改变测试准则的判断结果时，“假失败”，即程序实际运行正确但被判别为测试失败的测试用例数目将会使很多代码行对应的 a_{ef} 增加，因此实际错误的代码行将不会被排在与原来一样靠前的位置。对于 Naish2，相比于 a_{ep} 来讲 $\frac{a_{ep}}{a_{ep}+a_{np}+1}$ 这一部分的数值很小，起决定性作用的仍然是 a_{ep} 的值，因此测试准则错误带来的影响与 Russel&Rao 相似。对 Ochiai，由于程序执行测试时的运行轨迹不会受测试准则错误的影响，因此影响最终结果的只是 $\frac{a_{ef}}{\sqrt{a_{ef}+a_{nf}}} = \sqrt{a_{ef}}\sqrt{\frac{a_{ef}}{a_{ef}+a_{nf}}}$。由于我们对测试准则判断结果的改变是随机的，$\frac{a_{ef}}{a_{ef}+a_{nf}}$ 这一部分的数值应当基本没有变化，于是主要的影响因素就只剩下 $\sqrt{a_{ef}}$。这就解释了为什么 Ochiai 受到的影响比 Russel&Rao 和 Naish2 少，但是精确度损耗随错误比率的变化趋势相类似。最后，对于 Tarantula，公式中的两个部分 $\frac{a_{ef}}{a_{ef}+a_{nf}}$ 和 $\frac{a_{ep}}{a_{ep}+a_{np}}$ 所受的影响都比较有限，这也与实验数据吻合。

以上的实验数据和分析可能会引起一个疑问，既然 Tarantula 受测试准则错误的影响并不十分明显，为什么还需要对测试准则错误进行处理，而不直接使用 Tarantula 呢？首先，实验数据表明在测试准则没有错误的情况下 Tarantula 的性能不如其他三个计算公式。因此，如果能够识别或部分识别测试准则的错误并加以改正，其他三个计算公式的性能可能会超过 Tarantula。其次，Tarantula 并不是完全不受测试准则错误的影响，改正错误仍然能够改善 Tarantula 的精度损耗。总之，尽管实验结果显示 Tarantula 容错能力较强，但这并不意味着修改测试准则中的错误没有意义。

从等高线投影图可以看到，性能损耗随着测试准则错误率的增加而增加。这一点在 Ochiai 的图表上表现的特别明显，当错误率增加时，等高线越来越远离“mutation”轴。尽管在实验中我们仅记录了 r 在 $[0.1, 1)$ 时的数据，但这一趋势显示当 $r \geq 0.1$ 时测试准则错误带来的负面影响将越来越显著。

实验数据中的另一个有趣的现象是，两个理论分析结果中最优的计算公式对测试准则错误的敏感程度超过了 Ochiai，而 Ochiai 又超过了 Tarantula。直觉上，这一现象可以解释为表现更优秀的计算公式应该对测试过程中的数据利用的更充分，因此也会对数据的错误更加敏感。但是实验数据也显示 Naish2 的实测结果不如 Ochiai。由此这几个计算公式之间的优劣关系仍然有待研究。

在本节中，实验数据说明 SFL 算法的定位精度确实会因测试准则的错误受损，测试准则的错误率越高，SFL 算法的准确率受损程度越高。因此，为了使得 SFL 算法能够在“生成-验证”系统中起到应有的作用，应尽量识别并改正测试准则中的错误。

2.4　测试 Oracle 纠错算法

修正测试准则的错误判断具有一定的难度。首先，测试准则的设计初衷是检查和判断程序的正确性，而不是作为被程序检查的对象。换言之，没有其他的标准可以用来检查测试准则的正确性。考虑到测试准则最初是开发人员根据自己对程序行为的理解建立的，一种可能的解决办法是人工进行多次检查。然而，检查测试准则可能与建立一样耗时耗力。哪怕我们愿意接受这些成本，考虑到测试准则中的错误本身很难发现，重新检查一遍也未必能够清理掉这些错误。

幸运的是，在大多数情况下，测试准则中的错误可能只占很少的比例。如果利用绝大多数测试准则的正确判断，我们或许可以在某种程度上预测程序的行为，并判断程序是否应该通过某些测试用例。

受基于覆盖率的测试集压缩领域工作[5][6] 的启发，我们观察到执行路径相似

的测试用例通常有相似的测试结果，即这些测试用例通常是同时通过或同时不通过。研究者称，如果在保持测试过程中的基本块覆盖率（Block coverage）不变的前提下，大幅缩减测试集的规模不会对错误定位的准确度造成明显的影响。换言之，测试集中具有相似执行路径的测试用例，如果它们的基本块覆盖情况相同，通常也会有相同的测试结果（通过或不通过）。基于这一观察，本文提出利用测试用例对程序代码行的覆盖情况识别和修改测试准则中的错误。

Algorithm 1 Correct the test oracle

Input: $P, O(T), n, thres$

Output: $O'(T)$

1: **for** each $t_i \in T$ **do**
2: Record $Tr(P(t_i))$;
3: **end for**
4: **for** each $t_i \in T$ **do**
5: Find $T_i = \{t_{i1}, t_{i2}, \ldots, t_{in}\} \subset T$ nearest to t_i;
6: $Suspicion(t_i)$ = vote (T_i);
7: **if** $Suspition(t_i) > thres$ **then**
8: $O'(t_i) = \neg O(t_i)$;
9: **else**
10: $O'(t_i) = O(t_i)$;
11: **end if**
12: **end for**
13: **return** $O'(T)$;

算法1描述了测试准则错误修正的方法框架。其输入包括程序 P，测试集 T 及其对应的测试准则 $O(T)$，投票测试用例数 n 以及“可疑值阈值”$thres$。其输出是修正后的测试准则 $O'(T)$。算法分为两个步骤。步骤一，执行所有的测试用例 $t_i \in T$，记录程序 P 执行测试用例 t_i 的运行路径 $Tr(P(t_i))$。步骤二，首先，对所有的测试用例 t_i，找到与之距离最近，即最相似的 n 个测试用例构成集合 $T_i = \{t_{i1}, t_{i2}, \ldots, t_{in}\}$。接着，我们设计一种投票策略，让 T_i 中的测试用例的测试结果决定 t_i 的测试结果是否应该通过或不通过。投票策略的输入是 t_i 和 T_i，输出则是一个量化的可疑值 $Suspicion$，表示测试准则对 t_i 的测试结果判断 $O(t_i)$ 出错的概率。如果 T_i 中的测试用例投票认为 t_i 的测试结果更可能是“通过”，但测试准则判断其结果为“不通过”，并且 $Suspicion(t_i)$ 超过了预设的阈值 $thres$, 那么算法将判断结果取反，修改

为“通过”，并记录下作为输出的 O' 对 t_i 的判断结果。

上述算法框架结构比较简单，但具体实现时仍有以下问题需要解决:

- 算法的第一步需要记录测试用例的执行路径。由于已有许多成熟工具能够辅助获取执行路径（如 `gcov` 等），本文对这一步骤不做详细讨论。
- 算法的第 5 行在计算 T_i 时要求我们对测试用例之间的相似度做出定义，即需要定义一种依据执行路径量化测试用例之间相似性的度量公式。
- 算法第 5 行要求我们确定集合 T_i 的规模 n 的合理取值。
- 算法第 6 行要求设计一种计算测试准则判断结果可疑值 *Suspicion* 的策略，称为投票策略。
- 算法的第 7 行要求我们给出 *thres* 的合理取值。

