中图分类号：TP311.5

论 文 编 号：10006BY1103139

北京航空航天大學

博士学位论文

自适应软件缺陷定位理论与技术研究

作者姓名　　张逍怡

学科专业　　导航、制导与控制

指导教师　　蔡开元 教授

培养学院　　自动化科学与电气工程学院

Study on Adaptive Software Fault Localization: Theory and Technology

A Dissertation Submitted for the Degree of Doctor of Philosophy

Candidate: Zhang Xiao-Yi

Supervisor: Prof. Cai Kai-Yuan

School of Automation Science and Electrical Engineering,
Beihang University, Beijing, China

中图分类号：TP311.5

论 文 编 号：10006BY1103139

博 士 学 位 论 文

自适应软件缺陷定位理论与技术研究

作者姓名	张逍怡	申请学位级别	工学博士
指导教师姓名	蔡开元	职　　称	教授
学科专业	导航、制导与控制	研究方向	软件控制论
学习时间自	2011 年 09 月 01 日	起至	2018 年 4 月 25 日止
论文提交日期	2018 年 04 月 25 日	论文答辩日期	2018 年 6 月 5 日
学位授予单位	北京航空航天大学	学位授予日期	年　月　日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期： 年 月 日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。保密学位论文在解密后的使用授权同上。

学位论文作者签名：_____ 日期： 年 月 日

指 导 教 师 签 名：_____ 日期： 年 月 日

摘　　要

随着信息全球化进程的加速，软件在人们的生产生活中扮演着愈发重要的角色。为了能够适应信息流通，技术更新，以及观念变化的节奏，当代软件的质量与敏捷性需求都变得日益强烈。在软件开发周期中，软件测试与调试花费了大部分时间与资源，这无疑成为软件满足需求过程中遇到的严峻挑战。其中，软件缺陷定位环节作为调试环节的第一步，在整个测试 -调试 -修复 -再测试这一循环中起着至关重要的作用。近十年来，自动化软件缺陷定位方法一直作为学术界研究的热点而被广泛关注。目前这些方法大多通过收集软件测试中获得的程序执行信息，并利用统计学、逻辑推理等方法推测缺陷语句在程序中所处的位置。

然而软件的多样性以及软件开发环境的多样性使得不同软件工程项目中测试信息的获取方式，以及相应测试信息空间的表述及分布都存在很大区别。尽管自动化的缺陷定位方法很多，但很难找到一种普适的缺陷定位方法。因此，软件缺陷定位方法的适应性就成为目前自动化软件缺陷定位这一研究领域所面临的一大挑战。自动化软件缺陷定位的适应性主要涉及两方面问题：

（1）对软件本身的适应性；

（2）对软件开发环境的适应性。

基于上述问题，本文针对自适应软件缺陷定位的理论与方法展开研究，通过在缺陷定位相关的各个过程中合理地引入自适应机制，以提升软件缺陷定位在不同目标程序及缺陷定位环境中的适应性。此外，我们构建对软件多样性特征具有较强表达能力的分析框架，以此为工具对缺陷定位方法的执行机理进行刻画与观测，并对其适应性进行合理地评估。

本文的主要贡献和创新点体现在以下四个方面：

（1）构建了自适应软件缺陷定位的研究框架

对软件缺陷定位过程的各个环节进行深入剖析，对各环节的自适应机制进行描述，并对其输入输出进行定义，进而构建自适应软件缺陷定位的整体研究框架。此外，建立缺陷定位的基本模型，并在此基础上讨论自适应机制的合理性与必要性。

（2）提出了面向缺陷定位的自适应测试方法

针对面向缺陷定位的测试问题进行深入研究，提出了基于输入空间划分的测试策

略，以及基于程序覆盖信息多样性的测试策略。其中，基于程序覆盖信息的策略中，针对动态开发环境中的测试信息多样性问题提出了面向缺陷定位的度量准则。

（3）提出了基于测试分类的无标签测试用例利用方法

研究测试 Oracle 问题对缺陷定位造成的影响，在此基础上提出基于测试分类的解决方案，对由 Oracle 问题导致的无标签测试信息进行处理，并为处理后的信息设计缺陷定位环境下的具体利用方案。其中，测试分类器的设计基于对当前缺陷定位结果的观测与辨识，包含了自适应思想。

（4）构建了可视化的软件缺陷定位理论分析与机理描述框架

引入程序谱图与公式曲线的概念，并在此基础上构建程序谱缺陷定位方法的机理描述与理论分析框架，该框架能够直观地描述不同程序谱的特征差异性。在该框架下对单缺陷下缺陷定位可疑度计算公式的性能进行分析，并在考虑适应性的基础上得到更为丰富的比较结论。此外，在去除单缺陷假设的一般条件下，分析了程序谱空间的区域特征，并提出基于程序谱空间划分的语句过滤方法，以提升各可疑度计算公式对不同目标程序的适应性。

关键词：软件缺陷定位，软件测试，自适应方法，程序谱缺陷定位，测试信息处理，测试分类，程序谱分析

Abstract

Along with the acceleration of globalization, software has been playing a more and more important role in industrial production and in our daily life. In adapt to the fast pace of information communication, technological innovation and changes of concepts, the requirements of quality and agility for current software products are increasingly highlighted. Thus the process of testing and debugging that consumes the most time and resources becomes a serve challenge to satisfy these requirements. Particularly, as the first activity of debugging process, fault localization takes very important effects in the "test-find-fix" cycle. In the last decade, the study on automatic fault localization approaches has become a hot topic and received widespread attention. To date, various fault localization approaches and techniques have been proposed. Most of these approaches are based on test information collected during the testing process. They use the derived information to predict the location of faulty statements based on the knowledge borrowed from the science of statistics or logical reasoning.

Nevertheless, because of the variety of software products and software development environments, different software projects will yield different expressions and distributions of the test information. As a result, although there are many fault localization approaches, we cannot find the one that has good performance in all the situations. Therefore, the adaptability of fault localization approaches is a big challenge in the research of software fault localization. Considering the adaptability of automatic software fault localization, two problems should be addressed:

1) the adaptability to the variety of software product,

2) the adaptability to the variety of software development environments.

Based on the above problems, this thesis studies the theory and approach of the adaptive fault localization, aiming to introduce rational adaptive mechanisms to improve the adaptability to various situations. In addition, we try to propose the framework that has strong ability to describe the variety of software, use this framework as a tool to observe the mechanism of the fault localization techniques on different subject programs, and then evaluate their adaptability. The contribution of this thesis can be summarized as follows:

(1) Introduce the research framework for adaptive fault localization

We comprehensively analyse the procedures during the activity of fault localization,

and describe the related adaptive mechanisms, defining the inputs and outputs for each of them. Then we introduce the general research framework of adaptive fault localization. The rationality of each adaptive mechanism is discussed based on this framework.

(2) Propose the fault localization oriented testing approaches

Comprehensively study the testing process conducted aiming to facilitate fault localization. Propose the Fault Localization Prioritization (FLP) approaches based on partition and coverage information, respectively. Particularly, in the coverage-based approach, the measurement of test information diversity especially for fault localization is introduced.

(3) Propose the test classification based approach to make use of the unlabelled test cases

Explore the influence of oracle problem on fault localization. Propose the test classification based approach to tackle the unlabelled test cases, and to utilize the processed test information in fault localization. In particuler, the design of test classifier is based on the observation and identification of current fault localization result, which reflects the idea of adaptive.

(4) Establish the visualization framework to make description and theoretical analyses for fault localization

Introduce the terms of spectra plot and metric curve, and then introduce a visual framework to make description and theoretical analyses for fault localization. This framework can vividly describe the distinctions of the characteristics for different programs. We use this framework to analyse the performance for each fault localization metric under single-fault assumption and achieve more results under the consideration of adaptability. In addition, without single-fault assumption, we universally analyse the area features of the program spectra space, and propose the statement filtering approach based on the division of spectra space that improves the adaptability of each fault localization metric.

Keywords: Software fault localization, Software Testing, Adaptive approach, Spectra-based fault localization, Test information processing, Test classification, Program spectra analysis

目　　录

插图目录

表格目录

第一章　绪论

软件测试与软件调试是软件工程过程中非常重要的环节，其在执行过程中会消耗大量时间与劳动力。有统计称，在整个软件的开发与运维周期中，50% 至 80% 的资源都会花费在测试与调试任务上[1, 2]；这其中调试任务因其过程复杂、难度高、包含不确定性等特点而显得尤为艰巨。据不完全统计，大多数软件工程师都会把一半以上的时间花费在软件调试上[3]。而本论文所讨论的软件缺陷定位就是调试环节的第一步，也是至关重要的一步[4]。

随着信息产业全球化的逐步推进，人们无时无刻不在获取着大量的信息，亦无时无刻不在渴望着新的信息。软件作为信息获取、处理及传递的重要媒介也在人类社会生活中扮演着愈发重要的角色。这使得软件的应用范围越来越广，单个软件所承载的功能性需求也越来越多，而软件工程中对于软件开发及更新速度的需求也日益迫切。然而，对于这些具有复杂结构与行为的软件来说，耗时巨大并且难于管理的软件调试环节无疑成为满足需求时所遇到的最大障碍之一。

本论文以软件控制论的基本思想为指导，以更好地满足“可靠性”、“敏捷性”需求为目标，针对软件调试环节中“如何高效地在程序中寻找导致软件失效的代码”这一问题引入自适应思想，构建自适应软件缺陷定位的研究框架。在该框架下，借助智能决策、数据挖掘以及机器学习相关的理论与技术对软件缺陷定位执行过程中各个子环节的适应性进行评估，并在此基础上通过合理地引入自适应机制（或是对现有的自适应机制进行改进完善），来切实提升自动化软件缺陷定位方法的性能与实际执行效率，进而提升软件缺陷定位在整个软件工程过程中的自动化程度，以及软件缺陷定位方法本身的应用价值。

1.1　引言

本节首先介绍论文的项目与应用背景，并在此基础上从自动化软件缺陷定位的必要性、软件缺陷定位的适应性需求、当前软件缺陷定位方法存在的问题以及相关理论与技术的研究价值这几个方面阐述本文的选题依据和意义。最后，我们给出软件缺陷定位研究中涉及到的相关概念。

1.1.1　研究背景与选题依据

本论文研究受到国家自然科学基金“基于复杂网络可控性的 GUI 软件回归测试方法研究（61402027）”以及“面向运行环境依赖缺陷的软件自动化调试技术研究（61772055）”的支持。

论文的研究背景将从以下三个方面阐述：1）计算机技术的广泛应用以及软件工程理论与技术的日益发展[5] 是自动化软件缺陷定位技术的应用前提；2）软件与软件开发环境的多样性以及缺陷定位任务本身的复杂性为软件缺陷定位技术的适应能力带来挑战，这是本文研究的出发点；3）软件控制论思想[6] 以及现有软件缺陷定位技术为本文的研究工作提供了理论依据和方法论基础。

（1）随着计算机技术的广泛应用以及软件工程理论与技术的日益发展，软件在人们生产生活的各个角落发挥着重要作用，而软件缺陷带来的影响也越来越严重。因此如何提升软件调试的效率逐渐成为热点研究问题。

有研究指出，程序员在编码时的出错率为 0.5%-3%[7]，也有研究称编码出错率在 0.1% 左右[8]。然而，无论具体的出错率如何，程序代码中都不可避免地会存在错误（Bug）。

随着社会经济的发展，人们对生活质量的追求也越来越高，而计算机技术的广泛应用使得软件所能完成的功能越来越多。软件一方面代替人类完成了很多高难度任务，例如飞行控制、智能编队及任务规划、信息处理及决策等；另一方面，其还能够表达人们精神世界中更为复杂的概念与事物，例如游戏软件。然而，功能层面上广度与深度的拓展使得软件的规模不断增大，各组件间的逻辑关系也会变得更为复杂。假设程序员写每行代码的出错率不变，那么软件规模的增大就会直接导致程序出错的可能性增加。而软件内部复杂的逻辑关系则进一步导致软件失效很难被预测，且失效机理难以被理解。

此外，由于软件所承担的任务越来越重要，软件失效所带来的后果也愈发严重。1994 年 3 月 23 日，罗斯航空客的空中客车 A310-304 在由莫斯科舍列梅季耶夫国际机场飞往香港启德机场途中坠毁在俄罗斯西伯利亚地区荒郊，机上 75 名机员及乘客全部罹难[9]；而这个惨剧正是由于控制系统的缺陷造成的。又如，2012 年 8 月 1 日，美国大型证券中介公司——骑士资本美洲公司——由于其开发的交易算法软件的 Bug，导致大量虚假订单的出现，造成超过 4 亿美元的损失，并最终导致公司的破产[10]。

软件缺陷变得难以理解，软件失效后果愈发严重，这些都加重了软件测试及调试任务的强度。然而，在这快节奏的时代，科学技术飞速发展，人们的需求也在时刻改变。尽管软件的测试及调试任务变得更加严峻，人们对软件开发与更新效率的需求却日益提升[11]。如此一来，耗时巨大的软件调试任务就成为满足需求过程中的一大障碍，而如何缩短调试环节的时间及资源消耗就成为必须考虑的问题。

随着系统科学、信息科学与控制科学的发展，用于程序代码及其执行过程的观测与分析的理论与手段被广泛研究与应用，这使得我们能够得到大量对软件调试有用的数据。此外，我们还可以借鉴这些学科的先进思想，设计各种方法及自动化流程来辅助传统的手工调试过程，进而节约大量的人力与时间。

此外，近年来开源软件及开源社区迅速发展[12]。一份 2008 年的报告称，开源社区软件每年能够为使用者节省约 600 亿美元 [13, 14]。截止到 2017 年 4 月，著名的开源代码库以及版本控制系统 GitHub[15, 16] 已经拥有约 2000 万用户以及 5700 万版本库[17]。这些开源平台提供了大量真实的程序。这些程序为调试技术与方法的验证与实践提供了大量的实验对象与辅助工具。

（2） 对不同软件来说，其开发模式、设计架构以及所描述的对象会有很大不同，这使得执行缺陷定位任务时能够利用的信息在表达、分布以及获取形式上也各不相同。这些都会为缺陷定位方法中信息处理以及定位准则的设计带来困难。此外由于软件缺陷的种类繁多且很多缺陷的失效机理会比较复杂，使得软件缺陷定位过程成为一个多维信息空间中的复杂推理过程。

- 软件本身的多样性

软件是所有计算机程序及相关文档的统称，而其本质是一种关联思维的具象化。思维本身是无限的，能够描述世间万物，甚至刻画一些超越物质范畴的抽象概念，因此软件本身一定是多种多样的。

根据需求，软件开发可能遵循不同的编程思想与架构模式。就编程思想而言，就存在面向过程[18]、面向对象[19]、命令式与符号编程[20] 等思想。此外，就架构模式[21] 这个维度来看，软件又包含分层模式（Layers）、面向服务模式（Service-Oriented Architecture，SOA）[22, 23, 24]、事件驱动架构（Event-driven architecture）等。

不同的描述对象、编程思想以及架构模式都会导致软件分析时所涉及的信息空间在构成与数据分布（如程序的输入空间、静态分析时各组件间的拓扑结构与调用关

系[25, 26, 27] 以及程序执行空间中语句覆盖信息的分布等）上存在较大的差异。因此，在进行软件缺陷的相关分析时，如何合理地在不同的软件架构与编码方式下获取信息，以及如何在当前的信息空间中抽取与软件缺陷相关的特征就成为迫切需要解决的问题。

- 软件缺陷定位任务的复杂性

近十年来，软件工程领域的研究取得了许多开创性成果，而与之相配的平台及工具也被也陆续被开发出来。目前，大型的集成开发平台（如 Visual Studio[28]、Eclipse[29]、PyCharm[30] 等）、功能强大的编辑器（如 Sublime[31]、SpaceMacs[32] 等）以及各种第三方插件（如 Resharper[33] 等）都极大地提升了代码的规范化程度、软件的开发效率以及交付产品的可靠性。通常情况下，编码过程中的词法错误及语法错误（例如使用了未定义的变量、语句结尾忘记“;”等）都能够很好地被这些工具发现并准确定位。

然而，最后遗留在程序中的编码错误大多都是语义错误，或是非常复杂的逻辑错误。例如，在某个条件判定语句中将“&&”写成“||”，或者对某个变量进行赋值时与其它变量名混淆，又或者将各对象间的调用关系弄错。这些错误无法通过编译器查出，而开发人员在编码过程中也很可能意识不到它们。若程序在测试或者运行过程中发生失效，调试人员就必须在大量的代码中，通过较为复杂的推理过程来推断这些缺陷代码的位置。例如，程序员将“>”写成“>=”，那么在测试过程中，只有当相关谓词在边界处取值时缺陷才会被触发；而触发之后调试人员需要通过反复的测试，根据测试结果进行推理与修正，并最终锁定缺陷的位置。又如对于将“x%6 == 0”写成“x%3 == 0”这种缺陷，连触发条件都不易掌握，故而在对其进行调试时，调试人员需要分析大量的测试信息，以掌握其失效规律，并根据自己的编程经验去推断可能的缺陷位置。而如何引入相关技术去解析并模拟这一缺陷推理过程并设计能够切实提升缺陷定位效率的方法，就成为需要解决的问题。

本文所涉及的缺陷，都是指语义缺陷。由于软件是一种固化的思维，软件的错误多是思维的错误；反之，所有思维上的错误都有可能会体现在程序代码上。因此软件的缺陷种类也是多样化的。DiGiuseppe 等人[34] 在针对软件缺陷进行研究时讨论了经典的缺陷分类框架，即概念性错误与执行错误分类（Smith92[35]），组件与资源的错误分类（Firesmith92[36]），基于软件的抽象与封装的错误分类（Hayes94[37]），以及基于缺陷位置及其对使用方法与目的的影响等多种因素的多级软件分类（Hayes11[38]）。

Trivedi 等人[39, 40] 根据缺陷的触发、传播及导致失效的条件建立负载与环境依赖缺陷的分类体系，并在此基础上研究缺陷复现的相关问题。

不同种类的缺陷所表现出来的行为也可能不同。例如，条件语句的错误可能触发边界失效。而这种错误也会导致大量的测试用例即使经过了缺陷语句也不会发生失效，我们称这种现象为巧合一致性现象[41]。而对于一些程序主干上的数值计算错误，则可能导致程序的大面积失效。通常情况下对缺陷行为的识别需要借助调试人员丰富的编程经验，以及对目标软件较为深刻的了解。因此如何在自动化或半自动化的方法中识别并应对不同的缺陷行为也是软件缺陷定位研究中需要面对的挑战。

另一个挑战来自于调试信息的限制，这是由于缺陷定位的复杂性以及软件开发环境的多样性共同导致的。缺陷定位是软件调试环节的第一步，其衔接软件测试与缺陷修复两个环节。在软件开发及运维过程中，软件工程中的各个过程是一个有机的整体，任何流程上的改变都会对软件调试环节产生影响。对于软件缺陷定位来说，最主要的影响就是调试输入信息的获取。近些年来，随着软件功能不断增多、复杂度不断提高且更新频率不断加快，敏捷性在软件开发中被越来越多地强调[11]，各种相关的软件工程技术已经被广泛研究与应用，而如何在不断变化的需求中快速更新软件并保证软件的质量需求就成为软件工程研究的热点。例如在持续集成（Continuous Integration, CI）[42] 开发过程中，软件的测试、调试与更新等工作同步交互进行。在该环境下，软件的调试工作不会等待软件测试全部完成，一旦软件发生失效，调试工作就立即开始；而与此同时，测试工作也在并行执行（详见第三章所述）。此时，能够辅助调试的测试信息非常有限，而新的信息仍在动态地生成。在这种动态环境下，调试人员如何对导致程序失效的软件缺陷进行有效地搜索就是非常值得研究的问题。

总的来说，与软件测试相比，软件缺陷定位是更复杂的任务。试想程序员的手工查错过程：选择一些与出错相关的输入来运行程序并观察其输出，通过输出信息以及调试人员自己对于程序缺陷的理解推断缺陷可能的位置；在此基础上，其还会通过运行更多的输入对这一缺陷推断进行细化，并最终将其确定到某一段代码上 [4]。整个定位过程与软件测试、静态分析、软件的需求分析、概要设计以及逻辑推理机制都有着密切的关系；而对于整个过程中的每一个步骤，软件的多样性以及软件开发环境的多样性都会对最终的缺陷定位执行效率产生影响。

（3）软件控制论思想与软件工程的本质相契合，这为软件测试与调试机理的描述，以及基本框架的搭建提供了理论基础。此外，以自动化软件测试与调试技术为基

础的缺陷定位理念、方法与技术已经陆续被提出，这些都是本文的研究基础。

一方面，控制论是探究系统结构、约束及对各种可能性的观测与管理的方法论，它由 Wiener 在 1948 年提出[43]。Wiener 将其定义为“面向动物与机器的通讯与控制的科学研究”。在当代，一切面向目标系统的控制方法与技术都在控制论所讨论的范畴中。而“软件控制论”这一概念最早是由蔡开元教授提出的[44]，其将控制论中的理论与方法系统地引入软件工程领域，为软件工程过程及方法提供了坚实的理论基础 [6]。一般系统论、控制论和信息论被称为系统科学领域的“三论”。而针对软件系统，现有的方法主要是基于信息论，例如，基于统计学的缺陷预测方法[45]、基于信息多样性的软件测试方法[46] 以及基于信息熵的软件缺陷定位方法[47] 等。而软件工程，尤其是软件的测试与调试，在很大程度上可以看作是人对计算机的管理、控制并与之协作的过程，这与控制论的核心理念是契合的。因此，软件控制论思想对软件工程问题的描述以及相关研究框架的搭建有着天然的优势。

软件控制论中的“软件”可以用来广泛地指代软件开发、进化和运维中的各个过程以及所有与之相关的事物[48]；“控制”一词指代作用在被控对象上用来达成某种目的的一系列动作；而包含施加控制的控制器与接受控制的被控对象的系统则被称作“控制系统”。在此基础上，“软件控制”这个概念被解释为“将软件视作被控对象并加以控制，以达到某种目的”。这里软件工程中的一切过程、环节及相关实体都可以作为被控对象，而施加的控制可以是对软件过程参数的鉴别与估计，也可以是对软件本身或是软件工程过程的改变与调整。

图1.1 列举了软件控制论的框架及其在一些领域上的应用。实际上很多软件工程问题都可以被放在软件控制论框架下去研究。这类研究的基本思想是将所考虑的过程转化为一个控系统，明确控制器与受控对象，并且设计合理的控制策略，以优化给定的目标函数。例如，在解决软件测试相关问题时，被测软件就是受控对象而软件测试的策略就是相应的控制器，运行测试后的输出就是系统的观测值[49]。在研究软件设计问题时，软件运行环境就成了被控对象而软件本身则成为控制器，而整个控制任务就是要根据软件在当前环境中的表现来设计软件本身[50]。此外，在解决面向服务类软件的资源分配问题时，被控对象为服务系统，而资源分配策略为控制器，用户的服务质量就是系统的输出，整个控制系统根据当前用户所拥有的服务质量（Quality Of Service，QoS）设计合理的资源分配策略，并最终在资源消耗与服务质量之间取得平衡。最后，老化重生方面的研究面向的是存在软件老化现象的软件系统，旨在设计合理的重生策

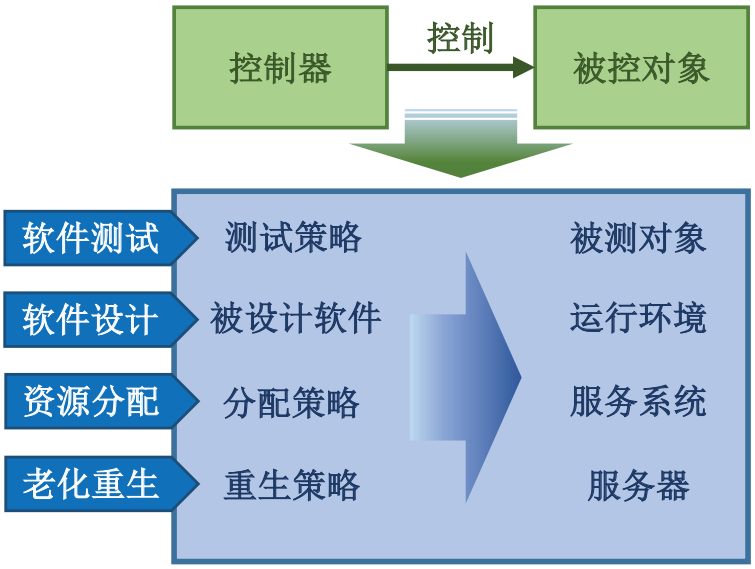


图 1.1　软件控制论及其涉及的研究领域

略在改善软件老化问题的同时，保障系统的利用效率[51]。

对于软件缺陷定位来说，软件同样是被控对象。软件的多样性以及软件开发环境的多样性均可以理解为被控对象的多样性。因此，通过软件控制论来搭建软件缺陷定位的研究框架，并且在该框架下进一步开展讨论以提升缺陷定位效率就成为非常可行的方案了。

另一方面，软件缺陷定位在近些年来作为研究领域的热点问题而被学者们广泛关注。目前很多面向软件调试及软件缺陷定位的方法与技术相继被提出来（具体见第1.3.1节中对国内外研究工作的阐述）。这些研究工作从多个方面提升了软件缺陷定位的效率。此外，很多相关的工具与实验包都被开发出来。这些方法与技术为本文的研究工作提供了较为全面的方法论基础，本文的很多方法都是在已有方法的基础上加以改进得来的。

然而，大多数现有的研究工作都是针对具体问题，软件缺陷定位缺少相应的研究框架，针对软件多样性及软件开发环境多样性的研究没有系统地开展，相关技术的普适性以及潜在的基本假设也不易被讨论。因此，尽管本文的研究工作基于现有工作展开，但是本文对于软件缺陷定位的适应性进行了系统性的讨论，原创性地提出了相应的研究框架以及该框架下的方法与技术。相信本文的工作对软件缺陷定位理论体系的搭建，缺陷定位方法的可靠性及严谨性，以及软件缺陷定位效率的切实提升能够起到较大的促进作用。

以上关于软件缺陷定位任务、其在软件工程中所处的地位及当前所面临挑战的概述为本论文的研究提供了方向。本文针对自适应软件缺陷定位的理论与方法展开研究，试图通过在缺陷定位的各个环节合理地引入自适应机制，来切实提升软件缺陷定位方法对软件多样性及开发环境多样性的适应性，从而提升缺陷定位的准确度以及其在整个软件工程过程中的执行效率。下面从概念、理论和方法等不同角度具体阐述论文的选题依据，其总体结构如图1.2 所示。

（1）面向软件多样性与软件开发环境多样性的缺陷定位技术研究是软件缺陷定位实现自动化的必要途径，对整个软件工程领域的方法整合与运筹管理具有重要的价值。

如何切实提升软件缺陷定位的效率是缺陷定位研究的根本目标。如前文所述，软件缺陷定位是软件开发与运维过程中非常重要且消耗大量资源的环节。因此，随着软件可靠性以及软件开发敏捷性需求的日益增加，软件缺陷定位也逐渐成为非常艰巨的任务，而如何提升软件缺陷定位的效率也已经成为非常具有研究价值的问题了。软件缺陷定位即指根据现有的测试信息推断软件缺陷位置的推理过程，因此理解软件缺陷的机理与测试信息的特征关系并以此设计自动化或半自动化的软件缺陷定位方法，从而在本质上提升软件缺陷定位的效率就成为首要的研究点。此外，程序代码组织的多样性、软件缺陷的多样性，测试信息的多样性，缺陷定位任务流程的多样性这些都可以归结为软件与软件开发环境多样性，它是软件缺陷定位自动化过程中所面临的挑战，也是本论文研究的侧重点。

（2）适应性是自动化软件缺陷定位方法研究的重要需求，也是突破软件缺陷定位方法研究瓶颈的关键。

软件缺陷定位相关的多样性问题是本文研究的侧重点，而适应性就是本文的主要研究方向。从字面意义上讲，适应性表示目标事物与周围环境的相合能力。本论文在讨论软件缺陷定位的适应性时采用更广义的解释。某个缺陷定位的方法的适应性指的是该方法在处理多样性问题时相应机制的完善程度，其既可以是该方法在不同的软件及软件开发环境下保持较高缺陷定位效率的能力，也可以是度量与预测该方法本身在各种条件下执行效率的能力。在实际软件开发过程中，如果一种缺陷定位方法能够取得工业界的信赖并带来收益，那么该方法应该能够在相对多样的环境下都能有较好的表现，这样才值得将相应的预算分配在与该方法相关的工具平台的引入与搭建以及相

关人才的培训上。此外，软件缺陷定位方法还可以具备相应的适应性预测与度量机制，使调试人员能够明确其具体的适用条件，以控制使用该方法所带来的风险。如此一来，缺陷定位方法就能够被应用到正确的场合，其价值也得以体现。因此，本文以提升适应性为目标，从提出具备较好普适性的软件缺陷定位方法，以及提出合理的机制与方法来度量相应软件缺陷定位方法的适用条件这两个角度展开研究。

（3）由于软件测试与调试贯穿整个软件生命周期，而缺陷定位是连接软件测试与软件调试的桥梁，因此对于软件缺陷定位适应性的讨论需要考虑到与缺陷定位相关的多个任务，并设计相应的自适应机制。

整个软件开发过程是一个有机的整体。尽管软件缺陷定位任务本身的目的是要找到程序中导致软件失效的代码，但其与软件工程过程中的很多环节都存在联系。首先软件缺陷定位的输入信息需要通过软件测试过程得到。此外，软件缺陷定位的输出也要作为程序员进行错误修正的参考。从广义上讲，在所有与软件缺陷定位相关的各种流程中，凡是以软件缺陷定位为主要目标而进行的活动（如面向缺陷定位的测试用例运行、代码走查、输入空间划分等）都属于软件缺陷定位这个研究领域所涉及的范畴。因此，欲使软件缺陷定位环节在软件开发过程中得到真正的优化，广泛研究以缺陷定位为目标的各个过程的运筹调度及资源优化问题是非常有价值的。

（4） 若要从根本上解决软件缺陷定位方法的适应性问题，建立相应的研究框架并且在此基础上完善软件缺陷定位的研究体系非常重要。

目前针对软件缺陷定位的研究中，方法偏多而理论偏少。Wang 等人[52] 调研了 2016 年以前与软件缺陷定位相关的研究工作，发现其中关于方法的研究远多于理论研究工作，且现有的理论工作大多集中于某个技术细节，亦或是研究问题的理论推导。而一个研究领域中的方法与技术若能够取得很好的应用，很可能得益于该领域拥有一套完整的理论体系。在一套理论体系的指引下，我们可以更好地对各方法的适应性进行分析，并在此基础上进行现有方法的改进与新方法的设计。因此，关于软件缺陷定位的研究体系本身的研究也是非常有价值的。

总结前面的论述，本文的研究背景与选题依据如图1.2 所示。作为本论文的应用背景，计算机与软件工程技术得到了广泛的研究与应用，为本论文的研究工作提供了较为丰富的理论与方法基础、工具平台与实验对象。而另一方面，科学技术的发展使得

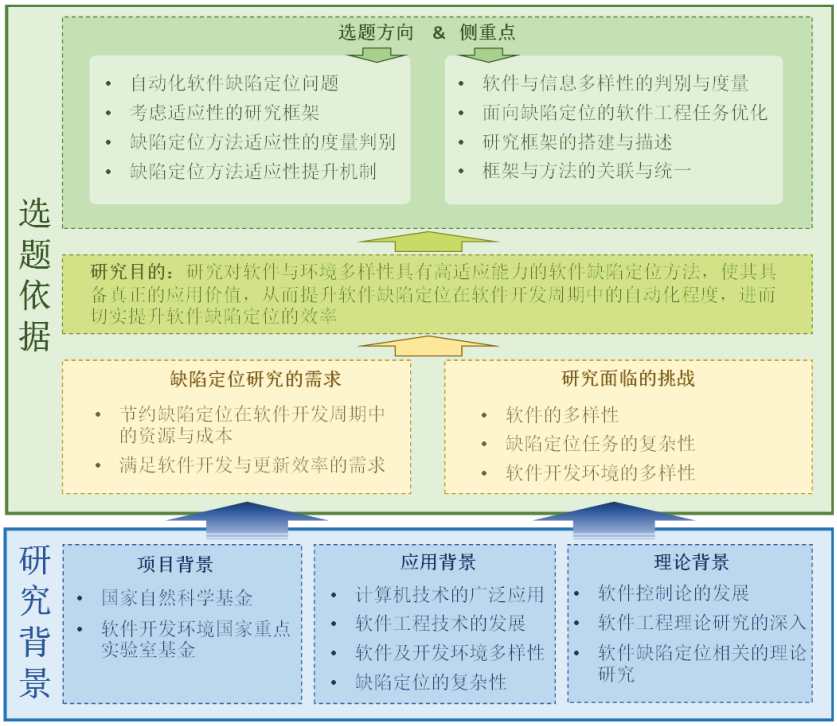


图 1.2　论文研究背景与选题依据示意图

人们对软件可靠性与软件开发效率有了更高的需求；然而与之矛盾的是，当今的软件种类繁多，软件内部的逻辑关系非常复杂，使得程序员在编码时的出错机会增大。当软件失效后，人们需要对软件进行调试，以消除导致软件失效的缺陷代码，从而提升软件的可靠性。然而，软件的多样性及开发环境的多样性使得调试人员在调试时获取的信息空间、信息量以及信息分布都是多样的。此外，软件缺陷定位本身就个非常复杂且艰巨的任务。由此可见，软件缺陷定位既是研究热点，也是非常有价值的研究领域。

作为本论文的理论背景，软件控制论在上世纪九十年代被提出并且在近十年取得了非常大的发展。软件控制论试图从人与机器的通信、交互以及人对机器的操作角度为软件工程过程进行建模。由于软件调试与测试过程就是一种人对计算机软件的观测与操作行为，因此控制论思想就为软件调试与测试环节的描述与改善提供了理论基础。

从研究背景可以引出本文的研动机以及需要面对的挑战。首先，切实提升软件缺陷定位执行效率使其适应软件工程的需求，就成为本文主要的研究动机。具体来说，我们旨在节约软件缺陷定位的资源消耗以及缩短缺陷定位所需要的时间，使其更好地满足软件的质量及更新速度的需求。

尽管目前有很多关于软件缺陷定位的研究，然而软件本身的多样性、缺陷的多样

性、信息的多样性以及软件缺陷定位任务的复杂性使得目前的软件缺陷定位方法仍未在工程中被普遍使用并带来很大的收益，软件调试过程冗长且单调乏味的现状并没有的到实质性改善。因此，我们希望研究对软件与环境多样性具有高适应能力的软件缺陷定位方法，使其具备真正的应用价值，从而提升软件缺陷定位在软件开发周期中的自动化程度，进而切实提升软件缺陷定位的效率。

本文的选题方向与研究侧重点主要体现在以下几个方面。首先初步建立考虑适应性的软件缺陷定位框架，并在其中讨论适应性的提升方法与相关判别机制，明晰自适应缺陷定位的概念及研究内容。此外，针对与缺陷定位相关的多个过程，提出具体的自适应方法以及适应性的度量与判别机制，以更好地解决软件缺陷定位方法的适应性问题，从而切实提升软件缺陷定位方法的可信度与应用价值。在研究中，我们着眼于缺陷定位过程中信息空间以及信息多样性的判别与度量，面向缺陷定位的软件工程任务优化，自适应软件缺陷定位研究框架的搭建与描述，以及理论与方法的关联与统一。当然，软件工程领域涉及的活动种类繁多，本文不可能将缺陷定位的各个子环节及其相关活动都涉及到，但我们试图通过对典型适应性问题的讨论，以点带面，细化自适应软件缺陷定位的研究框架，为该方向的后续研究打下基础。

1.1.2 软件缺陷定位相关的重要概念

本小节给出本文针对软件缺陷定位的研究中涉及到的若干重要概念。

（1）软件

计算机软件（简称软件），是由数据和计算机指令所组成的计算机系统中与物理硬件相对应的部分。在计算机科学中软件指代所有由计算机系统、程序与数据处理的信息，这包括计算机程序、库以及与之相关的不可执行数据（如文档、多媒体数据）。在计算机系统中，软件没有实体，它与硬件相互依存。本文认为，软件是通过计算机语言写成的人类思维的具象化，而软件所描述的抽象思维通过硬件来实现。

（2）软件工程

在 IEEE 软件工程标准中，软件工程被定义为“关于软件的开发与运维的系统的、标准的、可量化的方法的运用范例[53]”，其具体含义可以分为两个方面解释。首先，将软件视作产品，从工程的角度对其开发及运维过程进行应用与管理，使其标准化、工程化，以提高软件的可靠性并缩短开发周期。其次，软件工程也作为一个研究学科[20]，旨在构建和维护有效、实用以及高质量的软件所需要的工程化方法及相关理论建模方法，其涉及程序设计语言、数据库、软件开发工具、系统平台、标准、设计模式等研

究领域。

（3）软件生命周期

软件生命周期指的是软件计划、开发、测试、运维、升级直到报废的整个过程。具体来说，周期内包含问题定义、可行性分析、总体描述、系统设计、编码、调试和测试、验收与运行、维护升级到废弃等阶段。为了在工程中确立软件开发和演化过程中各个阶段的次序限制以及各个阶段活动的准则，在工程设计阶段通常会建立相应的软件生命周期模型。软件生命周期模型描述软件开发过程中各项活动如何执行，常用的有瀑布模型、增量模型以及螺旋模型等。

在瀑布模型中，软件开发过程中的各项活动会严格按照线性方式进行，当前阶段的活动接受上一阶段活动的工作结果，完成所需的工作内容。增量模型则将目标项目模块化，把每个模块视为一个增量组件，从而分批次地分析、设计、编码和测试这些增量组件。运用增量模型的软件开发过程是递增式的过程。而螺旋模型则根据系统包含的风险看待软件开发过程，其将开发活动和风险管理结合起来，以更好地控制风险。

（4）软件测试

软件测试是指通过人工手段或自动化工具来运行某个软件，并将实际输出进行比较或分析的过程。软件测试是软件质量保障中一个重要的技术，并且是软件开发过程中必不可少的环节。通过测试能够暴露被测软件中存在的缺陷，衡量软件可靠性，此外还能评估软件的设计合理性[54]。

由于测试空间无法穷尽，并且无法证明软件的正确性，软件测试本身就是一个需要消耗大量资源的任务。随着软件规模及复杂度的提高，对测试效率的要求也越来越高，软件测试与整个软件开发过程也越来越不可分割。如果软件测试以文档审查或需求审查的形式越早地参与到软件的生命周期中，就能减少越多的开发和维护成本。若将软件测试与软件质量保障相结合，发掘更多潜在的缺陷特征，并以此来改进整个软件的开发过程，就能在保障可靠性的同时，提升软件开发效率。

（5）软件调试 调试是测试之后的活动。当软件测试过程中发现待测软件中存在缺陷时，排除这些缺陷以提升软件的质量就是非常重要的；而软件调试的根本目的就是诊断和改正程序中潜在的缺陷。具体来说，软件调试环节可分为三个子环节[4]：1）缺陷定位，即在程序中找到导致软件失效的缺陷代码；2）缺陷理解，即探索缺陷的触发机理，理解该缺陷是如何导致软件失效的；3）缺陷修复，即通过修改源代码或是添加异常处理机制来排除由目标缺陷导致的软件失效。

（6）软件缺陷定位

软件缺陷定位是指当软件发生失效时，调试人员通过分析程序源代码，找到引发该失效的缺陷代码的行为[55, 56]。软件缺陷定位是软件调试的第一步，也是至关重要的一个步骤。如果我们能够准确找到导致软件失效的错误代码位置，那么对于缺陷触发机理的理解以及缺陷的修复都会变得容易；反之，如果缺陷的位置找不到，那么之后的调试步骤都无法正常进行。此外，对于规模较大，代码行数较多的软件，准确地找到缺陷代码的位置就成为一个非常困难的个过程。

1.2 自适应软件缺陷定位

本节通过分析软件缺陷定位任务的性能指标和自适应需求，明确给出自适应软件缺陷定位的含义，并阐明本文所研究的理论与技术范围——自适应软件缺陷定位的研究框架，以及在此基础上针对软件及开发环境多样性的自适应机制。

1.2.1 软件缺陷定位的综合需求

软件缺陷定位是指根据当前的测试信息去推断程序中导致软件失效的缺陷代码位置的过程，其性能同时受内部及外部两方面因素影响。其中，内部因素主要涉及软件缺陷定位方法的合理性与可行性；而外部因素则主要涉及软件缺陷定位输入信息（即测试信息）的信息量以及信息质量。在整个软件开发环境中，软件缺陷定位方法需要讨论以下性能指标：

（1）准确性

通常情况下，软件缺陷定位方法的输出结果是对软件缺陷位置的一个预测，那么其准确性就是指该预测对实际缺陷位置的指示能力。

（2）响应速率

若将软件缺陷定位过程放在整个开发环境中考虑，在最开始测试信息不是很充足，而软件缺陷定位方法的准确性也会较差。随着测试过程的进行，可利用的测试信息会逐渐增多，而软件缺陷定位的响应速率就是指当测试信息增加时，软件缺陷定位结果趋于准确的速率。

从上述讨论可知，软件缺陷定位的性能与方法本身以及测试信息都有关系。欲使软件缺陷定方法能够得到切实的应用，那么其必须满足不同方面的性能需求，而影响这些性能的各个因素也都需要考虑。

1.2.2　自适应软件缺陷定位的含义

在第1.1.1节的讨论中提到，软件缺陷定位方法所面临的一大挑战就是其对软件多样性及环境多样性的适应性。而本文中自适应软件缺陷定位的含义就是根据缺陷定位执行时的当前环境（包括目标程序的特征、当前的测试信息以及缺陷定位的中间结果）动态调整缺陷定位相关的流程、方法及参数（包括考虑软件缺陷定位效率的测试策略、测试信息的处理以及缺陷定位方法参数的调整），以提升整个缺陷定位过程对目标软件与开发环境的适应性，并最终满足缺陷定位环节的性能需求。具体来说，自适应软件缺陷定位包括以下内容：

（1）缺陷定位环境的实时辨识

软件缺陷定位存在适应性问题的根本原因是软件以及缺陷定位环境的多样性。由于目标程序、缺陷、以及软件开发环境多种多样，我们很难找到一种放之四海皆准的缺陷定位方法及流程。因此，欲使缺陷定位方法实现缺陷定位过程的自动化并最终创造价值，就必须在特定的环境下采用与之相适应的缺陷定位方法与流程。然而实际情况中，当前缺陷定位环境通常是未知的，这导致我们不知道当前的环境与所采用的缺陷定位方法是否合适，进而也就无法对其进行合理的改善。而这种对环境因素的未知性，是限制自动化软件缺陷定位在实践中得到普遍应用的根本原因。因此，对目标软件及环境中的未知信息进行观测及辨识，就是实现自适应软件缺陷定位所需要考虑的第一个关键问题。

在辨识过程中，首先需要对缺陷定位的环境进行观测，并从观测的结果中挖掘能够准确反映当前环境状态及参数的信息。其次，需要根据这些观测信息实时监测测缺陷定位过程中当前的环境状态，并将其在与缺陷定位性能指标相关的特征空间中进行量化。在软件缺陷定位的执行过程中，随着观测、采样及分析的持续进行，对当前内部与外部环境因素的量化与表征也会更加准确。随着“未知”逐渐变为“已知”，我们就可以对当前软件缺陷定位方法的适应性进行合理准确的评估，并对缺陷定位过程及方法本身进行调整。

（2）缺陷定方法及相关过程的适应性评估

当影响软件缺陷定位的环境信息被较为精确地掌握且表达后，接下来要做的就是分析这些环境因素对缺陷定位的具体影响，发现其与缺陷定位性能指标的内在联系，并在此基础上，评估所采用的自动化软件缺陷定位方法对当前环境的适应性。在评估

过程中，需要对所关注的软件缺陷定位性能指标，以及其与各环境因素间的联系进行详细分析，并建立合理的评估及度量模型。接着，我们从环境信息库中抽取与该性能指标相关的信息，并根据该指标的评估模型进行适应性计算。有了适应性度量做支撑，在使用自动化缺陷定位方法时的可信度就会提升，风险就会减小；换言之，即使没有下文中所讨论的具体自适应机制，若我们能够通过自动辨识掌握缺陷定位方法与流程在当前场景下是否适用，这个方法本身就能够获得更好的应用了。因此，对于适应性的辨识本身就是解决适应性问题时必要的自适应机制。

（3）提升软件缺陷定位环节相关方法及过程适应性的具体自适应机制

在适应性评估的基础上，自适应缺陷定位将会讨论引入具体的自适应机制对当前环境下软件缺陷环节中的各个过程进行调整及改善，以切实提升自动化软件缺陷定位方法对软件多样性及软件开发环境多样性的适应性。

适应性的评估与提升能够使现有的软件缺陷定位方法及技术得到更好的应用，并切实带来价值。首先，根据适应性评估结果，若当采用的方法及相关配置适用于当前的软件缺陷定位环境，可以通知调试人员，以提升自动化缺陷定位方法给出缺陷位置预测的参考价值；相反，如果缺陷定位方法对当前环境因素的适应性程度较差，同样可以通知调试人员放弃对该方法的使用，以避免其提供的误导性信息产生多余的代价。此外，一套完善的自适应机制通过对自动化软件缺陷定位的相关配置参数进行动态调整，使其适应当前环境，从而切实提升了软件缺陷定位方法的可用性。

1.2.3 自适应软件缺陷定位的关键研究问题

研究自适应软件缺陷定位时需要重点考虑以下关键问题。

（1）缺陷定位相关信息空间的建模

由于多样性是软件缺陷定位适应性问题产生的根源，因此对于多样性的建模与分析就是必须要讨论的问题。正如第1.1.1节所述，软件开发环境中不同的软件需求、程序输入空间、开发者、设计者、软件结构与开发环境及流程都会涉及不同的论域。因此，如何分析各种信息之间的关系，构建相应的特征空间，并对该空间内的信息样本进行量化就成为需要解决的关键问题之一。

（2）软件缺陷定位各个子环节适应性的度量以及解决方案

软件缺陷定位是一个复杂的任务，很多与其相关的软件工程过程都会对实际缺陷定位的效率产生影响。因此，自适应机制也需要考虑与缺陷定位相关的各个过程。首先自动化软件缺陷定位的核心算法模块需要引入自适应机制，该自适应机制需要对当前的环境参数进行估计与辨识，并以此为基础调整缺陷定位方法本身，使之更好地适应当前的缺陷定位环境，并最终给出合理的缺陷位置预测结果。其次，对于软件缺陷定位来说，测试信息是其必要输入，其质量直接影响到最终缺陷位置预测的准确度。因此，在测试环节就引入自适应机制也是很有必要的。

（3）相关研究框架的搭建

若要更好地对适应性问题展开研究，一个较为完整的研究框架是必要的。针对缺陷定位相关过程中的各个子环节以及各种内部与外部的环境因素，我们能够提出多种自适应机制。而一套完整的研究框架不仅便于分析各自适应机制的使用条件及相互关系，还能够为这些自适应机制的整合提供思路与指引。

1.3　国内外研究现状

本节简述自动化软件缺陷定位的研究与发展状况，并对目前较为流行的缺陷定位方法的基本思想与执行过程进行介绍。最后，对现有方法中存在的自适应思想进行描述与分析。这些思想对本文工作的提出与开展起着铺垫作用。

1.3.1　自动化软件缺陷定位方法

上世纪 70 年代末，Mark Weiser[57] 提出程序切片的概念，标志着自动化软件缺陷定位研究的开始。目前的自动化软件缺陷定位技术大多为半自动化的，它们并不直接给出缺陷语句的位置，而是提供一个程序成分（如语句、函数等）的子集，而缺陷语句很可能或者一定包含在这个子集中。具体来说，现有的自动化缺陷定位方法大致可以分为四大类：基于程序切片的软件缺陷定位方法、基于模型的软件缺陷定位方法、基于程序谱的缺陷定位方法以及基于程序变异的软件缺陷定位方法。这些方法之间也会有相互的交叉与融合。

（1）基于切片的软件缺陷定位方法

程序切片最早由 Weiser 提出[57]。假设程序员对程序中某个特定的输出变量感兴趣，则其可以利用数据之间的依赖关系从程序中抽取出一个语句子集，称之为程序切片。切片中的语句均与这个特定的输出变量有关，而切片外的语句则与这个输出变量

```
1. int r = 3;
2. float area = r*3.141f;
3. float circ = 2.f*r*3.141;
```

图 1.3　基于模型的软件缺陷定位方法示例

无关。这样程序员在进行代码分析及调试工作时就可以只关注程序切片而不是整个程序。最早的程序切片是静态切片，然而在很多情况下静态切片集合依然包含过多的元素，这使得缺陷语句不易被锁定。之后，动态切片被引入[58, 59]，其仅抽出程序在执行过程中与特定输出相关的语句，规模通常要比静态切片小得多，而这在很大程度上提升了程序切片在缺陷定位过程中的参考意义。目前为止，研究者出于不同目的，引入了各种切片技术，例如对象切片[60]、超谱等[61]。由于程序切片的固有特性，很容易想到从其中提取与程序错误输出相关的代码。目前将程序切片应用在程序调试中的相关研究很多[58, 59, 62]。张等人[60] 通过实验证实了程序切片可以有效地提供与程序错误相关的语句集。基于切片的软件缺陷定位方法具有很高的确定性与可靠性，几乎可以保证缺陷语句一定存在于所提取出的程序切片内（这里不考虑漏写语句的情况），然而其缺点是精度比较低，很多情况下，即使是动态切片，其包含的代码量依然很多，而对于同一切片内的语句，它们包含缺陷可能性无法区别，因此其有可能无法在程序员纠错过程中提供有效的指导。

（2）基于模型的软件缺陷定位方法

基于模型的故障诊断方法首先由 Davis 提出[63]，并由 Reiter[64] 完善。而基于模型的程序调试方法（Model-based software debugging, MBSD）于 1993 年被 Console 等人[65] 提出。该方法的基本思想是通过程序的逻辑推理模型来推断出缺陷语句的位置[66]。具体来说，将程序运行时观测到的失效行为通过事先建立的故障诊断模型来解释，将最合理的解释所对应的故障组件视作程序中的缺陷，即做出“若假设该组件存在问题，那么所有现象就解释的通了”这样的推断。例如，图1.3 中的求圆形面积的程序[69]。

该程序的故障诊断模型可以表示为以下规则：

规则 1：若语句 1 正确 $\rightarrow r$ 正确；

规则 2：若 r 正确 $\wedge$ 语句 2 正确 $\rightarrow area$ 正确；

规则 3：若 r 正确 $\wedge$ 语句 3 正确 $\rightarrow circ$ 正确。

若实际观测中发现输出 r 正确，$area$ 正确，而 $circ$ 错误，则根据上述推理模型，最终得到的故障诊断结果为语句 3 错误。目前很多研究都从理论或技术上扩展了 MBSD；例如，基于不同的程序语言（如 VHDL[67]、JAVA[68]）的 MBSD，利用外部信息（例如输入区域划分）来促进 MBSD[69, 70]，以及引入各种程序模型等[71]。

（3）基于程序谱的软件缺陷定位技术

Jones 等人[72] 首先提出了利用测试程序谱进行缺陷定位（SBFL，Spectrum-Based Fault Localization）的思想。这里，程序谱指的是程序运行过程中执行特征的统计信息，其最早在调试解决千年虫[73] 问题时被提出。程序谱方法通常假设错误语句在程序运行成功与失效时的行为特征存在较大的差异，而种差异能够在程序谱中被反映出来。在实际执行时，首先收集测试用例的执行信息，并以此计算各语句的程序谱信息。不同的 SBFL 技术所用到的程序谱信息是不同的，有些用到各个谓词在成功（或失效）测试用例中为真（或为假）的次数，而有些则用到各语句在成功（或失效）测试用例中被覆盖的次数。随后，SBFL 方法根据各个语句的程序谱信息计算其包含缺陷的可疑度，并将所有语句按照它们的可疑度进行排序，可疑度高的语句被排在前面，而可疑度低的语句被排在后面。当语句被排好顺序后，调试人员按照顺序进行排查。如果程序谱方法足够合理，那么缺陷语句将被排在靠前的位置，而调试人员也能够很快地找到它。尽管上文中都是在使用“缺陷语句”一词，实际上序谱方法的定位粒度可以是多种多样的，例如语句[2, 72, 74, 75, 76, 77]、谓词[78, 79, 80, 56, 81, 82]、分支[83] 以及机器码[84]，当然也有整合不同粒度的程序成分进行缺陷定位的研究[85]。程序谱方法的核心是可疑度计算公式[86, 87]，即如何根据程序谱信息计算各语句的可疑度。不同的可疑度计算公式依赖于不同程度的启发式信息。目前的可疑度计算公式有上百个，Naish 等人[75] 首次统筹了主要的 30 个可疑度计算公式，并把他们划分成等价公式类。Xie[2] 等人在单缺陷假设（即假设程序只包含一条缺陷语句）下证明了 Op 和 Wong1 公式为 Maximal 公式。而对于多缺陷的情况，则很难给出确定性的理论证明。由于程序谱缺陷定位方法是目前使用最为普遍且具有良好实用性的方法，因此本文将其作为的基本方法，并在后面的章节中做详细介绍。

