北京大学

博士研究生学位论文

题目： 针对自动生成补丁的审阅提升技术

姓　　名：梁晶晶
学　　号：1601111285
院　　系：信息科学技术学院
专　　业：计算机软件与理论
研究方向：程序设计模型及语言
导　　师：熊英飞 副教授

二〇二一 年 十二 月

版权声明

摘要

软件是信息基础设施最基本的构成要素。软件缺陷引发的故障会导致不可预期的严重后果。实际开发中，开发者需要花费大量时间和精力来调试和修复程序中的缺陷。随着软件规模的迅速增长和功能日益复杂，保证软件可靠性的成本也急剧上升。因此，程序自动修复技术对于降低软件缺陷修复成本，保证软件质量具有十分重要的意义。

目前主流的程序自动修复技术通常采用“生成-验证”模型，即在补丁生成之后使用规约验证补丁正确性。然而由于实际修复场景中规约不完备，修复技术本质上无法保证补丁的正确性，人工审阅补丁不可避免。现有研究表明，由于审阅过程的困难，无正确性保障的修复技术不一定能提升开发者的修复效率与正确率，甚至可能导致降低效率与正确率，使得程序修复技术难以在实际中发挥应有的作用。

为辅助开发者在提供候选补丁时高效准确修复缺陷，提升自动修复技术的实用价值，本文提出针对自动生成补丁的审阅提升技术。该技术面临的**主要挑战**是：**如何在审阅阶段帮助开发者？**目前尚未存在对补丁审阅方法的相关研究。针对该挑战，本文提出两种方案辅助开发者审阅补丁：*(1) 过滤无效信息*。在人工审阅补丁之前，为开发者自动过滤部分错误补丁，可以减轻开发者的审阅负担。已有自动化补丁过滤技术过滤错误补丁的数量有限，无法准确为开发者过滤较多错误补丁。*（2）引导开发者关注关键信息*。辅助审阅工具需要识别可以判别补丁正确性的关键信息，并引导开发者关注该关键信息，目前尚未存在相关研究。

具体而言，为探究如何辅助开发者审阅补丁，本文首先进行了补丁审阅实证研究。基于实证研究结果，本文提出了一套**针对自动生成补丁的审阅提升技术**，具体包括两个重要组成部分：（1）基于动静态信息结合的补丁过滤技术。该技术结合历史补丁的静态修改特征和动态覆盖特征，自动过滤掉明显错误的补丁，减轻人工审阅补丁的负担；(2) 交互式补丁过滤技术。该技术根据补丁的静态特性和动态运行时特性向开发者提问，回答这些问题一方面过滤掉错误补丁，另一方面引导开发者理解缺陷，完成修复过程。实验表明，本文提出的技术可以有效辅助开发者审阅补丁，提升修复效率和正确率。综上，本文主要的工作和创新如下：

补丁审阅实证研究。提出了一个通用的补丁解释模型，其中包含五个元素，可用于辅助开发者审阅补丁。实证研究中的定量实验表明，可同时采用多个补丁特性对补丁正确性进行判别。实证研究的发现不仅为本文的补丁审阅提升技术提供了指导，还具有对补丁解释生成，衡量缺陷报告质量等其它科学研究方向的参考价值。

基于动静态信息结合的补丁过滤技术（*Ceres*）。提出了一种同时基于静态修改信息

与动态覆盖变化信息的自动补丁过滤技术。在人工审阅补丁之前自动过滤掉部分错误补丁，减轻人工审阅的负担。*Ceres* 保留了补丁以代码节点为粒度的修改信息、修改位置的上下文信息及整个函数的代码结构信息，采用 S-Transformer 模型对补丁静态修改特征进行编码学习；采用 RAT 模型对补丁修改所导致的测试覆盖变化进行学习，同时结合静态修改特征学习结果对补丁正确性进行判别。本文首次在跨补丁和跨缺陷划分数据集两种场景下验证基于历史数据学习的补丁过滤技术的效果，实验结果表明，在两种场景下，*Ceres* 过滤错误补丁的数量均多于已有方法。在跨补丁划分场景下，*Ceres* 相对于已有技术分别多过滤 12.9% 和 34.6% 的错误补丁。

交互式补丁过滤技术（*InPaFer*）。提出了一种与用户交互过滤错误补丁从而辅助审阅的技术。针对无法被自动过滤技术判别的补丁，*InPaFer* 通过询问与补丁特性相关的问题与用户交互，引导用户关注关键补丁特性，从而有效辅助用户理解补丁及缺陷并过滤错误补丁。为提高交互效率，减少提问次数，*InPaFer* 定义了询问选择问题。并证明了询问选择问题与最优决策树构建问题在多项式时间内可规约，从而引入决策树中最小化最大分支算法作为询问策略，使得期望询问次数最少。本文将 *InPaFer* 实现为一个 Eclipse 插件，并设计了包含过滤视图和差异视图的用户交互界面，方便用户在交互过程中审阅补丁，完成修复。用户实验结果表明，使用 *InPaFer* 修复缺陷的开发者的修复效果（修复准确率和效率）显著优于另外三组基准调试场景下开发者的修复效果，其中，相比于没有任何辅助下开发者自己完成修复，准确率提升了 62.5%，修复时间减少了 25.3%。该实验结果说明：在合适的工具支持下，补丁审阅过程可以帮助开发者理解缺陷，提升修复效率和正确率。

关键词：程序修复，补丁审阅，补丁过滤，修复效率与准确率

Improving the Review Process of Automatically Generated Patches

Jingjing Liang (Computer Software and Theory)
Directed by Prof. Yingfei Xiong

ABSTRACT

Software is the basic component of the information infrastructure. However, software defects can lead to unexpected serious consequences. Eliminating the defects in software (i.e., debugging) can be very time consuming, especially for complex software systems in a large scale. To reduce manual debugging efforts, automated program repair (APR) has been proposed to automatically generate patches for buggy programs with minimal human intervention.

Modern APR techniques often follow a “generate&validate” procedure, which first generates patches and then validates the generated patches according to specifications. However, due to the insufficiency of specifications in practice, the patches satisfying specifications cannot always be guaranteed as correct. All generated patches need to be manually reviewed by developers. When given the candidate patches with unknown correctness, as the developers need to understand the correctness of the patches and compare the differences of multiple patches, their repair performance (i.e., debugging time and success rates) will not necessarily be improved or even decreased. Automated repair technique is difficult to play an important role in practice.

To help developers to efficiently and accurately repair bugs when providing automatically generated patches and improve the practical value of APR, this thesis proposes a technique to improve the review process of automatically generated patches. This technique faces a **key challenge**: how to help developers in patch review? There is no previous research about supporting patch review. We have two ideas to assist developers in patch review: *(1) filtering the invalid information.* Specifically, how to filter as many incorrect patches as possible in high precision to reduce the number of patches reviewed by developers? （2） *guiding developers to focus on the key information.* How to guide developers to focus on the key information? Which information is the key information for different bugs and patches, to help developers understand the bugs and patches?

To investigate how to help developers review patches, we first conduct an empirical study

about the patch review. Based on the findings of this study, we propose **a technique to improve the review process of automatically generated patches**, which consists of two stages: (1) coverage aware neural discriminator with semantics for patches. This work automatically assesses the correctness of patches by combining the static repair features and the dynamic coverage features of the historical patches, which improves the recall of filtering incorrect patches and reduces the burden of manually reviewing patches. (2) interactive patch filtering as debugging aid. This work interactive with developers by providing questions about the static features and dynamic run time features of patches. In the interaction, this technique can help developers understand patches and filter incorrect patches. The evaluation shows that the supporting technique can effectively filter incorrect patches, assist developers in patch review, and finally improve the repair performance (i.e., debugging time and success rates). In summary, the contributions of this work are presented as follows:

An empirical study about patch review. This thesis proposes a general patch explanation model, consisting of five elements, which can assist developers in patch review. The empirical study also conducts a quantitative experiment and finds that multiple characteristics of patches can be used together to assess the correctness of patches. The findings of this study not only guide the patch review supporting technique, but also are valuable for other researches, such as patch explanation generation, the measurement of the patch quality and so on.

Coverage aware neural discriminator with semantics for patches (*Ceres*). This thesis proposes an automated patch filtering technique based on the static modification and the dynamic change of coverage information. Before manually reviewing the patch, we can automatically filter out some incorrect patches to reduce the burden of developers. *Ceres* utilizes the modification of code node granularity, the context of modification, and the code structure of modified method, and leverages the S-Transformer network to learn essential static features; *Ceres* leverages the RAT network to learn the change of coverage information caused by patches and combines the results of S-Transformer network to assess the correctness of patches. This work evaluates the patch assessment approaches based on historical data in two scenarios(i.e., partitioning data sets cross patch and cross bug) for the first time. The experimental results show that the recall of *Ceres* is better than existing techniques in two scenarios. Specifically, in cross patch scenario *Ceres* outperforms existing approaches by 12.9% and 34.6% in recall, respectively.

Interactive patch filtering as debugging aid (*InPaFer*). This thesis proposes a technique assisting developers to review patches by interacting with developers and filtering incorrect

patches. For the patches that cannot be automated filtered out, *InPaFer* interacts with users by asking questions about the features of patches, and guiding users to focus on the key feature of patches so as to help users understand patches and filter out incorrect patches. In order to improve the efficiency of interaction and reduce the number of questions, *InPaFer* defines the question selection problem and theoretically proves that the question selection problem can be reduced to the construction of the optimal decision tree in polynomial time. Therefore, the minimum-maximum branch algorithm in the decision tree is introduced as the query strategy to minimize the expected number of questions. This work implements *InPaFer* as an Eclipse plugin project and designs a graphic user interface consisting of filter view and diff view to facilitate users to review patches. The results of the user study show that compared with three debugging scenarios of baselines, the debugging time and success rates of users using *InPaFer* are significantly improved. Specifically, our approach improves the repair performance of developers with 62.5% more successfully repaired bugs and 25.3% less debugging time on average. The results show that with proper tool support, the patch review process helps developers to understand the bug, and improves the repair performance.

KEY WORDS: Automated program repair, Patch review, Patch filtering, Efficiency and precision

目录

第一章　引言

1.1　研究背景

1.1.1　软件缺陷及修复问题

软件缺陷（software defects）是指计算机软件或程序中存在的某种破坏正常运行能力的问题、错误，或者隐藏的功能缺陷①。在软件开发过程中，由于程序功能复杂性或者开发人员的疏忽，软件缺陷不可避免的会被引入。而缺陷可能会带来巨大的经济损失，甚至会危及人的生命安全。例如，Ariane 5 号无人卫星，由于系统软件中的数值溢出缺陷，在发射 36 秒后偏离轨道，被引爆自毁，造成约 5 亿美元的损失[1]。海湾战争中，爱国者导弹防御系统由于一个软件缺陷未能侦测到对一处军营的攻击，造成 28 名士兵丧生[2]。因此，在软件开发过程中，及时发现并且修复程序中的缺陷占有重要地位。

然而，缺陷修复是一件十分困难并且耗时的工作。近年来，随着计算机技术的飞速发展，软件规模变得越来越大，软件维护也越来越困难。其中，缺陷修复是软件维护中的一项重要内容。研究表明，缺陷修复活动通常占软件产品的全部开发成本的 50% 左右[3,4]。开发人员平均需要 28 天来修复一个安全缺陷[5]，并且缺陷产生速度远快于缺陷修复速度[6]。2013 年剑桥大学的一项研究表明，全球每年大概要花费 3120 亿美元在软件调试上。因此，研究自动化程序修复技术对于提升软件开发效率降低软件维护成本具有十分重要的意义。

1.1.2　程序缺陷自动修复技术

为了降低软件缺陷修复成本，保证软件质量，学术界致力于研究缺陷自动修复技术（automated program repair）。这项技术的核心思想是：自动化生成一个代码片段使得程序能够正常执行。该代码片段可以在通过开发人员验证或被适当修改之后应用。从而实现减轻缺陷修复工作量的目标，缓解开发者缺陷修复的压力。

目前主流的自动化缺陷修复技术通常采用“生成-验证”模型（如图1.1所示），其输入是一个带有缺陷的程序及该程序的规约。一种常见规约是单元测试集合，并且至少有一个测试用例该程序无法通过。缺陷修复技术的输出是补丁。应用该补丁可使原程序满足给定的程序规约，即通过全部测试用例。除了测试用例集合，程序规约还可以是描述程序行为的文档，形式化逻辑规则等等。本文介绍的自动修复技术中的规约均指测试用例集合。具体修复步骤包含以下三个阶段：

---

① 在本文中，defect、bug 和 fault 均指缺陷，不作区分。

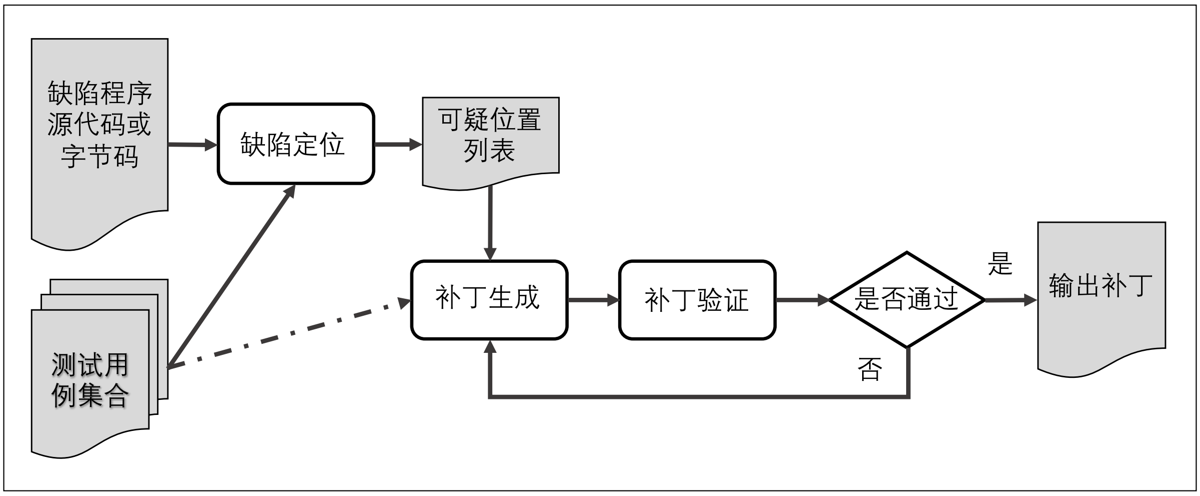


图 1.1 基于“生成-验证”模型的自动修复技术流程图

- 缺陷定位阶段 定位出错位置，一般采用缺陷自动定位技术，根据缺陷程序和测试结果获取一个可疑位置列表，作为可能的错误位置。
- 补丁生成阶段 依次在出错位置采用生成策略生成补丁，该阶段是自动修复技术的核心，具体补丁生成策略见第1.2.1节。
- 补丁验证阶段 根据程序规约过滤候选补丁，即过滤掉无法通过全部测试用例的补丁。

自 2009 年，自动化缺陷修复技术 GenProg[7–9] 被提出以来，自动化缺陷修复技术逐渐被学术界关注，并成为软件工程和程序语言领域的研究热点。随着众多修复技术被提出[10–31]，自动修复技术已经取得了很大的进步。目前，缺陷自动修复技术能为一些对于开发者也比较困难的缺陷生成补丁[24], 并且生成的质量也逐渐接近人工修复质量，最高的修复准确率可以达到 70% 以上[32]. 同时，研究者发现缺陷自动修复技术还可以提升缺陷定位技术的效果[33]。除此之外，自动修复技术在工业界也得到了一些应用，例如，Facebook 公司开发的自动修复工具 GetaFix[34] 通过静态分析技术修复程序中潜在的空指针异常缺陷，并且已经被部署到工业化的生产环境中。另外，该公司开发的 SapFix[35] 能够修复千万行代码级别的商业项目中存在的缺陷。由此可见，自动修复技术在数十年来取得了显著成就，缓解了开发者修复缺陷的压力。

1.1.3 尚待解决的问题

经过了数十年的研究，自动修复技术虽然取得了显著的成就，但是距离实现辅助开发人员高效修复缺陷，甚至在修复过程中完全解放开发人员的目标还有一些距离。其中，审阅开销问题是阻碍自动修复技术在实际中广泛应用的主要原因。

如前文介绍，目前的修复技术采用“生成-验证”模型。可以通过程序规约（通

常是一个测试用例集合）的补丁就会被返回给开发者，这种补丁也被称为*似真补丁*（*plausible patch*），而一个补丁修复了缺陷且不对程序产生其它影响才会被判定为是正确的。Long 等人[36] 研究发现，由于规约的不完备，真实修复中大部分似真补丁都是错误补丁，即，即使通过全部测试用例也可能依然无法正确修复缺陷程序，甚至是删除程序原有功能，这通常被称为*过拟合问题*[37–39]。大多数自动程序修复技术都存在过拟合问题。事实上，由于实际修复中不存在完备规约，修复技术本质上无法保证补丁的正确性，人工审阅补丁不可避免。

然而，审阅补丁对于开发者来说十分具有挑战性。这是因为：*（1）开发者在审阅过程中需要理解补丁与代码上下文信息，并判断补丁的正确性*。例如，实证实验[40] 中的开发者表示，自动修复工具生成的一些补丁会令人困惑，甚至会产生误导。对于不熟悉项目的开发者来说，提供补丁无法带来作用，开发者还是更倾向于理解程序和人工修复。*(2) 当向开发者提供多个候选补丁时，开发者需要审阅较多补丁，比较候选补丁的不同，判断每个补丁的正确性*。例如，已有实证研究[41] 观察到当提供给开发者多个补丁时，开发者需要先后浏览不同的补丁，并专门地比较补丁。这些会给开发者带来额外的负担，甚至影响到开发者的修复效率与正确率。事实上，已有研究表明[40,41]，**向开发者提供正确性未知的补丁时，尤其是提供错误补丁或多个候选补丁，开发者的修复效率和正确率并不一定会被提升，甚至会下降**。为了缓解开发者审阅补丁的压力，目前自动修复技术采取了很多保守策略提升生成补丁的准确率①。例如，为每个缺陷返回最可能正确的一个补丁[42–46]，一些修复方法只针对特定类型的缺陷：频繁出现的缺陷[47,48]、具有可参考实现的缺陷[49,50]、在之前版本上出现过的缺陷[51]。然而，这些策略不可避免地会影响生成补丁的召回率②，影响了修复技术为开发者提供帮助的程度。表1.1列举了目前准确率和召回率较高的修复工具在广泛使用的数据集 Defects4J[52] 上的修复结果。从该表中可以看出目前修复技术均无法保证准确率，且召回率较低。鉴于以上事实，如果可以放宽对于修复准确率的要求，将会为程序自动修复领域带来更多的可能性。例如，合并不同修复工具生成的补丁。这种方式下，对于任意一个缺陷，如果合并之后的任意一个技术可以为该缺陷产生正确补丁，那么这个缺陷就会被合并技术修复。未来修复工具可以采用更激进的策略，尽可能多的生成补丁。这样，单一修复技术的召回率也会被提升。

综上，人工审阅自动生成的补丁在实际修复中不可或缺。在人工审阅过程中，开发者需要理解补丁正确性，甚至比较较多补丁，这些都会影响开发者的修复效率与正确率，使得自动修复技术难以发挥预期作用，目前亟需相关工作降低开发者审阅补丁的负担。

---

① 该准确率是修复技术为其生成补丁的所有缺陷中，修复技术为其生成正确补丁的缺陷所占比例。
② 该召回率是指所有缺陷中，修复技术为其生成正确补丁的缺陷所占比例。

表 1.1　修复技术的准确率与召回率对比

修复技术	准确率较高			召回率较高		
	ACS[32]	FixMiner[53]	CapGen[43]	Hercules[27]	TBar[54]	Recoder[55]
准确率	78.3%	80.6%	**84.0%**	51.9%	73.0%	56.4%
召回率	5.0%	6.3%	5.9%	10.6%	12.9%	**13.4%**

1.2　相关研究现状

本节对相关工作进行介绍。补丁生成技术是自动修复方法的核心步骤，也是研究者尝试提升补丁生成准确率的关键步骤，因此本节首先介绍补丁生成的相关工作。然后，介绍现有关于提升补丁生成准确率，缓解补丁过拟合问题的相关研究，最后本节对现有研究进行总结。

第1.2.1节将介绍补丁生成的相关工作。根据生成方式的不同，补丁生成可以分为以下四类：

- *基于搜索的生成* 定义一组搜索策略在一个程序空间内尝试搜索并复用代码片段作为补丁。
- *基于模板的生成* 定义一组模板，在生成过程中套用模板。
- *基于语义的生成* 将程序的输入输出转为约束条件，并采用约束求解器来生成满足约束条件的代码作为补丁。
- *基于统计学习的生成* 根据开源项目或者本项目内的代码学习一个模型，用来指导补丁生成。

第1.2.2节将介绍缓解补丁过拟合问题的相关研究，现有研究主要以下三种方式来解决补丁过拟合问题：

- *基于启发式规则* 定义一些启发式规则来判断补丁正确性或者选择生成正确补丁需要的代码原料。
- *基于增强规约* 引入其他形式的规约来判断补丁正确性。
- *基于统计学习* 基于历史数据和统计学习模型来判断补丁正确性或者选择生成正确补丁需要的代码原料。

1.2.1　补丁生成技术

1.2.1.1　基于搜索的生成

在缺陷自动修复技术领域，最早被提出的补丁生成方法是基于启发式搜索的[7]，基于搜索的补丁生成技术的核心思想是程序中存在相似甚至重复的代码片段，通过定义一组搜索策略，可以在已有代码中找到并且复用这些代码片段。这些搜索算法一般是

基于启发式规则，比如遗传算法[7–9,56]，随机搜索[57]，代码相似性搜索[27,58–61]。搜索到的候选代码片段可以通过在抽象语法树（AST）中删除，修改和插入节点，或者修改程序语句中单独的操作符和变量等转换复用到修改位置。

GenProg[7–9] 是自动修复领域的开创性工作，GenProg 首先将源代码转换成 AST 表示形式，采用遗传算法搜索程序中的代码片段。具体而言，在每次迭代中，GenProg 首先确定可能的缺陷位置，采用变异和交叉两种操作来生成新的补丁，变异是指在候选补丁上插入一条新的语句或者删除一条已有的语句。交叉是指随机交换两个候选补丁的两条语句，生成新的候选补丁。为了提升搜索效率，GenProg 定义了一个适应度函数，该函数根据通过和失败测试用例的数量来衡量每个候选补丁的质量，即应用候选补丁之后的程序具有越多通过测试用例越少失败测试用例，该补丁被保留的概率越大。

Marriagent[62] 是一种类似于 GenProg 的技术，但使用了不同的交叉算法。由于随机或基于适应度选择个体进行交叉会导致选择相似的个体，产生相似的后代，从而只能略微提高候选补丁的总体质量，因此，Marriagent 从有利于多样性的角度来选择个体进行交叉。在实践中，它通过考虑每个程序相对于原始程序的共同更改和总体更改，来衡量一对程序之间的差异。选择一对程序进行交叉运算的概率取决于这两个程序之间的多样性。

GenProg 和 Marriagent 采用遗传算法搜索对补丁空间进行搜索，此搜索空间通常很大且难以遍历。AE[56] 利用语义等价检查来缩小搜索空间，节省运行时间。语义等价检查包括检测语法上不同但语义相同的候选补丁，然后在不运行测试用例的情况下丢弃多个候选补丁，但这通常代价很高。AE 主要从三方面来判断两个候选补丁是否是语义等价的：（1）语法相等，仅因存在重复的变量名或重复的语句而有所不同的程序；（2）死代码，仅因存在一些死代码而有所不同的程序；（3）指令调度，程序在某些相邻指令的顺序上有所不同，这些指令不访问公共资源，可以在不影响程序语义的情况下对其进行重新排序。

RSRepair[57] 使用随机搜索来指导生成过程。与 GenProg 相比，RSRepair 不使用演化，因此它不会对候选补丁应用变异和交叉操作。这是基于假设：与自动程序修复领域中的演化算法相比，随机搜索的性能可能更好，因为在演化算法中，很难定义一个良好的渐进演化。在每次迭代中，RSRepair 都会简单地生成一个候选补丁，并对其进行验证，如果未通过所有测试案例，则将其丢弃。实验证明，随机搜索算法比遗传算法在 23/24 个缺陷修复上具有更高的修复效率。

早期的自动修复技术仅依靠是否通过测试用例判断补丁的正确性，Qi 等人[39] 人工检查了 GenProg，RSRepair 和 AE 生成的补丁发现大部分似真补丁都等价于删除操作，依据此发现设计了一个仅有删除操作的自动修复技术 Kali，可生成 3 个正确补丁。

尽管在某些情况下 Kali 可能会成功修复缺陷，但该修复技术的主要目的是通过实验研究似真但不正确补丁的问题。实际上，通常可以通过简单地删除功能来获得通过测试用例的程序，但是这些似真补丁通常不会被开发人员接受，他们通常在不损失程序功能的情况下修复缺陷程序。

基于搜索的补丁生成技术中，代码相似性这一特征经常被用来指导修复过程，依据代码相似性对候选补丁排序，提升修复技术的准确率或者修复效率。相似性既可以是根据候选补丁与历史修复补丁的相似性[58,59]，也可以是候选补丁与待修复位置的相似性[61]，或是同时考虑这两种信息[27,60]。

HDRepair[58] 基于历史修复信息自动修复缺陷。它基于假设：反复出现的缺陷修复在应用程序中很常见，并且历史修复模式可以为自动修复技术提供有用的指导。HDRepair 从开源项目的历史修复中挖掘频繁出现或者常见的修复模式，然后采用一些现有的变异算子生成候选补丁，与 GenProg 直接对候选补丁进行验证不同，HDRepair 根据与挖掘到的历史修复模式的相似性对与候选补丁排序，并优先验证相似度更高的候选补丁。采用这种方式，可以更快的修复一个缺陷，实验表明，HDRepair 平均修复一个缺陷的时间在 20 分钟左右。

ssFix[59] 将代码片段看做纯文本，采用 TF-IDF 模型对可疑位置的代码片段与已有的代码仓库中的代码片段进行相似度匹配，生成候选补丁。实验表明，ssFix 可以成功修复 20 个 Defects4J 中的缺陷。

CapGen[61] 是一个上下文敏感的修复技术。CapGen 通过细粒度代码搜索确保生成正确补丁，根据上下文信息对变异算子选择顺序进行优化来约束搜索空间，最后将所有候选补丁根据自定义的启发式规则进行排序。该方法在成功修复了 21 个 Defects4J 的缺陷，修复准确率达到了 84%。是 2020 年及以前方法中在 Defects4J 数据集上准确率最高的修复技术。

SimFix[60] 同时考虑历史修复信息和修复程序的上下文信息。具体而言，SimFix 从开源软件的历史修复信息挖掘频繁修改模式，构成一个候选补丁空间，并且在缺陷程序内部，搜索与缺陷程序抽象语法树相似的代码片段，构成另外一个候选补丁空间。这两个空间的交集即为最终补丁空间。该方法可以修复 34 个 Defects4J 缺陷。

HERCULE[27] 利用缺陷程序内代码相似性和缺陷程序的历史演化版本两个信息实现对多位置缺陷进行修复。针对一个缺陷位置，HERCULE 进行语法和语义相似性分析找到程序内相似的缺陷位置，同时 HERCULE 还会考虑历史演化版本中多个缺陷位置是否进行了相似的演化，进一步确定找到真正的需要同时修复的位置，然后针对这些缺陷位置依次进行修复。该方法可成功修复 46 个 Defects4J 的缺陷，在 2020 年及之前的已有方法中，HERCULE 达到了在 Defects4J 数据集上最高的召回率。

1.2.1.2 基于模板的生成

基于模板的生成一般是定义一组模板，在生成补丁过程中套用模板。这个模板可以是预定义的[63–65]，也可以是基于历史数据手工提取的[54,66,67]。

SPR[63] 采用预定义模板的方式修复条件类型的缺陷。首先，SPR 预定义了一组通用转换模式 (transformation schema) 作为模板，并且应用于缺陷程序，这些模板是参数化的，每个模板代表一类程序转换。然后，对于每个转换后的程序，SPR 确定可以使修复成功的参数值（如果存在）。由于模板通常需要一个条件作为参数，因此，SPR 尝试合成一个条件使得其满足确定的参数值。如果该过程成功，则将合成条件插入程序中，并最终确定修复结果。SPR 中定义的参数模板可能会修改条件（向现有条件添加子句并生成新条件）和修改变量值（替换变量名和常量值），也可能会从源程序的其他地方复制语句。通过这种先定义模板再确定参数的分阶段修复方式，可以快速跳过许多错误的修复，并专注于最可能正确的修复。

基于预定义模板对于特定类型的缺陷的修复更具有针对性，比如空指针异常[65]，崩溃[64] 以及静态分析违反规约的缺陷[68,69]。Durieux 等人[65] 预定义了 9 种修复模板来解决空指针异常问题。这 9 种模板可以分为两类：使用一个对象来替换异常抛出和跳过异常抛出的代码片段，并且采用执行时动态分析程序填充模板，相比于静态分析程序模板填充，该方法可以探索更多的潜在补丁。AVATAR[69] 是一个针对静态分析工具检测出的缺陷的修复技术，不同于已有方法从所有开发者提交的补丁中提取修复模板，AVATAR 只提取同样修复静态分析工具检测出的缺陷的补丁中的模板，实验表明，AVATAR 可以修复 34 个 Defects4J[52] 数据集中的缺陷。

操作符变异操作也可以看做一种预定义模板。PraPR[70] 在 JVM 字节码上定义了一系列变异操作获取补丁，包括传统变异测试中基础的变异操作和频繁出现在缺陷修复中的一些变异操作，例如，替换域访问和函数调用。一方面，在字节码上修改避免了源码级别的编译开销，另外，字节码修复可以实现跨语言的修复，可针对任意基于 JVM 的语言（JAVA，Kotlin）程序上的缺陷进行修复。实验表明，PraPR 可成功修复 43 个 Defects4J[52] 数据集中的缺陷，并且相对于已有的修复方法，修复速度可提升十倍。

Kim 等人人工分析了超过 60000 个 Eclipse 上真实修复的补丁并定义了 10 个典型的模板，提出了修复技术 PAR[66]。它的假设是，从真实的补丁中提取的模板会比其他模板更有效和更被接受。PAR 将人工提取的模板编码为一系列 AST 重写规则，采用遗传算法，将重写规则应用于缺陷定位技术分析的可疑位置，生成候选补丁，直到成功修复程序。这种基于人工定义的模板可能具有复杂的修改格式，影响程序的一个或多个语句，无法通过简单的变异修改及其组合获取。实验结果表明，在包含 119 个缺陷的数据集上 PAR 可修复 27 个缺陷，高于 GenProg 的修复个数（16 个）。并且在人工实

验中，实验参与者认为 PAR 生成的部分补丁比原始的人工生成的补丁更容易被接受。

Liu 等人调研了当前基于模板的 17 篇缺陷修复论文，总结了前人研究中基于模板的修复方法，并提出了 TBar[54]，一个集成了 15 个类别下 35 个修复模板的修复技术，截止 2020 年为止，TBar 是集成最多修复模板的技术，具体修复模板参见表1.2。实验验证 TBar 可成功修复 43 个 Defects4J[52] 数据集中的缺陷，该研究说明基于模板的缺陷修复方法的有效性。随后他们还基于 16 个开源 Java 程序修复工具进行了大规模实证研究[71]，实验结果表明相比于其它类别的修复技术，基于模板的修复方法虽然能够修复更多的缺陷，但是也可能会产生很多无关的不正确补丁。

表 1.2　TBar 中集成的 15 类修复模板

修复模板名称	**修复模板具体描述**
插入类型转换检查	在具有强制类型转换的语句前增加 *instanceof* 检查
插入空指针检查	在被访问的对象或表达式前增加 *null* 检查
插入越界检查	在数组或集合访问前增加越界检查
插入丢失语句	在某些语句前后增加一些 *return/try-catch/if* 语句
变异类的实例生成	将一个类的实例生成替换为 *super.clone()* 语句
变异条件语句	删除子表达式或插入新的表达式
变异数据类型	将声明或类型转换时的变量类型改变
变异整除操作	将整数除法结果替换为浮点数类型
变异常量表达式	将数值或字符串常量替换为其它相关常量
变异函数调用表达式	替换调用的函数名
	至少替换一个函数参数
	移除一个函数参数
	插入一个函数参数
变异操作符	替换成同类别内的其他操作符
	改变数值操作符的优先级
	将等值操作替换为 *instanceof*
变异返回值语句	将返回值语句中的表达式替换为其它表示式或变量
变异变量	将缺陷语句中的一个变量替换为其它可编译的变量
移动语句	将一个缺陷语句移动到程序的其它位置
删除缺陷语句	从程序中删除整个缺陷语句

1.2.1.3　基于语义的生成

基于语义的程序修复技术是将程序的输入输出转为约束条件，并采用约束求解器来生成满足约束条件的代码作为补丁，主要分为三步：(1) 行为分析，分析待修复程序并且生成该程序的正确行为和错误行为的语义信息。为此，行为分析可以利用程序的

源代码和测试用例集合，通过符号执行[16,72–74] 或者程序的动态执行信息[75] 生成语义信息。(2) 问题生成，利用行为分析收集的信息，形式化表示待修复的程序问题，它的解就是实际修复的补丁。(3) 补丁生成，尝试解决上一步生成的问题，或者生成一个满足条件的解，或者无法找到满足条件的解，或在有限时间内无法找到。补丁生成通常采用预言指导的基于组件的程序综合技术[76] 来生成补丁，即被编码为一阶逻辑约束问题，并使用可满足性模理论（SMT）求解器求解。

SemFix[72] 是基于语义自动修复缺陷的开创性工作，修复单个语句缺陷。它首先使用 SBFL 对程序进行缺陷定位，根据定位结果，从可疑度最高的语句开始执行修改操作。每次修改一条语句，SemFix 会通过修改一个分支语句的谓词或者赋值语句的右半边来生成修复。SemFix 用一个符号表达式替换要被修复的程序中的表达式，该符号表达式要么表示一个通用条件，要么表示一个通用的变量赋值。然后执行测试用例，其中具体执行未被修改的部分，符号执行被替换的部分。执行的结果会产生关于符号表达式的一组约束，这组约束对符号表达式必须满足的条件进行编码，使得程序能通过所有的测试用例。根据这组约束跟一组基本组件，通过程序综合问题来生成具体的代码。为了使生成的表达式尽可能简单，用于综合算法的基本组件可以分为多个级别，并逐级提供给综合算法。该方法由于需要对全部程序执行符号执行，因此只适用于规模较小的程序。

DirectFix[73] 采用了类似 SemFix 的修复技术，在 SemFix 的基础之上，降低了修复表达式的复杂性，提升了可接受性。DirectFix 的优化主要是将缺陷定位过程和补丁生成过程结合到了一起，并不是严格的按照定位结果的先后顺序尝试修复，而是首先选择比较简单的语句进行修复，这样可以尽可能生成简单的修复补丁。其次，DirectFix 在进行符号执行的过程中全部用符号量进行表示，规定了一些必须满足的条件（硬条件：hard condition）和尽可能满足的条件软条件：soft condition）。最终是将其转换为部分最大可满足问题（partial MaxSAT）。该问题在求解时是在满足硬条件的前提下尽可能多的满足软条件，目的同样是期望尽可能生成简洁的修复补丁。当一个错误表达式有多个修复时，基于简单的修复不会引入回归缺陷的假设，DirectFix 选择最简单的修复。

Angelix[74] 是基于 SemFix 和 DirectFix 的改进，其目的是在保持可扩展性的同时能生成多行修复，DirectFix 可以生成多行修复，但可扩展性不好，SemFix 是可扩展的，但仅限于单行修复。Angelix 利用天使路径 (angelic path) 和天使森林 (angelic forest) 的概念，在不具有可扩展性的情况下生成多行修复。天使路径将修复程序部分编码为一组三元组，每个三元组包含一个可疑表达式（通过基于 SBFL 定位的结果），它的天使值（测试通过时表达式的返回值）和天使状态（在表达式位置上可访问的一组变量）。

天使路径是使用符号执行提取的。天使森林将修复问题完全编码为一组天使路径，然后修复引擎基于天使森林生成多行修复。

以上三种方法只考虑了程序语义信息，Le 等人在考虑程序语义的同时还考虑了程序语法信息，提出了自动修复技术 S3[16]。与以上方法类似，S3 通过符号执行获取程序规约，使用 DSL 定义了一个补丁空间，不同的是，在补丁空间内搜索候选补丁之后，采用了一些特征对候选补丁排序。排序的假设是，候选补丁应与原程序在语法与语义上都具有相似性。该技术在不产生任何错误补丁的情况下，可以修复 52 个 Java 缺陷中的 22 个。

基于语义的补丁生成中，除了通过符号执行获取程序规约之外，还可以通过程序的动态执行信息来获取。采用动态执行信息的优势在于可延展性更好，可修复大型程序中的缺陷。针对条件语句缺陷修复方法 Nopol[75] 采用了天使定位（angelic localization）的方法收集程序约束。天使定位是指获取测试执行中条件变量的期望值并且快速定位出错位置。其次，Nopol 收集程序变量在执行时的实际值，包括简单数据类型和面向对象数据的特性（例如，空值检查）。最终，Nopol 将收集到的约束转为 SMT 问题求解。

1.2.1.4 基于统计学习的生成

基于统计学习的生成是根据开源项目或者本项目内的代码学习一个模型，用来指导补丁生成。

Long 等人[17] 提出了一个基于机器学习的自动推断代码转换的修复技术 Genesis。Genesis 通过学习历史修改中，修改前后抽象语法树的转换，然后对缺陷程序推断其代码转换并且生成代码组件作为候选补丁。Genesis 采用了整数线性规划 (integer linear program，ILP) 模型，能够使得推断补丁的搜索空间尽可能覆盖训练数据集，同时控制补丁生成系统能高效遍历补丁空间从而找到正确补丁。实验表明，Genesis 在空指针，数组越界和强制类型转换这三类 Java 程序缺陷上，能够有效修复 21 个缺陷（共 49 个缺陷）。

ELIXIR[77] 是针对面向对象程序的修复技术。ELIXIR 定义了一个丰富的补丁搜索空间，包含函数调用，部分变量，域和常量。为了对搜索空间内的候选补丁排序，ELIXIR 提取了 4 个重要的补丁特征：（1）标识符在当前上下文中使用的频率，（2）修复位置与最后一次使用位置的距离，（3）相似名称是否出现在修复上下文中，以及（4）缺陷报告中是否出现了标识符或者部分标识符。并采用逻辑回归算法对模型进行训练和预测。实验表明，ELIXIR 成功修复了 26 个 Defects4J 的缺陷。Noda 等人[78] 将 ELIXIR 应用在一个由 150 多个 Java 项目和 13 年开发历史组成的大型工业软件上。实验结果表明：ELIXIR 仅能修复 7.7% 的缺陷，能为其生成补丁的缺陷中，仅有 10% 的正确率。另外工业化实际场景中还面临着缺乏触发缺陷的测试用例的挑战（90%）。不

过作者通过增加一些简单的模板，例如，交换缺陷代码中函数的参数，使得修复成功率提高了 40%。作者认为 APR 工具的实际应用还有很大的提升空间。

Getafix[34] 基于层次化聚类算法 (hierarchical clustering algorithm) 学习重复出现的修复模式来修复静态分析工具检测出的缺陷。首先提取历史上修复前后的代码的 AST 形式，使用 Anti-Unification 算法将所有的修复前后 AST 代码对进行聚类和抽象，抽象是为了将类似的修改被归为同一类中，例如，两个修复是替换了两个不同变量，将变量用一个通配符表示，那么这两处修改会被归为一类。将学习到的修复模板应用到新的缺陷上时，Getafix 会先将缺陷代码和模板中修复前的模式进行匹配，如果成功匹配，那么会使用该修复模式生成补丁。该方法的修复效果取决于训练集数据中的缺陷类型，一般适用于常见的缺陷类型，例如，可能的空引用，API 的错误使用，特定语言构造函数的错误使用。

DeepFix[79] 是第一个基于深度学习的缺陷修复技术。DeepFix 修复程序语法错误而非功能错误。它通过训练一个基于注意力机制的多层序列对序列神经网络，用来预测缺陷位置以及所需的正确程序语句。在包含 6971 个缺陷的 C 程序学生作业上，DeepFix 可以正确修复 1881 个（27%）缺陷。

AutoGrader[80] 结合人工提取模板和深度学习预测两种方式生成修复选项，并将缺陷程序和修复选项转为基于 SKETCH 的程序合成问题，进行补丁求解。其中，深度学习预测部分采用长短期记忆网络 (LSTM) 将缺陷位置前的正确代码序列化，并截取固定长度作为模型的输入。反复迭代产生预测。实验表明，该方法在学生作业的数据集上可修复 14/19 个缺陷。

DeepRepair[81] 采用递归神经网络推理如何为补丁原料（repair ingredients）排序和如何从代码库中学习代码转化。相比于基准方法 jGenProg，DeepRepair 能更快的找到可编译的补丁原料，但是并没有生成更多补丁。