以上问题的解决方案将影响到算法整体的有效性、时间和空间复杂度。以下我们将分节讨论这些问题。

2.4.1　相似性度量

测试用例相似性度量公式需要尽量将相似的测试用例聚集在一起，并将相异的测试用例尽量分离。令 $tr_i = \{i_0, i_1, \ldots, i_{n_i}\}$ 表示测试用例 t_i 覆盖的代码行，称为 t_i 的测试轨迹集合（trace set）。量化两个集合 A 与 B 相似性的一种常用度量是 $\frac{|A \cap B|}{|A \cup B|}$，受此启发，我们可以定义两个测试用例的相似度为其测试轨迹集合的相似度，即

$$Sim(t_i, t_j) = Sim(tr_i, tr_j) = \frac{|tr_i \cap tr_j|}{|tr_i \cup tr_j|}$$

对以上公式的直观理解是，两个测试用例所共同覆盖的代码行越多越相似。但是，实际程序执行过程中，常常有很多代码行被所有测试用例都执行。例如，单元测试中的初始化代码负责构造被测单元的实例对象，在每个测试方法执行前都需运行。这部分运行路径所占比例过大将导致事实上并不相似的测试路径在上述度量上相似。事实上，被所有测试方法均覆盖的代码行不具有区分度，而被越少的测试方法覆盖的代码行越具有代表意义。因此，在依据上述公式度量两个测试用例相似度时，应为相应测试轨迹集合中的代码行赋予一定的权重。

赋权公式的设计参考了 $tf\text{-}idf$[33]。$Tf\text{-}idf$ 是 *term frequency - inverse document frequency* 的简称，最初用于量化一个单词对一个词汇集合（corpus collection），或称为文档（document）的代表性。令 D 表示一个文档集合，$d \in D$ 是集合中的一个文档，t 表示 D 中的一个词汇。词频（*Term frequency*）$tf(t, d)$ 表示 t 在文档 d 中出现的次数。倒排词频（*Inverse document frequency*）度量 t 在文档集合 D 中出

现的频率，定义为

$$idf(t, D) = \log \frac{|D|}{|\{d|d \in D \wedge t \in d\}|}$$

基于此，对于文档 $d \in D$，t 对 d 的权重定义为

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

回到对测试用例相似性的度量，我们可以将每个代码行看做为一个单词 t，将每个测试路径集合看做文档 d，将所有测试路径集合看做文档集合 D。在此模型下，对 $t_i \in T, i_p \in tr_i, tf(i_p, t_i) = 1$，因此

$$tfidf(i_p, t_i, T) = idf(i_p, T) = \log \frac{|T|}{|\{t_l|t_l \in T \wedge i_p \in t_l\}|}$$

将权重与集合相似度公式相结合，最终我们定义两个测试用例 t_i 和 t_j 之间的相似度为

$$Sim(t_i, t_j) = \frac{\sum_{i_p \in tr_i \cap tr_j} tfidf(i_p, t_i, T)}{\sum_{i_p \in tr_i \cup tr_j} tfidf(i_p, t_i, T)}$$

注意，此处 $tfidf(i_p, t_i, T)$ 实际上 t_i 无关，因此 $Sim(t_i, t_j) = Sim(t_j, t_i)$，即公式对 t_i, t_j 满足交换律。此外，$tr_i \cap tr_j \subset tr_i \cup tr_j$，因此 $0 \leq Sim(t_i, t_j) \leq 1$ 恒成立。

2.4.2　投票策略

对测试用例 t_i，我们计算其与所有其他测试用例的相似度，并选出其中与之相似度最高的 n 个测试用例构成投票组。购票组中测试用例的测试结果将用于决定 t_i 的测试结果是否正确。设 $T_{i,p} = \{t_{iq}|O(t_{iq}) = \mathcal{P}, t_{iq} \in T_i\}$，$T_{i,f} = \{t_{iq}|O(t_{iq}) = \mathcal{F}, t_{iq} \in T_i\}$，分别表示投票组中被实际测试准则 O 判别为“通过”和“不通过”的测试用例集合。定义 $Vote_{ip} = \sum_{t_{iq} \in T_{i,p}} Sim(t_i, t_{iq})$，$Vote_{if} = \sum_{t_{iq} \in T_{i,f}} Sim(t_i, t_{iq})$，分别表示投票为“通过”和“不通过”的计票数，则 t_i 的测试结果可疑值 Suspicion 按如下公式计算。

$$Suspicion(t_i) = \begin{cases} \frac{Vote_{ip}}{Vote_{ip}+Vote_{if}} & if\ O(t_i) = \mathcal{F} \\ \frac{Vote_{if}}{Vote_{ip}+Vote_{if}} & if\ O(t_i) = \mathcal{P} \end{cases}$$

2.4.3 Parameter Settings

算法1涉及两个重要的参数，其一是投票组中的测试用例数 n，二是可疑度阈值 $thres$。这两个参数都是预设的常数，因此其值应科学选取。从直观上分析，若 n 太小，如 $n = 2$，那么很可能投票组中的两个测试用例的测试结果也有错误，那么投票结果也是不可信的。而如果 n 过大，则投票组中将很可能包含许多与所关心的测试用例并不相似的测试用例，其投票结果同样不可靠。类似的，若参数 $thres$ 设置的过高，那么测试准则的判断错误将可能被漏掉。相反，若 $thres$ 太低，则很多实际正确的测试结果会被误认为错误。

理想的参数配置使算法能够尽量保留正确的测试结果，同时识别出错误的测试结果，也即应当降低假阳（false positive，简称 FP）和假阴（false negative，简称 FN）的数目。设置参数的困难在于，在算法的实际应用中，我们预先并没有一个完全正确的测试准则作为参考，自然也不能对 FP 和 FN 准确计数。换言之，哪怕将所有的 n 和 $thres$ 的组合遍历一次，我们仍然无法选出哪一组参数的设置是最合理的。

事实上，从给定所期望的 FP 和 FN 出发寻找恰当的 n 和 $thres$ 是比较困难的，但从相反的方向分析 n 和 $thres$ 的不同取值对 FP 和 FN 的影响则是可行的。本节我们分析 FP、FN 与 n、$thres$ 的概率关系。分析基于以下两个假设：

假设 *1*：$\forall t_{iq} \in T_i \cup \{t_i\}, O_c(t_{iq}) \in \{\mathcal{P}, \mathcal{F}\}$ *i.i.d.*.