（4）基于程序变异的软件缺陷定位方法

近些年，一种新的软件缺陷定位方法——基于变异的软件缺陷定位方法——被提

出。该方法通过人为改变程序中的某些语句，来获得更多程序失效时的行为信息，并以此来推断被变异的语句中哪些是真正的缺陷语句。例如，Debroy[88] 等人试图通过手动植入变异来修正程序中的错误。在他们的实验中，20.70% 的错误得到了修正。尽管效果一般，但是软件测试调试自动化的最终目标就是缺陷的自动修复，他们的工作向这个目标迈出了重要一步。Moon 等人[89] 提出了基于程序变异的缺陷定位方法（MUtation-baSEd fault localization，MUSE）。MUSE 的基本思想是，当我们对程序进行修改时，修改原本就存在错误的语句对程序行为的影响不会很大，而修改原本正确的语句将会在很大程度上改变程序的行为。因此，如果修改某条语句后，程序的行为不会发生变化，或者成功的测试用例反而增加，则该语句就很可能是原本的缺陷语句；相反，如果修改某条语句后，失效测试用例明显增多，则原本该语句很可能不存在错误。

（5）多种方法融合的缺陷定位策略

上述缺陷定位方法各有优劣。例如切片方法具有很高的确定性，但其定位精度较低；而程序谱方法的定位精度高，但其结果中包含较多的启发式信息。为了取各方法之长处，一些学者在研究中尝试结合多种方法进行缺陷定位。例如，Abreu 等人[66] 将模型方法与程序谱方法结合，提高了程序谱方法对于多缺陷程序的定位效果；文献[63, 90] 将程序谱方法同切片方法进行结合，希望能够同时满足准确度和精度上的要求。Moon 等人[89, 91] 在提出变异方法时也将程序谱方法作为辅助工具。

1.3.2 目前存在的面向缺陷定位的自适应机制

第1.2.3节中提到，软件缺陷定位的各个部分都有必要引入自适应机制。目前的研究中，在测试用例选取与缺陷定位方法上都存在符合自适应基本理念的机制及策略。

（1）面向缺陷定位的测试策略

面向缺陷定位的软件测试技术主要涉及如何生成与选取测试用例使之更加适应缺陷定位的需求，其具体的优化目标有很多，例如减少测试用例的执行数量、减少测试人员检查测试结果的工作量或是使缺陷定位更准确等。Yoo 等人[47] 提出了面向缺陷定位的测试用例选取技术，旨在减少测试用例的执行代价。该技术将执行测试用例与缺陷定位并行进行，利用信息论中信息熵的概念评估当前缺陷定位方法的不确定性，并以此为依据选取接下来要执行的测试用例。Yoo 等人的方法利用当前缺陷定位的结果对各个测试用例的价值进行评估，符合自适应机制的基本特征。然而，他们的

方法用到了测试用例的覆盖信息，这在实际软件测试中往往不易获取。此外，将信息熵引入软件缺陷定位时，本身也会产生一些问题。Compos 等人[7] 同样用熵的方法讨论了测试用例的生成问题。在软件缺陷定位的程序谱方法中，如何解决巧合一致性（coincidental correctness）问题一直是一个难题，这里巧合一致性是指某些测试用例虽然覆盖了缺陷语句却没有产生失效的现象。巧合一致性的成例很大程度上干扰了程序谱方法的预测结果。Masri 等人[92, 41] 通过聚类方法从某种程度上滤除了那些干扰测试用例，不过他们的方法并不是自适应的。针对此问题，Bandyopadhyay 等人[93] 提出了一个基于反馈的方法，他们根据调试人员的实际调试情况，实时改变当前的缺陷预测结果。不过他们的方法仅仅考虑了当可疑度最大的语句不包含缺陷时的调整策略，在严格意义上也不是自适应的。

（2）缺陷定位方法的适应性度量及改善

相对于测试过程，关于缺陷定位方法本身自适应机制的研究目前并不是很多。2002 年，Zeller 等人[94, 95, 96, 97] 提出了经典的 Delta Debugging 方法，通过找到导致程序失效的临界输入来搜索失效的原因，进而找出导致程序失效的代码。该方法经常需要以静态分析及对目标程序的理解做为基础。

对于程序谱方法，Li 等人[98] 提出了根据程序各历史版本的调试数据来选择可疑度计算公式的方法。Le 等人[99] 统计历史版本的调试信息，并将其作为输入提交给支持向量机进行训练，并利用训练后的预测模型评估当前版本得到的缺陷预测的准确性。同样，Daniel 等人[100] 根据当前的测试信息及程序语句间的关系，对程序谱方法的缺陷位置预测结果进行了评分。尽管评分本身并不能实质性地提升缺陷定位效率，但其完全可以看成是包含自适应思想的，因为如果在调试人员手动介入之前就能够对当前的缺陷定位方法所提供的预测作出较为准确的评估，那么我们就有足够的时间对当前的定位方法进行调整，或是抛弃当前的缺陷位置预测结果。Lo 等人[101] 针对现有的缺陷定位方法性能进行了评估，并对现有的可疑度计算公式进行了整合。

1.3.3　现状分析与问题总结

近年来，具备自适应思想的软件缺陷定位技术相继被提出。然而，目前大多数工作只是在起步阶段，所采用的研究框架并不完善，很多重要问题也没有被详细讨论。因此自适应软件缺陷定位理论与技术的研究还有广阔的发展空间。具体可以从以下几个方面体现：

（1）现有的针对测试过程的自适应机制，包括测试用例选择/生成与测试用例优化两个方面。前者旨在不影响缺陷定位效果的前提下减少测试用例的执行代价，而后者旨在滤除或处理那些产生不好影响的测试用例，从而提升整个测试用例集的性能，当然这两方面并不是完全分开的。在测试用例的选择生成问题上，目前的白盒方法较多，而黑盒方法也开始被提出。另一方面，对于测试用例优化问题，目前大多数方法旨在减少巧合一致性现象所带来的负面影响，因为巧合一致性确实是影响缺陷定位准确度的一个非常重要的因素。以此为目的的相关方法在今后仍会持续被提出与扩展。然而，尽管在软件测试领域关于测试策略的研究已经比较成熟，但是面向软件缺陷定位的测试用例选择/生成与过滤的研究工作目前仍比较少，有待进一步研究。本文也将在此方面对原有的方法进行改进，并对新问题进行探索。

（2）关于面向缺陷定位方法自适应机制的研究在近几年开始出现，不过尚处于起步阶段，大多数工作集中在对缺陷预测准确度的预先评价或是对现有方法进行融合上，而满足自适应条件的闭环策略则非常少，体系化的研究框架还有待进一步探索。本文将针对上述问题对软件缺陷定位方法本身进行自适应机制的研究与探索。

总的来说，到目前为止已经有很多学者针对缺陷定位的自适应策略展开研究，但是由于软件缺陷定位任务的复杂性，很多比较重要的问题并没有被考虑到或是很好地得到解决。此外自适应软件缺陷定位还缺少整体的研究框架。

1.4 主要研究内容

本论文致力于自适应软件缺陷定位研究框架的提出与完善。根据本章对研究背景、选题依据以及自适应软件缺陷定位的讨论，自适应软件缺陷定位的理论以及面向与缺陷定位环节相关的各个过程的自适应机制都是值得研究的。本文的研究框架如图1.4 所示。

本文关于自适应软件缺陷定位的研究可分为以下两个部分：

（1）研究框架的搭建及形式化表述，旨在从理论方面论证自适应软件缺陷定位的合理性与可行性，其中包括：

- 自适应软件缺陷定位含义的详细描述，旨在借助软件控制论思想对自适应的概念进行形式化表达，以对其进行精确定义，并以此为基点进行关于引入自适应机制来提升软件缺陷定位方法性能及实际执行效率的一系列讨论。

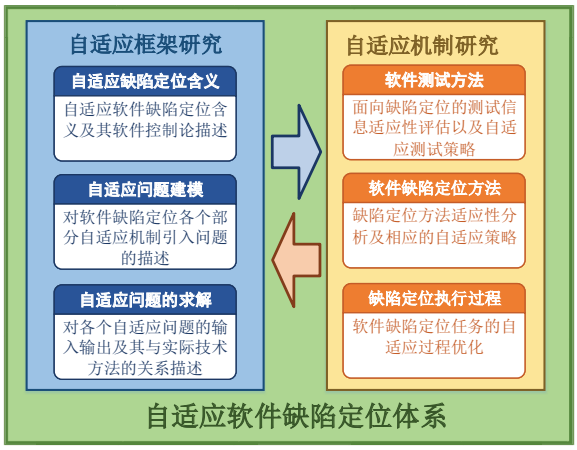


图 1.4　自适应软件缺陷定位研究体系示意图

- 自适应软件缺陷定位相关问题的建模，旨在规范化描述软件缺陷定位相关各个过程的自适应机制。具体来说，在上述对自适应软件缺陷定位含义的描述基础上，对整个缺陷定位流程中各个部分的相关参数及运行机制进行描述，以明确各自适应机制所涉及的对象、参数及相关假设和约束。

（2）具体自适应机制研究，主要讨论实际缺陷定位问题中自适应机制的设计及应用，旨在通过在各缺陷定位的关键问题上引入自适应机制并设计相应的自适应策略，全方位提升软件缺陷定位的性能与执行效率。本文拟从两个方面对自适应机制的引入进行讨论，其中包括：

- 面向缺陷定位的测试信息适应性评估以及自适应测试策略。软件测试是软件缺陷定位的前序过程，而测试过程中够得到的测试信息是软件缺陷定位的输入，测试信息的质量是保障缺陷定位性能的基础。对于不同的环境，测试信息的获取过程、表达方式以及特征分布都会呈现截然不同的性质，而本论文将引入多种自适应机制，以在不同的开发环境想定下提升缺陷定位方法对测试信息的适应性。

- 软件缺陷定位方法的适应性分析及相应的适应性提升策略。软件缺陷定位方法是自动化软件缺陷定位环节的核心部分，其接受软件测试过程中得到的测试信息（例如程序执行路径、输出结果等），并根据这些信息通过相应算法来推理出软件缺陷的位置。尽管前面论述了测试信息的分布差异会受到测试过程的影响，然而软件及缺陷的多样性会从根本上导致输入测试信息具备不同的特征分布。因此，

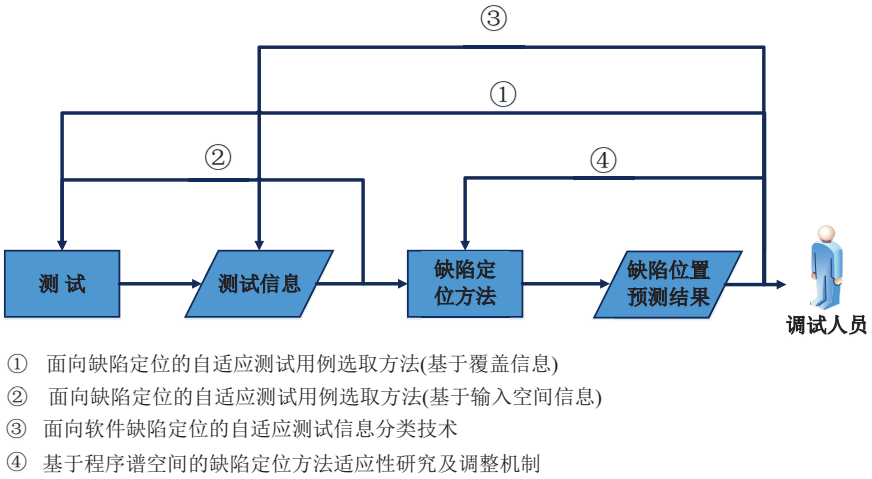


图 1.5　本文涉及的软件缺陷定位自适应机制示意图

软件缺陷定位方法本身作为一种基于测试信息的推理与搜索机制，对具备不同分布特征的测试信息的适应性也是必须要讨论的内容。

尽管我们承认目前很多关于缺陷定位的研究工作都体现了自适应思想，然而这些研究基本是针对特定的问题点，将包含自适应思想的方法或技术作为工具，设计解决方案。本文旨在构建自适应软件缺陷定位的研究框架，并以此为基础，在更高的层面，以面带点，提出实际的自适应机制。一方面，自适应软件缺陷定位的研究框架明确提出了自适应软件缺陷定位这一概念，解释其含义，并在此基础上描述了软件缺陷定位的各个过程，进而对相应的自适应问题，以及解决这些问题的自适应机制给出了数学描述。该研究框架明确了自适应软件缺陷定位的含义、需要引入自适应机制的子环节、各自适应机制的执行过程与输入输出以及它们的设计思路。框架可以作为实际自适应机制提出的基点，对自适应问题及约束的建立以及自适应方法的设计起指导作用。而实际自适应机制的提出与验证是对自适应软件缺陷定位研究框架的实例化，这些自适应机制具体实践并验证了自适应软件缺陷定位理论的可行性，以及其在提升软件缺陷定位效率上的有效性。研究框架以及具体的自适应机制互相关联、相辅相成，形成自适应软件缺陷定位的研究体系。

本论文考虑到的软件缺陷定位相关的自适应问题及自适应机制如图1.5 所示。图中描绘了整个软件缺陷定位的执行过程。首先对目标软件进行测试并且得到测试信息，而这些测试信息作为输入提供给所采用的自动化软件缺陷定位方法，自动化软件缺陷定位方法根据测试信息自动化地推理缺陷代码位置，并将推理的结果以缺陷预测的形

式传递给调试人员。最后，调试人员在该缺陷位置预测结果的指导下进行软件缺陷的排查。

针对上述流程，我们试图通过对缺陷定位过程中的状态与参数的进行实时观测与辨识，评估其适应性，并合理地引入自适应机制对其进行调整。如图所示，本论文对自适应机制的讨论主要面向与软件缺陷定位相关的三个过程及相关数据——软件测试、测试信息以及软件缺陷定位方法。具体来说，本论文针对缺陷定位的具体自适应问题与技术展开如下研究：

（1）面向缺陷定位的测试用例选取方法

本部分旨在通过对缺陷定位方法所给出的缺陷位置预测结果进行辨识，评估当前测试信息对软件缺陷定位的适应性并以此为基础在线调整测试策略。传统缺陷定位都假设测试用例执行信息是完备的，而在一些存在敏捷性开发需求的软件及软件开发环境中，这一点是很难做到的。很多情况下，测试用例的执行是同缺陷定位并行的，例如持续集成（Continuous Integration，CI）的软件开发模式[42, 102, 103]。软件测试过程中一旦发现错误，则调试环节立即启动，而不必等待所有的测试用例都运行完毕。此时，测试过程还在继续执行，而调试过程中还会继续接受测试过程提供的测试信息；此时测试不仅可以发现错误，还能够为软件缺陷定位提供更多的测试信息。而问题就是，应该采取什么样的测试策略来提升缺陷定位的效率呢？

Jiang 等人[42] 指出，当以缺陷定位为目的时，现存的那些测试用例选取策略[104, 105, 106] 就不一定合适了，它们的性能甚至还不如纯随机测试。Yoo 等人[47] 首次提出了面向缺陷定位的测试用例选取问题（Fault Localization Prioritization Problem, FLP）。在测试与缺陷定位并行执行的环境下，通过合理地挑选测试用例，使得那些对缺陷定位有帮助的测试用例优先被执行，进而使得缺陷语句能够更为快速地被找出来。Yoo 等人通过引入信息熵的概念来对测试用例进行挑选。然而，信息熵的计算是在概率意义下进行的，而程序谱方法所给出的排位表并没有严格的概率意义，其方法显得过于复杂且不透明。因此，本文采用直接面向程序谱方法的测试用例选取策略，力求达到更好的效果，所涉及的流程如图1.5 中流程①所示。

（2）基于输入空间划分的缺陷定位测试用例选取方法

图1.5中流程①需要用到测试用例的执行信息，即假设测试用例在实际执行前其覆盖信息就是已知的，这在很多情况下是不易被满足的。因此，依据执行信息设计的测

试策略在很多实际情景中是无法实现的，本文提出更具普适性的测试用例选取策略，使得面向缺陷定位的测试策略能够得到更广范的应用。该方法通过观测当前测试用例的执行结果，评估当前的测试信息对软件缺陷定位的有效性，并以此为依据安排接下来的测试用例执行过程，所涉及的闭环如图 1.5 中流程②所示。

（3）面向软件缺陷定位的自适应测试信息分类技术

本部分自适应机制所涉及的相关闭环如图 1.5 中流程③所示。在软件测试过程中，Oracle 问题会导致测试信息的不完整，进而影响软件缺陷定位的准确性。而本文对缺陷定位开始时的执行信息空间分布，以及当前缺陷定位结果进行观测，根据观测到的数据训练合理的分类器，并通过分类技术对不完备的测试信息进行合理的处理与利用，进而提升软件缺陷定位的准确性。

（4）基于程序谱空间的缺陷定位方法适应性评估及改善策略

基于程序谱的缺陷定位方法（Spectra-Based Fault Localization, SBFL）是目前软件缺陷定位研究中最常用的方法，也是本论文的研究重点之一。SBFL 旨在根据测试用例运行后的执行信息（即程序谱）推理出源程序缺陷的可能位置，并赋予相应的可疑度作为缺陷位置预测结果。尽管合理的测试策略能够改善程序谱空间的信息分布，然而决定程序语句特征在程序谱空间分布的最根本原因是软件本身的特性以及软件缺陷的属性。对于不同的软件与缺陷，其程序谱分布可能截然不同，而 SBFL 中不同的缺陷可疑度算法对不同程序谱分布的适应性也是不相同的。因此，本文提出相应的适应性评估机制，对所采用的 SBFL 可疑度计算公式在当前程序谱下的适应性进行评估，并且设计相应的调整算法，根据对程序谱空间的分析调整 SBFL 的执行过程，以提升该可疑度计算公式的适应性。所涉及的相关闭环如图 1.5 中流程④所示。

第二章　自适应软件缺陷定位的研究框架

本章初步建立自适应软件缺陷定位的研究框架，为后续章节中具体的适应性分析和自适应机制的提出与实现打下基础。首先我们对软件缺陷定位过程进行描述，并建立自适应软件缺陷定位的基本框架。在此基础上，分析软件缺陷定位环节的各个过程，并提出相应的自适应问题及研究思路。进一步，我们对本文所基于的程序谱缺陷定位方法进行详细介绍，并讨论其与各自适应问题的联系。此外，本章还对自适应软件缺陷定位技术的仿真与实验平台进行介绍；该平台作为本文研究工作的验证手段对后续章节的方法及相关的缺陷定位流程进行了实现与仿真。

2.1　软件缺陷定位及相关自适应机制的过程描述

软件缺陷定位任务执行过程中，调试人员对目标软件进行一系列交互与操作，并通过获取到的信息来推断软件缺陷的位置。该过程可以被看成是一个控制过程，其流程框图如图2.1所示。

其中，目标程序作为被控对象。在缺陷定位过程中，目标程序实质上是一个固有存在的系统：程序的各个组件（语句、函数等）以及它们之间的相互作用决定了系统的内部结构及运作机制，由人或者其它人造物（如测试工具）将输入信息作用到这些语句或函数上，从而得到系统的输出。在测试过程中，测试人员采用某种测试策略规划被测软件的输入序列以得到期望的输出（例如系统的失效），这与控制系统中通过特定的方式安排被控系统的输入而得到理想的输出特性的过程是吻合的。

与软件测试过程类似，相应的测试方法与测试策略则可以看成是控制系统的控制

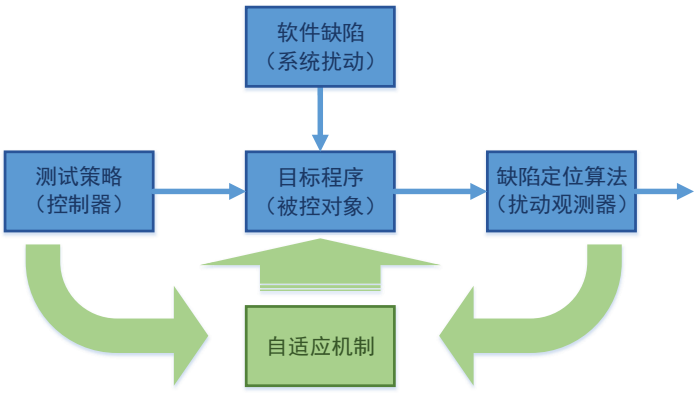


图 2.1　自适应软件缺陷定位的基本框图

器。对于作为被控对象的目标软件来说，测试用例就是其输入。而控制测试用例的生成、选取与执行的测试方法与测试策略就可以被看成是控制器。通过对控制器进行设计与调整可以改变系统的输入信息特征。在控制论中，控制器主要用来生成被控对象的输入，并对输入与被控对象间的相互作用与交互过程作出安排。这与软件测试策略对测试输入序列的规划作用是吻合的。

与软件测试不同的是，缺陷定位任务除了需要对测试用例执行后得到的信息进行判别与分析，还要根据这些测试信息，推断缺陷代码所处的位置。具体来说，我们需要通过软件缺陷定位算法对测试信息进行处理，将其转化为关于缺陷代码位置的指示信息，并输出对缺陷代码位置的预测结果。而在控制论视角下，若将没有缺陷的完美软件视为原始系统，将软件的缺陷看成系统的扰动，那么软件缺陷定位方法就相当于一个扰动观测器。因此在控制论视角下，整个软件缺陷定位环节的过程描述如下。

软件缺陷定位过程： 给定目标程序作为被控对象，测试策略作为控制器，而软件缺陷定位方法作为扰动观测器。控制器（测试策略）生成并选取相应的输入量（测试用例），并将其传递给被控对象（目标程序）。而被控对象（目标程序）接受输入信号（测试用例），并在被控系统自身作用下产生输出。此时，扰动观测器接受系统的输出并对其进行处理，最终生成扰动（缺陷）的预测信息。在控制过程中，我们希望观测器给出的观测值能够反映干扰的实际状况，即准确地预测出缺陷代码。

整个控制框架中，作为被控对象的待测软件与作为干扰信号的软件缺陷是客观存在且不易被改变的（基于变异的软件缺陷定位方法相当于在系统中植入人为干扰，本文不作讨论）。而作为控制器的测试策略以及作为扰动观测器的软件缺陷定位方法共同对软件缺陷定位的准确度产生影响。

在进行方法设计过程中，待测软件的内部结构是未知的，因此由于软件及开发环境的多样性所致，实际所采用的相关技术很可能不适用于当前的被控对象。而这种由被控对象及环境的不确定性带来的控制器及观测器的设计问题与自适应控制的适用情景是相吻合的。

从图2.1可以看出，在控制论视角下，自适应软件缺陷定位的含义就是要在整个控制系统（软件缺陷定位过程）的某些位置设置观测器，通过观测系统的某些状态，动态地对系统的结构及特性进行辨识，并根据辨识结果对相应的控制器（测试策略）与扰动观测器（软件缺陷定位方法）进行动态调整，以提升整个控制系统的性能（即软件缺陷定位的效率）。

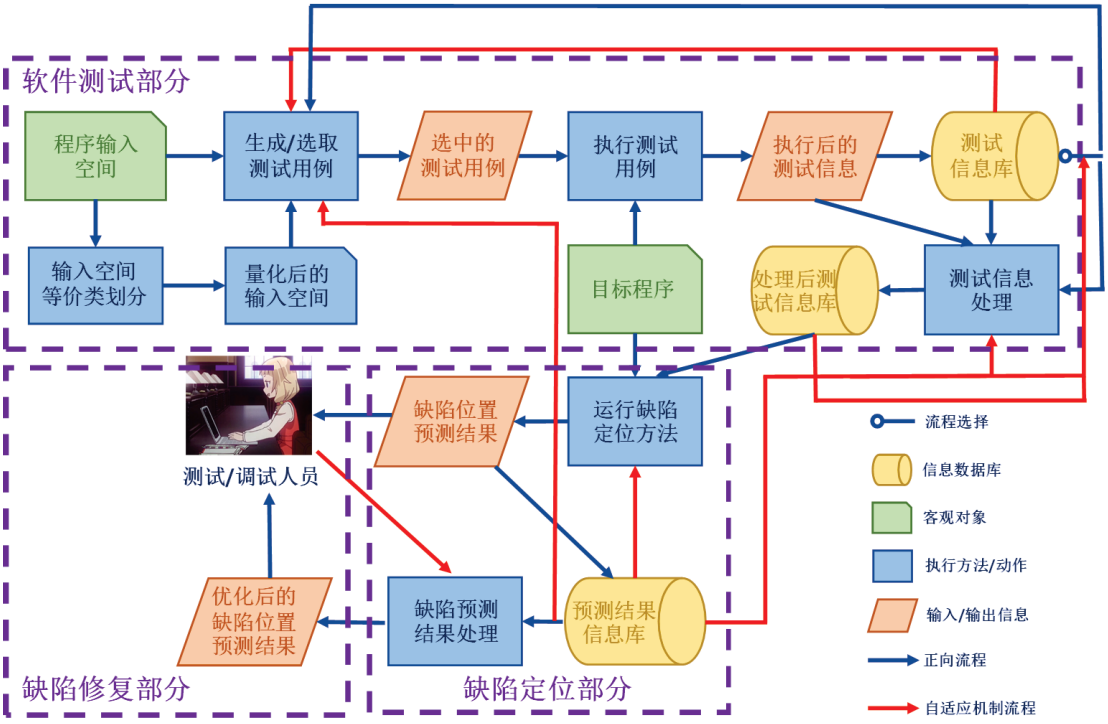


图 2.2　与软件缺陷定位相关的各过程及自适应机制的控制流程图

2.2　自适应软件缺陷定位研究框架

本节将针对软件缺陷定位的详细控制过程进行讨论，并在此基础上对软件缺陷定位的适应性及其相应的自适应机制进行描述。软件缺陷定位的详细控制流程如图2.2 所示。该流程图描绘了采用自动化方法执行缺陷定位任务时可能涉及的相关执行动作、客观对象、信息、信息库以及它们之间的流程及交互。其中，图中的方框表示相应的执行动作（例如运行程序或是执行某个方法）、平行四边形表示信息片（例如测试用例、缺陷定位输出信息等）、横向圆柱表示信息库（即对历史信息片的存储）、而缺角的矩形表示不易在缺陷定位过程中被人为改变的固有对象（如待测软件与其输入空间）。此外，图中蓝色箭头表示原始缺陷定位流程中的动作方向及输入输出，而红色箭头则表示引入相关自适应机制的位置及方向。

如绪论所述，与软件缺陷定位过程相关的活动包含测试、缺陷定位以及缺陷修复三个部分。其中，测试与缺陷修复分别是软件缺陷定位的前序及后序环节，而执行相应的方法对缺陷代码进行搜索的过程则是软件缺陷定位的核心。

在测试环节中，测试人员已知软件的输入空间，并采用适当的策略在测试空间中对测试用例进行选择或对其执行顺序进行调整。此时，需要对目标程序的输入空间进

行量化，使其能够被所采用的自动化测试用例生成及选择方法解析并且进行相关计算。此后，测试人员将得到的测试用例输入到目标程序中执行，观察测试的结果，收集执行过程中的信息，并将得到的测试信息存入测试信息库。当测试过程中发现软件失效时，就会考虑转入缺陷定位过程。此时，测试人员可能需要适当的评估准则来判断当前测试信息的充分性，以确定测试过程是否要继续进行。此外，测试或调试人员也可能采用相应的方法或技术对的测试信息做进一步处理，使其更有利于缺陷定位。

整个流程的核心是对缺陷语句位置的预测，即缺陷定位环节。首先，调试人员获取测试环节中得到的测试信息（或是处理后测试信息），将其输入给相应的缺陷定位技术与方法。假设调试员借助自动化软件缺陷定位方法来定位缺陷，则他会将目标程序与测试信息库作为输入，执行自动化软件缺陷定位方法与技术，并获得相应的输出。一般情况下，软件缺陷定位方法会输出软件缺陷代码位置的预测信息。这些预测会提交给调试人员，而调试人员也将根据这些缺陷预测信息进行最后的缺陷排查工作。

从图2.2中可以看出，整个缺陷定位过程中目标程序及其输入空间被视为固有对象，而需要进行改变、调整及优化的对象就是图中方框所表示的动作及方法。具体来说，欲采取某种措施或者手段提升软件缺陷定位的效率或是优化整个软件缺陷定位流程，就需要对缺陷定位过程中相关的动作、方法及流程控制进行合理的改进与调整。图2.2中“执行测试用例”这一动作在本文中被视为固定的动作而不做讨论；“输入空间等价类划分”是属于测试环节的活动，故在其被涉及时我们采用软件测试领域的经典划分方法；而本文在研究软件缺陷定位的优化问题时主要讨论以下内容：

（1）面向缺陷定位的测试用例生成/选取优化策略；

（2）面向软件缺陷定位的测试信息处理策略；

（3）软件缺陷定位方法适应性的评估与优化；

（4）缺陷预测结果处理方法及其应用。

上述流程中，关于测试用例的生成、选取策略以及软件缺陷定位方法都存在相应的现有研究工作。然而，由于软件的多样性所致，对于任何方法，其都会受当时缺陷定位环境的影响。而本文所讨论的自适应机制就是要识别缺陷定位时的环境特征，并以此为依据调整相关的方法与控制过程。图2.2中，红色箭头所表示的就是各自适应机制及其作用方向。这里，代表自适应机制的红色箭头与仅表示操作流程的蓝色箭头的区别在于，红色箭头（自适应机制）不仅表示操作流程（包括正向与反馈流程），还表示对当前环境进行辨识以及对控制过程及相关方法进行调整与优化。

自适应机制对环境进行辨识的过程需要基于整个流程执行与信息存储过程中获取的各种历史数据，包括用来存储测试用例及其执行信息的测试信息库或是经过处理后的测试信息库，以及用来存储缺陷定位输出结果的缺陷代码位置预测信息库。而自适应机制的作用位置就是前文中提到的可以被调整及改变的流程与方法，即测试用例生成/选取策略、测试信息处理策略、软件缺陷定位方法、缺陷位置预测结果处理以及测试与缺陷定位的过程控制。从所依据信息的作用方向来看，图中包含的红色箭头及其所描述的自适应机制如下（下文中的 $\rightarrow$ 代表红色箭头的指向）：

（1）测试信息库 $\rightarrow$ 测试用例生成/选择策略

该自适应机制旨在通过对现有测试信息的观测，动态调整测试策略，使其更适应当前缺陷定位的环境需求，其流程与自适应测试的基本流程是一致的。因此，各种现有的自适应测试方法，如自适应测试（Adaptive Testing，AT）[49, 107]、自适应随机测试（Adaptive Random Testing，ART）[46, 108] 、动态随机测试（Dynamic Random Testing，DRT）[109, 110] 等，均可以被采用。然而，这些方法在软件缺陷定位这一目标下的表现则需要重新被评估。本部分研究中，我们将借助软件缺陷定位的相关理论及启发式准则来识别有利于软件缺陷定位的测试信息特征，分析这些特征与测试过程之间的关系，并以此为基础设计相应的测试策略。

（2）缺陷预测结果信息库 $\rightarrow$ 测试用例生成/选择策略

该自适应机制旨在通过对缺陷预测结果的观测，动态调整测试策略，使其更适应缺陷定位的需求。相比于测试信息，当前的缺陷定位结果是软件与缺陷特性最直接的体现。因此对当前缺陷定位结果的观测，更能掌握当前测试用例集对缺陷语句的辨识度。以此为依据能够较为有效地对测试充分性及测试信息的表达能力进行判定，并对测试策略做出合理的调整。由于需要对缺陷定位结果进行观测，经典的自适应测试方法不再适用，故而对于适应性的判别准则也需要重新被设计与检验。

（3）缺陷预测结果信息库 $\rightarrow$ 测试信息处理

该自适应机制通过对却缺陷预测结果进行观测，并以观测结果为指导，对测试阶段得到的测试信息进行处理，使得处理后的测试信息能够更好地暴露缺陷的位置。软件缺陷定位结果的准确性在很大程度上依赖于测试信息的质量，而通过观测当前的缺陷预测结果信息库，能够对于当前测试信息的质量进行初步的度量。以此为依据，我们除了能够像前面所述那样去调整测试策略，还能够对已经取得的测试信息进行适当

地处理（如提升其多样性程度等）。

（4）缺陷预测结果信息库 → 缺陷定位方法

该自适应机制旨在通过对待测软件的程序谱，以及缺陷代码位置历史观测结果的评估与分析，改进软件缺陷定位方法，使得调试人员能够针对当前的程序进行正确的缺陷定位决策。通过对程序谱及缺陷位置预测结果的观测可以了解程序谱的性质、待测软件的结构特征以及语句间的依赖关系及其在失效程序谱上的反映。目前软件缺陷定位的相关算法被广泛研究，然而这些算法都有其擅长与不擅长处理的缺陷与软件；而通过适应性分析，我们可以有针对性地选择适当的缺陷定位算法或是对当前算法的相关参数进行合理地调整，以得到更准确的缺陷定位结果，亦或是对是否采用自动化软件缺陷定位进行合理的决策。

2.3　自适应软件缺陷定位方法的数学描述

2.3.1　软件缺陷定位方法的基本数学表述

为了便于分析与论述，本小节给出软件缺陷定位方法的一个数学模型。设目标程序为 PG，其包含的 n 条语句集合为 S，其中的某一条语句记为 s。软件的测试用例集为 T，设测试用例集 T 包含 m 个测试用例 $t_1, t_2, \ldots, t_m$。对某个特定测试用例 t，其包含的信息可以用多元组 $< inp^t, C^t, o^t, \ldots >$ 来表示，这里 $inp^t \in I$ 表示程序的输入信息，其中 I 表示目标程序 PG 的输入空间，$C^t \subset S$ 表示测试用例 t 被执行时经过的程序语句集合；而布尔值 $o^t \in \{passing, faling\}$ 则表示程序输出的正确性。当然，实际方法可能会用到更多的信息（例如具体的输出信息与谓词真值信息等），因此测试信息多元组的维度也会按照需求进行扩充或是缩减。例如，本文主要用到的程序谱缺陷定位方法中，测试信息主要用 $< C^t, o^t >$ 来表示。

现考虑缺陷定位问题，记缺陷语句为 s^f，这里 s^f 是缺陷语句的标记，其并不一定仅仅代指一条语句；例如，s_1、s_2 都是缺陷语句，则可以同时将其记为 $s^f = s_1, s^f = s_2$。记软件缺陷定位方法的执行过程为 FL，则软件缺陷定位过程可以写成

$$\hat{s}^f = FL(PG, T) \tag{2.1}$$

其中 FL 代指所采用的缺陷定位方法，目标程序 PG 与测试用例集 T 作为方法的输入，而 $\hat{s}^f$ 是缺陷定位方法给出的缺陷位置预测结果。对于一种缺陷定位方法，可以将

其表示为下面的规划模型：

$$\min J(PG, T, h(s^f))$$

满足：

$$Constraint_1(PG, T, h(s^f))$$

$$Constraint_2(PG, T, h(s^f))$$

$$\dots \tag{2.2}$$

其中，J 为目标函数，一般情况下，其可以是一种关于软件缺陷位置的一个启发式准则；而 $Constraint_1, Constraint_2, \dots$ 为一系列约束，其表示在 $h(s^f)$ 假设下的测试信息特征约束。例如，本文主要针对基于语句覆盖的缺陷定位方法进行讨论，则模型 (2.2) 可以写成如下形式：

$$\min J \tag{2.3}$$

满足：

$$\mathbf{c}^t\mathbf{x} \geq 1 \quad \forall t \in TF \tag{2.4}$$

$$J = \sum_{t \in TP} \mathbf{c}^t\mathbf{x} \tag{2.5}$$

其中 $TF \subset T$ 与 $TP \subset T$ 分别表示 T 中的失效测试用例集以及成功测试用例集。决策变量 $\mathbf{x}$ 可以理解为缺陷位置预测结果 $h(s^f)$ 的向量表示，即 $\mathbf{x} = [x_1 x_2 \dots x_n]^T$，满足

$$x_k = \begin{cases} 1, & s_k \in h(s^f) \\ 0, & s_k \notin h(s^f) \end{cases}$$

而 $\mathbf{c}^t = [c_1^t c_2^t \dots c_n^t]$ 为测试用例 t 的覆盖信息 C^t 的向量表示，即

$$c_k^t = \begin{cases} 1, & s_k \in C^t \\ 0, & s_k \notin C^t \end{cases}$$

缺陷语句对测试信息的限制作用由约束 (2.4) 表示，其含义可以解释为任何测试用例至少经过一条缺陷语句；而目标函数 J 由约束 (2.5) 计算得到，其表示缺陷语句被成功测试用例经过的次数，是一种启发式准则。总的来说，模型 (2.3)–(2.5) 根据给定的测

试信息约束来推测缺陷代码的位置，并且通过启发式准则对推理结果进行细化。

现考虑该模型的求解问题。约束 (2.4) 实际上是由每个失效测试用例所对应的 $|TF|$ 个具体约束组成的。这些约束中，每一条都是对缺陷语句绝对性的限制。首先，若有失效测试用例 t，则可以通过约束

$$\mathbf{c}^t\mathbf{x} \geq 1 \tag{2.6}$$

将模型的解（即缺陷代码位置）锁定为

$$s^f \in C^t \tag{2.7}$$

若有另一个测试用例 t'，则得到约束

$$\mathbf{c}^{t'}\mathbf{x} \geq 1 \tag{2.8}$$

现假设目标程序只有一个缺陷，则可以确定其一定存在于两个测试用例覆盖语句的交集中，即

$$s^f \in C^t \cap C^{t'} \tag{2.9}$$

若去掉单缺陷假设，则关于缺陷代码位置的推理则要更为复杂一些。具体来说，有一条缺陷语句 s^f 会存在于 $C^t \cap C^{t'}$ 中亦或是在 $C^t - C^{t'}$ 以及 $C^{t'} - C^t$ 中各存在至少一条缺陷语句，即

$$(\exists s^f \in C^t \cap C^{t'}) \vee (\exists s_1 = s^f, s_2 = s^f, s_1 \in (C^t - C^{t'}) \wedge s_2 \in (C^{t'} - C^t)) \tag{2.10}$$

然而无论如何，通过额外测试信息 t' 的引入，我们对软件缺陷的位置有了更多的了解。

也就是说，t' 的引进可以使我们将缺陷语句在语句集合 $(C^t - C^{t'}) \cup (C^{t'} - C^t)$ 中的可能性部分排除（由于还要考虑多缺陷时缺陷语句分别在这两个集合中的情况，因此这里说“部分排除“）。而若 t 与 t' 的覆盖信息存在包含关系，例如 $C^t \subset C^{t'}$，则有 $C^t - C^{t'} = \emptyset$，这样仍只能得到 $s^f \in C^t$，此时 t' 并没有带来有用的信息。相反，若 t 与 t' 的差异性很大，则 $|(C^t - C^{t'}) \cup (C^{t'} - C^t)|$ 就越大，而我们就能排除掉更多的语句。

现考虑成功执行的测试用例所建立的约束 (2.5)。成功测试用例的约束不像失效测试用例约束那样对缺陷语句的推理具有确定性，其只能作为启发式准则或者是理论假设。然而，若所依据的启发式准则或是理论假设是正确合理的，那么其同样会对缺陷定位结果准确性的提升起积极作用。此时，我们就希望目标函数值 $\sum_{t \in TP} \mathbf{c}^t\mathbf{x}$ 的计算是在预想的测试剖面下进行的，且希望测试过程中随机因素带来的干扰尽可能小。因

此，我们仍需要足够的测试数据作为支持。

总结来说，为了缩小推理范围，我们希望拥有更多的测试信息以保证约束的数量，并且各约束之间需要有足够的差异性。此外，对于模型本身与测试信息的契合程度也会影响缺陷定位的准确度。

2.3.2 自适应机制在数学模型中的含义及合理性

本小节试图将2.2节所列出的自适应机制放在2.3.1小节所述的缺陷定位模型中进行讨论并分析其合理性。

（1）面向缺陷定位的自适应测试用例生成/选择策略

如上所述，在缺陷定位时，我们希望测试信息尽可能的多样化。在测试过程中，假设测试用例 $t_1, t_2, \ldots, t_{m'}$ 已经被执行，则接下来在从输入空间中选取测试用例，或是在安排剩余测试用例的执行顺序时，更期望选取与当前测试信息库中的测试信息差异性较大的测试用例优先执行。这样，当缺陷定位任务在任意时间开始时，都可以保证当前的测试信息具有较高程度的多样性。给定一个未执行的测试用例 t，则如何判断 C^t 与现有测试信息库中覆盖信息的区别就成为该自适应策略的关键。首先，可以采用相似性度量机制计算现有测试覆盖信息 $C^{t_1}, C^{t_2}, \ldots, C^{t_{m'}}$ 与 C^t 的相似性，并选择相似性最低的测试用例。该方法利用测试信息库的内容，对应于**测试信息库 $\rightarrow$ 测试用例生成/选择策略**流程，本文具体会在第四章中对其进行讨论。此外，还可以通过分析覆盖信息 C^t 对当前缺陷定位方法给出的预测结果 $h(s^f)$ 的区分度来度量 t 的价值，该部分内容具体会在第三章中进行讨论。

（2）面向缺陷定位的自适应测试信息处理策略

除了多样化，我们还希望测试的信息量是足够的。一方面，对于失效测试用例约束 (2.4) 来说，缺陷定位时的信息量与信息多样性程度是相关的。在测试策略较为完备时，可利用的测试信息越多，其多样性程度也会越高。此外，对于成功执行的测试用例约束 (2.5) 来说，也需要更多的测试用例来保证测试剖面能够被正确地体现出来。而本部分的自适应机制就旨在提高测试用例的利用率。若现有执行过的测试用例集为 T，并假设由于一些原因使的 T 中的部分测试用例无法被当前的缺陷定位方法直接采用，并设这些测试用例的集合为 T'。则本部分的自适应机制旨在根据缺陷定位执行过程中所得到的 $h(s^f)$ 信息，以及 T 中那些可利用的测试信息，对 T' 进行合理利用，具体将在第五章讨论。

（3）缺陷定位方法的适应性分析以及改进策略

此外，模型本身的适应性也是软件缺陷定位的关键。对于约束 (2.4)，其可以限定软件缺陷的位置，可是并不能将缺陷位置唯一确定；也就是说，仅考虑失效测试用例约束会产生多解问题。例如，当整个程序全都是缺陷语句时（即 $\forall s \in S \rightarrow s = s^f$），约束 (2.4) 就是满足的。其实不难看出当缺陷语句越多时，失效约束就越容易满足。然而，通常情况下，已经进入测试阶段的程序是不会存在那么多缺陷代码的。因此我们有必要通过约束 (2.5) 在模型可行域中进行进一步筛选。此时，整个模型是否与当前的软件结构，或是测试环境相匹配，即缺陷定位方法的适应性，也是需要解决的问题。例如，某些倾向于选择失效测试用例的测试剖面中[49]，巧合一致性的成功测试用例会有更大的概率被选中[41]，这使得即使是正确的缺陷预测假设也会得到较大的目标函数值。又例如当软件缺陷处在程序主干时，无论测试用例是否能够暴露缺陷，其都会经过缺陷语句，进而导致正确的缺陷位置预测也会得到很大的目标函数值。而该自适应机制就旨在通过分析测试信息空间（这里指维度为 $|S|$ 的语句覆盖空间）以及空间上的信息分布（这里指测试覆盖信息 $C^{t_1}, C^{t_2}, \ldots, C^{t_2}$），对缺陷定位方法的适应性进行判别，具体在第六章进行讨论。

2.4　程序谱缺陷定位方法及其与缺陷定位模型的关系

本章讨论了软件缺陷定位流程中自适应机制的含义、合理性及研究框架。对于任何软件缺陷定位方法都会有相应的自适应机制，而本文则主要针对程序谱缺陷定位方法的自适应机制进行讨论。基于程序谱的缺陷定位方法（Spectra-Based Fault Localization, SBFL）是目前被研究得最为广泛的一类自动化缺陷定位方法，其根据测试用例的覆盖信息以及测试结果的正确性信息来为每个程序语句都附上一个可疑度值，以表征其包含缺陷的可能性。为了方便后面章节的叙述，本小节将阐述 SBFL 的数学表述。

基于第 2.3.1小节给出的定义。设目标程序为 PG，其包含 n 条语句 $s_1, s_2, \ldots, s_n$，这些语句所组成的程序语句全集为 S，用 s 表示其中的一条语句。设缺陷定位时考虑采用的测试用例集为 T，其包含 m 个测试用例 $t_1, t_2, \ldots, t_m$，用 t 表示其中一条测试用例（其也可以是从输入空间中生成的新测试用例）。对于一条测试用例 t 来说其测试信息可以用二元组 $< C^t, o^t >$ 来表示。这里，$C^t = \{s \in S | s\text{被}t\text{经过}\}$ 为测试用例 t 的覆盖信息，而 o^t 为输出结果的正确性信息，其中 $o^t = failing$ 表示该测试用例触发了

程序失效，而 $o^t = passing$ 则表示测试用例没有触发失效。

SBFL 则根据 T 中的测试信息提炼出程序谱，即各个语句的覆盖信息。基于测试用例 T，对于目标程序的每条语句 s，其覆盖信息都可以表示为四元组 $< a_{ef}^{s,T}, a_{ep}^{s,T}, a_{nf}^{s,T}, a_{np}^{s,T} >$。其中 $a_{ef}^{s,T}$ 与 $a_{ep}^{s,T}$ 分别表示经过语句 s 的失效测试用例数量与成功测试用例数量，而 $a_{nf}^{s,T}$ 与 $a_{np}^{s,T}$ 则分别表示没有经过语句 s 的失效测试用例数量与成功测试用例数量。此外，对于全局性的覆盖特征，我们令 F^T 和 P^T 表示成功测试用例的总数与失效测试用例总数，即我们有

$$\begin{aligned} a_{ef}^{s,T} &= |\{t \in T | o^t = failing, s \in C^t\}| \\ a_{ep}^{s,T} &= |\{t \in T | o^t = passing, s \in C^t\}| \\ a_{nf}^{s,T} &= |\{t \in T | o^t = failing, s \notin C^t\}| \\ a_{np}^{s,T} &= |\{t \in T | o^t = passing, s \notin C^t\}| \\ F^T &= |\{t \in T | o^t = failing\}| \\ P^T &= |\{t \in T | o^t = passing\}| \end{aligned} \tag{2.11}$$

这里，语句的覆盖信息会根据需要用到不同的表示方法。例如，当测试用例 T 给定并不作为讨论对象时我们将这些覆盖信息统计量写成 a_{ef}^s、a_{ep}^s、a_{nf}^s、a_{np}^s、F 与 P；而当强调语句 s 的对覆盖信息的影响时，可以将其写成 $a_{ef}(s)$、$a_{ep}(s)$、$a_{nf}(s)$ 与 $a_{np}(s)$。

利用以上覆盖信息统计量，SBFL 会计算每个语句的可疑度，以表征其包含缺陷的可能性，而可疑度的计算主要基于以下两个基本理论：

SBFL 基本原理 1： 在失效测试用例中执行频率高的语句更有可能是缺陷语句。

SBFL 基本原理 2： 在成功测试用例中执行频率低的语句更有可能是缺陷语句。

这两个基本原理在后续章节中也会被提及并讨论。具体来说，一条缺陷语句的可疑度会通过包含该语句覆盖信息统计量且能够反映 SBFL 基本原理的计算式得到，称该计算式为可疑度计算公式，其可以用一个函数 R 表示。而通过 R 计算出的语句 s 的可疑度记为 $r^{s,T}$，即

$$r^{s,T} = R(a_{ef}^s, a_{ep}^s, F, P) \tag{2.12}$$

可疑度计算公式的设计是 SBFL 领域的主要研究问题之一，不同的可疑度计算公式都在反映 SBFL 基本原理的前提下基于不同的启发式信息。早期提出 SBFL 时用到

的可疑度计算公式是 Tarantula[86]，其表达式为

$$R_{Tarantula}(a_{ef}^{s,T}, a_{ep}^{s,T}) = \frac{a_{ef}^{s,T}/F}{a_{ef}^{s,T}/F + a_{ep}^{s,T}/P} \tag{2.13}$$

$R_{Tarantula}$ 的实质是将一条语句被执行后的失效率当做其可疑度值。而进一步研究发现，在软件缺陷定位过程中，失效测试用例会起到更大的作用。也就是说，在可疑度计算公式中 a_{ef} 应该被赋予更大的权值。例如 Abreu 等人[76] 提出的 Ochiai 公式

$$R_{Ochiai}(a_{ef}^{s,T}, a_{ep}^{s,T}) = \frac{a_{ef}^{s,T}}{\sqrt{F(a_{ef}^{s,T} + a_{ep}^{s,T})}} \tag{2.14}$$

就相当于在 Tarantula 的等价公式上多乘以一个 $a_{ef}^{s,T}$。而在单缺陷假设下，可以将 $a_{ef}^{s,T}$ 的权值设为足够大，如此就得到了 Naish 等人[75] 提出的 Op 公式：

$$R_{Op}(a_{ef}^{s,T}, a_{ep}^{s,T}) = a_{ef}^{s,T} + a_{ep}^{s,T}/(P+1) \tag{2.15}$$

实际上，可疑度计算公式的设计是有很大自由度的。任何能够反映 SBFL 基本原理的，且由 $a_{ef}^{s,T}$ 与 $a_{ep}^{s,T}$ 组成的算式都可以看作是可疑度计算公式。目前已经有超过上百种公式被提出并讨论，本文的第六章将对 SBFL 可疑度计算公式做详细介绍。

借助可疑度公式可以计算各语句包含缺陷的可疑度值，这里记语句 s 的可疑度值为 $r^{s,T}$（当不对 T 进行讨论时，同样可以将其简化为 r^s）。而 SBFL 的输出则是一个可疑度列表，记为 L^T；其中各语句按照其可疑度值降序排列。若缺陷语句的可疑度很大，则其会在 L^T 中排在靠前的位置。如此，调试员就会在纠错时优先对缺陷语句进行考查，进而提升了缺陷定位效率。具体 L^T 的定义式如下：

$$L^T = [\lambda_1^{L^T} \lambda_2^{L^T} \cdots \lambda_n^{L^T}] \tag{2.16}$$

其中

$$\lambda_k^{L^T} = \langle s_k^{L^T}, r_k^{L^T} \rangle,$$

$s_k^{L^T}$：在 L^T 中排名为 k 的语句

$r_k^{L^T} = r^{s_k^{L^T}}$：语句 $s_k^{L^T}$ 的可疑度值

最后在对 SBFL 的效率进行评价时通常选择定位代价 (Expense) 作为标准，记可

疑度列表 L^T 的 Expense 值为 E^{L^T}，其计算式为

$$E^{L^T} = (i/|L^T|) \times 100\%, \quad s_i^{L^T} = s^f \tag{2.17}$$

其含义为按照可疑度列表 L^T 的排序进行检查时，遇到缺陷语句时所检查过的语句占程序全部语句的百分比。

SBFL 通过各语句覆盖信息的统计量来对缺陷语句的位置进行推断。其中，对语句的 a_{ef} 值的考查是为了强调缺陷语句更趋向于被失效测试用例所覆盖，这与第 2.3.1小节所给出模型的失效测试用例约束（即约束 (2.4)）是吻合的。而对语句 a_{ep} 值的考查与成功测试用例形成的约束（即约束 (2.5)）相吻合。实际上，SBFL 可以看成是该模型的简化。首先，在单缺陷假设下（即假设目标程序只有一条缺陷语句），则采用 Op 可疑度公式（即式 (2.15) ）进行计算后，排在可疑度列表 L^T 第一位的语句同样是第 2.3.1小节中模型的解。也就是说，二者是等价的。

