北京大学

博士研究生学位论文

题目：面向非频繁程序缺陷的自动修复技术研究

姓　　名：姜佳君

学　　号：1501111332

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

导师姓名：熊英飞　副教授

二〇二〇年六月

版权声明

摘要

程序缺陷在软件中普遍存在，对国家的经济和人们的生命安全构成了巨大的威胁。而在软件开发的过程中，程序缺陷是不可避免的。研究表明，软件开发者平均需要花费大概一半的开发时间用来调试和修复程序中的缺陷。同时，程序缺陷使得软件的开发和维护成本增加。因此，研究缺陷的自动修复方法来辅助程序开发者的调试过程具有重要的科学意义和实用价值。

缺陷修复技术的主要挑战是非频繁缺陷的修复。目前缺陷修复的主要方法是依赖大量的重复修改历史学习补丁模板指导修复过程，该方法的基本原理决定了其只能修复频繁缺陷，不能修复非频繁缺陷。然而，实践中的绝大多数缺陷都是非频繁缺陷。研究发现，超过 80% 的真实缺陷是非频繁缺陷，只有 15%~20% 的缺陷会重复发生，而且其中超过 50% 的缺陷重复次数仅为个位数。如果不能解决非频繁缺陷的问题，则缺陷修复技术始终只能处理实际中缺陷的较小部分，难以在实践中大规模应用。但是，解决非频繁缺陷并不容易。主要难点是：(1) 非频繁缺陷的重复修改历史数据规模小，大部分缺陷仅出现一到两次。因此，已有技术无法从中学习修复模板。(2) 非频繁缺陷种类多，不同缺陷的修复方式差异大。目前针对非频繁缺陷的主要方案是依赖人工定义修复模板或者暴力搜索的方式，但都无法实际解决该问题。

本文围绕非频繁缺陷的修复问题，针对非频繁缺陷重复修改少 (或不存在)、种类多导致补丁生成困难的挑战，提出了**利用代码的频繁性克服修改的非频繁性，并依据此提出了一个面向非频繁缺陷的自动修复方法 IBFix**。IBFix 包含三个重要组成部分: (1) 基于单个历史修改样例的补丁模板提取技术，可以利用历史修复对补丁生成进行指导，同时克服了对大量重复修改的依赖；(2) 基于相似代码的补丁生成技术，克服了不存在重复历史修改时的补丁生成难题；(3) 基于程序状态划分的缺陷定位技术，提升了定位的准确率以满足非频繁缺陷修复对定位准确率的更高要求。最后的实验表明，本文所提出的缺陷修复技术 IBFix 有效地克服了非频繁缺陷的修复难题，推动了自动修复技术的实用化进程。综上，本文主要的工作和创新如下：

1. **通过人工实证研究详细地分析和总结了开发者修复程序缺陷的过程，为本文提出的缺陷修复技术提供了必要指导**。本文通过记录开发者修复真实程序缺陷，对其修复过程进行了详细地分析和总结。根据实证研究结果，本文总结了以下重要发现：开发者综合使用不同的数据信息来定位程序缺陷，并且根据有限的历史修复经验和项目中相似代码指导其编写修复补丁。上述发现为本文的自动修复技术提供了必要指导。

2. **提出了基于程序状态划分的缺陷定位技术 (PredFL)**。本文形式化地定义了统一

的缺陷定位模型，实现了基于程序频谱与基于统计性调试定位技术的深度结合。基于该模型，PredFL 将程序元素覆盖转换为状态谓词覆盖，使两种定位技术所依赖的互补数据信息实现共享。结合之后的方法具有更加丰富的状态谓词，对程序状态实现了更细粒度划分，具有更强的识错能力，从而有效地提升了缺陷定位的准确率。

3. **提出了基于单个修改样例的补丁模板提取技术 (GenPat)**。本技术通过从海量的开源代码中分析代码属性分布，指导补丁模板的抽象过程，从而克服了模板提取对大量重复修改的依赖。GenPat 采用代码超图模型表示补丁模板，图中丰富的边和节点属性可以更好地反映代码结构以及语义特征，增强了其表达能力和灵活性。基于该模型，补丁模板的抽象过程被转化成代码超图中的节点以及边的选择过程，实现了代码结构和代码属性特征解耦，从而可以利用海量开源代码指导不同属性的抽象过程，而不依赖重复修改。因此，本方法可以有效修复非频繁缺陷，具有更强的通用性。

4. **提出了基于相似代码的补丁生成技术 (SimFix)**。本技术通过对比缺陷代码与相似代码的差异实现对缺陷代码的智能修改，从而完全克服了缺陷修复对补丁模板的依赖。SimFix 将搜索到的相似代码作为缺陷代码的参考，通过对比两段代码的语法树差异，提取有针对性的修改操作。此外，为了进一步提升补丁的质量，SimFix 使用开源项目中频繁用于修复缺陷的修改操作对上述得到的修改进行过滤。通过组合不同的候选修改操作，可以生成丰富的修复补丁。因此，即使针对不存在重复历史修复的非频繁缺陷，该方法依然可以提供有效的修复指导。

5. 在基准数据集 Defects4J 上的系统性对比实验证明了本文提出的自动修复方法 IBFix 及其各组件的有效性。相比之前的缺陷自动修复技术，IBFix 可以有效修复非频繁缺陷，大幅度提升了正确修复缺陷的数量 (提升 ≥65.4%)。其正确修复的缺陷中，35.6% 的缺陷不能被其他任意技术修复。下列是 IBFix 中不同组件的实验结果：

- PredFL 不仅可以提升单一定位技术的准确率 (提升 32.6%-227.9%)，而且与目前已知的定位技术具有互补性，与已有技术集成可以进一步提升 20.8% 的准确率。
- GenPat 在缺陷修复中，为 19 个缺陷提供了正确补丁模板。在系统性代码修改中，相比已知的最好方法 Sydit，其实现了 4.5 倍的效果提升。具有较强通用性。
- SimFix 可以正确修复 34 个缺陷。相比已有方法，修复数量提升了至少 30.8%。2020 年的最新研究 (本工具发布于 2018 年) 通过实验对比 16 种最新缺陷修复工具表明，SimFix 的正确修复数量仍然最多，且独立修复 (其他工具均不能修复) 的数量也是最多。

关键词：软件维护，缺陷修复，缺陷定位，非频繁缺陷

Research on Automatic Program Repair Technology for Infrequent Defects

Jiajun Jiang (Computer Software and Theory)
Directed by Prof. Yingfei Xiong

ABSTRACT

Program defects are prevalent among programs, which may cause big threats to both national economics and human lives. However, they are inevitable in software development. According to the study, developers spend in average half of their developing time on debugging and repairing those defects. In addition, program defects can also increase the cost of software development and maintenance. Therefore, it is an important research to study automatic program repair techniques to assist software developers in the debugging process.

The key challenge of automatic program repair techniques is how to repair infrequent defects. Existing techniques depend on a large number of repetitive history repairs to learn repair patterns, which, in theory, can repair only those frequent defects, but not infrequent ones. As a matter of fact, most of defects are infrequent in practice. Based on prior study, more than 80% defects in real world are infrequent, and only 15%~20% defects repetitively occurred in the history, in which more than 50% repeated only several times. Therefore, if the infrequent defects could not be fixed, automatic program repair techniques could only repair a small number of defects, and would not be applied to large-scale applications in practice. However, it is not easy to repair infrequent defects. The major challenges are twofold: (1) They have limited repetitive history repairs, and most of them occurred once or twice, making it impossible for existing techniques to learn repair patterns. (2) They tend to be diverse and require different repair patterns. Existing techniques leverage human-defined patterns to repair infrequent defects, or simply utilize brute-force search without sufficient guidance, which would not solve the challenges.

In this thesis, facing the challenge of repairing infrequent defects, **we propose an automatic program repair approach for infrequent defects, i.e., IBFix, which takes advantage of the frequency of source code to overcome the infrequency of repetitive repairs.** IBFix consists of three major components: (1) a repair pattern extraction technique based on singu-

lar examples, which benefits from historical repairs and overcomes the dependency on large number of repetitive repairs; (2) a patch generation technique based on similar code, which overcomes the patch generation for infrequent defects when no repetitive repair exists; (3) a fault localization technique based on the division of program states, which significantly improves the fault localization accuracy to meet the high demand of repairing infrequent defects. Finally, the evaluation shows that IBFix can effectively repair infrequent defects, and facilitate the practicality of automatic program repair techniques. In summary, the contributions of this work are presented as follows.

1. **An empirical study to investigate the program repair process of developers, providing insightful guidance for the new repair approach**. Via analyzing the repair process of human developers, we summarize a set of useful strategies, which will benefit the research on automatic program repair techniques. According to the result, we have the following findings: the developer usually combine different data source to locate defects. In addition, he can repair unfamiliar defects with the help of previous experience or similar code in the project. Those findings inspire the following automatic approaches.

2. **A new fault localization technique based on the division of program states (PredFL)**. We have given the formal definition of a unified model, which combines the strength of spectrum-based fault localization (SBFL) and statistical debugging (SD). Under this model, PredFL maps the coverage of program elements into the coverage of a kind of predicates in SD, making it possible to share complementary information used by the two different approaches. Therefore, the combined technique has more plentiful predicates to effectively identify incorrect program states, and thus improves the precision of fault localization.

3. **A technique to extract repair patterns from singular historical bug fixes (GenPat)**. GenPat leverages distributions of code attributes analyzed from huge open-source code to guide the abstraction of repair patterns, which does not depend on repetitive repairs for pattern extraction. It employs the hypergraph of code to represent repair patterns. Edges and attributes of nodes in the hypergraph represent the structure and semantics of source code, which is expressive and flexible. Under this model, the repair pattern abstraction process is converted to the selection of edges and nodes. Therefore, code structures and attributes are decoupled and can be tackled separately, making it possible to analyze node attributes from open-source code to guide pattern abstraction without repetitive repairs. As a consequence, GenPat is not only effective for repairing infrequent defects, but general for other potential applications.

4. **A patch generation technique based on similar code (SimFix)**. SimFix takes the similar code as a reference and repairs the defective code based on the differences from similar

code, and thus it overcomes the dependence on repair patterns. SimFix searches the entire program for similar code and extracts fine-grained code changes on abstract syntax tree via diffing similar and defective code, which are more targeted. Besides, to further improve the quality of generated patches, SimFix leverages frequent repair changes collected from open-source programs to filter all candidates. Then, patches can be generated by combining different changes. Therefore, even though for infrequent defects without repetitive repairs, SimFix can still repair them effectively.

5. A systematic evaluation of IBFix on a commonly used benchmark, Defects4J, to demonstrate its effectiveness. Compared with prior techniques, IBFix can effectively repair infrequent defects, and significantly outperforms the others with correctly repairing at least 65.4% more defects. In addition, 35.6% defects repaired by IBFix cannot be repaired by any other approaches. We list the experimental results for different components of IBFix as follows.

- PredFL not only can improve the fault localization accuracy of SBFL and SD (32.6%-227.9%), but also complements existing techniques, where it can further improve 20.8% on accuracy after integrating with existing techniques.
- GenPat can extract reusable code change patterns from singular examples, which can be applicable to different applications. In program repair, it can correctly repair 19 real-world defects. In systematic editing, it significantly outperforms state-of-the-art Sydit with 5.5x correctly changed code.
- SimFix can correctly fix 35 defects based on similar code, which is at least 30.8% more than existing techniques. Additionally, the most recent study in 2020 (SimFix was published in 2018) empirically compares the most latest 16 automatic program repair tools, and reports that SimFix repairs the most number of defects. In addition, the unique repairs of SimFix (cannot repaired by others) is also the most.

KEYWORDS: Software maintenance, Bug repair, Fault localization, Infrequent defects

目录

表格

插图

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 (Software) 在现代的计算机科学中发挥着越来越重要的作用，现代信息系统的发展离不开软件的发展。软件开发在国家的发展战略中占据重要地位，是国家的“十三五”信息化规划中的一个重要方面。随着软件规模的提升，软件的后期维护成本不断增加。研究表明，软件维护的成本占软件开发成本的 90% 以上[89]，其中修复程序中的缺陷是软件维护中一项重要内容。因此，软件缺陷 (Software Defects)①成为影响软件质量的重要因素。软件缺陷会影响软件的正常功能甚至导致程序崩溃而无法继续运行，造成重大的经济损失甚至对生命造成巨大威胁。2013 年，剑桥大学的一项研究②表明，全球每年大概要花费 3120 亿美元在软件缺陷调试上。此外，软件缺陷造成了 2018 和 2019 年两架波音 737 Max 8 型客机坠毁事故③，造成了上百人的死亡。软件缺陷在软件开发的过程中是不可避免被引入的，虽然一些缺陷极少出现，但是其发生时所带来的损失是巨大的。因此，及时修复软件中的缺陷十分重要。

然而，修复软件中的缺陷是一件十分困难而耗时的工作。有研究表明，对于开发人员来说，平均需要 28 天来修复一个安全缺陷[94]，并且程序缺陷还在不断地快速增加[3]。此外，对 Linux 开发者的一项调研表明，在程序的开发过程中，大约一半时间是用在了缺陷修复上[10]。不仅如此，在现代企业级的软件开发过程中，一方面软件的规模增加，另一方面开发人员的流动性很大，导致大部分程序开发人员不能很好的理解完整软件系统代码的功能逻辑。因此，程序缺陷的修复难度增加，甚至会引入新的软件缺陷[122]，或者难以修复而在软件系统中存在非常长的时间④。所以，研究和开发高效的缺陷修复方法对于降低软件维护成本和开发人员的负担具有重要意义。

1.1.2　自动化软件缺陷修复

为了降低开发人员的负担以及软件维护的成本，从 2009 年开始，自动化缺陷修复的研究成为了一个热门的课题。经过了十多年的研究，很多优秀的缺陷修复方法被提出[28,71,134,135]，甚至一些方法已经被应用到工业化的生产环境中辅助开发人员修复缺陷。比如国际社交媒体公司 Facebook 开发的软件自动修复工具 GetaFix[7]，它可以有效

①本文不区分 Defect、Bug 和 Fault 的不同，均用来指程序中的缺陷，可互换使用。
②https://www.prweb.com/releases/2013/1/prweb10298185.htm
③https://en.wikipedia.org/wiki/Boeing_737_MAX_groundings
④https://www.cvedetails.com/cve/CVE-2014-6352/

地修复程序中潜在的空指针异常缺陷等。因此，自动化缺陷修复技术可以极大的辅助开发者的繁重缺陷修复过程。

目前主流的自动化缺陷修复方法是基于测试的“生成-验证”修复模型 (如图1.1所示)，该模型依赖测试定位程序中的缺陷和验证生成补丁的正确性：即首先根据测试运行的结果定位程序中可能的出错位置；然后在出错位置尝试应用代码修改操作以及构造修复代码片段生成修复补丁；最后验证补丁能否正确通过所有测试。实际上，自动化的缺陷修复过程本质上是一个修复补丁的搜索过程，其搜索空间由三个子空间共同确定。首先是由缺陷定位确定的**位置空间**，其次是在补丁生成过程中定义的**修改空间**以及代码**元素空间**。缺陷修复的搜索空间是以上三者的笛卡尔积。

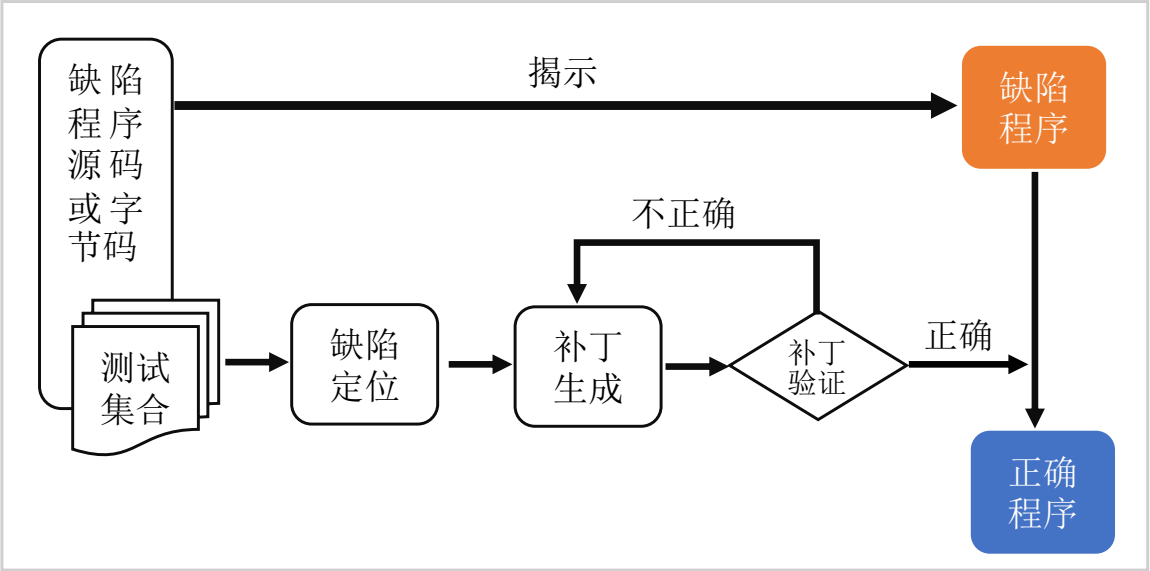


图 1.1　自动化缺陷修复过程

缺陷定位作为修复的首要过程，其结果的准确度将直接影响缺陷修复的效果。根据测试运行的结果，定位方法对程序中所有位置的代码进行排序，目标是尽可能将出错代码的位置排在前面。我们将候选列表中出错代码位置之前的所有位置定义为缺陷修复的**位置空间**，因此定位的准确率决定了位置空间的大小。Liu 等人[59] 研究发现，提升缺陷定位的准确率 (缩小位置空间) 可以有效提升缺陷修复能力。为此，很多不同的定位方法被提出，比如基于程序频谱的定位方法，基于状态的定位方法等。缺陷定位的粒度根据需要可以是出错的文件、方法或者代码语句等。在自动化缺陷修复过程中，一般定位到方法或者代码语句级别。

修复补丁生成是缺陷定位的后继过程，也是缺陷修复的**核心**。修复补丁的生成能力直接决定修复工具的修复能力和适用范围。比如一些修复方法仅修复空指针异常[18] 或者内存泄漏[26] 缺陷等，这些方法不能修复其他类型的缺陷。对于补丁生成过程，首先

是决定对错误代码所允许的修改操作集合，比如插入、删除等，即所谓的**修改空间**。然后，在给定的代码**元素空间**中搜索代码元素作为原材料用于生成修复补丁。比如 GenProg[51] 的修改操作包括语句插入、语句删除以及语句替换操作，而生成的补丁代码严格要求来源于项目中现存代码。因此，补丁的生成过程即在给定的空间中的搜索过程。所以，补丁搜索空间 (修改空间和元素空间) 的定义在补丁生成过程中至关重要。如果搜索空间太小，会使得很多正确的修复补丁被排除在空间之外导致对应缺陷在理论上就不可能被正确修复；相反的，如果搜索空间定义得太大，会导致正确的补丁在整个空间中所占比例降低，进而增加搜索 (修复) 的难度。为此，很多的方法被提出用来优化补丁空间，比如人工定义修改模板、从大量重复历史修改中学习等。

1.2 尚待研究的问题

尽管经过了十多年的研究，已经有众多的缺陷自动修复技术被提出，并取得了一定的效果。但是，现有的自动修复技术严重依赖大量重复的历史修改数据，使得其主要针对比较常见的简单类型的缺陷，比如空指针缺陷等。因此，对于非频繁缺陷目前尚缺乏有效的修复方法。顾名思义，非频繁的缺陷即指在软件的开发和维护过程中重复发生频率比较低的缺陷类型。非频繁的缺陷在真实地软件缺陷中占有非常大的比例。研究表明[124]，真实程序缺陷中超过 80% 为非频繁缺陷，只有 15%~20% 的缺陷会重复出现。而重复出现的缺陷中超过 50% 重复次数仅为个位数。此外，据统计分析，在缺陷修复领域广泛使用的真实缺陷数据集 Defects4J[44] 中，超过 60% 的缺陷属于非频繁缺陷。因此，不解决非频繁缺陷的修复问题，自动化缺陷修复技术只能修复比较小的部分缺陷，难以大规模实用化。

然而，修复非频繁缺陷具有非常大的挑战。在上文的1.1节中已经介绍，缺陷修复的补丁搜索空间太大或者太小都会严重影响修复技术的修复效果：太小会导致很多缺陷理论上不能修复；而太大会导致大量不正确的修复以及耗费更多的时间和资源。因此，缺陷修复的补丁生成需要定义合适的补丁搜索空间。目前的修复方法中效果比较好的技术主要是基于人工定义和基于统计的修复方法，但是它们主要针对频繁的缺陷。其主要原因是：基于人工定义的修复方法需要依赖开发者人工定义修复模板。对于修复方式复杂多变的非频繁缺陷来说，人工定义所有的修复模板不具可操作性。因此，尽管该方法的修复效果比较好，但是并不适用于修复非频繁的缺陷。另一方面，基于统计的修复方法通常依赖大量重复的历史修改，通过统计频繁出现的缺陷修复或者训练机器学习模型从历史数据中自动化地提取修复的补丁模板。该方法相对于人工定义的方式，不需要人工编写模板，理论上可以针对所有类型缺陷生成修复模板。然而事实上，由于非频繁缺陷缺少足够的重复历史修复，基于统计的方法并不能从少量的 (几个

甚至只有一个) 修改样例中学习通用的补丁模板，造成该类方法也主要修复比较常见的缺陷，在理论上并不适用修复非频繁的缺陷。比如 Genesis[61] 和 GetaFix[7] 只修复空指针异常等常见缺陷。此外，对于已有的其他修复方法，基于启发式搜索和基于约束求解的方法，因为缺少上述两种方法中的模板对补丁空间进行较好的约束以及面临计算延展性问题，因此，在实际中只能修复少数常见的简单缺陷，在真实的生产环境中难以大规模使用。

根据已有的研究[124]，在真实开发中非频繁缺陷的重复频率是非常低的，部分缺陷在历史修复中的重复次数仅为个位数，甚至很多缺陷并不存在重复的修复历史。因此，针对非频繁缺陷的该特征研究和开发有效的自动化修复技术是必要的。

1.3 本文主要工作和创新点

本文围绕非频繁缺陷的修复问题，针对上文中所面临的历史重复修复数据量少 (或不存在) 的挑战，提出**利用代码的频繁性克服非频繁缺陷的修复难题**，并提出了一个面向非频繁缺陷的自动修复方法。众所周知，最近几年随着开源软件的增加，海量的代码可以被公开的访问和使用。因此，虽然非频繁缺陷的修复历史数据有限，本文通过从大量的可访问代码数据中分析代码属性来指导对非频繁缺陷的修复过程。

为了更好地理解非频繁缺陷修复的过程，本文首先进行了一个人工修复缺陷的实证研究。通过分析和总结人工修复的过程，为非频繁缺陷的自动修复方法提供指导。根据人工修复中的启发以及利用上述的代码频繁特征，本文提出了一个面向非频繁缺陷的修复方法 IBFix，它主要包含三部分：基于单个修改样例的补丁模板自动提取技术、基于相似代码的补丁生成技术以及基于程序状态划分的缺陷定位技术。具体来讲，IBFix 针对非频繁缺陷修复所面临的两种情况分别使用不同的补丁生成技术 (如图1.2)。首先，针对缺陷的修复历史存在但是比较少的情况，IBFix 使用**基于单个修改样例的补丁模板自动提取技术**，通过从海量的开源代码数据中学习代码属性分布情况，指导修复模板的提取过程。该方法仅根据一个历史修复补丁提取可复用的补丁模板，在应用了历史修复知识的同时克服了已有方法在学习补丁模板时对大量重复修复的依赖。最后通过与缺陷代码匹配并应用模板生成修复补丁。其次，对于历史中不存在重复的缺陷而言，IBFix 使用**基于相似代码的补丁生成技术**。该方法通过从本项目中自动搜索与出错代码具有相似功能的代码指导缺陷代码的修复。由于同一个项目通常由一个开发者或者一个开发团队所编写，相似的代码所实现的功能具有相似性。通过对比缺陷代码与相似代码的差异，提取代码修改操作。同时，通过复用相似代码中的代码元素，为修复补丁提供原材料并约束代码的修改空间。此外，面对非频繁缺陷的自动修复对缺陷定位提出的更高准确率要求，IBFix 使用一种基于程序状态划分的缺陷定位技术，通

过白盒的方式结合基于程序频谱和基于程序状态的两种定位方法，可以有效提升缺陷定位的准确率，辅助提升非频繁缺陷的修复数量。所以，在修复非频繁缺陷时，IBFix使用两种不同的补丁生成方法分别尝试生成修复补丁并对其进行排序以提升补丁的准确率。

综上，本文所提出的缺陷自动修复技术通过应用海量的开源代码和大量的相似代码有效地克服了非频繁缺陷中的重复修改样本少的挑战。本文主要研究内容以及创新点如下：

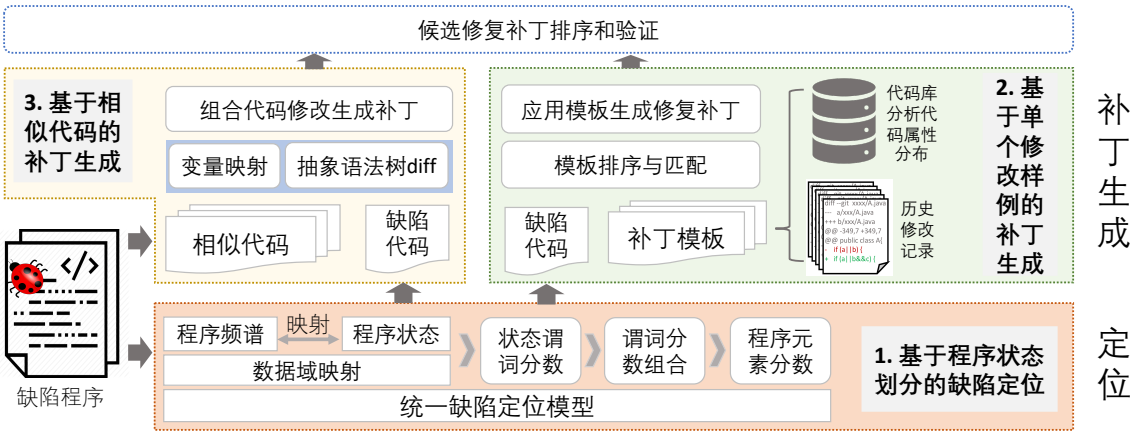


图 1.2　本文提出的自动修复技术概览

1. 为了了解非频繁缺陷的修复过程，给自动修复方法提供指导，本文进行了探究程序开发者人工修复缺陷的实证研究。通过对人工修复过程进行分析，本文总结了人工修复中的重要发现，为本文的自动修复方法提供了必要指导。
2. 提出了一个新的缺陷定位技术，形式化地定义了统一的缺陷定位模型用来结合程序的频谱和状态信息，提升了缺陷定位的准确率以满足非频繁缺陷修复对定位的更高要求。
3. 提出了一个基于单个代码修改样例提取可复用补丁模板的技术，克服了非频繁缺陷重复历史修改样本数量少的问题，为非频繁缺陷的补丁生成提供模板。
4. 提出了一个基于项目中相似代码的修复补丁生成技术，克服了非频繁缺陷不存在重复历史修改数据时的修复难题，依赖相似代码指导补丁生成。
5. 一个系统化的对比实验分析，验证了本文提出的缺陷自动修复方法 IBFix(如图1.2) 及其所包含的各组件的有效性。

1.3.1　缺陷修复实证研究

尽管缺陷修复技术经过了十多年的快速发展，最新的缺陷修复方法的修复能力依然较弱，主要针对一些比较常见的缺陷类型，比如空指针异常等。然而，对非频繁缺

陷，现有方法还缺乏足够的修复能力，只有极为少数的缺陷可以被正确修复。在现有“生成-验证”模型指导下，为了更好地了解人工修复缺陷的过程为修复非频繁缺陷提供指导，本文进行了一个实证研究来探索程序开发者如何人工修复真实的缺陷。其具体过程是模拟自动化缺陷修复方法，为程序开发者提供相同的缺陷修复环境，由开发者者修复缺陷。在该研究中我们主要关注以下三个问题：

1. 在相同的修复环境下，有多少缺陷可以被人工修复？
2. 人工修复过程中如何定位程序中的缺陷？
3. 人工修复的补丁是如何生成的，特别是如何生成非频繁缺陷的补丁？

通过第一个问题我们可以在一定程度上了解目前的自动化缺陷修复方法相比于开发人员的修复能力：即除了常见的程序缺陷，依赖测试提供的程序规约能否指导非频繁缺陷的正确修复。虽然在某些情况下，自动化的缺陷修复工具具有更强的能力，比如强大的计算力等。但一般情况下，在相同的条件下程序开发者难以修复的缺陷，我们可以近似认为自动化的修复方法也难以克服。而最后两个问题的目标是通过分析人工修复的过程，总结出一些开发者在修复程序缺陷中所使用的具体定位以及补丁生成方法，特别的，本文重点关注非频繁缺陷的修复方法。

该项研究中的**一个关键挑战**是：如何记录和描述人工修复的定位过程及补丁生成过程。众所周知，对于自动化的缺陷修复过程而言存在一套具体可描述的算法逻辑。但是对于人工修复而言，大部分的“算法”都是在开发者的大脑内进行的。同时，对于不同的缺陷甚至同一个缺陷，开发者也可能使用不同的方法实现正确修复。因此，如果要求开发者针对每个缺陷都编写一个具体的算法在实际中并不可行。对此本文提出了一种基于**策略**描述的缺陷定位和补丁生成方法，每条策略会有对应的过程以及使用到的信息描述。不同的策略可以进行组合使用以共同修复一个缺陷。通过这种方式一方面为开发人员记录修复过程提供了具体可行的方法，另一方面可以方便不同策略的分析和对比，同时为之后的具体算法实现提供易于理解的指导。

实证研究表明，大概 82% 的缺陷可以正确地被开发者修复，其中包括非频繁的代码缺陷。而现有的自动化缺陷修复方法只能正确修复不到 10% 的缺陷，且主要是比较常见的缺陷类型，如空指针错误等。实验结果表明目前的缺陷修复方法依然存在很大的提升空间，并有希望克服非频繁缺陷的修复。通过分析程序员的修复过程，本文分别总结了七个用于缺陷定位和补丁生成的策略，可以为自动化缺陷修复技术提供必要的指导。更重要的，通过分析开发者修复的过程，本文总结出了对于自动化修复非频繁缺陷的有效方法。首先，开发者通常可以根据少数的示例代码提取可复用的代码修改操作，该过程开发者主要依赖其已有的开发经验而并不过度依赖其历史修复经验。其次，即使开发者在完全不理解程序功能的情况下，根据程序中的相似代码可以指导缺陷修复的补丁生成。最后，开发者在定位和修复的过程中会综合使用不同的数据信息

提升修复的准确率。上述发现为本文提出的面向非频繁缺陷的自动修复技术提供了重要的指导。该研究工作已发表在中国科学 (Science China Information Sciences) 2019 上。

1.3.2　基于状态划分的缺陷定位技术

根据已有的研究表明[59]，提升定位准确率可以有效提升缺陷修复的能力。对于非频繁缺陷而言，定位准确率显得更为重要。主要原因是非频繁缺陷由于补丁生成更加困难，当定位准确率比较低时会将该困难放大。因此，为了有效修复非频繁的缺陷，提升定位的准确率也是其中一项重要研究内容。

根据实证研究我们发现，在人工的修复过程中开发人员通常会结合程序的不同特征信息用来辅助缺陷定位，比如程序运行过程中的变量取值信息以及程序执行的路径信息等。基于此，本文提出了一个基于程序状态划分的缺陷定位方法。该方法结合了基于程序频谱和基于程序状态的两种缺陷定位方法，通过将两种方法所依赖的特征信息变换到一个特征域 (程序状态域) 进行结合后实现对程序状态的细粒度划分，从而提升缺陷定位的准确率。

该项研究存在的**关键挑战**是：如何实现特征空间的变换。根据相关研究介绍，基于程序状态的定位方法依赖程序状态信息，而基于程序频谱的定位方法依赖程序执行时的元素 (比如语句) 覆盖信息。为了使两者在程序状态域实现信息共享和结合，我们需要首先将程序频谱特征映射为程序的状态特征，然后与前者进行结合。此外，由于根据程序状态定位方法得到的结果是不同程序状态与缺陷的关系大小，而缺陷修复过程需要定位到出错的代码元素。因此，如何将定位得到的程序状态反映射回代码元素空间也需要考虑。

面对上述挑战，本文形式化地定义了一个统一的缺陷定位模型，通过定义特征转换函数可以实现程序频谱和状态之间的相互转换。其具体过程是，假设每个程序元素位置存在一个常量谓词[①]“TRUE”，则程序代码的覆盖可以一对一地映射到对该常量谓词的覆盖，实现与基于状态定位方法中所定义谓词的结合。状态空间中丰富的谓词种类，对程序状态实现更细粒度检查，提升对程序缺陷的甄别能力。最后，根据不同谓词被定义的具体代码位置，将谓词与缺陷的关联关系映射为代码元素与缺陷的关系实现缺陷代码的定位。

综上，本文形式化的定义了一个统一的基于程序状态的定位模型，可以为系统化地研究如何结合程序频谱和程序状态提供指导。更具体说，本文提出的缺陷定位模型综合考虑了状态谓词种类、谓词怀疑度计算公式、状态信息的粒度以及不同谓词的组合形式对定位结果的影响，通过系统化的实验研究提出了一个新的缺陷定位方法。实验表明，该定位方法可以有效提升现有缺陷定位的准确度：相比于传统的基于程序频

[①]程序谓词用于描述程序的状态。

谱和程序状态的定位方法可以达到 32.6% 和 227.9% 的准确率提升。该研究工作已发表在 ASE 2019 上。

1.3.3 基于单个样例的补丁生成技术

根据相关研究介绍，由于非频繁缺陷重复历史修改数据量小，导致现有缺陷修复方法无法从中提取有效的补丁模板用于修复。实际上，从我们的人工实证研究中发现，开发者在根据代码修改样例提取修改模板时并不过分依赖于修改样例的重复性。相反的，他所依赖的是代码特征的重复性。即对于给定的一段代码，哪些特征在相似功能的代码中大概率同样存在，比如循环迭代的过程中通常会有“**for**”结构等。基于此，开发者可以较准确的应用历史的修复到当前的缺陷代码上。受上述启发，本文提出基于单个历史修改的补丁模板提取技术指导补丁生成，针对非频繁缺陷存在有限的重复修改历史时指导修复补丁生成。

比如下面的一个例子，图1.3中所示的代码是真实软件中的缺陷修复补丁，来自于缺陷修复的基准数据集 Defects4J(v1.2.0)[44] 中的 Mockito-22。其中，以“+”起始的行代表新插入的代码。

```
12.  public static boolean areEqual(Object o1, Object O2) {
13.+    if ( o1 == o2 ) {
14.+       return true;
15.+    }
16.     if ( o1 == null || o2 == null )
```

图 1.3 缺陷 Mockito-22 补丁代码

从图中可以看出，该缺陷的修复是添加了对传入的两个参数的恒等比较操作 (13-15 行)。实际上，在开源的代码仓库中存在非常相似的缺陷修复历史①，其修改如图1.4所示。

```
526.  boolean _equasComplexEL(Object left, Object right, ...) {
527.+    if ( left == right ) {
528.+       return true;
529.+    }
530.     if ( Decision.isSimpV(left) && Decision.isSimpV(right) ) {
```

图 1.4 开源项目中的历史修复示例

从上面的两幅图中可以发现，虽然两处修改的代码有所差异，但是采取的修改操作是同样的模式：添加两个传入参数的恒等比较，并在条件为真的情况下返回常量 true。因此，如果我们可以在该历史修改中提取出上述的修改模板，我们可以更容易地实现

①https://github.com/clitnak/mcrailo/commit/8e76da8

对 Mockito-22 的正确修复。实际上，上述相似的历史修改在过去的六年时间里，在开源代码中只出现过一次。因此，如何从这单个的样例中自动化地提取出可复用的补丁模板并不容易，因为历史修改代码和新的缺陷代码并非完全一致。

根据上面的描述，从单个修改样例中提取可复用的通用补丁模板主要面临**两个挑战**：(1) 如何定义修复补丁模板的通用表达形式；(2) 如何克服样本数据量小的问题。对于第一个挑战，修复补丁模板的表达形式需要具有灵活的表达能力以使其具有较强的通用性。对于给定的一个代码修改样例，当从中提取修改的模板时，我们需要决定哪些代码属性需要保留 (在应用模式时需要被匹配)、哪些属性不需要保留 (不需要被匹配)。而在不同的上下文中，提取出来的模板所具有的属性也可能存在差异。因此，灵活的表达能力是提高补丁模板在不同场景下通用性的必要条件。对于第二个挑战，在用于学习的样本数据非常有限时 (如图1.4中的修改)，如何决定补丁模板中应该保留的代码特征？根据上面的例子，如果提取的模板包含太多的代码特征信息 (比如变量“`left`”)，会导致其与出错代码不匹配而不适用修复上述缺陷。相反的，如果其包含的特征信息太少，会导致其被用于修复不相关的缺陷。

针对上述的第一个挑战，本文提出一种基于程序图的修改模式表达形式。图模型在静态的软件分析中被经常使用，因为其具有灵活多变、表达能力强的特性，比较适合描述代码结构。但是由于图模型匹配的时间复杂度比较高，因此在海量的代码匹配中被很少使用。目前的研究中多以树型结构用来表示代码修改模板。本文中提出的代码图表达形式通过应用代码图的扩展层概念，可以根据需要灵活控制图的大小。同时，由于本文中使用的补丁模板抽取算法可以在保证模板表达能力的同时避免图的爆炸式增长。因此，在实际使用中并不会导致时间复杂度过高而不可计算。针对上述中的第二个挑战，本文提出了使用海量的开源代码数据弥补样例数据量不足的问题。通过分析海量开源代码中不同代码属性的分布情况，用来指导单个代码片段中的代码属性抽象过程。其基本思想是在不同项目中广泛分布的代码属性具有通用性，因此在提取的补丁模板中应该保留，否则应该删除。该方法将模板提取过程中对重复修改的依赖转变为对属性分布的依赖，从而将对重复样例代码的依赖转嫁到了海量的无关代码上。

通过从海量的开源代码中分析代码属性的分布情况，克服了补丁模板学习对大量重复修改的依赖，从而解决了非频繁缺陷修复中补丁模板提取的难题。同时，由于其使用的图模型可以有效的保存代码上下文信息，可以为修复补丁的生成提供必要的指导。综上，本文提出的基于单个历史修改样例的补丁模板提取方法可以对非频繁缺陷修复的补丁空间提供有效的指导，从而增加生成补丁种类同时提升修复质量。最终实验验证表明，该方法在不依赖重复修改样例的情况下，可以生成高质量的修复补丁模板，并能修复真实项目中的非频繁缺陷。该研究工作已发表在 ASE 2019 上。

1.3.4 基于相似代码的补丁生成技术

上文提出的基于单个历史修改提取补丁模板方法可以在一定程度上克服非频繁缺陷的补丁生成问题。然而，重复的历史修改并不总是存在的。当修复的缺陷在历史中不存在时，上述方法也会失效。从人工缺陷修复实证研究中发现，程序开发者在遇到从未接触过的缺陷时，会浏览当前的缺陷程序代码，根据相似功能的代码片段修复缺陷的代码。其原因是，同一个项目的代码通常是由同一个开发者或同一个团队的开发者所编写，对于相似功能的代码，其代码风格具有一定的相似性。因而，根据相似的代码可以帮助开发者更好的理解程序的功能以及编写修复补丁。比如图1.5中所展示的是数据集 Defects4J 中的缺陷 Closure-57 的修复补丁。

```
1.+  if ( target != null && target.getType() == Token.STRING ) {
2.-  if ( target != null ) {
3.       className = target.getString();
4.    }
```

图 1.5　缺陷 Closure-57 补丁代码

修复该缺陷需要增加一个全新的条件约束`target.getType()==Token.STRING`。事实上，正确生成该缺陷的修复补丁并不容易，其原因主要有两点：首先，我们需要知道当前需要增加的条件是上述的形式 ("==”比较) 以及需要对比的是`target.getType()`的返回值。其次，即使知道以上的修改形式，其对比的对象`Token.STRING`也很难被定位，因为在`Token`类中存在接近 100 种常量可以用来作对比。所以如何确定此处应该使用`STRING`而不是别的常量十分困难。另一方面，历史修改中不存在上述修改记录。但是，通过分析数据我们发现，与出错代码在同一个文件中存在另一处代码，其形式如下图1.6所示。

```
1.  if ( last != null && last.getType() == Token.STRING ) {
3.      String propName = last.getString();
4.      return (propName.equals(methodName));
5.  }
```