解释: 一个测试用例的运行结果与其他测试的运行结果不相关，因此每个 $t_{iq} \in T_i \cup \{t_i\}, O_c(t_{iq})$ 都是独立的。另外，由 T_i 的生成方式可知，任一 $t_{iq} \in T_i$ 与 t_i 均足够相似，因此我们可以合理的假设这些测试用例的测试结果的概率分布也是相同的。

假设 *2*：$\forall t_{iq_1}, t_{iq_2} \in T_i \cup \{t_i\}, iq_1 \neq iq_2, \mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) = sim_i$ $(constant)$

解释: 由于 $O_c(t_{iq}) \in \{\mathcal{P}, \mathcal{F}\}$ *i.i.d.*, 显然存在常数 $\alpha_i \in [0, 1]$ 满足 $\forall t_{iq_1}$，$t_{iq_2} \in T_i \cup \{t_i\}$，$iq_1 \neq iq_2$，$\mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) = \alpha_i$。根据上文中的观察，测试用例越相似，具有相同结果的概率就越高，因此我们可以用 sim_i 估计 α_i。

在以上两条假设下，我们可以得到以下结果：对任意 $t_i \in T$，经算法1计算得到的 $O_c(t_i)$ 是 FN 的概率是

$$\mathbf{P}(fn(t_i)) \approx r \sum_{w=0}^{\hat{n}} \mathrm{C}_n^w \phi_i^w (1 - \phi_i)^{(n-w)} \tag{2-2}$$

而 $O_c(t_i)$ 是 FP 的概率是

$$\mathbf{P}(fp(t_i)) \approx (1-r)\sum_{w=\hat{n}+1}^{n} \mathrm{C}_n^w (1-\phi_i)^w \phi_i^{(n-w)} \tag{2-3}$$

其中 $\phi_i = \beta_{i2} + r - 2\beta_{i2}r$，$\beta_{i2} = \frac{1+\sqrt{2sim_i - 1}}{2}$，$sim_i$ 是 T_i 中所有测试用例与 t_i 相似度的平均值，$\hat{n} = \lfloor n \times thres \rfloor$，$r$ 是测试准则的错误率。

公式2-2和2-3的详细推导过程请参见附录。

上述公式可以指导 n 和 $thres$ 的取值决策过程。决策过程的第一步是分析 ϕ_i 的取值情况。ϕ_i 是 r 和 sim_i 的函数，图**??**展示了三者之间的量化关系。如图所示，$0 < \phi_i < 1$。由于我们假定测试准则的大部分判断是正确的，即 $r > 0.5$，并且在这一范围下，$\phi_i > 0.5$，因此显然 $\mathbf{P}(fn(t_i))$ 随 $\hat{n}$ 的增加而剧烈增加，$\mathbf{P}(fp(t_i))$ 随 $\hat{n}$ 的增加而剧烈减少。合理的参数取值应当使 FN 和 FP 的比例同时尽可能低。直观上似乎应使 $thres = 0.5$，这样 FP 和 FN 都不会太高，而量化分析的结果也确实如此。表2.4列出了 $P(fn(t_i))/r$ 和 $P(fp(t_i))/(1-r)$ 两个算式在 $n = 3,4,5$，$\hat{n} = 1,\cdots,4$ 及 $\phi_i = 0.7, 0.8, 0.9$ 所有组合下的取值。表格中标红的参数组合是在确定 n 的取值下我们选出的最优参数组合。选择的理由有两个，首先 r 相对较小，一个较大的 $P(fn(t_i))/r$ 与之相乘时计算结果也不会很大，但是一个较大的 $P(fp(t_i))/(1-r)$ 与 $(1-r)$ 相乘则计算结果也会较大，因此在 $n = 4$ 时我们选择 $\hat{n} = 2$。第二，实验结果表明 FP 的存在对 SFL 算法精度的负面影响更加明显，使 $\hat{n}$ 与 n 距离稍远可以带来更好的错误定位精度，因此在 $n = 5$ 时我们选择 $\hat{n} = 2$。

表 2.4　$fp(t_i)$ 和 $fn(t_i)$ 的概率随 ϕ_i，n 和 $\hat{n}$ 的变化

n		3		4			5			
$\hat{n}$		1	2	1	2	3	1	2	3	4
$\phi_i = 0.7$	$\mathbf{P}(fn(t_i))/r$	0.2160	0.6570	0.0837	0.3483	0.7599	0.0307	0.1630	0.4717	0.8319
	$\mathbf{P}(fp(t_i))/(1-r)$	0.2160	0.0270	0.3483	0.0837	0.0081	0.4718	0.1630	0.0308	0.0024
$\phi_i = 0.8$	$\mathbf{P}(fn(t_i))/r$	0.1040	0.4880	0.0272	0.1808	0.5904	0.0067	0.0579	0.2627	0.6723
	$\mathbf{P}(fp(t_i))/(1-r)$	0.1040	0.0080	0.1808	0.0272	0.0016	0.2627	0.0579	0.0067	0.0003
$\phi_i = 0.9$	$\mathbf{P}(fn(t_i))/r$	0.0280	0.2710	0.0037	0.0523	0.3439	0.0005	0.0085	0.0814	0.4095
	$\mathbf{P}(fp(t_i))/(1-r)$	0.0280	0.0010	0.0523	0.0037	0.0001	0.0814	0.0085	0.0005	0.0000

表2.4中给出的参数选择方案仅考虑了如何针对某一个特定 t_i 降低 $P(fn(t_i))$ 和 $P(fp(t_i))$。若要降低测试集整体的 FP 和 FN 数目则需要知道所有测试用例的 ϕ_i。但 ϕ_i 是 n 的函数，因此在确定 n 前我们无法得知 ϕ_i 的值。打破这一循环的关键在于，从表2.4中的数据我们可以得到这样的结论，当给定某一 n 时，$\hat{n}$ 的最佳取值与 ϕ_i 的变化无关。因此，尽管测试用例的 sim_i 各不相同，一旦 n 确定后，我们不需要对每个测试用例单独设置一个 $thres$ 值。换句话说，尽管我们无法将所有测试用例的 $P(fn(t_i))$ 和 $P(fp(t_i))$ 逐一计算出，我们仍然可以为所有测试用例统一选择一个最优的 $thres$ 值。

上述分析将问题简化为选择一个合理的 n。根据表2.4中的数据，ϕ_i 值越大，$P(fn(t_i))$ 和 $P(fp(t_i))$ 的值越小。因此，如果 ϕ_i 较小，则 n 可相应的取一个较大的值用以保证算法的正确率。但是当 $\phi_i > 0.8$ 时，取 $n = 4$ 算法的正确率已经比较好。在实际应用中，如果有先验知识可以用来估计 ϕ_i，那么可以使用表2.4作为确定 n 取值的参考。如果没有先验知识，可以按照如下步骤选择 n：（1）执行所有测试用例，获取测试路径集合，（2）在测试集上随机选取一个子集作为样本，为子集中所有测试用例找到全集中的 10 个距离最近的邻居，（3）计算这些邻居间的相似度。根据这一数据，我们可以估计 sim_i 的值，也能够迅速了解每个测试用例周围的邻居有多少个。如果 sim_i 很低，或者邻居很少，可能我们需要为测试集补充新的测试用例。反之，如果 sim_i 足够高，邻居也比较充足，那么只要 r 不太大，算法最终的正确率应当仍有保证。不管怎样，即使是在需要补充新的测试用例的情况下，由于测试准则本身是有错的需要修改，我们并没有为测试人员增加额外工作量。