而在一般情况下（即去除单缺陷假设的情况下），SBFL 本身也可以用模型的形式做如下表述：

$$\max J = R(a_{ef}, a_{ep}) \tag{2.18}$$

满足：

$$a_{ef} = \textstyle\sum_{t \in TF} \operatorname{sgn} \mathbf{c}^t \mathbf{x} \tag{2.19}$$

$$a_{ep} = \textstyle\sum_{t \in TP} \operatorname{sgn} \mathbf{c}^t \mathbf{x} \tag{2.20}$$

模型 (2.18)–(2.20) 中将约束 (2.4) 简化为求 a_{ef} 的值。而整个求解 SBFL 的过程也等价于循环求解该模型（每求解一遍会将当前解剔除），并最终生成可疑度列表。而由于其基本的思想与第 2.3.1小节提出的数学模型是相似的，测试信息的数量与多样性程度同样在定位过程中起着重要的作用。也就是说，前面讨论的自适应软件缺陷定位的研究框架在 SBFL 中也同样适用。

2.5 自适应软件缺陷定位技术仿真与实验平台

本节将对本文研究所用到的实验平台进行介绍。该平台完成了本文方法验证工作的所有实验。该实验平台的架构具有较好的可扩展性，有利于为更多实验对象的加入，以及缺陷定位方法在现实软件中的应用。

2.5.1　实验平台总体部署

2.5.1.1　缺陷定位实验及实验对象特点

相比于其它研究工作的实验仿真系统，自适应软件缺陷定位实验平台存在以下特点：

（1）实验对象的分散性

很多研究领域都能够找到适合的该领域的特定研究对象，如无人机任务规划领域中适合的飞行器及场景仿真模型[111] 以及供应链优化领域中的标准数据模型[112]。而对于软件测试及缺陷定位领域来说，实验对象就是软件本身。不同软件的架构、编码与接口都是多样的，软件的多样性也正是自适应软件缺陷定位所需要面对的挑战。因此，在验证一个软件缺陷定位方法时，需要将多个软件作为实验对象进行实验，而实验平台也必须具备操作多种软件对象的能力。具体来说，平台应具备良好的可扩展性，为各类软件测试脚本的编写提供接口，并为相关的输入输出数据提供处理接口。

（2）功能的分散性

区别于对单一研究问题的探索，自适应软件缺陷定位涉及缺陷定位方法本身、面向缺陷定位的测试过程、以及测试信息处理等多个环节。而各研究问题的求解流程，也需要各环节间的相互协调。因此，对于实验平台来说，既需要各个环节内部功能的封装与接口暴露，亦需要各个环节之间的组织协调与交互过程脚本。

2.5.1.2　实验平台的组成与交互

基于第2.5.1.1小节的讨论，本文所用到的实验平台由标准程序库、实验对象脚本库、实验方法过程库以及实验数据处理四部分组成。不同部分所涉及的流程与功能不同，而这些流程及功能所适合的编程模式以及相关的 IDE 也各不相同。因此，该平台的部署方式是分散的，不同部分使用不同的语言及开发工具编写，并在不同系统上运行。

首先，本文主要选取 SIR 标准程序库中的 C 程序作为实验对象。即 *flex* 、*grep* 、*gzip* 、*sed* 、*Siemens* 测试套件以及 *Space* 程序，这些标准程序的标准运行环境都是 Linux。因此，平台的实验对象脚本部分在 Linux 平台上采用 shell 脚本和 python 代码编写。具体来说，该部分分为测试脚本以及接口脚本。其中，对于每个实验对象，我们都需要相应的测试脚本以控制该对象的测试过程——运行该对象的测试用例、检测程序的失效并收集测试用例的执行信息与执行结果。此外，接口脚本则负责将测试时

收集的信息转化为能够读取的形式并储存起来，为下一步缺陷定位环节的执行做准备。

实验方法过程库主要用来实际实现缺陷定位方法、测试方法以及本文提出的各种自适应机制。由于这一部分需要对程序、测试以及各种方法进行建模与实现，因此我们采用功能比较强大的 Visual Studio 作为开发环境。这部分主要包括方法模块以及缺陷定位流程脚本。其中，方法模块用来对程序进行建模，以及实现各种测试方法与缺陷定位方法，该部分基于面向对象思想进行编写。缺陷定位流程脚本用来实现本章介绍的缺陷定位任务的各个子环节，并灵活地实现各自适应机制。由于该部分的核心在于各种方法的编排以及对整个流程的控制，因此采用脚本方式编写。本部分不会实际运行实验对象，而是通过对实验对象的程序模型以及实验脚本库中得来的测试信息进行操作，来实现与执行本文提出的各种方法。这其中，程序对象的模型存储在 SQLSever 数据库中，而测试信息则从 Linux 测试脚本生成的文本文件中读取。最后，我们也会用 Matlab 等软件来对实验的结果进行处理及分析。下面我们将对实验对象脚本库和实验方法过程库做进一步介绍。

2.5.2 实验对象脚本库

本文的实验对象采用 SIR 程序库中的 C 程序作为实验对象，对于每个实验程序，SIR 提供了源码、缺陷版本以及测试框架。本部分中，每一个特定目标程序的控制在一个目录下完成。该目录下各子目录中都存放着相应的信息，这里介绍比较重要的几个子目录。

- version 目录：存放实验对象各个版本的源代码以及缺陷版本信息。
- version.alt：在实验运行过程中即时存放目标代码的目录。由于实验程序会存在很多缺陷版本，因此需要通过相关脚本将缺陷版本配置好后，生成实际需要运行的临时代码。而这些临时代码就存放在 version.alt 中。
- inputs：存放目标程序中测试用例的输入信息。
- output.alt：存放实验过程中各测试用例的输出信息。
- traces.alt：存放各测试用例的覆盖信息。
- scripts：存放测试脚本。

在该部分代码编写过程中，我们需要在 scripts 目录下，编写相应的脚本以控制整个实验流程，下面对主要的文件进行介绍。

- Experiment.py：运行整个实验的脚本，包括生成测试脚本、运行所有测试用例、获取测试用例的输出及覆盖信息以及区分成例与失例。该脚本用 python 语言编写。

- v∗.sh：原始目标程序的测试脚本，v1-v5 分别是程序的版本号。需指出，该脚本运行的是没有植入缺陷时的正确程序，其运行结果会作为相应版本的测试 Oracle。

- ScriptsForGcov.py：动态生成程序缺陷版本的测试脚本 v∗.gcov.sh。该脚本作为缺陷程序的测试脚本，运行缺陷程序，并获取其输出及测试覆盖信息。

- FaultSelection.py：该脚本为缺陷植入脚本，通过修改程序头文件 FaultSeeds.v∗.h 中的宏定义来向目标程序植入缺陷。

在实验过程中，先通过执行原始程序的测试脚本 v∗.sh 生成程序的测试 Oracle；再通过 FaultSelection.py 向程序中植入缺陷，并运行 ScriptsForGcov.py 以及其生成的测试脚本 v∗.gcov.sh，以执行缺陷程序的测试用例，并获取其覆盖信息；最后，将缺陷程序的输出与 v∗.sh 脚本得到的标准输出进行比对，以判断程序的失效。

2.5.3　实验方法过程库

实验方法过程库用来实现各种软件测试方法以及缺陷定位方法，其可以分为对象、方法类和过程脚本两部分。对象、方法类封装了本文研究所需要的程序模型及测试用例模型，以及各种测试方法的实现及调用接口，其主要包含以下几个类：

- FLProgram 类：缺陷定位用到的程序模型，包含语句、函数、缺陷语句等属性，还包含了各属性成员之间的相互索引。该类的 ReadProgram 方法能够根据数据库中储存的程序信息将该程序模型实例化。此外，程序的语句及函数都有相对应的模型类（即 FLStatement 类与 FLFunction 类）。

- FLTestSuite 类与 FLTestCase 类：缺陷定位所用到的测试用例集与测试用例模型。其中 FLTestCase 类储存测试用例的属性，包括编号、输入输出及语句覆盖信息等。而 FLTestSuite 类储存测试用例集的信息，包括该测试用例集包含的测

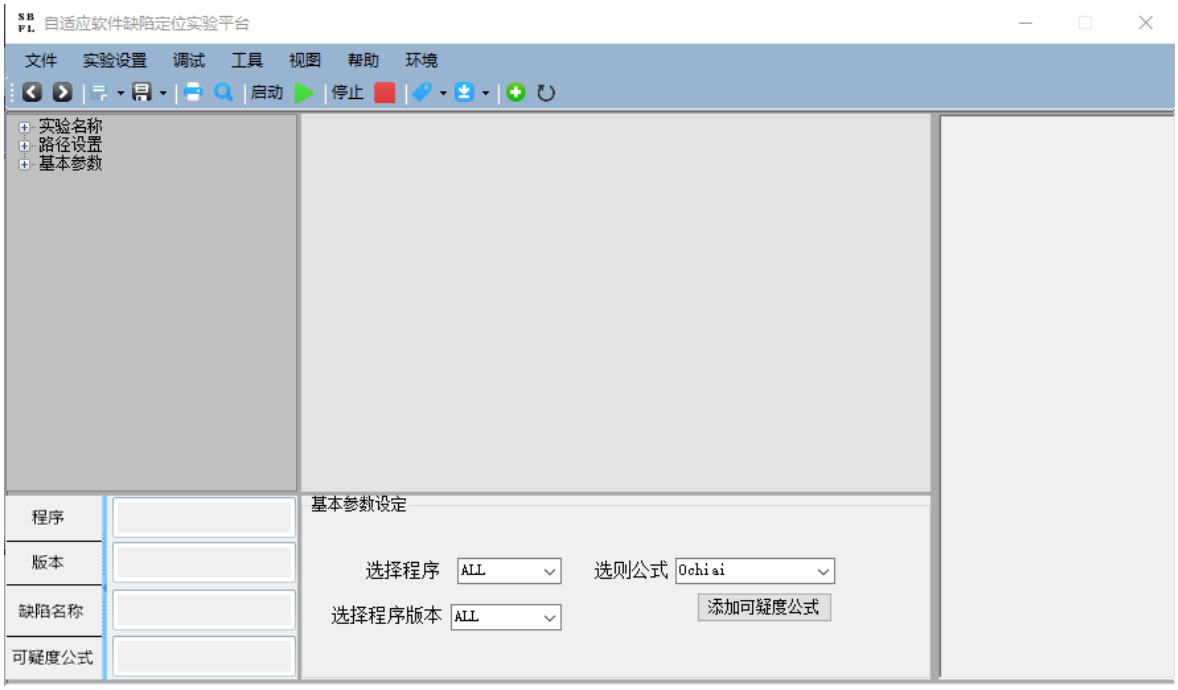


图 2.3　方法过程库主界面

试用例，成例集和失例集等。其中，该类的 ReadTestSuite 方法能够根据 Linux 平台得到的测试信息将测试用例集模型实例化。

- FLDebugger 类：封装了本文提出的各种缺陷定位方法以及面向缺陷定位的测试方法。其包含目标程序及测试用例集属性，这些属性需要通过将前面介绍的 FLProgram 类与 FLTestSuite 类实例化来赋值。此外，FLDebugger 类实现了本文提出的各种方法；例如，该类中的 SBFL_Normal 方法实现普通的 SBFL，而该方法的若干重载形式分别为成功测试用例与失效测试用例的加权预留了接口。此外 FLResult 类、FLMetric 类、ProgramSpectum 类以及 StatementSuspicious 类分别建立缺陷定位的输出、可疑度计算公式、程序谱、以及语句可疑度值的模型。

过程脚本则统一封装在 TestingProgram 类中，其中每一个方法都模拟一套缺陷定位的流程，而每个流程都对应着本文所涉及的一个缺陷定位问题及其求解过程。例如，TestingProgram 类的 UnlabelFL 方法实现了本文第五章提出的无标签测试信息处理方法。此外，我们为方法过程库设置了相应的界面，如图2.3所示。在该界面中我们可以选择希望执行的缺陷定位过程，并配置相关方法的参数。

2.5.4　数据库设计

在对缺陷定位方法进行检验时，我们需要知道实验对象的版本信息及缺陷信息，这些信息的存储则通过 SQLSever 数据库实现。首先实验对象数据表储存了实验对象的基本信息，其中一个实验程序的一个发布版本作为一条记录，以 ID 为主键。该表作为主表对其它数据表起约束作用。而缺陷设置表储存了为各实验程序配置的缺陷信息，以 ID 与缺陷编号组成复合主键，每一条记录对应了一个特定目标程序的一组缺陷配置。而具体的配置信息记录在缺陷描述表以及缺陷桩点对照表中。在实验过程中，方法库通过组合读取实验程序表以及缺陷配置表的信息来对实验对象的相关类进行实例化。

第三章　基于覆盖信息多样性的缺陷定位测试优化方法

缺陷定位测试用例优化（Fault Localization Prioritization, FLP）问题旨在通过对测试用例的执行顺序进行合理的安排，来提升缺陷定位的效率。FLP 问题是自动化测试与缺陷定位领域的重要研究问题，其关键是提出并量化对缺陷定位起积极作用的测试用例特征。本章通过测试用例的语句覆盖信息来评价其质量，并提出基于覆盖特征与随机测试的缺陷定位测试方法（COverage-based and Random approach for FLP, COR）来解决 FLP 问题。具体来说，我们根据作为缺陷定位输入的测试用例对缺陷定位方法产生的影响，提出并量化了两个对缺陷定位起积极作用的测试用例覆盖特征，即缺陷位置预测结果多样化特征与失效趋向性特征。此外，该方法还融入了随机测试策略，保证了方法的稳定性。实验证明，COR 能够在面向缺陷定位的测试过程中对测试用例的质量进行有效地评估，并提升了软件缺陷定位的效率。

3.1　问题分析与描述

通常情况下，软件缺陷定位方法通常需要足够多的测试用例作为输入，以对缺陷代码位置进行较为准确的推断。这就需要在缺陷定位开始之前预先执行大量的测试用例，以得到较为充足的测试信息。然而，在很多大规模项目中，执行大量的测试用例消耗的时间会非常漫长，有时这么做本身就是不切实际的。因此，在实际软件开发环境（如持续集成环境）中，测试信息的扩充与缺陷定位是并行执行的。

考虑测软件 PG，其语句的集合记为 S，测试用例记为 t，测试用例的集合记为 T。对任意测试用例 t，C^t 为其覆盖信息，而 o^t 为其执行结果。本章中，我们设测试用例全集为 T^a，当前已经被执行了的测试用例集为 T^e，余下没有被执行的测试用例集为 T^r。假设调试人员采用程序谱缺陷定位方法（Spectra-Based Fault Localization, SBFL）进行缺陷定位。设 SBFL 给出的缺陷预测结果为 L。

上述情境下，测试与缺陷定位的执行流程如图3.1所示。在测试过程中，当发现一个失败测试用例时则直接转入缺陷定位流程。此时 SBFL 接受当前已经执行的测试用例 T^e，并且给出缺陷位置预测结果 L。与此同时，新的测试用例也在不断地被执行。每执行一个测试用例，测试人员通过相应的工具获取其测试信息，并将该其加入到 T^e 中。由于 T^e 变化了，SBFL 给出的结果 L 会被重新计算，而新的结果也将实时提供给调试人员。

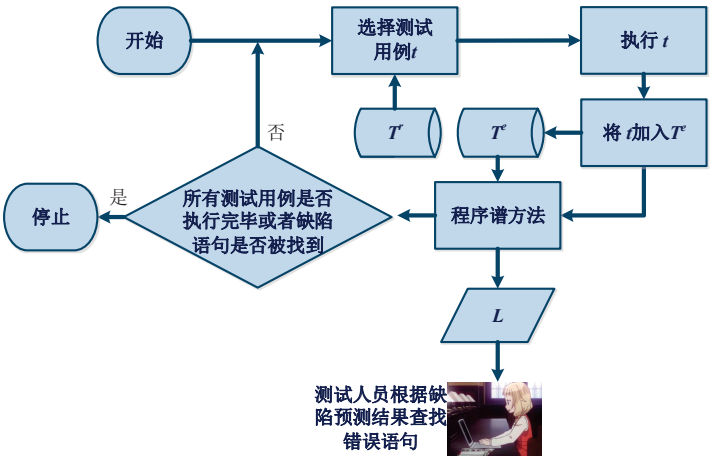


图 3.1　缺陷定位测试用例优化问题描述

通过此办法，尽管在初始状态下由于 T^e 中的测试用例比较少，SBFL 输出的缺陷位置预测可能会不准确，但是随着更多的测试用例被执行，L 的准确度也会随之提升。而一旦程序员发现了缺陷语句，那么就可以直接转入到缺陷修复当中，剩下 T^r 中的测试用例就不需要被执行了（至少是不需要再以定位已经找到的缺陷代码为目的执行了）。该流程符合调试人员排查缺陷的习惯——当错误被发现时先依靠经验与直觉搜索可能出错的地方，并在搜索过程中借助自动化缺陷定位技术给出的缺陷位置预测结果将搜索范围进行细化；此外还能够节约大量测试用例执行的时间。在以上开发环境及流程的想定下，FLP 问题的含义如下：

若目标软件在测试过程中发生失效，那么该如何决定剩余测试用例的执行优先级，使得调试人员能够更好地对引发失效的缺陷代码进行定位？

换言之，FLP 问题旨在定义一个最优的测试用例执行序列，使得那些对缺陷定位有帮助的测试用例优先被执行。如果缺陷定位方法给出的缺陷位置预测结果的准确度能够快速提升，那么在其帮助下，缺陷语句就会更有效率地被找到了，进而整个缺陷定位环节的执行效率也就得到了提升。

尽管在软件测试领域已经有很多较为成熟的测试用例优化方法，然而这些方法主要面向软件测试的基本目标，如发现错误、评估可靠性等；而与面向缺陷定位的测试用例优化技术相关的工作还比较少。其中，较为著名的是 Yoo 等人[47] 于 2013 年提出的基于信息熵的方法（Fault Localization using Information Theory, FLINT）。FLINT 方法利用测试的覆盖信息去计算当前软件缺陷定位结果的信息熵，并以此来衡量每个

测试用例的价值。然而，该方法并没有试图通过分析测试用例与缺陷预测结果的相关性，来探索对软件缺陷定位是有意义的测试用例特征。

为此，本章提出了基于覆盖特征与随机测试的缺陷定位测试方法（COverage-based and Random approach for FLP, COR）。COR 通过测试用例覆盖信息是否具备特定的特征，即 1）缺陷位置预测结果多样化特征与 2）失效趋向性特征，来评价一个测试用例在缺陷定位问题上的质量。本章对这两个特征与缺陷定位的关系进行详细分析，论证它们的合理性，设计相应的标准对测试用例满足这些特征的程度进行度量，并以此为依据设计相应的 FLP 自适应机制。此外，COR 还结合了随机测试的理念，在测试选取过程中加入随机因素，提升了方法的稳定性，降低了算法的复杂度。

3.2 FLP 定义与现有方法分析

3.2.1 FLP 定义

测试用例优化（Test Case Prioritization, TCP）技术是软件测试领域被广泛研究与应用的经典技术，其旨在通过合理地安排测试用例的生成或执行顺序，来提升软件测试效率。以此为借鉴，缺陷定位测试用例优化问题（即 FLP 问题）的定义如下：

定义 3.1 (**缺陷定位测试用例优化问题** (FLP))：

假设： 1）已知测试用例全集为 $T^a = T^e \cup T^r$，其中 T^e 表示已经执行过的测试用例集合，其包含至少一个失效的测试用例，T^r 表示还未执行的剩余测试用例集合。2）SBFL 以 T^e 为输入进行缺陷位置预测的相关运算并输出缺陷位置预测结果 L。这里 T^r 中的测试用例由于没有被执行，我们无法知道其输出结果是否失效，因此它们无法被缺陷定位方法直接使用。3）若 T^r 中的测试用例还将继续被执行，并且每执行一个新的测试用例 t，其执行结果将被记录下来，该测试用例也将从 T^r 被移入 T^e 中。而每当 T^e 发生变化，SBFL 也会将新的 T^e 作为输入重新被执行，并且更新缺陷位置预测结果 L

问题： 在 T^r 中找到一组次序置换 $Pe(T^r)$，使得当测试用例依照 $Pe(T^r)$ 中的测试用例顺序执行时，L 的准确度能够以最快的速率上升。

假设调试人员采用 SBFL 进行缺陷定位，则每当一条新测试用例 $t \in T^r$ 执行完毕，其会添加到 T^e 的测试信息中，而 SBFL 的结果也将被更新。假设第 i 条测试用例执行后得到的缺陷位置预测结果为 L^i，而评价其性能的 Expense 值为 E^{L^i}；假设缺陷定位

开始时 T^r 中有 m 条测试用例，则 T^r 中所有测试用例都执行完毕后，我们会得到一系列缺陷位置预测结果 $L^1, L^2, \ldots, L^m$，其对应的 Expense 值为 $E^{L^1}, E^{L^2}, \ldots, E^{L^m}$，则整个过程的效率可以通过这些 Expense 值的平均（Average Expense，AE）来表示：

$$AE = \frac{\sum_{i=1}^{m} E^{L^i}}{m} \tag{3.1}$$

如果 AE 值很低，则说明 SBFL 的 Expense 值下降速率很快；反之，如果 AE 值很高的话，整个 FLP 效果就不会很理想。这种度量方法在测试用例优先级的相关研究中被广泛采用[47, 104]。

3.2.2　信息熵方法分析

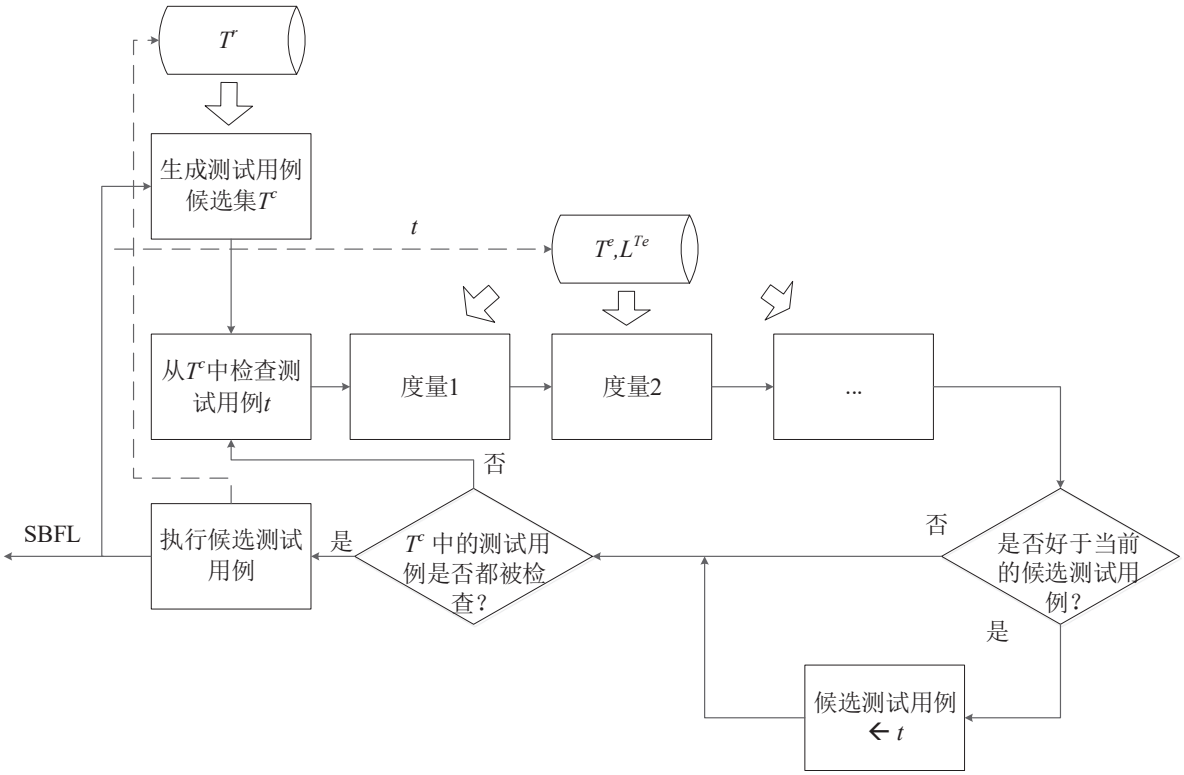


图 3.2　基于覆盖信息的缺陷定位测试用例优化自适应机制执行流程

FLP 本质上是个自适应问题，其需要根据缺陷定位过程中的实时信息，对当前测试信息的分布与其对缺陷代码的暴露情况进行观测，并以此安排后续的测试进程。近年来 Yoo 等人[47] 提出了 FLP 问题，并尝试引入信息论领域的知识对其进行求解，即软件缺陷问题的信息论方法（Fault Localization using Information Theory, FLINT）。

对于剩余测试用例集 T^r 中的每一个测试用例 t，FLINT 试图计算其加入到 T^e 后

得到缺陷位置预测 $L^{T^e\cup\{t\}}$ 的信息熵，并且选择 T^r 中那些能够最大程度减小信息熵的测试用例优先执行。

由于 T^r 中的测试用例都还没有被执行，因此他们不能直接作为输入而被 SBFL 方法所接受，这使得 $L^{T^e\cup\{t\}}$ 的准确值无法实现计算，因此 FLINT 尝试对其进行估计。首先，对于任意 $t\in T^r$，其 o^t 的预测如下：

$$\begin{aligned}\hat{p}(o^t=passing)&=|\{u|u\in T^e,o^t=passing\}|/|T^e|\\\hat{p}(o^t=failing)&=1-\hat{p}(o^t=passing)\end{aligned}\tag{3.2}$$

这里 $\hat{p}(o^t=passing)$ 与 $\hat{p}(o^t=failing)$ 分别是测试用例 t 成功与失效的概率估计值。现设 $L_p^{T^e\cup\{t\}}$ 与 $L_f^{T^e\cup\{t\}}$ 分别为当测试用例成功与失效时将其考虑在内而得到的缺陷位置预测 $L^{T^e\cup\{t\}}$。则 $L^{T^e\cup\{t\}}$ 的估计值如下：

$$\hat{L}^{T^e\cup\{t\}}=[\lambda_1^{\hat{L}^{T^e\cup\{t\}}}\lambda_2^{\hat{L}^{T^e\cup\{t\}}}\dots\lambda_n^{\hat{L}^{T^e\cup\{t\}}}]\tag{3.3}$$

其中

$$\begin{aligned}\lambda_k^{\hat{L}^{T^e\cup\{t\}}}&=<s_k^{\hat{L}^{T^e\cup\{t\}}},r_k^{\hat{L}^{T^e\cup\{t\}}}>\\r_k^{\hat{L}^{T^e\cup\{t\}}}&=\hat{p}(o^t=passing)r_l^{L_p^{T^e\cup\{t\}}}+\hat{p}(o^t=failing)r_j^{L_f^{T^e\cup\{t\}}}\\s_k^{\hat{L}^{T^e\cup\{t\}}}&=s_l^{L_p^{T^e\cup\{t\}}}=s_j^{L_f^{T^e\cup\{t\}}}\end{aligned}$$

以此为基础，执行测试用例 $t\in T^r$ 后，预期获得缺陷预测结果 $\hat{L}^{T^e\cup\{t\}}$ 的熵为：

$$entropy(t,T^e)=-\sum_{n}^{k=1}\log(r_k^{\hat{L}^{T^e\cup\{t\}}})\tag{3.4}$$

FLINT 方法的基本思想是如果缺陷定位方法给出的缺陷预测结果的信息熵越低，说明其包含的不确定性就越少，准确度就越高。因此，优先执行那些使得信息熵减小的测试用例就是合理的。然而，信息熵的方法基于测试用例的覆盖信息，但其没有明确给出测试用例覆盖信息与缺陷预测结果之间的联系，从而缺乏对 FLP 问题中，测试用例的执行对 SBFL 结果的作用关系的剖析。本章后续内容将针对此问题进行分析及方法改善。

3.3　基于覆盖信息的缺陷定位测试用例优化方法设计

基于覆盖信息的缺陷定位测试策略（COR）假设软件已经进行了至少一轮的更新迭代周期，其已经拥有了一个测试用例执行的全集 T^a。对于这个全集中的任意测试用例 t，其覆盖信息 C^t 都是已知的。此假设在很多关于回归测试的研究中均被采用。然而，由于程序进行了更新，测试用例 t 的测试结果 o^t，还需要重新检验。

COR 方法的流程如图3.2所示。COR 利用测试用例全集 T^a 中的覆盖信息以及已执行测试用例集 T^e 中的执行结果信息对未执行测试用例 T^r 中的测试用例进行排序。COR 通过一系列度量准则来评估一个测试用例在软件缺陷定位中的质量。对于一个新的未执行测试用例 $t \in T^r$，每一个度量都反映了其覆盖信息 C^t 对某一覆盖特征的隶属程度，而这些覆盖特征是在启发式意义下对软件缺陷定位起促进作用的。然而，由于软件与程序缺陷的多样性，我们不能保证这些特征永远对缺陷定位有益处，因此 COR 还引入了随机测试思想，以提升其稳定性。具体来说，当选择接下来要执行的测试用例时，COR 先从 T^r 中随机地抽取测试用例，生成候选集 T^c，并在此候选集所包含的测试用例中进行选择。最终，在所提出的度量下满足所考察的覆盖特征程度最高的测试用例将被选中。

COR 方法的关键是如何选择对缺陷定位起积极作用的测试覆盖特征并对这些特征进行度量。本章节中，我们根据测试输入信息对软件缺陷定位结果的作用机理提出以下两个覆盖特征：

（1）缺陷预测结果多样化特征，即覆盖信息能够提升当前 SBFL 软件缺陷预测结果的可疑度列表多样性；

（2）失效趋向性特征，即覆盖信息使得该测试用例更趋向于失效测试用例。

在本章后续内容将对这两个特征进行描述与度量，并阐述其合理性，进而提出完整的 COR 方法。

3.3.1　缺陷位置预测结果多样化特征及其度量

在自动化测试与调试中，多样性一直是一个非常重要的概念。在软件测试领域，测试用例输入的多样性以及覆盖信息的多样性在测试方法相关的研究工作中被广泛讨论。例如，一些传统测试用例优化方法会试图选择那些能够覆盖到更多语句的测试用例以丰富语句覆盖的多样性[113]。而在软件缺陷定位中，多样性也非常重要的。如第二章所述，缺陷语句 s^f 一定会被至少一个失效测试用例所覆盖，即满足

$$s^f \in \bigcup_{t \in T^e, o^t = failing} C^t \tag{3.5}$$

若进一步假设程序中只包含一个缺陷（即单缺陷假设），则 s^f 的位置就会更严格地被限制——其一定包含在所有失效测试用例的交集中，即

$$s^f \in \bigcap_{t \in T^e, o^t = failing} C^t \tag{3.6}$$

这里，记 $I^f = \bigcap_{t \in T^e, o^t = failing} C^t$，则在讨论多样性问题时只有 I^f 中的语句会被涉及。在单缺陷假设下，SBFL 的本质是区分那些属于 I^f 的语句的缺陷可疑度。也就是说，I^f 中的语句区分度越高，缺陷定位的精度就越高。由此，SBFL 所涉及的多样性问题就可以表述为：

SBFL 所给出的缺陷位置预测结果中能够区分不同语句包含缺陷可能性的程度。

为了更加准确地描述一个 SBFL 可疑度列表的多样性，这里引入“子列”这一概念。设 SBFL 的结果为一可疑度列表 L（本章中的可疑度列表如果没有特殊说明均基于 T^e 中的测试用例而得到，因此为了更简洁我们省略定义中上标 T^e）。将 L 分为若干个组，满足：1）同一个组中的语句具有相同的可疑度值；2）不同组中的语句具有不同的可疑度值。此时，每一个组就可以定义为 L 中的一个“子列”，记为 Sl，其定义式如下：

$$L = [Sl_1^L Sl_2^L \dots Sl_v^L] \tag{3.7}$$

满足：

1) 对任意 $Sl_l^L = [\lambda_k^L \ \lambda_{k+1}^L \dots \lambda_{k+n_l-1}^L]$，有 $r_k^L = r_{k+1}^L = \dots = r_{k+n_l-1}^L$；

2) 对任意 $j < l$，且对 Sl_j^L 中的任意 λ_k^L，以及 Sl_l^L 中的任意 λ_h^L，有 $r_k^L > r_h^L$。

显然，如果两条语句处在可疑度列表的同一子列中，则它们在 SBFL 的缺陷位置预测结果中包含缺陷的可能性相同，进而不能被区分开来。而称一个新的测试用例具有很高的缺陷预测结果多样化特征，就是说该测试用例的语句覆盖信息具备能够区分当前缺陷位置预测结果 L^{T^e} 中隶属于相同子列的语句的能力。这里我们只讨论 I^f 中的语句（具体见公式 (3.6)）。

现考虑一测试用例 t 与一个子列 Sl_k^L，若 t 的覆盖信息 C^t 覆盖了 Sl_k^L 中的一部分

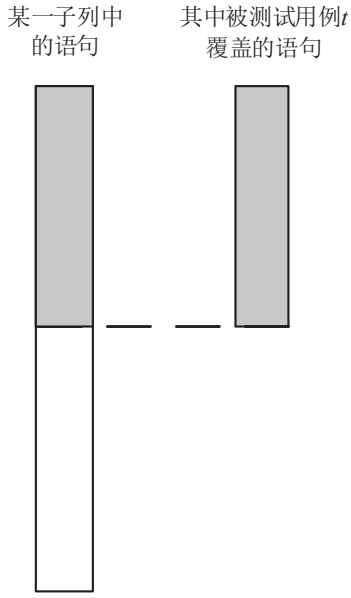


图 3.3　测试用例区分一个子列示意图

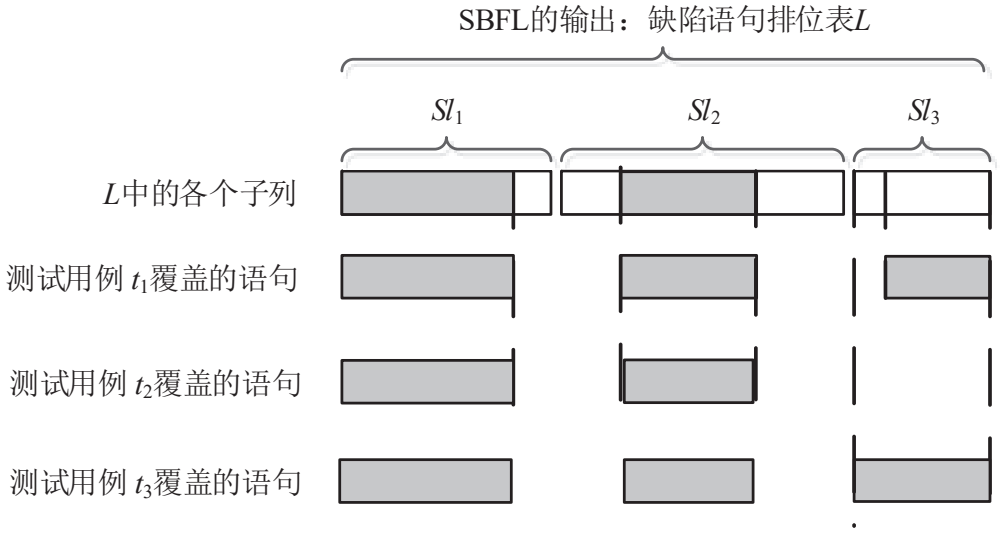


图 3.4　不同测试用例区分子列示意图

语句且没有覆盖全部语句，则我们称 t 能够区分 Sl_k^L，记作 $dist(t, Sl_k^L) = 1$，如图3.3所

示。这里 $dist$ 为子列区分函数，其定义式如下：

$$dist(t, Sl_k^L) = \begin{cases} 1 & \text{条件 3.1 被满足} \\ 0 & \text{条件 3.1 不被满足} \end{cases} \tag{3.8}$$

条件 3.1：$\exists s \in Sl_k^L$ 满足 $s \in C^t$，且 $\exists s' \in Sl_k^L$ 满足 $s' \notin C^t$。

例如，图3.4中可疑度列表 L 包含三个子列 Sl_1，Sl_2 和 Sl_3，现考察 T^r 中的三个新测试用例 t_1，t_2 和 t_3。从图中不难发现，测试用例 t_1 能够区分所有这三个子列。而对于测试用例 t_2 和 t_3 来说，测试用例 t_2 所覆盖的语句全部都不属于 Sl_3；相反，测试用例 t_3 却覆盖了 Sl_3 中所有的语句，因此它们都不能够区分子列 Sl_3。

以此为依据，一个测试用例具备缺陷预测结果多样化特征的程度就可以通过其能够区分的子列的个数进行度量。设根据已执行测试用例 T^e 所给出的当前缺陷位置预测结果为 L^{T^e}，现在 T^r 中考察一新的未执行测试用例 t，其缺陷预测结果多样化特征的度量（Diversity Measure，DM）的定义式为：

$$DM(t, T^e) = \sum_{k=1}^{v} dist(t, Sl_k^{L^{T^e}}) \tag{3.9}$$

3.3.2 失效趋向性特征及其度量

作为 SBFL 的基本假设之一，失效的测试用例在缺陷定位中的作用要大于成功执行的测试用例。如上节中式 (3.5) 所描述的那样，缺陷语句至少被一条失效测试用例所覆盖，这是一个确定的结论，这一点在第二章的论述中也有提到。因此，失效趋向性，即一个测试用例失效的可能性也能够作为衡量该测试用例质量的标准。需指出，由于在对测试用例质量进行评估时，该测试用例并没有真正意义上地被执行，因此我们并不知道其实际失效与否，只能根据其覆盖信息去估计其失效的可能性。也就是说，在对一测试用例进行考察时，其覆盖特征使其越像是失效测试用例，其执行价值就越大。在本小节中，我们将基于测试用例的覆盖信息对其产生失效的可能性进行评估，并将评估的结果作为其失效趋向性特征的度量。

对于任意已经执行的测试用例 $t \in T^e$，假设其覆盖信息 C^t 与执行结果信息 o^t 已知，则可以基于 T^e 运行 SBFL 并得到作为缺陷定位结果的可疑度列表 L^{T^e}。现依据此列表对当前的每条语句 s 中包含缺陷的可能性做初步估计，其估计方法见式 (3.10)：

$$\hat{p}(s = s^f | T^e) = r_k^{L^{T^e}} \tag{3.10}$$

这里

$$\lambda_k^{L^{T^e}} =< s_k^{L^{T^e}}, r_k^{L^{T^e}} >$$
$$s_k^{L^{T^e}} = s$$

现进一步假设语句 s 就是缺陷语句（即 $s = s^f$），则考察一个未执行的新测试用例 $t \in T^r$ 时，其失效率的估计如下：

$$\hat{p}(o^t = failing | s = s^f) = \begin{cases} q_e^{s,T^e} & s \in C^t \\ 0 & s \notin C^t \end{cases} \tag{3.11}$$

这里

$$q_e^{s,T^e} = a_{ef}^{s,T^e} / (a_{ef}^{s,T^e} + a_{ep}^{s,T^e})$$

在式 (3.11) 中，在语句 s 为缺陷语句的条件下，如果测试用例 t 没有覆盖语句 s，则其失效率为 0；反之，若其覆盖了语句 s，则其失效概率可以通过基于 T^e 的先验概率做估计。因此，当 $s = s^f$ 假设成立时，该估计在概率意义上是准确的。最终，我们通过全概率公式对测试用例 t 的失效可能性做如下估计：

$$\hat{p}(o^t = failing | T^e) \approx \sum_{s \in S} p(o^t = failing | s = s^f) \hat{p}(s = s^f | T^e) \tag{3.12}$$

公式 (3.10) 中关于 $\hat{p}(s = s^f | T^e) = r_k^{L^{T^e}}$ 的计算将程序谱方法给出的可疑度信息看作对语句 s 包含缺陷的一个粗略估计；尽管可能不是很准确，但其比式 (3.2) 中仅仅通过基于 T^e 的先验概率估计要合理。首先，当 T^e 中只有一条测试用例时，失效率不至于被武断地估计为 1；此外，由于本文的估计还基于 T^r 中测试用例的覆盖信息，因此对于 T^r 中不同的测试用例，其估计值也是不同的。该方法的基本思想还将在后面的章节中使用。

最终，考察测试用例 $t \in T^r$，其失效趋向性特征的度量（Fail-Like Measure，FM）如下：

$$FM(t, T^e) = \hat{p}(o^t = failing | T^e) \tag{3.13}$$

The COR approach:
Input: T^e, T^r, L^{T_e}
Output: The next executed test caes t^*
1. if $T^r = \varnothing$ **,return** null**;**
2. Set $T^c \leftarrow \{t \mid t$ is randomly chosen from T^r, $|C|=10\}$;
3. $dm^* \leftarrow -1$, and $fm^* \leftarrow -1$;
4. foreach $t \in T^c$
5. Calculate $DM(t,T^e)$;
6. Calculate $FM(t,T^e)$;
7. **if** $DM(t,T^e) > dm^*$
8. $t^* \leftarrow t$; $dm^* \leftarrow DM(t,T^e)$; $pm^* \leftarrow FM(t,T_e)$;
9. **else**
10. **if** $DM(t,T^e) = dm^*$
11. **if** $FM(t,T^e) > fm^*$
12. $t^* \leftarrow t$; $dm^* \leftarrow DM(t,T^e)$; $fm^* \leftarrow FM(t,T^e)$;
13. return t^*

图 3.5　基于覆盖信息的缺陷定位测试用例优化自适应方法执行过程

3.3.3　基于覆盖信息与随机测试的缺陷定位测试用例优化方法

基于覆盖特征与随机测试的（COverage-based Random，COR）自适应方法旨在解决面向缺陷定位的测试用例优化问题，其通过对 SBFL 给出的缺陷位置预测结果进行实时观测，了解当前缺陷定位结果的特征，并以此为依据评估待执行测试用例能够改善当前缺陷位置预测结果的能力，最终在剩余测试用例集中选择具有高 DM 和 FM 值的那些测试用例并优先执行，以快速提升 SBFL 的效率。

图3.5描述了整个 COR 方法的执行过程，其输入为已经执行的测试用例集 T^e、未执行的剩余测试用例集 T^r 以及 SBFL 给出的当前可疑度排位表 L^{T^e}。首先，从 T^r 中随机选出一候选测试用例集 T^c。接下来对于 T^c 中的每一个测试用例，COR 方法会计算其 DM 与 FM 值，以衡量其价值。在评判过程中，DM 会被优先考虑；具体来说，在比较两个测试用例质量时，先看它们的 DM 值，DM 值高的测试用例具有更高的质量，如果二者 DM 值相同，则 FM 值高的测试用例会被赋予更高的优先级。这样的操作是因为测试信息多样性是保证缺陷定位准确度的基本特征；相反如果多样性特征不能得到保证，那么测试用例本身是成功还是失效都是没有意义的。最后，T^c 中具有最高的质量估值的测试用例会被赋予最高的优先级，并被选为接下来要执行的测试用例。

在整个缺陷定位测试过程中，COR 方法会被持续执行，直到 T^r 中所有的测试用例都被执行完毕，亦或是缺陷代码已经被成功定位时停止。

需指出的是，COR 方法并不仅限于 DM 与 FM 度量，其本身是一个方法框架。任何基于测试覆盖信息的度量均可以被考虑。因此，COR 具有较强的可扩展性。此外，分析测试覆盖特征与缺陷定位方法性能的关系，使得软件缺陷定位方法的机理更容易被理解，而以此设计的自适应机制也能够从本质上提升缺陷定位方法的适应性，这对软件缺陷定位方法的广泛应用是必不可少的工作。

3.3.4　COR 方法应用举例

本小节以实际程序上的缺陷定位过程为例，来更为直观地阐明 COR 方法的机理，以及采用 COR 方法进行缺陷定位测试的收益。

例 3.1：本例子主要用来体现缺陷定位结果多样化特征度量（DM）的作用。

假设原始程序、其缺陷版本以及其现有的测试信息如图3.6所示。从图中可以看出，该程序共有 7 条可执行语句 $s_1, s_2, \ldots, s_7$，其中 s_6 是缺陷语句。具体来说，程序员将 *C=A*写成了 *C=1*。与此同时，该程序一共有 5 个测试用例，即 $T^a = \{t_1, t_2, \ldots, t_5\}$，其中已经执行的测试用例集为 $T^e = \{t_1, t_2, t_3\}$，未执行的测试用例集为 $T^r = \{t_4, t_5\}$。这里黑点表示各测试用例所覆盖的语句，而覆盖信息表的最后一栏中 P 与 F 分别表示其执行结果成功与失效。而 $r^{L^{T^e}}$ 列中的数值表示 SBFL 依据当前 T^e 中的测试信息所计算出的各语句的可疑度（采用 Op 公式）。

从图中可以看出，语句 s_5 被赋予了最高的可疑度。此时如若采用 FLINT 方法进行测试，其会趋向于维持并加强当前的可疑度列表，因此 t_4 会被选中。然而，t_4 与 t_3 具有相同的覆盖特征，其并没有引入新的有用信息。就结果而言，当前的缺陷列表并没有被改变，而真正的缺陷语句 s_6 的排位也没有被提前。而在 COR 方法中，t_5 具有更高的 DM 值，因为其能够区分当前最大的子列，即语句 s_2、s_3 、s_4 与 s_6 所组成的子列。执行了 t_5 后，真正的缺陷语句 s_6 会排在 s_2 与 s_3 之前，进而使得整个缺陷定位的准确度得到提升。从这个例子可以看出，测试用例对可疑度列表的多样化过程更本质地反映了测试用例与缺陷定位结果之间的作用机制，而以此设计的 DM 度量也能够更为有效地评估测试用例在缺陷定位环节中的质量。

例 3.2：本例子主要用来体现失效趋向性特征及其度量 FM 的有效性。

假设原始程序、其缺陷版本以及测试信息如图3.7所示。该程序共有 13 条可执行语

缺陷定位结果多样化特征示例程序

原始程序	缺陷版本

Input (*R*, *A*, *Al*){
int *L*, *C*, *S* = 0;
if (*R* > 5){
L=1;
if (*R* > 10){
if(*Al* =true)
S=1;
C = *A*;}}
return [*L*, *C*, *S*];
	Input (*R*, *A*, *Al*){
	int *L*, *C*, *S* = 0;
	if (*R* > 5){
	L=1;
	if (*R* > 10){
	if(*Al* =true)
	S=1;
	C = 1;}}
	return [*L*, *C*, *S*];

测试用例覆盖信息、输出结果
以及当前语句的可疑度

	t_1	t_2	t_3		t_4	t_5
	I:12, 1, false	I: 4, 2 true	I:12, 0, true	r^{L^e}	I:15, 1, true	I:8,1 , true
	•	•	•	1	•	•
s_1	•	•	•	1	•	•
s_2	•		•	2	•	•
s_3	•		•	2	•	•
s_4	•		•	2	•	
s_5			•	3	•	
$s_6 = s^f$	•		•	2	•	
s_7	•	•	•	1	•	•
	P	*P*	*F*		*P*	*P*

图 3.6　COR 方法中多样化特征及其度量示例

句 $s_1, s_2, \ldots, s_{13}$，其中 s_4 是缺陷语句，具体来说，程序员将 *Ob=1*写成了 *Ob=2*。该程序一共有 3 条测试用例，即 $T^a = \{t_1, t_2, t_3\}$，其中已经执行的测试用例集为 $T^e = \{t_1\}$，未执行的测试用例集为 $T^r = \{t_2, t_3\}$。图中各符号所代表的含义与图 3.6 相同。

从图中可以看出，已执行测试用例集 T^e 只包含一条失效测试用例 t_1，也就是说，T^e 中的先验失效率为 1；显然由于测试信息不足，此概率非常不准确。若仅仅以此作为 T^r 中测试失效率的估计值，那么 t_2 与 t_3 具有相同的失效率，而按照 FLINT 方法，t_2 将被赋予很高的优先级。然而，t_2 实际上是一条成功测试用例；运行 t_2 后，语句 s_1 的排位会提升，而缺陷语句 s_4 的排位会下降，整个缺陷定位结果也会变差。

反观 COR 方法，t_2 与 t_3 具有相同的 DM 值，并且 t_3 具有更高的 FM 值，即 t_3 更趋向于失效测试用例。因此，相比于 t_2，t_3 会被赋予更高的优先级，并最终被选中。

失效趋向性特征示例程序及其缺陷版本

Original Program	**Faulty Version**

Input (*Coo*, *L*, *Ob,R*){
double y_1, y_2,*cl*;
if (*Ob*< 0){
Return null}
If (*R*= 1)
If(*Ob* =1)
{y_1= *Coo*;
y_2= *Coo* +L;
cl = 'r';
Return [*y1 y2 cl*];
Else
{y_1= *Coo-5L*;
Y_2= *Coo-5/L*;
cl = 'y';
Return [*y1 y2 cl*];
}
 	
else
Return [Coo, Coo, '']
	Input (*Coo*, *L*, *Ob,R*){
	double y_1, y_2,*cl*;
	if (*Ob*< 0){
	Return null}
	If (*R*=1)
	If(*Ob* =2)
	{y_1= *Coo*;
	y_2= *Coo* +L;
	cl = 'r';
	Return [*y1 y2 cl sh*];
	Else
	{y_1= *Coo-5/L*;
	y_2= *Coo-5/L*;
	cl = 'y';
	Return [*y1 y2 cl*];
	}
	
	Else
	Return [Coo, Coo, '']

测试用例覆盖信息、输出结果
以及当前语句的可疑度

	t_1	$r^{L_{Tc}}$	t_2	t_3
	I:1,2,1, 1,		I:1,1,0, 1	I:3,3,2 ,1,
s_1	•	1	•	•
s_2		0	•	
s_3	•	1		•
s $_4$***=f***	•	1		•
s_5	•	1		
s_6	•	1		
s_7	•	1		
s_8	•	1		
s_9		0		•
s_{10}		0		•
s_{11}		0		•
s_{12}		0		•
s_{13}		0		
	F		*P*	*F*

图 3.7　COR 方法中失效趋向性特征及其度量示例

表 3.1 实验对象

目标程序	包含版本	所包含缺陷数	代码行数（LOC）
flex	v1-v5	55	12,438-14,259
grep	v1-v4	19	12,670-13,377
gzip	v1, v4, v5	17	6,592- 8009
sed	v2-v7	23	6,687-12,006

而事实上，t_3 也确实是失效测试用例；其被执行后，语句 s_5、s_6、s_7 与 s_8 的可疑度会下降；相应地，缺陷语句 s_4 的排位会随之上升；整个缺陷定位结果会变得更加准确。从该例子可以看出，COR 中 FM 采用了更为合理的失效率估计，对缺陷定位效率的提升起到了促进作用。

3.4 实验与结果分析

3.4.1 研究问题

本节将通过实验对 COR 方法的有效性与执行效率进行检验。我们将 COR 方法、纯随机测试、以及 FLINT 方法进行比较，在此基础上对 COR 方法的性能进行评估，并对其中随机性的影响进行探究。本节主要针对以下问题展开研究。

研究问题 1：COR 方法能否在缺陷定位测试过程中更快地提升缺陷定位结果的准确度（即 COR 方法的有效性问题）?

研究问题 2：COR 方法的运行是否会额外消耗更多的计算资源（即 COR 方法本身的执行效率问题）?

研究问题 3：如果 COR 方法是有效的，那么促使软件缺陷定位效率提升的因素有那些?

研究问题 4：COR 方法中加入随机因素是否能够切实得到收益?

