北京大学

博士研究生学位论文

题目：基于程序估计的缺陷自动修复方法研究

姓　　名：王博

学　　号：1501110654

院　　系：软件与微电子学院

专　　业：软件工程

研究方向：软件分析与测试

导　　师：熊英飞 副教授

二〇二〇年十二月

版权声明

摘要

随着软件规模和复杂程度的迅速增长，提升软件的质量和可靠性的代价也急剧上升。相关研究表明，软件缺陷的调试与修复往往需要占用软件开发一半以上的时间。为了缓解这一问题，研究者提出基于“生成-验证”模式的软件缺陷修复技术。

“生成-验证”模式的软件缺陷修复技术在生成补丁之后使用规约来验证补丁的正确性。该类技术的输入为一个有缺陷的程序以及一组规约，输出为一个补丁列表。其中规约用于描述程序的正确性，并且能揭示程序中的缺陷。通过对缺陷程序应用修复技术输出的补丁，可以使得该程序满足规约。常见的规约形式有测试和约束。在现有方法中，测试主要采用单元测试的形式，即描述程序应满足的输入和输出的对应关系；约束主要采用程序正确性属性，即程序语义或状态应满足的逻辑公式。缺陷修复技术可以根据补丁验证阶段所依赖的规约种类进行分类。其中，以补丁通过测试集为标准的为*基于测试的修复*；以补丁满足给定约束为标准的为*基于约束的修复*。

为了提升缺陷自动修复方法的性能，研究者从补丁生成和补丁验证等多方面展开研究。然而，缺陷自动修复技术依然面临以下两大挑战：

1. **补丁过拟合**问题；
2. **修复效率不高**的问题。

*补丁过拟合*是指由于规约不完全，修复方法生成了满足了规约但是错误的补丁，即补丁拟合在了给定规约上。已有研究表明，基于测试的修复和基于约束的修复都存在严重的过拟合问题。为了缓解过拟合问题，已有方法试图引入统计信息指导补丁的生成过程，例如高精度修复的代表方法 ACS[1] 和 CapGen[2] 利用补丁原料在开源代码仓库中出现的频率对原料进行排序。然而，已有缓解过拟合的方法主要针对特定类型的缺陷，缺乏系统性方法。

*修复效率不高*是由修复方法内在的计算复杂性导致的。一方面，基于测试的修复依赖运行测试集检验补丁，即应用补丁后编译软件并测试，检查补丁是否能通过全部测试。在真实软件中，该过程需要一定的计算量，代价不可忽略。在修复实践中，往往需要验证大量补丁才会遇到通过验证的补丁。另一方面，基于约束的修复通常使用符号执行、程序的动态踪迹等代价较高手段提取约束并将约束传播到修复位置。当程序规模较大时，该部分会存在严重的效率问题。

为了针对性解决上述挑战，本文首先定义了程序估计问题。程序估计问题目的是求出给定上下文中概率最大且满足规约的程序。在给定修复位置，以周围代码为上下文和程序正确性为规约，可以通过求解程序估计问题得到补丁，并替换修复位置的代

码来实现修复。因此，缺陷修复问题可以在程序估计问题框架内解决。程序估计问题主要包括以下四个子问题：(1) 定义搜索空间；(2) 赋以程序概率；(3) 查找空间中概率最大的程序；(4) 高效验证程序。为了更精准、更高效地生成程序，本文提出基于数据驱动的玲珑框架依次解决上述问题。

具体来说，本文针对缺陷自动修复中的基于测试的修复和基于约束的修复两个子问题展开研究，以缓解缺陷修复技术面临的两大挑战。

针对补丁过拟合问题，本文研究将基于测试的修复和基于约束的修复归约为程序估计问题的方法。本文在玲珑框架中解决程序估计问题的前三个子问题，从而能够返回在给定上下文中概率最大的补丁。之后，本文提出修复方法 Hanabi 和 ExtractFix，分别将基于测试的修复和基于约束的修复归约为程序估计问题，以生成高质量的补丁，最终缓解补丁过拟合问题。

针对效率不高的问题，本文在玲珑框架中提出了通过静态分析尽早地判断一个部分程序无法生成满足规约的程序，通过剪枝提高搜索效率；针对基于测试的修复提出加速方法 AccMut，约减通过测试验证补丁时的冗余计算；针对基于约束的修复在 ExtractFix 中提出了基于模板的约束提取和基于受限符号执行的约束传播方法，提高约束的提取以及传播的效率。

本文的主要研究工作以及创新如下：

一、本文定义了程序估计问题，并提出了一种基于数据驱动解决方案：**玲珑框架**。玲珑框架需要提供程序代码语料库作为训练集，统计学习出代码中的上下文与程序片段的关系，并以此估计代码片段的概率。具体而言，玲珑框架包含四个组成部分：(1) **扩展规则**，是上下文无关文法的泛化，支持程序按不同方向展开，而基于扩展规则的扩展树是抽象语法树的泛化版本，描述了程序的展开步骤；(2) **统计学习模型**，赋予按照扩展规则展开的概率；(3) **搜索算法**，将程序空间表现为顶点为程序、扩展规则为边的有向图，将程序搜索建模为路径查找问题；(4) **静态分析剪枝**，通过静态分析将不满足约束的部分程序排除。

二、提出了将基于测试的程序自动修复归约为程序估计问题的方法 **Hanabi**，用于修复 Java 程序的条件语句缺陷。即以给定缺陷代码位置代码为上下文，以类型约束和程序规模为规约，求解得到候选补丁。在大型程序中测试难以归约为针对补丁的规约，因此 Hanabi 需要运行测试验证补丁。实验表明，Hanabi 在针对条件语句的修复上，准确率和召回率均显著优于现有技术，并且成功修复了 8 个现有方法无法修复的缺陷。

三、提出了加速基于测试的自动修复的验证的方法 **AccMut**。为了提升基于测试验证阶段的执行效率，本文提出一种动态分析方法 AccMut 以削减测试执行过程中的冗余计算。本文提出了状态同余关系，将同一位置的补丁分组，组内可以共享计算。实验表明，AccMut 相比最新的分支流执行技术加速 2.56 倍。

四、提出了将基于约束的程序自动修复归约为程序估计的方法 **ExtractFix**。本文针对几种可提取约束的崩溃缺陷将其归约为程序估计问题。ExtractFix 根据崩溃信息基于模板提取约束后，通过依赖分析确定修复位置，再用受限的符号执行将约束传播至修复位置。之后，ExtractFix 以修复位置程序为上下文，以约束为规约，将补丁生成问题归约为程序估计问题。实验表明，ExtractFix 可以针对内存越界、整数溢出等几种类型的缺陷提取约束并修复，其修复效果和效率显著超越现有方法。更一般地，只要能提供关于补丁的正确性约束，ExtractFix 就可以尝试进行修复。

关键词：软件维护，软件调试，程序估计，缺陷自动修复，“生成-验证”修复方法

Approaches for Automated Program Repair Based on Program Estimation

Bo Wang (Software Engineering)
Directed by Prof. Yingfei Xiong

ABSTRACT

As software scale and complexity rapidly grow, the cost of maintaining is rising sharply. Existing research has shown that debugging and repairing bugs could take more than half of the cost during development. To alleviate the problem, researchers have proposed "generation-and-validation" pattern approaches, which aim to automatically fix software defects. The inputs of the approaches are a buggy program and a set of specifications that reveal the bug(s), and the outputs are a list of patches that repair the program to fit the specifications. In the existing approaches, two common specification forms are test and constraint. The approaches that evaluate a patch by checking whether it can pass the tests are called *test-based repair*. While the approaches that evaluate a patch by checking whether it can satisfy the constraints are called *constraint-based repair*.

To improve the performance of the repair approaches, researchers focus on patch generation and patch verification. However, the repair approaches still face two major challenges:

1. The patch overfitting problem,
2. The low efficiency problem.

The patch overfitting problem means a repair approach generates a wrong patch that meets the given specifications. In other words, the patch *overfits* the given specification. Existing studies show that test-based repair and constraint-based repair both suffer from a severe overfitting problem. To alleviate the problem, existing studies involve statistical information from the big code to guide patch generation. For example, the high-precise repair approaches, ACS[1] and CapGen[2], sort candidate patches by the frequency of their ingredients taken from open-source repositories. However, existing approaches mainly focus on some specified types of defects, which are not systematic.

The low efficiency problem is caused by the internal complexity of repair approaches. On one hand, the test-based approaches evaluate patches by executing tests, i.e., they need

to compile each patch and then run it against the tests. Practically, the evaluation procedure could introduce non-negligible computation costs. The approaches often need to verify a large number of patches to find one that passes all the tests. On the other hand, the constraint-based approaches usually extract and propagate constraints by symbolic execution or execution trace analysis, which introduces heavy costs when repair large-scale programs.

This thesis studies how to alleviate the two challenges in the two aforementioned types of repair approaches, i.e., *the test-based repair* and *the constraint-based repair*.

To overcome the patch overfitting problem, this thesis defines the problem to synthesize patches guided by statistic methods as *the program estimation problem*, and studies how to reduce test-based repair and constraint-based repair to the program estimation problem. The program estimation problem is to find the program which meets the given specification and has the largest conditional probability under the current context. Formally, given a program space $PROG$, a specification S, and a context C, the program estimation problem is to search a program $prog$ that satisfies $argmax_{prog \in PROG \wedge prog \models S}(P(prog|C))$. Given a fix location, if we treat the surrounding code as the context, and the methods to evaluate patches as the specification, we can generate candidate patches by solving the program estimation problem to replace the buggy code snippet. To solve the problem, we need to conquer the following subproblems: (1) define the search space; (2) assign a probability to a program; (3) find the program with the largest probabilistic. This thesis proposes the Linglong framework to solve these problems. Then this thesis proposes Hanabi and ExtractFix, which respectively reduce test-based repair and constraint-based repair to the program estimation problem. The two approaches can generate patches of high quality and alleviate the patch overfitting problem.

To overcome the low efficiency problem, this thesis proposes several approaches. First, a static analysis approach for Linglong framework to filter invalid partial programs as early as possible, which improves repair efficiency by pruning. Second, this thesis proposes AccMut, an accelerating approach for test-based repair to reduce redundant executions during testing. Third, this thesis proposes a limited symbolic execution for ExtractFix to improve the efficiency of extracting and propagating constraints.

In summary, the contributions of this thesis are presented as follows.

1. This thesis defines the program estimation problem and proposes a data-driven solution: **Linglong Framework**. The framework takes code corpus as a training set, statistically learns the relationship between a code snippet and its context, and uses the models to estimate the probability of a program. Lingling framework has four main components: (1) expansion rules, which generalize the context-free grammar rules, support to expand

a program by different orders and build expansion trees which describe the expansion steps; (2) statistical learning models, which assign probabilities to expansion steps; (3) search methods, which convert the program generation procedure to the path-finding problem, by treat programs as vertexes and expansion rules as edges; (4) statically pruning analysis, filters invalid partial programs by static analysis.

2. This thesis presents **Hanabi**, an approach to reduce test based program repair to the program estimation problem, and then repairs conditional statements for Java programs. Given the surrounding code of the fix location as the context, and the type system and the program size as the specification, it tries to find the patch to meet the specification and has the largest probability. Finally, as it is usually impossible to generate specifications via tests, Hanabi verifies the patches by running tests. The evaluation results show that Hanabi performs better than all existing approaches in terms of fixing conditional statements and generates eight fixes that other approaches unable to successfully fix.
3. This thesis presents **AccMut**, an approach to accelerate the verification of test-based repair. AccMut is a dynamic analysis approach to reduce redundant executions during testing. This thesis presents a relationship called equivalence modulo states. Based on the relationship we can group the patches from the same location and share executions inside a group. The evaluation result shows that AccMut outperforms the state-of-the-art approach with a speedup of 2.56X.
4. This thesis presents **ExtractFix**, an approach to reduce constraint-based repair into the program estimation problem. For several specified types of crash bugs, this thesis presents methods to extract constraints and reduces constraint-based repair to the program estimation problem. ExtractFix first extracts constraints by the template-based extraction of the targeted crash types, and localizes fix localizations by dependency analysis. Then it propagates the constraints by controlled symbolic execution between the fix location and the crash location. If we treat the code on the fix location as the context, the propagated constraints as the specification, we can solve the repair problem by the program estimation problem. The evaluation results show that ExtractFix outperforms existing approaches both in repair effect and efficiency.

KEYWORDS: Software Maintenance, Software Debugging, Program Estimation, Automated Program Repair, Generate-and-Validate Repair

目录

表格

插图

第一章　绪论

1.1　研究背景

1.1.1　软件缺陷及其修复描述

计算机软件已经深入人们生产生活的方方面面，起到了不可替代的作用。随着软件的规模的越来越大、复杂程度越来越高，软件维护的成本也不断增高。研究表明，软件维护的成本占软件开发成本的 90% 以上[3]。其中，软件缺陷的修复是软件维护的核心问题，甚至会消耗 50% 以上的开发时间[4]。软件缺陷 (software defect) ① 是指软件中存在的，导致产品无法满足软件需求和其规格要求，需要进行修复的瑕疵、问题[5]。软件缺陷不但会造成严重的经济损失，还会造成人员伤亡。例如，欧洲航天局的 Ariane 5 火箭因整数溢出错误在发射 40 秒后偏离轨道，从而触发了火箭自毁爆炸，直接损失高达 5 亿美元。波音 737 MAX 型客机因为操控系统软件缺陷导致两次坠机事故，共导致 300 余人死亡。

惨痛的教训说明了软件缺陷会带来灾难性的后果。为了避免软件缺陷带来的危害，开发者需要及时地调试并修复缺陷。然而在一些大型软件中，缺陷的增加的速度经常会超过修复的速度。例如，开源软件 Eclipse 大概每日发现 76 个缺陷，其中大部分得不到及时地修复[6]；而 Mozilla 公司每日发现大约 300 个缺陷，大大超出了该公司的软件维护能力[7]。无法及时地修复软件缺陷，无疑给软件运行带来极大的隐患，威胁着人们的生命财产安全。因此，软件缺陷的调试、定位和修复技术对于节约开发成本和保证程序质量有着十分重要的意义。

1.1.2　程序缺陷自动修复

为了降低软件缺陷修复的时间成本和人力成本，工业界和学术界对软件缺陷调试和修复方法展开研究。其中备受关注的是缺陷自动修复方法 (automated program repair)，即以全自动的方法帮助开发者修复缺陷。

形式地讲，该类方法的输入是一个程序 P 及其需要满足的规约 (specification) S，并且 P 不满足 S(即 $P \not\models S$)；该类方法的输出是一个或多个补丁，通过应用补丁能够使得 P 满足 S(即 $P \models S$)。

在缺陷自动修复技术中，软件规约描述了软件应该符合的行为。规约可以有多种形式，例如开发过程中用于描述软件行为的文档、形式化的逻辑公式以及软件测试集

①在本文语境下名词 defect、bug 和 fault 均翻译为缺陷，视为同义词；动词 fix、patch 和 repair 均翻译为修复，视为同义词。

等等[8]。其中，规约可以是完全规约也可以是不完全规约。完全规约是指所提供的规约能完整地描述程序应满足的行为，而不完全规约只能规定一部分程序应满足的行为。在现有的缺陷自动修复技术中，主要还是使用不完全规约。一种典型的不完全规约是单元测试集，即规定若干组程序应该满足“输入-输出”序对 (input-output pair) 来约束程序的行为。由于完全规约一般情况下提取较为困难、形式过于复杂，目前已有的修复技术主要针对一些具有简单且完整规约的特殊类型的缺陷，例如数组越界、空指针解引用、除零异常、数据竞争等。

现有缺陷自动修复技术基本上被认为符合“生成-验证”模式 (generate-and-validate pattern)[8–12]，其流程概览如图1.1所示。“生成-验证”模式的缺陷修复技术主要包含 3 个阶段：(1) 缺陷定位阶段；(2) 补丁生成阶段；(3) 补丁验证阶段。缺陷定位阶段是依赖于缺陷自动定位技术，根据错误程序、程序输入等信息给出一个按出错可疑程度排序的代码位置序列。补丁生成阶段是修复技术的核心，在给定可疑位置的基础上，通过修复方法特有的补丁生成算法产生补丁序列。已有研究表明，在补丁空间中，正确的补丁十分稀疏，而能通过测试集等验证手段的错误补丁却很多[13]。补丁生成阶段的关键是描述合适的补丁空间并且设计高效的搜索策略，以便尽可能地在补丁空间内找到正确的补丁。补丁验证阶段则使用软件规约过滤所生成的候选补丁，尽可能过滤掉非法补丁、留下正确补丁。

自该方法在 2009 诞生之后就一直是软件工程和程序语言等领域的学术研究热点，并且 Facebook、华为和阿里巴巴等科技公司也将该技术应用到实际工业生产中[14,15]，取得了良好的效果。目前，缺陷自动修复已经能够修复一些对开发者来说也相对困难的缺陷[16]。缺陷修复方法的补丁质量也逐渐接近人工修复，在最近的 Java 修复工具产生的正确补丁中，70% 以上都与开发者提供的补丁等价[17]。缺陷定位结果的质量对修复的质量有直接影响[18]，不过近期研究者发现自动修复方法可以反过来提升缺陷定位的效果[19]。

根据补丁验证阶段所依赖的规约类型，可以将修复技术分为*基于测试的修复*和*基于约束的修复*。基于测试的修复是将应用补丁后的程序运行测试，检测其是否能通过测试。基于约束的修复是根据程序应满足的属性提取出补丁在修复位置应满足的约束，检验补丁的语义是否满足该公式。其中，约束一般以逻辑公式的形式表示。在实际程序中，在修复位置上的规约一般难以取得，尤其是对于程序完整执行的“输入-输出”样例难以映射为修复位置的规约。因此目前主流的验证手段依旧是应用补丁后执行全部测试，观察完整程序的运行结果是否符合预期。测试验证阶段是计算密集型的任务，往往占总的修复时间的较大比重。因此，高效的补丁验证方法能够缩短修复时间，提升正确修复的可能，并且能减少开发者的等待时间，提升自动修复技术的实用性。

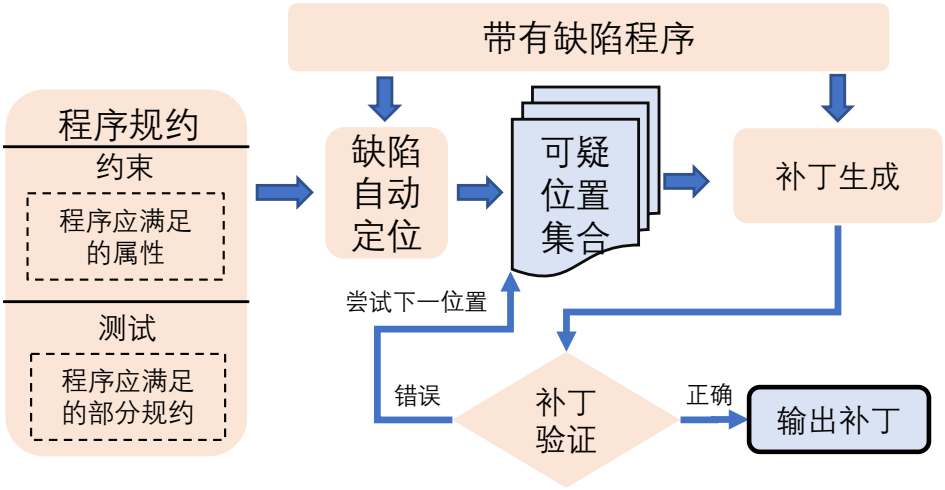

图 1.1　基于“生成-验证”模式的缺陷修复技术流程图

1.2　缺陷自动修复技术中尚待解决的问题

自从 Weimer 等人于 2009 年提出首个缺陷自动修复方法 GenProg[20] 以来，研究者针对该问题展开广泛研究，研究方向包括提出新的修复方法以及提升修复的精确度和效率，详情请见2.1节。

缺陷自动修复领域相关研究的最高目标是使修复技术能够满足工业生产的要求，产生的补丁达到或接近人工修复的质量。然而，当前自动修复方法中，存在**补丁过拟合**和**修复效率过低**两大挑战，阻碍了该方法的实际应用。

1.2.1　补丁过拟合问题

补丁过拟合问题 (overfitting problem) 是缺陷修复系统生成的补丁能够满足所提供的规约，但是该补丁不是正确的。例如，由于测试集的局限性，自动修复工具经常出现能够通过全部测试却与正确的补丁语义不等价的情况，即补丁拟合在测试集上。

在缺陷修复技术发展的早期，研究人员没有重视生成补丁的质量，只关注补丁能否通过全部测试。例如，在 GenProg[20]、AE[21] 和 RSRepair[22] 等方法中，只要补丁通过很弱的测试，就被认为是正确的。Simth 等人在分析 GenProg[20] 和 TrpAutoRepair[23] 的修复过程发现，高质量的测试集有利于提升补丁质量，而且这两个修复方法经常产生删除未被测试的功能的过拟合补丁[24]。Qi 等人进一步人工检视了 GenProg、AE 和 RSRepair 所产生的修复，发现它们的实际修复效果远低于文章报告的结果。在上述文章用于实验验证的 105 个缺陷中，GenProg 和 RSRepair 仅仅产生了 2 个与开发者的修复语义等价的补丁，而 AE 仅仅产生 3 个[25]，远低于原始论文中报告的修复数量。该论文指出补丁即便可以通过测试集规约的检验，仅能说明是“貌似正确的 (plausible) 补丁”，而不一定满足其他形式的规约。相关结论引起自动修复研究领域对过拟合补丁的

重视和讨论。随后，Le 等人发现基于语义的修复中也存在严重的过拟合现象[26]，并试图从所采用的程序综合引擎解释原因[27]。

龙凡等人解释了产生过拟合补丁的原因[13]。在整个补丁空间中，正确的补丁是十分稀疏的，而能通过全部测试的错误补丁则多出好几个数量级，因此补丁验证阶段需要其他辅助信息分析“貌似正确的补丁”的正确性。在实验中，一个缺陷对应的上千个“貌似正确的补丁”中只有一两个是正确的补丁。更进一步，作者发现“更大的”和“更丰富的”补丁空间往往不会带来修复性能的提升，甚至会导致产生更少的正确修复。作者分析造成这个原因是：(1) 对候选补丁的验证次数增多；(2) 通过全部测试的错误补丁阻止了正确的补丁被发现。因此，缺陷自动修复工作需要对补丁空间的大小以及丰富程度做出权衡。该论文引发了针对补丁评价方法的广泛讨论。在此之后的修复研究中，主要依靠作者人工检验的方法来验证“貌似正确的补丁”的正确性。

为了缓解补丁过拟合问题，研究者主要从两方面开展研究：(1) 提出更好的补丁生成方法，从而提升找到正确补丁的可能性；(2) 增强验证手段，尽可能地过滤错误补丁、选择正确补丁。第一类方法尝试引入统计信息，并针对特殊类型缺陷引入启发式规则提高补丁搜索的精确度。例如，ACS[1] 和 CapGen[2] 通过对开源代码的模式出现的频率建立模型，优先返回频繁模式的补丁。第二类方法主要从引入其他形式的规约。例如，测试集补强[28]，引入程序的安全属性作为预言 (oracle)[29]，引入相同功能代码的其他实现作为规约[30] 以及分析补丁在测试上的执行行为[31] 等手段。然而，当前缺陷自动修复方法大部分依然存在较为严重的补丁过拟合问题，已有缓解过拟合问题的方法主要针对特定类型的缺陷，缺乏系统性方法。

1.2.2 效率过低问题

修复效率不高是由修复方法内在复杂性导致的。本节分别介绍基于测试的修复和基于约束的修复的效率问题。

1.2.2.1 基于测试的修复中的效率问题

在基于测试的修复方法中，验证方式是检验补丁是否通过测试集。现有方法往往需要探索较多的补丁才能发现可以通过全部测试的补丁。在实际修复中，绝大多数执行时间都消耗在测试验证上。因此，补丁验证阶段是基于测试的修复的性能瓶颈。下面详细分析测试验证阶段影响修复效率的原因。

缺陷自动修复的搜索空间主要包含三个维度：(1) 可疑位置空间 (fault space, FS)，即缺陷定位方法返回的位置序列；(2) 变异操作空间 (operation space, OS)，即修复预定义的用于产生补丁的操作集合；(3) 补丁原料空间 (ingredient space, IS)，即用于生成补丁的代码组件集合[2]。而最终补丁的搜索空间，是这三个集合的笛卡尔积:

$$P = FS \times OS \times IS \tag{1.1}$$

而在现实的修复中，尤其是在大型软件上，$|P| = |FS| \cdot |OS| \cdot |IS|$ 的值往往十分巨大，而验证一个补丁的开销又不可忽略，这造成补丁验证阶段十分消耗时间和计算资源。在以测试为规约的修复系统中，验证阶段总的时间代价 T_v 为：

$$T_v = \sum_{p \in P} t_{cmpl,p} + \sum_{p \in P} \sum_{tc \in TC} t_{test,p,tc} \tag{1.2}$$

即总时间 (T_v) 包括针对每个补丁 (p) 植入补丁后编译得到可执行软件的时间 ($t_{cmpl,p}$) 以及执行测试集 (TC) 中每个测试的时间 ($t_{test,p,tc}$)。上述朴素的运行过程是十分消耗计算资源的，严重影响修复技术的可延展性 (scalability)。现有研究表明，修复总时间中的大多数时间用于补丁验证阶段的反复执行回归测试上[21]，验证阶段执行测试的时间可以占据全部修复时间的 60% 以上[32]。

在现有研究中，为了保证实验的可行性，往往对修复时间设置上限。然而，当前技术很多采用了较长的时间上限，例如，SimFix[33] 和 GenPat[34] 等方法将超时时间设为 5 小时，AE[21] 、Fix2Fit[35] 和 Prophet[36] 等方法的超时时间甚至设置为 12 小时。过长的修复时间无法及时地为程序员提供反馈和参考，不利于该技术的实际应用。

在缺陷自动修复技术发展的早期，Weimer 等人就已提出基于测试的缺陷修复与变异分析 (mutation analysis) 是本质一样的技术[21]。二者都需要对程序大量的相似变体反复编译并执行相同的测试集。变异分析加速作为软件测试领域的经典问题，已有丰富的相关工作可以参考。目前，已有一些变异分析的加速方法被应用在缺陷修复方法中，以提升补丁测试验证阶段的效率，例如测试选择 (test selection)[37]、等价变异体过滤[21]、变异提要 (mutant schemata)[38]、测试等价性分析[39] 等。然而，当前基于测试的修复流程中依然存在大量冗余计算，严重降低了修复的效率。

1.2.2.2　基于约束的修复中的效率问题

在基于约束的修复方法中，约束是验证补丁的规约。约束是描述正确程序应满足的属性。常见的约束形式是描述程序语义的逻辑表达式。

在实际程序中，约束往往不能直接获得，需要对缺陷程序进行分析提取。在约束提取的过程中，往往需要使用符号执行或者程序执行的动态踪迹等计算代价较高程序分析方法。例如，SemFix[40]、DirectFix[41] 和 Angelix[42] 等方法需要对大型软件实施符号执行以获得补丁应满足的约束；Nopol[43] 和 JAID[38] 等方法需要收集程序在测试集上的执行踪迹以提取程序运行时关于程序状态的约束。

符号执行由于路径爆炸等问题，运行代价较高。在大型程序上使用符号执行收集和传播约束的方法修复效率不高。使用程序踪迹提取状态约束需要记录大量测试执行运行时的状态，运行代价较高，而且这样提取的约束并不能保证可以完全刻画测试集规约。为了保证修复质量，Nopol[43] 和 JAID[38] 等现有方法最终仍然会使用测试集验证，这导致这类方法也面临与基于测试的修复方法补丁验证阶段同样的效率问题。