2.4.4　时间与空间复杂度分析

算法1包含两个阶段，即测试轨迹获取和测试准则调试。第一阶段的时间和空间复杂度取决于所使用的的轨迹获取工具，例如 $gcov$。但存储所有测试路径集合需要的空间复杂度为 $O(|T|N)$。在第二阶段，首先我们需要计算每一行代码的权重，即 $tf-idf$ 值，这将带来 $O(|T|N)$ 的时间复杂度，但并不需要申请新的空间。接着我们计算任意两个测试用例之间的“相似度”，这一操作的时间复杂度为 $(|T|^2O(sim))$，其中 $O(sim)$ 是计算给定两个测试用例之间的相似度的时间复杂度，空间复杂度是 $O(|T|^2)$。在本文的实现中，$O(sim) = O(N)$。算法的下一步骤是计算每个测试用例 t_i 的邻居集合 T_i，这一步骤最简单的实现需要 $O(|T|^2n)$ 的时间复杂度和 $O(|T|n)$ 的空间复杂度。最后，算法计算每个测试用例的“可疑值”，时间复杂度是 $O(|T|O(Sus))$，其中 $O(Sus)$ 是计算一个给定测试用例可疑值的时间复杂度，空间复杂度最多为 $O(|T|)$。在本文实现中，$O(Sus) = O(n)$。因此，算法的总时

间复杂度是 $\max\{O(|T|^2O(sim)), O(|T|^2n), O(|T|O(Sus))\} = O(|T|^2N)$，总空间复杂度是 $O(|T|N)$。这一结果说明，时间复杂度随程序规模的增加而线性增加，随测试集规模的增加而平方增加，空间复杂度随程序规模和测试规模的增加而线性增加。

2.5 实验结果及分析

A comprehensive evaluation of our approach should inspect its performance on two aspects, (1) how much “correcter" is the debugged oracle compared to the original one and (2) how much the debugged oracle improves SFL algorithms' performance. To answer these two questions, we conducted two corresponding experiments. The results are presented in the following two sections.

2.5.1 Repair the Erroneous Test Oracle

Our first experiment evaluates the quality of the debugged oracle. In this experiment, we used the Siemens Test Suite as our subject because it contains enough similar entities of programs, which ensures a statistical significant result. Same as when we inspected erroneous oracles' impact on SFL algorithms, we first generated the correct oracles of the Siemens Test Suites and randomly mutated them with a mutation rate mr ranging from 0.01 to 0.1. Remember the value of two input parameters needs to be decided, i.e. the number of voters n and the debugging threshold $thres$. To achieve better performance, we first analysed a sample of the subject test suites and discovered that nearly all test cases own more than 10 neighbours with similarity above 0.9. This was confirmed by a later analysis as approximately over 95% test cases own such number of neighbours. Referring to Table 2.4 we decided $n = 4$ is proper for gaining satisfactory performance. As with $thres$, according to our analysis, as long as $2 < \hat{n} < 3$ the final performance should be satisfactory. For convenience we fixed it on 0.58 for all experiments.

With $n = 4$ and $thres = 0.58$, we applied our debugging algorithm to the oracle mutants and compared the output oracles to the original correct ones. According to[32] we repeated our experiment 5 times to achieve a stable and convincing result. Three quantitative index values are selected as assessment criteria, namely, “recall", representing our algorithm's ability to acutely discover the error, “precision", indicating the ability to preserve the already correct judgements and “f1 measure", synthesizing the former two.

Fig. **??** presents the overall distribution of recall, precision and f1 measure for all oracle mutants and error rates. For recall rec and a mutation rate mr, the proportion value

$p(rec, mr)$ is defined as the ratio of mutants whose corresponding output oracle achieved recall value greater or equal than rec in all mutants mutated at mr. For precision pre and f1 measure $f1$, $p(pre, mr)$ and $p(f1, mr)$ are similarly defined.

As shown in the figure, averagely our algorithm achieves 85% recall rate on over 90% mutants, meaning over 85% oracle errors are successfully identified. As to precision, mutants at different mutation rates respond differently. As the median, when $mr = 0.05$, precision reaches 75% for over 80% mutants, i.e. over 75% of the highlighted test cases are truly misjudged in the input oracle. This fraction expands to 80% as mr increases to 0.1. Figure of F1 measure is similar to precision as value of recall is close to 1.

Contour projection on the x-y panel presents some interesting phenomena. Firstly, when mr is lower than 0.02, over 20% highlighted test cases are false positives for 50% mutants, and over 40% for 30% mutants. This accords with intuition when considering the relative high recall. 90% errors in around 80% oracle mutants are marked out, thus it is very likely that many other oracle judgements are wronged. Secondly, with the increase of mr, recall falls slightly while precision rises promptly. This matches our theoretical analysis as with a fixed $thres$ and n, both $\mathbf{P}(fn(t_i))/r$ and $\mathbf{P}(fp(t_i))/(1-r)$ stay almost the same. Therefore as r increases, $\mathbf{P}(fn(t_i))$ increases while $\mathbf{P}(fp(t_i))$ decreases.

2.5.2 Cure SFL Algorithms

The second experiment is designed to evaluate how much our algorithm can reduce the extra time consumption caused by test oracle errors during the debugging process. This evaluation is conducted on not only the Siemens Test Suite, but also on `grep`, a real world program which scales up to over 13,000 lines of code. The results are presented as follows.

2.5.2.1 Results on the Siemens Test Suite

As a reference, we first applied the four SFL algorithms on each program variants of the Siemens Suites with the erroneous oracle mutants we generated and recorded their performance. Then we performed the oracle debugging process and generated a corresponding “debugged oracle" for each oracle mutant. Finally we applied the four SFL algorithms with the “debugged oracles" and compared their performance with the record with the erroneous oracle mutants. In consistency with Section 2.3, time required for debugging is quantified as the score defined by equation 2-1. Again the experiment was

repeated 5 times. The results are presented in Fig. **??**.

Fig. **??** and **??** compare the performance of four SFL algorithms with the erroneous oracle mutants and the oracles debugged by our algorithm. For a mutate ratio mr, a certain score s_x, the proportion value $p(mr, s_x)$ in both figures is defined as the proportion of program variants that, with the erroneous oracle at error rate mr, can be successfully debugged by checking less than s_x of its code. Visually, surfaces in Fig. **??** for all 4 SFL algorithms are clearly “lifted" from their place in Fig. **??**, demonstrating a remarkable improvement in fault localization accuracy. Typically, under the erroneous oracles, Russel&Rao achieved score 0 (hitting the bug at the first line it suggests to be checked) on approximately 20% program variants, while this number expands to 50% and even higher for less erroneous oracles. For Ochiai, this rate rose from less than 60% to over 70%. Even for Tarantula, the least sensitive algorithm to oracle errors, this rate rose from 25% to 40%.

For clearer view of the differences between Fig. **??** and Fig. **??**, we plotted two other set of figures demonstrating the absolute (Fig. **??**) and relative (Fig. **??**) improvement. To be precise, in addition to $s_0(v_i, \omega_j)$ and $s_r(v_i, \omega_j)$ defined in subsection 2.3.2.4, we define $s_d(v_i, \omega_j)$ as the score of ω_j applied to v_i with the debugged oracle. Then the absolute performance improvement of ω_j for v_i at error rate r is defined as

$$\Delta^{+}_{|\cdot|}(\omega_j, v_i, r) = s_r(v_i, \omega_j) - s_d(v_i, \omega_j)$$

and the relative improvement is defined as

$$\Delta^{+}_{\%}(\omega_j, v_i, r) = \frac{s_d(v_i, \omega_j) - s_0(v_i, \omega_j)}{s_r(v_i, \omega_j) - s_0(v_i, \omega_j)}$$

For both figures, proportion p relative to a certain improvement value Δ_x at mutate rate r represents the rate of program variants on which the SFL algorithms achieved greater improvement than Δ_x using oracles mutated and debugged at mutate rate r. As shown in Fig. **??**, different algorithms improve at various extent. For Russel&Rao, approximately 30% program variants can be successfully debugged by checking 10% less code. These numbers reflect dramatic reduction in debugging time.