图 1.6　Closure 项目中的相似代码片段

通过分析，上述两段代码具有很强的相似性：两段代码具有相同的结构并且对同类型的两个变量 (`target`和`last`) 采取的判断操作是一致的。因此，根据图1.6中的相似代码，修复补丁的搜索空间将被极大地削减 (可以直接定位到常量`Token.STRING`)。然而，在应用上述相似代码生成补丁时，由于两段代码不完全一致而且依赖不同的局部变量，根据其生成修复补丁同样存在挑战。

在该项研究中主要面临以下**两个挑战**：(1) 如何有效的搜索指导补丁生成的参考代码；(2) 如何复用参考代码生成修复补丁。首先，从上述的示例程序可以发现，在根据

参考代码生成修复补丁时，并不是所有的代码片段都具备指导作用，只有功能相近的代码片段才有较强的指导意义。因此如何搜索功能相似的代码是一个关键难题。现有的方法主要考虑代码的文本特征，而没有很好的考虑代码的语义特征。因此，搜索到的代码不具有较好的指导作用，导致容易产生错误的修复补丁。其次，当给定一个参考代码片段，如何根据参考代码生成修复补丁是另一关键难题。参考代码与出错代码之间存在共性的同时也存在差异性，通常情况下不能直接复用参考代码生成修复补丁，如示例程序中的参考代码中存在额外的`return`语句在补丁代码中无用。

针对上述的研究挑战，首先本文提出了一种基于代码语法和语义的相似代码搜索算法，通过将代码的语法和语义特征编码为向量的表示形式实现代码的高效搜索。该方法不仅可以较好的识别代码的语义信息，提升所搜索到的参考代码的质量，可以有效避免任意的代码修改。此外，在根据相似代码生成修复补丁的过程中，本文提出了一种基于代码差异化的补丁生成算法。具体来讲，我们首先计算出错代码和相似代码在抽象语法树上的差别，然后根据其差异获得对代码的细粒度修改操作并实现对相似代码的细粒度复用。然而，细粒度的修改会导致补丁的搜索空间变大。为此，本文在复用相似代码时，同时结合了历史补丁数据，通过统计历史修复中的操作分布对代码复用进一步过滤，从而可以有效地约束补丁搜索空间。

综上，本文提出了一种基于相似代码的补丁生成技术，可以细粒度的复用项目中的相似代码用来生成修复的补丁，可以为非频繁的缺陷修复提供有效指导。相似代码约束了用于生成补丁的代码修改操作以及代码元素，而历史补丁进一步优化了代码修改。该技术克服了非频繁缺陷的修复对补丁模板的依赖。由于修复补丁来源于已有的人工代码，可以有效避免补丁的随机性，提升补丁质量。通过实验证明，该方法可以有效利用相似代码的指导作用，相比最新的缺陷修复技术，修复数量提升了 30.8% 到 33 倍。该研究工作已发表在 ISSTA 2018 上。

1.4　论文的组织结构

本文组织分为六个部分，具体如下:

- **第一章 引言**。主要阐述论文工作的背景，并指出现有研究中的不足以及介绍本文的主要工作和创新点。
- **第二章 相关研究现状**。全面地回顾和总结缺陷修复领域的相关研究工作。
- **第三章 缺陷修复实证研究**。详细介绍人工缺陷修复的实证研究。通过分析人工修复过程，总结了一系列缺陷定位和补丁生成策略，为本文的非频繁缺陷自动修复方法提供指导。

- **第四章 面向非频繁缺陷的自动修复技术**。详细介绍本文提出的面向非频繁缺陷的自动修复方法 IBFix，其中主要包含以下三个主要部分。
    1. **基于状态划分的缺陷定位技术**。形式化地定义了统一的缺陷定位模型用来结合基于程序频谱和基于统计性调试的定位方法，并基于该模型系统性研究不同的结合方式。最后提出基于程序状态划分的定位方法提升定位的准确率以满足非频繁缺陷修复对定位提出的更高要求。
    2. **基于单个样例的补丁生成技术**。提出基于单个样例的补丁模板提取技术。该方法使用图模型表示修复补丁模板。通过分析代码不同属性在海量开源代码中的分布情况，为补丁模板的提取提供有效的指导来克服非频繁缺陷补丁学习样本数据少的问题。
    3. **基于相似代码的补丁生成技术**。提出参考相似代码指导补丁生成的技术。该技术通过对相似代码和缺陷代码的差异分析提取补丁生成的代码修改操作和代码元素。此外，为了进一步优化补丁的搜索空间，通过分析历史补丁中的修改操作分布，对修复补丁进行过滤以提升补丁的质量，克服非频繁缺陷无重复历史修改作为参考时的补丁生成难题。
- **第五章 实验验证**。通过对比实验验证本文提出的自动修复方法 IBFix 的修复效果，同时验证其不同组件 (定位和补丁生成技术) 的有效性。
- **第六章 结论及展望**。总结本文研究工作，并展望未来的可能研究方向或问题。

第二章　相关研究现状

本章对缺陷修复领域的研究现状进行较全面的回顾和总结，并针对缺陷修复过程中所涉及的定位和补丁生成两个过程进行分类介绍。

2.1　缺陷定位技术

根据缺陷定位 (Fault Localization) 技术所使用的程序信息的差异，目前使用比较广泛的缺陷定位方法主要包含基于程序频谱和基于程序状态的两种定位方法。本节将就以上两类定位方法分别介绍其研究现状。

2.1.1　基于程序频谱的定位

程序频谱 (Program Spectra) 是程序在运行的过程中某方面行为的分布情况，比如程序中每条语句被不同测试所执行的情况 (也被称为程序的覆盖) 等。该概念最早由 Reps 等人[86] 在 1997 年提出来用于解决计算机的“千年虫”问题 (Year 2000 Problem) [①]。后来，Harrold 等人[31] 在 2000 年对程序频谱做了进一步的细化分类并通过实验证明了程序的频谱和程序的状态是存在关联的。对于一个软件程序来说，在同一个测试集的测试下，程序中存在错误和不存在错误时测试运行过程中所表现出来的程序频谱是不同的。其具体的分类如表2.1所示。该发现使得程序频谱被广大的科研学者所关注，并应用到了实际的缺陷定位中[2,43]。

2.1.1.1　基于程序频谱的基本定位方法

最早的基于程序频谱 (Spectrum-Based) 的定位方法是 2002 年由 Jones 等人[42,43] 提出来的 Tarantula，也是被广泛使用的基于程序频谱的缺陷定位方法。该方法通过记录所有测试程序在运行过程中的完整路径频谱，根据不同测试的频谱分布情况计算每条语句出错的概率。其基本思想是：对于一条语句来说，如果该语句被越多的未通过的测试所执行到、被越少的通过的测试所执行到，则该语句出错的可能性越大。相反的，如果语句被越多的通过的测试所执行到、被越少的未通过测试所执行到，其出错的可能性越小。因此统计每条语句被通过和未通过测试执行的次数，根据预先定义好的公式可以计算得到每条语句的出错概率。由于该类方法是基于频谱定位方法的基础，为了方便理解，下面将结合一个具体的示例程序对该过程做进一步的介绍。

[①] https://en.wikipedia.org/wiki/Year_2000_problem

表 2.1 程序频谱分类

程序频谱	描述
分支命中频谱 (Branch Hit Spectra)	程序中每个条件分支语句是否被执行
分支计数频谱 (Branch Count Spectra)	程序中每个条件分支语句被执行的次数
完整路径频谱 (Complete Path Spectra)	程序运行的完整路径
路径命中频谱 (Path Hit Spectra)	过程间的无环执行路径
路径计数频谱 (Path Count Spectra)	过程间的无环执行路径被执行的次数
数据依赖命中频谱 (Data-dependence Hit Spectra)	程序中的每个“定义-使用”对是否执行
数据依赖计数频谱 (Data-dependence Count Spectra)	程序中的每个“定义-使用”对被执行的次数
输出频谱 (Output Spectra)	程序执行输出的结果
执行踪迹频谱 (Execution-trace Spectra)	程序完整的执行路径 (实际执行的指令)

表2.2所示为一个存在缺陷的程序以及测试用例对该程序进行测试的情况，其中程序中的第 7 行是缺陷代码位置。表中的每个方格内的 “✓’’ 表示对应测试会执行对应行所在的代码，也称为覆盖所在行的代码。表格的最后一行是每个测试的测试结果，其中“P”表示该测试通过，“F’’ 表示该测试失败。Tarantula 根据每个测试对程序中每条语句的覆盖情况以及对应测试的测试结果计算每条语句可能出错的概率。计算的公式如下所示：

$$susp(s) = 1 - hue(s) = \frac{failed(s)/totalfailed}{passed(s)/totalpassed + failed(s)/totalfailed}$$

其中，$failed(s)$ 和 $passed(s)$ 分别表示覆盖了程序元素 s 并且未通过和通过的测试函数的数量。而 $totalfailed$ 和 $totalpassed$ 分别表示未通过的测试函数总数和通过的测试函数总数。根据上面的公式可以对程序中的每个元素计算怀疑度值并排序。在实际的应用中，程序元素的粒度可以根据需要设定。表2.2中的最后两列分别为根据上面的公式计算的对应程序语句的怀疑度值及其对应排序位置。根据表中的结果，第 7 条出错语句排到了第 1 的位置，因此补丁生成过程会优先尝试对第 7 条语句进行修复。

Eric Wong 等人[20] 在 Tarantula 的基础之上提出了一种启发式的方法用来提高 Tarantula 的准确性。对于每个通过的测试语句，Tarantula 方法认为它们所起的作用都是一样的，而 Eric Wong 等人提出不同的测试所起到的作用应该是不一致的。对于一个程序元素来说，第一个覆盖该元素并且通过的测试应该给了开发人员比较大的信心认为该语句不太可能会出错，而对于后来覆盖该语句同时通过的测试来说，其对于开发人员判断该语句不会出错所增加的信心是相对较弱的。基于以上的想法，他们提出了为不同的测试赋予不同的权重，从而改善定位方法的准确性。

表 2.2　缺陷程序的测试覆盖情况

	mid(){ int x, y, z, m;	测试输入 3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	susp	rank
1:	read("Enter 3 numbers:",x,y,z);	✓	✓	✓	✓	✓	✓	0.5	7
2:	m = z;	✓	✓	✓	✓	✓	✓	0.5	7
3:	if(y < z)	✓	✓	✓	✓	✓	✓	0.5	7
4:	if(x < y)	✓	✓			✓	✓	0.6	3
5:	m=y;		✓					0.0	13
6:	else if(x < z)	✓				✓	✓	0.7	2
7:	m=y; //bug	✓					✓	0.8	1
8:	else			✓	✓			0.0	13
9:	if(x > y)			✓	✓			0.0	13
10:	m=y;			✓				0.0	13
11:	else if(x > z)				✓			0.0	13
12:	m=x;							0.0	13
13:	print("Middle number is :", m);	✓	✓	✓	✓	✓	✓	0.5	7
	}								
	测试是否通过状态:	**P**	**P**	**P**	**P**	**P**	**F**		

在此之后，又有很多类似的定位技术被先后提出来，比如 O/Op[41]，Crosstab[110] 等。不同的方法都是对上面的怀疑度计算公式进行一些修改来提高定位的准确性，但其基本思想都是一样的，这里不再一一赘述。在 2006 年，Abreu 等人[2] 通过实验比较了四种采用不同计算公式的基于程序频谱的缺陷定位方法：Jaccard[16]，Tarantula，AMPLE[19] 和 Ochiai[2]。实验结果表明 Ochiai 公式的的表现效果明显优于其他的三种公式，Jaccard 的效果要优于 Tarantula，AMPLE 表现效果最差。平均下来，针对不同的项目，Ochiai 公式的定位效果相比于 Jaccard (效果第二好) 提升了 2.4%-10%。因此，在最新的缺陷修复方法中，Ochiai 公式也被广泛使用[59,115]。

通过程序的频谱对程序中的缺陷进行定位是目前被应用最为广泛的定位方法，也是实验中被认为是最好用的方法。其原因主要包含以下两点：(1) 过程简单，计算方便；(2) 对于较大的项目不存在不可计算问题，时间开销和存储开销不大。虽然如此，基于程序频谱的缺陷定位方法依然面临着定位准确度不高的问题，在测试集的测试能力不足的情况下，定位的效果会非常容易受到影响。因此，出现了一些相关的优化方法，比如对测试函数进行提纯[119] (Purification)，通过简化未通过的测试函数来排除与出错不相关的程序元素的干扰；以及通过机器学习模型结合不同的计算公式的定位结果[118] 以发挥不同方法的优点；甚至通过加入程序员的反馈信息[121] 来改善现有方法的准确性等等。

2.1.1.2 基于变异的定位

变异测试[①] (Mutation Testing) 在软件测试中被经常用来衡量一个测试集的测试能力，其基本想法和上文提到的基于程序频谱的方法类似，都是应用测试集的覆盖信息。对于一个拥有比较强的错误检查能力的测试集来说，当程序中的某个元素被修改之后，覆盖该程序元素的测试函数会因为被修改语句计算了错误结果导致测试失败。

基于该发现，Papadakis 和 Le Traon[77,78] 提出了一种基于变异 (Mutation-Based) 的缺陷定位方法 Metallaxis-FL。该方法的基本假设是：对于一个给定的程序 p 和测试集 T，将程序中的某条语句变异成不同的形式从而获得变异后的程序 p_1 和 p_2，测试集 T 对 p_1 和 p_2 分别测试所得到的测试结果应该是类似的。即测试程序 p_1 时，通过的测试函数，在测试 p_2 时应该有大的概率通过；对于未通过的测试函数类似。该假设的依据是对于每条语句来说，运行该语句的测试是确定的，对于没有覆盖被变异语句的测试函数来说，在程序 p_1 和 p_2 中的表现应该是完全一样的，都能通过测试。而对于覆盖被变异语句的测试函数来说，由于不同的变异都是将正确的语句改成了错误的语句，在理想情况下这些测试在 p_1 和 p_2 中应该都不能通过测试。而在实践中，即使测试集 T 在程序 p_1 和 p_2 上的结果不完全一样，他们应该具有很高的相似性。所以，在论文中提出了一种使用已知变异来推测未知出错位置的方法实现高效定位 (未知的错误被视为被变异的程序)。类似的，在计算语句怀疑度值时使用了 Ochiai[2] 公式。

此外，Moon 等人[72] 于 2014 年提出另一种基于变异的缺陷定位方法 MUSE。其基本思想是如果变异的语句刚好是错误的语句，那么原本未通过的测试函数变成通过的可能性会比变异了原本正确的语句大，而对于原本通过的测试这个结论刚好相反。因此，根据程序植入变异前后测试结果的变化情况来对程序中的未知错误进行定位，类似的提出了一个新的公式：

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right)$$

其中 $mut(s)$ 为所有的变异算子，$f_P(s)$ 为覆盖语句 s 并且未通过的测试函数集，$p_P(s)$ 为覆盖语句 s 并且通过的测试函数集，p_m 和 f_m 分别为应用变异算子 m 后通过和未通过的测试函数集合，α 是可调的常系数。根据公式可以看出其中的前半部分计算了在语句 s 上应用变异算子 m 之后，从未通过变成通过的测试数量的正相关比例，而后半部分计算了从通过变成未通过的测试的负相关比例。通过实验验证，该方法可以在一定程度上改善基于程序频谱的缺陷定位方法的准确性。

分析上面的两种基于变异的缺陷定位方法发现，通过引入变异算子可以获得更丰

[①] https://en.wikipedia.org/wiki/Mutation_testing

富的变异程序，通过错误代码之间的相似性可以帮助更准确定位程序中的错误代码。但同时需要注意，使用基于变异的定位方法在项目比较大的数据集上会增加定位的时间开销，虽然 Papadakis 和 Le Traon 在扩展的期刊论文[78] 中对 Metallaxis-FL 在较大项目上 (代码行数 <1.5w) 的效果也做了验证，但相比于真实场景下的大规模项目 (代码行数 >10w) 还有距离，因此在目前的自动化缺陷修复方法中使用相对较少。同时对于面向对象的编程语言，存在的很多缺陷使用现有的变异算子不能很好的模拟 (比如 Object 类型) 也使得该类方法受到一定的制约。

2.1.1.3　基于程序切片的定位

类似于基于程序频谱的定位方法，基于程序切片 (Program Slicing) 的定位同样是考虑程序语句的覆盖情况。然而，与传统的基于程序频谱的定位方法的不同点是基于程序切片的定位方法所依赖的“频谱”并不是仅仅根据程序 (测试) 执行的覆盖确定的，而是同时结合程序中变量值的迁移关系所确定的程序元素的覆盖情况。简单说，基于动态程序切片的定位方法会根据未通过测试的输入 (或结果) 来正向 (或反向) 追踪被输入值影响 (或影响输出值) 的语句，这些语句才真正会影响测试执行的最终结果。因此，相比于基本的基于程序频谱的定位方法，该方法可以进一步缩小候选出错位置的搜索空间。

程序切片最早在 1981 年由 Weiser 提出[103]，其文中主要对程序切片的计算方法进行研究，静态切片技术只依赖于对程序的静态代码进行分析，不需要执行对应的程序。该方法面临计算复杂度高和切片不精确的缺点。1988 年由 Korel 和 Laski[49] 提出了一种动态程序切片技术，动态切片中只保留了程序中和具体的某一次程序执行相关的语句，相比于静态程序切片可以大大减少程序切片的规模，同时获得的结果更加精确。使得程序切片技术得到更加广泛的应用。

Zhang 等人[128] 在 2006 年提出了一种通过对程序的动态切片进行剪枝的方法对程序中的缺陷进行定位。该方法通过搜集程序的运行时信息对动态切片的结果做进一步的筛选。该方法首先将程序中的语句分成了三类：(1) 只被未通过的测试执行的语句集合 F，(2) 只被通过的测试执行的语句集合 S，(3) 被通过和未通过的测试都执行到的语句集合 FS。接下来通过程序动态切片技术获取到原始的程序切片 (数据依赖和控制依赖) Q。其中集合 S 中的语句可以直接删除，因为其中的语句对未通过的测试没有影响；集合 F 和集合 Q 的交集全部保留，该部分语句只被未通过的测试执行过，因此具有最大的出错概率 (类似于基于程序频谱的定位方法)；最后是将未通过测试和通过测试都执行过的语句作为候选的语句 ($FS \cap Q - F \cap Q$)。最后通过静态代码分析对切片中的不同语句进行过滤。该方可以很大程度上缩小候选出错语句的数量 (缩小 1.79-190.57 倍)，可以有效减小候选错误代码位置空间大小。

文万志等人[136] 提出了一种基于层次切片谱的错误定位技术，根据面向对象 Java 语言的层次结构特性，从包层次、类层次、方法层次到语句层次对程序逐层进行切片，且从粗粒度到细粒度逐层计算出错怀疑度比较大的对应程序结构 (包、类等)，并根据得到的结果对出错概率比较高的程序结果进一步细化。即低一层次的切片依赖高一层次的定位结果。比如当前得到 A 类出错的概率最大，接下来仅考虑 A 类内的所有函数而排出其他类中的函数，从而降低了程序切片的开销。因此，该过程是一个不断优化的过程，实验结果表明该方法可以提升定位的效率。

此外，Soremekuno 等人[91] 通过实验证明程序切片技术在定位技术中具有重要的辅助作用，对于提高现有技术的定位准确性有很好的促进作用。

2.1.2 基于程序状态的定位

在之前的基于程序频谱的定位方法中，所使用到的主要信息是测试函数对待测试程序的代码覆盖信息。然而，程序中的错误通常是因为程序中的某些操作导致一些不满足当前环境的值 (程序状态) 的出现，引起程序崩溃或者计算结果出错 (如对`null`指针解引用导致程序崩溃)。因此，这种通过某种方式定义程序在特定环境下的合法状态，然后通过判断未通过测试函数在不同环境 (比如不同函数) 中的状态是否与合法状态矛盾来定位程序的出错位置形成了基于程序状态的定位方法。本文将基于状态的定位方法主要分为以下五种类型：统计性调试、谓词翻转、天使调试、差异化调试和不变量挖掘。

2.1.2.1 统计性调试

统计性调试 (Statistical Debugging) 最早是由 Liblit 等人[55] 提出来用于程序状态的远程采样，其目标是辅助开发人员远程检测程序的运行状态是否正常以及发现异常之后帮助开发人员定位缺陷的位置。该方法通过在程序代码中注入一些预定义的谓词来检查程序的执行状态。相比于之前介绍的缺陷定位方法，其输出的结果是预定义谓词与当前缺陷的关联密切程度，而并不是程序元素的出错概率。比如 Liblit 等人[56] 提出来的通过计算谓词被通过测试和未通过测试的覆盖情况来衡量不同谓词的重要程度。其基本思想和基于程序频谱的定位方法类似：被更多的未通过测试覆盖则其重要程度越高。相反的，被越多的通过的测试覆盖其对应的重要程度会降低。基于统计性调试的思想，很多后续工作对其进一步优化。

Liu 等人[14,57] 对传统的谓词覆盖方法进行了更细粒度的统计。在上述介绍的传统方法中，谓词的覆盖情况被记录为布尔变量，而没有考虑其被覆盖的具体次数。对此，Liu 等人提出了一个新的的计算公式 Sober，该公式通过分别计算谓词在通过测试和未通过测试的运行中的覆盖 (具体讲为分布) 情况，然后对比在两种情况下的分布差异计

算谓词的重要性。由于该方法对通过测试和未通过测试分别独立计算状态分布，因此可以在一定程度上降低由于两种测试覆盖位置不同导致的定位不准。

Jiang 和 Su[39] 提出将不同的状态谓词进行聚类，然后根据聚类后的谓词构造具有可疑度的控制流路径。其具体过程是根据程序运行时搜集到的谓词覆盖信息对谓词进行聚类，根据聚类之后的结果以及每个谓词的覆盖情况在程序的控制流图上遍历节点获得可能出错的控制流路径，使得该路径可以覆盖一个类别中的不同谓词位置。该方法将谓词和执行路径相关联实现缺陷代码的定位。类似的，Zheng 等人[129] 采用交互式的聚类方法对状态谓词进行聚类，然后根据多类谓词可以定位出程序中存在的多处缺陷。Chilimbi 等人[17] 同样结合了谓词的状态以及执行路径信息，同时还结合缺陷报告的内容辅助定位代码错误，但该过程依赖于程序开发人员的交互。

由于统计性调试方法需要对程序进行插桩以及运行测试来检查程序的执行状态，当插桩谓词数量比较大的情况下，运行时间开销会增加。Zuo 等人[133] 提出了分步插桩的方式对其进行加速。其基本思想是对程序进行不同的粒度划分，从粗粒度到细粒度逐步迭代精化插桩。比如先在代码的函数级别插桩，然后再对重要函数内的语句进行插桩。该方法可以有效降低程序的执行开销。与此不同，Misherghi 和 Su[70] 提出对测试的输入进行逐步化简以提升定位的准确性，该方法结合了差异化调试的方法 (见下文)，因此也受到差异化调试方法的局限：即要求测试输入是结构化的数据以方便化简操作。

2.1.2.2　谓词翻转

谓词翻转 (Predicate Switching) 是 2006 年由 Zhang 等人[127] 提出来的一种针对条件错误的高效定位方法，即所对应的缺陷代码位置需要增加新条件或者删除、修改现有条件。该方法通过翻转程序中的条件语句取值，然后运行测试来捕获该条件语句对未通过的测试是否存在影响，从而判断该条件语句是否出错。其定位过程是：(1) 查找错误的值：运行未通过的测试找到导致程序崩溃的值最早出现的位置；(2) 找到需要翻转的谓词：再次运行测试，记录运行过的所有分支语句集合 PT (每个函数入口被认为是一个分支) 以及运行的顺序；(3) 找到关键谓词：对于 PT 中的每个谓词，根据其运行顺序的倒序逐个翻转谓词，然后运行测试，找到可以使未通过测试通过的谓词翻转，则得到的对应谓词即出错的条件语句位置。其翻转的过程示意图如图2.1所示：从执行路径中的谓词序列 $P_m...P_1$ 中，依次翻转谓词，直到输出正确的结果 $O_{correct}$。

尽管在实验中该方法定位效率可以接受，大部分缺陷可以在 1 分钟内实现定位结果输出。然而，根据上面的算法得到的结果在某些类型错误上的定位效果会非常差。原因是程序中的某些错误虽然通过翻转条件可以使测试通过，但真正出错的语句不一定就是当前的条件语句。因此，在上述过程之后该方法通常会结合程序切片技术对程序中可能出错的语句进一步定位。但该方法主要被应用到一些简单的谓词结构上，比如

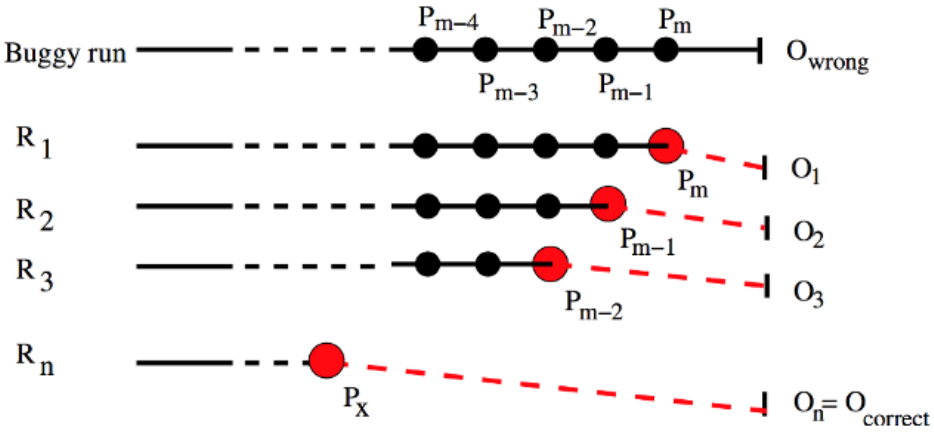


图 2.1　谓词翻转示意图

和常量比较等。2007 年，Nainar 等人[4] 提出了一种利用三值逻辑和程序的静态结构将该方法从简单谓词结构扩展到了复杂的谓词结构上，改善了该方法的使用价值。尽管如此，该方法主要被应用在条件语句错误的定位上。在通用的缺陷定位中效果依然不够理想。

2.1.2.3　天使调试

天使调试 (Angelic Debugging) 技术是 2011 年由 Chandra 等人[13] 提出。该方法只针对程序中的现有语句存在错误的情况，并且出错的语句只能有一处。否则该方法的定位效果会很差。该方法的主要想法和上面介绍的谓词翻转类似，不同点在于此处可以使用任意的常量值替换任意的程序元素，而谓词翻转只针对布尔类型的条件表达式。具体来讲，该方法通过尝试使用常量来替换程序中的表达式的值使之前不能通过的测试可以正常通过。如果可以找到这样的一个常量替换表达式，则对应的表达式即可能是出错的代码位置。

由于该方法在对程序运行过程中的值替换过程依赖符号执行技术来求取，导致代码的规模一般不能太大。因此，在实际的应用中，通常依赖于人工等方式提前划定一个可疑的有限范围，然后对该范围内的所有语句进行逐一筛选。此过程中采用了受控的符号执行技术，即在应用符号执行的过程中只将选中的语句用符号值代替，其他的位置依然使用程序运行的真实值。比如图2.2展示的是一个符号执行的完整过程，图中左侧的代码是存在错误的程序，其中的第 2 条语句出错，正确的表达式应该是“`sgn=1`”。

在图中的符号执行树中，符号执行的过程中会记录每条分支的条件以及当前所涉及变量的具体值。在符号执行树中连接两个节点的边上的编号代表即将要执行的语句，而对应的两个节点分别是执行对应语句前后的程序状态。上图中的执行树针对的是语句 1，因此将变量`x`替换成了符号变量 X，在进行语句 1 的条件判断时，由于 X 是一个

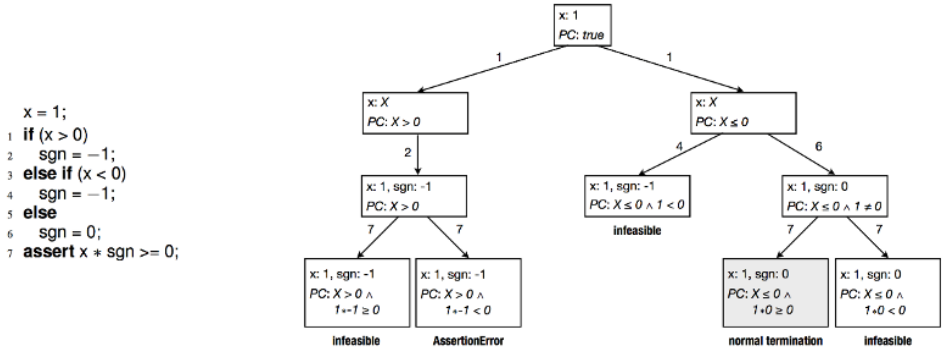


图 2.2　程序源代码及对应的符号执行树

符号变量，因此对其所有的可能取值进行尝试 ($X > 0$ 或者 $X <= 0$)。而在运行其他的语句 (非语句 1) 时，变量x依然使用其真实的具体值。如执行树最左侧的叶子节点中变量x的值为 1 而非未知。这种方法可以得到程序的所有运行状态。最后在这些状态中找到可以被满足的程序状态 (如图中的“normal termination”)，其对应的符号变量 X 的取值则对应了修复该程序的约束条件。相反的，如果不能找到被满足的程序状态，则当前所替换的表达式被认为是正确的表达式。此外，该方法不仅在较大规模的程序上存在效率问题，该方法受符号执行技术的限制，不能处理数组下标变量等。

综上，该方法对于处理简单的程序缺陷或者数值计算类的语句错误效果比较明显。但是对于复杂的程序，由于受到方法和符号执行技术的限制不能得到很好的应用。实际上，在 2008 年 Jeffrey 等人[36] 就已经采用了类似的定位思想，但是他们针对程序元素值的替换部分采用了不同的处理方法，通过搜集不同测试在运行的过程中程序的状态，然后将未通过测试的某些状态使用其它测试的状态进行替换来判断当前的语句是否可能存在错误。相比于依赖符号执行的方法适用性更强。但其对测试集的依赖更强，导致有些状态可能通过现有的测试数据获取不到，导致更大的不确定性。

2.1.2.4　差异化调试

差异化调试 (Delta Debugging) 最早是由 Zeller 和 Hildebrandt[126] 提出来用于简化失败测试的输入。其基本的想法是对测试失败的测试输入数据不断的应用预先定义好的修改 δ (比如 δ 为对测试输入每次删除一个字符，该方法通常适用于输入数据为字符串类型)，然后不断的应用一系列的修改 $\delta_1 ... \delta_n$ 得到新的测试输入并重新运行修改之后的测试，直到当前的测试输入可以导致程序出错并且不能进一步简化。

2002 年，Zellers[125] 将该方法应用到缺陷定位上。其具体过程是首先采用差异化调试技术将失败的测试输入简化到最小。在此过程中，可以找到导致程序测试失败的最小输入差异。然后同时运行包含这个最小差异的两个测试，对两个测试程序的运行

中间状态分别采样 (在统一采样点)。比较两个测试在采样点处的内存状态差异 (变量数量、变量取值等)，分别将其对应的内存状态映射为一个内存状态图，然后使用差异化调试方法不断修改通过的测试的内存状态，找到最小的内存状态改变使得通过的测试变成不能通过。其输出是给程序员提供程序运行的状态差异，最后需要依赖程序员判断当前的内存中呈现对应状态是否符合要求，如果是非法状态则可以定位到该状态产生的位置即为程序出错的位置。

该方法通用性很强，其核心思想是通过不断地最小化差异来缩小目标搜索空间。此后，Burger 和 Zeller[11] 进一步将其与程序切片相结合来减少未通过测试运行过程中的无用函数调用。在其实验中，一个涉及 1 万 4 千多个函数调用的测试被简化成 2 个函数调用，同样产生相同的错误。该方法可以有效减少程序员对出错位置的排查。但是存在的问题是由于涉及内存状态图上的比较，因此时间开销会比较大，同时因为在调试的过程中涉及到对内存状态的修改等，实现复杂且需要有工具支持。另一方面，由于其输出的结果依赖程序员对其进行筛选和判断，在当前的自动化缺陷修复方法中还难以得到应用。

2.1.2.5 不变量挖掘

研究发现，在程序中会存在一些函数或代码片段，其输入的数据和输出的数据是满足某种不变量关系的。比如对于一个要求输入小时 (时间) 的函数，其输入的数据通常应该小于 24，如果输入的数据大于 24，则程序会出错。再比如对于一个计算除法的函数，其输入的除数应该不为 0。通过不变量挖掘对程序中的错误进行定位就是利用了上面的程序特性。首先通过预定义谓词[6] 或者数据挖掘[80] 的方法获得程序中的不变量关系，然后通过未通过测试在对应位置的真实值与不变量关系的比较来缩小候选代码位置空间。然而，单纯地依赖不变量信息，存在较大的不确定性。一方面是并不是所有的缺陷代码位置都能发现不变量的存在。另一方面，不变量关系的来源依赖测试的覆盖或者统计信息，其准确性不能得到有效的保证。

B. Le 等人[6] 提出了结合程序中的不变量关系和程序频谱定位结果的一种基于排序学习模型 (Learning to Rank) 的缺陷位置排序方法 Savant。Savant 主要分为四步：挖掘不变量、基于程序频谱定位、联合训练模型、缺陷定位。具体来讲：挖掘不变量是指通过应用 Daikon[21] 工具挖掘通过的测试中的不变量关系；基于程序频谱定位在上文中已经介绍，其中，Savant 采用了多种不同计算公式的定位结果；最后的训练和定位过程是将前两个过程中得到的数据转换为特征表示形式，训练一个排序模型并用其对候选缺陷进行预测。

图2.3展示了 Savant 对程序中的错误进行定位的完整过程。其中用到了测试用例筛选和聚类方法来降低模型训练过程中的时间和存储开销。Savant 不仅使用了程序的不

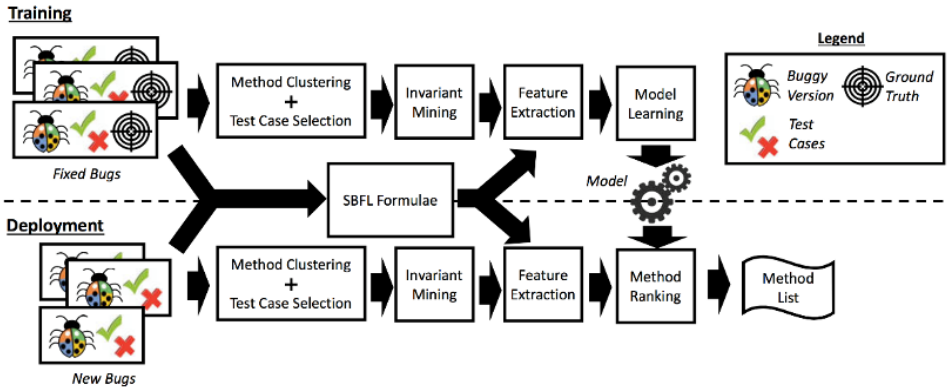


图 2.3　Savant 缺陷定位过程概览

变量信息，同时结合了基于程序频谱的定位方法的结果，使得最终实现了比较好的定位效果：和效果最好的基于程序频谱的定位方法相比，准确度平均提升了 51.37%。但是该方法因为使用了机器学习的方法来训练模型，需要足够的训练数据集。和单纯地结合基于程序频谱的定位方法[118] 相比，不变量关系信息在定位中所起的作用大小并不清晰。

2.1.3　讨论与小结

根据上面的介绍可以发现，基于程序频谱的定位方法主要依赖于测试集对程序的覆盖信息，覆盖率越高其定位的效果表现会越好。由于其方法简单，在缺陷修复领域被广泛使用。然而，在真实的项目中，测试的覆盖率并不能满足准确定位的需求。对于非频繁的缺陷而言，由于这类缺陷在实际开发中发生频率比较低，开发人员对其缺乏足够的经验和重视，导致对其测试更加不充分。因此，测试不充分会为缺陷定位带来更大的挑战。另一方面，基于程序状态的定位方法分析变量的取值情况辅助定位，由于过程复杂，时间开销较大导致较少被使用。另一方面是该类技术通用性较差。比如谓词翻转方法主要针对条件语句类型缺陷，天使调试方法由于依赖符号执行和约束求解导致在较大项目上延展性较差，差异化调试对测试的输入有要求 (结构化数据)，而不变量挖掘定位效率较低。虽然统计性调试方法通用性较强，但是该方法不能定位出错代码，而是返回可疑的程序状态，不能直接集成到自动化缺陷修复方法中。同时，由于该类方法基于经验知识定义谓词，谓词数量同样会限制其定位的效果。

2.2　补丁生成技术

补丁生成 (Patch Generation) 实际上是研究如何定义一个合适的补丁空间 (**操作空间**和**元素空间**) 以及在这个空间中准确而快速地找到正确的修复补丁。通常该过程返回一个或者排序后的多个修复补丁。目前主流的技术主要包含四种类型：基于启发式搜

索，基于人工定义，基于约束求解以及基于统计的补丁生成方法。

2.2.1 基于启发式搜索的补丁生成

基于启发式搜索的程序修复技术主要思想是程序中的代码具有相似性和重复性，尤其是在同一个项目中。因此基于启发式搜索的方法通过定义一组启发式搜索策略在已有的代码中尝试搜索并复用这些代码生成修复补丁。

该方法在早期的缺陷修复中应用比较广泛，如缺陷修复方法 GenProg[24,51]。该方法将源代码转换为抽象语法树 (Abstract Syntax Tree) 的表示形式，然后通过定义启发式搜索规则应用遗传算法搜索可以用来复用的代码片段。该过程中的遗传算法主要应用两种代码修改操作：交叉 (Crossover) 和变异 (Mutation)。其中交叉操作指的是对程序两个执行路径上的代码语句进行重新组合和拼接，而变异操作是指在程序的执行路径上插入一条新的代码语句或者删除、替换一条已有代码语句。为了在搜索的过程中提高搜索的效率，文中定义了带权重的路径。具体来说，根据每条语句被测试用例执行的次数为其赋一个权重，使被执行次数比较高的程序片段具有更大的几率被搜索到。同时，如果某条语句只是被正确通过的测试用例执行，可以通过设置其权重为 0 来防止在之后的变异操作中删除该条语句。GenProg 的补丁生成算法主要利用了程序中的重复代码，在此过程中同时采用类似差异化调试的方法对修复的补丁进行最小化处理。

在此之后，该方法又被进一步优化。Weimer 等人[102] 提出了对生成的修复补丁和测试的运行顺序进行排序，避免相似的修复补丁被重复验证来节省运行的时间。此外，为了尽早地排除掉不正确的补丁，在每次验证新生成的补丁时优先运行在之前验证中未能通过的测试。基于此设计并实现了修复工具 AE。然而，Qi 等人[81] 通过实验证明 GenProg 中的遗传算法所起作用并不大。他们提出缺陷修复方法 RSRepair，该方法保留了 GenProg 的生成框架，只是将其中的遗传算法改成了随机搜索算法。实验结果发现随机搜索算法相比于遗传算法在 23/24 个测试用例上具有更高的效率。

早期的自动化缺陷修复方法仅仅依赖是否通过测试判断修复补丁的正确性，该方法在测试集比较弱 (测试覆盖率低) 的情况下难以保证补丁正确性，导致补丁的质量低下，大量不正确的补丁被误认为正确修复。Qi 等人[82] 在对 GenProg、RSRepair 以及 AE 生成的修复补丁进行了人工验证分析之后发现其生成的修复补丁虽然可以通过测试的验证，但实际上并不能真正修复程序中的缺陷，大部分的修复补丁通过删除原有程序的部分功能使测试通过。据此，它们进一步提出了仅应用代码删除操作的修复方法 Kali，并通过实验证明其与上述三种方法实现相似修复效果。实验验证暴露了仅依赖测试验证补丁正确性的局限性。在此之后，补丁的验证方法在原有的基于测试的验证基础之上增加了人工验证的步骤，可以更有效的提升补丁验证的准确性。

Le 等人[52] 同样采用了遗传算法提出了缺陷修复方法 HDRepair。该方法通过统计

开源项目中的常用缺陷修复模式对最后生成的修复补丁进行排序，使得正确的补丁可以有效地排在比较靠前的位置。类似的，Xin 和 Reiss[113] 提出 ssFix 直接复用开源代码库中已有代码作为修复补丁的原材料。ssFix 首先使用 Apache Lucene 的文本搜索接口在开源代码库中搜索相似代码，然后通过映射出错代码和相似代码中的变量对应关系实现代码的复用。最后根据补丁的修改大小对候选补丁进行排序。