Tufano 等人为评估使用机器翻译技术为真实缺陷生成补丁的可行性，进行了实证实验[82]。作者首先从 GitHub 上收集了百万个缺陷修复的数据集，然后使用修复前后的代码训练了神经机器翻译模型（neural machine translation, NMT）。实验结果表明，在 9%-50% 的情况下，NMT 模型能够为缺陷代码预测出补丁，并且超过 82% 的候选补丁是语法正确的。此外，给定缺陷程序，NMT 模型能够在小于一秒钟的情况下生成多个候选补丁。

CoCoNut[83] 利用集成学习 (ensemble learning) 将卷积神经网络 (convolutional neural network，CNN) 和 NMT 组合起来自动修复缺陷。CoCoNut 使用了上下文感知 NMT 模型来表示缺陷代码的上下文信息，并且使用 CNN 模型进行预测。相比于循环神经网络 (recurrent neural network，RNN)，CNN 模型可以提取分层特征，并在不同的粒度级别

（例如语句和函数）更好地表示源代码。此外，CoCoNut 利用超参数中的随机性来构建多个模型修复不同的缺陷，并使用集成学习来组合这些模型修复更多的缺陷。在包含四种编程语言的六个通用数据集上，CoCoNut 能成功修复 509 个缺陷，其中 309 个是已有方法无法修复的。

DLFix[84] 是一种基于上下文信息的代码变换修复方法。它采用了两层深度学习模型来解决已有的深度学习修复方法对补丁上下文信息表示具有局限性的问题。第一层网络是一个基于树的循环神经网络用来学习补丁的上下文信息，并将该信息作为第二层网络的输入。第二层网络是一个卷积神经网络主要学习缺陷修复的代码变换以生成补丁。实验结果表明，DLFix 的修复效果不弱于传统基于模板的修复方法。

基于统计学习方法生成补丁最初使用较为简单的逻辑回归模型，后来开始逐渐使用神经网络等复杂深度学习模型，表1.3总结了上述相关工作的名称，文献以及采用的统计学习模型。

表 1.3 基于统计学习的生成中使用到的学习模型

修复方法名称	**文献**	**学习模型**
Genesis	[17]	整数线性规划模型
ELIXIR	[77]	逻辑回归模型
Getafix	[34]	层次化聚类算法
DeepFix	[79]	递归神经网络
AutoGrader	[80]	长短期记忆网络
DeepRepair	[81]	递归神经网络
实证试验	[82]	机器翻译模型
CoCoNut	[83]	机器翻译模型 + 卷积神经网络
DLFix	[84]	卷积神经网络 + 循环神经网络

1.2.2 缓解补丁过拟合问题的技术

1.2.2.1 基于启发式规则

基于启发式规则缓解补丁过拟合的技术是定义一些规则，尽可能生成正确补丁。这些规则可以基于人工定义的模板[67] 排除有可能是错误的补丁，或者基于代码语法或者语义相似性[43,85] 对补丁排序，或者根据开源代码中出现的频率，优先返回频繁模式的补丁[32]，或者基于执行动态信息的结果排除错误补丁[86]。

在基于模板的修复技术 PAR 的启发下，Tan 等人[67] 提出了反模板 (Anti-Pattern)，即人工总结规则过滤补丁空间内错误补丁。作者提出了 7 个反模板，如表1.4所示。作者将这些反模板应用在基于搜索的补丁生成技术上，例如，GenProg 和 SPR 生成的补

丁上，显著提升了产生补丁质量和速度。尤其是 GenProg 中经常出现删除功能的错误补丁，通过将删除动作设置为反模板，该类别的非法补丁显著减少。除此之外，反模板有效地约减了搜索空间，提升了修复速率。

表 1.4 Tan 等人提出的 7 个反模板

反模板名称	**反模板具体描述**
反删除 CFG 退出语句	不删除 *return,exit,goto* 等语句
反删除控制语句	不删除 *if,switch,loop* 等语句
反删除单行语句	不删除一个代码块内只有一行的语句
反删除 *if* 周围的语句	不删除 *if* 前定义，且在 if 中使用的变量
反删除循环计数更新语句	不删除控制循环体结束的变量更新语句
反提早退出	不能在一个代码块内除最后一行外增加 *return,goto* 语句
反增加朴素条件	不能增加一个恒为 *true* 或 *false* 的条件表达式

如第1.2.1.1介绍，很多基于搜索的算法在生成补丁过程中都根据相似性信息对补丁进行排序，优先返回正确的补丁，提升修复准确率，Asad 等人[85] 同时结合了补丁与缺陷代码的语义相似性和语法相似性用于补丁的排序。在生成补丁之后，采用 AST 节点相似性和变量相似性来描述语义相似性，其中，AST 节点相似性指补丁与缺陷代码在函数范围内内不同类型节点的频数的相似度，变量相似性包括了变量名和变量类型。同时，采用词法级别的最长公共子序列度量语法相似性。该方法在一个基于学生作业的数据集（IntroClassJava[87]）上能成功修复 22 个缺陷，且正确率达到 100%。

上下文敏感的修复技术 CapGen[43] 分别对补丁原料和补丁进行排序。生成补丁过程中基于上下文相似度对补丁原料进行排序，包括：（1）补丁和缺陷代码变量相似度；（2）补丁和缺陷代码 AST 结构相似程度（包括祖先和兄弟节点）（3）补丁和缺陷代码影响到的上下文代码的相似性。生成补丁之后，CapGen 基于每个补丁的缺陷位置，变异算子的频数和补丁与缺陷代码的上下文相似程度三个维度进行排序。相比于不排序的结果，对补丁排序可以使正确补丁排在 98.78% 的不正确补丁之前。CapGen 修复准确率达到了 84%。是在 2020 年及以前方法中在 Defects4J 数据集上准确率最高的修复技术。

ACS[32] 是一个针对条件语句缺陷修复的技术。作者提出三个启发式规则对变量和谓词进行排序，即在条件表达式中应该出现哪些变量和这些变量应该具有怎样的谓词逻辑。这三个规则为：（1）基于依赖关系排序：按照变量使用的局部性原则排序，依赖关系的拓扑排序中，最近的变量更有可能被用到条件语句中；（2）文档分析：根据源代码中的 javadoc 注释，当注释中条件表达式提及某个变量时，修复该类的条件表达式仅会使用提及的变量；（3）谓词挖掘：对 GitHub 等开源网站中条件表达式进行分析，

按照表达式中谓词出现的频率排序。实验结果表明，在数据集 Defects4J 上，ACS 的修复准确率达到了 78.3%。相比之前工作[18]（jGenProg：18.5%，Nopol：14.3%）的准确率，ACS 大幅提高了修复的准确度。

熊等人[86] 基于补丁相似性和测试相似性两个观察，在不需要完整的测试预言的情况下，根据执行动态信息排除不正确的补丁。具体而言，补丁相似性是指，当应用正确补丁之后，通过测试用例的行为跟之前是相似的，而失败测试用例的行为跟之前是不同的。测试相似性是指，当两个测试用例具有相似的执行时，它们的执行结果往往是相同的。依据补丁相似性，可以通过比较缺陷程序打补丁前后，原始测试集合中，通过测试行为和失败测试行为是否相似来判断该补丁是否正确。当通过测试行为越相似，失败测试行为越不相似时，该补丁正确的可能性越大。当新生成测试时，可以根据测试相似性来确定新生成测试的结果，新生成测试执行行为跟原始测试中失败测试的执行行为越相似，那该新生成测试的结果是失败的概率越大。他们基于这两个观察并设计了一个为每个补丁正确性概率打分的公式，实验结果表明，该公式可以排除掉约 56% 的不正确补丁。

1.2.2.2 基于增强规约

增强规约的方法主要通过增加测试用例来增强现有规约[64,88–90]，或者引入其它形式的规约，例如，相似正确程序的输入输出[91]，引入相同功能代码的其他实现作为规约[92]。

DiffTGen[88] 根据缺陷程序和修复程序之间的语义差异生成测试用例，并且假设存在一个测试预言，它可以为新生成的输入提供相应的输出。如果修复后的程序产生的输出违反了测试预言提供的输出，则该补丁是不正确的。将该测试用例加入到原测试集中可以增强原测试集并且防止修复技术再次产生相似过拟合的补丁。DiffTGen 在 4 个修复工具生成的 89 个补丁的数据集上，可检测出 49.4%（39/79）的不正确补丁。在实际中，测试预言无法自动生成，需要人工提供，因此这种方法并不普适。

Fix2Fit[64] 也是通过新增加测试用例来避免补丁过拟合，与 DiffTGen 不同的是，Fix2Fit 在修复阶段集成了测试生成来检测补丁是否过拟合以及它将程序崩溃作为测试预言。Fix2Fit 采用更好的划分搜索空间内的候选补丁的测试来指导模糊测试。具体而言，在生成测试时，Fix2Fit 会首先将候选补丁根据现有测试集划分为测试等价的补丁集合，然后将能进一步划分同一等价集合内补丁的测试赋予更高的权值。实验表明，根据程序是否崩溃可以过滤掉候选补丁空间内 60% 的补丁。

Opad[89] 同样采用模糊测试的方法。相比于 Fix2Fit，除了验证候选补丁在新生成的测试上是否会导致程序崩溃之外，还考虑是否会产生内存泄露问题，是否内存泄露可以通过现有的静态分析工具 Valgrind[93] 来获得。同时，Opad 还提出了一个度量补丁

是否崩溃的公式：如果存在测试用例在缺陷程序上通过而补丁程序上失败了，那么这个补丁就是过拟合补丁。采用该方法，Opad 可以过滤 75.2% (321/427) 的不正确补丁。

Yu 等人[90] 将过拟合问题分为两类：不完全修复和引入回归缺陷，然后提出 UnsatGuided，同样使用测试用例生成增强已有的开发者测试用例集合从而缓解补丁过拟合问题的方法。为了避免自动生成错误的测试预言误导补丁生成的过程，该方法会过滤掉与已有测试预言产生矛盾的新生成测试。实验表明，UnsatGuided 可有效缓解引入回归缺陷类型的过拟合问题，并不会影响到已生成的正确补丁，另外还可以帮助 Nopol 生成额外的正确补丁。

Le 等人[94] 的实验证明，在他们的数据集上（8 个自动修复工具生成的 189 个补丁），通过增加额外的测试用例，排除五分之一左右的不正确补丁，减轻了开发者人工验证的负担，不过，没有被排除的补丁仍然无法确定是正确的。

Refactory[91] 是一个针对学生作业缺陷的实时修复方法。给定一个缺陷程序，Refactory 根据控制流图找到与之最相似的经过重构的正确程序，并且将正确程序对应代码块的“输入-输出”作为缺陷程序代码块的一个规约，将该规约作为原来测试集合的补强规约，最后采用基于搜索的修复方法生成补丁。

为了缓解基于语义的修复系统的过拟合问题，Mechtaev 等人提出了利用功能相同的参考程序作为规约输入的修复工作 SemGraft[92] 。一些库函数和经典算法有多个语义等价实现的版本，对参考程序实现执行符号分析可以自动推断出期望行为规约，可作为补丁生成的补充规约，缓解仅依靠测试用例作为规约引发的补丁过拟合问题。此外，与基于相似代码搜索的补丁生成不同，SemGraft 是基于语义分析生成补丁，修复后的程序与参考程序可能会有不同的实现方法。该技术在 GNU-Coreutils 和 Linux Busybox 上进行实验，相比于不使用参考程序推断规约的方法，SemGraft 可以生成更多正确的补丁。

1.2.2.3　基于统计学习

基于统计学习缓解补丁过拟合问题与基于统计学习的补丁生成类似，都是基于历史信息学习模型，具体地，基于统计学习缓解补丁过拟合问题主要通过学习正确补丁的特征训练模型来预测新生成的补丁是否是正确的。

Prophet[95] 是第一个采用学习算法缓解补丁过拟合问题的技术。它采用对数线性模型 (log-linear probability model) 对基于模板的修复方法 SPR[63] 生成的候选补丁进行排序，使正确的补丁优先被验证。Prophet 抽取两种类型的特征：修改特征和程序值特征。修改特征包括补丁的修改类型以及缺陷位置上下文的语句类型和补丁修改类型的关系。程序值特征是指缺陷程序在修复前后变量和常量值的使用情况。为了学习正确补丁的修复模式，Prophet 收集了一个人工提交的正确补丁集合作为训练集。实验表明，

相比于 SPR 直接验证补丁，排序之后的结果使更多缺陷的正确补丁被第一个验证。类似地，ELIXIR[77] 也是通过提取补丁特征进行学习训练对补丁排序，提升生成准确率。

Wang 等人[96] 进行了一个关于补丁正确性判断的实证研究。作者首先调研了已有的缓解补丁过拟合技术，包括使用 Evosuite[97]，Randoop[98]，DiffTGen[88] 和 Opad[89] 生成测试用例，通过 Daikon[99] 判断修复前后动态行为，Xiong 等人[86] 提出的 PatchSim 技术，反模板[67] 等七种技术。此外还选取了现有补丁生成技术中采用的八个静态代码特征，使用了包括随机森林在内的六种机器学习模型进行训练。作者有以下发现：（1）静态代码特征能够有效判断补丁正确性；（2）基于动态信息的技术准确率较高，而基于静态特征技术的召回率较高；（3）现有方法可能只对特定项目和修复工具生成的补丁有效；(4）现有技术是高度互补的。例如，单个技术最多只能检测 53.5% 的过拟合补丁，而当测试语义信息可用时，93.3% 的过拟合补丁可以被至少一种技术检测。基于以上发现，作者设计了一个集成策略，首先通过学习排序算法集成静态代码特征，然后通过多数投票原则与其他技术生成的结果集成。实验表明，该集成策略能显著提高现有补丁正确性评估技术效果。

ODS[100] 通过提取修复前后程序的静态特征并采用梯度提升算法（gradient boosting）训练补丁正确性分类模型。该工作提取了三个类别共 202 个特征，是截止至 2021 年关于缓解补丁正确性相关工作中提取最多静态特征的方法，其特征类别及描述见表1.5。其中，代码描述特征，修复模式特征，上下文语法特征，分别代表了基于不同粒度描述代码的文法特征，人工提取的补丁修复类别和缺陷语句的上下文特征。作者在 Defects4J, Bugs.jar 和 Bears 三个数据集上共超过 10000 个补丁上对 ODS 识别补丁正确性的效果进行了验证，实现了 94.7% 准确率和 70.0% 的召回率。同时，ODS 的效率也远高于采用动态信息的技术 PatchSim[86]（速度提升 *138* 倍）。

以上都是采用了人工提取特征采用机器学习方法来训练模型，Csuvik 等人[101] 采用表示学习来判断补丁正确性，即将修复前后的代码采用代码嵌入模型 Doc2vec[102] 和 Bert[103] 转为向量表示，然后根据修复前后代码的向量表示相似度来判断补丁正确性，作者认为与缺陷代码越相似的补丁越可能是正确补丁。特别地，作者探索了三种不同类型的代码表示方式：源代码，AST 形式和 AST 子树形式。实验结果表明，将源代码直接输入代码嵌入模型效果最好，可以在 QuixBugs 数据集上过滤掉 45% 的错误补丁。

Tian 等人[104] 进一步探索了多种不同基于表示学习的方法对补丁正确性判断的效果，与 Csuvik 等人[101] 类似，作者首先采用了四种代码嵌入模型，Doc2vec[102]，Bert[103]，code2vec[105] 和 CC2Vec[106]，计算修复前后代码在以上四种模型上表示结果的相似度，发现正确补丁与错误补丁分别与缺陷程序的相似度有很大不同。基于此观察，作者分别将不同的补丁表示方法采用余弦相似度和机器学习模型来判断补丁正确性。作者采

表 1.5 ODS 中提出的静态补丁特征

特征名称	**子类**	**子类描述**	**数目**
代码描述特征	操作符	是否包含某些操作符，例如关系操作符	14
	变量	变量的修饰符，例如是否为局部或者全局的	5
	语句	语句的类型，例如是否是赋值语句	21
	AST 操作	AST 操作类别，例如插入一个变量	10
修复模式特征	包裹缺陷语句	在缺陷语句上增加 *if* 或 *try/catch* 等代码	9
	表达式	修改表达式类型的代码	4
	条件	增加或删除条件语句	4
	空值检查	增加判空或者非空检查	1
	其它模式	其它修复模式，例如单行语句删除	27
上下文语法特征	类型	缺陷语句及上下文语句的类型	9
	函数	函数调用的特征	7
	相似度	是否存在相似的对象或函数调用	4
	用法	变量或域的使用及定义情况	6

用了三种机器学习模型：逻辑回归，决策树和朴素贝叶斯。实验结果表明，基于 Bert 预训练模型的表示方法和逻辑回归模型的组合过滤错误补丁的效果最好，可达到 72% 的 F1，但是该工作效果差于基于动态信息的 PatchSim[86] 和静态特征的 ODS[100]，作者认为基于表示学习的方法可以与基于静态特征的方法互补。

1.2.3 研究现状小结

本节首先介绍了程序自动修复技术中的核心步骤：补丁生成。根据生成方式的不同划分为四类：基于搜索的生成、基于模板的生成、基于语义的生成和基于统计学习的生成。根据调研可知无论是基于何种方式生成补丁，现有技术都采用了一个不完整的规约，即给定的测试用例集合，因此以上技术都无法从根本上保证补丁正确性。

接下来，本节调研了现有提升修复正确率，缓解过拟合问题的相关工作，根据缓解方式的不同可分为三类：基于启发式规则、基于增强规约和基于统计学习。从自动修复不同阶段的维度来看，大多数缓解补丁过拟合问题的技术都是在生成补丁时，研究者设计修复方法，采取一些策略，例如，定义避免过拟合的模板[67]，根据相似性规则和频繁模式选取最有可能正确的补丁或是补丁原料[18,43]，从而提升正确补丁的生成概率，或者是在补丁生成之后采取一些排序算法使得正确的补丁尽早被验证[77,95]。此类方法在一定程度上提升修复准确率，但是可能会影响修复技术的其它性能，例如召回率和修复效率。目前准确率较高的两种技术 CapGen[43] 和 ACS[32] 的召回率都比较低（CapGen 和 ACS 的召回率均低于 6%）。Fix2Fit[64] 的修复中经常出现用满限定 12 个小

时的情况。

少量工作研究了补丁已经生成的情况下，如何对补丁正确性进行判别，例如增加额外的测试规约过滤触发缺陷的错误补丁[89]、根据历史数据统计信息对补丁正确性进行判别[100]。但是这些方法目前也存在一些局限性：（1）召回率①较低且无法保证准确率②，目前最先进的识别补丁正确性的工作 PatchSim[86] 和 ODS[100] 的召回率只能达到 57% 左右，且更换数据集或者超参数（例如，判断正确性的阈值）之后，就无法保证准确率达到 100%。（2）部分方法只能缓解特定类型的过拟合问题，例如 Opad[89] 以程序崩溃和内存泄露为新增测试用例的规约，只能识别会导致这两种情况的错误补丁。

由上述讨论可知，（1）由于规约的不完备性，且难以获得完备的规约，自动修复技术无法保证生成补丁的正确性；（2）已有技术只能在一定程度上提升修复准确率，且可能会影响到修复技术的召回率和效率。最终开发者还是需要人工审阅生成的补丁。

1.3 本文研究思路

基于向开发者提供正确性未知的补丁时，开发者的修复效率和正确率并不一定会被提升，甚至会下降的现状。本文提出针对自动生成补丁的审阅提升技术，通过帮助开发者判断补丁正确性，高效完成补丁审阅，提升开发者的修复效率和正确率。该技术面临的**总体挑战**是：**如何在审阅阶段帮助开发者？**目前还没有已有工作对这个方面进行相关研究。针对该挑战，本文提出两种方案辅助开发者审阅补丁：（1）*过滤无效信息*。修复工具可能会为一个缺陷产生多个通过测试用例的补丁，过滤错误概率较高的补丁即可帮助开发者过滤掉部分无效信息。在这个过程中，存在的问题是：如何准确地过滤掉尽可能多的错误补丁？目前自动化补丁过滤技术过滤错误补丁数量有限，且部分技术只针对特定类型的缺陷。因此提升自动化错误补丁过滤数量才能减轻人工审阅补丁的负担。（2）*引导开发者关注关键信息*。目前自动修复技术只提供了生成的代码片段，即补丁。开发者在理解补丁时需要考虑多种信息，如修改位置，对测试执行的影响等。实际上，针对不同的缺陷或不同的补丁，需要关注的信息是不同的，引导开发者关注关键信息是十分重要的。因此，辅助审阅工具需要识别关键信息，并引导开发者关注这些关键信息。

本文的研究思路如图1.2所示，为了探究如何辅助开发者审阅补丁，本文首先进行了一个**补丁审阅实证研究**。目前开发人员在撰写补丁的时候辅助补丁审阅人员的主要手段是提供补丁解释信息。本文通过收集和分析开源网站上开发者提供的辅助补丁审阅的信息，提出了一个通用的补丁解释模型，并依据定性分析与定量分析结果，指导

---

① 该召回率是指识别错误补丁占全部错误补丁的比例。

② 该准确率是指识别出的错误补丁是真正的错误补丁的比例。

设计补丁审阅提升技术。

基于实证研究结果，本文提出一套**针对自动生成补丁的审阅提升技术**，其主要包括两部分：基于动静态信息结合的补丁过滤技术和交互式补丁过滤技术。在开发者人工审阅补丁之前，可采用自动化的补丁正确性判别技术将错误补丁过滤，即帮助开发者过滤无效信息。根据实证研究，开发者从补丁的静态特性与动态特性两方面解释补丁正确性，说明这两方面特性对于补丁正确性判别具有关键作用。因此，本文使用**基于动静态信息结合的补丁过滤技术**。该技术通过 S-Transformer 模型对历史数据中补丁的静态修改特征进行编码分析。然后采用 RAT 模型对历史补丁的动态覆盖特征进行编码分析，提取判别补丁正确性的关键特征，同时结合静态分析结果，从而对任意修复技术生成补丁的正确性进行判断。

自动化的过滤技术可以帮助开发者过滤掉部分错误补丁，提升了开发者的修复效率。但是仍有部分错误补丁无法通过全自动方法区分，最终仍然需要开发者参与到审阅过程中。为了进一步帮助开发者理解补丁正确性，高效过滤全部错误补丁，最终完成审阅过程，本文提出了一个**交互式补丁过滤技术**。实证研究发现，补丁审阅者通常是熟悉项目内容的，因此开发者提交抽象概括非正式的补丁解释信息也可以帮助审阅者理解补丁，例如提供补丁特性。该技术通过询问补丁的一些静态修改性质与动态测试执行性质相关的问题与用户进行交互，引导用户关注关键的补丁特性。候选补丁均是可以通过测试用例的，本文只选取候选补丁的差异特性向用户询问，作为关键的补丁特性。在交互过程中，用户在回答系统询问的同时，关注到关键的补丁特性，从而理解当前缺陷与补丁。系统根据用户的反馈过滤掉错误补丁，最终帮助用户完成审阅过程。为提升交互效率、优化询问次数，本文定义了询问选择问题，并理论证明了询问选择问题与最优决策树构建问题在多项式时间内可规约，从而引入决策树中的最小化最大分支算法作为解决询问选择问题的多项式时间内的最优近似算法。最后，为方便开发者在交互过程中调试缺陷，提升修复效率，本文设计了一个包含过滤视图与差异视图的交互界面。

综上，为了辅助开发者在给定候选补丁时高效准确修复缺陷，提升自动修复技术的实用价值，本文提出了针对自动生成补丁的审阅提升技术辅助开发者审阅补丁、完成修复。本文的主要贡献和创新点如下：

1. 提出了一个通用的补丁解释模型，可用于辅助开发者审阅补丁。实证研究中的定量实验表明，可同时采用多个补丁特性对补丁正确性进行判别。实证研究的发现不仅为本文的补丁审阅提升技术提供了指导，还具有对补丁解释生成，衡量缺陷报告质量等其它科学研究方向的参考价值。
2. 提出了一个自动化的基于动静态信息结合的补丁过滤技术，在人工审阅之前过

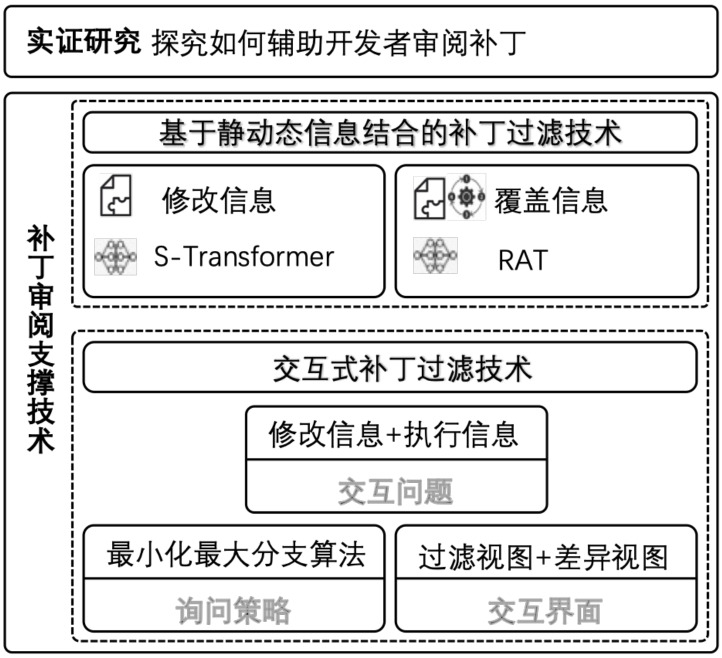


图 1.2　本文研究技术总结

滤掉部分无效信息，即错误补丁。该技术结合补丁的静态修改信息和动态覆盖信息并通过神经网络进行编码分析学习，提升了过滤错误补丁的数量。

3. 提出了一个交互式补丁过滤技术，通过询问补丁特性相关问题与用户进行交互，引导用户关注有助于理解补丁及缺陷的关键补丁特性，并根据用户反馈过滤无效信息，即错误补丁。定义了询问选择问题，对交互次数进行理论分析，提升交互效率。将该技术实现为一个 Eclipse 插件，并设计了用户图形界面，包含过滤视图与差异视图，方便用户审阅补丁调试缺陷，最终进行了定量实验和用户实验综合验证技术效果。

实验结果表明，本文的技术可以提升开发者的修复效率及正确率。该结果表明支持和改进补丁审阅过程也是一个有前途的研究方向，未来可开展更多的研究工作。同时，实验结果也表明，通过适当的工具辅助补丁审阅过程，低准确率的修复技术也能在实践中发挥作用。未来的修复技术可以放宽对补丁准确率的限制，从而在更多的缺陷上对开发者提供帮助。

1.3.1　补丁审阅实证研究

由于规约不完备，目前自动修复技术都无法保证补丁正确性，即生成的补丁即使可以通过全部测试用例，也不一定是正确补丁。因此自动生成的补丁不会直接被开发者接收，而需要验证其正确性，这可能会影响最终修复效率。尤其是在生成大量错误补丁的情况下，补丁质量问题影响了自动修复技术在实际场景中广泛应用的效果。为

了缓解该问题，现有很多技术尝试不同的方法判别补丁的正确性，提升自动修复技术的准确率，然而，实际中难以获得完备的规约，已有的方法无法从根本上保证补丁正确性，自动生成的补丁最终都需要开发者人工审阅，而目前缺乏对补丁审阅相关的研究。

为了更好的了解开发者如何审阅补丁，为提出补丁审阅提升技术提供指导，本文进行了一个实证研究：人工审阅了开源网站上开发者提交的补丁解释信息，总结有助于辅助开发者判别补丁正确性，理解补丁与缺陷的补丁特性。在该项研究中，主要关注以下问题：（1）补丁解释信息中包含了哪些可用于开发者进行补丁正确性判别的补丁特性？（2）这些补丁特性在调研的数据集内的分布情况是怎样的？

第一个问题定性分析了开发者依赖哪些补丁特性判断补丁正确性、完成补丁审阅过程。通过该问题，研究者可以在一定程度上了解：（1）哪些补丁特性有助于判别补丁正确性，甚至是自动化过滤错误补丁；（2）在补丁审阅过程中需要引导开发者关注哪些补丁特性辅助理解补丁，完成审阅过程。第二个问题的目标是定量分析这些补丁特性被使用的频繁程度或者相关程度。该研究结合定性分析与定量分析，为提升过滤错误补丁的数量最终辅助开发者审阅补丁提供参考思路。

这项研究中的**一个关键挑战**是：*如何从补丁解释信息中总结出补丁审阅依赖的信息*。开源网站上开发者提供了补丁解释信息以及与补丁审阅者之间交流信息都是自然语言形式，且描述通常与特定的缺陷相关，因此需要对所有解释信息进行抽象概括，提取通用的补丁特性。对此，本文基于开放编码 (open coding) 原则，首先人工审阅补丁解释信息，并对采样的数据进行编码，总结生成补丁特性相关的元素类型，提取编码模型，然后根据派生的元素类型标记解释信息，最后完成定量分析。通过该方法，可以探究补丁审阅过程中依赖的通用补丁特性信息，方便之后实现辅助审阅任意缺陷的补丁。

通过分析人工解释补丁信息，实证研究具有以下发现：(1) 提出了一个补丁解释通用模型，包含了五个核心元素，分别代表了开发者从不同方面解释补丁正确性，具体包括：补丁位置信息，缺陷原因以及补丁修改信息 (三类静态信息) 和缺陷触发条件及补丁应用前后的结果信息 (两类动态信息)，该发现说明，静态信息与动态信息对判别补丁正确性以及过滤错误补丁有帮助。（2）补丁审阅者通常是熟悉项目内容的，因此开发者提交的抽象概括性补丁描述信息也可以帮助审阅者理解补丁，该发现说明，向审阅者提供不完整或非正式的补丁解释信息，比如某些补丁特性，也有助于开发者理解补丁及缺陷。（3）定量研究结果表明，开发者会倾向于选择 2-3 个元素来解释一个补丁，说明多种元素结合，共同用于区分正确补丁与错误补丁相对于单一元素来解释补丁更为开发者接受，此发现也侧面说明了，补丁过滤及补丁审阅中可尝试同时采用

多种特性来判别补丁正确性，过滤错误补丁，辅助审阅。上述发现为本文提出的补丁审阅提升技术提供了重要的指导。

1.3.2 基于动静态信息结合的补丁过滤技术

如上文所述，通过帮助开发者过滤无效信息的方式可辅助开发者审阅补丁。在人工审阅补丁之前，可采用自动化技术过滤掉一部分错误补丁，减轻人工审阅的负担。近些年来，研究者已经提出许多自动化判别补丁正确性的技术，从使用信息源来看，这些技术或是根据补丁的静态代码特征，例如修改特征、代码相似性特征，或是根据程序的动态执行特征，例如生成测试用例是否可以触发缺陷、应用补丁前后测试覆盖情况。虽然目前这些技术取得了一些进展，但是其过滤错误补丁的数量仍不理想。

根据实证研究结果及已有工作研究，提升补丁正确性判别技术过滤错误补丁召回率主要面临**两个挑战**：*（1）如何充分利用静态信息；（2）如何结合静态修改信息与动态覆盖信息判别补丁正确性*。对于第一个挑战，已有的基于表示学习的补丁正确性判别技术只是将补丁修改代码当做字符串，未考虑代码的结构信息，修改的上下文信息，以及细粒度的修改信息。事实上，代码结构信息和修改的上下文信息也蕴含了补丁正确性判别的有效信息，例如语句类型、未修改语句的特征。细粒度代码节点级别的代码变更，而非语句级别的修改，有助于精确分析修改信息。因此，充分利用静态信息，对于过滤错误补丁是非常有帮助的。对于第二个挑战，已有补丁正确性判别技术都只采用了单一的信息源，即静态信息或动态信息。事实上，静态信息与动态信息对判别补丁正确性都非常重要，部分补丁在应用前后的测试覆盖不会发生任何改变，无法通过动态覆盖信息判别其正确性，部分补丁的静态修改不具备判断其正确性与否的特征，无法通过静态信息判别。因此同时结合静态与动态两方面信息，可提升过滤错误补丁效果。

针对上述的第一个挑战，本文将补丁应用前后所在函数的代码表示为一个基于抽象语法树的静态修改表征图，该表征方法可以显示以代码节点为粒度的修改信息，同时保留修改位置的上下文信息及整个函数的代码结构信息。针对上述的第二个挑战，本文将补丁判别分为两部分，第一部分是以修改表征图为输入的静态修改特征模型学习，即将修改表征图通过 S-Transformer 模型进行编码和分析，从中提取判别补丁正确性的关键静态信息，并对补丁正确性进行判别；第二部分首先构造了覆盖表征图，即以补丁应用前后行级别的代码和测试用例为节点，以两种节点之间是否存在覆盖关系构建边，特别地，根据覆盖情况不同，该图中的边具有四种不同的类型：代码节点在补丁应用前后都未被测试节点覆盖、在补丁应用前后都被测试节点覆盖、只在补丁应用前被测试节点覆盖和只在补丁应用后被测试节点覆盖。然后将修改特征模型的判别

结果与该覆盖表征图作为输入，通过 RAT 模型学习动态覆盖信息并结合修改信息对补丁正确性进行判别。

综上，本文提出了一个基于动静态信息结合的补丁过滤技术，可在人工审阅补丁之前自动化过滤具有较大概率错误的补丁。具体而言，本文提出的补丁过滤技术综合考虑了以代码节点为粒度的补丁修改信息，程序代码的结构信息，修改位置的上下文信息，动态覆盖信息等不同的信息源。同时，考虑到基于历史数据进行学习的补丁过滤技术的应用场景为使用历史其它缺陷的补丁数据来预测当前缺陷的补丁正确性，因此在验证其效果时应跨缺陷划分数据集，即同一缺陷的不同补丁只能出现在测试集或训练集中。已有技术均是只在跨补丁场景下进行验证，本文首次同时在这两种场景下验证基于动静态信息结合的补丁过滤技术的效果。实验研究结果表明，该技术可有效提升现有技术过滤错误补丁的数量：在跨补丁划分场景中，相比于只使用静态特征与只使用动态特征的技术，本文提出技术可多过滤 12.9% 和 34.6% 的错误补丁，而在跨缺陷划分场景中，本文技术也优于已有技术。

1.3.3 交互式补丁过滤技术

如前文所述，修复技术无法从根本上保证补丁正确性。目前自动化过滤错误补丁的数量有限，无法被过滤的补丁仍然需要开发者人工审阅，判断其正确性，即自动修复技术生成的补丁不可避免的需要开发者审阅。在给定正确性未知的补丁的情况下，开发者的修复效率并不一定会提升[40]。因此，若能辅助开发者在审阅补丁过程判别补丁正确性，提升修复效率，将扩大自动修复技术在实际场景中的应用影响力。

本文提出交互式补丁过滤技术，在给定一组正确性未知的候选补丁情况下，通过与用户进行交互过滤错误补丁，辅助审阅补丁。该技术主要面临三**个挑战**：（1）选择何种类型的问题与用户进行交互；（2）如何提升交互效率，减少交互次数；（3）如何设计交互界面。对于第一个挑战，询问问题的选择十分重要，需要考虑其划分错误补丁与正确补丁的能力，若是正确补丁与错误补丁均具有的性质，则无法过滤掉错误补丁。除此之外，还需要考虑对于用户来说回答询问的难易程度，若是用户无法提供正确答案，或是需要长时间思考才可以提供正确答案，那么交互式过滤错误补丁的效率将低于用户审阅补丁的效率。最后，通过提问引导用户关注补丁及缺陷相关的关键信息，帮助用户理解补丁及缺陷、甚至直接完成修复过程。对于第二个挑战，同挑战一类似，交互次数是交互效率的另外一方面的体现，若是需要回答的询问过多，也会给用户带来沉重负担。对于第三个挑战，交互式技术需要设计良好的交互界面，方便用户在回答询问过滤错误补丁的同时审阅补丁、调试缺陷，最终完成修复的目标。

针对上述第一个挑战，本文基于三种程序属性：修改方法（静态代码属性）、执行

路径和变量值（动态运行时特性），设计了三种类型的补丁过滤准则（即补丁特性相关的问题）。由于程序属性的收集耗时较长，在线收集程序属性并生成询问问题与用户交互是不现实的。为了及时响应用户反馈，完成交互过程，该技术分为两阶段来完成：准备阶段离线收集程序属性生成询问问题，交互阶段在线与用户交互过滤错误补丁，辅助审阅。针对上述第二个挑战，在给定补丁空间及该空间补丁的概率分布、询问空间和答案空间情况下，本文首先定义了询问选择问题，并理论证明了询问选择问题与最优决策树的构建问题在多项式时间内可规约，进而引入决策树中的最小化最大分支算法作为询问策略。已有工作已经理论证明，最小化最大分支算法是解决最优决策树构建问题的最优多项式时间内的近似算法，即每次选择尽可能将当前候选补丁等分的询问，可以使得最终的询问次数最少。针对上述第三个挑战，本文实现了一个基于 Eclipse 插件的交互界面，该界面不仅包含了用户与系统进行问答交互的过滤视图，还包含了审阅补丁的板块与查看补丁应用前后的差异视图，方便用户在交互过程中理解缺陷，进而完成修复过程。

综上，本文提出了一个交互式补丁过滤技术，利用了静态代码属性和动态运行时特性区分错误补丁与正确补丁，同时通过向用户询问补丁相关特性，引导用户关注关键信息，根据用户反馈过滤错误补丁，即过滤无效信息，最终帮助用户完成修复。为了验证其效果，本文进行了定量实验和用户实验，其中定量实验验证了本文技术过滤错误补丁的能力，用户实验则是在真实场景下，验证本文技术是否能帮助开发者提升修复正确率和效率。用户实验结果表明，使用交互式补丁过滤技术修复缺陷的开发者的修复效果（修复准确率和效率）显著优于另外三组基准调试场景下开发者的修复效果。其中，相比于没有任何辅助下自己完成修复的开发者准确率提升了 62.5%，修复时间减少了 25.3%。该实验结果表明：在合适的工具支持下，补丁审阅过程可帮助开发者理解缺陷，提升修复效率和准确率。支持和改进补丁审阅过程也是一个有前途的研究方向，未来可开展更多的研究工作。

1.4 论文组织

本文章节结构如下：

- **第一章 引言**。介绍本文的研究背景，相关研究现状以及当前尚未解决的问题，提出本文的研究思路和主要创新点。
- **第二章 补丁审阅实证研究**。提出了一个通用的补丁解释模型。通过分析开发者提交的补丁解释信息，总结可以用于辅助开发者理解补丁及缺陷、判别正确性的补丁特性，从而为设计补丁审阅支撑技术提供指导。
- **第三章 基于动静态信息结合的补丁过滤技术**。提出了一个自动化补丁过滤技术。

结合补丁的静态修改信息与动态覆盖信息，通过神经网络进行编码分析，提取对于判别补丁正确性的关键特征，进而过滤错误补丁，提升自动化补丁过滤技术的效果。

- **第四章 交互式补丁过滤技术**。提出了一个交互式补丁过滤技术。根据静态代码属性与动态运行时属性设计了与用户交互的询问问题，并定义了询问选择问题，引入最小化最大分支算法优化询问策略，最终实现了一个以 Eclipse 插件为载体的交互界面，提升了开发者修复效率和正确率。
- **第五章 结论及展望**。总结本文研究工作，并展望未来可能的研究方向。

第二章　补丁审阅实证研究

2.1　引言

近十年来，随着众多程序自动修复技术被提出[10–27]，程序修复领域已经取得了很大进展，极大缓解了开发者修复缺陷的压力。目前的程序修复技术都是基于测试用例，即输入一个缺陷程序和至少包含一个失败测试的测试用例集合，输出一个能通过所有测试用例的补丁集合。然而，该方法会存在“弱测试集”或“补丁过拟合”问题[37–39]，即使一个补丁能通过全部测试用例，也可能是不正确补丁。因此自动生成的补丁不会直接被开发人员接收，而需要人工验证，这可能会影响修复效率和正确率，尤其在生成的补丁是错误的情况下[40]。补丁质量问题会影响到自动修复工具在实际场景中广泛应用。

由于实际修复中规约不完备且无法获得完备的规约，修复技术无法从根本上保证补丁正确性，最终生成的补丁都需要开发者人工审阅，而将正确性未知的补丁提供给开发者，开发者的修复效率与正确率并不一定会提升，甚至可能会下降[40]。目前对于支撑开发者审阅补丁，提升修复效率的技术还缺乏研究。

为了系统探索，如何辅助开发者审阅补丁，判断其正确性，最终完成修复，本章进行了一个实证研究。具体而言，开源网站上开发者提交的补丁被审阅之后才会被接收，开发者在提交补丁的同时也会提交解释信息，以便审阅者理解并快速接收补丁。审阅者通过审阅补丁及解释信息完成审阅过程最终决定接收或拒绝补丁。本实证实验通过分析开发者提交的解释信息，总结有助于审阅的补丁特性。具体而言，有助于判别补丁正确性和开发者理解补丁及缺陷的补丁特性。该过程中的一个**关键挑战**是：**如何从补丁解释信息中总结补丁审阅依赖的信息？**开源网站上开发者提供的以及与补丁审阅者交流的补丁相关信息都是通过自然语言，且补丁信息一般都是与当前项目的缺陷相关，列举每个补丁的详细说明信息可能无法泛化到全部补丁上。因此，本章基于开放编码（open coding）原则，针对开发人员提供的补丁解释信息进行人工分析，并提出一个包含五种元素的通用补丁解释模型。除了进行定性分析之外，为了探究每种元素被使用的频率，元素之间的相关性等统计数据，本实证实验还进行了一些定量分析。通过定性和定量分析，本章具有以下关键发现，可为设计补丁审阅提升技术提供指导。

1. 提出了一个包含五个元素的通用补丁解释模型，组成该模型的各个元素代表了开发者从不同方面解释补丁正确性，具体包括：补丁位置信息、缺陷原因及补丁修改信息（三类静态信息）和缺陷触发条件及补丁应用前后的结果信息（两类动态信息），该发现说明：静态特性与动态特性对判别补丁正确性以及过滤错

误补丁均有帮助。

2. 补丁审阅者通常被认为是熟悉项目代码的，因此开发者提交的抽象概括性补丁描述信息也可以帮助审阅者理解补丁，该发现说明，向审阅者提供不完整非正式的补丁解释信息，比如补丁特性，也有助于开发者理解缺陷和补丁。
3. 定量研究结果表明，开发者会倾向于选择 2-3 个元素来解释一个补丁，说明多种元素结合，共同用于区分正确补丁与错误补丁相对于使用单一元素来解释补丁更为开发者所接受。此发现也从侧面说明了，补丁审阅提升技术中可尝试同时采用多种特性来辅助开发者判别补丁正确性，过滤错误补丁。

本章通过分析开发者提交的补丁说明信息，总结可用于辅助审阅的补丁特性。本章的组织结构如下：第2.2节介绍实证研究的实验设置；第2.3节描述实证研究的结果以及结果分析；第2.4节揭示了实证研究结果对未来研究的启示；第2.5表明了本实证研究的有效性；第2.6节对本章内容进行讨论和总结。

2.2 实验设置

本节介绍补丁审阅实证研究的实验设置，包括：研究问题（第2.2.1节）、数据集的选择（第2.2.2节）和实验过程（第2.2.3节）。

2.2.1 研究问题

本实证实验主要探究以下三个研究问题：

- **RQ1：补丁解释信息中包含哪些补丁特性及其具体类别？**
- **RQ2：补丁解释信息中不同补丁特性的分布是怎样的？**
- **RQ3：补丁解释信息中不同补丁特性的具体类型（表达方式）的分布是怎样的？**

2.2.2 数据集

本实证研究是针对开源项目中开发者提交的补丁解释信息进行人工分析，本小节将从项目和解释信息分别进行介绍。

本章选取了 GitHub 上 6 个流行的开源 Java 项目。为了增加数据样本的多样性，本章在选择这些项目时充分考虑了它们之间各方面的差异性。如表2.1所示，这 6 个项目来自不同的开发组织，涵盖了库程序和应用程序。其中，*RxJava* 是一个用于异步编程的库，通常用于构建 Android 应用程序[107]。*Spring* 是一个用于创建 Java 网络应用程序的框架[108]。*PocketHub* 是 GitHub 的 Android 客户端[109]。*Nextcloud* 是用于文件共享和通信的 Android 客户端[110]。*IntelliJ* 是一个被广泛使用的 Java 集成开发环境[111]。*Lang* 是 Java 实用类的库程序[112]。

表 2.1　开源项目详细信息

组织名称/项目名称	**项目类型**	**标星数目**	**PR 数目**	**代码行数**
ReactiveX/**RxJava**	库程序	35,688	51	267.2K
Spring/**Spring**-Boot	框架程序	29,646	56	243.3K
PocketHub/**PocketHub**	应用程序	9,311	40	157.3K
JetBrains/**IntelliJ**-Community	应用程序	6,647	64	3,544.7K
Apache/Commons-**Lang**	库程序	1,450	36	761.8K
Nextcloud/Android	应用程序	1,090	53	59.4K


LANG-1118: Fix StringUtils.repeat(char, int) #72

rikles wants to merge 2 com pache:master from rikles:fix-LANG-1118

rikles commented on 28 Apr 2015

Close PR #68. Recreated due to rebase on master.