1.3 本文的研究思路

缺陷自动修复技术研究的关键问题是如何缓解其两大挑战，即如何缓解补丁过拟合问题与提升修复效率。本文大致的研究思路如图1.2所示，其中上部分的实线矩形内是本文针对的目标问题和挑战，下半部分虚线矩形内是本文为了缓解这些问题而展开的研究工作。

本文主要的目标问题是缓解程序修复问题所面临的补丁过拟合问题和修复效率较低问题。在程序修复技术中，如果按照验证手段分类，目前两个主流的方法是基于测试的修复和基于约束的修复。两种修复方法在缺陷定位、补丁生成和补丁验证等修复的关键步骤上均有所不同，而且面临的过拟合问题和效率问题的原因和表现也不同。为此，本文针对这两种修复分别展开研究，探索如何将两种修复方法分别归约 (reduce) ①为程序估计问题以缓解补丁过拟合挑战，并研究如何提升修复效率。本文的主要工作以解决修复问题的两大挑战展开。

针对补丁过拟合问题，首先本文定义了程序估计问题，并提出了玲珑框架用于解决程序估计问题。玲珑框架通过描述合适的程序搜索空间、查找程序生成路径并计算程序的条件概率，从而能够生成高质量的补丁。对于基于测试的修复，本文提出了针对条件语句缺陷的修复方法 Hanabi 。对于基于约束的修复，本文提出针对崩溃缺陷的修复方法 ExtractFix 。通过 Hanabi 和 ExtractFix ，本文分别将两种修复方法归约为程序估计问题，并通过生成高质量的补丁缓解补丁过拟合问题。

针对修复效率问题，本文在玲珑框架中添加了静态分析剪枝方法。该方法可以保证在早期过滤掉不可能满足规约的程序，从而提升程序估计问题的求解效率，继而在修复实践中能够减少需要验证的补丁数量。由于两种修复方法的性能瓶颈不同，导致低效的原因也不同，因此本文针对各自的特点研究加速方法。针对基于测试的修复，本文提出基于状态同余的测试验证加速方法 AccMut ，削减冗余计算；针对基于约束的修复，本文在 ExtractFix 中提出了受限的符号执行方法，限制探索的路径数量。实验验证表明，上述方法缓解了修复系统的效率不足问题。

综上所述，本文针对两种修复方法面临的挑战展开深入研究。实验结果表明，本

①https://en.wikipedia.org/wiki/Reduction_(complexity)

文的方法缓解了两种修复方法的补丁过拟合问题并提升了修复效率。

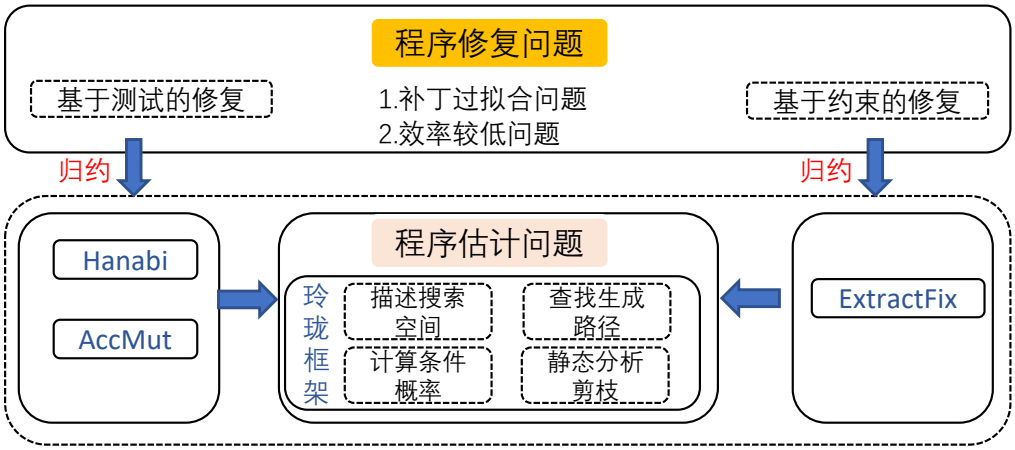

图 1.2　本文研究思路示意图

1.4　本文的主要贡献

如前文研究思路所述，本文拟解决以下科学问题：

1. 如何解决程序估计问题？需要依次解决的子问题主要有：如何定义程序空间，如何计算程序的条件概率以及如何分析程序是否满足规约。
2. 如何将程序自动修复问题归约为程序估计问题？具体来说，我们需要分别实现缺陷自动修复的三个主要阶段，即缺陷定位阶段、补丁生成阶段和补丁验证阶段。其中，基于测试的修复技术和基于约束的修复技术的定位、修复以及验证阶段均有所差异，面对具体问题如何进行归约？
3. 如何提升基于测试的修复方法和基于约束的修复方法的修复效率？

为了解决上述的科学问题，本文主要做出如下创新和贡献。

1.4.1　定义程序估计问题

在很多情况下，我们需要根据已有代码、开发文档等上下文，生成满足归约并且是最有可能出现的程序。本文将其称为**程序估计问题 (program estimation problem)**，即在给定一个上下文 (context) 内搜索程序空间内满足规约且最有可能的程序。很多需要自动或半自动生成代码的问题都可以归约为程序估计问题。例如，当软件维护需要添加新功能时，开发人员提供一组自然语言描述作为程序规约，就可以通过程序估计问题求解满足规约且概率最大的程序。事实上“生成-验证”模式的缺陷自动修复就是程序估计问题的一个实例。例如在基于测试的修复中，可以用程序估计问题针对修复位置生成补丁，即以修复位置为上下文，以类型匹配等程序合法性约束为规约生成概率

较大的程序，并在最后使用测试对补丁进行验证。因此，如果我们能够将程序修复问题归约为程序估计问题，就能够通过求解程序估计问题来解决修复问题。

在此形式地定义程序估计问题，给定上下文 C 和一个规约集合 S，我们需要在程序空间 $PROG$ 中搜索到程序 $prog$ 使之满足：

$$argmax_{prog \in PROG \wedge prog \models S}(P(prog|C)) \tag{1.3}$$

与程序估计问题相关的两个问题是程序综合问题和基于统计的代码生成问题。在所生成程序必须满足规约这一点上，程序综合问题与程序估计问题是一致的。但程序估计问题还需要所得程序在给定上下文的条件下概率最大。形式地讲，程序综合问题是寻找程序 $prog$ 使之满足 $prog \in PROG \wedge prog \models S$。基于统计的代码生成问题和程序估计问题都需要所得程序在上下文下概率最大，但是程序估计问题还需要所得程序满足规约。形式地讲，代码生成问题是在上下文 C 的条件下，寻找程序 $prog$ 使之满足 $argmax_{prog \in PROG}(P(prog|C))$。从各自的定义来看，上述三个问题是不同的。

1.4.2 玲珑框架

本文定义程序估计问题之后，提出了面向该问题的解决方案：**玲珑框架**。

如前文所述，程序估计问题输入为上下文和规约，输出为满足规约的程序以及该程序关于当前上下文的条件概率。由于当前开源软件和在线代码仓库等资源十分丰富，一个可行的方案是使用数据驱动的方式、利用统计学习方法训练模型来预测程序概率。

玲珑框架玲珑框架包含四个组成部分：

1. **扩展规则**，是上下文无关文法的一种一般化表示，能够形式化描述程序片段从不同方向的生成过程。根据扩展规则我们可以得到抽象语法树的泛化版本扩展树。扩展树可以支持程序按不同的方向展开。在每一步扩展中，可以从匹配的扩展规则中选择一个应用。扩展规则描述了全部的程序空间。
2. **程序统计模型**，用于估计出程序的概率。根据扩展树，程序生成过程被转化为若干步针对扩展规则的分类问题。该分类问题可以使用统计学习模型预测选择某个扩展规则的条件概率。程序的最终概率为各个扩展步骤条件概率的乘积。
3. **搜索算法**，本文将程序按照扩展规则展开的过程建模为有向图。图中的顶点为扩展树描述的程序，边为扩展规则。两个相邻的顶点表示出发顶点的程序应用扩展规则后扩展为到达顶点的程序。程序的生成过程就是沿着一条从代表空程序的顶点到一个代表完全展开程序的顶点的完整路径。通过将程序生成问题转化为路径查找问题，就可以使用图上路径查找算法来解决。
4. **静态分析剪枝**，是一组针对部分程序的静态分析过滤方法，能够尽早地将不满

足规约的选择过滤掉，避免生成非法的程序，提升生成效率。

1.4.3　在程序估计问题的框架内解决基于测试的修复

在玲珑框架的基础上，本文将基于测试的程序修复归约为程序估计问题。修复条件语句是开发中常见的任务，面向条件语句的修复有较为重要的实用价值。条件语句缺陷是指发生在分支语句中的缺陷，是常见的一种缺陷类型。据文献[44] 统计，在大型开源数据集 Defects4J[45] 中，有超过 60% 以上的缺陷是条件语句缺陷。条件语句缺陷受到广泛关注，已有 SPR[46]、Nopol[43]、ACS[1] 等工作对此展开研究。因此若能广泛而精确地修复条件语句缺陷，可以极大地降低软件维护的人力物力代价。

本文尝试在程序估计框架下解决基于测试的程序自动修复问题。我们分别实例化了基于测试的程序自动修复技术的三个主要阶段，即缺陷定位阶段，补丁生成阶段和补丁验证阶段，最终整合为面向条件语句缺陷的修复方法 **Hanabi** ①。

在缺陷定位阶段，Hanabi 与主流的基于测试的修复技术一致，采用了基于频谱的缺陷定位技术 (spectrum-based fault localization)[47,48]，即在统计某程序位置在通过测试和未通过测试上的执行次数来计算其出错可疑度。为了将修复问题规约为程序估计问题，本文面向生成 Java 条件表达式的问题对玲珑框架进行实例化，实现了各个子模块并通过实验选出最合适的配置。在补丁生成阶段，Hanabi 根据给定的可疑位置的代码上下文提取特征并以类型约束和程序规模约束为规约，通过实例化的玲珑框架生成目标程序。最终得到候选目标条件表达式 (condition expression) 列表，该列表按照表达式的概率排序。Hanabi 将条件表达式根据模板生成补丁。由于在大型程序中测试无法直接转化为关于补丁的规约，在补丁验证阶段 Hanabi 使用测试集对所生成的补丁进行验证。最终 Hanabi 返回可以通过全部测试的补丁。

本文在两个学术界常用缺陷数据集 Defects4J[45] 和 Bugs.jar[49] 的中的 6 个大型真实开源项目的 269 个缺陷上进行验证。Hanabi 产生了 42 个修复，其中经过人工检验有 32 个正确修复。在修复条件语句缺陷的能力上，Hanabi 在召回率 (recall) 和精确率 (precision) 上都超越了其他现有方法。尤其值得指出的是 Hanabi 能够修复 8 个其余方法无法修复的缺陷。实验结果说明 Hanabi 能够缓解补丁过拟合问题，并且展示了将基于测试的修复问题归约为程序估计问题是有效的。

1.4.4　基于状态同余的测试验证的加速方法

在基于测试的修复的补丁验证阶段需中，修复工具要反复、大量地运行测试集。期间会消耗大量的计算资源，使补丁验证成为修复的效率瓶颈。补丁往往是相对缺陷程

①Hanabi 为日语中的花火 (はなび)，寓意其采用以自底向上的展开方式来生成补丁的过程。

序的微小改动，在自动生成的补丁之间绝大部分的代码都是相同的，并且共享相同的测试输入集合。这意味着测试验证过程中，在编译和执行阶段都存在大量的冗余计算。

本文提出一种动态分析方法，约减验证阶段的冗余计算。在同一代码位置的两个程序变体，尽管语义不等价，在特殊的测试输入下，其状态可能一致。例如，对于某个缺陷程序，在同一位置的两个补丁，`int c=a+b` 和 `int c=a*b` 在输入 `a` 和 `b` 都为 `2` 的时候，会进入相同的状态，并最终得到相同的输出。在本文中，称这两个补丁**状态同余**。对于状态同余的补丁，我们可以将其合并为一个等价类，对于每个等价类仅需要实际执行其中的任意一个的代码变体，因为它们都会有相同的运算结果。我们可以分析找到状态同余的补丁并共享计算，从而加速基于测试的修复的补丁验证速度。

本文提出了基于动态分析的同余状态检测方法 **AccMut** ①来削减冗余的计算。在程序执行的过程中，当遇到修复位置，AccMut 尝试将该位置上的补丁按照同余状态关系，根据补丁执行结果的状态将其分为等价类。对于每个等价类，AccMut 使用 `POSIX fork()` 系统调用分支出新的进程，代表该等价类的全部补丁完成剩余的计算。由于补丁验证和变异分析的等价性[21]，该技术也可以用于加速变异分析的执行速度。

在大规模的开源软件上的实验验证表明，处理单行位置的 AccMut 比变异提要技术加速了 8.95 倍，比当前最新方法分支流执行提升 2.56 倍左右。AccMut 可以大幅提升基于测试的修复的补丁验证效率。

1.4.5 在程序估计问题的框架内解决基于约束的修复

约束是描述程序应满足的属性，可以根据安全性属性或者程序语义提取。与基于测试的修复方法不同，基于约束的修复方法是通过约束作为验证规约来检查补丁。一般基于测试的修复对测试集的质量需求较高，然而在很多情况下高质量测试集并不存在，这限制了基于测试的修复系统的能力。其原因主要有：(1) 基于测试使用的基于频谱的定位方法依赖于测试质量；(2) 补丁正确性验证依赖于测试，在测试不足的情况下，基于测试的修复更容易产生过拟合的补丁。另外，新发现的缺陷往往没有测试集的覆盖信息，仅有用户提供的导致出错的输入样例，无法利用基于频谱或者基于变异的缺陷定位技术。因此，为了实现基于约束的修复，需要新的解决方法。本文提出方法 **ExtractFix** ②，将基于约束的修复归约为程序估计问题。

为了实现归约，ExtractFix 需要依次解决约束提取、错误定位、约束传播和补丁生成这几个问题。给定一个能触发程序崩溃的输入，首先我们需要提取出约束。ExtractFix 在通过 Sanitizer 报告的崩溃位置，将具体崩溃信息与程序代码元素联系起来，再基于模板得到避免崩溃发生的约束。随后，在没有测试集的覆盖统计信息下，我们需要解

①AccMut 为 Accelerating Mutation Analysis 的缩写。
②ExtractFix 寓意提取约束后修复。

决定位问题。ExtractFix 通过静态分析进行缺陷定位与动态执行覆盖情况得到候选的修复位置集合。由于修复位置和崩溃位置经常不在同一位置，我们需要将约束传播至修复位置。ExtractFix 从修复位置到崩溃位置实施符号执行，并且只符号化崩溃位置相关的变量，通过受限的符号执行求出崩溃位置上保持正确状态的最弱前条件，将约束反向传播至修复位置。最后，以修复位置语句为上下文，以反向传播来的约束为规约，将其归约为程序估计问题并生成补丁。其中，基于模板的约束提取和受限的符号执行能够缓解当前基于约束的修复所面临的效率问题。

实验结果表明，与已有基于测试的修复方法相比，ExtractFix 的产生的补丁质量更高，而且平均用时仅 9.46 分钟。因此，ExtractFix 能够缓解补丁过拟合问题并提升修复效率。

1.5　论文组织结构

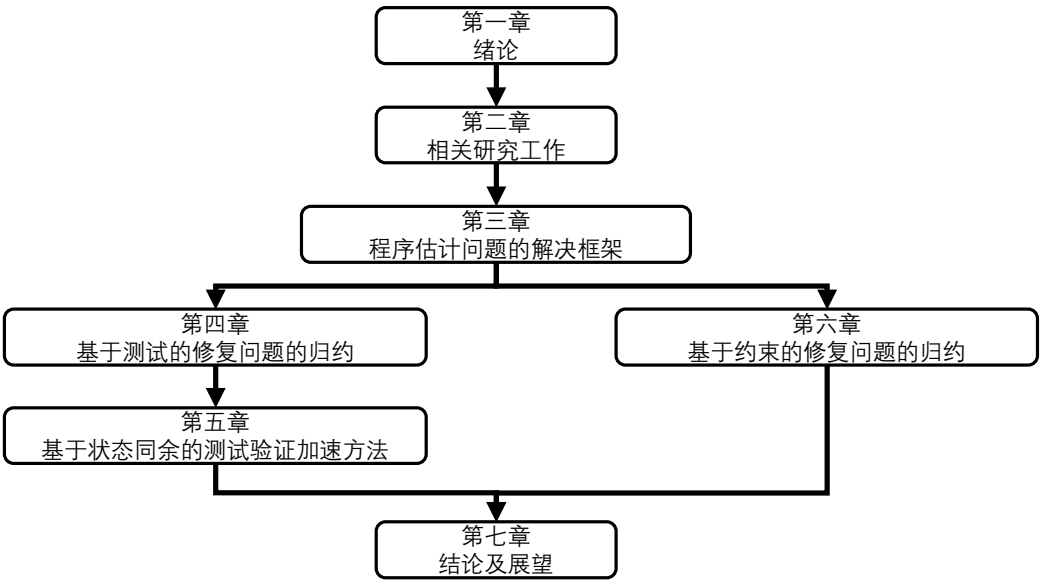

图 1.3　本文组织结构示意图

本文共分七个章节，组织结构如图1.3所示。第一章绪论给出程序估计问题和程序修复问题等关键的定义，介绍缺陷修复面临的两大挑战；第二章介绍相关研究工作；第三章介绍程序估计问题的解决方案；第四章和第五章分别介绍将基于测试的修复归约到程序估计问题上并求解的方法以及验证中的优化；第六章是介绍将基于约束的修复归约到程序估计问题并求解的方法；第七章总结。

对本文各章节的简介如下：

- **第一章 绪论**。主要介绍本文的研究背景，提出本文的研究思路，并介绍本文的主要贡献和创新。

- **第二章 相关研究工作**。主要分析和总结现有相关工作，主要包括缺陷自动修复技术、程序综合技术和基于统计的代码生成技术。
- **第三章 程序估计问题的解决框架**。提出玲珑框架来解决程序估计问题。定义了扩展规则和扩展树等概念并分析其性质。提出了 AST 和扩展树的转化方式。提出了将程序生成问题转化为路径搜索问题，以及估计程序概率的方法。最后提出针对部分程序的静态分析剪枝方法，尽早地过滤非法程序。
- **第四章 基于测试的修复问题的归约**。针对生成 Java 条件表达式的问题实例化玲珑框架并探索实例化的最合适的配置选项。提出基于测试的修复方法 Hanabi ，将基于测试的程序修复问题归约为程序估计问题。
- **第五章 基于状态同余的测试验证加速方法**。提出一种针对基于测试的修复验证阶段的加速方法以消除验证过程中的冗余计算。
- **第六章 基于约束的修复问题的归约**。提出将基于约束的缺陷修复归约为程序估计问题的方法 ExtractFix 。
- **第七章 结论及展望**。总结本文工作并展望未来研究。

第二章　相关研究工作

本章对相关领域的研究现状进行分析总结，主要包括缺陷自动修复技术、程序综合技术和基于统计的代码生成技术。其中，本文核心目标是缓解缺陷自动修复技术中的补丁过拟合问题和效率较低问题，因此程序修复技术是本章介绍与分析的重点。首先，介绍程序修复方法的两个代表性分类，即基于测试的修复技术和基于约束的修复技术。之后，介绍已有缓解针对补丁过拟合挑战和修复效率过低挑战的方法。最后，对已有方法进行分析总结，简要介绍已有方法的不足。

程序修复技术的核心是合成补丁，而补丁是一种程序片段。本文定义了程序估计问题，并将两种代表性的修复方法归约到该问题。根据前文的定义，求解程序估计问题可以获得满足给定归约的程序，并且可以估计所得程序的概率。程序估计问题与其他程序合成相关方法高度相关，尤其是*程序综合技术*和*代码生成技术*。程序综合技术可以生成满足给定归约的程序，代码生成技术可以预测出程序的概率。虽然这两个方法也可以用于生成程序片段，二者的定义与程序估计问题并不相同。程序综合技术仅生成满足规约的程序，不提供所得程序的概率；代码生成技术仅提供所得程序的概率，不保证满足规约。而程序估计问题是二者的结合，所生成的程序即满足约束也提供概率。本章最后简要介绍和分析这两个领域相关的重点工作。

2.1　缺陷自动修复技术

如前文所述，缺陷自动修复技术对于降低软件维护成本和加强软件可靠性有着十分重要的意义。自 2009 年 GenProg[20] 被提出以来，全世界的研究者提出了大量方法。另外，有研究者对程序员人工修复中的模式和方法展开研究[50–53]。这些研究对改进已有修复方法和提出新的修复方法都很有启发。

目前已有的修复方法符合“生成-验证”模式，即*生成补丁之后通过规约进行验证*。缺陷自动修复方法可根据补丁验证阶段所依赖的规约种类进行分类。其中，主要的两种修复方法是*基于测试的修复方法*和*基于约束的修复方法*。

基于测试的修复以补丁通过测试集为验证标准，而基于约束的修复以补丁满足给定约束为验证标准。其中，测试描述了程序应满足的“输出-输出”对应关系，是一种典型的不完全规约。约束描述了补丁的语义应满足的属性，一般以逻辑公式的形式表示，根据缺陷的类型，约束既有可能是完全规约也有可能是不完全规约。由于测试往往比约束更容易获得，基于测试的修复一直是该领域的研究热点。本节系统地介绍缺陷修复相关研究现状，并分析现有工作的不足。更加系统的分类与描述详见已有的通

信文章[54] 和综述文章[8–12]。

2.1.1 基于测试的修复方法

基于测试的修复方法一般由缺陷定位阶段、补丁生成阶段和补丁验证阶段组成。其中，缺陷定位阶段一般使用基于频谱的定位方法，根据通过测试和未通过测试的覆盖信息计算代码位置的出错可疑度；补丁生成阶段是修复方法的核心，根据缺陷程序自身代码、外部代码库和缺陷报告等数据来源合成补丁；补丁验证阶段主要使用程序自身测试集和程序正确性属性等规约来验证补丁。本小节按照补丁生成阶段所采用的方法，将该类方法分为*基于启发式搜索的修复方法*、*基于枚举的修复方法*和*基于机器学习的修复方法*这三个类别进行介绍。

2.1.1.1 基于启发式搜索的修复方法

与程序综合方法类似，在缺陷自动修复中被最早提出和实践的补丁生成方法也是基于启发式搜索的。特别是一些典型的启发式搜索算法在程序自动修复领域的早期研究中起到了重要作用，例如遗传算法和随机搜索算法等。

遗传算法将搜索空间定义为种群 (population)，搜索空间中每个个体 (individual) 持有表示其特征的基因 (gene)。该算法通过用户定义的适应度函数，计算每个个体的适应值，并基于生物界的优胜劣汰思想，将适应值高的个体留下，根据其基因进行交叉 (crossover) 和变异 (mutation)，得到下一代的种群。遗传算法会迭代上述步骤直至找到满足适应值的个体或者触发停止条件 (例如超时等)。

由 Le Goues 和 Weimer 等人提出的 GenProg[20,55] 是缺陷自动修复领域的开创性工作。GenProg 是基于遗传算法的修复方法。该方法在 C 语言的抽象语法树 (AST) 上操作，即将某代码段的抽象语法树视为个体。GenProg 将程序修复的动作抽象为三种操作，即添加、删除和替换。GenProg 假设用于修复缺陷的补丁的原料就在程序内部某个位置。GenProg 根据测试集的执行覆盖情况为每个语句赋以权重。若一条语句被失败的测试用例覆盖的越多，则越有可能被搜索到。其变异过程是：遗传算法随机从程序的抽象语法树上的某个语句开始，然后在带权路径下进行变异。如果当前语句的权重超过变异阈值，则从插入、替换和删除三个操作中随机选择一个应用。由于修复的目标是通过全部测试，因此适应度函数则为补丁通过测试的数量。作者在后续论文中进行了大规模实验验证，并在 8 个大型真实开源的 C 语言程序中选取了 105 个真实缺陷。其中，GenProg 可以成功修复 55 个，平均修复每个缺陷的成本仅为 8 美元[56]。

Le Goues 等人[57] 分析并验证自动修复方法诞生以来研究者提出的一系列改进。作者以 GenProg 为基础平台，添加 2009 年至 2012 年间提出的新修复方法，验证这些方法的实际效果，并分析修复面临的挑战和未来的研究方向。

随后，Weimer 等人提出了基于程序等价性的改进方法 AE[21]。作者发现，在 GenProg 的实验过程中，由于生成的补丁的搜索空间过大，导致验证过程运行全部测试的时间占了整个修复时间的主要部分[32]。作者们首次将变异分析和程序缺陷修复关联在一起，指出这两个方法本质上是一样的。因此，变异分析中的等价变异 (equivalent mutant) 检测技术可以用于分析修复产生的补丁。AE 按照语义等价关系对补丁进行等价类划分，每个等价类只执行其中一个补丁。所谓语义等价，是指两个不同的程序在所有测试上运行的结果相同。语义等价有句法等价 (syntactic equality)、语句重排之后代码相同以及死代码中的补丁等形式。AE 虽然不能保证找出全部语义等价的补丁，但是确保所找到的语义等价是正确的。在与 GenProg 的对比实验中，AE 的搜索空间显著缩小，并且将修复成本降至 1/3。

然而，Qi 等人将 GenProg 中的遗传算法替换为随机搜索算法，提出了修复方法 RSRepair[22]。在实验验证中，RSRepair 在 24 个缺陷中都可以生成与 GenProg 等价的补丁，而且 23 个生成补丁的速度比 GenProg 更快。因此，该文对 GenProg 所采用的遗传算法的有效性产生了质疑。并且指出随机算法应该作为搜索方法的比较基准 (baseline)，相关技术都应该与之做对照实验。该文章引起了对修复方法的广泛讨论。

GenProg 及相关后续研究一直沿用着原论文中的 105 个真实的 C 程序的缺陷数据集，用来评价修复系统的能力。然而，在 2015 年 Qi 等人指出 GenProg 等工作实验设置上的重大错误，导致原作者对实验结果做出过于乐观的估计[25]。作者重新检查了 GenProg 和 AE 在 105 个缺陷上的修复补丁，发现 GenProg 报告的 55 个可修复的缺陷中，有 37 个在修复后无法通过全部测试；而 AE 报告的 54 个可修复的缺陷中，有 27 个在修复后无法通过全部测试。更进一步，在人工检视全部的 105 个补丁后，发现 GenProg 的修复结果只有 2 个是语义正确的，而 AE 的结果中只有 3 个是语义正确的，剩下的修复均无法满足原有程序的正确性规约。作者分析原因指出该数据集的测试判定过弱导致了这一现象。例如，有的测试把补丁能使程序正常退出判定为正确的，如果修复工具在程序最开始处插入 `return 0;` 就可以“修复”该缺陷。为了说明测试过弱的危害，作者实现了 Kali 工具，该工具的唯一功能就是删除已有语句。在修复 GenProg 的数据集的实验中，Kali 的修复效果超过了 GenProg 等一系列工作。Kali 可以完全正确修复 3 个补丁，多于 GenProg 的 2 个，AE 的 3 个。Kali 产生了 27 个“貌似正确 (plausible)”的补丁，多于 GenProg 的 18 个、RSRepair 的 10 个、AE 的 27 个。