Wen 等人[104] 提出了一种依赖代码上下文的补丁生成方法 CapGen。该方法同样在开源代码中统计历史修复中经常使用的修改操作，与之前的方法不同的是在统计修改的过程中同时记录与修改相关的上下文信息。比如在 ssFix 中只统计替换变量的出现频率，而 CapGen 在统计的时候会考虑所替换变量所在的表达式类型 (比如在中缀表达式中)。通过修改上下文的约束，可以缩小修改操作的适用范围，有效避免任意修改操作生成的低质量补丁。同时，CapGen 在生成修复补丁时所使用的程序元素会根据其在项目中出现的频率进行排序，优先使用比较频繁的程序元素。该方法通过上述两种方式，大幅度地提升了缺陷修复方法的准确率，减少错误补丁的生成。

2.2.2　基于人工模板的补丁生成

基于人工定义的补丁生成是指由程序员预定义一些修复的模板，在生成修复补丁的时候直接套用模板。该方法的好处是生成的修复补丁更接近人工修复的补丁，易于理解和维护。但是由于需要人工预定义，而模板的种类和数量很难确定，难以覆盖所有修复补丁类型或者导致模板数量过大引起预定义和搜索空间的变大而难以实用化等。

Kim 等人[46] 提出的 PAR 是具有代表性的基于人工定义模板的补丁生成方法。他们通过分析真实缺陷的修复补丁发现其中 8 种用于补丁生成的模板，可以为 30% 的缺陷生成正确的修复补丁。根据此发现，PAR 预定义了 10 类修复模板并为每个模板实现补丁生成过程，不同模板之间相互独立，如表2.3所示。PAR 根据缺陷定位的候选出错位置逐一尝试应用模板生成修复补丁，直到补丁可以通过所有测试。其修复过程如图2.4所示。该方法在实际的验证中效果显著，可以有效提升生成补丁的质量。

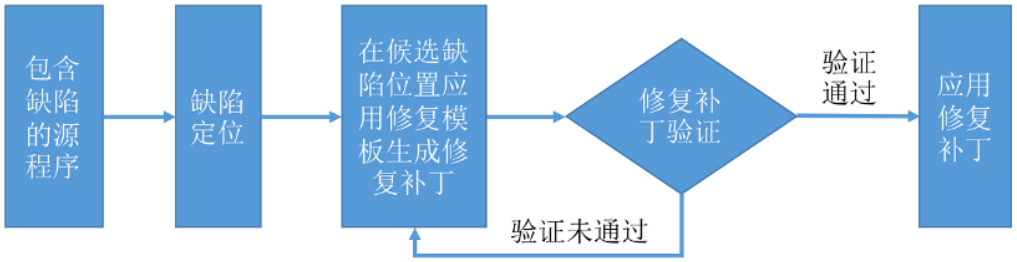


图 2.4　PAR 缺陷修复流程图

表 2.3 PAR 的修复模板

模板名称	功能描述
参数替代	在作用域范围之内，选择一个类型匹配的变量或表达式替换函数参数。
函数替代	用一个具有相同参数和返回类型的函数替换某个函数调用。
参数添加和删除	对于一个函数调用，如果存在多态函数，删除其中的多余参数或者在作用域范围之内找一个类型兼容的变量或者表达式作为新添加的参数，使得调用同名多态函数中的另一个。
表达式替代	在作用域范围之内，使用其他的表达式替换条件判断语句或者三元操作中的条件表达式。
添加和删除表达式	在条件表达式中添加一个合法的表达式或者删除一个已有表达式。
空指针检查	当语句中存在对象引用，添加`if`条件判断语句对该引用对象做空指针判断。
对象初始化	在函数调用的前面对函数调用的参数进行初化。
数组范围检查	对于数组引用，添加一个`if`条件判断语句检查引用的下标是否越界。
集合大小检查	对于集合类型变量，添加`if`条件判断语句检查下标引用是否越界。
类型强制转换检查	对于对象的强制类型转换，添加`instanceof`条件对转换的对象判断兼容性。

Long 和 Rinard[62] 提出了一种基于代码变换的补丁生成方法 SPR。由于该方法通过定义了一系列的代码变换模式 (Transformation Schemas)，类似于 PAR 中用于生成补丁的模板，因此同样可以被总结为基于人工定义的方法。其区别在于 SRP 将代码的变换模式定义得更加详细，不同修改模式可以相互组合而实现复杂修复，因此表达能力更强。该方法只针对条件语句的修复，首先在出错的代码位置应用变换模式修改代码并用符号表达式表示修改之后的代码，然后根据程序运行过程中表达式的取值情况，搜索对应的替换表达式来生成最终的修复补丁。

Tan 等人[95] 通过分析之前缺陷修复方法所产生的修复补丁发现，一些特定的代码修改操作更容易导致不正确的修复补丁，如删除程序中的部分语句 (功能) 比较容易通过测试的验证，从而生成不正确的修复补丁。基于此，她们提出了定义反模板 (Anti-Pattern) 对 GenProg 和 SPR 生成的补丁进行过滤。实验效果证明该方法虽然并不能提升现有修复工具的修复能力，但是可以显著地提升修复工具的修复效率：对 GenProg 和 SPR 分别有大概 1.4x 和 1.8x 的效率提升。其根本原因是使用反模式可以尽早地将不正确的补丁排除掉，从而节省了不正确补丁的验证时间。

Saha 等人[87] 提出了一种主要针对面向对象语言 (Java) 中的函数调用错误的缺陷修复方法 ELIXIR。该方法同样是预定义了一些程序转换的模式 (Repair-Expressions)，然后在出错的位置使用合法的程序元素 (如变量、数组、函数调用等) 来实例化变换模式生成修复的补丁。为了提升修复补丁的质量，ELIXIR 进一步结合修改代码的上下文特征信息对候选补丁进行排序。

Chen 等人[15] 提出了缺陷修复方法 JAID。与之前的方法类似，JAID 首先是通过定义了一些表达式的变换模板对原程序中的代码进行一些修改生成新的可复用代码片

段。然后在补丁生成阶段根据预先定义的补丁生成模式 (如修改表达式、修改程序执行流等)，用上一步新生成的代码片段实例化修复补丁。

与上面提到的方法类似，Hua 等人[34] 提出的 SketchFix 方法同样定义了一系列的代码语法树修改的变换模板 (Transformation Schemas) 用于修改出错位置代码生成修复补丁。该方法相比上述方法的不同是对补丁生成和验证的过程进行了优化。相比于“生成-验证”的交互式补丁生成方式，SketchFix 将程序中的候选缺陷位置使用抽象表达式进行替换。这里所谓的抽象表达式只是为了保证程序可以被正常编译和执行，并不能保证测试的通过，其具体功能在后续可以被方便替换。然后在运行测试的过程中，如果测试未通过则会在执行到的抽象表达式处生成新的修复补丁进行替换。其优点是只有被替换的表达式需要重新编译，而不需要重复编译整个项目的代码，可以有效提升补丁验证的效率。

Ghanbari 等人[29] 同样针对补丁生成过程中的效率提升问题，提出了在 JVM 字节码层面的缺陷修复方法 PraPR。该方法同样是定义了一些代码变换模板 (或变异算子) 来对代码进行修改。由于该方法在字节码层面修改代码，可以避免处理由于人工编写代码引起的代码形式不确定的问题。同时，由于目前很多编程语言 (如 Java，Kotlin，Scala 等) 都使用字节码形式，因此该方法对于修复不同语言的程序缺陷有更好的通用性。实验证明，PraPR 由于直接修改字节码可以避免反复编译代码，可以大幅度提升修复的效率。

除此之外，后续有很多针对特定缺陷的修复方法被提出。相比于通用的缺陷类型，特定的缺陷模式比较固定，更容易定义高效的修复模板。Xiong 等人[115] 提出了 ACS，针对 Java 语言的条件语句修复方法。该方法定义了条件语句的修改模式，比如增加条件、删除一个条件等。Gao 等人[26] 提出了 LeakFix 针对 C 语言的内存泄漏缺陷。该方法通过自动插入内存释放语句 (`free(pointer)`) 来实现修复，修复模板简单。然而，LeakFix 通过静态代码分析结合一系列的约束可以保证修复的正确性。

2.2.3　基于约束求解的补丁生成

根据前面的介绍我们不难发现，基于搜索和基于模板的缺陷修复方法中的补丁都是来源于现有的程序或者人工给定模板。而基于约束求解的缺陷修复方法是将当前程序中的输入输出的对应关系转换为一系列的约束条件，然后由约束求解器 (SMT Solver) 进行约束求解来构建补丁代码。其优点是可以生成种类更为丰富的修复补丁而不受已有的程序限制。但是其缺点也正是由于其生成的修复补丁种类庞杂，对补丁的准确率更难以控制。

Nguyen 等人[74] 提出的 SemFix 是较早使用约束求解技术辅助生成修复补丁的方法。该方法首先采用符号执行技术对测试程序提供的约束信息进行收集，然后将收集

到的约束转换为相应约束求解器的约束条件，并将其作为约束求解器的输入进行约束求解，最后求解的结果会被转换成源代码生成修复补丁。SemFix 采用了一种基于组件 (component-based) 的补丁生成方法。具体来讲，通过搜集程序出错位置处可用的程序元素 (如变量、常量等)，根据约束对其进行随机组合得到满足要求的代码片段。由于其依赖符号执行以及约束求解过程，在比较小的程序上效果比较好，因为该方法可以充分利用现有约束求解器的高效搜索算法求解。在比较大规模的软件程序上难以适用。

Mechtaev 等人[67] 提出的 DirectFix 采取了和 SemFix 相似的策略，同样是依赖于约束求解器进行求解。不同的地方是 DirectFix 对 SemFix 的补丁生成过程做了优化。SemFix 依赖于错误定位方法对错误的代码进行定位，然后按照错误位置的排名依次尝试生成修复。DirectFix 的优化主要是将错误定位的过程和补丁的生成过程结合到了一起，并不是严格的按照定位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，其目的是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，然后规定了一组硬条件 (Hard Condition) 和软条件 (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。

Mechtaev 等人[68] 后续又进一步提出了对 SemFix 的改进方法 Angelix。该方法同样继承了 SemFix 的基本思想，采用基于组件的程序综合方法生成补丁。但是，Angelix 采用了受限的符号执行 (Controlled Symbolic Execution)，可以克服原本的传统符号执行方法在较大的项目上不可计算的问题。同时在新的方法中通过引入天使调试定位方法对原来的定位方法做了优化：对测试函数依据运行的路径的不同进行分类，然后转化为约束条件进行求解。

Ke[45] 提出的基于代码语义搜索的补丁生成方法 SearchRepair 通过约束求解器映射变量之间的对应关系来复用人工编写的代码。该方法首先是建立了一个人工代码片段的代码库，同时为每个代码片段的输入输入建立约束关系。对于给定位置的缺陷代码，该方法通过搜集程序的约束条件然后到数据库中去搜索满足相同条件的代码片段。然后再通过约束求解器求解复用代码中变量在缺陷代码位置的映射关系，最后直接使用替换变量之后的复用代码替换掉原来的缺陷代码生成修复的补丁。该方法由于生成的修复补丁来源于人工编写的代码，补丁质量较高。然而，由于该过程中需要进行约束求解以及海量代码的匹配操作，当候选代码空间较大时会面临效率问题。

除此之外，还有针对特定类型的基于约束求解的补丁生成方法，比较代表的工作是针对 Java 语言中条件修复的 NOPOL[117]。通过修改原有的条件语句或者添加新的条件语句使得生成的补丁通过所有的测试用例。该方法首先是依赖谓词翻转的定位方法得到候选的出错位置列表。在此过程中会搜集程序执行过程中变量值信息以及缺陷条

件代码的约束 (**true**或**false**)，最后将约束条件转换为 SMT 问题使用微软的 Z3[①]约束求解器进行约束求解，将求得的结果再最终转化为源代码生成修复补丁。

2.2.4　基于统计的补丁生成

基于概率统计的补丁生成方法通过统计数据分布，根据已有补丁自动抽取修复的模板。其中比较典型的方法是基于统计学习的补丁生成和过滤方法。通过定义抽象模型，应用大量的训练数据作为输入对定义的抽象模型进行不断的参数优化调整，最终使得获得的具体模型可以拟合大部分甚至所有的训练数据，然后使用该模型对新的数据进行预测。近几年统计学习在软件的缺陷修复领域也得到了越来越多的关注和使用。

Long 和 Rinard[63] 提出了 Prophet。它使用机器学习算法训练一个排序模型，对现有的修复补丁生成工具 SPR 的修复补丁进行优先级排序，使得正确的修复补丁可以尽早地被验证。其基本思想是缺陷程序的补丁与缺陷发生位置的上下文是相关的。比如出错的位置是条件语句，出现的错误很可能会和符号的错误使用相关等。基于此，Prophet 通过搜集大量的缺陷程序以及对应的修复补丁，然后对缺陷程序和补丁的特征 (如表2.4) 进行抽取。根据提取的特征信息建立一个缺陷程序到修复补丁的一个分布概率模型。在补丁生成阶段，该模型可以对 SPR 生成的修复补丁进行排序。虽然该方法并没有新的补丁生成方法，但由于其对 SRR 补丁验证的顺序进行了优化，使得最终的修复效果有一定的提升。上文介绍的 ELIXIR[87] 中所采用的补丁排序过程与此类似。

表 2.4　Prophet 提取的部分程序特征

修复补丁的类型特征	插入控制语句，插入判断条件，替换判断条件，插入非条件语句等。
缺陷位置的错误类型	赋值或输出相关，循环，continue，break，比较或一元操作相关等。

统计模型除了被用于候选补丁的排序过程，还可以直接用来生成修复的补丁。Long 等人[61] 提出 Genesis，它是一个修复补丁的生成模型，通过分析缺陷代码和正确代码之间的关系来建立他们的映射。该方法使用代码的抽象语法树表示形式，在补丁的生成阶段会尝试将缺陷代码与模型中的代码进行代码树匹配以指导补丁代码的搜索。因此，生成的修复补丁需要在训练数据集中存在，否则将不能生成相应的修复。此外，由于其学习过程对数据量的要求比较大，该方法目前只针对一些特殊类型的缺陷，比如空指针异常等。

Tufano 等人[96] 将代码转换为文本的表示方式，然后使用自然语言中的神经机器翻译 (Neural Machine Translation) 模型对缺陷修复补丁进行预测。具体讲，该方法将修改前后的代码变成字符序列，然后使用循环神经网络 (Recurrent Neural Network) 的编码器-解码器 (Encoder-Decoder) 模型训练修改前后代码的对应关系。为了使训练的模型更

[①] https://github.com/Z3Prover/z3/wiki

具通用性，在代码的序列化时会抽象掉一些常量和变量名称，比如所有的`String`类型常量用“STRING”表示等。在最后预测生成补丁时再借助集束搜索 (Beam Search) 恢复被抽象掉的变量。

Bader 等人[7] 提出了缺陷修复方法 GetaFix，该方法应用层次聚类 (Hierarchical Clustering) 算法从历史修复中学习补丁模板。其具体过程是首先提取代码修改前后的树形表示形式，并形成代码对 $\langle before, after \rangle$，其中 $before$ 是代码被修改之前的表示而 $after$ 是修改之后的表示。将所有的历史修改提取成上述代码对形式形成代码对的集合，在聚类过程，GetaFix 使用 Anti-Unification 算法将所有的代码对进行聚类和抽象。为了使不同的修改可以用统一的模板表示，一些具体的代码会被抽象成通配符表示形式，比如两个修改是替换两个不同的变量，则通过抽象掉变量名信息可以使两个修改达到统一进而被归到一类中。因此，聚类之后相似的修改会被抽象为一个删掉了部分代码属性的代码模板。在应用其进行修复新的缺陷时，GetaFix 尝试将缺陷的代码和模板中的 $before$ 进行匹配。如果匹配成功，则使用对应的修复代码 $after$ 生成修复补丁。该方法理论上可以修复任何类型的缺陷，但是根据其算法过程，对于训练数据比较少的缺陷类型该方法会得到比较具体的修改模板，从而限制其适用范围。

White 等人[105] 提出使用神经网络 (Neural Network) 模型来搜索相似代码用于构建修复补丁代码的原材料并实现了修复工具 DeepRepair。由于程序语言和自然语言具有很大的差异性，为了可以更好的识别程序代码的特性。DeepRepair 首先基于大量的源代码数据训练了一个语言模型用来编码代码中的符号，比如变量名以及操作符等。在补丁生成的过程中，DeepRepair 同样是采用了基于模板的补丁生成方法，不同点是在实例化修复模板时它使用神经网络模型搜索相似的代码。然而，该方法在实际中表现一般，难以修复真实的复杂程序缺陷。

2.2.5 讨论与小结

基于启发是搜索的方法 (如 GenProg) 方法简单易于实现，依赖于精心设计的启发式规则。然而，由于该方法效率比较低，只能在比较粗粒度复用代码，限制了其补丁生成能力。基于模板的补丁生成方法依赖人工定义补丁的生成模板，由于模板由人工编写，通常生成的补丁质量比较高。但是，当缺陷的类型增加之后，该类技术难以实用化。基于约束求解的补丁生成方法与基于搜索的方法类似，主要差别在于该方法不需要人工设计启发式规则，而是依赖约束求解器的优化搜索算法进行求解，方法简单粗暴。但由于其依赖符号执行和约束求解过程，在大项目上延展性比较差。最后，基于统计的方法可以在一定程度上克服其他补丁生成方法的局限性，利用了已有数据的先验知识。但是该类方法引入了新的挑战：依赖大量重复的修改训练补丁生成模型。因此，该方法对常见的缺陷效果明显，对非频繁的缺陷由于缺乏训练样本数据而难以适用。

第三章　缺陷修复实证研究

3.1　引言

自动化软件缺陷修复技术虽然经过了十多年的发展，已经有很多自动化的修复方法被提出[51,52,113,115]。但是，现有的缺陷修复方法所能修复的缺陷数量依然比较少，主要针对比较常见的缺陷类型，如空指针错误等[7,61,62]。对于非频繁的缺陷，现有的修复技术依然缺乏有效的方法，只有非常少数量的缺陷可以被正确修复。根据第一章的介绍，目前主流的自动化缺陷修复方法是依赖测试的“生成-验证”修复模型，即依赖测试提供程序的规约信息尝试生成缺陷的修复补丁，最后再通过测试过滤不正确的修复。研究表明[63,82]，现实中程序所提供的测试集合通常比较弱，在可以通过测试的候选补丁中依然存在大量不正确的修复补丁。在缺陷修复基准数据集 Defects4J[44] 上的一个实验表明[64], 测试集合并不能保证自动化修复工具生成补丁的正确性。

为了提升缺陷修复方法生成补丁的质量，增加正确修复的数量，现有的很多研究主要专注于比较常见的缺陷。原因是这部分缺陷存在大量的历史数据，同时开发人员对这些类型的缺陷比较熟悉，因此可以通过人工定义模板[46,87] 或者通过从大量重复数据中学习[7,61] 来提升生成补丁的质量。但是，对于非频繁的缺陷来说，由于开发人员以及研究人员缺少足够的经验知识，历史重复的修复数量有限，导致难以生成正确的修复。因此，目前对于非频繁的缺陷而言，依然缺乏有效方法。

为了更好地了解在该模型的指导下非频繁缺陷的有效修复方法。本章进行了一个实证研究，通过分析开发者人工修复程序缺陷的过程，为非频繁缺陷的自动化修复提供有效的指导方法。具体讲，本章通过分析人工修复过程，从中总结出一些开发者所使用的方法。该过程中的一个关键挑战是：如何记录和描述人工修复的过程？自动化的缺陷修复方法通常具有比较明确的修复算法。但是，对于人工修复，为每个缺陷的修复过程编写一个修复算法，在实际中是难以实现的。所以，本章将针对人工缺陷修复的过程仔细分析，并提出一种基于“策略”的描述方式用于有效地描述人工修复的过程。通过实验结果分析发现，根据缺陷程序和未通过的测试，虽然开发者同样会生成错误的修复补丁，但是对于大部分 (82%) 的缺陷开发者可以编写正确修复补丁。表明通过分析人工修复的过程，有希望提升非频繁缺陷的自动化修复能力。为此，针对人工缺陷修复中的定位以及补丁生成过程，本章分别总结了七条策略，用于指导非频繁缺陷修复的新方法。

本章通过分析开发者人工修复缺陷的过程，总结非频繁缺陷的有效修复方法。本章接下来的组织结构如下：第3.2节介绍实证研究的过程设置；第3.3节描述实证研究的

结果以及结果的分析。第3.4节对本章内容进行讨论和总结。

3.2 实验设置

本节介绍人工修复缺陷的实验设定，主要分为两个方面：(1) 实验数据集的选择；(2) 实验的过程描述。接下来，本节对以上两方面分别详细介绍。

3.2.1 数据集

为了研究人工修复真实缺陷的过程，本文选取了在缺陷修复领域被经常使用的一个基准数据集 Defects4J[44]。本文在实证研究中使用了该数据集的 v1.0.0 版本[①]，其详细信息如表3.1所示。本研究选择此基准数据集作为实验对象有以下两方面原因。首先，该数据集目前被广泛使用，因此可以方便与目前已有的自动化修复方法进行对比分析。其次，数据集包含的项目信息具有多样性，在一定程度上具有比较好的代表性。数据集包含五个不同类型的项目，由不同的软件公司所开发，比如 Closure 项目是由谷歌公司所开发的用于 JavaScript 代码的优化编译器，而项目 Math 则是由 Apache 公司所开发的编程语言 Java 上的数学计算库。

表 3.1 Defects4J 数据集详细信息

项目名称	缺陷个数	平均代码行数 (kLoC)	平均测试个数
JFreeChart (**Chart**)	26	96	2205
Closure compiler (**Closure**)	133	90	7927
Apache commons-math (**Math**)	106	85	3602
Apache commons-lang (**Lang**)	65	22	2245
Joda-Time (**Time**)	27	28	4130
合计	**357**	**321**	**20109**

从上表数据可以看出，该数据集一共包含 357 个来自真实项目开发中的缺陷。由于本实证研究需要开发者人工修复缺陷，为了降低开发者的负担，本实验只采用部分数据。为此，本文采用随机抽样的方式从每个项目中随机选取 10 个缺陷作为实验数据集。最终一共选取了 50 个缺陷完成接下来的实证研究。为了方便描述，本文将对应缺陷根据其所对应项目名称从 1 到 10 编号，即 Chart-1 表示 Chart 项目中的第一个缺陷，而 Closure-10 则表示 Closure 项目中的第 10 个缺陷。

[①] https://github.com/rjust/defects4j/releases/tag/v1.0.0

3.2.2　实验过程

由于本实验的目标是通过分析人工修复软件缺陷的过程为自动化的修复方法提供指导。因此，为了使得人工修复的过程与自动化修复方法的环境一致，本文对人工修复的过程做以下约束。

- 执行缺陷修复的开发者对待修复的缺陷项目没有先验知识，即除了项目的测试集合所提供的程序规约信息不知道缺陷程序的完整语义。
- 在人工修复的过程中主要依赖程序源代码和对应的测试来生成修复。
- 开发者可以解析源代码中的注释以及对应的 Java 文档，同时开发者也可以上网搜索相关知识，但不能直接搜索和待修复缺陷具有直接关系的内容，比如对应的修复补丁是什么。

根据上述约束，本文作者作为实验开发者进行该实证研究过程。在人工调试过程中，开发者可以使用集成开发环境 (IDE) Eclipse 辅助调试过程。该 IDE 提供了比较全面的 Java 程序调试工具，包括运行单元测试以及设置程序断点单步执行等。和自动化修复工具类似，开发者每次给定一个缺陷程序进行调试，首先是通过运行测试分析程序可能出错的原因，定位出错代码位置。然后尝试修改出错的代码生成修复补丁。如果开发者编写的修复补丁可以通过测试，则对比开发者的修复补丁和数据集提供的正确修复补丁。如果两个补丁语义等价，则判定人工修复正确，否则判定人工修复错误。此外，为了限制实验时间，本文限制每个缺陷的修复时间不超过 5 个小时。如果开发者在 5 个小时内未能正确修复对应的缺陷，则认为该缺陷很难被修复。为了方便对实验过程数据的分析，实验的过程中，作者将每一次缺陷修复的过程完整记录，包括修复时间、修复用到的数据以及错误位置和修复补丁的推导过程[①]。最后实验结束根据每个缺陷修复的描述，采用卡片分类法 (Card Sorting) [②]将所使用的数据以及方法相同的修复实例归类，并概括描述每类方法，作为该一类缺陷所使用的“策略”。因此，本章中所使用的“策略”即对修复过程的一个概括性描述。需要注意的是，不同的策略之间是不互斥的。即在修复一个程序缺陷时，多个策略可以同时使用。

3.3　实验结果与分析

本节将介绍人工修复缺陷的结果并进行详细分析。首先，第3.3.1节介绍人工修复的总体效果，并与已有的自动化缺陷修复方法进行对比。3.3.2介绍人工修复过程中所采用的定位方法，并与相关的已有方法对比分析。3.3.3介绍人工修复过程中采用的补丁生成方法，并与相关方法对比分析。

[①] https://sites.google.com/site/d4jinpection
[②] https://en.wikipedia.org/wiki/Card_sorting

3.3.1 修复结果概述

经过实验，在选定的 50 个程序缺陷中，开发者最终正确修复 41 个，同时在另外 4 个缺陷上生成了错误的修复补丁。表3.2中列出了具体的实验数据，最后一列代表人工修复的结果。表中每个单元格内的“X/Y”表示在对应的项目上正确修复“X”个缺陷和错误修复“Y”个缺陷。表中对比的已有工具实验结果均来自已经发表的论文数据：Nopol[64]，ACS[115]，HDRepair[52]，ssFix[113]，ELIXIR[87]，JAID[15]，CapGen[104]。

表 3.2 人工修复结果与已有工具对比

项目名称	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	人工修复
Chart	1/1	0/0	2/-	1/3	3/1	0/2	2/0	7/3
Closure	-/-	-/-	1/-	0/1	-/-	0/1	-/-	8/1
Lang	0/0	1/0	2/-	1/1	1/0	0/0	1/0	10/0
Math	0/0	3/0	1/-	0/4	1/1	1/0	1/0	7/2
Time	0/0	0/0	0/-	0/1	1/0	0/0	0/0	9/0
合计	**1/1**	**4/0**	**6/-**	**2/10**	**6/2**	**1/3**	**4/0**	**41/6**

从上述实验结果可以发现，仅仅依赖已有的缺陷程序以及测试集合，开发者可以正确修复大部分 (82%) 缺陷。相比之下，目前已有的缺陷修复方法，修复缺陷的数量不超过 12%。其中一个主要的原因是现有的缺陷修复方法只能修复比较频繁的缺陷类型，而对于非频繁缺陷会直接过滤掉导致不能被修复。该实验结果表明，对于非频繁缺陷的修复方法，即使仅仅提供有限的程序规约信息 (测试)，开发者依然可以在保证比较高的修复准确率的同时大幅度增加修复的数量。因此，通过分析人工修复的过程，对于提出非频繁缺陷的修复方法具有很大的指导意义。

然而，如表3.2所示，在实验中开发者未能正确修复 9 个缺陷，其中在 6 个缺陷上生成了错误的修复补丁，在其余 3 个缺陷上没有生成任何可以通过测试的补丁。本文对上述情况进一步分析发现其主要原因是程序测试所提供的程序规约不完整导致生成的补丁过拟合到测试上或者由于开发者缺少领域特定知识而修复失败。比如缺陷 Chart-10，其缺陷位置的代码功能是替换掉传入的字符串参数中的特殊字符。根据给定的测试，开发者可以知道当传入字符串中存在字符“\”时应该替换为“&quot”。根据此，开发者生成的修复补丁为替换该字符。然而实际上，通过对比正确修复补丁发现，传入参数中不仅仅字符“\”需要被替换，还有很多其他字符也需要类似替换。因此，虽然开发者生成的修复补丁可以正确通过给定的测试，却不是真正正确的修复。再比如缺陷 Time-6，正确修复该缺陷需要插入一个新的函数调用，由于开发者对整体项目不够熟悉，导致未能生成通过测试的补丁，最终修复失败。

通过上述的实验结果，尽管根据测试提供的程序规约作为指导修复程序缺陷在某些情况下会引入不正确的修复，但是在通常情况下开发者依然可以正确修复大部分缺

陷。实验表明，根据已有的测试和源代码等数据有希望正确修复非频繁的缺陷。

3.3.2　定位方法分析

本节对人工修复过程中的定位方法进行分析和总结。同时，通过与已有的相关研究对比，分析未来研究的可能改进方向。为了方便对人工修复过程的描述，以及为自动化缺陷修复方法提供直接的指导，本文提出了使用“策略”的方式用来描述人工修复的过程。具体来讲，每条策略是对一种定位方法的概括性描述。通过分析开发者修复的过程，使用卡片分类法对其进行分类，总结成策略。对于定位过程而言，本节共总结出七条策略。下表3.3中列出了每条策略的详细信息，其中最后一列是根据卡片分类过程记录的在修复过程中使用了对应策略的缺陷列表。本节将对以下的定位策略进行详细分析。

表 3.3　人工修复采用的定位策略

序号	策略	描述	对应修复的缺陷
1	排除未被执行的语句	排除没有被未通过测试所执行的语句	所有缺陷
2	排除出错可能性低的语句	根据程序元素的功能和复杂度以及来源过滤出错可能性低的位置	L-1,2,4,7,9; M-5,10; Ch-2; Cl- 9; T-1,4,10
3	执行堆栈状态分析	根据未通过测试执行过程中抛出的堆栈信息定位	L-1,5,6; M-3,4,8; Ch-4,9; Cl-2; T-2,5,7,8,10
4	定位不期望的变量值修改	定位将输入变量值修改成为最终导致测试失败的语句	L-8; Cl-1,3,5,7,8,10; T-3,9
5	定位不期望的变量值修改	根据之前的编程习惯检查违反常规编程习惯的代码	L-6,8; Ch-1,7,8
6	谓词翻转	尝试翻转条件语句的取值使未通过测试通过	L-3; Ch-1,9; Cl-10
7	程序理解	理解程序的执行逻辑和功能	L-10; M-6,9; Ch-3; Cl-9; T-3,9

表中 L，M，Ch，Cl 和 T 分别代表项目 Lang，Math，Chart，Closure 和 Time。

定位策略 *1:* 排除未被执行的语句。本策略的基本思想非常简单：当一条程序语句没有被未通过的测试所执行时，其不可能为出错的代码位置。实际上，该条策略在人工调试的过程中是被隐式地使用。由于开发者在使用 Eclipse 调试出错程序时，通常会使用其单步调试功能，即运行未通过的单元测试函数，跟踪程序的每步执行过程以及执行结果。未被执行的语句将自动被开发者所忽略掉。因此，在调试的过程中，所有的缺陷都使用到了该条策略。目前的缺陷定位方法基本都会采用该条策略，此外，一些已有方法通过程序动态切片技术对候选的出错位置进一步优化[49,128]，以提升定位的准确率，缩小缺陷位置空间。

定位策略 *2:* 排除出错可能性低的语句。当给定一个候选出错位置列表之后，通过

分析出错位置处的代码以及程序结构和功能可以有效排除掉一些看起来不像出错的代码位置。该条策略通常比较依赖开发者的经验知识。在实际的使用中，该条策略可以针对不同粒度的程序元素，比如一条语句或者一个函数等。分析开发者的调试过程发现，在函数级别使用该条策略通常具有比较好的表现。即当给定一个候选出错函数列表，开发者通过分析函数特征排除掉一些正确的函数。此外，在调试过程中，开发者发现以下两条启发式规则可以辅助其实现高效地排除正确位置。

1. 当一个函数是测试函数本身或者来自于某个通用的库，则其出错的概率比较小。
2. 在 Java 程序中，由于面向对象编程语言的多态特性。同功能函数可能存在多个，但实际上只有一个函数实现了具体的算法功能，其他函数只是为了设置一些默认参数方便调用 (通常称为 wrapper 函数)。通常这些 wrapper 函数不太可能出错。

需要注意的是，上述排除候选出错位置的方法需要依赖程序员的开发经验。严格说，即使被排除掉的代码位置也是存在出错的可能性的，只是相比较而言，其出错的概率会比较小。实验证明，该方法在人工调试的过程中非常有效，一些缺陷可以仅仅依赖本策略以及策略1就可以实现准确的错误函数位置定位。

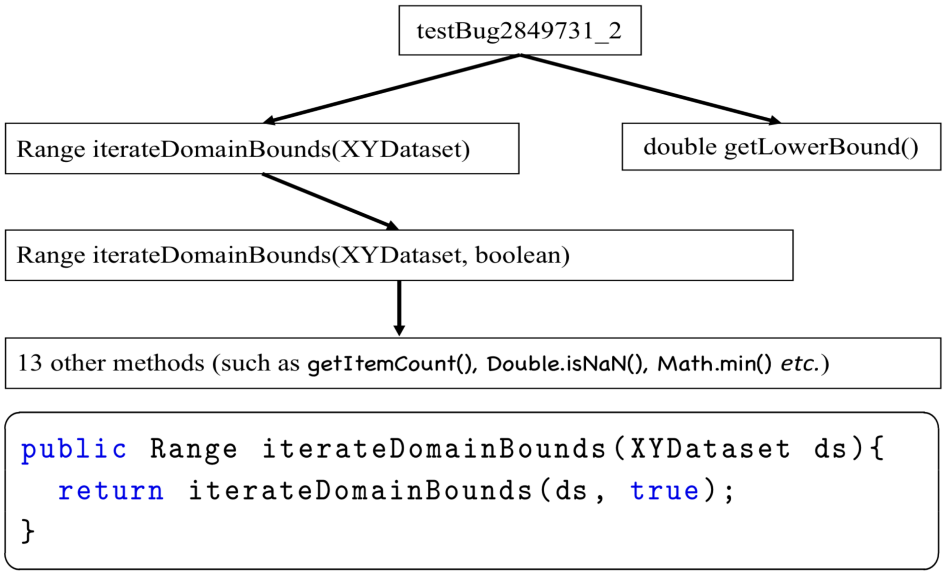


图 3.1　缺陷 Chart-2 的调用关系示意图

比如上图3.1展示的是缺陷程序 Chart-2 的函数调用关系示意图，上图中已经排除掉了未执行的函数。排除掉库函数 (比如`Double.isNaN`) 和简单的 wrapper 函数 (比如`iterateDomainBounds(XYDataset)`，图3.1下部分展示了该函数源代码)，剩下可能出错的函数则只有`iterateDomainBounds(XYDataset, boolean)`。而实际上，该函数就是出错的代码所在位置。通过分析可以发现，上述过程并不需要对程序有完整的理解同时也不需要知道程序的完整规约信息。已有缺陷预测方法[84] 与上述过程比较类似，该方法通过训练一个预测模型为每个候选函数的出错可能性打分。但是，现有的缺陷预测方法主要依赖

程序的静态特征信息，没有考虑程序运行时的特征，比如上例中介绍的函数调用关系等。因此，结合程序的动态运行时特征有希望进一步提升现有方法。

定位策略 *3:* *执行堆栈状态分析*。当运行未通过测试时，如果程序中有异常 (Exception) 抛出，则可以通过分析打印的异常信息辅助定位。通常情况下，在打印出来的异常信息中包含函数调用栈，即测试执行过程中函数的先后调用关系。根据经验，程序出错的位置通常距离运行时抛出异常的位置比较近。因此根据打印出的位置列表可以帮助开发者高效定位。在人工调试的过程中，该策略在 15 个程序缺陷上实现准确定位。比如下图3.2中所展示的是缺陷程序 Lang-1 在运行测试时触发的程序异常信息。从图中可以发现候选出错位置有七处，根据策略2我们首先可以排除掉前五个出错位置 (第 1-5 行) 以及最后一个 (第 7 行)。其中前四个位置是库函数而另外两个位置分别是 wrapper 函数 (图中展示了对应的源代码) 和测试函数本身。因此，唯一可能出错的函数位置即为候选位置中的第六个位置。实际上，上述代码的真实缺陷位置在上述候选函数中，如图中所展示的第 469 和 472 行代码。已有的一些研究已经意识到该信息的作用[106,130]，并应用其提升现有方法的效果。

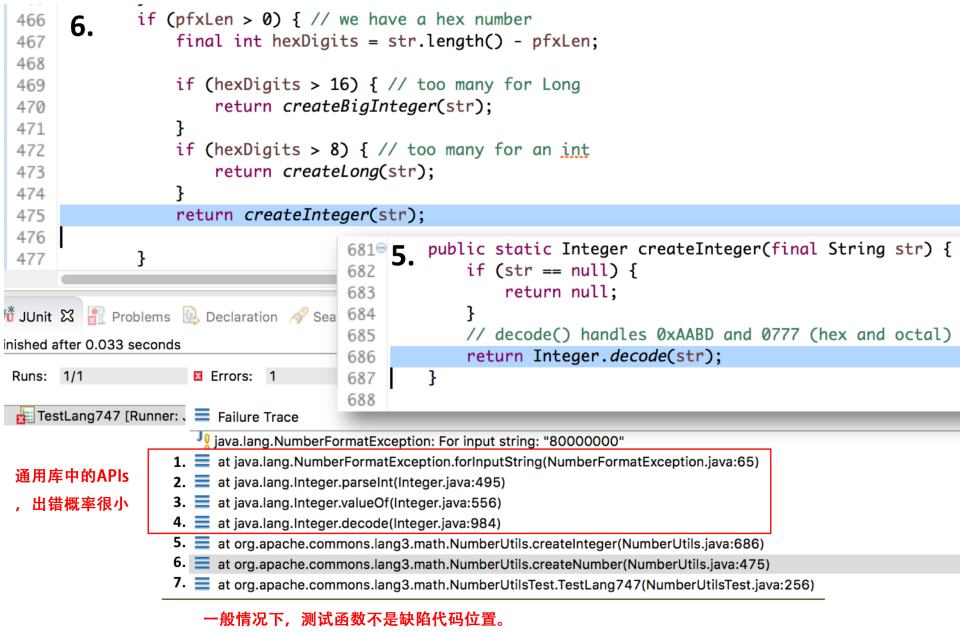


图 3.2　缺陷 Lang-1 的调用栈示意图

定位策略 *4:* *定位不期望的变量值修改*。在某些情况下，当未通过测试运行失败时会同时输出期望的值与真实的运行值。然而，实际上在测试运行的过程中测试的期望输出值可能已经产生，在随后的计算过程中由于不正确的修改导致测试失败。因此，找到程序运行过程中期望值被修改成为出错值的位置可以帮助定位出错代码。此时，修改值的语句或者其所依赖的语句有很大可能出错。

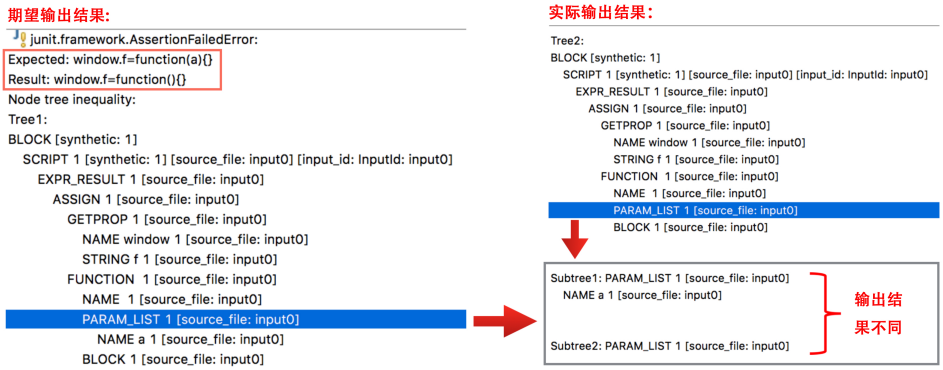


图 3.3　缺陷 Closure-1 失败测试运行结果

实际上，该策略只针对特定的测试程序会有比较好的效果，比如测试的输入数据具有比较好的结构性。在实验过程中，该策略主要被用到 Closure 项目上的相关缺陷定位，因为其测试的输入数据是 JavaScript 代码片段，具有比较好的结构信息。比如图3.3中所示为运行缺陷 Closure-1 的未通过测试的输出信息。其测试的输入是代码片段`window.f=function(a){}`。分析测试输出发现，代码片段中的参数“`a`”被删除了，而期望的输出是应该保持不变。根据其输出以及跟踪测试运行可以找到上述的输出错误是由于调用了一个函数导致的。因此，该函数或者调用该函数的代码存在错误。与此方法相似的已有研究是差异化调试，同样是通过对比程序运行时的内存状态信息实现缺陷定位。但是已有的方法依赖人工判断内存状态是否合法。所以，开发自动化的错误状态识别方法可以有效降低开发者的调试负担。