Now doing what is said in JavaDoc comment :
when passing a negative repeat value to the *StringUtils.repeat(char, int)* function, it returns an empty String instead of throwing a *NegativeArraySizeException*.

britter commented on 29 Apr 2015

Hello @rikles I'm still trying to find out how to correctly merge PRs from the command line. Looks like it didn't work with this PR. Your changes have been merged. Can you close this PR please? :) Thank you!

rikles commented on 1 May 2015

Merged into master. Close PR.

图 2.1　GitHub 上 Lang#27 的一个拉取请求示例

GitHub 上开发者提交的补丁说明信息一般为拉取请求 (pull request，PR) 的形式。图2.1展示了 Lang 项目中一个拉取请求的截图，一个拉取请求一般分为以下四部分:

- 标题总结该拉取请求。
- 描述信息详细解释了该拉取请求。例如，该拉取请求修复了什么缺陷和如何修复该缺陷的。
- 讨论内容展示了提交者和审阅者之间的讨论。

- *代码变更信息*展示了代码提交, 修改文件和代码改动差异，这三项信息都可以在对应的标签栏内找到。

本实证研究首先收集了每个项目在 2019 年 1 月 21 日（本实证实验开始时间）之前的所有拉取请求，然后根据以下过滤准则来选择符合条件的拉取请求。

- *拉取请求必须已经被接收*。被接收的拉取请求已经被开发者验证为是有效的补丁，并且说明它的解释信息在一定程度上是足够补丁审阅者理解该补丁的。本实证研究遵从以下规则来收集被接收的拉取请求，包括通过其它机制或者是 GitHub 用户界面合并的拉取请求。
    - 如果一个拉取请求在 GitHub 上被标记为接收，那么本实证实验认为它已经被接收了。
    - 如果一个提交（commit）关闭了当前的拉取请求，并且此提交出现在上游项目，本实证实验认为它已经被接收了。
    - 如果一个拉取请求的讨论涉及到一个提交 SHA，那么本实证实验认为它已经被接收了。特别地，本实证实验遵从 Gousios 启发式规则[113] 来收取被接收的拉取请求：（1）讨论中包含了一个提交 SHA；（2）提交的 SHA 出现在项目的分支中；（3）拉取请求对应的讨论内容可以被以下正则表达式匹配：

$$(?:merg|appl|pull|push|integrat)(?:ing|i?ed)$$

- *拉取请求修改的是 Java 代码*。即使一个项目的主要开发语言是 Java，修改代码也可能涉及其他语言，例如 XML 或者 Markdown。因为本文只关注 Java 程序中的缺陷，因此，本实证实验只考虑主要修改 Java 代码的拉取请求，并且通过人工查看具体代码修改来检查选取的拉取请求是否满足该条件。
- *拉取请求只涉及缺陷修复*。一个拉取请求可能有不同的目的，例如，增加新功能，重构或者缺陷修复。当前实证研究只关注缺陷修复相关的拉取请求，并且通过以下条件来判断一个拉取请求是否为缺陷相关：（1）拉取请求已经被标记为“*缺陷*”；或者（2）在拉取请求的描述信息或讨论内容中存在缺陷报告链接，并且该链接被标记为“*缺陷*”。为了保证选取的拉取请求只涉及缺陷修复，本文选择只涉及一个标签（“*缺陷*”）的拉取请求。

对于每个项目，本实证实验随机选择符合上述标准的拉取请求，直到分析结果达到饱和。如果一个拉取请求的描述信息只有“代码提交”或者“缺陷修复”等没有信息量的描述，本实证实验会把这样的拉取请求排除。最后，本章总共人工审阅了 300 个拉取请求。

2.2.3　实验过程

回答 RQ1 的实验过程：本实证实验遵循开放编码原则[114]（open coding），首先生成元素类型，然后根据派生的元素类型标记拉取请求。具体过程如下：首先，两位作者通过阅读标题、描述信息和讨论内容对采样数据分别进行编码，并总结每个拉取请求中包含的元素；然后，提取编码模型。最后将编码分组处理并去重，在某些情况下对它们进行泛化或特殊化。处理之后的编码被应用于所有拉取请求。当新的编码出现时，它们会被整合到编码集合中。在此过程中，代码变更信息可以用来更好地理解拉取请求并验证编码结果。此步骤的输出是补丁解释模型中包含的*元素*（elements）以及这些元素之间的关系。一旦两位作者之间出现不一致，就与其他作者讨论，直到达成共识。在识别所有元素之后，本实证实验按照相同的过程将每个元素进一步细分为*表达形式*（expressive forms），以便捕获不同元素的具体类别。最后，本实证实验建立了补丁解释模型。

回答 RQ2 的实验过程：在完成定性分析之后，本实证实验还进行了对补丁模型中元素的定量分析，并通过以下三方面回答 RQ2。

- **RQ2a:** 每个拉取请求中通常包含几个元素？
- **RQ2b:** 在解释补丁时，什么元素会被更经常使用？
- **RQ2c:** 在拉取请求中，任意两个元素的出现是否有关联？

为了回答 RQ2a，本实证实验人工标记解释模型中的元素是否出现在每个拉取请求中，并统计每个拉取请求中的元素个数。为了回答 RQ2b，本实证实验统计包含每种元素的拉取请求的个数，并计算其在所有拉取请求中的比例。为了回答 RQ2c，本实证实验采用了一个统计度量方式 *lift*[115]，可以衡量元素 A 出现时，元素 B 出现增加的概率，如公式2.1所示，*per(A)* 是指包含元素 A 的拉取请求占所有拉取请求的比例，*per(A,B)* 指同时包含元素 A 和元素 B 的拉取请求占所有拉取请求的比例。如果 *lift(A,B)* 的结果为 1，即 $per(A,B) = per(A) \times per(B)$ 说明元素 A 和 B 是不相关的。如果 *lift(A,B)* 的结果大于 1，说明元素 A 和元素 B 出现在同一个拉取请求中的概率更大，如果 *lift(A,B)* 的结果小于 1，说明元素 A 和元素 B 出现在同一拉取请求中的概率更小。

回答 RQ3 的实验过程：除了对补丁模型中的元素进行定量分析，本实证试验还进一步对各元素中的子类进行了定量分析，RQ3 的实验过程与 RQ2 类似，通过以下两个方面回答 RQ3。

- **RQ3a:** 每个元素中一般包含几种表达形式？
- **RQ3b:** 每个元素中，什么表达形式会被经常使用？

由于目前收集数据的数量不足以进行对表达形式之间的相关性分析，因此未在 RQ3 中进行相关性分析。

表 2.2　元素的特点

元素	解释信息	缺陷/修复相关	静态/动态
条件（Condition）	缺陷发生的条件（When）	缺陷相关	动态
结果（Consequence）	缺陷导致的后果（What）	缺陷相关	动态
位置（Position）	缺陷发生的位置（Where）	缺陷及修复相关	静态
原因（Cause）	为什么缺陷会发生（Why）	缺陷相关	静态
修改（Change）	缺陷如何修复（How）	修复相关	静态

$$lift(A,B) = \frac{per(A,B)}{per(A)\times per(B)} \tag{2.1}$$

2.3　实验结果与分析

2.3.1　RQ1：补丁解释模型

图2.2为本实证实验定义的补丁解释模型，它定义了一个补丁解释中可能包含的补丁特性。在这个模型中，每行 ::= 左侧的元素由右侧的元素组成，右侧的元素至少存在一个且不需要同时存在。

从该模型中可以看出，一个补丁解释由五个元素组成（第一行）：条件（*Condition*）、结果（*Condition*）、位置（*Position*）、原因（*Cause*）、修改（*Change*）。表2.2总结了这五个元素的特点，条件、结果和原因描述了缺陷相关的信息，修改描述了补丁修复相关的信息，位置既描述了缺陷相关信息也描述了修复相关信息。条件指触发缺陷的发生条件，结果是指缺陷导致的后果，位置是源码中缺陷发生的位置或修复的位置，原因是缺陷发生的原因，修改是在源码中如何修复了缺陷，换言之，这五个元素分别代表了缺陷发生的条件（When）、位置（Where）、原因（Why）、导致的后果（What）以及它是如何被修复的（How）。此外，位置、原因和修改描述了程序中静态的语法特征，条件和结果描述了程序的动态的执行信息。

每个元素都可以被扩展为其子类（第 2-6 行），并且每个子类可以被进一步扩展，如果一个子类可以进一步扩展就称之为中间表达形式（[] 标记的元素），否则称之为叶子表达形式（<> 标记的元素）。

图2.3是一个补丁解释的示例①，它包含了一个条件元素和两个结果元素。条件元素描述了一个动态的程序状态，即函数的参数是一个负值。一个结果元素描述了期望的程序状态，即返回一个空的字符串，另外一个结果元素描述了一个实际发生的事件，即抛出一个异常。

---

① https://github.com/apache/commons-lang/pull/72。

```
1.   EXPLANATION  ::=  Condition, Consequence, Position, Cause, Change
2.   Condition    ::=  [dynamics]
3.   Consequence  ::=  [expected], [actual]
4.   Position     ::=  <file>, <inner_class>,
                       <method>, <variable>, <module>
5.   Cause        ::=  <missing_process>, <wrong_process>
6.   Change       ::=  <insertion>, <deletion>,
                       <replacement>
7.   [expected]   ::=  [dynamics]
8.   [actual]     ::=  [dynamics]
9.   [dynamics]   ::=  <state>, [event]
10.  [event]      ::=  <missing_event>, <occurred_event>
```

图 2.2　补丁解释模型
[ ] → 中间表达形式, <>→ 叶子表达形式

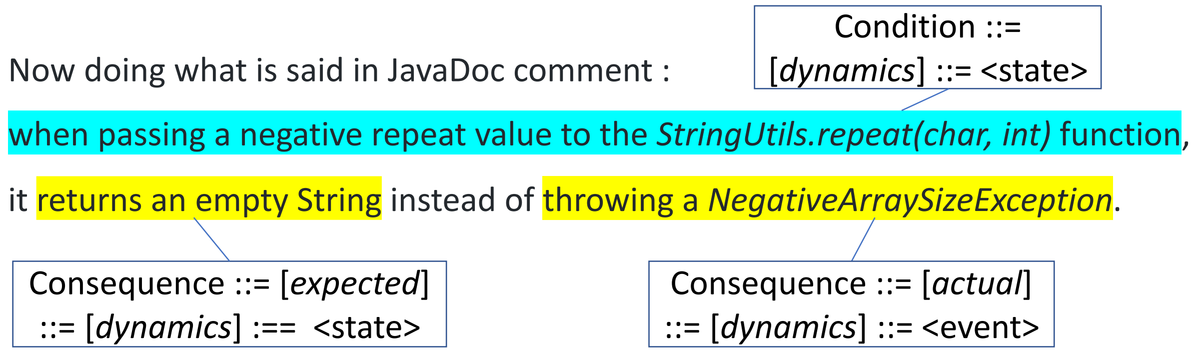


图 2.3　一个补丁解释模型示例（[Lang#72]）

接下来本章详细介绍解释模型中的元素特征。

条件　表示运行时触发缺陷的条件，它是一个动态的（dynamics）特征。动态特征可以进一步被划分为状态 (state) 和事件（event），说明一个动态的条件可以是程序状态的条件也可以是执行过程中一系列事件的条件。类似地，事件可以进一步被划分为未发生的事件（missing_event）和已发生的事件（occurred_event），即触发基于事件的条件，既可能是一系列事件的发生，也可能是一系列事件未发生。

- 状态类的条件：当使用状态描述缺陷触发条件时，典型模式是使用对应位置上的变量声明一个谓词。常见的谓词模式是一个变量是否等于某个值或者具有某种类型。例2.1中缺陷触发条件是第二个参数不是特定（Wildcard）类型。谓词描述可能更抽象和泛化，例2.2展示了一个缺陷触发条件是输入值中包含小写字母。除了以上使用具体程序代码元素描述具体的条件之外，一个条件也可能是

由一个抽象概念且开发者都熟悉的词来描述抽象的条件。例如，例2.3展示了一个抽象条件：需要多种数据源出现，其中“数据源 (datasources)”就是一个抽象概念且开发者都了解。

- 事件类的条件：除了使用状态描述缺陷触发条件，还可以使用程序执行中一系列事件来描述缺陷触发条件。一个事件既可以是具体的事件，比如函数调用或者抛出异常，也可以是一个抽象事件，比如打开文件，还可以是用户事件，比如点击按钮。例2.4展示了一个抽象事件：设置一个端口 *management.port*，实际是创建一个对象并且调用它的函数。例2.5展示了发生在应用程序图形界面中的用户事件。

理论上看，所有程序事件都会导致程序状态发生变化从而触发缺陷，因此所有基于事件序列的条件都可以通过状态来表述。例如，在例2.5中，点击按钮和标签会导致程序中相应变量的值发生变化。补丁提交者仍然更多（61.1%）使用事件序列来描述缺陷，说明开发者使用自然语言描述补丁时，使用事件序列更方便。在当前的编程语言中，使用程序状态描述缺陷条件是很容易的，但是使用事件序列描述缺陷条件并不容易。本实证实验表明，如果可以在编程语言中提出一种类似的机制，开发者在描述补丁正确性时可能会更加便捷。

例 2.1 *fixes bug in TypeUtils.equals(WildcardType, Type) where it was incorrectly returning true when* ***the second argument was not a Wildcard type****. [Lang#73]*

例 2.2 *Currently, IllegalArgumentException occur* ***if contains lowercase into log level.****[Spring#7914]*

例 2.3 ***When there are multiple datasources present,*** *make sure that the AutoCongureTest-Database annotation marks the embedded source as primary. [Spring#7217]*

例 2.4 *Webflux doesn't require Servlet.class,* ***when setting management.port****, auto-configuration would fail with class not found exception.[Spring#10590]*

例 2.5 ***Click one issue of "New" tab in Home page. Click home/back button on the toolbar in the opened issue page.*** *Will see a strange loading view on the toolbar.[PocketHub#1082]*

结果 结果是缺陷发生时导致的意外结果。类似于条件，结果也描述了程序运行时的行为，可以划分为两个方面：期望行为（expected）和实际行为（actual）。例2.6描述了程序的期望行为是 *Follow* 应该被隐藏。例2.7描述了程序缺陷的实际行为为一个

IllegalArgumentException 异常被抛出。跟条件一样，期望与实际结果都是动态(dynamics)特征，可以被扩展为状态和事件。

发现 1 期望行为和实际行为在大多数情况下可以互相推导出，*87.2%* 的拉取请求中只包含其中一种结果。 一般情况下，一个补丁解释中仅出现一种结果，但是并不是所有的拉取请求中只描述一种缺陷结果，例2.8同时描述了两种结果，因为这两种结果无法互相推导，同时描述两种结果可以为理解补丁正确性提供更多的信息。

例 2.6 ***Hides 'Follow' on*** *if the viewed user is the current user.[PocketHub#955]*

例 2.7 *Currently,* ***IllegalArgumentException occurs***
if contains lowercase into log level.[Spring#7914]

例 2.8 *The following scenario did not work...*
expected result: result stream emits the combined event
actual result: result stream does not emit anything *[RxJava#5494]*

位置 位置表示源代码中缺陷发生和修复的位置。位置可以通过具体的源码来说明，也可以通过一些总结的术语说明。在解释模型中，位置可以被扩展为五种形式：文件（*file*），内部类（*inner_class*），函数（*method*），变量（*variable*），模块（*module*）。其中前 4 种是具体的代码元素，例如，例2.9中 *StrBuilder* 是一个文件名，*replaceImpl* 是一个函数名。位置并未被扩展为类（class），因为在 Java 文件中，文件名与主类名是一致的。同时，位置也没有被扩展为语句或者行号，本实证实验探究此原因为：具体的行号或者语句可以通过代码改动查看，因此补丁提交者不需要再次提及。另外，如例2.9所示，较小粒度的代码位置不一定包含较大粒度的代码位置，即明确缺陷函数 *replaceImpl* 并不一定可以明确出错文件 *StrBuilder*，因为具有此名称的函数可以出现在不同的类中。

例 2.9 *Fix issue of buf using nonupdated buffer in* ***StrBuilder replaceImpl****. Avoid array OoB error by keeping variable buf consistent with buffer.[Lang#200]*

发现 2 拉取请求中的位置信息没有包含代码元素的类型。在例2.9中，代码提交者并未明确说明 *StrBuilder* 是一个文件，*replaceImpl* 是一个函数。这是因为代码提交者默认代码审阅者是十分熟悉代码的。

发现 3 变量也是缺陷位置的表达形式之一。例2.10提及了一个变量形式的缺陷位置 *swappedPair*。现有缺陷定位技术[116,117] 和缺陷预测技术[118]，往往在语句、方法、类或者文件级别定位或预测缺陷。该发现为这些技术提供了新的研究思路：将变量作为缺陷位置。

例 2.10 *There seems to be a bug in the current implementation of the isRegistered implementation, where the* ***swappedPair*** *is constructed similarly to the existing pair to check their existence in registry.[Lang#282]*

原因 原因描述了程序中的缺陷为什么会发生，具体可以分为两类：未执行操作（missing_process）和错误操作（wrong_process）。未执行操作描述了开发者忽略处理某些情况。一种可能是代码忽略处理输入域中的一个子集，即开发者忘记处理一些边界情况。例2.11描述了程序缺少对订阅者中是否订阅进行检查。另一种可能是代码忽略了一个步骤。例2.12显示开发者忘记清除缓存。错误操作描述了错误处理代码的情况。例2.13描述了程序没有正确排除两个操作导致缺陷产生。

例 2.11 *Previously SingleFromCallable* ***did not check if the subscriber was unsubscribed*** *before emitting onSuccess or onError.[RxJava#5743]*

例 2.12 *Logout* ***never cleared the WebViews cookies*** *so you could not switch you accounts, we also need to clear the cached items in the database.[PocketHub#1109]*

例 2.13 *The logic* ***didn't properly mutually exclude the timer action and the onNext action****, resulting in probabilistic emission of the same buffer twice. [RxJava#5427]*

修改 修改描述了应用到源代码上的具体修改，可以划分为三类：插入（insertion）、替换（replacement）和删除（deletion）。具体的补丁在代码改动中可以查看，因此拉取请求中的解释信息只是抽象总结了具体的代码改动。例2.14提及到了删除操作，补丁提交者虽然没有提及删除的具体位置，但是抽象总结了删除的位置和内容。例2.15是一个描述插入的例子，“关闭连接”是总结了插入的语句，“完成检查之后执行”总结了插入的位置。

例 2.14 *This pull request removes the started check on stop() for Jetty and Tomcat.[Spring#8227]*

例 2.15 *Close connection after performing the actual check to release resources.[Spring#10153]*

发现 4 *在包含修改的拉取请求中，86.4% 的拉取请求抽象总结了具体的补丁修改。*

2.3.2　RQ2：元素在补丁解释中的分布

2.3.2.1　RQ2a：每个解释中涉及的元素数量

发现 5 *接近 70% 的拉取请求包含 2-3 个元素。* 表2.3展示了包含对应元素个数（1-5 个元素）的拉取请求的百分比，总体来看，只有 2.0%（6 个）拉取请求的说明信息中涉及了全部 5 个元素，接近 70% 的拉取请求会涉及 2 个或 3 个元素。

一个补丁解释中只出现 2-3 个元素是合理的。首先，有些元素很难单独使用，例如*位置*和*条件*。因此，只包含一个元素的拉取请求所占比例很小。其次，没有必要用太多的元素来解释一个补丁。有两个可能的原因。首先，不同的元素可能传递重叠的信息。在例2.11中，缺陷出现的原因是开发人员忘记检查订阅者是否取消订阅，一个直观修复该缺陷的方式是增加检查。在这种情况下，*原因*和*修改*是重复的，所以在解释信息中*修改*相关的描述未被提及。其次，有些元素对说明补丁正确性可能并不重要。例2.15中没有说明触发该缺陷的条件，而*条件*信息在解释该补丁正确性时并不重要。这一观察结果表明，选择哪些元素来说明补丁正确性很重要。

表 2.3　包含不同元素个数的拉取请求的比例

元素个数	RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	总计
#1	7.8%	19.6%	32.5%	18.8%	8.3%	30.8%	20.4%
#2	29.4%	42.9%	42.5%	45.3%	50.0%	63.5%	45.5%
#3	47.1%	23.2%	17.5%	23.4%	36.1%	3.8%	24.1%
#4	11.8%	12.5%	7.5%	7.8%	5.6%	1.9%	8.0%
#5	3.9%	1.8%	–	4.7%	–	–	2.0%

#x：包含 x 个元素的拉取请求

每个项目内的分析结果与总体结果是相似的，对于其中三个项目，包含 2-3 个元素的拉取请求所占比例是最大的（RxJava：76.5%、Spring：66.1% 和 Lang：86.1%），另外三个项目（PocketHub，Nextcloud 和 IntelliJ）结果略有不同，包含 1 个元素的拉取请求所占百分比高于包含 3 个元素的拉取请求的百分比，通过进一步分析这个问题，可以发现，在 PocketHub，Nextcloud 和 IntelliJ 中包含的元素主要是*结果*。出现这种现象的可能原因是：它们的项目类型不同。相比于库程序和框架程序，应用程序类项目可能需要更少的解释信息。在大多数情况下，一个期望的*结果*可以描述应用补丁之后的程序，足以解释补丁的正确性。例2.16展示了应用修改之后一个期望的结果：可以正确显示一些特征。

例 2.16 “*modified to display characters of some languages(eg. chinese) correctly*” *[PocketHub#466]*

发现 6 修改和结果单独使用可以解释补丁，位置元素往往跟其它元素一起使用。此外，本实证实验发现，当一个拉取请求只包含一个元素时，这个元素通常是修改或结果，说明这两个元素单独使用可以在一定程度上解释补丁的正确性。当一个拉取请求中同时包含两个元素时，通常会包含位置信息，说明位置信息虽然不能单独解释一个补丁，但是可以跟其它元素一起使用。

2.3.2.2 RQ2b：在解释信息中更常被使用的元素

发现 7 结果和条件是两个最多被使用的元素，原因是最少被使用的元素。 表2.4展示了包含每种类型元素的拉取请求的百分比。从表中可以看出，结果是最经常被使用的元素（75.7%），这是符合预期的，因为大多数补丁提交者了解缺陷出现的后果，因此提交补丁时会描述缺陷的结果。条件是第二经常被使用的元素（48.3%），这是由于缺陷往往在一定条件下被触发，当描述缺陷产生后果时，补丁提交者也会描述触发缺陷的条件。原因是最少被使用的元素。一个可能的原因是，当说明修改时，开发人员可以自动推测出缺陷原因。例如，例2.17只包含了修改信息没有对缺陷进行解释，但是可以通过修改内容推断出缺陷发生的原因是缺少空检查。

表 2.4 包含每种类型元素的拉取请求的比例

元素	RxJava	Spring	PocketHub	IntelliJ	Lang	Nextcloud	总计
位置	76.5%	48.2%	12.5%	37.5%	88.9%	3.8%	31.7%
条件	51.0%	41.1%	57.5%	53.1%	36.1%	49.1%	48.3%
结果	76.5%	60.7%	80.0%	81.3%	50.0%	98.1%	75.7%
原因	27.5%	23.2%	30.0%	21.9%	19.4%	13.2%	22.3%
修改	43.3%	55.4%	25.0%	42.2%	44.4%	9.4%	37.0%

例 2.17 *DiffBuilder: Add null check on fieldName when appending Object or Object[]. [Lang#121]*

对于每个项目，结果也是相似的。例如，结果在每个项目中的使用频率都很高，它是 Lang 中第二被经常使用的元素，也是其它项目中最经常被使用的元素。但是，不同项目之间也存在一些差异。例如，框架程序和应用程序中涉及位置和条件的拉取请求更多，而应用程序项目中的拉取请求很少使用位置。一个可能的原因是 PocketHub 和 Nextcloud 中有 40% 拉取请求是用 UI 交互描述的，很难直接映射到源代码中的某个位置。

2.3.2.3 RQ2c：元素的相关性分析

发现 8 位置与修改具有很强的正相关性，条件和结果彼此之间具有正相关性，与其他元素均具有负相关性。 表2.5展示了任意两个元素使用 *lift*[115] 统计度量相关性的

结果。加粗数字（1.17）表示了位置和修改具有较强的正相关性，斜体数字（0.61）表示了结果和修改具有较强的负相关性。

从表格中可以看出，具有正相关性的共有 3 对元素，其中，位置与修改和原因均具有正相关性。原因在于，当解释一个原因或者修改时，位置常常可以用作辅助信息并且它的存在依赖于其它元素。除此之外，条件和结果也具有正相关性，这种现象也是合理的：一个单独的触发条件没有缺陷结果是没有意义的。另外一方面，条件和结果都与其它元素具有负相关性。特别是结果与修改负相关性很高。本实证实验探究了原因，发现许多拉取请求虽然只包含条件和结果，但是足以让补丁审阅者推断出缺陷及补丁的其他信息，从而判断补丁的正确性。

表 2.5　元素之间的相关性分析（lift 度量结果）

	位置	条件	结果	原因	修改
位置	–	–	–	–	–
条件	0.91	–	–	–	–
结果	0.79	1.11	–	–	–
原因	1.15	0.80	0.80	–	–
修改	**1.17**	0.71	*0.61*	0.97	–

2.3.3　RQ3：表达形式在元素中的分布

2.3.3.1　RQ3a：元素中表示形式的数量

本文发现一些元素的表达形式本身是互斥的：一个元素在一个拉取请求中只会有一种表达形式，比如条件，原因和修改。因为补丁提交者可能会使用一些抽象语言概括这些元素，因此在一个描述补丁的拉取请求中，所有的缺陷会被一种条件触发，具有单一的结果，并且需要一种操作修复缺陷。因此，在本文收集的数据集中，这三类元素只有一种表达形式。

对于位置和结果，不同表达形式之间是互补的。表2.6和表2.7分别表示了包含这两类元素的不同表达形式个数的拉取请求的比例。

补丁提交者在描述位置元素时，为了更精确的指向一个位置，往往会使用多个粒度的位置表达方式。从表2.6可以发现，在本文的数据集中的一个拉取请求中，提交者最多使用三种粒度的位置表达方法，其中，包含一种或者两种位置表达方式的拉取请求较多，分别为 53.5% 和 41.9%，少量的拉取请求包含了三种位置表达方式，为 4.7%。不同项目之间的情况稍有不同，首先，PocketHub 项目中的拉取请求对位置信息描述最为简单，一般只有一个位置信息，这种现象的主要原因是 PocketHub 为 GUI 应用程序，大部分拉取请求都是通过用户行为描述进行说明而不具体指向程序中的代码。其

次，相比于其它项目，Spring 项目具有较多种类的位置信息表达形式，一个可能原因是作为框架程序，Spring 往往需要更为精确的代码缺陷位置来描述补丁。

如例2.8所示，一些拉取请求中可能会同时描述期望行为和实际行为，因此，结果元素往往具有多种表达形式。从表2.7可以看出，大约 87.2% 的拉取请求只提及了一种结果，而 12.8% 的拉取请求同时提及了期望结果和实际结果。每个项目的情况几乎是一样的：包含一种结果信息的拉取请求比例远高于包含两种结果信息的拉取请求。

表 2.6　包含位置元素的不同表达形式个数的拉取请求比例

数目	**RxJava**	**Spring**	**PocketHub**	**IntelliJ**	**Lang**	**Nextcloud**	**总计**
#1	71.8%	51.9%	100.0	62.5%	18.8%	50.0%	53.5%
#2	28.2%	33.3%	–	33.3%	78.1%	50.0%	41.9%
#3	–	14.8%	–	4.2%	3.1%	–	4.7%

数目：位置元素中不同表达形式的数目，位置元素包含五种不同的表达形式，#x 代表只包含 x 种位置的表达形式的拉取请求。

表 2.7　包含结果元素的不同表达形式个数的拉取请求的比例

数目	**RxJava**	**Spring**	**PocketHub**	**IntelliJ**	**Lang**	**Nextcloud**	**总计**
#1	89.7%	88.2%	84.4%	88.5%	77.8%	88.5%	87.2%
#2	10.3%	11.0%	15.6%	11.5%	22.2%	11.5%	12.8%

数目：结果元素中不同表达形式的数目。本实验只考虑结果元素中两类表达形式：实际行为和期望行为，#1 代表只涉及实际行为或期望行为的拉取请求。

2.3.3.2　RQ3b：在解释信息中更常被使用的表达形式

表2.8展示了五种补丁解释元素中各子类表达形式的具体分布情况，接下来依次分析每种元素中表达形式的分布。

位置。如表2.8所示，38.0% 的位置元素都包含文件名，它是最常见的位置信息。37.2% 的位置元素包含函数名。更多细粒度的位置信息比如变量，是很少被使用的。这个观察跟之前的研究一致[119]，开发者希望自动定位技术提供函数级别的信息。模块的表达形式只出现在 RxJava 项目中，这是因为只有该项目具有明确的模块的概念，且补丁提交者和审阅者之间都可以互相理解。

发现 9 粗粒度的位置信息的使用程度（文件和函数）比细粒度的位置信息（变量）的使用程度更频繁。

条件。从表2.8可见，在本实验的数据集中，未发生的事件从未被使用解释触发条件，另外，状态也是比发生事件更少使用，说明对于解释触发条件，事件比状态更重

要。

结果。结果可以从实际行为和期望行为两方面描述，从表中可以看出，相比于期望行为，提交者更倾向于使用实际行为。另外，同条件一样，事件的使用比状态的使用更多。当事件被使用时，具体是未发生事件，还是发生事件是取决于缺陷类型的，并且可以看出发生事件和未发生事件各占一定比例。请注意，实际未发生事件与期望发生事件是彼此对应的。

原因。原因被如何描述是取决于缺陷类型，而不是提交者自己可选的。一般而言，一个缺陷发生的原因可分为两种情况：一个是错误处理了某些情况，另外一个是未处理某些情况。表2.8展示了这两种情况占比分别为 74.6% 和 25.4%。

修改。和原因类似，修改如何被描述也取决于缺陷类型。如表2.8所示，三种表达形式所占比例为：插入（36.7%）、删除（5.1%）和替换 (58.2%)。这一结果说明大多数缺陷是通过增加或是替换程序语句被修复的，少部分缺陷是通过删除语句。这一发现也与现有修复工具将删除作为一个反模板一致[67]。

从表2.8中，还可以观察到表达形式具有一些项目特定特征。如上文所述，与其他项目相比，Lang 项目的拉取请求在位置元素上使用了更多种类的表达形式。此外，还可以观察到，与其他项目相比，Lang 项目中更频繁地使用函数这一表达形式。就条件而言，GUI 应用程序（PocketHub 和 Nextcloud）和基于事件的项目（RxJava）更多使用事件，而库项目（Lang）更多使用状态。有趣的是，应用程序 IntelliJ 比其它两个应用程序（PocketHub 和 Nextcloud）更频繁地使用状态。作者进一步调查了 IntelliJ 项目的拉取请求，发现主要原因与 IntelliJ 项目的复杂性有关，由于 IntelliJ 是一个复杂的 IDE 项目，触发缺陷的状态通常可以通过不同的事件序列达到，并且通过状态来说明条件更容易。例如，一些缺陷与插件之间的冲突有关，通常使用状态来描述：“同时存在 XX 和 XX 插件”，而不是描述插件安装过程。

2.4　实证实验的启示

本实证实验的发现对不同方面的研究均有启示作用：

对于补丁正确性判断，本实证实验发现开发者通常从静态和动态两方面解释补丁的正确性，其中静态包括：位置、原因与修改，动态包括：条件和结果。开发者通常被认为是熟悉项目的，向审阅者提供不完整非正式的解释信息，例如补丁特性，也会有助于开发者理解补丁正确性。此外，本章的定量实验发现，补丁提交者通常采用不止一种元素描述补丁，这启示可同时向开发者提供多种补丁特性。以上发现同样适用于提升自动化补丁过滤技术，即根据补丁的静态特征，例如补丁修改信息，位置信息，与动态特征，例如补丁的测试执行信息，自动判别补丁正确性。

表 2.8　五种元素中表达形式的使用频率

		RxJava	**Spring**	**PocketHub**	**IntelliJ**	**Lang**	**Nextcloud**	**总计**
位置	文件	38.5%	44.4%	60.0%	58.3%	12.5%	50.0%	38.0%
	函数	23.1%	22.2%	–	29.2%	78.1%	50.0%	37.2%
	内部类	–	–	–	–	3.1%	–	0.6%
	变量	–	33.8%	40.0%	12.5%	6.3%	–	12.4%
	模块	38.5%	–	–	–	–	–	11.6%
条件	未发生事件	–	–	–	–	–	–	–
	发生事件	80.8%	43.5%	81.8%	58.8%	–	73.1%	61.1%
	状态	19.2%	56.5%	18.2%	41.2%	100.0%	26.9%	38.9%
结果	实际状态	–	17.6%	25.0%	11.5%	27.8%	–	11.5%
	实际未发生事件	7.7%	14.7%	15.6%	9.6%	11.1%	–	8.8%
	实际发生事件	71.8%	88.2%	31.3%	57.7%	61.1%	96.2%	70.0%
	期望状态	–	8.8%	21.9%	–	–	–	7.5%
	期望未发生事件	–	–	–	–	–	–	–
	期望发生事件	30.8%	8.8%	21.9%	23.1%	–	15.4%	18.5%
原因	未执行操作	50.0%	30.8%	25.0%	14.3%	14.3%	–	25.4%
	错误操作	50.0%	69.2%	75.0%	85.7%	85.7%	100.0%	74.6%
修改	插入	36.4%	33.3%	–	40.7%	50.0%	20.0%	36.7%
	删除	9.1%	11.1%	–	–	6.3%	–	5.1%
	替换	54.5%	55.6%	80.0%	59.3%	43.8%	–	58.2%

对于补丁解释，本实证实验的发现有助于生成补丁解释。本章提出的补丁解释模型捕获了补丁解释中应该涉及的基本元素以及表示它们的表达形式，为解释生成奠定了基础。本实证研究的定量研究揭示了哪些元素和表达形式更常用，哪些元素具有相关性，哪些元素是具有排他性的。本章还发现了一些关于补丁审阅者的特性，例如，可以假设审阅者熟悉项目。虽然生成完整的解释是很困难的一件事，但本章的发现也揭示了可以只生成部分解释，对于补丁审阅者来说也是有用的。例如，如果一个补丁修改了项目中的多个位置，则可以指出核心修改，这样的任务比生成完整的解释更容易自动化。

对于衡量缺陷报告质量，目前有很多研究是针对缺陷报告质量建模[120]，本章的结果有助于扩展这些模型并对拉取请求质量建模。许多公司为缺陷报告提交者提供指导，本章的结果有助于进一步指导提交者提交拉取请求。

对于定位技术，本实证实验发现一个新的可被定位技术采纳的程序元素粒度：变量。本章同时确认了在自动定位技术中，相比于细粒度的定位结果，例如语句级别，程序员更愿意采纳粗粒度的定位结果，例如函数级别。

对于程序语言设计，编程语言设计的一个目标是让开发人员以一种自然的方式说

明程序。本章的结果发现，条件通常是通过事件序列来说明，但主流编程语言不支持这种说明，这表明语言设计者可以考虑一个新特性。

2.5 实证实验的有效性

实证实验中不可避免的会存在一些因素影响实验结论的有效性，本节简要介绍影响实证实验有效性的因素，并且说明本实验的设置如何缓解这些因素以保证实验发现的有效性。

*内部威胁*在于实证实验中人工审阅拉取请求。为了减少人工审阅过程的主观性，实验采用了开放编码原则，两位作者独立分析数据集生成补丁解释模型直至分析结果达到饱和，即当分析超过 20 个拉取请求之后，该模型内的元素及表达形式都不会发生变化。

另外一个*内部威胁*在于实证实验中只采用了被合并的拉取请求。本实验假设合并的拉取请求是补丁提交者提供了充分解释的，但是即使一个拉取请求没有被充分解释，也可能会被合并。为了缓解这方面的威胁，本实验研究了来自 6 个不同项目的 300 个拉取请求。此外，实验中的很多发现，例如补丁解释模型中包含哪些元素，是不会受到个别补丁解释信息是否充足的影响。本实验审阅的约一半拉取请求都是由项目中的核心开发者提交，事实上，一个拉取请求是否来自当前项目的核心开发者也不会影响到补丁解释模型。

*外部威胁*在于收集的数据集。由于每个项目都有各自的特点，项目参与者提交的拉取请求也具有各自的特点，本实验中的结果可能无法泛化到其它项目中。为了缓解这方面的威胁，实验中的数据集包含来自不同组织，不同领域的各种类型的项目。然而，实验中的数据集只包含了 GitHub 中的 Java 项目，目前还需要更多的研究来理解当前结论是否能推广到其它编程语言和其它开源平台中的项目上。

*构造威胁*在于使用的统计度量方式。本实验采用 *lift* 作为标准来度量两个元素之间的相关性。*lift* 可能不是最充分的度量方式，并且当前还存在很多其它的度量方式，然而，其它度量方式可能不适合当前的场景，例如，Pearson 相关性分析[121] 可以度量两个元素之间是否线性相关，但是本实验中解释模型中的元素不具有线性关系。未来可以尝试使用更多种度量方式统计元素之间的相关性。

