

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

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

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

本文的创新点主要有：

- 1;

关键词： \TeX ; \LaTeX ; CJK; 模板; 论文

Abstract

Key words: T_EX; L^AT_EX; CJK; template; thesis

目 录

第1章 绪论	1
1.1 研究背景	1
1.2 研究现状	3
1.2.1 错误定位算法	4
1.2.2 修复建议生成	4
1.2.3 修复建议验证	6
1.3 研究思路	6
1.4 论文贡献	7
1.5 论文结构	8
第2章 基于统计的错误定位	9
2.1 引言	9
2.2 相关工作	11
2.2.1 SFL 算法精度测评	11
2.2.2 影响 SFL 精确度的因素	12
2.2.3 测试准则的自动生成	12
2.3 测试 Oracle 纠错的必要性	13
2.3.1 例程	14
2.3.2 在西门子测试集上的实验	15
2.4 测试 Oracle 纠错算法	22
2.4.1 相似性度量	24
2.4.2 投票策略	25
2.4.3 参数设置	25
2.4.4 时间与空间复杂度分析	29
2.5 实验结果及分析	30
2.5.1 修复测试准则错误	30
2.5.2 SFL 算法精度恢复	31
2.6 讨论	35
2.7 本章小结	37

第 3 章 搜索引擎优化	38
3.1 引言	38
3.2 相关工作	40
3.2.1 搜索空间设计	40
3.2.2 搜索算法设计	40
3.2.3 搜索空间分析	42
3.3 “预过滤”算法	43
3.3.1 算法概述	43
3.4 搜索引擎实现	46
3.4.1 替换表达式的生成	47
3.4.2 预过滤算法的嵌入	48
3.4.3 Filter 算法实现	48
3.5 实验结果及分析	53
3.6 预过滤算法的局限性	57
3.7 本章小结	58
第 4 章 框架扩展	60
4.1 引言	60
4.2 交互式调试	61
4.2.1 概述	61
4.2.2 相关工作	62
4.2.3 交互模式	63
4.2.4 系统结构	64
4.2.5 扩展的错误定位	65
4.2.6 调试进度控制	65
4.2.7 实验结果	66
4.3 针对单类别错误的可扩展框架	67
4.3.1 概述	67
4.3.2 相关工作	68
4.3.3 框架设计	69
4.3.4 扩展示例	74
4.4 本章小结	77
第 5 章 SmartDebug 工具设计与实现	78
5.1 引言	78

目 录

5.2 功能模块.....	78
5.2.1 检查点管理模块	78
5.2.2 修复建议提示与应用模块	79
5.3 应用示例	79
5.4 本章小结	85
第 6 章 总结与展望	87
6.1 工作总结	87
6.2 研究展望	87
插图索引	89
表格索引	91
公式索引	92
参考文献	93
附录 A 公式 2-2 和 2-3 的证明	99
个人简历、在学期间发表的学术论文与研究成果	105

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	聚吡咯
PMDA-BDA	均苯四酸二酐与联苯四胺合成的聚吡咯薄膜
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

主要符号对照表

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，使应用该建议修改后的程序能够通过测试集，从而达到修复错误的目的。

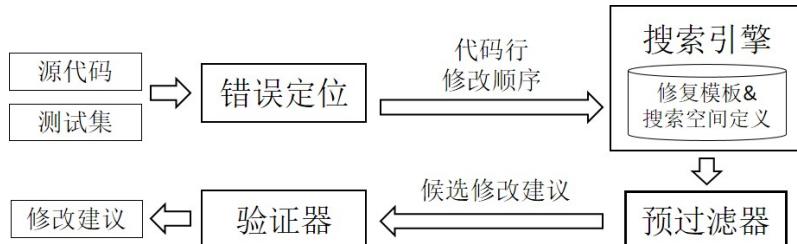


图 1.1 研究思路

图1.1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，代码1.1与代码1.2 展示了一个简单 Java 程序的源代码。主程序包含一个类 Calculator，并提供两个公有方法 add 和 minus，分别处理加、减两种操作。类 Calculator 对应的 JUnit 单元测试类 CalculatorTest 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，代码??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 将在测试代码第 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，如果不间断，“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就越长，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。这一类算法通常耗时较长，实现也稍显复杂，因此一般现有“生成-检验”系统常采用的是另一类算法，称作基于统计的错误定位算法（Spectrum-based Fault Localization，简称 SFL 算法）。

SFL 算法根据程序测试集中测试用例的覆盖（Coverage）信息，依据“被越过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，被越过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^[72]，很多实验^{[14][10]}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。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。^[12]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 2 组共 10 个计算公式在理论上超过其他现有公式。然而^[13]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是搜索空间规模与搜索时间之间的冲突。

基于搜索算法：GenProg^[37]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 程序测试集上修复成功率能够达到 77%。在此基础上，^[40] 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^[43] 首先提出了基于模式库的修复建议生成算法。在^[43] 中，该团队分析了多个开源代码项目中的代码和 6 万余条错误修改日志，提出了 9 条常见的错误修改模板。在此基础上，他们实现了系统 Par，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 119 个错误中的 27 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^[50] 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，并没有考虑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，^[46] 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 5 个错误。^[47] 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。^[48] 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。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^[46] 将修改限定为 If 条件，并且^{[47][48]} 的实验程序规模都比较小。

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，

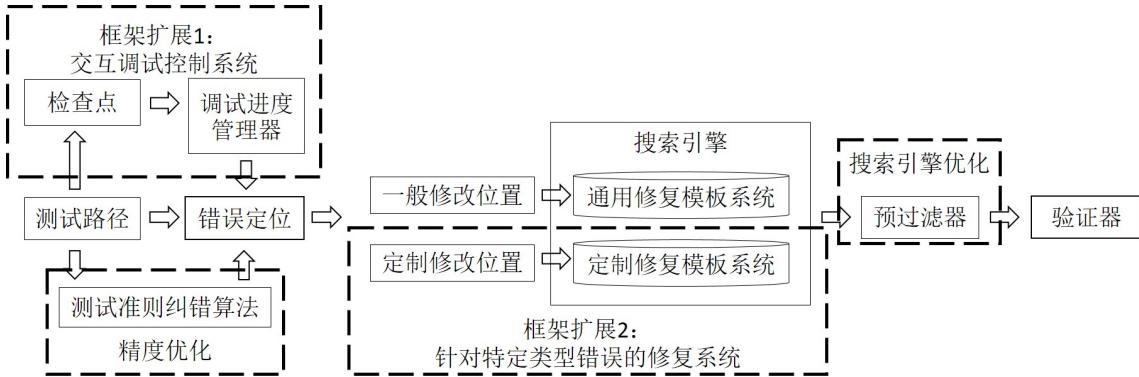


图 1.2 研究思路

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 19/69。

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。

1.3 研究思路

“生成-检验”系统研究的关键问题是如何提高系统的修复正确率与修复效率。图1.2展示了本文的研究思路。本文首先着眼于系统框架内部，针对“生成-检验”系统工作流中的两个关键步骤进行性能优化。步骤一是“错误定位”。本文提出实际程序中测试准则可能并不完善，而这将使已有的错误定位算法精度降低。本文

提出一种依据测试用例执行路径相似性判断及其测试结果识别测试准则错误的方法，使得在修正了被识别的测试准则后，已有错误定位算法精度恢复到原有水平，系统整体的修复效率不会受损。步骤二是“搜索修复方案”。本文提出当搜索引擎内部使用搜索模板系统定义搜索空间时，可以使用一种“预过滤”策略将搜索过程中生成的与表达式替换相关修复方案预先过滤、压缩，减少检验器最终需要验证的修复方案数目，提高系统整体效率。由于搜索速度提高，系统能够负担的搜索空间也相应扩大，系统的修复正确率也超过现有其他工作。

在上述工作基础上，本文提出在系统框架层面进一步扩充系统功能。本文提出了两种系统框架方案，第一种是引入开发人员与系统的交互，使得系统能够利用开发人员对程序运行状态的基本了解，更准确地定位错误，提高系统运行效率。第二种则是规范系统模块实现接口，使得开发者可以方便的在系统框架上进行二次开发，将针对特定类型的错误修复算法集成到系统中，与原系统形成优势互补。

总之，本文以提高系统的修复正确率和修复效率为目标，从模块层算法优化入手，以框架层改进方案为结束，对“生成-检验”框架进行了深入研究，具有系统性。

1.4 论文贡献

本文以基于“生成-检验”框架的程序错误自动修复算法为研究内容，从系统内部模块优化、系统框架扩展两个方面提出提高系统的修复正确率和效率的技术方案，最终形成原型工具 SmartDebug。主要贡献如下：

1. 提出不完全正确“测试准则”存在的普遍性，并以实验说明测试准则错误对“生成-检验”系统中常用的基于频谱的错误定位算法（SFL）错误定位精度的负面影响。针对这一问题，本文提出了一种基于测试用例执行路径相似性识别测试准则错误的算法。实验表明，算法能够识别出测试准则中的错误，并且 SFL 算法精度在修正后的测试准则下能够成功恢复。
2. 针对搜索引擎中与表达式替换、修改相关的修复模板，提出“预过滤”策略，根据测试集中已通过和未通过的测试用例的运行时状态在备选修复方案进入到检验器前过滤掉其中不符合要求的修复方案，同时压缩搜索空间，减轻检验器的工作负担。实验表明，“预过滤”策略能够过滤掉搜索空间中约 90% 的修复方案，使得系统效率大幅提升，系统整体修复正确率也超过现有的其他系统。
3. 在系统框架层提出两种扩展方案。第一是提供使用者与系统交互的接口，使得系统能够利用使用者的经验减少无用搜索操作，提高系统效率。第二是提

供二次开发接口，使得开发人员能够根据需求方便的引入针对特定类型错误的修复算法，使得系统本身的修复能力得到补充。

4. 实现原型系统 SmartDebug，系统中包含了本文提出的优化技术以及扩展框架，并集成在 Eclipse 平台中供开发者在日常开发过程中使用。

1.5 论文结构

本文共分为 6 个章节，其中第二章介绍测试准则错误修正算法，第三章介绍“预过滤”算法，第四章介绍两种框架扩展方案，第五章介绍系统功能与使用案例，第六章总结本文工作并给出未来研究方向。

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，测试准则是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方法相结合，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
			•		•	•	0.5	1.0
	•	•	•	•	•	•	0.5	0.5
	•			•	•	•	1.0	0.6
	•			•	•	•	1.0	0.6
	•				•		1.0	0.3
							—	—
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一行和 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。

如果有两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 $\{l_{f1}, \dots, l_{fk}\} \subset \{1, 2, \dots, N\}$ 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

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

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	$\frac{a_{ef}}{\frac{a_{ef} + a_{nf}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}}$
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2.1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2.1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排位前 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2.1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 $r \in \{0.01, 0.02, \dots, 0.1\}$ ，并记录四种 SFL 算法在所有程序变体上的精度损耗 $\Delta_{|.|}^-(\omega_j, v_i, r)$ 。图2.2展示了四种算法的绝对精度损耗随 r 的变化情况。

在图2.2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：

$$p_{\omega_j}(\Delta_x, r) = \frac{|\{k | \Delta_{|.|}^-(\omega_j, v_k, r) \leq \Delta_x\}|}{|V|}$$

其中 ω_j 表示所采用的 SFL 算法。图2.2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。

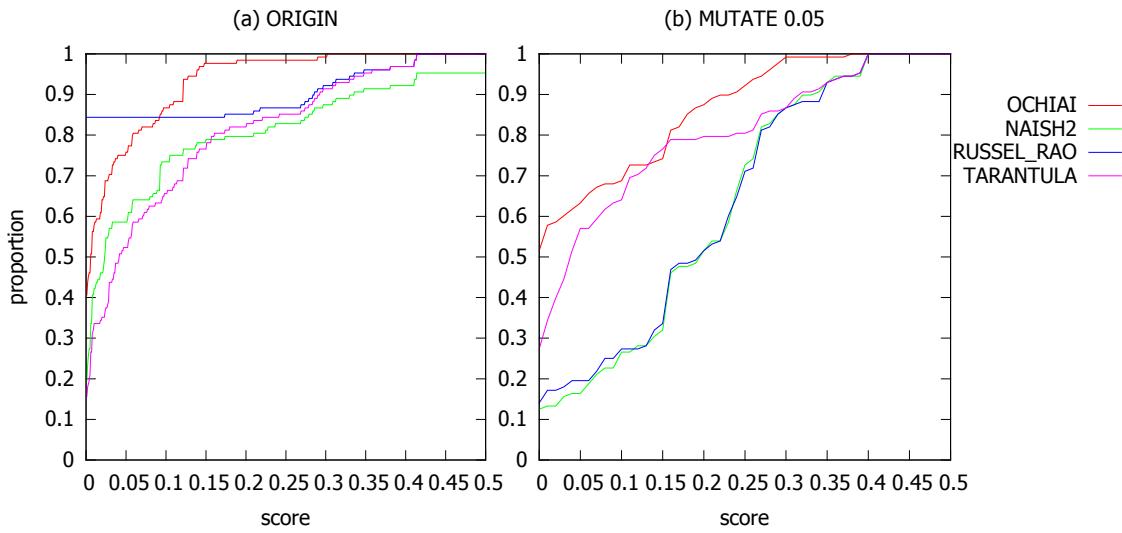


图 2.1 4 种 SFL 算法在使用正确和错误率为 0.05 的测试准则时分数 (score) 分布情况

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。

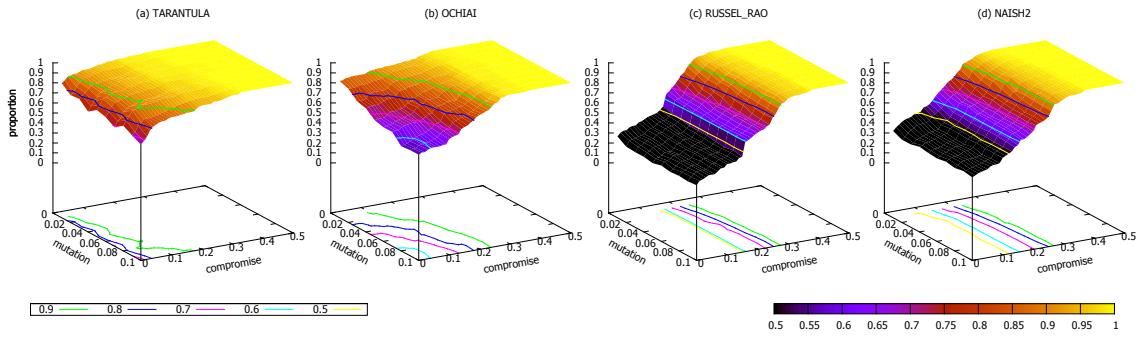


图 2.2 4 中 SFL 算法在测试准则错误率从 0.01 到 0.1 之间是精度损耗分布情况

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。

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 $T_i = \{t_{i1}, t_{i2}, \dots, t_{in}\}$ 。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