随后，Monperrus 团队在 Java 缺陷数据集 Defects4J[45] 中的 4 个项目，总计 224 个缺陷上进行了大规模实证研究[58]。原始版本的 GenProg 和 Kali 修复的是 C 语言程序，作者实现了对应的 Java 版本：JGenProg 和 JKali。作者使用 JGenProg、JKali 和 Nopol[43] 实施修复对比实验。实验结果表明，JGenProg、JKali 和 Nopol 可以生成“貌似正确”的补丁 (即可以通过全部测试集的补丁) 分别为 27 个、22 个和 35 个。而其中真正与开发

者的修复语义等价的补丁，JGenProg、JKali 和 Nopol 分别产生了 5 个、1 个和 5 个。上述三个修复方法的精确性不足 20%。该实证研究说明，在测试不强的客观条件下，缺陷修复存在严重的补丁过拟合问题。修复工具报告的补丁需要人工检查，修复的精确率有待提高。

另外，Monperrus 团队将 JGenProg、JKali 等六个修复方法继承到修复工具框架 ASTOR 中，为研究者提供实验平台[59,60]。

Yuan 等人提出了新的基于遗传搜索的修复方法 ARJA[61]。相比于传统的遗传修复方法 GenProg，ARJA 采用了更细粒度的补丁表示，实现对可疑位置、操作类型和补丁原料等子搜索空间的解耦合。在新的补丁表示的帮助下，ARJA 把缺陷修复问题转化为多目标搜索问题，并使遗传算法可以更有效的探索搜索空间。在数据集 Defects4J 的实验验证上，ARJA 的性能相比于 JGenProg[58] 大幅提升。随后，Yuan 等人提出了其改进方法 ARJA-e[62]。ARJA-e 使用了语句级冗余假设和修复模板这两种修复原料，并且改进了遗传算法的适应性函数。在 Defects4J 的 224 个缺陷上，ARJA-e 成功修复了 39 个，修复效果相比已有方法有了显著提升。

Yu 等人提出了针对智能合约 (smart contract) 的修复方法 SCRepair[63]。SCRepair 使用遗传搜索算法从程序变异中搜索补丁。作者针对智能合约设计了变异算子，并设计了一组适应性函数。

研究者将遗传算法和随机搜索用于缺陷修复之后，针对程序修复和代码的特点，提出了众多的启发式搜索算法。

Le 等人利用历史补丁数据指导补丁排序，提出了修复方法 HDRepair[64]。HDRepair 首先使用变异操作产生补丁集合。在补丁集合中，如果某个补丁符合开源代码中经常出现的修复模式，则会被赋予高优先级。

熊英飞等人提出了精确条件生成技术 ACS[1] 用于精确地修复条件语句错误。ACS 总结了 3 个启发规则，并将其结合起来对补丁进行排序，包括：(1) 对缺陷结构的研究；(2) 对缺陷程序的文档分析；(3) 对已有程序的 `if` 条件表达式分析。ACS 在 Github [①]等开源项目网站上收集 `if` 条件表达式，并按照表达式的谓词的出现频率排序。实验结果表明，在数据集 Defects4J 上，ACS 的修复精度达到了 78.3%，召回率为 8%。相比之前工作不足 40% 的精确率，ACS 大幅提高了修复的精确度。

Hua 等人将程序修复问题归约为基于 Sketch 的程序综合问题，提出了修复方法 SketchFix[65,66]。如果把缺陷修复视为先将缺陷位置“挖空”，并保留其余上下文代码，再生成补丁填入空处。此行为与基于 Sketch 的程序综合一致，因此可以使用 Sketch 综合技术求解补丁。

Koyuncu 等提出基于缺陷报告的修复工作 iFixR[67]。在很多场景下，新提出的缺陷报

[①] `https://github.com/`

告问题 (bug report issue) 没有对应的测试，需要在此场景下根据缺陷报告的描述作为规约验证。作者实现了基于信息检索的定位 (information-retrieval-based fault localization)，可以避免像基于频谱的方法一样运行测试集。该方法通过模板生成补丁，并使用三个启发式规则进行排序。

姜佳君等人提出基于小样本学习的修复技术 GenPat[34]。GenPat 通过将代码表示为超图，从而将补丁模板的抽象过程转化为图中路径的选择过程，可以利用较少的样本得到修复的模板。

基于代码相似度被证明是一种有效的启发式规则，一些工作利用相似代码指导修复过程。

Qi 等人提出基于代码本文相似度的修复工作 ssFix[68]。该工作将代码视为纯文本，定位到可疑的代码块之后，通过搜索引擎技术在代码库中搜索相似的代码。

文明等人提出上下文敏感的修复技术 CapGen[2]。CapGen 使用自定义的启发式方法对补丁进行排序。作者人工指定描述代码上下文的线性函数，通过该函数对补丁进行打分排序。该方法在 Defects4J 的 224 个缺陷中成功修复了其中的 21 个，正确修复率达到了 84%。值得说明的是该方法的精确率是在已有方法中最高的。

姜佳君等人同时考虑了缺陷软件代码内部的上下文和其他开源软件的数据，提出了修复方法 SimFix[33]。SimFix 利用补丁和相似代码之间抽象语法树的差异程度进行排序，并参考开源数据中修复模式的出现频率。该方法在 Defects4J 的 5 个项目 357 个缺陷中，正确修复了 34 个缺陷。在 2018 年及之前的已有方法中，SimFix 达到了在 Defects4J 数据集上最高的召回率。

Asad 等人同时结合了代码的文法相似性和语义相似性用于补丁的排序[69]。作者使用 AST 祖先节点以及变量名来描述语义相似性，使用正则化的最长公共子序列描述文法相似性。

Saha 等人利用代码演化实现修复多个缺陷位置的修复方法 HERCULE[70]。在已有研究中，仅有 Angelix 等少数工作能实现对多个缺陷位置同时进行修复。作者通过上下文挖掘修复位置的演化兄弟位置，对这些多个位置实现同时修复。所谓演化兄弟，是指代码演化过程中的相似代码，而非简单的代码克隆 (code clone)。作者克服了挖掘演化兄弟过程的中的噪声等困难，实现精度较高的搜索方法。

Kim 等人提出了基于上下文修改应用的修复方法 CCA[71]。该方法收集抽象子树的修改及其对应的 AST 上下文，并将修改应用在待修复的位置。所谓抽象子树是保留修复原料的结构，但是将项目特定的标识符抽象。通过 AST 上下文匹配，可以有效缩减搜索空间。

表 2.1 PAR 总结的 10 个修复模板

修复模板名称	功能描述
参数替换	选择一个合法变量或表达式替换参数。
函数替换	选择合法函数替换当前函数。
参数添加或删除	通过添加或删除函数的参数，更换重载函数。
表达式替换	使用其他合法表达式替换当前条件表达式。
表达式的添加和删除	添加或删除一个合法的表达式或删除一个判断条件。
空指针检查	为对象类型的引用添加空指针检查。
对象初始化	函数的参数进行初始化。
数组范围检查	添加数组访问是否在边界以内的检查。
集合大小检查	对集合类型的变量，判断其下标访问是否合法。
类型强制转换检查	在类型强制转换之前使用 `instanceof` 判断是否合法。

2.1.1.2 基于枚举的修复方法

基于枚举的修复方法与基于枚举的程序综合方法高度相关，即在划定搜索空间后，使用一种启发式的顺序对空间进行遍历式的搜索。

在缺陷修复方法的发展早期，Debroy 等提出基于变异分析的缺陷修复方法[72]。该方法对于可疑的代码位置应用变异，并将通过所有测试集的变异，即“存活 (live)”下来的变异，视为潜在的补丁。

Kim 等人提出基于模板的修复方法 PAR[73]。PAR 通过人工总结出 10 个典型的修复模式，并根据这些模式划定搜索空间枚举。其总结的模式如表 A.1所示。在实验验证中，PAR 在选取的 119 个缺陷的数据集上修复了 27 个，而 GenProg 仅修复了 16 个。同时，作者还邀请了 117 个学生和 68 个开发者对 PAR 的补丁进行评价。作者发现用户对一些 PAR 的补丁的评价比程序员的原始补丁更高。该论文获得了 2013 年的 ICSE 杰出论文奖。然而，PAR 的实验结果和研究方法遭到置疑。Monperrus 团队在 ICSE 2014 上指出 PAR 的算法和实验的不足[74]，继而引出了自动修复技术的评价体系问题。

在 PAR 的启发下，Tan 等人提出了反模式 (anti-pattern)[75]，即通过人工总结的启发式规则过滤补丁空间内的错误补丁。作者提出了 7 个反模式，如表 2.2所示。作者将反模式并应用在 GenProg 和 SPR 上，显著提升了二者的产生补丁质量和速度。尤其是 GenProg 中经常出现删除功能的错误补丁，通过将删除动作设置为反模式，该类别的非法补丁显著减少。反模式有效地约减了搜索空间，后续很多技术都采用了反模式。

龙凡等人提出了分阶段的条件表达式生成技术 SPR[46]。SPR 可以修复条件缺陷或者 `if` 语句丢失的缺陷。SPR 预先定义了一些代码转化模式 (transformation schema)。例如，对于分支条件表达式错误，SPR 定义了将条件收紧 (通过添加逻辑与运算符 &&) 或将条件放松 (通过添加逻辑或运算符 ||) 的转化模式。SPR 对模式迭代尝试，直至发现

表 2.2　7 个反模式

反模式名称	功能描述
反删除 CFG 退出语句	不可删除 `return` 等语句。
反删除控制语句	不能删除 `if`、`switch` 和循环。
反删除单语句 CFG	不能删除只有一个语句的 CFG 节点。
反删除 `if` 周围语句	不能在删除 `if` 之前的，`if` 语句中使用的变量。
反删除循环迭代更新	不能删除控制循环次数的变量。
反早退出	不能在 CFG 的最后一个语句之前插入 `return`、`exit` 等退出语句。
反添加朴素条件	`if` 的条件的扩展必须有意义，不能改变条件之后所有测试真值不变。

可以通过全部测试的补丁。与 PAR 等简单的模板技术相比，SPR 支持多种模板的组合，能够产生较为复杂的补丁。

Rolim 等人提出了 Refazer 方法[76]，从基于样例的程序综合角度出发，以当前已有的正确修复为样例学习得到修复模板。通过用户自定义的 DSL 划定搜索空间，并归纳出细粒度的修复模板，该方法在学生作业修复任务上可以修复 87% 的同类缺陷。

Le 等人提出了语义制导和语法制导相结合的修复方法 S3[77]。S3 通过对程序实施符号执行获得语义信息并得到程序规约。为了生成满足规约的部分程序，可将生成补丁的问题归约为程序综合问题。S3 通过 DSL 定义了一个相对简单的补丁空间，从而限制了空间的复杂程度。在搜索空间内的程序按照与原程序的相似程度排序进行枚举验证。在实验验证中，S3 能修复 52 个大型 Java 程序缺陷中的 22 个，100 个小型程序中的 20 个，并且没有产生错误的修复。

高祥等人提出避免崩溃的修复方法 Fix2Fit[35] 。Fix2Fit 通过枚举预定义的生成模板产生补丁，并通过受引导的模糊测试 (fuzzing testing) 生成覆盖缺陷位置的测试输入，之后通过 Sanitizer 为测试预言过滤导致缺陷的补丁。

Facebook 公司的 Marginean 等人提出基于变异的修复方法 Sapfix[15]，并成功部署在公司内部的大型软件项目上。该系统是第一个被部署到实际应用场景的“端到端”修复系统。

Ghanbari 等人提出基于字节码修改的缺陷自动修复方法 PraPR[78]。源码中的形式往往很复杂，但是转化为字节码等中间表示之后代码模式的种类大大减少。而且，在中间代码级别可以实现跨语言的修复。例如，Java、Scala 和 Closure 等基于 Java 虚拟机的程序都可以在字节码级别共享相同的修复技术。作者通过设计字节码上的变异操作来描述补丁空间，在通过变异得到候选补丁后，进行枚举验证。修改字节码避免了源码级别的编译开销。在实验验证中，PraPR 实现了修复速度加速 10 倍的效果。

刘逵等人提出了修复方法 LSRepair[79]，使用在线搜索方法在代码仓库中搜索补丁原料。作者指出代码开发过程中，开发者经常会写出重复性的和功能类似的代码，甚

至直接克隆已有代码。若类似代码被修复，则可以用于指导与之类似的缺陷。

刘逵等人总结并分析了当前已有的基于模板的修复方法，并提出综合的基于模板的修复方法 TBar[80]。在实验验证中，TBar 在 Defects4J 中成功修复了 43 个缺陷，证明了基于模板的修复方法的有效性。

刘逵等人随后进行大规模实证研究，实验结果表明已有的不同种类的修复方法互补性很强，其中基于模板的修复方法更为有效[81]。

由于测试在一些情况下并不可访问，刘魁等人提出使用基于静态分析的修复方法 AVATAR[82]。AVATAR 针对静态分析报告的缺陷提取补丁模板并修复。

Koyuncu 等人提出了修复模式的挖掘方法 FixMiner[83]。作者利用带有上下文感知的可描述 AST 修改的富编辑脚本 (rich edit script) 来对相似修改进行聚类。在实验验证中，FixMiner 可以从数千个补丁中提取较为准确的修复模板。

2.1.1.3 基于机器学习的修复方法

基于机器学习的修复方法从开源代码仓库或者是项目本身中统计学习得到模型，并用模型指导补丁生成的过程。

龙凡等人提出 Prophet[36] 是首个使用到基于统计机器学习的修复方法。作者收集一个由正确的补丁集合作为训练集，目标是训练所得的模型能够赋以正确的补丁较高的概率。Prophet 学习补丁周围的代码“环境”以及抽象补丁的特点，以完成特征提取和模型训练。对于补丁，Prophet 提取的特征包括用于捕捉变化和常量被使用的信息的程序值的特征，以及用于捕捉补丁中的操作和补丁应用关系的修改特征。其中，Prophet 使用对数线性模型 (log-linear probability model)，并根据该模型对 SPR[46] 生成的补丁进行排序。

Prophet 对候选补丁集合利用统计学习进行排序，但是补丁的生成方法依旧是 SPR。之后，龙凡继续提出基于机器学习的方法 Genesis[84]，用于推断补丁的生成过程。作者提到 Genesis 是首个自动推导补丁转化过程的工作。Genesis 通过学习有缺陷的代码与修复后的代码的抽象语法树的对应关系建立统计模型，从而避免了人工定义代码转化模板。Genesis 没有使用传统的统计学习模型，而是使用了整数线性规划 (integer linear program，ILP)，即推理出的抽象补丁应尽可能多地覆盖训练集中的数据。Genesis 在空指针 (null pointer)、数组越界 (out of bounds) 和类型强转 (class cast) 这 3 种类型的缺陷上进行了实验。实验结果表明，Genesis 的修复效果显著超过 PAR。

Soto 等人提出了预测缺陷修复概率模型[85]。作者通过开源软件中的补丁的修复模式频率建立统计模型。作者针对修复采用的变异操作符设计了两层的概率模型，第一层为抽象的变异操作算子，第二层为更细粒度的变异操作。

ELIXIR[86] 是 Saha 等人提出的针对面向对象程序的修复工作。ELIXIR 首先针对面

向对象程序的缺陷进行实证研究，分析其特点。作者发现以下 4 方面的信息对于生成补丁十分重要：(1) 标识符在当前上下文出现的频繁程度；(2) 被修复的位置的距离上次使用的距离；(3) 相似的名字是否出现在修复上下文中；(4) 标识符或者部分标识符是否出现在缺陷报告 (bug report) 中。因此，ELIXIR 从上下文中人工提取 4 个重要特征用于训练模型和预测。ELIXIR 使用较为简单的文法描述补丁空间，并使用逻辑回归 (logistic regression) 学习算法。在数据集 Defects4J 上，ELIXIR 成功修复了 26 个缺陷。

随后，Noda 等人将 ELIXIR 部署到实际工业生产场景上以检验其修复效果[87]。作者使用了超过 150 个实际 Java 程序以及 13 年的开发历史数据。基于实验结果，作者指出当前基于测试的修复方法在实际场景中面临着补丁召回率低 (仅为 7.7%)、缺乏揭示缺陷的测试 (90% 的缺陷没有对应测试) 和修复成功率低 (仅为 10%) 等缺点。另外，作者指出只需要对 ELIXIR 做出微小改动，就可将修复成功率提升到 40%。

Gupta 等人提出了 DeepFix[88]，首次将深度学习技术应用于缺陷修复。DeepFix 针对的是程序语法错误而非功能错误。DeepFix 通过多层的序列对序列的递归神经网络 (recursive neural network, RNN) 处理输入代码，最终可以预测出错位置和正确的语句。在对学生编程作业的修复上，DeepFix 可以完整修复其中 27% 的缺陷。

周风顺等人通过将基于深度学习的程序综合与基于模板的修复技术相结合，提出了 AutoGrader 修复方法[89]。AutoGrader 使用 RNN 中的长短期记忆网络 (LSTM) 进行填补 Sketch 的程序综合任务，之后使用人工总结的修复模式来修复 C/C++ 中的缺陷。AutoGrader 在修改学生作业的任务上实现了较高的修复率。

White 等人提出了基于深度神经网络的修复方法 DeepRepair[90]。DeepRepair 试图推理如何为修复原料排序以及如何从代码库中学习转化生成补丁。DeepRepair 使用了递归神经网络对变量名和操作符等补丁原料进行排序，并使用模板生成补丁。

Facebook 公司提出的 Getafix[14] 试图从某一类缺陷的重复修改中学习得到修改。Getafix 是基于层次化的聚类算法 (hierarchical clustering algorithm) 把历史补丁根据其抽象程度归类到不同的层次。Getafix 没有直接地探索巨大的补丁空间，而是根据上下文对补丁进行简单有效地排序。Getafix 是首个工业界开发的缺陷修复工具。

Tufano 等人针对神经机器翻译修复方法的效果在开源软件补丁上进行了大规模的实验验证[91]。作者使用基于循环神经网络的神经机器翻译模型，从开发历史的海量补丁中学习修复模式。

Li 等人提出基于上下文信息的代码转换修复方法 DLFix[92]。当前已有基于深度学习的方法在表示补丁的上下文时仍存在局限。为了解决这一问题，DLFix 提出了两层的深度学习模型。一层基于树的 RNN 模型学习补丁周围代码上下文信息，另一层是卷积神经网络 (convolutional neural network，CNN) 模型用于代码转化以生成补丁。通过分离上下文模型和代码转化模型能够使得 DLFix 更好地分析缺陷位置的周围代码。在

基准数据集 Defects4J 和 Bugs.jar 的实验验证上，DLFix 首次实现了不弱于传统修复方法的效果。

Lutellier 等人利用集成学习 (ensemble learning) 将从不同角度描述缺陷代码和安全代码关系的模型结合起来，提出了修复方法 CoCoNut[93]。CoCoNut 使用了上下文感知神经机器翻译来表示缺陷代码的上下文信息，并且使用卷积神经网络模型进行预测。作者认为卷积神经网络相比于循环神经网络更适合用描述代码中的不同层次的粒度信息。在大规模的实验验证中，CoCoNut 成功修复了 509 个缺陷，其中 309 个是已有方法无法修复的。

基于机器学习的修复方法最初普遍使用逻辑回归等较为简单的模型，后来逐渐开始使用深度神经网络等较为复杂的模型。模型数量也从一个模型进行全部预测，演变为多个模型根据各自的优势处理不同任务。表 2.3 列举了上述相关工作的名称、文献以及机器所采用的机器学习模型。

表 2.3　基于机器学习的修复方法所用的学习模型

工具名称	文献	机器学习模型
Prophet	[36]	对数线性模型
Genesis	[84]	整数线性规划
ELIXIR	[86]	逻辑回归
DeepFix	[88]	递归神经网络
AutoGrader	[89]	长短期记忆网络
修复概率模型	[85]	频率统计
DeepRepair	[90]	递归神经网络
Getafix	[14]	聚类算法
NeuralCodeTranslator	[91]	递归神经网络
DLFix	[92]	递归神经网络 + 卷积神经网络
CoCoNut	[93]	卷积神经网络

2.1.2　基于约束的修复方法

基于约束的修复通过分析得到正确软件应满足的属性，继而指导补丁的生成。约束一般以逻辑表达式的形式描述，用于刻画程序语义或者程序运行状态应满足的条件和属性。修复方法产生的补丁或程序的语义应满足该逻辑表达式。约束可以使用符号执行、软件执行踪迹分析和静态分析等手段来获得。使用符号执行和执行踪迹的方法计算代价较高，难以扩展到大规模软件上。已有实证研究表明基于约束的修复方法也存在较为严重的补丁过拟合问题。目前，使用符号执行和执行踪迹归纳的修复方法是通过搜集使程序满足测试集的语义来获取约束，当测试集质量不高时也不易获得高质量的约束；而基于静态分析获得的约束往往过松。因此，已有的基于约束的修复方法

容易产生过拟合的补丁。

2.1.2.1　使用符号执行提取约束的修复

该类方法通过符号执行获得补丁应满足的约束，并将约束传播至修复位置，再根据约束综合得到补丁。这类方法也被称为基于语义的修复 (semantics-based program repair)。

SemFix 是该类方法的开创性工作[40]。该方法通过符号执行收集通过全部测试的程序应该满足的约束，之后使用基于组件的程序综合[94] 将约束转化为补丁。生成方法是通过预定义一些“组件”，例如变量和操作符，通过求解约束，找到一种将组件组和起来的方式，从而得到补丁。由于需要对全程序实施符号执行，该方法局限在结构简单、规模较小的程序上。

随后，Mechtaev 等人针对 SemFix 提出了改进方法 DirectFix[41]。DirectFix 将约束分为必须被满足的“硬约束”和可以不被满足的“软约束”，并且在生成补丁的过程中，满足所有硬约束，并且最大限度地满足软约束。随后 DirectFix 通过应用部分 MaxSAT 约束求解和基于组件的程序综合来得到补丁。相比于 SemFix，DirectFix 生成的补丁形式更简单，通过回归测试的补丁更多。

Ke 等人提出了基于语义搜索的缺陷修复方法 SearchRepair[95]。SearchRepair 通过对代码实施符号执行获得约束。SearchRepair 首先建立一个人工编写的代码块的数据库，并对代码块提取出“输入-输出”的 SMT 约束。对于给定的缺陷位置，SearchRepair 推导出该位置的“输入-输出”约束关系。根据可疑位置的约束，在代码库中搜索语义类似的代码，并生成补丁替换可疑出错的代码。最后使用测试集对补丁进行验证。在实验验证中，SearchRepair 比 GenProg 、AE 以及 RSRepair 多修复了 20% 新缺陷。

Mechtaev 等人提出了二阶约束符号执行方法 SE-ESOC 并应用在修复上[96]。该方法使用二阶符号变量，避免了一些传统一阶符号变量所面临的路径爆炸问题。

上述基于约束求解的修复方法主要依托 C/C++ 平台上的符号执行引擎 Klee，无法处理 Java 等面向对象语言。Le 等人提出 Java 上基于符号执行引擎 PathFinder 的修复工作 JFix[97]。

由于符号执行的内在缺点，例如路径爆炸、不能支持未建模的库函数等，SemFix 和 DirectFix 仅能处理规模较小的程序。为了克服这一缺点，Mechtaev 等人进一步提出了 Angelix[42]。Angelix 提出了天使森林 (angelic forest) 概念，用于实施部分符号执行，从而避免了从程序最开始进行符号执行。部分符号执行使 Angelix 的可延展性大大增强。相比之前基于语义的修复方法，Angelix 可以成功修复大规模程序的缺陷，并且能够产生多行修复。

Le 等人实施了大规模实证研究以探索基于语义的修复方法的补丁过拟合问题[27]。作者将 Angelix 的程序综合引擎换为基于枚举的程序综合[98] 和 CVC4[99]，得到两种新

的基于约束的修复方法，并与 SemFix[40]、Angelix[42] 一同进行对比。该实证研究表明基于语义的修复方法存在严重的过拟合问题，并且测试集的质量和程序综合引擎对过拟合问题有显著影响。

2.1.2.2 使用程序动态踪迹提取约束的修复方法

符号执行有诸多局限，例如路径爆炸问题、不支持浮点数、不支持第三方库以及不支持内联汇编等等，因此有研究者使用程序动态踪迹来提取约束。但是该类方法不保证约束能够正确地刻画测试集等规约。因此该类方法仍然需要运行全部测试来验证补丁。从这个角度来看，该类方法也可以归类为基于测试的修复方法。

玄跻峰等人提出的 Nopol[43,58] 是针对条件语句修复的方法。为了避免符号执行带来的可延展性差的问题，Nopol 使用根据测试执行的踪迹收集“天使值 (angelic value)”[100]，并根据“天使值”生成约束。所得到的约束经过基于组件的程序综合得到相应的补丁。同时，由于针对的是条件语句缺陷，尤其是 `if` 条件的缺陷，使用天使调试也利于快速、准确地确定缺陷位置。相比于 SemFix、DirectFix 等基于符号执行的修复方法，Nopol 可以处理大型的程序，而且也可以生成面向对象相关的补丁。

Chen 等人提出基于契约 (contract) 的修复方法 JAID[38]。JAID 通过代码运行测试时的具体状态分析得到对于状态的谓词描述，即“契约”。根据补丁应满足的契约指导补丁生成的过程。相比于其他启发式规则，JAID 采用了动态分析技术，从而能获得目标代码的状态信息。

2.1.2.3 其他基于约束的修复方法

某些特殊类别的缺陷的约束更适合用静态分析的方法提取，例如内存泄漏、资源泄漏、配置错误和并发错误等。

高庆等人提出修复 C 语言程序的内存泄露的方法 LeakFix[101]。LeakFix 通过静态分析得到内存泄漏的位置以及需要插入 `free` 语句的位置，可以避免产生双重释放 (double free) 和释放后使用 (use-after-free) 等错误，保证产生的修复是正确的。

Lee 等人提出 MemFix[102] 可以修复内存泄漏、双重释放、释放后使用等内存释放相关缺陷。MemFix 通过一种类型状态静态分析 (typestate static analysis) 的变体，寻找一个合法的 `free` 语句集合。由于其依赖的静态分析是可靠的 (sound)，MemFix 可以保证产生修复的正确性。

Rijnard 等人提出了基于静态分析的修复技术 FootPatch[103] 以修复资源泄漏、指针泄漏和空指针解引用等安全相关的缺陷。该方法利用分离逻辑 (separation logic)，推理程序中已有代码块的语义并构造补丁。

Xu 等人提出针对 Java 程序空指针异常的修复方法 VFix[104]。VFix 使用静态的值流 (value-flow) 分析获得关于补丁的约束，从而生成修复空指针异常的补丁。

Huang 等人使用循环展开等静态分析方法获得关于内存访问越界的约束，从而修复相关类型的安全漏洞[105]。

高庆等人提出修复方法 QACrashFix 修复安卓应用的崩溃缺陷[106]。QACrashFix 通过分析缺陷报告的调用栈信息获得修复位置，并使用编程社区的在线问答网站 (如 StackOverflow ①) 中提问者和回答者提供的代码片段提取修复，最后使用 AST 差异性比较 (diff) 和编译器过滤补丁。

Liu 等使用轻量级的静态分析与动态分析相结合的方法修复安卓应用 (Android app) 的资源泄漏[107]。该方法保证产生的补丁是安全的，不会影响程序的正常运行。

配置错误经常出现在软件产品线中，为了解决配置冲突和错误，研究者提出了交互式的配置修复方法 RangeFix[108] 和 Tortoise[109]。

并发错误是在多线程程序中，指令之间的不同执行顺序造成了不符合预期的相互影响，主要包括数据竞争、死锁、原子性违反、序列化等[110]。研究者提出了一系列修复多线程缺陷的工作，包括修复原子性违背的 AFix[111]、Axis[112]、Grail[113]、HFix[114]，修复死锁的 DFixer[115]，以及综合修复多种并发错误的 CFix[116]。