3.4.2 实验设置

实验采用四个 *UNIX* 程序作为目标程序，分别是 *flex* ，*grep*，*gzip* ，以及 *sed* 。这些源程序从“软件基础信息库（Software Infrastructure System，SIR）[114]”中获取，它们的代码量从 6000 行至 14000 行不等，每个程序都会附带一些标准错误。整个实验中，总共有 114 个错误版本将被考查。实验对象的名称、版本号、代码行数及所包含的缺陷数量由表3.1 给出。

本章考察单缺陷实例，即实验基于单缺陷假设。当对程序的某一缺陷版本进行验

表 3.2　FLP 方法性能比较

目标程序	程序版本号	COR→ 随机测试		FLINT→ 随机测试	
		提升缺陷数	下降缺陷数	提升缺陷数	下降缺陷数
flex	v1	7	2	6	7
	v2	5	1	4	4
	v3	8	0	4	3
	v4	5	0	2	7
	v5	2	0	2	0
grep	v1	4	0	2	2
	v2	2	0	1	1
	v3	4	1	3	4
	v4	3	0	2	1
gzip	v1	3	2	5	0
	v4	3	0	3	0
	v5	1	1	2	2
sed	v2	1	1	0	2
	v3	2	1	1	3
	v4	0	0	0	0
	v5	3	0	2	0
	v6	4	0	0	5
	v7	1	0	0	2
总计		58	9	39	43

证时，我们植入目标缺陷，并且屏蔽掉其它缺陷。此外，我们同样会运行程序的正确版本，并且将测试用例在正确版本下的输出作为测试 Oracle，通过比较输出的一致性来判断在缺陷版本下运行的测试用例是否失效。此外，COR 方法所用到的测试用例覆盖信息通过通用的程序覆盖分析工具 gcov[115] 获得。

对于每个程序的每一个错误，我们都会分别采用 COR，FLINT 以及纯随机测试来进行缺陷定位测试与缺陷定位流程，即整个 FLP 过程（参照第3.1节中图3.1及其相关描述）。对于每个 FLP 实例，我们计算其平均 Expense 值（*AE*）作为度量该 FLP 自适应方法有效性的标准。由于在最初测试阶段，以及在 COR 方法的执行过程中都会存在随机因素，因此，实验中每个实例都会执行 30 次，并且通过 p 值为 0.05 的 t-检验来比较不同方法的优劣。

3.4.3　COR 方法的有效性

作为对**研究问题 1** 的回答，我们将各 FLP 方法进行了 t-检验比较，其结果在表3.2中给出。表中前两列分别是实验对象的程序名和版本号，第三、四列是 COR 方法与纯随机测试的比较，而第五、六列是 FLINT 方法与纯随机测试的比较；其中“提升缺陷数”一栏是采用了相应方法后 *AE* 值降低的（即缺陷定位效率提升）的缺陷个数，而“下降缺陷数”一栏为采用了相应方法后，*AE* 值升高的（即缺陷定位效率下

表 3.3 FLP 方案的执行时间

目标程序	版本号	计算时间（s）		
		随机测试	FLINT	COR
flex	v1	13.8	110.1	29.6
	v2	16.4	102.5	35.7
	v3	13.6	102.8	31.0
	v4	16.1	125.1	33.0
	v5	21.6	128.4	45.8
grep	v1	18.0	167.0	37.8
	v2	19.3	165.5	45.0
	v3	22.8	203.6	50.2
	v4	25.3	202.7	53.3
gzip	v1	1.2	6.7	2.8
	v4	1.7	1.0	4.0
	v5	1.3	5.0	2.4
sed	v2	10.3	45.5	21.5
	v3	3.8	30.3	9.5
	v4	0.0	3.0	0.0
	v5	2.0	20.3	4.8
	v6	1.0	20.0	3.2
	v7	1.8	23.0	4.0
平均时间		10.5	81.2	23.0

表 3.4 CO 与 COR 方法区别

	CO	COR
起积极作用的版本数	50	58
起消极作用的版本数	29	9
平均计算时间（s）	46.7	23.0

降）的缺陷个数。

总结来说，FLINT 在 39 个缺陷版本中比纯随机测试的定位效率要高，但在 43 个缺陷版本中效率反倒下降了。而本章提出的 COR 方法则能有明显地提升软件缺陷定位的效率，很好地解决了 FLP 问题。在所有的缺陷版本中，有 58 个缺陷版本的缺陷定位效率获得了提升，而 COR 方法起负面影响的版本数只有 9 个。

进一步分析，COR 方法在 15 个程序版本中都得到了较好的结果。不过对于不同的程序版本，COR 方法的提升程度是不同的。在一些版本中（例如 *flex.v3*），COR 在所有的缺陷版本下都提升了缺陷定位的效率。而在如 *gzip.v1* 这样的程序中，提升并不明显。可见，即使是自适应机制本身，也会受到软件及缺陷多样性的影响。总的来说，COR 方法通过对 SBFL 即时输出的观测，实时调整测试方法，在 FLP 问题所描绘的流程中，很大程度上提升了缺陷定位的效率。

3.4.4 COR 方法的执行效率

作为对**研究问题 2** 的回答，每种 FLP 解决方案（包括 COR 方法、FLINT 方法以及纯随机测试）的执行时间在表3.3中给出。可以看出，对于不同的实例，由于程序代码量以及测试用例集规模的不同，缺陷定位测试用例优化的执行时间也各不相同。例如，在 *grep* 程序中，纯随机方法要花费超过 20 秒的时间，而对于 *sed* 程序，由于其规模要小于 *grep* ，纯随机方法的执行时间不到 3 秒。

然而，不同 FLP 解决方案所需要的相对时间是存在明显区别的。具体来说，纯随机测试所需的计算时间最少。对于所有的版本，随机测试的平均计算时间为 10.5 秒，在最坏情况下不过 25.3 秒。FLINT 方法花费的时间最长，在最坏的情况下需要 203.6 秒，所有缺陷版本中平均需要 81.2 秒。而对于 COR 方法，尽管执行效率依然不如纯随机测试，但其通过引入了随机思想生成测试用例被选集，节约了大量的计算时间。在最坏实例中，其花费 53 秒；而在所有实例中，其花费的平均时间仅为 23 秒，这与纯随机方法已经很接近了。

总的来说，相对于 FLP 流程中，测试用例本身的执行时间、缺陷定位方法（及 SBFL）的执行时间以及调试人员人工参与的时间，FLP 解决方案所花费的时间非常少。例如，假设 T^r 中包含 1000 条测试用例，那么平均下来，即使速度最慢的 FLINT 方法在每一条测试用例选择上的计算时间还不到 0.2 秒。因此，正常情况下，FLP 解决方案的计算时间代价通常都在可接受范围之内。总的来说，对于各 FLP 方案，纯随机测试的执行效率最高，其次是 COR 方法，而 FLINT 方法的执行效率最低，但各方法的执行效率在解决 FLP 问题中都是可以接受的。

3.4.5 COR 方法执行机理分析

通过上述实验，COR 方法的有效性已经得到了验证。而为了回答**研究问题 3** ，我们对 COR 方法的执行过程进行了详细地分析，以对其提升缺陷定位效果的机理进行进一步探究。图3.8 给出了 COR 方法在一些典型缺陷实例上执行的详细信息。每个子图中，x 轴代表执行测试用例的数量，而 y 轴代表缺陷定位的准确度，即 SBFL 的 Expense 值。其中虚线与实线分别刻画了纯随机测试与 COR 方法的性能。在缺陷定位的最开始阶段，T^e 中只包含在单纯测试阶段所执行的测试用例，而在最后所有测试用例都执行完备时，T^e 的规模就等于所有测试用例的全集 T^a，即 $T^e = T^a$。因此，对于任何 FLP 方法，其在最开始以及整个过程执行完毕的时，缺陷定位的准确性是一样

的，不同的是从开始到结束的过程中缺陷定位准确度的变化。FLP 的性能指标 AE 对应的就是性能曲线与坐标轴围成的面积。从图3.8中可以看出，COR 方法从以下两个方面降低了 SBFL 的 AE 值：

（1）滤除冗余的测试信息

在图3.8(a) 与图3.8(b)中缺陷定位方法最终能够得到很低的 Expense 值。在此类实例中，测试开始时影响 SBFL 性能的是测试信息量，而缺陷定位测试中最关键的问题就是选取有用的测试信息，使得在有限的测试信息量下尽快提升缺陷定位的效率。随着测试的执行，测试用例集包含的有用信息逐渐增多。然而，不是所有测试信息都是有用的。例如一条与前面测试用例的执行信息相同的测试用例就很可能对缺陷定位没有帮助。假设我们采用纯随机测试，这些没有用的冗余测试用例就非常有可能被选中，这样就浪费了很多测试用例的执行时间。而 COR 方法则不然，通过度量 DM 与 FM，我们可以分辨出有用的测试覆盖特征，进而滤除掉那些冗余的测试用例，最终使得 SBFL 给出的 Expense 值迅速下降。

（2）去除一些存在干扰的测试用例

如图3.8(c)与图3.8(d)所示，SBFL 最终的 Expense 值并不是最低的，这种现象表明有一些测试用例的信息是对缺陷定位有误导性的。例如，某些测试用例经过了缺陷语句却没有产生失效（即巧合一致性的测试用例，Coincidental Correctness Test cases，CCT）。而 COR 方法的 DM 度量使得相似的测试用例不会被执行很多次，因此这在一定程度上抑制了产生负面影响的测试用例被反复执行。

最后，为了回答**研究问题 4**，本部分对 COR 方法以及去掉随机因素的自适应方法（即每次不选出候选集，而是度量所有 T^r 中测试用例的 DM 与 FM 值，记该机制为 CO）进行比较。从表3.4中可以看出，CO 在 50 个缺陷版本中比纯随机测试要好，在 29 个版本中产生负面影响；而 COR 方法在 58 个缺陷版本中取得了提升，而下降的只有 9 个缺陷版本。此外，COR 的执行时间也比 CO 缩短了近一倍。总的来说，添加随机机制以后，COR 在性能与执行时间上都取得了很好的效果。

3.5 潜在的风险

COR 方法的一个潜在的风险是其对软件缺陷定位方法本身的适应性。本文只考察了 SBFL 的 Op 公式，而在其它可疑度计算公式中，很多不属于 I^f 的语句也有可能

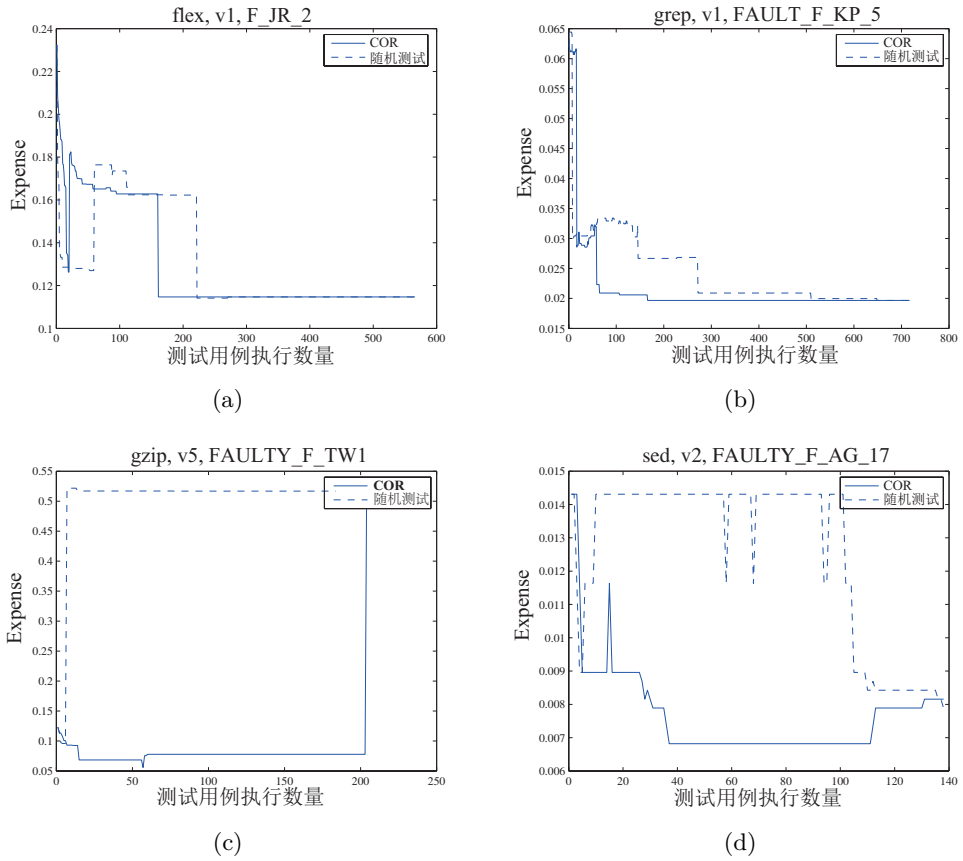


图 3.8　T^r 中测试用例执行数量与缺陷定位准确度的关系

会被赋予较高的可疑度值。此时，引入适当的过滤机制，对缺陷可疑度列表进行噪声过滤就可能成为必要的步骤。在实际执行时，我们可以根据所选择的公式，在可疑度列表中设置一个阈值，并在阈值范围内定义子列。例如，我们将阈值设置为 δ（δ 为正数），则在定义子列时可用公式：

$$L = [Sl_1^L\ Sl_2^L \ldots Sl_v^L] \tag{3.14}$$

满足：

1) 对任意 $Sl_l^L = [\lambda_k^L\ \lambda_{k+1}^L \ldots \lambda_{k+n_l-1}^L]$ 中的任意 r_{k+u}^L，有 $r_{k+u+1}^L - \delta < r_{k+u}^L < r_{k+u+1}^L + \delta$（这里 $u < n_l$）；

2) 对任意 $j < l$，且对 Sl_j^L 中的任意 λ_k^L，与 Sl_l^L 中的任意 λ_h^L，有 $r_k^L > r_h^L + \delta$。

用公式 (3.14) 计算子列，并且进一步计算测试用例的 DM 值，可以在一定程度上缓解由不同可疑度计算公式带来的适应性问题。

另一个使用风险是 COR 方法需要用到测试用例的覆盖信息，而在实际评估某条

测试用例质量时，它并没有被执行过，因此实际过程中测试覆盖信息的获取很可能会存在困难。然而，在实际的回归测试中，调试人员可以从其历史执行数据来获取其覆盖信息。由于 COR 方法主要是基于缺陷定位列表多样性的计算，因此其对于历史覆盖信息的少许偏差是有较高的容忍度的。此外，即使测试用例覆盖信息无法预先获得，COR 方法也存在其价值。在很多情况下，程序输出的正确性需要通过测试与调试人员手动去判别，因此即使一条测试用例执行完毕，要想将其用于软件缺陷定位，仍需要花费较长的时间，在此期间，所有的测试用例的覆盖信息已知，而调试人员就可以通过运行 COR 方法，优先对那些对缺陷定位有帮助的测试用例进行判断了。此外，COR 中的随机因素也可以减轻由于信息的偏差所带来的负面影响，从而在一定程度上提升了方法的鲁棒性。而在后面的章节中，我们也会对测试用例的覆盖信息无法取得时的 FLP 问题及其解决方案做进一步研究。

3.6 本章小结

本章提出 COR 方法去解决面向缺陷定位的自适应测试问题。COR 方法利用测试的覆盖特征，通过对 SBFL 给出的缺陷位置预测结果进行实时观测，动态对当前缺陷定位的状态进行识别，根据识别的结果，COR 方法通过相应的评判准则，选择那些对缺陷定位有用的测试用例优先执行，从而在整个流程中提升了缺陷定位效率。具体来说，COR 方法的贡献如下：

（1）通过分析测试用例覆盖信息与软件缺陷定位结果的关系，识别对软件缺陷定位起到促进作用的测试覆盖特征；

（2）提出了这些覆盖特征的量化与度量标准；

（3）COR 方法引入随机因素，提升了方法执行效率，并且在一定程度上减小了方法本身的适应性问题带来的负面影响。

总的来说，本章提出了直接基于 SBFL 缺陷定位方法运行机制的 FLP 问题的解决方案。而 COR 方法本身也是一个框架，其具有良好的可扩展性。通过对 SBFL 机理的进一步分析，更多对缺陷定位起积极作用的测试覆盖信息特征也都可以纳入其中。其对 FLP 相关自适应机制的进一步研究起到很好的指引作用。

第四章　基于输入信息多样性的缺陷定位测试优化方法

4.1　引言

在第三章中，缺陷定位测试用例优化问题旨在通过在缺陷定位过程中调整测试用例的执行顺序，从而提升软件缺陷定位的效率。该问题的实质是根据缺陷定位系统输出的观测值，辨识当前缺陷定位的状态，并以此为依据动态调整测试输入策略的自适应控制问题。第三章提出了解决此问题的 COR 方法，然而该方法需要用到测试用例的覆盖信息，其在测试用例没有执行的情况下是比较难获取的，因此该方法在一些实际情况下不一定适用。以提出具有更强普适性的方法为目的，本章试图利用程序本身的输入空间来量化各个测试用例的特征，并在此基础上评估各个测试用例对缺陷定位的用途，进而提出相应的缺陷定位测试用例优化方法。

在软件测试领域，关于测试用例优化策略的研究工作中，有很多方法是基于程序输入空间进行设计[46, 107]，而这些方法旨在解决发现错误或是可靠性评估这类测试问题。而面向缺陷定位的测试方法中，利用输入空间信息的方法（即黑盒方法）还比较少。本章试图通过范畴划分（Category-Partition Method）来设计面向定位的测试用例优化方法。CPM 是软件测试中的经典方法，其将输入信息划分为若干个范畴（Category），并根据测试用例在各个范畴中的取值选择（Choice）对其特征进行量化[116, 117, 118, 119]。我们称一系列范畴和在范畴上相应的取值选择所构成的组合约束为一个测试框架（Test Frame），而在相同测试框架下的测试用例被视为具有相似的行为。本章通过分析测试用例所隶属的测试框架与缺陷定位准确性之间的关系，优化测试用例的执行过程，进而提升缺陷定位的效率。

在本章方法中，我们将各测试框架量化为一组 Choices 的取值，用不同测试框架之间的差异来度量各测试用例间的距离，并基于此距离空间分别提出基于测试全集、基于成功测试用例以及引入随机机制的缺陷定位测试用例优化算法。

实验证明，这些自适应机制能够很好地解决面向缺陷定位的测试用例优化问题。此外，由于测试框架的能够依据软件的需求分析而构建，其通常是黑盒的。因此，该方法具有较好的可行性与实际应用价值。

4.2　基于程序输入空间的缺陷定位测试优化方法

4.2.1　范畴及取值选择的划分

如前文所述，范畴划分方法（Category Partition Method，CPM）能够将程序输入空间划分成若干子功能单元，其作为很多测试信息处理技术被广泛应用（尤其是在那些具有非数值型输入的程序中）。一个程序实现的任何功能和方法都会依赖于特定的输入参数[117]，而程序的一个范畴（Category）指的就是程序输入参数或环境变量的某个属性。每个范畴都会被分割为若干个取值选择（Choice），其中每个取值选择对应于该范畴的某个特定的取值或取值范围。

通过该划分方法，每个测试用例都对应于各范畴中某个取值选择的组合。给定目标程序 PG，设 Ch 为 PG 所有取值选择的集合，则一个测试用例的输入空间划分信息 Ch^t 就被定义为：

$$Ch^t = \{c | c \in Ch, c\text{被测试用例}t\text{选中}\} \tag{4.1}$$

这里通过一个例子对基于范畴与取值选择的测试空间划分方法做进一步说明。表4.1 的示例程序中包含三个输入参数：$Risk$、$AutoMode$和 $Alarm$，这三个参数用于指代三个范畴。其中 $Risk$ 是个连续变量，而 $AutoMode$ 与 $Alarm$ 分别是两个布尔变量。

因此在进行取值选择的划分时，范畴 $Risk$ 可以被分成三个取值选择，分别是 $Risk \leq 5$、$5 < Risk \leq 10$ 以及 $Risk > 10$；$AutoMode$可以被分成两个取值选择，即 $AutoMode = 0$ 与 $AutoMode = 1$；而 $Alarm$也分成两个取值选择，即 $Alarm = 0$ 与 $Alarm = 1$。由此，最终得到的选择集合为 $Ch = \{Risk \leq 5, 5 < Risk \leq 10, Risk > 10, AutoMode = 0, AutoMode = 1, Alarm = 0, Alarm = 1\}$。假设我们有测试用例 t，其输入为 $< Risk = 12, AutoMode = 1, Alarm = 1 >$，则其所对应的取值选择集合（即测试框架）为：$Ch^t = \{Risk > 10, AutoMode = 1, Alarm = 1\}$。需指出，在分割输入空间时，我们不需要用到覆盖信息，只依靠软件的需求分析相关的信息就可以完成这一步骤。

通常情况下，一个测试用例的输入会反映其很多方面的特征，而本章提出的方法主要基于输入空间划分的三个基本认识：

CPM 基本假设 1：具有相似取值选择的测试用例很可能覆盖相似的语句；

CPM 基本假设 2：取值选择差异较大的测试用例的覆盖信息差异也可能较大；

表 4.1 范畴及取值选择的划分实例

```
1   Input (Risk, AutoMode, Alarm){
2     int Light, Control, Sound = 0;
3     if(Risk> 5){
4       Light=1;
5       if(Risk> 10){
6         if(Alarm= true)
7           Sound=1;
8         Control=AutoMode}};
9     return [Light, Control, Sound];
```

CPM 基本假设 3：失效测试用例与成功测试用例很可能会属于不同的测试框架。

尽管这些假设不是绝对的，但其合理性在各种研究工作中得到证实[120]。

4.2.2 基于程序输入空间划分的缺陷定位测试方法

对于面向缺陷定位的测试方法来说，我们希望其能够评判当前未执行的测试用例可能给缺陷定位准确度的提升带来的预期收益，进而选择那些最有用的测试用例优先执行。记被选为接下来要执行的测试用例为 t^{next}，则一个 FLP 解决方案就可以表示为一函数 G：

$$t_{next} = G(T^e, T^r) \tag{4.2}$$

若仅考虑输入空间划分方法，我们不会用到未执行测试用例的覆盖信息，然而由于程序谱缺陷定位方法（SBFL）本身就会用到已经执行测试的覆盖信息，因此我们假设已经执行的测试用例的覆盖信息是已知的。仍设测试用例的全集为 T^a，已经执行的测试用例集合为 T^e，而未被执行的剩余测试用例结合为 T^r，则如果我们在方法设计时只用到 T^e 与 T^r 中测试用例的输入信息，则整个测试用例选取过程在整个测试过程实际开始前就能够完成，而若方法用到测试用例 T^e 中的执行结果信息，那么测试用例选取过程则必须在测试过程中进行。

在本章中，我们基于自适应测试思想，对测试用例进行排序。具体来说，我们提出三种方法，分别是基于测试用例全集输入差异性的测试方法（Adaptive Fault Localization strategy based on All executed test cases，AFLA）、基于成功测试用例全集输入差异性的测试方法（Adaptive Fault Localization strategy based on executed test cases which Pass，AFLP）以及基于成功测试用例集输入差异性的随机测试方法（Adaptive Fault Localization strategy based on executed test cases which Pass，AFLRP）。其中，AFLA 仅仅用到测试用例的输入特征，是非实时的；而 AFLP 与

AFLRP 则需要在每一次执行完毕后判断被执行的测试用例是否失效，因此是实时的。另外，由于随机测试是基本且有效的策略（如第三章所述），我们同样在设计测试策略时引入随机因素。具体来说，当对接下来要执行的测试用例进行选择时，首先从 T^r 中随机选出候选集 T^c，并在此候选集中根据 AFLA、AFLP 与 AFLRP 各自基于的启发式信息对测试用例进行选择。

4.2.2.1　AFLA 方法

AFLA 方法基于以下启发式思想。

AFLA 启发式： 覆盖信息多样性程度高的测试用例更有利于缺陷定位。

在软件测试领域，测试用例的多样性是非常重要的性质。运行多样性程度高的测试用例集会观察到更多的软件行为，进而更容易触发软件的失效。而在缺陷定位中，测试用例的多样性也是非常重要的。如第三章所述，如果两个测试用例的覆盖信息是一样的，那么其中一个测试用例大概率是冗余的，甚至是起误导作用的。如果两个测试用例的覆盖信息与结果的正确性都相同，那么它们所能提供的测试信息是一样的，而其中一条测试用例就很可能是冗余的；而如果两个覆盖信息相同的测试用例的正确性不同，那么其中没有产生失效的测试用例一定是巧合一致性测试用例，其会对缺陷定位产生误导作用。执行这些测试用例是对时间及资源的一种浪费，其降低了整个缺陷定位过程的效率。因此，这些疑似冗余的测试用例最好放到后面执行，而那些能够带来更加丰富的覆盖信息的测试用例应该尽可能优先执行。

根据 **CPM 基本假设 1**与 **CPM 基本假设 2**，我们可以通过对输入空间的划分来度量测试用例的多样性。设两个测试用例 t_i 与 t_j，其输入空间的取值选择信息分别为 Ch^{t_i} 与 Ch^{t_j}，则 t_i 与 t_j 的差异性（即其距离）D^{t_i,t_j} 的度量方法如下：

$$D^{t_i,t_j} = |Ch - Ch^{t_i} \cap Ch^{t_j}| \tag{4.3}$$

进一步，给定一个测试用例 t 与测试用例集 T，则该测试用例与测试用例集之间的距离可由下式计算：

$$D^{t,T} = \min_{t' \in T} D^{t,t'} \tag{4.4}$$

现给定测试用例集 T^r，从其中随机选出候选测试用例集 $T^c \subset T^r$，AFLA 会从 T^c 中选出与已经执行过的测试用例集 T^e 最近的测试用例作为接下来要执行的测试用例

t^{next}，如式4.5所示：

$$t^{next} \leftarrow \arg\max_{t \in T^c} D^{t,T^e} \tag{4.5}$$

4.2.2.2 AFLP 方法

AFLP 基于以下启发式思想：

AFLP 启发式 1： 覆盖信息多样性程度越高的测试用例更有利于缺陷定位。

AFLP 启发式 2： 覆盖信息趋向于使目标程序失效的测试用例更有利于缺陷定位。

其中 **AFLP 启发式 1**和 **AFLA 启发式**是相同的，而 AFLP 还依据 **AFLP 启发式 2**，即失效测试用例在缺陷定位中起着更大的作用，这一点与第三章中 FM 度量的依据是一致的。一方面，缺陷语句一定包含在失效测试用例中，进一步在单缺陷假设下，缺陷语句一定会出现在所有失效测试用例所覆盖语句交集中。而成功测试用例则没有一个非常确定的有利于缺陷定位的性质。

AFLP 与 AFLA 的实际执行过程是相似的，唯一不同的是，AFLP 仅仅用到 T^e 中的成功测试用例作为衡量测试用例距离的标准。由于成功测试用例通常情况下会多于失效测试用例，其往往分布于整个测试空间中，因此仅仅以成功测试用例作为衡量测试用例距离的标准仍然有助于提升测试信息的多样性。此外，根据 **CPM 基本假设 3**，优先选择那些距离成功测试用例比较远的测试用例更有助于找到那些产生失效的测试用例。

设 $T^{e,p} \subset T^e$ 为已经执行的测试用例中成功测试用例的集合，则 AFLP 会从测试候选集 T^c 中选择那些距离 $T^{e,p}$ 最远的测试用例作为接下来优先执行的测试用例，如下式所示：

$$t^{next} \leftarrow \arg\max_{t \in T^c} D^{t,T^{e,p}} \tag{4.6}$$

4.2.2.3 AFLRP 方法

本小结给出本章中的最后一个基于输入空间划分的缺陷定位测试方法——基于成功测试用例集输入差异性的随机测试方法（AFLRP），其基于以下启发式信息。

AFLRP 启发式 1： 覆盖信息多样性程度高的测试用例更有利于缺陷定位；

AFLRP 启发式 2： 覆盖信息趋向于使得程序失效的测试用例更有利于缺陷定位；

AFLRP 启发式 3： 基于 T^e 中的部分测试用例计算测试用例间的距离可以提升

方法的稳定性。

在软件缺陷定位过程中，一些测试用例包含的信息反而会起到误导作用，例如巧合一致性（Coincidental Correctness）测试用例。Masri 等人[41] 通过实验发现巧合一致性现象普遍存在，而 CC 测试用例也会降低软件缺陷定位的准确度。若程序中没有巧合一致性现象，则有 $a_{ep}^{sf}=0$ 以及 $a_{ef}^{sf}=F$，那么缺陷语句都会被赋予最高的可疑度，而 SBFL 的准确度也会很高。相反，如果测试用例中存在巧合一致性现象，则 a_{ep}^{sf} 就不为 0，甚至是个比较大的值，这会导致缺陷语句的可疑度下降，进而降低缺陷定位的准确度。

而作为缺陷定位测试方法来说，4.2.2.1与 4.2.2.2小节提出的 AFLA 与 AFLP 的性能就很可能受到巧合一致性现象的影响。以 AFLP 为例，AFLP 以测试用例集 $T^{e,p}$ 为基准计算测试用例间的距离，并且以此衡量测试用例的质量；其旨在提升测试用例多样性的同时，保证程序失效空间的遍历效率。现假设测试用例 t 覆盖了缺陷语句 s^f 但其结果并没有发现失效，即 t 为巧合一致性测试用例，则根据 **CPM 基本假设 3**，其输入空间的分割信息 Ch^t 很可能与失效测试用例更为相似。尽管相对于测试用例全集来说，巧合成例不一定很多，但其对 AFLP 产生的影响却可能很大。假设有一个巧合成例 $t\in T^e$，其输入空间取值选择信息为 Ch^t，而 t' 是一个真正的未执行的失效测试用例，其取值选择信息为 $Ch^{t'}$，则很可能存在关系 $Ch^{t'}=Ch^t$。一旦如此，根据式 (4.4)，会有 $D^{t',T^e}=0$；则根据 AFLP 的选择规则（见式 (4.6)），该失效测试用例，以及与之相似的其它失效测试用例就很难在接下来的测试过程中被选中了。

而 AFLRP 就试图通过引入随机因素来减轻诸如 CC 测试用例那样在缺陷定位中起消极作用的测试用例带来的影响。AFLRP 与 AFLP 的区别体现在测试用例距离的计算上。与 AFLP 的距离计算公式 $D^{t,T^{e,p}}$ 不同，AFLRP 会实现从已执行成功测试用例集合 $T^{e,p}$ 中，随机抽取一定数量的测试用例集，组成规模较小的已执行测试用例子集作为比较基准，记为 T^{e,p,k^b}，其中 k^b 为该测试用例集的规模。其具体的测试用例选择过程如式 (4.6) 所示：

$$t^{next}\leftarrow \arg\max_{t\in T^c} D^{t,T^{e,p,k^b}} \tag{4.7}$$

与 AFLP 相比，AFLRP 采用双随机策略。一方面，如 AFLP 那样，AFLRP 从 T^r 中随机选出测试用例候选集 T^c，以保证 T^r 中的测试用例都有可能被选择到；另一方面，AFLRP 从已经执行的成例 $T^{e,p}$ 中同样随机选出作为距离比较基准的测试用例

集 T^{e,p,k^b}，以减轻一些带有误导性的测试用例对距离计算的影响。双随机策略在达到提升测试多样性目的的同时，保证了方法的稳定性。此外，由于随机生成的候选集 T^c 与作为距离计算基准的测试集 T^{e,p,k^b} 的规模都比较小，AFLRP 实际执行时所需要的计算量也会因此而减少。

4.3 实验与结果分析

本节通过实际的例子验证了本章提出的测试方法在解决缺陷定位测试问题时的有效性。

4.3.1 研究问题

本章实验部分所讨论的研究问题如下：

研究问题 1：基于输入空间划分的缺陷定位测试方法能否提升缺陷定位的效率？

研究问题 2：本章提出的三种方法各自具有什么特点，性能如何？

4.3.2 实验对象及其输入空间划分

本实验重点考察两个 *Siemens* 测试套件中的程序 [114]，实验对象及其附带的一系列标准缺陷版本如下：

schedule：拥有 9 个缺陷版本，其功能是安排不同进程之间的调度，使这些进程能够按照正确的顺序去执行。程序首先初始化三个不同优先级的进程队列，之后在运行过程中实时读取输入的控制字，对这三个队列中的进程按要求进行操控，或是对队列之间的优先级进行调整。

tcas：拥有 41 个缺陷版本，其功能是飞行器的防撞预警与控制。在运行时，其估计各机体之间发生碰撞的可能性，并在合理的时机向操作人员进行报警。

在对程序的输入空间进行划分时，*schedule* 的测试用例可以分为两个部分。首先，输入文件的开始会给出三个参数，分别代表初始化时不同优先级队列中的进程个数。其次，其输入还包含了在程序运行时对这三个队列或是这三个队列中的进程进行的一系列操作。具体来说，合法的操作有以下七种：*NewJob*、*UpgradePriority*、*Block*、*Unbolck*、*QuantumExpire*、*Finish* 以及 *Flush*。初始化的进程队列参数、标准操作以及程序在执行这些操作时的具体情况可以被看作是输入空间的范畴，由此得到的 *schedule* 程序具体范畴与取值选择划分如表4.2所示。

相较于 *schedule*程序，*tcas* 程序的输入空间则相对简单。一条正常的 *tcas* 测试用

表 4.2　*schedule* 范畴与取值选择划分

♯	范畴	取值选择
1	无效输入（包括进程数 < 0，优先级提升率 *UpgradePriority*< 0 等)	是
		否
2	进程总数	$= 0$
		$\neq 0$
3	优先级为 1 的进程数	$= 0$
		$\neq 0$
4	优先级为 2 的进程数	$= 0$
		$\neq 0$
5	优先级为 3 的进程数	$= 0$
		$\neq 0$
6	包含操作 *NewJob*	是
		否
7	包含操作序列：*NewJob*$\rightarrow \ldots \rightarrow$ *Flush*	是
		否
8	包含操作 *UpgradePrority*	是
		否
9	包含操作序列：*Start* $\rightarrow$ *UpgradePrority*$\rightarrow \ldots \rightarrow$*Flush* 或 *Flush*$\rightarrow \ldots$ *NewJob*$\rightarrow \ldots$*UpgradePrority*$\rightarrow \ldots \rightarrow$*Flush*	是
		否
10	*UpgradePrority Ratio*$= 0$	是
		否
11	*UpgradePrority Ratio*$= 1$	是
		否
12	包含操作 *Block*	是
		否
13	包含操作序列：*Start* $\rightarrow \ldots \rightarrow$ *Block*$\rightarrow \ldots \rightarrow$*Flush* 或 *Flush*$\rightarrow \ldots \rightarrow$*NewJob*$\rightarrow \ldots \rightarrow$*Block* $\rightarrow \ldots \rightarrow$*Flush*	是
		否
14	包含操作 *Unblock*	是
		否
15	包含操作序列：*Start*$\rightarrow \ldots \rightarrow$ *Block*$\rightarrow \ldots \rightarrow$ *Unblock*$\rightarrow \ldots \rightarrow$*Flush* 或 *Flush*$\rightarrow \ldots \rightarrow$*NewJob*$\rightarrow \ldots \rightarrow$*Block*$\rightarrow \cdots \rightarrow$ *Unblock*$\rightarrow \ldots \rightarrow$*Flush*	是
		否
16	*Unblock Ratio* $= 0$	是
		否
17	*Unblock Ratio* $= 1$	是
		否
18	包含操作 *QuantumExpire*	是
		否
19	包含操作序列：*Start*$\rightarrow \ldots \rightarrow$*uantumExpire*$\rightarrow \ldots \rightarrow$*Flush* 或 *Flush*$\rightarrow \ldots \rightarrow$*NewJob*$\rightarrow \ldots \rightarrow$*QuantumExpire*$\rightarrow \ldots \rightarrow$*Flush*	是
		否
20	包含操作 *Finish*	是
		否
21	包含操作 *Flush*	是
		否
22	*Flush*$\rightarrow \ldots \rightarrow$(及不存在 *NewJob*)$\rightarrow$...$\rightarrow$*Flush*, 及连续的 flush 操作	是
		否

表 4.3 *tcas* 范畴与取值选择划分

♯	范畴	取值选择
1	无效输入（包括输入参数个数小于 < 13，间距 *Up_Separation* < 0 等)	是
		否
2	TCAS 是否有效	是
		否
3	*Up_Separation* 与 *Down_Separation*的关系	$>$
		$<$
		$=$
4	*Up_Separation*+ *NOZCROSS (100)* 与 *Down_Separation*的关系	$>$
		$<$
		$=$
5	*Up_Separation* 与 *ALIM*（在 *Alt_Layer_Value*$= 0, 1, 2, 3$ 时分别取 $400, 500, 640, 740$）的关系	$>$
		$<$
		$=$
6	*Down_Separation* 与 *ALIM*的关系	$>$
		$<$
		$=$
7	*Up_Separation*+ *NOZCROSS* 与 *ALIM*的关系	$>$
		$<$
		$=$
8	*Own_Tracked_Alt* 与 *Other_Tracked_Alt* 的关系	$>$
		$<$
		$=$
9	*Climb_Inhibit*	$= 0$
		$= 1$
10	*Alt_Layer_Value*	$= 0$
		$= 1$
		$= 2$
		$= 3$

例包含 13 个输入参数（其中并没有操作指令信息），它们用来描绘主机体当前的状态以及其与其它机体的位置关系。因此在划分输入空间范畴及确定各范畴的取值选择时，每个输入参数本身或者是某些特定的输入参数组合被看作范畴，其具体范畴及选择的划分如表4.3 所示。

4.3.3 实验设置

如上所述，*schedule* 程序附带 9 个标准缺陷而 *tcas* 程序附带 41 个标准缺陷。在进行实验时，我们仔细检查了每一个缺陷版本，将一些宏定义缺陷以及缺陷语句分布在多个程序分支上的缺陷版本去除掉，因为这些缺陷版本实质上是不满足单缺陷假设。最终保留并在实验中进行讨论的缺陷如表4.4 所示。

三种基于程序输入空间的缺陷定位测试方法——AFLA、AFLP 与 AFLRP——将分别被执行并评估。此外，纯随机测试也会作为比较对象而被采用。在检验一种测试方法时，首先对程序进行纯随机测试，当发现程序失效时，缺陷定位测试开始，而待

表 4.4　实验考查的缺陷版本

schedule			
FAULT_V2	FAULT_V3	FAULT_V4	FAULT_V8
tcas			
FAULT_V1	FAULT_V2	FAULT_V4	FAULT_V6
FAULT_V10	FAULT_V11	FAULT_V25	FAULT_V28
FAULT_V29	FAULT_V30	FAULT_V31	FAULT_V32
FAULT_V35	FAULT_V36	FAULT_V37	FAULT_V37
FAULT_V39	FAULT_V41		

表 4.5　各策略间的比较结果

比较的测试策略（机制）对	缺陷版本数	
	优于	劣于）
AFLA→ 纯随机测试	8	4
AFLP→ 纯随机测试	10	2
AFLRP→ 纯随机测试	10	1
AFLP→AFLA	1	0
AFLRP→AFLA	4	7
AFLRP→AFLP	4	6

考察的测试方法将被运行。每一个新的测试用例执行完毕后，SBFL 将被执行并利用当前已经执行的测试用例（包括刚刚被执行的测试用例）进行缺陷定位，并给出作为缺陷位置预测结果的可疑度列表。如第三章所示，整个缺陷定位过程中所有缺陷位置预测结果的 Expense 平均值（即 AE 值）将被作为评价此次缺陷定位测试过程效率的标准。此外，我们还记录每个测试方法的执行时间，以观测各个方法的执行代价。由于随机因素的存在，对于每个程序、缺陷版本以及测试方法所组成的实验实例，我们都重复运行 100 次，通过假设检验来评估该实例的缺陷定位效率。

4.3.4　实验结果

图4.1展示了在 *shedule* 与 *tcas* 程序的所有缺陷版本上各测试方法的 AE 值。为了方便，我们将实验结果分为六个子图进行展示，将具有相似 AE 值的缺陷版本放在同一个坐标系下。图中，x 轴列出被展示的缺陷版本的名称，而各个测试方法所得到的 AE 值在 y 轴上标记。在每个子盒须图中，盒中的平行线分别标记了对应缺陷版本上进行的 100 次实验中低第四分位、平均以及高四分位次数所对应的 AE 值。例如低四分位线表示 1/4 的运行次数中 AE 值小于该水平线所对应的 y 轴刻度。而红色、蓝色、青色与绿色所对应的结果分别代表了纯随机测试、AFLA、AFLP 与 AFLRP 的性能。需指出，具备越低的 AE 值的实例缺陷定位效率越高。

表4.5显示了各个策略之间的对比结果，表中的数字代表了一个测试方法好（差）

表 4.6 各策略的平均计算时间

策略名称	平均执行时间（秒）	
	schedule	*tcas*
纯随机测试	0.22	0.06
AFLA	8.30	2.40
AFLP	6.50	2.30
AFLRP	0.41	0.13

于另一个测试方法的缺陷版本数。这里，我们根据 p 值为 0.05 的 t-假设检验来决定一个测试方法是否在某一缺陷版本中优于另一个测试方法。

最后，为了观察每个自适应机制的执行效率，我们将这些方法在所有缺陷版本下的平均执行时间记录在表4.6中。

4.3.5 实验结果分析

本小节将对得到的实验结果进行分析，并在此基础上回答第4.3.1节中列出的研究问题。

对于**研究问题 1**（基于输入空间划分的缺陷定位测试方法能否提升缺陷定位的效率？），我们对图4.1 给出的各测试策略（包括自适应测试方法及纯随机测试）在解决缺陷定位测试用例优化问题中的性能进行了比较与分析。结果表明，总体上，随机测试的 AE 值比采用 AFLA、AFLP 以及 AFLRP 所的到的 AE 值明显要高。此外，从表4.5 中可以进一步了解到，采用自适应测试方法后，缺陷定位效率提升的缺陷版本数比下降版本数至少多一倍，甚至多达 10 倍（采用 AFLRP 后的结果）。这说明本章提出的测试方法能够切实地在缺陷定位测试中提升缺陷定位效率。

此外，从图4.1 中还能看出，采用自适应机制后的盒须图上下两个四分位线的距离比纯随机测试短很多，这说明得到的 AE 值与纯随机测试相比具有更小的方差。例如在 *tcas* 程序的 FAULT_V38 缺陷上，随机测试得到的 AE 值会在 0.2% 与 2.8% 之间变化，两个四分位线会在 0.9% 与 1.4% 之间；而采用自适应测试方法后 AE 值的浮动会缩小到 0.2% 与 1.4% 之间，而上下四分位线分别为 0.9% 与 1.1%。类似的现象在过半的缺陷版本中都被观测到，而相反的现象并没有出现。如果 AE 值的方差很大，则意味着缺陷定位效率具有非常大的不确定性，也意味着在采用自动化缺陷定位方法时会存在较大的风险；而较低的 AE 值方差说明了方法具有较好的稳定性，这使得调试人员在实际开发环境中能够有更充足的信心去使用这些方法。

总结来说，所有自适应测试方法性能都好于纯随机测试，它们能够得到更低的 AE

值与方差。也就是说，采用这些方法，能够提升缺陷定位的效率并降低自动化缺陷定位带来的风险。实验数据进一步说明了缺陷定位测试问题的意义，以及采用输入空间划分方法解决该问题的合理性。

对于**研究问题 2**（本章提出的三种方法各自具有什么特点，性能如何？），我们从三个方面对本章自适应测试方法的性能进行了比较分析：1）相对于纯随机测试的提升程度；2）三种自适应策略之间的比较；3）所需的计算时间。

由表4.5 可以看出，AFLA 与纯随机测试相比在 8 个缺陷版本上定位效率提升，在 4 个缺陷版本上下降；AFLP 在 10 个版本上提升，在两个缺陷版本上下降；而 AFLRP 同样在 10 个缺陷版本上提升，仅在 1 个缺陷版本上下降。这些结果显示，与仅仅考虑了覆盖信息的多样性的 AFLA 相比，AFLP 进一步考虑了遍历失效空间的重要性，进而取得了更好的缺陷定位效果。而与 AFLP 相比，AFLRP 采用双随机机制，在提升缺陷定位效率的基础上，保证了方法的稳定性。我们发现，当 AFLA 提升了缺陷定位效率时，AFLP 与 AFLRP 也都得到了很好的结果。而类似地，在 AFLP 起积极作用的缺陷版本上，AFLRP 也同样起到了积极作用。

此外，这三种方法之间的比较也展示在表4.5 中。可以看到，AFLP 在三者比较中取得了最好的效果，其在一个缺陷版本上比 AFLA 好，且并不存在 AFLP 比 AFLA 差的缺陷版本。此外 AFLP 在 6 个缺陷版本上比 AFLRP 好，而在 4 个缺陷版本上比 AFLRP 差。这说明 AFLP 的启发式信息（即快速遍历失效空间）是有意义的。在另一方面，AFLRP 在三者比较中是最差的（其比 AFLA 与 AFLP 差的缺陷版本数分别为 4 个和 6 个，而好的版本数均为 4 个）。可见 AFLRP 的启发式信息（即双随机策略）提升了方法的稳定性，有效的避免了方法带来的消极因素的同时，也在一定程度上减小了对缺陷定位效率的提升幅度。也就是说在 AFLA 与 AFLP 起积极作用的缺陷版本上，AFLRP 也能够起积极作用，然而不如 AFLA 与 AFLP 明显。但需指出，稳定性是一个自适应方法具备应用价值的前提。

表4.6 列出了各个方法在执行时的计算代价。首先，随机测试在一个 *shedule* 与一个 *tcas* 实例的缺陷定位测试上花费的平均时间分别为 0.22 到 0.06 秒，其执行代价是最小的，这主要因为其不需要任何的加乘计算。仅随其后的是 AFLRP 方法，其在 *schedule* 与 *tcas* 的缺陷版本上平均分别花费了 0.41 秒与 0.13 秒的计算时间，这与随机测试的差距并不是很大。由于采用了双随机策略，其选择一个测试用例所花费的时间复杂度是常量级的（与随机测试相同），不会随着测试过程以及测试用例集规模的增大

而增大。可见双随机策略在保证效率的基础上很大程度上节省了测试方法的计算代价。而 AFLA 与 AFLP 考虑了全部的已执行测试用例（即 $t \in T^e$），因此其执行代价会随着缺陷定位测试过程呈线性增长趋势。它们在 *schedule* 上的平均执行时间分别为 8.3 秒与 6.5 秒，而在 *tcas* 上的平均执行时间分别为 2.4 秒与 2.3 秒。

总结来说，AFLRP 在缺陷定位测试中能够最稳定地起着积极作用，尽管其对于缺陷定位测试效率的提升幅度有时不如 AFLP 或者 AFLA 明显。此外，双随机策略降低了 AFLRP 的计算复杂度，使其执行效率接近于纯随机策略。尽管，AFLP 在很多缺陷版本上的表现比 AFLRP 要好，在实际开发环境中 AFLRP 的稳定性与轻量级的执行代价是更重要的。因此，在输入空间比较容易划分的程序中，AFLRP 方法是我们所推荐的，AFLP 也可以被考虑。另外，尽管 AFLA 方法在稳定性与执行效率上不如 AFLP，但其可以在缺陷定位测试实际开始之前就进行完毕，因此在实际环境中可以作为预处理策略而被采用。

4.4 潜在的风险

本章所提出方法的主要潜在风险在于程序的输入空间划分方法。对于文档说明不够详细的程序，从不同的角度去解读输入空间可能会得到不同的划分方案。然而，我们的方法仅仅利用不同输入范畴与取值选择的方法来描述测试用例间的差异，其只利用了输入空间划分的最基本性质。因此无论从哪个角度划分输入空间，只要划分方法合理，不同测试用例间的差异性，以及成功与失效测试用例间的差异性都能被反应出来。当然，面向 FLP 问题的输入空间划分方法也是值得被进一步研究的。

另一个潜在风险是本章的实验只基于 Op 公式，然而无论是 AFLA、AFLP 还是 AFLRP，其执行过程均与可疑度计算公式无关，因此我们相信该方法在多种 SBFL 公式上都是适用的。此外，Op 公式也是单缺陷下的 Maximal 公式，因此在此公式上面取得好的效果本身就是重要的。

最后，还有一个问题就是我们只用了 *Siemens* 程序集中的 *schedule* 与 *tcas* 两个实验程序对本章的方法进行了验证。在后续工作中，我们希望能够将本章提出的测试方法在更多标准程序上做进一步检验。

4.5 本章小结

本章提出了三种基于输入空间划分的缺陷定位测试用例优化方法，来提升面向陷定位的测试效率。在这三种方法（即 AFLA、AFLP 与 AFLRP）都通过测试用例在输

入空间划分取值上的差异性，来量化测试用例之间的距离，并以此为依据对测试用例进行选择，试图提升测试信息的多样性程度。此外，AFLP 与 AFLRP 通过强调对输入空间中程序失效区域的遍历，在缺陷定位测试中取得了更好的效果。

这三种方法仅仅用到了未执行测试用例的输入空间信息，因此其在实际软件工程中的获取难度要低于第三章中基于程序覆盖信息的方法。实验证明，与纯随机策略相比，本章提出的测试方法在缺陷定位测试中确实起到了积极作用。总结来说，本章的贡献主要可以归纳为以下三点：

（1）提出了基于输入空间划分的黑盒缺陷定位测试优化方法；

（2）将范畴与取值选择的划分方法用在软件缺陷定位上，取得了很好的效果；

（3）提出了缺陷定位在输入空间划分上的启发式思想，分析了其与缺陷定位的关系，并将其用在面向缺陷定位测试方法的设计上。

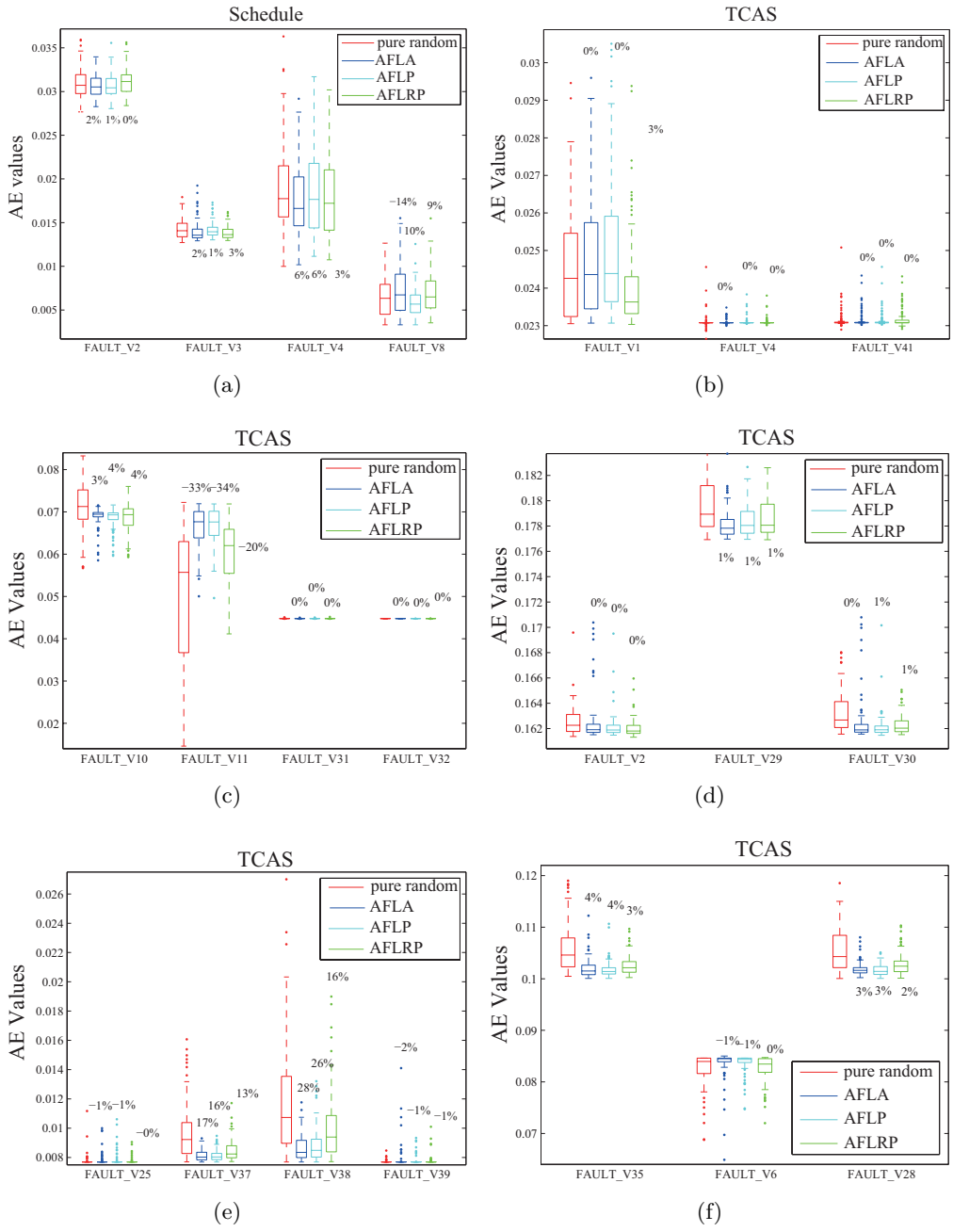


图 4.1　各测试策略在不同缺陷版本下的盒须图

第五章　基于测试分类的自适应缺陷定位测试信息处理方法

5.1　引言

在软件工程中，测试人员通过软件测试来检查被测软件是否能够正确运行。如果发现失效，则转到缺陷定位环节，以找到引发软件失效的缺陷位置，并通过反复的调试过程修复该缺陷。而软件缺陷定位所利用的信息，正是测试过程中获取到的测试信息。因此，提高测试信息的质量，是提升软件缺陷定位方法性能的关键。本文第三章与第四章讨论的就是通过合理地安排测试过程，加速测试用例集质量的提升速率，而本章则主要讨论对于已经获取的测试信息的利用问题。

由于自动化软件缺陷定位方法在推理缺陷代码位置时，需要用到测试信息的统计数据，因此其对测试信息的完整度会有较强的依赖性。而如何对测试过程中产生的测试信息进行尽可能合理地利用，是提升测试信息质量，并最终提升软件缺陷定位效率的关键。更进一步，充分地利用测试信息不仅是缺陷定位的基本需求，同样是较为迫切的需求。在软件测试过程中，测试信息的完整性与准确性问题（例如测试用例覆盖信息的不完整性[47]、测试 Oracle 的缺失[119] 等）一直是研究热点[121] 。而这些测试中存在的问题都会对软件缺陷定位产生影响。本章主要针对 Oracle 缺失或不完整的条件下，缺陷定位测试信息的利用问题进行讨论。

在软件测试领域，Oracle 问题一直是一个热点研究问题[122]，其主要讨论判断被测软件输出正确性机制（即测试 Oracle）的各种问题（如缺失、不完整、不正确等）所造成的影响及其相应的解决方案。目前，各种基于软件规范、异常行为检测、逻辑推理等用来解决 Oracle 问题的方法被提出[123, 124]。这些方法使得测试人员能够在一定程度上解决 Oracle 问题，即在测试 Oracle 存在问题的情况下进行测试，并完成既定的测试目的。然而它们并不能取代 Oracle 本身来判断整个测试用例集的正确性。因此，实际测试环境中 Oracle 缺失的情况依然是普遍存在的[125]。

当 Oracle 缺失时，测试人员会考虑采用现有的测试方法来保证测试过程能够有效地进行下去。一旦软件在测试过程中发生失效，则需转入缺陷定位环节来对导致失效的软件缺陷进行排查。此时，调试人员会尝试使用自动化缺陷定位方法或工具以辅助其对缺陷位置进行推断。缺陷定位方法需要高质量的测试信息作为输入，然而测试时所采用的测试方法虽然能够在一定程度上保证软件失效的鉴别与分析，其并不能改变 Oracle 缺失的实质。因此，尽管测试人员能够利用测试过程中的相关数据来发捕获程

序运行过程中产生的失效[126, 127, 128]，其仍没有办法对所有测试用例的正确性进行标定。例如，当测试数值计算程序时，我们可以通过将一些特殊值（如 sin 函数中 30°、45°、60°、90°、22.5° 等）作为输入来检验输出的正确性[129]。又如，对于一些求解优化模型的程序来说，我们可以通过其他文献中的历史数据以及经典数据集上的测试结果对其正确性进行验证[130, 112]。然而由于这些程序的输入空间是连续的，而连续空间中大部分输入值的正确性都是无法判定的。因此，从缺陷定位角度，针对测试 Oracle 问题的解决方案仍是需要进一步研究的。

在软件测试过程中，测试人员会记录各种测试信息，而缺陷定位所需要的信息主要是程序执行时的语句覆盖信息以及输出结果正确性信息。本章用测试用例的“标签”来描述一个测试用例输出的正确性。对于一个测试用例，其标签为“成功”或是“失效”分别指该测试用例的输出是正确的或是产生了失效。进一步，称一个测试用例是“带标签的”当且仅当其被赋予了一个“成功”或是”失效的标签；否则我们称其为“无标签的”。我们知道，Oracle 的缺失并不会阻碍被测软件的运行，因此测试人员仍然可以用任何测试用例作为输入来运行待测软件，并且得到这些测试用例的执行信息。由于这些执行信息记录了程序在不同输入下的行为，因此其能够在一定程度上反映软件结构与数据关系，进而为程序缺陷位置的推断提供支持。然而，由于 Oracle 的缺失，尽管一些测试用例能够被运行且执行信息能够被记录，测试人员却无法为其附上标签，即无法识别其输出的正确性。这导致在缺陷定位开始时，只有少部分测试用例可以被贴上标签（即通过特殊方法对一些输入值的正确性进行判断），大多数测试用例都处于无标签状态。然而大多数自动化缺陷定位方法都将测试用例的执行信息与其正确性之间的关系作为推断缺陷位置的重要线索，因此这些无标签的测试用例就会被传统的缺陷定位方法视为“没有利用价值”并被舍弃。根据前文所述，缺陷定位对于测试信息质量的依赖性非常高，而能够被利用的少量带标签测试用例所包含的有用信息却非常有限。结果，由于测试信息的不充分，所采用的缺陷定位方法就很可能无法给出准确的缺陷位置预测结果。

由于 Oracle 的缺失导致的对无标签测试信息的浪费是产生上述问题的根源。尽管这些无标签测试用例的正确性无法被直接验证，它们所表现出的行为模式都可以在一定程度上暴露程序的内部结构以及逻辑关系。现有研究发现，行为相似的测试信息在软件失效上的表现也很可能相似[131, 132]，这说明无标签测试用例所包含的行为信息同样能够反映出软件的失效机理，进而对软件缺陷的理解与分析起到促进作用。因此，

在不引入任何额外领域知识的前提下，尽可能充分地利用这些无标签测试用例，是在Oracle 缺失条件下提升缺陷定位方法性能的可选方案。

本章对软件缺陷定位过程中测试 Oracle 缺失问题的解决方案进行探索，提出合理的测试信息挖掘方法以及以软件缺陷定位为目的的测试信息利用方法。本章具体针对以下两个问题展开研究：

（1）无标签测试用例究竟对软件缺陷定位有没有帮助?