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  $T_i = \{t_{i1}, t_{i2}, \dots, t_{in}\} \subset T$  nearest to  $t_i$ ;
6:    $Suspicion(t_i) = \text{vote}(T_i)$ ;
7:   if  $Suspicion(t_i) > thres$  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  $O'(T)$ ;
```

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 $tr_i = \{i_0, i_1, \dots, i_{n_i}\}$ 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要运行。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 $tf-idf$ ^[33]。 $tf-idf$ 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。

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

2.4.3 参数设置

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，也即应当降低假阳性（false positive，简称 FP）和假阴性（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： $\forall t_{iq_1}, t_{iq_2} \in T_i \cup \{t_i\}, iq_1 \neq iq_2, \mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) = sim_i$

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

$$\mathbf{P}(fn(t_i)) \approx r \sum_{w=0}^{\hat{n}} C_n^w \phi_i^w (1 - \phi_i)^{(n-w)} \quad (2-2)$$

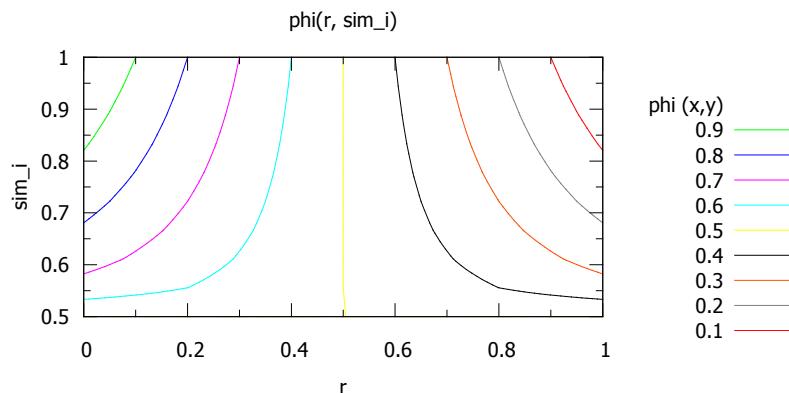
而 $O_c(t_i)$ 是 FP 的概率是

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

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A。

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2.3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 $r > 0.5$ ，并且在这一范围内， $\phi_i > 0.5$ ，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 FN 和 FP 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 $n = 3, 4, 5$, $\hat{n} = 1, \dots, 4$ 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

图 2.3 ϕ_i 随 r 和 sim_i 的变化情况

$P(fn(t_i))/r$ 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
$\phi_i = 0.7$	$\mathbf{P}(fn(t_i))/r$	0.2160	0.6570	0.0837	0.3483	0.7599	0.0307	0.1630
	$\mathbf{P}(fp(t_i))/(1-r)$	0.2160	0.0270	0.3483	0.0837	0.0081	0.4718	0.1630
$\phi_i = 0.8$	$\mathbf{P}(fn(t_i))/r$	0.1040	0.4880	0.0272	0.1808	0.5904	0.0067	0.0579
	$\mathbf{P}(fp(t_i))/(1-r)$	0.1040	0.0080	0.1808	0.0272	0.0016	0.2627	0.0579
$\phi_i = 0.9$	$\mathbf{P}(fn(t_i))/r$	0.0280	0.2710	0.0037	0.0523	0.3439	0.0005	0.0085
	$\mathbf{P}(fp(t_i))/(1-r)$	0.0280	0.0010	0.0523	0.0037	0.0001	0.0814	0.0085

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的数目。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 $\max\{O(|T|^2 O(sim)), O(|T|^2 n), O(|T| O(Sus))\} = O(|T|^2 N)$ ，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

算法1性能需两个方面评价，一是纠错后的测试准则的判断结果正确性有多大程度的提升，二是使用纠错后的测试结果应用 SFL 算法的错误定位精度有多大程度的提升。本节将以两个实验及其实验数据分别对这两个问题做出回答。

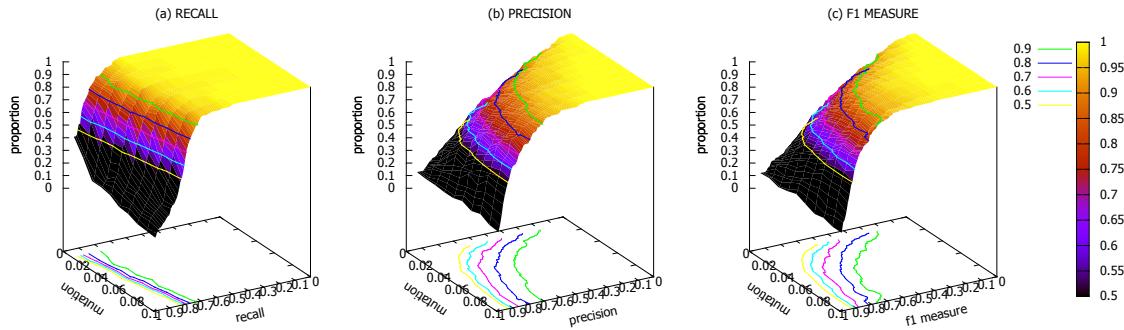
2.5.1 修复测试准则错误

实验一的目的是评价纠错后的测试准则的正确率，并与纠错前的测试准则作对比。在本实验中，我们使用西门子测试集作为对象程序，这是由于该测试集中的每个程序均含有足够的测试变体，从而能够保证测试结果的统计显著性。与第三节中的实验相似，我们首先生成正确的测试准则，并将测试准则的判断按照概率 mr 随机改变，其中 $mr \in [0.01, 0.1]$ 。

算法1在具体实现时需要确定两个参数的值，即投票集合大小 n 和纠错阈值 $thres$ 。遵照上节中的参数选择方法，我们首先随机选取了一部分测试用例作为样本，结果表明几乎所有的测试用例都有至少 10 个相似值超过 0.9 的邻居。完整的实验数据表明，事实上整个测试集中约有超过 95% 的测试用例拥有相同数量的邻居。参考表2.4中的数据，我们认为 $n = 4$ 已经足够获得较好的性能。另一方面，在 $thres$ 的设置上，根据前文的分析，只要 $2 < \hat{n} < 3$ 就可以得到一个较好的结果。最终在所有的程序上我们都使用了 $thres = 0.58$ 这一参数值。

将 $n = 4$ 和 $thres = 0.58$ 带入到算法框架中，我们在待修改的测试准则上运行了调试算法，并将输出结果与输入结果对比。按照^[32] 中的要求，为获得稳定可靠的结果，我们将实验重复了五次。实验结果的观察指标共三个，包括（1）召回率（recall），表示算法发现测试准则错误的能力，（2）精度（precision），表示算法保留测试准则正确判断的能力，（3）F1 指数（f1 measure），表示前两个指标的综合成绩。

图2.4展现了三个指标在所有程序变体和不同错误率下的分布情况。其中，给定召回率 rec 、错误率 mr ，纵坐标比率 $p(rec, mr)$ 的含义是，在错误率 mr 时，所有输出测试准则的召回率不低于 rec 的测试用例占测试集的比例 mr 。对精度 pre 和 F1 指数 $f1$ ， $p(pre, mr)$ 和 $p(f1, mr)$ 也有类似定义。如图所示，平均来看算法在超过 90% 的程序变体上达到了 85% 的召回率，也即超过 85% 的测试准则错误

图 2.4 mr 在 $[0.01, 0.1]$ 时召回率、准确率和 f1 指数分布情况

被成功发现。另一方面，算法在不同错误率的测试准则上达到的精度也各不相同。当 mr 取中位数 0.05 时，在超过 80% 的程序变体上算法的精度达到了 75%，即算法所识别出的测试准则错误中超过 75% 的确是错误。当 mr 增长到 0.1 时，算法精度上升到 80%。这一趋势也在 F1 指数的图像上显示出来，当 mr 趋向 1 时，由于召回率接近 1，F1 指数的图像与精度类似。

观察在 x-y 平面上的等高线投影可以发现一个有趣的现象。首先，当 mr 低于 0.02 时，在 50% 的测试用例上，算法识别出的错误中超过 20% 是 FP，在 30% 的测试用例上，FP 比率超过了 40%。另一方面，召回率在约 80% 左右的程序变体上超过了 90%，可见实验数据与直觉完全相符，当召回率很高时，FN 较低，FP 较高，精度也较低。其次在第四节的理论分析中，我们得出结论当 $thres$ 和 n 保持不变时， $\mathbf{P}(fn(t_i))/r$ 和 $\mathbf{P}(fp(t_i))/(1 - r)$ 均基本保持不变，因此当 r 升高时， $\mathbf{P}(fn(t_i))$ 上升而 $\mathbf{P}(fp(t_i))$ 下降。实验数据也验证了这一结论，当 mr 升高时，召回率略微下降，但精度显著上升。

2.5.2 SFL 算法精度恢复

实验二的目的是研究 SFL 算法的定位精度在使用纠错后的测试准则判断的测试结果时会恢复到怎样的程度。实验用的对象程序包括西门子测试集以及代码规模超过 13,000 行的程序 grep。

2.5.2.1 西门子测试集上的实验结果

作为基准参考，我们首先使用错误的测试准则，统计四种 SFL 算法在所有西门子测试集中的程序变体上错误定位的精度。接着使用算法 1 对测试准则纠错。最终重新统计四种 SFL 算法错误定位的精度，并与之前的精度进行对比。与第 2.3 节中的数据统计方式保持一致，错误定位精度被量化为公式 2-1 定义的分数（score）。实验重复了 5 次，实验结果如图 2.5 所示。

图2.5(a)和图2.5(b)展示了测试准则纠错前后SFL的精度的对比情况。给定错误率 mr 和分数 s_x , 图中的纵坐标比率(proportion) $p(mr, s_x)$ 表示当测试准则的错误率为 mr 时, 所有错误定位精度高于 s_x 的程序变体在测试集中所占的比例。

直观的看, 图2.5(b)中四种算法对应图像的曲面都比图2.5(a)中对应的曲面明显提高, 这意味着SFL算法的定位精度也有了明显提升。具体而言, 在使用错误的测试准则时, Russel&Rao在约20%的程序变体上定位精度分数为0(即错误代码行被排在第一位), 而当使用纠错后的测试准则时, 这一比例扩大到超过50%。对Ochiai, 这一比例从少于60%扩大到超过70%。在第三节的实验中, Tarantula表现出较好的容错性, 但这一比例从25%扩大到40%, 说明其定位精度也有了较大的提高。

为了更清晰的观察图2.5(b)和图2.5(a)之间的区别, 我们绘制了另外两组图片来展示SFL算法精度的绝对(图2.5(c))和相对(图2.5(d))提高。准确起见, 除在第2.3.2.4小节定义的 $s_0(v_i, \omega_j)$ 和 $s_r(v_i, \omega_j)$ 之外, 我们定义 $s_d(v_i, \omega_j)$ 用以表示SFL算法 ω_j 以纠错后的测试准则判断作为输入应用于程序变体 v_i 达到的定位精度, 则算法 ω_j 在测试准则错误率为 r 应用于 v_i 时精度的绝对提高定义为

$$\Delta_{\text{abs}}^+(\omega_j, v_i, r) = s_r(v_i, \omega_j) - s_d(v_i, \omega_j)$$

相对提高定义为

$$\Delta_{\text{rel}}^+(\omega_j, v_i, r) = \frac{s_d(v_i, \omega_j) - s_0(v_i, \omega_j)}{s_r(v_i, \omega_j) - s_0(v_i, \omega_j)}$$

在这两张图中, 给定 Δ_x 和测试准则错误率 r , 纵坐标比率 p 的含义是所有在测试准则纠错前后SFL的精度提高超过了 Δ_x 的测试用例所占比例。如图2.5(c)所示, 不同算法的精度在不同程度上有所提高。例如, 使用算法Russel&Rao, 在约有30%的程序变体上算法精度提高了10%, 这意味着在实际的“生成-检验”系统实现中, 搜索引擎所需检查的代码减少了10%, 搜索效率大大提升。

从图2.5(c)中的图像来看, 似乎在许多程序变体上SFL的精度提高并不明显。这是因为对于大多数程序而言, 哪怕是使用错误的测试准则作为输入, SFL的定位精度分值仍然在30%以下, 因此绝对提升的空间有限。但是图2.5(d)显示, 对超过50%的程序变体而言精度的相对提升幅度较大, 提升最明显的是Russel&Rao, 在超过50%的程序变体上定位精度的相对提升幅度超过了50%。

从图2.5中可以看出, SFL算法的定位精度在测试准则经过纠错后有了明显恢复。由于这四种算法都是研究公认的“最优”或应用最广泛的是算法, 因此我们认为

表 2.5 grep 基本信息

版本号	植入错误数	可检测的错误数	代码行数
v1	18	2	12654
v2	8	1	13231
v3	18	2	13374
v4	12	2	13360
v5	1	0	13293

为这一结论可以推广到 SFL 一族的其他算法上。

2.5.2.2 grep 上的测试结果

grep 是类 Unix 系统中常用的字符串匹配程序，其功能是在文本文件中查找符合某一模式的字符串。SIR^[7] 提供了 grep 的 5 个发布版本，每个版本都包括 18 个有植入错误的程序变体及其相应的测试集。初步测试显示，很多程序变体中的错误无法被测试集中的测试触发，因此本节实验的程序对象将这部分程序变体移除了。特别的，版本 5 只有一个程序变体，但其中的错误无法被触发，因此版本 5 整体被移除了。表2.5列出了基本发行版本的基本信息。

与在西门子测试集上的实验设计相同，我们首先利用未植入错误的程序版本为所有程序变体生成了正确的测试准则，接着按概率 $mr \in [0.01, 0.1]$ 随机改变测试准则的输出获取错误的测试准则，最终比较 SFL 算法在错误的测试准则和纠错后的测试准则上的定位精度。实验重复了 5 次，图2.6展示了实验结果。

图2.6展示了在四个版本的 grep 上 SFL 的错误定位精度。鉴于总共只有 4 个版本，我们可以更清晰的看到算法1对每个程序的影响。对使用同一 SFL 算法的每个版本的程序，红线表示当测试准则错误率在 $[0.01, 0.1)$ 上变化时定位精度的变化情况，绿线表示使用纠错后的测试准则时的定位精度。在大多数程序变体上，所有 SFL 算法的定位精度都有所提高。其中变化最显著的是 Russel&Rao, Ochiai 的精度虽没有明显提高，但也没有降低。

值得注意的是，西门子测试集中，每个程序都有约 4000 个测试用例，因此算法能够很方便的利用测试用例之间的相似性达到较好的召回率和精度。与之对比，SIR 为 grep 提供的测试用例集是“能够代表实际开发过程中使用的测试用例”^[7]，所包含的测试用例只有 199 个，实验数据显示算法仍然取得了较好的效果。这说明算法1能够应用于实际程序的真实测试集。

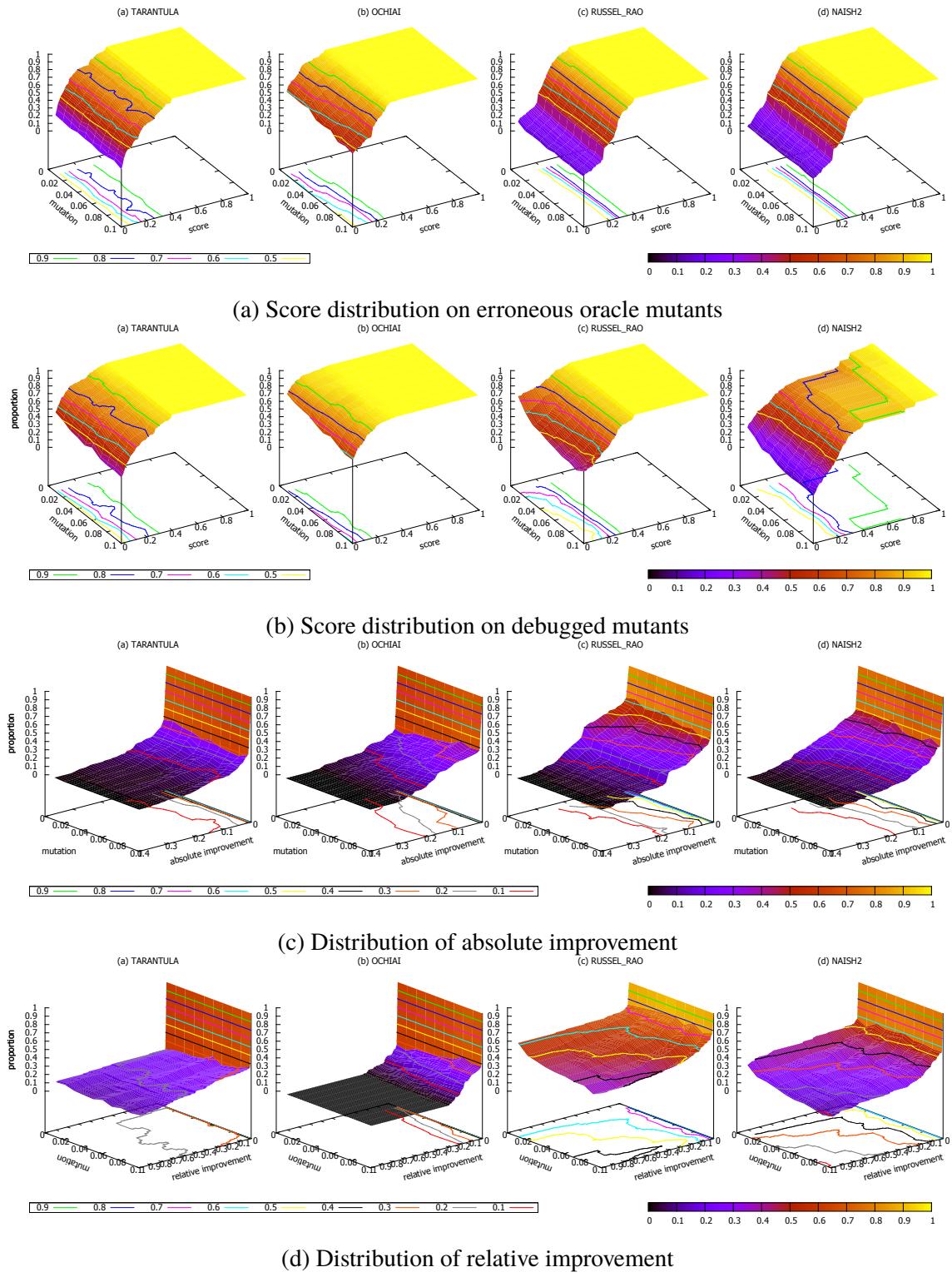


图 2.5 4 种 SFL 算法在修正后的测试准则下的定位精度恢复情况

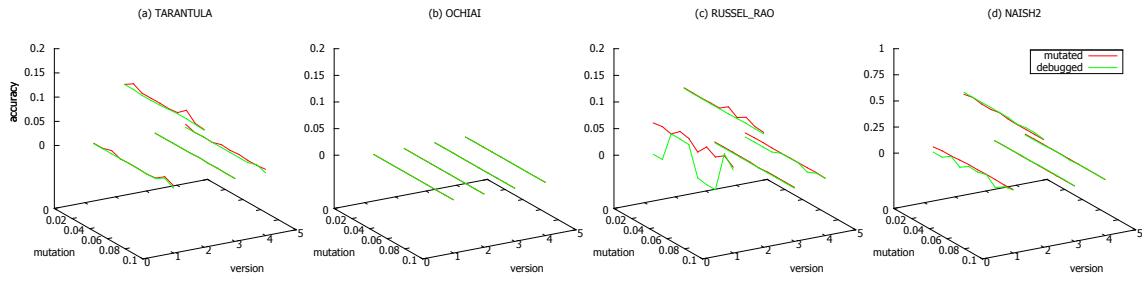


图 2.6 SFL 算法在 grep 程序上使用正确与错误测试准则的精度对比

2.6 讨论

在第2.5节的实验二中，我们观察到随着测试准则错误率的上升，SFL 算法在纠错后的测试准则上的精度提升有小幅度的下降。这一趋势恰好与召回率随测试准则错误率的变化趋势相同，那么召回率与 SFL 算法的精度提升幅度之间是否有不通过测试准则错误率的直接关系呢？考虑到 Russel&Rao 的精度提升最明显，我们计算了绝对精度提高和相对精度提高在不同召回率上的分布情况，图像如图2.7所示。

在图2.7中，我们画出了当绝对（相对）精度提升在 0 和 1 之间取值时，达到不同召回率的程序变体所占的比率。尽管曲面有许多皱褶，我们仍然可以从 $x - y$ 平面的等高线投影上看到精度提升随着召回率的升高而升高的大致趋势。

从这一趋势可以得出的一个直接结论是，与测试准则的错误率无关，一般而言较高的召回率也对应着较大的 SFL 的精度提高幅度。因此，只要调整算法中的参数获得较好的召回率，本节所提出的方法也能够在测试准则错误率超过 0.1（本实验中的上限）场景下应用。

此外，我们还可以推断出测试准则的哪一类错误，即“假通过”或“假失败”，对 SFL 算法的精度影响更大？这一问题的重要性在于，在实际的软件开发过程中，尽量避免测试准则错误总是比使用 SFL 定位错误更加经济。分析实验数据我们发现，对于西门子测试集中的绝大多数程序变体，大约有 5% 的测试用例能够触发程序中的错误，这导致大多数的测试准则错误是“假失败”，而“假通过”很少发生。当召回率较高时，几乎所有的测试准则错误都会被纠正过来，这大幅减少了“假失败”的数量，但对“假成功”影响较小。这样一来，召回率实际上可以被当做纠错后的测试准则其中所包含的错误类型分布的观察指标，即较高的召回率通常意味着较少的“假失败”和较多的“假成功”。由图2.7中的趋势可以看出，较少的“假失败”也和更明显的 SFL 定位精度提高同时出现。因此我们可以得出结论，“假失败”的确对 SFL 定位精度的影响更严重，在实际开发过程中应当尤其注意避免。

本章中第三节与第五节中的实验均属于经验研究，其结论的正确性可能受到

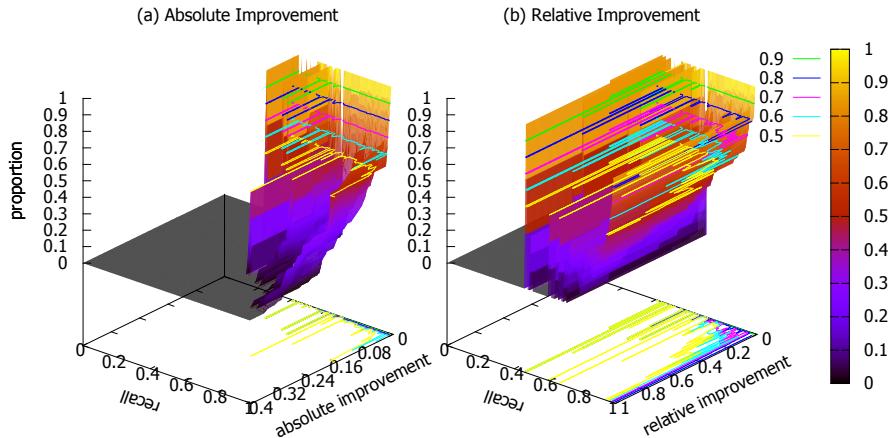


图 2.7 绝对与相对精度提升与召回率的关系

多种因素的影响。我们参考^[32] 中总结出的几个方面作如下分析。

第一个因素是被研究程序对象是否具有代表性。在第2.3节的实验研究中，研究对象是西门子测试集。西门子测试集的一个问题是其中包含的被测程序的规模都比较小，因此实验得出的结论可能无法完全准确的反映测试准则错误对大型程序的影响。但是这并不影响我们得出的测试准则错误会对错误定位精度产生影响这一结论。在第2.5节，我们使用超过 13,000 行代码的真实程序 `grep` 作为研究对象，并在第2.5节证明错误定位准确度由于修正了测试准则而提高，由此我们认为本章所提出的算法可以应用于一般规模的真实程序。

第二个因素是被研究程序的错误种类是否全面，即是否同时考虑了单一错误及多种错误两种情况。本章中的实验对象均是单一错误对象，但是在第2.3.2.4小节的分析中我们指出，许多单一错误对程序的实际影响可以模拟多处错误，例如一个错误的常量定义可能导致程序中所有使用这一常量的地方都是错误的。实验中我们也刻意保留了这些错误，使实验结果能够更贴近实际应用。

第三个因素是采样规模，即被研究程序对象的数量。^[32] 中强调需要至少 300 个被测程序实验的统计结果才能稳定，因此我们进行了 5 次重复实验来减少实验结果的随机性。

最后^[32] 中提到的一个因素是测试集的质量，即测试集的规模和覆盖率。西门子测试集为每个被测程序都提供了充足的测试用例，测试集的覆盖率也足够好。程序 `grep` 只包含了 199 个测试用例，但由于我们希望实验结果能够“代表实际开发过程中的程序”，^[7] 因此没有对测试集作进一步的扩充。综合两种类型的测试集，我们认为本章的实验结果能够充分支持实验结论。

2.7 本章小结

在本章中，我们首先提出测试准则错误会对“生成-检验”系统中的第一步“错误定位”造成负面影响。为了确认这一影响，我们在西门子测试集上进行试验，测试 SFL 算法的定位精度受测试准则错误的影响。与使用完全正确的测试准则相比，实验数据使用有错的测试准则会使得不同的 SFL 算法的定位精度有不同程度的下降。因此得出结论，应在错误定位这一环节前修正测试准则错误。

基于一个简单的假设，即“覆盖相似代码行的测试用例通常同时通过或不通过”，我们提出了基于代码行覆盖情况度量测试用例间相似度的方法，并在此基础上提出利用邻居测试用例投票估计测试准则对测试结果的判断出错的可疑程度，并将可疑程度超过某一阈值的测试准则判断翻转，作为测试准则纠错的结果。实验显示，这一算法能够识别出大多数的测试准则判断错误。更进一步的，我们在西门子测试集和 grep 上比较了 4 种 SFL 算法在使用纠错前和纠错后的测试准则判断时的定位精度。实验结果显示，4 种 SFL 算法的定位精度均有恢复。最后，通过对实验数据的进一步分析，我们得出结论，本章算法可以适用于测试准则的错误率超过实验中所设的上限 0.1 的情况。

第3章 搜索引擎优化

3.1 引言

在“生成-检验”系统中，搜索引擎负责生成程序变体，最终将其交由检验模块来验证该变体是否能够通过测试集。在搜索引擎模块的设计中，搜索空间的规划以及搜索算法直接影响系统整体的修复正确率和运行速度。采取一个广大的搜索空间可以提高系统的修复正确率，但是也意味着需要搜查的范围扩大，系统运行速度降低。因此，提高搜索算法的效率可以使得系统在可接受的时间范围内能够搜索到的空间更大，使得提高修复正确率成为可能。

现有系统的搜索算法可以分为两大类。第一类是基于构造或程序综合方法精准的生成程序变体。算法首先确定程序变体的基本结构，如需要替换表达式的具体位置和语法类型，接着利用静态分析、符号执行等技术，结合“程序变体需通过测试集中的所有测试”这一要求，生成逻辑公式描述替换表达式所应满足的语义要求，最后构造或利用程序综合技术直接生成满足要求的表达式。这一方法的优点在于“精确”，即理论上最终生成的表达式一定是满足逻辑公式要求的，因此将其替换进程序所生成的程序变体应直接通过测试集，几乎不需要再经过检验模块的验证。然而实际系统实现时我们会发现，生成逻辑公式这一过程非常耗时，因此将该算法应用于大规模程序有一定的难度。

另一类搜索算法基于预定义的修改模板生成程序变体。搜索算法首先在引擎内部预设了一组修改模板，用于描述在某些代码上下文中可能出现的错误情形以及相应的程序变体生成方法。例如，模板空指针检查器（Null-Checker）用以消除空指针错误，其内容是在某个指针的解引用前（上下文），插入一个判断指针是否为空的检查语句（变体生成方法）。依照这些模板，搜索算法能够针对不同的程序上下文生成合理的程序变体。因此，模板设置的合理性对系统的正确性和搜索的效率起到了决定性的作用。当模板能够覆盖的错误情形较多时，搜索算法能够覆盖到的程序变体范围也比较广，系统的修复正确率也可能有所提高。但若模板类型过多，搜索算法则依照许多并不常见的修改方式生成程序变体，而最终检验模块将耗费大量时间运行和抛弃这些程序变体，如此系统的整体效率也会降低。

近年 MIT 人工智能实验室发表的系统 SPR 提出了结合以上两种方法的搜索算法“分阶段的程序修复（Staged Program Repair，简称 SPR）”。借鉴基于修改模板的搜索引擎，SPR 首先预定了能够覆盖其他系统搜索空间的一组修改模板，接着在依照与 If 条件相关程序变体时，对 If 条件中的布尔表达式执行 SPR 算法。算法的

基本思想是，首先通过尝试给出某一布尔值序列，使得目标布尔表达式在每次求值时若依照这一序列取值则程序能够通过测试集。接着在程序执行过程中，计算目标表达式所处的程序环境下其他合法布尔表达式的取值情况，若有与该序列相符的布尔表达式则可用它替换目标表达式。由于 SPR 只生成一种布尔值序列，因此在所有合法的布尔表达式集合中只有非常少的一部分能够符合要求，因此实际被生成出来的程序变体数目也极少，事实上在实验中 SPR 报告有超过 90% 的布尔表达式在第二步时被丢弃，这导致最终检验模块需要处理的程序变体较以往系统大大减少。

从实验效果来看，SPR 是目前最好的搜索算法，但它仍有一定的局限性。SPR 算法对搜索空间的压缩是通过减少 If 条件中的布尔表达式的搜索空间完成的，但是对更一般的表达式的搜索空间没有起作用。事实上，在 SPR 系统采用的修复模板中，仍有许多条是直接与一般表达式相关的，因此若能够将一般表达式的搜索空间以相似的方式进行压缩，SPR 的搜索效率会进一步提高。然而，在 SPR 框架下，这一压缩方式无法扩展到一般表达式上。这是因为 SPR 的第一步需生成目标表达式的取值序列，然而对于一般表达式，其可能的取值远不止 true 或 false 两种，生成一个确定的取值序列是不现实的。

针对这一问题，本章提出一种新的搜索空间压缩算法，称为“预过滤（prefilter）”策略。算法的基本思想是利用不同测试用例的执行过程生成对替换表达式的约束条件。具体而言，由于能够成功修复错误的程序变体必须保证原本已经通过的测试用例仍然能够通过，因此替换表达式在已通过的测试用例中的取值应与目标表达式一致。而另一方面，程序变体在未通过的测试用例上的执行过程应与原程序不同，因此替换表达式在未通过的测试用例上的取值应与目标表达式不同。利用这一约束，算法在每次程序对目标表达式求值的同时也对搜索空间中的替换表达式求值，并将不符合约束的目标表达式滤除。这一过程发生在生成程序变体之前，因此称为“预过滤”。

预过滤方法对目标表达式的语法类型没有严格限制。它能够兼容 SPR 算法针对的布尔类型表达式，也能够扩展到其他基本类型以及面向对象语言（如 Java）中的对象类型。在本章中，我们实现了一个针对 Java 程序的“生成-检验”系统 PFDebug，并将预过滤方法实现在搜索引擎中。我们在 defect4J 测试集上进行了实验，结果表明，预过滤技术可以将一般类型表达式的搜索空间压缩 90% 左右。这也使得系统可以负担较宽泛的搜索空间，最终系统能够成功修复其中的 24 个错误，这是目前在该测试集上报告的最好成绩。

本章的主要贡献如下：

- 提出预过滤搜索算法思想，并给出在常用修复模板上该算法的实现方法
- 实现了一个完整的“生成-检验”系统 PFDebug，完成预过滤算法的具体实现，并在 defects4J 测试集上进行实验，给出预过滤算法对搜索空间的压缩效果的量化分析
- 将预过滤算法与 SPR、Nopol 等系统的搜索空间进行对比分析，提出优缺点，并给出可能的改进方向

3.2 相关工作

搜索引擎的设计是“生成-检验”系统设计的核心问题，现有研究工作可分为三个部分。其一是搜索空间的设计，包括搜索空间的组织结构及覆盖范围。二是搜索算法的设计，即给定搜索空间后如何高效搜索。三是对已有系统搜索空间的分析，即分析比较搜索空间不同实现对系统整体修复正确率和效率的影响。

3.2.1 搜索空间设计

针对搜索空间设计的研究主要目标是划定搜索空间的合理范围，使得被覆盖的程序变体能够有更大的可能成为正确的修复方案。例如，在^[34]中，作者分析了 7 个大型 Java 开源项目代码库，将其中与“修正错误”“补丁”等相关的代码提交会话找出，在抽象语法树（Abstract Syntax Tree，简称 AST）层面上分析这些会话中前后两个版本代码之间的差异，并总结出 9 大类常见代码修改方式。在^[35]中，作者借助代码比较工具 ChangeDistiller^[36] 分析了 14 个开源 Java 项目，并统计了在所有“修复回话（fix-commits）”中 ChangeDistiller 所定义的修改类型（Change Type）及修改实体类型（Change Type Entity Type）所占的比例。作者得出结论，不同修改类型所占的比例在不同项目中不一致，因此搜索引擎内部预置一个概率模型指导搜索过程在某些程序上会导致搜索速度降低。无论怎样，以上两篇文章均认为搜索引擎可以依照有限的模式生成程序变体，使得修复系统能够覆盖一定比例的常见修复方案。

3.2.2 搜索算法设计

搜索算法设计的研究工作较多，我们按照设计思想和发展关系将其分为以下几组分别介绍：

GenProg, RSRepair, AE, Kali: 这一组算法的设计思想都是“搜索框架 + 程序变换”。GenProg^{[37][38]}最早提出了使用遗传算法搜索框架 + 程序变换的方式解决错误修复问题。作者认为，对程序中的一段错误代码，通常可以在程序的其他

位置找到能够将其替换和修复的正确代码。基于这一观察，GenProg 在对象程序源代码中剪切粘贴代码片段生成程序变体，将其作为“种群”，以程序变体所能通过的测试数目作为评价函数，交由遗传算法框架进行搜索。其优点是搜索空间较大，对错误类型也无限制。而缺点是搜索方向难以控制，搜索时间也比较长。^[39] 则提出设计更加的评价函数（Fitness Function）提高遗传算法搜索效率。文章^[40] 实现了系统 RSRepair，并指出在实验过程中发现，采用随机搜索算法能够达到的修复成功率与 GenProg 接近，且系统运行时间短效率高。AE^[41] 舍弃了随机化搜索框架，它首先固定了程序变体的生成方式，接着提出一种将语义等价但语法不等价的程序变体归为一个等价类，最终以程序（或程序变体）所需执行的测试数目为指标定义了适应函数套用适应性搜索算法。文章中报告的实验结论认为该算法的效果优于 GenProg。但另一篇文章^[42] 认为，以上三种算法生成的程序补丁实际上没有修好错误，而是“删除了系统功能”。为了证实这一说法，作者开发了“功能移除系统” Kali，该系统仅移除代码而不生成新代码。实验表明，Kali 的“修复”效果不弱于前几个系统，因此这些算法的有效性有待考察。

Par:^[43] 最早提出了基于修改模板的搜索算法。作者首先分析了 60,000 余个开源代码库中的代码补丁，并得出结论约有 30% 的错误可以被 8 种修改模式覆盖。在此基础上，作者实现了系统 Par，并在 119 个真实错误上进行实验，成功修复了其中的 27 个错误。另外，作者认为由分析开发人员提交的代码补丁分析出的修改模板所生成的修改方案更容易被开发人员理解。为证实这一结论，作者邀请了学生与专业开发人员分别评价 Par 和 GenProg 生成的补丁，文章称评价人员认为 Par 生成的补丁更易被理解。Par 系统的优点是模板简单，生成较快，而缺点是模板覆盖的搜索空间有限，修复正确率也比较低。在^[42] 中，作者也认为 Par 系统的实验数据有误，但并没有得到 Par 作者的回应。因此我们无法判断 Par 真正的修复正确率是多少。

“天使调试”（**Angelic Debugging**）和 **NOPOL**: 天使调试（Angelic debugging）是在^[45] 中首次提出。其思想是，在测试执行过程中将目标表达式替换为符号表达式，执行符号分析，获取为使程序能够通过测试集中的所有测试的符号条件。如果存在符号变量的某一具体取值能够使得符号条件满足，则目标表达式是一个待选的修复位置。天使调试的输出结果是一列可能的修复位置而不是最终的修复方案。NOPOL^[46] 借鉴了天使调试的思想。NOPOL 只能修复错误的 If 条件表达式，首先它在测试运行过程中动态地修改条件表达式的取值，找到能使程序通过测试集的布尔值，即替换表达式应满足的取值条件。接着，它分析在目标表达式位置上的合法布尔表达式，用逻辑公式描述其语义并利用程序综合方法生成满足条件的替

换表达式。NOPOL 的优点是，其生成的布尔表达式结构可以比较复杂，但缺点是无法顾及其他可能的错误情况，且对目标表达式构成元素的逻辑分析也受符号执行器的能力限制。

SemFix and DirectFix: SemFix^[47] 是第一个使用符号执行技术生成目标表达式约束的完整系统。与天使调试类似，SemFix 用符号变量替换目标表达式，接着对通过和不通过的测试用例执行过程分别做符号分析，根据测试准则的要求生成符号变量需要满足的条件。最后利用程序综合技术利用某一范围内的构造元素（如变量、函数等）生成符合要求的替换表达式。该算法的优点是生成的替换表达式精确地符合测试集的要求，节省检验器的验证时间，而缺点是受限于现有符号执行器的处理能力，算法很难应用于较大规模的程序。此外，算法仅能生成替换单一表达式的修复方案，更复杂的情况将无法处理。在 SemFix 基础上，同一研究组发表了系统 DirectFix^[48]。DirectFix 的目标是生成更简单的修复方案，算法的思路是将目标表达式的位置与替换表达式应满足的条件全部用逻辑公式描述，并将其交给 MaxSAT 求解器进行求解。由于 MaxSAT 求解的结果通常比较简练，因此最终生成的替换方案也比较简单易懂。

SPR and Prophet: SPR 系统^[49] 由 MIT 人工智能实验室提出，其搜索引擎内部定义了一组修复模板，其搜索空间覆盖了 Par、GenProg、AE 等系统的搜索空间。同时，它针对其中与 If 条件布尔表达式相关的修复模板提出了分阶段修复算法（Staged Program Repair，简称 SPR）。SPR 的基本思路是，在测试运行过程中动态调整目标布尔表达式每次的取值，最终获取一个能够使不通过的测试用例通过的取值序列。接着 SPR 记录目标表达式位置处的合法表达式的取值情况，挑出符合该取值序列的目标表达式，将其填入模板的对应位置。实验显示，经过取值序列的过滤，约有 90% 的修复方案可以被滤除，大大压缩了搜索空间。在 SPR 基础上，Prophet^[50] 提出了一种对修复方案排序的算法，该算法主要应用于检验阶段，因此不在本章做详细介绍。

3.2.3 搜索空间分析

在^[52] 中，作者在 SPR 和 Prophet 系统的基础上使用搜索空间的不同配置在同一组对象程序上实验，并观察不同配置下搜索空间的变化与系统整体修复成功率和运行速度之间的关系。文章得出了两个主要结论。第一，搜索空间中正确的修复方案非常稀疏，而能够使测试集通过但事实上并不正确的修复方式却相对密集许多，因此利用测试集之外的知识进一步筛选修复方案非常重要。第二，盲目扩张搜索空间并不一定会带来修复正确率的提升，事实上在给定某一时间上限时，扩

张搜索空间会导致错误的修复方案增多，在搜索到正确修复方案之前所花费的时间也增多。

3.3 “预过滤” 算法

根据现有研究工作的进展情况，本文认为 SPR 将预定义的修复模板与搜索技术相结合的技术是“生成-检验”系统搜索引擎较好的设计方法。但由于其空间压缩算法的应用对象只能是 If 的布尔条件，SPR 的搜索算法仍有缺陷。本节我们提出一种新的搜索空间压缩办法，称为“预过滤”算法。该算法的应用对象能够从布尔表达式相关的修复模板扩展为一般表达式相关的修复模板，弥补了 SPR 算法的不足之处。

3.3.1 算法概述

“预过滤”算法的基本思路是利用测试集中已通过和未通过的测试用例的执行过程压缩替换表达式的搜索空间。具体而言，对于一个涉及替换表达式的修复模板，若生成的修复方案中将目标表达式 A 替换为表达式 B ，我们有以下观察：

- 为保证替换后的程序在已通过的测试用例上应仍能够保持通过， B 应在每次对 A 求值时与 A 的值保持一致
- 为保证替换后的程序能够在未通过的测试用例上通过， B 在所有对 A 求值的上下文中应至少有一次与 A 的值不相同

利用以上观察，在将 A 替换为 B 并经过检验器验证之前，搜索引擎可以判断该替换是否可能修正程序的错误并滤除不符合条件的替换表达式。这一过滤算法称为“预过滤”算法，我们用以下例程说明“预过滤”的基本思路。

Listing 3.1 错误示例 jfreechart-1.1/ShapeUtilities.java