2.1.3　补强规约来缓解补丁过拟合的方法

在现实软件开发中，软件单元测试集的质量并不能保证，较弱的测试集更容易产生“疑似正确的错误”补丁。缓解补丁过拟合问题的另一种思路是补强规约，增强单元测试或者补充其他形式的规约。

Qi 等人提出了 DiffTGen[28]。DiffTGen 试图发现错误程序和应用补丁之后的程序的语法上的结构差异。之后，通过测试生成工具 EvoSuit 构造测试输入来揭示差异。在得到该测试输入后，通过已有的开发者补丁，比较自动生成的补丁和人工补丁的输出差异。如果二者输出相同，则认为是正确补丁，否则认为是过拟合的补丁。

Yang 等人提出 Opad[29]。该方法首先使用模糊测试 (fuzzing testing) 随机产生满足覆盖要求的测试集，之后根据崩溃和内存安全作为补强的规约，即补丁不得触发这两个类型的错误，否则就被归类为过拟合的补丁。Fix2Fit[35] 也采用了类似的方法，但是选取了数组越界等更多、更严格的安全属性作为规约。

Yu 等人提出缓解过拟合方法 UnsatGuided[117]。UnsatGuided 使用测试生成方法来补强已有的开发者测试，并利用已有测试的预言过滤生成的错误测试。在 Nopol 的实验验证上，UnsatGuided 可以缓解过拟合问题。

①https://stackoverflow.com/

熊英飞等人提出了 Patch-Sim 和 Test-Sim[31] 两个指标来衡量补丁的正确性。为了解决使用随机测试生成的时候测试真言未知的问题，作者提出了补丁相似性（PATCH-SIM）和测试相似性（TEST-SIM）两个启发式规则。所基于的猜想是执行路径越相似的程序和测试输入，则其更可能有同样的正确性和通过性。

为了缓解基于语义的修复系统的过拟合 (overfitting) 问题，Mechtaev 等提出了使用功能相同的参考程序作为规约输入的修复工作 SemGraft[30]。一些库函数和经典算法有多个语义等价实现的版本，将已有版本的语义作为参考，可以用于判断补丁的正确性。

Hu 等人提出针对学生作业的修复方法 Refactory[118]。Refactory 利用代码的控制流图的相似性找到结构最相似的一组正确代码，并以正确代码的基本块的“输入-输出”对作为补强的规约指导修复。

在已有方法中，有的引入随机测试或者模糊测试，结果有一定随机性。而且这些测试生成方法往往会产生大量测试，继而加重修复效率问题。例如，Fix2Fit[35] 的修复中经常出现用满限定的 12 小时的情况。有的方法需要参考已有的正确软件，然而很多情况下难以取得类似的参考。

2.1.4 提升修复效率的相关研究

目前已有的缺陷修复工作普遍存在修复效率不高的问题。在基于测试的修复中效率瓶颈主要在测试验证阶段，而基于约束的修复效率瓶颈主要在约束提取阶段。本小节介绍已有工作中提升效率的相关方法。

已有的基于测试的修复方法主要引入测试相关优化技术来提升效率。Weimer 等人提出基于测试的缺陷自动修复方法本质上就是变异分析方法[21]，即二者都面临对大量相似程序变体执行同一组测试集的场景。在该想法的启发下，作者将变异分析中的等价变异体实别方法引入修复方法中，来过滤多余的补丁，并提升修复方法 AE 的效率。Fast 等人提出在修复中引入测试选择技术，用高质量的测试高效地验证补丁[37]。F1X[39] 利用测试等价技术[119] 将补丁按照能通过的测试划分等价类，每个等价类只需运行一个补丁，从而共享组内的运算。Mehne 等人提出针对基于搜索的修复方法的加速技术，通过筛选修复位置和测试降低修复的时间成本[120]。为了避免编译时的开销，PraPR[78] 通过在字节码上实施修复从而避免源码级别的编译，JAID[38] 引入了变异提要[121] 方法将所有补丁集成编译进同一个可执行文件。

已有的基于约束的修复方法主要考虑降低约束提取阶段中实施符号执行的开销。Angelix[42] 提出了符号森林，以部分符号执行避免对完整程序实施符号执行。SE-ESOC[96] 使用二阶符号变量大幅降低需要探索的路径数量。

2.1.5　现有缺陷修复方法研究总结

现有技术中，为了保证修复的成功率，现有方法主要是限定在特殊类型的缺陷上，如小节 2.1.1.1 中描述的 ACS 和 CapGen 等方法。但是，当缺陷类型增多、缺陷复杂度升高之后，目前的技术依然存在较为严重的补丁过拟合问题。

现有修复技术中，基于测试的修复和基于约束的修复都普遍存在修复效率不足的问题。例如，现有技术修复时间普遍过长，平均修复时间基本都在 1 小时以上。对于基于测试的修复，效率瓶颈主要在补丁验证阶段，验证过程中反复执行测试集引入了巨大的计算消耗。虽然目前已有一些修复方法引入软件测试的相关加速技术，目前基于测试的补丁验证手段仍然较为简单。在验证阶段中，存在大量的冗余计算。对于基于约束的修复，效率瓶颈在约束提取阶段所实施的符号执行或动态踪迹分析等计算代价高的方法。已有一些限制符号执行代价的方法，但是仍有很大的提升空间。

2.2　程序综合技术

程序综合技术 (program synthesis，亦可被称为程序合成)，目标是构建程序使之可被证明满足给定的高级形式化规约 ①。如何自动地生成程序，一直被认为是计算机软件理论研究中最为核心的问题之一[122]。与软件验证技术相反[123]，一般情况下程序综合技术的输入为规约，输出满足规约的程序集合。自 1930 年代，处在萌芽状态的计算机科学就提出并开始了程序综合的相关研究。程序综合技术也经历了与人工智能类似的发展路线，研究中心也从早期的基于推导的程序综合方法过渡到目前的基于归纳的程序综合方法。

基于推导的综合方法 (deductive program synthesis) 是从一般到特殊的推理方法[124]。1990 年之前，该领域主要的方法是基于自动机[125,126]、程序证明等推理手段[124,127,128]。基于归纳的程序综合 (inductive program synthesis) 是从特殊到一般的推理方法[122,129]，根据“输入-输出”样例等归纳得到程序的过程。目前，基于归纳的程序综合已经得到了实际应用，部署在微软的 Excel 等软件中，受到了广泛好评[130]，本节主要介绍该类别的相关研究。

微软的 Gulwani 将程序综合的问题分为三个维度，即用哪一种约束表示用户的意愿、用怎样的搜索空间以及用哪一种搜索算法[122]。其中，搜索算法是其综合技术的核心。本节以搜索算法维度，按基于枚举的方法、基于语法制导的方法和基于机器学习的方法介绍相关工作。

① https://en.wikipedia.org/wiki/Program_synthesis

2.2.1 基于枚举的方法

基于枚举的方法根据某种固定的顺序来搜索空间。此类方法虽然简单，却往往是有效的。在近期的 ICFP 和 SyGuS [①] 比赛中，该类方法取得了良好的成绩。基于枚举的方法需要首先定义一个启发式的搜索空间，之后在空间中使用一些启发式规则优先选取某些程序或者过滤某些候选程序。搜索空间可以通过自定义语法、上下文无关语言或者上下文有关语言等方式定义。

在给定的文法上，枚举的搜索方向可以是自顶向下的，也可以是自底向上的。所谓自顶向下，就是按照文法展开式，从抽象向着具体的方向展开[131,132]。而自底先上则是一个相反的，从具体一般化到抽象的过程。也有两个方向结合的搜索方法，用于处理几何构建[133] 和汇编语言超优化问题[134]。

2.2.2 基于语法制导的方法

基于语法制导的程序综合 (syntax-guided synthesis)[98] 是生成符合用户提供的标准语言的合法函数的技术。用户为了描述标准语言，需要提供：

- 待综合函数：一个有限的函数集合。
- 语法约束：一组描述待综合函数内的表达式的集合，该集合的元组满足给定的上下文无关文法。
- 语义约束：描述待综合函数的行为的逻辑公式。

在得到待综合函数所需要满足的约束后，通过约束求解器得到解。在 SyGuS 程序综合竞赛中，常用的求解器包括 CVC4[99]、EUSolver[135] 等等。

2.2.3 基于机器学习的方法

基于机器学习的程序综合学习程序分布以指导搜索，以便找到更有可能满足规约的程序。

Menon 等人提出了使用机器学习解决基于样例编程 (programming by example) 问题的方法[136]。该方法为上下文无关文法提供了表示概率的权重，得到对应的概率上下文无关文法 (probabilistic context-free grammar)。随后使用基于枚举的方法搜索程序。而文法规则的概率是通过代码仓库的“输入-输出”样例训练得到的。

Percy 等人提出了针对组合程序的有层次的贝叶斯方法，可以实现在多任务中共享部分程序[137]。作者提出了通过使用组合逻辑来表示程序，并针对这种表示提出了能够安全转换程序的 MCMC 算法。

也有研究者利用统计学习综合得到逻辑命题形式的不变量 (invariant)。Garg 等提出

[①] https://sygus.org/

基于反例引导的不变量综合技术 ICE[138]。ICE 包含“教师”和“学习者”两个角色，“学习者：统计归纳得到不变量提交给“教师”，“教师”负责提供反馈和反例返回给“学习者”，供其进一步产生更精准的不变量。Garg 等人之后使用决策树算法改进其学习模型[139]。

2.2.4　现有程序综合方法研究总结

本节介绍了现有主要的程序综合方法。程序综合方法的发展流程是从基于推理的方法逐渐过渡到基于归纳的方法。由于基于推理的方法需要完整规约，很难在大型程序上应用，而基于归纳的方法可以避免这个问题。在当前的缺陷修复方法中，主要使用的也是基于归纳的方法，例如 SketchFix[65] 使用 Sketch、Fix2Fix[35] 使用语法指导的综合、Angelix[42] 等使用基于枚举的综合方法。而这些方法所生成的程序都限制在较为简单的形式，无法支持 API 调用等复杂形式的补丁生成。

标准的程序综合问题的输入是规约，返回满足规约的程序。而程序估计问题的输入是程序上下文和规约，返回生成的程序及其存在的概率。从公式的形式上看，程序综合问题是寻找满足 $prog \in PROG \wedge prog \models S$ 的程序 $prog$；而程序估计问题是寻找满足 $argmax_{prog \in PROG \wedge prog \models S}(P(prog|C))$ 的程序 $prog$。可见虽然二者的目标都是为了生成程序，但是问题定义并不相同。尽管如此，程序综合问题可以直观地归约为程序估计问题，因为程序估计问题返回的就是满足规约的程序。与缺陷修复问题类似，基于归纳的程序综合问题也面临着过拟合问题，即在规约不完全的情况下，生成满足规约程序却不正确的程序。通过返回概率最高的程序，程序估计问题的解法也有助于提升程序综合方法的能力。

2.3　基于统计的代码生成技术

代码生成技术通过从大规模代码中学习程序模型，并通过应用模型生成程序[140,141]。由 Hindle 等人在 2012 年首次指出程序语言与自然语言都属于人类创造的语言，本质上都有一定的重复性[142]。二者必然存在统计特征，从而为利用统计学习乃至深度学习提供了可能。近年来，随着机器学习，尤其是深度学习等技术的推广，研究者提出一系列基于机器学习的代码生成方法，用于代码推荐和代码补全等任务。基于统计的方法对数据集的规模和质量有一定要求，随着开源软件社区的繁荣、Github 等开源代码仓库的兴起，为相关研究提供了基础支撑。

一种常用的处理方法是将代码视为文本，基于自然语言处理技术构建模型。通过在代码仓库中学习单词的概率分布，使用已有单词预测即将出现的单词的概率，最后将文本映射回代码的 AST 等结构以得到程序。

一种常用的统计模型是N-gram，它使用一个长度为N的窗口在序列$S = w_1w2\cdots w_m$上滑动，每个单词w_i的概率依赖于它前面的$n-1$个单词。即序列S的概率为：

$$P(S) = \prod_{i=1}^{m} P(w_i|w_{i-n+1} \cdot \ldots \cdot w_{i-1}) \tag{2.1}$$

使用 N-gram 模型用于代码补全和代码推荐的工作主要有以下代表性的工作。Raychev 等通过学习已有程序中的调用习惯，进行补全函数调用[143]。Tu 等人为 N-gram 模型添加 cache，增强其在代码相关任务的能力[144]。Raychev 等提出在有噪音的“大代码(big code)”中学习的代码生成模型的方法[145]。Raychev 等提出了基于决策树的代码概率模型用户代码补全[146]。

在学习模型选择上，研究者从早期的统计模型逐渐过渡到复杂的深度神经网络上。有研究者同样利用代码局部性，提升深度学习模型学习提取代码特征的能力[147,148]。Saxena 等人提出了 DeepCoder，使用深度学习的方法通过学习“输入-输出”样例综合程序[149]。Gu 等人提出使用 RNN 学习 API 的调用序列和自然语言规约的关系[150]。

基于统计的代码生成技术目标是在给定上下文，预测当前概率最大的程序。形式地讲，该技术是寻找空间 $PROG$ 中满足在上下文 C 下的条件概率最大的程序 $prog$，即求解满足式 $argmax_{prog \in PROG}(P(prog|C))$ 的 $prog$。可以看出，该问题与程序估计问题的定义不同，虽然两个问题都是求解上下文中概率最大的程序，程序估计问题还需要所得程序满足给定的规约。与程序综合技术类似，基于统计的代码生成技术也可以归约为程序估计问题。

第三章　程序估计问题的解决框架

3.1　引言

在许多场景我们需要自动地综合 (synthesize) 程序。该动作可视为在程序空间中搜索到满足规约且最有可能的程序。正如小节 1.4.1所述，我们将其定义为程序估计问题，即在程序空间 $PROG$ 中寻找程序 $prog$ 使之满足 $argmax_{prog \in PROG \wedge prog \models S}(P(prog|C))$。

如前文所述，程序估计问题作为一般性问题，我们可以将程序修复、程序综合和基于统计的代码生成等多个问题归约 (reduce) 到程序估计问题求解。而且程序估计问题存在多种解法。本文提出一般的抽象框架：**玲珑框架**，提供一种解决程序估计问题的方法。

程序可以表示为上下文无关文法表示的抽象语法树 (AST)。玲珑框架将上下文无关文法泛化为*扩展规则*，从而能够形式化描述了程序片段从不同方向的生成过程。类似地，AST 泛化为*扩展树*，即使用扩展规则表示树结构。扩展树可以支持程序按不同的方向展开。在每一步扩展中，可以从匹配的扩展规则中选择一个应用。扩展规则描述了全部的程序空间，扩展树描述了程序展开方式。玲珑框架将程序按照扩展规则展开的过程建模为有向图。图中的顶点为扩展树描述的程序，边为扩展规则。两个相邻的顶点表示出发顶点的程序应用扩展规则后扩展为到达顶点的程序。程序的生成过程就是沿着一条从代表空程序的顶点到一个代表完全展开程序的顶点的完整路径。通过将程序生成问题转化为路径查找问题，就可以使用图上的路径查找算法来解决。而程序的概率，就可以通过其展开路径上所有扩展规则的条件概率乘积获得。而选择扩展规则的概率，则由统计学习方法通过训练集中的代码学习模型来预测。另外，玲珑框架提供了静态分析提前将不可行的部分程序剪枝，避免消耗资源展开注定不能满足规约的程序，提升生成效率。

本章详细介绍玲珑框架：首先，3.2节以例子驱动概述该方法；其次，3.3节详细介绍玲珑框架的概念和方法，包括如何描述程序搜索空间和生成步骤 (小节3.3.1、小节3.3.2以及小节3.3.3)；再次，3.4节描述如何计算一个完整程序或部分程序的概率；最后，3.6节描述静态分析方法判断一个部分程序是否产生满足规约的程序。

3.2　方法概览

本节介绍以生成条件表达式 (例如 `if` 语句的跳转条件) 为任务的启发式案例作为玲珑框架的概览。

生成条件表达式是程序综合的典型任务，例如在基于 Sketch 的程序综合中，经常需要填补空缺的条件表达式。在程序估计问题中，可视为是在上下文中生成概率最大且满足规约的条件表达式。其中，周围的代码可被视为上下文，语言的句法、类型约束以及所需通过的测试作为规约。为了训练统计学习的模型，可将代码语料库中的条件表达式提取出来，根据它们的上下文提取特征作为训练集。

假设有这样一组文法规则用于定义条件表达式，其中 T 为根符号：

$$\mathtt{T} \rightarrow E$$

$$E \rightarrow E \text{ “} > 12\text{”} \mid E \text{ “} > 0\text{”} \mid E \text{ “} > \text{”} \ E \mid E \text{ “} + \text{”} \ E \mid \text{“}hours\text{”} \mid \text{“}value\text{”}$$

本节后续的示例均使用该文法描述程序。

3.2.1 计算程序的条件概率

一个程序可以按照文法规则自顶向下分解展开。例如，条件表达式 ”$hours > 12$” 可以认为是按照图3.1 中的三个文法规则自根符号依次向下展开而得的

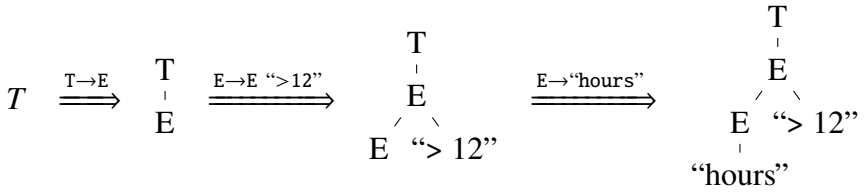

图 3.1　将程序 `hours>12` 按自顶向下方式分解

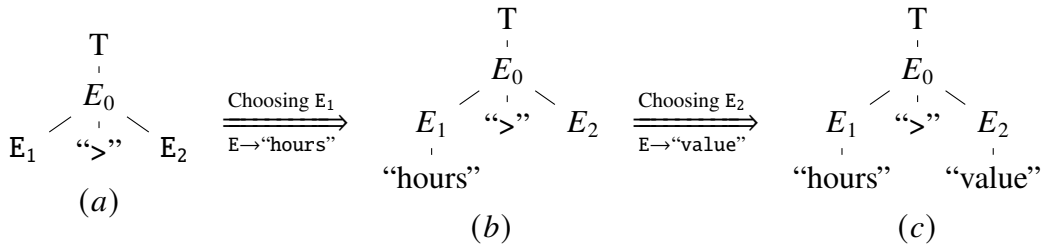

图 3.2　包括文法规则和非终结符的选择

展开选择的条件概率则可以通过训练统计学习模型来预测得到。展开选择的条件概率符合如下形式：$P(rule|context, prog, position)$。其中 $context$ 表示上下文，$prog$ 表示当前已生成的程序，$position$ 表示当前需要展开的位置，$rule$ 表示从当前 $position$ 展开所选择的规则。通过将程序按照展开过程分解，我们就能通过这个展开的事件序列来求程序的概率，即若能得到每个展开选择的条件概率，程序的概率就是所有展开时所用选择的条件概率之积。

玲珑框架需要统计学习算法能够预测概率，此外并不限定具体的学习方法，用户可以根据数据和任务特点选择合适的方法。直观上，针对不同类型的非终结符，选择不同的统计学习方法是合理的。由于输入仅包含一组程序及其上下文，在准备训练数据的时候需要将程序按扩展规则分解。在每次展开时，选中的选择当作正例 (positive instance)，而其余视为负例 (negative instance)。

在实践中，可以使用两种不同的策略选择统计学习方法。对于在实际代码中在不同的上下文内也经常出现的非终结符。例如，对于 `Statement` → `IfStatement` | `WhileStatement` | `BasicStatement`，玲珑框架训练一个多分类问题的模型，其中每个类别对应一个选择。对于变量等非终结符，由于在不同上下文环境下差异较大，因此玲珑框架训练一个二元分类器来给出该非终结符被选中的概率。值得注意的时，我们不可能将多分类器直接用于程序，因为其分类数量过于巨大，远超任何学习算法的处理能力。

当然，图3.2中所示的例子是比较特殊的，在每一步中只有一个非终结符需要被展开。但是在一般情况下，可能同时存在多个需要被展开的非终结符。因此，既要选择合适的非终结符展开选项，也要选择哪一个非终结符被展开。例如，图3.2展示的展开过程中引入了四次选择。在第一个选择中，部分程序包含两个待展开的非终结符 (E_1 和 E_2)。第一个选择选择了 E_1，第二个选择选择了文法规则 $E \rightarrow$ "$hours$"，第三个选择选择了 E_2，而第四个选择选择了文法规则 $E \rightarrow$ "$value$"。

对于非终结符节点的选择引发了如下问题：程序的概率是否等于全部选择的概率？如果是，我们如何计算展开节点的选择的概率？后文将论述，即便是引入了关于节点的选择，程序的概率依然只是文法规则选择的概率的乘积，即选择非终结符节点的概率是可以忽略的。即便是选择先展开 E_2，后展开 E_1，该程序的概率保持不变。这个性质可以使我们自由地选择被展开的节点。

3.2.2　定位空间中概率最大的程序

上一小节展示了如何估计一个程序关于上下文的条件概率，本小节展示如何定位搜索空间中概率最大的程序。

该问题可以转化为有向图上的路径查找问题。其中，图的顶点是部分 (partial) 或完整 (complete) 的程序，边是被标记为一个程序中的非终结符和一个文法规则。一个程序 p 通过被标记的边连接至另一个程序 p'，该边的标记是非终结符 v 和文法规则 g。如果将 p 的非终结符 v 按 g 展开，则得到 p'。例如图3.2展示的路径。

每条边的增益是选择文法规则的概率。该概率是通过统计学习模型估计而得。一条路径的增益是其所有边的概率乘积。起始顶点是空程序，目标顶点是一个满足给定规约的完整程序。我们的目标是找到一个从起始顶点到目标顶点且增益尽可能大的路

径。

玲珑框架并不局限路径选择的搜索算法，用户可以选择满足需求的算法。例如，Dijkstra 算法[151]、A* 算法[152] 等精确算法，或者 Beam 搜索[153] 和蒙特卡洛树查找[154] 等近似算法。

3.2.3 保证程序满足规约

路径查找问题需要一个目标顶点，其包含满足规约的完整程序。一种朴素的做法是每次生成完整程序时检测其是否满足规约。但是该做法并不高效，很多情况下，在路径查找的早期我们就能够判断一个部分程序是否能得到满足规约的完整程序。尽早发现不能满足规约的程序并将其剪枝就能够避免很多不必要的计算开销。

决定一个部分程序能否生成完整程序不是一个平凡的问题，因为部分程序中包括非终结符。例如，假设规约为如下“输入-输出”样例：当 `hours = 1` 且 `value = 10` 时，条件表达式的值为"`true`"。我们可以容易地判断图3.2(c) 中的程序无法展开得到满足的程序，而图3.2(b) 中的程序则可能展开为满足的程序。

为了解决这一问题，玲珑框架引入基于抽象解释[155] 的针对部分程序进行静态检查。通过使用合适的抽象域，我们可以分析得到非终结符所有可能的展开式的值的上界。如果根节点的上界不包含所需的输出，我们可知当前部分程序无法展开为满足规约的程序。假设我们使用整数的区间分析来展示这一过程。通过静态分析语法规则，我们已知 E 将会被展开为整数，且其取值范围是区间 $[1, +\infty]$。因为文法规则中只有加法 (+) 的存在，且 `hours` 和 `value` 都大于等于 1。并且易知 `hours` 的取值区间是 $[1, 1]$，继而有 $[1, 1] > [1, +\infty]$ 是肯定为 `false` 的。最终我们知道该部分程序不可能得到满足需求的程序。

3.2.4 其他扩展顺序

前文所述的过程是将程序按文法的自顶向下的顺序扩展。玲珑框架同样支持按照文法规则的其他扩展方向分解程序。例如，对于条件表达式 `hours>12`，我们可以假设扩展从最左边的叶子节点开始，即先决定使用哪个变量。在选择变量“`hours`’’ 之后，进一步为其选择双亲节点，即谓词“`> 12`’’。

在分解程序中，允许按照不同的顺序对于解决路径查找问题是很重要的。例如，在一些上下文情况下，我们容易预测出应该首先使用哪些变量，在变量的基础上可以较为准确地预测其谓词，最终所估计的完整程序也较为准确。另外，不同的扩展顺序在不同的路径查找算法下也会影响程序生成的速度。

玲珑框架通过将文法规则 (grammar rule) 泛化为扩展规则 (expansion rule) 来支持不同的生成顺序。给定一个文法规则，我们按照从左至右、从 0 开始的顺序为其非终

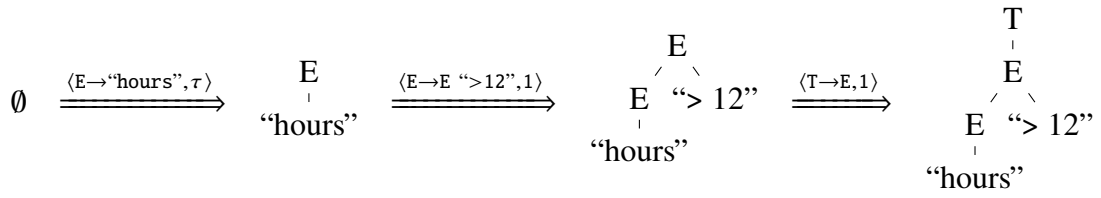

图 3.3　按自底向上的顺序分解程序

结符添加索引。例如，$\mathtt{N} \to \mathtt{N}$ " + " $\mathtt{N}$ 被标记为 $\mathtt{N}^0 \to \mathtt{N}^1$ " + " $\mathtt{N}^2$，其中上标是非终结符的索引。扩展规则符合形式 $\langle \mathtt{rule}, \mathtt{i} \rangle$，表示当第 i 个非终结符时出现时，可以应用这个规则。当 i 是 τ 时，该规则可以被应用在空树以产生第一个节点。例如，图3.3展示如何从最左边的变量按自底向上的方向使用扩展规则扩展得到表达式 "`hours > 12`"。

通过使用不同的扩展规则集合，我们可以将程序分解为一系列选择。例如，如果需要按照原始的文法规则一样以自顶向下的顺序展开，我们只需将每个语法规则和 0 配为序对，以得到扩展规则集；如果需要按照图3.3中展示的自底向上顺序展开，则可以使用如下扩展规则集：

$$
\begin{array}{ll}
\langle E \to \text{“}hours\text{”}, & \tau \rangle \quad \langle E \to \text{“}hours\text{”}, \quad 0 \rangle \\
\langle E \to \text{“}value\text{”}, & \tau \rangle \quad \langle E \to \text{“}value\text{”}, \quad 0 \rangle \\
\langle E \to E \text{ “} > 12\text{”}, & 1 \rangle \quad \langle E \to E \text{ “} > 12\text{”}, \quad 0 \rangle \\
\langle E \to E \text{ “} > 0\text{”}, & 1 \rangle \quad \langle E \to E \text{ “} > 0\text{”}, \quad 0 \rangle \\
\langle E \to E \text{ “} + \text{” } E, & 1 \rangle \quad \langle E \to E \text{ “} + \text{” } E, \quad 0 \rangle \\
\langle E \to E \text{ “} > \text{” } E, & 1 \rangle \quad \langle E \to E \text{ “} > \text{” } E, \quad 0 \rangle \\
\langle T \to E, & 1 \rangle
\end{array}
\tag{3.1}
$$

其中，左列是向上扩展树的规则，右侧是将剩余非终结符向下展开的规则。

为了保证空间中的所有程序能够被分解为一系列选择，扩展规则集合必须满足**完全性**。为了保证程序的概率是其所有选择的乘积，扩展规则集合必须满足**唯一性**。这两个性质能够保证用扩展规则无歧义地描述程序展开的过程。稍后本文会定义完全性和唯一性，并描述产生完全并唯一的扩展规则集合的方法。