One may notice that there seems to be not much absolute improvement on a majority of program variants. This is because for most of the cases, the faulty line can be localized within 30% of the total code even with the erroneous oracle mutants, which leaves not

表 2.5　Summary of `grep`

version	fault seeds	detectable seeds	line of code
v1	18	2	12654
v2	8	1	13231
v3	18	2	13374
v4	12	2	13360
v5	1	0	13293

much space of absolute improvement. However, as presented in Fig. **??**, there is a relative improvement for over 50% program variants. The most obvious improvement appeared in Russel&Rao, where accuracy for approximately 50% program variants improved 50%.

From Fig. **??**, we conclude that after debugging the test oracles, effectiveness of four subject SFL algorithms recovered remarkably. Since each of these four algorithms are either identified as “optimal" metrics or widely known, we believe they are representative of the whole SFL family.

2.5.2.2 Results on `grep`

`grep` is a command-line utility for searching plain-text data sets for lines matching a regular expression widely used on Unix-like systems. SIR[7] provided 5 versions of `grep`, each containing from 1 to 18 fault seeds and several corresponding test sets. The original test results showed that plenty of the fault seeds could not be detected, thus were excluded in our experiment. Especially, only one fault seed was provided for version 5 but was undetectable, so version 5 was not included. A summary of the usage of `grep` is presented in Table 2.5.

Following the same scheme as with the Siemens Test Suite, we first generated the correct test oracles for four versions of `grep` using the corresponding unseeded programs. Then we randomly mutated the test oracles at a mutate rate *mr* from 0.01 to 0.1. For comparison, we recorded the accuracy of the four SFL algorithms applied to the seeded programs using both the mutated oracles and the debugged oracles. The experiment was repeated 5 times. The results are plotted in Fig. 6.

Fig. 6 presents the evaluation results on four versions of `grep` using the four SFL algorithms. Since there are only four versions of programs we were able to look closer at the effect of our approach on each program. For each version using the same SFL algorithm, the red line represents the change of accuracy as *mr* changes from 0.01 to 0.1 with the mutated oracles, while the green line represents the accuracy with the

debugged oracles. Obviously, accuracy improved on most of the versions for different SFL algorithms. This is especially prominent with Russel&Rao. There seems to be no obvious improvement in Ochiai, but accuracy with debugged oracle is at least not worse than with the original oracle.

Another thing worth mentioning is the test sets provided by SIR. For programs in Siemens Test Suite, SIR provided approximately 4000 test cases each. Thus it is natural that we can debug the test oracle taking advantage of the duplicated test cases. However, for `grep` we adopted the test set which aimed to be *"representative of those that might be created in practice for these programs"*[7]. The test set contains merely 199 test cases, yet we still managed to improve the accuracy of SFL algorithms. This proves that our approach is applicable for realistic programs with realistic test suites.

2.6 讨论

During the second experiment in Section 2.5, we observed that as error rate rises, performance improvement of SFL algorithms on the debugged oracles decreases slightly. This trend coincides with the change of recall rate relative to error rate, which leads us to wonder if there is a direct connection between recall rate of the debugged oracles and performance improvement of SFL algorithms irrelevant to the error rate of the input oracles. Since Russel&Rao showed a more obvious improvement, we calculated the absolute and relative improvement distribution related to "recall" and the results are presented in Fig. **??**.

In Fig. **??**, we plotted the proportion of debugged oracles of identical recall rate that achieved absolute or relative improvement values between 0 and 1. Despite the layered surface, from the contour projection we are able to recognize the rough tendency that improvement grows along with the increase of recall.

A direct conclusion from this tendency is that generally a higher recall rate will lead to better improvement with the debugged oracle regardless of the actual error rate. Therefore, our approach can be applied to scenarios where the error rate is over 0.1 (the upper bound in our experiment), with only adjustment of parameters to achieve a proper recall rate.

Another deduction answers an important question: which kind of error, "false passes" or "false fails", impacts SFL algorithms more and needs to be avoided more carefully? This is a meaningful question since early prevention is always cheaper than later correction.

Intuitively we covered this issue in Section 2.1 and decided that “false fails" are worse. But statistical evidence is still required.

For most program variants of the Siemens Test Suite, only around 5% test cases can expose the error. Consequently the majority of oracle errors are “false fails" and “false passes" rarely happen. At a high recall rate, almost all errors in the oracle mutant would be corrected, which dramatically reduces the ratio of “false fails" while “false passes" are less influenced. As a result, “recall" can actually be used as an indicator for error composition of the debugged oracle, that is, a higher recall usually means less “false fails" whereas more “false passes". As the tendency shown in Fig. **??**, less “false fails" relates to more obvious performance improvement. Therefore, seen from the other side “false fails" do impact more than “false passes" and should be especially prevented in the initial creation of test oracles.

Similar to existing work, empirical studies on SFL algorithms are all subject to validity problems. Referring to[32], we summarize the potential threats to validity of our work as follows.

First is the representativeness of the subject programs. In our empirical study in Section 2.3, the subject programs are the Siemens Test Suites. One issue is that the size of the Siemens programs is rather small, therefore the conclusions may not fully reflect the exact impact of oracle errors on large scale programs. But this does not impede us from confirming the negative influence of oracle errors. In Section 2.5, we adopted `grep`, a widely used real world program scaling up to 13,000 lines of code as a subject program and proved that the fault localization accuracy improved with debugged test oracles in Section 2.5. Therefore we believe that our approach is scalable and applicable to real world programs.

The second issue is that all faulty variants are single-faulted, while in real world multi-faults are common. Fortunately as we have pointed out in subsection 2.3.2.4, many of the faults in fact influence a great proportion of the program and may even mimic multiple faults. For example, a wrongly defined constant value in “#define" would map to wrong values scattering all over the program. We kept these kind of errors intentionally to ameliorate the bias between our experiment and real world case.

The third issue is the sample size, or the number of subject programs.[32] states that to reach stable results at least 300 subject programs with errors are required. Therefore we repeated our experiments 5 times to smooth away the randomness.

Another factor mentioned in[32] is the quality of the test suites, including size and coverage. Siemens Test Suite provides ample test cases for each variant and achieves satisfactory test coverage. Test suite for `grep` contains only 199 test cases but was designed to be *"representative of the those that might be created in practice for these programs"*[7]. Therefore we believe that quality of test suites in our experiment is sufficient to come to stable and convincing conclusions.

2.7　本章小结

In this paper, we first proposed the problem of test oracle errors' impact on debugging process. To verify this impact, we conducted experiments on the Siemens Test Suites and tested the performance of four Spectrum-based Fault Localization algorithms using randomly mutated erroneous test oracles. Compared with when using the correct test oracles, fault localization accuracy is remarkably compromised. Therefore it is necessary to debug the test oracle before it is used to debug the program.

Based on a simple observation that test cases covering similar lines of code usually pass or fail simultaneously, we proposed to measure the similarity between test cases using coverage information and identify the unusually judged test cases from their neighbours. As demonstrated by later experiments, these test cases proved to be mostly wrongly judged by the erroneous test oracle. We further compared the accuracy of four SFL algorithms on both the Siemens Test Suites and `grep` using the debugged oracle with using the erroneous oracle mutants. Experimental results show that fault localization ability of all four SFL algorithms recovered in different degrees. Finally, we analysed the experiment data and concluded that our approach is applicable to oracles unlimited to an upper bound of error rate. In all, we proposed a simple but effective method to promote the quality of existing erroneous test oracles.