定位策略 *5:* *检查违反编程习惯的代码*。尽管程序只要符合语法约束就可以运行，但是开发者在编写代码时通常会遵循一些编程规范，以提升代码的可理解性和可维护性。比如，程序中定义的变量名称通常会反应变量的类型或者代表的语义信息。当程序中存在违反一些常规编程习惯的代码时会有比较大的出错概率。比如程序中出现一个变量“`size`”，则其取值大概率不会是负数，否则可能会触发程序出错。比如在 C 语言中如果存在一条语句“`if(a=0)`”，则该语句有比较大的可能是误把“`==`”写成了“`=`”。因此，正确识别程序中的违反正常编程习惯代码同样可以辅助缺陷的定位过程。缺陷静态检查工具的基本原理与该策略相近，比如 FindBugs[5] 根据一些预定义的不正确代码模式通过静态分析查找出错代码。

定位策略 *6:* *谓词翻转*。该策略主要针对条件语句错误，通过翻转条件语句的取值迫使程序运行到不同的条件分支中。如果该方法可以使得测试正确通过，其对应的条件语句即为出错的代码位置。该策略和已有的自动化谓词翻转方法类似，不同点在于人工调试过程中不会尝试逐一翻转所有的条件语句。只有当调试代码的分支条件不是很复杂，并且可以很容易判断出改变条件之后运行结果是否正确时，开发者才会使用

该策略。比如下面的代码 (Listing 3.1) 是缺陷程序 Lang-3 的错误代码。当测试的输入是`3.40282354e+38`时，其期望返回的是一个`Double`类型数据，而实际输出是`Float`类型变量。分析代码会发现，如果代码中第 3 行的条件为假，则第 6 行中的`Double`类型变量会作为结果返回。根据此，第 3 行中的条件有较大的概率出错。从该例子中可以发现，该过程开发者并不需要重复运行测试，只需要简单的代码分析就可以得到结果。但是已有的自动化谓词翻转方法相比人工方法具有更强的计算能力，因此适用的范围也会更广。

```
1. try{ // 当小数部分的长度不超过 7 时才可以使用 Float 类型保证精度
2.    Float f = createFloat(str);
3.    if (!(f.isInfinite() || (f.floatValue() == 0.0F && !allZeros))) return f;
4. } catch (NumberFormatException e) {}
5. try { // 当小数部分的长度不超过 16 时才可以使用 Double 类型保证精度
6.    Double d = createDouble(str);
7.    if (!(d.isInfinite() || (d.doubleValue() == 0.0D && !allZeros))) return d;
8. } catch (NumberFormatException e) {}
```

Listing 3.1　缺陷程序 Lang-3 的缺陷代码片段

定位策略 7: *程序理解*。分析上述的几条策略可以发现，大部分策略不需要完全理解程序的功能，只需要根据程序的部分特征信息即可实现较准确的定位。然而，实际上依然存在一些缺陷，通过上述的几种策略并不能定位出错代码。此时，对程序部分功能的理解是非常必要的。事实上，程序理解是一个比较复杂的人为过程，本章将其总结为开发者综合不同信息对候选出错位置的不断优化排序过程。在此过程中，开发者会尝试理解程序的实现代码、测试的执行情况、变量名字以及解读代码中的自然语言注释等。由于目前对该策略缺乏深入的理解，并且其对自动化的修复方法提供有效的指导信息有限，本章对其不做详细解释。

小结：根据实证研究中对开发者的定位方法介绍可以发现，一部分定位策略已经被现有的定位技术所采用。然而，依然存在一些策略并没有应用到已有的方法中 (比如策略2) 或者没有充分发挥其作用 (如策略4)。此外，根据表3.3，**开发者在调试过程中通常同时使用不同的策略实现准确定位**，比如定位缺陷 Lang-1 中的缺陷同时使用了策略2和策略3。因此，通过结合不同的策略有希望提升定位的准确性。

3.3.3　补丁生成方法分析

本节对人工修复过程中的补丁生成方法进行详细地分析和总结。同时，本节中会结合已有的缺陷修复方法分析人工修复过程中所采用方法的优缺点，为未来新方法提供有效的指导。与分析人工定位过程类似，本节同样采用“策略”的描述方式总结人工补丁生成方法。通过分析，本节总结了七条补丁生成策略如下表3.4所示。接下来，本

节将对这些策略逐条分析总结。

表 3.4　人工修复采用的补丁生成策略

序号	策略	描述	对应修复的缺陷
1	添加空指针检查	对象引用之前对其进行空指针检查以避免空指针异常	M-4; Ch-4; Cl-2
2	返回测试期望结果	根据测试函数中的 Assertion 直接返回期望的结果	L-2,7,9; M-3,5,10; T-1,3
3	替换一个相似变量	在作用域范围内使用具有相同类型和相似变量名的变量替换出错变量	L-6,8; Ch-7,8
4	对比测试执行路径	对比相似输入的测试执行路径信息	L-2,5
5	解析代码注释	解析自然语言描述的代码注释	M-9; Cl-1,5,7,9; T-8,9
6	模仿相似代码	对不出错代码和相似代码生成修复补丁	L-4,5; M-6,8; Ch-1,2,7,9; Cl-3,8,10; T-5,7,10
7	程序理解	理解程序的执行逻辑和功能	L-1,3,9,10; M-6,9; Ch-2,3; Cl-3,8; T-1,2,4,10

表中 L，M，Ch，Cl 和 T 分别代表项目 Lang，Math，Chart，Closure 和 Time。

补丁生成策略 *1:* 添加空指针检查。当程序引发空指针异常时 (`NullPointerException`)，一个常用的修复方法是为引发错误的变量添加空指针检查。如果异常是由变量`x`引起的，那么通常的修复方式是添加条件`x!=null`来限制对变量`x`的引用操作。比如下面的实例代码 (Listing 3.2) 中，运行测试函数时，由于在第 5 行引用了变量`r`导致了空指针异常，其修复方法是在第 4 行插入了对该变量的检查条件进行修复。事实上，上述例子中的修复并不能单纯根据该策略生成，因为其有可能是函数`getRendererForDataset(d)`出错导致返回了错误的空指针变量。开发者最后写出该修复补丁依据两点事实：(1) 应用该补丁可以避免空指针异常并通过所有测试；(2) 根据第 3 行中的代码，函数`getRendererForDataset(d)`返回`null`值应该是合理的，否者不需要对其进行检查。综合上述两点，修复补丁才得以正确生成。该方法和已有的基于人工模板的修复方法相似，比如 PAR[46] 中存在一个模板就是添加空指针检查。然而，由于需要人工定义模板，该方法适用范围有限。

```
1.   XYItemRenderer r = getRendererForDataset(d);
2.   if ( isDomainAxis ) {
3.     if ( r != null ) { result = Range.combine(result, r.findDomainBounds(d)); } ... }
4. + if ( r != null ) {
5.     Collection c = r.getAnnotations(); // 抛出 NullPointerException
6.     Iterator i = c.iterator(); ...
7. + }
```

Listing 3.2　缺陷程序 Chart-4 的修复补丁代码

补丁生成策略 *2:* 返回测试期望结果。在软件开发的过程中，通常存在一些特殊

的边界值需要进行特殊的处理。开发者有时会忘记对这部分特殊值的处理，引发程序出错。因此，修复该类型的错误，一般的修复方法是插入语句`if(c)return v;`。其中`c`是对特殊边界值的检查，当条件满足时直接返回一个特定的值`v`。因此，如果未通过的测试中给出了期望的返回值 (即`v`)，则可以套用上述模板实现修复。比如下面的示例代码 (Listing 3.3) 是 Math-3 的未通过测试，如果可以正确识别输入数组的长度`len`为 1 是边界值，则根据测试的期望结果 (即`a[0]*b[0]`) 可以很容易得到对应的修复为`if(len==1)return a[0]*b[0];`。实际上，已经存在修复方法 (ACS[115]) 采用该策略生成修复补丁。但是使用该策略的前提是要能正确识别出特殊边界值，否则容易导致修复补丁过拟合到给定的测试上而修复错误。因此，ACS 中采用了一些统计的方法避免错误修复补丁的生成。由于该策略的使用条件要求和边界值相关，因此主要适用于修复一些比较常见的缺陷。

```
1.  void testLinearCombination() {
2.      double[] a = { 1.23456789 }; double[] b = { 98765432.1 };
3.      Assert.assertEquals( a[0]*b[0], MathArrays.linearCombination(a,b), 0d);
4.  }
```

Listing 3.3　缺陷程序 Math-3 中未通过的测试函数

补丁生成策略 *3:* *替换一个相似变量*。当程序中存在多个变量具有相似的变量名称时，开发者会由于混淆变量的使用而导致程序错误。因此，一种有效的修复方法是用相似的变量替换出错变量。这一条策略通常与定位策略中的*检查违反编程习惯的代码*共同使用。由于该策略修复的缺陷通常是由于变量的错误使用导致的，所以其使用范围比较有限。在实验分析中，只有四个缺陷使用到了该策略。

补丁生成策略 *4:* *对比测试执行路径*。一般情况下，项目中会存在多个测试同时测试一个相同的功能模块。通过对比未通过和通过测试运行过程中的程序状态可以帮助分析导致测试失败的原因。比如在人工调试的过程中，测试函数的输入是一个字符串。通过对比测试运行数据发现，当传入函数中的字符串参数包含“♯”时，测试运行失败。相反的，当不包含“♯”时，测试全部通过。由此可以判断，字符串中包含字符“♯”有很大概率需要特殊处理。因此，该策略通常结合其他策略共同使用，比如*返回测试期望结果*。该策略和挖掘程序中的不变量类似[15,101]，但相比于挖掘程序中的不变量方法，该策略只对比一个正确的通过测试则可以得到边界值，但现有方法依赖很多相近的测试输入，在目前测试比较弱的情况下表现并不理想。因此，如何提升通过少量数据识别程序中的不变量对于提升补丁生成的质量有很好的帮助作用，同时可以在一定程度上克服对测试强度的依赖。

补丁生成策略 *5:* *解析代码注释*。程序开发的过程中，为了提升软件的维护，一些代码功能会附有代码注释说明用来描述代码的功能以及代码约束等，尤其是在 Java 语

言的程序中会有 Java 文档 (Javadoc) 注释。因此，通过分析代码中的注释可以帮助理解代码功能甚至直接找到引发错误的特殊值。比如修复下面的修复补丁 (Listing 3.4) 是插入了第 3 行代码对参数取值进行判断。根据其注释描述可知，参数`hours`的取值应该在-23 和 23 之间，结合其测试的期望输出 (抛出`IllegalArgumentException`异常)，生成下面的修复补丁会容易很多。目前，该策略在已有的修复方法中使用比较少，主要还是分析一些具有特定结构的数据，比如抛出异常类型[115]。其主要原因是注释通常由自然语言描述，而现有的自然语言理解技术还不能达到人工理解的程度。比如根据第 8 行的注释生成第 9 行的补丁代码，对于现在的自然语言处理方法还难以实现。

```
1.  // the offset in hours from UTC, from -23 to +23
2.  public DateTimeZone forOffsetHM(int hours,int minutes) throws IllegalArgumentException{
3. +   if ( hours < -23 || hours > 23) throw new IllegalArgumentException();
...
8.  //The DOS command shell will normalize "/" to "\", so we have to wrestle it back.
9. + filename = filename.replace("\\", "/");
```

Listing 3.4　解析代码注释指导修复

补丁生成策略 *6:* *模仿相似代码*。通常情况下，具有相似功能的程序代码也会具有相似的结构，尤其是对于同一个项目中的代码而言。因为同项目中的代码通常由同一个开发者或者同一个团队所编写，在代码风格上具有一定的相似性。此外，面向对象语言中多态函数也会导致一些相似功能函数的存在。因此，通过对比正确的相似代码修复出错代码一方面其搜索空间可以得到明显的缩小，另一方面相似代码具有较强的正确性指导作用。比如下面所列代码 (Listing 3.5) 是修复缺陷 Chart-9 的补丁，在第 2 行插入了一个新的`if`条件语句。观察代码可以发现，第 1 行中的已有代码可以为修复补丁提供指导信息：在特殊条件满足时应该设置变量`emptyRange`的值为`true`。该例子只是应用了很小一部分的相似代码，实际上在真实的修复过程中可以视情况决定相似代码的粒度。已有的一些方法也尝试复用项目中的已有代码，但是由于其方法不识别代码的相似性或者不能很好的识别相似代码的语义信息造成不正确的代码引用而生成错误补丁[51]。该策略在开发人员不理解程序的功能时具有很好的指导作用，在人工调试过程中，该策略主要被应用于修复同文件中的代码错误。实际上，对于自动化的缺陷修复方法而言，由于其具有更强的计算能力，可以进一步扩展相似代码的搜索范围，比如整个项目甚至开源项目库。

```
1.   if (endIndex < 0) emptyRange = true;
2. + if (startIndex > endIndex) emptyRange = true;
3.   if (emptyRange) { ... }
```

Listing 3.5　缺陷程序 Chart-9 的修复补丁

补丁生成策略 *7:* *程序理解*。和定位过程类似，虽然上述方法有效但并不能修复所有的缺陷。在上述方法无效的情况下，开发者会尝试理解程序的逻辑功能，并根据理解编写修复补丁。在此过程中，通过程序理解所生成的修复补丁可以进一步结合其他的策略对补丁的正确性进行判断，比如编写的补丁涉及到边界值检查，则可能需要结合开发经验、测试以及注释信息等。本章对该策略不做详细介绍，作为未来工作深入研究。

小结：和开发者使用的定位策略类似，部分补丁生成策略已经被现有的修复工具使用。但是，开发者所使用的策略对于开发新的自动化缺陷修复技术具有很好的指导作用。比如策略4通过对比多个测试执行过程中程序运行状态的差异来推断导致程序出错的边界特殊值可以作为现有方法的很好补充。此外，从上述的分析可以发现，开发者在修复的过程中会依赖其之前的修复经验，比如对于策略策略2和策略3，开发者需要准确判断出错的边界值以及是否违反变成准则等。事实上，上述过程开发者并不依赖大量的重复修改经验，即开发者可以从之前的某一次修改经验中便可提取可以复用的修改模式，其主要原因是**开发者在提取模板时可以根据其开发经验 (而不是修复经验) 辅助其提取通用的修改模板**，即判断一个具体修复中哪些信息是关键的，哪些信息是无关的。最后，策略6通过**对比已有的相似代码克服了现有方法在生成补丁时对修复模板的依赖**。上述发现对于非频繁的缺陷修复具有很好的指导意义。

3.4　讨论与小结

本节讨论在实证研究中的一些影响因素以及总结从中获得的启发，为未来开发新的缺陷修复方法提供指导，同时引出本文的缺陷修复技术。

在该实验中，影响实验结果的主要因素包括三个方面，分别是实验数据、实验人员以及实验结果的分析。首先，对于实验的数据。本章的实验从缺陷修复的 Java 程序基准数据集 Defects4J 中选取的部分数据作为人工调试的样例。但是，为了降低由于人工挑选引入的影响，实验中采取随机选取的方式从全集中挑选了 50 个缺陷进行实验。同时，为了使分析得到的结果更具有通用性，我们在不同的项目中分别选取一部分数据。此外，由于该数据集中的缺陷均来自于对应项目在真实开发场景中引入的缺陷，具有比较好的代表性。其次，本实验中的开发者由作者本人担任。由于实验人员数量比较少，对于不同的开发者其实验结果可能不具有普适性。但是，本实证研究的目标是通过分析人工调试的过程为自动化的缺陷修复方法提供指导，而不是研究开发者的普遍调试技术。因此，只要开发者所使用的方法可以进一步提升现有的修复方法，其结果就是有效的。然而，通过增加实验人员有希望提出更多有效的调试方法，本文将其作为未来工作进行系统研究。最后，在对实验结果进行分析时，本文采用了基于策略的

修复方法描述。由于修复过程的复杂性，可能会导致部分过程并不能较好的概括。为了克服上述问题，本文使用了卡片分类法对实验的结果进行了详细分析。同时，作者邀请其他开发者帮助确认对应描述的准确性，对存在疑问的策略进行修正和优化。

本章分析了开发者人工修复真实缺陷的过程，并对其使用的方法进行了总结，用于指导非频繁缺陷的修复方法。具体讲，本章针对人工修复分别总结了七条缺陷定位和补丁生成策略，为自动化方法提供直接指导。根据实证研究结果，即使没有完整的程序规约、开发者对所修复缺陷并不熟悉，大部分的程序缺陷 (82%) 依然可以被其正确修复。基于开发者调试过程中使用的策略，新的缺陷修复方法有希望实现对非频繁缺陷的修复。实际上，通过前文的分析，开发者在调试的过程中所使用的部分策略已经被已有的自动化方法或多或少地使用，并取得了一定的效果。但是依然有进一步优化的空间：个别策略在现有的自动化方法中并没有得到很好的利用，并且对于修复非频繁的缺陷具有非常重要的指导作用，比如补丁生成中的策略6。此外，无论是定位还是补丁生成过程，开发者通常会结合不同的策略以提升修复的数量和准确性，对新的自动化方法同样具有启发意义。综上，本章的重要发现主要包含以下内容：

- 结合不同的缺陷定位策略可以提升定位的准确率。
- 丰富的开发经验 (而不是修复经验) 可以指导开发者从单个的修改样例中提取高质量的修复模板。
- 程序中的相似代码可以为缺陷的修复提供指导，克服对修复模板的依赖。

上述发现将直接指导本文所提出的自动修复技术。然而，值得注意的是，本章的实证研究过程不仅仅为本文的非频繁缺陷修复提供了必要的指导，其中总结的很多人工定位以及补丁生成策略对于频繁缺陷的修复也具有很好的指导意义，为未来的缺陷修复技术指出了方向。本章研究工作已发表在中国科学 (Science China Information Sciences) 2019 上。

第四章　面向非频繁缺陷的自动修复技术

4.1　引言

根据相关研究的介绍，目前已有的缺陷修复方法主要针对常见的频繁缺陷。事实上，非频繁缺陷虽然在开发过程中重复发生的概率比较低，但是由于其通常和特定类型的项目或者上下文相关，导致其缺陷的种类比较多。因此，在所有的缺陷类型中，非频繁的缺陷依然占据很大的比例。通过人工分析发现，在缺陷定位和修复的基准数据集 Defects4J[44] 中，60% 的缺陷在历史修复中很少重复产生。所以，自动化修复程序中的非频繁缺陷对于提升自动修复技术的实用性具有非常重大的意义。

然而，自动化修复非频繁缺陷具有非常大的挑战。前文已经介绍，已有方法通过定义修复模板来指导修复的生成，但是该方法并不适用修复非频繁的缺陷。比如图4.1所示的修复补丁是 Defects4J 数据集中的一个真实程序缺陷。该缺陷是一个比较项目特定的缺陷，在其他的项目中很难找到类似的修改。为了正确修复该缺陷，4 条独立的代码语句 (2-4 行) 需要被插入到缺陷代码中。从修复的补丁可以发现，即使对于经验丰富的开发者，正确修复该缺陷也不是一个容易的过程。因此，已有的自动化修复技术对于此类缺陷的修复依然缺乏有效的方法。

```
1.    if(Math.abs(dt) <= Math.ulp(stepStart)){
2. +     interpolator.storeTime(stepStart);
3. +     System.arraycopy(y, 0, yTmp, 0, y0.length);
3. +     hNew = 0;
4. +     stepSize = 0;
5.       loop = false;
6.   } else { hNew = dt; }
```

图 4.1　非频繁缺陷修复示例

根据第三章中的实证研究分析，开发者在人工修复缺陷时，一方面可以凭借其丰富的开发经验从单个修改样例中抽象出可以复用的修改模式。另一方面，参考项目中相似的代码，开发者同样可以修复不熟悉的缺陷。比如图4.1所示的修复代码，在其对应的缺陷项目中存在相似功能的代码。通过对比相似代码，正确修复该缺陷的难度会极大降低。因此，基于实证研究的发现，本文提出面向非频繁缺陷的自动修复方法 IBFix，通过使用频繁的海量代码数据弥补重复修改的非频繁性。其包含 3 个关键技术，分别用于解决非频繁缺陷修复中的关键挑战。(1) 针对存在少量的重复修改历史的非频繁缺陷，本文提出了**基于单个样例的补丁生成技术**。该技术通过统计分析海量开源项目中的代码属性信息指导从单个的修改样例中提取可复用的补丁模板，从而克服模板提取

对大量重复修改的依赖。(2) 针对不存在重复修改的非频繁缺陷，本文提出了**基于相似代码的补丁生成技术**，通过参考相似代码生成修复补丁，从而克服修复对补丁模板的依赖。(3) 由于非频繁缺陷补丁难度增加，因此对于缺陷定位的准确率提出了更高的要求，以尽可能避免生成不正确的修复补丁。根据实证研究中得到的启发，开发者人工定位中通常综合使用不同的数据。依据此，本文提出了**基于状态划分的缺陷定位技术**，通过结合基于程序频谱和基于统计性调试的定位方法，实现不同定位技术的数据共享，进一步提升定位的准确率。

本章接下来的组织结构如下：第4.2节简要介绍自动修复方法 IBFix 的整体结构。第4.3节详细介绍基于程序状态划分的缺陷定位技术。第4.4节和第4.5节分别详细介绍基于单个样例的补丁生成技术和基于相似代码的补丁生成技术。最后，第4.6节讨论并总结本章内容。

4.2 缺陷自动修复方法 IBFix 概览

本节介绍非频繁缺陷修复方法 IBFix(Infrequent-Bug Fix)。该方法遵循传统的“生成-验证”模型的缺陷自动修复过程，(1) 通过缺陷定位方法根据测试确定程序中可能出错的位置，(2) 然后作为补丁生成的输入用于生成修复补丁，不同的补丁生成方法相互独立，相互之间可以并行生成修复补丁而相互不影响。(3) 最终，不同技术生成的补丁汇聚到一起由统一的补丁排序算法对所有补丁进行排序，并使用测试逐一验证补丁的正确性。图4.2是本文自动修复方法 IBFix 的示意图。从图中可以看出，其主要包含三个组成部分，分别是基于程序状态的定位以及基于单个修改样例的补丁生成技术和基于相似代码的补丁生成技术。

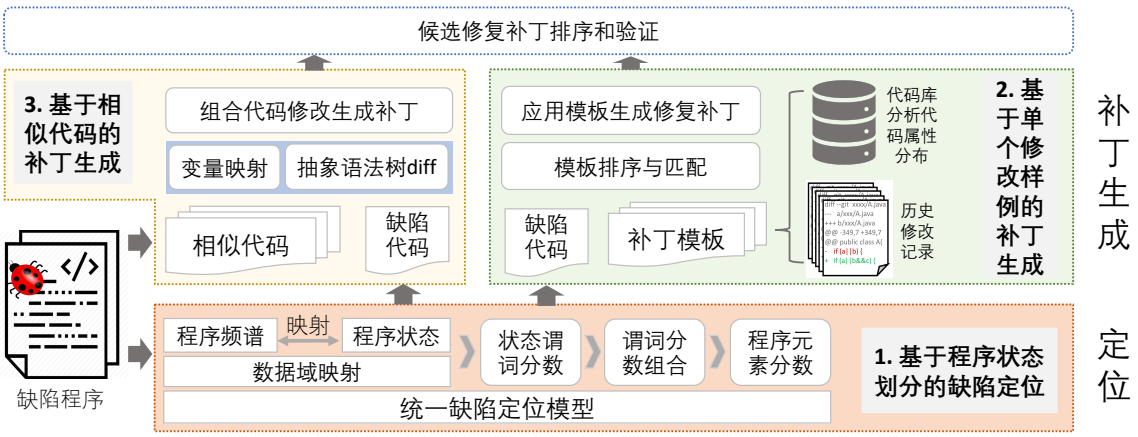


图 4.2 缺陷自动修复方法 IBFix

其中，基于程序状态的缺陷定位方法通过定义统一的缺陷定位模型，以白盒的方式

结合基于程序频谱和基于程序状态的两种定位方法。由于两种定位方法所使用的数据信息具有互补性，因此结合之后的方法可以提升缺陷定位的准确率。基于单个修改样例的补丁生成方法通过从历史的修复补丁中提取可复用修复模板来指导补丁的生成。在该过程中本文提出使用海量的开源代码中的统计信息指导模板提取过程，克服对重复修改的依赖。而基于相似代码的补丁生成方法在缺陷项目中搜索与缺陷位置相似的代码作为参考，指导缺陷修复，生成补丁。因此，该技术不依赖任何修改模板。最后，使用排序算法对不同方法生成的修复补丁进行排序，并逐一验证补丁的正确性。

4.3　基于状态划分的缺陷定位技术

研究表明，缺陷定位的准确率会直接影响缺陷修复的效果[59]。在过去的几十年里，已经有很多缺陷定位方法被提出[2,13,14,42,43,55,56,78,126]。但即使如此，目前的缺陷定位方法的准确率依然比较低。因此，研究者考虑通过结合不同的定位方法以提升定位的准确率[6,53,118,132]。然而，已有的结合技术通常将不同定位方法作为黑盒使用，利用排序学习 (Learning to Rank) 或者神经网络 (Neural Network) 模型将其结果重新组合排序，实现定位准确率的提升。该类方法存在两方面的缺点：(1) 没有充分考虑不同定位方法的特征，黑盒使用不同方法；(2) 学习模型通常需要依赖大量的训练样本数据，在不同类别的项目上效果难以保证。所以，系统性地研究不同定位方法的结合有希望进一步提升定位准确率，有助于提升对非频繁缺陷的修复能力：增加修复的数量以及提升修复准确率。

根据第二节介绍，已有的定位方法中，有两大类典型的定位方法：基于程序频谱的定位和基于程序状态的定位方法。本节首先系统性地研究两类定位方法的结合方式，并基于此提出新的方法提升定位的准确率。具体来讲，对于基于程序频谱和程序状态的定位方法，本节将分别采用其对应的具有代表性的定位方法：基于程序频谱的基础方法[2,16,19,43] (以下使用基于程序频谱的定位方法指基础方法) 和基于统计性调试 (Statistical Debugging) 的定位方法[14,17,39,56]。基于程序频谱的定位方法通过搜集测试执行过程中程序元素的覆盖信息为程序元素计算出错的概率。而基于统计性调试的定位方法是在程序中插入一些状态谓词 (Predicates)，根据谓词的覆盖以及取值情况计算每个谓词和代码缺陷的关系。因此，两种定位方法虽然存在相似性：都计算运行时的覆盖，但两种方法也存在很大的差异：一个考虑程序元素，另一个考虑状态谓词。因此，到目前为止，尚无研究探索两种定位方法的结合。

分析第三章实证研究中开发者的定位过程已知，通过结合多种定位策略可以有效提升定位的准确率。本节通过系统化的分析基于程序频谱和统计性调试定位方法的结合方式，根据其依赖信息以及计算过程的相关性，设计结合的方法。其中的挑战是：由

于两种定位方法所使用的数据状态不一致，想要在其所依赖的数据部分对其进行结合应如何实现数据的转换。针对此，本节提出了统一的定位模型 UniFL 用于结合两种定位方法，并给出了其形式化的定义。具体来讲，UniFL 可以将程序的频谱转换为程序的状态谓词覆盖，结合基于统计性调试方法中已有的状态谓词，实现对程序状态的细粒度划分。此外，为了使基于程序频谱定位方法中的程序元素出错概率公式可以用于计算状态谓词出错分数，本章提出将谓词覆盖映射到程序元素覆盖，使得出错概率的计算公式可以通用。除此之外，本节还会探索数据搜集的粒度 (如函数级别或语句级别) 以及不同谓词的结合方式对效果的影响。综上所述，在统一的模型 UniFL 的指导下，本节将在状态谓词种类、计算公式、数据搜集粒度以及谓词结合方式四个维度系统性地研究两种定位方法的结合以提升定位的准确率。

本章接下来将对两种定位方法的结合方法以及实验验证进行详细介绍。第4.3.1节介绍两种定位方法的背景，第4.3.2节介绍统一的定位模型 UniFL 并给出其形式化的定义。第4.3.3节详细分析在定位模型 UniFL 的指导下，两种定位方法的不同结合方式的定位效果。最后，基于前面的分析结果，第4.3.4节提出本文的缺陷定位技术 PredFL。

4.3.1 背景介绍

本节首先介绍基于程序频谱和基于统计性调试的缺陷定位技术的背景，通过详细分析两种定位方法的模型以及数据关联，为统一定位模型的提出铺垫基础。

4.3.1.1 基于程序频谱的定位方法

基于程序频谱的定位方法 (Spectrum-Based Fault Localization) 是目前一种主流的缺陷定位技术。由于该方法简单且效果好，在很多缺陷修复的方法中被采用[52,104,113,115,117]。其具体定位过程是，当给定一个缺陷程序以及对应的测试集合 (其中测试集合中至少存在一个未通过的测试)，典型的基于程序频谱的定位方法搜集程序在运行测试时每个元素被不同测试覆盖的情况，然后通过预定义的风险评估公式 (risk evaluation formula) 为每个程序元素计算一个分数，作为对应程序元素可能出错的概率。在实际的使用中，程序元素的粒度可以根据需要决定，可以是程序函数，也可以是程序语句等。不同的基于程序频谱的定位方法都遵循上述的定位模式，区别是使用的风险评估公式不同。

为了方便之后的描述，接下来本文将定义基于程序频谱定位方法的一般表示形式。对于给定的一个缺陷程序，我们定义 E 是该程序所包含的所有程序元素的集合，对于其中的任意元素 $e \in E$，定义以下符号表示：

- $failed(e)$：表示覆盖程序元素 e 的未通过测试的数量。
- $passed(e)$：表示覆盖程序元素 e 的通过测试的数量。
- $totalfailed$：表示测试集中所包含的未通过测试总数。

- $totalpassed$：表示测试集中所包含的通过测试总数。

根据上面的定义，给定的风险评估公式为每个程序元素计算出错概率。比如使用 Ochiai 公式[2] (被广泛使用)，程序元素 e 的出错概率计算如下所示：

$$Ochiai(e) = \frac{failed(e)}{\sqrt{totalfailed \cdot (failed(e) + passed(e))}} \tag{4.1}$$

上述的 Ochiai 公式反映了基于程序频谱定位方法的基本思想：对于一个程序元素，被更多的未通过测试覆盖、更少的通过测试覆盖，其出错的概率会增大；相反的，程序元素的出错概率减小。当给定一个风险评估公式 r，定位方法会为 E 中的每个程序元素计算一个分数，最后根据该分数将所有的程序元素进行排序。因此，定位方法的输出是一个排好序的列表，其中列表中的每个元素对应程序元素及其出错的概率分数。其一般的表示形式为：

$$SBFL^{r}_{simple}(E) = \{(e, r(e)) | e \in E\}$$

根据上面的介绍，在实际使用上述定位方法时，我们可以在不同的程序元素粒度实现定位。部分已有的研究使用更细粒度的程序元素覆盖信息实现定位。比如，使用程序语句级别的覆盖信息计算函数级别的定位[53,90]。其具体方法是在语句级别搜集覆盖信息，然后计算每条语句的出错概率，最后使用函数内所包含语句的最大分数表示对应函数的分数。实验证明，上述方法可以在一定程度上提升定位的准确率。为了表示上述的映射关系，本文定义粒度函数 $g : E \rightarrow 2^E$，表示映射程序元素与其子元素集合的关系。比如上述例子中将函数映射为语句的集合。特殊的，如果其定位的元素粒度和搜集覆盖信息的元素粒度是一致的，那么该函数会将每个程序元素映射到元素本身。基于上述定义，当给定一个粒度函数 g，基于程序频谱定位方法的更一般表示形式如下所示。

$$SBFL^{r,g}(E) = \{(e, max_{e_i \in g(e)} r(e_i)) | e \in E\}$$

4.3.1.2　基于统计性调试的定位方法

统计性调试 (Statistical Debugging) 最早由 Liblit 等人[55,56] 提出用于程序状态的远程采样，作为开发者调试的辅助信息。当给定一个缺陷程序，统计性调试的方法会在程序中插桩一些预先定义好的状态谓词 (State Predicates)。所谓谓词，即一些条件表达式用于判断程序是否满足特定的要求，比如`var>0`可以检查变量`var`是否满足大于 0。当程序部署之后，在执行的过程中会自动搜集谓词的取值情况返回给开发者。开发者根据谓词的取值状态来诊断程序是否正常运行或者出现缺陷的原因。与基于程序频谱的定位方法类似，通过计算测试的覆盖检查程序出错的原因。但与基于程序频谱的定位方法不同的是，本方法的输出排好序的谓词及其所对应的分数。

为了方便之后的描述，本文定义符号 P 表示程序中的状态谓词集合，对于任一个

谓词 $p \in P$，基于统计性调试的定位方法搜集谓词的以下覆盖信息：

- $F(p)$：表示在测试运行过程中，状态谓词 p 至少被满足一次的未通过测试个数。
- $S(p)$：表示在测试运行过程中，状态谓词 p 至少被满足一次的通过测试个数。
- $F_0(p)$：表示在测试运行过程中，状态谓词 p 被执行到的未通过测试个数。
- $S_0(p)$：表示在测试运行过程中，状态谓词 p 被执行到的通过测试个数。

和基于程序频谱定位方法类似，本方法使用一个预定义的公式计算每个状态谓词对于程序中的缺陷的重要性，因此被称为重要性评估公式 (importance evaluate formula)。该公式根据上述定义的状态谓词覆盖计算每个谓词的重要性。一个典型的重要性评估公式[56] 定义如下。

$$Importance(p) = \frac{2}{\frac{1}{Increase(p)} + \frac{1}{Sensitive(p)}} \tag{4.2}$$

其中：

$$Increase(p) = \frac{F(p)}{S(p) + F(p)} - \frac{F_0(p)}{S_0(p) + F_0(p)}$$
$$Sensitive(p) = \frac{\log(F(p))}{\log(totalfailed)}$$

在公式4.2中，谓词的重要性评估公式是两部分的调和平均数。其中，$Increase(p)$ 用来区分谓词 p 的状态在覆盖该谓词的未通过的测试执行中成立与否的分布情况，而 $Sensitive(p)$ 用来计算谓词 p 在所有未通过的测试执行中成立的比例。分析上述公式可以得出以下结论：当一个谓词被更大比例的未通过测试所覆盖并成立时，该谓词具有更大的重要性分数。相反的，如果其在越小比例的未通过测试中成立，其重要性分数会越小。因此，谓词的重要性分数反映了谓词与测试之间的关系。为了方便之后的描述，我们使用 SD_i 代表上述的公式4.2。特殊的，根据公式定义可知，当 $totalfailed = 1$ 时，根据已有的研究[56]，对应谓词重要性分数将直接设为 0，以避免除零错误。

典型的基于统计调试的方法通常预定义三类程序状态谓词[56]，其分别是 branches、scalar-pairs、returns。其具体定义如下所示。

branches 该类别的谓词表示程序中已有的所有条件表达式以及其取反表达式，如语句`if`中的条件表达式。

scalar-pairs 该类别的谓词表示将程序中赋值表达式 (包括变量初始化) 中的左值与对应位置可访问的同类型变量和常量进行对比。

returns 该类别的谓词表示将函数的返回值 (如果存在) 与常量 0 比较，在比较中使用比较符号 >、<、≤、≥、== 和 ≠。

根据上述定义，实际上不同类别的谓词在程序中的插桩位置是不同的。branches 谓词插桩在条件表达式的位置，scalar-pairs 谓词插桩在赋值表达式位置。最后，returns 谓

词插桩在函数的返回值 (即**return**语句) 位置。为了描述插桩谓词的种类，本文定义插桩函数 $s(e)$ 用来描述给定程序元素 e 时对应的插桩谓词集合。

综上所述，当给定一个重要性评估函数 i 以及插桩函数 s，对于给定的一个程序元素集合 E，统计调试定位方法返回一个状态谓词的列表，并且每个谓词会对应一个重要性分数。因此，其一般形式可表示如下所示。

$$SD^{i,s}(E) = \{(p, i(p)) | p \in s(e), e \in E\}$$

4.3.2　统一定位模型

根据第4.3.1节的介绍，基于程序频谱和统计调试的定位方法有很多共同点，比如两种方法所依赖的信息都是测试执行时的覆盖信息，且最后都是利用一个预定义的公式计算程序元素或者谓词的分数。因此，本节通过定义统一的定位模型 UniFL 实现两种定位方法的结合。

模型 UniFL 的基本思想是：将程序频谱映射为一种状态谓词。由于程序频谱定位方法使用的是程序元素的覆盖，则其覆盖情况可以等价于状态谓词“$True$”的覆盖。即程序元素被覆盖，对应的状态谓词 $True$ 同样会被覆盖，并且其取值为真恒成立。本文定义 $True^e$ 表示任一程序元素 e 所对应的谓词 $True$。因此，程序元素 e 的覆盖可以通过下面的转换关系映射成为谓词覆盖，从而实现将程序频谱定位方法中的覆盖数据映射到统计性调试方法中的状态谓词覆盖。

$$F(True^e) = F_0(True^e) = failed(e)$$

$$S(True^e) = S_0(True^e) = passed(e)$$

此外，为了使得基于程序频谱定位方法中的风险评估函数可以用来计算谓词的重要性分数，我们通过下面的公式定义对应的数据转换关系：

$$failed(p) = F(p)$$

$$passed(p) = S(p)$$

需要注意的是，在上面的转换关系中，我们使用 $F(p)$ 和 $S(p)$，而不是 $F_0(p)$ 和 $S_0(p)$。原因是如果使用后者，那么对于关联同一个程序元素的所有谓词将会有相同的分数。换句话说，我们将状态谓词作为程序中某些位置的状态划分。通过使用上述的转换关系，在接下来本文将不再区分基于频谱定位中使用的风险评估公式和基于统计调试定位方法中的重要性评估公式，并统一称作风险评估公式。

然而，根据上面的背景介绍我们知道两类定位方法的定位结果是不一样的：基于程序频谱的定位返回结果是程序元素及其分数，而基于统计性调试的定位方法返回结

果是状态谓词及其分数。为了使得最后定位结果统一，我们定义在统一的定位模型中，返回的结果为程序元素及其分数。我们定义函数 c 为同一程序元素所对应的所有不同谓词的结合函数。一种简单的形式可以是取最大值。因此，给定插桩函数 s 和风险评估函数 r 后，程序元素 e 的风险评估分数可以通过下面的公式计算。

$$c(s,r,e) = max_{p \in s(e)} r(p)$$

综上所述，当给定插桩函数 s、风险评估函数 r、粒度函数 g 以及谓词结合函数 c，我们定义统一的定位模型公式为：

$$\text{UniFL}^{s,r,g,c}(E) = \{(e, max_{e_i \in g(e)} c(s,r,e_i)) | e \in E\} \tag{4.3}$$

4.3.3 结合方式对比分析

基于第4.3.2节中提出的统一缺陷定位模型，针对模型中的四个可变维度，本节通过实验系统性地分析两种定位技术的结合方式。首先，第4.3.3.1节介绍对比分析实验中使用的数据集。其次，第4.3.3.2节介绍统一定位模型 UniFL 在实验中的具体配置。最后，第4.3.3.3节和第4.3.3.4节分别介绍结果分析的度量标准和方法实现。

4.3.3.1 数据集

为了系统性研究提出的缺陷定位模型的定位效果，本文在缺陷定位[6,53] 和缺陷修复研究[29,64,113,115] 中广泛使用的基准数据集 Defects4J[44] 上进行实验分析，表4.1中列出了该数据集的详细信息，其中包含来自 5 个项目的 357 个真实程序缺陷。

表 4.1 缺陷定位实验数据集

项目名称	缺陷个数	平均代码行数 (kLoC)	平均测试个数
JFreeChart (**Chart**)	26	96	2205
Closure compiler (**Closure**)	133	90	7927
Apache commons-math (**Math**)	106	85	3602
Apache commons-lang (**Lang**)	65	22	2245
Joda-Time (**Time**)	27	28	4130
合计	**357**	**321**	**20109**

4.3.3.2 分析维度配置

为了研究统一定位模型的效果，找到合适的结合方式。本文将针对统一定位模型中的四个可变维度 (见公式4.3) 分别探索不同结合方式的定位效果。在实验中，本文采用控制变量法研究不同维度的影响。因此，在研究某一具体维度对定位效果的影响时，其他维度将使用对应的默认配置保持不变。此外，本文在实验中在函数级别定位缺陷