通过将文法规则泛化为扩展规则，路径查找问题和基于静态分析的剪枝方法也可以被类似的方法定义。后文将展示该定义。

3.3　扩展规则

本节介绍定义了上下文无关文法 (context-free grammar, CFG) 和抽象语法树 (AST)，以及从文法推导得到扩展规则的过程。为了描述按不同方向展开，本节将文法规则泛

化得到扩展规则。类似地，也将基于文法规则的 AST 泛化为基于扩展规则的扩展树。本节描述扩展规则和扩展树的性质以及扩展树与 AST 之间相互转化的方法。

3.3.1 文法规则和抽象语法树

定义 3.1 (语法规则 (Grammar Rule)). 给定一个终结符集合 Σ，一个非终结符集合 N，以及一个根符号 T。其中，集合 Σ，N 和 $\{T\}$ 互不相交。一条**文法规则** r 是一个满足如下形式的符号序列：$n_0 \to n_1 \ldots n_k$，其中，称 $n_0 \in N \cup \{T\}$ 为**左符号** *(left symbol)*、$n_1 \ldots n_k \in N \cup \Sigma$ 为**右符号** *(right symbol)*。当左符号为根符号时，称该规则为**起始规则** *(start rule)*。

针对文法规则的相关符号，本文引入一些操作符。本文使用 $g[i]$ 表示文法规则 g 中的第 i 个右符号，其中 $i \in \mathbb{N} \wedge i \geq 1$。另外，$|g|$ 用于表示 g 的右符号中非终结符的数量。若 g 不是起始规则，则使用 $g[0]$ 表示其左符号，否则 $g[0]$ 是未定义的。同时，对于任意的 $i > |g|$，符号 $g[i]$ 也是未定义的。使用 $g[i]\downarrow$ 表示 $g[i]$ 是被定义的。

生成程序的过程可以被视为是从根符号开始一系列文法规则的应用。每次应用，如果上一步的规则 g' 生成的一个非终结符匹配了规则 g 的左符号，则可用 g 进一步展开。为了描述 g 和 g' 的关系，我们引入可连接性的定义。

定义 3.2 (语法规则的可连接性 (Connectivity)). 如果 $0 < i < |g'|$ 且 $g[0] = g'[i]$，则称文法规则 g 可在第 i 个符号连接至文法规则 g'，记为 $g \dashv\in g'[i]$。

例如，规则 $\mathtt{E} \to$ "hours" 可连接至规则 $\mathtt{E} \to \mathtt{E}$ " > 12" 的第一个右符号。

我们可以根据规则的连接过程定义抽象语法树 (AST)，可以将连接至右符号的规则视为孩子节点，将右符号中的非终结符被连接的规则视为双亲节点，如图3.4中所示的两个程序的语法树。

程序	hours>12	hours+value
AST	$\mathtt{T} \to \mathtt{E}$ $\mathtt{E} \to \mathtt{E}$ " > 12" $\mathtt{E} \to$ "hours"	$\mathtt{T} \to \mathtt{E}$ $\mathtt{E} \to \mathtt{E}$ " + " $\mathtt{E}$ $\mathtt{E} \to$ "hours" $\mathtt{E} \to$ "value"

图 3.4 抽象语法树样例

定义 3.3 (抽象语法树 (AST)). 一个 *(部分)* 抽象语法树是一个四元组 (V, p, G, l)，其中 V 是顶点 *(vertex)* 集合，$p : V \to ((V \times Nat) \cup \tau)$ 是一个单射的双亲函数，*(*其中，$p(v) = (v', i)$ 表示 v 是 v' 的第 i 个孩子，且对于根顶点 $p(v_{root}) = \tau$*)*，G 是一组文法规则，$l : V \to G$ 是标记函数，将每个顶点映射到一条语法规则，使得：

- 令 $q^0(v) = \{v\}, q^i(v) = \left\{v'' \mid \exists j, v' \in q^{i-1}(v) :\ p(v') = (v'', j)\right\}$，我们知道对于任意的 $v \in V$，不存在 k 使得 $v \in q^k(v)$，即没有环存在。
- $p(v) = (v', i)$ 可推出 $l(v) \dashv\in l(v')[i]$，即一个孩子规则可连接至其双亲规则。
- $p(v) = \tau \implies \neg l(v)[0]\downarrow$，即根顶点被关联至起始规则。

其中，双亲函数的单射性可保证根顶点的唯一性。

给定一个 AST (V, p, G, l)，我们使用 $nb^p(v, i)$ 表示 v 的第 i 个邻居顶点。当 $i = 0$ 时表示双亲顶点，$i > 0$ 时表示第 i 个孩子顶点。

如果 AST 中还存在未被展开的非终结符，则该部分 (partial) AST 可以被进一步展开。形式化地，对于任意的 (v, i) 且有 $0 < i < |g| \wedge \forall v' \in V, p(v') \neq (v, i)$ 成立的，被成为是**文法展开位置**。如果一个 AST 没有文法展开位置了，则称其为**完全的**。

3.3.2　扩展规则和扩展树

基于文法规则和 AST 的定义，我们可以将文法规则泛化为扩展规则 (expansion rule)。一个扩展规则包含一个文法规则和一个索引 (intex) 指示出该规则是从哪个非终结符展开的。如果索引是 τ，则该扩展规则是创始规则 (creation rule)，且被应用到创建 AST 的第一个顶点。

定义 3.4 (扩展规则 (Expansion Rule)). 一个**扩展规则** r 是一个序对 (g, i)，其中 g 是文法规则，$i \in Nat \cup \{\tau\}$，则有 $i \neq \tau \implies g[i]\downarrow$。当 $i = 0$ 时，r 被称作**自顶向下规则** *(top-down rule)*；当 $i > 0$ 时，r 被称作**自底向上规则** *(bottom-up rule)*；当 $i = \tau$ 时，r 被称作**创始规则** *(creation rule)*。本文用 r^g 表示其文法规则 g，r^i 表示其索引 i。当 i 不是 τ 时，我们称 g 的第 i 个符号是扩展规则 (g, i) 的**起始符号** *(start symbol)*。

与文法规则类似，我们也定义扩展规则的可连接性。直观上，如果两个扩展规则的文法规则是连接的，并且一个规则的起始符号是被另一个规则产生的，则两个扩展规则是连接的。

定义 3.5 (扩展规则的可连接性). 如果对于文法规则 g 和 g' 有 $g \dashv\in g'[j]$ 且 $(i = 0 \wedge i' \neq j) \vee (i \neq 0 \wedge i' = j)$ 成立，则称扩展规则 $r = (g, i)$ 在第 j 个非终结符**可连接**至扩展规则 $r' = (g', i')$，记作 $r \dashv\in r'[j]$。

抽象语法树描述了程序的关于文法规则集合的结构。将文法规则泛化为扩展规则之后，程序的生成过程也可以用扩展规则的应用来描述。为了用扩展规则描述程序的结构，本文引入**扩展树** (expansion tree) 概念。扩展树与抽象语法树类似，只是其顶点使用扩展规则标记，而非文法规则。

程序	hours>12	hours+value
扩展树	$(\mathtt{T} \rightarrow \mathtt{E}, 1)$ ↑ $(\mathtt{E} \rightarrow \mathtt{E}\ \text{“}>12\text{”}, 1)$ ↑ $(\mathtt{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)$	$(\mathtt{T} \rightarrow \mathtt{E}, 1)$ ↑ $(\mathtt{E} \rightarrow \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)$ ↗ ↘ $(\mathtt{E} \rightarrow \text{“}\mathtt{hours}\text{”}, \tau)$ $(\mathtt{E} \rightarrow \text{“}\mathtt{value}\text{”}, \mathbb{0})$

所示扩展树是基于式3.1中的扩展规则集合。

图 3.5　扩展树样例

定义 3.6 (扩展树 (Expansion Tree)). 一个**扩展树** 是一个元组 (V, p, R, ϕ)，其中 V 是顶点集，$p : V \rightarrow ((V \times Nat) \cup \tau)$ 是单射的双亲函数，R 是扩展规则集合，$\phi : V \rightarrow R$ 是将每个顶点映射至某个扩展规则的标记函数，并满足：

- 令 $q^0(v) = \{v\}, q^i(v) = \left\{v'' \mid \exists j, v' \in q^{i-1}(v) :\ p(v') = (v'', j)\right\}$，我们可知对于任意的 $v \in V$，不存在 k 使得 $v \in q^k(v)$，即保证无环。
- $p(v) = (v', i)$ 表明 $\phi(v) \dashv \phi(v')[i]$，即孩子规则应该可连接至其双亲规则。
- $\exists v \in V : \phi(v)^i = \tau$，即有且仅有一个 创始规则。

类似地，给定一个扩展树 (V, p, R, ϕ)，我们使用 $nb^p(v, i)$ 表示 $v \in V$ 的第 i 个邻居顶点。

例如，图3.5 展示了扩展树样例，其中箭头表示了扩展的次序。

引理 3.1. 只存在一个 $v \in V$ 使得 $\phi(v)^i = \tau$，即只有一个顶点有创始规则。此时，我们称 v 为该扩展树的初始顶点 *(initial vertex)*。

证明. 假设存在两个顶点都有创始规则。因为顶点集合会形成一个树，在这两个顶点之间只有一条路径。因此，必然有一个顶点在这个路径中。而该路径中有两个起始符号，产生矛盾。 □

与 AST 类似，一个扩展树如果存在一个**扩展位置** (expansion position)，则它可被进一步扩展。当扩展树不是空树时，扩展位置是还没有被其他规则进一步扩展的非终结符，当扩展树是空树时，扩展位置为 τ。其定义如下：

定义 3.7 (扩展位置 (Expansion Position)). 给定一个扩展树 $t = (V, p, R, \phi)$，一个**扩展位置**是以下某一种情形：

- τ，当 $V = \emptyset$，即 t 为空树时，或者
- 任意的 (v, i) 当 $|V| > 0$，使得：
  - $\phi(v)^g[i]\downarrow$，即 v 中存在非终结符；
  - $(i = 0 \implies p(v) = \tau)$，即该非终结符是左符号且该顶点没有双亲顶点；

- $(i \neq 0 \implies \forall v' \in V : p(v') \neq (v, i))$，即该非终结符是右符号且它没有对应的孩子顶点。

一个扩展树如果没有扩展位置，则被称为是**完全的**。

定义 3.8 (候选扩展 (Expansion Candidate)). 给定一个扩展树 $t = (V, p, R, \phi)$ 和一个扩展位置 ρ，一个关于 t 和 ρ 的 **候选扩展** 是以下某一种情形：

- 是一个创始规则，当 $\rho = \tau$；
- 是一个扩展规则 $r = (g, j)$，当 $\rho = (v, i)$ 满足：
  - $r \dashv\in \phi(v)[i]$ 且 $i \neq 0$；
  - $\phi(v) \dashv\in r[j]$ 且 $i = 0$。

最后，我们定义如何根据候选扩展实现扩展树的增长。

定义 3.9 (扩展后的树 (Expanded Tree)). 给定一个扩展树 $t = (V, p, R, \phi)$，一个扩展位置 ρ 以及一个扩展候选 $r = (g, j)$，一个关于 t、ρ 和 r 的 **扩展后的树** (V', p', R, ϕ') 是：

- $V' = V \cup \{v'\}$，其中 v' 是不在 V 中的新顶点；
- $\phi' = \phi \cup \{v' \mapsto r\}$；
- 当 $\rho = \tau$ 时，$p' = \{v' \mapsto \tau\}$；
- 当 $\rho = (v, i)$ 时，$(i \neq 0 \implies p' = p[v' \mapsto (v, i)]) \wedge (i = 0 \implies p' = p[v \mapsto (v', j)][v' \mapsto \tau])$。

其中，$p[v \mapsto (v', j)]$ 是一个新函数，将 v 映射到 (v', j)，且其余任意顶点 v'' 至 $p(v'')$。

3.3.3　抽象语法树和扩展树之间的相互转化

前文定义了两种树结构：基于文法规则的抽象语法树 (AST) 和基于扩展规则的扩展树。二者之间能够转化是十分重要的，因为玲珑框架的预测得到的是扩展树，而最终需要将扩展树转化为 AST 以得到最终程序；而在训练过程中，玲珑框架需要将 AST 转化为扩展树来获得训练集。

先考虑如何将扩展树转化为 AST。因为 AST 需要从根符号开始生成，因此不是所有的扩展树都可以转化为 AST，只有那些根部的扩展规则的文法符号是根符号的扩展树才可以。换句话说，可以进一步向上展开的扩展树是不能被转化的。转化过程很简单，只需将扩展树的每个扩展规则的文法规则提取出来，标记在相同树形的 AST 的对应顶点即可。

定义 3.10 (扩展树转化为 AST). 若要扩展树 (V, p, R, ϕ) 生成 *AST* (V, p, G, l)，使 $\forall v \in V, \phi(v)^g = l(v)$ 即可。

定理 3.1. 如果有 $v \in V$, $p(v) = \tau$ 且 $\neg\phi(v)^g[0]\downarrow$，则 (V, p, R, ϕ) 生成 $(V, p, \{r^g \mid r \in R\}, \{v \mapsto \phi(v)^g \mid v \in V\})$。

证明. 可从定义易得。 □

但是从 AST 转化为扩展树并不直观。首先，给定 AST (V, p, G, l) 和一个扩展规则集合 R，并不是能够保证转化一直成立。例如，如果 AST 中的一个顶点被关联至文法规则 g，则没有其他的 $r \in R$ 使得 $r^g = g$。其次，给定一个 AST 可能存在多种转化方式。例如，如果扩展规则包括全部的自顶向下规则和自底向上规则，则 AST 既可以沿着向下方向转化，也可以向上方向转化。

为了方便后续统计学习模型的训练，我们希望 AST 可以转化为唯一的扩展树。同时，这样做也有利于程序的条件概率的计算。因此，我们期待扩展规则满足**完全性** (completeness) 和 **唯一性** (uniqueness)。

定义 3.11 (完全性 (Completeness)). 如果对于任意的 *AST* (V, p, G, l)，存在一个扩展树 (V, p, R, ϕ) 可以生成该 *AST*，则称扩展规则集合 R 关于文法规则集合 G 是**完全的**。

定义 3.12 (唯一性 (Uniqueness)). 如果对于任意的 *AST* (V, p, G, l)，最多存在一个扩展树 (V, p, R, ϕ) 能够生成该 *AST*，则称扩展规则集合 R 关于文法规则集合 G 是**唯一的**。

如果一个规则集合是完全并且唯一的，我们称一个其为正则的。我们介绍使规则集合正则的充分条件。直觉上，这个规则集合需要能有创始规则用于生成初始顶点，并有一组自底向上的规则能够向上伸展树，直至达到树根，以及一组自顶向下规则生成剩余的非终结符。

定义 3.13 (正则规则集 (Regular Rule Set)). 关于一个文法集合 G 的 **正则规则集** 是满足如下条件的最小的扩展规则集合：

- $g \in G \wedge g[0]\downarrow \implies (g, 0) \in R$,
- $\forall root, child \in G : \neg(root[0]\downarrow) \wedge (child, root) \in ChildRules^* \implies isOK(child)$, 其中
  - $ChildRules = \{(child, parent) \mid \exists i : child \dashv\in parent[i] \wedge (parent, i) \in R\}$;
  - $isOK(g) ::= |BURules(g)| = 1 \vee (|BURules(g)| = 0 \wedge (g, \tau) \in R)$;
  - $BURules(g) ::= \{(g, i) \in R \mid i \neq 0\}$。

第一个条件保证了每条不是起始规则的文法规则包括了一条自顶向下规则。第二个条件保证了每个 AST 只会有一个顶点被创始规则创建，并且从有创始规则的顶点到树根的路径中只有一条是自底向上规则构成的。第二个条件是按照反方向检测的。首先，对于每个起始文法规则，检测 *isOK* 是否满足，即检查“它被唯一的自底向上规

则的扩展规则关联，或者它是创始规则”。换句话说，将起始规则转化为一个扩展规则的方法是唯一的。如果在 i 有起始符号被自底向上规则所关联，我们对可在 i 处连接至该起始规则任意文法规则再次实施这个检查。如此递归地检查，直至遇到创始规则。形式化地讲，这个递归检查形成一个关于 $ChildRule$ 关系的自反传递闭包，记作 $ChildRule*$。

例如，一个包含全部自顶向下规则的规则集合 (即 $\{(g,0) \mid g \in G \wedge g[0]\downarrow\} \cup \{(g,\tau) \mid g \in G \wedge \neg(g[0]\downarrow)\}$) 是一个正则规则集合。并且，规则集合式3.1是一个正则规则集。

接下来证明正则规则集是如何保证完全性和唯一性的。下面的引理说明了创始规则的位置决定了一个扩展树。

引理 3.2. *假设有 (V,p,R,ϕ) 和 (V,p,R,ϕ') 生成 (V,p,G,l).。如果 $\exists v \in V, g \in G : \phi(v) = \phi'(v) = (g,\tau)$，则有 $\phi = \phi'$。*

证明. 根据树的定义，顶点 v 和起始符号之间有且仅有一条路径。因此，顶点 v 的起始符号一定被关联至该路径，即对于顶点 v 存在唯一的扩展规则选择。 □

根据该引理，我们有如下定理：

定理 3.2. *一个关于 G 的正则规则集合 R 是完全且唯一的。*

证明. 首先，我们看根据任意的 AST (V,p,G,l) 构建扩展树 (V,p,R,ϕ) 的完全性。首先，令 v 为根顶点。根据正则规则集的第二个条件，我们可知肯定存在一个创始规则 $(l(v),\tau)$ 或一个自底向上规则 $(l(v),i)$。若是后者存在，令 v 是 v 的第 i 个孩子，并重复这一过程。因为树的高度是有限的且叶节点没有自底向上规则 (因右符号中没有非终结符)，该过程一定会在某个创始规则 $(l(v'),\tau)$ 存在的顶点 v' 处停止。之后我们为该过程中访问过的节点选择创始规则或自底向上规则，并为其余节点选择自顶向下规则。这些选择形成一个扩展树。

其次，我们来看唯一性。因为根顶点不能被标记为一个自顶向下规则，上述过程只可能在唯一的有创始规则的顶点处终止。又因为引理3.2，我们可知该扩展树唯一。 □

定义3.13同样提供一个根据一个文法规则创建正则扩展规则集的方法。算法3.1展示该方法。首先，我们为每个不是起始规则的文法规则添加对应的自顶向下规则。之后对每个起始规则 g，我们决定是添加对应的创始规则或者是自底向上规则。如果我们选择添加一个自底向上规则 (g,i)，我们需要找到所有可在 i 处连接至 g 的文法规则，并决定为这些文法规则是添加创始规则还是自底向上规则。重复上述过程直到没有剩余的自底向上规则。

Algorithm 3.1: 将一个文法规则集转化为一个正则扩展规则集

```
Input: G: 一个文法规则集合
Output: R: 一个扩展规则集合
foreach g ∈ G 满足 g[0]↓ do
    R ← R ∪ {g, 0}
end
foreach g ∈ G 满足 ¬g[0]↓ do
    AddRule(g)
end
Procedure AddRule(g)
    if |g| > 0 then
        决定是添加一条自底向上规则还是一条创始规则.
        if 选择的是 bottom-up then
            决定起始符号的索引 i, 其中 1 ≤ i ≤ |g|.
            R ← R ∪ {g, i}
            foreach g ∈ G 使得 g′ ∈ g[i] do
                AddRule(g)
            end
            return
        end
    end
    R ← R ∪ {g, τ}
end
```

最后，算法3.2展示了如何将一个扩展树转化为 AST。该算法的复杂度为 $O(nm^2)$，其中 n 是 AST 的节点数，m 是文法规则中右侧非终结符的最大个数。一般而言，m 较小，因此该算法实际复杂度接近线性。

算法的核心是一个顶点可以被映射的扩展规则是被“它的孩子顶点是否被向上或向下构建”决定的。该算法后序遍历树，根据孩子顶点的属性决定标记顶点为自顶向下规则、自底向上规则或者创始规则。其中，二维数组 `Feasible` 用于记录该信息，并记录用于产生结果的具体规则。

3.4 扩展树的概率

本小节展示一个扩展树的概率是其在扩展的过程中所选择的候选扩展的概率的乘积。其中选择扩展位置的概率可以被忽略。首先，本文引入概念**扩展序列**来表示扩展的过程。

定义 3.14 (扩展序列 (Expansion Sequence)). 扩展序列是一个形如 $(\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}\rangle)_{\mathtt{i=1}}^{\mathtt{n}}$ 的三元组序列，其中 $\mathtt{prog}_1 = (\emptyset, \emptyset, R, \emptyset)$；当 $1 \le i \le n$ 时，$\mathtt{pos_i}$ 是一个 $\mathtt{prog_i}$ 的扩展位置；$\mathtt{rule_i}$ 是关于 $\mathtt{prog_i}$ 和 $\mathtt{pos_i}$ 的一个候选扩展；$\mathtt{prog}_{\mathtt{i+1}}$ 是基于 $\mathtt{prog_i}$，pos_i 以及 $\mathtt{rule_i}$ 扩展的扩展树。更一般地，我们用 $\mathtt{prog}_{\mathtt{n+1}}$ 表示基于 $\mathtt{prog_n}$，pos_n 以及 $\mathtt{rule_n}$

Algorithm 3.2: 将 AST 转化为扩展树

Input: (V, p, G, l): AST
Input: R: 扩展规则集合
Output: ϕ: 从一个顶点到一个扩展规则的映射
Data: `Feasible`: 从一个顶点和一个布尔值到一个扩展规则的映射，或用 `Nil` 表示不可行。其中，布尔值表示是否沿着自顶向下的方向。

// 初始化 `Feasible`
foreach $v \in V$ **do**
    $\texttt{Feasible}[v, \mathit{true}] \leftarrow \texttt{Nil}$
    $\texttt{Feasible}[v, \mathit{false}] \leftarrow \texttt{Nil}$
end
// 计算后序遍历的解
foreach v 在 V 的后序遍历 **do**
    foreach $r \in R$ 使得 $r^g = l(v)$ **do**
        if $\texttt{feasible}(r, v)$ **then** $\texttt{Feasible}[v, i = 0] \leftarrow r$;
    end
    if $\neg \mathit{Feasible}[v, \mathit{true}] \wedge \neg \mathit{Feasible}[v, \mathit{false}]$ **then return** `Nil`;
end
// 将解恢复为前序遍历
$root \leftarrow$ 根节点
$\phi[root] \leftarrow \texttt{Feasible}[root, \mathit{true}] \neq \texttt{Nil}\ ?\ \texttt{Feasible}[root, \mathit{true}]\ :\ \texttt{Feasible}[root, \mathit{false}]$
foreach v 在 V 中的前序遍历，使得 $v \neq root$ **do**
    $(v_p, i) \leftarrow p(v)$
    $(g, j) \leftarrow \phi[v_p]$
    $\phi[v] \leftarrow \texttt{Feasible}[v, i \neq j]$
end
return ϕ
Function $\mathit{feasible}(r, v) : \mathit{Boolean}$
    // 确定 v 是否能映射到 (g, i).
    $(g, i) \leftarrow r$
    if $i \neq 0 \wedge \neg \mathit{Feasible}[nb^p(v, i), \mathit{false}]$ **then return** *false*;
    forall j 使得 $0 < j < |g| \wedge j \neq i$ **do**
        if $\neg \mathit{Feasible}[nb^p(v, j), \mathit{true}]$ **then return** *false*;
    end
    return *true*
end

扩展得到的扩展树。

若存在一个扩展序列 $(\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}\rangle)_{i=1}^{n}$ 满足 $\mathtt{prog} = \mathtt{prog_j}$，$\mathtt{prog'} = \mathtt{prog_k}$，且有 $0 \le j < k \le n+1$，则称扩展树 prog′ 可由 prog **扩展**得到。

假设有一种策略能够从多个扩展点 (expansion point) 中选择其一。该策略时一个函数，将一个不完全的扩展树映射到一个扩展点。自然地，可以做出如下假设：(1) 一个程序的概率与选择的策略独立；(2) 为一个扩展位置选择候选扩展的概率与该策略独立。

接下来介绍关于概率的定理及其证明。首先，需要证明以下有关扩展序列的唯一性的引理。

引理 3.3. *给定唯一的扩展规则集合 R，一个选择策略 policy 和一个 AST prog，至多存在一个基于 R 的扩展序列 $(\langle \mathtt{prog_i}, \mathtt{position_i}, \mathtt{rule_i}\rangle)_{i=1}^{n}$ 使得：对任意的 $1 \le i \le n$，有 $\mathtt{policy}(\mathtt{prog_i}) = \mathtt{position_i}$；且 $\mathtt{prog_{n+1}}$ 生成 t。*

证明. 根据唯一的扩展规则集合的定义可得。 □

定理 3.3. *给定一个唯一的扩展规则集合 R, AST prog, 上下文 context, 一个满足 $\mathtt{prog_{n+1}}$ 生成 $\mathtt{prog_n}$ 的扩展序列 $(\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}\rangle)_{i=1}^{n}$。我们可得：*

$$\Pr(\mathtt{prog} \mid \mathtt{context}) = \prod_{i} \Pr(\mathtt{rule_i} \mid \mathtt{context}, \mathtt{prog_i}, \mathtt{pos_i}).$$

证明. 在接下来的证明中，因为上下文存在于所有的条件概率的条件中，我们忽略 context。令 policy 产生上述扩展序列的策略。因为策略与当前程序无关，所以有 $\Pr(\mathtt{prog}) = \Pr(\mathtt{prog} \mid \mathtt{policy})$。根据引理3.3，我们有：

$$\begin{aligned}
& \Pr(\mathtt{prog}) \\
= \ & \Pr(\mathtt{prog} \mid \mathtt{policy}) \\
= \ & \Pr((\langle \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}\rangle)_{i=1}^{n} \mid \mathtt{policy}) \\
= \ & \Pr(\mathtt{prog_1} \mid \mathtt{policy}) \Pr(\mathtt{pos_1} \mid \mathtt{policy}, \mathtt{prog_1}) \Pr(\mathtt{rule_1} \mid \mathtt{policy}, \mathtt{prog_1}, \mathtt{pos_1}) \\
& \Pr(\mathtt{prog_2} \mid \mathtt{policy}, \mathtt{prog_1}, \mathtt{pos_1}, \mathtt{rule_1}) \ldots \\
& \Pr(\mathtt{prog_{n+1}} \mid \mathtt{policy}, (\mathtt{prog_i})_{i=1}^{n}, (\mathtt{pos_i})_{i=1}^{n}, (\mathtt{rule_i})_{i=1}^{n}).
\end{aligned}$$

考虑到 $\mathtt{prog_i}$，pos_i 和 $\mathtt{rule_i}$ 决定了 $\mathtt{prog_{i+1}}$，policy 决定了 $\mathtt{pos_i}$，以及 R 决定了 $\mathtt{prog_1}$，可知 $\Pr(\mathtt{prog_{i+1}} \mid \mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}, \ldots) = 1$，$Pr(pos_i \mid policy, \ldots) = 1, Pr(prog_1 \mid \ldots) = 1$。并且我们有：

$$\begin{aligned}
\Pr(\mathtt{prog}) &= \textstyle\prod_i \Pr(\mathtt{rule_i} \mid \mathtt{policy}, (\langle \mathtt{prog_j}, \mathtt{pos_j}, \mathtt{rule_j}\rangle)_{j=1}^{i-1}, \mathtt{prog_i}, \mathtt{pos_i}) \\
&= \textstyle\prod_i \Pr(\mathtt{rule_i} \mid \mathtt{policy}, \mathtt{prog_i}, \mathtt{pos_i}) \\
&= \textstyle\prod_i \Pr(\mathtt{rule_i} \mid \mathtt{prog_i}, \mathtt{pos_i}).
\end{aligned}$$