第 3 章　搜索引擎优化

3.1　引言

介绍搜索引擎在“生成-检验”系统中的功能、作用，基本设计思想、技术路线和存在问题。

3.2　相关工作

重点介绍 GenProg，Par，SPR，SemFix，DirectFix，Prophet，NOPOL 等较完整的系统。

3.3　“预过滤”优化技术

介绍“预过滤”的基本思想，详细介绍具体算法，讨论其理论上的优势与实际应用中的局限性。

3.4　实验结果及分析

“预过滤”策略对搜索空间的压缩作用：将其与 SPR、NOPOL 的压缩效果进行比较分析。

3.5　本章小结

3.6　Introduction

Debugging has long been recognized as one of the most time- and labour- costing activity in software engineering practice. To accelerate debugging process, recently a series of “generate-and-validate" systems are proposed to generate fix suggestions to programmers so that a buggy program can pass a certain test suite. However, two major drawbacks prevents existing systems from practical usage. First, most systems generate program mutations that include only one location of changes in the program. As a result, bugs that require fixes at multi-location, which widely exists in real coding practice, can never be fixed. Second, reported experimental results show that the fix generation time can be too long that programmers may take less time fixing the bug manually.

In this paper, we propose ***SmartDebug*** , an interactive Java debug assistant that implements a generate-and-validate system while aiming to overcome these two drawbacks. More than existing systems, it utilizes programmers' judgment of program running state by taking in "Checkpoints", a user-interface we provide for recording programmers' expectations of program running at arbitrary execution positions, so that complicated debug tasks can be split into small fragment of tasks that are solvable separately yet automatically. Therefore ***SmartDebug*** is able to work with bugs that require modifications on several locations to be fixed. ***SmartDebug*** presented in this paper is an upgraded system of our previous work[?]. It implements our newly proposed optimization strategy called "early filtration", which filter out mutations that are impossible to fix the program before they enter into the validation phase, so that the search space is dramatically reduced and generation process correspondingly accelerated.

SmartDebug is implemented as an Eclipse plug-in integrated with the JDT debugger. For evaluation, we first run ***SmartDebug*** against Defects4J to inspect the effectiveness of "early filtration". Experimental results show that ***SmartDebug*** is able to fix 24 versions, the largest amount ever reported on this benchmark. With early filtration, the mutations that enter into validation phase reduced to less than 10% on 12 versions, and 20% on 8 more versions. To prove that ***SmartDebug*** is able to actually speed up the debugging process, we also evaluate the plug-in on 25 real bugs that appeared during the development process of programing exam. Results show that ***SmartDebug*** is able to accelerate the debugging process on 7 bugs.

Structure of the paper is organized as follows. Background and related work are discussed in Section II. In Section III we explain the design and the two features of ***SmartDebug*** . Implementation details are presented in Section IV while evaluation results are reported in Section V. Finally Section VI concludes the paper.

3.7　Background and Related Work

"Generate-and-validate" systems are proposed as a solution to automatic general bug fixing. They generate possible fixes on the source code that makes the program under test pass a certain test suite. As the name suggests, they first generate a series of program mutations as fix candidates, then apply them to the program , and finally rerun the tests for validation. Major differences between existing systems reside in the "generate" process, and we cluster them as follows.

GenProg, RSRepair, AE and Kali: GenProg[?][?] performs genetic search to generate fixes that are composed of code ingredients from the software system itself. RSRepair[?] adopts a random search algorithm instead. AE[?] deterministically searches for patches and utilizes program equivalence to prune the search space. However analysis in[?] show that most of the patches generated by these systems actually removes program features rather than corrects bugs. To prove this analysis, Kali[?] is designed as a function remover and the experiments show that Kali works just as well. Therefore it remains a question whether these systems can actually work on real software bugs.

Par: In[?], the authors summarized 9 common fix patterns by analyzing fix patterns in open-source software repositories, and used them as templates to generate fixes. The authors claim that the patches generated by Par are more acceptable by developers than GenProg.

Angelic Debugging, NOPOL, SemFix and DirectFix: Angelic debugging[?] replaces expressions with an angelic variable-a variable that takes an arbitrary value-and see if the program can successfully pass the tests. NOPOL[?] narrows the target expression in[?] to condition expressions in Java. If such expression exists, a replacement expression is synthesized with an SMT solver. SemFix[?] uses symbolic execution to calculate the constraints that an replacement expression must satisfy for a potential buggy expression. It then uses an SMT-solver to synthesize such expression. DirectFix[?] improves SemFix as it fuses the fault localization and repair generation into one step by utilizing MaxSAT constraint solving and program synthesis to generate the simplest expression modifications.

SPR and Prophet: SPR, short for staged program repair[?], first generates possible sequences of the value of a target condition expression such that the failed test may take a correct execution route, then synthesizes such condition expression.[?] sorts the prioritizes fixes by their probability of being the correct fix.

SmartDebug differs from existing systems as it allows user annotation and specification of program execution, so that multi-site fixes are handled naturally. Also, the “early-filtration" technique is proposed and applied exclusively by us to sharply reduce the candidate fixes going into the validation process and speeds up the process dramatically. Most importantly, ***SmartDebug*** is a usable demo integrated in a widely used IDE, which can indeed improve debug efficiency for programmers.

表 3.1　Search Space Specification

Fix Pattern	Explanations
expression fix	change an expression to a type-compatible expression
branch introduction	insert if(...) return/break/continue;
branch expression fix	modify if-conditions as general expression fix and change expression “e" to “!e", “e |
array boundary checks	insert if(i >= 0 && i <= array.length){ array access statement; }
collection boundary checks	insert if (i >= 0 && i <= list.size()){collection.get(i) statement;}
null pointer checks	insert if(obj != null){ object access statement;}
cast type checks	insert if(obj instanceof Type){Supertype obj1 = (Type)obj;}
method overload	change a.b(c1, c2, ..., cn) to a.b’(c1, c2, ..., cn)
method override	change a.b(c1,c2,..., cn) to a.b(c1, c2, ..., cm)

3.8 Tool Design

Figure 1 shows the system architecture of ***SmartDebug*** . ***SmartDebug*** consists two major components, the *Interactive Frontend* and the *Fix Generation Backend*. The Interactive Frontend provides facilities for programmers to describe their judgments and expectations of the program running state through “Checkpoints" (*Checkpoint Manager*). It also tracks the debugging process of the program, i.e. the satisfaction status of each Checkpoint on each test case, guiding the programmer through the whole debugging task (*Debug Process Controller*). The *Fix Generation Backend* first localizes the bug by collecting and analyzing the execution trace of each test case (the *Fault Localizer)*, then generates program mutations within a predefined *Search Space* according to the generated ranking list of suspicious fix sites (*Search Engine*), after which some of the mutations go through a *Filter* that removes mutations impossible to fix the bug. Finally the survived mutations are validated by applying them back to the program and rerun the test suite (*Final Validator*).

Search space specification is vital in fix correctness rate in generate-and-validate systems. ***SmartDebug*** adopts the mutation patterns listed in Table I, whose first column lists the name of patterns and the second column provides brief explanations.