代码，即定位的结果是一个候选出错函数列表。接下来，本节将对模型中的不同维度的配置分别详细介绍。

(1) 谓词种类 (s)　根据上文的介绍，将程序频谱映射为程序谓词后，我们有更丰富的状态谓词。为了研究不同谓词对定位效果的影响，本文根据谓词的种类进行分组。对于统计性调试方法中的谓词依然使用其原有的分类方法，程序频谱转换得到的谓词单独作为一组，为了描述方便将其命名为 SBFL。因此，实验中一共有四类谓词，分别是 branches、scalar-pairs、returns 和 SBFL。默认情况下，实验中使用所有类别谓词。

(2) 风险评估公式 (r)　为了比较不同的风险评估公式对定位效果的影响，本文实验中使用了五个具有代表性的基于程序频谱定位中的风险评估公式，这些公式在之前的研究中被广泛使用[79]。表4.2中列出了具体的计算公式。此外，根据第4.3.1.2节的介绍，当 $totalfailed = 1$ 时，原本的公式 SD_i 会失效。为了降低其影响，我们额外又采用了一个新的计算公式 $NewSD$。该公式是由原本的 SD_i 作微小改动而来。最后，在实验中公式 SD_i(即公式4.2) 依然被采用。特殊的，在实验过程中，默认使用的公式是 Ochiai。

表 4.2　缺陷定位风险评估公式

名称	公式	名称	公式
Ochiai[2]	$\frac{failed(p)}{\sqrt{totalfailed \cdot (failed(p)+passed(p))}}$	**Tarantula**[42]	$\frac{failed(p)/totalfailed}{failed(p)/totalfailed+failed(p)/totalpassed}$
Barinel[1]	$1-\frac{passed(p)}{passed(p)+failed(p)}$	**DStar**†[108]	$\frac{failed(p)^*}{passed(p)+(totalfailed-failed(p))}$
Op2[73]	$failed(p)-\frac{passed(p)}{totalpassed+1}$	**NewSD**‡	$\frac{2}{1/Increase(p)+\log(totalfailed)/\log(F(p)+1)}$

† 公式中的变量 $*$ 是正整数，在本实验中我们采用之前研究中的设置[79]，将其设置为 2。
‡ 公式 NewSD 中的函数 $Increase(p)$ 和公式4.2中的一致。

(3) 数据搜集粒度 (g)　由于我们的目标是在函数级别定位错误代码位置。因此，在实验中我们采用两种数据搜集的粒度，分别是函数级别和语句级别。即对于函数级别的数据搜集，粒度函数 g 会将每个函数映射到它本身。而语句级别的数据搜集是将每个函数映射到该函数所包含的所有语句的集合。实验中，默认使用语句级别数据搜集。

(4) 谓词结合方式 (c)　实验中，我们采用两种方式来结合不同的谓词，分别是取所有谓词的最大分数以及线性组合不同谓词的分数。特殊的，当使用线性组合时，我们需要约束组合的项数。在实验中，我们采用线性组合谓词 SBFL 和其他所有的谓词 (统计性调试方法中的谓词)。为了方便描述，本文将上述两种结合方法分别称为 MaxPred(即 Max score of Predicates) 和 LinPred(即 Linear score combination of Predicates)。下面我们给出两种方法的详细定义，其中 s 和 r 分别是谓词插桩函数和风险评估函数。

MaxPred 给定一个程序元素 e，所有与该元素相关的谓词的最大风险评估分数作为该元素的风险分数，即 $c(s,r,e) = max_{p \in s(e)} r(p)$。

LinPred 给定一个程序元素 e，与其相关的谓词分为两部分：来自于频谱的转换 (P_1) 和来自统计性调试方法的定义 (P_2)。其中 $P_1 \cup P_2 = s(e)$。则元素 e 的风险分数由线性公式 $c(s,r,e) = (1-\alpha) \cdot max_{p \in P_1} r(p) + \alpha \cdot max_{p \in P_2} r(p)$ 确定，其中 $\alpha \in [0, 1.0]$。另外，实验中默认的结合方式是 LinPred，并且 α 的取值为 0.5。

4.3.3.3 度量标准

在实验中，本文采用了两种度量标准分析定位的效果。

***Top-k* 召回率：** 本度量标准用来评估有多少程序缺陷可以定位在候选列表的前 k 位置。调研表明[48]，73% 的参与者认为检查前五个候选出错位置是可以被接受的。几乎所有的参与者认为检查的位置应该不超过前 10 个位置。因此，在实验中本文采用 $k \in \{1, 3, 5, 10\}$。此外，对于自动修复技术而言，其出错代码排得位置越靠前，该缺陷被正确修复的概率同样会增加，同时可以有效避免产生不正确的补丁。特殊的，当多个元素具有相同的风险分数时，本文使用已有研究中的方法[79,111,118]，用中间位置作为它们的定位结果，并对排序位置向上取整。

***EXAM* 分数：** 本度量标准是用来评估开发者在找到出错位置时，需要检查的候选位置在所有位置中所占的比例。该标准反应了定位方法的平均有效性，被很多已有的研究所采用[79,107,109]。*EXAM* 分数越低代表定位效果越好。

4.3.3.4 方法实现

为了搜集程序运行过程中的动态覆盖信息，本文实现了一个轻量级的程序静态插桩方法。该方法应用 Java 语言的一个开发工具库 JDT (Java Development Toolkit)[①]实现对 Java 源码的插桩。该库提供和丰富的源代码修改操作，可以插入、替换或者删除任意一段代码。此外，该库提供了静态解析表达式类型的能力，为谓词的插桩提供了便利 (依赖类型信息)。而且，该插桩方法不会对原程序的执行造成任何影响。其对应的源代码提供开放访问：https://github.com/xgdsmileboy/StateCoverLocator。

4.3.3.5 对比结果分析

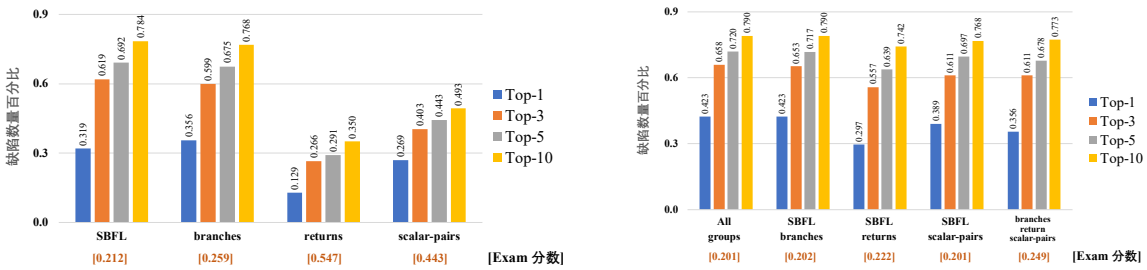
本节详细介绍实验的结果，并根据统一定位模型 UniFL 中的四个维度分别讨论。

不同种类谓词对定位效果影响 本节将对不同类型的谓词对定位效果的影响进行对比分析。首先，本文分别使用单一种类的谓词进行缺陷定位实验，其定位结果如图4.3a所

[①] https://www.eclipse.org/jdt

示。其中横坐标表示谓词的种类，纵坐标表示缺陷程序的比例。除此之外，柱状图的下面单独标出了对应的 *EXAM* 分数。在之后的实验结果展示部分，本文将使用相同的图展示方式，之后将不再赘述。

从图中可以看出，相比于其他种类的谓词，branches 在 *Top-1* 上的召回率最高，表明该谓词的有效性。此外，谓词 SBFL 在 *Top-3* 上实现了最好的效果，并且具有最小的 *EXAM* 分数。相比于其他两类谓词，即 scalar-pairs 和 returns，谓词 branches 在 *Top-1* 召回率上实现了大概 0.3-1.8 倍的效果提升。不仅如此，branches 和 SBFL 在 *EXAM* 分数上的表现都优于另外两类谓词。通过进一步分析发现，大部分的程序缺陷和条件语句相关。而 branches 和 SBFL 都包含条件谓词，因此可以比较准确地识别程序中的相关缺陷，该发现也再次确认了之前研究[92] 中的发现。进一步，为了研究不同类型的谓词之间是否存在互补性，我们采样了一些不同谓词的组合方式进行实验，图4.3b是实验的结果。



(a) 单一谓词　　(b) 谓词组合

图 4.3　不同谓词对定位效果影响

从定位结果可以看出，谓词 SBFL 和 branches 两类谓词结合之后的定位效果和使用所有谓词几乎一致，相比于其他的谓词组合方式，提升了 8.7% 到 42.4% 的 *Top-1* 召回率。通过对比图4.3a和图4.3b，结合不同谓词确实可以提升单一种类谓词的定位效果。而且，在 *Top-1* 召回率和 *EXAM* 分数两个方面都有明显的效果提升，表明不同种类的谓词具有一定的互补性。

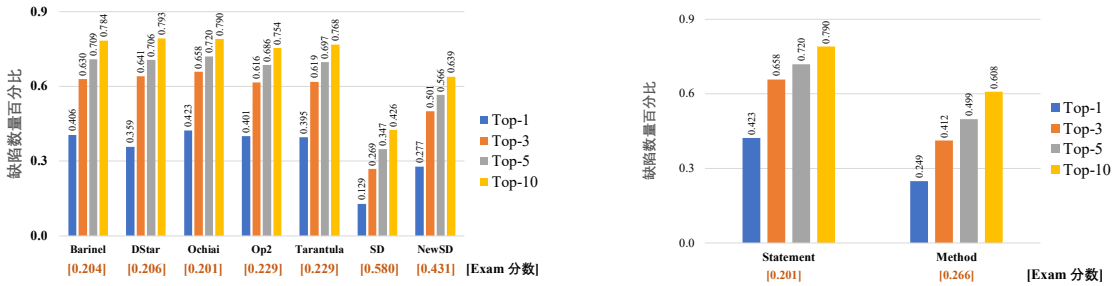
发现 1. *在所有类型的谓词中，程序中已有的条件谓词和 SBFL 谓词的定位效果相对最好，在定位中起到了非常重要的作用，尤其针对 Top-1 召回率。*

风险评估公式对定位效果影响　图4.4a展示了使用不同的风险评估公式时缺陷定位的效果。在图中，“SD”表示基于统计性调试方法中的原始公式 SD_i(公式4.2)，其余的公式定义参见表4.2。从图中可以看出，除了公式 SD_i 和 $NewSD$ 以外，其他的公式在定位效果上没有非常大的差异，该结论与之前的研究基本一致。然而，从结果中我们会发现，来自不同定位方法的风险评估公式效果表现差异很大。实验中，基于程序频谱定

位方法中的公式明显优于基于统计性调试方法中的公式。前者相比于后者的提升可以达到 227.9%。

实际上，SD_i 公式的效果表现较差的一个很重要原因是由于该方法最早是用于远程的程序状态采样，且通常情况下存在不止一个未通过的测试。然而，在本实验的场景设定中，通常只有一个未通过的测试。如第4.3.3.2节所解释的，本文对公式 SD_i 进行了微小的改动 (即 $NewSD$)，结果表明其实验结果相比于原始的 SD_i 公式在 *Top-1* 召回率上效果提升了两倍多 (26.9% vs 12.9%)。但即便如此，其定位结果依然比频谱定位中的公式效果差。上述结果表明，基于频谱定位方法中的风险评估公式在实验中优于基于统计性调试定位方法中的公式。

发现 2. *基于程序频谱定位中的风险评估公式的定位效果明显优于基于统计性调试定位方法中的公式。相比于 SD_i，前者可以实现 227.9% 的 Top-1 召回率提升。*



(a) 不同评估公式　　(b) 数据搜集粒度

图 4.4　风险评估公式和数据搜集粒度对定位影响

数据搜集粒度对定位效果影响　本节研究数据搜集的粒度对定位效果的影响。图4.4b中展示了分别使用语句 (Statement) 级别和函数 (Method) 级别搜集数据时定位结果。从图中的实验结果可以看出，在语句级别搜集数据可以实现更好的定位效果，无论是 *Top-k* ($k \in \{1,3,5,10\}$) 召回率还是 *EXAM* 分数。具体讲，其定位结果可以实现 69.9% 的 *Top-1* 召回率提升。综上分析，细粒度的数据搜集具有更好的缺陷状态区分能力。

发现 3. *相比于函数级别的数据搜集，语句级别的数据搜集可以提升 69.9% 的 Top-1 召回率。*

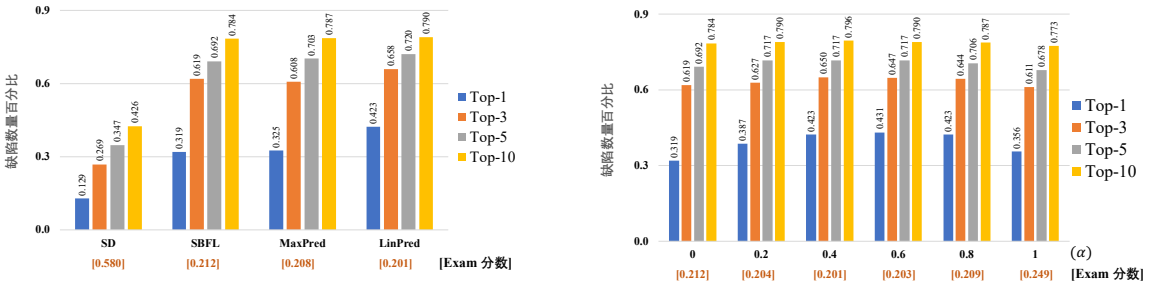
事实上，由于搜集谓词在程序运行时的覆盖信息，细粒度的数据搜集直观上是会影响定位的效率，导致更长的定位时间。为了研究数据搜集粒度对效率的影响，本文对定位的效率进一步分析。结果表明，在语句级别搜集数据相比于函数级别搜集数据会搜集大概 4.1 倍数量的谓词，导致 1.4 倍的运行时间。虽然如此，其整体的运行时间依然在三分钟之内，相比于一些其他的定位方法 (如基于变异的定位方法通常需要几个

小时)，由于状态谓词数量的增加对定位效率的影响并不大。

发现 4. *相比于函数级别的数据搜集，在语句级别搜集数据会增加 3.1 倍数量的状态谓词，进而导致增加 0.4 倍的定位时间。但整体定位时间依然不超过三分钟。*

谓词结合方式对定位效果影响　本节研究不同的谓词结合方式对定位效果的影响。如前所述，本实验中采用两种结合方式，分别是 MaxPred 和 LinPred。图4.5a展示了实验的结果。为了方便对比，图中同样展示了两种定位方法单独使用的定位结果。

从图中的结果可以发现，通过结合两种定位方法，定位的准确率都有一定的提升。但是，不同定位方法的效果提升有很大差别，尤其是针对 *Top-1* 召回率。实验结果表明，MaxPred 和原始的基于程序频谱的定位方法 (SBFL) 效果差异很小，而 LinPred 相比其他方法定位效果可以提升 32.6% 到 227.9%。事实上，造成上述的原因在之前的结果分析中已经提到，程序中已有的条件语句谓词可以有效提升定位的准确率。另一方面，通过取最大值的方式不能有效避免由于单一定位方法效果不好引入的噪音。



(a) 谓词结合方式　　(b) 线性组合参数

图 4.5　谓词结合方式及结合参数对定位影响

比如用缺陷程序 Math-72 作为例子进行解释。当使用单一的定位方法时 (基于程序频谱或者基于统计性调试)，总会有两个候选函数具有相同的风险评估分数 1.0 (基于程序频谱定位方法得到的两个函数分别是 m_1 和 m_2，而基于统计性调试方法得到的函数分别是 m_1 和 m_3)，导致真正出错的位置函数 m_1 排在候选第二位。然而，当使用线性组合的方式 (LinPred) 结合两类方法时，会将 m_1 与其他的函数区分开实现更准确的定位。相反的，使用取最大值的方式 (MaxPred) 结合时，不仅不能区分原有的 m_1 和 m_2，同时会引入 m_3，进一步降低定位的准确率。

实际上，两种独立的定位方法在理论上都存在各自的优缺点。对于基于程序频谱的定位方法，由于程序频谱映射到程序状态时对应了常量 *True*，因此，对于在同一条执行路径下的程序元素并没有区分能力，即使在运行的过程中程序的状态可能会存在差异[127]。对于基于统计性调试的定位方法，由于状态谓词由人工事先定义，只在一些特定的程序语句位置存在，比如赋值语句或者返回语句，导致在一些不存在谓词的程序

位置不能识别程序非法状态。通过结合两种定位方法，可以在一定程度上弥补单一方法的缺点，实现效果的提升。比如对于下面的缺陷代码 (Listing 4.1)，第 2 行的`return`语句存在错误，要求方法的返回值不小于 0。仅仅根据程序频谱定位，该出错代码在候选第八位置。但是统计性调试中的谓词`sumYY-sumXY*sumXY/sumXX<0`刚好捕获了该缺陷，将出错位置排到了第一位置。类似的，对于缺陷 Math-53，结合程序频谱定位方法使得出错位置由第六排到了第一。

```
1.  public double getSumSquareErrors() {
2.-     return sumYY - sumXY * sumXY / sumXX;
3.+     return Math.max(0d, sumYY - sumXY * sumXY / sumXX);
4.  }
```

Listing 4.1　缺陷程序 Math-105 的修复补丁

根据分析，两种不同的定位方法具有一定的互补性。为了进一步探究他们之间的互补性以尽可能克服各自的缺点，在实验中本文进一步研究了两种方法在线性组合模型下，系数 α 对结果的影响。图4.5b中显示了对应的实验结果。特殊的，当 $\alpha = 0$ 时，其定位结果退化为单纯使用程序频谱的定位结果。从图中可以看出，当系数在 [0.4, 0.8] 范围之内时，LinPred 方法达到最好的的定位效果。在本实验中，本文仅仅探索了两种比较简单的组合方式。尽管如此，从结果可以看出结合之后的方法始终优于单一的定位方法。

发现 5. *线性组合基于程序频谱和基于统计调试的定位方法 (LinPred) 相比于取最大值的方式 (MaxPred) 可以提升 23.2% 的 Top-1 召回率。特殊的，当线性组合的系数在 [0.4, 0.8] 范围内时，LinPred 有最好定位效果。*

4.3.4　缺陷定位技术 PredFL

根据前面的分析，通过结合两种不同的定位方法可以有效提升定位的准确率。同时，根据对统一定位模型 UniFL 四个维度的对比实验分析发现，使用线性组合两种方法可以获得最好的定位效果。基于此，本文提出了一个新的缺陷定位技术称为 PredFL。该方法采用 UniFL 模型结合基于程序频谱和基于统计性调试定位方法，并根据对比分析的结果对定位模型中的四个维度采用如下配置。

- **谓词种类**。从对比分析中可知，当使用所有种类的程序状态谓词时，缺陷定位的效果实现最佳。因此，在 PredFL 中使用所有状态谓词。
- **风险评估公式**。根据对比结果，PredFL 使用基于程序频谱中的 Ochiai 公式计算状态谓词的分数。

- **数据搜集粒度**。尽管在语句级别搜集状态谓词的覆盖信息会增加定位的时间，但是其总时间依然小于 3 分钟，而定位的准确率却可以提升将近 70%。因此，PREDFL 采用语句级别的数据搜集方式。
- **结合方式**。通过实验结果发现，使用线性组合两类状态谓词 (分别来自程序频谱和统计性调试) 的分数可以有效克服单一种类谓词的缺陷，实现定位效果提升。因此，PREDFL 使用线性组合方式结合两类谓词。

因此，当给定一个缺陷程序，通过运行测试集合 PREDFL 自动搜集程序运行时的程序元素和状态谓词的覆盖信息，使用本节定义的统一定位模型结合两种数据信息，最后返回候选出错位置列表。

4.4　基于单个样例的补丁生成技术

根据背景介绍，尽管经过了十多年的研究，已经有很多自动化缺陷修复方法被提出来[51,104,113,115,117]。但是目前的缺陷修复技术尚无有效方法修复非频繁缺陷，其中的一个关键难点是对于非频繁的缺陷，缺乏有效的补丁生成方法。目前修复效果比较好的修复工具主要是依赖修复模板的补丁生成方法，比如 PAR[46]，ELIXIR[87] 等但是对于非频繁的缺陷，一方面人工定义修复的补丁模板在实际应用中难以实用，另一方面现有的自动化修复模板学习方法难以处理非频繁缺陷的数据样本量小的问题。因此，研究如何从小样本数据中自动化学习修复补丁是提升非频繁缺陷修复的有效途径。

随着大量软件的开源，海量的软件缺陷修复历史可以被公开访问和使用，因此，一些自动化修复方法尝试从这些数据中自动挖掘缺陷修复补丁的生成补丁[7,61]，为自动化修复方法的修复模板提取提供了一种有效的途径。然而，根据修复历史提取通用的修复模板并不简单，主要包含两方面挑战：(1) 修复模板的表达形式，它将影响补丁模板的通用性以及表达的能力。所以，定义合适的模板形式是模板学习的一个重要内容。(2) 修复模板的抽象，由于代码修改来自于不同的项目，会包含一些项目特定的变量或者函数调用等，如何抽象补丁模板使得其在不同的项目上具有通用性，同时又能尽可能保证模板的正确适用场景是一个重要挑战。针对第一个挑战，已有的方法一般使用抽象语法树 (Abstract Syntax Tree) 的形式表示代码修改模板[7,61]，该方法由于主要考虑程序的语法结构，导致不能很好的表达程序的语义。对于第二个挑战，现有方法主要依赖从大量的修改样例中学习[7,61]，通过保留共有特征删除差异特征对修改模板进行抽象。因此，此类方法由于依赖大量重复的训练样本，主要修复常见的缺陷 (如空指针错误)，而不能处理非频繁缺陷。

针对上面的挑战，本节提出一种通用的代码修改自动提取方法 GENPAT，该方法克服了现有方法对大量重复修改历史的依赖，可以根据单个的代码修改样例提取代码修

改模板。根据第三章的分析，开发者在修复缺陷时可以根据丰富的开发经验 (不是修复经验) 适配历史中的某个修复到当前候选缺陷程序上。受上述启发，GenPat 的其基本思想是：**使用频繁的开源代码克服重复修改的非频繁性**。具体讲，GenPat 使用代码超图的结构表示代码修改模板，可以灵活地表达程序的语法结构以及程序语义信息。其次，GenPat 应用统计开源代码中的代码属性用来指导模板提取过程中的模板抽象过程，在准确描述模板的适用场景的同时保证修改模板的通用性。因此，GenPat 为修复非频繁的缺陷提供了有效的补丁模板自动提取方法。

4.4.1 GenPat 技术概览

本节介绍 GenPat 的具体工作流程。GenPat 包含两个阶段，分别是代码修改模板 (或代码变换) 推断阶段和代码变换应用阶段，如图4.6所示。在推断阶段，给定一对修改前后的样例代码，GenPat 会将其转换为代码超图的表示形式，通过对比代码差异提取代码修改操作，最后通过分析海量代码指导修改模板的抽象过程。在应用阶段，当给定一个代码片段，首先将其转换为代码超图表示形式，然后与已有的修改模板进行匹配。如果匹配成功，则将模板中的代码修改操作应用在候选代码上实现代码修改。在第4.4.2节，本文首先形式化定义 GenPat 中采用的代码超图表示。然后，基于该表示形式第4.4.3节和第4.4.4节分别介绍 GenPat 工作流程中的两个阶段。

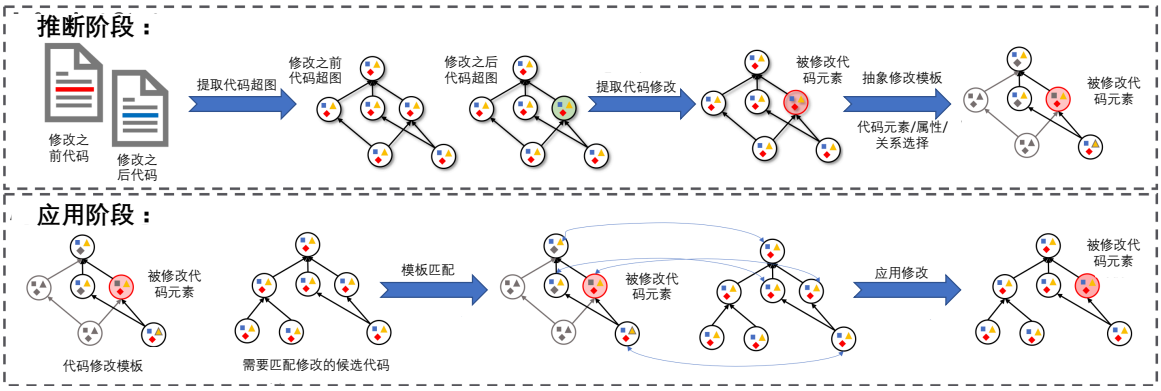


图 4.6 GenPat 技术概览

4.4.2 代码修改模板定义

在介绍代码修改模板的表示之前，我们先介绍一个代码修改的实例 (如 Listing 4.2所示)。在该示例代码中，以“-”起始的代码行表示修复代码缺陷时删除的代码，而“+”起始的代码行表示新增加的代码。其中，上部分代码修改是一个示例修改，我们需要根据该修改提取一个通用的修改模板。最后再将该模板应用在下半部分代码片段上，实现相似的修改操作。在接下来的介绍中，本文将使用该例子解释相关概念。

```
// first case for pattern generation
67.  Description createDescription(Class<?> testClass) {
68. -  return new Description(testClass.getName(), null, testClass.getAnnotations());
68. +  return new Description(testClass.getName(), testClass.getAnnotations());
69.  }
// candidate place to apply the above pattern
34.  Description createDescription(String name, Annotation... annotations) {
35. -  return new Description(name, null, annotations);
35. +  return new Description(name, annotations);
36.  }
```

Listing 4.2　代码修改实例

根据之前的介绍，现有的方法主要依赖程序的语法树结构。为了提升修改模板的表达能力 (语法和语义)，同时增加模板的表达灵活性。本文提出使用代码超图的结构表示代码的修改模板。为了方便描述，本节将给出一些名词的定义用于描述代码修改模板。对于给定一个代码片段，和已有的方法类似，本文的方法首先将代码解析成抽象语法树 (Abstract Syntax Tree) 结构进行分析，本文将语法树中的每个节点称为一个代码元素，因此每个代码元素会有一些代码属性信息，本文将其定义为该节点所对应的代码元素的属性。下面给出代码元素的形式化定义。

定义 4.1. *(代码元素 (Code Element))，一个代码元素可以表示成一个二元组 $\langle id, attrs\rangle$。其中 id 是代码元素的 ID，用于唯一标识该代码元素。$attrs$ 是关联该代码元素的代码属性的集合，而每个属性是一个二元组 $\langle name, value\rangle$，$name$ 是属性的名称、$value$ 是该属性的取值。*

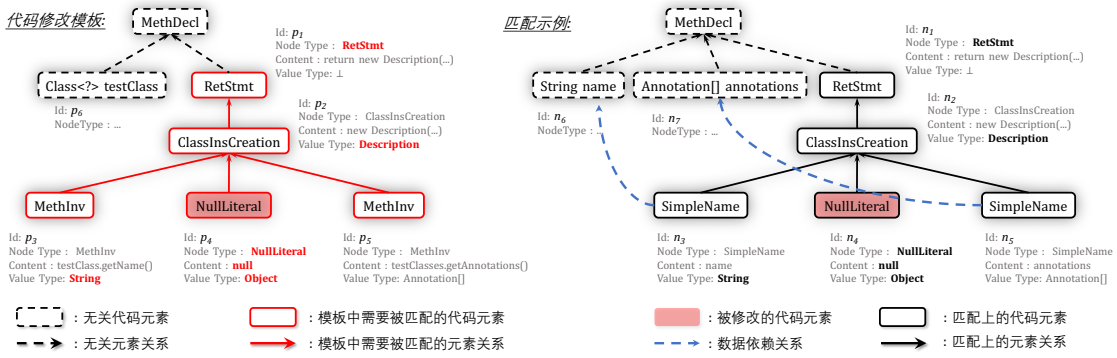


图 4.7　GenPat 应用实例

目前，GenPat 的实现主要考虑三种类型的代码属性，分别是语法树节点类型 (AST node type)、代码文本信息 (content) 和表达式的值类型信息 (static value type)。其中节点类型表示对应节点在抽象语法树中的类型，如 statement 或 variable 等；文本信息表示

代码的字符串表达形式，如`a+b`或`>=`等。最后，值类型表示某表达式在类型系统[①]中的具体取值情况，如字符串类型`String`或者整数值类型`int`等。特殊的，GenPat 使用表达式的静态分析类型。对于上述的示例代码，图4.7展示了对应代码的语法树结构 (实际上是完整的代码图结构，用于之后的介绍)。其中的节点 p_1 和 p_2 对应了代码元素，而 *RetStmt* 和 *Description* 分别是两个节点的节点类型和值类型属性。

根据上面的介绍，代码元素反映了代码中不同节点的特征，包括结构和类型等，却不能表达元素之间的关系。下面本文定义代码超图用来表示代码元素之间的关系。

定义 4.2. *(代码超图 (Code Hypergraph))，一个代码超图是一个二元组 $\langle E, R\rangle$。其中 E 是一个代码元素的集合，R 是一个边的集合，且每条边是一个二元组 $\langle rname, r\rangle$。其中 $rname$ 表示代码元素的关系名称，$r \subseteq E^k$ 表示该边所关联的 E 中的 k 个元素 (一般 $k \geq 2$)。*

理论上，代码超图中的关系 r 既可以是无向边也可以是有向边。实际中，GenPat 目前包含三种有向关系：(1) 父节点关系，用于描述在抽象语法树中一个代码元素是另一个代码元素的父节点，如图4.7中的 p_1 是 p_2 的父节点。(2) 祖先关系，用于描述在抽象语法树中一个代码元素是另一个元素的祖先节点。实际上祖先关系是父节点关系的传递闭包，如图4.7中的 p_1 是 p_3、p_4 和 p_5 的祖先节点。(3) 数据依赖关系，具体讲是过程间的变量“定义-使用”(define-use) 关系，如图4.7中的变量`annotations` (节点 n_5) 依赖函数的参数声明 (节点 n_7)。由此可见，前两种关系反应了代码的结构特征，而最后的数据依赖反应了程序的语义信息。在 GenPat 中本文忽略控制依赖来避免轻量级代码分析中的“过近似”(over-approximation) 问题[93,116]。从上面的定义可以发现，本文中所提出的代码超图不仅包含了抽象语法树的结构，同时包含代码的语义信息。此外，由于其中的代码元素关系可以根据需要扩展，因此其具有更强的表达能力。

接下来介绍代码图的匹配过程。当给定两个代码超图，GenPat 首先匹配两个超图中的代码元素，然后再匹配元素之间的关系。下面，首先给出两个代码元素的匹配定义。

定义 4.3. *(元素匹配 (Element Match))，一个代码元素 $\langle id, attrs\rangle$ 匹配另一个代码元素 $\langle id', attrs'\rangle$，当且仅当 $\forall \langle name, value\rangle \in attrs$，$\langle name, value\rangle \in attrs'$。*

根据上面的定义，本文定义两个代码超图的匹配定义如下：

定义 4.4. *(超图匹配 (Hypergraph Match))，一个代码超图 $\langle E, R\rangle$ 匹配另一个超图 $\langle E', R'\rangle$，当且仅当存在一个映射函数 $match : E \rightarrow E'$，其中 $\forall e \in E$，e 匹配 $match(e)$，并且 $\forall \langle rname, r\rangle \in R, \exists \langle rname', r'\rangle \in R', rname = rname' \wedge r \subseteq r'$。*

[①] https://en.wikipedia.org/wiki/Type_system

根据上面的超图匹配定义，本文称一个代码超图 g 比超图 g' 更抽象如果存在一个从 g 到 g' 的匹配。因此，当给定一个代码超图，通过抽象掉超图中的一些元素 (关系) 和元素属性可以使得抽象之后的超图匹配原始的超图。因此，对代码超图的抽象过程变成了对超图中的代码元素以及元素属性的选择过程。特别注意的是，本文中所提出的代码超图抽象方法可以支持图中的任何属性或者关系的抽象，但目前的 GenPat 实现只对代码元素和元素属性进行抽象，保留剩余元素之间的所有关系。

根据上面的定义，当给定一个代码片段，GenPat 首先将其映射成代码超图表示形式。然后，通过匹配来确定两段代码是否匹配，如果匹配，我们可以将一个超图中的代码修改操作应用到另一个超图所对应的代码上，实现代码修改复用。理论上，本文所介绍的方法对代码修改操作没有任何要求，可以允许任何形式的代码修改。在目前的实现中，GenPat 定义了五种代码修改操作。为了方便解释，本文定义函数 $preIDs(m)$ 表示代码修改操作 m 要求匹配的代码元素的集合。此外，本文定义函数 $preAttrs(m,id)$ 表示代码修改 m 要求匹配的代码元素 id 所对应的元素属性名称。此外，上述两个函数之间的依赖关系为：$preAttrs(m,id)$ 中的 id 应该满足 $id \in preIDs(m)$。下面列出了 GenPat 中所定义的五种代码修改操作的定义：

- $insert(id,id',i)$：插入代码元素 id' 作为 id 的第 i 个子节点。
- $insert\_str(id,str,i)$：插入文本代码 str 作为代码元素 id 的第 i 个子节点。
- $replace(id,id')$：用代码元素 id' 替换 id。
- $repalce\_str(id,str)$：用文本代码 str 替换代码元素 id。
- $delete(id,id')$：删除代码元素 id' 的子节点 id。

根据上面的定义，对于任一个代码修改操作 m，$preIDs(m)$ 返回的元素即为上述定义修改操作中的参数。比如 $preIDs(insert(id,id',i)) = \{id,id'\}$。$preAttrs(m,id)$ 对任何的 $id \in preIDs(m)$，总会返回“节点类型 (AST node type)”。因为，代码修改前后，抽象语法树中节点类型需要一致来保证程序的正确性。

最后，本文给出代码修改模板的定义。根据上面的描述，代码修改模板应该包含两部分：首先是被修改代码的样子 (代码超图)，用来匹配代码是否需要做相应修改；另一部分是代码修改操作序列。本文将上述两部分定义为代码变换 (Code Transformation)，下面是其定义描述。

定义 4.5. *(代码变换 (Code Transformation))，一个代码变换是一个二元组 $\langle g,\vec{m}\rangle$。其中，g 代表被修改代码的超图，而 $\vec{m}$ 是代码修改的序列。此外，$\forall m \in \vec{m}, id \in preIDs(m)$，以及 $attrName \in predAttrs(m,id)$，在 g 中存在一个元素 $\langle id',attrs'\rangle$，使得 $id = id' \wedge attrName \in attrs'$。*

因此，当给定一个代码超图 $g' = \langle E',R'\rangle$、一个代码变换 $t = \langle g,\vec{m}\rangle$，以及一个从超

图 g 到 g' 的匹配 $match$，在超图 g' 所对应的代码上应用变换 t 将会产生一个修改的序列 $\vec{m}[id_0 \backslash id'_0, ..., id_n \backslash id'_n]$，其中 $\langle di_0, id'_0 \rangle, ..., \langle id_n, id'_n \rangle \in match$。换句话说，在原始的代码超图中的元素使用其匹配代码超图中的对应元素进行替换，然后应用修改操作得到修改之后的代码。

4.4.3 模板离线推断

根据图4.6，代码修改模板的推断过程是一个离线过程，主要分为三个部分：(1) 提取代码超图；(2) 提取代码修改操作；(3) 抽象修改模板。

(1) 提取代码超图。当给定一对修改前后的代码，提取代码超图过程首先将代码转换为抽象语法树表示形式。然后，通过代码静态分析构建代码中的数据依赖关系，从而建立代码图模型。此外，在构建代码超图时，GenPat 同时分析代码元素的属性信息。根据之前的定义和介绍，GenPat 的目前实现中定义了三种代码属性信息，分别是节点类型、节点代码文本以及表达式的值。特殊的，语句表达式 (如`return`语句) 的默认值类型为 $\perp$。在分析代码的数据依赖时，GenPat 采用了流非敏感 (flow-insensitive) 的过程内 (intra-procedural) "定义-使用"(define-use) 分析[30,32]，即关注在同一个函数内部的代码修改。

(2) 提取代码修改。为了提取细粒度的代码修改操作，GenPat 使用 GumTree[22] 算法提取代码修改操作。然而，原始的 GumTree 算法中包含代码移动操作 (move)，GenPat 通过增加和删除代码操作实现相同的功能。

(3) 抽象修改模板。当给定一个代码图及其对应的代码修改操作时，GenPat 对修改模板进行抽象操作，删除一些与代码修改无关的代码元素以及简化一些代码元素的属性提升修改模板的通用性。所以，此部分包含两个过程，分别是代码元素选择和代码元素属性选择。

- **元素选择**。根据之前的定义，对于一个修改模板而言，其中的被修改元素应该都包含在模板中 (即 $predIDs(m)$)。因此，GenPat 首先包含该部分代码元素。基于此，GenPat 选择与上述所选元素相关联的元素作为上下文信息，选择的过程是沿着代码图中的边每次扩展直接关联的代码元素。本文定义每次扩展为"一层"扩展，通过定义扩展的层数，可以控制代码元素的上下文信息的多少。扩展层数在实际的使用中可以根据需要配置。实验中，GenPat 只扩展一层。比如图4.7中，由于代码元素 p_2 和 p_4 被修改，因此首先被选择。然后根据图中的连边扩展一层，元素 P_1、p_3 和 p_5 被包含在模板中。最后，剩余的其余代码元素将被抽象掉 (删除)。
- **属性选择**。和元素选择类似，GenPat 首先选择 $preAttrs(m, id)$ 中的属性。比如图4.7中，GenPat 首先包含元素 p_2 和 p_4 的"节点类型"属性。接下来，GenPat 将

对未被抽象掉的元素属性进行选择。具体讲，对于“代码文本”和“表达式值类型”属性，GenPat 通过分析海量代码中的对应属性的出现频率，然后计算包含对应属性的使用比例来表示该属性的通用性，其基本思想是在越多的项目中被使用的代码属性其通用性越强。基于此，当其频率超过某个阈值时，对应的属性将会被选择，否则将被抽象掉 (删除)。因此，对于某属性 $attr$，GenPat 使用下列公式计算其出现频率。GenPat 的默认阈值为 0.5%。

$$freq(attr) = \frac{|\{f|attr\ exists\ in\ file\ f\}|}{|\{all\ files\ in\ code\ corpus\}|}$$

最后，对于“节点类型”属性，保留所有语句类型 (statement) 元素的该属性，以保证代码匹配时结构的正确匹配。比如图4.7中，GenPat 保留节点 p_1 的节点类型。

4.4.4　模板在线应用

根据图4.6，应用阶段是一个在线过程。当给定一个代码修改模板和一个候选代码片段，该过程首先将代码片段转换为代码超图表示，并尝试与模板匹配。如果匹配成功，则将模板中所包含的修改操作应用到候选代码上实现相同修改操作。

(1) 模板匹配。当给定一个代码模板 (变换) $t = \langle g, \vec{m} \rangle$，以及一个候选代码片段 sp。GenPat 首先将 sp 转换为为一个代码超图 $g' = \langle E', R' \rangle$，然后尝试找到一个从 g 到 g' 的匹配 $match$。GenPat 采用下面的两个步骤搜索上述匹配。

1. 使用贪心算法尝试将 E 中的每个 e 与 E' 中的所有元素匹配。
2. 穷举所有可能的匹配组合使得模板中元素的关系在候选代码中同样满足。

比如在图4.6中，依赖元素的属性信息进行贪心匹配时，我们可以得到下面的匹配关系：

$$match(p_1) \in \{n_1\}, match(p_2) \in \{m_2\}$$
$$match(p_3) \in \{n_3\}, match(p_4) \in \{n_4\}, match(p_5) \in \{n_2, n_3, ..., n_7\}$$

接下来考虑不同的匹配组合。通过元素间的关系过滤，最后元素 p_5 的唯一匹配是 n_5，如下所示。

$$match(p_1) \in \{n_1\}, match(p_2) \in \{m_2\}$$
$$match(p_3) \in \{n_3\}, match(p_4) \in \{n_4\}, match(p_5) \in \{n_5\}$$

(1) 应用修改。根据上面的匹配结果，通过匹配元素的置换，GenPat 将模板中的代码修改应用到候选代码上。例子中对候选代码生成的修改操作为：

$$delete(n_2, n_4)$$

然而，在实际应用中，对于一个候选代码片段也可能存在多个合法匹配。GenPat 通过分析上下文的相似性来进一步对候选的匹配进行排序，以找到最佳的匹配。相似

性的计算公式如下所示：

$$node\_sim = \frac{|\{e|e \in E \wedge sameNodeType(e, match(e))\}|}{|E|}$$
$$text\_sim = \frac{1}{|E|}\Sigma_{e \in E}\frac{LCS(tokenize(e), tokenize(match(e)))}{|tokenize(e)|}$$
$$score = node\_sim + text\_sim$$