□

3.5　路径查找问题

本小节形式化定义了如何在扩展规则的基础上，将程序估计问题转化为路径查找问题。给定如下的项：

- 文法集合 G；
- 基于 G 的唯一且完全的扩展规则集合 R；
- 上下文 context；
- 规约 spec，它是一个关于定义在 G 上的完全的 AST 的谓词；
- 概率估计函数 pr，其输入是一个扩展树 prog，一个扩展位置 pos 以及一个候选扩展 rule，并返回关于概率 $\mathtt{Pr}(\mathtt{rule} \mid \mathtt{prog}, \mathtt{pos}, \mathtt{context})$ 的估计值。

我们可以定义路径查找问题如下：

- 图 (V, E)，其中：
  - V 是 (可能有限的) 集合，包含所有定义在 R 上的扩展树，即所有的扩展树满足形式 $(\_, \_, R, \_)$，
  - $(\mathtt{prog}_1, \mathtt{prog}_2) \in \mathtt{E}$ 当且仅当 $\mathtt{prog}_1$ 和 $\mathtt{prog}_2$ 都是扩展树，并且 $\mathtt{prog}_1$ 存在一个扩展位置 pos 和一个扩展候选 rule，使得 $\mathtt{prog}_2$ 是一个由 $\mathtt{prog}_1$，pos 和 rule 扩展而得的树。
- 一个增益函数 $gain$，定义了图中的一条路径的增益：

$$gain(e_1 e_2 \ldots e_n) = \prod_{i=1}^{n} pr(\mathtt{prog_i}, \mathtt{pos_i}, \mathtt{rule_i}).$$

  其中 $e_1 e_2 \ldots e_n$ 是 n 条构成图的边，$\mathtt{prog_i}$ 是 e_i 的起始顶点，$\mathtt{pos_i}$ 是 e_i 的扩展点，$rule_i$ 是 e_i 的候选扩展。
- 起始顶点 $(\{\}, \{\}, R, \{\}) \in V$。
- 一个目标顶点集合 $\{v \in V \mid$ 存在 AST t, 受限制于 t 是完全的, v 生成 t 且 $\mathtt{spec}(t)\}$。

路径查找问题的目标是找到一条从其起始顶点到目标顶点的增益尽可能大的路径。因为 R 是唯一的，基于定理3.3，我们查找的是估计概率值尽可能大的 AST。

在路径查找中，玲珑框架并不限制搜索单一的算法，用户可自行选择合适的算法，例如 Dijkstra 算法[151]、A* 算法[152]、、Beam 搜索[153] 或者贪心算法等。

3.6　非法部分程序的剪枝

本小节介绍如何在对扩展规则的静态分析的基础上剪除不可行 (infeasible) 的部分程序。不可行的部分程序其所有可能生成的完整程序都无法满足规约。首先，玲珑框架假设所有规约都能表示为关于扩展规则的函数，其被称为**约束函数**。每个约束函数都基于在其他非终结符上收集的值并在起始符号上产生输出。给定一个扩展树，约束

函数为每个顶点产生一个**顶点属性**。在初始顶点的顶点属性决定了该扩展树能否满足规约。

定义 3.15 ((具体) 约束函数 ((Concrete) Constraint Function)). 一个基于集合 D 的 *(具体的)* 约束函数 cs 将扩展规则 (g,i) 映射至一个函数。该函数的输入是规则 g 中除去第 i 个非终结符所对应的 D 的子集序列，输出是规则中的第 i 个非终结符 D 的子集。当 i 是 τ 时，该函数的输出为布尔值 *(boolean)*，表示规约是否被满足。

常见的规约可以用约束函数表达，如用于描述程序语义的“输入-输出”样例等。图3.6中左列给出了两个不同的输入-输出样例的规约。该图的右列仅展示扩展树中应用的扩展规则，但是在实际应用中，所有的扩展规则都需要定义程序语义函数。具体的约束函数不一定是可计算的。之后会介绍在部分扩展树上实施检查的抽象约束函数。

$$cs(\mathtt{T} \to \mathtt{E}, 1) = \{\mathtt{true}\}$$
$$cs(\mathtt{E} \to \mathtt{E}\ \text{“}>12\text{”}, 1)(E_0) = \{e_1 \mid e_0 \in E_0 \wedge e_0 = \mathtt{true} \to e_1 > 12 \wedge e_0 = \mathtt{false} \to e_1 \le 12\}$$
$$cs(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)(E_0) = \begin{cases} \mathit{true} & \text{if } 13 \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$[\![(\mathtt{T} \to \mathtt{E}, 1)]\!] = \{\mathit{true}\}$$
$$\uparrow$$
$$[\![(\mathtt{E} \to \mathtt{E}\ \text{“}>12\text{”}, 1)]\!] = \{v \mid v > 12\}$$
$$\uparrow$$
$$[\![(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)]\!] = \mathit{true}$$

(a) 对于“输出-输出”样例 `hours=13`、`ret=true` 和程序 `hours > 12`。

$$cs(\mathtt{T} \to \mathtt{E}, 1) = \{10\}$$
$$cs(\mathtt{E} \to \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)(E_0, E_2) = \{e_1 \mid e_0 \in E_0 \wedge e_2 \in E_2 \wedge e_0 = e_1 + e_2\}$$
$$cs(\mathtt{E} \to \text{“}\mathtt{value}\text{”}, 0) = \{1\}$$
$$cs(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)(E_0) = \begin{cases} \mathit{true} & \text{if } 13 \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$[\![(\mathtt{T} \to \mathtt{E}, 1)]\!] = \{10\}$$
$$\uparrow$$
$$[\![(\mathtt{E} \to \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)]\!] = 9$$
$$\nearrow \qquad \searrow$$
$$[\![(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)]\!] = \mathit{false} \qquad [\![(\mathtt{E} \to \text{“}\mathtt{value}\text{”}, \mathtt{0})]\!] = \{1\}$$

(b) 对于“输入-输出”样例 `hours=13`、`value=1`、`ret=10` 和程序 `hours + value`。

图 3.6　示例语义约束方程及其具体属性

除了程序的语义约束，类型约束也可以用约束函数表示。图3.7 左列展示了用于类型检测的约束函数。其基本思路是计算某非终结符所有可能的类型。在程序综合和缺陷修复的补丁生成中，还有一种常见的约束是程序规模约束，以避免生成过于复杂庞大的程序。图3.8 左列展示了程序规模约束，其中的约束函数通过将被展开的顶点的数量加一来计算顶点数量。

基于约束函数，接下来给出 (具体) 顶点属性的定义。

定义 3.16 ((具体) 顶点属性 ((Concrete) Vertex Attribute)). 给定一个完全的扩展树 (V, p, R, ϕ) 和一个约束函数 cs，对于顶点 $v \in V$ 的 *(具体)* 属性记作 $[\![v]\!]$，其定义如

$cs(\mathtt{T} \to \mathtt{E}, 1) = \{\mathtt{Boolean}\}$ $cs(\mathtt{E} \to \mathtt{E}\ \text{“+”}\ \mathtt{E}, 1)(E_0, E_2) =$ $\quad E_0 \cap E_2 \cap \{\mathtt{Int}, \mathtt{Float}\}$ $cs(\mathtt{E} \to \text{“value”}, 0) = \{\mathtt{Int}\}$ $cs(\mathtt{E} \to \text{“hours”}, \tau)(E_0) =$ $\quad \begin{cases} \mathit{true} & \text{if } \mathtt{Int} \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases}$	$[\![(\mathtt{T} \to \mathtt{E}, 1)]\!] = \{\mathtt{Int}\}$ ↑ $[\![(\mathtt{E} \to \mathtt{E}\ \text{“+”}\ \mathtt{E}, 1)]\!] = \{\mathtt{Int}\}$ ↗ ↘ $[\![(\mathtt{E} \to \text{“hours”}, \tau)]\!] = \mathit{true}$　　$[\![(\mathtt{E} \to \text{“value”}, \mathtt{0})]\!] = \{\mathtt{Int}\}$

图 3.7　示例类型约束方程及其具体属性

$cs(\mathtt{T} \to \mathtt{E}, 1) = 1$ $cs(\mathtt{E} \to \mathtt{E}\ \text{“+”}\ \mathtt{E}, 1)(E_0, E_2) =$ $\quad E_0 + E_2 + 1$ $cs(\mathtt{E} \to \text{“value”}, 0) = 1$ $cs(\mathtt{E} \to \text{“hours”}, \tau)(E_0) =$ $\quad \begin{cases} \mathit{true} & \text{if } E_0 + 1 \le 3 \\ \mathit{false} & \text{otherwise} \end{cases}$	$[\![(\mathtt{T} \to \mathtt{E}, 1)]\!] = \{0, 1, 2\}$ ↑ $[\![(\mathtt{E} \to \mathtt{E}\ \text{“+”}\ \mathtt{E}, 1)]\!] = \{0\}$ ↗ ↘ $[\![(\mathtt{E} \to \text{“hours”}, \tau)]\!] = \mathit{false}$　　$[\![(\mathtt{E} \to \text{“value”}, \mathtt{0})]\!] = \{1\}$

假设 AST 的最大大小为 3。

图 3.8　程序规模约束函数示例及其具体属性

下：

$$[\![v]\!] = cs(\phi(v))\left(\left([\![nb^p(v,j)]\!]\right)_{j=0}^{\phi(v)^i-1}, \left([\![nb^p(v,j)]\!]\right)_{j=\phi(v)^i+1}^{|\phi(v)|}\right).$$

为了检测一个完全扩展树是否满足一个约束，我们只需要检查对于该树的初始顶点 v_0，$[\![v_0]\!]$ 是否为真。例如，图3.6、图3.7和图3.8右列展示了四棵基于其约束函数的顶点属性的样例扩展树。

具体约束函数定义了完整程序上的规约的语义。为了判断部分程序是否可行，玲珑框架使用了一个基于抽象解释[155] 的用户定义的抽象域。抽象域用于进行两种计算：首先，针对扩展规则实施离线的静态分析为每个非终结符计算**抽象向上/向下属性** (abstract upward/downward attribute)。该抽象属性是覆盖所有具体的顶点属性的抽象值。其次，针对部分扩展树实施在线分析计算**抽象顶点属性**。该抽象属性覆盖所有可能的完整树产生的具体顶点属性。其中第二步分析依赖于第一步的分析结果。

用户定义的抽象域应该满足一些基本的需求。抽象解释框架通过伽罗瓦联结 (Galois connection) 表现这些需求。

定义 3.17 (伽罗瓦联结 (Galois Connection)). *给定具体域 D、有限的抽象域 A、A 上的偏序关系 $\sqsubseteq$、函数 $\alpha : 2^D \to A$ 和函数 $\gamma : A \to 2^D$，它们构成伽罗瓦联结 $(2^D, \subseteq) \leftrightarrows_{\alpha}^{\gamma} (A, \sqsubseteq)$ 当且仅当 $\forall d \in 2^D, a \in A:\ \alpha(d) \sqsubseteq a \Leftrightarrow d \subseteq \gamma(a)$。*

为了描述方便，假设存在伽罗瓦联结 $(2^D, \subseteq) \leftrightarrows_{\alpha}^{\gamma} (A, \sqsubseteq)$，并假设在抽象域 A 上存在与 $\sqsubseteq$ 一致的安全的并操作 $\sqcup$，有 $a_1 \sqsubseteq a_2 \Leftrightarrow a_1 \sqcup a_2 = a_2 \wedge \gamma(\alpha(a_1) \cup \alpha(a_2)) \sqsubseteq a_1 \sqcup a_2$。

抽象约束函数是具体约束函数的功能抽象，用于描述在抽象域上的计算。值得注意的是，对于创始规则，对于结果的抽象域是直接由用户提供：返回 `true` 表示具体约束函数可能返回 `true`，而返回 `false` 表明具体约束函数永不返回 `true`。

定义 3.18 (抽象约束函数 (Abstract Constraint Function)). 一个基于抽象域 A 的约束函数 acs 抽象一个基于 D 的约束函数 cs，当且仅当对于任意的扩展规则 $r = (g,i)$，$i \neq \tau$，$acs(r)$ 是单调的且是 $cs(r)$ 的安全的函数抽象，即有 $\alpha \circ cs(r)(\gamma(a_1),\ldots,\gamma(a_n)) \sqsubseteq acs(r)(a_1,\ldots,a_n)$，且对于任意的扩展规则 $r = (g,\tau)$ 有 $\exists d \subseteq \gamma(a) : cs(r)(d) \Rightarrow acs(r)(a)$。

例如，图3.9、图3.10和图3.11的左列展示了针对前述具体约束函数的抽象约束函数。对于“输入-输出”样例抽象域是区间。对于类型检查抽象域是类型集合。对于程序规模抽象域是程序大小的下界。

给定一个约束函数 cs，并针对其抽象得到可计算的约束函数 acs，我们可以计算前文所述两种抽象属性。

定义 3.19 (抽象向上/向下属性 (Abstract Upward/Downward Attributes)). 扩展规则集合 R 中的非终结符的抽象向上和向下属性集合是一个 A 中满足以下等式的最小值的集合。其中，对于非终结符 N，$\llparenthesis N \rrparenthesis$ 表示其抽象向上属性，$\llbracket N \rrbracket$ 表示其抽象向下属性。对于 $\exists g : (g,0) \in R \wedge g[0] = N$ 中的任意 N：

$$\llbracket N \rrbracket = \bigsqcup_{r=(g,0)\in R \wedge g[0]=N} acs\,(r)\left(\left(\llbracket g[j] \rrbracket\right)_{j=1}^{|g|}\right).$$

对于 $\exists g : (g,i) \in R \wedge g[i] = N \wedge i > 0$ 中的任意 N：

$$\llparenthesis N \rrparenthesis = \bigsqcup_{r=(g,i)\in R \wedge g[i]=N \wedge i>0} acs(r)\left(\llparenthesis g[0] \rrparenthesis, \left(\llbracket g[j] \rrbracket\right)_{j=1}^{i-1}, \left(\llbracket g[j] \rrbracket\right)_{j=i+1}^{|g|}\right).$$

因为抽象域是有限的，而且根据 acs 产生的函数是单调的，上述等式根据标准不动点 (fix-point) 算法求解[156]。

定义 3.20 (抽象节点属性 (Abstract Vertex Attribute)). 给定一个部分展开树 (V,p,R,ϕ)，对于节点 $v \in V$ 的抽象属性记作 $\|v\|$，定义如下：

$$\|v\| = acs(\phi(v))\left((attr(v,j))_{j=0}^{\phi(v)^i-1}, (attr(v,j))_{j=\phi(v)^i+1}^{|\phi(v)|}\right).$$

其中，

$$attr(v,j) = \begin{cases} \|nb^p(v,j)\| & if\ nb^p(v,j)\ exists \\ \llparenthesis \phi(v)[0] \rrparenthesis & elif\ j = 0 \\ \llbracket \phi(v)[j] \rrbracket & else \end{cases}$$

$$
\begin{array}{l}
acs(\mathtt{T} \to \mathtt{E}, 1) = \{\mathtt{true}\} \\
acs(\mathtt{E} \to \mathtt{E}\ \text{“}>12\text{”}, 1)(E_0) = \\
\quad \begin{cases} [-\infty, +\infty] & \text{if } \{\mathtt{true}, \mathtt{false}\} \sqsubseteq \mathtt{E}_{\emptyset} \\ [12, +\infty] & \text{elif } \{\mathtt{true}\} \sqsubseteq \mathtt{E}_{\emptyset} \\ [-\infty, 12] & \text{elif } \{\mathtt{false}\} \sqsubseteq \mathtt{E}_{\emptyset} \\ \emptyset & \text{else} \end{cases} \\
acs(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)(E_0) = \\
\quad \begin{cases} \mathit{true} & \text{if } [13, 13] \sqsubseteq E_0 \\ \mathit{false} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{c}
\|(\mathtt{T} \to \mathtt{E}, 1)\| = \{\mathit{true}\} \\
\uparrow \\
\|(\mathtt{E} \to \mathtt{E}\ \text{“}>12\text{”}, 1)\| = [12, +\infty] \\
\uparrow \\
\|(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)\| = \mathit{true}
\end{array}
$$

(a) 对于“输入-输出”样例 hours=13, ret=true 和程序 hours > 12。

$$
\begin{array}{l}
acs(\mathtt{T} \to \mathtt{E}, 1) = [10, 10] \\
acs(\mathtt{E} \to \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)(E_0, E_2) = \\
\quad (E_0 - E_2) \\
acs(\mathtt{E} \to \text{“}\mathtt{value}\text{”}, 0)(E_0) = [1, 1] \\
acs(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)(E_0) = \\
\quad \begin{cases} \mathit{true} & \text{if } [13, 13] \sqsubseteq E_0 \\ \mathit{false} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{c}
\|(\mathtt{T} \to \mathtt{E}, 1)\| = [10, 10] \\
\uparrow \\
\|(\mathtt{E} \to \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)\| = [-\infty, 9] \\
\nearrow \qquad \searrow \\
\|(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)\| = \mathit{false} \quad [\![\mathtt{E}]\!] = [1, +\infty]
\end{array}
$$

(b) 对于“输入-输出”样例 hours=13, value=1, ret=10 和部分程序 hours + E。

图 3.9　示例抽象语义约束方程及其抽象属性

$$
\begin{array}{l}
acs(\mathtt{T} \to \mathtt{E}, 1) = \{\mathtt{Boolean}\} \\
acs(\mathtt{E} \to \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)(E_0, E_2) = \\
\quad E_0 \cap E_2 \cap \{\mathtt{Int}, \mathtt{Float}\} \\
acs(\mathtt{E} \to \text{“}\mathtt{value}\text{”}, 0) = \{\mathtt{Int}\} \\
acs(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)(E_0) = \\
\quad \begin{cases} \mathit{true} & \text{if } \mathtt{Int} \in E_0 \\ \mathit{false} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{c}
\|(\mathtt{T} \to \mathtt{E}, 1)\| = \{\mathtt{Int}\} \\
\uparrow \\
\|(\mathtt{E} \to \mathtt{E}\ \text{“}+\text{”}\ \mathtt{E}, 1)\| = \{\mathtt{Int}\} \\
\nearrow \qquad \searrow \\
\|(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)\| \qquad [\![\mathtt{E}]\!] = \\
= \mathit{true} \qquad \{\mathtt{Int}, \mathtt{Float}, \mathtt{Boolean}\}
\end{array}
$$

图 3.10　示例抽象类型约束方程及其抽象属性

例如，图3.9、图3.10和图3.11的右列展示了抽象向上/向下属性和针对某些部分展开树计算的抽象节点属性。

当初始节点的抽象节点属性是 false 时我们剪除部分展开树，即该部分展开树无法产生可以满足规约的完整程序。接下来，证明这个剪枝过程是安全的。

定理 3.4. 给定一个部分的展开树 $\mathtt{prog} = (V, p, R, \phi)$，对于任意可由 prog 展开而得的完全展开树 $\mathtt{prog}' = (V', p', R, \phi')$，我们有 $[\![v_0]\!] \implies \|v_0\|$，其中 v_0 是初始顶点。

证明. 接下来本文以三个步骤证明该定理：(1) 证明当对任意的 $v \in V'$ 有 $i \neq 0 \implies [\![v]\!] \subseteq \gamma(\lfloor\!\lfloor N \rfloor\!\rfloor)$ 和 $i = 0 \implies [\![v]\!] \subseteq \gamma(\lceil\!\lceil N \rceil\!\rceil)$，其中 $i = \phi(v)^i$ 且 $N = \phi(v)^g[\phi(v)^i]$；(2) 对任意非初始顶点 $v \in V$，有 $[\![v]\!] \subseteq \gamma(\|v\|)$；(3) 根据具体属性的定义，任意具体属性可以通过应用函数 $cs(\phi(v))$ 来计算。令 k 为应用的最大数字，接下来以关于 k 的归纳法证明。

$acs(\mathtt{T} \to \mathtt{E}, 1) = [1, +\infty]$ $acs(\mathtt{E} \to \mathtt{E}\text{“+”}\,\mathtt{E}, 1)(E_0, E_2) =$ $E_0 + E_2 + [1, +\infty]$ $acs(\mathtt{E} \to \text{“}\mathtt{value}\text{”}, 0)(E_0) = [1, +\infty]$ $acs(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)(E_0) =$ $\begin{cases} true & \text{if } 3 \in (E_0 + [1, +\infty]) \\ false & \text{otherwise} \end{cases}$	$\lVert(\mathtt{T} \to \mathtt{E}, 1)\rVert = [0, 2]$ $\uparrow$ $\lVert(\mathtt{E} \to \mathtt{E}\text{“+”}\,\mathtt{E}, 1)\rVert = [0, 0]$ $\lVert(\mathtt{E} \to \text{“}\mathtt{hours}\text{”}, \tau)\rVert = false \quad \llbracket \mathtt{E} \rrbracket = [1, +\infty]$

图 3.11　示例抽象程序规模约束方程及其抽象属性

当 $k = 1$ 且 $i = 0$ 时，有：

$$\begin{aligned} \llbracket v \rrbracket &= cs(\phi(v))() && \text{具体顶点属性的定义} \\ &\subseteq \gamma(acs(\phi(v))()) && \text{抽象约束函数的安全性} \\ &\subseteq \gamma(\llbracket N \rrbracket) && \text{并 (union) 的安全性} \end{aligned}$$

类似地，可以有 $k = 1$ 且 $i \neq 0$。

假设该性质对所有的 $k < k'$ 成立，下面考虑 $k = k'$ 的情况。我们首先考虑 $i = 0$ 时的情形。令 $v_i = nb^{p'}(v, i)$ 且 $n = |\phi(v)|$，我们有

$$\begin{aligned} \llbracket v \rrbracket &= cs(\phi(v))(\llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket) && \text{具体顶点属性的定义} \\ &\subseteq \gamma\left(acs(\phi(v))(\alpha(\{\llbracket v_1 \rrbracket\}), \ldots, \alpha\{\llbracket v_n \rrbracket\})\right) && \text{抽象约束方程的安全性} \\ &\subseteq \gamma(acs(\phi(v))(\llbracket \phi(v)^g[1] \rrbracket, \ldots, \llbracket \phi(v)^g[n] \rrbracket)) && \text{归纳假设和 } \alpha、\gamma \text{ 与 } acs(\phi(v)) \text{ 的单调性} \\ &\subseteq \gamma(\llbracket N \rrbracket) && \text{并 (union) 的安全性} \end{aligned}$$

类似地可以证明 $i \neq 0$ 也成立。综上所述，步骤 (1) 得证。

(2) 用类似的方法证明步骤 (2)。我们实施关于 k 的归纳，其中 k 是在计算 $\lVert v \rVert$ 期间抽象顶点属性的个数。最开始，我们考虑 $k = 1$ 且 $i = 0$ 的情况。因为只有一个抽象顶点属性被计算了，可知 v 没有孩子，即 $nb^p(v, j)$ 对于任意的 $j > 0$ 不存在。因此，可知：

$$\begin{aligned} &\llbracket v \rrbracket = cs(\phi(v))(\llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket) && \text{具体顶点属性的定义} \\ &\subseteq \gamma\left(acs(\phi(v))(\alpha(\{\llbracket v_1 \rrbracket\}), \ldots, \alpha\{\llbracket v_n \rrbracket\})\right) && \text{抽象约束方程的安全性} \\ &\subseteq \gamma(acs(\phi(v))(\llbracket \phi(v)^g[1] \rrbracket, \ldots, \llbracket \phi(v)^g[n] \rrbracket)) && \text{(1) 的结果和 } \alpha、\gamma \text{ 与 } acs(\phi(v)) \text{ 的单调性} \\ &= \gamma(\lVert v \rVert) && \text{抽象顶点属性的定义} \end{aligned}$$

类似可以证明 $k = 1$ 且 $i \neq 0$ 的情况。

现在假设对所有的 $k < k'$ 成立。首先，考虑 $k = k'$ 且 $i = 0$ 的情况。我们证明 $\llbracket v \rrbracket \in \gamma(attr(v, j))$ 对于任意的 $1 \leq j \leq |\phi(v)^g|$ 成立：$attr(v, j)$ 只能为 $\lVert nb^p(v, j) \rVert$ 或 $\llbracket \phi(v)[j] \rrbracket$。根据归纳假设，第一个情况是成立的。而根据 (1) 的结果，第二个情况成

立。因此 $[\![v]\!] \in \gamma(attr(v,j))$ 成立。在这个性质的基础上，我们有：

$$
\begin{aligned}
&[\![v]\!] = cs(\phi(v))([\![v_1]\!],\ldots,[\![v_n]\!]) && \text{由定义可得}\\
&\subseteq \gamma\left(acs(\phi(v))(\alpha(\{[\![v_1]\!]\}),\ldots,\alpha\{[\![v_n]\!]\})\right) && \text{抽象约束函数的安全性}\\
&\subseteq \gamma(acs(\phi(v))(attr(v,1),\ldots,attr(v,n))) && \text{根据上面的讨论和 } \alpha、\gamma \text{ 与 } acs(\phi(v)) \text{ 的单调性}\\
&= \gamma(\|v\|) && \text{抽象顶点属性的定义}
\end{aligned}
$$

类似地，我们可以证明当 $k = k'$ 且 $i \neq 0$ 时的情况是成立的。

(3) 根据 (2) 的结果，最初的定理可以由抽象约束方程的定义获得。 □

3.7 小结

本章提出了一种解决程序估计问题理论：玲珑框架。玲珑框架包括定义程序空间的扩展规则，并提供了 AST 与扩展树相互转换的方法。通过扩展规则，可以控制程序生成的顺序。另外，玲珑框架将程序生成步骤转化为路径查找问题，并提供计算目标程序概率的方法。最后，玲珑框架提供了基于抽象解释的约束求解方法，用于尽早过滤不可能展开为符合规约的完整程序的部分程序。

第四章　基于测试的修复问题的归约

4.1　引言

正如1.2节中所述，补丁过拟合问题是缺陷自动修复当前所面临的主要挑战之一。在基于测试的修复方法中，由于测试是不完全规约，经常产生过拟合的补丁，即生成可以通过全部测试但是错误的补丁。已有研究表明，测试集的质量直接影响修复方法产生补丁的质量[13,24]。在实践中，测试集的质量经常难以得到保证，这使得补丁过拟合问题更为严重。

本质上，补丁过拟合问题的原因是在补丁空间中正确补丁过于稀疏，而能通过测试的错误补丁则多出数个数量级[13]。在缺陷修复技术中，一般称可以通过全部测试的补丁称为“貌似正确的补丁”(plausible patch)，而只将与程序员提供的补丁语义等价的补丁视为“正确补丁”(correct patch)①。在实践中，一个缺陷所对应的全部补丁内，可能有上千个“貌似正确的补丁”，然而只有一两个是正确的补丁。

在2.1节中，我们通过梳理缺陷自动修复方法的发展脉络，发现已有修复技术为了增强修复能力，往往使用“更大”和“更丰富”的补丁空间，即探索更多、表现力更丰富的补丁。但是过于复杂的补丁空间，会加剧程序过拟合问题，进一步降低“正确补丁”在“貌似正确补丁”中的密度。

解决补丁过拟合问题的核心挑战是如何在补丁空间中定位到正确的补丁。以程序估计问题的视角来看，缺陷自动修复的任务就是从补丁空间中搜索到概率最大的补丁。若能用误差较小的方式估计得到正确补丁的概率，则对正确地修复有极大的帮助。为了将基于测试的修复归约为程序估计问题，我们需要解决缺陷定位、补丁生成和补丁验证三个子问题。其中，在补丁生成阶段中生成高质量的补丁是缓解过拟合问题的关键。