SmartDebug distinguishes itself from peer systems by two major features. One is the interactive debugging usage model, which requires a collaboration among several components mentioned above. Second is the early-filtration strategy, implemented in *Filter*.

3.8.1 Interactive Debugging

The interactive debugging usage model aims to take advantage of programmers' understanding of the program under test to break down complicated bugs into simple fractions and narrow down possible bug locations to improve the debug efficiency.

From the programmers' perspective, ***SmartDebug*** works as a personal consultant. During the common debugging process using JDT debug frontend, the programmer is able to describe his judgment of the program running state at a certain point of execution to ***SmartDebug*** through *Checkpoints*. If the program executes correctly, the programmer can mark down this execution point as "correct". Otherwise, he or she can input a boolean expression that should evaluate to `true` if the program runs correctly as a description of his or her expectations.

SmartDebug tracks the current debugging process. Figure 2 shows a snapshot of the debug process control panel. The panel lists every checkpoint and their current satisfaction status. For each failing test case, ***SmartDebug*** will find the first failing checkpoint it encounters during execution. The programmer may choose any failing checkpoints as the debug target in the next step.

When searching for fixes, the fault localizer utilizes the information of checkpoint to achieve more accurate fault localization. We adopt the Ochiai[?] fault localization metric, however the execution trace of the test case containing the target checkpoint is divided into two segments by the last correct checkpoint. The first segment is counted as a successful execution trace, while the second failing. Only code lines covered by the second segment will be treated as possible fix positions. Therefore ***SmartDebug*** is able to focus on a possibly small fraction of programs.

In the validation process, we do not rigorously require the candidate fixes to pass all test cases, instead we report every fix that can fix the target checkpoint while we rank the fixes according to the number of test cases the program passes if they are applied.

3.8.2 Early Filtration

The huge search space has always been an obstacle in developing efficient generate-and-validate systems. "Early filtration", by the name, means to filter impossible fix candidates early in the generate phase, so that they do not flow to the validate phase, which leads to a dramatic compression of the search space and improves efficiency sharply. Unlike SPR[?], "early filtration" works on fixes that modifies expressions, or that

contains modifications of expressions, thus are not limited to only condition expression.

“Early filtration" is proposed based on the observation that if an expression A is to be replaced with an expression B, then:

- In order to ensure that all the passed tests remain passed, B should be “equal" to A at each time the program runs to the position where A is evaluated.
- In order to correct the failed tests, B should NOT be “equal" to A in at least one time when A is evaluated.

For fix pattern 1, this observation can be applied directly. For fix patterns 2-6, the modified or newly introduced branch condition should be seen as expression A in the above observation.

Note that the criteria of “equality" is critical in applying this observation. Equality between primitive values is straight forward. For non-primitive objects, equality is defined as the equality of all primitive-typed fields of the object.

We are currently working on Java programs. JVM makes it very convenient to “halt" the execution process at a certain point and evaluate the value of an expression on a hot stack, which is in orders of magnitude faster than a complete “rebuild-and-rerun" procedure. Therefore, through early filtration, we can save a large amount of time since mass majority of the candidates can be filtered out before entering to the validation process.

3.9 Evaluation

3.9.1 Effectiveness of Early Filtration

We propose “early filtration" strategy to improve the search efficiency of “generate-and-validate" systems. To evaluate its effectiveness, we tested ***SmartDebug*** against the Defects4J[?] benchmark, a collection of several hundred versions of 5 real world programs. We compared the number of fixes that needed to be validated in the final validator with or without “early filtration".

SmartDebug succeeded in fixing 24 bugs in Defects4J, the largest number currently reported on this benchmark. Table II presents the number of program mutations generated before and survived after filtration. Column “EXPR" records the number of expression related mutations where early filtration is applicable. “METHOD" denotes the number of method overriding and overloading mutations where early filtration is not applicable. Column “VAL" reports the number of mutations that need to be validated by the final validator, which includes the survived expression related mutations and also method

表 3.2　Early Filtration Effectiveness on Defects4J

Version_ID	EXPR	METH	VAL	RED
closure_compiler_1	154963	624	1878	1.21%
closure_compiler_10	249694	2783	3596	1.42%
closure_compiler_118	298024	1311	1635	0.55%
closure_compiler_125	212912	417	530	0.25%
closure_compiler_51	1812	54	58	3.11%
closure_compiler_62	61	1	4	6.45%
closure_compiler_63	327	77	80	19.80%
closure_compiler_73	1052	7	11	1.04%
closure_compiler_86	869	160	177	17.20%
closure_compiler_92	17300	279	1550	8.82%
closure_compiler_93	17475	279	1538	8.66%
commons_lang3_24	950	6	124	12.97%
commons_lang3_26	313	43	58	16.29%
commons_lang3_6	375	57	122	28.24%
commons_math_75	1	35	36	100.00%
commons_math_80	1033	100	391	34.51%
commons_math_82	704	80	95	12.12%
commons_math_85	230	39	42	15.61%
commons_math3_33	1527	56	69	4.36%
commons_math3_5	58	0	8	13.79%
jfreechart_1	7632	193	1134	14.49%
jfreechart_11	256	8	9	3.41%
jfreechart_24	54	26	40	50.00%
jfreechart_9	61	3	4	6.25%

related mutations. The last column computes the reduction rate. Results show that in 12 versions only 10% mutations needs to be validated while in 8 more versions only 20%. The reduction is especially remarkable when large number of mutations are generated as in closure compiler version 1, 10, 118, 125, 92 and 93.

3.9.2　Interactive Debugging Efficiency

We collected 25 versions of buggy programs from a coding exam for first year graduate students and asked students in the same year to debug these programs with or without help of ***SmartDebug*** . We recorded the time cost on both situations and compared them in Table III.

Experiments show that without human interference, ***SmartDebug*** is able to generate

correct fix suggestions for 8 of the buggy versions in acceptable time. On 7 versions of programs ***SmartDebug*** accelerated the manual debugging process. Generally with help of ***SmartDebug*** the debug task is finished within 5 minutes.

3.10 Conclusion

In this paper, we present ***SmartDebug*** , an interactive debug assistant for Java. ***SmartDebug*** distinguishes from existing “generate-and-validate" systems in two major features, the interactive usage paradigm and early-filtration optimization strategy. Experiments show that through early-filtration, program mutations are dramatically reduced when entering the validation phase, thus accelerating the fix generation process prominently. Through the interactive usage model, ***SmartDebug*** is capable of supporting multi-location fixes that cover wider practical usage scenarios. Experimental results show that ***SmartDebug*** is able to facilitate the manual debugging process.