2.6 讨论与小结

本章分析了开发者在开源网站上提交的补丁解释信息，总结了有助于判别补丁正确性和辅助补丁审阅的补丁特性，用于指导设计补丁审阅提升技术。具体而言，本章总结出了一个补丁解释通用模型，包含了五个核心元素，分别代表了开发者从不同方

面解释补丁正确性，具体包括两类：静态信息（补丁位置信息、缺陷原因以及补丁修改信息）和动态信息（缺陷触发条件及补丁应用前后结果信息），该发现说明，补丁的静态信息与动态信息均可用于补丁正确性判别。同时，实证研究发现补丁审阅者通常是熟悉项目内容的，向审阅者提供不完整或非正式的补丁解释信息，比如补丁特性，也有助于开发者理解补丁及缺陷。除此之外，本章还进行了定量实验，分析了解释补丁元素出现的频率、元素之间的相关性与排他性等统计数据。实验结果表明，开发者倾向于选择 2-3 个元素解释一个补丁，说明相比于使用单一元素，开发者更倾向于多种元素的结合解释补丁，启示补丁审阅提升技术可结合补丁的多种特性辅助开发者判别补丁的正确性，辅助审阅。

上述发现将直接指导本文所提出的针对自动生成补丁的审阅提升技术。值得注意的是，本章的实证研究的发现不仅仅为本文设计的补丁审阅提升技术提供指导，还对补丁解释生成、衡量缺陷报告质量等其它研究方向产生了很好的指导意义。

第三章　基于动静态信息结合的补丁过滤技术

3.1　引言

如第1.1.3节所述，目前自动修复技术本质上无法保证生成补丁的正确性，人工审阅补丁不可或缺。而给定正确性未知的补丁时，开发者的修复效率与正确率并不一定会提升，甚至可能会下降，自动修复技术无法发挥预期的作用。

为辅助开发者在给定候选补丁情况下高效准确修复缺陷，提升自动修复技术的实用价值，本文提出补丁审阅提升技术。具体而言，在开发者人工审阅补丁之前，可通过自动化技术帮助开发者过滤无效信息（即错误补丁）的方式减轻开发者人工审阅的负担。近些年来，研究者已经提出了许多判别补丁正确性的方法[67,86,96,100,104,122,123]。从使用信息源上可以将这些工作划分为两类：需要动态执行测试用例的技术与基于静态的代码特征的技术。其中基于动态信息的技术或是额外生成测试用例，根据是否可以触发缺陷判别正确性[122,123]；或是收集运行时的覆盖信息，根据补丁应用前后通过与失败测试用例覆盖变化，启发式判断补丁正确性[86]。基于静态信息的技术根据代码修改模式[67] 或是应用补丁前后的代码片段在语法和语义上的相似性[96,100,104] 判断补丁正确性。虽然目前的这些技术取得了一些进展，但是其过滤错误补丁的能力有待提升。特别地，已有工作存在以下局限性：（1）在考虑静态信息时，已有基于表示学习的技术只是将补丁当做字符串，未充分利用代码信息，例如代码结构信息、修改上下文信息、以及细粒度的修改信息；（2）已有技术都使用了单一信息源，即静态信息或动态信息，未同时考虑补丁的这两种信息源。然而，已有研究[86,124,125] 和本文的实证研究（见第二章）均发现，代码的结构信息、修改上下文等静态信息和补丁的动态覆盖信息对于补丁正确性判别都具有帮助。

本章提出基于历史数据学习的补丁正确性判断方法 *Ceres*，分别对补丁的两种信息源，即静态修改特征与动态覆盖特征，进行学习，最终判别结果由修改特征学习模型与覆盖特征学习模型两部分决定。特别地，在考虑修改信息时，将补丁应用前后所在函数的代码表示为一个基于抽象语法树 (即 AST) 的修改表征图。该表征方法可以显示以代码节点为粒度的修改信息、保留补丁修改的上下文信息、代码结构信息，例如，语句类型、变量名称与变量值。然后通过 S-Transformer 模型[126] 对该表征图进行编码和分析，从该图中学习到对判别补丁正确性有效的关键修改特征。在考虑动态信息时，提出覆盖表征图，即将源代码与测试分别抽象为不同的节点，并根据覆盖情况在两种节点之间建立四种类型的边（代码节点在补丁应用前后都未被覆盖，在补丁应用前后都被覆盖，只在补丁应用前覆盖和只在补丁应用后覆盖）。然后通过 RAT 模型[127] 对该

表征图进行分析学习关键的覆盖特征，同时结合静态修改模型的学习结果，对补丁正确性进行判别。

已有基于学习的方法在验证补丁正确性判别效果时采用了跨补丁的训练方法，即同一个缺陷的不同补丁可能分别出现了测试集和训练集中。而真实场景中，往往使用历史上其它缺陷的修复补丁进行模型训练进而预测新的缺陷的补丁，即同一缺陷的补丁只会出现在训练集或是测试集中。本章同时采用了跨补丁与跨缺陷两种验证场景对 *Ceres* 的判别效果进行验证。实验结果表明，在跨补丁划分场景下，*Ceres* 的过滤错误补丁的数量相对于只使用静态信息的技术 ODS[100] 和只使用动态信息的技术 PatchSim[86] 分别提升了 12.9% 和 34.6%。一个有趣的发现是，相比于跨补丁划分场景，基于学习的技术（ODS 与 *Ceres* ）在跨缺陷划分场景下的判别错误补丁效果都显著下降了，过滤错误补丁数量分别下降了 96.8% 和 74.3%。该发现为未来基于学习的补丁正确性判别技术研究提供了启示：跨缺陷划分场景与实际场景更契合，未来基于历史数据学习的补丁过滤技术需要考虑如何提升跨缺陷场景下的判别效果。虽然 *Ceres* 在跨缺陷场景下过滤错误补丁数量有所下降，不过判别效果仍优于 ODS 与 PatchSim。

本章的组织结构如下：第3.2节使用真实的补丁示例说明修改特征与覆盖特征的重要性，阐明方法的动机；第3.3节对方法中静态修改特征学习模型与动态覆盖特征学习模型进行详细描述；第3.4节对提出的方法进行实验验证；第3.5节对本文涉及到方法的局限性和适用场景进行讨论；第3.6对本章的内容进行总结。

3.2 示例说明

为了更好的说明静态修改信息与动态覆盖信息对于补丁正确性判别的重要性，本小节将使用两个示例分别进行说明，并进一步提出同时基于静态信息与动态信息的补丁正确性判别技术。

静态修改信息与补丁正确性判别。由于测试用例集合的不完备，修复工具生成能通过测试用例的错误补丁往往具有一定的相似性。图3.1是修复工具 CapGen[43] 为 Defects4J[52] 数据集中缺陷 Lang59 生成的错误补丁。缺陷 Lang59 是在一个字符串的后面增加字符串时，传入错误的长度变量导致发生数组越界，错误补丁一是为该字符串扩充了固定长度（即 4），使其不再发生越界问题。应用该补丁之后所有测试用例的覆盖信息都不会发生改变，因而无法通过动态覆盖信息判别其正确性。此时，CapGen 也为该缺陷生成了其它相似的错误补丁，如图3.2所示，错误补丁二与错误补丁一修复位置的上下文是一致的，并且它们的修复模式都是将 `width` 变量替换成了一个数字，若已知补丁二为错误补丁的情况下，可通过分析其静态信息，例如修复语句的语法特征，替换了同一变量的修复特征，修复语句的父节点为条件语句的上下文特征，或者将两

```
public StrBuilder appendFixedWidthPadRight(Object obj, int width,
   char padChar) {
    if (width > 0) {
        - ensureCapacity(size + width);
        + ensureCapacity(size + 4);
        ...
    }
}
```

图 3.1　CapGen[43] 为缺陷 Lang59 生成的错误补丁一

```
public StrBuilder appendFixedWidthPadRight(Object obj, int width,
   char padChar) {
    if (width > 0) {
        - ensureCapacity(size + width);
        + ensureCapacity(size + 5);
        ...
    }
}
```

图 3.2　CapGen[43] 为缺陷 Lang59 生成的错误补丁二

个补丁转换为词嵌入的向量比较相似性等方法，由历史数据补丁二判别补丁一为错误补丁。

动态覆盖信息与补丁正确性判别。错误补丁除了可能在静态特征上具有相似性，其应用前后对测试用例覆盖信息的影响也具有一定的相似性。图3.3和图3.4分别为修复工具 Kali[128] 为 Defects4J 中缺陷 Chart12 和修复工具 TBar[54] 为缺陷 Lang39 生成的错误补丁。从修改的静态特征来看，这两个补丁完全不一致，例如，缺陷 Chart12 的错误补丁增加了一个 `If` 块，而缺陷 Lang39 的错误补丁修改了原代码中的 `For` 循环中的条件判断，通过分析这两个补丁的静态修改特征无法判断补丁的正确性。然而，这两个补丁应用前后，测试用例覆盖信息的变化具有一定的相似性，特别地，这两个补丁应用之后不仅失败测试用例无法覆盖到修改位置之后的部分代码行，所有通过测试也无法覆盖到修改位置之后的部分代码行。基于之前的工作[86] 的发现：一个正确的补丁应用前后通过测试用例的覆盖信息应该是相似的，失败测试用例的覆盖信息应该发生改变。因此，缺陷 Chart12 补丁的正确性的判别，可通过学习历史数据中错误补丁应用前后通过与失败测试用例的改变情况完成。

以上两个示例说明，错误补丁在修改特征与覆盖特征上可能都具有相似性，因此静态修改信息与动态覆盖信息对于补丁正确性的判别同样重要，基于此观察，本文提出同时基于这两个特征对历史数据学习的补丁正确性判别方法：*Ceres* 。接下来将详细介绍该方法。

```
    public boolean hasListener(EventListener listener) {
        + if (true)
            + return true;
        List list = Arrays.asList(this.listenerList.getListenerList());
        return list.contains(listener);
     }
```

图 3.3　Kali[128] 为缺陷 Chart12 生成的一个错误补丁

```
    int increase = 0;
    - for (int i = 0; i < searchList.length; i++) {
    + for (int i = 0; i == searchList.length; i++) {
         int greater = replacementList[i].length() -
             searchList[i].length();
         if (greater > 0) {
              increase += 3 * greater;
         }
    }
    increase = Math.min(increase, text.length() / 5);
             ...
```

图 3.4　TBar[54] 为缺陷 Lang39 生成的一个错误补丁

3.3　方法介绍

给定缺陷程序、补丁、至少包含一个失败测试用例的测试集合和补丁应用前后的测试覆盖信息，*Ceres* 的工作流程如图3.5所示，主要分为两个步骤：静态修改特征学习和动态覆盖特征学习。其中，静态修改特征学习首先将提取补丁所在函数的抽象语法树以及修改表达式的抽象语法树，并为补丁定义新的修改操作节点，通过操作节点定义函数的抽象语法树和修改表达式的连接方式，最终构建成修改特征图，采用 S-Transformer 模型对修改特征图进行结构编码；动态覆盖特征学习步骤将所在函数表达式的抽象语法树根节点和测试用例作为图中的节点，以覆盖关系为边，构建覆盖特征图，采用 RAT 模型对补丁修改所导致的测试覆盖的变化进行学习，并建立与修改特征学习结果的关系，最终 *Ceres* 根据两个步骤的结果对补丁正确性进行判断，接下来分别介绍以上步骤。

3.3.1　静态修改特征学习

3.3.1.1　修改特征图

本小节介绍如何以图的形式表示修改信息并作为 S-Transformer 模型的输入。

定义 1　修改特征图 给定同时包含补丁应用前后代码的函数 m，$\mathcal{G}^m_{ast} = (\mathcal{V}^m_{ast}, \mathcal{E}^m_{ast})$ 代表其对应的抽象语法树（即 *Abstract Syntax Tree*(*AST*)），其中，$\mathcal{V}^m_{ast}$ 和 $\mathcal{E}^m_{ast}$ 代表所

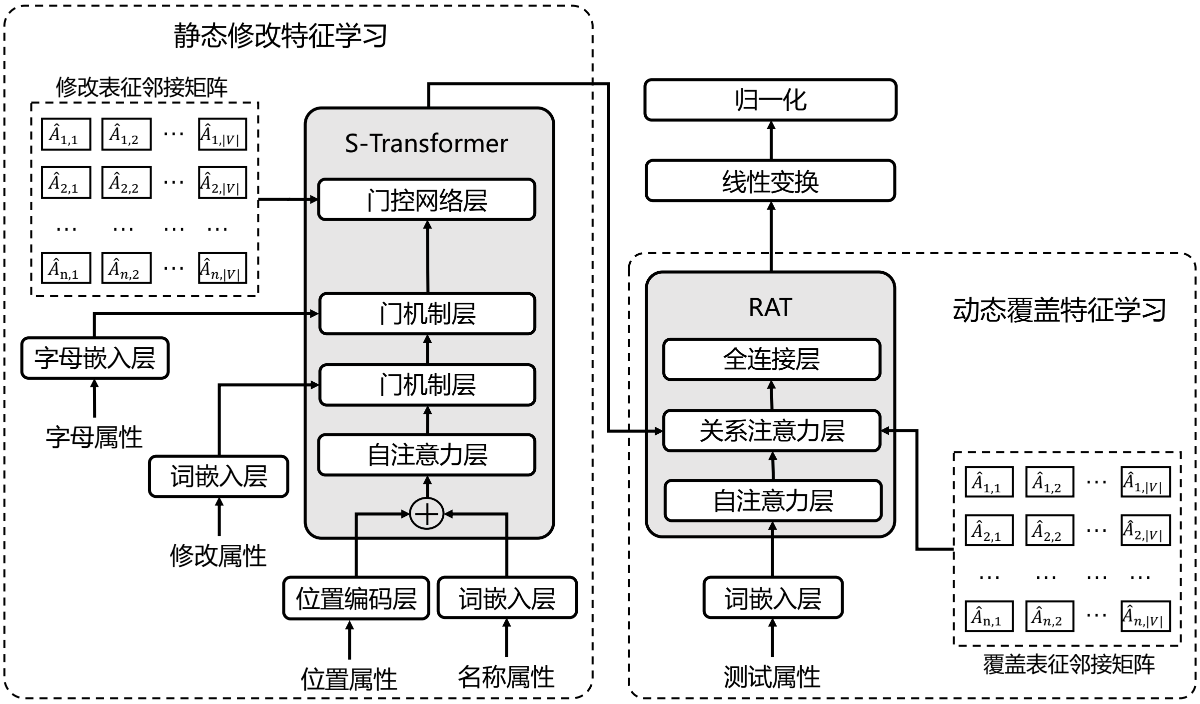


图 3.5　*Ceres* 概览

有代码节点和代码边。特别地，对于每个代码节点 $v_i \in \mathcal{V}_{ast}^m$，都具有两类节点属性：$attr_n(v_i)$ 和 $attr_m(v_i)$，分别表示节点名称属性的集合与修改属性的集合。其中 $attr_n(v_i)$ 标注了每个节点的名称，例如 *IfStatement*, *Literal* 等。$attr_m(v_i) \in \{same, add, delete\}$，分别表示该节点修改前后不变，修改后新增或是被删除。该图为无向无权图。

例 3.1 图*3.6*为图*3.1*中补丁示例的部分修改表征图。此图是将补丁应用前的抽象语法树与补丁应用后的抽象语法树合并而得到的。其中，v_1 节点为该修改函数的根节点，v_4 节点为合并后的抽象语法树中新增的一个节点，它具有两个子节点 v_5 和 v_6，分别代表了补丁应用前的代码 `size+width` 和补丁应用后的代码 `size+4`。图中每个节点都具有两个属性：名称属性和修改属性，v_4 节点的名称属性为自定义名称，其余节点的名称属性均为 *AST Parser* 解析而来，v_4 及其所有的父节点的修改属性均为 *same*，表示修改前后代码节点未发生变化，v_5 和 v_6 及其所有子节点的修改属性分别为 *delete* 和 *add*，分别表示补丁应用前后的代码。

相比于类或文件范围内所有代码的抽象语法树，*Ceres* 使用函数范围内的抽象语法树构建修改表征图，既保留了补丁修改的上下文信息，又极大缩小了表征图的规模（即图的节点数目和边的数目），降低了学习模型的计算代价。节点名称属性和修改属性表示了每个节点在修改前后是否发生变化以及节点的具体变化。接下来本文将介绍修改表征图的构造以及这两类属性的收集。

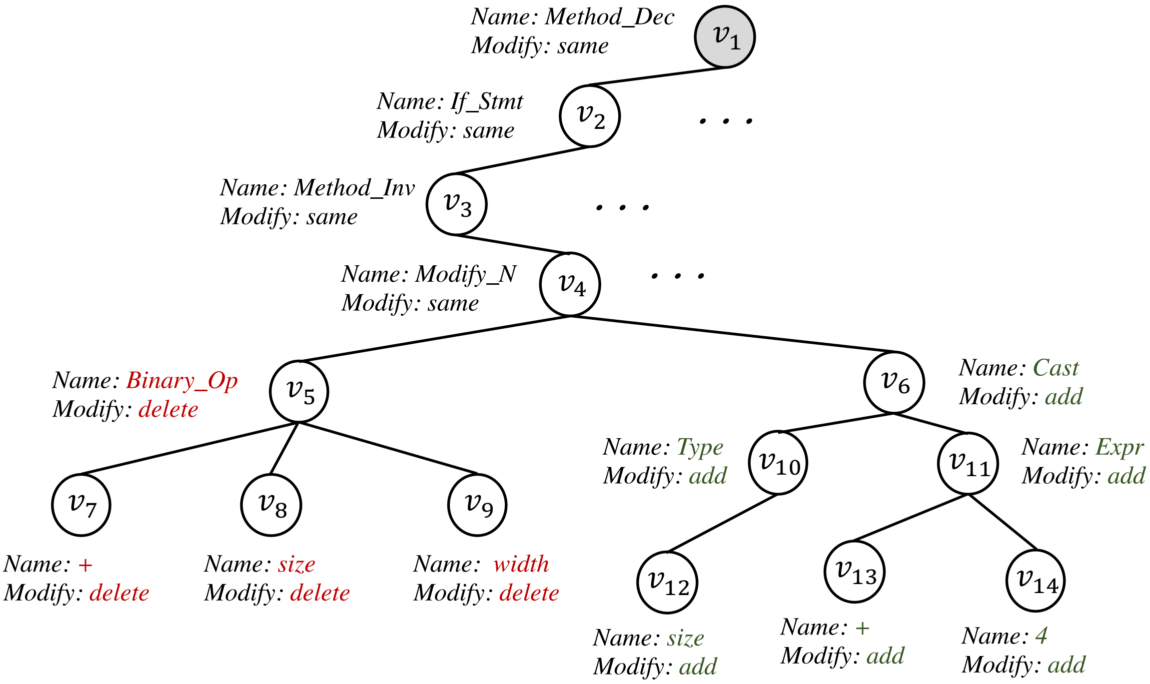


图 3.6　缺陷 Lang59 的错误补丁一的修改表征图

3.3.1.2　模型输入

给定缺陷程序和补丁，本文按照下述步骤构造修改表征图 $\mathcal{G}_{ast}^{m}$。

- *Ceres* 使用 Javalang 工具[129] 解析补丁修改函数在应用补丁前后的代码，从而获取修改前后补丁所在函数的抽象语法树 (*AST*)。
- *Ceres* 采用算法1 将补丁应用前后的函数表示为一棵抽象语法树，构建修改表征图。该算法的输入为补丁应用前后的函数的抽象语法树 $root_b, root_f$，若两棵抽象语法树完全一致则直接返回（第 2-4 行），若两棵语法树的根节点不一致，则返回这两个根节点（第 5-7 行），接下来遍历两个抽象语法树的子节点（第 8-38 行），若两个抽象语法树子节点数目一致，则依次递归调用 Construct 函数对比子节点，若返回节点不为 *None* 时，新增 `Modify_N` 节点，表示补丁是将子节点 $node_a$ 修改为了 $node_b$（第 8-14 行），注意，ConstructNode($root_{label}, childList$) 表示新建一个名为 $root_{label}$ 的根节点，并将 $childList$ 中的节点作为根节点的子节点，最后返回该根节点；若两个抽象语法树子节点数目不一致，则依次在修改后的抽象语法树 $root_f.child$ 中寻找和修改前抽象语法树 $root_b.child$ 中一致的节点，并建立映射关系 map(第 16-25 行)，然后以 $root_b.child$ 中未出现在映射关系表中的节点为子节点，构建名为 `Delete_N` 的父节点，以 $root_f.child$ 中未出现在映射关系表中的节点为子节点，构建名为 `Add_N` 的父节点，最后更新 $root_b.child$ 的子节点列表（第 26-37 行）。至此，补丁应用信息已经被完全表示

在了抽象语法树 $root_b$ 中。

- *Ceres* 在图中标注每个节点的节点名称属性和修改属性，其中，*Ceres* 使用 Javalang 解析的 AST 节点名称作为对应节点的名称属性。对于修改属性，*Ceres* 将图中以 `Delete_N` 为根节点的所有节点和 `Modify_N` 的第一个子节点及以它为根节点的所有节点的修改属性标注为 *delete*，将图中以 `Add_N` 为根节点的所有节点和 `Modify_N` 的第二个子节点及以它为根节点的所有节点的修改属性标注为 *add*，将图中剩余其它节点标注为 *same*。

为了进一步将构建的修改表征图 $\mathcal{G}_{ast}^{m}$ 转换为适合模型训练的输入格式，本文采用了邻接矩阵 A 表示图结构，特别地，对于 $\mathcal{G}_{ast}^{m}$，邻接矩阵 A 中的元素 $A_{v_i,v_j} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$ 表示节点 v_i 与节点 v_j 是否有边存在。为了避免矩阵中因度数累加而引发的梯度爆炸问题，矩阵 A 被正则化为 $\hat{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$，其中 D 为对角矩阵，即，$D_{v_i,v_j} = de(i)^{-\frac{1}{2}}$，其中 $de(v_i)$ 代表节点 v_i 的度。此外，为了将节点的名称属性和修改属性输入模型进行训练，本文分别采用属性序列 S^n 和 S^m 代表这两类属性，在名称属性序列 S^n 中，$S_{v_i}^n$ 代表了节点 v_i 对应的名称属性，在修改属性序列 S^m 中，$S_{v_i}^m$ 代表了节点 v_i 对应的修改属性，取值为 $\{same, add, delete\}$ 中的一种。

至此，修改表征图 $\mathcal{G}_{ast}^{m}$ 已经被转换为邻接矩阵 $\hat{A}$ 和两个属性序列 S^n 和 S^m，并作为 S-Transformer 模型的输入。

3.3.1.3 S-Transformer 模型

嵌入层。嵌入层将输入的文字序列编码为矩阵，作为神经网络的输入，具体包括以下三种嵌入。

词嵌入。词嵌入层将属性序列 S^n 和 S^m 分别编码为一个名称属性矩阵 $X_s^n \in \mathbb{R}^{|\mathcal{V}\times d|}$ 和修改属性矩阵 $X_s^m \in \mathbb{R}^{|\mathcal{V}\times d|}$，其中 d 代表每个属性词被编码的大小。

字母嵌入。在代码中，经常会出现相似的单词有相似的字符，例如，“expression” 和 “expressions”，为了利用这种特性，本文引入字母嵌入，对于属性序列 S^n 中每个名称属性 n_i，先分解为字母表示：$c_1^{n_i}, c_2^{n_i}, \cdots, c_s^{n_i}$，其中 s 代表 n_i 中字母长度，然后利用公式3.1将其编码为名称属性字母表示 $n_i^{(c)}$，最终所有名称属性组成矩阵 $X_s^c \in \mathbb{R}^{|\mathcal{V}\times d|}$，其中 d 代表每个属性词被编码的大小。

$$\boldsymbol{n}_i^{(c)} = W^{(c)}[\boldsymbol{c}_1^{(n_i)};\cdots;\boldsymbol{c}_s^{(n_i)}] \tag{3.1}$$

位置编码。Transformer 模型中的注意力机制不包含位置信息，即一个序列中的词在不同位置时，学习结果是一样的，为了表示词在序列中相对或绝对位置的信息，需要引入位置编码，特别地，本文沿用已有工作[130] 中变量的设置，使用公式3.2计算第 i 个词在第 b 次迭代中的位置编码，其中 $p_{i,b}[.]$ 索引了向量 $p_{i,b}$ 的维度，d 代表每个词

Algorithm 1 构造包含修改前后代码的抽象语法树

Require: 修改前函数的抽象语法树的根节点 $root_b$, 修改后函数抽象语法树 $root_f$。

Ensure: 包含修改前后代码的函数根节点 $root_b$。

```
function CONSTRUCT($root_b, root_f$)
    if $root_b$ 的抽象语法树与 $root_f$ 的抽象语法树一致 then
        return $None, None$
    end if
    if $root_b \neq root_f$ then
        return $root_b, root_f$
    end if
    if $root_b.child$ 与 $root_f.child$ 节点个数一致 then
        for $i = 0 \to len(root_b.child)$ do
            $node_a, node_b \leftarrow$ CONSTRUCT($root_b.child[i], root_f.child[i]$)
            if $node_a \neq None$ then
                $root_a.child[i] \leftarrow$ ConstructNode ($Modify\_N, [node_a, node_b]$)
            end if
        end for
    else
        $lastMatch_j \leftarrow 0, map \leftarrow \{\}$
        for $i = 0 \to len(root_b.child)$ do
            for $j = lastMatch_j \to len(root_f.child)$ do
                if $root_b.child[i] == root_f.child[j]$ then
                    $map[i] \leftarrow j$
                    $lastMatch_j \leftarrow j$
                    $break$
                end if
            end for
        end for
        $lastMatch_i \leftarrow -1, lastMatch_j \leftarrow -1, childList \leftarrow []$
        for $i = 0 \to len(root_b.child)$ do
            if $map[i] \neq \emptyset$ then
                $j \leftarrow map[i]$
                $childList.add($CONSTRUCTNODE$(Delete\_N, [root_b.child[lastMatch_i + 1] \sim root_b.child[i - 1]]))$
                $childList.add($CONSTRUCTNODE$(ADD\_N, [root_b.child[lastMatch_j + 1] \sim root_b.child[j - 1]]))$
                $childList.add(root_b.child[i])$
                $lastMatch_i \leftarrow i$
                $lastMatch_j \leftarrow j$
            end if
        end for
        $root_b.child \leftarrow childList$
    end if
end function
```

被编码的大小，与词嵌入层属性词编码大小一致。然后将序列中的词组成一个位置矩阵 $X_p \in \mathbb{R}^{|\mathcal{V}\times d|}$，最终与名称属性序列 S^n 词嵌入编码的矩阵 X_s^n 拼接，作为自注意力层的输入。

$$\begin{aligned} p_{b,i}[2j] &= \sin((i+b)/(10000^{2j/d})) \\ p_{b,i}[2j+1] &= \cos((i+b)/(10000^{2j/d})) \end{aligned} \tag{3.2}$$

神经元循环模块。*Ceres* 在该模块中进行六次循环迭代，每次迭代都经过三个神经元层（自注意力层（self-attention），门机制层 (gating mechanism)，门控图网络层 (gated graph neural network)），

自注意力层。自注意力层遵从 Transformer[131] 的架构，并且使用多头注意力机制来捕获长依赖信息。一个 Transformer 块通过多头注意力机制学习非线性特征，最终产生一个矩阵 $Y_b = [\boldsymbol{y}_{b,1}, \boldsymbol{y}_{b,2}, \cdots, \boldsymbol{y}_{b,V}]W_v$，其中 $Y_b \in \mathbb{R}^{|\mathcal{V}\times d|}$。综合多个注意力神经元的输出特征由公式3.3计算，其中 H 代表头的数目，W_h 是权重。注意力层被应用在每个头 $head_t$ 中，由公式3.4计算，其中其中，$d_k = d/H$ 是一个正则化系数，H 表示头的数目，d 表示隐藏层的大小。Q, K 和 V 由公式3.5来计算，其中，$W_Q, W_K, W_V \in \mathbb{R}^{|d\times d_k|}$ 都是模型参数，x_i 为自注意力层的输入，对于第一次迭代，输入为位置矩阵 X_p 与名称属性矩阵 X_s^n 拼接之后的矩阵，对于其它次迭代，输入为上次迭代后的输出与当前迭代的位置矩阵的向量和。

$$Y_b = \mathrm{concat}(head_1, \cdots, head_H)W_h \tag{3.3}$$

$$head_t = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.4}$$

$$[Q, K, V] = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_V]^T [W_Q, W_K, W_V] \tag{3.5}$$

自注意力层的输出 Y_b 将作为门机制层的输入，整合修改特征与字母特征。

门机制层。每个节点的输入特征都包含了丰富的其它信息，为了突出这些信息的作用，本文采用了两层门机制[132] 来分别加入修改特征，以及字母特征。门机制以 $\boldsymbol{q}, \boldsymbol{c}_1, \boldsymbol{c}_2$ 三个向量为输入，通过以 $\boldsymbol{q}$ 为控制向量的多头机制将 $\boldsymbol{c}_1$ 与 $\boldsymbol{c}_2$ 合并，其具体的计算方式如以下公式所示，

$$\begin{aligned} \alpha_i^{c_1} &= exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^{c_1})/\sqrt{d_k} \\ \alpha_i^{c_2} &= exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^{c_2})/\sqrt{d_k} \\ \boldsymbol{h}_i &= (\alpha_i^{c_1}\boldsymbol{v}_i^{c_1} + \alpha_i^{c_2}\boldsymbol{v}_i^{c_2})/(\alpha_i^{c_1} + \alpha_i^{c_2}) \end{aligned} \tag{3.6}$$

其中，$d_k = d/H$ 是一个正则化系数，H 表示头的数目，d 表示隐藏层的大小，$\boldsymbol{q}_i$ 是由一个全连接层在控制向量 $\boldsymbol{q}$ 上计算得来，$\boldsymbol{k}_i^{c_1}$ 和 $\boldsymbol{v}_i^{c_1}$ 是由另外一个全连接层在向量 $\boldsymbol{c}_1$ 上计算得来，$\boldsymbol{k}_i^{c_2}$ 和 $\boldsymbol{v}_i^{c_2}$ 也是由相同的层且带有不同参数的全连接在向量 $\boldsymbol{c}_2$ 上计算得来。

对于第 b 次迭代，本文使用上层神经网络对于节点 y_i 的输出 $\boldsymbol{y}_b^i$ 作为控制向量 $\boldsymbol{q}$ 来合并它本身与其它特征（修改特征与字母特征），特别地，在第 b 次迭代，对于合并修改特征的节点 y_i 的输出可由公式3.7计算得来，其中，$\boldsymbol{y}_b^i$ 是自注意力层的输出，$\boldsymbol{m}_{y_i}$ 代表节点 y_i 的修改特征。在第 b 次迭代，对于合并字母特征的节点 y_i 的输出可由公式3.8计算得来，其中，$\boldsymbol{c}_{y_i}$ 代表节点 y_i 的字母特征。

$$\boldsymbol{m}_b^{i\,\prime} = \mathrm{Gating}(\boldsymbol{y}_b^i, \boldsymbol{y}_b^i, \boldsymbol{m}_{y_i}) \tag{3.7}$$

$$\boldsymbol{m}_b^i = \mathrm{Gating}(\boldsymbol{m}_b^{i\,\prime}, \boldsymbol{m}_b^{i\,\prime}, \boldsymbol{c}_{y_i}) \tag{3.8}$$

门机制层的输出向量 m_b^i 将作为门控图网络层的输入，整合图中邻居节点的信息。

门控图网络层。和传统的图网络一样，为了提取代码图的结构信息，每个节点的编码信息应该整合其它邻居节点的编码信息。该层首先使用邻接矩阵计算邻居节点状态信息，然后使用门机制合并当前节点的状态信息与当前节点所有邻居节点的状态信息。

对于修改表征图 $\mathcal{G}$ 中的每个节点 y_i，使用公式3.9计算其邻居节点信息。

$$\boldsymbol{p}_b^i = \sum_{y_i \in \mathcal{G}} \hat{A}_{y_i, y_j} \boldsymbol{m}_b^j \tag{3.9}$$

为了将邻居节点的状态更新到当前节点上，本文对节点的当前状态 $\boldsymbol{m}_b^i$ 和邻居节点编码信息 $\boldsymbol{p}_b^i$ 应用相同的子层，具体而言，使用 $\boldsymbol{m}_b^i$ 作为控制向量 $\boldsymbol{q}$ 来合并这两个向量，由公式3.10完成状态更新。

$$\boldsymbol{o}_b^i = \mathrm{Gating}(\boldsymbol{m}_b^i, \boldsymbol{m}_b^i, \boldsymbol{p}_b^i) \tag{3.10}$$

为了避免训练过程中出现梯度消失的问题，每两个子层之间引入了残差连接[133]与层级归一化[134]。本模型中的输出是模型对静态代码修改特征学习的结果，同动态覆盖特征共同作为下一阶段模型（RAT 模型）的输入。

3.3.2　动态覆盖特征学习

3.3.2.1　覆盖特征表示

本小节介绍如何以图的形式表示代码节点与测试节点之间的覆盖信息并作为 RAT 模型的输入。

定义 2　测试表征 给定同时包含补丁应用前后修改代码的函数 m 和测试用例集合 $\mathcal{T}_{all}$，测试节点集合 $\mathcal{V}_{\mathcal{T}}$ 代表补丁应用前或应用后执行覆盖函数 m 的测试用例。其中，对于每个测试用例节点，$v_t \in \mathcal{V}_{\mathcal{T}}$，具有节点属性 $attr_r(v_t)$ 表示在补丁应用前缺陷程序上测试结果的集合，即 $attr_r(v_t) \in \{pass, fail\}$。

Ceres 只考虑测试执行覆盖到补丁修改函数的测试用例，将每个测试用例作为一个单独的节点，并通过它们在缺陷程序上的测试输出（即 $pass$ 或 $fail$）作为节点测试属性，进而区分通过测试用例与失败测试用例。

定义 3　代码表征 给定同时包含补丁应用前后修改代码的函数 m 和测试用例集合 $\mathcal{T}_{all}$，语句级别覆盖信息 $C_b(m,t)$ 和 $C_f(m,t)$ 分别代表了补丁应用前和应用后函数 m 中所有被测试用例 t 覆盖的程序语句集合，那么，补丁应用前后函数 m 中被测试用例集合 $\mathcal{T}_{all}$ 覆盖的程序语句集合 $stmt = \{C_b(m,t) \cup C_f(m,t) | t \in \mathcal{T}_{all}\}$。$\mathcal{V}_{\mathcal{S}}$ 代表该程序语句集合 $stmt$ 中抽象语法树上语句级别节点集合。

与修改特征一样，*Ceres* 只考虑补丁修改函数内测试执行覆盖信息的变化，相比于使用完整 AST 中的所有节点，*Ceres* 只考虑语句级别抽象语法树上的节点作为代码表征，因为语句级别的覆盖信息与词语级别的覆盖信息其蕴含的信息量是相同的，而后者会极大增加代码节点的数目，从而导致增加最终覆盖表征图的规模，增加了学习模型的计算代价。

定义 4　覆盖表征 给定同时包含补丁应用前后修改代码的函数 m 和测试用例集合 $\mathcal{T}_{all}$，语句级别覆盖信息 $C_b(m,t)$ 和 $C_f(m,t)$，覆盖表征为代码节点 $\mathcal{V}_{\mathcal{S}}$ 与测试节点 $\mathcal{V}_{\mathcal{T}}$ 之间边的集合 $\mathcal{E}_{cov}$，即 $\mathcal{E}_{cov} = \{< v_s, v_t > | s \in C_b(m,t) \cup C_f(m,t), t \in \mathcal{T}_{all}\}$，其中 $v_s \in \mathcal{V}_{\mathcal{S}}$ 代表语句 s 在 $\mathcal{V}_{\mathcal{S}}$ 中对应的代码节点，$v_t \in \mathcal{V}_{\mathcal{T}}$ 代表测试用例 t 在 $\mathcal{V}_{\mathcal{T}}$ 中对应的测试节点。其中，对于每条覆盖边，$e_{<v_s,v_t>} \in \mathcal{E}_{cov}$，具有权重集合 $weight_e \in \{0,1,2,3\}$，分别代表该节点在修改前后都未被覆盖，只在补丁应用前被测试节点覆盖，只在补丁应用后被测试节点覆盖和在补丁应用前后都被测试节点覆盖。

定义 5　覆盖表征图 给定同时包含补丁应用前后修改代码的函数 m 和测试用例集合 $\mathcal{T}$，语句级别覆盖信息 $\mathbb{C}$，覆盖表征图包含了 $\mathcal{G}^m_{cov} = (\mathcal{V}, \mathcal{E}_{cov})$，其中节点 $\mathcal{V} = \{\mathcal{V}_{\mathcal{S}} \cup \mathcal{V}_{\mathcal{T}}\}$ 是代码节点与测试节点的集合，$\mathcal{E}_{cov}$ 为覆盖表征边的集合。该图为有向有权图。

例 3.2 图3.7为图3.4中示例补丁的覆盖表征图，其中 v_a,v_b,v_1 到 v_7 为代码节点，v_1 到 v_7 依次对应补丁代码中的第 1 行到第 6 行及第 9 行，v_a,v_b 未在代码示例中显示，v_8 和 v_9 均为测试代码，分别具有测试结果属性 *fail* 和 *pass*。图中连接代码节点与测试节点的边有四种类型，分别使用四种不同的线表示，代表了补丁应用前后测试节点不同的覆盖情况。

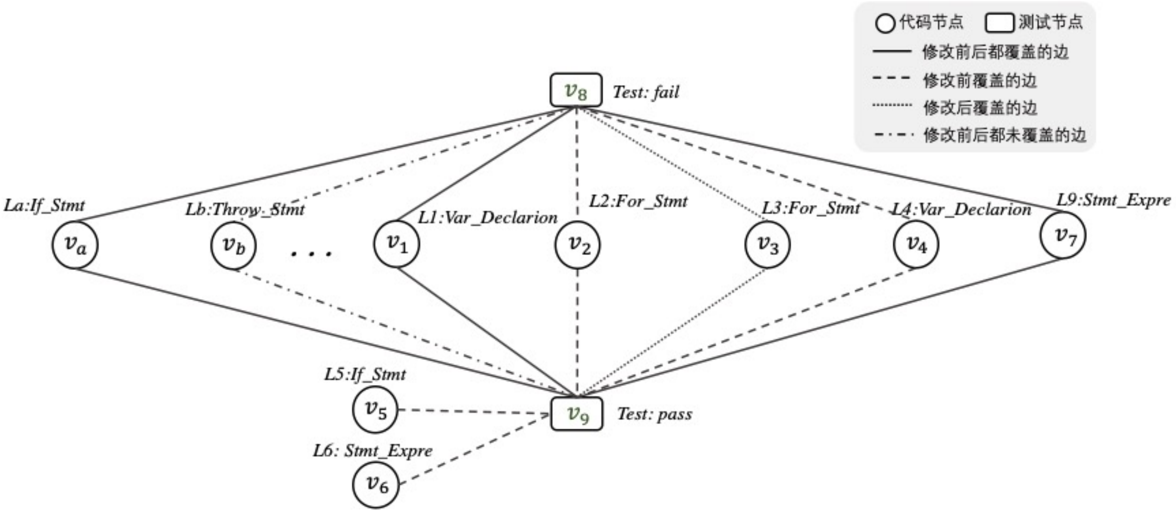


图 3.7 缺陷 Lang39 的错误补丁的部分覆盖表征图

3.3.2.2 模型输入

在给定缺陷程序、补丁、至少包含一个失败测试用例的测试用例集合以及语句级别的覆盖信息的前提下，本文按照下述步骤构造覆盖表征图 $\mathcal{G}_{cov}^{m}$。

- *Ceres* 在静态修改特征学习步骤中获取的补丁应用前后所在函数的抽象语法树中获取所有的语句级代码节点。
- *Ceres* 根据测试覆盖信息将测试节点与语句级代码节点进行连接，删除没有覆盖任何语句节点的测试节点。
- *Ceres* 根据补丁应用前后的覆盖信息，对图中每条边进行权重标记。
- *Ceres* 根据测试用例在缺陷程序上执行的结果在图中对测试节点的测试属性进行标注。

根据上文描述，*Ceres* 在覆盖特征学习中只考虑语句级别的代码节点信息，根据 Javalang 工具解析结果，共有 14 种代码节点。由于基于注意力机制的神经网络所能承载的节点数是有限制的，对于那些具有上千通过测试用例的缺陷，如果将其全部通过测试用例的覆盖信息输入神经网络会超出节点数的限制，进而影响网络的学习效果，因此需要对通过测试用例进行选择。由于已有工作[86] 已经说明补丁应用前后通过