在第三章中介绍了玲珑框架。玲珑框架为解决程序估计问题提供了一种理论上的解法。然而，玲珑框架作为一个一般性的框架，若要将其用于实际生成补丁，需要将框架中的子模块实例化，并针对修复场景优化。本章的目标是将玲珑框架实例化，针对 Java 语言生成条件表达式，即以条件表达式的周围代码为上下文，以 Java 文法规则、类型和程序规模等为规约，求解关于条件表达式的程序估计问题。为了实例化玲珑框架，本章依次探索以下 4 个子模块的不同配置选择：

- 扩展规则：探索不同的文法和展开顺序对效果的影响；

①术语 plausible patch 和 correct patch 最初由文献 [25] 提出，现已被缺陷自动修复的研究社区广泛采用。为保持与已有文献统一，本文将其直译为对应中文。

- 统计学习模型：探索不同学习算法对效果的影响；
- 搜索算法：不同的路径搜索算法对效果的影响；
- 抽象约束方程：约束方程对效果的影响。

由于各个模块有许多选项，难以理论上分析得到最优的配置项。为此，本章首先系统地探索若干有代表性的配置，并通过实验观察其表现。本文通过分析预测精度和时间消耗来判断配置的性能。

在配置中找到最优一组选项之后，在其基础上实现针对 Java 条件语句的缺陷自动修复方法 Hanabi 。条件语句缺陷是指出错位置发生在条件分支或者循环等语句内部的缺陷，例如分支语句 `if` 语句的条件表达式错误及其分支体错误、循环语句的条件表达式错误等等。在真实软件中，条件语句错误是常见的缺陷种类，例如，在大型开源数据集 Defects4J[45] 中，有超过 50% 以上的缺陷是条件语句缺陷[44]。因此，正确而高效地修复条件语句缺陷有着较高的实际意义。在已有技术中，SPR[46]、Nopol[43]、ACS[1] 等方法专门修复条件语句缺陷。

Hanabi 是基于测试的修复方法，即输入为缺陷程序及其单元测试集，其中测试集包含至少一个未通过的测试，返回可使全部测试通过的补丁或返回空。Hanabi 符合标准的基于测试的修复框架，包含缺陷定位、补丁生成和补丁验证三个阶段。Hanabi 使用基于频谱的缺陷定位，将有出错嫌疑的语句按其出错可能性排序。随后通过实例化玲珑框产生条件表达式，再根据所修复位置的特点生成对应模板的补丁。最后，使用测试集验证，即保留能够通过全部测试的补丁。

在实验验证中，在两个大型开源 Java 缺陷数据集 Defects4J 和 Bugs.jar 上，与已有的修复方法相比，Hanabi 在条件语句缺陷上实现了最高的修复精确度和召回率。而且，Hanabi 修复了 8 个其他方法没有成功修复的缺陷。实验结果说明 Hanabi 能够缓解补丁过拟合问题，并且说明了将基于测试的修复问题归约为程序估计问题是有效的。

本章的结构如下：首先在4.2节中分别介绍如何实例化玲珑框架的子模块以及不同的配置选择；其次在4.3节中介绍基于实例化的玲珑框架的修复工具 Hanabi ；再次在4.4节中探索不同玲珑框架不同实例化配置对性能的影响，并选择合适的配置用于 Hanabi ；之后在4.5节分析 Hanabi 的修复能力；最后在4.6节中小结。

4.2 玲珑框架的实例化

为了让玲珑框架实际地产生程序，我们需要对其实例化，即为其主要模块提供实现方案。本节主要介绍使用玲珑框架生成条件表达式的实例化工作，主要包括以下几个方面：

- 扩展规则模块的实例化，包括文法规则的设计并在其基础上设计扩展规则集合；

- 统计学习模块的实例化，包括准备训练数据以及学习算法的选择；
- 搜索算法的实例化；
- 约束函数的实例化。

4.2.1　扩展规则的实例化

扩展规则定义了玲珑框架的搜索空间，决定了非终结符的数量与非终结符展开时选择的数量，决定了后续的统计学习的特征提取与学习方法选择。扩展规则是影响玲珑框架实例的关键因素。

4.2.1.1　文法规则的设计

文法规则规定了所有可能生成程序的空间，也定义了程序分解的步骤。如3.3节所述，扩展规则是从文法规则泛化而来。为了针对条件表达式定义扩展规则，需要首先定义其文法规则。而由于条件表达式在不同的程序位置中，可访问的变量、方法等都是不同的，因此很难为所有位置定义一个唯一的语法规则集合。为此，本文对变量和方法等“无限的”字符串集合分别特殊处理。对于变量，由于不同程序位置的变量集合差异往往很大，因此对于不同位置的变量使用不同的集合。而对于方法，可以观察到不同程序位置的可访问的方法差距不是那么巨大。更进一步，可以观察到表达式经常是通过固定的模式组合在一起，例如，对于数组 `a`、`b` 和 `c`，`a.length > 0` 的使用频率远远大于 `a.length + b.length > c.length`。本文将布尔表达式中将变量用占位符替换而得的固定的组合称为**检验** (test)。为了生成足够精确简洁的文法，本文从训练集中挖掘所有的检验，并将其作为文法的一部分。因此，在本文的实例化中，文法是部分固定的。其中，操作符和检验是固定的，而变量部分是非固定的，是从当前上下文中提取的。

另一个需要考虑的设计是需要在更多非终结符和更多的选择之间做出权衡。更多的非终结符意味着抽象语法树更高，与此同时每一个节点的孩子节点的选择更少；而更多的选择则意味着其抽象语法树更矮，但每个节点的孩子节点选择更多。例如，如下的两个文法规则集合描述了相同的语言，但是规则集4.1有更多的非终结符，同时每个非终结符有更少的选择。

$$\mathtt{E} \rightarrow \mathtt{V} > \mathtt{L}, \quad \mathtt{V} \rightarrow \mathtt{a} \mid \mathtt{b}, \quad \mathtt{L} \rightarrow \mathtt{0} \mid \mathtt{1} \tag{4.1}$$

$$\mathtt{E} \rightarrow \mathtt{a} > \mathtt{0} \mid \mathtt{b} > \mathtt{0} \mid \mathtt{a} > \mathtt{1} \mid \mathtt{b} > \mathtt{1} \tag{4.2}$$

为了探索在玲珑框架的实例化中，究竟是更多的非终结符而更少的选择好，还是更少的非终结符更多的选择好，本章试图通过对照实验来求解该问题。本文设计了两

```
T  →  E
E  →  L                      | L && E
   |  L || E                 | !E
L  →  FastMath.abs(V) > V    | V.isEmpty()
   |  V > V                  | V == 0
   |  V != 0                 | V * V == 0 | ...
V  →  overflow               | a0
   | u | v                   | ...
```

图 4.1　更多的非终结符且包含递归的文法集合

```
T  →  E
E  →  FastMath.abs(V) > V    | V.isEmpty()
   |  V > V                  | V == 0
   |  V == 0 || V == 0       | V * V == 0 | ...
V  →  overflow               | a0
   | u | v                   | ...
```

图 4.2　非终结符有更多选择的文法集合

个对照的文法规则集合，如图4.1和图4.2所示。其中，图4.1拥有更多的非终结符。其中，训练集中的原子检验被提出到 L，而表达式 E 或者展开为原子的 L，或者展开为 L 通过逻辑操作符 (&&、|| 和 !) 与 E 连接，再递归地展开右侧的 E。另一方面，图4.2表示了一组更“平”的文法规则集，不包含需要递归展开的非终结符，而且逻辑操作符 (&&、|| 和 !) 被融合进检验，从训练集中直接提取，从而 E 的选择数量更多。尽管有些非终结符 (例如 L 和 V) 理论上有无穷多的选择，本文仅使用在训练集中出现过的符号。即从训练集中构建“词汇表”，未在其中出现过符号的视为“未知”。

4.2.1.2　扩展规则的设计

如前文所述，为了保证程序的概率计算是唯一的，我们需要使用正则的扩展规则集合，即保证其完全性和唯一性。正则的扩展规则集合可以使用算法3.1由文法规则集合导出。

对于扩展规则，首要需考虑的是从何处开始搜索，即在何处放置创始规则。为了通过对照试验求解该问题，本文在同一组文法规则的基础上，使用两种策略生成扩展规则集合。第一种策略是自顶向下，即在算法3.1的第 9 行中，永远在根顶点添加创始规则，使得扩展树保持向下扩展。第二种策略是自底向上，即在算法3.1的第 9 行中选择添加自底向上规则，并在第 11 行时选择 i 为 1，即选择最左的叶子顶点作为创始规则，并向上展开。对于图4.1中所示的包含递归的文法规则，选择自顶向下的策略，生

成如图4.3所示的扩展规则集合。在图4.2中所示的文法规则实行对照实验，同时使用自顶向下和自底向上两种策略，分别产生如图4.4和图4.5所示的扩展规则集合。

$$\begin{array}{llll} \langle T \to E,\ \tau\rangle & \langle E \to L \text{ “||” } E,\ 0\rangle & \langle L \to V \text{“ > ”} V,\ 0\rangle & \langle V \to \text{“}u\text{”},\ 0\rangle \\ & \langle E \to L,\ 0\rangle & \langle L \to V \text{“ == 0”},\ 0\rangle & \langle V \to \text{“}v\text{”},\ 0\rangle \\ & \cdots & \cdots & \cdots \end{array} \tag{4.3}$$

图 4.3　图4.1中带递归的文法集合的自顶向下扩展规则集

$$\begin{array}{llll} \langle T \to E,\ \tau\rangle & \langle E \to V \text{“ > ”} V,\ 0\rangle & \langle V \to \text{“}u\text{”},\ 0\rangle \\ & \langle E \to V \text{“ == 0 ||” } V \text{“ == 0”},\ 0\rangle & \langle V \to \text{“}v\text{”},\ 0\rangle \\ & \cdots & \cdots \end{array} \tag{4.4}$$

图 4.4　图4.2中不带递归的文法集合的自顶向下扩展规则集

$$\begin{array}{llll} \langle T \to E,\ 1\rangle & \langle E \to V \text{“ > ”} V,\ 1\rangle & \langle V \to \text{“}u\text{”},\ \tau\rangle \\ & \langle E \to V \text{“ == 0 ||” } V \text{“ == 0”},\ 1\rangle & \langle V \to \text{“}u\text{”},\ 0\rangle \\ & \cdots & \langle V \to \text{“}v\text{”},\ \tau\rangle \\ & & \langle V \to \text{“}v\text{”},\ 0\rangle \\ & & \cdots \end{array} \tag{4.5}$$

图 4.5　图4.2中不带递归的文法集合的自底向上扩展规则集

由于理论上扩展规则的连接顺序数量巨大，难以理论上分析得到最优解，因此本文仅选取上述三个有代表性的扩展规则集合进行经验性分析。图4.3中的扩展规则集合和图4.4中的扩展规则集合组成对照组，可以探索在相同的展开顺序策略下，不同的文法集合对性能的影响，即是非终结符层级多的文法规则集合较优，还是非终结符展开的选择多的文法规则集合较优。图4.4中的扩展规则集合和图4.5中的扩展规则集合组成对照组，可以探索在相同的文法规则集合的条件下，不同展开顺序策略对性能的影响。后文会对三组扩展规则的性能进行实验验证。

4.2.2　统计学习算法的实例化

为了实例化玲珑框架，对于每个非终结符和每个扩展方向策略 (即向上或向下) 需要训练统计学习模型来预测不同选择的概率。由于变量理论上是无穷集合，并且在不同程序位置变量集合差异较大，在玲珑框架的实例化中，对于由非终结符 `V` 向下展开的扩展规则使用二元分类，即应用该规则的概率。通过比较同一位置上不同变量的选择的概率，选择概率最高的规则，就是最应该被应用的规则。对于检验等其他的情形，则直接使用多分类模型进行预测。

为了训练模型，需要训练数据和选择学习算法。在基于机器学习的软件分析中，根据本项目和相关项目训练往往可以获得较好的效果，例如在缺陷预测技术中，项目内的预测效果远好于跨项目的预测[157,158]。究其原因，是软件之间功能、架构、代码风格和代码规范等等各不相同，同一项目内的数据训练的模型性能更好。因此，本文在实例化玲珑框架中，训练数据取自项目自身，即给定一个项目，将项目的条件表达式提取出来，将其中 90% 的条件表达式用于训练，其余 10% 用作效果验证。

而对于学习算法，玲珑框架并不限制，用户可以根据训练数据的特点进行选择。在本节的实例化中，提供了 XGBoost[159]、朴素贝叶斯以及 SVM 等学习算法的选择。如后文验证结果所示，XGBoost 在该任务中表现更好、相对时间消耗也不高。本文认为，其原因是基于决策树的学习模型对数据不均衡容忍程度较高，并且通过多组树形成的森林也可以降低过拟合问题。

4.2.2.1 代码中变量名等字符串的编码方法

在代码中，变量名、类名和方法名等字符串特征经常与其功能用途的自然语言描述相关或者有特定含义。在使用自然语言处理技术时，需要对字符串特征进行特殊处理。如果使用标签编码器 (label encoder)，则会造成标签集合过于庞大，也会丢失标识符之间相关联的信息。尤其是在变量名等标识符中，常见有缩写和多个单词连写的情况。例如，`length`、`len`、`xLen` 和 `arrLen` 等几个变量名都表示某种“长度”，而且拥有公共子字符串 `len`；而 `mutationRate`、`crosoverRate` 和 `elistimRate` 等几个变量名都由单词 `rate` 结尾，表示与某种“率”有关。

为了保留标识符之间重要的联系，本文提出将字符串通过字符 *bi-gram* 的布尔模型编码为向量的方法。若语言的字母表 (alphabet) 的字符数为 $\mathtt{n}$，则字符 bi-gram 的个数为 $\mathtt{n}\times\mathtt{n}$。构建长度为 $\mathtt{n}\times\mathtt{n}$ 的向量，与两个字符的排列方式一一对应。在某单词中，若某个 bi-gram 出现，则将对应的元素置为 1，否则设为 0。如图4.6所示，将变量 `len` 和 `xLen` 编码为两个 $26*26$ 的向量。可以看出，两个向量都置为 1 的元素存在一些重叠，因此二者关联性较高。因为该向量过长并且过于稀疏，本文使用 PCA[160] 对其进行降维，将向量压缩至不高于 20 维。

该编码方法能很好地捕捉到字符串相似程度，不单单适用于变量名等标识符，也适用于其他需要分析字符串相似性和相关性的特征。

4.2.2.2 特征设计

如前文所述，展开时的选择的条件概率为 $\mathtt{P(rule\mid context, prog, position)}$，即展开规则的选择是关于上下文、已生成程序和当前展开位置的条件概率。为了训练统计学习模型，我们需要对这四个方面设计特征。

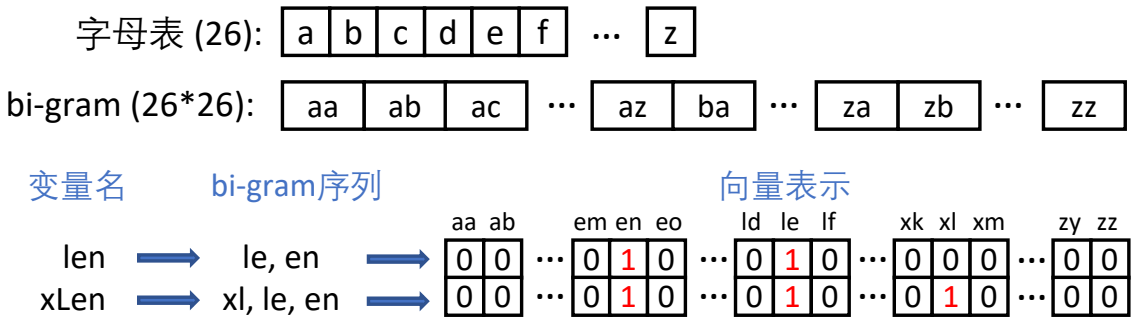

图 4.6　变量名的字符 bi-gram 编码示意图

上下文特征。上下文特征描述了周围代码的上下文信息，是最重要的一类特征。从空树展开选择创始规则时，唯一能凭借的就是上下文特征。而扩展的早期，能够正确预测出关键的扩展规则对于最终的完整结果至关重要。并且在后续的预测中，上下文特征都是重要程度较高的条件。

上下文特征包括了描述以下四个方面的特征：

- 代码位置：即需所生成的条件表达式在代码中的位置，文件名 (file name)、行数与列数等。
- 所在类型信息：代码位置所在的类 (class) 的信息，包括包名 (package name)、类名 (class name)、超类 (super class) 名、类的域 (field) 及其类型等。
- 所在方法信息：代码位置所在的方法 (method) 的信息，包括方法的返回类型、签名 (signature)、方法行数、参数及其类型等。
- 文法信息：代码位置的文法结构 (syntax structure)，包括可访问的局部变量 (local variable) 及其类型，代码位置周围的控制流结构 (例如 `if`、`for`、`return` 等控制流节点[158]) 等。

具体的上下文特征及其描述请见表 4.1。对于表 4.1 至 表 4.5 中的特征类型列，其中，数值表示可比较大小的浮点数特征值，布尔表示 True-False 二元特征值，标签表示不可比较大小的标签型特征值，字符串表示按照其字面值实施4.2.2.1小节中所述的字符串编码方法处理后得到其特征值向量。

已生成程序特征。已生成程序特征是从已有的部分程序中提取的特征。由4.2.1.2节中所描述的扩展规则，主要需要提取的项主要有三个，即 `L`、`E` 和 `V`。玲珑框架的实例化主要从两个方面考虑提取特征：(1) 程序的结构；(2) 项的内容信息。程序的结构相关的特征主要提取自程序的树形结构，即是否是创始规则、根节点类型、兄弟节点类型、父节点类型、距离树根的距离的等。项内容信息则主要根据项自身的特点提取特征。例如，`V` 代表变量，可以根据其变量名、变量类型、赋值的距离、后续的使用和 `JavaDoc` 中文档信息等提取相关特征；`E` 代表检验，可以提取其中的操作符 (operator)、被调用的

表 4.1　上下文的特征

特征名称	特征类型	特征描述
line	数值	所在行号
col	数值	所在列号
filename	字符串	所在文件名
tdname	字符串	所在类名
mtdname	字符串	所在方法名
mtdmod	数值	所在方法的修饰符 (`public`、`static`、`final` 等)
mtdln	数值	所在方法行数
locnum	数值	所在位置可访问本地变量个数
paranum	数值	所在位置的可访问参数个数
fldnum	数值	所在位置可访问的域个数
allloc	字符串	所有本地变量连接而得的字符串
allloctp	字符串	所有本地变量的类型连接而得的字符串
locintnm	数值	本地变量中的整型个数
locfltnm	数值	本地变量中的浮点型个数
locarrnm	数值	本地变量中的数组类型个数
allfld	字符串	所有域连接而得的字符串
allfldtp	字符串	所有域的类型连接而得的字符串
inloop	布尔	当前位置是否在循环体内
bodyctl	标签	当前语句块内的 `throw` 和 `return` 信息
befsyn	字符串	当前位置之前的控制流节点拼接而得的字符串
bdsyn	字符串	当前位置语句块之内控制流节点拼接而得的字符串
afsyn	字符串	当前位置语句块之后的控制流节点拼接而得的字符串
bes N	标签	当前位置之前第 N 个控制节点类型 ($0 \le N \le 5$)
bds N	标签	当前位置语句块之内第 N 个控制节点类型 ($0 \le N \le 3$)
afs N	标签	当前位置语句块之后第 N 个控制节点类型 ($0 \le N \le 3$)
lv N	标签	声明距离当前位置最近的第 N 个变量 ($0 \le N \le 3$)
pstmt N	字符串	当前之前位置最近的第 N 行代码的字面字符串 ($0 \le N \le 1$)
nstmt N	字符串	当前之后位置最近的第 N 行代码的字面字符串 ($0 \le N \le 1$)
befcd	字符串	当前位置之前最近的条件表达式
befpred	字符串	当前位置之前最近的条件表达式的检验

表 4.2　基于变量 V 的特征

特征名称	特征类型	特征描述
varname	字符串	变量名
vartype	字符串	变量类型
vnmlen	数值	变量名的字符个数
shortvn	布尔	是否是短而无意义的变量名 (如 a、b、p、q 等)
vnmwds	数值	变量名驼峰分词后的单词数
ltt N	标签	变量名中倒数第 N 个字母 ($0 \leq N \leq 2$)
wd N	标签	变量名中倒数第 N 个单词 ($0 \leq N \leq 2$)
isint	布尔	是否是整型
isflt	布尔	是否是布尔类型
isarr	布尔	是否是数组类型
iscoll	布尔	是否是 java.util.Collection 的子类
ispmtarr	布尔	是否是基本类型的数组
prmtandspl	布尔	是否既是基本类型又是短变量名
twdl	标签	驼峰分词后的最后一个单词
lastassign	标签	最后一次被赋值类型
assop	标签	最后一次被赋值时所使用的二元操作符
assmtd	标签	最后一次被赋值时所使用的方法
assnm	标签	最后一次被赋值时所使用的变量
assnum	数值	被赋值的次数
dis	数值	当前位置到最后一次赋值的行数
disl10	布尔	赋值是否离当前位置较近，即 $dis \leq 10$
disl20	布尔	赋值是否离当前位置不远，即 $dis \leq 20$
disg20	布尔	赋值是否离当前位置较远，即 $dis \geq 20$
preassnum	数值	当前位置之前被赋值的次数
isparam	布尔	是否是方法参数
isfld	布尔	是否是当前类的域
isfnl	布尔	是否被 final 修饰
isidxer	布尔	是否被用作数组下标
bodyuse	布尔	在当前位置的语句体是否被使用
casted	布尔	当前位置之前是否被强制类型转换
castedtp	标签	被强制转换的类型
outuse	布尔	是否在当前语句体之外被使用
incondnum	数值	在当前方法中出现在条件中的次数
filecondnum	数值	在当前文件中出现在条件中的次数
totcondnum	数值	在全项目中出现在条件中的次数
lastpre	字符串	使用该变量最近的检验
occpostime	数值	已有程序中 V 展开为该变量的次数
used	布尔	已有程序中 V 是否被展开为该变量
docexcp	标签	在方法 JavaDoc 中有该变量有关异常的描述
docop	标签	在方法 JavaDoc 中有该变量有关异常的操作符的描述
doczero	布尔	在方法 JavaDoc 中该变量与 0 比较
docone	布尔	在方法 JavaDoc 中该变量与 1 比较
docnon	布尔	在方法 JavaDoc 中该变量与 null 比较
docrange	布尔	在方法 JavaDoc 中有该变量的取值范围的描述
docincode	布尔	在方法 JavaDoc 的 <code> 标签中有无出现该变量

表 4.3　基于谓词 E 的特征

特征名称	特征类型	特征描述
pred	字符串	E 的字符串字面量
posnum	数值	待扩展的变量数
roottp	标签	E 的 AST 的各节点类型
ariop	标签	E 的 AST 中最高处的算数运算符
hight	数值	E 的 AST 树高
mtd	标签	E 的 AST 中最高处的方法调用
instcof	布尔	是否有 `instanceof`
num N	数值	E 第 N 个出现的自然数 ($0 \leq N \leq 1$)
hasnull	布尔	是否有 `null`

表 4.4　基于字面量 L 的特征

特征名称	特征类型	特征描述
parenttp	标签	当前 AST 的根节点类型
astsize	数值	当前 AST 的节点数
andno	数值	当前 AST 中与节点的数量
orno	数值	当前 AST 中或节点的数量
notno	数值	当前 AST 中非节点的数量

方法等相关特征。

与 V 相关的已有程序特征如表 4.2所示；与 E 有关的程序特征如表 4.3所示；与 L 有关的程序特征如表 4.4所示。

当前展开位置特征。当前展开位置特征描述了即将被展开的位置信息，例如它是双亲节点的第几个孩子、当前待展开位置是否是方法的参数等。其中，项 V 和项 L 的当前展开位置特征详见表 4.5。

表 4.5　当前展开位置的特征

特征名称	特征类型	特征描述
argused	布尔	当前待展开的 V 是否是函数参数
tpfit	布尔	当前待展开的 V 是否完全匹配当前位置的类型
isroot	布尔	L 是否是当前部分程序的 AST 根
siblingtp	标签	当前 L 的兄弟节点的类型
location	标签	当前 L 在双亲节点中的位置
depth	数值	当前 L 到 AST 树根的距离

训练数据的规范化。在真实代码中，实现同样的语义可以有多种写法，这将会带来分解程序时，产生很多不必要的选择。例如，在检验中浮点数 0 的写法可以有：0、.0、0.0、$0.0d$、$0.0f$、$.0d$、$.0f$、$1e-10$、$1E-10$、$1e-18$ 和 $1E-18$ 等多种形式。若简单地通过不同的字符串将检验分类，会引入很多不必要的选择。因此，在训练之前本文会对检

验的训练数据进行基于启发式规则的标准化 (normalization) 处理。对于数字形式，统一为单一格式，例如前文关于零的一系列写法统一为 0。对于有 `get/set` 方法的域，统一改成由其 `get/set` 方法读/写。对于整数上的比较运算，统一为同一种，例如 `a>=N` 统一替换为 `a>N-1`。

特征选择。由于上述四类特征是人工基于经验设计的，特征的重要程度和作用可能存在较大差异。在将字符类型的类型进行 bi-gram 编码并使用 PCA 处理之后，特征数量依旧较为庞大。因此，在训练之前使用方差分析进行特征选择 (feature selection)，保留前 50% 的特征。

4.2.3　搜索算法的实例化

玲珑框架将程序生成转化为有向图上的路径查找问题。玲珑框架不限定具体的路径查找算法，但是不同算法的表现也不相同。为了探索路径搜索算法对程序生成性能的影响，本文实现了两种算法：*精确算法* Dijkstra 算法以及*近似算法* Beam 搜索算法。Dijkstra 算法维护了图中所有访问过的顶点的列表，以及从起始顶点到某顶点的最高收益。在每一步，该算法将新的收益最高的顶点及其相邻节点添加进列表。持续迭代直至访问到一个表示完整程序的目标顶点。Beam 搜索算法类似，只是维护列表的过程中只保留收益最大的 K 个顶点。K 越大，Beam 搜索的结果就越接近精确算法的结果，所消耗的计算也越多。

玲珑框架也可以使用其他算法实例化，例如贪心搜索策略，即每步扩展时只选择当前概率最大的选项。在后文的验证中，本文将对 Dijkstra 算法和 Beam 搜索算法进行比较。

4.2.4　抽象约束方程的实例化

在本文的玲珑框架实例话中，实现了两种典型的约束方程：类型约束方程和程序规模约束方程。类型约束方程是根据 Java 的类型检测系统，过滤掉类型不匹配的部分程序，保证产生的程序是符合语法要求的。程序规模约束强制部分程序的 AST 树高在和孩子节点个数，保证程序不会太过复杂。在本文的实现中，AST 高度上限为 6，每个节点的孩子节点不超过 4 个。

4.3　Hanabi 缺陷修复过程

Hanabi 是在前文所述的玲珑框架实例化的基础上，用于修复条件语句错误的自动修复方法。Hanabi 的输入为一个有缺陷的程序及其测试集，其中测试集中至少包含一个未通过的测试；输出为能使全部测试通过的补丁。与传统的基于测试的自动修复一