上面的公式中，函数 $sameNodeType(e, e')$ 用来判断两个代码元素 e 和 e' 是否具有相同的节点类型，函数 $tokenize(e)$ 是将元素 e 的代码文本转换为符号 (token) 序列，而函数 $LCS(s_1, s_2)$ 用来计算给定的两个符号序列 s_1 和 s_2 的最长公共子序列。最后，GenPat 使用节点类型的匹配中的节点类型相似性 ($node\_sim$) 和文本相似 ($text\_sim$) 计算最佳匹配的相似性。上述的排序算法中前者反映了代码的语法结构相似性，而后者反映了代码的语义相似性，GenPat 依据此对多个匹配进行排序。

4.5 基于相似代码的补丁生成技术

程序缺陷的自动缺陷修复过程实际上可以被视为一个修复补丁的搜索过程，而该过程的搜索空间是所有可能的修复补丁。自动化缺陷修复方法的目标是尽快搜索到正确的修复补丁。如前所述，目前的基于测试的缺陷修复方法并不能保证生成修复补丁的正确性。因此，在修复的过程中除了尽可能生成正确的修复补丁，同时需要尽可能避免生成可以通过测试但不是正确的修复补丁。因此，对于补丁的生成算法而言，合适的搜索空间约束是提升缺陷修复质量的核心。尤其是针对非频繁的缺陷而言，搜索空间的定义至关重要。第4.4章中提出了一种基于小样本数据的修复模板自动提取方法，可以有效克服非频繁缺陷的模板学习对大量重复数据的要求。然而，实际情况中对于某些特定类型的非频繁缺陷，在历史的修复中并不存在相似的代码修改操作。因此，对于该类型的缺陷而言，通过从修改样例中学习补丁模板变得不现实。

第三章中的人工修复过程分析发现，当开发者对所修复缺陷一无所知时 (对项目不熟悉同时也没有相似缺陷的修复经验)，依然可以利用项目中的代码帮助生成正确的修复补丁。而在此过程中，一个非常重要的数据是项目中相似的代码。之前的研究表明[25,33]，实际中存在很多结构相同的代码。此外，一些研究[8,66,75,85] 发现，11% 到 52% 的缺陷修复补丁可以通过复用已有的代码或者历史的修复产生。因此，针对非频繁缺陷的补丁生成难题，本文提出一种基于已有相似代码的补丁生成方法 SimFix，该方法不依赖重复的历史修改信息，通过对比缺陷代码和相似代码为修复补丁提供修改指导。同时，SimFix 还利用开源项目中的历史修复信息，通过分析历史修复对补丁生成中的代码修改操作进一步过滤。

然而，根据相似代码生成修复补丁并不容易。首先，如何有效地搜索指导补丁生成的参考的相似代码？参考代码为生成补丁直接提供原材料，质量低的参考代码会降

低生成补丁的质量，从而影响修复的效果。其次是当给定一个相似代码片段，如何根据该代码为缺陷代码生成修复补丁？由于不同代码段的功能可能存在差异以及代码中所使用的变量或函数调用等不一定相同，直接复用代码不仅不会产生正确的修复补丁，甚至会降低补丁的质量。比如，已有方法直接复用项目中的代码[37,51,123]，导致大量不正确补丁被产生。最后，由于历史修复来自不同的缺陷程序，会包含一些项目特定的信息，如何结合相似代码和历史修复也是一个重要挑战。

针对上述的难点，SimFix 采用了一种基于代码特征的相似代码搜索方法，通过编码代码的语法和语义特征实现相似代码高效搜索。另一方面，SimFix 使用一种基于代码差异的补丁生成方法，通过对比相似代码和缺陷代码在抽象语法树 (Abstract Syntax Tree) 上的差异，根据差异生成对缺陷代码的修改操作。此外，在复用相似代码生成修复补丁时，SimFix 会根据上下文信息自动匹配两段代码中的局部变量信息，从而实现正确的变量替换。为了结合相似代码和历史修复信息，本文分别定义了具体代码修改和抽象代码修改的概念用来分析相似代码和历史修复，并基于此为两者分别构造具体的和抽象的代码修改搜索空间，通过定义两种修改的转换关系实现两个空间的求交运算，达到搜索空间优化的目的。

综上所述，本文提出一种基于相似代码的补丁生成方法 SimFix，通过参考相似代码，对缺陷代码实现细粒度修改。从而可以克服缺陷修复对重复修改历史的依赖，即使无参考历史修复的情况下，依然可以有效修复非频繁的缺陷。第4.5.1节首先概要介绍 SimFix 的组成。第4.5.2节形式化定义代码修改的搜索空间。第4.5.3节介绍代码修改提取算法。最后，第4.5.4节和第4.5.5节对 SimFix 工作流程中的两个阶段分别详细介绍。

4.5.1　SimFix 技术概览

本节概述 SimFix 技术。图4.8展示了 SimFix 的方法概览。从图中可以看出，SimFix 包含两个阶段：一个离线分析阶段和一个在线修复阶段。在离线分析阶段 (第4.5.4节)，SimFix 通过分析开源项目的历史修复数据搜集缺陷修复的常用修改操作，构成一个抽象的代码修改操作空间 S^A；在线修复阶段 (第4.5.5节) 对比缺陷代码与相似代码生成一些具体的代码修改操作用于生成修复补丁，构成一个具体的代码修改操作空间 S^C。最后，通过两个搜索空间的求交 ($S^A \cap S^C$) 实现修改操作搜索空间的优化。因此，接下来本文将先定义不同的搜索空间 (第4.5.2节) 以及代码修改提取算法 (第4.5.3节)。

4.5.2　搜索空间定义

本节介绍在补丁生成过程中涉及到的搜索空间问题。为了方便之后的描述，此处首先给出代码抽象语法树 (*AST*) 的形式化定义。在本节中，我们定义一个 *AST* 是一个有序的树型结构，即每个节点的所有子节点是有顺序的。此外，我们使用 *node* 表示语

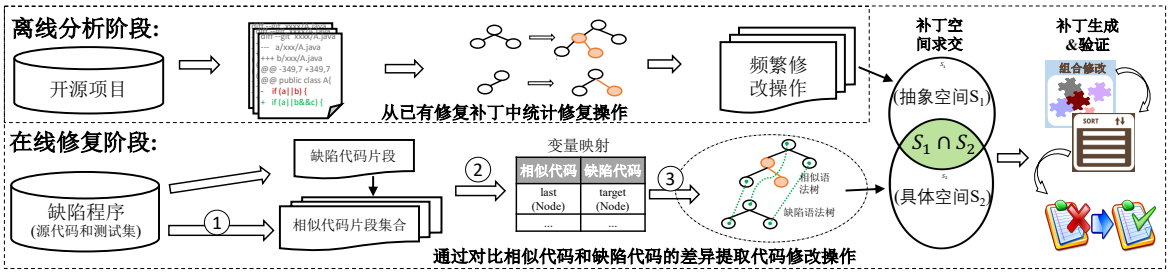


图 4.8　SimFix 方法概览

法树中的一个节点 (本章采用 Eclipse JDT 中定义的节点类型命名[①])，可以定位到语法树中的唯一位置。此外，为了方便描述，当给定一个抽象语法树 t 及其中的任一节点 n，本节定义以下函数用于实现特定的功能。

- $parent(n)$：表示节点 n 的父节点。
- $children(n)$：表示节点 n 的所有子节点列表。
- $index(n)$：表示节点 n 在其父节点的子节点列表中的索引。
- $type(n)$：表示节点 n 的类型。
- $root(t)$：表示语法树 t 的根节点。

特殊的，本节将语法树中的节点类型 ($type(n)$) 分为两类：第一类是元组类型，该类型的节点具有固定数量的子节点。比如函数调用节点 (`MethodInvocation`) 有三个子节点，分别是调用对象、函数名和参数列表。另外一类是列表类型，该类型节点的子节点数是不固定的。比如语句块节点 (`Block`) 的子节点是一个语句 (`Statement`) 列表，其中语句的数量不定。本文使用函数 $isTuple(n)$ 或者 $type(T)$ 表示节点 n 或者节点类型 T 是一个元组类型。除此之外，语法树中的一些叶子节点是有对应的值的，比如`SimpleName`类型节点的值是标志符的名称。我们使用函数 $value(n)$ 表示节点的值，当节点没有值时该函数返回 $\perp$。严格来说，上述的所有函数都依赖于节点所在的语法树。比如 $parent(n)$ 应该是 $parent(n,t)$，其中 t 是节点 n 所在的语法树。为了描述清晰，本文不列出对应参数，即用 $parent(n)$ 表示 $parent(n,t)$，其他函数类似。

根据上述语法树的定义，本文定义在抽象语法树 t 上的修改如下。

定义 4.6. *语法树 t 上的一个 (具体) 修改定义为下列中的某一个操作：*

- *$Insert(n,t',i)$: 插入一个语法树 t' 作为节点 n 的第 i 个子节点。*
- *$Replace(n,t')$: 使用语法树 t' 替换以节点 n 为根节点的子树。*

因此，修复补丁的搜索空间则是通过组合不同的修改所组成的空间。

定义 4.7. *给定一个抽象语法树 t 和一个在该语法树上的修改的集合 M，补丁搜索的 (具体) 修改空间是集合 M 的超集 2^M。*

[①] https://www.eclipse.org/jdt/core/r2.0/dom%20ast/ast.html

特殊的，本节中的语法树修改操作不包含删除操作。根据研究表明[82,95]，删除操作经常导致错误的修复补丁。本文在最后的实验验证部分会进一步讨论。根据上述的定义，当给定两个代码修改前后的片段时，通过对比两段代码可以获得讲一段代码改成另一段代码的修改序列，可以作为补丁生成的搜索空间。但是，根据上述的定义，修改操作依赖于特定的语法树。由于开源的项目中的代码修改依赖不同的语法树，为了不同项目中的修改可以统一进行统计。本文定义抽象的语法树修改操作以及其定义的抽象搜索空间。

定义 4.8. *抽象的语法树修改定义为系列中的某一个操作：*

- *INSERT*(T)*: 插入一个根节点类型是* T *的子树。*
- *REPLACE*(T_1,T_2)*: 用根节点类型是* T_1 *的子树替换根节点类型是* T_2 *的子树。*

定义 4.9. *假设* M^A *表示一个抽象的语法树修改集合，这一个抽象的搜索空间定义为该集合的超集* 2^{M^A}。

根据上述 (具体) 修改和抽象修改的定义，我们不难发现每一个具体的修改可以被映射到一个抽象的修改，本文使用 *abs*() 函数转换一个具体修改为对应的抽象修改。

$$\begin{aligned} abs(Insert(n,t,i)) &= INSERT(type(root(t))) \\ abs(Replace(n,t)) &= REPLACE(type(n),type(root(t))) \end{aligned}$$

最后，通过分析已有的历史修复补丁，我们可以得到一个常用的抽象修改集合，构成抽象的搜索空间 S^A。同时，通过对比相似的代码，我们可以得到一些在缺陷代码上的具体修改，构成具体的搜索空间 S^C。本文定义以下求交运算将两个空间求交运算实现修改空间的优化。

$$S^C \cap S^A = \{m | m \in S^C \wedge abs(m) \in S^A\}$$

4.5.3　代码修改提取

实际上，SimFix 的两个阶段中都需要提取代码的修改。离线分析阶段分析历史缺陷修复数据提取代码修改，而在线修复阶段通过对比相似代码和缺陷代码为补丁生成提取代码修改。因此，当给定一对修改前后的代码，本节算法实现代码修改的自动提取过程，使得在修改前的代码上应用对应的修改操作可以将其变成修改之后的代码。代码修改提取算法的输入是两个抽象语法树 a 和 b，其输出是一系列从 a 到 b 的修改。

本文所提出的修改提取算法和 GumTree[22] 具有相似的计算框架，我们根据需要对其进行了适当的修改和优化使得其输出符合本章中的修改空间定义。我们首先使用代码匹配算法对修改前后的两段代码进行匹配，然后根据匹配的结果提取代码修改操作。

直观上，在匹配语法树节点时，两个节点匹配当且仅当 (1) 他们有相同的节点类型、(2) 两个节点的祖先节点都正确匹配，并且 (3) 通过插入一些修改之后的语法树中的节点到修改之前的语法树中使得前两个条件成立。算法4.1展示了两个语法树的详细匹配算法。该算法通过自顶向下的匹配两个语法树中的节点。该算法的入口是 *match* 函数，其参数是两个语法树的根节点 (即 a 和 b)。如果两个节点可以匹配 (2-3 行)，我们递归地匹配其子节点。否则，我们检查是否可以通过插入一些 b 中的节点到 a 中使得两个节点匹配 (4-6 行)。当两个节点匹配之后，我们继续匹配其子节点。在该过程中，我们对节点的类型进行区分。元组类型的两个节点匹配的条件是其对应位置子节点实现匹配 (13-14 行)，非元素类型的节点，它们的子节点可以任意顺序匹配 (14-24 行)。

算法 4.1: 抽象语法树匹配算法

```
1  func match(a : ASTNode, b : ASTNode)
2      if type(a) = type(b) then
3          return {⟨a, b⟩} ∪ matchChildren(a, b)
4      else
5          for each b′ ∈ children(b) do
6              if match(a, b′) ≠ ∅ then return match(a, b′);
7          return ∅
8  func matchChildren(a : ASTNode, b : ASTNode)
9      if isTuple(a) then
10         return matchTuple(children(a), children(b))
11     else
12         return matchSet(children(a), children(b))
13 func matchTuple(as : Seq[ASTNode], bs : Seq[ASTNode])
14     return match(as.head, bs.head) ∪ matchTuple(as.tail, bs.tail)
15 func matchSet(as : Seq[ASTNode], bs : Seq[ASTNode])
16     result ← ∅
17     toMatch ← as × bs
18     while toMatch.size > 0 do
19         ⟨a, b⟩ ← a pair in toMatch
20         result = result ∪ match(a, b)
21         if match(a, b) is not empty then
22             remove all pairs containing a or b from toMatch
23             toMatch ← toMatch − ⟨a, b⟩
24         return result
```

根据上述的语法树匹配算法得到的结果，本文定义代码修改的提取方法。本文定义了四种生成代码修改的条件，当满足特定的条件时，生成相应的代码修改操作。在描述中，本文使用 $a \leftrightarrow b$ 表示节点 a 和节点 b 已经匹配，其中 a 来自于修改前的代码/缺陷代码 (简称原代码)，而 b 来自修改之后的代码/相似代码 (简称目标代码)。此外，

本文使用 $tree(a)$ 表示根节点为 a 的子树。

条件: $a \leftrightarrow b \wedge type(a) = type(b) \wedge value(a) \neq value(b)$

代码修改: $Replace(a, tree(b))$

即当两个叶子节点匹配，但是值不同时，使用目标节点 b 替换原节点 a。

条件: $parent(a) \leftrightarrow parent(b) \wedge isTuple(parent(a)) \wedge index(a) = index(b) \wedge a$ 没有匹配任何节点

代码修改: $Replace(a, tree(b))$

当两个元组类型节点匹配，但是存在子节点不匹配时，使用目标子节点替换原来的子节点。

条件: $parent(a) \leftrightarrow parent(b) \wedge \neg isTuple(parent(a)) \wedge a \leftrightarrow b$

代码修改: $\{Insert(parent(a), b', i) \mid b'$ 是 b 的兄弟节点，且没有匹配节点 $\wedge i = index(b') - index(b) + index(a)\}$

当两个非元组类型节点匹配时，首先找到其匹配的一对子节点。根据相对位置，将目标代码中的未匹配节点插入到原代码中。

条件: $a \leftrightarrow b \wedge parent(a) \leftrightarrow b' \wedge b' \neq parent(b)$

代码修改: $Replace(parent(a), t)$, 其中 t 是对 $tree(b')$ 应用修改 $Replace(b, tree(a))$ 得到的新节点。

当原代码中的节点和目标代码中的一个 (语法树中) 更深层次的节点匹配时，将原代码中的缺失的代码部分 (目标语法树中中间层未匹配的代码) 插入到原代码中。

4.5.4　离线分析阶段

在离线分析阶段，本文通过分析开源项目的历史修复数据统计常用于修复缺陷的抽象修改操作。因此，该阶段的输入是一个历史修复补丁的集合，集合中的每个元素是一个二元组，分别是修改之前和修改之后的代码。对于每一个修改元组，使用第4.5.3节中的算法分别提取具体的代码修改操作，然后使用 $abs()$ 函数 (见第4.5.2节) 将其转换为抽象的代码修改操作。最后，我们根据给定的阈值 k 选择出现比较频繁的抽象修改操作作为历史修复补丁确定的抽象搜索空间 S_1。在实验中，根据 Pareto 原则[9] 本文设置阈值 k 使得抽象搜索空间中的修改可以覆盖 80% 的历史补丁。此外，该阶段获得的抽象修改对于修复不同的缺陷是可以复用了，不需要重复分析，因此该数据可以离线分析。

本文从开源项目网站 GitHub 上下载了三个比较大的项目：Ant①、Groovy②和 Hadoop③。

①https://github.com/apache/ant
②https://github.com/apache/groovy
③https://github.com/apache/hadoop

表4.3中列出了上述三个项目的详细数据。参考已有的研究[40,76]，本文使用关键字对项目提交历史的说明信息进行过滤，比如"fix" 和“repair”等。本文只采用包含相关关键字的提交历史作为缺陷修复修改。最后，我们再对选取的修复历史进行人工过滤，排除掉无关缺陷修复的修改，比如“fix doc”和“repair format”等。对于每个修改历史，我们分别提取修改之前和修改之后的代码作为一组。

表 4.3　提取抽象修改操作的开源项目

项目名称	代码行数 (kLoC)	历史提交数	过滤后的修复历史数
Ant	138	13,731	270
Groovy	181	14,532	402
Hadoop	997	17,539	253
合计	1,316	45,802	925

在离线分析阶段，通过分析开源项目的修复历史我们获得了一个由频繁的修改操作构成的抽象搜索空间。表4.4中列出了不同修改操作在历史修改中所占的比例情况。从表中的数据可以发现，前三种最常使用的修复操作分别是 (1) 插入`if`条件语句、(2) 替换函数调用和 (3) 插入函数调用，该统计结果和已有的研究发现一致[12,46,65,104]。比如，Martinez 和 Monperrus[65] 发现缺陷修复中前五种最常用的插入和替换修改操作也包含在本文统计的结果中，类似于插入函数调用或者条件语句等。列表中的抽象修改操作构成了一个相对较小的搜索空间。实际上，根据抽象语法树的节点类型 (大概一共 40 种), 完整的搜索空间大概在 1640 种修改操作 (40×40 种替换操作和 40 种插入操作)。然而，通过分析已有的修复历史，本文一共搜集了 16 种常用修改操作 (见表)，使得补丁的抽象修改搜索空间实现了大约 102.5 倍的空间优化。

表 4.4　历史修改中抽象修改操作频率

替换操作 (Replacement)		插入操作 (Insertion)	
(MI, MI)	21.62%	**IFSTMT**	22.05%
(INFIXE, INFIXE)	8.54%	**ESTMT(MI)**	14.70%
(NAME, NAME)	5.40%	**VDSTMT**	11.67%
(NAME, MI)	3.89%	**ESTMT(A)**	5.73%
(INFOP, INFOP)	2.05%	**TRYSTMT**	2.49%
(TYPE, TYPE)	1.84%	**RETSTMT**	1.51%
(SLIT, SLIT)	1.84%	**TRSTMT**	1.19%
(BLIT, BLIT)	1.18%		
(NULIT, NAME)	1.08%		

MI : `MethodInvocation`　　IFSTMT : `IfStatement`
NAME : `Name`　　ESTMT(MI) : `ExpressionStatement(MethodInvocation)`
INFIXE : `InfixExpression`　　ESTMT(A) : `ExpressionStatement(Assignment)`
TYPE : `Type`　　VDSTMT : `VariableDeclarationStatement`
SLIT : `StringLiteral`　　INFOP : `InfixExpression.Operator`
NULIT : `NumberLiteral`　　RETSTMT : `ReturnStatement`
BLIT : `BooleanLiteral`　　TRYSTMT : `TryStatement`
TRSTMT : `ThrowStatement`

此外，为了分析所选取的项目是否数量足够具有代表性，本文进一步统计了当使

用不同数量的项目时，代码修改操作的分布情况，并与全集的统计结果进行对比。我们枚举了所使用的数据集中项目的所有可能组合，最终结果表明不同的项目组合所得到的修改分布情况和使用全集得到的结果一致。此外，为了更清楚的对比不同数量的修复补丁所确定的抽象修改空间，本文进一步对比不同的组合中频繁的代码修改操作所占的比例情况。图4.9是统计对比结果。从图中可以发现，当使用不同的修复补丁集合时 (不同项目) 各种频繁修改的比例非常接近。该结果说明本文所使用的历史补丁数据集足够代表性。不仅如此，当使用更少的修复历史来构造本文的抽象修改空间时同样具有很强的代表性。因此，本文将上述分析得到的结果作为代码修复的代表性修改操作集合，并最终使用其对修复补丁进行过滤。

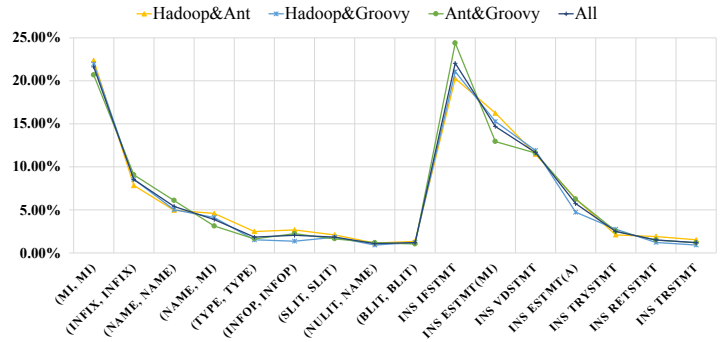


图 4.9　不同开源项目组合中抽象修改的比例

4.5.5　在线修复阶段

本节介绍 SimFix 的在线修复过程，从图4.8中可以看出，该部分包含三个过程，分别是相似代码搜索、变量映射和代码修改提取。下面本文将结合一个具体的示例修复介绍上述过程。Listing 4.3中显示的是 Defects4J 中缺陷 Closure-57 的修复补丁，其中的第 2 行代码中的条件被第 1 行中的条件替换。而 Listing 4.4中显示的代码是来自同一个项目中的相似代码，根据该相似代码 SimFix 成功修复了该缺陷。

```
1 + if(target != null && target.getType() == Token.STRING){
2 - if(target != null){
3     className = target.getString(); }
```

Listing 4.3　缺陷 Closure-57 的修复补丁

```
1   if(last != null && last.getType() == Token.STRING){
2       String propName = last.getString();
3       return (propName.equals(methodName)); }
```

Listing 4.4　修复缺陷 Closure-57 的相似

4.5.5.1 相似代码搜索

根据定位的结果，SimFix 首先将错误的代码位置扩展成一个代码片段然后搜索一组与该代码相似的代码进行对比以生成修改操作。

首先我们定义代码片段 (code snippet)。当指定代码中的行号 x 和 y，本文定义 $[x,y]$ 中的代码片段指在代码行 x 和 y 之间的所有语句 (Statement) 所构成的语句块。比如 Listing 4.4中的代码行 1 和 3 之间的代码片段包含完整的`if`条件语句，而对于行 1 和 2 之前的代码片段则只包含第 2 行中的初始化语句，因为完整的`if`语句没有在该范围之内。

为了识别相似的代码，SimFix 定义了三种相似性衡量方法用来计算代码片段之间的相似性，最终用三者的相似性总和作为两段代码的最终相似性分数。

结构相似性。结构相似性指的是两段代码所对应的抽象语法树的结构相似程度。本方法与 DECKARD[38] 类似，SimFix 为每个代码片段提取一个特征向量，而向量中的每一个维度代表语法树节点的一个特征。比如，其中的一个特征是代码片段中包含`for`语句的个数，或者代码片段中包含的算术操作符 $(+,-,*,/,\%)$ 的个数。然后，SimFix 使用余弦公式计算两个特征向量的相似性。

变量名相似性。变量名相似性用来衡量两段代码中所使用变量的相似程度。SimFix 首先根据驼峰命名法拆分代码中的变量名将其变成独立的英文单词。比如，变量`className`将被分解为单词`class`和`name`。对于两个代码片段，SimFix 使用 Dice 系数[①]计算两个单词集合的相似度。

函数名相似性。函数名相似性是用来衡量两段代码中所使用的函数调用相似性。对于函数名的相似性计算方法与变量名相似性的计算方法类似，不同是此处考虑的是函数的名称而不是变量的名称。

根据上面的代码片段定义，我们可以定位缺陷代码片段。当给定一个候选出错代码行，SimFix 会将其扩展成为包含 N 行代码的片段。假设出错代码行为 n，则 SimFix 将均等地向两侧扩展代码形成代码片段 $[n-N/2, n+N/2-1]$，作为出错代码片段。

接下来，根据出错代码片段 SimFix 在当前的项目中搜索与此相似的代码片段。其过程是使用一个大小为 N 的滑动窗口对所有代码文件中的代码进行逐一对比，通过提取滑动窗口内的代码特征使用上述三种衡量方法计算与出错代码片段的相似性。最后，根据相似性大小，对候选的相似代码进行排序。由于候选相似代码可能会很多，在实验中本文选择前 100 个最相似的代码作为参考代码用于补丁生成。此外，实验中我们设定代码行数 N 大小为 10。

[①] https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient

4.5.5.2　变量映射

通过上述过程，SimFix 可以得到一组与缺陷位置代码相似的候选代码。然而，由于代码片段来自于不同的功能函数，里面可能分别使用了不同的局部变量，直接复用相似的代码可能会导致编译错误。比如上面的 Listing 4.3和 4.4中的两段代码，虽然代码结构和功能类似，但是使用的变量却不尽相同。因此，接下来本文介绍两段代码中的变量映射关系。和搜索相似代码的过程类似，SimFix 通过计算变量之间的相似性来映射两段代码中的变量映射。具体说，SimFix 使用通过三种信息计算变量的相似性，分别是变量使用、变量类型以及变量名称。

用法相似性。用法相似性用来识别变量在代码片段中的使用情况，SimFix 应用变量在代码中的使用序列表示其所处的上下文环境。当给定一个代码片段，我们对抽象语法树后序遍历，输出变量在语法树中的父节点类型。因此，对于变量的使用是一个树节点类型的列表。比如对于示例代码 Listing 4.3和 4.4中的两个变量`target`和`last`的使用序列如下所示。

`target`: [INFIX_EXP, METHOD_EXP]

`last`: [INFIX_EXP, METHOD_EXP, METHOD_EXP]

然后，SimFix 计算两个变量的用法相似性通过计算其对应序列的最长公共子序列长度与总长度的比值作为其相似性分数。

类型相似性。当复用相似代码中的代码时，SimFix 需要首先替换其中对应的变量。因此，替换的变量应该保证类型是正确的：(1) 当一个变量被用作赋值表达式的左值 (left-value) 时，我们需要保证替换的变量的值类型是被替换变量的父类 (可以相同)；(2) 当一个变量被用作右值 (right-value) 时，其替换变量的值类型应该被替换变量值类型的子类 (可以相同)。理想状态下，上述两个条件应该必须满足。然而，由于在目前阶段我们还不知道具体复用哪一部分代码。因此，SimFix 使用类型相似性来衡量类型的兼容性。本文定义类型相似性是一个二值分数：如果两个变量的类型兼容，则其类型相似性分数为 1，否则为 0。

特殊的，在 Java 语言中，基本数据类型 (primitive type) 之间没有明确的子类关系。给定两个基本类型 T 和 T'，我们认为 T 是 T' 的子类型，当且仅当任何 T 类型的变量都可以不损失精度地转换为 T' 类型。

名称相似性 。名称的相似性计算和代码搜索中的变量名相似性计算方法类似，不同的是此处仅考虑两个变量的相似性。

最后，SimFix 通过上述三种相似性的加权求和作为两个变量的最终相似性得分。当计算任意一对变量 (一个来自缺陷代码一个来自相似代码) 之间的相似性之后，SimFix 通过贪心选取相似性分数最高变量对作为变量匹配。比如，对于 Listing 4.3和 4.4中的

两段代码，变量last和target，以及变量propName和className匹配。然后，根据变量的匹配关系，SimFix 会使用缺陷代码中的变量替换相似代码中的对应变量。之后，缺陷代码可以直接复用相似代码中的代码元素。

4.5.5.3 代码修改提取和过滤

根据上述步骤，相似代码中的变量已经替换成了缺陷代码中的对应变量。因此，我们可以直接复用相似代码中的元素生成修复补丁。之后本文所引用的代码均指替换变量之后的代码片段。接下来，SimFix 通过使用第4.5.3节中介绍的代码修改提取算法，根据缺陷代码和相似代码提取代码修改。比如对于 Listing 4.3和 4.4中的代码例子，我们可以得到图4.10中所示的匹配示意图。

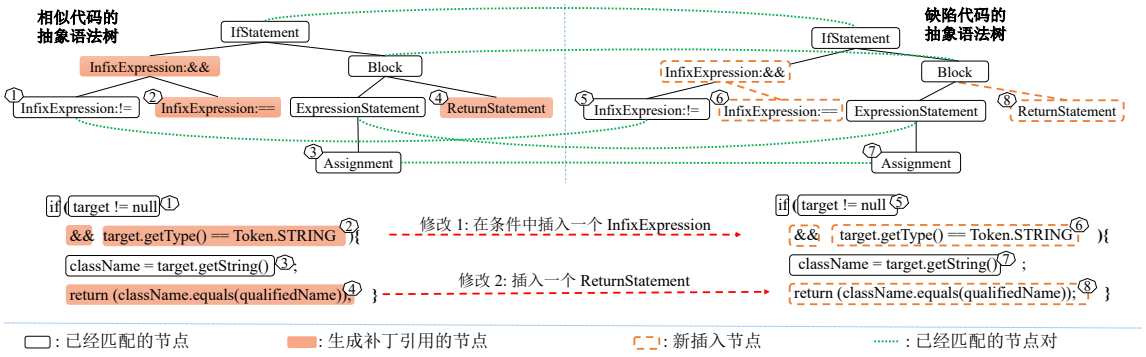


图 4.10 示例代码匹配示意图

根据匹配的结果，我们可以得到两个代码修改操作，分别是插入一个新的条件语句和插入一个**return**语句。最后，SimFix 应用在离线分析阶段得到的常用抽象代码修改操作与上述得到的具体修改操作求交集，得到最后候选的代码修改操作用于最终生成修复的补丁。由于根据相似代码提取的代码修改操作可能存在多处，为了使正确概率比较高的修复补丁被优先验证，SimFix 根据下面的规则对补丁进行分级排序，即满足前面规则的补丁具有更好优先级。

1. 修复补丁中存在一致性修改时具有较高优先级。此处的一致性修改[35,54] 指的是对于同一个变量存在多处同时修改。
2. 具有较少修改的补丁具有较高优先级。
3. 具有更多替换操作的补丁具有更高优先级。

上述第一条规则用于处理一些特殊情况。当一个变量被另一个变量替换时，其他位置的相同变量需要进行一致性修改。第二和第三条规则的依据是倾向更少、更简单的代码修改。根据上述规则，我们可以得到一个排好序的候选补丁列表用于之后的补丁验证。

4.6　讨论与小结

根据第三章实证研究对人工修复过程分析所获得的启发，本文提出了一个面向非频繁缺陷的自动修复技术 IBFix。本章对该缺陷自动修复技术进行了详细的介绍。其主要包含三个重要组成部分：(1) 基于状态划分的缺陷定位技术通过定义统一的缺陷定位模型实现了不同代码特征数据 (程序频谱和状态) 的结合，目标是进一步提升已有缺陷定位的准确率；(2) 基于单个样例的补丁生成技术通过分析海量的开源代码数据为补丁模板的提取过程提供指导，以克服该过程对大量重复修改的依赖，从而适用于修复非频繁缺陷；(3) 基于相似代码的补丁生成技术通过对比缺陷代码与相似代码的差异指导修复补丁生成，不依赖补丁模板，从而完全克服了对重复历史修改的依赖。本章介绍的自动修复技术 IBFix 针对非频繁缺陷的修复难题对修复的不同的阶段 (定位和生成补丁) 以及不同的情况 (存在和不存在重复修改) 分别使用对应技术予以克服。而且，不同技术之前存在互补性。如基于单个样例的补丁生成技术利用了历史修复知识，而基于相似代码的补丁生成技术利用了已有代码知识。因此，本章所介绍的自动修复方法比较系统而全面地处理了非频繁缺陷的修复难题。

根据本章对 IBFix 的介绍不难看出，该技术主要针对非频繁缺陷的特征和难点进行设计。但实际上，该技术也可以应用于修复频繁缺陷，但缺少必要的优化。比如 IBFix 中的定位技术也可以用于频繁缺陷缺陷的定位，并不局限于非频繁缺陷。此外，IBFix 中所提出的补丁生成技术也适用于指导修复频繁缺陷，尤其是基于相似代码的补丁生成技术，具有较强的普适性。然而，基于单个样例的补丁生成技术在用于修复频繁缺陷时依然有进一步优化的空间，原因是其并没有充分利用大量的可复用历史修改数据，由于本文主要针对非频繁缺陷修复，此处不做进一步展开讨论。

第五章　实验验证

本章系统性的验证本文所提出的面向非频繁缺陷修复方法 IBFix 的有效性。首先，第5.1节介绍本章验证所关注的研究问题。第5.2节介绍实验的具体设定。第5.3节分析 IBFix 的整体实验效果。最后，第5.4~5.7节详细分析 IBFix 中不同组件的实验效果。第5.8节讨论最新研究对本文工作的后续验证。

5.1　待验证问题

在实验中，本文对自动修复方法 IBFix 的有效性进行系统性验证，包括其整体修复效果以及其依赖的缺陷定位以及补丁生成技术的有效性。因此，本文主要针对以下五个研究问题分别验证。

问题 1: 相比已有的自动修复方法，IBFix 的修复效果如何?

问题 2: 缺陷定位技术 PredFL 能否有效提升定位准确率?

问题 3: 补丁生成技术 GenPat 单独使用修复效果如何?

问题 4: 补丁生成技术 SimFix 单独使用修复效果如何?

问题 5: 对于非缺陷修复应用，GenPat 能否提取通用代码修改模板?