测试用例覆盖信息的改变程度可用于判断补丁正确性的指标之一，特别地，如果所有通过测试用例中，应用补丁前后覆盖信息最大改变量仍然小于一定阈值，那么该补丁为正确补丁。因此，本文优先考虑应用补丁前后覆盖信息改变程度较高的通过测试用例。直观上，覆盖信息改变多的通过测试用例可以说明补丁对通过测试用例的影响程度，影响程度越小，正确的可能性越高。本文采用已有工作中[86] 类似的度量方式，如公式3.11所示，b, f 分别为一个测试用例在补丁应用前后缺陷函数上的覆盖节点集合，$|common(b, f)|$ 代表这两个集合公共节点个数，最终，一个测试用例在补丁应用前后覆盖信息改变分数被正则化为 0 到 1 之间的值，分数越大，说明改变程度越高。

$$score(b,f) = 1 - \frac{|common(b, f)|}{max(|b|, |f|)} \tag{3.11}$$

同静态修改特征学习一样，为了将构造的覆盖表征图 $\mathcal{G}^m_{cov}$ 转换为适合模型训练的输入模式，本文将 $\mathcal{G}^m_{cov}$ 转换为邻接矩阵 A，对于邻接矩阵 A 中的元素 $A_{v_s,v_t} \in \{0, 1, 2, 3\}^{|\mathcal{V}_s|\times|\mathcal{V}_t|}$ 表示代码节点 v_s 与测试节点 v_t 边的权重。同时为了表示测试节点的测试属性，采用属性序列 S^t 代表该属性，$S^t_{v_t}$ 代表了测试节点 v_t 的在缺陷程序上的测试结果，取值为 $\{pass, fail\}$ 中的一种。

至此，覆盖表征图 $\mathcal{G}^m_{cov}$ 以被转换为邻接矩阵 $\hat{A}$ 和一个属性序列 S^t，与静态修改特征学习的输出一起作为 RAT 模型的输入。

3.3.2.3　RAT 模型

嵌入层。*词嵌入层*词嵌入层将测试属性序列 S^t 编码为一个测试属性矩阵 $X^t_s \in \mathbb{R}^{|\mathcal{V}_t\times d|}$，其中 d 代表每个属性词被编码的大小。

循环神经元模块。 *Ceres* 在该模块中同样进行六次迭代，每次迭代经过三个子层 (自注意力层 (self-attention)，关系注意力层（relation-attention）和全连接层 (fully-connected-layer))。

自注意力层。 RAT 模型中的自注意力层与静态修改特征学习中的自注意力层一样遵从 Transformer 架构，使用公式3.3，公式3.4以及公式3.5进行计算。其中，对于第一次迭代，输入为测试属性矩阵 X^t_s，注意，由于测试序列与顺序无关，所以不需要对测试位置进行编码。对于其它次迭代，输入为上次迭代后的输出。自注意力层的输出将作为关系注意力层的输入，并整合静态修改特征学习的结果与覆盖特征，标识为 $\mathbf{s}_1, \cdots, \mathbf{s}_{\mathcal{V}_t}$。

*关系注意力层。*为了编码覆盖表征图中边的方向和种类属性，本文采用 RAT 模型[127] 中的关系注意力机制，该机制可以将覆盖表征图中的边的属性编码至测试的表征向量中。该神经元的输入为自注意力神经元的输出，$\mathbf{s}_1, \cdots, \mathbf{s}_{\mathcal{V}_t}$，以及静态表征模型

S-Transformer 的输出，$\boldsymbol{o}_1, \cdots, \boldsymbol{o}_{\mathcal{V}}$，其具体计算方法如公式3.12所示。

$$\begin{aligned} e_{ij}^{(h)} &= \frac{\boldsymbol{s}_i W_Q^{(h)} (\boldsymbol{o}_j W_K^{(h)} + \boldsymbol{r}_{ij}^K)^\top}{\sqrt{d_k}} \\ \boldsymbol{z}_i^{(h)} &= \sum_{j=1}^{n} \alpha_{ij}^{(h)} (\boldsymbol{o}_j W_V^{(h)} + \boldsymbol{r}_{ij}^V) \end{aligned} \tag{3.12}$$

其中，$d_k = d/H$ 是一个正则化系数，H 表示头的数目，d 表示隐藏层的大小，$\boldsymbol{r}_{ij}$ 表示测试样例 i 和函数语句 j 之间的已知关系。考虑关系特征 R，每个二元关系 $\mathcal{R} \subseteq X \times X (1 \leq s \leq R)$，关系注意力机制使用公式3.13表示关系图中的每条边 (i, j)，其中 ρ_{ij}^s 为关系 $\mathcal{R}^{(s)}$ 的嵌入表示。

$$\boldsymbol{r}_{ij}^K = \boldsymbol{r}_{ij}^V = \text{concat}(\rho_{ij}^1, \cdots, \rho_{ij}^R) \tag{3.13}$$

全连接层。与传统的 Transformer 架构一致，在 RAT 模型的最后一层增加了全连接层，其计算方法如公式3.14，即先进行线性变换，然后进行 ReLU 非线性激活，最终再执行线性变化，其输入 $\boldsymbol{x}$ 为关系注意力层的输出 $\boldsymbol{z}_i^{(h)}$。

$$FFN(\boldsymbol{x}) = \max(0, \boldsymbol{x} W_1 + b) W_2 + b_2 \tag{3.14}$$

为了避免训练过程中出现梯度消失的问题，每两个子层之间引入了残差连接[133]与层级归一化[134]。

3.3.3 推导

所有的 RAT 模型在迭代之后的输出都将经过一次线性转换和 $softmax$ 激活函数。其中对于节点（即单个补丁）$v_i, \boldsymbol{z}_i$ 代表其在全连接层中最后一轮迭代之后的输出。如公式3.15所示，$\boldsymbol{z}_i$ 通过线性变换之后成为二维向量 $\boldsymbol{y}_i$，其中 $W \in \mathcal{R}^{d \times 2}$，$b \in \mathcal{R}^{1 \times 2}$。如公式3.16所示，$softmax$ 函数将二维向量 $\boldsymbol{y}_i$ 进行归一化处理，即将 $\boldsymbol{y}_i$ 的值转换为范围在 $[0, 1]$ 且和为 1 的概率分布，分别代表该节点是正确补丁和错误补丁的概率，其中 C 代表预测结果的种类。

$$\boldsymbol{y}_i = W \boldsymbol{z}_i + b \tag{3.15}$$

$$p(\text{选择第 j 个预测结果}|) = \frac{e^{y_i^j}}{\sum_{c=1}^{C} e^{y_i^c}} \tag{3.16}$$

3.3.4　损失函数

与其它二分类问题一致，本文采用交叉熵损失函数[135](如公式3.17所示)，其中，$l(v_j) \in \{1, 0\}$ 代表补丁 v_j 的真实标签，$p_c(v_j)$ 和 $p_i(v_j)$ 分别代表 *Ceres* 推导补丁 v_j 为正确补丁和错误补丁的概率，n 代表补丁的总数。

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} l(v_j)\log\left(p_c(v_j)\right) + (1 - l(v_j))\log\left(p_i(v_j)\right) \tag{3.17}$$

3.4　实验验证

3.4.1　实验设置

3.4.1.1　研究问题

- **RQ1:** *Ceres* 的效果分析。
  - **RQ1a:** 在跨补丁划分测试集的情况下，与其它补丁正确性判别技术相比，*Ceres* 的效果如何？
  - **RQ1b:** 在跨缺陷划分测试集的情况下，与其它补丁正确性判别技术相比，*Ceres* 的效果如何？
- **RQ2:** *Ceres* 部件的影响分析。
  - **RQ2a:** 修改特征学习如何影响 *Ceres* 的效果？
  - **RQ2b:** 覆盖特征学习如何影响 *Ceres* 的效果？

3.4.1.2　数据集

为了验证 *Ceres* 对补丁正确性的判别效果，本文需要一个正确性已知的补丁数据集。现有的程序修复工具大多在 Defects4J[52] 数据集上验证，因此也产生了大量的正确与错误补丁。已有补丁正确性验证工作中已经收集了大部分程序修复工具产生的补丁，如表3.1所示，本文选取了两个被广泛使用的补丁数据集（Wang 等人[96] 收集了 902 个补丁和 Xiong 等人[86] 收集了 139 个补丁），每个补丁的正确性都被仔细标记。由于程序自动修复工具只产生了少量正确补丁，这导致了数据集中正例与负例比例不均衡，为了解决数据不均衡问题，本文效仿已有工作[104]，加入了 Defects4J 中开发者提交的补丁（即正确补丁）。由于这三部分补丁来源不同，基于 Defects4J 的版本不同，数据集中可能会有重复数据，或无法通过新版本 Defects4J 的测试用例集合，本文合并这三部分补丁数据之后，去除重复数据，并去除无法通过 Defects4J(V1.2.0)①上全部测试用例的补丁。最终，本文共收集了 1069 个补丁，包含 510 个正确补丁与 559 个错误补丁。

---

① 具体版本为：Commit:29d8448。

表 3.1　实验中使用的补丁数据集

	正确补丁	**错误补丁**	**总计**
Wang 等人[96]	248	654	902
Xiong 等人[86]	30	109	139
Defects4J(开发者)[52]	356	0	356
总数据集	634	763	1397
最终数据集	510	559	1069

3.4.1.3　对比技术

根据已有实验[100] 可知，目前判别补丁正确性效果最好的两种技术分别为 PatchSim[86] 和 ODS[100]。在 RQ1 中，本文将 *Ceres* 与这两种技术进行比较，它们分别基于不同的信息源：

- 基于动态信息的启发式方法（*PatchSim*)[86]。考虑补丁应用前后的覆盖信息变化情况，并设计了启发式公式判别补丁的正确性。
- 基于静态信息的机器学习方法（*ODS*)[100]。提取补丁应用前后源代码级别三类静态特征，包括代码描述、修改模式和上下文语义共 202 个特征，采用随机森林算法对特征进行训练从而预测补丁正确性。

对于以上两种方法，本实验直接采用其 GitHub 主页上的公开实现[136,137]，其中，ODS 技术原论文中采用十折交叉验证方式，本实验采用划分训练集与测试集的方式（见第3.4.1.4节），因此本文将其代码实现修改为仅在训练集中训练，测试集中测试效果。

在 RQ2 中，本实验考虑以下两种 *Ceres* 技术的变种：

- 基于静态修复特征学习模型（$Ceres_{mod}$）。仅考虑静态修改特征及 S-Transformer 模型的训练结果。具体来说，将 S-Transformer 模型的输出结果进行推导（进行线性转换和 $softmax$ 激活函数）从而进行补丁正确性判别。
- 基于动态覆盖特征学习模型（$Ceres_{cov}$）。仅考虑动态覆盖特征及 RAT 模型的训练结果，具体来说，将静态修改特征学习阶段的输出（包含代码结构的特征矩阵），替换为补丁应用前后的函数代码与位置序列作为动态信息的输入。

3.4.1.4　数据集划分

Ceres 是一个基于历史数据进行学习的补丁正确性判别技术，训练集和测试集的划分对于学习结果会存在一定影响。已有的基于学习的相关工作[96,100,104] 均是跨补丁划分数据集，即补丁数据集合随机划分为训练集和测试集，同一缺陷的不同补丁可能会同时出现在训练集与测试集中，而在实际场景中，开发者需要利用已经修复缺陷的补丁预测未修复缺陷补丁的正确性，即跨缺陷划分数据集，本文同时考虑这两种数据集划分方式，并探究这两种场景下，不同技术的判别效果。表3.2列出实验中两种划分

表 3.2　两种数据划分方式下的补丁个数

划分方式	**训练集**		**测试集**		**重合补丁**
	正确	错误	正确	错误	
跨补丁划分	458	492	52	67	93
跨缺陷划分	455	505	55	54	0

重合补丁：同一缺陷的不同补丁出现在训练集中的测试补丁数。

方式的统计结果，该结果是在满足以下原则下的随机划分结果：（1）测试集与训练集的比例均约为 9:1；(2) 在训练集与测试集中正例与负例约为 1:1。由表可以看出，在跨补丁划分数据集中，测试集中有 93 个补丁所修复缺陷的其它补丁处于训练集中，这些测试补丁的判别效果，可能会受到影响。

3.4.1.5　衡量指标

为了验证 *Ceres* 的效果，本文首先定义了如下术语：

- 真正类（TP）: 错误补丁被正确预测为错误补丁类别。
- 真负类（TN）：正确补丁被正确预测为正确补丁类别。
- 假正类（FP）：正确补丁被错误预测为错误补丁类型。
- 假负类（FN）：错误补丁被错误预测为正确补丁类别。

进而采用分类模型中被广泛使用的衡量指标：准确率 (公式3.18)，召回率 (公式3.19)，精确率（公式3.20），F1 分数 (公式3.21)。其中准确率衡量识别出的错误补丁中真实错误补丁的比例，补丁正确性判别技术需要具有高的准确率，因为不可以过滤掉正确的补丁。召回率衡量了识别出错误补丁占总体错误补丁的总数，一个高的召回率对于开发者来说也是至关重要的，因为可以减少审阅错误补丁的个数。一般而言，准确率与召回率是互相制衡的，例如将所有补丁标记为错误补丁时，召回率为 100%，而准确率处于较低水平。

为了解决以上问题，参照已有工作[86]，本文在不过滤掉任何正确补丁（准确率为 100%）的情况下，对比判别技术识别错误补丁数量（召回率）。ODS 技术与 *Ceres* 类似都是输出每个补丁为正确和错误的概率，因此可通过设置正确性阈值保障准确率为 100%。特别地，PatchSim 技术的实现中直接输出每个补丁的判别结果，因此在新的数据集上无法保证准确率达到 100%，在此情况下，可通过精确率与 F1 分数的值进行比较。

$$\text{准确率}\ (precision) = \frac{TP}{TP + FP} \tag{3.18}$$

表 3.3 *Ceres* 的效率

划分方式	**修改特征图**			**覆盖特征图**		
	节点数	边数	构建 (ms)	节点数	边数	构建 (ms)
跨补丁划分	315	750	4.08	39	242	0.99
跨缺陷划分	321	765	4.38	40	251	1.24

$$\text{召回率}\ (recall) = \frac{TP}{TP + FN} \tag{3.19}$$

$$\text{精确率}\ (accuracy) = \frac{TP + TN}{TP + FP + FN + TN} \tag{3.20}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall} \tag{3.21}$$

3.4.2 实现细节

数据收集。本文使用 JDT[138] 对源码级别代码进行插桩收集覆盖信息。为了与 ODS 技术进行对比，本文按照其说明使用开源工具 Coming[139] 提取补丁应用前后的静态特征。

时间开销。表3.3展示了在训练集上构建特征图的平均时间开销。由表可见两种划分方式下构建的特征图的边数与节点数没有显著区别，修改特征图要略大于覆盖特征图，这是因为修改特征图中包含了一个函数内的所有代码节点，而覆盖特征图只包含了测试用例节点与少量行级别代码节点，不过两个图的构建时间都是比较短的，平均只需要 0.99*ms* 到 4.38*ms* 之间。

超参数。本实验中所有的模型都采用大小为 256 的词向量维度。为了节省时间开销，所有项目的批量大小均设为 60 来最大化利用 GPU。在覆盖特征学习模型中限制通过测试用例的个数不超过 100 个。遵循已有工作[140]，因为跨缺陷与跨补丁训练模型的稳定速度不同，在跨缺陷训练模型中采用 80 为其迭代次数，跨补丁训练模型中采用 100 为其迭代次数。本实验中数据划分与训练过程中均设置了随机种子，以保证实验可复现。

实验环境。本文中的实验在以下配置的 Dell 工作站完成，包括 256G RAM、Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz、24G GPUs GeForce RTX 3090 和 Ubuntu16.04.12 LTS。实验使用的学习框架是 PyTorch V1.3.0[141] 。

3.4.3 实验结果

3.4.3.1 RQ1: *Ceres* 的效果

表3.4和表3.4分别展示了跨补丁和跨缺陷划分训练集与测试集两种场景下，三种技术对补丁正确性判别的效果。其中第一列为技术名称，第二到五列为补丁判别的具体结果，剩余列分别为准确率、召回率、精确率和 F1 分数指标。特别地，PatchSim 技术的实现不支持 Closure 项目下的补丁正确性判别任务①，在部分补丁上执行超时，PatchSim 行的统计数据是在可执行的补丁上结果。

RQ1a: 跨补丁划分数据集的效果比较。由表3.4可知，*Ceres* 在四项指标上均超过 PatchSim 和 ODS。PatchSim 技术无法保证补丁正确性判别的准确率达到 100%, 而 ODS 和 *Ceres* 可以通过设置补丁正确性的阈值保证准确率达到 100%。相比于 PatchSim 与 ODS，*Ceres* 在保证准确率 100% 的情况下，识别错误补丁的召回率分别提升了 34.6% 和 12.9%，这也说明，相比于单一使用静态信息或者动态信息的技术，*Ceres* 同时使用这两种信息，可以提升识别错误补丁的召回率。

RQ1b: 跨缺陷划分数据集的效果比较。由表3.5可知，*Ceres* 在四项指标上均超过了 ODS，即保证准确率 100% 的情况下，*Ceres* 识别错误补丁的召回率相比于 ODS 提升了 800%。PatchSim 的召回率虽然高于 *Ceres* 的召回率，但是其准确率却远低于 *Ceres* 的准确率。为了进一步公平的与 PatchSim 技术进行比较，本文降低了 *Ceres* 判别错误补丁的准确率，将判别正确性阈值设为识别接近一半正确补丁的值，此时 *Ceres* 达到 62.2% 的准确率（见表3.5的 $PCA_{p=60\%}$ 行），而判别错误补丁的召回率为 85.2%，比 PatchSim 提升了 130%。由此可见，在跨缺陷划分数据集情景下，*Ceres* 的判别错误补丁的效果也优于 PatchSim 和 ODS。

综合表3.4和表3.5来看，基于历史数据学习的补丁判别技术（ODS 和 *Ceres* ）的效果受数据集划分方式影响较大，相比于跨补丁划分方式，在跨缺陷划分场景下，ODS 召回率降低了 96.8%，*Ceres* 召回率降低了 74.3%。出现这种情况的原因是同一缺陷的不同补丁的特征是相似的，因此当同一缺陷的不同补丁分别出现在训练集与测试集中时，可能会出现数据泄露，例如图3.1和图3.2中的两个错误补丁分别出现在测试集与训练集时，可以很容易通过其中一个补丁的特性判别另外一个补丁的正确性。但是在实际调试场景中，只能以历史上其它缺陷的补丁数据作为训练集，不存在同一缺陷的其它补丁，跨补丁划分数据集的场景是不合理的，并且无法体现基于历史数据学习的补丁判别技术的真正效果。该发现可以为基于历史数据学习的补丁判别技术的未来研究提供新的思路。

---

① PatchSim 使用 Randoop 生成额外的测试用例，而 Randoop 不支持为 Closure 项目生成测试用例[86]。

表 3.4　在跨补丁划分场景下，*Ceres* 与其它技术的效果对比

技术	TP	FP	TN	FN	准确率	召回率	精确率	F1
PatchSim	26	7	20	28	78.8%	48.1%	56.8%	59.8%
ODS	31	0	52	36	100.0%	46.3%	69.7%	63.3%
Ceres	35	0	52	32	100.0%	**52.2%**	**73.1%**	**68.6%**

PatchSim 不支持 37 个补丁，在 1 个补丁上执行超时。

表 3.5　在跨缺陷划分场景下，*Ceres* 与其它技术的效果对比

技术	TP	FP	TN	FN	准确率	召回率	精确率	F1
PatchSim	20	22	17	30	47.6%	40.0%	41.6%	43.5%
ODS	1	0	55	53	100.0%	1.9%	51.4%	3.6%
Ceres	9	0	55	45	100%	**16.7%**	**58.7%**	28.6%
$Ceres_{p=60\%}$	46	28	27	8	62.2%	85.2%	67.0%	**71.9%**

PatchSim 不支持 17 个补丁，在 3 个补丁上执行超时。

3.4.3.2　RQ2: *Ceres* 部件的影响

图3.8和图3.9分别展示了两种划分数据集方式下，*Ceres* 只考虑修改特征的 $Ceres_{mod}$ 和只考虑覆盖特征 $Ceres_{cov}$ 判别补丁正确率 100% 的情况下，召回率与 F1 分数的值。实验结果进一步验证了本文的研究动机：两种划分方式下，修改特征与覆盖特征对于补丁正确性的学习都提供了帮助，去除任何一种特征，都会对效果造成影响。此外，*Ceres* 技术判别错误补丁的召回率远高于另外两种单一技术召回率之和，可见修改特征与覆盖特征有效判别的补丁集合并不是独立的，在一个补丁上同时考虑修改特征和覆盖特征才能提升判别召回率。

RQ2a: 修改特征学习对 *Ceres* 的影响。从图3.8和图3.9来看，在跨补丁划分数据集上，只考虑修改特征学习，召回率和 F1 分数分别下降了 82.9%(减少了 29 个) 和 76.1%。在跨缺陷划分数据集上，只考虑修改特征，召回率与 F1 分数均为 0。此结果与 ODS 在跨缺陷划分数据集上只识别了一个错误补丁是一致的，说明对于修改特征，跨补丁划分数据集与跨缺陷划分数据集结果具有较大差异，在跨补丁划分场景下，同一缺陷的不同补丁可能被分别划分在了测试集与训练集中，提升了判别效果。

RQ2b: 覆盖特征学习对 *Ceres* 的影响。从图3.8和图3.9来看，在跨补丁划分数据集上，只考虑覆盖特征学习，召回率和 F1 分数分别下降了 88.6%(减少了 31 个) 和 83.5%。在跨缺陷划分数据集上，只考虑覆盖特征，召回率与 F1 分数分别下降了 55.6%(减少了 5 个) 和 51.7%。两种划分方式下，只考虑覆盖特征都在一定程度上影响了最终的效果，但是两种划分方式下的效果是比较接近的，说明同 PatchSim 一样，数据集的划分方式对于基于动态覆盖信息的技术影响不大。

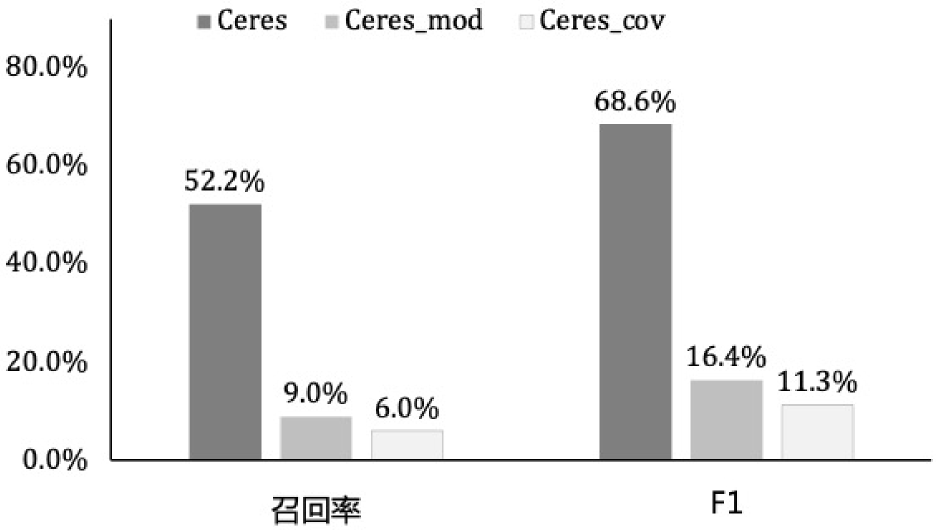


图 3.8　跨补丁划分数据集上各部件的影响

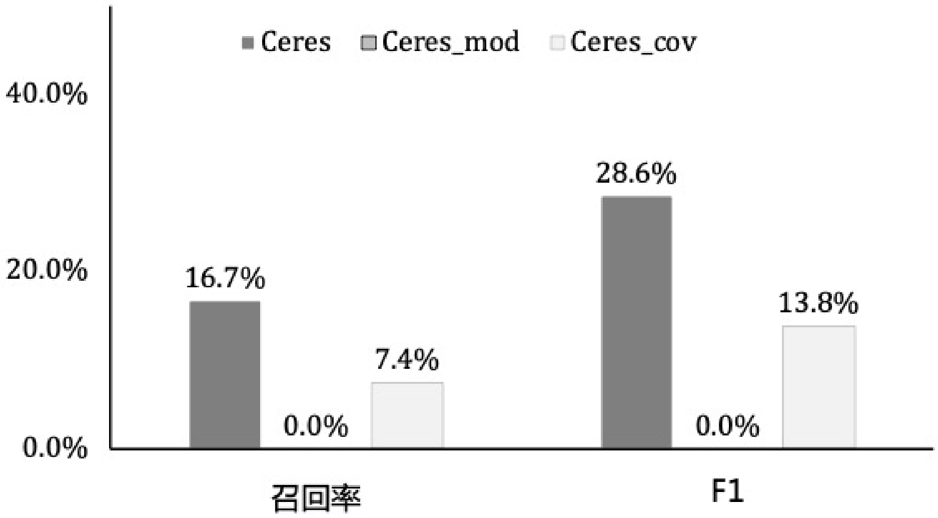


图 3.9　跨缺陷划分数据集上各部件的影响

3.5 局限性讨论

根据第3.4节的实验结果可知，本文涉及到的补丁正确性判别技术在不同的场景下判别效果各有不同。本节依据各项技术采用的判别方式和信息源的不同，对这些技术的局限性和适用场景进行讨论。

- *基于启发式规则依据动态信息的判别方法（PatchSim）*。基于启发式规则的判别技术不需要划分测试集与训练集，因此适用于任意场景下的补丁正确性判别。本文选取的 PatchSim 技术使用 Randoop 生成额外的测试用例，而 Randoop 目前不支持对某些项目生成额外的测试用例。因此，PatchSim 不支持 Closure 项目。
- *基于历史数据的判别方法*。该方法需要划分训练集与测试集，因此在不同划分场景下，各种技术判别效果会发生变化。
    - *只依据静态信息（ODS、$Ceres_{mod}$）*。此类技术依据补丁的静态语法及语义特征相似性对测试集中的补丁正确性进行判别。因此当历史数据中，存在与当前待判别补丁的静态特征相似的补丁时，例如存在同一缺陷的不同补丁，或者是相似缺陷的相似补丁，此类技术才可准确地判别补丁正确性。
    - *只依据动态信息（$Ceres_{cov}$）*。依据补丁的动态执行特征相似性对测试集中的补丁正确性进行判别。该技术依赖于补丁应用前后测试用例在程序上的覆盖变化情况，不依赖于具体的测试用例代码或程序补丁代码。因此，同 PatchSim 技术一样，对于两种数据集划分场景，该技术的判别效果没有明显区别。特别地，对于在应用前后，通过测试和失败测试用例覆盖变化情况不同的这类补丁，该判别技术的准确率更高。
    - *同时依据静态信息和动态信息（Ceres）*。该技术同时依赖于补丁的静态信息与动态信息。其中。利用静态信息部分，同 $Ceres_{mod}$ 一样，在测试集中，存在与当前待判别补丁的静态特征相似的补丁时，即跨补丁划分数据集场景下，该技术可以较准确的判别补丁正确性。而利用动态信息部分，同 $Ceres_{cov}$ 一样，在跨缺陷划分数据集场景下，也可以准确判别补丁正确性。因此，在两种数据集划分场景下，该技术均可以取得较好的效果。

3.6 讨论与小结

为提升开发者审阅效率和正确率，可在人工审阅之前自动过滤掉部分错误补丁。已有的补丁过滤技术只采用了补丁的静态修改信息或者动态覆盖信息等单一信息源，其判定错误补丁的召回率有待提升。本章提出同时基于静态修改特征与动态覆盖特征

的补丁过滤技术，将补丁应用前后的函数代码表示为一个基于抽象语法树的修改表征图，通过 S-Transformer 模型对该表征图编码与分析，学习对判别补丁正确性有效的关键修改特征。同时，构建覆盖表征图，将行级别的代码与测试用例抽象为图中的节点，根据覆盖关系在两个节点之间建立不同权重的边，通过 RAT 模型对该图编码与分析，学习对判别补丁正确性有效的关键覆盖特征，同时结合基于修改表征图的学习结果对补丁正确性进行判别。

本章在跨补丁划分数据集与跨缺陷划分数据集两种场景下对提出方法进行了验证。实验表明，在跨补丁划分数据集场景下，*Ceres* 相比已有方法分别多过滤了 12.9% 和 34.6% 的错误补丁，在跨缺陷划分数据集场景下，*Ceres* 的判别效果虽然有所下降，但是仍然优于已有方法。本文提出基于静动态信息结合的补丁过滤技术能有效过滤掉大部分错误补丁，减轻了人工审阅补丁的负担。

第四章　交互式补丁过滤技术

4.1　引言

第三章介绍了基于动静态信息结合的补丁过滤技术，该方法可以在人工审阅补丁之前，自动过滤掉部分错误补丁，减轻开发者的审阅负担，提升开发者的修复效率。不过，目前自动过滤技术过滤错误补丁的数量有限，即无法过滤掉全部错误补丁，无法被过滤的补丁仍然需要开发者人工审阅，判断其正确性。

自动修复技术生成的补丁最终不可避免的需要开发者审阅。当给定一组正确性未知的候选补丁时，为进一步帮助开发者判断补丁正确性，本章提出的技术尝试引导开发者关注关键信息，帮助开发者理解补丁及缺陷，判别补丁正确性，进而提升修复缺陷的效率和正确率。具体地，本章提出交互式补丁过滤技术，通过与用户交互的方式辅助开发者理解补丁正确性从而过滤错误补丁，完成修复。交互式补丁过滤技术需要解决以下问题：（1）选择何种类型的问题既可以区分正确补丁和错误补丁，又易于用户回答，并且回答该问题有助于用户理解缺陷及补丁；（2）如何设计询问策略，减少与用户的交互次数，提升交互效率；（3）如何设计交互界面可以方便用户在交互过程中调试缺陷，提升用户的修复效率。

本章提出 *InPaFer* 依次解决以上问题。具体而言，本章基于三种程序属性，即修改方法（静态代码属性）、执行路径和变量值（动态运行时特性），设计了三种类型的补丁过滤准则（即补丁特性相关的询问），例如，“类 `c` 中的方法 `m` 应该被修改”，“应该执行类 `c` 中第 n 行的语句” 和 “值 `val` 在程序执行期间应该分配给变量 `var`”，用于与用户交互过滤错误补丁。特别地，这些过滤准则在过滤掉错误补丁的同时也引导了用户关注这些关键补丁特性，从而有助于用户理解缺陷原因，最终完成修复。例如，在回答修改方法相关的问题时，会引导用户关注到可能出错的位置信息。由于收集程序属性需要很长时间，及时响应用户的反馈，实现与用户实时交互是不切实际的。因此本章将 *InPaFer* 分为两个阶段：准备阶段（收集程序属性和构建过滤准则，即关于补丁特性的问题）和交互阶段（与开发者进行在线交互，辅助审阅补丁并过滤错误补丁）。为提升与用户的交互效率，本章定义了询问选择问题，并证明询问选择问题与最优决策树构建问题在多项式时间内可规约。然后引入决策树中的最小化最大分支算法减少交互次数，即每次选择尽可能将当前候选补丁均分的过滤准则。理论证明，最小化最大分支算法是解决最优决策树构建问题的多项式时间内的最优近似算法。最后，本章将 *InPaFer* 实现为一个 Eclipse 插件，并设计了过滤视图与差异视图方便用户审阅补丁，调试缺陷。

为验证 *InPaFer* 过滤错误补丁的效果与辅助开发者提升修复效率和正确性的效果，本章分别进行了定量实验与用户实验。定量实验结果表明，对于 47.7% 的缺陷，*InPaFer* 可以过滤掉全部错误补丁。用户实验结果表明，使用 *InPaFer* 修复缺陷的开发者的修复效果（修复准确率和效率）显著优于另外三组基准调试场景下开发者的修复效果。其中，相比于没有任何辅助下自己完成修复的开发者准确率提升了 62.5%，修复时间减少了 25.3%。即使所有候选补丁均是错误补丁，开发者的修复结果也没有显著降低，仍然可以提升。

本章的组织结构如下：第4.2节介绍交互式过滤补丁的方法；第4.3节定义了询问选择问题并进行交互次数理论分析；第4.4节介绍了工具实现；第4.5节通过定量实验与用户实验对本章提出的方法进行验证；第4.6节对本章内容进行讨论和总结。

4.2 方法概述

本节首先用一个缺陷示例引入交互式补丁过滤的工作流程（第4.2.1节），然后再详细介绍交互式补丁过滤的两个阶段：*准备阶段*（第4.2.2节）和*交互阶段*（第4.2.3节）。

4.2.1 说明示例

图4.1展示了 Defects4J[52] 基准数据集中缺陷 Math41 的代码片段，当条件 `length>1` 时，它调用缺陷方法 `evaluate ()`（第 322 行）。为了修复这个缺陷，现有的 APR 工具生成了一组候选补丁。表4.1列出了三个候选补丁，它们可以通过全部测试用例。其中，第二列表示图4.1的代码中需要被修改的代码行号，最后两列显示补丁的详细信息及其正确性。

从表4.1可以看出，补丁和打补丁之后的程序显示出不同的动态和/或静态属性。例如，补丁 p_2 和 p_3 修改了程序中的 `eval ()` 函数，p_1 修改了函数 `evaluate ()`。此外，在应用候选补丁并执行测试之后，可以发现补丁 p_2 和 p_3 使失败测试用例进入第 321 行的 `then` 分支，相反，在应用补丁 p_1 时，程序执行将覆盖第 323 行和第 325 行之间的 `else` 分支。因此，开发人员可以通过检查这些应用补丁之后程序的一些属性是否正确来过滤出不正确补丁。例如，如果开发者认为缺陷存在于 `evaluate ()` 函数中，那么可以直接过滤掉修改其他函数的补丁（即 p_2 和 p_3）。

基于这一观察结果，本章设计了一个交互式补丁过滤技术 *InPaFer*。给定一组针对同一缺陷的候选补丁，*InPaFer* 分析哪些程序属性可以区分候选补丁，并向开发者提供关于程序属性是否正确的是/否选项，称为过滤准则（filtering criteria）。开发者选择其中一个过滤准则并验证其正确性，称为一次过滤（filtering step）。每次过滤，*InPaFer* 会自动过滤出与开发者提供的答案相矛盾的错误补丁（是代表认为属性正确，确认对

```
public double eval(final double[] val, ...){
     ...
  if(length == 1){//错误补丁修改位置
     var = 0.0;
  }else if(length > 1){
     Mean mean = new Mean();
     double m = mean.evaluate(val,weights,...);
     var = evaluate(val,weights,m,begin,length);
  }
     ...
}

public double evaluate(double[] values, ...) {
     ...
   for(int i=0;i < weights.length;i++){ //真正缺陷位置
      sumWts += weights[i];
   }
     ...
}
```

图 4.1　Math41 中的一个代码片段

应的补丁，否代表属性不正确，拒绝对应的补丁），并为剩余补丁更新过滤准则。

表4.2给出了表4.1中所列补丁的两个代表性过滤准则。假设开发者首先选择 f_2，然后拒绝它（即回答“否”）。f_2 对应的补丁将被过滤掉（即 p_2 和 p_3），而其他不满足这个过滤准则的补丁（p_1）会被保留。

为开发者提供过滤准则也可以帮助开发者理解缺陷，例如，当考虑过滤准则 f_1 是否正确时，开发人员会重点关注函数 `evaluate()`，而缺陷就发生在这个函数内。当考虑过滤准则 f_2 是否正确时，开发人员会理解变量 `var` 的值与缺陷相关（当把它设置为 0 时，所有测试用例都会通过）。因此，即使候选补丁集合中不包含任何正确补丁，回答这些过滤准则是否正确也有利于理解该缺陷，最终有助于开发者修复这个缺陷，

实现 *InPaFer* 有一个挑战：由于收集程序属性可能需要很长时间，例如，程序执行信息的获取，及时响应实时调试过程是不切实际的。为了克服这一挑战，本章利用了这样一个事实，即自动修复技术被假定为在日常工作之后，在第二天的工作时间之前。事实上，当前的自动修复技术通常需要几个小时才能修复一个缺陷，并且不能在线使用[57,72,75,142]。因此，本文计划将 *InPaFer* 设计为两阶段。如图4.2所示，准备阶段是一个收集程序属性和构建过滤准则的过程，而交互阶段是一个在线过程，通过与开发者的交互来执行补丁过滤，这可以在很短的响应时间内实现。接下来两节将详细介绍这两个阶段。

表 4.1　Math41 的候选补丁

补丁 ID	代码行号	代码修改	是否正确
p_1	520	`- for (i = 0; i < weights.length; i++){` `+ for (i = begin; i < begin + length; i++){`	是
p_2	320	`- if (length == 1){` `+ if (length == 5){`	否
p_3	320	`- if (length == 1){` `+ if ((length & 1) == 1){`	否

表 4.2　Math41 的前两个过滤准则

过滤准则 ID	程序属性	补丁	程序属性是否正确
f_1	修改类 *Variance* 中的函数 *evaluate()*	p_1	是/否
f_2	测试用例覆盖类 *Variance* 中的第 *321* 行代码	p_2,p_3	是/否

补丁是指满足对应程序属性的补丁。

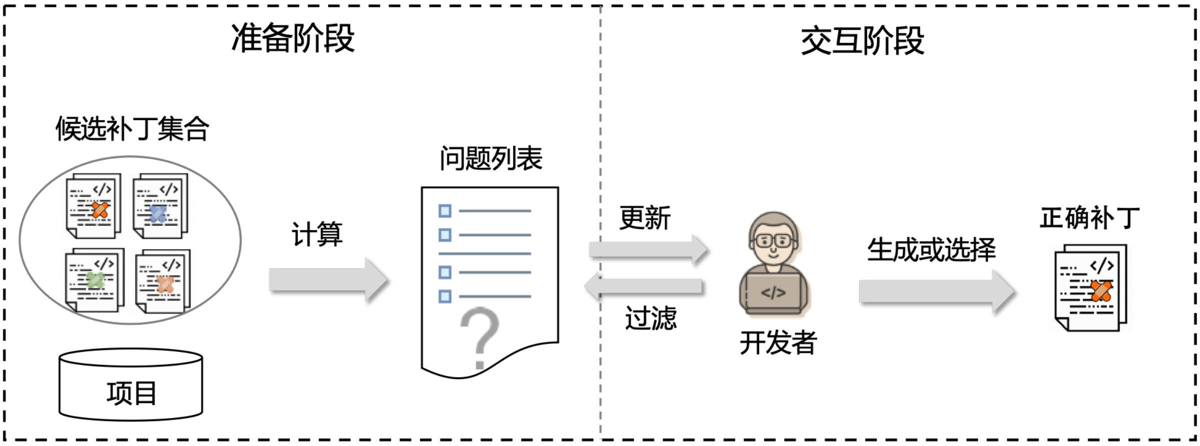


图 4.2　InPaFer 的工作流程

4.2.2　准备阶段

准备阶段是为交互阶段进行数据准备的离线过程。给定一组与缺陷相关的应用补丁之后的程序，*InPaFer* 会自动收集不同应用补丁之后的程序属性，最终利用这些属性构建一套交互过滤准则。

一般来说，可以采用多种属性来区分候选补丁。但是，应该优先选择那些能够区分更多补丁并且易于开发者理解的属性，因为它们可能会减少交互的数量并减轻开发者回答问题的负担。因此，*InPaFer* 计划实现三种程序属性，包括静态代码属性（修改方法）和动态运行时特性（执行路径和变量值）。根据先前的研究[40,143,144]，这些属性对于开发者在调试过程中理解程序并区分候选补丁非常有用。每个属性的详细信息如下所述。

- 修改方法可以解释为“类 c 中的方法 m 应该被修改”，其中 m 和 c 分别代表一个方法和一个类的名称。*InPaFer* 在准备阶段分析补丁并收集其修改方法。当一个补丁修改多个方法时，*InPaFer* 会收集所有这些方法。
- 执行路径可以解释为“应该执行类 c 中第 n 行的语句”，其中 n 和 c 分别表示行号和类。但是，完整的执行路径对于人工检查来说可能太长。*InPaFer* 目前只考虑由候选补丁修改的那些方法中的执行路径，因为它们与修改的代码很接近，因此更容易进行人工检查。
- 变量值可以解释为“值 `val` 在程序执行期间应该分配给 `var`”，其中 `val` 和 `var` 分别代表一个值和一个变量。尤其是，*InPaFer* 考虑了修改后方法的入口和出口位置具有原始类型的所有局部变量和类字段。之所以只考虑这些变量赋值，是为了不让开发者检查太多的过滤条件。

对于每种程序属性，*InPaFer* 都会构造一个用于交互阶段的过滤准则列表。每个缺陷都有多个通过测试用例，对于执行路径和变量值，考虑通过测试用例执行的结果可能会加重开发者的负担，因此 *InPaFer* 只使用失败测试用例的执行信息。如果一个缺陷有多个失败测试用例，那么 *InPaFer* 将会收集每个失败测试用例的执行信息，用于构造过滤准则。另外，*InPaFer* 会移除无法区分任何补丁的过滤准则。

4.2.3　交互阶段

交互阶段是一个使用准备阶段构建的过滤准则与开发者在线进行交互的过程（如图4.2所示）。这个阶段的输入是一个过滤准则列表和正在调试的完整项目。每次，InPaFer 都会收集开发者对某个过滤准则的反馈，并根据需要更新候选过滤准则和补丁。更具体地说，在交互式调试过程中，开发人员总共可以执行三种操作。