```

1 public static boolean equal(GeneralPath p1, GeneralPath p2) {
2     if (p1 == null) {
3         return (p2 == null);
4     }
5     if (p2 == null) {
6         return false;
7     }
8     if (p1.getWindingRule() != p2.getWindingRule()) {
9         return false;
10    }
11    PathIterator iterator1 = p1.getPathIterator(null);
12    //ERROR: 下句 p1 应为 p2
13    PathIterator iterator2 = p1.getPathIterator(null);
14    double[] d1 = new double[6];

```

```

15  double[] d2 = new double[6];
16  boolean done = iterator1.isDone() && iterator2.isDone();
17  while (!done) {
18      if (iterator1.isDone() != iterator2.isDone()) {
19          return false;
20      }
21      int seg1 = iterator1.currentSegment(d1);
22      int seg2 = iterator2.currentSegment(d2);
23      if (seg1 != seg2) {
24          return false;
25      }
26      if (!Arrays.equals(d1, d2)) {
27          return false;
28      }
29      iterator1.next();
30      iterator2.next();
31      done = iterator1.isDone() && iterator2.isDone();
32  }
33  return true;
34 }

```

代码3.1来自 defects4J^[55] 测试集中程序 JFreeChart 的一个错误版本。代码展示了一个计算两个路径 GeneralPath 对象是否相同的静态方法，其算法思路如下：首先判断两个输入对象是否为空值（2-7行），接着比较两个对象的 windingRule 属性是否一致（8-10行），最后比较两个路径对象（p1, p2）包含的实际数据是否一致（11-33行）。在最后一步中，代码首先构造了两个内部数据访问迭代器（11-13行），接着利用迭代器逐段比较两个路径是否一致（17-32行）。代码中的错误是，在 13 行构造路径 p2 的迭代器时应调用 p2 的方法而实际上调用了 p1，这导致两个迭代器实际上都是在同一个路径对象 p1 上遍历，因此只要方法输入参数不为空，返回值永远是 true。

方法 equals 被测试集中的两个测试用例所覆盖，分别是 ShapeUtilitiesTest.java 中的 testEqualShapes 和 testGeneralEqualPaths。在测试执行过程中，testEqualShapes 通过了，但 testGeneralEqualPaths 没有通过，其原因是 testEqualShapes 中只测试了两条路径相同的情况，因此 equals 方法的返回值永远是正确的，而 testGeneralEqualPaths 则覆盖了两条路径不相同的情况，因此错误被触发。

对“生成-检验”系统的搜索引擎来讲，当已经把错误定位到 13 行时，会为表达式 p1 生成一系列替换表达式。在此处的合法表达式有 p2，及 new GeneralPath() 等等。我们以这两个表达式为例说明“预过滤”算法如何利用前述观察滤除不合理的替换表达式。首先，由于 testEqualShapes 调用该方法时输入的

路径都是相等的，`p1 -> new GeneralPath()` 不能使已通过的 `testEqualShapes` 方法通过但 `p1 -> p2` 可以，因此应首先将 `new GeneralPath()` 这一表达式从替换表达式中剔除。另外对于未通过的测试用例 `testGeneralEqualPaths`，修改后的程序执行过程中迭代器 `iterator2` 应当发生改变，因此 `p2` 应当被保留。此时，经过预过滤处理，`new GeneralPath()` 被过滤掉，而 `p2` 被保留，最终搜索引擎会仅会生成 `p1 -> p2` 这一修复方案。

以上是预过滤策略的应用示例，下面以伪码给出算法描述：

Algorithm 2 预过滤算法 (Filter)

Input: $P, T_c, T_f, n, N, E_{in}, e_t$

Output: E_{out}

```

1:  $n \text{++;}$ 
2: for each  $t_i \in T_c$  do
3:   while ( $n < N$ ) do
4:     for each  $e_r \in E_{in}$  do
5:       if (!Equal(Eval( $e_r$ ), Eval( $e_t$ ))) then
6:         Remove( $E_{in}, e_r$ );
7:       end if
8:     end for
9:   end while
10:  end for
11:  Init( $E_{out}$ )
12:  for each  $t_i \in T_f$  do
13:    while ( $n < N$ ) do
14:      for each  $e_r \in E_{in}$  do
15:        if (!Equal(Eval( $e_r$ ), Eval( $e_t$ ))) then
16:          Update( $E_{out}, e_r$ );
17:        end if
18:      end for
19:    end while
20:  end for
21: return  $E_{out};$ 

```

算法2描述了预过滤策略在压缩单一表达式替换的搜索空间时的操作步骤。算

法输入包括 7 项，分别是目标程序 P ，已通过的测试用例集合 T_c ，未通过的测试用例集合 T_f ，当前过滤次数 n ，过滤次数上限 N ，替换表达式搜索空间 E_{in} 及目标表达式 e_t 。其中，搜索空间 E_{in} 表示所有合法替换表达式的集合。当前过滤次数 n 在每次调用 Filter 时增加 1，用于表示已对搜索空间 E_{in} 执行压缩操作的次数。过滤次数上限 N 表示对 E_{in} 压缩的次数上限，规定这一值的意义在于当程序执行到 e_t 的次数过于频繁时仍保证 Filter 算法仅被调用有限次，外层的搜索算法可以终止。

算法的执行过程分为两个部分，前半部分对应观察 1，即对于所有已通过的测试用例（第 2 行），替换表达式应与目标表达式的求值结果相同（第 5 行），否则应将其从搜索空间中剔除（第 6 行）。这一过程结束后我们获得了一个压缩后的 E_{in} ，此时进入算法的下半部分。下半部分对应观察 2，即对于所有未通过的测试用例（第 12 行），替换表达式应至少有一次与目标表达式的求值结果不同（第 15 行），若满足则将其添加到 E_{out} 中（第 16 行）。最终输出 E_{out} 。此处值得注意的是，算法 11 行将 E_{out} 初始化为空，而后半部分仅将新发现的符合要求的替换表达式 e_r 加入 E_{out} 而不改变 E_{in} ，这是由于替换表达式只要有一次与目标表达式求值结果不同即可，因此本次执行 Filter 时不满足条件仍可以在 E_{in} 中保留。

算法 2 描述了 Filter 算法的主体结构，但并未给出内部引入的方法 Equal, Eval, Remove 和 Update 的具体实现。由于这些方法的实现与所处理的程序对象密切相关，我们将在下节结合程序对象与搜索引擎的整体实现具体说明。

3.4 搜索引擎实现

为说明 Filter 算法如何嵌入在搜索引擎整体的搜索框架中，本节我们给出以 Java 程序为修复对象的搜索引擎的具体实现。在此基础上，结合 Java 语言本身特性算法 2 中几个未实现函数的具体实现方式。

算法 3 描述了搜索引擎的搜索逻辑。其输入包括对象程序 P ，已通过的测试用例 T_c ，未通过的测试用例 T_f ，过滤次数上限 N ，由错误定位模块生成的代码行排序列表 L ，预定义的搜索模板集合 S ，输出对象是修改方案集合，也即程序变体集合 M 。其中，错误定位结果 L 将所有代码行按出错可能性由高到低排序。算法按照 L 中给出的搜索顺序依次以代码行 l 为修改位置生成修复方案。对每一个代码行 l ，算法将逐一使用修复模板集合 S 中的修复模板 s 生成修复方案（第 2-5 行），构成集合 M_l 。随后，算法调用 Filter 函数对 M_l 进行过滤（第 8 行），最终将过滤结果存入 M 中。

搜索引擎的逻辑比较简单清晰，但具体实现则需考虑 Java 语言本身的特性。本章在 Java 语言操作部分是基于 Eclipse 的 JDT 组件^[56] 实现的，以下几小节将对

Algorithm 3 搜索框架**Input:** P, T_c, T_f, N, L, S **Output:** M