（2）如何在缺陷定位过程中合理地利用这些无标签测试用例？

本章的研究内容依然基于程序谱缺陷定位方法（SBFL）。由前面介绍可知，SBFL 将程序的执行轨迹与程序的失效相关联，并根据各个语句在失效与成功测试用例中的执行频率来推断其包含缺陷的可能性。因此，一条测试用例的成功与失效（即其标签）对于 SBFL 的执行来说是非常重要的。尽管目前很多面向带标签测试信息的信息利用方法已经被提出[47, 41]，然而仅仅利用带标签的测试用例对于准确定位缺陷语句来说很有可能是不够的。根据 Jiang 等人[42] 通过大量的实验得到的观测结论，当只有少部分带标签测试用例可用时，SBFL 的性能会有明显下降。因此，在 Oracle 缺失的情况下，通过对大量无标签测试用例的合理利用来丰富现有的测试信息库，进而提升 SBFL 的性能是非常值得研究的。

为此，本章提出一种基于测试分类的缺陷定位无标签测试用例利用方案。首先我们提出了可疑度概率（Suspiciousness-Probability-based，SP）分类器。SP 分类器基于 SBFL 本身的领域知识，根据当前已标记的测试用例来预测未标记测试用例的正确性，并为其赋予新的预测标签。分类过后，新标记的测试用例就能够作为输入参与到语句可疑度的计算中了。通过对大量原本无标签测试用例的预测与标定，整个可利用测试信息库的信息多样性程度被极大地提升了。这使得更多的语句能够通过当前的程序谱信息被区分开来。实验结果表明，我们的方法能够切实提升软件缺陷定位效率。

5.2 研究动机

本节通过对 **无人机航迹规划程序及其缺陷版本** 的示意代码进行分析来引出本章的研究动机。

无人机航迹规划程序及其缺陷版本： 考虑无人机航迹规划的控制程序（function for Unmanned Aerial Vehicle Path Planning problem, **UAVPP**[133]），其简化的示意代码及缺陷版本如图5.1 所示。**UAVPP**函数接收无人机当前的状态信息，将

语句	原程序	缺陷版本
	Input (*wallDis*, *objDis*){	Input (*wallDis*, *objDis*){
	double *step, ranDegree*;	double *step, ranDegree*;
	void (**method*)(double, double);	void (**method*)(double, double);
s_1	*method* = &*MRRT*;	*method* = &*MRRT*;
s_2	*ranDegree* =1;	*ranDegree* =1;
s_3	*step* = 2;	*step* = 2;
s_4	if(*wallDis* < 8){	if(*wallDis* < 8){
s_5=s^f	*step* = 1;	***step* = 2;**
s_6	if(*wallDis* < 5){	if(*wallDis* < 5){
s_7	*ranDegree* = 2;	*ranDegree* = 2;
s_8	if(*wallDis* < 1.5)	if(*wallDis* < 1.5)
s_9	*method* = &*RRT*;	*method* = &*RRT*;
	}	}
	}	}
	else{	else{
s_{10}	*method* = &*NN*;}	*method* = &*NN*;}
s_{11}	if(*objDis*<10){	if(*objDis*<10){
s_{12}	*step* = 1;	*step* = 1;
s_{13}	*method* = &*MRRT*;	*method* = &*MRRT*;
s_{14}	*ranDegree* =0.5;	*ranDegree* =0.5;
	}	}
s_{15}	(**method*)(*step*, *ranDegree*);	(**method*)(*step*, *ranDegree*);
s_{16}	return;	return;

图 5.1　求解航路规划问题的示例程序

其表示为两个变量 $objDis$ 和 $wallDis$。其中，$objDis$表示当前无人机到目标点之间的距离，而 $wallDis$表示当前无人机到距其最近的障碍物之间的距离；这里假设无人机的状态能够通过传感器被检测到并实时传输给 **UAVPP**。在实际航迹规划中，无人机会根据其当前所处的状态选择适当的航迹规划方法，以保证其安全性、实时性、航程与航迹的平滑度等。而 **UAVPP**的主要作用就是根据无人机当前所处的状态对航迹规划算法进行选择与实时切换。具体来说，当无人机远离障碍物时，会借助一张经过训练的神经网络，生成相对平滑的路径，并且执行程序语句 s_10 中的函数 NN，以进一步训练这张神经网络。而当 UAV 接近一个或者多个障碍物（即 $WallDis$ 减小到某个数值）时，程序会将规划算法切换为带有随机性的方法（如（$QS\text{-}RRT$ 与 RRT [134, 111]），以提升路径搜索的广度，保证无人机能够及时摆脱障碍物的包围，到达一个安全地带。在程序的缺陷版本中，语句 s_5 是缺陷语句，其在 $wallDis < 8$ 的条件下错误地设置了无人机的步长变量 $step$。

现假设有 5 个测试用例，它们的执行信息以及结果标签如图5.2所示。其中 t_1、t_2 与 t_3 为已标记测试用例。这其中 t_2 与 t_3 只经过了 $wallDis > 8$ 条件下的代码区域。而 t_1 是一个失效测试用例，但其覆盖了很多条语句（16 条语句中的 12 条）。因此该测试用例对缺陷语句位置的暴露能力不是很强。现假设我们仅仅用这三个带标签测试用

语句编号	t_1	t_2	t_3	t_4	t_5
	Input: 1.5, 50	*Input*: 10, 5	*Input*: 20, 45	*Input*: 4, 45	*Input*: 7, 60
s_1	•	•	•	•	•
s_2	•	•	•	•	•
s_3	•	•	•	•	•
s_4	•	•	•	•	•
s*5=*f	•			•	•
s_6	•			•	•
s_7	•			•	
s_8	•			•	
s_9	•				
s10		•	•		
s_{11}	•	•	•	•	•
s_{12}		•			
s_{13}		•			
s_{14}		•			
s_{15}	•	•	•	•	•
s_{16}	•	•	•	•	•
output	*failing*	*passing*	*passing*	?	?

图 5.2 **UAVPP**的测试用例库示例

例来进行软件缺陷定位，那么 SBFL 将无法区分 $wallDis > 8$ 条件下的所有语句。也就是说，语句 s_5 到 s_9 都会被赋予相同的可疑度值。然而，我们还有另外两个执行信息不相同的无标签测试用例 t_4 与 t_5，如果这些测试用例能够被正确的应用到缺陷定位中，测试信息库中的信息多样性程度将会随之提升，而 SBFL 所给出的缺陷位置预测结果也会更加准确。

从测试用例 t_1 至 t_3 可以看出，成功与失效测试用例的行为具有很大的差异。具体来说，失效测试用例 t_1 覆盖了 $wallDis < 8$ 条件下的语句；而成功测试用例 t_2 与 t_3 所覆盖的语句在条件 $wallDis > 8$ 与 $objDis < 5$ 之间。进一步考察测试用例 t_4 与 t_5，其与 t_2 和 t_3 的覆盖信息差异很大。此外，t_4 与 t_1 覆盖的语句相同，而 t_5 与 t_1 的覆盖语句中只有 3 条不同。因此，这两个测试用例有更大的概率会产生失效（即与失效测试用例 t_1 的行为相似），故我们可以赋给它们一个“失效”的预测标签。当这两个带有预测标签的测试用例添加到 SBFL 输入中时，缺陷语句 s_5 的可疑度会增大，其在缺陷可疑度排位表中的排位也会提升。由上所述，我们通过对无标签测试用例 t_4 与 t_5

进行合理利用，有效地提升了软件缺陷定位的准确度。

5.3　面向程序谱缺陷定位的无标签测试用例利用方法

5.3.1　问题描述

记 PG 为目标程序，其包含的语句数量为 n。记所有语句的集合为 $S = \{s_1, s_2, \ldots, s_n\}$。假设给定测试用例集 T，则关于 T 的程序谱信息可以写成在任意语句 s 上的四元组 $\langle a_{ef}^{s,T}, a_{ep}^{s,T}, a_{nf}^{s,T}, a_{np}^{s,T}\rangle$。其中，$a_{ef}^{s,T}$ 与 $a_{ep}^{s,T}$ 分别表示 T 中经过语句 s 的测试用例中失效与成功的测试用例个数；而 $a_{nf}^{s,T}$ 与 $a_{np}^{s,T}$ 则分别表示 T 中没有经过语句 s 的测试用例中失效与成功的测试用例个数。另外我们记 T 中失效测试用例的总数为 F^T，而成功测试用例的总数为 P^T。

若调试人员使用 SBFL 进行缺陷定位，则在 PG 给定的情况下，传统 SBFL 的执行过程可表示为关于 T 的函数：

$$L = SBFL(T) \tag{5.1}$$

其中测试用例集 T 作为输入变量。而 SBFL 输出的缺陷位置预测结果，即可疑度列表 L，在本章的定义如下：

$$L = [\lambda_1^L \lambda_2^L \cdots \lambda_n^L] \tag{5.2}$$

其中

$$\lambda_k^L = \langle s_k^L, r_k^L \rangle,$$

s_k^L：在 L 中排名为 k 的语句

$r_k^L = r^{s_k^L}$：语句 s_k^L 的可疑度值

在公式 (5.2) 中，λ_k^L 表示里列表 L 中的第 k 个元素，其中 s_k^L 与 r_k^L 分别表示该元素对应的语句以及其可疑度值。度量 L 性能的 Expense 值记为 E^L，其计算式入下：

$$E^L = (i/|L|) \times 100\%, \quad s_i^L = s^f \tag{5.3}$$

设 T^a 为已经执行的测试用例全集。对任意测试用例 $t \in T^a$，记 C^t 为其覆盖信息，其被定义为测试用例 t 覆盖的语句集合，即 $C^t = \{s|$语句 s 被测试用例 t 覆盖$\}$。此外记 o^t 为测试用例 t 的正确性标签，其中 $o^t = passing$ 表示测试用例 t 是成功测试用例，

而 $o^t = failing$ 表示 t 是失效测试用例。本章研究中，若 t 是无标签的，我们将其标签信息记为 $o^t = null$。

在测试 Oracle 缺失的情况下，有大量的测试用是无标签的。因此，测试用例全集 T^a 就可以分成 T^l 与 T^u 两个子集，其中 T^l 表示已经被标记的测试用例集，即 $T^l = \{t \in T^a | o^t = passing || o^t = failing\}$；而 T^u 表示未被附上标签的测试用例集，即 $T^u = \{t \in T^a | o^t = null\}$。本章假设无标签测试用例的数量要远远多于带标签测试用例的数量，因此我们有 $|T^u| \gg |T^l|$。

由于 SBFL 在可疑度计算中需要用到测试用例正确性信息，因此只有 T^l 中的测试用例能够被 SBFL 利用。故实际 SBFL 过程的表示函数为

$$L^l = SBFL(T^l) \tag{5.4}$$

而由于包含在 T^l 中的测试信息非常有限，传统 SBFL 得到的结果 L^l 很可能不准确，即 E^{L^l} 的值很大。

本章旨在对 T^u 中的测试信息进行合理利用，以提升 SBFL 的性能。具体来说，我们希望改善传统的 SBFL 过程函数 $L = SBFL(T)$ 为

$$L = SBFL(T_1, T_2) \tag{5.5}$$

与传统的 SBFL 过程函数不同，式 (5.5) 中的过程函数接收两个测试用例集 T_1 和 T_2 作为输入。该函数的输出同样是一个可疑度列表 L，该列表的构成与定义式 (5.2) 相同；所不同的是，各语句可疑度值 r^s 的计算将同时基于 T_1 与 T_2 这两个测试用例集中的信息。进一步，对于 T_1 中的测试用例，我们既可以利用其语句覆盖信息，也可以利用其测试结果的正确性信息（即标签信息）。然而，对于 T_2 中的测试用例，我们仅仅能利用其语句覆盖信息。现令 $T_1 = T^l$ 且 $T_2 = T^u$。这样，带标签测试用例集 T^l 与无标签测试用例集 T^u 就都可以在式 (5.5) 所描述的过程中被利用了。则当采用改进的 SBFL 方法进行缺陷定位时，执行过程可用下面函数表示：

$$L^{l,u} = SBFL(T^l, T^u) \tag{5.6}$$

其中 $L^{l,u}$ 表示基于 T^l 与 T^u 得到的可疑度列表。总的来说，解决面向 SBFL 的无标签测试用例利用问题就是要设计一个 SBFL 的改善流程 $L^{l,u} = SBFL(T^l, T^u)$ ，使其性能优于传统的 SBFL 过程 $L^l = SBFL(T^l)$。具体问题描述如下：

- 条件假设： 整个测试用例全集 T^a 分为带标签测试用例集 T^l 以及无标签测试用

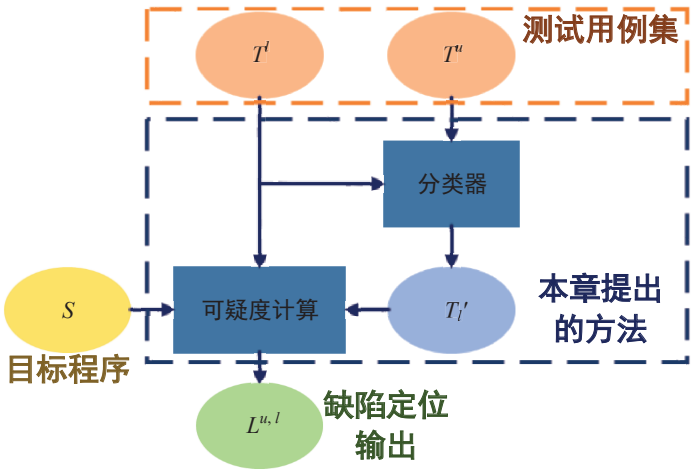


图 5.3　方法的整体框架

例集 T^u，满足 $|T^u| \gg |T^l|$。调试人员采用 SBFL 进行软件缺陷定位。

- 问题：将 SBFL 过程由 $L^l = SBFL(T^l)$ 改为 $L^{l,u} = SBFL(T^l, T^u)$ 使得 $E^{L^{l,u}} < E^{L^l}$。

5.3.2　求解框架

图5.3 描述的框架中，蓝色虚线框中的内容为本章提出的方法。该方法接受目标程序 PG 以及测试用例全集 T^a，包括带标签测试用例 T^l 以及无标签测试用例 T^u。与传统 SBFL 方法类似，其输出为可疑度列表 $L^{l,u}$。具体来说，方法的执行分为两个步骤——测试用例分类以及可疑度计算。

测试用例分类是该方法的核心。分类问题是模式识别与机器学习领域中的传统问题。假设论域中包含若干类别，每个类别都有一个属于自己的标签，记候选标签集合为 Y。此外，现有若干无标签样本，其集合记为 X。分类问题旨在鉴别各个无标签样本 $x \in X$ 的类别。也就是说，我们期望寻找一个决策函数 $h : X \to Y$，将无标签样本集合 X 映射到标签集合 Y 上。这里，决策函数 h 需要根据现有知识来构造。构造过程是一个基于已标记样本数据的训练过程，而这些已标记样本通常可以从历史数据或者领域知识中获得。

在本章面向 SBFL 的无标签测试用例分类过程中，标签集合 Y 就相当于测试用例正确性标签集合 $\{passing, failing\}$，待分类的无标签样本集合 X 就相当于无标签测试用例集合 T^u，而决策函数就可以看成是由 T^u 到 $\{passing, failing\}$ 的映射，即 $h : T^u \to \{passing, failing\}$。决策函数的训练过程需要通过基于已标记数据集 T^l 的训

练得到。记 $T^{l'}$ 为测试用例的分类结果，其表示在分类过程中被附上新的预测标签的测试用例集合，定义式如下：

$$T^{l'} = \{t | t \in T^u, h(o^t) = passing || h(o^t) = failing\} \tag{5.7}$$

这里，$h(o^t)$ 表示通过分类过程得到的测试用例 t 的预测标签。在 SBFL 中，一个测试用例只有在正确性被标记的情况下，才能够被应用到可疑度的算中。因此，测试分类是利用无标签测试用例的前提。

分类完成后，我们需要对分类结果进行合理处理并将其应用到 SBFL 的可疑度计算中。也就是说，可疑度计算过程将最初就标记好的测试用例集合 T^l，以及新标记的测试用例集合 $T^{l'}$ 作为输入，而输出则是最终得到的可疑度列表 $L^{l,u}$。

总结来说，整个 $L^{l,u} = SBFL(T^l, T^u)$ 过程可以分为以下两个步骤：

（1）设计用来预测 T^u 中测试用例正确性标签的分类器，并得到新的带有预测标签的测试用例集合 $T^{l'}$。

（2）设计合理的策略将测试分类结果应用到 SBFL 可疑度列表的计算中，以提升 SBFL 的性能。

5.3.3 基于可疑度概率的测试分类器

本小节提出基于可疑度概率的测试分类器设计方法。在一般性的分类过程中，我们需要从论域中选择一些特征，进而构建一个特征空间。而分类器训练过程就是要寻找一个估值函数来量化各个样本以及各个类别之间的相似性。最终，通过这个估值函数构建决策函数，来根据各样本与类别之间的相似性为其附上预测标签。

本章将程序每一条语句的覆盖情况视作特征。具体来说，一条测试用例 t 可以通过其覆盖信息 C^t 进行量化。之后，我们会度量该测试用例与各个类别（既标签为 $passing$ 的成功测试用例类和标签为 $failing$ 的失效测试用例类）的相似性。而相似性的度量标准则需要基于当前已经被标记的测试用例集合 T^l 进行训练。假设测试用例 t 覆盖了语句 s，则 $s \in C^t$ 与 $o^t = failing$ （或者 $o^t = passing$）的关系由两方面决定：

（1）语句 s 包含缺陷的可能性；

（2）如果 s 包含缺陷，那么其导致软件失效的可能性。

第一个方面的解答正是缺陷定位的目标本身。软件的缺陷组件与软件失效的产生之间的关系是非常复杂的，而 SBFL 方法本身就是在这方面较为完善的研究成果。在缺陷传播过程中，错误语句作为程序失效的源头，其一旦被触发，整个程序会被“感

染”。而这个感染状态还会影响到之后多条语句的执行结果，并最终影响程序的输出。而一个合理的可疑度计算公式正是对这一过程合理的体现与预测。因此，SBFL 可疑度计算公式本身的合理性同样可以用来指导测试分类器的设计。

传统 SBFL 基于带标签测试用例集 T^l 生成的可疑度列表很可能并不准确，当然这也正是利用无标签样本的直接目的。然而该可疑度列表切实地对缺陷语句做了估计，且由于 SBFL 的合理性，该估计在只考虑 T^l 的情况下是合理的。因此，若试图在 T^l 提供的数据基础上针对“语句 s 是否包含缺陷”这一问题进行预测，那么传统的 SBFL 本身就是一个合理的方案。

首先基于 T^l 执行传统 SBFL，即执行过程 $L^l = SBFL(T^l)$，并得到可疑度列表 L^l。这里 L^l 的定义式如下：

$$L^l = [\lambda_1^{L^l} \lambda_2^{L^l} \cdots \lambda_n^{L^l}] \tag{5.8}$$

其中

$$\begin{aligned}
&\lambda_k^{L^l} = \langle s_k^{L^l}, r_k^{L^l} \rangle \\
&s_k^{L^l} : \text{排在第 } k \text{ 位的语句}, \\
&r_k^{L^l} = R_{Ochiai}(s_k^{L^l}, T^l) \\
&= \frac{a_{ef}^{s_k^{L^l}, T^l}}{\sqrt{(a_{ef}^{s_k^{L^l}, T^l} + a_{nf}^{s_k^{L^l}, T^l})(a_{ef}^{s_k^{L^l}, T^l} + a_{ep}^{s_k^{L^l}, T^l})}}
\end{aligned} \tag{5.9}$$

接着，将 L^l 中的可疑度值归一化，使其在 $[0, 1]$ 之间变化。记 $\overline{L^l}$ 为 L^l 归一化后的列表，其中各语句的可疑度值 $r_k^{\overline{L_l}}$ 依下式进行重新计算：

$$r_k^{\overline{L^l}} = r_k^{L^l} / \sum_{i=1}^{n} r_i^{L^l} \tag{5.10}$$

对于每条语句 s，找到其在 $\overline{L^l}$ 中的排序。设语句 s 在 $\overline{L^l}$ 中排第 k 位，则有

$$\overline{r}^{s,T^l} = r_k^{\overline{L^l}}. \tag{5.11}$$

假设 SBFL 采用的可疑度计算公式是合理的，且能够反映各语句的缺陷倾向性，则给出一条语句 s，其包含缺陷的可能性可由下式计算：

$$\hat{p}(s = s^f | T^l) = \overline{r}^{s,T^l}, \tag{5.12}$$

其中 $s = s^f$ 表示语句 s 是缺陷语句。

而设计测试分类器所需要讨论的第二个方面，即“缺陷语句能否产生失效”，是对软件经过缺陷语句后，触发失效机率的讨论。这里，缺陷语句被经过后的失效率能够借助 T^l 中的信息通过条件概率公式来估计。假设 $s = s^f$，则测试用例 t 的失效率可以通过下式计算：

$$p(o^t = failing|s = s^f) = \begin{cases} q_e^{s,T^l}, & s \in C^t \\ 0, & s \notin C^t \end{cases} \tag{5.13}$$

其中

$$q_e^{s,T^l} = a_{ef}^{s,T^l}/(a_{ef}^{s,T^l} + a_{ep}^{s,T^l})$$

通过执行式 (5.12) 与式 (5.13)，就能够完成测试分类器的训练。则在实际分类过程中，给定测试用例 t，其失效概率可以通过全概率公式进行预测。

$$\hat{p}(o^t = failing) = \sum_{s \in C^t} \hat{p}(s = s^f|T^l)p(o^t = failing|s = s^f) \tag{5.14}$$

若 $\hat{p}(o^t = failing)$ 的值很大，那么运行 t 产生失效的概率就越大。$\hat{p}(o^t = failing)$ 可以看成是测试用例 t 与标签为 $failing$ 的失效类的相似性度量。而 t 与标签为 $passing$ 的运行成功测试用例类的相似性度量则可通过 $1 - \hat{p}(o^t = failing)$ 计算得到。因此分类器最终的决策函数如下式所示：

$$h(o^t) = \begin{cases} failing & \hat{p}(o^t = failing) > \theta_f \\ passing & \hat{p}(o^t = failing) < \theta_p \\ null & otherwise \end{cases} \tag{5.15}$$

此处，阈值 θ_f 与 θ_p 需要通过权衡测试分类所获得的收益与当分类器作出错误判断（即分类出错）时所付出的代价来确定。如果 θ_f 的值设得很大而 θ_p 设得很小，那么分类器错分带来的风险就会很低，但分类器的预测标准就会非常严格，预测能力也会随之下降。由于本章主要讨论一般缺陷定位环境下的测试信息利用问题，因此在实验中，我们在没有任何领域知识的情况下将 θ_f 与 θ_p 的值都设为 0.5。

总结来说，作为本小节中分类器设计的核心思想，我们基于已标记样本 T^l 运行传统的 SBFL，并将得到的分辨率准确度较低的粗糙可疑度列表作为各语句缺陷倾向性的初步评估模型，将其应用到分类器训练中。也就是说，整个测试分类器的训练所基于的领域知识正是 SBFL 本身。因此，我们将本章提出的分类器叫做基于可疑度概率

（Suspiciousness-Probability-based，SP）的测试分类器，即 SP 分类器。在控制论视角下，SP 分类器的设计过程可以看成是通过观测传统 SBFL 的结果，对当前缺陷定位系统运行状态进行辨识的过程。而辨识的结果则会被用来改善 SBFL 本身。因此 SP 分类器的设计理念包含了自适应思想。

现考虑 SP 分类器的复杂度，首先考虑公式 (5.8) 到 (5.12) 所描述的训练过程，其复杂度与传统的 SBFL 相同。此外，在对每个测试用例 t 进行分类的过程中，式 (5.13) 到式 (5.15) 都会被执行。这其中每个语句的覆盖情况都会被检查并进行相关计算。因此，整个分类过程的复杂度与无标签测试用例集 T^u 的规模以及可执行语句集 S 的规模成正比，即 $O(|T^u| \cdot |S|)$。

5.3.4　分类结果处理与最终可疑度的计算

测试分类完成后，会得到新标记的测试用例集合 $T^{l'}$。本小节将讨论如何将 T_l' 应用到可疑度计算上，进而提升缺陷定位的准确度。在传统的 SBFL 过程中，测试用例的标签只会在计算程序谱统计信息时会被用到。具体来说，基于 T^l，每条语句 s 都会关联相应的程序谱统计量 a_{ef}^{s,T^l}、a_{ep}^{s,T^l}、a_{nf}^{s,T^l} 以及 a_{np}^{s,T^l}。此外，整个测试用例集的失效与成功测试用例数为 F^{T^l} 与 P^{T^l}。

从各种缺陷定位可疑度公式中可知，当可疑度公式 R 被选中，那么这些统计量参与到可疑度计算的方式也就随之被确定了。因此，新标记的测试用例集合 $T^{l'}$ 参与到可疑度计算过程的途径就是将其融入到各个统计量的计算中。具体来说，我们修改传统 SBFL 中的统计量计算方法，使其同时考虑原始带标签测试用例 T^l 以及通过测试分类得到的新标记测试用例 $T^{l'}$，具体如下式所示：

$$
\begin{aligned}
a_{ef}^{s,T^l,\alpha T^{l'}} &= a_{ef}^{s,T^l} + \alpha|\{t \in T^{l'} | h(o^t) = failing, s \in C^t\}| \\
a_{ep}^{s,T^l,\alpha T^{l'}} &= a_{ep}^{s,T^l} + \alpha|\{t \in T^{l'} | h(o^t) = passing, s \in C^t\}| \\
a_{nf}^{s,T^l,\alpha T^{l'}} &= a_{nf}^{s,T^l} + \alpha|\{t \in T^{l'} | h(o^t) = failing, s \notin C^t\}| \\
a_{np}^{s,T^l,\alpha T^{l'}} &= a_{np}^{s,T^l} + \alpha|\{t \in T^{l'} | h(o^t) = passing, s \notin C^t\}| \\
F^{T^l,\alpha T^{l'}} &= F^{T^l} + \alpha|\{t \in T^{l'} | h(o^t) = failing\}| \\
P^{T^l,\alpha T^{l'}} &= P^{T^l} + \alpha|\{t \in T^{l'} | h(o^t) = passing\}|
\end{aligned}
\tag{5.16}
$$

那么通过式 (5.16) 计算得到的统计量就可以代替原来得到的统计量而参与到可疑度计

算中了。例如采用 Ochiai 公式，则语句 s 的可疑度值可通过下式计算：

$$R_{Ochiai}(s,T^l,T^{l'})=\frac{a_{ef}^{s,T^l,\alpha T^{l'}}}{\sqrt{(a_{ef}^{s,T^l,\alpha T^{l'}}+a_{nf}^{s,T^l,\alpha T^{l'}})(a_{ef}^{s,T^l,\alpha T^{l'}}+a_{ep}^{s,T^l,\alpha T^{l'}})}} \tag{5.17}$$

最终我们将根据式 (5.17) 得到的新可疑度值对各语句进行排序，并得到新的可疑度列表为 $L^{l,u}$。

式 (5.16) 考虑了新标记测试集 $T^{l'}$ 并赋予其一个权值 α。由于分类过程中会存在错分样本，因此在利用分类结果计算可疑度值时，测试用例集 $T^{l'}$ 的权值应该小于 T^l。本章将 α 的值设得足够小，使其对任意语句 s_1,s_2 满足

$$R(s_1,T^l)>R(s_2,T^l)\rightarrow R'(s_1,T^l,T^{l'})>R'(s_2,T^l,T^{l'}) \tag{5.18}$$

式 (5.18) 表示，对于任意两条语句，若在基于 T^l 得到的可疑度列表 L^l 中，其可疑度值已经能够被区分，那么它们之间的序关系在基于 T^l 与 $T^{l'}$ 得到的可疑度列表 $L^{l,u}$ 中不会被改变。

这样设置的目的在于保证 T^l 在缺陷定位中的高优先级地位。尽管 $T^{l'}$ 权值很小，但其仍然能够体现对无标签样本在缺陷定位过程中的价值。由于原始已标记的测试用例是不充足的，因此很多语句的可疑度值无法被分辨。这很可能导致大量语句的可疑度值与缺陷语句的可疑度值相同，进而使得可疑度列表 L^l 中存在很长一段不可分辨区间（即 tie interval)[2]。而考虑了 $T^{l'}$ 后，测试信息的多样性程度得以提升。无论 α 的值有多小，其都能够对这些不可分辨区间中的语句给予区分。因此，这种对 $T^{l'}$ 的利用方案实际上是对原始可疑度列表的进一步细化，使其分辨率得到切实提高。同时，较小的 α 值还能够在一定程度上避免由于分类错误而带来的一些负面影响。

5.4 实验研究

5.4.1 研究问题

本节将通过实验来评估无标签测试用例处理方法的性能，此外对于该方法的一些重要性质，我们都将做了进一步讨论。下面列出实验部分的主要研究问题。

研究问题 1： 本章提出的无标签测试用例利用方法是否能在 Oracle 缺失的缺陷定位环境下提升缺陷定位的准确度？

研究问题 2： 影响本方法性能的实验对象相关因素有哪些？

研究问题 3： 本方法在执行过程中需要多少计算量？

表 5.1　实验对象及其特征

程序集	程序名称	程序版本数	语句行数（LOC）	测试用例数量
UNIX	*flex*	53	10,459	567
	grep	17	10,068	809
	gzip	17	5,680	217
	sed	26	14,427	370
Siemens	*print_tokens*	5	472	1,608
	print_tokens2	9	399	2,650
	replace	29	512	2,710
	tcas	41	141	1,052
	schedule	5	292	4,130
	schedule2	9	301	4,115
	tot_info	23	440	5,542
Space	*space*	33	6,199	13,585

研究问题 4： 测试分类器的分类准确度与软件缺陷定位效率之间存在怎样的关系？

研究问题 1 主要关注在缺陷定位中利用无标签样本的可行性以及本章方法的有效性。**研究问题 2** 主要讨论方法的影响因素及详细特性。**研究问题 3** 关注方法的执行效率，这对其实用性是非常重要的。最后，通过对**研究问题 4** 的讨论，我们试图了解测试分类结果对 SBFL 准确性的影响，以及由于错分类而带来的风险，进而分析该方法的鲁棒性与稳定性。

5.4.2　实验设置

本章实验基于三组主体程序集，即包含四个主体程序的 *UNIX* 程序集、包含 7 个子程序的 *Siemens* 测试套件以及 Space 程序集。总的来说，实验对象包含 12 个主体程序。在 *UNIX* 程序中，*flex* 是一个词法分析工具，*grep* 是一个命令行文本匹配工具，*gzip* 是文件解压缩工具，而 *sed* 则是用于批处理的文本编辑工具。*Siemens* 套件包含七个小规模的程序，其本身就是为软件测试的研究而引入的。最后，*Space* 是一种定义语言的解释器。

与前面两章的实验一样，每个程序都有自己的标准缺陷版本。本实验将同时考虑单缺陷版本以及多缺陷版本。首先，尽管实际程序通常包含多个缺陷，对单缺陷的讨论依然是非常重要的。在实际调试过程中，调试员很可能会集中精力修改一个缺陷而过滤掉由其他缺陷引发的失效。对于每一个单缺陷版本，我们植入一个标准的缺陷语句，而屏蔽掉其它缺陷语句。另外，我们同样会对多缺陷版本进行实验。这里每一个多缺陷版本都是通过 2 至 3 条缺陷语句的随机组合生成的。对于双缺陷版本，所有缺

陷语句可能的排列组合都会被考查。而对于三缺陷版本，我们在每个程序上随机检查50 个缺陷语句组合。实验程序的源代码、标准缺陷版本都是从软件基础信息库中获取（Software Infrastructure Repository，SIR）[114]。实验执行过程中，每当一个缺陷版本被选定，那么该版本对应的缺陷语句将被编译执行，而与之相对的正确语句，将被注释掉。

表5.1 列出了每个实验对象的详细信息，包括标准缺陷版本数、代码行数以及附带的测试用例全集所包含的测试用例数量。为了回答**研究问题 1**–**研究问题 3**，我们采用本章所提出的无标签测试用例利用方法进行缺陷定位，并将结果与传统 SBFL（即仅仅基于原始带标签样本的 SBFL）得到的结果进行比较，并以此分析本章方法整体性能。

对于每个缺陷版本，将 SIR 中给定的测试用例集随机分成 T^l 与 T^u，并假设 T^l 中测试用例的标签已知而 T^u 中的标签未知。由于 SIR 提供的测试用例集规模较大，因此为了满足本章研究问题的基本假设，即 $T^l \ll T^u$，我们将 $|T^l|$ 与 $|T^a|$ 的比例设为一个较小的值 0.05。此外 0.1 与 0.15 这两个比值也会在分析带标签训练样本的规模对缺陷定位性能的影响（即对**研究问题 2**的探索）时被考虑到。考虑到分类过程中初始带标签样本集 T^l 的随机性，我们在考查不同方法在某个版本上的性能时会针对该版本进行 20 次重复试验，并采用 p-值为 0.05 的 t 检验对缺陷定位性能进行比较，以此来决定哪那个方案能够获得更低的 Expense 值。

很多研究指出，当 SBFL 给出的可疑度列表具有过高的 Expense 值时，那么调试人员很可能放弃对该列表进行检查，转而回到手工调试中，而此时再对 Expense 值进行讨论就没有意义了。因此在本章实验中，我们只考虑 Expense 值小于 0.1 的那些实例，本阈值的设定还可以在类似文章中见到[135]。具体来说，每当比较两种方法的性能时，我们先对每种版本进行单边的 t-检验，将 Expense 值小于 0.1 作为备择假设，并略去那些无论在传统 SBFL 还是在本章方法下都拒绝原假设的缺陷版本。

为了对**研究问题 4** 进行探索，我们在假设 $|T^l|$ 与 $|T^a|$ 的比值为 0.05 的情况下单独运行本章提出的 SP 测试分类器，并引入 F-measure 来度量测试分类的效果。在测试分类中，将“失效”与“成功”两个标签分别看成是“negative”与“positive”。则可定义

True Positives (TP)：被正确分类为成例的成例数量；

False Positives (FP) ：被错误分类为成例的失例数量；

True Negatives (TN)：被正确分类为失例的失例数量；

False Negatives (FN)：被错误分类为失例的成例数量；

基于以上定义可以进一步定义“精确率”与“召回率”：

$$Precision = \frac{TP}{TP + FP} \tag{5.19}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.20}$$

最终测试分类的 F-measure 值可由下式计算：

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.21}$$

此外，我们还考虑了一个完美分类器，即假设一个能够将所有无标签测试用例都正确分类的分类器，并观察在该分类器下 SBFL 的表现。这么做主要是为了进一步考查缺陷定位效率与分类器的分类性能之间的相关性。

5.4.3　实验结果

本小节主要展示实验结果并对其做简要的说明。表5.2 给出了利用无标签测试用例与仅考虑带标签测试用例所得到的缺陷定位效果的 t 检验比较结果，这里 $|T^l|/|T^a|$ 取值为 0.05。其中第一列指出实验对象是单缺陷版本还是多缺陷版本，第二列中的“B”与“W”分别表示利用无标签测试用例后的 SBFL 方法优于与劣于传统 SBFL 的缺陷版本数，而第 3 至 8 列分别列出了各个实验对象上的比较结果。例如，第 4 列第 2 行中的数据表示，在 *grep* 程序中，本章方法在 41.1% 的缺陷版本上好于传统 SBFL。最后，第 9 列列出了所有实验对象的平均值。这里，本章提出的方法好于传统 SBFL 指的是其生成的可疑度列表在 t 检验下具有更低的 Expense 值。另外，表5.3 与表5.4 分别列出了不同比例的训练样本集（即 $|T^l|/|T^a|$ 分别为 0.1 与 0.15）上的实验结果。

图5.4 详细描绘了无标签测试用例处理方法应用在 SBFL 上的性能，其中每个子图刻画了一个实验对象上的结果。其中 x 轴代表 Expense 值而 y 轴代表 Expense 值等于或低于相应 x 坐标的缺陷版本数。图中的一个点 (a, b) 就表示一共有 b 个缺陷定位实例的 Expense 值小于或等于 a。例如一条曲线通过点 $(0.01, 20)$ 表示有 20 个缺陷版本的 Expense 值小于或等于 1%。这些图可以解释为当程序员按照可疑度列表检查特定百分比的语句时能够定位的缺陷版本数。在每个子图中，黑色实线记录了本章方法的性能而绿色虚线则记录了传统 SBFL 的性能。此外，我们仍然假设一个“完美分类器”，

表 5.2　本章方法在 $|T^l|/|T^a| = 0.05$ 下的 t 检验结果

		flex	*grep*	*gzip*	*sed*	*Siemens*	*space*	总和
单缺陷版本	B(%)	18.8	41.1	53.3	37.5	6.6	10.7	17.4
	W(%)	7.5	11.7	20	16.7	6.6	0	8.1
多缺陷版本	B(%)	25.9	41.6	58.6	42.3	5.1	10.0	19.7
	W(%)	5.6	15.3	17	6.5	4.2	3.2	6.6

表 5.3　本章方法在 $|T^l|/|T^a| = 0.1$ 下的 t 检验结果

		flex	*grep*	*gzip*	*sed*	*Siemens*	*space*	总和
单缺陷版本	B(%)	18.8	29.4	53.3	25	5.8	3.6	14.3
	W(%)	9.4	17.7	13.3	8.3	4.1	3.6	6.9
多缺陷版本	B(%)	22.9	26.8	56.6	34.6	4.3	2.0	15.3
	W(%)	7.8	11.7	14.3	6.1	4.2	1.2	6.2

表 5.4　本章方法在 $|T^l|/|T^a| = 0.15$ 下的 t 检验结果

		flex	*grep*	*gzip*	*sed*	*Siemens*	*space*	总和
单缺陷版本	B(%)	15.1	17.7	53.3	29.1	2.8	0	11.2
	W(%)	9.4	17.7	13.3	4.2	4.1	3.6	6.5
多缺陷版本	B(%)	20.1	20.6	56.0	35.4	2.5	0.6	12.0
	W(%)	9.6	18.4	10.4	3.1	4.8	2.1	7.2

其能够完全正确地对任何无标签测试用例进行分类。这个“完美分类器”的性能则由青色虚线来刻画。

表5.5 列出了不同缺陷定位过程的计算时间。其中第二、三两行分别记录了执行传统 SBFL 以及考虑测试分类的 SBFL 花费的时间。此外，分类过程本身（即只分类而不执行 SBFL）的时间也在表格的第四行被记录下来。同样，表格的每一列都对应着特定的目标程序。

图5.5展示了各目标程序测试分类的 F-measure 值，这里单缺陷与多缺陷版本被放在一起。在每一个柱状图中，横坐标表示 F-measure 值从 $(0, 0.1]$ 到 $(0.9, 1]$ 的变化区间，而 y 轴则表示 F-measure 在特定区间的百分比。

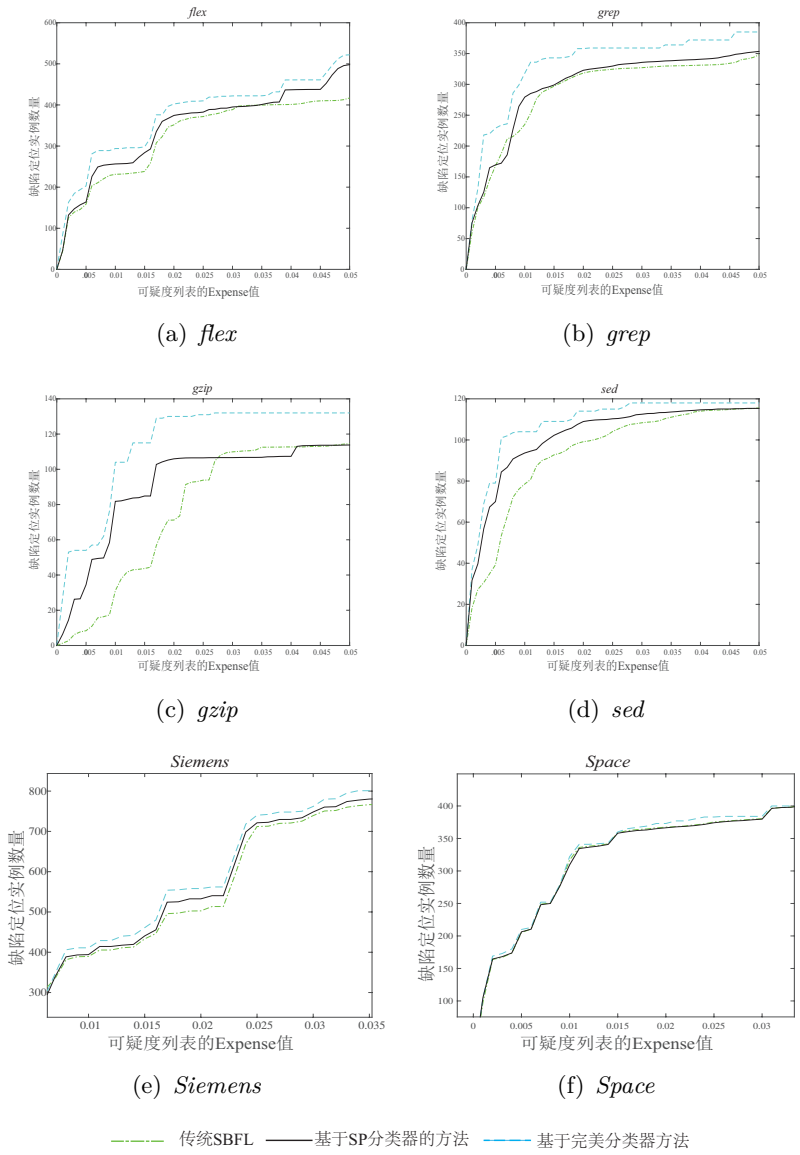


图 5.4　不同解决方案下 SBFL 的 Expense 值与小于该值的缺陷定位实例数量之间的关系

5.4.4　结果分析

本节将对第5.4.3节中的实验结果进行分析，并在此基础上回答**研究问题 1–研究问题 4**。

5.4.4.1　对研究问题 1 的探究

由表5.2 可以看出，对于单缺陷版本，本章方法比传统 SBFL 在 17.5%（45）的缺陷版本上有提升，是下降版本数的两倍多（下降版本比例为 8.1%）。而对于多缺陷版本，本章方法取得的提升更为明显，其在 19.7%（790 个）的版本上本章方法优于传统 SBFL，而仅仅在 6.6% (272 个) 版本上缺陷定位效率下降。取得提升的缺陷版本数是

表 5.5 SP 分类器应用在 SBFL 上的执行效率

Computation Time(s)	*flex*	*grep*	*gzip*	*sed*	*Siemens*	*space*
传统 SBFL	0.02	0.02	0.02	0.01	0.0	0.05
本章方法	0.37	0.38	0.08	0.11	0.02	7.06
SP 分类器单独运行	0.33	0.31	0.07	0.10	0.02	6.74

性能下降的缺陷版本数的三倍之多。尽管在一些实验对象上提升的版本与下降的版本相等甚至更少（例如 *Siemens* 程序的单缺陷版本以及 *Space* 程序在 $|T^l|/|T^a| = 0.15$ 时的缺陷版本中），但总体的结果清楚地显示了采用本章方法获得的收益。总的来说，采用本章提出的基于 SP 分类器的无标签测试用例处理方法后，SBFL 性能提升的缺陷版本数明显多于性能下降的缺陷版本数。考虑所有 T^l 的比例配置，本章方法在 2025 个缺陷版本上取得了提升，远多于性能下降的缺陷版本数（882 个）。

此外，从图5.4 中也可以直观地看出，对于大多数程序，用于描绘本章方法性能的曲线（即黑色实线）都在传统缺陷定位方法所对应的曲线（即绿色点划线）之上。这意味着在实际缺陷定位环境中，给定一个 Expense 阈值，本章方法能够定位更多的软件缺陷。

作为对**研究问题 1**的回答，与仅仅采用带标签样本的传统 SBFL 相比，通过对本章方法能够切实提升软件缺陷定位的效率。这说明在软件缺陷定位中，无标签测试用例确实能够在一定程度上反映缺陷语句的位置信息，而对这些测试用例的利用是有价值的也是值得研究的。

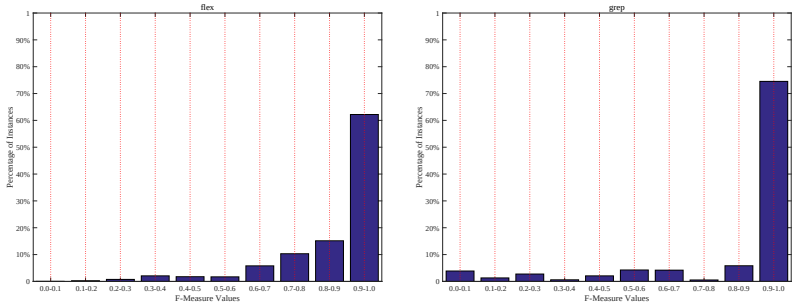
5.4.4.2 对研究问题 2 的探究

为了回答**研究问题 2**，我们对影响方法性能的实验对象相关因素进行进一步探索，这其中包括带标签测试用例（即训练样本）的数量、不同的实验对象及缺陷语句数量。

（1）原始带标签样本（即训练样本）的比例

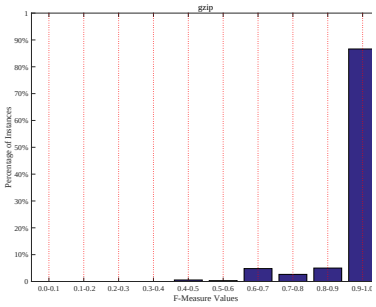
从表5.2 – 表5.4 的比较数据中可以看出，不同训练样本比例（即 $|T^l|/|T^a|$）对方法性能的影响是不同的。例如，当 $|T^l|/|T^a|$ 从 0.05 变为 0.1 时，本章方法性能在 *sed* 与 *gzip* 程序上取得了提升，在 *flex* 与 *Siemens* 程序上的性能没有太大变化，而在 *grep* 与 *Space* 程序上的性能反而下降了。当 $|T^l|/|T^a|$ 的比值从 0.1 变为 0.15 时，其在 *sed* 上的性能提升更加明显了，但其在其他程序上的性能有了一些下降。