- 过滤 确认或拒绝过滤准则以筛选候选补丁。
- 选择 从候选补丁中选择一个补丁作为正确的补丁。
- 生成 手动生成一个正确的补丁来修复缺陷。

对于每次过滤，*InPaFer* 按照算法2更新过滤准则列表和候选补丁列表。(1）通过只保留与开发者答案一致的补丁来更新候选补丁（第 3-7 行），(2）通过只保留与剩余补丁相对应的过滤准则来更新过滤标准（第 8-13 行），最后，向开发者展示更新后的候选补丁和过滤准则。当开发者选择或提供一个正确的补丁，或者没有过滤准则需要被回答时，该过程终止。

4.3　交互次数理论分析

本节首先使用一个示例说明询问选择策略的重要性 (第4.3.1节)，然后定义询问选择问题（第4.3.2节），对决策树进行介绍（第4.3.3节），并且引入最小化最大分支算法

Algorithm 2 更新算法

```
Require: $\mathbb{F}$: 过滤准则列表, $\mathbb{P}$: 候选补丁列表
         $f$: 被回答的过滤准则, $a$: $f$ 的答案
Ensure: $\mathbb{F}'$: 更新之后的过滤准则列表, $\mathbb{P}'$: 更新之后的补丁列表
$\mathbb{F}' \leftarrow \emptyset, \mathbb{P}' \leftarrow \emptyset$
if $\mathbb{F}! = \emptyset$ && $\mathbb{P}! = \emptyset$ then
    if $a ==$ yes then $f.attr$ 是正确的
        $\mathbb{P}' \leftarrow f.patches$ 中满足 $f.attr$ 的补丁
    else
        $\mathbb{P}' \leftarrow \mathbb{P} \setminus f.patches$ 中不满足 $f.attr$ 的补丁
    end if
    for 每个 $f' \in \mathbb{F}$ do 更新过滤准则列表
        $f'.patches \leftarrow f'.patches \cap \mathbb{P}'$
        if $f'.patches \neq \emptyset$ then
            $\mathbb{F}' \leftarrow \mathbb{F}' \cup \{f'\}$
        end if
    end for
end if
```

（*minimax branch strategy*）(第4.3.4节)，最后介绍询问选择问题与决策树之间的关系以及最小化最大分支算法在询问选择问题上的应用（第4.3.5节）。

4.3.1 说明示例

本节使用一个例子说明，交互式补丁过滤中，选择询问的方式对询问的数量的影响。考虑存在表4.3中的候选补丁集合，该补丁集合中包含 8 个语义等价类中 14 个候选补丁，其中 S_4 中的补丁是用户期望的补丁。

为了从这 14 个补丁中找到用户需要的补丁，一个交互式补丁过滤可能会向用户提出若干关于输入输出的询问。具体来说，在每一轮交互中，交互工具会选择一个输入，然后向用户询问该输入的输出。在这个过程中，不同输入的选择方式会显著影响询问的数量。例如，采用随机策略，按照 $x = 1, x = 3$ 和 $x = 2$ 的顺序对用户进行提问，最终获取正确的补丁集合 S_4，其具体过程如下所示：

询问次数	询问输入	用户提供的输出答案	剩余合法补丁
1	$x = 1$	$output = 0$	$\{S_3, S_4, S_5, S_6\}$
2	$x = 3$	$output = 2$	$\{S_3, S_4\}$
3	$x = 2$	$output = 0$	$\{S_4\}$

可以看出以上过程总共包含了三次询问。然而，如果首次询问 $x = 2$，那么可以只用一次询问排除其它非法补丁，补丁空间中只剩下目标补丁集合 S_4。

这个例子说明了询问输入的选择对于交互次数具有显著的影响，随机选择输入可

表 4.3　候选补丁集合

Id	补丁 1	补丁 2	(询问, 答案)	Id	补丁 1	补丁 2	(询问, 答案)
S_1	ret x	$o_1 = x - 1$ ret $o_1 + 1$	$(1,1)$ $(2,2)$ $(3,3)$ $(4,4)$	S_5	$o_1 = x + 1$ ret $x \ \& \ o_1$	$o_1 = x + 1$ $o_2 = o_1 - 1$ ret $o_1 \ \& \ o_2$	$(1,0)$ $(2,2)$ $(3,0)$ $(4,4)$
S_2	ret $x + 1$	$o_1 = x + 1$ ret $o_1 \ \& \ o_1$	$(1,2)$ $(2,3)$ $(3,4)$ $(4,5)$	S_6	$o_1 = x - 1$ $o_2 = x + 1$ ret $o_1 \ \& \ o_2$	$o_1 = x + 1$ $o_2 = x - 1$ ret $o_1 \ \& \ o_2$	$(1,0)$ $(2,1)$ $(3,0)$ $(4,1)$
S_3	ret $x - 1$	$o_1 = x - 1$ ret $o_1 \ \& \ o_1$	$(1,0)$ $(2,1)$ $(3,2)$ $(4,3)$	S_7	$o_1 = x - 1$ $o_2 = o_1 \ \& \ x$ ret $o_2 + 1$		$(1,1)$ $(2,1)$ $(3,3)$ $(4,1)$
S_4	$o_1 = x - 1$ ret $x \ \& \ o_1$	$o_1 = x - 1$ $o_2 = o_1 + 1$ ret $o_1 \ \& \ o_2$	$(1,0)$ $(2,0)$ $(3,2)$ $(4,0)$	S_8	$o_1 = x + 1$ $o_2 = o_1 \ \& \ x$ ret $o_2 - 1$		$(1,-1)$ $(2,1)$ $(3,-1)$ $(4,3)$

S_i 表示一组语义等价的补丁，(询问，答案) 表示给定输入 x 为 ‘询问’ 值时，该补丁的输出返回值为 ‘答案’。

能会需要用户回答更多的问题，给用户带来额外的负担。因此，采用一个最优询问策略对于交互式补丁过滤来说十分重要。

本节将会介绍最小化最大分支策略（*min-max branch strategy*），每次选择一个询问使得用户给出最坏答案时，能最好的削减搜索的补丁空间，其中最坏答案即为剩余合法补丁个数最多的答案。在这个例子中，该策略首先会询问 $x = 3$，最坏的答案 2 将补丁空间削减到两个等价类集合 $\{S_3, S_4\}$ ，相反，如果询问 $x = 1$，最坏答案为 0 时，补丁空间会剩下 4 个等价类 $\{S_3, S_4, S_5, S_6\}$，在这个例子中，对于 $x = 3$，用户恰好回答 2，然后该策略会继续选择 $x = 2$ 来区分 S_3 和 S_4。在这种情况下，只需要两次询问就可以完成交互过程。该算法的详细内容及证明将在接下来的章节中介绍。

4.3.2　询问选择问题

定义 6 语义函数 给定一个补丁空间 $\mathbb{P}$, 询问空间 $\mathbb{Q}$ 和答案空间 $\mathbb{A}$，一个函数 $\mathbb{D} : \mathbb{P} \to \mathbb{Q} \to \mathbb{A}$ 为每个补丁 p 和问题 q 关联一个答案 a, 记作 $\mathbb{D}[p](q)$，表示在 p 为用户目标补丁情况下，询问 q 对应的答案。

显然，交互式补丁过滤只能区分那些在语义函数上表现不同的补丁。

定义 7 可分区的补丁 两个补丁 p_1, p_2 是关于语义函数 $\mathbb{D}$ 可区分的当且仅当存在 $q \in \mathbb{Q}$，使得 $\mathbb{D}[p_1](q) \neq \mathbb{D}[p_2](q)$。否则这两个补丁是不可区分的。

交互式补丁过滤的目标是从补丁空间中找到一个和目标补丁 r 不可区分的补丁 p。为了方便接下来的讨论，本节进一步进入合法补丁的概念，用来表示和给定的询问-答案对一致的补丁形成的集合。同时，在接下来的讨论中，本节假设存在一个对于所有补丁，询问，答案空间都有效的通用语义函数 $\mathbb{D}$。

定义 8 合法补丁 假设 $C \in (\mathbb{Q} \times \mathbb{A})^*$ 是一个包含若干询问-答案对的序列，$\mathbb{P}$ 是一个补丁空间，$\mathbb{P}$ 上关于 C 的合法补丁，记作 $\mathbb{P}|_C$，是 $\mathbb{P}$ 中所有与 C 一致的补丁形成的集合，即 $\{p|p \in \mathbb{P}, \forall(q,a) \in C, \mathbb{D}[p](q) = a\}$。

接下来讨论交互部分，首先，本节把每一轮中的交互式补丁过滤的询问定义为一个关于交互历史的函数。

定义 9 询问选择函数 给定一个补丁空间 $\mathbb{P}$，询问空间 $\mathbb{Q}$ 和答案空间 $\mathbb{A}$，询问选择函数 $QS : (\mathbb{Q} \times \mathbb{A})^* \mapsto \{\top\} \cup \mathbb{Q}$ 是一个对于任意的交互历史 $C \in (\mathbb{Q} \times \mathbb{A})^*$ 和 $q^* = QS(C)$（即当交互历史为 C 时 QS 选择的询问），都满足如下两个条件的函数：

$$q^* = \top \iff \forall p_1, p_2 \in \mathbb{P}|_C, q \in \mathbb{Q}, \mathbb{D}[p_1](q) = \mathbb{D}[p_2](q) \tag{4.1}$$

$$q^* \neq \top \implies \exists p_1, p_2 \in \mathbb{P}|_C, \mathbb{D}[p_1](q^*) \neq \mathbb{D}[p_2](q^*) \tag{4.2}$$

其中，$\top$ 代表交互结束。对于给定的询问选择函数 QS，交互式补丁过滤过程如下所示（起始时，交互历史 $C = \emptyset$）：

1. 如果 $QS(C) = \top$，从合法补丁集 $\mathbb{P}|_C$ 中返回任意一个补丁。
2. 向用户提出询问 $QS(C)$，得到回答 a。
3. 把询问-答案对 $(QS(C), a)$ 加入交互历史 C 中，并返回到步骤 1。

$len(QS, r)$ 代表询问选择函数为 QS, 用户的目标补丁是 r 的交互过程中，询问的个数，φ 为补丁空间 $\mathbb{P}$ 上的一个先验概率分布，表示每个补丁是目标补丁的可能性。询问选择问题的目标是找到一个最优的询问选择函数使得期望询问次数尽可能小。

定义 10 最优询问选择函数 令 $\mathbb{P}$ 为补丁空间，φ 为该补丁空间上的概率分布。最优询问选择函数 OQS 是一个最小化 $\sum_{r \in \mathbb{P}} \varphi(r) len(OQS, r)$ 的函数，即最小化期望询问次数的询问选择函数。本节将这一期望值称为一个询问选择函数的代价。

定理 1 OQS 是一个关于补丁空间大小 $|\mathbb{P}|$，询问空间大小 $|\mathbb{Q}|$ 和答案空间大小 $|\mathbb{A}|$ 的 NP 难度问题。

本节将在第4.3.3节中给出详细证明。

4.3.3　背景介绍：决策树

决策树是离散函数中的基本模型之一，被广泛应用于很多领域。

定义 11 决策树 给定一个包含 m 个测试的集合 T，一个包含 n 个元素的结合 X 和一个结果集合 R，令元素 $x \in X$ 在测试 $t \in T$ 上的结果是 $t(x) \in R$。一个 T，X 上的决策树 DT 是一个满足如下条件的有根树：

- 每一个非叶子节点都被标记上了一个 T 中的元素。
- 每一个非叶子节点都至少有两个子节点。
- 每一条边都被标记上了一个 R 中的元素。
- 对于每一个非叶子节点，它连出的边（连向子节点的边）上的标记两两不同。
- 每一个叶子节点都被标记上了一个 X 的非空子集。
- 对于任意一条从根节点到叶子节点的路径，设路径上经过的标号依次为 $(t_1, r_1, ..., t_n, r_n, X_0)$，那么对于 X_0 中的任意一元素和任意一下标 $i \in [1, n]$，都有 $t_i(x) = r_i$。
- 对于任意两个元素 $x_1, x_2 \in X$，x_1, x_2 被标记在同一个叶子节点上当且仅当对于任意的 $t \in T$，都有 $t(x_1) = t(x_2)$。

例 4.1 表4.4展示了一个决策树的例子。在该例子中，元素集合包含 8 个整数 $\{0, 1, 6, 11, 12, 15, 16, 21\}$，可以使用的测试为分别计算一个数在除以 2，3，5 后的余数。图4.3 展示了使用这些集合构建的一颗决策树，决策树上的每一个非叶子节点都被标记了对应的测试，每一个叶子节点都被标记了对应的元素以及每一条边都被标记了对应的答案。

表 4.4　一个决策树的元素集合，测试集合及其对应的答案

		T_1	T_2	T_3
		mod 2	mod 3	mod 5
X_1	0	0	0	0
X_2	1	1	1	1
X_3	6	0	0	1
X_4	11	1	2	1
X_5	12	0	0	2
X_6	15	1	0	0
X_7	16	0	1	1
X_8	21	1	0	1

给出一棵决策树 $\mathcal{DT}$，定义元素 X_i 的代价 $c(X_i)$ 为它所在的叶子节点的深度（根节点的深度被定义为 0）。这个定义有着鲜明的现实意义：它等于这棵决策树在区分 X_i 时所需要使用的测试数量。例如，在图4.3中的决策树上，$c(X_1) = c(X_2) = c(X_4) = c(X_5) = 2, c(X_3) = c(X_6) = c(X_7) = c(X_8) = 3$。

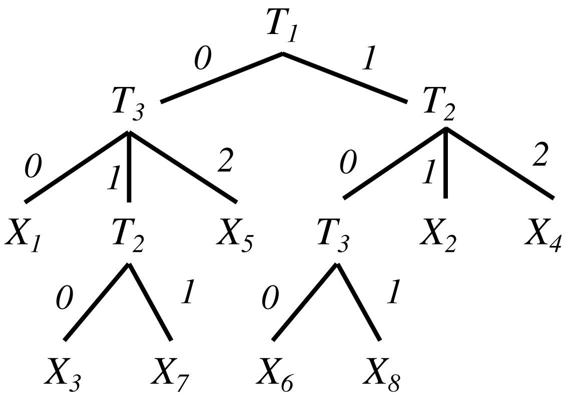


图 4.3　一棵建立在表4.4中集合上的决策树示例

基于决策树的定义和元素代价，可以形式化描述构造最优决策树问题。

定义 12 最优决策树问题 给定一个测试集合 T，一个元素集合 X，一个测试结果集合 R 和一个在 X 上的概率分布 φ，该概率分布赋予 X 中每个元素一个概率值且所有概率值相加和为 1，最优决策树问题 $\mathcal{DT}$ 的目标是构造一个决策树使得一个服从分布 φ 的随机元素的期望代价尽可能小，即找到一颗决策树来最小化 $\sum_{x\in X}\varphi(x)c(x)$。这个期望代价被定义为决策树的代价。

特别地，当 φ 被定义为元素集合 X 上的均匀分布时，该最优决策树问题被称为 $\mathcal{UDT}$。当结果集合大小等于 2 时（测试结果类型为波尔类型），最优决策树问题被称为 2-$\mathcal{DT}$。除此之外，当以上两种特殊情况同时发生时，最优决策树问题被称为 2-$\mathcal{UDT}$。

例 4.2 在例4.1中，当 φ 被定义为元素集合 X 上的均匀分布时，图4.3中决策树的代价是 $\frac{5}{2}$。当 φ 被定义为元素集合 $\{X_1, X_5, X_6\}$ 上的均匀分布，其它元素出现的概率为 0 时，图4.3中决策树的代价是 $\frac{7}{3}$。图4.4分别展示了对应的最优决策树。左侧树展示了 $\mathcal{UDT}$，它的最小代价是 $\frac{19}{8}$，右侧树展示了 φ 在元素集合 $\{X_1, X_5, X_6\}$ 上均匀分布时 $\mathcal{DT}$ 的最小代价 $\frac{5}{3}$。

4.3.4　最优决策树的理论结果

最优决策树的计算复杂度。随着决策树被广泛应用于各个领域，构造一棵最优决策树逐渐成为了一个重要的目标。在过去几十年里，众多理论结果和算法被提出。跟其它优化问题一样，2-$\mathcal{UDT}$ 已经被证明是一个 NP 难度问题[145]，这一结果也说明了 $\mathcal{UDT}$,2-$\mathcal{DT}$,$\mathcal{DT}$ 全部是 NP 难度问题。在意识到在多项式时间内构造一棵决策树的

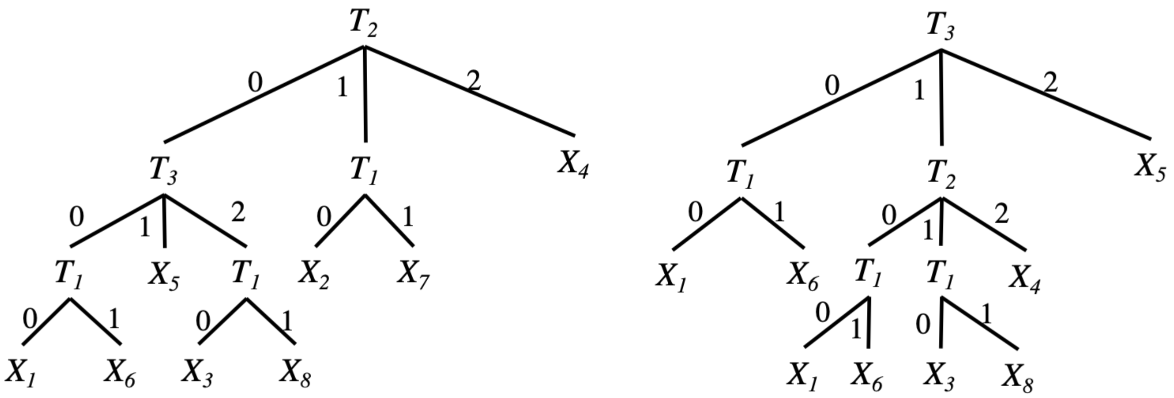


图 4.4　最优决策树的示例，左侧树展示了当 φ 在元素集合上均匀分布时的决策树，右侧树展示了当 φ 在元素集合 $\{X_1, X_5, X_6\}$ 上的均匀分布时的决策树。

难度之后，大多数后续工作都都致力于找到高效的近似算法以及证明最优决策树问题的难近似性。

一个近似算法。作为背景知识，本节首先给出近似算法，近似比和难近似性的定义。直观上来说，近似比衡量了近似算法给出的近似解与最优解之间的距离，而难近似性表现了为一个问题找到具有某个近似比的算法的困难程度。

定义 13 近似算法 给定一个优化问题 $\mathcal{P}$，它的目标是最小化一个目标函数 c, 算法 A 是问题 $\mathcal{P}$ 的近似比为 k 的近似算法当且仅当它满足如下两个条件：

1. 对于问题 $\mathcal{P}$ 的任何一个实例，A 给出的解总是合法的。
2. 对于问题 $\mathcal{P}$ 的任何一个实例，等式 $c(S) \leq kc(S')$ 对于任意的合法解 S' 都成立，其中 S 是算法 A 给出的解。

定义 14 难近似性 最优化问题 $\mathcal{P}$ 是关于近似比 k 难近似的当且仅当不存在任何在多项式时间内以近似比 k 近似 $\mathcal{P}$ 的算法，除非 $P = NP$。

最小化最大分支。本节接下来介绍 $\mathcal{DT}$ 上目前效果最好的算法：最小化最大分支算法[146]。构造决策树的核心步骤是构造每个内部节点时如何选择测试。该算法采用一个贪心策略：测试应该尽可能均等划分元素集合，但是均分是很难评估的，所以本文尝试使用最小化最大划分。

算法3展示了算法伪码。具体来说，给定函数 par 可以基于测试 $t_i \in T$, 将给定的元素集合 S 划分，即，$par(S, t_i) = \{S'_1, \ldots, S'_k\}$，其中，$S_i$ 对应了测试 t_i 在元素集合 S 上的一个可能结果。此时，测试 t_i 在元素集合 S 上的代价为 $cost(t_i, S) = \max_{S'_j \in par(S,t_i)} \sum_{x \in S'_j} \varphi(x)$，即，测试 t_i 在集合 S 上的代价为所有划分得到的子集中，概率和最大的那个子集的概率。算法3首先检查当前节点代表的元素集合是否还可以被进一步划分（第

2-6 行），然后找到一个具有最小代价的测试（第 7 行），并使用这个测试来构造当前节点的标记和子节点（第 8-9 行），其中，第 9 行，NEWCHILD 的第一个参数是子节点边的标记，第二个参数是子树。$t^*(S_i)$ 表示 t^* 在 S_i 中的测试结果。

例 4.3 *图4.3右侧的决策树是使用算法3在例4.1中元素集合* X *为均匀分布时，构造出来的结果。首先，在划分* X *时，测试* T_1, T_2, T_3 *的代价分别为 4，5，5，所以首先会选择测试* T_1 *划分* X *集合，*X *集合被分为两个集合* $\{X_1, X_3, X_5, X_7\}$ *和* $\{X_2, X_4, X_6, X_8\}$*。对于这两个集合，测试* T_2 *和测试* T_3 *分别是最优测试，经过划分之后，只剩下两个集合* $\{X_3, X_7\}$ *和* $\{X_6, X_8\}$ *中的元素无法被区分，然后使用测试* T_3, T_2 *分别划分这两个集合，最后，会得到图4.3中的决策树。*

Algorithm 3 使用最小化最大分支算法构造决策树

```
Require: 测试集合 T，元素集合 X，测试结果集合 R 和在元素集合上的概率分布 $\varphi$。
Ensure: 决策树的根节点
 1: function CONSTRUCT($T, X, R, \varphi$)
 2:     $node \leftarrow$ NEWNODE
 3:     if X 中所有的元素是相同的 then
 4:         $node$ 的标记 $\leftarrow X$
 5:         return $node$
 6:     end if
 7:     $t^* \leftarrow \arg\min_{t \in \mathbb{T}} cost(t, X)$
 8:     $node$ 的标记 $\leftarrow t^*$
 9:     $node$ 的子节点 $\leftarrow \{$NEWCHILD$(t^*(X_i))$, CONSTRUCT$(T - \{t^*\}, X_i, R, \varphi)) \mid X_i \in par(X, t^*)\}$
10:     return node
11: end function
12: return CONSTRUCT($T, X, R, \varphi$)
```

本节将这个贪心算法称之为最小化最大分支算法（*min-max branch*），代表每次选择的测试可以最小化最大分支的数量。目前已经有大量的研究成果证明最小化最大分支是一个高效地近似算法。本节接下来将会简要总结这些研究成果。

最小化最大分支的近似比。表4.5中总结了在 $\mathcal{DT}$ 各个分支问题上目前最好的结果。从这个表中可以发现，对前三个问题，最小化最大分支算法都具有已知最好的近似比。除了 $\mathcal{DT}$ 问题，在这个问题上最小化最大分支算法被证明为是一个近似比为 $O(\log^2 m)$ 的近似算法，而目前最优算法[147] 的近似比为 $O(\log m)$，然而相比于最小化最大分支算法，这个算法要更复杂。

基于假设 $P \neq NP$，对于任意的 $\epsilon > 0$，$2-\mathcal{UDT}$ 和 $\mathcal{UDT}$ 分别为 $2-\epsilon$ 和 $4-\epsilon$ 难近似，同时存在近似比 $r(m) = \Omega(\log m)$ 使得 $2-\mathcal{DT}$ 和 $\mathcal{DT}$ 都是 $r(m)$ 南近似的。特

别地，对于 $2-\mathcal{DT}$ 来说，最小化最大分支的近似比 $O(\log m)$ 已经达到了难近似性给出的理论下界 $\Omega(\log m)$。因此在 $P \neq NP$ 时，最小化最大分支算法是解决 $2-\mathcal{DT}$ 的一个最优的多项式时间近似算法。

表 4.5　相关研究成果的总结

分支问题	难近似性	相关文献	最小化最大分支	相关文献
$2-\mathcal{UDT}$	$2-\epsilon, \forall\epsilon > 0$	[146]	$1+\ln m$ *	[148]
$\mathcal{UDT}$	$4-\epsilon, \forall\epsilon > 0$		$O(\log m)$ *	[149]
$2-\mathcal{DT}$	$\Omega(\log m)$		$O(\log m)$ *	[146]
$\mathcal{DT}$	$\Omega(\log m)$		$O(\log^2 m)$	

表格第二列列举了各个分支问题的难近似性，第四列列举了目前已证明的最小化最大分支在各个分支问题上的近似比，其中，* 标注为目前已知的最小近似比。参数 m 表示元素集合的大小，即 |X|。

4.3.5　询问选择问题与决策树的关系

基于询问选择问题的定义，本节介绍询问选择问题与决策树构建问题之间的关系。简单来说，补丁空间中的每个补丁对应决策树中每个元素，每个询问对应测试，每个答案对应测试结果。

4.3.5.1　从最优决策树问题到询问选择问题

本节首先介绍如果把最优决策树问题规约到询问选择问题。首先，定义一个从决策树问题上的元素和测试到询问选择问题中输入和输出的映射 π，该映射如下：

$$\pi(X, T, R, \varphi) = (\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi')$$

其中，

- X 是一个元素集合。
- T 是一个测试集合。
- R 是一个测试结果集合。
- φ 是元素在 X 上的概率分布。
- $\mathbb{P} = \{\pi_x(t) \mid x \in X\}$ 是一个基于 X 定义的补丁空间。令 $t_1, t_2, \ldots, t_n$ 为 T 中的测试，对于每个元素 x，应用上补丁之后的程序 $\pi_x(t)$ 的定义如下：

```
// program for $x$
if input == $t_1$ then return test result of $t_1$ on $x$
else if input == $t_2$ then return test result of $t_2$ on $x$
...
else if input == $t_{n-1}$ then return test result of $t_{n-1}$ on $x$
else return test result of $t_n$ on $x$
```

- $\mathbb{Q}$ 是一个和 T 完全一致的询问集合。
- $\mathbb{A}$ 是一个和 R 完全一致的答案集合。
- φ' 是一个 $\mathbb{P}$ 上补丁的概率分布，其中 $\varphi'(\pi_x(t))$ 等于 $\varphi(x)$。

可以发现 π 是一个多项式时间复杂度的映射。

接下来本节提出算法4，通过使用询问选择函数 QS 构造一个决策树。该算法的核心是函数 Construct，它以一个询问选择问题的实例（$\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi'$）以及一个交互历史 C 为输入，将所有可能的后续交互过程转化为一颗决策树。最终决策树的根节点对应了一个空的交互历史调用 Construct 的返回值（第 14 行）。函数 Construct 首先调用询问选择函数 QS（第 3 行），如果返回值为 $\top$，则构造一个叶子节点（第 4 行），否则构造一个内部节点（第 6-9 行）。当构造内部节点时，该节点的子节点可以通过在对应的交互历史上递归地调用函数 Construct 来得到。

Algorithm 4 使用询问选择函数构造一颗决策树

Require: 测试集合 T, 元素 X, 测试结果集合 R, 在元素 X 上的概率分布 φ，和询问选择函数 QS。

Ensure: 决策树的根节点。

```
1:  function Construct($\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi', C$)
2:      $node \leftarrow$ NewNode
3:      if $QS(C) = \top$ then
4:          $node$ 的标记 $\leftarrow \mathbb{P}|_C$
5:      else
6:          $q^* \leftarrow QS[\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi'](C)$
7:          $node$ 的标记 $\leftarrow q^*$
8:          $A \leftarrow \{\mathbb{D}[p](q^*) \mid p \in \mathbb{P}|_C\}$
9:          $node$ 的子节点 $\leftarrow \{$NewChild$(o,$ Construct$(\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi', C \cup \{(q^*, a)\})) \mid a \in A\}$
10:     end if
11:     return node
12: end function
13: $(\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi') \leftarrow \pi(X, T, R, \varphi)$
14: return Construct($\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi', ()$)
```

引理 1 算法4返回了一颗合法决策树。

证明 1 为了证明算法4是否返回一颗决策树，只需要检查算法4的返回结果是否满足决策树的性质。首先，由算法4的过程中可以发现，树上的所有内部节点，树的边，叶子节点都被正确标记，并且同一个节点的不同出边被不同的标签标记。第二，QS 只会返回可以进一步区分补丁的询问，因此，每个内部节点至少有两个子节点。第三，因为每一个叶子节点的标记都是在对应的交互历史下的合法补丁，因此标记上的所有元素一定都与祖先节点上的测试标记以及对应树边的答案标记保持着一致。第四，因为 QS 在无法区分任何补丁时会返回 $\top$，因此同一叶子节点上的任何元素都不可以为任意测试区分。最后，因为决策树的构造过程考虑了所有元素，因此任意两个叶子节点到根节点的路径都不相同，并且任意两个不可区分的元素，都被标记在同一个叶子节点上。

引理 2 给定元素 $x \in X$，在算法4返回的决策树上，x 到根节点的深度一定等于 $len(QS, x)$。

证明 2 给定元素 x，依次调用 QS 生成一些系列询问，len 为当前询问的长度。对询问序列的前 n 项进行归纳，可以看到（1）算法4返回的决策树上总有一条路径对应这 n 个询问；（2）当 QS 返回 $\top$ 时，这条路径必定到达了一个叶子节点且这个叶子节点的标记中必定包含 x。基于以上两点即可得到该引理。

引理 3 如果 QS 是多项式时间，那么算法4也是多项式时间。

证明 3 首先，每次调用 Construct 都会创建一个新节点，因此，Construct 的调用次数与决策树的大小是一致的，决策树的大小为元素大小 $|X|$，是多项式的，因此，Construct 的调用次数也是多项式的。其次，当 $QS[\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi']$ 是多项式的，Construct 中的每条语句也是基于 $|X|$ 和 $|T|$ 的多项式时间。综上，整个算法是多项式时间的。

上述引理说明 $\mathcal{DT}$ 可以在多项式时间内规约到 $\mathcal{OQS}$ 问题上，因此很多在 $\mathcal{DT}$ 问题上的理论结果可以直接应用到 $\mathcal{OQS}$ 问题上。

定理 2 以 k 的近似比近似 $\mathcal{DT}$ 问题可以在多项式时间内规约到以 k 的近似比近似 $\mathcal{OQS}$ 问题。

证明 4 通过引理3，引理2和引理1可得到以上定理。

推论 1 $\mathcal{DT}$ 可在多项式时间内规约到 $\mathcal{OQS}$。

证明 5 该推论是定理 2 中 k 为 1 时的特殊情况。

该结论可以推导出第4.3.2中的定理1。

推论 2 (定理 1) $\mathcal{OQS}$ 是一个关于补丁空间大小 $|\mathbb{P}|$，询问空间大小 $|\mathbb{Q}|$ 和答案空间大小 $|\mathbb{A}|$ 的 *NP* 难度问题。

证明 6 反证法，如果 $\mathcal{OQS}$ 不是 *NP* 难度问题，那么结合定理2，$\mathcal{DT}$ 也不是 *NP* 难度问题，矛盾。

令 $2-\mathcal{OQS}$ 为 $\mathcal{OQS}$ 在 $|\mathbb{A}=2|$ 时的子问题，$\mathcal{UOQS}$ 为 $\mathcal{OQS}$ 在 φ 服从均匀分布的子问题，$2-\mathcal{UOQS}$ 为 $\mathcal{OQS}$ 为以上两种情况同时成立的子问题。

推论 3 $\mathcal{OQS}$ 和 $2-\mathcal{OQS}$ 都是 $\Omega(\log|\mathbb{P}|)$ 难近似的, 对于任意 $\epsilon>0$，$\mathcal{UOQS}$ 是 $4-\epsilon$ 难近似的，对于任意 $\epsilon>0$，$2-\mathcal{UOQS}$ 是 $2-\epsilon$ 难近似的。

证明 7 与推论2类似，可以通过反证法证明。

4.3.5.2 从询问选择问题到最优决策树问题

本节说明询问选择问题可以在多项式时间内规约到最优决策树问题上，与之前类似，本节首先引入一个从 $\mathcal{OQS}$ 实例到 $\mathcal{DT}$ 实例之间的映射 ρ。ρ 的定义如下所示：

$$\rho(\mathbb{P},\mathbb{Q},\mathbb{A},\varphi') = (X,T,R,\varphi)$$

其中，

- $\mathbb{P}$ 是一个补丁空间。
- $\mathbb{Q}$ 是一个定义在 $\mathbb{P}$ 上的询问空间。
- $\mathbb{A}$ 是一个定义在 $\mathbb{P}$ 上的答案空间。
- φ' 是一个定义在 $\mathbb{P}$ 上的概率分布。
- X 是一个与 $\mathbb{P}$ 完全相同的元素集合。
- T 是一个与 $\mathbb{Q}$ 完全相同的测试集合，即对元素 $x\in X$ 执行测试 $q\in\mathbb{Q}$ 结果等于 $\mathbb{D}[x](q)$。
- R 是一个与 $\mathbb{A}$ 完全相同的结果集合。
- φ 是一个定义在 X 上的概率分布，与 φ' 完全相同。

不难发现，如果 $\mathbb{P}$ 中所有的补丁都是多项式时间的话，那么 ρ 的实现也是多项式时间，为了更好的讨论 $\mathcal{OQS}$ 的计算复杂度，本节忽略 $\mathbb{P}$ 中补丁本身的时间复杂度，假设 $\mathcal{D}$ 总可以在常数时间内计算得到。

Algorithm 5 使用决策树构造一个询问选择函数

Require: 补丁空间 $\mathbb{P}$, 询问空间 $\mathbb{Q}$, 答案空间 $\mathbb{A}$, 定义在 $\mathbb{P}$ 上的概率分布 φ , 交互历史 C（一个包含若干询问、答案对的序列）。

Ensure: 提供给用户的下一个问题。

```
1:  $(X, T, R, \varphi') \leftarrow \rho(\mathbb{P}, \mathbb{Q}, \mathbb{A}, \varphi)$
2:  $d \leftarrow$ Construct$(X, T, R, \varphi')$
3:  if $d$ 上有一条对应着交互历史 $C$ 的路径 then
4:      $path \leftarrow d$ 中对应着 $C$ 的路径
5:  else
6:      return $\top$
7:  end if
8:  $node \leftarrow path$ 的终止节点
9:  if $node$ 是一个叶子节点 then
10:     return $\top$
11: else
12:     return 节点 node 的标记
13: end if
```

基于 ρ，本节提出算法5，该算法首先决策树构造器 Construct 构造一棵决策树（第 2 行），然后找到当前交互式历史在决策树中对应的路径（第 3-7 行），最后根据该路径返回下一个向用户提问的问题（第 9-13 行）。

对于算法5，本节有以下三个引理。

引理 4 算法5定义了一个合法的询问选择函数。

证明 8 为了证明算法5定义了一个合法的询问选择函数，只需要证明它满足询问选择函数的两条性质。第一条性质可以通过决策树的最后一条性质得到，第二条性质可以通过决策树的中间节点至少有两个子节点来得到。

引理 5 令算法5定义的询问选择函数为 QS^*，给定补丁 $p \in \mathbb{P}$，决策树中 p 到根节点的路径长度为 $len(QS^*, p)$。

证明 9 以 p 为目标补丁，QS^* 为询问选择函数时，整个交互过程与决策树上从根节点到 p 对应的叶子节点遍历过程一致，因此询问次数与叶子节点的深度一致。

引理 6 如果 *Construct* 和 $\mathbb{P}$ 中的所有补丁都是多项式时间复杂度，那么算法5也是多项式时间复杂度。

证明 10 如果假设成立，算法5中每一条语句都是多项式时间，并且算法中没有循环与递归，因此算法5也是多项式时间复杂度。

定理 3 以 k 的近似比近似 $\mathcal{OQS}$ 问题可以在多项式时间内规约到以 k 的近似比近似 $\mathcal{DT}$ 问题。

证明 11 由引理6，引理4和引理5可以直接推导出该定理。

推论 4 $\mathcal{OQS}$ 可在多项式时间内规约到 $\mathcal{DT}$ 。

证明 12 该推论是定理 3 中 k 为 1 时的特殊情况。

4.3.5.3 在询问选择问题上应用最小化最大分支

基于询问选择问题与决策树之间关系的介绍，本节介绍将最小化最大分支算法应用到询问选择问题中。它基于如下的直观想法：为了最小化询问次数的期望值，在每一轮交互中，被选择的问题应当尽可能多地排除补丁。对于一个补丁集合 P，定义它的加权大小 $w(P)$ 为 $\sum_{p_0 \in P} \varphi(p_0)$, 即一个服从概率分布 φ 的补丁落在集合 P 内的概率。最小化最大分支策略在每一轮中都最小化了最坏情况下剩余的合法程序的加权大小。

Algorithm 6 一个最小化最大分支算法的简单实现

Require: 补丁空间 $\mathbb{P}$, 询问空间 $\mathbb{Q}$, 答案空间 $\mathbb{A}$, 定义在 $\mathbb{P}$ 上的概率分布 φ , 交互历史 C （一个包含若干询问、答案对的序列）。

Ensure: $\{\perp\} \cup \mathbb{Q}$ 中的一个询问。

```
$Q_o, c_o \leftarrow \perp, +\infty$
for $Q \in \mathbb{Q}$ do
    $c \leftarrow 0$
    for $A \in \mathbb{A}$ do
        $cost \leftarrow 0$
        for $p \in \mathbb{P}$ do
            if $p$ 与 $C + (Q, A)$ 一致 then
                $cost \leftarrow cost + \varphi(p)$
            end if
        end for
        if $cost > c$ then
            $c \leftarrow cost$
        end if
    end for
    if $c < c_o$ then
        $Q_o, c_o \leftarrow Q, c$
    end if
end for
return $Q_o$
```

定义 15 (最小化最大分支策略) 最小化最大分支策略是一个询问选择函数 *minimax*，它的定义如下:

- 如果所有补丁无法被进一步区分，即，$\forall p_1, p_2 \in \mathbb{P}|_C$，$\forall q \in \mathbb{Q}$，$\mathbb{D}[p_1](q) = \mathbb{D}[p_2](q)$，$minimax(C)$ 为 $\top$.
- 否则，$minimax(C)$ 被定义为：

$$minimax(C) = \arg\min_{q \in \mathbb{Q}} \left(\max_{a \in \mathbb{A}} w\left(\mathbb{P}|_{C \cup \{(q,a)\}} \right) \right)$$

推论 5 最小化最大分支策略是一个对询问选择问题的多项式时间近似算法。它对 $2-\mathcal{UOQS}$ 的近似比是 $1+\ln m$，对 $\mathcal{UOQS}$ 和 $2-\mathcal{OQS}$ 的近似比是 $O(\log m)$，对 $\mathcal{OQS}$ 的近似比是 $O(\log^2 m)$，其中 $m = |P|$，即补丁空间的大小。

证明 13 首先，令 QS 是利用算法4 转化最优决策树上的最小化最大分支后得到的询问选择函数。根据定理3，QS 在询问选择问题及其子问题上，有着和在最优决策树问题及其子问题上一样的近似比。然后，根据 OQS 上最小化最大分支的定义，QS 总是选择和最小化最大分支完全一样的询问。因此，在询问选择问题及其子问题上，最小化最大分支有着和在最优决策树问题及其子问题上完全一致的近似比。

算法6介绍最小化最大分支算法在询问选择问题上一个简单的实现，算法共有三层循环，其中最外层循环通过遍历询问空间 $\mathbb{Q}$，找到具有最小代价的询问（第 2-18 行），中间层循环通过遍历答案空间 $\mathbb{A}$，找到具有最大代价的答案（第 4-14 行），最内层循环通过遍历补丁空间 $\mathbb{P}$ 计算给定询问和答案对的代价（第 6-10 行）。