```

1: for each  $l \in L$  do
2:   for each  $s \in S$  do
3:      $E_{in} = \text{GenE}(l)$ 
4:      $M_l = \text{GenM}(l, s, E_{in})$ 
5:   end for
6:    $n = 0, \text{Init}(h), \text{Init}(b)$ 
7:    $\text{Register}(h, b);$ 
8:    $M_f = \text{filter}(P, T_c, T_f, n, N, E_{in}, M_l.E_r, l.e_t)$ 
9:    $\text{Unregister}(h, b);$ 
10:   $\text{Add}(M, M_f)$ 
11: end for
12: return  $M;$ 

```

引擎实现的细节进行进一步描述。

3.4.1 替换表达式的生成

算法3第3行中，`GenE` 函数以代码行 l 为输入，生成在这一行上符合语法规则的表达式集合。在 Java 语法中，代码行 l 所处的环境及其可访问的表达式对应关系主要有以下几种：

- 动态成员方法中的语句：此处可访问的当前类的所有成员变量及成员方法，父类的非 `private` 类型成员变量和成员方法，程序中其他可公开访问的静态变量和静态方法，本地局部变量
- 静态成员方法中的语句：此处可访问当前类的静态成员变量及静态成员方法，父类的非 `private` 类型静态成员变量和静态成员方法，程序中其他可公开访问的静态变量和静态方法，本地局部变量
- 动态成员变量的声明和初始化语句：与动态成员方法中的语句相同，但没有本地局部变量
- 类的静态初始化语句块：与静态成员方法中的语句相同，但没有本地局部变量

`GenE` 函数的实现建立在 JDT 提供的 Java 抽象语法树操作 API 的基础上。JDT 提供了查找本地变量、类的成员变量、父类（包括父类和接口）、类中方法等接口

支持以上操作。

3.4.2 预过滤算法的嵌入

Filter 算法需要在每次测试执行过程中，在对目标表达式求值的同时计算替换表达式的值，也即我们需要介入测试用例的执行过程。这一操作可以借助 JDT 组件提供的 IJavaBreakpointListener 接口完成。IJavaBreakpointListener 是断点监听器的一种，当在程序中加入断点，同时向调试管理器注册断点监听器时，程序将在运行过程中暂停在断点处并触发监听器中的处理函数。利用这一接口，在搜索框架的实际实现中，我们将 Filter 函数实现为扩展的断点监听器的处理函数，通过注册监听器的方式将预过滤算法嵌入搜索引擎的整体结构中。

如算法3第 6-9 行所示，在执行过滤操作前，我们初始化一个目标表达式位置上的行断点 b ，以及对应的断点监听器 h ，将二者注册到调试管理器上（第 6-7 行）。接着，filter 函数内部将以“调试”模式运行所有的测试用例，并由此触发断点监听器中的 Filter 函数（第 8 行）。最后，将断点 b 和监听器 h 从调试管理器中移除。

按照上述实现方式，预过滤算法与搜索框架主体以监听器的方式相接，这使得控制过滤算法的执行次数变得比较容易。主程序在过滤前将过滤次数初始化为 0，每当过滤算法执行一次该计数器就加一，一旦超过了过滤次数上限则过滤算法会跳过不执行。这样，预过滤技术可以无缝嵌入到主程序的搜索过程中。

3.4.3 Filter 算法实现

3.4.3.1 表达式求值 (Eval)

算法2中需使用求值函数 (Eval) 对程序执行到某处时目标表达式 e_t 和替换表达式 e_r 求值并据此判断二者是否相等。其中，求值操作的实现依赖于 JDT 提供的在运行栈上对表达式求值的开放接口。当断点监听器被触发时，Filter 算法将可以获得此处的运行栈，并调用该接口对 e_t 和 e_r 求值。

采取这种方式实现 Eval 的优点是，在运行栈上对表达式求值的计算速度很快。一般而言计算一个表达式的时间一定在 1ms 之内，因此算法可以在几秒钟内在同一个运行栈上完成 1 万个表达式的求值计算过程。与之对比，若将这些表达式全部应用于程序，生成程序变体，重新编译运行所有测试，将耗费几个小时的时间。这一时间差距是预过滤技术能够缩短系统整体运行时间的关键。但是，这种实现方式的弊端在于，Java 提供的表达式求值功能是通过“编译运行表达式”实现的，因此在某些表达式的求值过程中会产生副作用。我们曾尝试借鉴系统 CodeHint^[57] 的实现方式，通过监听内存数据修改事件还原表达式求值的副作用，但实验过程

中我们发现这一监听过程时间代价太高，根本无法应用于一般规模的程序，而由于我们对替换表达式的复杂度有一定的限制，被求值的表达式几乎没有产生副作用。因此在最后的系统实现中，我们忽略了求值副作用可能带来的问题。

3.4.3.2 等价关系判断（Equal）

Filter 算法依据 Eval 方法对目标表达式 e_t 和替换表达式 e_r 的求值结果判断 e_r 是否满足要求。Eval 的求值结果是两个 Value 对象，判断二者相等的标准如下：

- 基本数据类型变量，包括 bool, byte, short, int, long, char, float, double，比较二者的数值，数值相等则相等
- 对于其他类实例，数据类型不相等则不相等，否则：
 - 都为空则相等
 - String 和 StringBuffer 类型变量，比较其 `toString()` 后的结果是否相等
 - 其他类型变量，递归的比较其基本类型的成员变量取值是否相等，可控制递归深度
 - 数组类型，比较数组长度以及抽查其中某些位置的元素取值是否相等

比较两个 Value 是否相等的最终目的在于判断 e_r 是否能够替换 e_t ，使得已通过的测试仍能通过，并使未通过的测试执行过程有所改变。但是想直接达到这一目的比较困难，因此我们依照经验制定了上述判断标准。显然，这一标准是不完备的，某些被这一标准判定为“应过滤掉”的替换表达式事实上可能能够替换目标表达式成为正确的修复方案。更严格的定义方式或许应结合静态分析技术（如符号执行等）分析替换表达式和目标表达式的语义，即对程序运行状态的影响是否相同。然而目前搜索空间的规模和搜索算法无法支撑这些耗时的技术，且实验过程中我们发现这一比较标准应用效果良好，因此最终系统采用了这一标准。

3.4.3.3 搜索空间的管理（Remove 和 Update）

Filter 算法在计算过程中根据目标表达式 e_t 和替换表达式 e_r 的求值结果决定 e_r 在搜索空间 E_{in} 中的去留。在系统实现过程中，我们发现可以通过将每次求值都相同的 e_r 划为等价类进一步压缩搜索空间。具体而言，在算法2的 2-10 行，我们仅保留了与所有已通过的测试用例每次对 e_t 求值均相等的 e_r （`Remove` 方法），那么在之后的“检验”步骤中使用这部分 e_r 生成的程序变体也应通过这些已通过的测试用例。但在算法2的 12-20 行，我们保留了所有与未通过的测试用例对 e_t 求值时至少有一次不同的 e_r ，因此使用这些 e_r 生成的程序变体在“检验”时的运行轨迹将受到 e_r 的不同影响，那么这些程序变体均需要被检验器检验一次，而这是很

耗时的。在文章^[57]中，试验结果表明如果将生成的表达式按照求值结果划分等价类，则搜索空间将被压缩 90% 左右。受此启发，在实际的算法实现中，算法2的 16 行 `Update` 方法会根据 e_r 的求值结果将目前已经通过过滤的集合 E_{out} 进一步划分等价类。因此， E_{out} 是一个等价类的集合，每个等价类都包含多个多次求值相等的替换表达式。最终检验时，同一等价类中的替换表达式对应的程序变体将以其中一个为代表元素被检验。

3.4.3.4 过滤次数上限 (N)

过滤次数上限 N 决定了每个表达式将通过过滤的最大次数。在^[49]中，SPR 尝试生成的布尔值序列长度为 10，这意味着 SPR 将待生成的 If 条件表达式的取值范围划分为 1024 个等价类，并最终选择了其中的一个等价类作为标准。结合上小节中提到的搜索空间的压缩管理方法，SPR 算法中的布尔值序列长度与算法2中的过滤次数上限 N 含义是相同的。实验中我们发现，将 N 定为 10 次时系统效率较低，且绝大多数情况下搜索空间的等价类划分在 5 次以内就可以稳定，最终我们将 N 定为 8 次。

3.4.3.5 修复模板系统

搜索引擎内置的修复模板系统界定了搜索引擎的搜索空间。我们借鉴了^[43]中针对 Java 程序总结出的常见修复模板，同时借鉴了^[49]中的搜索空间定义，定制了本文的修复模板系统。如表3.1所示，修复模板系统共包含 9 条修复模板。模板可分为三类，分别是表达式相关修改（一般表达式替换）、分支结构相关修改（引入新分支、分支语句条件修改、引入数组和集合边界检查、空指针检查、类型转换检查）和方法相关修改（重载和重写方法替换）。表格第二列为每条修复模板定义了代号，方便后文讨论。

修复模板能够指导搜索引擎生成修复建议，但要生成具体的修复建议则需要填补模板中的空白，而填补空白的具体实现决定了系统搜索空间的最终范围。例如，对于模板 ER，生成哪些同类型的表达式决定了根据 ER 生成的程序变体数量。除此之外，依照不同修复模板生成出的修复代码形式各不相同，例如使用 IB，AB，CB，NP，TC 这几个模板生成出的修复代码是一个新的 If 语句，此时预过滤算法无法直接应用。因此，在表格第 4-6 列我们给出了每条修复模板的详细搜索空间及使用预过滤算法的转换方式。其中，ER 可直接应用预过滤算法。IC 可将被修改的 If 条件看做目标表达式，将替换后的条件看做替换表达式。对于其他几个引入 If 语句的模板，由于原程序中没有这一语句，可以将使修改无效的条件值当做目

标表达式，将引入的 If 语句条件看做替换表达式。对于两个方法替换相关的修复模板，为减少引入副作用的可能，我们没有应用预过滤算法。

表 3.1 修复模板系统

修复模板	代号	含义	搜索空间说明	e_t	e_r
一般表达式替换	ER	将表达式 e_t 替换为同类型表达式 e_r	e_t 是任一变量, e_r 是此处能够访问到的合法变量组成的深度不超过 1 的表达式	e_t	e_r
引入新分支	IB	插入 结构 $\text{if } (e_b) \text{ return/break/continue;}$	e_b 是此处能访问到的合法变量组成的深度不超过 2 的布尔表达式	e_b	e_b
分支语句条件修改	IC	对 If 条件表达式 e_b , 将其按照一般表达式替换, 此外引入替换 $e_b \rightarrow e_b e_t, e_b \rightarrow e_b \&& e_t, e_b \rightarrow !e_b$	e_t 是此处能访问到的合法变量组成的深度不超过 2 的布尔表达式	e_b	$e_b \rightarrow e_b e_t, e_b \rightarrow e_b \&& e_t, e_b \rightarrow !e_b$
引入数组边界检查	AB	插入 $\wedge \text{if } (i >= 0 \ \&\& i <= \text{array.length}) \text{ visit(array[i]); }$	该模板只在有数组访问的语句时使用, 其中 i 是访问下标	true	$\text{true } i >= 0 \ \&\& i <= \text{array.length}$
引入集合边界检查	CB	插入 $\wedge \text{if } (i >= 0 \ \&\& i <= \text{list.size}() \ \&\& \text{collection.get}(i);)$	该模板只在使用下标访问 Collection 类型变量的语句前使用, 其中 i 是访问下标	true	$\text{true } i >= 0 \ \&\& i <= \text{list.size}()$
引入空指针检查	NP	插入 $\wedge \text{if } (\text{obj} != \text{null}) \{\text{visit(obj); }\}$	该模板只在解引用语句前使用, 其中 obj 是被访问的对象	true	$\text{true } \text{obj} != \text{null}$
引入类型转换检查	TC	插入 $\wedge \text{if } (\text{obj instanceof Type}) \{\text{Supertype obj1 = (Type) obj; }\}$	该模板在强制类型转换语句前使用, 其中 Type 应是 SuperType 的子类	true	$\text{obj instanceof Type}$
重载方法替换	OL	将 $\text{a.b(c1, c2, ..., cn)}$ 修改为 $\text{a.b'(c1, c2, ..., cn)}$	方法 b' 是 b 的重载方法	-	-
重写方法替换	OR	将 $\text{a.b(c1, c2, ..., cn)}$ 修改为 $\text{a.b(c1, c2, ..., cm)}$	重写方法中的参数最多与原方法参数有一处不同	-	-

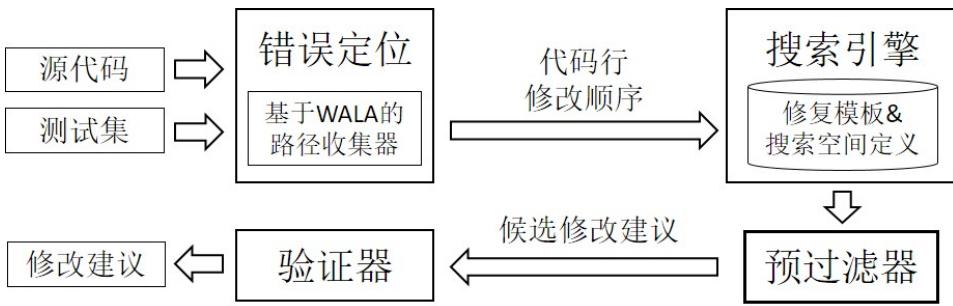


图 3.1 PfDebug 的系统结构

3.5 实验结果及分析

为检验预过滤算法对搜索空间的压缩效果，我们实现了一个完整的“生成-检验”系统 PFDebug 并将其在 Java 程序集 Defects4J^[55] 上进行实验。PFDebug 系统结构如图3.1所示，它以程序的源代码和测试用例为输入，内部包括错误定位模块、搜索引擎与验证器三个部分构成，最终输出可能的修复方案。其中，错误定位模块负责记录测试集的执行轨迹，这部分实现利用了 WALA^[58] 提供的 Java 字节码操作 API。其余部分均实现在 Eclipse JDT 上。

Defects4J 是“生成-检验”系统常用的程序测试集，它包含 5 个实际程序的 357 个版本，每个版本都包括完整的源代码和测试集，且至少有一个可以被检测到的错误。表3.2展示了测试集的基本信息。Defects4J 中的程序版本都是实际开发过程中提交的版本，其中的错误修改方式也比较复杂，现有的“生成-检验”系统能够覆盖的范围非常有限。PFDebug 的搜索空间能够覆盖其中 31 个版本，但在系统实现过程中在 Java 语言处理上存在一些细节问题，（例如 Java 内部类的字节码与源代码对应问题，循环太多导致插桩后程序运行速度过慢等），系统最终能够修复的版本数为 24 个。这是目前为止在此测试集上报告的最高数目。

表3.3展示了成功修复的 24 个版本程序在修复过程中与预过滤效果相关的运行数据：

表 3.2 Defects4J 测试集基本信息

程序名	代号	版本数	成功数	代码行数 (千行)	测试代码行数 (千行)	测试数	开发时长 (年)
JFreeChart	FC	26	4	96	50	2,205	7
Closure Compiler	CC	133	11	90	83	7,927	5
Commons Math	MA	106	6	85	19	3,602	11
Joda-Time	JT	27	0	28	53	4,130	11
Commons Lang	LA	65	3	22	6	2,245	12
总计		357	24	321	211	20,109	-

表 3.3 预过滤对搜索空间的压缩效果

版本号	代号	过滤前			过滤后			总计			压缩比率			
		EXPR	BOOL	CE	SUM	EXPR	BOOL	ES	BS	OL/OR	TOTAL	VAL	FULL	EXPR
CC1	IB	6362	256275	107674	154963	860	13999	291	963	624	15483	1878	0.01207	0.00809
CC10	OR	19223	385645	155174	249694	1778	4937	118	695	2783	9498	3596	0.01424	0.00326
CC118	IB	6229	468011	176216	298024	491	7637	76	248	1311	9439	1635	0.00546	0.00109
CC125	IC	5177	347358	139623	212912	141	1335	5	108	417	1893	530	0.00248	0.00053
CC51	IC	791	4500	3479	1812	0	29	0	4	54	83	58	0.03108	0.00221
CC62	IC	4	164	107	61	1	7	1	2	1	9	4	0.06452	0.04918
CC63	IC	90	560	323	327	1	7	1	2	77	85	80	0.19802	0.00917
CC73	IC	18	4486	3452	1052	0	18	0	4	7	25	11	0.01039	0.0038
CC86	ER	556	1127	814	869	95	166	0	17	160	421	177	0.17201	0.01956
CC92	OL	1479	23151	7330	17300	1051	2830	617	654	279	4160	1550	0.08817	0.07347
CC93	OL	1479	23194	7198	17475	1051	2767	617	642	279	4097	1538	0.08663	0.07205
LA24	ER	161	1253	464	950	30	261	7	111	6	297	124	0.12971	0.12421
LA26	OL	16	410	113	313	8	89	8	7	43	140	58	0.16292	0.04792
LA6	ER	240	385	250	375	68	51	44	21	57	176	122	0.28241	0.17333
MA75	OR	3	3	5	1	1	0	1	0	35	36	36	1	1
MA80	ER	752	522	241	1033	519	48	283	8	100	667	391	0.3451	0.2817
MA82	IC	295	1215	806	704	5	42	3	12	80	127	95	0.12117	0.02131
MA85	IC	157	238	165	230	0	10	0	3	39	49	42	0.15613	0.01304
MA33	ER	790	1707	970	1527	3	25	3	10	56	84	69	0.04359	0.00851
MA5	ER	12	68	22	58	4	24	4	4	0	28	8	0.13793	0.13793
FC1	IC	1222	7898	1488	7632	873	2526	816	125	193	3592	1134	0.14492	0.1233
FC11	ER	56	465	265	256	0	2	0	1	8	10	9	0.03409	0.00391
FC24	ER	56	9	11	54	34	0	14	0	26	60	40	0.5	0.25926
FC9	IC	25	107	71	61	0	1	0	1	3	4	4	0.0625	0.01639

表格第一列是所有成功修复的程序版本编号，对每一个程序版本，我们统计了成功修改改程序所使用的的修复模板（“代号”），在预过滤前系统产生的修复方案数（“过滤前”四列），在过滤后仍被保留的修复方案数（“过滤后”四列），与方法修改相关不参与过滤处理的修复方案数（“OL/OR”），最终需要被检验器检验的修复方案总数（“总计”两列）和搜索空间被压缩的压缩比率（“压缩比率”）。其中，“过滤前”的数据包括四组，分别是：与一般表达式替换相关，即使用模板 ER 生成的修复方案数（EXPR）；与布尔表达式相关，即使用 IB, IC, AB, CB, NP, TC 模板生成的修复方案数；编译不通过被首先剔除的修复方案数（CE^①）；与一般表达式和布尔表达式相关的合法修复方案总数（SUM）。“过滤后”也包括四组，分别是：ER 产生的修复方案过滤后的剩余数量（EXPR）；布尔表达式相关的修复方案过滤后剩余数量（BOOL）；ER 产生的修复方案划分的等价类数量（ES）；布尔表达式相关的修复方案等价类划分数量（BS）。在“总计”这两列中，TOTAL 表示过滤后的修复方案总数（EXPR+BOOL），VAL 表示需验证的等价类数量（ES+BS）。最后“压缩比率”这两列中，FULL 表示把方法相关的修复方案考虑在内，修复方案数在过滤后与过滤前的比值，EXPR 表示仅考虑一般表达式和布尔表达式相关的修复方案在过滤后与过滤前的比值。

从最终的压缩比率看，由于每个程序版本修复过程中生成的方法修复数都比较少，因此无论是否将其考虑进来（FULL/EXPR），过滤后需要被检验器验证的修复方案数远小于过滤前。在比较复杂的 Closure Compiler 程序上，搜索空间巨大，此时预过滤的效果体现的尤其明显，过滤后检验器的工作负担通常仅有过滤前的不超过 5%，在版本号 1,10,118,125,51 上，这一比率甚至低于 1%。对于较简单的程序，过滤效果虽然没有在 Closure Compiler 上明显，但这一比率也在 10%-20% 左右。值得一提的是，布尔表达式相关的修复方案数确实占了修复方案总数的绝大部分，这是由于生成一个布尔表达式比较容易，但生成一个同类型的一般表达式则相对困难。在压缩效果上，布尔表达式的压缩比率更小一些，这可能也是 SPR 专注于 If 条件搜索优化的一个原因。但是，一般表达式的修复方案也占了一定的比例，而且其数目远远超过最终过滤后的修复方案总数。例如 Closure Compiler 10 对应的一般表达式修复方案有 19223 个，如果不对这部分修复方案压缩，那么检验器的工作负担最少是 19223，远超过过滤后的 3596 个。这也说明，在研究搜索优化技术时将一般表达式替换考虑进来是很有必要的。

表格3.4 汇总了在 Defects4J 测试集上的其他工作的实验结果实验结果。参与比较的系统包括本文实现的 PFDebug（PF），Nopol、针对 Java 版本重新实现的

^① 由于系统实现时精确的控制表达式的生成过程使得仅生成编译通过的表达式比较困难，因此我们把一部分工作交给了编译器完成，因此会导致 CE 这一列的出现

表 3.4 修复成功率对比

版本	PF	Nopol	jGP	jKali	SPR	版本	PF	Nopol	jGP	jKali	SPR
CC1	•				•	LA58		•			
CC10	•					MA5	•		•		
CC51	•				•	MA33	•				
CC62	•					MA50		•	•	•	
CC63	•					MA53			•		•
CC73	•					MA70			•		
CC86	•					MA73			•		
CC92	•					MA75	•				
CC93	•					MA80	•				
CC118	•				•	MA82	•				•
CC125	•				•	MA85	•				•
LA6	•				•	FC1	•				•
LA24	•					FC5		•			
LA26	•					FC9	•				•
LA44		•				FC11	•				
LA55		•			•	FC24	•				

GenProg (jGP) 和 Kali (jKali) (实验结果来自^[59])。另外我们从搜索空间是否覆盖正确修复建议的角度比较了若实现一个 java 版本的 SPR 其可能的实验结果。

表格以 • 表示某系统能够正确修复某版本的程序。defects4J 中共有 357 个版本, PFDebug 总共可以修复 24 个版本。在^[59]的实验中作者只测试了不包括 Closure Compiler 的其他 4 个程序共 227 个版本, 因此 Nopol, jGenProg 和 jKali 在 Closure Compiler 上没有实验结果。PF 在所有 Closure Compiler 程序上可以修复 11 个程序版本。在其余的程序版本上, PFDebug 可以成功修复共 13 个程序版本, 而 Nopol、jGenProg 和 jKali 最多只能修复 5 个程序。一个可能的解释是这三个系统没有使用有效的搜索空间压缩手段, 所以无法覆盖较大的搜索空间, 修复正确率也比较低。

此外, 我们分析 SPR 的搜索空间, 预测它在其余几个系统能够成功修复的程序上的修复效果。不考虑具体实现, SPR 可以修复其中的 11 个程序, 主要原因是 SPR 的搜索空间只包含了简单语句的一般表达式替换, 没有包含方法替换。Java 语言中简单表达式较少, 方法的重写和重载也比较常见, 因此直接将 SPR 的搜索空间平移到 Java 语言程序中会有一定的问题。但 SPR 的搜索空间覆盖范围内能够产生的正确程序修复方案也超过了其他三个系统, 这也是 SPR 优化搜索算法优化效果体现。

3.6 预过滤算法的局限性

实验结果显示，预过滤算法对搜索空间的压缩效果非常显著，但是在算法实现中我们用到了大量基于经验假设的设计，而非严格的理论推导。例如，在已通过的测试用例执行过程中，替换表达式的值与目标表达式不同不一定会导致已通过的测试用例运行失败。此外，两个表达式取值等价性的判断标准对类对象实例也是比较粗糙的。这两点导致算法有时会将合理的替换表达式滤除掉，系统最终错过了正确的修复建议。

Listing 3.2 错误示例 commons-lang59/StrBuffer.java