实验结果表明，缺陷定位效率与训练样本集规模之间的关系是比较复杂的。一方

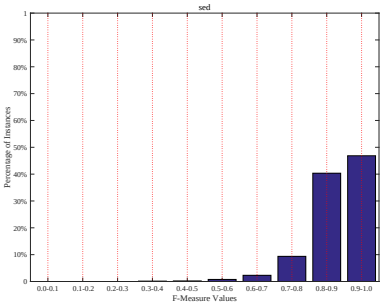


(a) *flex*

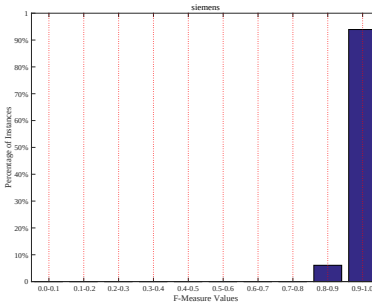
(b) *grep*



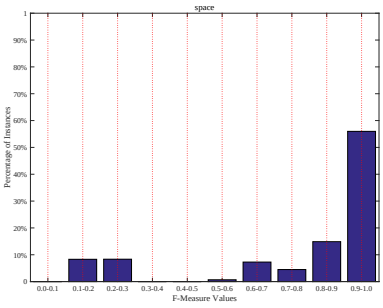
(c) *gzip*



(d) *sed*



(e) *Siemens*



(f) *Space*

图 5.5　SP 分类器分类效果的 F-measure 分布图

面，当 T^l 中的测试信息不充足且不足以支持缺陷定位任务时，随着 T^l 规模的增大，训练样本逐渐增多，分类的准确率提升，这使得更多的无标签信息能够被正确利用，进而提升了 SBFL 的性能。另一方面，对于一些实例来说，T^l 足够充足，以至于仅仅依靠 T^l 就能取得好的缺陷定位结果。此时，利用 T^u 中无标签测试用例的信息所获得的收益就不那么明显了，甚至分类错误所带来的负面效果会占据主导。事实上，这些实例并不满足本章所讨论问题的基本假设，即由于 Oracle 的缺失所导致带标签测试用例不足以支撑缺陷定位的需求。对于一些实验对象来说，SIR 中提供的测试用例集规模足够庞大，以至于仅仅取其中一小部分测试用例就足以支持缺陷定位了。然而，在实际应用过程中，如果 Oracle 缺失，那么当前可以鉴别的测试用例很可能是不能支撑 SBFL 执行的，故我们的方法也会起到积极作用。当然，当前可用测试用例集的充分性

问题也是值得进一步探索的。实际上，考虑不同覆盖准则与缺陷定位效率相关性的研究已经被广泛开展[42, 135]，这些研究工作值得被与本章相关的后续工作所借鉴。

（2）不同程序的影响

从表5.2 中可以看出，对于不同的实验对象，测试分类所带来的收益也是不同的。对于 *flex* 、*gzip* 与 *grep* 程序来说，测试分类所获得的提升非常明显。在这些程序中，测试分类后 SBFL 性能提升的版本数至少是性能下降版本数的两倍。而相反，对于 *Siemens* 的单缺陷版本，本章方法在 8 个缺陷版本上定位效率提升，同时也在 8 个版本上定位效率下降，而在 *Space* 程序上也能观察到类似现象。尽管在这些程序上我们的方法没有取得明显的提升，但从图5.4 中可以看出，我们的方法与传统 SBFL 在 *Siemens* 与 *Space* 两个程序集上的性能曲线非常接近；这说明在这些实验对象上，测试分类同样不会带来太多的消极影响。

（3）单缺陷与多缺陷假设

在对比表5.2 中单缺陷与多缺陷版本的数据后，我们发现本章方法在多缺陷版本的表现要好于其在单缺陷版本下的表现。总的来说，对于多缺陷版本，我们的方法在多于三倍的缺陷版本上好于传统的 SBFL。而在单缺陷版本中，我们的方法近在两倍的缺陷版本上好于传统的 SBFL。我们推测，相比于多缺陷版本，单缺陷实例的缺陷语句是比较容易被定位的，这意味着原始 T^l 中的测试用例对缺陷位置的暴露程度会更高。然而在实际情况中，目标程序通常会有多个缺陷语句，而本章的方法也会更有意义。

从上面对**研究问题 2** 的讨论中可知，本章方法会受到与实验对象相关的若干因素影响，但其在各种配置下的大多数缺陷版本中都能够提升传统 SBFL 的定位性能。另外，即使本章方法受到了某种因素的影响，其也不会产生很大的负面效应。当然，将该方法与现有的缺陷定位技术或评估手段结合，以更清晰地度量这些影响也是非常有意义的研究问题。

5.4.4.3 对研究问题 3 的探究

从表5.5 中可知，本章提出的结合测试分类的 SBFL 方法相比于传统的 SBFL 确实要花费更多的计算代价。而这其中，SP 分类器的分类过程花费的代价是最高的。然而，在实际缺陷定位环境中，SBFL 是一个半自动化的缺陷定位技术。也就是说，其 SBFL 自动生成的可疑度列表还需要经过程序员的进一步检查和确认。因此，与整个缺陷定位过程相比，测试分类所花费的额外代价就非常小了。例如，在 *space* 程序中，分

类超过一万条测试用例所花费的时间少于 10 秒。因此，我们的方法在计算代价上是完全可以接受的。

5.4.4.4　对研究问题 4 的探究

本章方法主要基于 SP 分类器对无标签测试用例进行分类，SP 分类器的分类效果如图 5.5 所示。从图中可以看出，超过 90% 实例的 F-measure 值在 0.95 左右。特别地，对于 *grep* 和 *gzip* 程序，超过 80% 的缺陷版本的 F-measure 值大于 0.95。而对于 *flex* 程序，接近 65% 的实例的 F-measure 值大于 0.95，这个百分比要远远小于 *grep* 与 *gzip* 程序。对于 *sed* 程序，大多数实例的 F-measure 值大于 0.6，而多于 20% 的实例的 F-measure 值在 0.5 至 0.7 之间。

对于 *Siemens* 和 *space* 程序，几乎所有实例的 F-measure 值都在 0.95 以上。然而，当我们检查分类细节时发现这两个程序集中成功测试用例占了绝大部分，而失败测试用例非常的少，这使得 SP 分类器会将所有的测试用例都视为成功测试用例。然而，在这些缺陷版本中，成功测试用数量远远大于失效测试用例，因此尽管失效测试用例不容易被鉴别，其分类的准确度还是很高的。

通过进一步对比分析图5.4 与图 5.5 可以看出，测试分类的准确率与最终缺陷定位的准确度之间确实是正相关的。首先在 *flex* 、*grep* 与 *gzip* 程序中，SP 分类器能够得到比较准确的分类结果，而将其应用到 SBFL 上时也确实能够明显提升软件缺陷定位的准确度。此外，图5.4 刻画了利用完美分类器得到的缺陷定位结果。可以看出，当分类器能够对任何无标签测试样本作出完美的分类时，SBFL 的性能提升是最明显的。另外，尽管 SP 分类器在 *Siemens* 、*sed* 与 *space* 这三个目标程序上的表现并不如其在 *flex* 、*grep* 与 *gzip* 上的表现，我们的方法在 *sed* 上的提升比其他程序还要明显。对于 *Siemens* 程序，尽管方法并没有提升 SBFL 的性能，但从其与传统 SBFL 比较结果可知，应用 SP 分类器本身并没有带来明显的负面影响。这样的实验结果可以从两方面进行分析。一方面，我们发现 SP 分类器在大多数分类实例下都能做出准确地分类，而且测试分类确实是将无标签样本信息应用在缺陷定位上的有效方法。此外，本章中基于 SP 分类器的方法与完美分类器下得到的缺陷定位结果仍存在明显差距。这一现象更说明了缺陷定位准确度的提升对测试分类的准确性是有依赖性的。也就是说，当我们能够得到更加准确的测试分类结果时，缺陷定位的性能就会取得更大的提升。这也说明了我们的方法在分类器设计环节还有很大的提升空间。

而另一方面，对于那些分类效果不是很好的实例，SP 分类器也没有带来过多的消

极影响。考虑其原因，SBFL 的可疑度列表是基于所有的测试用例全集得到的，包括原始的带标签测试用例以及新标记的测试用例。因此，测试用例正确性标签中存在少量的错误并不会对最终的可疑度列表产生太大的影响。另外一个原因是在第5.3.4章对分类结果进行处理时，我们赋予新标记的测试用例一个很小的权值，这在一定程度上限制了可疑度列表的变动幅度。需指出大量正确标记的测试用例仍足以对基于 T^l 得到的可疑度列表进行细化。例如，当巧合成例非常多时（例如 *Siemens* 程序），分类器的训练样本集中会存在很多误导信息，因此分类器非常容易将一个失效测试用例的标签标记为 $passing$。然而，此时 T^l 中包含的误导信息除了会影响分类结果以外同样会影响传统 SBFL 给出的 L^l 的准确度。这使得测试分类的不准确不会对基于 T^l 产生的可疑度列表有太大的改变。对于 *sed* 这样的程序，原始带标签测试用例集 T^l 本身是很不充足的，因此尽管测试分类结果不是很准确，但其仍能够提升 SBFL 的定位性能。

5.5 方法的潜在风险

本章方法的其中一个外在风险来自于实验对象本身。我们在实验研究中选择三组程序作为实验对象。这三组程序规模各不相同，从只有几百行代码到上万行代码，其在软件缺陷定位的研究中经常被使用 [74, 47, 42, 135, 136]。然而，考虑到软件的多样性，本章方法的有效性以及适应性需要在更多程序、编程语言以及失效模式下进行进一步检验。

另一个风险来自于软件缺陷。本章实验对象所有的缺陷版本都是从软件基础信息库 (SIR) 中获取的，而多缺陷版本则是通过合并不同的单缺陷版本来实现的。我们承认，这些缺陷版本以及合并机制不能覆盖所有实际编程中遇到的缺陷特征，更多的缺陷以及不同缺陷间的交互与融合机制有待进一步检验。

此外，在实验研究中，我们假设原始的带标签样本——其作为分类器的训练样本——是从整个测试用例全集随机选取并生成的。然而对于特定的测试策略（如边界测试）来说，测试用例选择机制与执行剖面分布都会有很大不同。因此，当前代标签测试用例作为测试分类器的训练样本是否充分，以及如何在不同测试用例分布的情况下设计与之相适应的测试分类器也是需要进一步研究的。

方法最主要内部风险则来自于 SP 分类器。在机器学习领域，已经被提出并研究的分类器种类非常多，它们在不同场景下都有比较坚实的理论与应用基础。而这些分类器（例如朴素贝叶斯、SVM、神经网络等）在测试分类中的表现以及对 SBFL 的促进

作用都是值得分析与研究的。实际上，在本章的研究过程中我们也对上述一些分类器进行了尝试；然而我们发现，在缺陷定位这个应用环境下为这些分类器设置适当的参数是较为困难的。此外，针对 SP 分类器机理的深入分析与探索，以及加入特定目标软件所涉及的领域知识都是值得进一步考虑的。

另一个潜在的内部风险就是在执行 SBFL 时，我们考查了 Ochiai 公式，而没有对其它 SBFL 可疑度计算公式进行考察。Ochiai 公式目前被认为是在单缺陷与多缺陷版本下表现都非常好的公式之一，其在很多研究中被推荐并使用[76, 74, 99, 34]。此外，在 Ochiai 公式中，各语句的可疑度值是通过连续平滑的函数计算的，因此其更适合在 SP 分类器中被当做先验概率来使用。当然，其他被证明有效的公式（如 Op 公式）也是值得进一步研究的。

5.6　结论与展望

本章讨论了在软件测试过程中 Oracle 缺失问题对软件缺陷定位的影响。在 Oracle 缺失的情况下，只有少部分测试用例的正确性能够被鉴别而大多数测试用例都是无标签的，这直接导致了软件缺陷定位效率的下降。本章通过合理利用大量的无标签测试用例，提升了测试信息的多样性，从而提高了缺陷定位的效率。具体来说，本章方法基于测试分类，即将少量已标记的测试用例作为训练样本，训练一个合理的分类器去预测其余测试用例的正确性标签。在此基础上，整合新标记的测试用例与原始测试用例，并将其应用到软件缺陷定位方法中。由于更多的测试用例在缺陷定位过程中被考虑，测试信息的多样性程度被提升了，而软件缺陷定位结果的准确度也随之取得了较大的提升。

我们的方法基于 SBFL。作为方法的一部分，我们提出一种可疑度概率（SP）分类器来对无标签测试用例进行分类。SP 分类器借助 SBFL 本身的领域知识，将传统缺陷定位得到的可疑度列表中各语句的可疑度值作为其包含缺陷的先验概率近似值。分类过后，新标记的测试用例会被赋予一个小的权值并应用到 SBFL 的可疑度计算中。实验结果表明，在大多数缺陷定位实例中，忽视无标签测试用例确实会降低缺陷定位的准确度，而本章方法也确实提升了 SBFL 的定位性能。

总结来说，本章做出以下贡献：

（1）详细讨论了在测试 Oracle 缺失环境下，无标签测试用例在软件缺陷定位过程中的作用。我们相信这在“测试 -定位 -修复”这个过程中是一个重要且实际的问题。

（2）我们设计了 SP 分类器，其基于 SBFL 本身的领域知识，设计理念中包含了自适应思想，属于软件缺陷定位过程中的一种自适应机制。通过实验，SP 分类器的可行性及其应用在 SBFL 方法上的有效性得到了验证。

（3）一整套基于 SBFL 的无标签测试用例利用方法被提出，方法在实际运行时的有效性也在实验中得到验证。

（4）无论是面向缺陷定位本身的分类器设计，还是其执行细节的实验分析都为该领域提出了新的研究方向。

在接下来的研究工作中，最主要的任务就是检验更多的目标程序以及测试分类器，以获得更为普适性的实验结论。我们希望能够设计针对无标签测试用例的更为有效的分类器，并希望这些分类器能够考虑更多细节，例如测试剖面以及程序说明信息等。此外，由于不同软件缺陷所表现出来的特征及行为不同，另一个非常有前景的研究方向就是去鉴别不同的失效特征，或是程序执行模式，并在此基础上针对各种程序架构以及缺陷类型设计特定的分类器。

第六章 SPICA: 基于程序谱图的缺陷定位适应性分析框架

6.1 引言

在软件缺陷定位中，欲准确地找到缺陷代码的位置，除了需要高质量的测试信息外，还需要一套合理的缺陷定位方法来根据现有的信息对缺陷代码位置进行推理。目前最常用的方法就是基于程序谱的缺陷定位方法（Spectra-Based Fault Localization，SBFL）。SBFL 将测试信息转化为程序谱信息，并跟据程序谱信息的统计特征来推断目标程序可能的缺陷位置。SBFL 的核心算法是可疑度计算公式，其决定了 SBFL 根据程序谱统计信息得到各个语句包含缺陷的可疑度的具体计算过程，而一个可疑度计算公式也可以看成是一个具体的 SBFL 技术或算法。因此，对可疑度计算公式的研究也就是对 SBFL 本身机理的研究。对于同一程序，不同可疑度公式计算出来的可疑度列表可能会存在很大差异，这也在很大程度上影响了 SBFL 的定位性能。而如何设计并选择一个合理的可疑度计算公式，以得到更为准确的缺陷位置预测结果则成为 SBFL 研究的一个共同主题。

目前很多学者都针对 SBFL 可疑度公式的合理性问题展开研究。该领域的研究工作大致可以分为两个部分：1）实验研究，即通过各可疑度公式在不同标准程序中的定位效果来判断其合理性[99, 74]；2）理论分析，即从理论上证明公式间的关系，并试图寻找最优公式[75, 2, 137]。对于实验研究来说，SBFL 在标准程序上的缺陷定位性能是其有效性的最直接证据，这些实验数据能让 SBFL 得到最基本的认可。看到数据后，调试人员会认可 SBFL 是一个经过基本验证的方法。此外，通过实验研究，还可初步筛选一些缺陷定位准确度比较好的公式，并且初步排除一些在实验中性能比较差的公式。另一方面，理论分析也是目前非常流行的研究方法。现有的理论工作大多通过综述现有的可疑度计算公式，并在理论上对这些公式进行比较，来鉴别各公式之间的性能差异。借此，我们能够掌握各公式在理想条件下的优劣关系，并且能够对 SBFL 的极限与瓶颈进行初步探索。例如 Naish 等人[75] 发现很多公式尽管在语句可疑度值的计算上存在很大差异，但其反映在 SBFL 最后输出的可疑度列表上是等价的。Xie 等人[2] 对 Naish 等人的工作进行了扩展，他们在单缺陷假设下，比较了各个公式的性能，并且给出了单缺陷假设下的 Maximal 公式。

然而，尽管目前的研究工作提供了大量的理论与实验结果，SBFL 可疑度计算公式在实际缺陷定位环境下的选择与应用仍然存在很大困难。一方面，理论分析能够提供

一些关于各公式性能比较关系的确定性结论，然而作为基本假设的单缺陷假设过于理想，在实际情况中很难被满足。此外，Yoo 等人[137] 得到的最新证明结论表明，即使在单缺陷假设下，也不存在一个最优的可疑度计算公式。这个结论在一定程度上标志着在此方向上整个理论工作的完成，但同时也是在宣告我们无法再试图通过理论证明的方式去寻找最优公式。尽管，各可疑度计算公式间的优劣可以被比较，但其本身的性能则不易被评估与验证。即使某个可疑度计算公式优于其它公式，我们也不确定其在特定的被测对象与测试环境下是否能够对缺陷代码位置进行准确地推断。

而另一方面，尽管实验研究能够给出关于 SBFL 性能的实际数据。然而，由于软件及缺陷的多样性所致，我们无法肯定实验中用到的标准程序及标准缺陷版本对于验证某种 SBFL 技术的有效性来说是足够的。此外，在目前众多实验研究中，不存在任何一个可疑度公式能够在所有的实验设置下一直取得较好的缺陷定位效果。因此在实际应用 SBFL 时，调试人员仍无法确定其在当前的配置环境下会不会有效。总的来说，当前的研究工作无法对调试人员正确地选择及应用 SBFL 技术（即可疑度计算公式）作出系统性指导。这使得调试人员并没有足够的信心去接纳并实际使用 SBFL。因此，无论理论上还是实验上，SBFL 方法本身的研究都面临着巨大挑战。

“一切恐惧都源于未知”。在 SBFL 应用上的最大障碍就是对其基本理论的了解不够深入。这使得调试人员很可能无法在实际缺陷定位过程中正确地选择并使用适当的 SBFL 技术。并且，最关键的问题是其无法排除由 SBFL 适应性问题带来的风险。SBFL 所给出的缺陷位置预测结果是一个可疑度列表，列表中各语句都依其可疑度被排序。通过检查该可疑度列表，调试人员能够知道哪些语句被赋予了更高的缺陷“可疑度”。然而问题是，“可疑”是一个非确定的概念，即使我们知道某些语句在包含缺陷这一事件上很“可疑”，也不能确定该语句是否为缺陷语句，甚至连其包含缺陷的概率都无从知晓。这时，如果不能清楚地了解这些语句“可疑”的原因，调试人员可能放弃对 SBFL 的使用并转而采用传统的手工调试。可见，SBFL 的适应性以及实际缺陷定位环境的未知性给 SBFL 的实际应用带来较大的障碍，而深入了解 SBFL 的执行机理的并对其进行合理地描述对消除这种障碍来说是非常重要的。

深入了解 SBFL 执行机理的途径是对 SBFL 的理论基础进行进一步探索。对于 SBFL 来说，无论是哪种可疑度公式，其内在思想都是模仿人类根据测试信息对软件缺陷的位置进行推理的过程。而描述由测试信息中得到的程序谱与软件缺陷位置之间关系的基本理论就是 SBFL 可疑度计算公式的最根本设计依据。如第二章所述，目前的

可疑度计算公式大多基于以下两个基本原理：

SBFL 基本原理 1： 在失效测试用例中执行频率高的语句更有可能是缺陷语句；

SBFL 基本原理 2： 在成功测试用例中执行频率低的语句更有可能是缺陷语句。

这两个基本原理描述了测试信息、程序失效以及软件缺陷位置之间的最基本联系，是被广泛认同的。然而，这种关系对于定位程序中缺陷语句的具体位置来说是比较弱的。一方面，如本文第一章所述，缺陷语句与程序最终失效之间的因果关系非常复杂。一条测试用例可能经过失效语句，然而其是否会触发软件的失效，还与程序经过缺陷语句时的状态有关；而这种复杂的缺陷传播过程正是软件缺陷定位任务的难点所在。一个经验丰富的调试人员的优势正是体现在对这一过程的领悟上。SBFL 若要获得更好的应用，其依据的理论也必须能够对这种复杂关系进行更为深刻的解读；而这是仅仅依靠上述两个基本原理所不易做到的。另一方面，在失效测试用例中执行频率高与在成功测试用例中执行频率低都不是精确的概念。有些语句在失效与成功测试用例中的执行频率都很高（例如程序主干上的语句），而一些语句的执行率在失例与成例中都很低（如一些不常用的功能函数中的语句）。对于这些语句，上述两个基本原理所给出的判断就会比较模糊，我们也很难仅仅通过这两个基本原理为这些语句安排一个绝对的序关系。这也说明了，为了更准确地对缺陷位置进行推理，我们有必要去探索更为复杂与精确的理论来刻画语句的程序谱信息与其是否包含缺陷之间的关系。

上述 SBFL 的基本原理仅仅涉及单个语句的执行频率，而讨论整个程序谱分布与与软件缺陷之间关系的研究还非常少。目前超过 100 种可疑度计算公式被提出，而这些公式都是对上述两个原理不同形式的表达。尽管不同的可疑度计算公式都拥有与自己的方式来组织程序谱的相关统计参数并计算语句的可疑度，但它们所基于的推理机制大多借鉴于其它领域的概念与技术，而不是来自于程序谱或是程序谱分布本身。

通过上面的论述，本文认为对于程序谱分布本身理论的深入研究，以及基于这些理论对 SBFL 可疑度计算公式对缺陷语句的推理机制进行描述、解释与分析，是评估 SBFL 适应性，并最终提升其应用价值的关键。因此，为了能够在实际中对 SBFL 可疑度计算公式进行更好地选择与利用，以切实提升 SBFL 本身的应用价值，本章试图挖掘程序谱分布的基本理论，并刻画 SBFL 可疑度计算公式的执行机理，进而对其适应性进行评估与分析。具体来说，本章研究从以下两个方面展开：

（1）从程序谱空间划分角度解释不同的可疑度计算公式对缺陷语句的推理机理；

（2）挖掘程序谱分布与缺陷语句之间的关系，以及其对不同可疑度计算公式性能

的影响。

其中，第一个方面主要针对 SBFL 可疑度计算公式本身。尽管一个可疑度计算公式很可能是从其它领域（如几何学、生物学等）的概念与方法中引入的，但其应用到 SBFL 时，除了能够反映 **SBFL 基本原理 1**与 **SBFL 基本原理 2**，还一定有其自己的推理原则——即在不违背基本原则的前提下判断什么样的语句更像是缺陷语句。而可疑度计算公式这种内在的推理机理就是第一个研究方面的侧重点。而第二个方面主要是对程序谱分布基本理论的挖掘，并在此基础上结合各可疑度计算公式的推理原则，分析各公式的合理性与适应性。

具体来说，本章研究采用理论分析与实验数据结合的方式展开，提出一种基于程序谱图与公式曲线的可视化分析框架（Spectra Plot based metrIC Analytical framework for spectrum-based fault localization，SPICA）。SPICA 利用程序谱图（Spectra Plot）与公式曲线（Metric Curve）这两个概念，对程序谱的分布规律以及各可疑度计算公式的执行机理进行描述。在该框架下，整个程序谱空间被抽象为一般性的坐标图，而各语句的程序谱信息以及各可疑度计算公式分别被表示为图中的点与曲线。借此，我们就得到一个整体的、统一的以及可视化的分析框架。

以 SPICA 为基本框架，本章首先对 SBFL 中的可疑度计算公式进行了深入的研究，并分析其缺陷推理机理。其中，我们在单缺陷假设下对各可疑度计算公式间的优劣关系进行了的证明，得到了更为丰富与直观的结论，并在此基础上对 Maximal 公式的本质做出解释。此外，去除单缺陷假设，我们对程序谱空间中的语句分布规律进行分析。以此为基础，我们提出基于程序谱空间划分的语句滤除算法，以提升各可疑度公式的适应性。最后，结合该算法的实验结果，我们对经典可疑度公式的适应性差异进行了分析，并得到了一些有用的结论。

6.2 SPICA 框架概述

6.2.1 程序谱图（SP）

基于程序谱图与公式曲线的可视化分析框架（Spectra Plot based metrIC Analytical framework for spectrum-based fault localization，SPICA）的核心思想是将 SBFL 可疑度计算公式的执行机理进行可视化，并在此基础上对 SBFL 及可疑度计算公式的基本理论进行分析。而程序谱图（Spectra Plot，SP）与可疑度公式曲线（Metric Curve，MC）就是该可视化框架的基本组成部分。

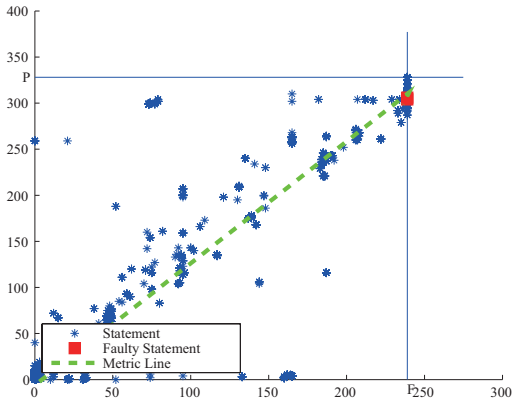


图 6.1　程序谱图（SP）示例

程序谱图（SP）指的是将目标程序 PG 中各语句的程序谱信息投影到直角坐标系中形成的分布图（如图6.1 所示），其具体定义如下：

SPICA 定义 1 (**程序谱图**（SP）)：在表示 a_{ef} 含义的横坐标与表示 a_{ep} 含义的纵坐标所组成的坐标系下绘制的各种图的统称。特别地，记 $G^{PG,T}$ 或 $G(PG,T)$ 为基于目标程序 PG 与测试用例集 T 所形成的程序谱图。

在程序谱图中，横坐标表示 a_{ef} 值，纵坐标表示 a_{ep} 值。而图中坐标为 (x,y) 的点就可以看成是 $a_{ef}=x$ 且 $a_{ep}=y$ 的可执行语句映射到该程序谱图中的成像。如此，假设 PG 的一个测试用例集 T 已经被执行并且执行信息以及输出结果的正确性信息已知，则 PG 中的任何一条语句 s 在程序谱图 $G(PG,T)$ 中都能够找到相应的成像点。而整个程序谱分布就可以可视化为程序谱图中一系列的点（如图6.1 中的蓝色“*”所示）。这里记一条具体的语句的横坐标为 $a_{ef}(s)$，纵坐标为 $a_{ep}(s)$；则程序谱图中代表特定语句 s 的成像点就可以记为 $po(s)=(x(s),y(s))=(a_{ef}(s),a_{ep}(s))$。与前面关于 SBFL 的介绍类似，这里的 $a_{ef}(s)$ 表示测试用例集 T 中覆盖语句 s 的失效测试用例个数，而 $a_{ep}(s)$ 则表示测试用例集 T 中覆盖语句 s 的成功测试用例个数。当然，作为程序谱来说还有另外两个参数 $a_{nf}(s)$ 与 $a_{np}(s)$。它们分别表示测试用例集 T 中，没有覆盖 s 的失效与成功测试用例个数。而由于 a_{nf} 与 a_{np} 的取值没有自由度（分别由 $F-a_{ef}$ 与 $P-a_{ep}$ 的值决定），因此程序谱图只有两个维度。此外，对于程序谱的总体统计参数 F 与 P 来说（F 代表 T 中失效测试用例的总数，而 P 代表成功测试用例总数），对于任意语句 s，显然有 $x(s)\leq F, y(s)\leq P$，因此程序谱图是一个用边界围起来的盒子，其中横坐标的范围为 $[0,F]$，而纵坐标的范围为 $[0,P]$。

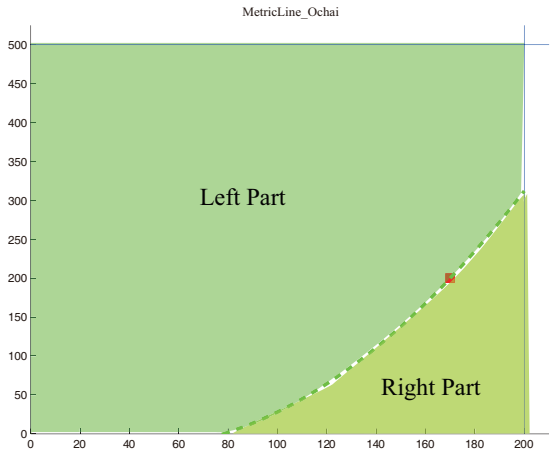


图 6.2　公式曲线（MC）示例

现考虑缺陷语句在程序谱图中的成像点。假设程序中存在错误，则缺陷语句 s^f 同样会在程序谱图上被映射为一个点，其坐标为 $(a_{ef}(s^f), a_{ep}(s^f))$。在实际缺陷定位过程中，缺陷语句一开始不会在图上被着重标出来，因为调试人员并不知道其成像的具体位置。而本章在进行分析时，为了显示清楚，将其用红色方点着重标出。在程序谱图概念下，整个 SBFL 过程就可以表述为在特定程序谱图的点集中寻找缺陷语句成像点的过程。

6.2.2　公式曲线（MC）

SPICA 的另一个重要的概念是可疑度公式曲线，其将可疑度计算公式的性能以曲线（或是曲线族的形式）在程序谱图上表示出来。假设调试人员采用的可疑度计算公式为 R，则 R 对于各语句的可疑度评判机制就可以表示为程序谱图上的一条分割曲线，如图6.2 所示。公式曲线的具体定义如下：

SPICA 定义 2 (**可疑度公式曲线**（MC）)：给定目标程序为 PG，其缺陷语句为 s^f。给定测试用例集 T，其中成例总数为 P，失例总数为 F 。对于程序谱图 $G^{PG,T}$，s^f 有程序谱信息 $a_{ef}(s^f)$ 与 $a_{ep}(s^f)$。则考虑可疑度计算公式 R，定义曲线 $R(x,y) = R(a_{ef}(s^f), a_{ep}(s^f))$ 为 R 在 $G^{PG,T}$ 上的公式曲线，记为 MC^R。

从图6.2 与 **SPICA 定义2** 中可以看出，对于任意可疑度公式曲线来说，其必须经过点 $po(s^f)$，并且将整个程序谱图分割成两个部分。其中一个部分中语句的可疑度高于缺陷语句可疑度，我们称其为错分辨区域。另一部分中语句可疑度低于缺陷语句可疑度，我们称其为可分辨区域。而对于成像在公式曲线上的语句，其可疑度与缺陷语句可疑度相等，因此我们称曲线本身包含的区域为不可分辨区域。若给定在目标程序

PG、测试用例集 T 以及基于它们的程序谱图 $G^{PG,T}$，讨论可疑度公式 R 的公式曲线 MC^R，则记曲线 MC^R 分割而成的可分辨区域为 G^i，不可分辨区域为 G^n，而错分辨区域为 G^e。

通过对程序谱空间的划分，公式曲线将特定的可疑度计算公式性能形象地刻画出来。假设某特定的缺陷定位实例形成的程序谱图为 $G^{PG,T}$，公式曲线 MC^R 将其分割成 G^i、G^n 与 G^e。记 PG 的语句集 S 中成像落在 G^i 上的语句集合为 S^i，成像落在 G^n 上的语句集合为 S^n，而成像落在 G^e 上的语句集合为 S^e，则可疑度公式 R 在该实例上的 Expense 值为（这里 tie-break 策略为取平均值）：

$$E = (|S^e| + |S^n|/2)/|S| \tag{6.1}$$

因此，通过对比程序谱图中各部分语句的比例，我们就能直观地分辨某个可疑度公式的性能了。

在实际情况中，PG 与 T 是已知的，但我们无法预先知道缺陷语句本身的程序谱信息，因此也就无法实际地画出可疑度公式曲线。这就使得实际执行 SBFL 时的定位准确度无法通过程序谱图和公式曲线直接看出来；若如此，就相当于我们已经知道缺陷语句的位置了，那么 SBFL 以及软件缺陷定位这个过程本身也就变得没有意义了。然而，无论能否被画出，可疑度公式曲线本身都是客观存在于程序谱图中的，而本章所做的讨论也是对客观存在事物性质的一种探索。此外 SPICA 本身是一个程序谱基础理论分析框架，而 SP 与 MC 都是相应的分析工具，因此利用这些工具去探索 SBFL 的基本理论实际上是对 T 与 PG 的泛化讨论。不同的可疑度公式 R 分割程序谱图的方式是不同的，这使得公式曲线以及被其分割出来的区域也会呈现出不同的几何特征。例如图6.2 中，公式曲线的几何特征就呈现为抛物线形状。而通过分析该几何特征所反映出的程序谱图划分机理，我们就能够了解相应可疑度计算公式的缺陷推理机制。

需指出，对于一些不连续的可疑度计算公式，其公式曲线很可能是一个面，也可能是一个不包含图中任何点的开区间。因此，这里所谓的“曲线”实际上是广义上的，无论其是否符合几何中“线”的定义，其都具备划分程序谱空间的能力，且其几何特性都可以被鉴别与分析。这里为了简化分析过程，我们对普通曲线公式进行如下定义：

SPICA 定义 3：若可疑度计算公式 R 的公式曲线 MC^R 在定义域 $[0, F] \times [0, P]$ 内连续、单调递增且在各点存在有限的左右导数，则称 MC^R 为普通公式曲线，并称 R 为普通曲线公式。

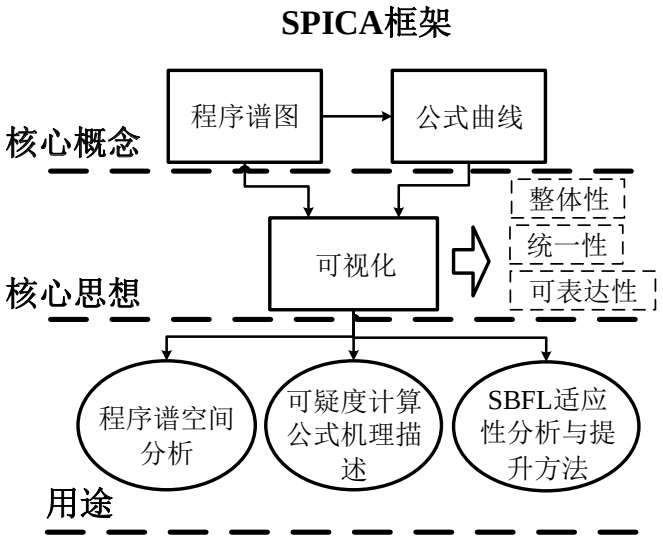


图 6.3　SPICA 框架示意图

本章主要针普通公式曲线以及普通曲线公式进行讨论。

最后讨论公式曲线的绘制方法。考虑程序谱图 $G(PG,T)$，假设缺陷语句 s^f 在 $G(PG,T)$ 的像为 $(a_{ef}(s^f),a_{ep}(s^f))$，则考虑可疑度计算公式 R，其公式曲线可以通过以下步骤绘出：

步骤 1：通过可疑度计算公式 R 计算缺陷语句本身的可疑度 $r^{s^f}=R(a_{ef}(s^f),a_{ep}(s^f))$；

步骤 2：根据 **SPICA 定义2** 列出公式曲线的表达通式，即 $r^{s^f}=R(x,y)$；

步骤 3：绘出该曲线。

在实际绘制时，可以考虑将曲线的表达式转化成 $y=\Phi_x(x)$ 或 $x=\Phi_y(y)$ 的形式，若得到的是标准的初等函数，则可以直接手动绘出；反之，如果解析式 $r^f=R(x,y)$ 比较复杂，则可以通过数学软件（如 Matlab 的 plotz 函数）绘出。

需指出 **步骤 1** 中关于 $a_{ef}(s^f)$ 与 $a_{ep}(s^f))$ 的假设是泛化的，并没有为其赋予具体的值。在实际分析时可以视情况去掉该假设，此时我们会假设不同位置的缺陷语句并画出一个曲线族。然而，本章为了分析的简洁，在绘制公式曲线时均假设 $(a_{ef}(s^f),a_{ep}(s^f)$ 已知。

6.2.3　SPICA 框架概述

整个 SPICA 框架（如图 6.3 所示）可以从核心概念、核心思想以及应用三个层面来诠释。

首先，SPICA 的核心概念包括程序谱图与公式曲线。程序谱图是整个框架的容器，

而可疑度公式曲线则是通过该框架对 SBFL 相关理论进行分析时的关键概念。其次，SPCIA 的核心思想是可视化。通过可视化，可以将目标程序、测试用例集、程序谱、可疑度计算公式以及 SBFL 输出结果统一并形象地刻画出来。最后一个层次是 SPICA 的用途，通过 SPICA，我们可以对程序谱空间进行分析，进而对程序谱图与公式曲线的几何意义进行探索。在此基础上，分析 SBFL 的执行过程以及可疑度公式的性能及缺陷推断机理。此外，SPICA 的用途还包括对 SBFL 基本理论的探索以及对这些理论的实验验证方法的设计。在自适应软件缺陷定位这个主题下，通过 SPICA，我们还可以对 SBFL 的适应性进行分析，并且提出改善方法。

SPICA 具有整体性、统一性以及可表达性的三个特点。首先，程序谱以及可疑度计算公式在不同程序中的性能都可以被表示为不同的点集与曲线，而 SBFL 的执行机理就能够通过点的分布以及曲线的特征趋势被整体地表现出来。此外，SPICA 将不同架构的程序及软件缺陷转化为程序谱图上不同的点集分布，而不同可疑度计算公式中包含的理念以及启发式思想也都能用不同公式曲线的几何特征来刻画。如此，所有可疑度计算公式都可以放在统一的容器内进行分析。最后，在该框架下，与 SBFL 相关的各种基本理论都可以通过描述曲线趋势、谱分布规律以及二者之间关系的几何语言来表述，这使得框架本身具备较强的可表达性。

除了上述三个特点，SPICA 还具有非常好的可扩展性。首先，除了程序谱图以及可疑度公式曲线这两个核心概念，语句成像点等衍生概念都会在基于该框架的研究中被涉及到。而后续研究中，可以根据需要引入更多的概念及工具（如公式曲线族等）。此外，SPICA 的分析方法也是开放的。除去本章中涉及 SBFL 性能的分析外，诸如缺陷定位稳定性[138] 与鲁棒性[47] 这些 SBFL 的相关性质也都可以借助 SPICA 进行进一步分析。最后，SPICA 得出的任何基本理论假设都可以作为启发式而应用到实际缺陷定位流程及方法设计中。本章主要对 SPICA 进行最基本的搭建，并在此基础上进行一些基础性研究。对于核心概念，我们主要考虑程序谱图、公式曲线及一些必要的衍生物（如图上的点、语句成像等）。对于 SPICA 的用途，本章在该框架下对 SBFL 的执行机理进行一些基础性分析（例如程序谱分布、可疑计算度公式的推理机制及其合理性与适应性等），并在此基础上设计 SBFL 适应性提升算法。

6.3 基于 SPICA 的 SBFL 可疑度公式机理分析

本章对程序谱图与公式曲线的基本物理意义，以及其几何性质进行讨论，并进一步对 SBFL 可疑度计算公式的推理机制进行描述。

表 6.1　本章在 SPICA 框架中分析的公式

公式名称		Standard Formula Expression
OP(ER1)		$a_{ef} - a_{ep}/(P+1)$
ER2	Jaccard [139] Anderberg [75] Sϕrensen-Dice [75] Dice [75] Goodman [75]	$\frac{a_{ef}}{a_{ep}+F}$
ER3	Tarantula [72] q_e [140] CBI Inc. [78]	$\frac{a_{ef}/F}{a_{ef}/F+a_{ep}/P}$
ER4	Wong2 [141] Hamann [75] Simple Matching [75] Sokal [75] Rogers & Tanimoto [75] Hamming etc. [75] Euclid [75]	$a_{ef} - a_{ep}$
ER5	Wong1 [141] Russel&Rao [75]	a_{ef}
ER6	Scott [75] Rogot1 [75]	$\frac{1}{2}\frac{a_{ef}/(a_{ef}+a_{ep}+F)}{a_{np}/(a_{nf}+a_{np}+P)}$
Kulczynski2 [75]		$\frac{1}{2}(\frac{a_{ef}}{F} + \frac{a_{ef}}{a_{ef}+a_{ep}})$
Ochiai [76]		$\frac{a_{ef}}{\sqrt{F(a_{ef}+a_{ep})}}$
M2 [75]		$\frac{a_{ef}}{2F+P-a_{ef}+a_{ep}}$
AMPLE2 [75]		$a_{ef}/F - a_{ep}/P$
Wong3 [141]		$a_{ef} - h$, where $h = \begin{cases} a_{ef} & \text{if} a_{ep} \leq 2 \\ 2 + 0.1(a_{ef} - 2) & \text{if} 2 < a_{ep} \leq 10 \\ 2.8 + 0.001(a_{ef} - 10) & \text{if} a_{ep} > 10 \end{cases}$
Arithmetic Mean [75]		$\frac{2Pa_{ef}-2Fa_{ep}}{-(a_{ef}+a_{ep})^2+(P+F)(a_{ef}+a_{ep})+FP}$
Cohen [75]		$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$
Fleiss [75]		$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$

讨论泛化的程序 PG，设 PG 的语句集为 S，其中包含缺陷语句 s^f。设已执行的

测试用例集为 T；对于任意测试用例 $t \in T$ 其执行信息与结果的正确性已知。则根据 PG 与 T，我们能够得到程序谱图 $G(PG,T)$。若采用可疑度计算公式 R 进行缺陷定位，则根据 **SPICA 定义 2**，得到公式曲线 $R(x,y) = R(a_{ef}(s^f), a_{ep}(s^f))$，现对公式曲线的基本性质进行分析。

首先从定义式可以得到公式曲线的最基本的几何特征，即

SPICA 基本性质 1：任何可疑度公式曲线必经过 s^f 在程序谱图上的成像点 $po(s^f)$。

现讨论 SBFL 的两个基本原理，即

SBFL 基本原理 1： 在失效测试用例中执行频率高的语句更有可能是缺陷语句；

SBFL 基本原理 2： 在成功测试用例中执行频率低的语句更有可能是缺陷语句。

尽管实际情况中这两个基本原理可能存在冲突，但我们说可疑度公式 R 符合或者是不违背这两个基本原理，就是说当这两个基本原理不存在冲突时，各语句可疑度值之间的比较关系要与这两个基本原理的描述相符。具体来说，有

$$\forall s_1, s_2 \in S, (a_{ep}(s_1) \leq a_{ep}(s_2)) \wedge (a_{ef}(s_1) \geq a_{ef}(s_2)) \rightarrow R(a_{ef}(s_1), a_{ep}(s_1)) \geq R(a_{ef}(s_2), a_{ep}(s_2)) \tag{6.2}$$

从另一个角度看式 (6.2)，其正是对公式曲线在程序谱图内增减性的描述。基于此，我们给出一个合理的可疑度计算公式 R 所需要具备的基本特征。

SPICA 基本性质 2 (MC **基本性质**)：若将可疑度公式曲线看成是在程序谱图坐标系中关于 x 的函数，则其单调递增。

SPICA 基本性质 3：若将可疑度公式曲线看成是在程序谱图坐标系中关于 y 的函数，则其单调递减。

这里的单调性并不需要是严格的。因此，形如 $x = k$ 这种与坐标轴垂直的直线，也被视作满足 **SPICA 基本性质2**，但这种公式曲线并不属于普通公式曲线。对于不连续的公式曲线以及非普通曲线公式本章不做讨论。

总结来说，**SPICA 基本性质1** 由公式曲线的定义式得到，任何可疑度计算公式的公式曲线都满足该性质。而 **SPICA 基本性质2** 与 **SPICA 基本性质3** 则基于 SBFL 的两个基本原理。我们认为符合这两个基本性质的可疑度计算公式是合理的，反之则是不合理的。

由公式曲线的增减性可知，其对程序谱图 $G(PG,T)$ 的划分中，可分辨区域 G^i 会

在公式曲线的左上部分，而错分辨区域 G^e 会在公式曲线的右下部分。而在程序谱图中，左上方意味着更小的 a_{ef} 值与更大的 a_{ep} 值，而右下方则意味着更大的 a_{ef} 值与更小的 a_{ep} 值。因此，公式曲线划分的可识别区域与 SBFL 的基本原理是吻合的。而不同可疑度公式的差异则能够从 G^i 与 G^e 更为细节的几何特征中反映出来。本章的后续内容将会对这些细节特征做进一步分析。

6.4　单缺陷假设下可疑度公式性能的可视化分析

尽管现实中的待测软件通常是多缺陷的，单缺陷假设下各可疑度计算公式的性能同样是衡量该公式质量的一个重要指标。一方面，当调试人员去关注一个缺陷时，其很可能会忽略由其他缺陷触发的失效。另一方面，一个可疑度计算公式在单缺陷假设下的表现是评价其优劣的基础。如果该可疑度计算公式在单缺陷假设下的准确度都无法保证，那么其本身就不是合理的。

如第6.1 节所述，Xie 等人[2] 在集合论框架下对 30 个经典的可疑度公式进行了比较与证明。然而他们的证明过程需要比较复杂的公式推导，并且缺少关于单缺陷假设下可疑度公式性能的一般性结论（如相关的性质、引理等）。因此，仅仅基于集合论的比较框架不易给调试人员带来关于 SBFL 性能的直观感受，从而为其所接受。此外，如果新的可疑度公式被设计出来，我们同样不易对其进行新一轮的比较与证明。

为解决此问题，本节试图通过 SPICA 对单缺陷假设下的可疑度公式性能进行比较与证明。借助 SPICA，我们在证明过程中尽量避免了复杂的理论推导。与之相对地，我们通过程序谱图直观地将单缺陷下 SBFL 的基本理论刻画出来，并且以此为依据对不同可疑度计算公式间的性能进行更为直观的比较。比起证明结论，SPICA 更强调对各可疑度公式在单缺陷假设下性能的描述。基于 SPICA，调试人员不仅能够了解各 SBFL 可疑度公式在单缺陷假设下的性能比较结果，还能够了解比较结果中蕴含的各可疑度计算公式间执行机理的差异。此外，当对新的可疑度计算公式进行考查时，SPICA 还提供了更为简单的定性比较方法。

6.4.1　SPICA 在单缺陷假设下的基本理论

假设程序 PG 只包含一条缺陷语句 s^f，则测试过程中任何失效测试用例必是由该缺陷语句导致的。若失效测试用例总数为 F，则有 $a_{ef}(s^f) = F$。将其转化至程序谱图中，则如图6.4 中红色方点所示。进而我们可以得到 SPICA 框架在单缺陷假设下的基本理论。

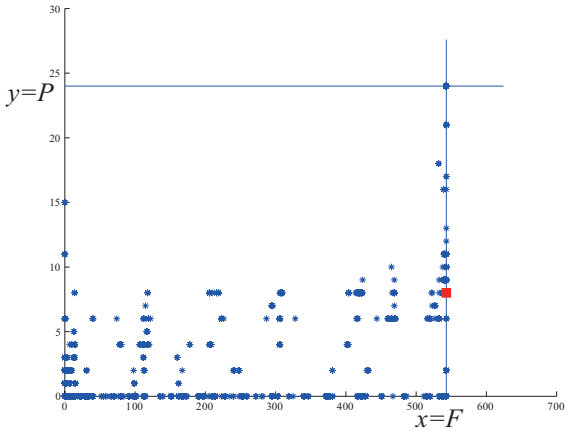


图 6.4　单缺陷程序谱图实例

SPICA 定理 1 (**单缺陷缺** SP **定理**)：若目标程序 PG 只包含一条缺陷语句 s^f，则对于任意 T，其在程序谱图 $G^{PG,T}$ 中的成像位置一定在直线 $x = F$ 上，即 $po(s^f) \in x = F$。

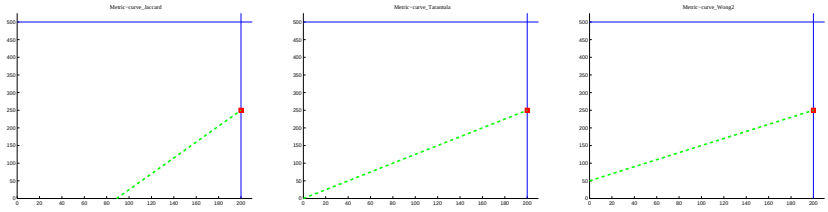
根据 Xie 等人[2] 的理论，对于可疑度计算公式 R，定义 S_R^A 为可疑度小于缺陷语句 s^f 的语句集合，S_R^B 为可疑度大于缺陷语句 s^f 的语句集合，而 S_R^F 为可疑度等于缺陷语句 s^f 的语句集合。若一个可疑度计算公式 R_1 比另一个公式可疑度计算公式 R_2 好，则有 $S_{R_2}^A \subseteq S_{R_1}^A$ 且 $S_{R_1}^B \subseteq S_{R_2}^B$ 。而将其转化到程序谱图上，则可以得到如下定理：

SPICA 定理 2 (**公式比较定理**)：若可疑度计算公式 R_1 绝对优于可疑度计算公式 R_2，则对于任意 PG 与 T 形成的程序谱图 $G(PG,T)$，R_1 的公式曲线划分出来的区域 $G_{R_1}^i$ 与 $G_{R_1}^e$，以及 R_2 的公式曲线划分出来的区域 $G_{R_2}^i$ 与 $G_{R_2}^e$，需满足关系

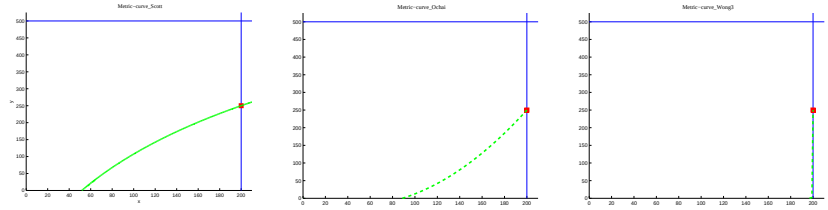
$$G_{R_1}^e \subseteq G_{R_2}^e \wedge G_{R_2}^i \subseteq G_{R_1}^i \tag{6.3}$$

也就是说，R_1 的公式曲线 MC^{R_1} 整体在 R_2 的公式曲线下方（不包含缺陷语句的成像点 $po(s^f)$）。

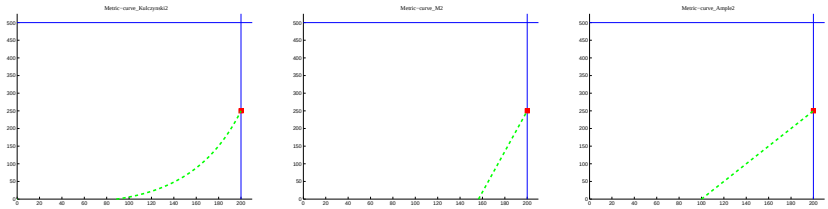
SPICA 定理2 与文献 [2] 的理论是等价的，然而 **SPICA 定理2** 将这一理论以程序谱图中可疑度公式曲线的几何特征来呈现；相比之下，这种表述方式更为直观。并且，通过这种描述，可以从几何的角度引出更多的定理，得到更为精细的证明框架以及更为丰富的结论。这里，我们仍用 $R_1 \to R_2$ 表示可疑度公式 R_1 优于可疑度公式 R_2。