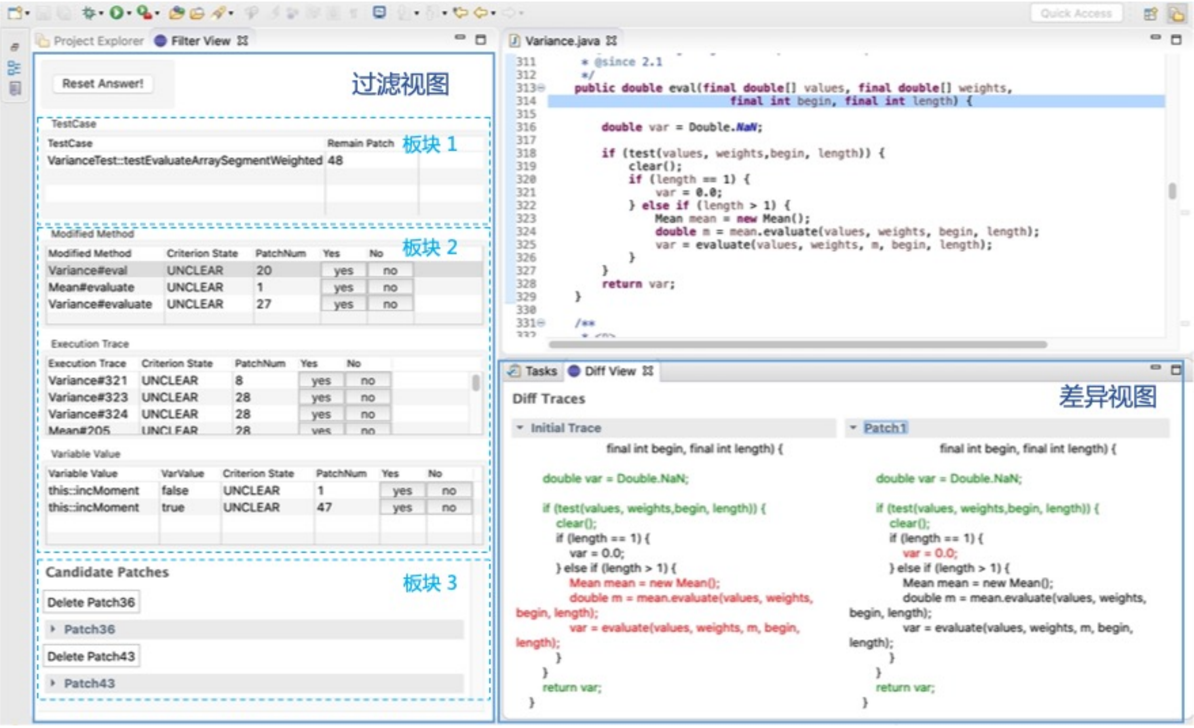
4.4　工具实现

本文将 *InPaFer* 技术实现为一个同名原型工具，该工具是一个 Eclipse 插件，图4.5展示了在调试图4.1中示例时 *InPaFer* 插件的截图。插件可以分为两个视图：过滤视图和差异视图。

过滤视图是插件的主视图，展示了过滤准则和候选补丁。它可以分为三个板块：(1) 板块 1 展示了失败测试用例集合和候选补丁的数目；（2）板块 2 展示了选中的失败测试用例对应的过滤准则的集合，每一行包括了属性的详细信息，该过滤准则对应候选补丁的个数以及该准则当前的状态（YES 代表肯定，NO 代表否定，UNCLEAR 代表还未回答）。属性详细信息对应了第4.2节中介绍的三类属性。例如，在 `Execution Trace` 栏中 *Variance#321* 行表示 “在类 *Variance* 中第 *321* 行代码应该被执行”。对于任意一个过滤准则，用户可以选择 ‘Yes’ 来过滤掉所有不满足该准则的候选补丁或者选择 ‘No’ 来过滤掉所有满足该准则的候选补丁，同时该行的状态也会随之改变；（3）当选中一个过滤准则时，板块 3 展示了该准则对应的候选补丁。另外，该插件提供了一键回滚功能退回到最初状态。

差异视图是一个辅助视图，可以可视化缺陷程序应用补丁前后失败测试用例执行路径的差异，绿色代码行代表修改前后都会被执行，红色代码行代表修改前后执行发生差异的语句，剩余行为修改行或修改前后均未被执行过的代码。用户可以通过这种方式清楚地理解补丁应用对程序执行的影响。

为了方便调试，所有的视图或者板块都是逻辑相关的，选择任何一个部分都可能会触发其它部分的响应或更新。例如，当选中板块 1 中的一个测试用例时，板块 2 会显示出与该测试用例相关的过滤准则。当选中板块 3 中的一个补丁时，差异视图会立刻刷新该补丁应用前后的程序执行差异。过滤视图也与 Eclipse 的代码编辑器相关，如果用户选择了 `Modified Method` 相关的过滤准则，代码编辑器中的光标会跳转到该修改函数的起始行。类似地，用户在选中一个补丁的同时可以直接在代码编辑器中看到该补丁的修改位置。



TestCase

TestCase	Remain Patch
VarianceTest::testEvaluateArraySegmentWeighted	48

Modified Method

Modified Method	Criterion State	PatchNum	Yes	No
Variance#eval	UNCLEAR	20	yes	no
Mean#evaluate	UNCLEAR	1	yes	no
Variance#evaluate	UNCLEAR	27	yes	no

Execution Trace

Execution Trace	Criterion State	PatchNum	Yes	No
Variance#321	UNCLEAR	8	yes	no
Variance#323	UNCLEAR	28	yes	no
Variance#324	UNCLEAR	28	yes	no
Mean#205	UNCLEAR	28	yes	no

Variable Value

Variable Value	VarValue	Criterion State	PatchNum	Yes	No
this::incMoment	false	UNCLEAR	1	yes	no
this::incMoment	true	UNCLEAR	47	yes	no

Candidate Patches

Delete Patch36

Patch36

Delete Patch43

Patch43

```
     * @since 2.1
     */
    public double eval(final double[] values, final double[] weights,
                       final int begin, final int length) {

        double var = Double.NaN;

        if (test(values, weights,begin, length)) {
            clear();
            if (length == 1) {
                var = 0.0;
            } else if (length > 1) {
                Mean mean = new Mean();
                double m = mean.evaluate(values, weights, begin, length);
                var = evaluate(values, weights, m, begin, length);
            }
        }
        return var;
    }

    /**
```

Diff Traces

Initial Trace

```
                 final int begin, final int length) {

    double var = Double.NaN;

    if (test(values, weights,begin, length)) {
        clear();
        if (length == 1) {
            var = 0.0;
        } else if (length > 1) {
            Mean mean = new Mean();
            double m = mean.evaluate(values, weights,
begin, length);
            var = evaluate(values, weights, m, begin,
length);
        }
    }
    return var;
  }
```

Patch1

```
                 final int begin, final int length) {

    double var = Double.NaN;

    if (test(values, weights,begin, length)) {
        clear();
        if (length == 1) {
            var = 0.0;
        } else if (length > 1) {
            Mean mean = new Mean();
            double m = mean.evaluate(values, weights,
begin, length);
            var = evaluate(values, weights, m, begin,
length);
        }
    }
    return var;
  }
```

图 4.5　*InPaFer* 插件的屏幕截图。板块 1 展示了当前剩余候选补丁的数量和失败测试用例集合；板块 2 展示了基于三种属性的过滤准则（即关于补丁特性的询问列表）；板块 3 展示了候选补丁的详细信息。

4.5　实验验证

4.5.1　实验设置

4.5.1.1　研究问题

为了验证 *InPaFer* 的效果，本节探究下述研究问题：

- RQ1： *InPaFer* 在过滤错误补丁上的效果如何？
- RQ2：不同程序属性如何影响 *InPaFer* 的过滤效果？
- RQ3：开发者的回答的正确率如何影响 *InPaFer* 的过滤效果？
- RQ4：在实际调试场景中，*InPaFer* 能否帮助开发者提升修复效率和正确率？
- RQ5：开发者是否认为 *InPaFer* 有用，如果是，*InPaFer* 的有用性提现了哪些方面？
- RQ6：与使用自动定位信息相比，使用 *InPaFer* 辅助调试效果如何？

本文通过对照实验度量 *InPaFer* 对错误补丁的过滤比率和询问次数定量回答前三个研究问题，除了对照实验之外，本文还通过两个用户实验定性回答剩下三个研究问题，其中第一个用户实验（用户实验 I）通过验证参与者使用 *InPaFer* 与其它两种调试场景下（不提供任何辅助信息和提供候选补丁）的调试效果差异来回答 RQ4 与 RQ5。根据第一个用户实验的结果，本文进行了第二组用户实验（用户实验 II），通过验证参与者使用 *InPaFer* 与自动定位结果之间的调试效果差异来回答 RQ6。在每个用户实验中，本文向开发者提供了一个 Eclipse 插件，并验证在该插件的辅助下，开发人员的缺陷修复准确率和效率，用户实验均使用威尔科克森秩和检验（Paired Wilcoxon rank-sum test）来确定统计显著性。

4.5.1.2　数据集

为了验证 *InPaFer* 在修复工具召回率较高但是准确率较低时是否仍然能辅助开发者修复缺陷，本文收集了截止至 2019 年 5 月（本实验的开始时间）已经发布在program-repair.org网站①上的修复工具在 Defects4J[52] 生成且开源的补丁，Defects4J 是在缺陷自动修复领域被广泛使用的 Java 缺陷上的数据集。最终，本文共收集 13 个自动修复工具在 5 个项目的 88 个缺陷上生成的补丁。其中，自动修复工具的详细信息如表4.6所示，收集到的补丁及项目信息如表4.7所示。

---

① 缺陷自动修复领域内著名的社区，收集了缺陷自动修复领域内的相关论文，工具，数据集等信息。

表 4.6　验证 *InPaFer* 效果涉及到的修复工具集合

工具名称	**工具介绍**
jKali[18]	Kali[128] 工具的 Java 版本实现, 生成删除代码的补丁。
jGenProg[18]	GenProg[150,151] 工具的 Java 版本实现, 使用遗传算法修复缺陷。
kPAR[152]	PAR[153] 工具的 Java 版本实现, 使用预定义的模板修复缺陷。
Nopol[15]	使用约束求解修复条件类型缺陷。
jMutRepair[154]	使用预定义的变异算子修复缺陷
Cardumen[155]	基于挖掘的模板修复缺陷。
Avatar[13]	修复静态分析检测出的缺陷。
HDrepair[14]	基于历史修复信息修复缺陷。
ACS[42]	在开源网站上利用统计学习信息修复条件类型缺陷。
3sfix[156] CapGen[43] SimFix[44]	基于相似代码匹配修复缺陷。
DeepRepair[81]	jGenProg 的扩展版本，使用代码相似度修复缺陷。

表 4.7　实验中使用的数据集

项目	**代码行**	**缺陷数**	**平均补丁数**	**包含/不包含**
Chart	96K	17	225	9/8
Closure	90K	13	6	3/10
Lang	22K	13	66	8/5
Math	85K	42	92	15/27
Time	28K	3	9	1/2
总计	321K	88	99	36/52

平均补丁数：平均每个缺陷的候选补丁数量；**包含/不包含：**候选补丁中包含正确补丁的缺陷个数和不包含正确补丁的缺陷个数。

4.5.2　实验过程

4.5.2.1　回答对照实验的过程

RQ1: 为了模拟使用 *InPaFer* 交互式过滤补丁，对于一个给定的缺陷及其对应的补丁集合，*InPaFer* 每次随机选择一个过滤准则，然后根据应用正确补丁之后的程序分析结果，为选取的过滤准则提供一个正确的答案，当不存在任何过滤准则的时候，交互过程结束。本文实验重复该过程十次并计算十次的平均结果。在这个过程中，只模拟了过滤步骤，并且不会有正确补丁被过滤出去。

本实验还研究了询问方式对于询问次数的影响，特别的，本实验设计了八种不同的询问顺序，覆盖了不同的应用场景。其具体策略如表4.8所示，随机策略模拟了开发者每次随机选择一个过滤准则回答，最小化最大分支策略为第4.3节中介绍的目前效果最好的交互式询问策略，剩余六种策略按照程序属性顺序选择过滤准则，一种类型的程序属性内随机选择一个过滤准则。同样，为了避免实验随机性造成的干扰，本实验

表 4.8　八种不同的询问顺序

询问顺序	**询问顺序介绍**
随机策略	每次随机选择一个过滤准则
最小化最大分支策略	每次选择最坏情况下最多过滤错误补丁的过滤准则
M-T-V	按照 修改方法-执行路径-变量值的顺序选择过滤准则
M-V-T	按照 修改方法-变量值-执行路径的顺序选择过滤准则
T-V-M	按照 执行路径-变量值-修改方法的顺序选择过滤准则
T-M-V	按照 执行路径-修改方法-变量值的顺序选择过滤准则
V-M-T	按照 变量值-修改方法-执行路径的顺序选择过滤准则
V-T-M	按照 变量值-执行路径-修改方法的顺序选择过滤准则

重复该过程十次并计算平均结果。

RQ2: 为了探究不同类型程序属性对 *InPaFer* 过滤错误补丁效果的影响，本节又进行了三次实验，每次移除一种类型的程序属性对应的过滤补丁，采用随机策略度量每次实验下过滤错误补丁的比率。

RQ3: 目前为止，以上实验都假定开发者在交互过程中不会回答错误，在实际中，开发者不可避免的会错误回答过滤准则。为了模拟交互过程中开发者以一定的错误率回答过滤准则，在本组实验中，*InPaFer* 随机选择一个过滤准则，然后以一定错误率给出一个答案。本组实验考虑的错误率的范围为从 2% 到 10%，以 2% 递增。同样，该实验也被重复十次，并取平均结果。

4.5.2.2　回答用户实验 I 的实验过程

用户实验设计: 为了研究 *InPaFer* 在实际修复场景中对开发者的帮助程度，本节设计了如下三种调试场景：

- 人工修复：在不提供候选补丁和 *InPaFer* 的情况下，参与者修复缺陷。
- 补丁修复：在提供候选补丁的情况下，参与者修复缺陷。
- *InPaFer* 修复：在提供 *InPaFer* 的情况下，参与者修复缺陷。

在以上三种调试场景中，参与者都可以按需求使用 Eclipse 的调试工具并执行测试用例。

任务: 因为缺陷对应的候选补丁集合可能不存在正确补丁，因此本实验中设计了两种类型的任务对应两种场景：（1）存在正确补丁；（2）不存在正确补丁。*InPaFer* 的目标是辅助开发者过滤错误补丁最终找到或提供正确补丁，因此本实验只考虑候选补丁数目大于 3 的缺陷，并且修复该缺陷不需要专业领域知识。在对照实验中使用的 Defects4J 数据集（见表4.7）中，本文共识别了符合以上条件的 38 个缺陷，这些缺陷对应的候选补丁中位数为 20。在这 38 个缺陷中，26 个缺陷的候选补丁集合中包含正确补丁，12 个缺陷的候选补丁集合中不包含正确补丁。本文从以上两类数据中随机选

表 4.9　用户实验中的缺陷

任务 ID	缺陷 ID	补丁数 (正确补丁数)	剩余错误补丁数
任务 1	Chart9	26 (4)	12
任务 2	Math41	48 (1)	1
任务 3	Lang14	8 (0)	0
任务 4	Lang22	24 (0)	0

择了 2 个缺陷及其对应的补丁，如表4.9所示，任务 1 和任务 2 对应候选补丁集合包含正确补丁的缺陷，任务 3 和任务 4 对应候选补丁集合不包含正确补丁的缺陷。实验参与者在实验过程中不会被告知每个任务中的候选补丁集合中是否包含正确补丁。

在用户实验中增加调试任务数量是一件相对困难的事情，目前本实验中涉及到的调试任务数量已经不小于最近多篇发表于顶会工作的用户实验中的任务数量了（1-4 个任务）[157–160]。为了缓解任务量少对实验结论的影响，用户实验中不仅考虑统计数据上的实验结果，还分析了用户在不同任务上修复效果不同的原因。另外，用户实验中选取四个任务都是可以体现 *InPaFer* 过滤效果的代表性任务（RQ1 中的错误补丁过滤率和询问次数），在这四个任务中，*InPaFer* 可以全部过滤掉三个任务的错误补丁，需要询问次数平均在 2 到 4.8 次不等。

试点实验（pilot study）：为了探究用户实验的一些设置，例如任务的描述，任务的限时等，在正式的用户实验之前，还进行了一个试点实验。在试点实验中，十名来自计算机系的学生作为参与者使用 *InPaFer* 插件在两个任务上进行了调试实验。完成该实验之后，根据参与者的建议，作者对 *InPaFer* 的 UI 设计以及用户实验的一些设定做了改进，试点实验的参与者不会参与最终的实验。为了保证实验在有限参与者的情况下结果显著，试点实验还对每个调试任务的限时做了评估，试点实验评估在人工修复情景下调试时间约为 25 分钟，在 *InPaFer* 修复情景下调试时间约为 20 分钟，因此在最终用户实验中设定每个调试任务的限时为 30 分钟。如果在限定时间内一个调试任务被正确修复，那么认为该认为是一个成功的调试任务。

参与者: 共有 30 名来自计算机系的学生参与到用户实验 I 中，这些参与者至少有三年编程经验并且熟悉 Eclipse 调试。另外，参与者对于实验中的所有缺陷调试任务没有任何先验知识。

实验步骤: 如图4.10所示，用户实验 I 中的每个参与者被随机平均分配到三个组中（A 组，B 组和 C 组），每个参与者需要完成两种调试场景下的四个调试任务，例如，A 组中的参与者需要人工修复（在不提供候选补丁和 *InPafer* 的情况下修复）任务 1 和任务 3，补丁修复（在提供候选补丁的情况下修复）任务 2 和任务 4。总得来看，在用户实验 I 共包含了 120（30 × 4）个调试单元，这个规模已经显著大于最近发表于顶会的

表 4.10　用户实验 I 的分组情况

分组 (参与者 ID)	人工修复	补丁修复	*InPaFer* 修复
A 组 (P1-P10)	任务 1+3	任务 2+4	–
B 组 (P11-P20)	任务 2+4	–	任务 1+3
C 组 (P21-P30)	–	任务 1+3	任务 2+4

交互调试工作的规模（20-48 个调试单元）[157–160]。

为了方便参与者熟悉 *InPaFer* 的使用，在正式实验之前，参与者可以先完成一个训练任务：在介绍完 *InPaFer* 的使用方法之后，B 组和 C 组的参与者使用 *InPaFer* 对一个不相关的缺陷进行调试。当参与者完成一个调试单元之后，作者会人工检查缺陷是否被正确修复。

在完成所有调试任务之后，针对每个调试单元，本文设计了一个关于调试体验和建议的开放问题访谈。具体而言，对于每个参与者。作者会询问两种调试场景下的差异，对于使用 *InPaFer* 进行调试过的参与者，作者会额外询问对 *InPaFer* 的改进意见。

访谈分析: 本文主要从定性角度分析参与者通过完成调试任务学习到的内容以及他们如何与 *InPaFer* 进行交互。本工作的两个作者按照定性实证实验研究的标准方法对访谈内容进行转录和编码[161]。

4.5.2.3　回答用户实验 II 的实验过程

在用户实验 I 中，对于所有的调试任务，至少有一名参与者提及 *InPaFer* 帮助他们定位到缺陷代码的位置，因此用户实验 II 探究 **RQ6: 与使用自动定位信息相比，使用 *InPaFer* 辅助调试效果如何？**如果使用 *InPaFer* 的参与者的调试效果没有比使用定位信息的参与者的调试效果更优，那么只提供定位信息不提供补丁也可以帮助开发者修复缺陷。

用户实验 II 中，参与者被分为两组，一组参与者使用 *InPaFer* 修复缺陷，另外一组参与者使用候选缺陷位置列表修复缺陷。特别地，本文选了语句级别的缺陷位置，因为语句级别的缺陷位置能够提供给开发者更细致的信息以及与 *InPaFer* 提供的定位信息粒度一致。本文选了目前在 Defects4J 数据集上语句级别定位效果最好的工具 CombineFL[162] 的定位结果提供给参与者。

实验设计: 用户实验II的实验设置与用户实验I的实验设置是一样的(见第4.5.2.2节)，除了控制组的调试场景与参与者的不同。用户实验 II 设置了两种调试场景：

- 位置修复：在提供一个有序的缺陷定位列表的情况下，参与者修复缺陷。
- *InPaFer* 修复：在提供 *InPaFer* 的情况下，参与者修复缺陷。

用户实验 II 共有 20 名参与者，包括 12 个计算机系的学生和 8 个来自公司的开发

表 4.11　用户实验 II 的分组情况

分组 (参与者 ID)	位置修复	*InPaFer* 修复
D 组 (P1-P10)	任务 1+3	任务 2+4
E 组 (P11-P20)	任务 2+4	任务 1+3

表 4.12　*InPaFer* 的过滤效果

错误补丁过滤率	0	(0, 100%)	100%	Total
缺陷数目	21	25	42	88

错误补丁过滤率: *InPaFer* 对一个缺陷的候选补丁中错误补丁的比率。

者。他们都至少有一年的 Java 编程经验并且熟悉 Eclipse 的使用和调试。如表4.11所示，参与者被平均分为两组（D 组和 E 组），每个参与者在两种调试场景下完成四个修复任务。和用户实验 I 一样，为了理解参与者的调试过程，在完成所有任务之后，本文对每个参与者进行一个访谈。

4.5.3　实验结果

4.5.3.1　RQ1：过滤效果与询问次数

过滤效果：表4.12展示了 *InPaFer* 在所有缺陷上的过滤效果。结果表明，对于 47.7% (42/88) 的缺陷，*InPaFer* 可以过滤掉全部错误补丁，对于 23.9% (21/88) 的缺陷，*InPaFer* 不能过滤掉任何错误补丁，对于 28.4% 的缺陷，*InPaFer* 可以部分过滤掉错误补丁。

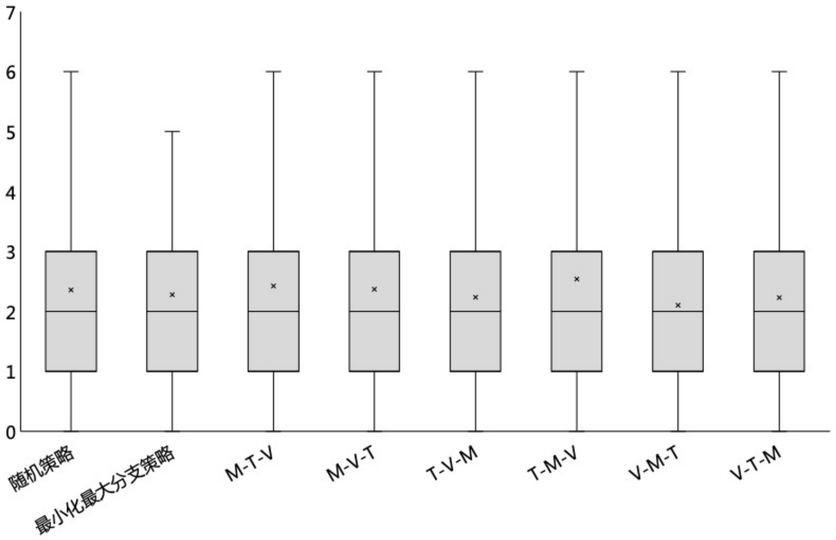


图 4.6　不同询问策略下的询问次数

为了研究 *InPaFer* 无法过滤掉错误补丁的原因，本文从 46 个缺陷中随机抽取了 5

```
1  // correct patch                 | // incorrect patch
2  - if (dataset != null) {         |  - if (dataset != null) {
3  + if (dataset == null) {         | + if (AbsRenderer.ZERO == null) {
4       return result;              |       return result;
5    }                              |    }
```

图 4.7　无法被区分的两个候选补丁示例

个缺陷，并人工检查了对应的候选补丁集合。结果表明，执行失败测试用例时应用这些补丁的程序状态是一样的。例如，图4.7是 *Chart1* 的候选补丁经过过滤之后无法被区分的两个补丁示例，左侧是正确补丁，右侧是错误补丁，它们都修改了第 2 行的 `if` 条件，并且具有相同的执行路径，因此当前实现版本的 *InPaFer* 无法区分它们。然而，*InPaFer* 可以通过增加其他种类的程序属性来提升过滤效果，例如不变量信息[163]，这可以留作未来工作。

询问次数：图4.6展示了八种询问策略的询问次数结果，从表中可以看出，八种策略的询问次数没有显著差异。当采用随机策略时，*InPaFer* 需要中位数 2.2 次询问（标准差为 2.6 次）。当前数据集中每个缺陷对应的候选补丁数目中位数为 7，也就是说，*InPaFer* 可以将审阅 7 个补丁的工作量转化为回答 2.2 个询问。

本文随机选取了十个缺陷，并人工检查对应的补丁过滤结果，发现通常只有一个或两个程序属性可以区分候选补丁，这可能导致了不同顺序的程序属性的询问次数是接近。例如，如果一组候选补丁只能通过修改方法和变量值对应的过滤准则区分，那么 M-V-T 与 T-M-V（T：执行路径）两种顺序的询问次数是一致的。最小化最大分支策略，每次会优先选择覆盖接近一半补丁的过滤准则，而本文发现，不同过滤准则对应的补丁数量往往比较接近，因此这种策略与其它策略相比也没有显著差异，但是当询问数量较多且每个询问对应的补丁个数差异较大时，不同询问策略的询问次数会有较大差异，而最小化最大分支策略会优于其它策略。

4.5.3.2　RQ2：**程序属性的敏感性分析**

为了研究 *InPaFer* 过滤效果对程序属性的敏感性，本文进行了三次实验，每次移除掉一种类型的程序属性对应的过滤准则，采用随机策略度量每次实验过滤错误补丁的比率。结果如表4.13所示，移除任何一种类型的程序属性的过滤准则都会影响 *InPaFer* 过滤错误补丁的效果，所有的属性对于过滤错误补丁都十分重要，其中，变量值的影响最大，当移除掉变量值类型的过滤准则之后，*InPaFer* 过滤掉全部错误补丁的缺陷数量减少了 15 个（42-27）。

表 4.13　程序属性的敏感性分析

缺陷 过滤率 属性	$< 100\%$	$= 100\%$
所有属性	46	42
所有 - 修改方法	51	37
所有 - 执行路径	52	36
所有 - 变量值	61	27

过滤率：过滤错误补丁的比率。

4.5.3.3　RQ3：**错误率的敏感性分析**

为了研究 *InPaFer* 过滤效果对开发者错误答案的敏感性，本文度量了开发者以一定错误率给出询问的答案时，*InPaFer* 可以过滤掉所有错误补丁并保留至少一个正确补丁的缺陷数量，特别地，本实验考虑错误率范围为 2% 到 10%，以 2% 递增。结果如表4.14所示，可以发现缺陷数目与错误率呈负相关。另外，即使开发者为过滤准则提供答案的错误率为 10% 的情况下，*InPaFer* 仍然能为 42%（37/88）的缺陷过掉错误补丁。

表 4.14　错误率的敏感性分析

错误率	0	2%	4%	6%	8%	10%
缺陷数目	42	40	40	38	37	37

缺陷数目：*InPaFer* 过滤掉全部错误补丁
且保留正确补丁的缺陷个数。

4.5.3.4　RQ4：**用户实验 I 的结果**

正确率： 表4.15展示了用户实验 I 中三种修复场景下成功完成的调试单元个数，*InPaFer* 修复场景下成功修复的缺陷个数比人工修复场景下成功修复的缺陷个数显著多 62.5%($p = 10^{-4}$)，比补丁修复场景下的成功修复的缺陷个数显著多 39.3%($p = 10^{-3}$)。特别地，当候选补丁集合中包含正确补丁时（任务 1 和任务 2），*InPaFer* 修复场景下，所有参与者都可以成功修复缺陷，但是补丁修复场景下虽然具有相同的补丁，但是并不是所有参与者都可以成功修复缺陷。

调试时间： 图4.8展示了用户实验 I 中三种调试场景下的调试时间。对于所有的修复任务，*InPaFer* 修复场景下的调试时间比补丁修复场景下的修复时间显著缩短了 28.0%（$p = 4 \times 10^{-3}$），比人工修复场景下的修复时间显著缩短了 25.3%（$p = 10^{-5}$）。即使所有的候选补丁都是错误补丁（任务 3 和任务 4），在 *InPaFer* 修复场景下的成功修复缺陷个数比人工修复场景的成功修复缺陷个数多 27%，修复时间缩短了 13%。

表 4.15　用户实验 I 中成功完成的调试单元

任务 ID	**人工修复**	**补丁修复**	***InPaFer* 修复**
任务 1	1	9	10
任务 2	8	6	10
任务 3	5	8	10
任务 4	10	5	9
总计	**24**	**28**	**39**

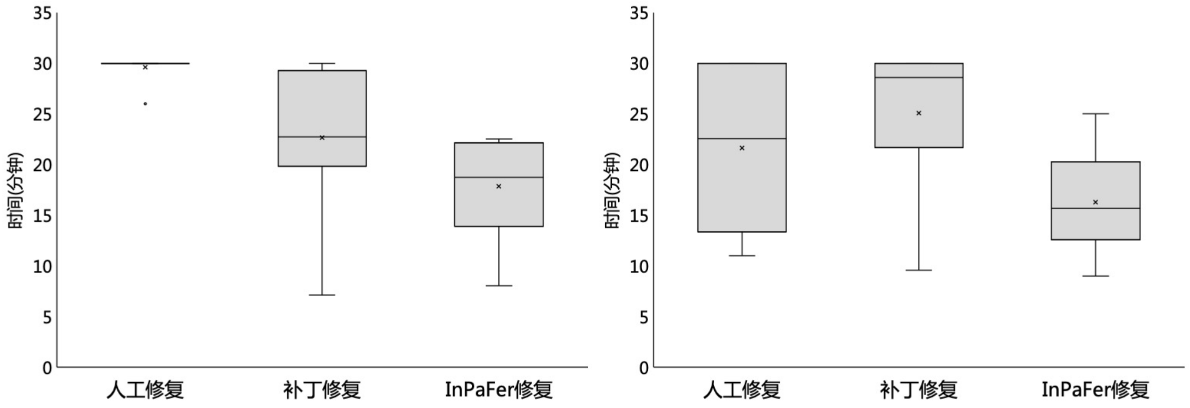
参与者调试结果与补丁作用之间的关系：本文注意到，并不是在所有任务上 *InPaFer* 修复场景下的修复结果都最优：在任务 4 上，*InPaFer* 修复场景下的结果略差于人工修复场景下的结果。另外，补丁修复场景下的修复效果并不取决于候选补丁集合中是否包含正确补丁：补丁修复场景下的修复结果在任务 1 和任务 3 上优于人工修复场景下的结果，而任务 1 的候选补丁中包含一个正确补丁，任务 3 的候选补丁都是错误补丁。

出现以上现象的可能原因是补丁的作用。本文观察到，即使一个补丁是错误的，它也可能为参与者修复缺陷提供启示。任务 1 中几乎所有候选补丁都修改了缺陷位置，这也提示了参与者可能的缺陷位置。例如，参与者 P23 说“补丁帮助我定位了缺陷代码”。任务 3 中的补丁，虽然是错误补丁，但是包含了部分正确补丁代码，这也有助于参与者完成修复。例如，参与者 P22 说“补丁为我提供了应该使用的 *API* 函数调用”。用户实验访谈表明，在补丁修复和 *InPaFer* 修复两种场景下 20 名参与者中有 14 名参与者（P11-P14, P17-P20, P22-P25, P29, P30）提及任务 1 和任务 3 中的补丁有助于他们定位缺陷位置或提供部分正确代码。另外一方面，任务 2 中所有错误补丁都修复了不同的非缺陷位置，任务 4 中的错误补丁提供了可能会误导参与者的无意义代码。

以上发现表明未来的自动修复技术研究工作不仅可以关注补丁正确性，还可以关注补丁的有用性，即补丁是否可以为开发者提供有用的信息。当前自动修复技术的衡量指标为生成正确补丁的个数，在此，本文认为可能存在一个更细粒度的衡量指标，该指标可以考虑每个补丁为开发者提供帮助的程度。

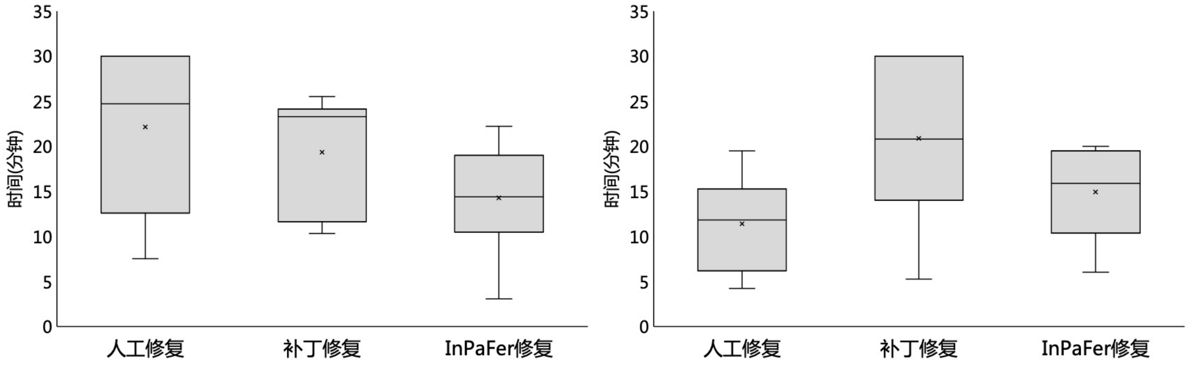
本文注意到，即使所有补丁都是错误且无用的，例如，任务 4，*InPaFer* 修复场景下的结果与人工修复场景下的结果是接近的，这一现象也可以证实本文的假设：*InPaFer* 的过滤过程也可以帮助开发者理解缺陷，有助于提升整个修复效果。

发现 10 与人工修复和补丁修复两种场景下的修复结果相比，*InPaFer* 修复场景下的修复正确率分别提升了 *62.5%* 和 *39.3%*，修复时间分别缩短了 *25.3%* 和 *28.0%*。即使所有候选补丁都是错误的，开发者的修复结果也没有显著降低，仍然可以提高。



(a) 任务 1 (Chart9)

(b) 任务 2 (Math41)



(c) 任务 3 (Lang14)

(d) 任务 4 (Lang22)

图 4.8　用户实验 I 中的调试时间

4.5.3.5　RQ5：**用户实验 I 的访谈结果**

本文分析了访谈结果，参与者从以下三个方面说明了 *InPaFer* 的有用性：

- **过滤功能可以帮助参与者减少审阅补丁的数量：** 12 个参与者（共 20 个）（P13, P17, P20, P22-P30）提及 *InPaFer* 可以帮助他们快速的过滤出不正确补丁，例如，P20 提及“经过几步过滤之后，我只需要审阅少量的补丁，工具十分可靠”。
- **与 *InPaFer* 交互可以帮助参与者理解缺陷：** 很多参与者自己生成补丁修复缺陷，不过仍然肯定了交互过程帮助他们理解了缺陷，例如，任务 2 的候选补丁集合中包含了很大比例的错误补丁，P21 说“一开始我不知道如何修复该缺陷，但是我可以确认或拒绝每个过滤准则，这样交互几次之后，我逐渐理解了这个缺陷并且知道了如何修复它”。总的来说，使用 *InPaFer* 调试的参与者对 16 个调试单元（共 40 个）提及过滤补丁可以帮助他们理解缺陷。
- **UI 界面设计帮助参与者浏览和访问关键信息：** Eclipse 插件（图4.5）将不同的视图联系起来，这有助于用户浏览缺陷。例如，当选中板块 3 中的一个补丁时，差异视图会立刻刷新该补丁应用前后的程序执行差异，参与者对于这一设计给

表 4.16　用户实验 II 中成功完成的调试单元

任务 ID	**位置修复**	**InPaFer 修复**
任务 1	2	9
任务 2	8	9
任务 3	10	10
任务 4	7	9
总计	**27**	**37**

出了积极的反馈，例如，对于候选补丁集合中不包含任何正确补丁的任务 3 来说，P11 说“当我选中修改方法这一过滤准则时，工具会将缺陷定位到 `equals` 函数，这帮助我理解了 `equals` 函数应该被调用。此外，差异视图展示出了测试通过时，哪个程序分支应该被执行，这纠正了我之前的理解错误”。

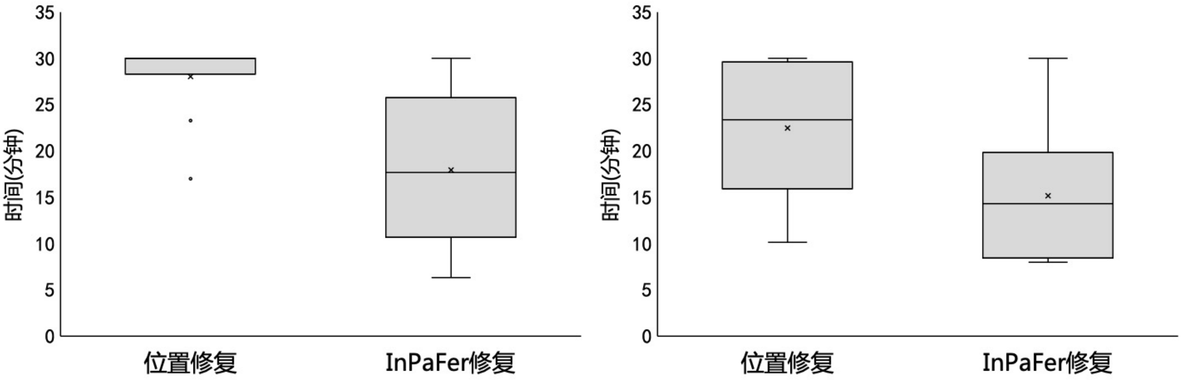
4.5.3.6　RQ6：**用户实验 II 的结果**

正确率：表4.16展示了用户实验 II 中成功完成的调试单元个数，*InPaFer* 修复场景下的正确修复的缺陷数目显著比位置修复场景下的正确修复的缺陷数目多 37.0%（$p = 2 \times 10^{-3}$）。

调试时间：图4.9展示了用户实验 II 中两种调试场景下的调试时间。对于任务 3&4 来说，两种调试场景下的时间没有显著差异（$p > 0.1$）。这是因为在 CombineFL 为这两个任务生成的定位结果中，真实的缺陷位置分别在第一位和第二位。实验参与者也确认当检查了代码和缺陷位置之后，很容易可以生成补丁。

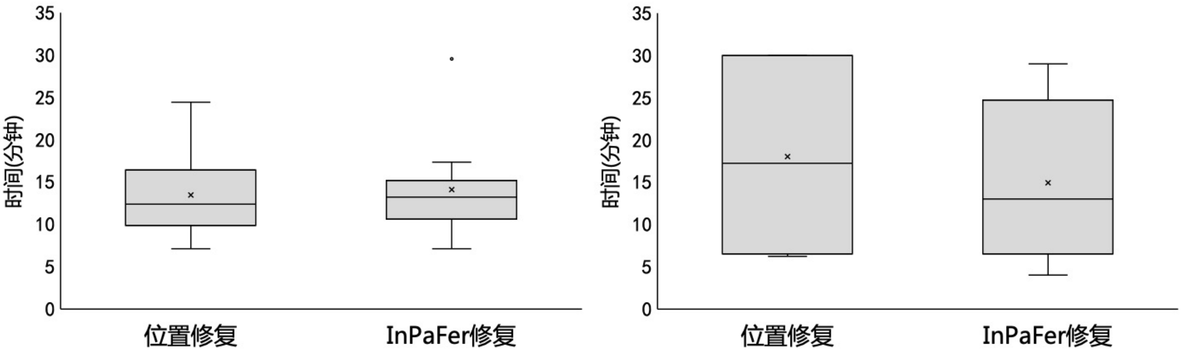
在任务 1&2 中，真实的缺陷位置分别排在第十三位和第十八位，这为参与者调试缺陷带来的收益较小。相比于位置修复场景下的修复时间，*InPaFer* 修复场景下的修复时间缩短了 34.4% （$p = 4 \times 10^{-4}$）。11 个参与者（共 20 个）认为如果给定的有序缺陷列表中的前三位不是真实的缺陷位置，那么他们将不会继续去参考剩下的缺陷位置，而对于在 *InPaFer* 修复场景下的参与者来说，除了位置信息之外，*InPaFer* 还可以提供更多有用的信息，比如执行信息（P31，P32，P37，P38，P43）和候选补丁的提示信息（P32，P34，P35，P37，P38，P40，P47）。这个结果表明候选补丁能够为修复过程提供除了位置信息之外的更多帮助，因此，对于自动修复技术来说生成候选补丁的时间并没有浪费。

发现 11　相比于位置修复场景，*InPaFer* 修复场景下的修复正确率提升了 *37.0%*，修复时间平均缩短了 *24.2%*。



(a) 任务 1 (Chart9)

(b) 任务 2 (Math41)



(c) 任务 3 (Lang14)

(d) 任务 4 (Lang22)

图 4.9　用户实验 II 中的调试时间

4.5.3.7　实验有效性

内部威胁在于访谈中存在交流问题可能会影响最终结论。为了缓解这方面的威胁，当访谈中的问题引起困惑时，会重新解释访谈内容，并且在每个访谈中都有两位研究者参与。尽管本文按照开放式问题的原则精心设计了访谈的步骤，但是这个过程中不可避免的会存在“确认偏差（*comfirmation bias*）’’，例如参与者可能会遗漏某些重要观点。为了缓解这个威胁，本文主要关注访谈中隐藏的见解而不仅仅是显式声明的内容。

外部威胁在于以下三方面。首先，用户实验中关注 Defects4J 数据集中的四个缺陷调试任务，该结果可能不能泛化到其它情况。为了缓解这个威胁，本文不仅仅依赖于四个调试任务上的统计数据结果（正确率和调试时间），还细致地分析了每个任务上调试结果差异的原因。另外，就 *InPaFer* 在整个数据集上性能而言，本文认为这四个任务是具有代表性的，在这四个任务中，*InPaFer* 可以在其中三个任务上过滤掉完全错误补丁，整体的询问次数平均为 2-4.8 次。在用户实验中增加调试任务数量是一件相对困难的事情，目前本实验中涉及到的调试任务数量已经不小于最近多篇发表于顶会工作中用于验证交互式调试技术的用户实验中的任务数量了（1-4 个任务）[157–160]。其次，和类似的实验一样，用户实验的研究结果可能会存在参与者选择偏差，为了减少选择偏差，