5.2　实验设置

本节介绍实验中的具体配置情况，包括实验所使用数据集 (第5.2.1节)、实验结果度量标准 (第5.2.2节) 、对比技术 (第5.2.3节) 以及实验环境和配置 (第5.2.4节) 等。

5.2.1　数据集

本节介绍实验中使用的数据集。在实验中，本文使用了两个数据集，分别作为修复模板学习和修复验证数据集。

5.2.1.1　补丁模板提取数据集

根据之前的方法介绍，GenPat 需要在历史的修复中提取补丁模板，同时在修复模板的抽象过程中需要分析海量的代码数据作为指导。因此，本文从开源软件网站 GitHub[①]下载了从 2011 年到 2016 年间，所有 Java 程序的缺陷修复历史，一共包含两百

[①]https://github.com

多万条记录。为了获得和缺陷修复相关的修改，我们使用关键字 (如“repair”、“fix”、“bug”、“issue”、“problem”、“error”等) 对修改的提交说明 (commit message) 进行过滤。研究表明[52,96]，当被修改代码比较多时，容易引入无关代码修改。因此，借鉴之前的研究，本文过滤掉涉及 6 行以上源代码修改或者涉及超过 5 个文件修改的历史。此外，为了避免直接应用验证数据集中对应项目的修改历史，本文删除来自 Defects4J 中的项目修改历史。经过上述过滤，最后我们获得超过一百万条的代码修改样例。此外，本文采用同样的数据集对模板抽象过程中的属性频率进行分析，辅助抽象过程。最后，根据研究[27,47]，在软件的修复历史中会存在重复的修复历史。为了避免重复计算，GenPat 对提取的修复模板进行聚类。聚类的标准是，如果两个修复模板 A 和 B 可以实现相互匹配，即两个模板具有相同的代码图结构和代码属性，那么它们被认为是同样的修复模板，被分到一类。因此，本文最终共搜集 689,546 个不同的修复模板。

5.2.1.2 缺陷修复验证数据集

为了验证修复方法 IBFix 的有效性，本文在缺陷修复基准数据集 Defects4J[44] (v1.2.0) 上进行对比实验。该数据集共包含来自 6 个开源项目的 395 个真实程序缺陷。表5.1列出了实验数据集的详细信息，包含每个项目中涉及的程序缺陷数量、每个项目的平均代码行数以及单元测试数量。从表中的数据可以发现，该数据集中的项目均为广泛使用的较大开源软件，具有较好的维护。此外，其中的项目差异性也比较大，如 Chart 项目是 Java 上的图表库，而 Closure 是谷歌公司开源的 JavaScript 代码优化编译器等等。因此，该数据集具有一定的代表性，并在缺陷修复领域被广泛使用[87,104,113,115]。

表 5.1 缺陷修复技术验证数据集

项目名称	缺陷数量	代码行数 (kLoC)	测试数量
JFreechart (**Chart**)	26	96	2,205
Closure compiler (**Closure**)	133	90	7,927
Apache commons-math (**Math**)	106	85	3,602
Apache commons-lang (**Lang**)	65	22	2,245
Joda-Time (**Time**)	27	28	4,130
Mockito (**Mockito**)	38	45	1,457
合计	395	366	21,566

5.2.2 度量标准

本文的修复实验中最后通过人工验证修复补丁的正确性。候选修复补丁被认为是正确修复当且仅当自动修复方法所生成的修复补丁与验证数据集中提供的正确补丁语义等价。在对比不同的修复方法的修复效果时，本文采用两种度量标准，分别是召回

率 (Recall) 和准确率 (Precision)。其定义如下所示。

$$Recall = \frac{TP}{|All\ Bugs|}, \qquad Precision = \frac{TP}{TP + FP}$$

其中，TP 和 FP 分别表示正确修复和错误修复的缺陷数量。因此，召回率是指修复方法正确修复缺陷占全部缺陷的比例，而准确率指的是在可以生成候选修复补丁的缺陷中，修复方法正确修复的缺陷所占的比例。

5.2.3　对比技术

在对比修复方法的效果时，本文一共对比了 11 种已有的代表性缺陷修复方法，其中包含本文研究工作发表之后的最新缺陷修复技术。对比方法分别是 jGenProg (GenProg[51] 的 Java 实现)、jKali (Kali[82] 的 Java 实现)、Nopol[117]、ACS[115]、HDRepair[52]、ssFix[113]、ELIXIR[87]、JAID[15]、CapGen[104]、PraPR[29] 以及 SketchFix[34]。对于以上修复方法在该数据集上的实验结果均来自对应的论文，同时本文还参考已有的实证研究论文[64,99] 对缺失结果数据进一步完善。

5.2.4　环境和配置

在实验中，本文对于定位得到的结果并行使用基于历史修复和相似代码的技术生成修复补丁，最后对补丁进行排序。除了两种补丁生成方法自身的排序策略，IBFix 会对两种生成的补丁进行排序。目前 IBFix 在对候选补丁进行排序采用了一种简单的策略：基于相似代码生成的修复补丁排在基于历史修改样例生成的补丁。该排序策略主要基于以下两点观察。(1) 基于相似代码生成的修复补丁准确率更高。主要原因是参考代码来自同一个项目，而历史修复是根据跨项目信息生成补丁，前者的指导性相对更强一些。(2) 基于相似代码生成的修复补丁不依赖重复修改信息。因此，原则上前者的使用范围会更广泛。实际上，在最终的实验验证部分也表明上述排序策略具有很好的表现。本实验在 64 位的 Linux 操作系统上进行实验，每个程序缺陷的修复时间为不超过 5 个小时，每个缺陷最多生成 10 个候选修复补丁。

5.3　自动修复方法 IBFix 的整体修复效果

5.3.1　缺陷修复数量

表5.2列出了缺陷修复的实验结果。表格中的“X(Y)”表示对应方法在对应项目上正确修复“Y”个缺陷程序 (不考虑正确补丁的排序)，其中“X”缺陷的正确补丁排在候选补丁中的第一位置。特殊的，由于一部分缺陷修复方法只生成一个修复补丁，因此我们

省略了部分数据的“Y”。此外，方法 HDRepair 和 SketchFix 没有给出对应项目的详细修复数据，对应的“X”数据被省略。类似的，由于部分修复方法只在 Defects4J 的前五个项目上进行实验，因此，对于 Mockito 项目部分修复方法没有数据。表格中灰色高亮的单元格代表对应项目被正确修复的最多缺陷数量。

表 5.2　IBFix 与已有缺陷修复方法对比

项目	IBFix	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
Chart	**6(7)**	0	0	1	2	-(2)	3	4	2(4)	4(4)	4(7)	-(6)
Closure	7(8)	0	0	0	0	-(7)	2	0	5(9)	5(5)	**12(14)**	-(3)
Math	**16(16)**	5	1	1	12	-(7)	10	12	1(7)	12(13)	6(10)	-(7)
Lang	**12(12)**	0	0	3	3	-(6)	5	8	1(5)	0(0)	3(6)	-(3)
Time	1(1)	0	0	0	1	-(1)	0	**2**	0(0)	0(0)	1(3)	-(0)
Mockito	**1(1)**	-	-	-	-	-	-	-	-	0(0)	0(3)	-
合计	**43(45)**	5	1	5	18	13(23)	20	26	9(25)	21(22)	26(43)	9(19)

从表中的数据可以看出，IBFix 无论是考虑排在第一位置的正确修复补丁还是任意位置的正确补丁，在该数据集上都实现了最多数量的正确修复，尤其是对于排在第一位置的正确修复补丁，IBFix 正确修复了 43 个 (10.1%) 缺陷，相比于 PraPR 和 ELIXIR (效果次好方法) 提升了 65.4%。而相比其他的缺陷修复方法修复数量增加了 1-42 倍。当不考虑修复补丁排序时，IBFix 增加了两个正确修复，分别是 Chart-4 和 Closure-2，其正确补丁在候选列表中分别排在了第 5 和第 2 的位置。该实验结果表明本文所提出的修复方法 IBFix 可以有效提升缺陷修复的数量。其主要原因是 IBFix 有效地提升了非频繁缺陷的修复能力，使得修复数量实现很大提升。比如 Listing 5.1是 Defects4J 中 Math-71 的正确修复补丁。从修复补丁中可以发现，正确修复该缺陷需要插入 4 条独立的程序语句 (2-5 行)，由于其代码功能与上下文密切相关并且修改比较复杂，在历史修复中并不存在相同修复，是典型的非频繁缺陷。已有的修复方法均不能正确修复该缺陷。然而，本文提出的 IBFix 通过搜索缺陷程序中的相似代码可以正确修复该缺陷，反映了 IBFix 方法的有效性。

```
1.   if(Math.abs(dt) <= Math.ulp(stepStart)){
2. +    interpolator.storeTime(stepStart);
3. +    System.arraycopy(y, 0, yTmp, 0, y0.length);
4. +    hNew = 0;
5. +    stepSize = 0;
6.      loop = false;
7.   } else {...}
```

Listing 5.1　缺陷程序 Math-71 的修复补丁

为了进一步对比 IBFix 与已有修复方法的修复效果，本节将不同修复方法所修复的具体缺陷详细列于表5.3中。其中，第二列代表 Defects4J 中的缺陷编号，后面的每列

表示对应修复方法的修复结果。○表示对应缺陷可以被修复工具正确修复 (不考虑正确补丁的排序)，●表示对应缺陷仅被 IBFix 方法正确修复。

分析表中数据可以发现，目前所有对比修复方法 (包括 IBFix) 所能修复的缺陷总数为 101 个，而本文方法 IBFix 则可以正确修复接近一半 (44.6%)。除此之外，在所有正确修复的缺陷中，有 16 个 (15.8%) 缺陷只有 IBFix 可以正确修复。上述实验结果表明，IBFix 不仅可以有效地提升缺陷修复方法对非频繁缺陷的修复能力，同时更具有较强的通用性。从表5.2中可以看出，当只考虑第一个候选修复补丁时，只有 IBFix 可以在所有类型的项目上都能实现部分缺陷正确修复。此外，从不同的项目角度分析，在大部分项目上 (**Chart**、**Math**、**Lang** 和 **Mockito**)，IBFix 实现修复数量最多。

最后，通过分析 IBFix 中的两种修复补丁生成技术所产生的候选补丁发现，在 IBFix 正确修复的所有 45 个缺陷中，基于相似代码的补丁生成技术 (SimFix) 正确修复了 34 个，基于历史修复的补丁生成技术 (GenPat) 正确修复了 19 个。即针对其中的 8 个程序缺陷，两种技术均生成了正确的修复补丁。此外，采用 SimFix 补丁优先的排序方法成功避免了 GenPat 产生的 6 个不正确修复。同时，针对 SimFix 未能正确修复的缺陷，GenPat 同样未能产生正确补丁，说明 SimFix 修复方法具有更强的修复能力 (事实上，之后会验证 GenPat 具有更强的通用性)。上述结果同样反映了本文所提出的补丁生成技术具有很好的互补性，实现了非频繁缺陷的有效修复。

表 5.3　不同缺陷修复方法详细结果

ID	Bug	IBFix	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
1	**Chart-1**	○					○	○	○	○	○	○	○
2	**Chart-3**	●											
3	**Chart-4**	●											
4	**Chart-5**				○								
5	**Chart-7**	●											
6	**Chart-8**						○		○		○	○	○
7	**Chart-9**								○	○			○
8	**Chart-11**	○							○		○	○	○
9	**Chart-12**											○	
10	**Chart-14**					○							
11	**Chart-19**					○							
12	**Chart-20**	○						○				○	○
13	**Chart-24**	○						○		○	○	○	○
14	**Chart-26**									○		○	
15	**Cloure-2**	●											
16	**Cloure-10**						○					○	
17	**Cloure-11**											○	
18	**Cloure-14**	○					○	○				○	○

表 5.3　不同缺陷修复方法详细结果 (续)

ID	Bug	IBFix	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
19	Cloure-18									○		○	
20	Cloure-31									○		○	
21	Cloure-33									○			
22	Cloure-40									○			
23	Cloure-46											○	
24	Cloure-51						○						
25	Cloure-57	●											
26	Cloure-62	○					○			○		○	○
27	Cloure-63	○								○		○	
28	Cloure-70						○			○		○	
29	Cloure-73	○					○			○		○	
30	Cloure-86	○										○	
31	Cloure-92											○	
32	Cloure-93											○	
33	Cloure-115	○						○					
34	Cloure-126						○			○		○	○
35	Math-3					○							
36	Math-4	○				○							
37	Math-5	○	○			○	○		○	○	○	○	○
38	Math-22	○					○						
39	Math-25					○							
40	Math-30							○	○		○		
41	Math-32									○			
42	Math-33	○						○	○		○	○	○
43	Math-34						○		○			○	
44	Math-35	○				○							
45	Math-41	○						○					
46	Math-50	○	○	○	○		○	○	○	○		○	○
47	Math-53	○	○				○	○		○	○		
48	Math-57	○						○	○		○		
49	Math-58								○			○	
50	Math-59	○						○	○		○	○	○
51	Math-61					○							
52	Math-63	○									○		
53	Math-65										○		
54	Math-68										○		
55	Math-70	○	○				○	○	○		○	○	○
56	Math-71	●											
57	Math-73		○										
58	Math-75	○							○		○	○	
59	Math-79	○						○					

表 5.3　不同缺陷修复方法详细结果 (续)

ID	Bug	IBFix	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
60	Math-80							○		○			
61	Math-82					○	○		○	○		○	○
62	Math-85					○			○	○	○	○	○
63	Math-89					○							
64	Math-90					○							
65	Math-93					○							
66	Math-98	●											
67	Math-99					○							
68	Lang-6						○	○	○		○	○	○
69	Lang-7					○							
70	Lang-10						○					○	
71	Lang-16	●											
72	Lang-21	○						○					
73	Lang-22												
74	Lang-24					○			○				
75	Lang-26								○		○	○	
76	Lang-27	●											
77	Lang-33	○						○	○	○		○	
78	Lang-35					○							
79	Lang-38								○	○			
80	Lang-39	●											
81	Lang-41	●											
82	Lang-43	○					○	○	○		○		
83	Lang-44				○								
84	Lang-45									○			
85	Lang-47	●											
86	Lang-50	●											
87	Lang-51	○					○			○			
88	Lang-55				○					○			○
89	Lang-57						○		○		○	○	
90	Lang-58	○			○								
91	Lang-59						○	○	○		○	○	○
92	Lang-60	●											
93	Time-4								○			○	
94	Time-7	●											
95	Time-11											○	
96	Time-15					○			○				
97	Time-19						○					○	
98	Mockito-5											○	
99	Mockito-22	●											
100	Mockito-29				○							○	

表 5.3　不同缺陷修复方法详细结果 (续)

ID	Bug	IBFix	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
101	**Mockito-38**											◯	

5.3.2　缺陷修复准确率

本节分析缺陷修复方法的修复准确率。图5.1中展示了不同修复方法的实验结果。由于部分修复方法同时给出了正确修复补丁的排序，因此在图中我们分别画出了仅考虑第一个修复补丁的准确率以及不考虑补丁排序的准确率 (即只要候选中存在正确补丁即可)[①]。从图中可以看出，相比于大部分缺陷修复方法，IBFix 具有较高的补丁准确率，但依然比 ACS 和 ELIXIR 的正确率低。一方面原因是 ACS 的设计目标是高准确率的条件修复，故其修复的数量相比要少很多 (如表5.2所示)。而 ELIXIR 方法是通过人工定义修复补丁模板的方法实现修复，因此其主要适用于比较常见的缺陷错误类型，通用性有限。需要特殊说明的是 CapGen 方法原本在四个项目中验证 (不包括 Closure 和 Mockito 项目)，其准确率为 84%。本文同时结合了已有的实证研究论文[29,99] 对其实验结果进行补全，发现 GapGen 在另外的两个项目上产生了很多不正确的修复补丁，从而导致其准确率降低很多。此外，当不考虑修复补丁的位置时，IBFix 的准确率并没有特别大的提升，原因是 IBFix 只考虑前 10 个正确修复补丁，因为补丁数量太多会增加开发者验证补丁的负担。因此，实用性降低。然而，从图中可以看到 JAID 和 SketchFix 的准确率提升了很多。事实上，JAID 的很大一部分 (10/25) 正确修复补丁已经排在了很靠后的位置，只有 15 个缺陷的正确修复补丁排在前 10 位置。因此，如果只考虑前 10 的正确修复补丁，其准确率只有 48.4%。而 SketchFix 并没有给出所有补丁的排序，因此只能考虑所有位置的正确补丁。

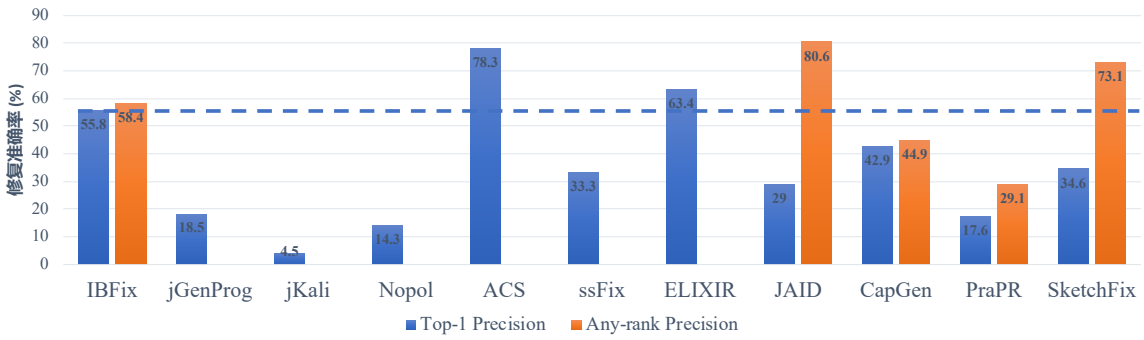


图 5.1　缺陷自动修复方法准确率对比

[①]HDRepair 没有给出错误补丁的数量，因此图中省略其数据。

从上面的分析结果可以看出，尽管 IBFix 没有实现最好的修复准确率，但是其准确率依然优于大部分修复方法。此外，目前很多研究通过生成测试或者或程序分析的方式过滤不正确的修复补丁[112,114,120]，这些方法可以集成到我们的修复技术中，有希望进一步提升补丁的质量。

5.4　IBFix 中缺陷定位技术的有效性验证

本节分析 IBFix 中所使用的缺陷定位技术 PredFL 的有效性。通过第4.3.3节中的对比分析我们已经知道该定位技术相比于结合之前的基于程序频谱定位和基于统计性调试的定位方法都有了明显的定位效果提升。然而，近几年研究者提出了一些通过结合不同定位方法提升现有方法的定位准确率。由于绝大部分已有的定位方法已经被集成到统一的定位方法中，即大部分的程序信息已经被已有的定位方法所使用。为了进一步验证本文所提出的定位方法是否依然与现有的方法互补，可以进一步提升现有定位方法的准确率，本节将 PredFL 与现有定位方法进行对比。在对比实验中，本文通过将 PredFL 与已有的定位技术结合框架进行集成来探索其与已有技术的互补性。本文使用了一个目前最新的开源定位方法集成框架 CombineFL[132] 作为对比。该框架集成了目前已有的绝大部分定位技术。

表 5.4　CombineFL 定位框架中的集成级别

定位时间级别	分类	技术工具
Level 1 (几秒到几十秒)	基于历史 (history-based) 基于堆栈信息 (stack trace) 基于信息检索 (IR-based)	Bugspots stack trace BugLocator
Level 2 (几分钟)	程序切片 (slicing) 基于频谱 (spectrum-based)	union, intersection and frequency Ochiai and DStar
Level 3 (约 10 分钟)	谓词翻转 (predicate switching)	predicate switching
Level 4 (几小时)	基于变异 (mutation-based)	Metallaxis and MUSE

CombineFL 集成了七种已有定位方法，覆盖基于程序频谱 (Spectrum-based)[2]、基于变异 (Mutation-based)[72,78]、基于历史 (History-based)[83]、基于信息检索 (IR-based) 定位方法[131]，以及堆栈分析 (Stack trace analysis)[88]、动态切片 (Dynamic slicing)[97,100]、和谓词翻转 (Predicate switching)[127] 方法。并且不同的定位方法之间可以随意组合。根据定位方法所用时间的差异，该定位框架有四个定位时间级别，如表5.4所示。其中高级别的集成方法包含所有低级别的定位方法。根据之前的分析，PredFL 定位时间在三分钟之内。因此，在与其进行集成时，本文同样考虑相同定位时间的方法集成以不影响定

位的效率。此外，本文也会集成所有方法，来探索本文新的定位方法是否对已有方法具有提升效果。因此，本文在 Level 2 和 Level 4 两个级别与 CombineFL 定位框架集成进行对比实验。需要注意的是，由于该方法已经证明其效果比其他的定位效果好，如 FLUCCS[90]。本文将直接与该定位框架的结果进行对比，而不再重复与其他方法对比。另外，本对比实验依然在 Defects4J 数据集上进行对比。由于该定位框架在 Defects4J 的前五个项目上进行实验 (不包含 Mockito 项目)，本文实验在同样数据集上验证。本节采用第4.3.3节对比分析中同样的方法实现与度量标准。

表 5.5　PredFL 定位方法对已有方法的提升效果

	Level 2							
	Top-1		**Top-3**		**Top-5**		**Top-10**	
	-	+	-	+	-	+	-	+
Chart	11	**13**	19	**18**	21	**18**	23	**25**
Math	49	**58**	78	**81**	84	**87**	90	**94**
Lang	42	**45**	55	**55**	57	**56**	60	**60**
Time	12	**12**	15	**16**	17	**19**	18	**20**
Closure	30	**46**	47	**64**	52	**68**	59	**81**
合计	144	**174**	214	**234**	231	**248**	250	**280**
百分比	40.3%	**48.7%**	59.9%	**65.5%**	64.7%	**69.5%**	70.0%	**78.4%**
	Level 4							
	Top-1		**Top-3**		**Top-5**		**Top-10**	
	-	+	-	+	-	+	-	+
Chart	13	**14**	19	**18**	21	**18**	24	**20**
Math	63	**67**	83	**86**	85	**91**	93	**95**
Lang	47	**51**	59	**57**	61	**61**	61	**61**
Time	10	**12**	12	**17**	16	**19**	19	**20**
Closure	35	**41**	57	**64**	64	**76**	74	**88**
合计	168	**185**	230	**242**	247	**265**	271	**284**
百分比	47.1%	**51.8%**	64.4%	**67.8%**	69.2%	**74.2%**	75.9%	**79.6%**

表格中的 “-’’ 列表示原始的 CombineFL 定位结果，“+’’ 列表示集成了 PredFL 方法之后的定位结果。

表5.5展示了实验中的定位结果。从实验结果可以发现，尽管现有的定位方法已经集成了很多程序信息，本文所提出的方法 PredFL 依然可以进一步提升其定位的准确率，提升幅度从 4.8% (75.9% vs 79.6%) 到 20.8% (48.7% vs 40.3%)。不仅如此，分析表中数据可以发现，对于单个项目的定位效果，PredFL 依然具有提升作用。特别的，当集成 PredFL 之后，在 Level 2 时间级别的定位准确率比 Level 4 还要高，体现了方法 PredFL 的定位有效性以及与目前所有定位方法的互补性。

小结：从实验分析中可以看出，通过结合两种不同定位方法可以有效提升缺陷定位的准确率。但是对于本文中所提出的定位方法，依然存在可以进一步优化的空间。对于未来工作，基于本文所提出的统一定位模型，可以从以下两个方面继续优化以提升 PredFL 的定位准确率：(1) **谓词筛选**。从第4.3.3节中的分析可知，不同的谓词在实际的

定位中所起的作用是不一致的，部分谓词辅助定位，同时有部分谓词会造成噪声干扰影响定位效果。比如 Listing 5.2中所展示的是 Chart-9 的修复补丁，其出错的代码是第 5 行中的条件代码，而在实验中发现，虽然该缺陷实现了准确的定位，其中发挥作用的谓词是在第 1 行中的条件而不是第 5 行的条件。因此，通过筛选候选状态谓词，有希望提升状态谓词的质量，进而可以为人工或者自动化的修复过程提供指导信息。(2) **风险评估公式**。目前 PREDFL 采用的是基于程序频谱定位方法中的风险评估公式。然而，实际上该公式的设计目标并不是针对状态谓词。因此，为其设计更合适的计算公式有可能进一步提升定位准确率。

```
1.  if(start == null) {
2.     throw new IllegalArgumentException();
3.  }
4.  ...
5.- if(endIndex<0){
6.+ if((endIndex<0)||(endIndex<startIndex)){
```

Listing 5.2　Defects4J 数据集中 Chart-9 的修复补丁

综上所述，本文提出缺陷定位技术 PREDFL 可以有效提升缺陷定位的准确率，并与目前已有的定位方法存在互补性。

5.5　基于单个样例的补丁生成技术有效性验证

为了验证 GENPAT 单独使用时在自动化缺陷修复中的效果，本节将其配置成了一个可以独立使用的缺陷修复工具。为了和已有的缺陷修复方法进行对比，该修复工具中同样使用已有的缺陷定位技术定位程序中的缺陷，并将结果与已有方法进行对比。该实验中，本文同样设置每个缺陷的修复时间不超过 5 个小时。表5.6中列出了不同缺陷修复方法的实验结果。

表 5.6　GENPAT 与已有缺陷修复方法对比

项目	GENPAT	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
Chart	3(4)	0	0	1	2	-(2)	3	**4**	2(4)	**4(4)**	**4**(7)	-(6)
Closure	5(6)	0	0	0	0	-(7)	2	0	5(9)	5(5)	**12(14)**	-(3)
Math	3(4)	5	1	1	12	-(7)	10	12	1(7)	**12(13)**	6(10)	-(7)
Lang	4(4)	0	0	3	3	-(6)	5	**8**	1(5)	0(0)	3(6)	-(3)
Time	0(0)	0	0	0	1	-(1)	0	**2**	0(0)	0(0)	1(3)	-(0)
Mockito	**1(1)**	-	-	-	-	-	-	-	-	0(0)	0(3)	-
合计	16(19)	5	1	5	18	13(23)	20	**26**	9(25)	21(22)	**26(43)**	9(19)

从实验结果可以看出，GENPAT 一共正确修复了 19 个缺陷。其中，对于 16 个缺陷，GENPAT 返回的第一个即是正确补丁，优于一些最新的缺陷修复方法，如 SketchFix、

JAID 和 HDRepair 等。然而，GenPat 的修复数量依然比一些已有的缺陷修复方法少，如 PraPR、CapGen 和 ACS 等。结果表明 GenPat 依然有进一步提升的空间。此外，GenPat 在修复的过程中产生了 23 个不正确的修复补丁，其修复补丁准确率为 45.2%，高于绝大部分已有的自动修复方法 (参考图5.1)。通过详细分析，GenPat 有两个方面依然可以继续提升。首先，由于 GenPat 是一个通用的代码修改模板提取方法，目前的实现缺少高效的搜索算法，导致在大量的补丁模板中难以定位正确修复。其次，实验中 GenPat 提取模板时只扩展一层上下文信息，会导致一些模板的错误使用。为了进一步提升 GenPat 的缺陷修复能力，通过开发高效的搜索算法以及增加上下文信息有希望提升模板使用的正确性，增加修复的数量。

```
12   public static boolean areEqual(Object o1,Object o2){
13 +   if(o1==o2){
14 +     return true;
15 +   }
16     if(o1==null||o2==null){
```

Listing 5.3　Mockito-22 的修复补丁

此外，GenPat 正确修复的 19 个缺陷中，4 个缺陷从未被任何已有的修复方法所修复。表明该方法可以提升现有方法对非频繁缺陷的修复。比如 Listing 5.3中展示的是 Mockito-22 的修复补丁，目前已有的修复方法都不能正确修复该缺陷。然而，在实验中，GenPat 在开源项目的修复历史中成功地提取出了用于修复该缺陷的补丁模板，Listling 5.4中列出了该历史修改代码[①]。事实上，在本文的补丁模板提取数据集中，类似修改只存在一个。反映了根据单个的代码修改样例提取修复的模板对于修复非频繁缺陷的重要性。GenPat 根据该示例修改可以正确提取补丁模板并修复该缺陷，证明了 GenPat 对于修复费频繁缺陷的有效性。

```
526  boolean _equalsComplexEL(Object left,Object right,...){
527 +   if(left==right){
528 +     return true;
529 +   }
530     if(Decision.isSimpV(left)&&Decision.isSimpV(right)){
```

Listing 5.4　修复 Mockito-22 的参考历史修复

然而，尽管 GenPat 对于修复非频繁的缺陷有比较强的适应性，但其正确修复的缺陷总数依然不是很理想，依然有进一步优化的空间。一方面是上面所介绍的算法优化问题，另一方面是 GenPat 是一个通用的代码修改模板提取方法，缺少对缺陷修复的针对性优化。在第5.7节中我们将验证其具有较强的通用性。

[①] https://github.com/clitnak/mcrailo/commit/8e76da8

5.6　基于相似代码的补丁生成技术有效性验证

5.6.1　整体修复效果

本节讨论 SimFix 对于修复真实缺陷的有效性。在该实验中，本文将 SimFix 配置成为一个独立的缺陷修复工具，其中缺陷定位技术使用的是已有的缺陷定位技术，最后将修复的结果与已有方法进行对比。实验中，本文设置每个缺陷的修复时间不超过 5 个小时。表5.7中列出了不同修复方法的详细结果。

表 5.7　SimFix 与已有缺陷修复方法对比

项目	SimFix	jGenProg	jKali	Nopol	ACS	HDRepair	ssFix	ELIXIR	JAID	CapGen	PraPR	SketchFix
Chart	**4(4)**	0	0	1	2	-(2)	3	4	2(4)	**4(4)**	**4(7)**	-(6)
Closure	6(6)	0	0	0	0	-(7)	2	0	5(9)	5(5)	**12(14)**	-(3)
Math	**14(14)**	5	1	1	12	-(7)	10	12	1(7)	12(13)	6(10)	-(7)
Lang	**9(9)**	0	0	3	3	-(6)	5	8	1(5)	0(0)	3(6)	-(3)
Time	1(1)	0	0	0	1	-(1)	0	**2**	0(0)	0(0)	1(3)	-(0)
Mockito	0(0)	-	-	-	-	-	-	-	-	0(0)	0(3)	-
合计	**34(34)**	5	1	5	18	13(23)	20	26	9(25)	21(22)	26(43)	9(19)

从表中对比结果可以看出，SimFix 在实验中一共正确修复了 34 个缺陷。相比于目前已有的修复方法，正确修复了最多数量的程序缺陷。当考虑在不同项目上的修复效果时，SimFix 在一半的项目上都实现了正确修复数量最多，表明了 SimFix 具有更强的补丁生成能力，尤其是针对非频繁的缺陷。比如第4.5.5节 Listing 4.3中所展示的缺陷代码，已有的缺陷修复方法都不能正确修复，然而通过参考相似代码，SimFix 正确修复了该缺陷。事实上，该缺陷在历史中并无参考相似修复存在，所以通过提取补丁模板的方法也不能有效修复。

除此之外，SimFix 一共生成了 22 个错误的补丁，其修复准确率为 60.7%。根据图5.1，其准确率高于大部分缺陷修复方法 (4.5%-42.9%)，和 ELIXIR 的准确率 (63.4%) 非常接近。值得注意的是，尽管如此，SimFix 的准确率依然比 ACS (78.3%) 要低。主要原因是 ACS 专注于高精度的条件错误修复。此外，一些已有的方法[112,114,120] 提出可以通过过滤自动生成的修复补丁来提升补丁的准确率，这些方法可以用于 SimFix 补丁的过滤，有希望进一步提升补丁的质量。

最后，为了分析 SimFix 修复非频繁缺陷的能力，本文对比了其修复缺陷与已有修复方法的交集。维恩图5.2展示了不同修复方法的修复补丁异同。特殊的，为了描述清晰，在该图中本文只列出了所有 SimFix 可以正确修复的缺陷，其他未列出的缺陷也有可能被其他方法正确修复 (参考表5.7)。从图中可以看出，SimFix 正确修复的缺陷中，12 个缺陷都不能被已有的方法修复。实验结果表明 SimFix 和现有的缺陷修复方法具有互补性。其中一个主要的原因是 SimFix 可以修复很多非频繁的缺陷。比如，除了 Listing 4.3中所示的缺陷外，第5.3节 Listing 5.1中所展示的缺陷代码同样是根据相似代

码实现正确修复。因此，实验结果表明 SimFix 可以有效修复非频繁的缺陷，同时不依赖任何重复修复。

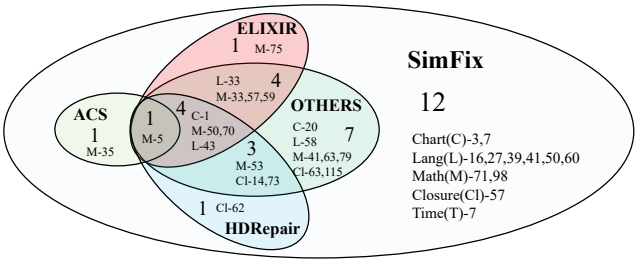


图 5.2　SimFix 修复结果与已有方法交集

5.6.2　历史修改对修复的影响

根据之前的介绍，SimFix 通过使用相似代码为非频繁缺陷的修复提供了修改指导和补丁原料，而根据历史的修复补丁所构造的抽象修改空间进一步优化了补丁生成的质量。本节验证历史修复补丁在修复中的影响。为此，我们对 SimFix 进行修改，删除了抽象修改空间对生成补丁的约束，即仅仅根据相似代码生成修复补丁。为了方便描述，本节称该方法为 SimFix-A。

表5.8中列出了实验对比结果。分析表中数据发现，当不适用历史修复对代码修改进行约束时，12 个缺陷将不能被正确修复。其主要原因有两个方面：(1) 由于代码修改空间变大，补丁搜索空间中会存在更多的不正确补丁，使得错误的修复补丁在正确补丁之前被生成。对比补丁的准确率可以看出，SimFix-A 生成了更多不正确的修复补丁。(2) 当补丁的搜索空间变大，在有限的修复时间之内会导致正确的修复补丁不能得到验证。在实验中，相比 SimFix, SimFix-A 平均为每个缺陷多生成了 2.3 倍数量的候选修复补丁才实现正确修复，使得 SimFix-A 大概花费两倍的正确修复时间。该实验结果表明，使用历史修复对补丁空间进行优化具有重要作用。

表 5.8　SimFix 变体效果对比.

方法变体	Chart	Closure	Math	Lang	Time	Mockito	合计	补丁准确率
SimFix	4	6	14	9	1	0	34	60.7%
SimFix-A	2	2	11	7	0	0	22	37.9%
SimFix-D	3	6	11	9	0	0	29	46.0%

5.6.3　删除操作对修复的影响

根据第4.5.2节的介绍，由于删除操作容易导致错误的修复补丁，SimFix 中只包含代码插入和代码替换操作。本节验证删除代码操作对修复效果的影响。为此，本文对

SimFix 进行了一些修改使得其允许删除代码操作，称为 SimFix-D。表5.8中列出了该方法的修复结果。从表中可以发现，SimFix-D 相比于 SimFix 减少了 5 个正确修复，同时使得修复的准确率降低了 14 个百分点。其原因和 SimFix-A 类似，搜索空间的增大会导致更多不正确的修复补丁，为补丁搜索增加难度，从而降低补丁的质量。

5.6.4　细粒度代码修改对修复的影响

根据第4.5.1节的介绍，相比于之前方法[45,51,81,102,123] 在语句级别复用代码，SimFix 通过抽象语法树上的代码差异算法实现了对缺陷代码的细粒度 (表达式级别) 修改。为了分析其对修复效果的影响，本文人工分析了 SimFix 生成的正确修复补丁，统计有多少补丁可以通过语句级别的代码复用被生成。分析结果表明，当只允许复用语句时，原来的 17 个正确修复补丁不能被生成，因为它们都是复用了相似代码中更细粒度的代码元素实现的修复。因此，结果表明 SimFix 的细粒度代码复用也是提升缺陷修复数量的一个重要因素。

5.7　代码修改模板提取技术的通用性验证

根据之前的介绍，本文所提出的代码修改模板提取方法 GenPat 不仅可以应用于修复非频繁的缺陷，同时在其他类似的应用场景下具有通用性。本节通过系统性代码修改 (Systematic Editing)[69] 应用验证其通用性，并与目前最好的技术 Sydit 进行对比实验。系统性代码修改指的是开发者提供一个示例代码修改，自动化工具根据示例提取代码修改模板并自动修改其他位置相似代码。

数据集　为了验证 GenPat 的通用性，本实验在两个数据集上验证 GenPat 的通用性。第一个数据集是由 Meng 等人[69] 提出用来验证方法 Sydit 的有效性，称为 Sydit 数据集。另外一个数据集是由 Kreutzer 等人[50] 提出，通过聚类算法将相似的修改进行归类，称为 *C*3 数据集。表5.9列出了数据集的详细信息。从表中数据可以看出，两个数据集中包含示例的数量不同。此外，两个数据集搜集的过程也有所不同。Sydit 数据集中的修改代码是使用 ChangeDistiller[23] 提取代码修改，并要求相似修改之间必须拥有至少一个相同的修改操作。此外，被修改的代码要满足至少 40% 的文本相似性。因此，该数据集对相似修改的要求比较严格。相反的，数据集 *C*3 不仅包含更多数量的代码修改对，同时其数据的搜集是通过聚类算法将不同的代码修改进行聚类，同一类别内的修改变化性更大，并且每个类别中可能多于两个修改。本文对多于两个相似修改的类别进行随机采样，选择其中两个作为实验数据。

表 5.9 GenPat 通用性验证数据集

数据集名称	包含项目	修改实例对数
Sydit	-	56
$C3$	junit	3,904
	cobertura	2,570
	jgrapht	2,490
	checkstyle	13,263
	ant	25,063
	fitlibrary	3,199
	drjava	31,393
	eclipsejdt	73,109
	eclipseswt	63,446
合计		218,493

过程 在该实验中，给定相似的一对代码修改 ($v_a \rightarrow v_{a'}, v_b \rightarrow v_{b'}$)，本文选择其中的一个修改作为提取修改模板的训练数据 (如 $v_a \rightarrow v_{a'}$)，然后尝试使用该模板修复另外一个代码 (如 v_b)，并生成修改之后的代码 $v_{b'*}$。最后，通过比较 $v_{b'*}$ 和正确的代码 $v_{b'}$，判断修改是否正确。在实验中，本文根据语法等价来判断两个代码是否相同。除此之外，我们人工采样了一些修改实例，人工验证其正确性。对于每个代码修改操作，本文设定实验时长为 1 分钟。特殊的，在该过程中，GenPat 使用缺陷修复中使用的相同历史修复数据集统计代码属性频率，辅助模板抽象。

结果分析 首先，本文在完整的数据集上验证 GenPat 的效果。表5.10列出了 GenPat 的实验结果，其中第三列表示实验中使用的代码修改样例数。第四列表示 GenPat 可以从给定的代码修改实例中提取修改模板并成功应用在测试代码上的数量。最后一列表示使用对应修改之后所得到的代码与给定的正确代码语法等价的数量。表中的括号内是比例数据。从实验结果可以看出，GenPat 可以为 39.1% 的代码修改提取修改模板并适配目标代码。特殊的，GenPat 可以正确修改 16.0% 的代码，保证代码修改后与目标代码完全一致。

此外，为了对比 GenPat 的效果，本文与 Sydit 做对比实验。其中，对于 Sydit 数据集上的实验，本文直接采用其论文中的实验结果。对于其他的项目，由于 Sydit 要求转换的代码前后是完整的可编译项目，在部分实验数据上由于缺少依赖或者引发程序异常导致不能正常实验。因此，最后本文在三个项目上进行对比实验。实验结果如表5.11所示。特殊的，由于在 Sydit 的原始论文中通过人工检查修改之后的代码是否与目标代码语义等价来判定代码转换是否正确。为了公平对比，本文在实验中同样人工检查 GenPat 在 Sydit 数据集上的实验结果。对于 $C3$ 数据集上的实验，由于实验总数较多，本文依然使用语法等价判断代码修改结果是否正确。表格中的最后一列表示在工具 Sydit 可以正确适配的修改代码中，GenPat 可以实现语法等价修改的数量。

表 5.10　GenPat 在完整数据集上的通用性验证结果

数据集	项目名称	相似修改对数	正确适配数	语法等价修改数
Sydit	-	56	49 (87.5%)	27 (48.2%)
*C*3	junit	3,904	1,088 (27.9%)	412 (10.6%)
	cobertura	2,570	769 (29.9%)	305 (11.9%)
	jgapht	2,490	547 (22.0%)	226 (9.1%)
	checkstyle	13,263	5,918 (44.6%)	1,679 (12.7%)
	ant	25,063	10,428 (41.6%)	4,398 (17.5%)
	fitlibrary	3,199	922 (28.8%)	374 (11.7%)
	drjava	31,393	11,391 (36.3%)	4,151 (13.2%)
	eclipsejdt	73,109	32,037 (43.8%)	14,150 (19.4%)
	eclipseswt	63,446	22,218 (35.0%)	9,206 (14.5%)
合计		218,493	85,367 (39.1%)	34,928 (16.0%)

表 5.11　GenPat 和 Sydit 对比实验结果

数据集	项目名称	修改对数	正确适配数		语法/语义等价修改数		Sydit 正确适配中 GenPat 语法等价修改数
			GenPat	Sydit	GenPat	Sydit	
Sydit	-	56	49(87.5%)	46(82.1%)	-/40(71.4%)	-/39(69.6%)	-
*C*3	jgrapht	1,314	354(26.9%)	20(1.5%)	211(16.1%)	6(0.5%)	7
	junit	1,208	383(31.7%)	240(19.9%)	206(17.1%)	57(4.7%)	110
	cobertura	1,021	293(28.7%)	235(23.0%)	113(11.1%)	1(0.1%)	0
	总计	3,543	1,030(29.1%)	495(14.0%)	530(15.0%)	64(1.8%)	117
合计		**3,599**	**1,079(30.0%)**	**541(15.0%)**	**570(15.8%)**	**103(2.9%)**	-

表格中，“-” 表示缺失数据。

通过对比实验结果，在 Sydit 数据集上，GenPat 的实验效果和 Sydit 几乎一致。然而，对于其他的实验项目，无论是正确适配的修改数量还是语法等价修改的数量，GenPat 的结果明显优于 Sydit。总体上，GenPat 相比于 Sydit，可以正确适配 2 倍的代码修改 (1079/541)，实现 5.5 倍 (570/103) 语法等价的代码修改。当考虑非语法等价的修改时，GenPat((1079-570)/1079=47.2%) 的效果依然比 Sydit((541-103)/541=81.0%) 好。此外，即使在 Sydit 可以实现正确适配的代码修改数据上，GenPat 依然明显优于 Sydit(117 *vs* 64)。除此之外，当对比两种方法在单个项目上的实验结果时，GenPat 相比于 Sydit 有更好的结果。总的来说，GenPat 的实验结果明显优于 Sydit。

实际上，通过语法等价判断修改之后的代码是否正确并不是精确的判定方法。因为在语法不等价的情况下，修改之后的代码依然有可能是正确的，比如修改了局部变量的命名。为了分析语法不等价的代码修改中有多少是正确的修改 (语义等价)，本文进一步采样了部分非语法等价修改代码进行人工分析。本文从 *C*3 数据集的每个项目中随机选取 20 个样本，最终选取了 60 个 GenPat 的结果和 54 个 Sydit 的结果 (项目 jgrapht 中只有 14 个样本)。分析之后发现，GenPat 的结果中 11.7% 的修改是语法等价的，而对于 Sydit，该比例降到了 9.3%。分析结果表明，GenPat 可以实现更高比例的语法等价代码修改。

从实验结果可以看出，虽然 GenPat 效果明显优于 Sydit，但其依然不能对很多代码进行正确修改 (语法或语义等价)。因此，为了进一步研究 GenPat 不能正确修改的原因，本文随机选取了 100 个错误的代码修改样本进行人工分析。其原因主要包括以下四个方面：(1) 一个主要的原因是实验的数据集中存在很大的噪音。如前所述，$C3$ 数据集通过自动化的聚类算法得到，因此难免会引入一些不正确的代码修改。比如在示例代码中将变量`runners`修改为`fRunners`，但是目标代码却是要将`fRunners`改为`runners`。所以，根据示例代码不可能正确转换目标代码。样本中有 64% 的数据属于这个原因。(2) 目前的 GenPat 实现尚不支持一些类型的代码修改，比如修改函数的签名或者同时修改两个函数等。样本中 27% 的数据属于该原因。(3) GenPat 目前支持的代码修改操作不能用于描述期望修改。比如一些示例程序需要在代码的相对位置中插入一些代码，而目前 GenPat 的修改操作只支持在绝对位置插入，不能刻画相对位置。样本中有 3% 的数据属于该原因。(4) 最后，GenPat 没有正确提取修改模板。比如提取的太多的上下文信息导致不能与目标代码正确匹配。6% 的样本数据由于该原因导致错误。需要注意的是，前两个原因并不是本章所提出方法的缺陷，可以通过更好的方法实现对应的功能。而最后两点为进一步改进 GenPat 指出了新的研究方向。

小结：根据实验结果，GenPat 可以有效地从单个代码修改样例中提取代码修改的模板，该技术并不局限于程序缺陷的自动修复，在其他类似的应用场景中具有较强的通用性。但是，根据实验结果可以发现，目前的 GenPat 方法依然有很多改进的空间值得进一步研究，本文将其作为未来研究工作。

5.8 相关工作验证

通过本章的实验验证，本文所提出的面向非频繁缺陷的自动修复方法 IBFix 可以有效修复程序中的非频繁缺陷。特别的，本文所提出的两种补丁生成技术具有互补性，可以有效克服非频繁缺陷修复对大量重复修改的依赖。本节讨论本文研究工作的后续相关研究对本文工作的相关验证。

尽管本文所提出的自动修复工具 SimFix[①]已经发布了两年多 (发布于 2018 年)，2020 年国际旗舰会议 ICSE 上的最新研究[60] 通过对比 16 种最新的缺陷自动修复工具表明，SimFix 不仅实现了正确修复的缺陷数量最多 (25 个[②])，并且其独立修复的缺陷 (其他工具都不能正确修复) 数量也是最多 (11 个)。同时，实验结果表明，SimFix 不会生成任何无意义的 (nonsensical)[③] 修复补丁，而其它绝大部分的修复工具都会产生无意义的修复补丁，其中包括基于人工模板的自动修复技术 kPAR[58]。此外，目前主

[①] https://github.com/xgdsmileboy/SimFix
[②] 该论文将本文中的 5 个小时的修复时间限制修改为最多 10,000 次修复尝试，因此结果有差异。
[③] 该论文中定义使程序不能通过编译的修复补丁为 nonsensical 补丁。

流的缺陷修复方法通过人工验证候选补丁与正确补丁的语义等价性来判断补丁的正确性。2019 年的最新研究表明[98]，当仅考虑候选修复补丁与正确补丁完全一致 (论文中的 SLSM 类别) 时，SimFix 相对于比较的其它 9 个最新修复工具，依然实现了最多的正确修复数量。

上述最新研究中，作者均通过重新运行本文工作的工具获得最新的实验结果，不依赖本文作者开展的实验。从最新的研究可以发现，本文中提出的自动修复技术相比于已有的技术 (包括最新工作) 具有较大的优势，为本文方法的有效性提供了更强的证明。

第六章　结论及展望

6.1　本文工作总结

软件缺陷在软件的开发和维护过程中难免被引入，而修复程序中的缺陷不仅成为了开发者的一项繁重任务，同时也给公司以及用户带来了巨大的经济损失。因此自动化的缺陷修复是一项有希望缓解甚至最终解决上述问题的一项重要研究。

尽管经过了十多年的发展，缺陷修复中的关键问题：非频繁缺陷的自动修复问题，依然缺乏有效的解决方案。本论文围绕非频繁缺陷修复，针对其面临的种类多、重复修复样本少导致修复模板难以学习的重要挑战，提出了**利用代码的频繁性克服非频繁缺陷的修复难题**，并依据此提出了一个面向非频繁缺陷的修复方法 IBFix。具体来讲，本文首先通过实证研究分析了开发者人工修复缺陷的过程，并从中学习针对非频繁缺陷的修复方法，为自动化方法提供有效的指导。基于此，本文提出的缺陷修复方法由三个主要部分组成：(1) 基于状态划分的缺陷定位技术用来提升定位的准确率以满足非频繁缺陷修复对定位提出的更高要求，(2) 基于单个样例的补丁生成技术和 (2) 基于相似代码的补丁生成技术用来克服非频繁缺陷修复中的补丁生成难题。最后，通过系统性的对比实验分析表明，本文所提出的自动修复技术能够有效地修复程序中的非频繁缺陷，克服了对大量重复历史修改的依赖，促进了程序缺陷自动修复技术的实用化进程。值得注意的是，本文所提出的缺陷修复技术不仅仅可以用来修复非频繁的缺陷，同样适用于修复频繁的缺陷类型。根据最新的研究表明，本文所提出的缺陷修复技术不仅优于已有研究，同时相比于后续的最新研究也具有非常大的优势 (正确修复数量依然最多)。

6.2　未来工作展望

(1) 本文方法的后续优化　根据正文的介绍，本文所提出的方法在一定程度上解决了修复非频繁缺陷的挑战，但是目前的缺陷修复方法在修复数量和准确率方面依然效果有限，进一步提升自动化缺陷修复方法依然具有很大的空间。

首先，对于自动化缺陷修复方法。虽然本文提出的方法进一步提升了定位的准确率，但从结果可以看出只有大概 50% 左右的缺陷可以实现正确定位 (候选列表中的第一个)。因此，如何提高缺陷定位的准确率将依然是以后自动化软件调试技术中的一个重要研究方向。从人工修复的实证研究也可以看出，人工调试中的一些定位方法在自动化的定位方法中没有得到充分地利用。此外，重要的是人工调试的过程中通常会结

合不同的策略实现共同定位过程。本文中所提出的定位集成方法在此方向上已经进行了初探，更深入的研究需要未来更多的探索。

其次，对于缺陷修复中的补丁生成。本文提出的技术对于解决非频繁缺陷具有更好的通用性，但从对比实验中可以发现，目前已有的一些通过人工定义修复模板的方式对于修复常见的频繁缺陷类型具有更好的效果。通过结合不同的方法以发挥各自的长处是提升缺陷修复整体实用性的一个重要途径。实际上，本文中所提出的两种修复补丁生成方法虽然验证了其有效性，但依然可以进一步优化。一方面，对于从历史中提取修复修改模板而言，本文的方法仅仅针对单个函数内部的代码修改，不能处理跨函数缺陷修复。在实验中发现，存在很多缺陷需要同时修改多个函数实现正确修复。另一方面，对于基于相似代码的补丁生成方法，现有的代码搜索范围局限在当前的出错项目中。随着开源项目的增加，更多的源代码可以用来辅助修复的过程，但其中要解决的难点是实现针对海量跨项目代码的准确搜索问题。除此之外，本文最后通过一个缺陷修复技术将本文所提出的方法进行集成，其依据同样是基于人工修复中的多策略组合，对其更加深入的探索有希望提升正确修复缺陷的数量和补丁的正确率。

(2) 新的机遇与挑战 事实上，从本文的研究可以发现，通过小样本数据学习通用的代码修改是可行的，甚至可以在某些方面超过主流的基于大量数据的模式学习方法。该发现为我们提供了解决软件工程问题的一个新思路和新的方向：基于小样本学习的软件复用。众所周知，软件开发过程中的很多任务具有重复性，比如本文研究的代码缺陷修复，以及系统性代码修改甚至代码生成等。因此，更好地复用开发过程中的重复数据以提升软件的自动化水平有巨大的潜力，同时具有重大的研究意义和实用价值。但由于代码逻辑的复杂性，不同任务之间会存在一定的差异，导致大部分任务的重复数据量并不可观，是目前实现大规模软件复用的一个关键挑战。然而，基于小样本学习的软件复用技术为该过程提供了新的解决思路，有很大的潜力推动软件复用技术的大规模实用化进程。本文所提出的基于小样本历史修改数据的补丁生成方法在该方向已经进行了初探，并取得了初步的成效。然而，对于更加复杂的软件工程任务，研究和开发基于小样本的高效复用技术依然需要更多的深入探索。

参考文献

[1] Rui Abreu, Peter Zoeteweij and Arjan J. C. van Gemund. "*Spectrum-Based Multiple Fault Localization*". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, **2009**: 88–99. https://doi.org/10.1109/ASE.2009.25.