```

1 //----- source code-----
2 public StrBuilder appendFixedWidthPadRight(Object obj, int width,
3     char padChar) {
4     if (width > 0) {
5         ensureCapacity(size + width);
6         String str = (obj == null ? getNullText() : obj.toString());
7         int strLen = str.length();
8         if (strLen >= width) {
9             // 此处 strLen -> width
10            str.getChars(0, strLen, buffer, size);
11        } else {
12            int padLen = width - strLen;
13            str.getChars(0, strLen, buffer, size);
14            for (int i = 0; i < padLen; i++) {
15                buffer[size + strLen + i] = padChar;
16            }
17        }
18        size += width;
19    }
20    return this;
21 }
22
23 //----- test code-----
24 public void testAppendFixedWidthPadRight_int() {
25     StrBuilder sb = new StrBuilder();
26     sb.appendFixedWidthPadRight(123, -1, '-');
27     assertEquals("", sb.toString());
28 }
29
30
31 // See: http://issues.apache.org/jira/browse/LANG-299
32 public void testLang299() {
33     StrBuilder sb = new StrBuilder(1);
34     sb.appendFixedWidthPadRight("foo", 1, '-');
35     assertEquals("f", sb.toString());
36 }
```

例如，在代码3.2中，被测对象是 `StrBuffer` 类中的方法 `appendFixedWidthPadRight`，方法的功能是在当前 `StrBuffer` 对象的字符缓冲区右端附加对象 `obj` 转换的字符串，要求附加的长度为 `width`，不足的部分用 `padChar` 补齐。在代码第9行，调用方法 `getChars` 时第二个参数应当是 `width`，表示截取的是前面 `width` 个字符，但误写成了 `strLen`，这导致虽然 `StrBuffer` 对象表示 `buffer` 长度的 `size` 属性是正确的，但是 `buffer` 数组在超过 `size` 的位置上仍有字符。这个错误在上一版本程序中并没有暴露出来，原因是测试代码只包含了类似 `testAppendFixedWidthPadRight_int()` 的测试，由于 `StrBuffer` 类重写了 `toString()` 方法，虽然 `buffer` 数组在超过 `buffer` 长度的位置仍有字符，但 `assertEquals` 只会对 `StrBuffer` 中不超过 `buffer` 长度的字符做判断，于是测试会通过。在下一个版本中，一个专门的测试用例 `testLang299` 被加入了测试集，这一问题才被暴露。

当“生成-检验”系统使用预过滤策略时，搜索空间中存在将 `strLen` 替换为 `width` 的修改方案，然而显然很多情况下 `strLen` 和 `width` 的值并不相等，于是这一方案被过滤掉了。在这一版本程序中我们没有发现更好的等价修复方案，因此最终 PFDebug 无法成功的修复该版本程序。

在现有预过滤算法的框架下，这一问题可能比较难解决。一种可能的解决思路是引入静态分析技术（如符号执行等），分析将 `strLen` 替换成 `width` 之后对测试用例结果的影响。事实上 SemFix 等算法的思路也是如此。但是按照现在的技术发展情况，按这一思路实现的系统计算量较大，速度较慢，可能无法处理 Closure Compiler 类似规模的程序。

3.7 本章小结

“生成-检验”系统的核心模块是搜索引擎。如何定义搜索空间的范围、设计搜索算法，平衡搜索速度与范围之间的关系是搜索引擎设计的关键问题。在本章中，我们分析了现有工作的搜索空间和搜索算法设计方案，借鉴了 SPR 等工具将搜索模板系统与搜索空间压缩技术相结合的设计思路，提出了“预过滤”搜索算法。算法利用“已通过的测试用例仍应通过，未通过的测试用例执行过程应有变化”这一观察，对搜索模板系统中与表达式修改相关的搜索模板所生成的修复方案进行过滤处理。此外，算法的具体实现利用表达式值的等价关系将搜索空间以等价类结构组织起来，进一步缩减检验模块的工作量。

基于预过滤算法，我们实现了“生成-检验系统” PFDebug。在 Defects4J 上的测试结果表明，检验模块的工作量减少到了过滤前的 10%-20% 左右，对于规模较

大搜索空间复杂的程序（如 Closure Compiler）等，搜索空间被压缩到了原来的 1% 以下。与现有的其他工作相比，PFDebug 能够成功修复其中 24 个程序，修复正确率超过 Nopol, jGenProg, jKali 等现有系统。

第4章 框架扩展

4.1 引言

基于“生成-检验”框架的程序自动修复系统以源代码和其对应的测试用例为输入，输出一组能够使测试集中所有测试用例通过的程序变体供开发人员参考。从设计的目标来看，“生成-检验”系统希望能够接手开发人员的调试工作，提高软件开发效率。然而从实验数据来看，现有系统在修复规模稍大的程序时速度仍然比较慢。例如，SPR 在实验对象程序 php 上常常需要几个小时才能完成修复，我们实现的系统 PFDebug 在修复 Closure Compiler 上的错误时也要花费几个小时。另一方面，由于系统中搜索引擎能够在有限时间内搜索完的搜索空间有限，现有系统一般只能在单一位置生成如表达式替换、方法替换等比较简单的修改方案，导致现有系统的修复正确率也不太高，例如 SPR 在 GenProg 的测试集上修复了 39/69 个错误，PFDebug 修复了 24/357 个错误。由于速度慢、正确率低，基于“生成-检验”框架的自动修复系统与实际应用仍有一定的距离。

从使用者的角度，遵循“生成-检验”框架开发的系统在计算过程中间不需要开发人员的参与，也不需要了解程序错误相关的更多信息。这一特点使得系统使用非常方便——只要启动程序，等待结果就可以了。但是，考虑到现有自动调试技术的发展水平，这种模式未必是“生成-检验”系统的最佳利用方式。事实上，如果能够让开发人员与系统有一定的互动，充分利用开发人员的调试经验、对调试任务的理解、错误类型的初步判断等信息，系统的修复速度可能得到进一步的提高，搜索空间也可以更大，正确率也随之提高。

基于上述想法，本章提出扩展“生成-检验”框架，使系统能够与开发人员充分共享信息，优化修复效果。扩展方式有两种，第一种是“交互式调试”，基本思想是，利用开发人员对程序的理解，向开发人员提供接口描述他对程序运行状态的判断，使系统能够将错误限定于较窄的范围内，提高错误定位的准确度。此外，系统允许开发人员将调试任务分为几个小任务逐个解决，这使得系统可以处理需要在多个位置修改才能完成的调试任务。第二种是针对单类别错误修复的可扩展框架，即规范“生成-检验”框架的基本结构，使得错误定位方法易于替换，搜索空间易于剪裁，方便在此系统上进行二次开发，形成针对特定错误类型的修复系统。

为验证框架扩展的有效性，本章首先在已有框架上实现了“交互式调试”使用模式，形成了新的系统 SmartDebug，并以多个实际程序为调试对象比较使用 SmartDebug 和人工调试的调试效率。实验表明，使用 SmartDebug 确实加速了调试

任务完成过程。此外，本章整理了已有系统的框架结构，设计了方便扩展的开放接口，并在此框架上实现了针对空指针异常（`NullPointerException`）的修复系统。我们在 `CWE_NullPointerDereference` 测试集上进行试验，成功修复了测试集中 6 个子类错误中的 5 个子类。

本章的主要贡献如下：

- 提出“交互式调试”扩展方式并实现，形成新的系统 `SmartDebug`。在一组真实程序上进行实验，结果表明 `SmartDebug` 能够加速开发人员完成调试任务的过程
- 提出将现有系统扩展为针对特定类别错误修复系统的扩展方式，并给出开放接口定义，将现有系统重构为可扩展框架 `xDebug`
- 在 `xDebug` 框架内实现针对 Java 空指针异常的修复系统 `NPEDebug`，并在 `CWE_NullPointerDereference` 测试集上完成实验，成功修复其中 6 个子类错误中的 5 个子类，证实 `xDebug` 的可用性

4.2 交互式调试

4.2.1 概述

“交互式调试”是指在利用自动修复系统完成调试任务的过程中，开发人员可以通过特定方式向系统提供信息，影响系统计算过程的一种设计模式。现有的“生成-检验”框架不依赖开发人员对程序的先验判断，这使得系统虽然使用简单，但也由于没有人工指导常常花费许多时间生成和检验无效的修复方案。在本节我们给出“生成-检验”框架的一种扩展方式，使得开发人员可以方便地输入他对程序运行状态的期望及是否正确的判断，而系统能够根据这些信息缩减搜索空间、分解调试任务、提高运行效率。

具体而言，我们首先在“生成-检验”框架中引入“检查点”概念。“检查点”在程序执行过程中某一具体位置上的一个标记点。在工具使用者的角度，开发人员可以自定义检查点，同时附上对该位置上程序运行状态的判断，包括“程序是否运行正确”，“如不正确，运行状态应当是什么样子”。而对于系统来说，由于检查点将程序的执行过程分成了多个段落，利用开发人员的判断，系统可以将程序错误位置缩小到其中的某一段或某几段上，将调试任务分解，降低难度。

实现“交互式调试”需解决两方面的问题，一是开发人员应如何输入检查点，而是系统应如何利用检查点上的信息优化搜索过程。本节我们将从这两方面阐述交互式调试框架扩展方式。

4.2.2 相关工作

将用户交互与调试工具相结合的工作可分为两大类。第一类是以辅助工具本身的功能为中心，引入开发人员的输入，使得辅助工具的性能更佳。例如，在文章^[60]中，作者将开发人员引入 SFL 算法错误定位的过程。作者将代码行的可疑程度分为“安全”，“敏感”和“危险”三个等级。这时，开发人员应人工检查标为“危险”的代码行，去除其中的错误。在此基础上，开发人员还应选择覆盖了“危险”和“敏感”代码行的测试用例，这些测试用例的执行过程和测试结果将作为下一轮 SFL 错误定位的输入。由于 SFL 算法在程序错误有多处时定位精度常常不稳定，将开发人员的判断引入错误代码行在迭代过程中逐步浮现，多处错误可逐个修改。类似的，文章^[61]要求开发人员对 SFL 算法给出的可疑代码行给出判断，标出正确的代码行，将其从可疑列表中剔除出去。这一反馈过程会使 SFL 的定位结果越来越精确。

另一个典型的研究工作是 Microbat^[62] 工具。该工具集成在 Eclipse IDE 中，其主要目的是引导开发人员的调试过程。对于一个出错的程序，Microbat 记录开发人员使用跟踪调试时程序的执行轨迹，从程序运行状态和上下文中分析出跟踪的哪些步骤可能出错并提示给开发人员。开发人员则分析提示给出的调试步骤，反馈给工具这一步骤是否真的有错，工具则据此调整输出，给出新的提示。

第二类是针对特殊类型的程序为开发人员提供方便的调试辅助设施。例如，针对并发程序的“记录-重放”工具 LEAP^[63], ODR^[64], ORDER^[65], PRES^[66], bbr^[67] 等能够记录多线程程序的具体执行路径并稳定回放，这使得开发人员可以在一个确定的执行路径上完成程序调试，而避免了多线程程序运行过程的随机性。工具 Sherlog^[68] 能够通过分析已部署程序的运行日志还原程序的执行过程。在还原过程中，当遇到不确定因素时，SherLog 会自动产生日志记录语句，从而确认程序的真实还原路径，最终方便开发人员定位错误。此外，BigDebug^{[69][70]} 为开发人员提供程序执行过程的记录、重放和可视化功能，使得开发人员在使用已有的断点调试、日志分析技术基础上更高效的修复程序错误。

在辅助调试类工具外，代码片段提示工具 CodeHint^[57] 也利用了用户输入信息逐步精化工具的输出结果。CodeHint 的功能是根据用户的要求自动生成表达式。用户描述需求的方式有三种，一是限定表达式的语法类型，二是直接给定表达式的值，三是通过 CodeHint 定义的 pdspec 语法描述表达式应满足的一条性质。CodeHint 在一定的搜索空间中查找合法的表达式，并根据用户需求过滤结果。在使用过程中，用户可以逐步精化对表达式的要求，CodeHint 也会不断扩大搜索空间生成新的结果。这一交互过程使得 CodeHint 能够不断将搜索方向靠近用户的期望，优化搜索

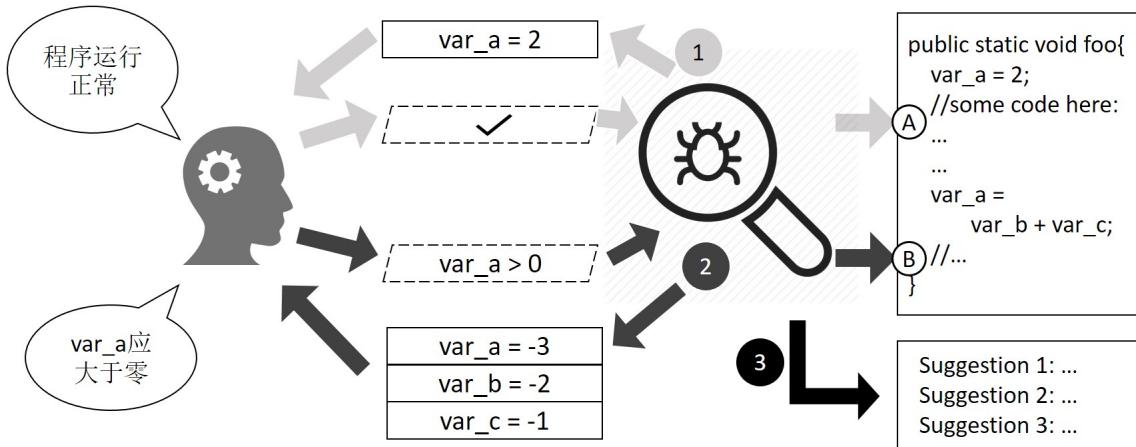


图 4.1 交互模式示意图

结果。

现有的研究工作尚未在“生成-检验”框架中引入用户交互。本文在此方向进行探索性的研究，也希望对其他研究人员带来一定的启发。

4.2.3 交互模式

将用户交互引入“生成-检验”系统，首先需要设计用户与系统的具体交互模式。图4.1以一段示意代码展示了开发人员与系统合作完成调试任务的主要步骤。

在示意代码中，函数 `foo` 内部包含了几个变量 `var_a`, `var_b`, `var_c`。在一般的调试过程中，开发人员将在 A 点和 B 点各加一个断点。启动调试器（例如 JDT Debugger）后，程序将首先暂停在 A 点，开发人员观察当前栈上的变量值，发现此时程序运行正常，因此恢复调试进程，程序继续运行。接着程序暂停在 B 点，此时开发人员发现变量 `var_a` 的值为 -3，然而此时 `var_a` 本应是正值，这说明在 A 点和 B 点之间程序运行出错，于是开发人员需要在两个断点之间再加入新的断点，逐步查找程序中的错误。

在上述调试过程中，开发人员在 A 点和 B 点对程序的运行状态进行判断，使得错误调试的范围缩小在两点之间。为利用这一判断，我们在现有“生成-检验”框架下设计了如下的交互模式：

1. 开发人员根据对程序的初始判断，在认为与错误相关的地方加入断点
2. 开发人员启动调试器，并判断断点处程序的运行状态是否正确。如果正确则标识“通过”（图4.1中路线①），否则输入对程序运行状态的期望描述，例如“`var_a > 0`”（图4.1中路线②）。系统将记录这一判断，生成带标注的“检查点”
3. 开发人员选择一个状态不正确的检查点作为调试目标，系统搜索可能的修复

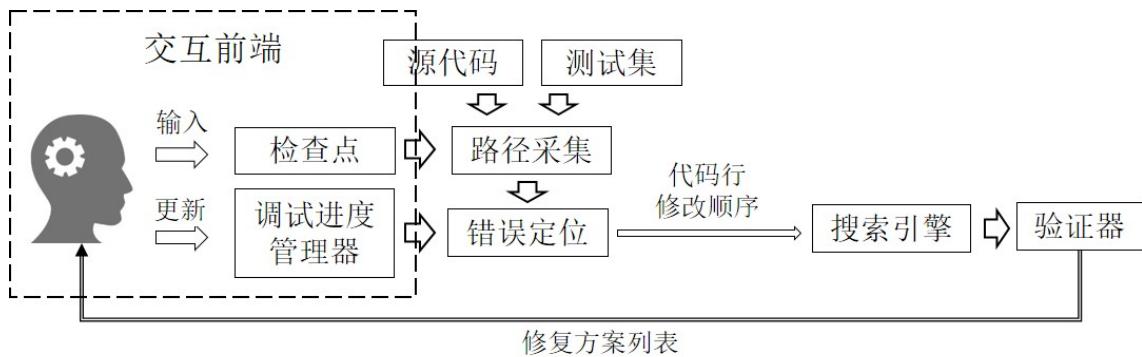


图 4.2 系统结构示意图

方案，返回给开发人员（图4.1中路线③）

4. 如果系统在一定时间内没有返回合理的修复方案，则开发人员可增加更多的检查点，返回第2步

上述交互模式综合了开发人员和系统在调试任务上各自的优势。一方面，开发人员对调试任务的先验知识可对“生成-检验”系统的错误定位结果起到补充作用。当程序错误比较复杂需要修改多处代码时，通过设置检查点可以将程序执行过程分解为小段，每小段中包含一处需要修改的代码，这样修复系统能够处理。另一方面，由于使用了“生成-检验”系统的修复方案自动生成功能，开发人员可以减少调试过程中“重启调试→单步跟踪”的次数。例如在图4.1中，当发现B点程序运行出错时，不需要重启程序并退回到A点开始单步跟踪。

4.2.4 系统结构

图4.2展示了框架扩展后的系统结构。如图所示，在一般的“生成-检验”模块前，我们加入了一个“交互前端”，方便开发人员与后端的修复生成系统交互。交互前端内部包含两个模块。一是检查点及其相关管理模块，负责接收用户对程序运行状态的判断。检查点具有以下属性：

- 关联测试用例。被测程序的测试集通常包含多个测试用例，每个检查点最多关联一个测试用例
- 位置信息。即在某个测试用例执行过程中的一个特定位置，以代码行号和经过这一行的次数表示
- 执行状态信息，包括开发人员对程序运行到此处的状态的判断，“正确”或“不正确”，如不正确，还包含以一个合法 Java 布尔表达式描述的对程序运行状态期望。特殊的情况是，如果需要表示程序应该在每次经过该检查点所在的代码位置时都满足所输入的布尔表达式，那么应在位置信息的“次数”这一属性上填入“一直满足”

二是调试进度管理器，负责实时更新当前所有检查点的运行状态。在程序调试过程中，可能出现多个检查点运行状态都不正确的情况，此时用户可以在调试进度管理器中每次选择一个检查点作为调试目标，逐步修改程序以完成调试任务。

开发人员通过交互前端输入其对程序的先验知识，系统则根据这些信息调整搜索过程。原有框架中有两个模块的设计和实现需要调整，一是错误定位模块，其定位结果应依照检查点的状态有所改变，而是检验器，其检验标准应随调试目标的变化而变化。在以下两个小节中我们重点介绍这两个模块的调整方案。

4.2.5 扩展的错误定位

在原框架下，错误定位模块的输入是由路径采集模块记录的测试用例执行路径。定位算法采用 SFL 定位算法中的 Ochiai^[72] 算子，对每个代码行 l ，其出错的概率估计 sus_l 为：

$$sus_l = \frac{a_{ef}}{\sqrt{(a_{nf} + a_{ef})(a_{ef} + a_{ep})}}$$

其中， a_{ef} 表示经过该代码行且执行失败的测试用例数， a_{ep} 表示经过该代码行且通过的测试用例数， a_{nf} 表示未经过该代码行且通过的测试用例数。

在扩展框架下，定位算法应利用检查点的属性生成更精确的定位结果。在引入检查点后，一个未通过的测试用例对应的路径可能被检查点分成了很多段。精确地说，设一个未通过的测试用例 t_f 的测试路径为 $L = l_{1,1}, \dots, l_{1,m}, cp_p, l_{u,1}, \dots, l_{u,v}, cp_f$ ，其中 cp_f 是第一个未通过的检查点， cp_p 是 cp_f 前的最后一个检查点，那么我们知道在 $L_1 = l_{1,1}, \dots, l_{1,m}$ 这段路径上程序运行正确，而在 $L_2 = l_{u,1}, \dots, l_{u,v}$ 这一段上必然有至少一处代码是错误的。因此，在统计 SFL 算子的输入时，路径 L 会被拆分为两条路径 L_1 和 L_2 ，而错误定位模块最终输出的调试顺序列表也仅包含 L_2 中的代码行。由此，开发人员借助检查点圈定了错误存在的范围，系统在后续搜索过程中也可以节省时间。

4.2.6 调试进度控制

当被测程序错误比较复杂时，可能出现多个测试都未通过且程序需要修正的位置不止一处的情况。此时，开发人员可以为每个不通过的测试用例都加入一组检查点，逐一程序在检查点上的运行状态，从而将调试任务分解为几个子任务逐步解决。在解决问题的过程中，不同测试用例的检查点状态也不相同。系统为检查点规定了三种状态，“通过”，“不通过”和“未知”。其中，“通过”表示检查点

所属的测试用例在这一位置上运行正确或在修复了某些错误后检查点上的“用户期望”表达式得到满足。相反的，“不通过”表示测试用例运行不正确，“用户期望”表达式未被满足。“未知”表示未对这一检查点做判断。

图4.3展示了系统的调试进度控制面板，面板上列出了当前测试用例及其检查点的状态。每次对修改程序后，刷新控制面板，系统将自动执行所有测试用例，并判断检查点的状态，更新面板。这一功能实现在 JDT Debug 插件上，对每个测试用例，系统首先将每个检查点转化为 JavaLineBreakpoint 注册到 JDT 调试管理器中，同时实现一个断点监听器，在回调函数中判断检查点中的期望表达式是否被满足。接着以调试模式运行程序，在程序运行过程中，监听器会被触发，一旦某个检查点上程序没有通过，则标为“不通过”，之后未触发的检查点都标为“未知”。

在分步调试过程中，开发人员可以选择任何一个当前具有未通过检查点的测试用例作为调试对象。此时，调试的目标是使调试进程“前进一步”，即通过这个检查点。因此，后台系统的检验模块验证一个备选修复方案是否通过的标准将从“通过所有测试用例”变为，在保证其他已通过的检查点仍然通过的基础上，使被选中的测试至少多通过一个检查点。这样，程序错误将逐步修复。

4.2.7 实验结果

在 PFDebug 的基础上，我们将交互式调试引入“生成-检验”框架，并实现为一个 Eclipse 插件工具 SmartDebug。为评价 SmartDebug 的实际使用效果，我们设计了如下的对比试验。

首先，我们记录了一次研究生一年级上机编程考试过程中 25 名同学解答一道考题的编程过程。参加考试的同学均使用 Eclipse 集成开发环境，我们在后台记录了程序的编辑历史，包括每次保存操作、调试操作对应的程序版本。考试结束后，我们恢复出开发过程中出错的版本作为被测程序。最后，我们邀请了 20 余名未参加考试的同年级同学参加 SmartDebug 的评价实验。参与人员随机分为两组，一组仅通过人工方式调试被测程序，一组使用 SmartDebug 辅助调试。我们记录了两组

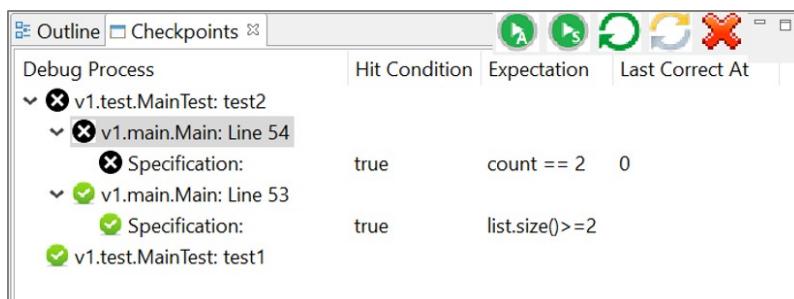


图 4.3 调试过程控制面板

表 4.1 SmartDebug v.s. Human

序号	错误简介	SD(s)	H(s)
1	variable replacement & operator change	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

同学完成调试任务所需的时间并比较两种情况下的调试时间。

表4.1列出了实验结果。SmartDebug 的搜索空间覆盖了 25 个错误版本中的 8 个，错误类型包括使用了错误的局部变量、循环变量、错误的操作符等。在这 8 个程序上，使用 SmartDebug 一般可以在 5 分钟以内完成程序修复。在 7 个版本上，借助 SmartDebug 的修复效率高于人工修复。其中值得注意的是，1 号程序有两个错误，SmartDebug 可以通过分段调试的方式分别解决两个问题。

以上实验结果表明，引入交互式调试模式的“生成-检验”系统能够对开发人员的调试工作起到一定的帮助作用。

4.3 针对单类别错误的可扩展框架

4.3.1 概述

基于“生成-检验”框架的自动修复系统本身并不针对特定类型的错误修复程序，其能够修复的错误范围是由系统内部搜索引擎的搜索空间决定的。这种设计的优点是系统在理论上具有普适性，例如当程序中的错误来自软件的业务逻辑层而非底层实现时，系统仍有可能成功修复程序。但是，如上文中所分析的，目前“生成-检验”系统的搜索引擎能够负担的搜索空间比较小，很多实际程序中的错误都无法覆盖，这导致这一类系统投入实用还比较困难。

实现错误自动修复的另一条技术路线是针对特定类型的错误研究专门的修复方案。这一类工作通常针对较底层的错误，如空指针异常、数组越界、死锁、数据竞争等。这类错误在程序执行过程中表征明显，与软件本身的业务逻辑关系不大，分析算法也比较成熟，因此针对特定类型的错误修复算法容易做到准确、高效。

事实上，以上两种技术路线具有一定的互补性。前者不限制错误类型，但准确度和效率较低，后者性能较好但错误类型受限。由此，本文提出扩展“生成-检

验”系统框架，使得针对特定类型错误的各类修复算法可以方便的集成到框架中，而在使用时开发人员可以根据具体情况选择所需的算法，则扩展系统可以兼具两种技术路线的优点。

为达成上述目标，框架扩展过程中需解决以下两个问题。第一，不同修复算法计算步骤各不相同，若将其统一在同一个框架中，则需在合适的粒度上规范各个算法的操作流程，使得各个算法具有规整的代码结构，但仍能有不同的功能实现。第二，为方便算法集成，应合理设计“生成-检验”系统各模块间的接口，将公用功能抽出，为其他算法提供较好的基础设施，尽量避免重复代码。

在本节中，我们从“生成-检验”框架的结构出发，分析针对特定类型错误的修复算法一般流程与“生成-检验”系统的计算流程的共同点，最终将不同修复算法以“子搜索引擎”方式集成在系统中。此外，我们将已有系统中路径收集模块、错误定位模块以及检验器模块的接口规范化，设计 API，方便不同算法实现重用已有功能。最后，我们以一个空指针异常修复系统 NPEDebug 为例，说明了如何利用已有系统完成针对特定类型错误的修复算法定制开发。我们将 NPEDebug 用于修复 CWE 空指针引用（Null Pointer Dereference）类别下的 6 类测试用例，实验结果表明 NPEDebug 可以成功修复其中 5 类错误。这一结果证明了将针对特定类型修复算法融合进“生成-检验”框架的可行性。

4.3.2 相关工作

4.3.2.1 针对特定类型错误的修复算法

空指针（**Null Pointer**）错误修复算法：^[73]提出了针对 Java 程序中的运行时异常的错误定位和修复算法，理论上其能够处理的异常类型包括空指针（NullPointerException）、算数运算（ArithmaticException）、类型错误（ArrayStoreException）。该算法将动态分析与静态后向数据流分析相结合，能够定位错误的赋值语句，同时也分析出该语句在不同运行场景下可能触发的其他异常。最终作者将该方法实现并应用于定位和修复空指针异常。^[74]中，作者参考“生成-检验”框架中的“修复模板”概念，提出了修复空指针异常的 9 条修复模板，并通过在程序的执行过程中监控变量取值找出适合使用模板的程序位置，最终生成修复方案。

数值错误：工作^[75]中，作者针对“整数溢出导致缓冲区溢出（IO2BO）”错误提出检测和修复算法，并实现工具 IntPatch。IntPatch 利用类型论和数据流分析框架检测可能的 IO2BO 错误，并能够在编译阶段在程序中插入运行时检查语句。此外，IntPatch 为程序员提供了检查整数溢出错误的接口。实验表明，IntPatch 能够成功捕捉到 IO2BO 错误，并且处理后的程序运行效率平均仅降低 1%。^[76]针对 C 程

序中的整数缺陷提出修复方案。作者提出，许多整数缺陷可以通过适当增加整数变量的精度完成修复，并基于此开发了工具 CIntFix。实验表明，CIntFix 成功修复了 Juliet 测试集中来自 7 个缺陷类别的 5414 programs，在 SPEC CINT2000 测试集上的处理速度达到了 0.157s/千行代码，修复后的程序平均有 18% 的效率损失。

数据结构错误:^[77] 针对关键数据结构应满足的规约提出了一种描述语言，并能够动态检测和修复程序中不满足规约属性的数据结构，使程序能够在数据结构受损时稳定运行。进一步地，文章^[78] 提出对数据结构规约条件的自动生成算法。算法利用不变式检测工具 Daikon 推断数据结构应满足的一致性条件，开发人员可泛化或精化这些条件以获得合理的数据结构规约。将上述工作结合，开发人员可以较容易的描述、检测和修复系统中的数据结构错误。

与获取数据结构的形式化规约不同,^[79] 提出将程序中的 Assert 语句看做程序规约并给出相应的数据结构修复算法。其基本思想是，首先利用符号执行计算当 Assert 语句被满足时受损的数据结构所应满足的约束条件，接着搜索能够使数据结构满足约束条件的修改方式。实验表明通过动态修改数据结构内容，程序能够从错误状态恢复并稳定运行。受符号执行技术限制，该算法通常可以修复规模在几百个节点的数据结构。在^[80] 中，作者引入静态分析技术提高算法的规模并开发新的工具 STARC。给定一个描述数据结构约束的 Java 谓词方法，STARC 通过静态分析识别该方法用于遍历数据结构的成员域以及目标数据结构中成员变量之间的约束关系。在程序运行过程中，STARC 执行该谓词方法识别受损的数据结构，接着在之前静态分析结果的指导下系统性的修改受损数据结构。实验表明改进的算法能够将处理的数据结构规模扩大到几万个节点。

4.3.3 框架设计

图4.4展示了扩展的“生成-检验”框架。为设计合理的扩展框架，我们首先将原有框架整理，并抽出针对一般类型错误和特定类型错误的不同修复算法可重用的功能模块，以实线框图表示，包括与用户交互的“检查点”、“调试进度管理器”，负责处理路径信息的路径收集和错误定位模块，负责“生成”的搜索引擎框架和通用修复模板系统，提供搜索空间压缩的与过滤器以及负责“检验”的验证器。接着，我们分析上小节中提到的针对特定类型错误的算法设计思路，发现无论修复何种错误，均需经过“错误定位”和“修复生成”两个步骤。因此我们将第一个步骤对应的计算模块抽象为“特定类型错误分析器”，其计算结果将生成“特定类型错误修复位置”列表。将第二个步骤抽象为“定制修复模板系统”，封装不同的“特定类型错误修复算法”。最终，当需要在系统中添加新的错误定位算法时，只

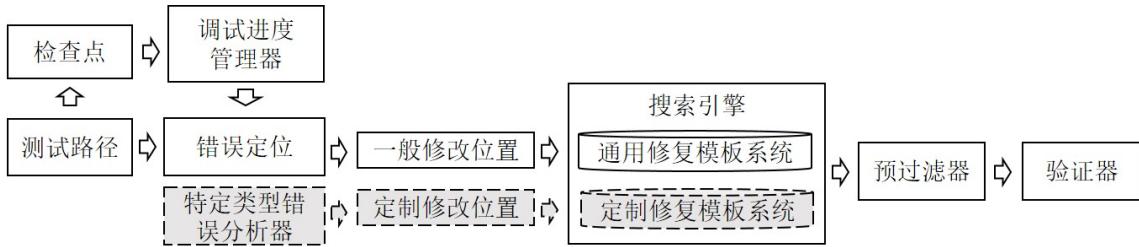


图 4.4 扩展框架设计

需实现一组图中灰色框图模块，与其他系统固有模块组合即可。

下面我们将详细介绍系统各模块所提供的的具体功能和可扩展接口。

4.3.3.1 搜索引擎

搜索引擎是扩展框架的核心部分，其功能是提供一个公用搜索流程控制接口，使得在实现针对不同类型错误的生成算法时只需关注针对某一特定位置的修复算法而不需重复编写整个流程控制代码。代码4.1展示了搜索引擎（SearchEngine）实现的代码框架：

Listing 4.1 搜索框架

```

1  public class SearchEngine extends Job {
2
3      private FixGenerator fixGenerator;
4      private FixValidator fixValidator;
5      private BugFixSession session;
6
7      public SearchEngine(BugFixSession session,
8          FixGenerator fixGenerator){...}
9
10     @Override
11     protected IStatus run(IProgressMonitor monitor) {
12         try {
13             session.initFixSession();
14             fixValidator = new FixValidator(session);
15             fixGenerator.init();
16
17             while(session.getBBProgress() <
18                 session.getSuspectList().size()) {
19
20                 for(int i = session.getBBProgress();
21                     i < allSuspiciousBlockNum; i++) {
22                     BasicBlock bb = session.getSuspectList().get(i);
23                     ArrayList<Fix> fixes = fixGenerator.searchFixes(bb);
24                     session.getCandidateQueue().appendNewFixes(fixes);
25                     session.increaseBBProgress();
26

```