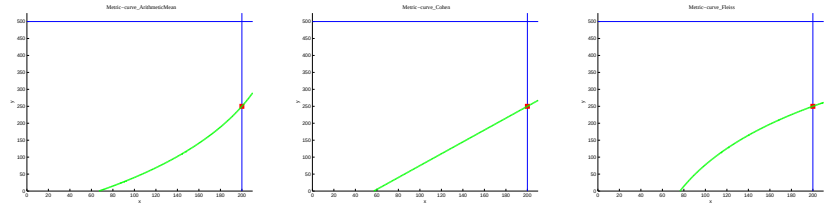
(a) ER2　(b) ER3　(c) ER4



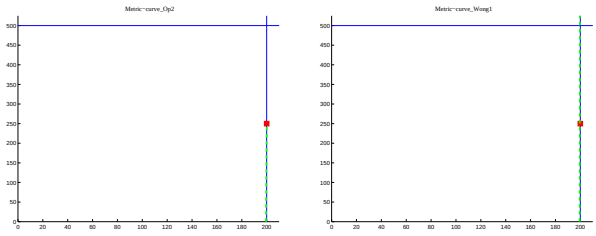
(d) ER6　(e) Ochiai　(f) Wong3



(g) Kulczynski2　(h) M2　(i) Ample2



(j) ArithmeticMean　(k) Cohen　(l) Fleiss



(m) Op　(n) Wong1

图 6.5　单缺陷假设下各可疑度公式的 MC 实例

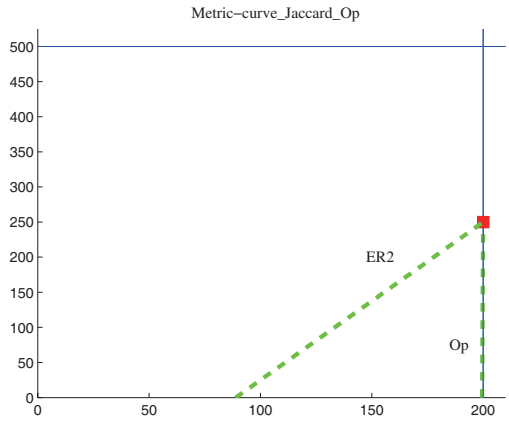


图 6.6　线性 MC 公式 OP 与 ER2 在单缺陷假设下的比较

6.4.2　可疑度公式性能比较

首先，表6.1 列出了本章将要在 SPICA 框架中进行分析的公式。这些可疑度计算公式的公式曲线实例如图6.5 所示。可以看出，不同的可疑度计算公式的公式曲线各不相同，既有线性曲线（即直线），也有非线性曲线。而非线性曲线又可以分为凸曲线和凹曲线两大类。其中每个种类所包含的可疑度计算公式如下：

- 直线公式：ER2、ER3、ER4、M2、Ample2、Cohen；
- 凸曲线公式：ER6、Fleiss；
- 凹曲线公式：ArithmeticMean、Ochiai、Kulczynski2。

6.4.2.1　直线公式之间的比较

显然，具有线性公式曲线的可疑度计算公式是最基本也是最简单的。因此，我们首先讨论两个线性公式之间的优劣关系。由 **SPICA 定理1** 可知，缺陷语句在程序谱图中的位置一定在整个图的右边界线（即 $x = F$）上。因此，我们很容易得到以下定理：

SPICA 定理 3：若公式 R_1 与公式 R_2 都是线性公式，且对于程序谱图 $G(PG,T)$，有 $k^{MC^{R_1}} > k^{MC^{R_2}}$ 以及 $k^{MC^{R_1}}, k^{MC^{R_2}} < +\infty$，则有 $R_1 \rightarrow R_2$，这里 $k^{MC^{R_1}}, k^{MC^{R_2}}$ 分别为 MC^{R_1} 与 MC^{R_1} 的斜率。

在单缺陷假设下，两个线性公式 OP 与 ER2 的比较实例如图6.6 所示。图中描绘了 R_{Op} 的公式曲线 $MC^{R_{Op}}$ 以及 R_{ER2} 的公式曲线 $MC^{R_{ER2}}$ 在单缺陷假设

下的几何特征。可以看出这两条公式曲线都是直线，并且不难求得 $MC^{R_{Op}}$ 的斜率 $k^{MC^{R_{Op}}}$ 为 $P+1$，而 $MC^{R_{ER2}}$ 的斜率 $k^{MC^{R_{ER2}}}$ 为 $(a_{ep}(s^f)+F)/a_{ef}(s^f)$。易证 $k^{MC^{R_{ER2}}}=1+(a_{ep}(s^f)/F<1+P=k^{MC^{R_{Op}}}$。此外，由 SP 单缺陷定理（即 **SPICA 定理1**）可知，单缺陷下的错分区域 G^e 为公式曲线、x 轴以及直线 $x=F$ 所围成的三角形区域。显然由于 $k^{MC^{R_{ER2}}}<k^{MC^{R_{Op}}}$，由 $MC^{R_{Op}}$ 围成的错分区域 G^e_{Op} 包含在由 $MC^{R_{ER2}}$ 围成的错分区域 G^e_{ER2} 中。因此，我们有 $R_{Op}\rightarrow R_{ER2}$。

依据 **SPICA 定理3** 我们可以得到以下公式比较关系：$R_{Op}\rightarrow R_{ER3}$、$R_{Op}\rightarrow R_{ER4}$、$R_{Op}\rightarrow R_{Cohen}$、$R_{Op}\rightarrow R_{M2}$、$R_{Op}\rightarrow R_{Ample2}$、$R_{Op}\rightarrow R_{Wong3}$、$R_{Op}\rightarrow R_{Cohen}$、$R_{ER2}\rightarrow R_{ER3}$、$R_{ER2}\rightarrow R_{ER4}$、$R_{M2}\rightarrow R_{AMPLE2}$、$R_{M2}\rightarrow R_{ER3}$ 以及 $R_{M2}\rightarrow R_{ER4}$。

6.4.2.2　直线公式与非线性公式之间的比较

进一步考虑直线公式曲线与非线性公式曲线之间的关系。这里，我们假设公式曲线连续且左右导数存在（即普通公式曲线，这种假设被本章讨论的所有非线性公式满足）。由于不存在一个固定的斜率，因此对于非线性公式曲线的比较就要复杂一些。为此，我们进一步对公式曲线所表现出来的更多几何性质（包括导数以及凹凸性）进行讨论。首先，一个连续可导函数在某点的导数等于其在该点切线的斜率，由此我们很容易得到以下定理：

SPICA 定理 4：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，斜率为 $k^{MC^{R_1}}$，而 R_2 的公式曲线 MC^{R_2} 是曲线。若 MC^{R_2} 在定义域内任意一点的导数都小于 $k^{MC^{R_1}}$，则有 $R_1\rightarrow R_2$。

尽管 **SPICA 定理4** 的结论比较直观，但在实际应用时，我们必须先求出 MC^{R_2} 的导函数并证明该导函数在程序谱图中恒大于（或小于）$k^{MC^{R_1}}$。这就需要较大的计算量与较为繁琐的数学推导。而借助 SPICA，则可以根据公式曲线的几何特征给出更多的定理，来辅助我们对公式机理的解读以及公式性能的比较。

首先，在同一程序谱图中，一线性公式曲线与一凹凸性一致的曲线公式曲线最多只能有两个交点，并且其中一个交点为 $po(s^f)$。因此，对于纯凸曲线或者纯凹曲线，我们可以得到以下定理：

SPICA 定理 5：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，斜率是 $k^{MC^{R_1}}$，而 R_2 的公式曲线 MC^{R_2} 是凹曲线；若 MC^{R_2} 与 x 轴交点处导数大于 $k^{MC^{R_1}}$，则有 $R_2\rightarrow R_1$。

SPICA 定理 6：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，斜率是 $k^{MC^{R_1}}$，而 R_2 的公式曲线 MC^{R_2} 是凹曲线；若 MC^{R_2} 与直线 $x = F$ 交点处导数小于 $k^{MC^{R_1}}$，则有 $R_1 \to R_2$。

SPICA 定理 7：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，其与 x 轴交于 $(x_1, 0)$，而 R_2 的公式曲线 MC^{R_2} 是凹曲线，其与 x 轴交于 $(x_2, 0)$；若 $x_1 \leq x_2$，则有 $R_2 \to R_1$。

SPICA 定理 8：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，斜率是 $k^{MC^{R_1}}$，而 R_2 的公式曲线 MC^{R_2} 是凸曲线；若 MC^{R_2} 在其与直线 $x = F$ 轴交点处（即缺陷语句成像点）的导数大于 $k^{MC^{R_1}}$，则有 $R_2 \to R_1$。

SPICA 定理 9：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，斜率是 $k^{MC^{R_1}}$，而 R_2 的公式曲线 MC^{R_2} 是凸曲线；若 MC^{R_2} 与 x 轴交点处导数小于 $k^{MC^{R_1}}$，则有 $R_1 \to R_2$。

SPICA 定理 10：考虑可疑度计算公式 R_1 与可疑度计算公式 R_2，其中 R_1 的公式曲线 MC^{R_1} 是直线，其与 x 轴交于 $(x_1, 0)$；而 R_2 的公式曲线 MC^{R_2} 是凸曲线，其与 x 轴交于 $(x_2, 0)$；若 $x_1 \geq x_2$，则 $R_1 \to R_2$。

利用 **SPICA 定理5**–**SPICA 定理10**，我们很容易可以证明 $R_{Ochiai} \to R_{ER2}$、$R_{Kullczynski} \to R_{ER2}$、$R_{M2} \to R_{Ochiai}$、$R_{Op} \to R_{Ochiai}$、$R_{Op} \to R_{ArithmeticMean}$、$R_{Op} \to R_{Fleiss}$、$R_{Op} \to R_{Kulczynski2}$ 以及 $R_{Op} \to R_{ER6}$。

6.4.2.3 两个非线性曲线公式之间的比较

当两个待比较的可疑度计算公式均具有非线性公式曲线时，最坏情况下我们只能根据 **SPICA 定理1**分别计算两个曲线的导函数，或是直接计算两个曲线表达式的差来对其性能进行比较。然而，更加深入地分析公式曲线的几何特征，我们仍能得到以下定理：

SPICA 定理 11：讨论公式 R_1 与公式 R_2，其中 R_1 的公式曲线为 MC^{R_1}，R_2 的公式曲线为 MC^{R_2}。设 MC^{R_1} 与 x 轴交点为 $(x_1, 0)$，而 MC^{R_2} 与的 x 轴交点为 $(x_2, 0)$，则若 MC^{R_1} 与 MC^{R_2} 的二阶导数就分别为常数 k_1' 与 k_2'，且有 1）$k_1' < k_2'$，2）$x_1 \leq x_2$，则 $R_2 \to R_1$。

根据 **SPICA 定理11**我们可以证明 $R_{Kulczynski2} \to R_{Ochiai}$。

6.4.3 SPICA 下对 Maximal 公式的讨论

在 Xie 等人的理论中，在单缺陷假设下对于 Maximal 公式的定义如下：

定义 6.1 (Xie **等人对于** Maximal **公式的定义**)：*考虑可疑度公式 R，若对于任意 $R' \to R$ 都能推出 $R \leftrightarrow R'$，则称可疑度计算公式 R 为* Maximal *公式。*

本小节试图将 Maximal 公式的概念表示在程序谱图中。对于任意可疑度计算公式 R，其公式曲线为 $R(x,y) = R(a_{ef}(s^f), a_{ep}(s^f))$。由于在单缺陷假设下有 $a_{ef}(s^f) = F$，则 R 的公式曲线就可以写成 $R(x,y) = R(F, a_{ep}(s^f))$。现设计一个可疑度计算公式 $R'(a_{ef}, a_{ep}) = R(a_{ef}, a_{ep}) + ka_{ef}$，则 R' 的公式曲线为 $R(x,y)+kx = R(F, a_{ep}(s^f))+kF$，则易证 $R' \to R$。根据 **SPICA 定理2**，$MC^{R'}$ 会在 MC^R 的右侧。

现考虑两者的错分区域为 G^e_R 和 $G^e_{R'}$，则有

$$G^e_{R-R'} = (G^e_R \cup MC^R) - (G^e_{R'} \cup MC^{R'}) \tag{6.4}$$

这里 $G^e_{R-R'}$ 表示包含在 R 的错分区域中且不包含在 R' 的错分区域中的区域，其在程序谱图上就表现为由 MC^R、$MC^{R'}$ 以及坐标轴围成的区域（如果两公式曲线都交于同一坐标轴，则该区域为三角形，否则为四边形）。在此情况下，若 R 是 Maximal 公式，则有 $R \leftrightarrow R'$，那么就必须保证 $G^e_{R-R'}$ 在任何情况下都不包含任何语句的成像点。由于在软件缺陷定位中，各语句的程序谱统计信息，即 a_{ef} 与 a_{ep}，的值必须是整数。因此对于连续单调递增的公式曲线来说，满足 Maximal 条件就等价于整个公式曲线右侧区域（即 G^e 区域）的跨度必须在程序谱图上小于 1，即我们有以下定理：

SPICA 定理 12：若可疑度计算公式 R 的公式曲线 MC^R 在定义域上连续单调递增，则以下两个命题等价

（1）R 是 Maximal 公式

（2）MC^R 交于 x 轴，且交点 $(x, 0)$ 满足 $F - 1 < x < F$

SPICA 定理12 将 Maximal 这一概念通过程序谱图的方式形象地表现出来。如图6.7 所示，凡是公式曲线在绿色区域内的可疑度计算公式，都是 Maximal 公式。进一步，由**定义 6.1** 对 Maximal 的定义可知，若只考虑普通公式曲线，则有如下定理：

SPICA 定理 13：若普通曲线公式 R 是 Maximal 公式，则对于任意的非 Maximal 普通曲线公式 R'，有 $R \to R'$。

SPICA 定理 14：若普通曲线公式 R_1 与 R_2 都是 Maximal 公式，则有 $R_1 \leftrightarrow R_2$。

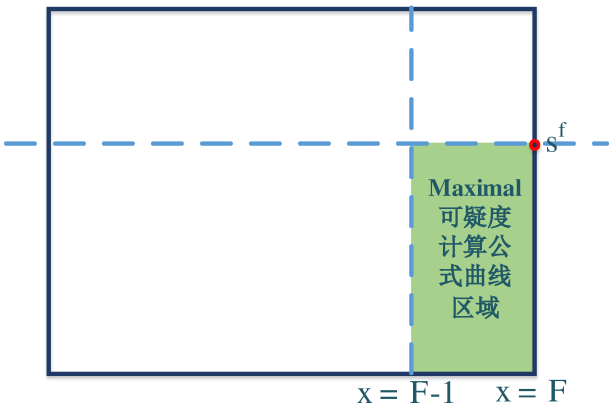


图 6.7　单缺陷下 Maximal 公式曲线所在区域

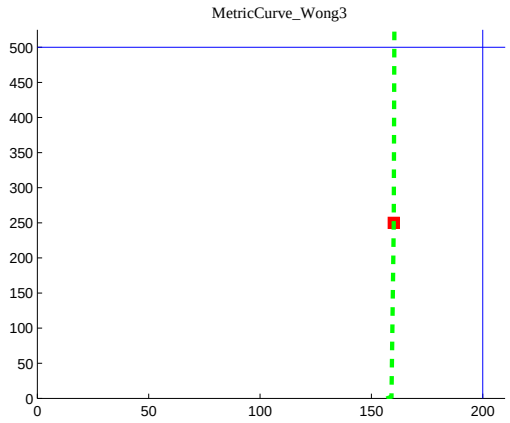
由于单缺陷下普通公式曲线与直线 $x = F$ 相交，且交点正是 s^f 的成像点。如果一个可疑度计算公式是 Maximal 公式，则其公式曲线与 x 轴交点的横坐标值必须大于 $F-1$。则我们可以得到一个判定可疑度计算公式是 Maximal 公式的充分条件。

SPICA 定理 15：可疑度公式 R 是 Maximal 公式的一个充分条件是 R 在各点的导数均大于 $a_{ep}(s^f)$

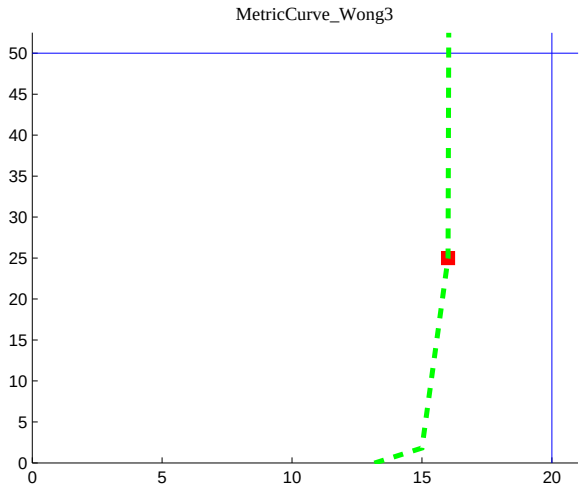
由于 $MC^{R_{Op}}$ 的斜率为 $P+1$，这个值一定大于 $a_{ep}(s^f)$，因此根据 **SPICA 定理15** 可证 Op 是 Maximal 公式。进一步根据 **SPICA 定理13**，我们可以对分段线性公式 Wong3 进行分析。

实际上，R_{Wong3} 的公式曲线是分段线性的，三段的斜率分别是 1、10 和 1000。当 P 很大且 $a_{ep}(s^f)$ 很大时，由于公式曲线的斜率很大（斜率值为 1000），其在程序谱图上与 Maximal 公式类似（见图6.8(a)）。然而，当 $a_{ep}(s^f)$ 较小时，从图6.8(b)可以很清晰地看出 $MC^{R_{Wong3}}$ 是一个过 $(F-2,0)$ 点的曲线，因此其并不是 Maximal 公式。

最后，对于公式 Wong1 来说，其只符合 **SBFL 基本原理 1** 而并不考虑 **SBFL 基本原理 2**。表现在公式曲线上，R_{Wong1} 的公式曲线 $MC^{R_{Wong1}}$ 是一条斜率为 $+\infty$ 的直线。因此 R_{Wong1} 并不是普通曲线公式。此外，R_{Wong1} 的公式曲线并不与直线 $x = F$ 相交，而是与其重合。因此对于任何成像在 $x = F$ 上且 a_{ep} 值小于 $a_{ep}(s^f)$ 的语句，其都会被 $MC^{R_{Wong1}}$ 划分在 G^n 中，而这些语句会被普通公式曲线划分在 G^e 中，也就是说 R_{Wong1} 在对于这些语句的区分上表现要好。相反，对于任何成像在 $x = F$ 上且 a_{ep} 值大于 $a_{ep}(s^f)$ 的语句，其会被普通公式曲线划分在 G^i 中，而 $MC^{R_{Wong1}}$ 同样会



(a) 成例数 P 值较大



(b) 成例数 P 值较小

图 6.8　Wong3 公式的公式曲线示例

将其划分自在 G^n 中，也就是说 R_{Wong1} 在这些语句的区分上表现要差。因此，可以说 R_{Wong1} 也是 Maximal 公式，但其与其它任何普通曲线公式都不存在优劣性的序关系。

总结来说，通过对 Maximal 公式进行讨论和分析，可以得到以下结论：

（1）R_{OP} 是普通曲线公式且是 Maximal 公式，因此其在单缺陷假设下性能优于除 R_{Wong1} 外任何本章所讨论的普通曲线公式；

（2）R_{Wong1} 是 Maximal 公式，但其不是普通曲线公式，与其它普通曲线公式不存在优劣性的序关系。

6.4.4 公式性能之间的条件比较

在 SPICA 中，通过对各可疑度计算公式的公式曲线几何性质进行比较，我们能够得到各可疑度计算公式除数学表达式以外更多的性能特征。我们还可以通过对这些特征进行更加深入的分析，得到更为丰富的结论。这也是 SPICA 将 SBFL 可疑度计算公式进行可视化带来的好处之一。在单缺陷假设下，很多公式之间拥有绝对的优劣关系；而对于很多公式，它们之间的优劣关系不是绝对的，其只有在特定条件下才会显现。具体来说，各公式曲线的几何性质通常和 F 与 P 的具体值相关，而这两个值是在实际缺陷定位运行之前就可以得到的。因此讨论在特定 F 与 P 条件下各公式曲线之间的关系，并进行公式性能间的比较是有意义的。这里我们用 $R_1 \dashleftarrow\dashrightarrow R_2$ 表示可疑度计算公式 R_1 与 R_2 在特定条件下存在优劣关系。对于条件优劣关系的证明需根据特定公式曲线的几何特征进行具体分析。作为示例，本节中我们讨论以下关系：

$R_{Ample2} \dashleftarrow\dashrightarrow R_{ER4}$：

易求得 R_{Ample2} 的公式曲线 $MC^{R_{Ample2}}$ 为一条斜率是 P/F 的直线，而 $ER4$ 的公式曲线 $MC^{R_{ER4}}$ 为一条斜率是 1 的直线。由此可见，当 PG 和 T 满足 $P > F$ 时，我们有 $k^{MC^{R_{Ample2}}} > k^{MC^{R_{ER4}}}$，则根据 **SPICA 定理3**，可以得到 $R_{Ample2} \dashrightarrow R_{ER4}$。而当 $P < F$ 时，$k^{MC^{R_{Ample2}}} < k^{MC^{R_{ER4}}}$，则可证 $R_{Ample2} \dashleftarrow R_{ER4}$。因此，我们得到 $R_{Ample2} \dashleftarrow\dashrightarrow R_{ER4}$。

此外，当比较条件涉及 $a_{ef}(s^f)$ 与 $a_{ep}(s^f)$ 时，我们无法在缺陷定位实际执行之前就确定其序关系。但在带有 $a_{ef}(s^f)$ 与 $a_{ep}(s^f)$ 的条件下进行比较仍能够为预测公式性能提供有力的支持。关于这个方面的研究我们会在今后研究工作中展开。

6.4.5 SPICA 框架下可疑度计算公式的定性分析

在实际应用环境中，对于比较复杂的可疑度计算公式，或是新设计出来的可疑度计算公式，调试人员可能不易对其进行直接推导与证明。而借助 SPICA，我们可以通过绘制公式曲线的方式对其在单缺陷下的性能进行定性分析。下面通过实例对该方法进行介绍。

考虑两个较为复杂的可疑度计算公式 $R_{Kulczynski2}$ 与 R_{Ochiai}，其公式曲线都是非线性的。这里我们通过仿真方法对 $R_{Kulczynski2}$ 与 R_{Ochiai} 的公式曲线进行可视化分析。具体来说，我们可以对被测软件的实际情况进行假设，并在假设情况下对公式的性能进行考察。这里我们考察 $F > P$、$F < P$、$F > 100$ 与 $F < 100$ 等想定，具体如表6.2

表 6.2　假设的代表性场景

$F=200$ 且 $P=500$	$F=500$ 且 $P=200$	$F=20$ 且 $P=50$	$F=50$ 且 $P=20$
$a_{ef}^{sf}=F$ 且 $a_{ep}^{sf}=P/4$	$a_{ef}^{sf}=F$ 且 $a_{ep}^{sf}=P/2$	$a_{ef}^{sf}=F$ 且 $a_{ep}^{sf}=3P/4$	

所示。在此基础上，绘制每个假设下 $R_{Kulczynski2}$ 与 R_{Ochiai} 的公式曲线，如图6.9所示。从图中可以看出，在所有假设场景下，$MC^{R^{Ochiai}}$ 与 $MC^{R^{Kulczynski2}}$ 都是凹曲线。此外，我们还能发现 $MC^{R^{Ochiai}}$ 与 $MC^{R^{Kulczynski2}}$ 交 x 轴于同一点，且 $MC^{R^{Kulczynski2}}$ 的二阶导数要大于 $MC^{R^{Ochiai}}$。则根据 **SPICA 定理5**，我们可以得到 $R^{Kulczynski2} \rightarrow R^{Ochiai}$。尽管，我们只是在假设场景下进行仿真，然而 $MC^{R^{Ochiai}}$ 与 $MC^{R^{Kulczynski2}}$ 的几何性质会在这些场景中被综合地表现出来，进而其在单缺陷假设下的基本性能也能被直观地观测到。而事实上，$MC^{R^{Ochiai}}$ 与 $MC^{R^{Kulczynski2}}$ 确实交 x 轴于同一点，并且有 $R^{Kulczynski2} \rightarrow R^{Ochiai}$。

6.4.6　比较结果总结

总的来说，SPICA 通过分析公式曲线的几何特征来对 SBFL 可疑度计算公式的性能进行比较。相比于传统的集合论方法，SPICA 从一种几何的视点引入一系列基于几何关系的定理来进行观测与证明。这些定理能够直观地表现在程序谱图上，并且具备相应的几何意义。因此，在单缺陷假设下，SPICA 不仅仅是证明的工具，也是对各公式之间性能关系的一种可视化描述。

图6.10列出了使用 SPICA 对单缺陷假设下各 SBFL 可疑度计算公式进行比较的结果。其中，图6.10(a) 是 Xie 等人基于集合论的比较结果，而图6.10(b) 列出了本章基于 SPICA 框架的比较结果。可以看出，我们在 Xie 等人的工作基础上进一步得到了 $R_{Ample2} \rightarrow R_{ER3}$、$R_{M2} \rightarrow R_{Ochiai}$、以及条件比较结果 $R_{Ample2} \dashleftarrow\dashrightarrow R_{ER4}$。这些结果的证明用到了基于几何特征关系的定理，相比于更为复杂的集合推导，其更为简单与直观。此外，由于几何特征包含更多的物理含义，因此相比于集合论方法，我们也得到了更为丰富的结论，这些结论仍有较大的扩展空间。

此外，借助 SPICA 框架，本节设计了单缺陷假设下可疑度公式性能的可视化定性分析方法。该方法在多个假设场景下对公式曲线进行仿真，直观地将各可疑度计算公式的性能及它们之间优劣关系刻画出来。该方法使公式间的比较更为简单直观，并且能够被自动化执行；其能够在实际缺陷定位环境下更为有效地进行公式性能的比较、公式的选择以及 Maximal 的满足性检验。反过来，通过仿真能够形象化地体现各公式曲线的几何特性，进而为 SBFL 可疑度计算公式的进一步分析提供思路。

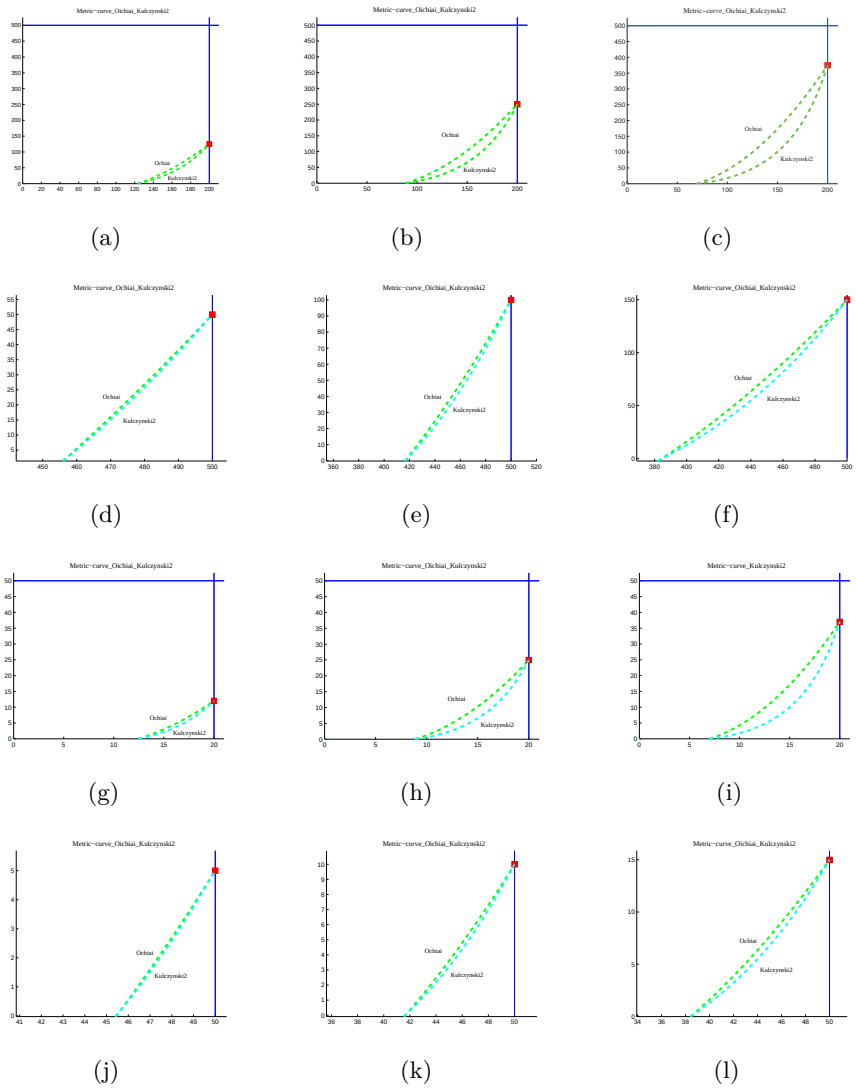
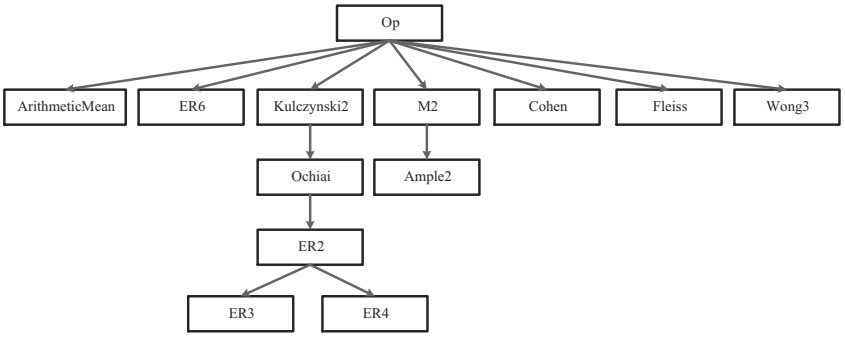
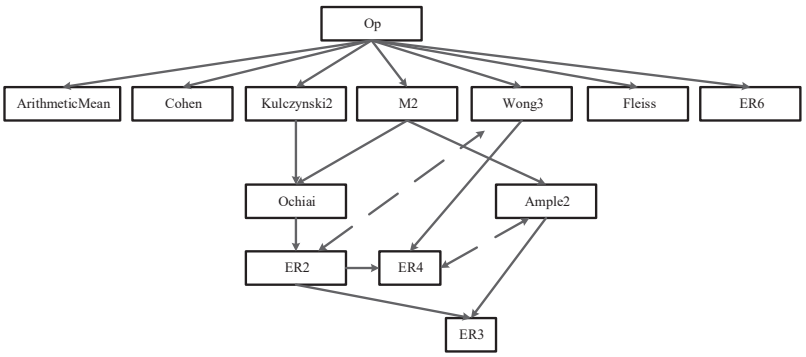


图 6.9　典型场景下 Ochiai 与 Kulczynski2 公式的公式曲线比较

最后，我们通过对 Maximal 公式的进一步分析，更为深入地在几何框架下解释了“Maximal”的含义，并得到了 **SPICA 定理13**– **SPICA 定义15**。，而这些理论对于 SBFL 公式的研究与应用有着很大的指导意义。首先，由 **SPICA 定理13** 可知，任何 Maximal 的普通曲线公式都好于其它普通曲线公式，这就说明了在单缺陷假设下 Maximal 公式不仅是“非劣”的，其在某种程度上也是“最优”的。然而，考虑 **SPICA 定理14**，所有 Maximal 的普通曲线公式在单缺陷假设下是等价的，这说明我们应该将“Maximal”作为考察一个公式的指标之一，而不是将设计出 Maximal 公式作为目标。具体来说，在可疑度计算公式的设计过程中，我们应着眼于设计合理的语句鉴别机制，使其能够对程序谱空间进行合理地划分，而不是仅仅以发现新的 Maximal 公式为目的。



(a) Xie 等人通过集合理论证明的结果[2]



(b) 基于 SPICA 的比较结果

图 6.10　单缺陷假设下各 SBFL 可疑度计算公式性能比较结果

6.5　普适条件下程序谱分布与可疑度计算公式适应性分析

尽管基于单缺陷假设的分析很重要，在实际软件工程过程中，大多数被测软件是包含多个缺陷的。目前很多错误聚类技术 [34, 142] 都试图将同一缺陷导致的失效测试用例聚于一类，并且在定位某一缺陷时滤除掉由其它缺陷引起的失效测试用例。然而，由于实际软件开发环境的复杂性，满足单缺陷假设的缺陷定位环境仍然无法保证。而这正是目前基于单缺陷假设下的 SBFL 公式性能分析研究所面临的最大挑战，也是整个关于 SBFL 的研究所面临的巨大挑战。单缺陷假设实际上是对程序谱做了一个比较极端的想定，即 $a_{ef}(s^f) = F$。而在此想定下得到的 Maximal 公式也是对可疑度计算公式中 a_{ef} 参数的权重进行最大化的结果，这种最大化在程序谱图中就表现为非常“陡峭”的公式曲线（例如最小导数大于 $P+1$ 的曲线）。然而通常情况下，较为极端的设置很可能造成抗干扰能力的下降。也就是说，当单缺陷假设不再被满足时，这些公式的性能很可能会受到很大影响。

事实上，第6.4 节中的理论及定理都是基于单缺陷假设建立的，而多缺陷场景下的

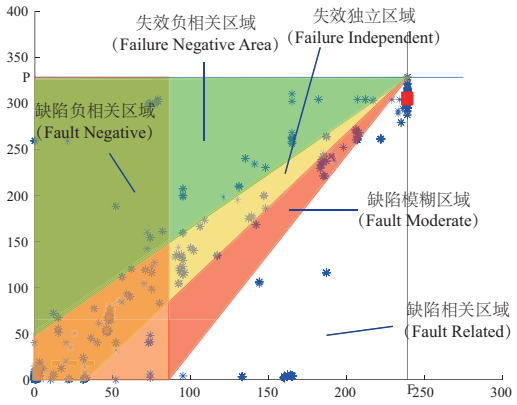


图 6.11　程序谱图中对于程序谱空间的划分

潜在风险为 SBFL 的性能评估、公式选择及实际应用都带来巨大挑战。在去除单缺陷假设后，各公式的性能，以及各公式之间的比较关系都需要进一步讨论。我们需要在去除该假设后的谱适场景下观察各可疑度公式的执行机理，并对其合理性进行分析与判别。本节在 SPICA 下对程序谱空间进行划分，并在此基础上对普适性的 SBFL 理论以及各可疑度计算公式的适应性及合理性进行探究。进而我们提出了一种提升可疑度计算公式适应性的算法。

6.5.1　程序谱空间划分

从第6.2.3节中可知，程序谱图实际上是由以 a_{ef} 为物理意义的横坐标轴以及以 a_{ep} 为物理意义的纵坐标轴形成的特征空间，其实质是程序谱空间的平面投影。通过对该空间的特征及理论进行分析与探究，能够加深我们对程序谱空间本身的了解。

考虑语句 s，其在程序谱图上的成像坐标是 $(a_{ef}(s), a_{ep}(s))$。可见，$a_{ef}(s)$ 的值越大，s 在程序谱图中的成像就越靠右侧，而 $a_{ep}(s)$ 的值越小，则 s 在程序谱图中的成像就越靠下侧。根据 SBFL 的基本原理，a_{ef} 值越大且 a_{ep} 值越小的语句越容易包含缺陷，而其在程序谱图上的表述就是越靠近右下的区域就越可能包含缺陷语句的成像点 $po(s^f)$。

因此，可以根据成像在不同位置的语句包含缺陷的可能性，将程序谱图分为若干个区域，具体的划分情况如图6.11 所示。图6.11是 *flex* 缺陷程序实例的程序谱图。这里将所划分的区域用不同的颜色标注出来。我们将绿色、黄色与红色所标注出来的区域分别称为失效负相关区域、失效独立区域以及缺陷模糊区域，而将白色区称为缺陷相关区域。此外，我们还将左边的矩形区域称为缺陷负相关区域。这种划分并不是一

种绝对的定论，而是一种对程序谱空间的认知理论。该划分是 SPICA 理论之一，可以用来解释实际程序的程序谱及缺陷谱分布规律。由于这种划分理论需要在今后的研究中通过大量实验数据进行进一步验证，因此我们称其为 SPICA 理论假设，其基本划分思想如下所述。

首先，对于一条语句 s，可以将其统计量 $a_{ef}(s)$ 与 $a_{ep}(s)$ 写成

$$\begin{aligned} a_{ef}(s) &\approx F * \rho_{ef}(s) \\ a_{ep}(s) &\approx P * \rho_{ep}(s) \end{aligned} \tag{6.5}$$

其中 $\rho_{ef}(s)$ 与 $\rho_{ef}(s)$ 分别表示语句 s 在失效与成功测试用例中被执行的概率。现假设语句 s 的执行与否与程序是否失效无关（这类语句很可能会出现在松耦合程序模块中），则有 $\rho_{ef}(s) = \rho_{ep}(s) = \rho_e(s)$。这里 ρ_e 是语句 s 在整个测试用例集 T 中的被执行率。因此有

$$\frac{a_{ef}(s)}{a_{ep}(s)} = \frac{F}{P} \tag{6.6}$$

这也就是说，语句 s 在程序谱图中的位置 $po(s)$ 会依概率收敛于对角线上。显然，对于成像在程序谱图中从原点到 (F, P) 点的对角线以下部分的语句，其在失效测试用例中的执行概率要高于成功测试用例中的执行概率，因此这些语句与测试过程中软件的失效具有更强的相关性。相反，对于成像在对角线以上区域的语句，其在成功测试用例中的执行概率要高于在失效测试用例中的执行概率，因此这类语句与软件的失效呈负相关趋势。

此外，有研究指出，尽管多缺陷的存在是普遍的，但缺陷之间的干涉对缺陷语句程序谱特征的影响并不会很强 [34]。因此，尽管去除单缺陷假设后，$a_{ef}(s^f) = F$ 这种强约束关系不再成立，然而我们仍然可以假设缺陷语句具有较大的 $a_{ef}(s^f)$ 值与较小的 $a_{ep}(s^f)$ 值，这与 SBFL 的基本原理也是吻合的。为此，可以相对保守地认为当某条语句具有很小的 a_{ef} 值与很大的 a_{ep} 值时，该语句很可能不是缺陷语句。基于此，图6.11中给出了 SPICA 下对程序谱图的区域划分，具体的划分理论如下所述。

SPICA 理论假设 1：给定程序谱图 $G(PG, T)$，将 $(0, 0)$ 到 (F, P) 这条对角线附近的区域称为失效独立区域，记为 A^{fi}。

失效独立区域是程序谱图中从 $(0, 0)$ 到 (F, P) 这条对角线附近的区域。此外，x 值越大表示相关语句的 a_{ef} 值越大，故对该区域的划定应更为严格；相反，x 值越小则表示语句的 a_{ef} 值越小，故对该区域的划定就可以比较宽松。因此我们将程序谱图中失

效独立区域的范围划定为由 $(0,0)$、$(0,P/10)$、$(F/5,0)$ 以及 (F,P) 点所围成的区域，如图 6.11中黄色部分所示。

SPICA 理论假设 2：给定程序谱图 $G(PG,T)$，将失效独立区域下方附近的区域称为缺陷模糊区域，记为 A^{bf}。

显然，成像在失效独立区域下方的语句具有更大的 a_{ef} 值与更小的 a_{ep} 值，这些语句的 a_{ef}/a_{ep} 值大于 F/P，因此这些语句与软件失效之间存在着某种关系。然而由于这些语句的成像仍然在失效独立区域附近，因此这种关联性相对较弱。虽然这一区域中的语句与程序失效存在某种关联，但其仍不太可能是缺陷语句，故称这一区域为缺陷模糊区域。具体来说，我们将该区域划分为 $(F/5,0)$、(F,P) 以及 $(F/3,0)$ 这三个点围成的三角形区域。

SPICA 理论假设 3：给定程序谱图 $G(PG,T)$，将缺陷模糊区域下方到程序谱图的右下角点 $(F,0)$ 之间的部分称为缺陷相关区域，记作 A^{br}。

当语句的位置靠近程序谱图的右下区域时，其 a_{ef} 的值较大而 a_{ep} 较小。这些语句所表现出的行为很可能与软件的缺陷相关。因此，我们将程序谱图的右下区域称为缺陷相关区域。一般来说，缺陷语句在程序谱图中更可能会成像于这一区域。图6.11 中，缺陷相关区域用白色表示。这里，为了减小区域划分的容错率，我们将该区域的范围划分得比较大（即点 $(F/3,0)$、(F,P) 以及 $(F,0)$ 围成的三角形区域）。

SPICA 理论假设 4：给定程序谱图 $G(PG,T)$，将失效独立区域上方的区域称为失效负相关区域，记作 A^{fn}。

对于 失效独立区域上方的语句，由于其 a_{ef}/a_{ep} 值很小，因此这些语句与程序失效是呈负相关的。也就是说，当测试用例执行该区域的语句时，软件会有较低的失效率。

SPICA 理论假设 5：给定程序谱图 $G(PG,T)$，将左方 y 轴附近的区域称为 缺陷负相关的区域，记作 A^{bn}。

软件缺陷的行为主要体现在两个方面，一方面是失效率，而另一个方面是其触发的失效测试的数量。考虑语句 s，其失效率为 $a_{ef}(s)/(a_{ef}(s)+a_{ep}(s))$。而我们认为缺陷语句会有更高的失效率，这一点在失效独立区域与缺陷模糊区域的划分上得以体现。此外，T 中有 F 个失效测试用例，这些失效测试用例本身一定是由缺陷语句所触发的。也就是说，相比于其它语句来说，缺陷语句应有更高的 a_{ef}/F 值。已知在单缺

陷假设下有 $a_{ef}(s^f)/F = 1$，现将其推广到多个缺陷的情况。假设程序中存在两条缺陷语句，则即使在两个缺陷完全不相关的情况下，其中的一条缺陷语句 s^f 也一定满足 $a_{ef}(s^f)/F \geq 1/2$。而对于存在更多缺陷语句的情况，我们有如下定理：

定理 6.1：*设程序 PG 中存在 k 条缺陷语句，则对于任意能够触发失效的测试用例集 T，必存在一条缺陷语句 s^f 满足*

$$a_{ef}(s^f)/F \geq 1/k \tag{6.7}$$

进而我们有

SPICA 定理 16：给定程序谱图 $G(PG,T)$，设程序 PG 中存在 k 条缺陷语句，则必有一条缺陷语句成像在直线 $x = F/k$ 的右侧。

以此为依据，我们将直线 $x = F/3$ 左边的部分，划为 *缺陷负相关区域*。首先，若目标程序的缺陷语句数小于 3，则必存在至少一条缺陷语句在直线 $x = F/3$ 的右侧。而当程序的缺陷语句很多时，缺陷之间的干涉会更加严重，同一条失效测试用例很可能覆盖多个缺陷语句，因此起主导作用的缺陷语句仍然非常可能满足 $x = F/3$。此外，如果目标程序的缺陷数量过多，那么其会变得更加脆弱，这种情况下 SBFL 本身就很可能不需要被采用。

6.5.2　基于程序谱空间划分的语句过滤算法

根据程序谱空间的划分理论，成像于 *失效独立区域*A^{fi}、*缺陷负相关区域*A^{bn}、*失效负相关区域*A^{fn} 与*缺陷模糊区域*A^{bf} 的语句包含缺陷的可能性较低，因此我们可以在实际缺陷定位中可以将其过滤并排在可疑度列表的后面，具体的算法过程如下所示。

基于程序谱空间划分的语句过滤算法：

步骤 1： 根据给定的 PG 与 T 计算各语句的 $a_{ef}(s)$ 与 $a_{ep}(s)$ 值，以及全局统计量 F 与 P 的值；

步骤 2： 分析程序谱图 $G(PG,T)$，对*失效独立区域*A^{fi}、*缺陷负相关区域*A^{bn}、*失效负相关区域*A^{fn} 以及*缺陷模糊区域*A^{bf} 进行划分；

步骤 3： 根据划分的区域对语句进行筛选，得出需要保留的语句集合 S^{remain} 以及需要过滤的语句集合 S^{filter}；

步骤 4： 选取可疑度计算公式 R，并计算各语句 s 的可疑度 r^s，进而得到原始

的可疑度列表 L；

步骤 5： 根据集合 S^{remain} 与 S^{filter} 对可疑度列表进行处理，生成经过处理的可疑度列表 L^*，使得在 L^* 中属于 S^{remain} 的语句排在前面，属于 S^{filter} 的语句排在后面，而 S^{remain} 或 S^{filter} 内部语句间的相对排位保持不变。

具体操作时，可以先计算当前选择的可疑度计算公式 $R(a_{ef}, a_{ep})$ 所能取到的最大值（即 $R(F,0)$），以及最小值（即 $R(0,P)$），并将所有 $s \in S^{filter}$ 的可疑度依下式重新计算：

$$r^s = R(a_{ef}(s), a_{ep}(s)) - (R(F,0) - R(0,P)) \tag{6.8}$$

最后，对各语句按照其新得到的可疑度进行排序，并得到最终的可疑度列表 L^*。

6.5.3 算法在经典可疑度计算公式上的性能

表 6.3 基于程序谱空间划分的 SBFL 语句过滤算法效果展示

	Op		Ochiai		Tarantula		Jaccard		Total	
	FBO	FWO	FBO	FWO	FBO	FWO	FBO	FWO	FBO	FWO
flex	32	0	39	14	741	77	51	15	863	106
grep	8	0	26	13	269	23	40	13	343	49
gzip	9	3	3	4	154	15	3	4	169	26
sed	38	0	2	0	114	0	2	0	156	0
Siemens	52	0	89	30	637	98	470	66	1248	194
Space	15	0	33	18	322	78	43	20	413	116
Total	154	3	192	79	2237	291	609	118	3192	491

本小节在标准程序上对基于程序谱空间划分的语句过滤算法性能进行检验，以验证其有效性。这里仍然选用与第五章相同的实验对象，即 UNIX 程序（包括 *flex* 、*grep* 、*gzip* 与 *sed*）、*Siemens* 测试套件以及 *Space* 程序，并且同样按照第五章的设置来生成各程序的单缺陷及多缺陷版本。在各个缺陷版本上我们分别运行传统的 SBFL 以及包含语句滤除机制的 SBFL，并将二者的 Expense 值进行比较，其结果在表6.3 中显示。这里我们对 Op、Ochiai、Tarantula 以及 Jaccard 这几个公式进行考查。其中 Op 是单缺陷下的 Maximal 公式，Ochiai 是目前公认效果较好的可疑度计算公式，Tarantula 是早期比较著名的可疑度计算公式，而 Jaccard 是 SPICA 下线性曲线公式中比较著名且效果较好的可疑度计算公式。表 6.3 中题为“FBO”（即“Filter Better than Origin”）的列表示语句过滤算法起到积极作用的缺陷版本数，而题为“FWO”（即”Filter Worse than Origin”）的列表示算法起消极作用的缺陷版本数。

从中可以看出，在 SBFL 中加入基于程序谱空间划分的语句过滤机制能够明显地提升缺陷定位的效果。表6.3中，除了 *gzip* 程序上基于 Ochiai 公式得到的缺陷定位结果外（算法在 3 个缺陷版本上取得了提升，在 4 个版本上下降），算法都能够取得较为明显的提升。统合所有数据，应用基于程序谱空间划分的语句过滤算法使得 SBFL 的性能在 3192 个缺陷版本上取得了提升，而仅在 491 个缺陷版本上性能下降，上升版本数是下降版本数的 6.5 倍。

对于不同的实验对象来说，算法在 *flex* 、*grep* 、*sed* 与 *siemens* 上提升明显。在 *flex* 、*grep* 与 *siemens* 上，提升版本数分别为 863、343 与 1248，分别是下降版本数的 8 倍、7 倍与 6 倍。在 *sed* 上，算法在 156 个缺陷版本上取得了提升，而 SBFL 性能下降的版本数为 0。算法在 *Space* 以及 *gzip* 程序上也取得了性能提升，但提升程度不如 *flex* 、*grep* 与 *sed* 显著。在 *Space* 程序上，SBFL 在 413 个缺陷版本上的性能得到提升，在 116 个缺陷版本上性能下降，上升版本数是下降版本数的 3.5 倍。而在 *gzip* 程序中，虽然上升版本数是下降版本数的 6.5 倍，然而语句滤除算法在 Ochiai 公式与 Jaccard 公式上并没有起到积极作用。可见，算法的效果会受到程序结构及缺陷本身性质的影响。

此外，对于不同的可疑度计算公式来说，算法的性能也有很大的不同。首先，使用 Tartantula 公式时，算法在所有实验对象上都取得了显著提升，取得提升的缺陷版本总数达到了 2237，而 SBFL 性能降低的缺陷版本数仅为 291。在 *flex* 、*grep* 与 *gzip* 程序中，SBFL 性能提升的版本数分别达到了下降版本数的 9 倍、10 倍与 11 倍。而在 *sed* 程序中 SBFL 效果上升的版本数为 114 个，下降版本数为 0。在 Op 公式上，语句过滤算法同样带来了较大的收益。尽管其在各目标程序上取得提升的缺陷版本数量不多（在 *flex* 、*grep* 、*gzip* 、*sed* 、*Siemens* 与 *Space* 上 SBFL 取得性能提升的缺陷版本数分别为 32、8、9、38、52 以及 15），然而算法起消极作用的版本数也非常少（在 *gzip* 程序上有 3 个，而在其它程序上全部为 0）。对于 Ochiai 与 Jaccard 公式，语句滤除算法所起到的作用不大。算法在这两个可疑度计算公式上取得性能提升缺的陷版本数分别是起消极作用的缺陷版本数的 2.4 倍与 5 倍。此外，这两个公式在 *gizp* 程序上起消极作用的缺陷版本数（3 个）要稍多于 SBFL 取得性能提升的缺陷版本数（4 个）。

总结来说，该算法能够在一定程度上过滤缺陷定位过程中由于目标程序以及软件缺陷的多样性带来的干扰信息，提升 SBFL 可疑度公式的适应性。实验结果表明，基于程序谱空间划分的语句滤除算法是较为有效的，这也说明本节基于 SPICA 对程序谱

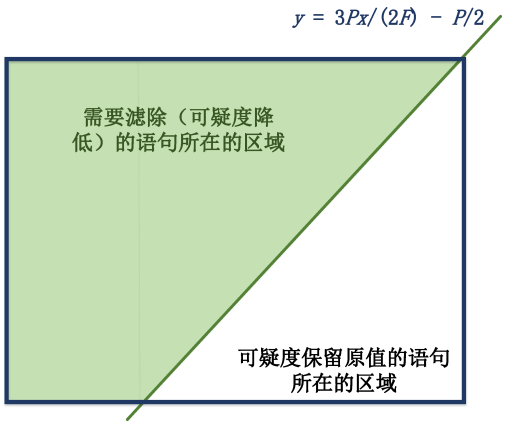


图 6.12　被过滤的语句在程序谱图中的位置

空间的划分理论是较为合理的。然而，算法本身的适应性还是会受到不同目标程序的影响。此外，对于不同的可疑度计算公式来说，该算法能够取得提升的程度也会不同。在下一小节，我们将根据本小节中的实验结果在 SPICA 下对程序谱空间以及各可疑度公式的适应性做进一步分析。

6.5.4　程序谱分布与公式曲线几何特征分析

本小节将对第6.5.3小节中基于程序谱图空间划分的语句过滤算法所得到的结果进行分析，并以此为依据，对程序谱图上的语句谱分布规律以及公式曲线更多的性质进行探究，进而分析可疑度计算公式的适应性及合理性。

首先，图6.12 画出了本节所提出的语句过滤算法所需要过滤的语句在程序谱图中的成像区域。可以看出，当同时考虑失效独立区域、缺陷负相关区域、失效负相关区域以及缺陷模糊区域（即 $A^{fi} \cup A^{bn} \cup A^{fn} \cup A^{bf}$）时，实际滤除的语句在程序谱图中的成像均在直线 $y = \frac{3P}{2F}x - \frac{P}{2}$ 上方，记该区域为 A^{filter}，即

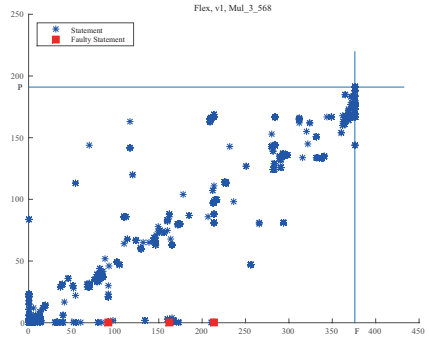
$$A^{filter} = A^{fi} \cup A^{bn} \cup A^{fn} \cup A^{bf} \tag{6.9}$$

同时，记该区域的补区域为 $A^{remain} = G(PG,T) - A^{filter}$，也就是维持原来可疑度值的语句所成像的区域。

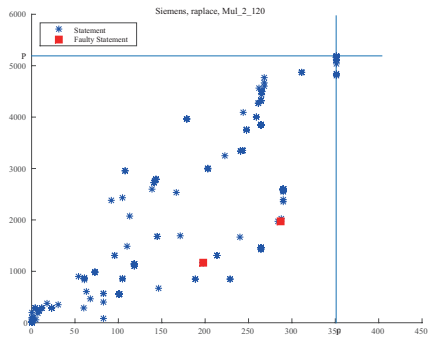
若经过语句过滤后 SBFL 的性能取得了提升，则所采用的可疑度计算公式、程序的缺陷谱以及程序谱分布反映在程序谱图 $G(PG,T)$ 中应该具有以下特征：