[2] Rui Abreu, Peter Zoeteweij and Arjan J.c. Van Gemund. "*An Evaluation of Similarity Coefficients for Software Fault Localization*". In: *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. 2006-12: 39–46. https://doi.org/10.1109/PRDC.2006.18.

[3] John Anvik, Lyndon Hiew and Gail C. Murphy. "*Who Should Fix This Bug?*" In: *Proceedings of the 28th International Conference on Software Engineering*. ACM, **2006**: 361–370. https://doi.org/10.1145/1134285.1134336.

[4] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin *et al.* "*Statistical Debugging Using Compound Boolean Predicates*". In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, **2007**: 5–15. https://doi.org/10.1145/1273463.1273467.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer *et al.* "*Using Static Analysis to Find Bugs*". *IEEE Software*, **2008**, *25*(5): 22–29. https://doi.org/10.1109/MS.2008.130.

[6] Tien-Duy B. Le, David Lo, Claire Le Goues *et al.* "*A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants*". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, **2016**: 177–188. https://doi.org/10.1145/2931037.2931049.

[7] Johannes Bader, Andrew Scott, Michael Pradel *et al.* "*Getafix: Learning to Fix Bugs Automatically*". *Proc. ACM Program. Lang.* 2019-10, *3*. https://doi.org/10.1145/3360585.

[8] Earl T. Barr, Yuriy Brun, Premkumar Devanbu *et al.* "*The Plastic Surgery Hypothesis*". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, **2014**: 306–317. https://doi.org/10.1145/2635868.2635898.

[9] George E.P. Box and R. Daniel Meyer. "*An Analysis for Unreplicated Fractional Factorials*". *Technometrics*, **1986**, *28*(1): 11–18. https://doi.org/10.1080/00401706.1986.10488093.

[10] Tom Britton, Lisa Jeng, Graham Carver *et al.* "*Reversible debugging software*". In: *University of Cambridge-Judge Business School, Tech. Rep.* **2013**.

[11] Martin Burger and Andreas Zeller. "*Minimizing Reproduction of Software Failures*". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, **2011**: 221–231. https://doi.org/10.1145/2001420.2001447.

[12] Eduardo C. Campos and Marcelo A. Maia. "*Common Bug-Fix Patterns: A Large-Scale Observational Study*". In: *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, **2017**: 404–413. https://doi.org/10.1109/ESEM.2017.55.

[13] Satish Chandra, Emina Torlak, Shaon Barman *et al.* "*Angelic Debugging*". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, **2011**: 121–130. https://doi.org/10.1145/1985793.1985811.

[14] Chao Liu, Long Fei, Xifeng Yan *et al.* "*Statistical Debugging: A Hypothesis Testing-Based Approach*". *IEEE Transactions on Software Engineering*, **2006**, *32*(10): 831–848. https://doi.org/10.1109/TSE.2006.105.

[15] Liushan Chen, Yu Pei and Carlo A. Furia. "*Contract-Based Program Repair without the Contracts*". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, **2017**: 637–647.

[16] Mike Y. Chen, Emre Kıcıman, Eugene Fratkin *et al.* "*Pinpoint: problem determination in large, dynamic Internet services*". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002-06: 595–604. https://doi.org/10.1109/DSN.2002.1029005.

[17] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra *et al.* "*HOLMES: Effective Statistical Debugging via Efficient Path Profiling*". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, **2009**: 34–44. https://doi.org/10.1109/ICSE.2009.5070506.

[18] Benoit Cornu, Thomas Durieux, Lionel Seinturier *et al.* "*NPEFix: Automatic Runtime Repair of Null Pointer Exceptions in Java*". *ArXiv*, **2015**, *abs/1512.07423*.

[19] Valentin Dallmeier, Christian Lindig and Andreas Zeller. "*Lightweight Defect Localization for Java*". In: *Proceedings of the 2005 European Conference on Object-Oriented Programming*. Springer, **2005**: 528–550.

[20] W. Eric Wong, Vidroha Debroy and Byoungju Choi. "*A Family of Code Coverage-Based Heuristics for Effective Fault Localization*". *J. Syst. Softw.* 2010-02, *83*(2): 188–208. https://doi.org/10.1016/j.jss.2009.09.037.

[21] Michael D. Ernst, Jake Cockrell, William G. Griswold *et al.* "*Dynamically discovering likely program invariants to support program evolution*". *IEEE Transactions on Software Engineering*, 2001-02, *27*(2): 99–123. https://doi.org/10.1109/32.908957.

[22] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc *et al.* "*Fine-Grained and Accurate Source Code Differencing*". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, **2014**: 313–324. https://doi.org/10.1145/2642937.2642982.

[23] Beat Fluri, Michael Wuersch, Martin PInzger *et al.* "*Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction*". *IEEE Transactions on Software Engineering*, 2007-11, *33*(11): 725–743. https://doi.org/10.1109/TSE.2007.70731.

[24] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer *et al.* "*A Genetic Programming Approach to Automated Software Repair*". In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, **2009**: 947–954. https://doi.org/10.1145/1569901.1570031.

[25] Mark Gabel and Zhendong Su. "*A Study of the Uniqueness of Source Code*". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, **2010**: 147–156. https://doi.org/10.1145/1882291.1882315.

[26] Qing Gao, Yingfei Xiong, Yaqing Mi *et al.* "*Safe Memory-Leak Fixing for C Programs*". In: *Proceedings of the 37th International Conference on Software Engineering*. **2015**: 459–470.

[27] Qing Gao, Hansheng Zhang, Jie Wang *et al.* "*Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T)*". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. 2015-11: 307–318. https://doi.org/10.1109/ASE.2015.81.

[28] Luca Gazzola, Daniela Micucci and Leonardo Mariani. "*Automatic Software Repair: A Survey*". *IEEE Transactions on Software Engineering*, **2019**, *45*(1): 34–67. https://doi.org/10.1109/TSE.2017.2755013.

[29] Ali Ghanbari, Samuel Benton and Lingming Zhang. "*Practical Program Repair via Bytecode Mutation*". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, **2019**: 19–30. https://doi.org/10.1145/3293882.3330559.

[30] Ákos Hajnal and István Forgács. "*A precise demand-driven definition-use chaining algorithm*". In: *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. 2002-03: 77–86. https://doi.org/10.1109/CSMR.2002.995792.

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

[32] Mary Jean Harrold and Mary Lou Soffa. "*Efficient Computation of Interprocedural Definition-Use Chains*". *ACM Trans. Program. Lang. Syst.* 1994-03, *16*(2): 175–204. https://doi.org/10.1145/174662.174663.

[33] Abram Hindle, Earl T. Barr, Zhendong Su *et al.* "*On the Naturalness of Software*". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, **2012**: 837–847.

[34] Jinru Hua, Mengshi Zhang, Kaiyuan Wang *et al.* "*Towards Practical Program Repair with On-Demand Candidate Generation*". In: *Proceedings of the 40th International Conference on Software Engineering*. ACM, **2018**: 12–23. https://doi.org/10.1145/3180155.3180245.

[35] Patricia Jablonski and Daqing Hou. "*CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE*". In: *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange*. ACM, **2007**: 16–20. https://doi.org/10.1145/1328279.1328283.

[36] Dennis Jeffrey, Neelam Gupta and Rajiv Gupta. "*Fault Localization Using Value Replacement*". In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ACM, **2008**: 167–178. https://doi.org/10.1145/1390630.1390652.

[37] Tao Ji, Liqian Chen, Xiaoguang Mao *et al.* "*Automated Program Repair by Using Similar Code Containing Fix Ingredients*". In: *Proceedings of the IEEE 40th Annual Computer Software and Applications Conference*. **2016**: 197–202. https://doi.org/10.1109/COMPSAC.2016.69.

[38] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su *et al.* "*DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones*". In: *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, **2007**: 96–105. https://doi.org/10.1109/ICSE.2007.30.

[39] Lingxiao Jiang and Zhendong Su. "*Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths*". In: *Proceedings of the Twenty-Second IEEE/ACM International*

Conference on Automated Software Engineering. ACM, **2007**: 184–193. https://doi.org/10.1145/1321631.1321660.

[40] Guoliang Jin, Linhai Song, Xiaoming Shi *et al.* "*Understanding and Detecting Real-World Performance Bugs*". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, **2012**: 77–88. https://doi.org/10.1145/2254064.2254075.

[41] James A. Jones, James F. Bowring and Mary Jean Harrold. "*Debugging in Parallel*". In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, **2007**: 16–26. https://doi.org/10.1145/1273463.1273468.

[42] James A. Jones and Mary Jean Harrold. "*Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique*". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, **2005**: 273–282. https://doi.org/10.1145/1101908.1101949.

[43] James A. Jones, Mary Jean Harrold and John Stasko. "*Visualization of Test Information to Assist Fault Localization*". In: *Proceedings of the 24th International Conference on Software Engineering*. ACM, **2002**: 467–477. https://doi.org/10.1145/581339.581397.

[44] René Just, Darioush Jalali and Michael D. Ernst. "*Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs*". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, **2014**: 437–440. https://doi.org/10.1145/2610384.2628055.

[45] Yalin Ke, Kathryn T. Stolee, Claire Le Goues *et al.* "*Repairing Programs with Semantic Code Search*". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, **2015**: 295–306. https://doi.org/10.1109/ASE.2015.60.

[46] Dongsun Kim, Jaechang Nam, Jaewoo Song *et al.* "*Automatic Patch Generation Learned from Human-Written Patches*". In: *Proceedings of the 2013 International Conference on Software Engineering*. **2013**: 802–811.

[47] Sunghun Kim, Kai Pan and E. E. James Whitehead. "*Memories of Bug Fixes*". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, **2006**: 35–45. https://doi.org/10.1145/1181775.1181781.

[48] Pavneet Singh Kochhar, Xin Xia, David Lo *et al.* "*Practitioners' Expectations on Automated Fault Localization*". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, **2016**: 165–176. https://doi.org/10.1145/2931037.2931051.

[49] Bogdan Korel and Janusz Laski. "*Dynamic program slicing*". *Information processing letters*, **1988**, *29*(3): 155–163.

[50] Patrick Kreutzer, Georg Dotzler, Matthias Ring *et al.* "*Automatic Clustering of Code Changes*". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, **2016**: 61–72. https://doi.org/10.1145/2901739.2901749.

[51] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest *et al.* "*GenProg: A Generic Method for Automatic Software Repair*". *IEEE Transactions on Software Engineering*, 2012-01, *38*(1): 54–72. https://doi.org/10.1109/TSE.2011.104.

[52] Xuan Bach D. Le, David Lo and Claire Le Goues. "*History Driven Program Repair*". In: *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. 2016-03: 213–224. https://doi.org/10.1109/SANER.2016.76.

[53] Xia Li, Wei Li, Yuqun Zhang *et al.* "*DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization*". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, **2019**: 169–180. https://doi.org/10.1145/3293882.3330574.

[54] Zhenmin Li, Shan Lu, Suvda Myagmar *et al.* "*CP-Miner: finding copy-paste and related bugs in large-scale software code*". *IEEE Transactions on Software Engineering*, 2006-03, *32*(3): 176–192. https://doi.org/10.1109/TSE.2006.28.

[55] Ben Liblit, Alex Aiken, Alice X. Zheng *et al.* "*Bug Isolation via Remote Program Sampling*". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM, **2003**: 141–154. https://doi.org/10.1145/781131.781148.

[56] Ben Liblit, Mayur Naik, Alice X. Zheng *et al.* "*Scalable Statistical Bug Isolation*". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, **2005**: 15–26. https://doi.org/10.1145/1065010.1065014.

[57] Chao Liu, Xifeng Yan, Long Fei *et al.* "*SOBER: Statistical Model-Based Bug Localization*". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, **2005**: 286–295. https://doi.org/10.1145/1081706.1081753.

[58] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé *et al.* "*You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems*". In: *Proceedings of 12th IEEE Conference on Software Testing, Validation and Verification*. IEEE, **2019**: 102–113. https://doi.org/10.1109/ICST.2019.00020.

[59] Kui Liu, Anil Koyuncu, Dongsun Kim *et al.* "*TBar: Revisiting Template-Based Automated Program Repair*". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, **2019**: 31–42. https://doi.org/10.1145/3293882.3330577.

[60] Kui Liu, Shangwen Wang, Anil Koyuncu *et al.* "*On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs*". In: *Proceedings of the ACM/IEEE International Conference on Software Engineering*. **2020**.

[61] Fan Long, Peter Amidon and Martin Rinard. "*Automatic Inference of Code Transforms for Patch Generation*". In: *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, **2017**: 727–739. https://doi.org/10.1145/3106237.3106253.

[62] Fan Long and Martin Rinard. "*Staged Program Repair with Condition Synthesis*". In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, **2015**: 166–178. https://doi.org/10.1145/2786805.2786811.

[63] Fan Long and Martin Rinard. "*Automatic Patch Generation by Learning Correct Code*". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, **2016**: 298–312. https://doi.org/10.1145/2837614.2837617.

[64] Matias Martinez, Thomas Durieux, Romain Sommerard *et al.* "*Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset*". *Empirical Software Engineering*, **2016**, *22*: 1936–1964.

[65] Matias Martinez and Martin Monperrus. "*Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing*". *Empirical Softw. Engg.* 2015-02, *20*: 176–205. https://doi.org/10.1007/s10664-013-9282-8.

[66] Matias Martinez, Westley Weimer and Martin Monperrus. "*Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches*". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, **2014**: 492–495. https://doi.org/10.1145/2591062.2591114.

[67] Sergey Mechtaev, Jooyong Yi and Abhik Roychoudhury. "*DirectFix: Looking for Simple Program Repairs*". In: *Proceedings of the 37th International Conference on Software Engineering*. **2015**: 448–458.

[68] Sergey Mechtaev, Jooyong Yi and Abhik Roychoudhury. "*Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis*". In: *Proceedings of the 38th International Conference on Software Engineering*. ACM, **2016**: 691–701. https://doi.org/10.1145/2884781.2884807.

[69] Na Meng, Miryung Kim and Kathryn S. McKinley. "*Systematic Editing: Generating Program Transformations from an Example*". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, **2011**: 329–342. https://doi.org/10.1145/1993498.1993537.

[70] Ghassan Misherghi and Zhendong Su. "*HDD: Hierarchical Delta Debugging*". In: *Proceedings of the 28th International Conference on Software Engineering*. ACM, **2006**: 142–151. https://doi.org/10.1145/1134285.1134307.

[71] Martin Monperrus. "*Automatic Software Repair: A Bibliography*". *ACM Comput. Surv.* 2018-01, *51*(1). https://doi.org/10.1145/3105906.

[72] Seokhyeon Moon, Yunho Kim, Moonzoo Kim *et al.* "*Ask the Mutants: Mutating Faulty Programs for Fault Localization*". In: *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014-03: 153–162. https://doi.org/10.1109/ICST.2014.28.

[73] Lee Naish, Hua Jie Lee and Kotagiri Ramamohanarao. "*A Model for Spectra-Based Software Diagnosis*". *ACM Trans. Softw. Eng. Methodol.* 2011-08, *20*(3). https://doi.org/10.1145/2000791.2000795.

[74] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury *et al.* "*SemFix: Program repair via semantic analysis*". In: *Proceedings of the 35th International Conference on Software Engineering*. 2013-05: 772–781. https://doi.org/10.1109/ICSE.2013.6606623.

[75] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham *et al.* "*Recurring Bug Fixes in Object-Oriented Programs*". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, **2010**: 315–324. https://doi.org/10.1145/1806799.1806847.

[76] Thanaporn Ongkosit and Shingo Takada. "*Responsiveness Analysis Tool for Android Application*". In: *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*. ACM, **2014**: 1–4. https://doi.org/10.1145/2661694.2661695.

[77] Mike Papadakis and Yves Le Traon. "*Using Mutants to Locate "Unknown" Faults*". In: *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012-04: 691–700. https://doi.org/10.1109/ICST.2012.159.

[78] Mike Papadakis and Yves Le Traon. "*Metallaxis-FL: Mutation-Based Fault Localization*". *Softw. Test. Verif. Reliab.* 2015-08, *25*(5–7): 605–628. https://doi.org/10.1002/stvr.1509.

[79] Spencer Pearson, José Campos, René Just *et al.* "*Evaluating and Improving Fault Localization*". In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, **2017**: 609–620. https://doi.org/10.1109/ICSE.2017.62.

[80] Long H. Pham, Jun Sun, Ly Ly Tran Thi *et al.* "*Learning Likely Invariants to Explain Why a Program Fails*". In: *Proceedings of the 22nd International Conference on Engineering of Complex Computer Systems*. 2017-11: 70–79. https://doi.org/10.1109/ICECCS.2017.12.

[81] Yuhua Qi, Xiaoguang Mao, Yan Lei *et al.* "*The Strength of Random Search on Automated Program Repair*". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, **2014**: 254–265. https://doi.org/10.1145/2568225.2568254.

[82] Zichao Qi, Fan Long, Sara 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*. ACM, **2015**: 24–36. https://doi.org/10.1145/2771783.2771791.

[83] Foyzur Rahman, Daryl Posnett, Abram Hindle *et al.* "*BugCache for Inspections: Hit or Miss?*" In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, **2011**: 322–331. https://doi.org/10.1145/2025113.2025157.

[84] Santosh S. Rathore and Sandeep Kumar. "*Predicting Number of Faults in Software System using Genetic Programming*". In: *Proceedings of the 2015 International Conference on Soft Computing and Software Engineering*. **2015**: 303–311. http://www.sciencedirect.com/science/article/pii/S1877050915025892.

[85] Baishakhi Ray and Miryung Kim. "*A Case Study of Cross-System Porting in Forked Projects*". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, **2012**. https://doi.org/10.1145/2393596.2393659.

[86] Thomas Reps, Thomas Ball, Manuvir Das *et al.* "*The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem*". In: *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Springer, **1997**: 432–449. https://doi.org/10.1145/267895.267925.

[87] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida *et al.* "*ELIXIR: Effective Object Oriented Program Repair*". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, **2017**: 648–659.

[88] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg *et al.* "*Do stack traces help developers fix bugs?*" In: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010-05: 118–121. https://doi.org/10.1109/MSR.2010.5463280.

[89] Robert C. Seacord, Daniel Plakosh and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. USA: Addison-Wesley Longman Publishing Co., Inc., **2003**.

[90] Jeongju Sohn and Shin Yoo. "*FLUCCS: Using Code and Change Metrics to Improve Fault Localization*". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, **2017**: 273–283. https://doi.org/10.1145/3092703.3092717.

[91] Ezekiel O Soremekun, Marcel Böhme and Andreas Zeller. "*Programmers should still use slices when debugging*". *Technical Report*, **2016**.

[92] Mauricio Soto, Ferdian Thung, Chu-Pan Wong *et al.* "*A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions*". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, **2016**: 512–515. https://doi.org/10.1145/2901739.2903495.

[93] Manu Sridharan, Stephen J Fink and Rastislav Bodik. "*Thin slicing*". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. **2007**: 112–122.

[94] Symantec. "*Internet security threat report*". 2006-09. http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf.

[95] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad *et al.* "*Anti-Patterns in Search-Based Program Repair*". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, **2016**: 727–738. https://doi.org/10.1145/2950290.2950295.

[96] Michele Tufano, Cody Watson, Gabriele Bavota *et al.* "*An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation*". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, **2018**: 832–837. https://doi.org/10.1145/3238147.3240732.

[97] F. Umemori, K. Konda, R. Yokomori *et al.* "*Design and implementation of bytecode-based Java slicing system*". In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. **2003**: 108–117. https://doi.org/10.1109/SCAM.2003.1238037.

[98] Shangwen Wang, Ming Wen, Liqian Chen *et al.* "*How Different Is It Between Machine-Generated and Developer-Provided Patches? : An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques*". In: **2019**: 1–12.

[99] Shangwen Wang, Ming Wen, Xiaoguang Mao *et al.* "*Attention Please: Consider Mockito When Evaluating Newly Proposed Automated Program Repair Techniques*". In: *Proceedings of the Evaluation and Assessment on Software Engineering*. ACM, **2019**: 260–266. https://doi.org/10.1145/3319008.3319349.

[100] Tao Wang and Abhik Roychoudhury. "*Using Compressed Bytecode Traces for Slicing Java Programs*". In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, **2004**: 512–521.

[101] Yi Wei, Yu Pei, Carlo A. Furia *et al.* "*Automated Fixing of Programs with Contracts*". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, **2010**: 61–72. https://doi.org/10.1145/1831708.1831716.

[102] Westley Weimer, Zachary P. Fry and Stephanie Forrest. "*Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results*". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. **2013**: 356–366. https://doi.org/10.1109/ASE.2013.6693094.

[103] Mark Weiser. "*Program slicing*". *IEEE Transactions on software engineering*, **1984**(4): 352–357.

[104] Ming Wen, Junjie Chen, Rongxin Wu *et al.* "*Context-Aware Patch Generation for Better Automated Program Repair*". In: *Proceedings of the 40th International Conference on Software Engineering*. ACM, **2018**: 1–11. https://doi.org/10.1145/3180155.3180233.

[105] Martin White, Michele Tufano, Matias Martinez *et al.* "*Sorting and transforming program repair ingredients via deep learning code similarities*". In: *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. **2019**: 479–490.

[106] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang *et al.* "*Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis*". In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, **2014**: 181–190. https://doi.org/10.1109/ICSME.2014.40.

[107] Eric Wong, Tingting Wei, Yu Qi *et al.* "*A Crosstab-based Statistical Method for Effective Fault Localization*". In: *Proceedings of 1st International Conference on Software Testing, Verification, and Validation*. **2008**: 42–51. https://doi.org/10.1109/ICST.2008.65.

[108] W. Eric Wong, Vidroha Debroy, Ruizhi Gao *et al.* "*The DStar Method for Effective Software Fault Localization*". *IEEE Transactions on Reliability*, **2014**, *63*: 290–308. https://doi.org/10.1109/TR.2013.2285319.

[109] W. Eric Wong, Vidroha Debroy, Richard Golden *et al.* "*Effective Software Fault Localization Using an RBF Neural Network*". *IEEE Transactions on Reliability*, 2012-03, *61*(1): 149–169. https://doi.org/10.1109/TR.2011.2172031.

[110] W. Eric Wong, Vidroha Debroy and Dianxiang Xu. "*Towards Better Fault Localization: A Crosstab-Based Statistical Approach*". *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 2012-05, *42*(3): 378–396. https://doi.org/10.1109/TSMCC.2011.2118751.

[111] W. Eric Wong, Ruizhi Gao, Yihao Li *et al.* "*A Survey on Software Fault Localization*". *IEEE Trans. Softw. Eng.* 2016-08, *42*(8): 707–740. https://doi.org/10.1109/TSE.2016.2521368.

[112] Qi Xin and Steven 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*. ACM, **2017**: 226–236. https://doi.org/10.1145/3092703.3092718.

[113] Qi Xin and Steven P. Reiss. "*Leveraging Syntax-Related Code for Automated Program Repair*". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, **2017**: 660–670.

[114] Yingfei Xiong, Xinyuan Liu, Muhan Zeng *et al.* "*Identifying Patch Correctness in Test-Based Program Repair*". In: *Proceedings of the 40th International Conference on Software Engineering*. ACM, **2018**: 789–799. https://doi.org/10.1145/3180155.3180182.

[115] Yingfei Xiong, Jie Wang, Runfa Yan *et al.* "*Precise Condition Synthesis for Program Repair*". In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, **2017**: 416–426. https://doi.org/10.1109/ICSE.2017.45.

[116] Tianyin Xu, Xinxin Jin, Peng Huang *et al.* "*Early Detection of Configuration Errors to Reduce Failure Damage*". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, **2016**: 619–634.

[117] Jifeng Xuan, Matias Martinez, Favio Demarco *et al.* "*Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs*". *IEEE Transactions on Software Engineering*, 2017-01, *43*(1): 34–55. https://doi.org/10.1109/TSE.2016.2560811.

[118] Jifeng Xuan and Martin Monperrus. "*Learning to Combine Multiple Ranking Metrics for Fault Localization*". In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. 2014-09: 191–200. https://doi.org/10.1109/ICSME.2014.41.

[119] Jifeng Xuan and Martin Monperrus. "*Test Case Purification for Improving Fault Localization*". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, **2014**: 52–63. https://doi.org/10.1145/2635868.2635906.

[120] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu *et al.* "*Better Test Cases for Better Automated Program Repair*". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, **2017**: 831–841. https://doi.org/10.1145/3106237.3106274.

[121] Cemal Yilmaz, Amit Paradkar and Clay Williams. "*Time Will Tell: Fault Localization Using Time Spectra*". In: *Proceedings of the 30th International Conference on Software Engineering*. ACM, **2008**: 81–90. https://doi.org/10.1145/1368088.1368100.

[122] Zuoning Yin, Ding Yuan, Yuanyuan Zhou *et al.* "*How Do Fixes Become Bugs?*" In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, **2011**: 26–36. https://doi.org/10.1145/2025113.2025121.

[123] Haruki Yokoyama, Yoshiki Higo, Keisuke Hotta *et al.* "*Toward Improving Ability to Repair Bugs Automatically: A Patch Candidate Location Mechanism Using Code Similarity*". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, **2016**: 1364–1370. https://doi.org/10.1145/2851613.2851770.

[124] Ruru Yue, Na Meng and Qianxiang Wang. "*A Characterization Study of Repeated Bug Fixes*". In: *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*. **2017**: 422–432. http://doi.org/10.1109/ICSME.2017.16.

[125] Andreas Zeller. "*Isolating Cause-Effect Chains from Computer Programs*". In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, **2002**: 1–10. https://doi.org/10.1145/587051.587053.

[126] Andreas Zeller and Ralf Hildebrandt. "*Simplifying and isolating failure-inducing input*". *IEEE Transactions on Software Engineering*, 2002-02, *28*(2): 183–200.

[127] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Locating Faults through Automated Predicate Switching*". In: *Proceedings of the 28th International Conference on Software Engineering*. ACM, **2006**: 272–281. https://doi.org/10.1145/1134285.1134324.

[128] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Pruning Dynamic Slices with Confidence*". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, **2006**: 169–180. https://doi.org/10.1145/1133981.1134002.

[129] Alice X. Zheng, Michael I. Jordan, Ben Liblit *et al.* "*Statistical Debugging: Simultaneous Identification of Multiple Bugs*". In: *Proceedings of the 23rd International Conference on Machine Learning*. ACM, **2006**: 1105–1112. https://doi.org/10.1145/1143844.1143983.

[130] Hao Zhong and Hong Mei. "*Mining repair model for exception-related bug*". *J. Syst. Softw.* **2018**, *141*: 16–31.

[131] Jian Zhou, Hongyu Zhang and David Lo. "*Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports*". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, **2012**: 14–24.

[132] Daming Zou, Jingjing Liang, Yingfei Xiong *et al.* "*An Empirical Study of Fault Localization Families and Their Combinations*". *IEEE DamingTransactions on Software Engineering*, **2019**. https://doi.org/10.1109/TSE.2019.2892102.

[133] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo *et al.* "*Low-Overhead and Fully Automated Statistical Debugging with Abstraction Refinement*". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, **2016**: 881–896. https://doi.org/10.1145/2983990.2984005.

[134] 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*(2).

[135] 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*(3): 588–610.

[136] 文万志, 李必信, 孙小兵 *et al.* "一种基于层次切片谱的软件错误定位技术". 软件学报, **2013**, *24*(5): 977–992. https://doi.org/10.3724/SP.J.1001.2013.04342.

个人简历及博士期间研究成果

个人简历

姜佳君，1992 年 01 月 04 日出生于辽宁省建昌县，2011 年 09 月考入西北工业大学计算机学院，专业为计算机科学与技术。2015 年 07 月本科毕业，并获得工学学士学位。2015 年 09 月保送进入北京大学信息科学技术学院攻读计算机软件与理论专业博士学位至今。

发表论文

[1] **Jiajun Jiang**, Luyao Ren, Yingfei Xiong, Lingming Zhang. "Inferring Program Transformations From Singular Examples via Big Code". In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. **ASE 2019**, *San Diego, CA, USA. November 11-15, 2019, pp. 255-266.* https://doi.org/10.1109/ASE.2019.00033

[2] **Jiajun Jiang**, Ran Wang, Yingfei Xiong, Xiangping Chen, Lu Zhang. "Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study". In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. **ASE 2019**, *San Diego, CA, USA. November 11-15, 2019, pp. 502-514.* https://doi.org/10.1109/ASE.2019.00054

[3] **Jiajun Jiang**, Yingfei Xiong, Xin Xia. "A manual inspection of Defects4J bugs and its implications for automatic program repair". In: *Science China Information Sciences* 62.10 (2019): 200102. https://doi.org/10.1007/s11432-018-1465-6

[4] **Jiajun Jiang**, Yingfei Xiong, Hongyu Zhang, Qing Gao, Xiangqun Chen. "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, Netherlands. July 16-21, 2018, pp. 298-309.* https://doi.org/10.1145/3213846.3213871

[5] Chenglong Wang, **Jiajun Jiang**, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, Zhenjiang Hu. "Transforming Programs between APIs with Many-to-Many Mappings". In: *Proceedings of the 30th European Conference on Object-Oriented Programming*. **ECOOP 2016**, *Rome, Italy. July 18-22, 2016, pp. 25:1–25:26.* https:

//doi.org/10.4230/LIPIcs.ECOOP.2016.25

[6] 王博, 卢思睿, **姜佳君**, 熊英飞. ‘‘基于动态分析的软件不变量综合技术综述”. *软件学报, (2020) 31(6),* https://doi.org/10.13328/j.cnki.jos.006014

申请专利

[1] **姜佳君**，任路遥，熊英飞。基于单个样例的代码转换推导方法和装置。申请号 (已公布)：201910861206.6

参与课题

[1] 数据驱动的软件自动构造与演进方法研究，国家重点研发计划课题，No. 2017YFB1001803。

[2] 软件维护，国家自然科学基金优秀青年基金项目，No. 61922003。

[3] 软件缺陷修复动作的识别与推荐，国家自然科学基金面上项目，No. 61672045。

致谢

白驹过隙，时光荏苒，从 2015 年 9 月至今，近五年的燕园生活给我留下了太多美好而深刻的记忆。回首燕园故事，记忆犹新，往事历历在目，感慨万千。它也注定将是我漫长人生路上一道亮丽的风景线，我会把这段记忆好好珍藏。回顾博士生涯，一点一滴的进步都离不开老师、同学、朋友以及家人的支持和帮助。是你们的鼓励，让我以一种积极乐观的心态去迎接未知的挑战。

首先，衷心感谢杨芙清院士，以及她所领导的研究团队，为我提供了优秀的学习环境以及浓厚的科研氛围。感谢梅宏院士，让我有机会接触最前沿的科学研究、与世界资深的学者沟通和交流、领略科研的魅力。能在此团队中度过我的博士研究生活，我感到无比荣幸和万分感谢。

衷心感谢我的导师熊英飞副教授。从入学到博士论文答辩，熊老师付出了大量的精力悉心教我科研的方法，手把手帮我修改论文，指导我报告的技巧，使我从一个科研小白逐渐开始掌握了科学研究的方法。熊老师对学术的执著和热爱一直感染着我在科研的道路上不断自我完善，其缜密的逻辑思维和严谨的治学态度将一直指导我未来的科研之路。熊老师生活中积极乐观的人生态度和对学生的关心，更使我在面对挫折时一往无前、砥砺前行。

衷心感谢陈向群教授。陈老师在我的博士研究期间给了我很多建议和指导。每天到实验室都能见到陈老师已早早开始工作，陈老师对科研的热爱以及对学生的关心一直影响着我，让我在科研的道路上时刻保持一颗求知若渴的心。衷心感谢胡振江教授和张路教授在科研道路上为我指导方向，指导我修改论文，给了我很多宝贵的意见，让我的科研能力不断进步。他们严谨的科研态度和广博的专业知识让我深深敬佩，激励着我不断努力奋进。而两位老师低调谦和的处世之道更感染着我在生活中要学会谦卑。

衷心感谢张洪宇教授 (澳大利亚纽卡斯尔大学) 以及张令明教授 (美国德克萨斯大学达拉斯分校)。有幸到两位老师小组访问学习是我博士学习中的宝贵经历。他们严谨的治学态度以及敏捷的科研洞察力令我折服，让我不断地虚心学习。在讨论和交流中，他们的宝贵意见也指导我在科研道路上不断进步。

衷心感谢微软亚洲研究院的张冬梅院长、林庆维老师。在实习期间他们给了我很多宝贵的意见，包括科学的研究方法以及未来的人生规划。感谢亚研院 DKI 小组的所有老师和同学，感谢你们的支持和帮助使我在 3 个多月的实习期间收获颇丰。

衷心感谢金芝教授、郝丹副教授、黄罡教授、谢涛教授、谢冰教授、王千祥教授、郭耀教授、焦文品教授、曹东刚教授、王亚沙教授、刘譞哲副教授、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过程中，给予了我很多宝贵的指导意见，让我不断进步。

衷心感谢我的师兄李军博士，师兄指导我完成了本科的毕业设计，开启了我的博士科研生涯。此外，本博士论文的完成也离不来很多合作老师的指导以及同学的帮助：感谢陈向群老师、张洪宇老师和高庆师兄对本文第4.5节内容的指导；感谢张路老师和陈湘萍老师对本文第4.3节内容的指导；感谢张令明老师对本文第4.4节内容的指导；感谢夏鑫老师对本文第三章内容的指导；感谢王然同学帮助共同完成本文第4.3.3节中的实验分析，感谢任路遥同学帮助共同完成本文第5.7节的实验验证。此外，感谢所有一起合作过的同学：陈俊洁、李夏、王博、梁晶晶、汪成龙、罗翔宇、章嘉晨，感谢你们对我科研上的帮助。感谢编程语言和开发环境小组的悦茹茹、米亚晴、王杰、杨小东、刘鑫远、曾沐焓、陈逸凡。感谢软件所其他小组的马郓、张洁、唐浩、臧琳飞、徐梦炜、邹达明、王冠成、孙泽宇、娄一翎、陈震鹏、李丰、朱琪豪。在学习期间，他们都给过我很多帮助和支持，与他们交往的日子是我生命中的宝贵财富。

衷心感谢我的家人。感谢爸爸、妈妈、哥哥、嫂子在我读博期间一直的支持和鼓励。在我人生重大的决定中帮助我分析、给予我支持。他们无微不至的关怀让我有勇气去面对生活和学习中的各种挫折和挑战。感谢我的爱人马莹，在我读博期间，她对我贴心的照顾与理解让我无后顾之忧。在困惑时，她的开导使我豁然开朗，维持乐观豁达的心态迎接每一个挑战。

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：姜任君 日期：2020年 5月24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：姜任君 导师签名：

日期：2020年 5月 24日