```

27         if (session.getCandidateQueue().getsize() > 30)
28             break;
29     }
30
31     while (session.getCandidateQueue().hasNextFix()) {
32         Fix fix = session.getCandidateQueue().getNextFix();
33         fixValidator.validate(fix);
34     }
35 }
36 } finally {
37     monitor.done();
38 }
39 return Status.OK_STATUS;
40 }
41 ...
42 }
```

SearchEngine 类包含了三个主要成员域，其类型是修复方案生成器 FixGenerator，修复方案验证器 FixValidator 以及修复会话 BugFixSession。在初始化时，系统需指定所需的 FixGenerator 实例并将其传入。不同的 FixGenerator 实现对应不同的修复算法，其功能体现在搜索流程实现方法 run 中。

在 run 方法内部，首先初始化修复会话 session（第 13 行），完成收集信息收集、利用 SFL 算法计算错误定位结果等操作，接着新建修复建议验证器（14 行），调用 FixGenerator 的 init 接口完成修复建议生成器所需的初始化操作（15 行）。此处的 init 是开放接口，由于此时修复会话内部的数据结构已经建立，不同的搜索算法可在此基础上完成各自所需的分析工作。搜索流程的核心结构是，对错误定位结果列表中的所有基本块上逐组完成“生成”和“检验”两个计算步骤（第 20-34 行 while 循环）。循环内部分为两个步骤，第一是生成修复方案（第 20-28 行）并将其添加到候选方案列表中，其中调用了 FixGenerator 的修复方案生成接口 searchFixes(BasicBlock bb)，用于封装不同修复算法的具体实现。第二是利用 FixValidator 提供的 validate 方法逐个验证候选方案列表中的修复方案是否满足要求（第 31-34 行）。为保证修复生成与检验的进度平衡，在候选修复建议超过 30 条时，搜索过程将从“生成”转向“检验”（第 27-28 行），这是由于“检验”通常比较耗时，在正确的修复方案出现后应尽早将其检验并返回给使用者。

代码4.1中省略了两个细节。第一，SearchEngine 继承自 Eclipse 的“任务” Job 类，run 方法是重写方法，方法内部利用类 IProgressMonitor 实现了搜索进度的控制与界面展示，方便使用者了解算法的运行进度。第二，计算过程中

可利用系统提供的日志管理器打印日志，在开发过程中了解系统的执行过程，方便调试。

不同类型修复算法均需使用上述搜索框架，框架内涉及到的开放接口或系统模块包括修复会话（BugFixSession），修复生成器（FixGenerator）以及修复验证器 FixValidator。以下将详细介绍三个类的使用方式。

4.3.3.2 修复会话（BugFixSession）

修复会话表示“一次修复方案计算过程”，在类 BugFixSession 内部包含此次计算过程所需的基本数据。

数据分为三类：

- 修复任务的基本信息，包括被测程序、测试集、每个测试用例的测试结果。在原有系统框架中，被测程序是 Eclipse 中的一个 Java 工程（Project）对象，测试集是一组 JUnit 测试方法集合，测试结果则是 JUnit 测试结果，包括通过/不通过，以及不通过时错误方法调用栈。这类信息在 BugFixSession 初始化时完成收集
- 根据基本信息产生的一些分析结果，包括测试用例的覆盖情况以及 SFL 算法据此计算出的初始错误定位排序结果。
- 与使用者交互相关信息，包括修复算法的选择，即具体使用了哪一类修复修复方案生成器，用户添加的检查点、各个检查点的当前状态、用户此次选择的修复目标。系统向不同修复算法开放这部分功能，方便实现过程中缩小错误定位范围，提高系统效率，也增加与使用者的交互性。

以上三类信息均对搜索引擎的搜索过程起到指导作用，因此均提供了开放接口以供其他模块查询。

4.3.3.3 修复生成器（FixGenerator）

修复生成器封装了针对不同错误修复算法的具体实现。框架将各修复算法的计算步骤抽象为两步，一是根据测试过程中产生的数据分析错误的原因，对应原框架中的“错误定位”，二是根据错误定位的结果针对每个可能出错的位置生成“修复方案”，对应原框架中搜索引擎中预置的“搜索模板”。不同算法在这两步的具体实现不同，但为嵌入整体框架，算法需继承 FixGenerator 类，并将两个步骤的算法封装在以下两个接口中：

```
public void init();
```

`init` 方法对应“错误定位”部分，参照它在 `SearchEngine` 中被调用的位置，可以看出此时 `BugFixSession` 已经完成了基本的信息收集，也有了初步的定位结果，若不同算法内部还需根据这些信息做进一步的分析计算，则可在 `init` 方法中完成，实现特定算法 `FixGenerator` 的初始化。

```
public ArrayList<Fix> searchFixes(BasicBlock bb);
```

`searchFixes` 方法针对一个具体的修复位置产生修复方案。其中输入参数 `BasicBlock` 是原有框架中表示 Java 代码基本块的数据结构。该结构中包含了基本块所在的代码文件、文件行号和具体代码内容。返回类型是 `ArrayList<Fix>` 是一组对应该修改位置的修复方案，其中 `Fix` 是系统中所有修复方案的抽象类型，具体细节在下一节介绍。

在实现上述接口后，`FixGenerator` 的不同实现均可无缝嵌入搜索引擎。

4.3.3.4 检验器（FixValidator）

检验器的功能是将修复方案应用回源代码，重新编译并运行测试集，检验修复方案是否能够完成本次修复会话的修复目标。系统提供了完整的检验器功能实现，新加入的算法可直接利用这一功能。但由于不同算法生成的修复方案也不相同，为利用已有检验器功能，所有修复方案都应继承自类 `Fix` 并实现以下三个接口：

```
public void doFix();
public void undoFix();
public IFile[] getModifiedFiles();
```

在检验过程中，`FixValidator` 首先调用修复方案的 `doFix` 方法完成对源代码的修改操作，接着调用 `getModifiedFiles()` 查看被修改的文件列表，编译程序并判断是否有编译错误，如果没有则重新运行测试查看结果。若通过检验，修复方案将被添加到修复方案列表中提交给使用者。最后，`FixValidator` 调用 `undoFix()` 还原源代码。

三个接口的具体实现通常需要调用 Eclipse 代码操作 API，原有系统中已有示例实现可供参考。

4.3.3.5 其他公用设施

系统提供了一些其他公共设施方便新算法的实现。首先是表达式生成器 `ExpressionGenerator`，此类实现了在不同上下文中生成符合要求的表达式集合的多种方法，包括生成可替换 If 条件的布尔表达式、替换特定语法类型的一般表

达式，根据目标表达式所在的代码位置获取局部变量、同类成员域、静态成员域，将不同表达式以运算符、方法调用等方式结合起来生成复合表达式等。生成方法实现在 Eclipse AST 上，因此输入参数也是 AST 中的元素。

第二是第三章“预过滤”策略的封装。算法生成的修复方案需要利用预过滤策略对搜索空间剪枝，则首先修复方案应继承自 `FilterableFix` 类并实现其中的相应接口。`FilterableFix` 扩展了类 `Fix`，增加的主要成员域是“目标表达式”与“替换表达式”。将修复方案转化为 `FilterableFix` 的方案可参见第三章表3.1。需要注意的是，搜索引擎并没有主动调用预过滤器，这是由于不同算法对预过滤器的需求不同，因此各个算法应在 `FixGenerator` 的 `generateFix` 方法中完成预过滤，将最终结果返回给 `SearchEngine`。

最后，为方便算法开发与调试过程，系统提供了一个公用的日志生成器 `Logger`，其提供的打印日志接口是：

```
public void log(int mode, String cause, String info);
```

其中，`mode` 表示日志条目类型，`cause` 表示记录事件类型，`info` 表示日志信息。该方法会将上述信息按照统一格式整齐打印，并添加时间戳。

4.3.4 扩展示例

Java 语言中空指针异常（`NullPointerException`，简称 NPE）是一种常见的运行时异常。当异常意外触发而没有合适的捕获处理机制时，该异常将导致程序执行中断，无法完成任务。针对 NPE 常见的修复方式是在合适位置提前加入空指针检查，或在空指针解引用操作出现时提供合适的非空对象。^[74] 中，作者提出了 9 条修复模板，事实上也分别属于这两类操作。在本节，我们在扩展框架上以实现针对空指针异常的修复系统 `NPEDebug` 为例，说明扩展框架二次开发的基本方法。

在扩展框架中实现针对特定类型的修复系统，基本想法是尽量利用系统中的已有功能快速开发，同时通过引入对特定错误的分析算法提高错误定位精度和修复方案生成准确度，提高系统效率。在 NPE 异常修复中，我们首先利用反向切片算法进一步缩小错误范围，接着定制搜索引擎中的错误生成模板，缩小搜索空间，从而达到较好的修复效果。

4.3.4.1 NPEFixGenerator

类 `NPEFixGenerator` 是 `NPEDebug` 系统中针对空指针异常定制的修复方案生成器，按照系统框架的要求，该类需实现 `init` 与 `generate` 两个方法，分别用于初始化和生成修复方案。

表 4.2 NPE 修复模板系统

修复模板	含义	搜索空间说明	e_t	e_r
引入新分支	插入 <code>if(obj == null) return/break/continue;</code>	<code>obj</code> 是下文将解引用的对象	false	<code>obj == null</code>
空指针检查	插入 <code>if(obj != null){ visit(obj); }</code>	<code>obj</code> 是下文将解引用的对象	true	<code>obj != null</code>
空值变量赋值	插入 <code>if(obj == null){ obj = e' }</code>	<code>obj</code> 是下文将解引用的对象	false	<code>obj == null</code>

在 `init` 方法中, `NPEFixGenerator` 根据异常信息定位触发异常的位置, 并使用反向切片技术分离代码中与异常触发相关的程序语句, 将其保存。其中, 反向切片是通过调用 `Wala` 分析框架中的 `BackwardSlicing` 接口完成的。进行这一操作的原因是, 原有系统给出的错误定位算法, 其代码排序的基本单位是“基本块 (**Basic Block**)”, 即在任意执行路径上都会同时执行的代码语句组合。同一基本块中的语句虽然会同时执行, 但语句之间并不存在必然的控制流与数据流依赖关系。若完全按照这一错误定位结果生成修复方案, 则可能会在“执行了”但“与异常触发”无关的语句上浪费搜索时间。因此将后向切片结果与 `SFL` 算法结果相结合, 可以得到所有与 `NPE` 异常触发相关语句的合理优先级排序结果。

参考系统中针对一般类型错误的修改生成器实现, 在获得反向切片结果后, `NPEFixGenerator` 借助类 `NPEFixSiteManager` 存储切片结果、管理与 `NPE` 相关语句所在位置。在 `searchFixes()` 方法中, `NPEFixGenerator` 首先根据输入的 `BasicBlock` 对象, 在 `NPEFixSiteManager` 中查找相关的程序语句, 接着分析可能存在空指针解引用操作的表达式, 并使用针对 `NPE` 的修复模板生成修复方案。这里使用的修复模板分为两类, 一类是空指针检查, 一类是为空值对对象赋值。具体的修复模板以及使用预过滤算法的方式参见表4.2。

4.3.4.2 NPEFix

`NPEFixGenerator` 生成出的修复方案类型为 `NPEFix`, 继承自 `FilterableFix`。依照框架规定, `NPEFix` 需实现 `doFix`, `undoFix` 和 `getModifiedFiles()` 三个接口。扩展框架没有限定上述方法的具体实现方案, 但提供了示例代码可供参考。在实际开发过程中, `NPEFix` 直接调用了父类 `FilterableFix` 的实现。因此这里我们简单解释原系统中这几个函数的实现方式。

`Eclipse` 平台提供了对 Java 工程、包、文件等的封装结构和操作 API。修复方案影响到的被修改文件是在生成过程中记录的。`doFix` 和 `undoFix` 利用其中的

表 4.3 CWENPD 测试集

类型	数量	说明
binary if	17	在 if 判断条件中应使用 && 阻断右端表达式的 NPD
deref after check	17	在判断对象是空值后仍然访问对象
int array	81	访问空值数组
Integer	81	访问空值整数
null check after deref	17	在访问对象后进行空值检查
String	81	访问空值字符串
StringBuilder	81	访问空值字符构造器

文件操作 API，将修复方案中对代码的修改方案实现为 Eclipse 代码编辑框架中的一个 `TextEdit`，并利用其自带的 `apply` 操作完成代码编辑，同时获取反向操作的编辑对象。

4.3.4.3 NPEDebug 实验

为检验 NPEDebug 的实际应用效果，我们在 CWE Juliet 测试集中的 Null Pointer Dereference 类测试程序上对 NPEDebug 进行测试。测试集共包含 375 个测试用例，表4.3列出了测试集的基本信息。

测试用例错误可分为六类，其中“null check after deref”并不会导致程序崩溃，因此 NPEDebug 无法生成修复方案。对于其余的 5 类错误，NPEDebug 可以生成有效的空指针检查语句，使得程序避开空指针异常正常执行。但对于第 1 类错误，NPEDebug 无法生成最简单的修复方案。

Listing 4.2 CWE476_NULL_Pointer_Dereference__binary_if_12.java

```

1  public void bad() throws Throwable
2  {
3      if (IO.staticReturnsTrueOrFalse ())
4      {
5          {
6              String myString = null;
7              /* FLAW: Using a single & in the if statement
8                  will cause both sides of the expression to be
9                  evaluated thus causing a NPD */
10             if ((myString != null) & (myString.length () > 0))
11             {
12                 IO.WriteLine ("The string length is greater than 0");
13             }
14         }
15     }
16     else { ... }
17 }
```

代码4.2展示了其中一个 binary-if 错误示例。在代码第 10 行的 if 条件中，中间使用的 & 符号应改为 && 符号，从而使右端表达式在左端为空时不再计算。NPEDebug 生成的修复方案是在 if 语句整体包裹在一个对 myString 做空指针检查的语句内部。这样固然可以使程序避开空指针异常，但是并不是最佳的解决办法。

一种可能的解决方案是扩展 NPEDebug 的搜索空间，增加对 If 条件表达式的修改。这可以覆盖这一类 NPD 错误，但程序的运行速度也必然有一定程度的下降。考虑到目前 NPEDebug 生成的修复方案可以使程序避免空指针异常，也比较容易理解，因此暂时保持了目前的系统版本。

4.4 本章小结

在本章中，我们分析了现有“生成-检验”修复框架存在的缺陷，并对此提出了两种框架扩展方案。第一种是“交互式调试”模式扩展，通过为使用者提供接口描述对程序状态的判断，使得系统能够利用开发人员对程序的理解，提高错误定位的精度，从而提高系统的整体效率。在实验中，我们比较了人工调试与用系统辅助调试完成调试任务所需时间，结果表明，系统辅助调试所耗时间小于人工调试时间，这说明系统确实能够提高调试效率。

第二种扩展方案是规范现有框架中的模块功能与接口，使“生成-检验”系统成为能够融合针对特定类别错误的修复算法的可扩展框架。本章描述了框架设计，并在扩展框架结构上实现了针对空指针异常的修复系统 NPEDebug。实验表明，NPEDebug 能够修复 CWE 空指针解引用类别下的测试程序。

第5章 SmartDebug 工具设计与实现

5.1 引言

“生成-检验”系统的目的是自动修复程序中的代码错误，使其能够通过对应的测试集。在当前的研究状态下，“生成-检验”系统修复正确率和效率都与实际应用有一定距离。在前几个章节中，本文提出了针对搜索引擎模块的“预过滤”优化算法、“交互式调试”框架扩展以及“针对特定类型错误的扩展框架”几种技术方案，目的都是提高“生成-检验”系统的修复准确率和系统效率。在本章中，我们将所提出的技术实现为工具原型 SmartDebug，从使用者角度介绍系统功能，并以一个示例程序说明系统的使用方式。

5.2 功能模块

SmartDebug 工具的核心功能是为开发人员提供程序的修复建议，提高程序修复速度。为方便开发人员使用，SmartDebug 实现在 Eclipse 插件平台上，与 Eclipse IDE 无缝集成在一起。

图5.1展示了 SmartDebug 的工具界面。其中，代码编辑器、调试窗口、JUnit 测试界面等都是 Eclipse Java 调试视图中的一部分，SmartDebug 加入的窗口是“检查点（Checkpoints）”管理窗口和“修复建议列表（Fix Suggestions）”窗口。其中，检查点窗口管理和展示所有已注册的检查点，检查点的状态表示着当前调试任务的进展情况，因此我们也将其称为调试进度管理窗口。这两个窗口分别对应系统的两个主要功能模块，即检查点管理模块和修复建议提示与应用模块。

5.2.1 检查点管理模块

检查点管理模块为用户提供查看和修改代码中注册的检查点的接口。如图所示，检查点按照其与测试用例的所属关系以树形结构展示在面板中，检查点的两个属性则以列表方式显示。每个检查点的状态以不同颜色图标显示，绿色表示已通过，黑色表示不通过，黄色表示未知。使用者可以在面板中完成对检查点的增、删、改、查操作。其中，增加和修改检查点功能只在程序执行某一测试用例过程中触发，这是由于检查点的增加和修改意味着开发人员在观察程序运行状态时得到了某一判断。删除则可以通过选中某一检查点，点击右上角的“X”型按钮在任意时刻完成。

5.2.2 修复建议提示与应用模块

修复建议列表窗口展示了在当前调试目标下系统找到的符合要求的修复建议。在使用过程中，系统可能会产生多条修复建议，这说明程序使用其中任一条修复建议都可以完成此次修复会话的调试目标，此时开发人员需要逐一检查这些建议条目，并选择合适的修复建议，点击右上角“**A**”字按钮，则程序将被修改。

5.3 应用示例

本节以一个具体调试任务为例说明 SmartDebug 的使用方法。代码5.1来自于本校研究生 Java 程序水平考试中某位同学答题时程序的一个中间版本。题目的要求是，给定一个素数，找出将其分解为不大于它本身的连续素数之和（包括素数本身）的方法数目。题目给出了两个测试数据点，2 和 53，对应的输出分别是 1 和 2。

代码5.1包含主程序代码和测试程序代码。在主程序中，算法分为两个步骤，第一是计算出不大于输入 `input` 的素数列表，第二是在素数列表中计算连续素数的和，查找有多少次与 `input` 相等。代码中共有两处错误，第一是在 `isPrime` 方法中判断一个数是否是素数时第 26 行的判断条件写错，应将 `i<=n` 改为 `i<n`。第二是在 `split` 方法中求连续素数和时第 41 行上的列表下标写错，应将 `j` 写成 `i`。测试代码分别对应题目给出的两个数据点。

Listing 5.1 应用示例

```

1 //----- 主程序 -----
2 public class Main {
3
4     public static int splitCount(int input) {
5         ArrayList<Integer> primeList = getPrimeList(input);
6         int count = split(input, primeList);
7         return count;
8     }
9
10    private static ArrayList<Integer> getPrimeList(int n) {
11        ArrayList<Integer> list = new ArrayList<Integer>();
12        for (int i=1; i<=n; i++) {
13            if (isPrime(i)) {
14                list.add(i);
15            }
16        }
17        return list;
18    }
19
20    private static boolean isPrime(int n) {
21        if (n == 1)

```