（1）所采用公式 R 的公式曲线 MC^R 需同时经过区域 A^{filter} 与 A^{remain}；

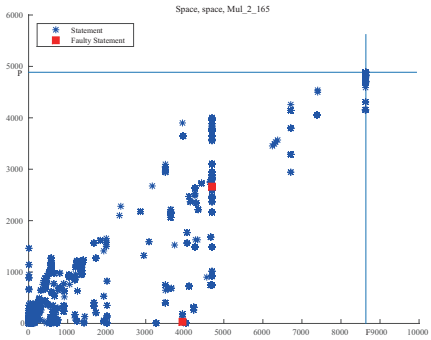
（2）缺陷语句 s^f 在程序谱图中的成像满足 $po(s^f) \in A^{remain}$；



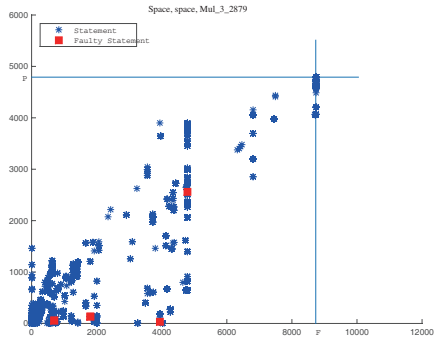
(a) *flex*, v1, Mul3_568



(b) *Siemens*, *replace*, Mul2_120



(c) *Space*, *space*, Mul2_165



(d) *Space*, *space*, Mul3_2879

图 6.13　语句滤除方法在四种可疑度计算公式上都取得了较好效果的实例

（3）存在非缺陷语句 s 满足 $po(s) \in A^{filter}$，且 s 成像于公式曲线划分的不可分辨区域 G^n 或错分区域 G^e 中，即 $\exists s, po(s) \in A^{filter} \cup G^n \cup G^e$。

为此，我们对实验数据进行进一步观察，找出了过滤算法对于所有可疑度计算公式都取得了提升的实验对象及缺陷版本，其典型实例的程序谱图如图6.13 所示。可以看出四个实例都是多缺陷实例，且主要的缺陷语句的 a_{ef} 都小于 F，并且过滤区域 A^{filter} 中包含 a_{ef} 较大或者是 a_{ep} 较小的非缺陷语句。这些语句虽然在某一方面具有类似缺陷语句所表现出来的特征，然而这种特征并不显著，并且其表现出很多缺陷语句并不应该表现出来的特征。

相对地，若考虑语句滤除算法对 SBFL 产生消极作用的情况，则所采用的可疑度计算公式、程序的缺陷谱以及程序谱分布反映在程序谱图 $G(PG, T)$ 中应该具有以下特征：

（1）所采用公式 R 的公式曲线 MC^R 需同时经过区域 A^{filter} 与 A^{remain}；

（2）缺陷语句 s^f 在程序谱图中的成像满足 $po(s^f) \in A^{filter}$；

（3）存在非缺陷语句 s 满足 $po(s) \in A^{remain}$，且 s 成像于公式曲线划分的可分辨

区域 G^i 中，即 $\exists s, po(s) \in A^{remain} \cup G^i$。

表 6.4　算法对于各可疑度公式单独取得性能提升的缺陷版本数量

R	Op	Ochiai	Tarantula	Jaccard
No.	38	21	1572	15

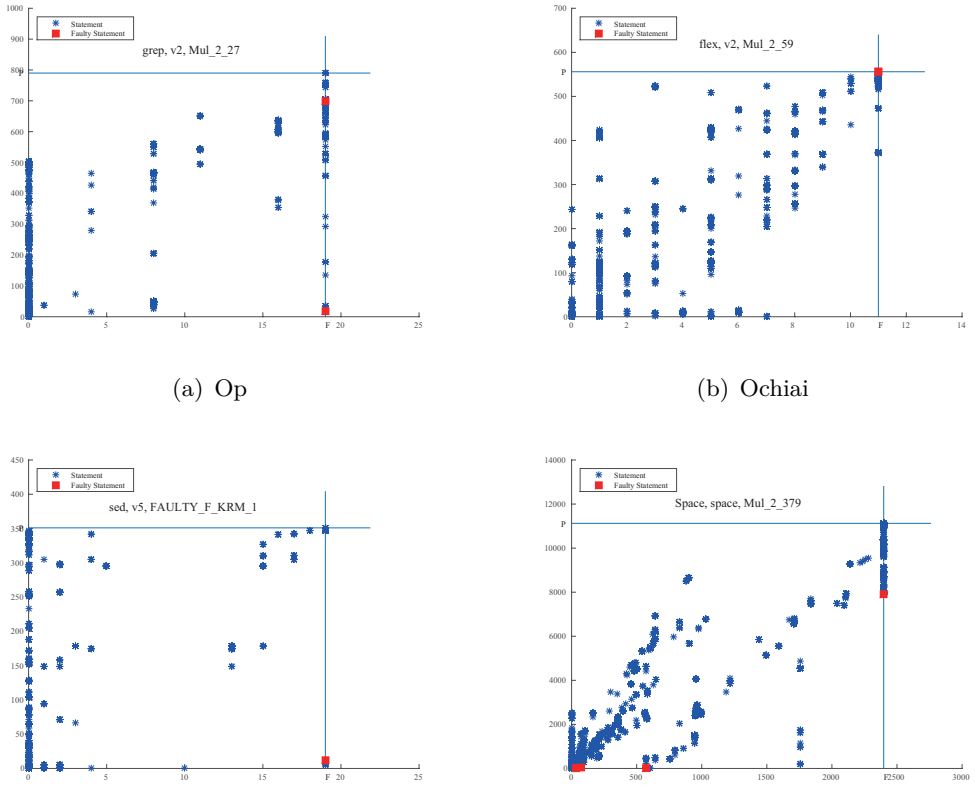
然而我们发现，在所有的实验数据中，不存在一组实例使得语句过滤算法在四个可疑度计算公式下都起消极作用，这也从另一个方面证明了算法的有效性。更重要的是，这说明了不同可疑度计算公式在程序谱空间的划分机理上存在着明显的差异。

一般来说，所采用的可疑度计算公式在执行语句过滤算法后能够取得提升，正说明该可疑度计算公式在某方面的适应性出现了问题。也就是说，在某个实例中，该公式的公式曲线经过了不太合理的区域。为此，表6.4 列出了语句过滤算法在各可疑度公式下单独取得性能提升的实例数量。可以看出，算法在 Ochiai 和 Jaccard 公式下单独取得提升的缺陷版本数较少（分别是 23 个和 15 个），这说明 Ochiai 和 Jaccard 这两个可疑度计算公式具有较好的适应性。而算法在 Op 上单独取得提升的缺陷版本数较多（38 个），这说明 Op 公式在某些情况下的适应性存在问题。而算法在 Tuantula 公式下单独取得提升的缺陷版本数则非常多（1572 个），这说明 Tuantula 公式的公式曲线在非常多的缺陷版本下都会穿过一些不太可能包含缺陷的区域。因此，其适应性不如 Op、Ochiai 以及 Jaccard 好。

这里我们从各可疑度计算公式上语句过滤算法起积极作用的实例中，选择代表性的实例，并将其程序谱图绘制在图6.14 中。我们看到，由于 Op 公式的公式曲线斜率非常大，因此只要 a_{ef}^{sf} 的值不等于 F（即多缺陷场景），其公式曲线就会经过程序谱图右上的失效无关及负相关的区域。而程序谱图的右上角区域通常会存在较多的非缺陷语句（例如程序 main 函数中的语句）。因此，尽管其在单缺陷假设下是 Maximal 公式，Op 公式对多缺陷情景的适应性较差。

而 Tarntula 公式的公式曲线为连接缺陷语句的成像点与坐标原点的直线，因此其必经过坐标原附近的区域。然而，在程序谱图坐标原点附近的区域中，语句的 a_{ef} 值非常小，因此赋予其很高的可疑度值是不合理的。当该区域存在较多语句时，Tarntula 公式的性能会比较差。而实际程序中，那些很少被经过的语句（例如异常处理机制以及一些不常用的功能模块中的语句）都大概率会成像在这个区域，因此 Tarntula 公式的适应性比较差。

相比之下，由于 Ochiai 公式与 Jaccard 公式的公式曲线在缺陷语句成像点上斜率



(a) Op　(b) Ochiai

(c) Tarantula　(d) Jaccard

图 6.14　语句滤除算法下各可疑度计算公式单独取得性能提升的示例

都不至于过大，并且其与 x 轴的交点同样比较靠右侧。因此 Ochiai 与 Jaccard 的公式曲线大概率不会穿过 A^{filter} 区域，因此其具有较好的适应性。

进一步，图6.15 绘出了统计所有程序以及它们的缺陷版本而得到的缺陷语句的谱分布图。由于不同缺陷版本的 F 和 P 值不同，因此我们将其归一化，即所有缺陷语句的 a_{ef}^{sf} 值除以该缺陷版本下得到的 F 值，而所有缺陷语句的 a_{ep}^{sf} 值除以该缺陷版本下得到的 P 值。从图中可以看出，绝大多数缺陷语句都成像在本节划分的缺陷相关区域中，且在 x 坐标轴（即满足 $a_{ep}=0$）以及直线 $x=F$ 附近。这个现象是合理的，若程序符合单缺陷假设，则 $a_{ef}^{sf}=F$ 得到满足；而若程序中存在多个缺陷且在缺陷之间存在干涉现象，则尽管对于缺陷语句会有 $a_{ef}^{sf}<F$，但多个缺陷的相互作用会使其 a_{ep} 值更小，因此其更加趋近于 x 轴。这一方面说明了 SBFL 方法本身的合理性，即无论是单缺陷还是多缺陷程序，缺陷语句的程序谱都有很大可能具备鲜明的鉴别特征。而另一方面，从缺陷谱的分布可以看出，可疑度计算公式必须对缺陷语句成像于 x 轴附近，以及直线 $x=F$ 附近（即程序谱图下侧与右侧两个边界）的情况具有良好的适应性。Op 公式虽然对单缺陷假设下 $a_{ef}^{sf}=F$ 的情况具有良好的适应性，然而其对缺陷语句位

于 x 轴附近时的情况不够敏感。Trantula 公式着眼于缺陷语句位于 x 轴上的情况，然而其公式曲线一定会经过原点，这使得原点附近（缺陷负相关区域）的语句被错误的分辨，因此其适应性较差。而反观 Ochiai 公式与 Jaccard 公式，其公式曲线同时兼顾了这两类缺陷语句（即成像在 x 轴附近，以及直线 $x = F$ 附近的缺陷语句），因此具有较好的适应性。

6.6 本章小结

本章引入 SPICA 这一可视化的程序谱分析框架对 SBFL 方法本身，以及可疑度计算公式的合理性与适应性进行分析。具体贡献包括以下五点：

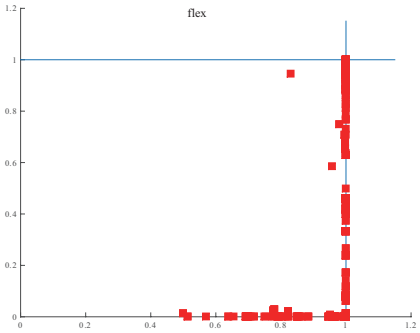
（1）提出可视化分析框架——SPICA。该框架具有整体性、统一性以及可表达性这些特点。能够对程序谱空间的基本性质进行可视化描述与分析。

（2）借助 SPICA，本章对程序谱本身的分布特征进行探索，并对 SBFL 可疑度计算公式对程序谱空间的划分机理进行描述。这些工作对于 SBFL 基础理论的进一步探究起到了促进作用。

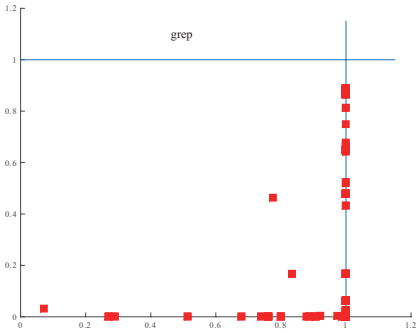
（3）在 SPICA 下对各可疑度公式在单缺陷假设下的执行机理进行描述，并对其性能进行了比较与证明，得到了更为丰富的证明结果以及较为深刻的结论。

（4）在 SPICA 下提出了程序谱空间划分理论，并基于该理论设计了相应的语句过滤算法，提升了经典 SBFL 可疑度计算公式的适应性。

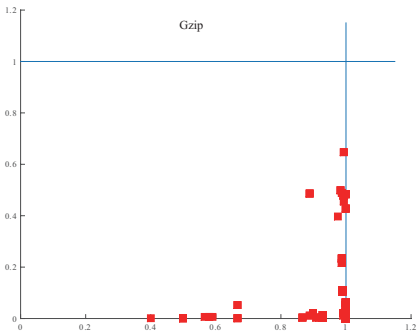
（5）结合程序谱空间划分理论以及各可疑度计算公式对程序谱空间的划分机理，在去除单缺陷假设的一般情况下，对经典可疑度计算公式的适应性及合理性进行分析。



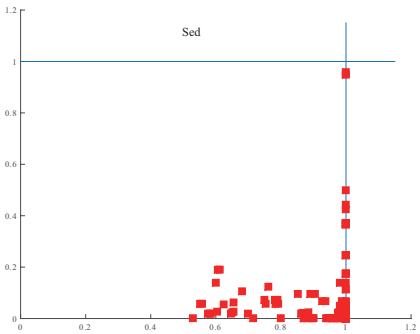
(a) *flex*



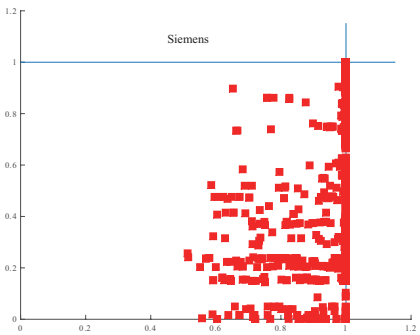
(b) *grep*



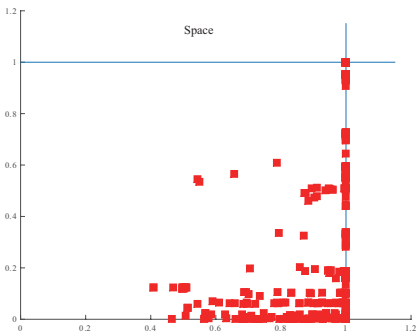
(c) *gzip*



(d) *sed*



(e) *Siemens*



(f) *Space*

图 6.15　各实验对象上不同缺陷版本的缺陷在程序谱图上的分布

结束语

本文面向软件质量与开发效率需求，以软件控制论思想为指导，围绕着软件工程中的软件缺陷定位环节的自动化方法及其适应性问题展开研究。进而提出自适应软件缺陷定位研究框架，并在此基础上提出相关的自适应机制，以提升软件缺陷定位方法对软件及开发环境多样性的适应性，从而切实提升软件缺陷定位方法的性能及应用价值。

论文根据软件控制论思想对软件缺陷定位的执行过程进行描述，以控制系统的视角分析其适应性，并借助控制领域的前馈及反馈思想引入自适应软件缺陷定位的概念。在此基础上，分析了软件缺陷定位各环节的实施流程、输入输出及其与相关环境因素间的相互作用机制，对其适应性进行讨论，并针对相应的环节引入适当的自适应机制。

根据研究内容和各部分之间的逻辑关系，正文内容可以分为三大部分：自适应软件缺陷定位研究框架、自适应方法的设计与算法实现、缺陷定位方法理论分析框架的建立以及在此基础上的适应性分析。这三部分相互依存，自适应缺陷定位研究框架从一个较高的视角诠释了软件缺陷定位中“自适应”的含义，并且指导相关自适应机制的设计；自适应机制的设计以及缺陷定位方法本身的适应性分析分别从内部与外部两个方面对自适应问题及其解决方案进行探究；而具体方法的实现以及从实验平台中得到的数据也为自适应软件缺陷定位研究领域本身的完善以及今后的发展铺平了道路。各部分内容的具体阐述如下：

（1）第一部分包括第一章和第二章。该部分首先提出软件缺陷定位的适应性需求，以控制论视角初步阐明自适应软件缺陷定位的具体含义，即在软件缺陷定位流程中加入前馈或反馈控制，亦或是适应性观测与评估机制，以提升自动化缺陷定位方法的适应性。进而我们对缺陷定位环节中的各个流程进行分析，并对其中相应的自适应问题及自适应机制进行描述（包括对其输入输出与前馈反馈路径）。在此基础上，我们建立缺陷定位的一个较为通用的数学模型，并借此说明这些自适应机制的合理性以及设计理念。

（2）第二部分包括第三章到第五章，主要针对软件缺陷定位的各个环节设计并实现了具体的自适应机制。这包括，基于覆盖信息多样性的缺陷定位测试优化方法（第三章）、基于输入信息多样性的缺陷定位测试优化方法（第四章）以及基于程序谱反馈的自适应缺陷定位信息处理方法（第五章）。其中，面向缺陷定位的测试优化问题旨在

通过对面向缺陷定位的测试过程进行优化，使得缺陷定位方法能够更有效率地接收对缺陷代码定位有用的测试信息。对此，第三章与第四章分别利用测试用例的覆盖信息以及程序的输入空间信息来设计相应的自适应机制。其中测试用例的覆盖信息更为精确，但在实际缺陷定位环境中不易获取；而输入空间信息则容易获取，但其没有测试覆盖信息精确，且需要额外的信息空间变换过程。此外，缺陷定位信息处理问题旨在通过对测试过程中得到的测试信息进行适当的处理，来提升缺陷定位过程中的信息利用效率。本文利用测试分类方法，对无标签测试用例的输出正确性进行预测，并将预测结果应用在程序谱缺陷定位方法中。其中，我们在分类器设计过程中引入了反馈思想，将仅利用原始测试信息进行缺陷定位所得到的“粗糙”结果应用到分类器的训练中，取得了较好的效果。

（3）第三部分包括第六章。这部分的重点并不是具体的自适应机制或算法，而是我们尝试去挖掘程序谱缺陷定位的深层理论，进而去完善自适应软件缺陷定位以及软件缺陷定位本身的理论基础。这一章中，我们提出了程序谱缺陷定位方法的可视化适应性分析框架（SPICA），并在该框架下对缺陷程序谱、程序谱分布以及可疑度计算公式进行深入分析，发现了很多新的现象与结论，这些结论对于缺陷定位方法的合理性及适应性评估来说是非常有意义的。此外，这些分析与结论还引出了很多新的研究问题以及研究思路，对程序谱缺陷定位这一研究领域的扩展，起到了积极作用。该部分内容不仅仅局限于自适应缺陷定位的研究，还是本文在对软件缺陷定位这一领域进行各种探索后，对其本质的理解与概括。

延续本论文的问题、思路和技术路线，未来可能的研究方向主要涉及到：

- 完善研究框架的理论基础。本文提出了自适应软件缺陷定位的理论框架，并且对其合理性进行了论述；在未来的研究工作中，我们将进一步针对框架中涉及的自适应问题在理想条件下进行数学建模并求解，建立比较完整的自适应软件缺陷定位理论体系。

- 完善具体的自适应机制。对于面向缺陷定位的自适应测试、调试人员与系统的交互策略、测试信息处理等问题，在更多实际的情景想定下提出更为实用的解决方案及技术，以丰富研究框架的内容。

- 对本文提出的方法在更多标准程序（如 Defect4J 程序库）上进行实验，并对实验数据进行更为深入的分析，以得到关于软件缺陷定位适应性的更多结论。

- 在 SPICA 框架下，进一步探索关于程序谱分布的基础理论，以及程序谱分布与不同程序结构及缺陷之间的关系，以得到更多的结论，提升 SBFL 的应用价值。

参考文献

[1] Collofello J S, Woodfield S N. Evaluating the effectiveness of reliability-assurance techniques[J]. Journal of systems and software, 1989, 9(3): 191-195.

[2] Xie X, Chen T Y, Kuo F C, et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2013, 22(4): 31.

[3] 张银奎. 软件调试 [M]. 电子工业出版社, 2008.

[4] Parnin C, Orso A. Are automated debugging techniques actually helping programmers?[C]//Proceedings of the 2011 international symposium on software testing and analysis. ACM, 2011: 199-209.

[5] Pfleeger S L, Atlee J M. Software Engineering: Theory and Practice (4th Edition)[M]. Prentice Hall, 2009.

[6] 蔡开元. 关于软件可靠性和软件控制论的若干认识 [J]. 中国科学基金, 2004, 18(4): 201-204.

[7] Campos J, Abreu R, Fraser G, et al. Entropy-based test generation for improved fault localization[C]//Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2013: 257-267.

[8] Lipow M. Number of faults per line of code[J]. IEEE Transactions on Software Engineering, 1982 (4): 437-439.

[9] Dugan J B, Van Buren R. Reliability evaluation of fly-by-wire computer systems[J]. Journal of Systems and software, 1994, 25(1): 109-120.

[10] 金钟河. 致命 Bug[M]. 人民邮电出版社, 2016.

[11] Fitzgerald B, Stol K J, O'Sullivan R, et al. Scaling agile methods to regulated environments: An industry case study[C]//Proceedings of the 35th International Conference on Software Engineering (ICSE2013). IEEE, 2013: 863-872.

[12] Androutsellis-Theotokis S, Spinellis D, Kechagia M, et al. Open source software: A survey from 10,000 feet[J]. Foundations and Trends® in Technology, Information and Operations Management, 2011, 4(3–4): 187-347.

[13] Zhang W, Yang Y, Wang Q. An empirical study on identifying core developers using network analysis[C]//Proceedings of the 2nd international workshop on evidential assessment of software technologies. ACM, 2012: 43-48.

[14] Rothwell R. Creating wealth with free software[J]. Free Software Magazine, 2008.

[15] Gousios G, Vasilescu B, Serebrenik A, et al. Lean GHTorrent: GitHub data on demand[C]//Proceedings of the 11th working conference on mining software repositories. ACM, 2014: 384-387.

[16] Heller B, Marschner E, Rosenfeld E, et al. Visualizing collaboration and influence in the open-source software community[C]//Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, 2011: 223-226.

[17] Celebrating nine years of github with an anniversary sale [OL]. (2017) https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversarv-sale.

[18] White G, Sivitanides M. Cognitive differences between procedural programming and object oriented programming[J]. Information Technology and management, 2005, 6(4): 333-350.

[19] Chidamber S R, Kemerer C F. A metrics suite for object oriented design[J]. IEEE Transactions on software engineering, 1994, 20(6): 476-493.

[20] I. Sommerville. Software Engineering (9th Edition). Addison Wesley, August 2006.

[21] Taylor R N, Medvidovic N, Dashofy E M. Software Architecture: Foundations, Theory, and Practice[M]. Wiley Publishing, 2009.

[22] C*â*ndido G, Barata J, Colombo A W, et al. SOA in reconfigurable supply chains: A research roadmap[J]. Engineering applications of artificial intelligence, 2009, 22(6): 939-949.

[23] Papazoglou M P, Van Den Heuvel W J. Service oriented architectures: approaches, technologies and research issues[J]. The VLDB journal, 2007, 16(3): 389-415.

[24] Erl T. Service-oriented architecture: concepts, technology, and design[M]. Pearson Education India, 2005.

[25] Gao Y C, Zheng Z, Qin F Y. Analysis of Linux kernel as a complex network[J]. Chaos, Solitons & Fractals, 2014, 69: 246-252.

[26] Wang H Q, Chen Z, Xiao G P, et al. Network of networks in Linux operating system[J]. Physica A: Statistical Mechanics and its Applications, 2016, 447: 520-526.

[27] Xiao G P, Zheng Z, Wang H Q. Evolution of Linux operating system network[J]. Physica A: Statistical Mechanics and its Applications, 2017, 466: 249-258.

[28] Welcome to Visual Studio 2015 - MSDN - Microsoft[OL]. (2017) https://msdn.microsoft.com/en-us/library/dd831853.aspx.

[29] The Eclipse Foundation-Filtered UDC Data[OL]. (2016). http://archive.eclipse.org/projects/usagedata.

[30] PyCharm Edu: Easy and Professional Tool to Learn & Teach Programming with Python[OL]. (2017), https://www.jetbrains.com/pycharm-edu/

[31] Skinner J. Sublime Text 3[OL]. http://www.sublimetext.com/3

[32] Le A. SpaceMac Library[OL]. (2011), http://www.ics.uci.edu/anhml/software.html#spacemac.

[33] JetBrains Inc. ReSharper plugin for VisualStudio[OL]. http://www.jetbrains.com/resharper/.

[34] DiGiuseppe N, Jones J A. Fault density, fault types, and spectra-based fault localization[J]. Empirical Software Engineering, 2015, 20(4): 928-967.

[35] Smith M D, Robson D J. A framework for testing object-oriented programs[J]. Journal of Object-Oriented Programming, 1992, 5(3): 45-53.

[36] Kung D C, Gao J, Kung C H. Testing object-oriented software[M]. IEEE Computer Society Press, 1998.

[37] Hayes J H. Testing of object-oriented programming systems (OOPS): A fault-based approach[M]//Object-oriented methodologies and systems. Springer, Berlin, Heidelberg, 1994: 205-220.

[38] Hayes J H, Chemannoor I R, Holbrook E A. Improved code defect detection with fault links[J]. Software Testing, Verification and Reliability, 2011, 21(4): 299-325.

[39] Trivedi K S, Mansharamani R, Kim D S, et al. Recovery from failures due to Mandelbugs in IT systems[C]//Proceedings of the 17th Pacific Rim International Symposium on Dependable Computing (PRDC2011). IEEE, 2011: 224-233.

[40] Cotroneo D, Pietrantuono R, Russo S, et al. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation[J]. Journal of Systems and Software, 2016, 113: 27-43.

[41] Masri W, Assi R A. Prevalence of coincidental correctness and mitigation of its impact on fault localization[J]. ACM transactions on software engineering and methodology (TOSEM), 2014, 23(1): 8.

[42] Jiang B, Zhang Z, Chan W K, et al. How well does test case prioritization integrate with statistical fault localization?[J]. Information and Software Technology, 2012, 54(7): 739-758.

[43] Wiener N. Cybernetics or Control and Communication in the Animal and the Machine[M]. MIT press, 1961.

[44] Cai K Y. On the Concepts of Total Systems, Total Dependability and Software Cybernetics[J]. Centre for Software Reliability, City University, London, 1994.

[45] Dejaeger K, Verbraken T, Baesens B. Toward comprehensible software fault prediction models using bayesian network classifiers[J]. IEEE Transactions on Software Engineering, 2013, 39(2): 237-257.

[46] Chen T Y, Kuo F C, Merkel R G, et al. Adaptive random testing: The art of test case diversity[J]. Journal of Systems and Software, 2010, 83(1): 60-66.

[47] Yoo S, Harman M, Clark D. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2013, 22(3): 19.

[48] Cai K Y, Cangussu J W, DeCarlo R A, et al. An overview of software cybernetics[C]//Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on. IEEE, 2003: 77-86.

[49] Cai K Y. Optimal software testing and adaptive software testing in the context of software cybernetics[J]. Information and Software Technology, 2002, 44(14): 841-855.

[50] Marchand H, Samaan M. Incremental design of a power transformer station controller using a controller synthesis methodology[J]. IEEE Transactions on Software Engineering, 2000, 26(8): 729-741.

[51] Grottke M, Trivedi K S. Fighting bugs: Remove, retry, replicate, and rejuvenate[J]. Computer, 2007, 40(2).

[52] Wong W E, Gao R, Li Y, et al. A survey on software fault localization[J]. IEEE Transactions on Software Engineering, 2016, 42(8): 707-740.

[53] Radatz J, Geraci A, Katki F. IEEE standard glossary of software engineering terminology[J]. IEEE Std, 1990, 610121990(121990): 3.

[54] Myers G J, Sandler C, Badgett T. The art of software testing[M]. John Wiley & Sons, 2011.

[55] Wong W E, Debroy V. Software Fault Localization[J]. Encyclopedia of Software Engineering, 2010, 1: 1147-1156.

[56] 李伟, 郑征, 郝鹏, 等. 基于谓词执行序列的软件缺陷定位算法 [J]. 计算机学报, 2013, 36(12): 2406-2419.

[57] Weiser M. Program slicing[C]//Proceedings of the 5th International Conference on Software engineering. IEEE Press, 1981: 439-449.

[58] Korel B, Laski J. Dynamic program slicing[J]. Information processing letters, 1988, 29(3): 155-163.

[59] Agrawal H, Horgan J R. Dynamic program slicing[C]//Acm Sigplan Notices. ACM, 1990, 25(6): 246-256.

[60] 张锋. 程序切片技术在软件测试中的应用 [J]. 电子技术与软件工程, 2014 (17): 70-70.

[61] Ju X, Jiang S, Chen X, et al. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices[J]. Journal of Systems and Software, 2014, 90: 3-17.

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

[63] Davis R. Diagnostic reasoning based on structure and behavior[J]. Artificial intelligence, 1984, 24(1-3): 347-410.

[64] Reiter R. A theory of diagnosis from first principles[J]. Artificial intelligence, 1987, 32(1): 57-95.

[65] Console L, Friedrich G, Dupré D T. Model-based diagnosis meets error diagnosis in logic programs[C]//International Workshop on Automated and Algorithmic Debugging. Springer, Berlin, Heidelberg, 1993: 85-87.

[66] Abreu R, Zoeteweij P, Van Gemund A J C. Spectrum-based multiple fault localization[C]//Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE2009). IEEE, 2009: 88-99.

[67] Friedrich G, Stumptner M, Wotawa F. Model-based diagnosis of hardware designs[J]. Artificial Intelligence, 1999, 111(1): 3-39.

[68] Wieland D. Model-based debugging of Java programs using dependencies[M]. na, 2001.

[69] Mayer W, Stumptner M. Model-based debugging using multiple abstract models[J]. arXiv preprint cs/0309030, 2003.

[70] Yilmaz C, Williams C. An automated model-based debugging approach[C]//Proceedings of the twenty-second IEEE/ACM International Conference on Automated software engineering. ACM, 2007: 174-183.

[71] Mayer W, Stumptner M. Evaluating models for model-based debugging[C]//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE2008). IEEE Computer Society, 2008: 128-137.

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

[73] Reps T, Ball T, Das M, et al. The use of program profiling for software maintenance with applications to the year 2000 problem[M]//Software Engineering—ESEC/FSE'97. Springer, Berlin, Heidelberg, 1997: 432-449.

[74] Abreu R, Zoeteweij P, Van Gemund A J C. On the accuracy of spectrum-based fault localization[C]//Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007. IEEE, 2007: 89-98.

[75] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis[J]. ACM Transactions on software engineering and methodology (TOSEM), 2011, 20(3): 11.

[76] Abreu R, Zoeteweij P, Van Gemund A J C. An evaluation of similarity coefficients for software fault localization[C]//Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on. IEEE, 2006: 39-46.

[77] Abreu R. Spectrum-based fault localization in embedded software[D]. TU Delft, Delft University of Technology, 2009.

[78] Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation[J]. Acm Sigplan Notices, 2005, 40(6): 15-26.

[79] Liu C, Fei L, Yan X, et al. Statistical debugging: A hypothesis testing-based approach[J]. IEEE Transactions on software engineering, 2006, 32(10): 831-848.

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

[81] 郝鹏, 郑征, 张震宇, 等. 基于谓词执行信息分析的自适应缺陷定位算法 [J]. 计算机学报, 2014, 37(3): 500-511.

[82] Hao P, Zheng Z, Gao Y, et al. Statistical fault localization in decision support system based on probability distribution criterion[C]//IFSA World Congress and NAFIPS Annual Meeting (IFSA/NAFIPS), 2013 Joint. IEEE, 2013: 878-883.

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

[84] Saadatmand M, Scholle D, Leung C W, et al. Runtime verification of state machines and defect localization applying model-based testing[C]//Proceedings of the WICSA 2014 Companion Volume. ACM, 2014: 6.

[85] Perez A, Abreu R, Riboira A. A dynamic code coverage approach to maximize fault localization efficiency[J]. Journal of Systems and Software, 2014, 90: 18-28.

[86] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique[C]//Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering. ACM, 2005: 273-282.

[87] Le T D B, Thung F, Lo D. Theory and practice, do they match? a case with spectrum-based fault localization[C]//Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM2013). IEEE, 2013: 380-383.

[88] Debroy V, Wong W E. Combining mutation and fault localization for automated program debugging[J]. Journal of Systems and Software, 2014, 90: 45-60.

[89] Moon S, Kim Y, Kim M, et al. Hybrid-MUSE: mutating faulty programs for precise fault localization[R]. Technical report, KAIST, 2014.

[90] Mao X, Lei Y, Dai Z, et al. Slice-based statistical fault localization[J]. Journal of Systems and Software, 2014, 89: 51-62.

[91] Moon S, Kim Y, Kim M, et al. Ask the mutants: Mutating faulty programs for fault localization[C]//Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST2014). IEEE, 2014: 153-162.

[92] 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[C]//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). ACM, 2009: 1-5.

[93] Bandyopadhyay A, Ghosh S. Tester feedback driven fault localization[C]//Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST2012). IEEE, 2012: 41-50.

[94] Cleve H, Zeller A. Locating causes of program failures[C]//Proceedings of the 27th International Conference on Software Engineering (ICSE2005). IEEE, 2005: 342-351.

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

[96] Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching[C]//Proceedings of the 28th International Conference on Software Engineering (ICSE2006). ACM, 2006: 272-281.

[97] Zeller A.. Why programs fail: a guide to systematic debugging [M]. Access Online via Elsevier, 2009.

[98] Li H, Liu Y, Zhang Z, et al. Program structure aware fault localization[C]//Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices. ACM, 2014: 40-48.

[99] Le T D B, Lo D. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools[C]//Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM2013). IEEE, 2013: 310-319.

[100] Daniel P, Sim K Y. Spectrum-based Fault Localization: A Pair Scoring Approach[J]. Journal of Industrial and Intelligent Information Vol, 2013, 1(4).

[101] Lo D, Xia X. Fusion fault localizers[C]//Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE2014). ACM, 2014: 127-138.

[102] Stolberg S. Enabling agile testing through continuous integration[C]//Proceedings of the 2009 Agile Conference (AGILE2009). IEEE, 2009: 369-374.

[103] Duvall P M, Matyas S, Glover A. Continuous integration: improving software quality and reducing risk[M]. Pearson Education, 2007.

[104] Zhang X, Chen T, Liu H. An application of adaptive random sequence in test case prioritization[C]// Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute Graduate School, 2014: 126-131.

[105] Rothermel G, Untch R H, Chu C, et al. Test case prioritization: An empirical study[C]//Proceedings of the 1999 International Conference onSoftware Maintenance (ICSM1999). IEEE, 1999: 179-188.

[106] Elbaum S, Malishevsky A G, Rothermel G. Test case prioritization: A family of empirical studies[J]. IEEE transactions on software engineering, 2002, 28(2): 159-182.

[107] Lv J, Yin B B, Cai K Y. On the asymptotic behavior of adaptive testing strategy for software reliability assessment[J]. IEEE transactions on Software Engineering, 2014, 40(4): 396-412.

[108] Chen T Y, Merkel R. An upper bound on software testing effectiveness[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2008, 17(3): 16.

[109] Lv J, Hu H, Cai K Y, et al. Adaptive and random partition software testing[J]. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2014, 44(12): 1649-1664.

[110] Li Y, Yin B B, Lv J, et al. Approach for Test Profile Optimization in Dynamic Random Testing[C]//Proceedings of the 39th Annual Computer Software and Applications Conference (COMPSAC2015). IEEE, 2015, 3: 466-471.

[111] Zheng Z, Zhang X, Liu W, et al. Adapting Real-Time Path Planning to Threat Information Sharing[M]//Knowledge Engineering and Management. Springer, Berlin, Heidelberg, 2014: 317-329.

[112] Zhang X Y, Zheng Z, Cai K Y. A Fortification Model for Decentralized Supply Systems and Its Solution Algorithms[J]. IEEE Transactions on Reliability, 2018, 67(1): 381-400.

[113] Rothermel G, Untch R H, Chu C, et al. Prioritizing test cases for regression testing[J]. IEEE Transactions on software engineering, 2001, 27(10): 929-948.

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

[115] gcov—a Test Coverage Program[OL]. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[116] Ostrand T J, Balcer M J. The category-partition method for specifying and generating fuctional tests[J]. Communications of the ACM, 1988, 31(6): 676-686.

[117] Chen T Y, Poon P L, Tse T H. A choice relation framework for supporting category-partition test case generation[J]. IEEE transactions on software engineering, 2003, 29(7): 577-593.

[118] Chen T Y, Poon P L, Tang S F, et al. DESSERT: a DividE-and-conquer methodology for identifying categorieS, choiceS, and choicE Relations for Test case generation[J]. IEEE Transactions on Software Engineering, 2012, 38(4): 794-809.

[119] Chen T Y, Poon P L, Xie X. METRIC: METamorphic Relation Identification based on the Category-choice framework[J]. Journal of Systems and Software, 2016, 116: 177-190.

[120] Briand L C, Labiche Y, Liu X. Using machine learning to support debugging with tarantula[C]// Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE2007). IEEE, 2007: 137-146.

[121] Orso A, Rothermel G. Software testing: a research travelogue (2000–2014)[C]//Proceedings of the on Future of Software Engineering (FSE2014). ACM, 2014: 117-132.

[122] Harman M, McMinn P, Shahbaz M, et al. A comprehensive survey of trends in oracles for software testing[J]. University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01, 2013.

[123] Afzal W, Torkar R, Feldt R. A systematic review of search-based testing for non-functional system properties[J]. Information and Software Technology, 2009, 51(6): 957-976.

[124] Aichernig B K. Automated black-box testing with abstract VDM oracle[C]//International Conference on Computer Safety, Reliability, and Security. Springer Berlin Heidelberg, 1999: 250-259.

[125] Ali S, Briand L C, Hemmati H, et al. A systematic review of the application and empirical investigation of search-based test case generation[J]. IEEE Transactions on Software Engineering, 2010, 36(6): 742-762.

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

[127] Bozkurt M, Harman M. Automatically generating realistic test input from web services[C]//Proceedings of the 6th International Symposium on Service Oriented System Engineering (SOSE2011). IEEE, 2011: 13-24.

[128] Pastore F, Mariani L, Fraser G. Crowdoracles: Can the crowd solve the oracle problem?[C]//Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST2013). IEEE, 2013: 342-351.

[129] Zhou Z Q, Huang D H, Tse T H, et al. Metamorphic testing and its applications[C]//Proceedings of the 8th International Symposium on Future Software Technology (ISFST2004). 2004: 346-351.

[130] Zhang X Y, Zheng Z, Zhu Y N, et al. Protection issues for supply systems involving random attacks[J]. Computers & Operations Research, 2014, 43: 137-156.

[131] Chen T Y, Kuo F C, Liu H, et al. Code coverage of adaptive random testing[J]. IEEE Transactions on Reliability, 2013, 62(1): 226-237.

[132] Zhou Z Q. Using coverage information to guide test case selection in adaptive random testing[C]//Proceedings of the 34th Annual Computer Software and Applications Conference Workshops (COMPSACW2010). IEEE, 2010: 208-213.

[133] Zheng Z, Liu Y, Zhang X Y. The more obstacle information sharing, the more effective real-time path planning?[J]. Knowledge-Based Systems, 2016, 114: 36-46.

[134] Liu W, Zheng Z, Cai K Y. Distributed on-line path planner for multi-UAV coordination using bi-level programming[C]//Proceedings of the 25th Chinese Control and Decision Conference (CCDC2013). IEEE, 2013: 5128-5133.

[135] Jiang B, Zhai K, Chan W K, et al. On the adoption of MC/DC and control-flow adequacy for a tight integration of program testing and statistical fault localization[J]. Information and Software Technology, 2013, 55(5): 897-917.

[136] Zhang X Y, Towey D, Chen T Y, et al. Using partition information to prioritize test cases for fault localization[C]//Proceedings the 39th Annual Computer Software and Applications Conference (COMPSAC2015). IEEE, 2015, 2: 121-126.

[137] Yoo S, Xie X, Kuo F C, et al. Human competitiveness of genetic programming in spectrum-based fault localisation: theoretical and empirical analysis[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2017, 26(1): 4.

[138] Guo Y, Zhang X, Zheng Z. Exploring the instability of spectra based fault localization performance[C]//Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC2016). IEEE, 2016, 1: 191-196.

[139] Chen M Y, Kiciman E, Fratkin E, et al. Pinpoint: Problem determination in large, dynamic internet services[C]//Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN2002). . IEEE, 2002: 595-604.

[140] Lee H J, Naish L, Ramamohanarao K. Study of the relationship of bug consistency with respect to performance of spectra metrics[C]//Proceedings of the 2nd International Conference on International Computer Science and Information Technology (ICCSIT2009). IEEE, 2009: 501-508.

[141] Wong W E, Qi Y, Zhao L, et al. Effective fault localization using code coverage[C]//Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC2007). IEEE, 2007, 1: 449-456.

[142] Huang Y, Wu J, Feng Y, et al. An empirical study on clustering for isolating bugs in fault localization[C]//Proceedings of the 2013 International Symposium on Software Reliability Engineering Workshops (ISSREW2013). IEEE, 2013: 138-143.

攻读博士学位期间取得的研究成果

1. 学术成果

[1] **Xiao-Yi Zhang**, Zheng Zheng, Kai-Yuan Cai. Exploring the usefulness of unlabelled test cases in software fault localization[J]. Journal of Systems and Software, 2018, 136: 278-290. (**SCI**: 000418979100017, IF(2016): 2.444, JCR: Q1; EI: 20173204031173)

[2] **Xiao-Yi Zhang**, Zheng Zheng, Kai-Yuan Cai. A Fortification Model for Decentralized Supply Systems and Its Solution Algorithms[J]. IEEE Transactions on Reliability, 2018, 67(1): 381-400. (**SCI**: 000426678500031, IF(2016): 2.79, JCR: Q1; EI: 20174804476810)

[3] **Xiao-Yi Zhang**, Zheng Zheng, Yue-Ni Zhu, Kai-Yuan Cai. Protection issues for supply systems involving random attacks[J]. Computers & Operations Research, 2014, 43: 137-156. (**SCI**: 000329383300014, IF(2016): 2.6, JCR: Q1; EI: 20134216864362)

[4] **Xiao-Yi Zhang**, Zheng Zheng, Shao-Hui Zhang, Wen-Bo Du. Partial interdiction median models for multi-sourcing supply systems[J]. The International Journal of Advanced Manufacturing Technology, 2016, 84(1-4): 165-181. (**SCI**: 000374403900013, IF(2016): 2.209, JCR: Q2; EI: 20153701256403)

[5] **Xiao-Yi Zhang**, Dave Towey, Tsong Yueh Chen, Zheng Zheng, Kai-Yuan Cai. Using partition information to prioritize test cases for fault localization[C]//In the Proceedings the 39th Computer Software and Applications Conference (COMPSAC2015), Taichung, China: IEEE Press, 2015, 2: 121-126. (**EI**: 20161502208027)

[6] **Xiao-Yi Zhang**, Dave Towey, Tsong Yueh Chen, Zheng Zheng, Kai-Yuan Cai. A random and coverage-based approach for fault localization prioritization[C]//In the Proceedings of the 28th Chinese Control and Decision Conference (CCDC2016), Yinchuan, China: IEEE Press, 2016: 3354-3361. (**EI**: 20163602765944)

[7] **Xiao-Yi Zhang**, Zheng Zheng, Yue-Ni Zhu, Kai-Yuan Cai. A Distributed Protective Approach for Multiechelon Supply Systems[C]//In the Proceedings the 37th Computer Software and Applications Conference Workshops (COMPSACW2013), Kyoto, Japan: IEEE Press, 2013: 621-626. (**EI**: 20134316879209)

[8] **张逍怡**, 郑征, 朱悦妮, 蔡开元. 基于分级网络的供求系统防护资源分配方法 [J]. 系

统工程与电子技术, 2014, 36(10): 1982-1993. (**EI**: 20144400136034)
(**Xiao-Yi Zhang**, Zheng Zheng, Yue-Ni Zhu, Kai-Yuan Cai. Hierarchical-network based fortification resources allocation approach for supply systems[J]. Systems Engineering and Electronics, 2014, 36(10): 1982-1993)

[9] Yuan-Chi Guo, **Xiao-Yi Zhang**, Zheng Zheng. Exploring the instability of spectra based fault localization performance[C]//In the Proceedings the 40th Computer Software and Applications Conference (COMPSAC2015), Atlanta, Georgia, USA: IEEE Press, 1: 191-196. (**EI**: 20163902831992)

[10] Zheng Zheng,**Xiao-Yi Zhang**, Wei Liu, Wenlong Zhu, Peng Hao. Adapting Real-time Path Planning to Threat Information Sharing[C]. In the Proceedings of the 7th International Conference on Intelligent Systems and Knowledge Engineering (ISKE2012), Beijing, China: Springer Verlag, December, 2012: 317–329 (**EI**: 20134917050928)

[11] Zheng Zheng, Yang Liu, **Xiao-Yi Zhang**. The more obstacle information sharing, the more effective real-time path planning?[J]. Knowledge-Based Systems, 2016, 114: 36-46. (**SCI**: 000389396800004, IF(2016): 4.529 , JCR: Q1; EI: 20164603022992)

[12] Yue-Ni Zhu, Zheng Zheng, **Xiao-Yi Zhang**, Kai-Yuan Cai. The r-interdiction median problem with probabilistic protection and its solution algorithm[J]. Computers & Operations Research, 2013, 40(1): 451-462. (**SCI**: 000329383300014, IF(2016): 2.6, JCR: Q1; EI: 20134216864362)

[13] Zheng Zheng, Ze Guo, Yue-Ni Zhu, **Xiao-Yi Zhang**. A critical chains based distributed multi-project scheduling approach[J]. Neurocomputing, 2014, 143: 282-293. (**SCI**: 000340982800028, IF(2016): 3.317, JCR: Q1; EI: 20143118007272)

[14] 朱悦妮, 郑征, **张逍怡**, 蔡开元. 关键基础设施防护主从对策模型及其求解算法 [J]. 系统工程理论与实践, 2014, 34(6): 1557-1565. (**EI**: 20143118012280)
(Yue-Ni Zhu, Zheng Zheng,**Xiao-Yi Zhang**, Kai-Yuan Cai. Leader-follower hierachical decision model for critical infrastructure protection and its solving algorithm[J]. Systems Engineering Society of China, , 2014, 34(6): 1557-1565.)

2. 参与的主要科研项目

时间	项目名称	来源	本人承担的主要工作
2018.1-2018.4	面向运行环境依赖缺陷的软件自动化调试技术研究	国家自然科学基金	1, 复杂缺陷的失效机理分析 2, 复杂环境下的软件缺陷定位及相关自适应机制研究
2014.1-2017.6	基于复杂网络可控性的 GUI 软件回归测试方法研究	国家自然科学基金	1, 面向软件缺陷定位的自适应测试策略研究
2012.11-2013.11	多无人机协同控制数字仿真平台	南航无人机所	1, 多无人机协同任务分配子系统的设计与编码实现 2, 平台测试大纲的设计 3, 平台的单元测试与集成测试
2016.10-2017.9	×× 机载飞控软件优化项目研究	×× 某院	1, 机载软件测试用例优化方法的设计与平台编码 2, 机载软件测试用例优化以及可靠性分析子软件的编码与整合

致　　谢

搁笔之际，已是深夜。望着窗外的星空，Deneb、Altair、Vega，那是夏季大三角；记忆中的星空永恒 而人世间已过七载。时光如白驹过隙，转瞬间，日月轮回，沧海桑田。当初那个即将结束的夏天，来到这里。对未来的梦想，以及心中无尽的希望，永远都无法忘记。回想着，那些不为人知的故事，那些欢笑与泪水，那些抹不去的思念。谨以此感谢所有帮助过我的人以及经历过的事。

首先对我的导师蔡开元教授致以最衷心的感谢和诚挚的敬意。蔡老师渊博的学识，敏锐的洞察力，开阔的想象力，以及缜密的思辨能力让我受益匪浅，使我对研究问题的捕获以及探索能力都有了很大的提升。蔡老师在倡导学术自由、鼓励创新的同时，对强调严谨的学风。他的严格要求直接影响了我追求细节的科研态度，也激发了我的学术研究潜能。此外，其在科研中实事求是、尊重尊重学术成果的理念给我留下了深刻的印象。我想，蔡老师“科学求真，技术求实，宁静致远”的理念一定会影响我今后整个科研生涯。

感谢郑征副教授这几年的悉心指导与合作。本文的主题以及很多研究思路都是来源于和郑老师长期的交流与讨论。郑老师的在学术上、生活上以及心理上都给了我莫大的帮助，也让我切实地感到了来自师长与朋友的温暖。他严谨做事、真诚做人的风格对我整个博士生涯、我的为人理念、以及我的人生观都具有非同寻常的意义。此外，他对事情合理的规划执行力、充沛的精力、强烈的科研动力以及科研热情都激励着我在逆境中不断向前。此外，论文研究还得到了实验室其他老师的帮助和指导，在此向我的硕士导师白成刚教授、殷蓓蓓老师、全权老师和奚知宇老师表示衷心的感谢。

感谢汤新宇老师在日常事务和生活中的支持和关怀，她的帮助为我的研究带来了很大的便利。我能够在实验室安心科研与汤老师的帮助是分不开的。和她在一起，总是很放松，很开心。

感谢澳大利亚斯威本科技大学的 *Tsong Yueh Chen*教授。和他的数次讨论对我科研思路的梳理以及科研思维的培养起到了非常重要的作用。更感谢他在我澳洲访学的三个月里对我在学术上的指导，以及生活上的悉心照顾。

感谢 刘伟师兄在编程问题以及编码思维方式上对我的悉心指导，感谢张瑞峰师兄、宿敬雅师姐、李伟师兄、林树民师兄、赵立蒙师姐、郭泽、朱悦妮、宫成、王楠、高已超、刘洋、仇坤与我愉快的合作、娱乐以及沟通。感谢北航新主楼 D635 和 D608

房间所有已经毕业和正在奋斗的各位师兄弟姐妹。这些年，我们共同创造了一个单纯、温馨，却又进取、富有激情的生活与科研环境；我们共同见证了实验室一步一步的发展，从播种，到开花，最后结出今天的果实。在这个充满暖意的空间中发生的点点滴滴，我永远不会忘记。

感谢我的父母，他们赋予了我生命，让我能够在这美好的世界创造一片新的天地。他们的理解、支持和鼓励给我减轻了很多压力。真心希望我的奋斗能够使他们过上更好的生活。

感谢我生活中的朋友们，他们会在我迷茫与怀疑自我的时候坚定地和我站边。他们的鼓励以及最纯粹的支持是我在逆境中的光芒。感谢我的爱好，动漫与围棋。在遇到困难、挫折、甚至感到痛苦的时候，看一话动漫，静静地下一盘棋，让我找回那最初的美好与幸福，获得了继续追寻这世上真、善、美的勇气。感谢那些动漫里的人物，樱宁宁、五更琉璃、凉风青叶..... 你们的执着、对梦想的追寻以及在逆境中的努力给了我继续向前的信心，让我看到了真正的 Spica。

读博期间还得到了国家自然科学基金、航空科学基金、北航研究生卓越学术基金、北航研究生短期访学基金以及相关横向项目的支持，在此谨向提供上述经费的单位表示感谢。

最后，对所有参与本论文评阅和答辩的老师们表示由衷的感谢，你们的每一条意见和建议都是我宝贵的财富，对我的研究非常重要。

张逍怡

二零一八年夏于北航新主楼

作者简介

张逍怡，男，1988 年 1 月 7 日出生于北京市东城区，汉族。

2006 年 9 月至 2010 年 7 月就读于北京航空航天大学自动化科学与电气工程学院，并于 2010 年 7 月取得工学学士学位。

2010 年 9 月至 2011 年 7 月成为北京航空航天大学 自动化科学与电气工程学院的硕士研究生，控制科学与工程专业，导航、制导与控制专业，并于 2011 年转为博士研究生。

2011 年 9 月至今就读于北京航空航天大学自动化科学与电气工程学院，控制科学与工程学科，导航、制导与控制专业，师从蔡开元教授攻读博士学位。在此期间获得北航研究生发表优秀学术论文奖，2016 年度博士研究生卓越学术基金等。目前主要研究方向和兴趣包括软件测试及缺陷定位、供应链系统的防护决策与优化、无人机任务分配及航路规划等。