表 3.3　SmartDebug v.s. Human

No.	Bug Summary	SD(s)	H(s)
1	wrong usage of loop variables	282	300
2	wrong operator	95	691
3	wrong usage of a local variable	1055	694
4	wrong usage of loop variables	260	423
5	wrong usage of loop variables	309	410
6	wrong usage of loop variables	198	341
7	wrong usage of a numeric variable	215	829
8	wrong usage of a local variable	228	600

第 4 章　“生成-检验”系统的扩展

4.1　引言

4.2　相关工作

4.3　交互式调试

介绍“交互式调试”的基本思想，阐述将其引入后“生成-检验”框架应做出的相应调整及其对系统错误定位模块和搜索模块的正面作用。

4.4　针对单类别错误的可扩展框架

举例说明“生成-检验”系统在修复特定类别错误上的局限性，阐述将针对特定类别错误修复算法整合进“生成-检验”系统中的架构设计，以空指针（NPE）为例具体说明该架构的可扩展性。

4.4.1　框架设计

4.4.2　扩展示例

单类别错误修复实例：在 CWE-Null-Dereference 测试集上的评测结果。

4.5　本章小结

第 5 章　SmartDebug 工具设计与实现

5.1　引言

介绍 SmartDebug 工具的集成环境及应用对象，其所含基本功能模块。

5.2　功能模块

检查点管理模块；修复策略配置模块；修复建议提示与应用模块。

5.3　应用实例

在 defects4J 中较大程序上的应用；在实际编程过程中收集的真实错误上的应用。

5.4　本章小结

第 6 章　总结与展望

6.1　工作总结

总结本文针对现有“生成-检验”自动错误修复框架提出的多项优化技术，分析优点和局限性。

6.2　研究展望

分析能够进行的进一步研究，如“预过滤”技术的扩展应用、单类别错误的支持类型的扩展。

插图索引

表格索引

公式索引

参考文献

[1] Peters D, Parnas D L. Generating a test oracle from program documentation: Work in progress. Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA: ACM, 1994. 58–65.

[2] Harman M, McMinn P, Shahbaz M, et al. A comprehensive survey of trends in oracles for software testing. Technical report, Technical Report Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield, 2013.

[3] Manolache L, Kourie D G. Software testing using model programs. Software: Practice and Experience, 2001, 31(13):1211–1236.

[4] McMinn P, Stevenson M, Harman M. Reducing qualitative human oracle costs associated with automatically generated test data. Proceedings of the First International Workshop on Software Test Output Validation. ACM, 2010. 1–4.

[5] Wong W E, Horgan J R, London S, et al. Effect of test set minimization on fault detection effectiveness. Software Engineering, 1995. ICSE 1995. 17th International Conference on. IEEE, 1995. 41–41.

[6] Wong W E, Horgan J R, Mathur A P, et al. Test set size minimization and fault detection effectiveness: A case study in a space application. Journal of Systems and Software, 1999, 48(2):79–89.

[7] Do H, Elbaum S G, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal, 2005, 10(4):405–435.

[8] 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, New York, NY, USA: ACM, 2005. 273–282.

[9] Abreu R, Zoeteweij P, Gemund A. An evaluation of similarity coefficients for software fault localization. Dependable Computing, 2006. PRDC '06. 12th Pacific Rim International Symposium on, 2006. 39–46.

[10] Abreu R, Zoeteweij P, Golsteijn R, et al. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 2009, 82(11):1780 – 1792. SI: {TAIC} {PART} 2007 and {MUTATION} 2007.

[11] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol., 2011, 20(3):11:1–11:32.

[12] XIE X, CHEN T Y, KUO F C, et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology (TOSEM), 2013..

[13] Le T D, Thung F, Lo D. Theory and practice, do they match? a case with spectrum-based fault localization. Software Maintenance (ICSM), 2013 29th IEEE International Conference on, 2013. 380–383.

[14] Yu Y, Jones J A, Harrold M J. An empirical study of the effects of test-suite reduction on fault localization. Proceedings of the 30th International Conference on Software Engineering, New York, NY, USA: ACM, 2008. 201–210.

[15] Masri W, Abou-Assi R, El-Ghali M, et al. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), New York, NY, USA: ACM, 2009. 1–5.

[16] Masri W, Assi R. Cleansing test suites from coincidental correctness to enhance fault-localization. Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, 2010. 165–174.

[17] Masri W, Assi R A. Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol., 2014, 23(1):8:1–8:28.

[18] Wang X, Cheung S C, Chan W K, et al. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. Proceedings of the 31st International Conference on Software Engineering, Washington, DC, USA: IEEE Computer Society, 2009. 45–55.

[19] Kochhar P S, Le T D B, Lo D. It's not a bug, it's a feature: Does misclassification affect bug localization? Proceedings of the 11th Working Conference on Mining Software Repositories, New York, NY, USA: ACM, 2014. 296–299.

[20] Herzig K, Just S, Zeller A. It's not a bug, it's a feature: How misclassification impacts bug prediction. Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA: IEEE Press, 2013. 392–401.

[21] Memon A M, Pollack M E, Soffa M L. Automated test oracles for guis. SIGSOFT Softw. Eng. Notes, 2000, 25(6):30–39.

[22] Aggarwal K K, Singh Y, Kaur A, et al. A neural net based approach to test oracle. SIGSOFT Softw. Eng. Notes, 2004, 29(3):1–6.

[23] Peters D, Parnas D. Using test oracles generated from program documentation. Software Engineering, IEEE Transactions on, 1998, 24(3):161–173.

[24] Davis M D, Weyuker E J. Pseudo-oracles for non-testable programs. Proceedings of the ACM '81 Conference, New York, NY, USA: ACM, 1981. 254–257.

[25] Brown D, Roggio R, Cross I, et al. An automated oracle for software testing. Reliability, IEEE Transactions on, 1992, 41(2):272–280.

[26] Chen T, Tse T H, Zhou Z. Fault-based testing in the absence of an oracle. Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International, 2001. 172–178.

[27] Chen T, Tse T, Zhou Z Q. Fault-based testing without the need of oracles. Information and Software Technology, 2003, 45(1):1 – 9.

[28] Murphy C, Kaiser G E. Automatic detection of defects in applications without test oracles. 2010..

[29] Murphy C, Shen K, Kaiser G. Automatic system testing of programs without test oracles. Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, New York, NY, USA: ACM, 2009. 189–200.

[30] Cheon Y, Kim M Y, Perumandla A. A complete automation of unit testing for java programs. 2005..

[31] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. Proceedings of the 24th international conference on Software engineering. ACM, 2002. 467–477.

[32] Steimann F, Frenkel M, Abreu R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. Proceedings of the 2013 International Symposium on Software Testing and Analysis, New York, NY, USA: ACM, 2013. 314–324.

[33] Wikipedia. Tf-idf — wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Tf%E2%80%93idf&oldid=589593815. [Online; accessed 21-January-2014].

致　谢

衷心感谢导师孙家广教授和顾明老师、宋晓宇老师对本人的精心指导。他们的言传身教将使我终生受益。

感谢清华大学学生艺术团舞蹈队的同学们，与你们在一起的时光总是非常美妙，台上台下我们共享世间美好。

感谢软件学院羽毛球队的所有同学，特别感谢嘉祥、朱晗、崇哥对我的耐心指导，你们为我打开了新世界的大门，让我感受到竞技体育特殊的魅力。

感谢父母多年来的支持。

感谢好友们一路相伴，特别感谢熊曦、包子、丢丢，是你们在我最困难的时候陪伴我纾解痛苦，让我仍能坚持到最后。

感谢实验室的兄弟姐妹，多年互相支持鼓励甚为不易。相聚是缘，后会有期。

感谢 ThuThesis，感谢 LATEX，论文因你们而优雅。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　公式证明

这是 FL 论文的公式证明

个人简历、在学期间发表的学术论文与研究成果

个人简历

1990 年 4 月 19 日出生于吉林省长春市。

2008 年 8 月考入清华大学软件学院计算机软件专业，2012 年 7 月本科毕业并获得工学学士学位。

2012 年 9 月免试进入清华大学软件学院攻读工学博士学位至今。

发表的学术论文