```

22     return false;
23     if(n==2)
24         return true;
25     int i = 2;
26     while (i<=n) { // ERROR HERE: it should be (i < n)
27         if((n % i)==0){
28             return false;
29         }
30         i++;
31     }
32     return true;
33 }
34
35 private static int split(int n, ArrayList<Integer> list){
36     int count=0;
37     int size = list.size();
38     for(int i=0;i<size;i++) {
39         int sum = 0;
40         for(int j=i;j<size;j++) {
41             sum+=list.get(i); // ERROR HERE: it should be get(j)
42             if(sum==n){
43                 count++;
44             }
45         }
46     }
47     return count;
48 }
49 }
50
51 //----- 测试代码 -----
52 public class MainTest {
53
54     @Test
55     public void test1(){
56         int input = 2;
57         int result = Main.splitCount(input);
58         Assert.assertEquals(1, result);
59     }
60
61     @Test
62     public void test2(){
63         int input = 53;
64         int result = Main.splitCount(input);
65         Assert.assertEquals(2, result);
66     }
67 }

```

使用 SmartDebug 完成调试任务的流程如下。首先，执行 JUnit 测试，会发现测试用例 test2 不通过，于是在主程序入口处加断点，单步调试该测试用例。当

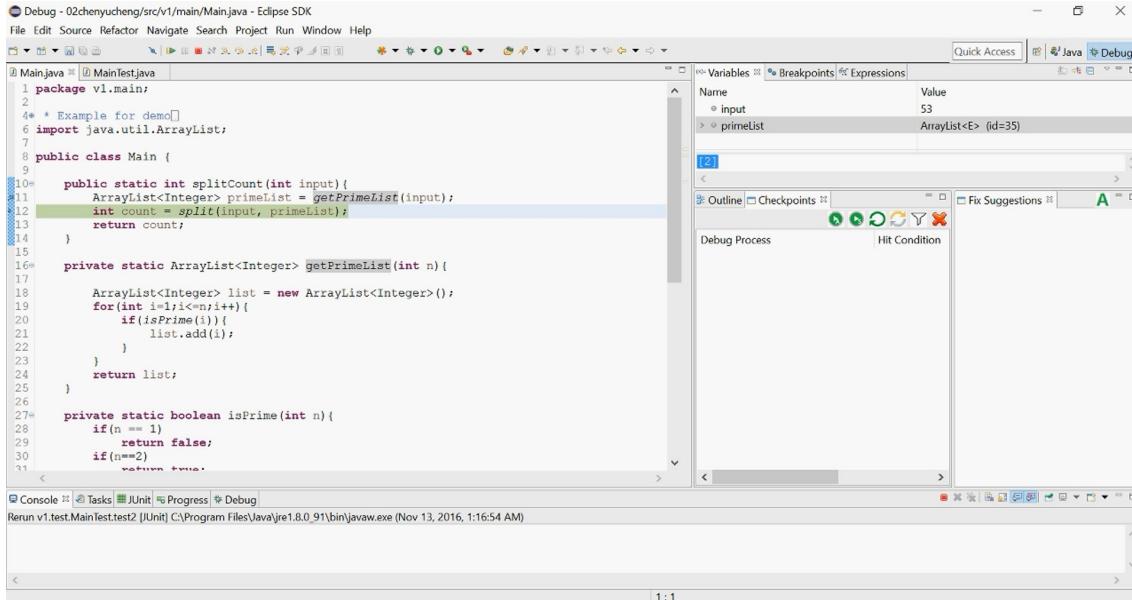


图 5.1 (1)

程序执行到图5.1中所示位置时，程序中变量 `primeList` 内容不对，应包含 53 以下的所有素数。因此，我们在此处按住 `Ctrl` 并左键单击代码编辑框左边栏，加入第一个检查点。

图5.2展示了检查点属性输入对话框。对话框左边栏输入此时执行到当前位置的次数，右边栏键入此程序状态应满足的条件。这里由于这一位置只执行一次，则右边的条件在每次程序执行到此处时都需成立，因此我们在左侧直接输入“`true`”表示右侧条件恒成立。在右侧我们输入条件表示此处 `primeList` 的大小应至少为 2。点击确定，则此时可看到检查点管理面板中已注册了这一检查点（图5.3），而代码编辑框左边栏对应位置也有了标志。

程序继续执行，在图5.4位置上，变量 `count` 表示程序找到的素数分拆方法数，此时该变量的值应为 2，因此在此处可以类似的加入检查点，将条件设置为“`count==2`”。点击确定，则如图5.5所示，新的检查点被添加。此时，点击“刷新”按钮，系统将执行程序，并计算检查点当前的状态。如图中所示，首先被加入的检查点状态是“不通过”，而后加入的检查点状态是“未知”，这是由于调试进度控制模块在更新检查点状态时，一旦在一个测试用例执行过程中遇到一个不通过的检查点就会停止运行，这使得调试任务按照一定顺序逐步完成。

图5.6标出了“预过滤”优化算法的启用按钮。由于“预过滤”算法有时会将有效的修复建议滤除，产生一定的漏报，因此 SmartDebug 提供开关供用户选择是否启用这一功能。在设置检查点后，使用者可以选择当前一个状态为“不通过”的检查点作为调试目标，点击“开始调试”按钮，则系统进入修复建议搜索状态。图5.8展示了搜索过程中系统的运行状态。SmartDebug 使用了 Eclipse 提供的进度

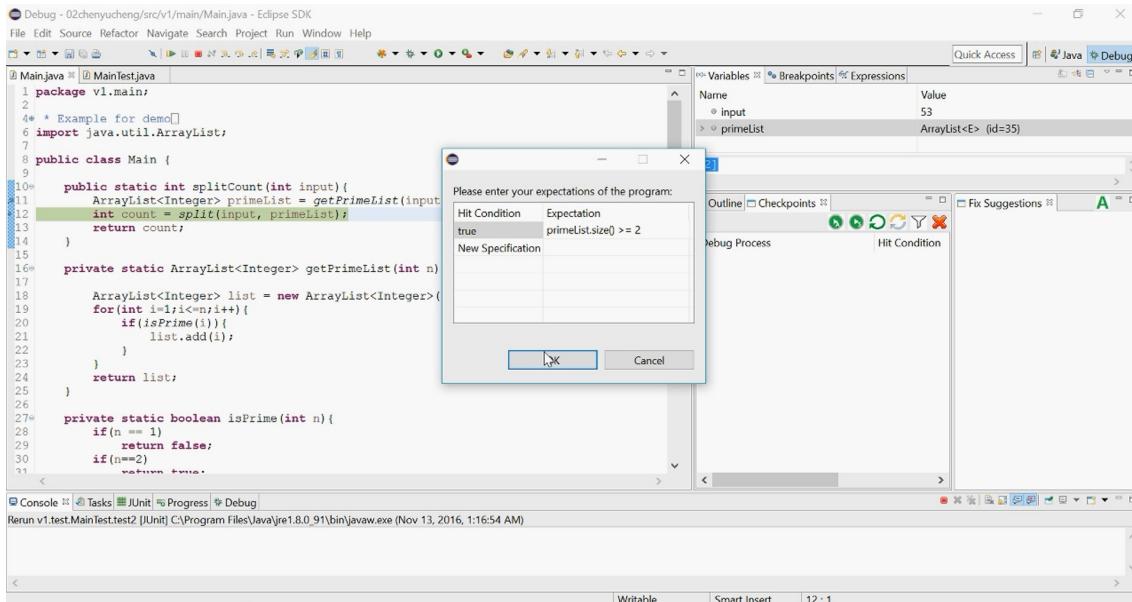


图 5.2 (2)

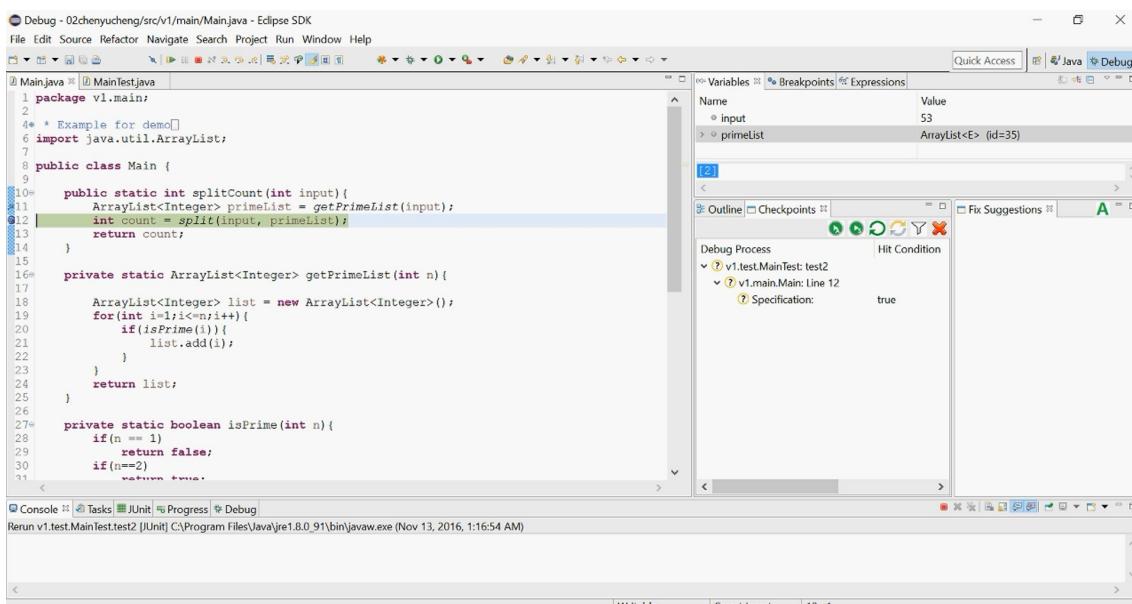


图 5.3 (3)

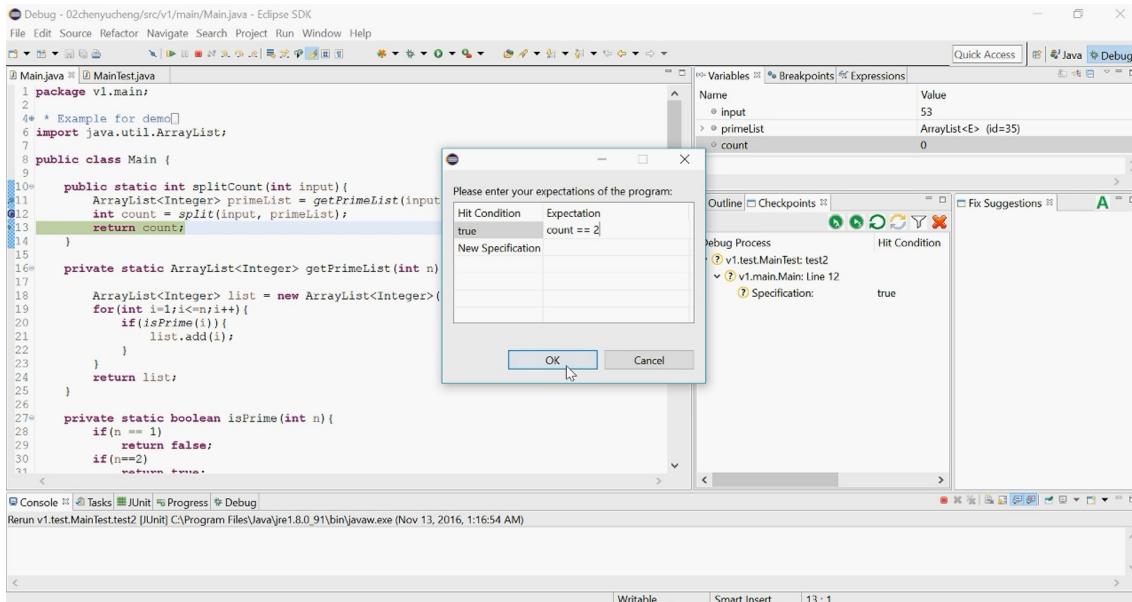


图 5.4 (4)

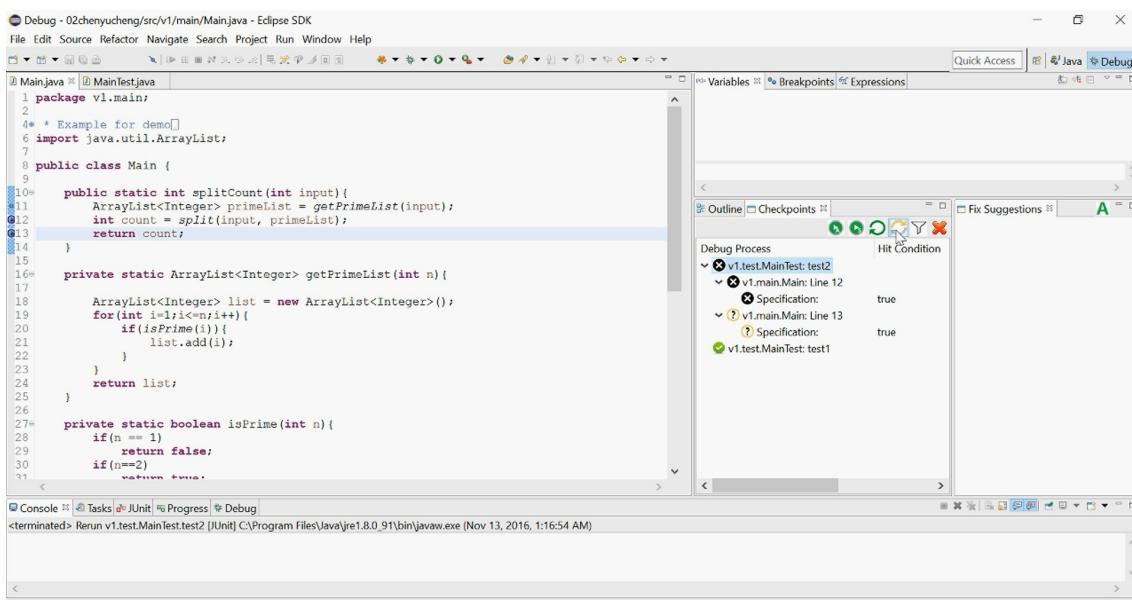


图 5.5 (5)

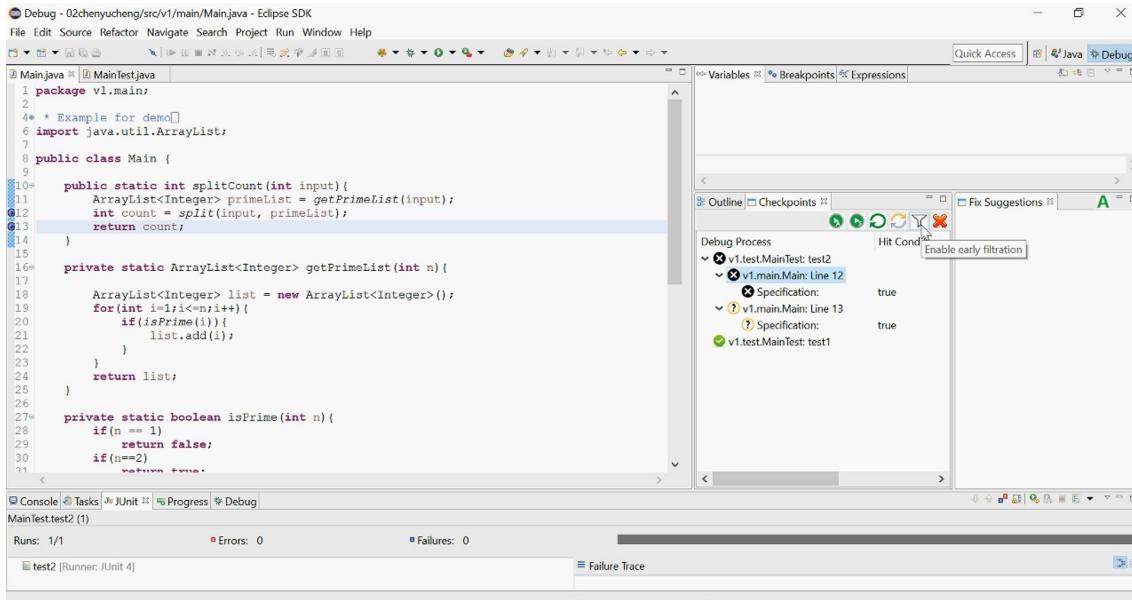


图 5.6 (6)

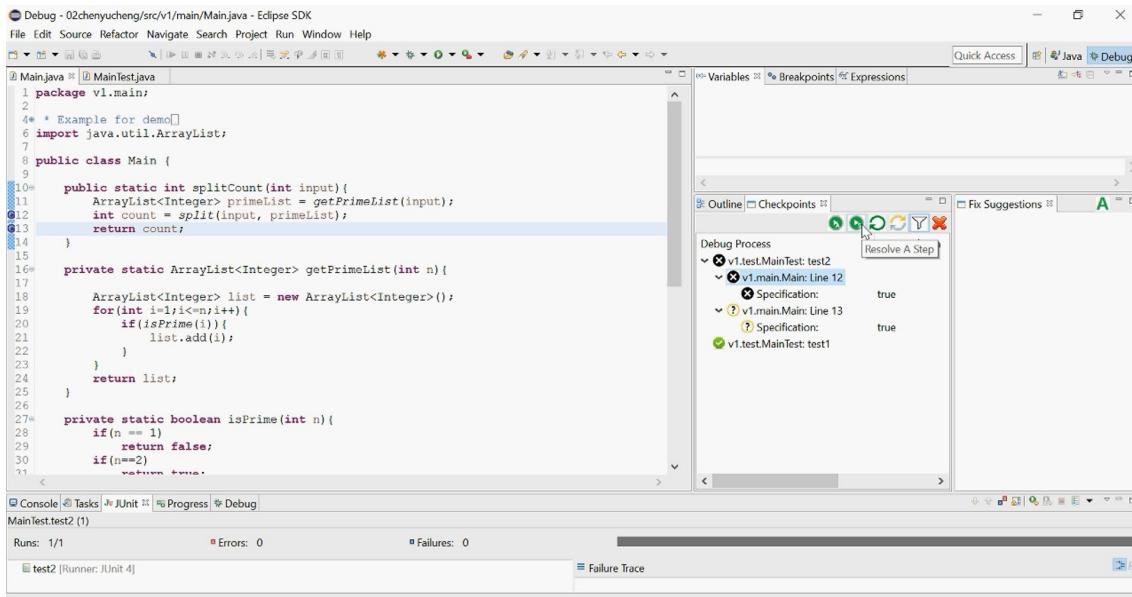


图 5.7 (7)

提示框方便使用者控制搜索过程。使用者可随时点击“取消”暂停当前搜索过程，也可在暂停后点击“继续”按钮继续刚才的搜索过程。搜索过程中合理的修复建议会实时被添加到修复建议列表框中。

当搜索过程结束时，使用者可在右侧修复建议列表框中选择合适的修改建议，点击“A”字按钮，则程序会被相应修改。

完成修改后，刷新检查点管理面板，可以看到第一个检查点状态已变成“通过”，而第二个检查点状态变为“不通过”。此时，可重复上述步骤，启动新的修复

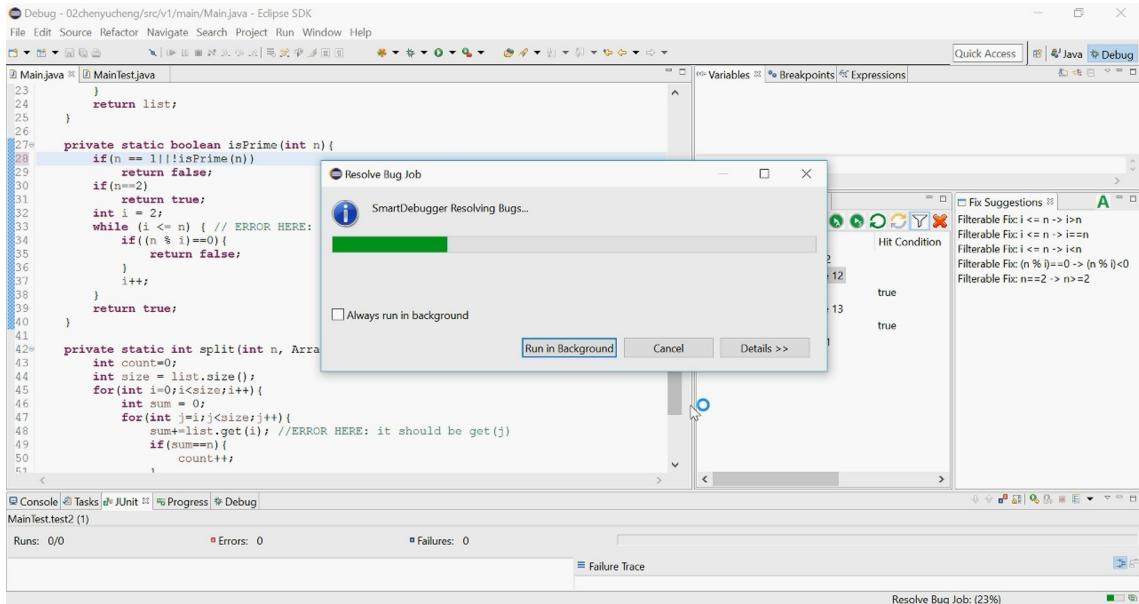


图 5.8 (8)

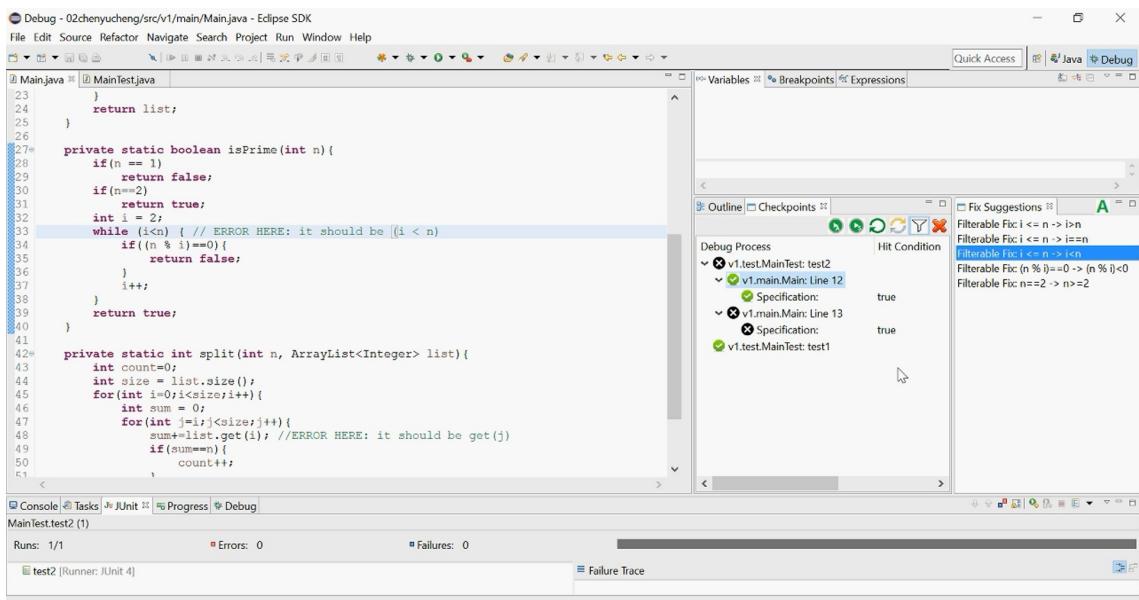


图 5.9 (9)

会话，生成新的修复建议（图5.10），将其应用于程序。此时再刷新检查点管理面板，可以看到所有检查点都通过（图5.11），重新执行测试用例，也可看到程序通过了测试。至此，调试任务已经完成。

5.4 本章小结

基于前两章的研究成果，本文实现了系统 SmartDeubg 工具原型。本章介绍了 SmartDebug 工具的主要功能，并以一个实际调试任务展示了工具的使用方式。

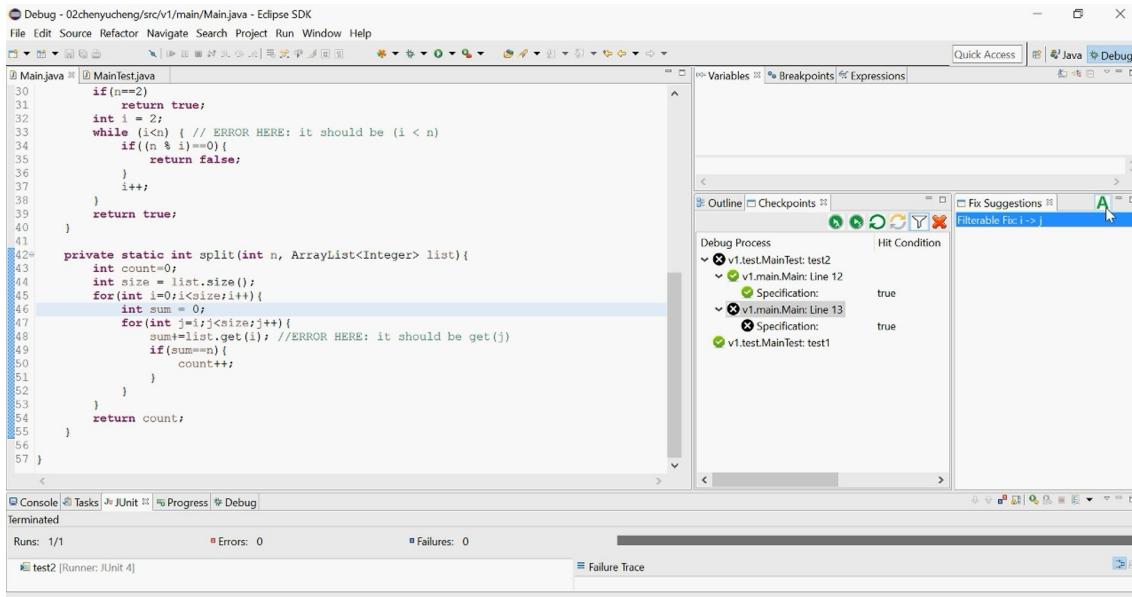


图 5.10 (10)

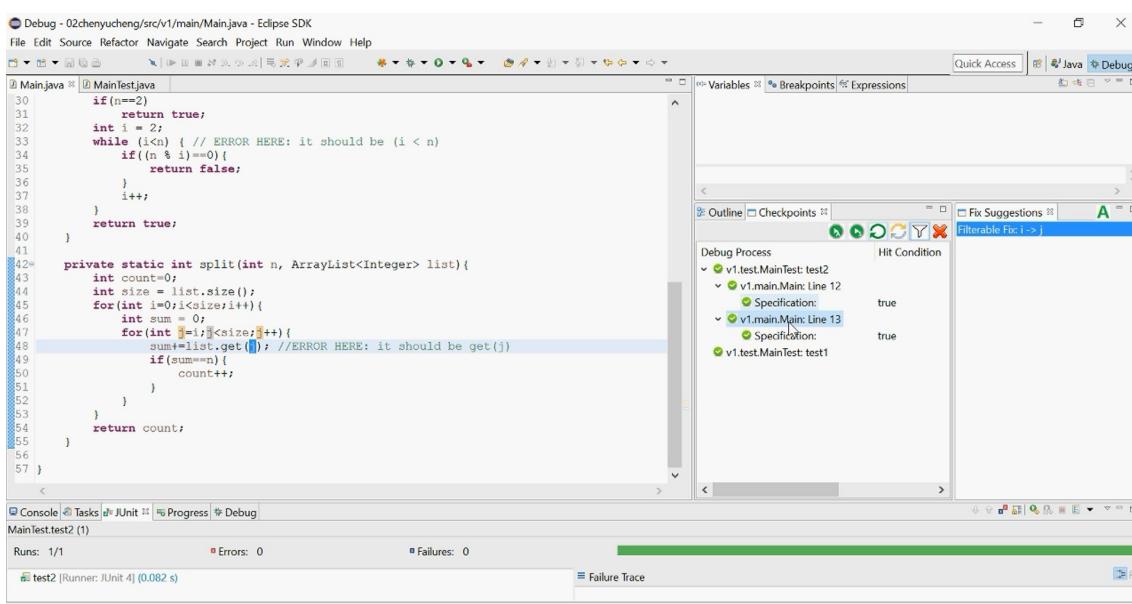


图 5.11 (11)

第6章 总结与展望

6.1 工作总结

实现错误自动修复一直是软件工程研究的美好愿景。近年来基于“生成-检验”框架的程序自动修复系统由于输入简单、错误类型限制小引起了很多研究者的注意。然而现有研究工作所实现的系统修复正确率和系统效率都难以令人满意，“生成-检验”系统推向实际应用仍有一定的距离。

基于上述背景，本文从系统内部模块优化、系统框架扩展两个方面提出提高系统的修复正确率和效率的技术方案。首先，针对框架内部“错误定位模块”，本文提出不完全正确“测试准则”存在这一事实，并以实验说明测试准则错误对“生成-检验”系统中常用的基于频谱的错误定位算法（SFL）错误定位精度的负面影响。针对这一问题，本文提出测试准则错误修复算法，使系统中错误定位算法尽量不受测试准则错误的负面影响。接着，本文针对系统的核心计算模块“搜索引擎”提出搜索优化算法。针对搜索引擎中与表达式替换、修改相关的修复模板，本文提出“预过滤”算法。该算法根据测试集中已通过和未通过的测试用例的运行时状态在备选修复方案进入到检验器前过滤掉其中不符合要求的修复方案，压缩搜索空间，减轻检验器的工作负担。实验表明，“预过滤”策略能够过滤掉搜索空间中约90%的修复方案，系统效率有较大提升，在Defects4J上的实验效果也说明本文实现系统的修复能力已超过现有其他系统。

在模块优化工作基础上，本文提出在框架层扩展系统，进一步提高系统的实用性。首先，本文提出为使用者提供与系统交互的接口，使得系统能够利用使用者的经验减少无用搜索操作，提高系统效率。接着，本文提出系统应提供二次开发接口，使得开发人员能够根据需求方便的引入针对特定类型错误的修复算法，从而使系统本身的修复能力得到补充。

最后，本文将以上算法与扩展方案实现在原型系统SmartDebug中，并将该原型集成在Eclipse平台中供开发者在日常开发过程中使用。

6.2 研究展望

“生成-检验”系统的研究前景非常广阔。在本文工作基础上，有以下几个问题可以继续深入研究。

首先，本文提出的预过滤算法应用场景仍有一定限制。目前预过滤算法只能

应用于与表达式替换、修改相关的修复模板生成的修复方案上。如何利用类似的思想将其扩展到一般修复方案中是一个有价值的研究问题。此外，目前算法实现中仍有许多基于经验假设的算法设计，例如判断两个表达式等价的标准、表达式等价与测试用例测试结果之间的关系假设等。如能在此处引入静态分析、逻辑推理等更加严格的方法，则算法的适用场景将更广泛。

其次，本文提出将使用者对程序错误的理解引入系统计算过程中，使得系统能够更准确地定位错误。在目前的设计中，使用者需要手动设置程序断点，引导系统运行程序。更友好的方式是系统能够根据计算结果提示用户应将断点设置在何处。这一提示算法也是值得探究的设计问题。

最后，本文提出了将系统模块接口开放，使系统可以方便引入针对特定类型错误的修复算法。本文仅以空指针错误为示例实现了 NPEDebug 系统。而更多其他类型错误的修复算法与“生成-检验”框架的结合应能使“生成-检验”系统的修复能力更进一步。

插图索引

图 1.1 研究思路	1
图 1.2 研究思路	6
图 2.1 4 种 SFL 算法在使用正确和错误率为 0.05 的测试准则时分数 (score) 分布情况	21
图 2.2 4 中 SFL 算法在测试准则错误率从 0.01 到 0.1 之间是精度损耗分布情况	22
图 2.3 ϕ_i 随 r 和 sim_i 的变化情况	27
图 2.4 mr 在 $[0.01, 0.1)$ 时召回率、准确率和 f1 指数分布情况	31
图 2.5 4 种 SFL 算法在修正后的测试准则下的定位精度恢复情况	34
图 2.6 SFL 算法在 grep 程序上使用正确与错误测试准则的精度对比	35
图 2.7 绝对与相对精度提升与召回率的关系	36
图 3.1 PfDebug 的系统结构	53
图 4.1 交互模式示意图	63
图 4.2 系统结构示意图	64
图 4.3 调试过程控制面板	66
图 4.4 扩展框架设计	70
图 5.1 (1)	81
图 5.2 (2)	82
图 5.3 (3)	82
图 5.4 (4)	83
图 5.5 (5)	83
图 5.6 (6)	84
图 5.7 (7)	84

图 5.8 (8)	85
图 5.9 (9)	85
图 5.10 (10).....	86
图 5.11 (11).....	86

表格索引

表 2.1	Debugging Process of a Sorting Program	14
表 2.2	不同 SFL 算法中的计算公式.....	16
表 2.3	西门子测试集简述	18
表 2.4	$fp(t_i)$ 和 $fn(t_i)$ 的概率随 ϕ_i , n 和 \hat{n} 的变化	28
表 2.5	grep 基本信息	33
表 3.1	修复模板系统.....	52
表 3.2	Defects4J 测试集基本信息.....	53
表 3.3	预过滤对搜索空间的压缩效果	54
表 3.4	修复成功率对比	56
表 4.1	SmartDebug v.s. Human	67
表 4.2	NPE 修复模板系统	75
表 4.3	CWENPD 测试集.....	76

公式索引

公式 2-1	16
公式 2-2	26
公式 2-3	26
公式 A-1	99
公式 A-2	99
公式 A-3	99
公式 A-4	99
公式 A-5	100
公式 A-6	101
公式 A-7	101
公式 A-8	101
公式 A-9	101
公式 A-10	102
公式 A-11	102
公式 A-12	102
公式 A-13	103
公式 A-14	103
公式 A-15	103
公式 A-16	104
公式 A-17	104
公式 A-18	104

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].
- [34] Pan K, Kim S, Whitehead Jr E J. Toward an understanding of bug fix patterns. Empirical Software Engineering, 2009, 14(3):286–315.
- [35] Martinez M, Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering, 2015, 20(1):176–205.
- [36] Fluri B, Wuersch M, PInzger M, et al. Change distilling:tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering, 2007, 33(11):725–743.
- [37] Goues C L, Nguyen T, Forrest S, et al. Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering, 2012, 38(1):54–72.
- [38] Goues C L, Dewey-Vogt M, Forrest S, et al. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. Software Engineering (ICSE), 2012 34th International Conference on, 2012. 3–13.
- [39] Fast E, Le Goues C, Forrest S, et al. Designing better fitness functions for automated program repair. Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA: ACM, 2010. 965–972.
- [40] Qi Y, Mao X, Lei Y, et al. The strength of random search on automated program repair. Proceedings of the 36th International Conference on Software Engineering, New York, NY, USA: ACM, 2014. 254–265.
- [41] Weimer W, Fry Z P, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, 2013. 356–366.
- [42] Qi Z, Long F, Achour S, et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems (supplementary material). 2015..
- [43] Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches. Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013. 802–811.
- [44] Tan S H, Yoshida H, Prasad M R, et al. Anti-patterns in search-based program repair. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA: ACM, 2016. 727–738.

- [45] Chandra S, Torlak E, Barman S, et al. Angelic debugging. *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011. 121–130.
- [46] DeMarco F, Xuan J, Le Berre D, et al. Automatic repair of buggy if conditions and missing preconditions with smt. *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*. ACM, 2014. 30–39.
- [47] Nguyen H D T, Qi D, Roychoudhury A, et al. Semfix: program repair via semantic analysis. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013. 772–781.
- [48] Mechtaev S, Yi J, Roychoudhury A. Directfix: Looking for simple program repairs. *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1. IEEE, 2015. 448–458.
- [49] Long F, Rinard M. Staged program repair with condition synthesis. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA: ACM, 2015. 166–178.
- [50] Long F, Rinard M. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2016. 298–312.
- [51] Gopinath D, Malik M Z, Khurshid S. Specification-based program repair using sat. *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, Berlin, Heidelberg: Springer-Verlag, 2011. 173–188.
- [52] Long F, Rinard M. An analysis of the search spaces for generate and validate patch generation systems. *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: ACM, 2016. 702–713.
- [53] Martinez M, Weimer W, Monperrus M. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. *Companion Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA: ACM, 2014. 492–495.
- [54] Qi Z, Long F, Achour S, et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM, 2015. 24–36.
- [55] Just R, Jalali D, Ernst M D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM, 2014. 437–440.
- [56] JDT E. Eclipse java development tools(jdt).
- [57] Galenson J, Reames P, Bodik R, et al. Codehint: Dynamic and interactive synthesis of code snippets. *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA: ACM, 2014. 653–663.
- [58] T. j. watson libraries for analysis (wala), 2017. wala.sourceforge.net. [Online; accessed 12-March-2016].

-
- [59] Durieux T, Martinez M, Monperrus M, et al. Automatic repair of real bugs: An experience report on the defects4j dataset. CoRR, 2015, abs/1505.07002.
 - [60] Sun X, Peng X, Li B, et al. Ipsetful: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra. Frontiers of Computer Science, 2016, 10(5):812–831.
 - [61] Gong L, Lo D, Jiang L, et al. Interactive fault localization leveraging simple user feedback. 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012. 67–76.
 - [62] Lin Y. Feedback-based debugging, 2016. <http://linyun.info/microbat/index.html>.
 - [63] Huang J, Liu P, Zhang C. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA: ACM, 2010. 207–216.
 - [64] Altekar G, Stoica I. Odr: Output-deterministic replay for multicore debugging. Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2009. 193–206.
 - [65] Yang Z, Yang M, Xu L, et al. Order: Object centric deterministic replay for java. Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2011. 30–30.
 - [66] Park S, Zhou Y, Xiong W, et al. Pres: Probabilistic replay with execution sketching on multiprocessors. Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2009. 177–192.
 - [67] Cheung A, Solar-Lezama A, Madden S. Partial replay of long-running applications. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, New York, NY, USA: ACM, 2011. 135–145.
 - [68] Yuan D, Mai H, Xiong W, et al. Sherlog: Error diagnosis by connecting clues from run-time logs. Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2010. 143–154.
 - [69] Gulzar M A, Han X, Interlandi M, et al. Interactive debugging for big data analytics. 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), Denver, CO: USENIX Association, 2016.
 - [70] Gulzar M A, Interlandi M, Condie T, et al. Bigdebug: Interactive debugger for big data analytics in apache spark. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA: ACM, 2016. 1033–1037.
 - [71] Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA: ACM, 2016. 691–701.
 - [72] Abreu R, Zoeteweij P, Gemund A J C v. An evaluation of similarity coefficients for software fault localization. Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, Washington, DC, USA: IEEE Computer Society, 2006. 39–46.
 - [73] Sinha S, Shah H, Görg C, et al. Fault localization and repair for java runtime exceptions. Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, New York, NY, USA: ACM, 2009. 153–164.

-
- [74] Cornu B, Durieux T, Seinturier L, et al. Nprefix: Automatic runtime repair of null pointer exceptions in java. CoRR, 2015, abs/1512.07423.
 - [75] Zhang C, Wang T, Wei T, et al. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010: 71–86.
 - [76] Cheng X, Zhou M, Song X, et al. Automatic fix for c integer errors by precision improvement. 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), volume 1, 2016. 2–11.
 - [77] Demsky B, Rinard M. Automatic detection and repair of errors in data structures. SIGPLAN Not., 2003, 38(11):78–95.
 - [78] Demsky B, Ernst M D, Guo P J, et al. Inference and enforcement of data structure consistency specifications. Proceedings of the 2006 International Symposium on Software Testing and Analysis, New York, NY, USA: ACM, 2006. 233–244.