本文通过公共渠道招募参与者，并将他们随机划分为五组。为了避免由于项目的真实开发者与实验中的学生参与者之间的差异造成的影响，用户实验中的调试任务都是只需要基本的调试经验可以完成不需要特定领域知识。总的来说，本文选择的参与者是具有不同的背景并且没有观察到任何系统性差异。最终，在用户实验 I 共包含了 200（50 × 4）个调试单元，这个规模已经显著大于最近发表于顶会的交互调试工作的规模（20-48 个调试单元）[157–160]，并且本文认为该实验规模已经使结果达到饱和。最后，在定量实验中，本文模拟了八种策略下的交互式补丁过滤询问方式，但是可能仍然存在其它询问情况未进行探讨，为了缓解这方面的威胁，对照实验都重复了十次，并取平均值作为最终结果，请注意，虽然定量实验的目标是通过某些统计数值衡量过滤准则的过滤效果，*InPaFer* 的有效性和有用性还是由用户实验来验证。

4.6　讨论与小结

为辅助开发者判别补丁正确性，完成补丁审阅，提升修复效率与正确率，本章提出交互式补丁过滤技术 *InPaFer* 。通过三种类型的过滤准则与开发者进行交互，辅助开发者理解缺陷及补丁并过滤错误补丁。为了及时响应用户的反馈，*InPaFer* 采用了两阶段算法，即准备阶段收集程序属性和构建过滤准则，交互阶段与用户在线交互，辅助审阅补丁并过滤错误补丁。为了提升交互效率、减少与用户之间的交互次数，本章定义了询问选择问题，并理论证明了询问选择问题与最优决策树之间的关系，进而引入最优决策树中的最小化最大分支算法优化交互次数。最终，本章设计与用户的交互界面，将 *InPaFer* 实现为一个开源 Eclipse 插件。为验证 *InPaFer* 的效果，本章同时进行了定量实验与用户实验。用户实验结果表明，使用 *InPaFer* 进行调试的开发者比对照组开发者修复准确率提升了 62.5%，修复时间减少了 25.3%。该实验结论说明：在合适的工具的辅助下，补丁审阅阶段也可以帮助开发者理解缺陷，完成修复，低准确率的修复工具在实际中也可以发挥作用。

本章的研究结果也为未来的自动修复技术提供了一些可能性：

- **未来自动修复技术可以通过降低对准确率的要求，生成更多补丁从而修复更多缺陷**。为了避免加重开发者审阅补丁的负担，目前的修复技术通常为一个缺陷只生成一个补丁。本章实验结果说明 *InPaFer* 在修复工具具有较低准确率时可以提升开发者的修复效率与正确率。该结果表明，只要有合适的辅助审阅工具支持，即使生成了很多不正确补丁的自动修复工具仍然是有用的。本文建议未来修复技术可以考虑尽可能提升修复召回率。具体而言，它们可以通过扩大搜索空间和放宽生成条件来为更多的缺陷生成补丁，使用 *InPaFer* 交互式过滤补丁可以保证在混合了正确补丁与错误补丁的情况下，仍然可以提升开发者的修

复效果。

- **度量自动修复技术的指标除了补丁正确性之外，还可以考虑补丁对于开发者的有用性**。本章的用户实验揭示了即使是不正确的补丁也可以帮助开发者修复缺陷，这一发现启示未来的研究工作可以考虑探究如何度量补丁的质量特性。未来度量自动修复技术的指标除了正确性之外，还可以增加补丁为开发者提供有用信息量这一维度。

第五章　总结和展望

5.1　本文工作总结

软件缺陷在软件开发过程中不可避免的会被引入，而缺陷会带来巨大的经济损失，甚至威胁人的生命安全，因此在软件开发过程中，及时发现并且修复程序中的缺陷十分重要。然而，缺陷修复是一件十分困难且耗时的工作。在此背景之下，程序缺陷自动修复技术被广泛研究，该方向经过十余年的发展，已经取得了很大进步。由于目前修复技术普遍面临过拟合问题，无法从根本上保证生成补丁的正确性，最终生成的补丁都需要开发者人工审阅。然而在给定正确性未知的候选补丁时，开发者的修复效率与正确率并不一定会被提升，甚至会下降。

为辅助开发者在提供候选补丁时高效准确修复缺陷，提升自动修复技术的实用价值，本文提出针对自动生成补丁的审阅提升技术，希望通过辅助开发者审阅补丁，提升其修复效率和正确率。具体而言，为探究如何辅助开发者审阅补丁，本文首先进行了一个补丁审阅实证研究。根据研究结果，本文提出一套**程序自动修复中的补丁审阅支撑技术**。通过过滤无效信息和引导开发者关注关键信息的方式来支撑补丁审阅过程，具体包括两个步骤：基于动静态信息结合的补丁过滤技术和交互式补丁过滤技术。本文的研究内容总结如下。

补丁审阅实证研究。为探究如何辅助开发者审阅补丁，本文对开源网站上开发者提交的补丁解释信息进行了人工审阅，并提出了一个补丁解释模型，包括：补丁位置信息、缺陷原因及补丁修改信息（三类静态信息）和缺陷出发条件及补丁应用前后的结果信息（两类动态信息），以上补丁特性对于补丁正确性判别及过滤错误补丁都具有潜在帮助。实证研究中还对补丁特性及其子类进行了定量分析。研究结果表明，开发者通常采用 2-3 个元素解释补丁。该发现说明，可同时结合不同的补丁特性辅助开发者审阅补丁。上述发现为本文的补丁审阅提升技术提供了必要指导。

基于动静态信息结合的补丁过滤技术。在人工审阅之前可为开发者过滤掉部分错误补丁，减轻开发者审阅负担。本文提出同时基于静态修改特征和动态覆盖特征相结合的补丁过滤技术 *Ceres* ，将补丁应用前后所在函数的代码表示为一个基于抽象语法树的修改特征图，采用 S-Transformer 模型对修改特征图进行结构编码，提取用于判别补丁正确性的关键修改特征；动态覆盖特征学习将所在函数中表达式的抽象语法树根节点和测试用例作为图中的节点，根据覆盖关系在两种节点间建立不同权重的边，构建覆盖特征图。采用 RAT 模型对补丁修改所导致的测试覆盖的变化进行学习，并建立与修改特征学习结果的关系，最终 *Ceres* 根据两个步骤的结果对补丁正确性进行判断。

本文在跨补丁和跨缺陷这两种数据集划分场景下验证 *Ceres* 的过滤效果。实验结果表明，*Ceres* 在两种划分方式下均超越了现有方法，缓解了补丁过拟合问题，减轻了人工审阅补丁的负担。

交互式补丁过滤技术。自动过滤技术只能过滤部分错误补丁，剩余补丁均需要开发者人工审阅。本文提出交互式补丁过滤技术 *InPaFer*，为辅助开发者审阅补丁，*InPaFer* 通过三种类型的补丁特性相关的询问与开发者进行交互，辅助开发者理解补丁并根据反馈过滤错误补丁。为及时响应开发者的反馈，*InPaFer* 构建了两阶段算法，离线收集程序属性构建询问，在线与开发者交互。为提升交互效率，减少用户回答询问的次数，*InPaFer* 定义了询问选择问题，并理论证明了询问选择问题与最优决策树的构建问题在多项式时间内可规约，从而引入最小化最大分支算法优化交互次数。最终，本文设计了一个与用户交互的图形界面，将 *InPaFer* 实现为一个开源 Eclipse 插件。为验证 *InPaFer* 的有用性，本文进行了定量实验和用户实验。用户实验表明，即使是在修复工具具有较低准确率的场景下，相对于另外三组基准调试场景下，使用 *InPaFer* 修复缺陷的开发者的修复效果（修复准确率和正确率）仍然可以显著提升。

综上，本文的实验结果表明：即使在修复工具具有较低准确率的场景下，通过辅助开发者审阅补丁的方式，也可以提升开发者的修复效率与正确率。该结果说明：*在合适的工具支持下，补丁审阅过程可以帮助开发者理解缺陷，最终有助于完成修复过程*。支持和改进补丁审阅过程是一个有前途的研究方向，未来可开展更多的研究工作。另外，该结果也为自动修复领域提供了新的可能性：通过采取适当的工具辅助补丁审阅过程，低准确率的修复技术也能在实践中发挥作用。基于本文提出的技术，未来的修复技术可以放宽对补丁准确率的限制，能够修复更多缺陷，从而为开发者提供帮助。

5.2 本文工作展望

本文方法的进一步提升。本文提出补丁审阅提升技术，辅助开发者判断补丁正确性，提升了修复效率和准确率。但是目前缺陷修复领域所面临的挑战依然存在，进一步提升补丁正确性判别能力。辅助补丁审阅仍有很大空间，距离自动修复技术实用化也有一定距离。

本文的补丁过滤方法可从以下几方面进一步扩展提升。

1. 自动化补丁过滤技术中，特征收集与编码方式可以进一步扩展。具体而言，本文的方法仅仅针对单个函数内部的覆盖信息变化提取特征，尚未考虑更大范围内的覆盖信息，例如文件或项目范围内。其次，本文收集的覆盖信息目前被编码为集合形式，即当前编码只能表示代码节点是否被覆盖，无法表示代码覆盖次数以及覆盖序列。未来研究工作可进一步收集覆盖信息，并且探究如何更全

面表示收集的信息，提升过滤错误补丁的能力。

2. 交互式补丁过滤技术中，可进一步扩展用于过滤错误补丁的询问类型或为开发者提供更多的补丁信息。具体地，询问更多类型的问题可以进一步提升区分错误补丁与正确补丁的能力，例如有关不变量信息的问题。其次，可为开发者提供额外的补丁信息，例如，利用已有补丁正确性判别技术提供每个生成补丁的正确性概率，方便开发者回答询问、提升交互效率。

新的机遇与挑战。本文在实验过程中也发现了很多未来可探索的研究方向。

1. **度量补丁有用性的准则**。本文的用户实验揭示了即使是不正确的补丁也可能为开发者提供一些有用信息，帮助开发者修复缺陷，例如错误补丁提示了正确的修复位置，或提供了部分正确的修改。这种情况下，相比于不提供任何帮助开发者独立修复，开发者的修复效率和正确率可以得到提升。因此未来度量自动修复技术的效果指标除了正确性之外，还可以增加补丁为开发者提供有用信息量这一维度。
2. **交互式程序修复技术**。本文提出在补丁生成之后，以询问开发者补丁特性相关问题的方式与开发者交互，过滤错误补丁。实际上，在补丁生成阶段引入开发者，也可以提升最终的修复效率和正确率。例如，开发者在补丁生成阶段提供补丁原料、或划定的搜索空间，这样补丁生成技术可减少对生成错误补丁的尝试次数，提升生成效率并且提升生成正确补丁的概率。引入开发者参与补丁生成过程，提升补丁生成效果，可以减轻补丁审阅负担。

参考文献

[1] J.-L. F. Jacques-Louis Lions Lennart Luebeck. “*Ariane 5 ight 501 failure report by the inquiry board*”. *European space agency Paris*, **1996**.

[2] R. Skeel. “*Roundo error and the Patriot missile*”. *SIAM News*, **1992**, *25*(4): 11.

[3] T. Britton, L. Jeng, G. Carver *et al.* “*Reversible debugging software - quantify the time and cost saved using reversible debuggers*”. In: **2013**.

[4] U. Software. “*Increasing software development productivity with reversible debugging*”. In: *Undo Software,Tech. Rep. white paper*. **2014**.

[5] *Symantec Internet Security Threat Report*. **2006**. `http://eval.symantec.com/%20mktginfo/enterprise/white_papers/entwhitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf`.

[6] J. Anvik, L. Hiew and G. C. Murphy. “*Who should fix this bug?*” In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, **2006**: 361–370.

[7] S. Forrest, T. Nguyen, W. Weimer *et al.* “*A genetic programming approach to automated software repair*”. In: *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*. ACM, **2009**: 947–954.

[8] W. Weimer, T. Nguyen, C. L. Goues *et al.* “*Automatically finding patches using genetic programming*”. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, **2009**: 364–374.

[9] C. L. Goues, T. Nguyen, S. Forrest *et al.* “*GenProg: A Generic Method for Automatic Software Repair*”. *IEEE Trans. Software Eng.* **2012**, *38*(1): 54–72.

[10] L. Gazzola, D. Micucci and L. Mariani. “*Automatic Software Repair: A Survey*”. *IEEE Trans. Software Eng.* **2019**, *45*(1): 34–67.

[11] M. Monperrus. *Automatic Software Repair: a Bibliography* [techreport]. Association for Computing Machinery, **2017**: 1–24.

[12] *kPAR*. **2021**. `https://github.com/SerVal-DTF/FL-VS-APR/tree/master/kPAR`.

[13] K. Liu, A. Koyuncu, D. Kim *et al.* “*AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations*”. In: *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. **2019**: 456–467.

[14] X.-B. D. Le, D. Lo and C. L. Goues. “*History Driven Program Repair*”. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, **2016**: 213–224.

[15] J. Xuan, M. Martinez, F. Demarco *et al.* “*Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs*”. *IEEE Trans. Software Eng.* **2017**, *43*(1): 34–55.

[16] X.-B. D. Le, D.-H. Chu, D. Lo *et al.* “*S3: syntax- and semantic-guided repair synthesis via programming by examples*”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, **2017**: 593–604.

[17] F. Long, P. Amidon and M. Rinard. “*Automatic inference of code transforms for patch generation*”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, **2017**: 727–739.

[18] M. Martinez, T. Durieux, R. Sommerard *et al.* “*Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset*”. *Empir. Softw. Eng.* **2017**, *22*(4): 1936–1964.

[19] K. Wang, A. Sullivan and S. Khurshid. “*Automated model repair for Alloy*”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, **2018**: 577–588.

[20] J. Hua, M. Zhang, K. Wang *et al.* “*SketchFix: a tool for automated program repair approach using lazy candidate generation*”. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, **2018**: 888–891.

[21] J. Hua, M. Zhang, K. Wang *et al.* “*Towards practical program repair with on-demand candidate generation*”. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 12–23.

[22] M. Endres, G. Sakkas, B. Cosman *et al.* “*InFix: Automatically Repairing Novice Program Inputs*”. In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, **2019**: 399–410.

[23] T. Nguyen, W. Weimer, D. Kapur *et al.* “*Connecting Program Synthesis and Reachability: Automatic Program Repair Using Test-Input Generation*”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. **2017**: 301–318.

[24] M. Motwani, S. Sankaranarayanan, R. Just *et al.* “*Do automated program repair techniques repair hard and important bugs?*” In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 25.

[25] A. Weiss, A. Guha and Y. Brun. “*Tortoise: interactive system configuration repair*”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, **2017**: 625–636.

[26] Y. Ke, K. T. Stolee, C. Le Goues *et al.* “*Repairing Programs with Semantic Code Search (T)*”. In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, **2015**: 295–306.

[27] S. Saha, R. K. Saha and M. R. Prasad. “*Harnessing evolution for multi-hunk program repair*”. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE, **2019**: 13–24.

[28] 李斌, 贺也平 and 马恒太. “程序自动修复: 关键问题及技术”. 软件学报, **2019**, *30*(02): 54–75.

[29] 孔祥龙. 自动化程序修复技术及影响分析 [phdthesis].

[30] 王赞, 郜健, 陈翔 *et al.* “自动程序修复方法研究述评”. 计算机学报, **2018**, *41*(003): 588–610.

[31] K. Liu, D. Kim, T. F. Bissyandé *et al.* “*Mining fix patterns for findbugs violations*”. *IEEE Transactions on Software Engineering*, **2018**.

[32] Y. Xiong, J. Wang, R. Yan *et al.* “*Precise condition synthesis for program repair*”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE, **2017**: 416–426.

[33] Y. Lou, A. Ghanbari, X. Li *et al.* “*Can automated program repair refine fault localization? a unified debugging approach*”. In: *ISSTA ’20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, **2020**: 75–87.

[34] J. Bader, A. Scott, M. Pradel *et al.* “*Getafix: Learning to Fix Bugs Automatically*”. *Proc. ACM Program. Lang.* 2019-10, *3*.

[35] A. Marginean, J. Bader, S. Chandra *et al.* “*SapFix: automated end-to-end repair at scale*”. In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE, **2019**: 269–278.

[36] F. Long and M. C. Rinard. “*An analysis of the search spaces for generate and validate patch generation systems*”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, **2016**: 702–713.

[37] X.-B. D. Le, F. Thung, D. Lo *et al.* “*Overfitting in semantics-based automated program repair*”. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 163.

[38] E. K. Smith, E. T. Barr, C. L. Goues *et al.* “*Is the cure worse than the disease? overfitting in automated program repair*”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, **2015**: 532–543.

[39] Z. Qi, F. Long, S. Achour *et al.* “*An analysis of patch plausibility and correctness for generate-and-validate patch generation systems*”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, **2015**: 24–36.

[40] Y. Tao, J. Kim, S. Kim *et al.* “*Automatically generated patches as debugging aids: a human study*”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, **2014**: 64–74.

[41] J. P. Cambronero, J. Shen, J. Cito *et al.* “*Characterizing Developer Use of Automatically Generated Patches*”. In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*. IEEE Computer Society, **2019**: 181–185.

[42] Y. Xiong, J. Wang, R. Yan *et al.* “*Precise condition synthesis for program repair*”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE, **2017**: 416–426.

[43] M. Wen, J. Chen, R. Wu *et al.* "*Context-aware patch generation for better automated program repair*". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 1–11.

[44] J. Jiang, Y. Xiong, H. Zhang *et al.* "*Shaping program repair space with existing patches and similar code*". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, **2018**: 298–309.

[45] Q. Xin and S. P. Reiss. "*Leveraging Syntax-related Code for Automated Program Repair*". In: Urbana-Champaign, IL, USA, **2017**. `http://dl.acm.org/citation.cfm?id=3155562.3155644`.

[46] R. K. Saha, Y. Lyu, H. Yoshida *et al.* "*ELIXIR: effective object oriented program repair*". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. **2017**: 648–659.

[47] F. Long, P. Amidon and M. Rinard. "*Automatic inference of code transforms for patch generation*". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, **2017**: 727–739.

[48] J. Bader, A. Scott, M. Pradel *et al.* "*Getafix: Learning to Fix Bugs Automatically*". *Proc. ACM Program. Lang.* 2019-10, *3*.

[49] S. Mechtaev, M.-D. Nguyen, Y. Noller *et al.* "*Semantic program repair using a reference implementation*". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 129–139.

[50] Y. Ke, K. T. Stolee, C. L. Goues *et al.* "*Repairing Programs with Semantic Code Search (T)*". In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2015**: 295–306.

[51] S. H. Tan and A. Roychoudhury. "*relifix: Automated Repair of Software Regressions*". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, **2015**: 471–482.

[52] R. Just, D. Jalali and M. D. Ernst. "*Defects4J: a database of existing faults to enable controlled testing studies for Java programs*". In: C. S. Pasareanu and D. Marinov, eds. *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. ACM, **2014**: 437–440.

[53] A. Koyuncu, K. Liu, T. F. Bissyandé *et al.* "*FixMiner: Mining relevant fix patterns for automated program repair*". *Empir. Softw. Eng.* **2020**, *25*(3): 1980–2024.

[54] K. Liu, A. Koyuncu, D. Kim *et al.* "*TBar: revisiting template-based automated program repair*". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, **2019**: 31–42.

[55] Q. Zhu, Z. Sun, Y.-a. Xiao *et al.* "*A syntax-guided edit decoder for neural program repair*". In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, **2021**: 341–353.

[56] W. Weimer, Z. P. Fry and S. Forrest. “*Leveraging program equivalence for adaptive program repair: Models and first results*”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, **2013**: 356–366.

[57] Y. Qi, X. Mao, Y. Lei *et al.* “*The strength of random search on automated program repair*”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. ACM, **2014**: 254–265.

[58] X.-B. D. Le, D. Lo and C. L. Goues. “*History Driven Program Repair*”. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, **2016**: 213–224.

[59] Q. Xin and S. P. Reiss. “*Leveraging syntax-related code for automated program repair*”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, **2017**: 660–670.

[60] J. Jiang, Y. Xiong, H. Zhang *et al.* “*Shaping program repair space with existing patches and similar code*”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, **2018**: 298–309.

[61] M. Wen, J. Chen, R. Wu *et al.* “*Context-aware patch generation for better automated program repair*”. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 1–11.

[62] R. Kou, Y. Higo and S. Kusumoto. “*A Capable Crossover Technique on Automatic Program Repair*”. In: *7th International Workshop on Empirical Software Engineering in Practice, IWESEP@SANER 2016, Osaka, Japan, March 13, 2016*. IEEE Computer Society, **2016**: 45–50.

[63] F. Long and M. Rinard. “*Staged program repair with condition synthesis*”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, **2015**: 166–178.

[64] X. Gao, S. Mechtaev and A. Roychoudhury. “*Crash-avoiding program repair*”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, **2019**: 8–18.

[65] T. Durieux, B. Cornu, L. Seinturier *et al.* “*Dynamic patch generation for null pointer exceptions using metaprogramming*”. In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. IEEE Computer Society, **2017**: 349–358.

[66] D. Kim, J. Nam, J. Song *et al.* “*Automatic patch generation learned from human-written patches*”. In: *35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, **2013**: 802–811.

[67] S. H. Tan, H. Yoshida, M. R. Prasad *et al.* “*Anti-patterns in search-based program repair*”. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, **2016**: 727–738.

[68] K. Liu, D. Kim, T. F. Bissyandé *et al.* “*Mining Fix Patterns for FindBugs Violations*”. *IEEE Trans. Software Eng.* **2021**, *47*(1): 165–188.

[69] K. Liu, A. Koyuncu, D. Kim *et al.* “*AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations*”. In: *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. **2019**: 456–467.

[70] A. Ghanbari, S. Benton and L. Zhang. “*Practical program repair via bytecode mutation*”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, **2019**: 19–30.

[71] K. Liu, S. Wang, A. Koyuncu *et al.* “*On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs*”. In: *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, **2020**: 615–627.

[72] H. D. T. Nguyen, D. Qi, A. Roychoudhury *et al.* “*SemFix: program repair via semantic analysis*”. In: *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, **2013**: 772–781.

[73] S. Mechtaev, J. Yi and A. Roychoudhury. “*DirectFix: Looking for Simple Program Repairs*”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, **2015**: 448–458.

[74] S. Mechtaev, J. Yi and A. Roychoudhury. “*Angelix: scalable multiline program patch synthesis via symbolic analysis*”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, **2016**: 691–701.

[75] J. Xuan, M. Martinez, F. Demarco *et al.* “*Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs*”. *IEEE Trans. Software Eng.* **2017**, *43*(1): 34–55.

[76] S. Jha, S. Gulwani, S. A. Seshia *et al.* “*Oracle-guided component-based program synthesis*”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, **2010**: 215–224.

[77] R. K. Saha, Y. Lyu, H. Yoshida *et al.* “*ELIXIR: effective object oriented program repair*”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, **2017**: 648–659.

[78] K. Noda, Y. Nemoto, K. Hotta *et al.* “*Experience Report: How Effective is Automated Program Repair for Industrial Software?*” In: K. Kontogiannis, F. Khomh, A. Chatzigeorgiou *et al.*, eds. *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, **2020**: 612–616.

[79] R. Gupta, S. Pal, A. Kanade *et al.* “*DeepFix: Fixing Common C Language Errors by Deep Learning*”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, **2017**: 1345–1351.

[80] 周风顺, 王林章 and 李宣东. “*C/C++ 程序缺陷自动修复与确认方法*”. 软件学报. **2019**, *30*(5): 1243–1255.

[81] M. White, M. Tufano, M. Martinez *et al.* "*Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities*". In: *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, **2019**: 479–490.

[82] M. Tufano, C. Watson, G. Bavota *et al.* "*An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation*". *ACM Trans. Softw. Eng. Methodol.* **2019**, *28*(4): 19:1–19:29.

[83] T. Lutellier, H. V. Pham, L. Pang *et al.* "*CoCoNuT: combining context-aware neural translation models using ensemble for program repair*". In: *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, **2020**: 101–114.

[84] Y. Li, S. Wang and T. N. Nguyen. "*DLFix: context-based code transformation learning for automated program repair*". In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, **2020**: 602–614.

[85] M. Asad, K. K. Ganguly and K. Sakib. "*Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair*". In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, **2019**: 328–332.

[86] Y. Xiong, X. Liu, M. Zeng *et al.* "*Identifying patch correctness in test-based program repair*". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 789–799.

[87] T. Durieux and M. Monperrus. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs* [Research Report]. **2016**. `https://hal.archives-ouvertes.fr/hal-01272126`.

[88] Q. Xin and S. P. Reiss. "*Identifying test-suite-overfitted patches through test case generation*". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. ACM, **2017**: 226–236.

[89] J. Yang, A. Zhikhartsev, Y. Liu *et al.* "*Better test cases for better automated program repair*". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, **2017**: 831–841.

[90] Z. Yu, M. Martinez, B. Danglot *et al.* "*Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system*". *Empir. Softw. Eng.* **2019**, *24*(1): 33–67.

[91] Y. Hu, U. Z. Ahmed, S. Mechtaev *et al.* "*Re-Factoring Based Program Repair Applied to Programming Assignments*". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, **2019**: 388–398.

[92] S. Mechtaev, M.-D. Nguyen, Y. Noller *et al.* "*Semantic program repair using a reference implementation*". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 129–139.

[93] *Valgrind(2016)*. **2016**. `http://valgrind.org/`.

[94] X.-B. D. Le, L. Bao, D. Lo *et al.* “*On reliability of patch correctness assessment*”. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE, **2019**: 524–535.

[95] F. Long and M. Rinard. “*Automatic patch generation by learning correct code*”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, **2016**: 298–312.

[96] S. Wang, M. Wen, B. Lin *et al.* “*Automated Patch Correctness Assessment: How Far are We?*” In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, **2020**: 968–980.

[97] G. Fraser and A. Arcuri. “*EvoSuite: automatic test suite generation for object-oriented software*”. In: *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, **2011**: 416–419.

[98] C. Pacheco and M. D. Ernst. “*Randoop: feedback-directed random testing for Java*”. In: *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, **2007**: 815–816.

[99] M. D. Ernst, J. Cockrell, W. G. Griswold *et al.* “*Dynamically Discovering Likely Program Invariants to Support Program Evolution*”. In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE’ 99, Los Angeles, CA, USA, May 16-22, 1999*. ACM, **1999**: 213–224.

[100] H. Ye, J. Gu, M. Martinez *et al.* “*Automated classification of overfitting patches with statically extracted code features*”. *IEEE Transactions on Software Engineering*, **2021**.

[101] V. Csuvik, D. Horváth, F. Horváth *et al.* “*Utilizing Source Code Embeddings to Identify Correct Patches*”. In: *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. **2020**: 18–25.

[102] Q. V. Le and T. Mikolov. “*Distributed Representations of Sentences and Documents*”. In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. JMLR.org, **2014**: 1188–1196.

[103] J. Devlin, M.-W. Chang, K. Lee *et al.* “*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, **2019**: 4171–4186.

[104] H. Tian, K. Liu, A. K. Kaboré *et al.* “*Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair*”. In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, **2020**: 981–992.

[105] U. Alon, M. Zilberstein, O. Levy *et al.* “*code2vec: learning distributed representations of code*”. *Proc. ACM Program. Lang.* **2019**, *3*(POPL): 40:1–40:29.

[106] T. Hoang, H. J. Kang, D. Lo *et al.* “*CC2Vec: distributed representations of code changes*”. In: *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, **2020**: 518–529.

[107] *RxJava*. **2021**. `https://github.com/ReactiveX/RxJava`.

[108] *Spring*. **2021**. `https://github.com/spring-projects/spring-boot`.

[109] *PocketHub*. **2021**. `https://github.com/pockethub/PocketHub`.

[110] *Nextcloud*. **2021**. `https://github.com/nextcloud/android`.

[111] *Intellij*. **2021**. `https://github.com/JetBrains/intellij-community`.

[112] *Lang*. **2021**. `https://github.com/apache/commons-lang`.

[113] G. Gousios, M. Pinzger and A. v. Deursen. “*An exploratory study of the pull-based software development model*”. In: *Proceedings of the 36th International Conference on Software Engineering*. **2014**: 345–355.

[114] S. H. Khandkar. “*Open coding*”. *University of Calgary*, **2009**, *23*: 2009.

[115] *Lift*. **2021**. `https://en.wikipedia.org/wiki/Lift_(data_mining)`.

[116] J. Zhou, H. Zhang and D. Lo. “*Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports*”. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. **2012**: 14–24.

[117] J. A. Jones, M. J. Harrold and J. T. Stasko. “*Visualization of test information to assist fault localization*”. In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. **2002**: 467–477.

[118] C. Catal and B. Diri. “*A systematic review of software fault prediction studies*”. *Expert Syst. Appl.* **2009**, *36*(4): 7346–7354.

[119] P. S. Kochhar, X. Xia, D. Lo *et al.* “*Practitioners’ expectations on automated fault localization*”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. **2016**: 165–176.

[120] Y. Huang, Q. Zheng, X. Chen *et al.* “*Mining Version Control System for Automatically Generating Commit Comment*”. In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*. **2017**: 414–423.

[121] *Pearson Correlation*. **2021**. `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`.

[122] H. Ye, M. Martinez and M. Monperrus. “*Automated patch assessment for program repair at scale*”. *Empir. Softw. Eng.* **2021**, *26*(2): 20.

[123] P. D. Marinescu and C. Cadar. “*KATCH: high-coverage testing of software patches*”. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, **2013**: 235–245.

[124] K. Liu, D. Kim, A. Koyuncu *et al.* “*A Closer Look at Real-World Patches*”. In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, **2018**: 275–286.

[125] Z. Chen, S. Kommrusch, M. Tufano *et al.* “*SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair*”. *IEEE Trans. Software Eng.* **2021**, *47*(9): 1943–1959.

[126] V. J. Hellendoorn, C. Sutton, R. Singh *et al.* “*Global Relational Models of Source Code*”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, **2020**.

[127] B. Wang, R. Shin, X. Liu *et al.* “*RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, **2020**: 7567–7578.

[128] Z. Qi, F. Long, S. Achour *et al.* “*An analysis of patch plausibility and correctness for generate-and-validate patch generation systems*”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, **2015**: 24–36.

[129] *Javalang*. **2021**. `https://github.com/c2nes/javalang/`.

[130] L. Dong and M. Lapata. “*Coarse-to-Fine Decoding for Neural Semantic Parsing*”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, **2018**: 731–742.

[131] A. Vaswani, N. Shazeer, N. Parmar *et al.* “*Attention is All you Need*”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. **2017**: 5998–6008.

[132] Z. Sun, Q. Zhu, Y. Xiong *et al.* “*TreeGen: A Tree-Based Transformer Architecture for Code Generation*”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, **2020**: 8984–8991.

[133] K. He, X. Zhang, S. Ren *et al.* “*Deep Residual Learning for Image Recognition*”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, **2016**: 770–778.

[134] L. J. Ba, J. R. Kiros and G. E. Hinton. “*Layer Normalization*”. *CoRR*, **2016**, *abs/1607.06450*.

[135] R. Rubinstein. “*The cross-entropy method for combinatorial and continuous optimization*”. *Methodology and computing in applied probability*, **1999**, *1*(2): 127–190.

[136] *ODS Experiment*. **2021**. `https://github.com/SophieHYe/ODSExperiment`.

[137] *PatchSim Experiment*. **2021**. `https://github.com/xinyuan-liu/Patch-Correctness`.

[138] *JDT*. **2021**. `http://www.eclipse.org/jdt`.

[139] *Coming*. **2021**. `https://github.com/SpoonLabs/coming`.

[140] X. Li, W. Li, Y. Zhang *et al.* "*DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization*". In: D. Zhang and A. Møller, eds. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, **2019**: 169–180.

[141] *Pytorch*. **2021**. `https://pytorch.org`.

[142] F. Demarco, J. Xuan, D. L. Berre *et al.* "*Automatic repair of buggy if conditions and missing preconditions with SMT*". In: *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, Hyderabad, India, May 31, 2014*. ACM, **2014**: 30–39.

[143] B. Johnson, Y. Brun and A. Meliou. "*Causal testing: understanding defects' root causes*". In: G. Rothermel and D.-H. Bae, eds. *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, **2020**: 87–99.

[144] M. Böhme, C. Geethal and V.-T. Pham. "*Human-In-The-Loop Automatic Program Repair*". In: *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, **2020**: 274–285.

[145] H. Laurent and R. L. Rivest. "*Constructing optimal binary decision trees is NP-complete*". *Information processing letters*, **1976**, *5*(1): 15–17.

[146] V. T. Chakaravarthy, V. Pandit, S. Roy *et al.* "*Decision trees for entity identification: Approximation algorithms and hardness results*". In: *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. **2007**: 53–62.

[147] A. Gupta, V. Nagarajan and R. Ravi. "*Approximation Algorithms for Optimal Decision Trees and Adaptive TSP Problems*". *Math. Oper. Res.* **2017**, *42*(3): 876–896.

[148] M. Adler and B. Heeringa. "*Approximating optimal binary decision trees*". In: *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*. Springer, **2008**: 1–9.

[149] V. T. Chakaravarthy, V. Pandit, S. Roy *et al.* "*Approximating decision trees with multiway branches*". In: *International Colloquium on Automata, Languages, and Programming*. **2009**: 210–221.

[150] W. Weimer, T. Nguyen, C. L. Goues *et al.* "*Automatically finding patches using genetic programming*". In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, **2009**: 364–374.

[151] C. L. Goues, T. Nguyen, S. Forrest *et al.* "*GenProg: A Generic Method for Automatic Software Repair*". *IEEE Trans. Software Eng.* **2012**, *38*(1): 54–72.

[152] K. Liu, A. Koyuncu, T. F. Bissyandé *et al.* "*You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems*". In: *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, **2019**: 102–113.

[153] D. Kim, J. Nam, J. Song *et al.* "*Automatic patch generation learned from human-written patches*". In: *35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, **2013**: 802–811.

[154] M. Martinez and M. Monperrus. "*ASTOR: a program repair library for Java (demo)*". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, **2016**: 441–444.

[155] M. Martinez and M. Monperrus. "*Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor*". In: *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*. Springer, **2018**: 65–86.

[156] Z. Chen and M. Monperrus. "*The Remarkable Role of Similarity in Redundancy-based Program Repair*". *CoRR*, **2018**, *abs/1811.05703*. `http://arxiv.org/abs/1811.05703`.

[157] Y. Lin, J. Sun, Y. Xue *et al.* "*Feedback-based debugging*". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE, **2017**: 393–403.

[158] X. Li, S. Zhu, M. d'Amorim *et al.* "*Enlightened debugging*". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, **2018**: 82–92.

[159] A. J. Ko and B. A. Myers. "*Debugging reinvented: asking and answering why and why not questions about program behavior*". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, **2008**: 301–310.

[160] S. Kaleeswaran, V. Tulsian, A. Kanade *et al.* "*MintHint: automated synthesis of repair hints*". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, **2014**: 266–276.

[161] J. Saldaña. *The coding manual for qualitative researchers*. Sage, **2015**.

[162] D. Zou, J. Liang, Y. Xiong *et al.* "*An Empirical Study of Fault Localization Families and Their Combinations*". *IEEE Trans. Software Eng.* **2021**, *47*(2): 332–347.

[163] P. Cashin, C. Martinez, W. Weimer *et al.* "*Understanding Automatically-Generated Patches Through Symbolic Invariant Differences*". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, **2019**: 411–414.

个人简历及博士期间研究成果

个人简历

梁晶晶，1994 年 9 月出生于河北省南宫市；2012 年 9 月考入华东师范大学软件工程学院，专业为软件工程，2016 年 7 月本科毕业并获得工学学士学位；2016 年 9 月保送进入北京大学信息科学技术学院计算机软件与理论专业攻读博士学位至今。

发表论文

[1] **Jingjing Liang**, Ruyi Ji,Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, Gang Huang. Interactive Patch Filtering as Debugging Aid. In *Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution*,12 pages. **[ICSME'2021 , CCF-B] IEEE TCSE Distinguished Paper Award.**

[2] Ruyi Ji, **Jingjing Liang**, Yingfei Xiong, Lu Zhang, Zhenjiang Hu. Question Selection for Interactive Program Synthesis. In *Proceedings of 41th ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 12 pages. **[PLDI'2020, CCF-A]**

[3] Daming Zou, **Jingjing Liang**, Yingfei Xiong, Michael Ernst, Lu Zhang. An Empirical Study of Fault Localization Families and Their Combinations. In *IEEE Transactions on Software Engineering*,12 pages. **[TSE'2019, CCF-A]**

[4] **Jingjing Liang**, Yaozong Hou, Shurui Zhou, Junjie Chen, Yingfei Xiong, Gang Huang. How to Explain a Patch: An Empirical Study of Patch Explanations in Open Source Projects. In *Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering*, 12 pages. **[ISSRE'2019, CCF-B]**

参与课题

1. 深度学习系统测试技术，国家重点研发计划政府间合作项目，No. 2019YFE0198100。
2. Testing Technologies for Deep Learning Systems, the Innovation and Technology Commission of HKSAR, No. MHP/055/19。
3. 软件维护，国家自然科学基金优秀青年基金，No. 61922003。
4. 基于情境的安全攸关软件的构造方法与运行机理研究，973 计划青年科学家专题项目，No. 2014CB347701。

致谢

时光飞逝，转眼间已经在燕园度过了五年半的时光。在这段时光里，我曾为遇到挫折而困惑迷茫过，也曾为达成阶段性目标而内心喜悦过。在逐渐实现目标的过程中，我也逐渐成长为一个内心更加坚定的人。回首这段弥足珍贵的时光，最值得感谢的还是那些陪伴着我的人。

首先，感谢谢杨芙清院士和梅宏院士。两位院士为中国软件工程事业的发展做出了卓越的贡献。何其幸运可以进入两位院士领导的北京大学软工所进行学习和科研。在如此优秀的学习环境与浓厚的科研氛围内，我可以接触到世界前沿的科研问题与内容，与世界优秀的科研学者进行沟通交流，最终完成博士研究生活。

感谢我的导师熊英飞副教授。熊老师是我的科研领路人。从最初接触科研到最终完成博士论文，熊老师都为我花费了很大的精力。从想法的提出到完成，从论文的撰写到演讲，熊老师都提供了悉心指导。从熊老师身上我学会了很多道理：在科研方向上，熊老师教导我们要做有影响力的研究；在科研心态上，熊老师鼓励我们每个人都需要突破自己的瓶颈。我深深地敬佩着熊老师一直以来对学术的执着与热情。熊老师严谨的科研态度、缜密的思维逻辑以及乐观积极的人生态度我都铭记于心，并将之视为工作与学习的榜样。

感谢天津大学陈俊洁副教授、姜佳君助理教授和多伦多大学的周抒睿助理教授。三位老师在我的科研陷入瓶颈时，都曾向我施以援手。不仅花费时间和精力为我提供可行的建议和指导，而且还真诚地鼓励我提升信心、直面挫折。他们不仅是优秀的学者，还是温暖的前辈。

感谢组里的指导老师胡振江教授和张昕助理教授。两位老师在科研课题上的深入见解，渊博的专业知识、为人处世的真诚与谦逊令我深深佩服。在我的科研课题上两位老师都给予过宝贵的意见，使我受益良多。

感谢软工所里的黄罡教授、谢冰教授、金芝教授、郝丹副教授、周明辉教授、焦文品教授、郭耀教授、邹艳珍副教授、陈泓婕副教授、赵海燕副教授。他们在我的博士培养期间提供了宝贵的指导意见，让我不断进步。

感谢朱琪豪同学和吉如一同学和我一起进行了程序修复的相关研究。朱琪豪参与了本文第三章内容的讨论，协助完成了部分实验。吉如一参与了本文第四章内容的讨论，协助完成了部分理论证明。

感谢我们小组的王杰、悦茹茹、王博、陈逸凡、章嘉晨、郜哲、曾沐焓、刘鑫远、张星、任路遥、张云帆、张石然、沈若冰、吴宜谦、叶振涛、杨晨阳、关智超、谢睿峰、

顾宇晨、鄢振宇、张煜皓、卢思睿同学。感谢其它小组的高庆、唐浩、张洁、娄一翎、邹达明、王冠成、周建 **YI**、冯致远、李丰、董谨豪、陈震鹏、李念语、傅英杰、陈天宇。感谢他们出现在我的博士生活中。我们一起解决过问题也一起度过过快乐的时光。

感谢我的父母对我毫无保留的爱与关心，永远是我坚实的后盾。正是因为你们的存在，我才会毫无畏惧的去迎接生活的挑战。感谢我的人生伴侣侯耀宗先生，陪伴我度过了人生中最宝贵的时光。感谢无论在我沮丧还是开心的时候，你始终都站在我的旁边。愿今后余生与你一起品味生活的真谛。

感谢一路走来遇到了那么多真诚与善良的人，他们带来的温暖让我感受到了生活的美好。在以后的工作与生活中，我必将继续努力传递这份温暖，成为社会中有责任感、有价值的一分子。

北京大学学位论文原创性声明和使用授权说明

原创性声明

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

论文作者签名：　　　　　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

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

（保密论文在解密后遵守此规定）

论文作者签名：　　　　　　导师签名：

日期：　　年　　月　　日