 - [79] Elkarablieh B, Garcia I, Suen Y L, et al. Assertion-based repair of complex data structures. Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA: ACM, 2007. 64–73.
 - [80] Elkarablieh B, Khurshid S, Vu D, et al. Starc: Static analysis for efficient repair of complex data. SIGPLAN Not., 2007, 42(10):387–404.
 - [81] Nokhbeh Zaeem R, Khurshid S. Contract-Based Data Structure Repair Using Alloy. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010: 577–598.
 - [82] Jin G, Song L, Zhang W, et al. Automated atomicity-violation fixing. Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA: ACM, 2011. 389–400.
 - [83] Liu P, Zhang C. Axis: Automatically fixing atomicity violations through solving control constraints. Proceedings of the 34th International Conference on Software Engineering, Piscataway, NJ, USA: IEEE Press, 2012. 299–309.
 - [84] Jin G, Zhang W, Deng D, et al. Automated concurrency-bug fixing. Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2012. 221–236.
 - [85] Liu P, Tripp O, Zhang C. Grail: Context-aware fixing of concurrency bugs. Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA: ACM, 2014. 318–329.
 - [86] Khoshnood S, Kusano M, Wang C. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. Proceedings of the 2015 International Symposium on Software Testing and Analysis, New York, NY, USA: ACM, 2015. 165–176.
 - [87] Cai Y, Cao L. Fixing deadlocks via lock pre-acquisitions. Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA: ACM, 2016. 1109–1120.

附录 A 公式 2-2 和 2-3 的证明

Let $\mathbf{P}(*)$ denote the probability of $*$. Let O_c be the correct test oracle. There are two cases of false negatives and two cases of false positives. For expression simplicity, we encode them as follows:

1. $P_O F_{O_c} P_{O'} : O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F} \wedge O'(t_i) = \mathcal{P}$
2. $F_O P_{O_c} F_{O'} : O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P} \wedge O'(t_i) = \mathcal{F}$
3. $P_O P_{O_c} F_{O'} : O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P} \wedge O'(t_i) = \mathcal{F}$
4. $F_O F_{O_c} P_{O'} : O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{F} \wedge O'(t_i) = \mathcal{P}$

Let r be the error rate of the test oracle, i.e. the ratio of faulty oracle judgements to all oracle judgements. Then the probability that t_i is false negative is

$$\mathbf{P}(fn(t_i)) = \mathbf{P}(P_O F_{O_c} P_{O'}) + \mathbf{P}(F_O P_{O_c} F_{O'}) \quad (\text{A-1})$$

And the probability that t_i is false positive is

$$\mathbf{P}(fp(t_i)) = \mathbf{P}(P_O F_{O_c} P_{O'}) + \mathbf{P}(F_O P_{O_c} F_{O'}) \quad (\text{A-2})$$

And now we calculate $\mathbf{P}(P_O F_{O_c} P_{O'})$, $\mathbf{P}(F_O P_{O_c} F_{O'})$, $\mathbf{P}(P_O F_{O_c} P_{O'})$ and $\mathbf{P}(F_O P_{O_c} F_{O'})$.

$$\begin{aligned} & \mathbf{P}(P_O F_{O_c} P_{O'}) \\ &= \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F} \wedge O'(t_i) = \mathcal{P}) \\ &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \cdot \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \end{aligned} \quad (\text{A-3})$$

$$\begin{aligned} & \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \\ &= \mathbf{P}(Suspicion(t_i) \leq thres) \\ &= \mathbf{P}\left(\frac{Vote_{if}}{Vote_{ip} + Vote_{if}} \leq thres\right) \\ &= \mathbf{P}\left(\frac{\sum_{t_{iq} \in T_{i,f}} Sim(t_i, t_{iq})}{\sum_{t_{iq} \in T_i} Sim(t_i, t_{iq})} \leq thres\right) \end{aligned} \quad (\text{A-4})$$

We assume that test cases in T_i are similar enough to t_i and the similarity are almost the same. Then approximately $\forall t_{iq} \in T_i$, $\text{Sim}(t_i, t_{iq}) \approx sim_i$, where sim_i represents the average similarity of all $\text{Sim}(t_i, t_{iq})$. Then we have

$$\begin{aligned}
& \mathbf{P}\left(\frac{\sum_{t_{iq} \in T_{i,f}} \text{Sim}(t_i, t_{iq})}{\sum_{t_{iq} \in T_i} \text{Sim}(t_i, t_{iq})} \leq thres\right) \\
& \approx \mathbf{P}\left(\frac{|T_{i,f}| \times sim_i}{|T_i| \times sim_i} \leq thres\right) \\
& = \mathbf{P}\left(\frac{|T_{i,f}|}{|T_i|} \leq thres\right) \\
& = \mathbf{P}(|T_{i,f}| \leq |T_i| \times thres) \\
& = \sum_{w=0}^{\hat{n}} \mathbf{P}(|T_{i,f}| = w)
\end{aligned} \tag{A-5}$$

where $\hat{n} = \lfloor n \times thres \rfloor$.

Precise calculation of $\mathbf{P}(|T_{i,f}| = w)$ (given w) is obviously very complicated. Fortunately in our case an accurate estimation is good enough. To simplify the problem, we make the following two assumptions:

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

Assump. 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)

The reasons for making such assumptions are explained in Section 3.3 thus not repeated here. Based on these two assumptions,

$$\begin{aligned}
& \mathbf{P}(|T_{i,f}| = w) \\
& = C_n^w (\mathbf{P}(O(t_{iq}) = \mathcal{F}))^w (\mathbf{P}(O(t_{iq}) = \mathcal{P}))^{(n-w)} \\
& = C_n^w (\mathbf{P}(O_c(t_{iq}) = \mathcal{F}) \mathbf{P}(O(t_{iq}) = O_c(t_{iq})) + \mathbf{P}(O_c(t_{iq}) = \mathcal{P}) \mathbf{P}(O(t_{iq}) \neq O_c(t_{iq})))^w \\
& \quad \cdot (\mathbf{P}(O_c(t_{iq}) = \mathcal{P}) \mathbf{P}(O(t_{iq}) = O_c(t_{iq})) + \mathbf{P}(O_c(t_{iq}) = \mathcal{F}) \mathbf{P}(O(t_{iq}) \neq O_c(t_{iq})))^{(n-w)}
\end{aligned}$$

where t_{iq} is an arbitrary test case in T_i .

Remember r is the error rate of the test oracle, then $\mathbf{P}(O(t_{iq}) = O_c(t_{iq})) = 1 - r$, $\mathbf{P}(O(t_{iq}) \neq O_c(t_{iq})) = r$. Use β_i to represent $\mathbf{P}(O_c(t_{iq}) = \mathcal{F})$, $t_{iq} \in T_i$, then $\mathbf{P}(O_c(t_{iq}) =$

$\mathcal{P}) = 1 - \beta_i$, and

$$\begin{aligned} & \mathbf{P}(|T_{i,f}| = w) \\ &= C_n^w (\beta_i(1-r) + (1-\beta_i)r)^w ((1-\beta_i)(1-r) + \beta_i r)^{(n-w)} \\ &= C_n^w (\beta_i + r - 2\beta_i r)^w (1 - (\beta_i + r - 2\beta_i r))^{(n-w)} \end{aligned} \quad (\text{A-6})$$

r is completely dependent on the oracle error itself, so we cannot compute its value from elsewhere. However, β_i can be deducted through sim_i . $\forall t_{iq_1}, t_{iq_2} \in T_i \cup \{t_i\}$, $iq_1 \neq iq_2$, according to *Assump. 1*,

$$\begin{aligned} & \mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) \\ &= \mathbf{P}(O_c(t_{iq_1}) = \mathcal{P}) \mathbf{P}(O_c(t_{iq_2}) = \mathcal{P}) + \mathbf{P}(O_c(t_{iq_1}) = \mathcal{F}) \mathbf{P}(O_c(t_{iq_2}) = \mathcal{F}) \\ &= (1 - \beta_i)^2 + \beta_i^2 \end{aligned} \quad (\text{A-7})$$

while according to *Assump. 2*,

$$\mathbf{P}(O_c(t_{iq_1}) = O_c(t_{iq_2})) = sim_i$$

Therefore,

$$(1 - \beta_i)^2 + \beta_i^2 = sim_i$$

Solving this equation, we get $\beta_{i1} = \frac{1+\sqrt{2sim_i-1}}{2}$ or $\beta_{i2} = \frac{1-\sqrt{2sim_i-1}}{2}$. Since we have assumed that all test cases in $T_i \cup \{t_i\}$ are similar enough, sim_i should be close to 1, and $2sim_i - 1 > 0$. As we know that $O_c(t_i) = \mathcal{F}$, therefore

$$\mathbf{P}(O_c(t_{iq}) = \mathcal{F}) = \beta_{i1} = \frac{1 + \sqrt{2sim_i - 1}}{2} \quad (\text{A-8})$$

Synthesizing equation A-4, A-5 and A-6,

$$\begin{aligned} & \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \\ &= \sum_{w=0}^{\hat{n}} C_n^w (\beta_{i1} + r - 2\beta_{i1}r)^w (1 - (\beta_{i1} + r - 2\beta_{i1}r))^{(n-w)} \end{aligned} \quad (\text{A-9})$$

Similarly, we have

$$\begin{aligned}
 & \mathbf{P}(F_O P_{O_c} F_{O'}) \\
 = & \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P} \wedge O'(t_i) = \mathcal{F}) \tag{A-10} \\
 = & \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P}) \cdot \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P})
 \end{aligned}$$

and

$$\begin{aligned}
 & \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P}) \\
 = & \sum_{w=0}^{\hat{n}} \mathbf{P}(|T_{i,p}| = w) \\
 = & \sum_{w=0}^{\hat{n}} C_n^w \cdot (\mathbf{P}(O(t_{iq}) = \mathcal{P}))^w \cdot (\mathbf{P}(O(t_{iq}) = \mathcal{F}))^{(n-w)} \tag{A-11} \\
 = & \sum_{w=0}^{\hat{n}} C_n^w \cdot ((1 - \beta_i)(1 - r) + \beta_i r)^w \cdot (\beta_i(1 - r) + (1 - \beta_i)r)^{(n-w)}
 \end{aligned}$$

where β represents $\mathbf{P}(O_c(t_{iq}) = \mathcal{F})$. In this case, $O_c(t_i) = \mathcal{P}$, therefore

$$\mathbf{P}(O_c(t_{iq}) = \mathcal{F}) = \beta_{i2} = \frac{1 - \sqrt{2\sin m_i - 1}}{2}$$

Since $\beta_{i1} + \beta_{i2} = 1$, we have

$$\begin{aligned}
 & \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P}) \\
 = & \sum_{w=0}^{\hat{n}} C_n^w ((1 - \beta_{i2})(1 - r) + \beta_{i2} r)^w (\beta_{i2}(1 - r) + (1 - \beta_{i2})r)^{(n-w)} \tag{A-12} \\
 = & \sum_{w=0}^{\hat{n}} C_n^w (\beta_{i1}(1 - r) + (1 - \beta_{i1})r)^w ((1 - \beta_{i1})(1 - r) + \beta_{i1} r)^{(n-w)} \\
 = & \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F})
 \end{aligned}$$

Synthesizing equations A-1, A-10, A-3, A-9 and A-12, we have

$$\begin{aligned}
 & \mathbf{P}(fn(t_i)) = \mathbf{P}(P_O F_{O_c} P_{O'}) + \mathbf{P}(F_O P_{O_c} F_{O'}) \\
 &= \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F} \wedge O'(t_i) = \mathcal{P}) + \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P} \wedge O'(t_i) = \mathcal{F}) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \cdot \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \\
 &\quad + \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P}) \cdot \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P}) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \\
 &\quad \cdot \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) + \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{P}) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \cdot \mathbf{P}(O(t_i) \neq O_c(t_i)) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \cdot r \\
 &\approx r \sum_{w=0}^{\hat{n}} C_n^w \phi_i^w (1 - \phi_i)^{(n-w)} \tag{A-13}
 \end{aligned}$$

where $\phi_i = \beta_{i1} + r - 2\beta_{i1}r$.

Following the same routine,

$$\begin{aligned}
 & \mathbf{P}(P_O P_{O_c} F_{O'}) \\
 &= \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P} \wedge O'(t_i) = \mathcal{F}) \tag{A-14} \\
 &= \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P}) \cdot \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P})
 \end{aligned}$$

and

$$\begin{aligned}
 & \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P}) \\
 & \approx \sum_{w=\hat{n}+1}^n \mathbf{P}(|T_{i,f}| = w) \\
 &= \sum_{w=\hat{n}+1}^n C_n^w (1 - (\beta_{i1} + r - 2\beta_{i1}r))^w (\beta_{i1} + r - 2\beta_{i1}r)^{(n-w)} \tag{A-15} \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{F})
 \end{aligned}$$

then

$$\begin{aligned}
 \mathbf{P}(fp(t_i)) &= \mathbf{P}(P_O P_{O_c} F_{O'}) + \mathbf{P}(F_O F_{O_c} P_{O'}) \\
 &= (\mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P} \wedge O'(t_i) = \mathcal{F}) + (\mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{F} \wedge O'(t_i) = \mathcal{P})) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P}) \cdot \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P}) \\
 &\quad + \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{F}) \cdot \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{F}) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{F} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P}) \\
 &\quad \cdot \mathbf{P}(O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{P}) + \mathbf{P}(O(t_i) = \mathcal{F} \wedge O_c(t_i) = \mathcal{F}) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \cdot \mathbf{P}(O(t_i) = O_c(t_i)) \\
 &= \mathbf{P}(O'(t_i) = \mathcal{P} | O(t_i) = \mathcal{P} \wedge O_c(t_i) = \mathcal{F}) \cdot (1 - r) \\
 &\approx (1 - r) \sum_{w=\hat{n}+1}^n C_n^w (1 - \phi_i)^w \phi_i^{(n-w)} \tag{A-16}
 \end{aligned}$$

where $\phi_i = \beta_{i1} + r - 2\beta_{i1}r$.

Finally, we get

$$\mathbf{P}(fn(t_i)) \approx r \sum_{w=0}^{\hat{n}} C_n^w \phi_i^w (1 - \phi_i)^{(n-w)} \tag{A-17}$$

$$\mathbf{P}(fp(t_i)) \approx (1 - r) \sum_{w=\hat{n}+1}^n C_n^w (1 - \phi_i)^w \phi_i^{(n-w)} \tag{A-18}$$

where $\phi_i = \beta_{i1} + r - 2\beta_{i1}r$, $\beta_{i1} = \frac{1+\sqrt{2sim_i-1}}{2}$, sim_i is the average similarity of all test cases in the same T_i , $\hat{n} = \lfloor n \times thres \rfloor$, r is the error rate of the test oracle.

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