致，Hanabi 主要包括缺陷定位、补丁生成和补丁验证三个阶段。其中，缺陷定位阶段和补丁验证阶段是基于测试的，即使用基于频谱的缺陷定位和使用测试集这个不完全规约验证补丁。而补丁生成阶段是 Hanabi 的核心，其依靠玲珑框架实例化生成条件表达式，继而得到候选补丁。Hanabi 尝试产生补丁，如果补丁可以使全部测试通过则返回该补丁，否则迭代处理剩余补丁，直至处理完所有候选补丁或者超时退出。Hanabi 生成补丁的方式是：首先，通过玲珑框架的实例化，生成一个条件表达式列表，并按照表达式的存在概率由高到低排序；其次，针对每个条件表达式，根据当前位置特点应用补丁生成模板生成补丁。在得到候选补丁集合之后，使用已有测试集过滤，并返回第一个能使全部测试通过的补丁。Hanabi 的流程图如图4.7所示。

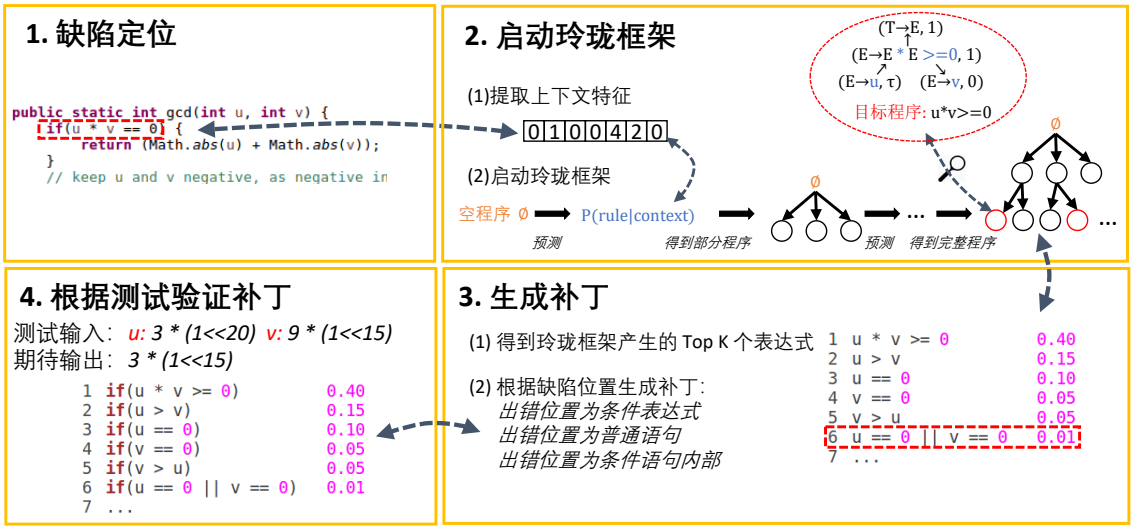

图 4.7　Hanabi 修复流程图

4.3.1　缺陷定位

缺陷定位提供一个可疑位置列表，并按照其出错可疑度从高到低排序。Hanabi 使用基于频谱的缺陷定位方法[161]，即统计程序位置在测试中的覆盖信息，通过公式计算出其错误可疑度。已有实证研究和理论研究表明，Ochiai 公式[162] 是基于频谱的定位方法中较为有效的[48,163,164]。另外，许多现有修复技术也在缺陷定位中使用 Ochiai 公式，例如 Nopol[43]、ACS[1]、ELIXIR[86]、CapGen[2]、SketchFix[65]、SimFix[33] 和 TBar[80] 等。

由于 Hanabi 是针对条件语句的修复工具，当 Ochiai 定位到 `if` 语句的条件表达式时，Hanabi 进一步应用谓词翻转 (predicate switching)[165]，判断该条件表达式是否是错误位置。谓词翻转是将条件表达式的真值反转，观察是否能使未通过测试通过。若能通过，则认为该语句的出错嫌疑最高。

4.3.2　条件语句补丁模板

在使用玲珑框架的实例化预测得到一组按出现概率排序的条件表达式之后，Hanabi 根据疑似出错位置的特点，根据相应的补丁模板生成补丁。Hanabi 的补丁模板与目前最新的条件语句修复方法 ACS[1] 一致，主要有*插入边界检查*和*条件表达式替换*两个模板。两个模板的描述如下：

- **模板 1：插入边界检查**

  $\Rightarrow$ `if (`c`) return` v

  $\Rightarrow$ `if (`c`) throw` e

  当失败的测试希望返回某个值或者抛出某个异常时，Hanabi 尝试在疑似出错位置之前插入 `if` 语句。该 `if` 语句体是直接返回测试希望的值或者是抛出对应的异常。其中，v 和 e 由失败测试提取，条件表达式 c 根据玲珑框架的实例化生成。该模板受到的启发是，程序员经常忘记捕捉某些特殊边界条件，并且该边界值已经体现在程序代码或测试中。
- **模板 2：修复缺陷条件表达式**

  `if (`c_o`)` $\Rightarrow$ `if (`c`)`

  Hanabi 使用玲珑框架实例化预测的表达式 c 替换当前的条件表达式 c_0。

当可疑语句是非条件语句表达式时，Hanabi 尝试应用模板 1，否则尝试模板 2。对于每个可疑位置，Hanabi 验证玲珑框架产生的条件表达式中的前 200 个。对于一个疑似出错的程序位置上补丁集合，Hanabi 的验证方式是首先运行失败测试，如果通过则运行剩余全部测试，否则迭代验证下一个补丁，直至所有候选补丁验证完毕。Hanabi 返回第一个能使全部测试通过的补丁。

4.3.3　补丁验证

Hanabi 在玲珑框架中采用的规约能够保证生成满足 Java 类型约束且程序规模不大的条件表达式。然而由于在大型软件中，很难根据测试提取针对补丁的语义规约。因此与基于测试的修复方法一致，Hanabi 使用测试集对候选补丁进行验证，即返回能通过全部测试的补丁。

为了能高效地验证补丁，Hanabi 采用了测试重排的策略。首先判断补丁能否通过原始程序未通过的测试，如果可以通过则判断能否通过原始未通过测试所在测试类的测试，如果可以通过则判断该补丁能否通过同一测试包内的测试。如果补丁可以顺利通过上述步骤，最终使用全部测试集对其验证。通过测试重排对补丁进行增量化地验证，能够降低补丁测试验证阶段的开销。

4.4 实验与结果分析

本节首先探索玲珑框架实例化中的各种选择的有效性，所采用的标准包括预测的准确程度和时间消耗。为了验证玲珑框架中不同配置如何影响最终的性能，本节针对其子模块系统地实施了一系列对照试验，主要探索了不同的文法规则集合、不同的扩展顺序、不同的统计学习算法以及不同的约束方程。在对比不同配置的性能之后，根据修复任务的特点，选择出一组配置应用到 Hanabi 上，并验证其实际的修复效果。

4.4.1 玲珑框架子模块配置的评估

4.4.1.1 实验设定

表 4.6 玲珑框架实例化的配置评估实验所用项目信息

项目名称	缺陷 ID	代码行数 (KLoc)*	测试代码行数 KLoc*	if 数	测试集个数
Apache Commons Math	106	9	12	635	64
Apache Commons Lang	65	16	28	1880	186
Joda Time	27	27	50	1879	187
JFree Chart	26	79	37	5261	524
合计	-	131	127	9655	961

KLoc 由统计代码行数的工具 *cloc* 采集。

为了评估不同的配置对玲珑框架性能的影响，本文从大型真实项目的开源 Java 缺陷数据集 Defects4J[45] 中选取了四个项目，分别是：Apache Commons Lang、Apache Commons Math、Joda Time 以及 JFreeChart。在 Defects4J 中一个项目包括按照缺陷 ID 编号的多个历史版本，我们选取的是其中最早的版本。所采用项目的统计如表 4.6所示，上述四个项目代码行数超过 13 万行，共有 9655 个 `if` 条件语句。将所有的 `if` 条件语句的条件表达式的特征提取出来，选取其中 90% (8694 个 `if` 语句) 作为训练集，10% (961 个 `if` 语句) 作为验证集。需注意的是，验证集的数据不出现在训练集中。对于每个项目的训练集，本文将原始的条件表达式的扩展树拆分为扩展规则选择，并提取特征。对于验证集，则使用玲珑框架生成预测表达式，检查正确表达式是否出现在 **Top K** 中。

对于 Hanabi 实验中所使用的 XGBoost 等学习算法的参数设置，详见附录 A。在提取的训练数据中，最大的规模大约是不足 10 万个实例，该数据量不足以支持深度神经网络的训练。因此本文没有使用深度神经网络模型。

4.4.1.2 研究问题

本文通过回答以下的研究问题确认玲珑框架配置的性能：

- **问题 1**：在图4.3、图4.4和图4.5中所示的三个扩展规则集合中，哪种是表现最好的？
- **问题 2**：对于统计学习算法，在 XGBoost 和朴素贝叶斯中哪一种更合适？
- **问题 3**：对于路径搜索算法，在精确的 Dijkstra 算法和近似算法 Beam 查找算法中哪一个更合适？
- **问题 4**：约束方程是否能提高预测性能？

为了回答上述四个研究问题，本文通过一系列对照试验比较性能，如表 4.7所示。表 4.7 中共有 12 个配置，分为四组 (分别为序号 1-3、4-6、7-9 和 10-12 的配置)。每组都包括了三个扩展规则集合 TD、BU 和 RECUR，在其他配置都一样的情况下，可以观察扩展规则集合对性能的影响。通过第一组 (配置 1-3) 和第二组 (配置 4-6) 的对比，可以观察学习模型对性能的影响。通过第一组 (配置 1-3) 和第三组 (配置 7-9) 的对比，可以观察路径搜索算法对性能的影响。通过第一组 (配置 1-3) 和第四组 (配置 10-12) 的对比，可以观察约束方程对性能的影响。

实验比较的性能指标包括预测精确度和预测时间。其中，预测精确度是指成功预测出正确条件表达式的数量，精确度越高意味着系统的预测能力更强；预测时间是指完成预测的运行时间，在实践中时间消耗越少的系统实用价值越高。

表 4.7　玲珑框架实例化中探索的配置项

序号	扩展规则集合 *	统计学习模型	路径查找算法	约束方程
1	TD	XGBoost	Beam 搜索	类型约束
2	BU	XGBoost	Beam 搜索	类型约束
3	RECUR	XGBoost	Beam 搜索	类型约束
4	TD	朴素贝叶斯	Beam 搜索	类型约束
5	BU	朴素贝叶斯	Beam 搜索	类型约束
6	RECUR	朴素贝叶斯	Beam 搜索	类型约束
7	TD	XGBoost	Dijkstra 算法	类型约束
8	BU	XGBoost	Dijkstra 算法	类型约束
9	RECUR	XGBoost	Dijkstra 算法	类型约束
10	TD	XGBoost	Beam 搜索	无
11	BU	XGBoost	Beam 搜索	无
12	RECUR	XGBoost	Beam 搜索	无

图4.3所示扩展规则带有递归展开，因此记为 RECUR (**RECUR**sive)；图4.4所示规则扩展顺序自顶向下，记为 TD (**T**op-**D**own)；图4.5所示规则扩展顺序自底向上，记为 BU (**B**ottom-**U**p)。所有配置分为四组，每组有相同的统计学习模型、搜索算法和约束方程，并包含三个 TD、BU 和 RECUR 的配置。

接下来介绍本次实验中子模块的参数设定。Beam 算法的参数是每次扩展保留收益最高的前 800 个选项，在下一次扩展时，从前 400 个选项展开。Dijkstra 算法的参数是每次扩展保留全部选项列表，直至生成 400 个完整程序。在 XGBoost 模型中选用梯度

提升树算法，而在朴素贝叶斯中选取高斯贝叶斯算法。类型约束根据 Java 文法自身的类型函数产生。

4.4.1.3 研究问题的回答

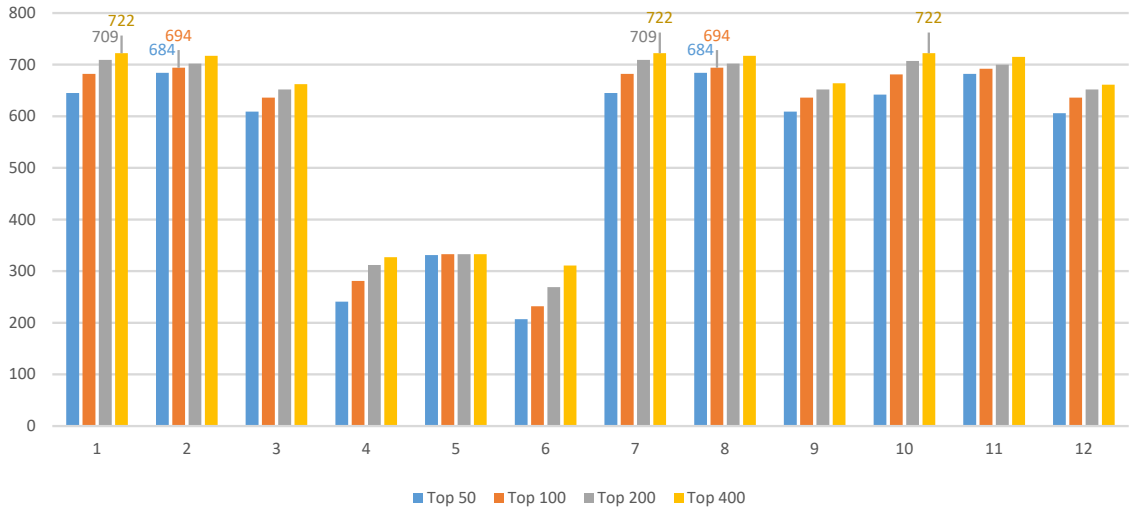

图 4.8 不同配置的成功预测数目

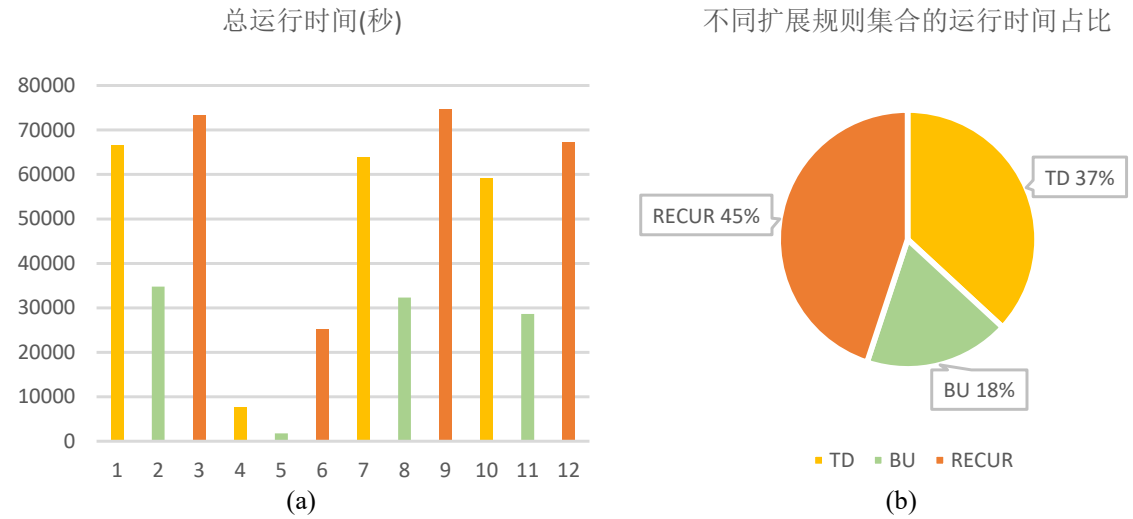

图 4.9 不同配置的时间消耗

实验结果整体概览。针对表 4.7中的 12 个配置，实例化玲珑框架之后在表 4.6中项目的验证集上进行预测，结果如图4.8所示。其中，*Top K* 表示正确的条件表达式排名出现在结果列表的前 K 个 (图中所示 $K \in \{50, 100, 200, 400\}$)。其中，在相同的 *Top K* 之间，表现最优的配置标注其预测成功的数目。由图可知，预测 *Top 50* 以内 (所有蓝色柱) 以及预测 *Top 100* 以内 (所有红色柱) 的最多的是都是配置 *2* 和配置 *8*；预测 *Top 200* 以内

表 4.8　表 4.7中不同配置的详细实验结果

		C26	L65	M106	T27	总计			C26	L65	M106	T27	总计
编号	指标	524	186	64	187	**961**	编号	指标	524	186	64	187	**961**
1	**Top 5**	229	99	18	82	428	**7**	**Top 5**	229	99	18	82	428
	Top 10	292	114	24	96	526		**Top 10**	292	114	24	96	526
	Top 50	356	138	34	117	645		**Top 50**	356	138	34	117	645
	Top 100	374	142	38	128	682		**Top 100**	374	142	38	128	682
	Top 200	383	148	41	137	709		**Top 200**	383	148	41	137	709
	Top 400	387	154	42	139	**722**		**Top 400**	387	154	42	139	**722**
	时间 (秒)	36765	11380	4011	14339	66495		时间 (秒)	35788	10891	3914	13278	63871
2	**Top 5**	325	122	28	101	576	**8**	**Top 5**	325	122	28	101	576
	Top 10	351	129	33	109	622		**Top 10**	351	129	33	109	622
	Top 50	379	139	40	126	684		**Top 50**	379	139	40	126	684
	Top 100	381	143	40	130	694		**Top 100**	381	143	40	130	694
	Top 200	381	146	41	134	702		**Top 200**	381	146	41	134	702
	Top 400	383	153	45	136	717		**Top 400**	383	153	45	136	717
	时间 (秒)	17391	6608	2890	7873	34762		时间 (秒)	16475	6365	2558	6923	32321
3	**Top 5**	234	96	20	82	432	**9**	**Top 5**	234	96	20	82	432
	Top 10	278	109	25	92	504		**Top 10**	278	109	25	92	504
	Top 50	337	122	34	116	609		**Top 50**	337	122	34	116	609
	Top 100	350	127	38	121	636		**Top 100**	350	127	38	121	636
	Top 200	359	131	38	124	652		**Top 200**	359	131	38	124	652
	Top 400	361	136	38	127	662		**Top 400**	362	137	38	127	664
	时间 (秒)	42263	12995	4337	13682	73277		时间 (秒)	42099	14394	4609	13635	74737
4	**Top 5**	36	31	7	29	103	**10**	**Top 5**	227	99	18	82	426
	Top 10	61	36	8	38	143		**Top 10**	289	114	24	95	522
	Top 50	120	52	14	55	241		**Top 50**	355	138	33	116	642
	Top 100	148	53	16	64	281		**Top 100**	373	142	38	128	681
	Top 200	166	60	18	68	312		**Top 200**	381	148	41	137	707
	Top 400	179	61	18	69	327		**Top 400**	387	154	42	139	**722**
	时间 (秒)	3574	1002	193	3010	7779		时间 (秒)	32189	10952	3530	12516	59187
5	**Top 5**	132	42	9	51	234	**11**	**Top 5**	323	122	27	101	573
	Top 10	144	51	10	52	257		**Top 10**	350	129	30	107	616
	Top 50	192	59	11	69	331		**Top 50**	378	139	39	126	682
	Top 100	192	61	11	69	333		**Top 100**	380	142	40	130	692
	Top 200	192	61	11	69	333		**Top 200**	381	145	41	133	700
	Top 400	192	61	11	69	**333**		**Top 400**	383	153	44	135	715
	时间 (秒)	1009	272	57	426	1764		时间 (秒)	13050	6295	2453	6825	28623
6	**Top 5**	42	13	3	27	85	**12**	**Top 5**	232	96	20	82	430
	Top 10	62	25	5	34	126		**Top 10**	276	109	25	92	502
	Top 50	107	38	12	50	207		**Top 50**	336	122	33	115	606
	Top 100	120	46	13	53	232		**Top 100**	350	127	38	121	636
	Top 200	143	52	15	59	269		**Top 200**	359	131	38	124	652
	Top 400	175	57	18	61	311		**Top 400**	361	136	38	126	661
	时间 (秒)	16223	3732	800	4420	25175		时间 (秒)	38598	12914	3820	11905	67237

C26: Chart-26，**L65**：Lang-65，**M106**：Math-106，**T27**：Time-27。

(所有灰色柱) 的最多是配置 *1* 和配置 *7*；预测 *Top 400* 以内 (所有黄色柱) 的最多是配置 *1*、配置 *7* 和配置 *10*，可以成功预测 961 个条件中的 722 个，预测率在 75% 以上。

图4.9 (a) 展示了 表 4.7中的 12 个配置预测总共的时间消耗。其中使用 *TD* 扩展规则的标记为黄色，使用 *BU* 扩展规则的标记为绿色，使用 *RECUR* 扩展规则的标记为红色。可以看出，在其他配置都一样的情况下，不同扩展规则集合对时间有重大影响，4 组配置中，均是 *BU* 最快，*TD* 其次而 *RECUR* 最慢。图4.9 (b) 展示三个扩展规则集合所消耗时间占总时间的比重，可知 *BU* 的时间消耗约是 *TD* 的 50%，*TD* 的时间消耗约是 *RECUR* 的 82%。

详细的实验预测结果和分析请见表 4.8。其中展示了每个配置的正确的条件表达式在前 5(Top 5)、前 10(Top 10)、前 50(Top 50)、、前 100(Top 100)、前 200(Top 200) 和前 400(Top 400) 的预测数量，以及其预测时间。在前 400 预测最多结果数的被标红 (722)。前 100 和前 400 的背景色分别涂成灰色和淡黄色以方便检索。

问题 1：扩展规则的表现。在预测精确度方面，根据图4.8和表 4.8，取 *Top 100* 及以内时，*BU* 的预测精确度最高；取 *Top 200* 时，*TD* 略优于 *BU*；取 *Top 400* 时，*TD* 的精确度最高。另外，由图4.9所示，*BU* 执行速度最快。不论在哪组配置中，*RECUR* 是预测效果最差且执行最慢的。*BU* 可以在扩展的早期将所需探索的空间大幅缩小，预测次数明显低于其余二者，因此 *BU* 的速度最快。因此，可以根据任务需求选择不同的扩展规则集合。当需要预测较快且结果列表较少时 (例如只生成 100 个)，可以使用 *BU*。当不计时间成本，需要尽可能探索较多结果时，可以使用 *TD*。

问题 2：统计学习算法的表现。表 4.7中第一组 (配置 1-3) 和第二组 (配置 4-6) 只有机器学习模型不同。根据这两组的对比，可以发现朴素贝叶斯的运行时间较快，用时仅为 XGBoost 的 45%；但是成功预测数远远低于 XGBoost，在 *Top 400* 以内仅为 XGBoost 的 46%。朴素贝叶斯作为简单的模型，不适合当前的任务，在目前这种需要较高的预测精确度的场景，XGBoost 更加适合。

问题 3：路径搜索算法的表现。表 4.7中第一组 (配置 1-3) 和第三组 (配置 7-9) 只有路径搜索算法不同。第一组使用近似算法 Beam 搜索，而第三组使用精确的 Dijkstra 算法。在 Beam 算法的参数设置为保留当前最大的 800 个选项时，可以发现二者性能较为相近，Dijkstra 仅微弱领先。

问题 4：约束方程的表现。表 4.7中第一组 (配置 1-3) 和第四组 (配置 9-12) 差异仅在是否应用类型约束方程。参照表 4.8的结果，引入类型约束方程会略微提高预测成功的数量，但是由于引入了更多的计算，也会使稍微降低运行速度。

4.4.2　Hanabi 的修复效果评估

4.4.2.1　实验设定

表 4.9　缺陷自动修复实验数据集的统计信息

项目名	缺陷数	数据集	代码 KLoc	测试 KLoc	测试数	开发年数	项目简介
Apache Commons Math	106	Defects4J	84	86	497	11	科学计算库
Apache Commons Lang	65	Defects4J	22	38	128	12	API 辅助库
Joda Time	27	Defects4J	28	53	155	11	时间处理库
JFree Chart	26	Defects4J	96	49	389	7	图表制作库
Apache Accumulo	17	Bugs.jar	154	20	147	6	分布式工具
Apache Camel	28	Bugs.jar	116	130	2,277	6	网络引擎
总计	269	-	500	376	3,593	-	-

为了验证 Hanabi 的修复效果，本文使用了 Defects4J 和 Bugs.jar 两个大型开源真实 Java 缺陷基准程序。根据已有的缺陷自动修复工作[1,2,33,34,38,58,59,64,66,68,80,86]，本文采用 Defects4J 的 1.0.0 版本。在 Defects4J 中选取了 4 个项目，即 Apache Commons Math、Apache Commons Lang、Joda Time 以及 JFree Chart。因为 Closure 无法支持缺陷定位工具 GZoltar[166]，本文没有采用该项目。类似地，有些现有工作[1,2,38,58,64,68,86] 也采取相同的策略。由于 Defects4J 中的四个项目都是库 (library) 项目，可能会因为项目类型对实验结果带来偏差。而且，Durieux 等人指出当前修复方法存在“过拟合”在 Defects4J 数据集上的现象[167]。因此，本文在另一个大型数据集 Bugs.jar 中选取了两个非库的项目：Apache Camel 和 Apache Accumulo。与 ELIXIR[86] 相同，对于 Bugs.jar 中的项目，只保留出错位置只有一处的缺陷。最终如表 4.9所示，本文从六个项目中选取了 269 个缺陷作为 Hanabi 的修复验证数据集。

本文使用 Hanabi 修复上述的六个项目的全部 269 个缺陷。为每个缺陷设定了 180 分钟的时间限制，超时记作修复失败。Hanabi 遇到第一个能使全部测试通过的补丁后返回该补丁并停止运行。

若 Hanabi 返回了能通过全部测试的补丁，则认为其是“疑似正确的”补丁，并人工验证是否与开发者提供的补丁语义等价，如果等价则报告为正确补丁。Hanabi 所有的“疑似正确”补丁都由两个程序员进行检查。通过人工检查补丁正确性也与已有工作一致[1,2,33,38,58,59,68,86]。为了避免人工检查引入的偏差，按照 Le 等人对于衡量缺陷自动修复所产生补丁正确性的建议[168]，本文进一步使用 EVOSUITE[169] 在程序员的修复版本上产生补充的测试集。如果 Hanabi 提供的“疑似正确”补丁不能通过新的测试集，则记为失败的修复。

在评估修复结果时，主要使用两个指标：**精确率** (precision) 和**召回率** (recall)。其

表 4.10　Defects4J 用于训练的缺陷及其修复的范围

训练	修复范围	年份	训练	修复范围	年份
M12	M1-M12	2013	L5	L1-L5	2013
M37	M13-M37	2012	L14	L6-L14	2012
M59	M38-M59	2011	L24	L15-L24	2011
M75	M60-M57	2010	L35	L25-L35	2010
M94	M76-M94	2009	L43	L36-L43	2009
M102	M95-M102	2008	L48	L44-L48	2008
M104	M103-M104	2007	L55	L49-L55	2007
M106	M105-M106	2006	L65	L56-L65	2006
T11	T1-T11	2013	C4	C1-C4	2009
T17	T11-T17	2012	C16	C5-C16	2008
T24	T18-T24	2011	C26	C17-C26	2007
T27	T25-T27	2010			

M：Math，L：Lang，T：Time，C，Chart。

中，精确率是指所有被“疑似正确”补丁修复的缺陷中被正确修复的缺陷所占百分比；召回率是指在全部缺陷中被正确修复的缺陷所占百分比。

缺陷自动修复往往需要探索很多疑似出错的位置，因此对于补丁生成的时间和精确程度都有较高要求。在上一小节中，我们系统地探索和比较了不同配置对玲珑框架性能的影响，发现 *BU* 扩展规则时间消耗较低，同时在 *Top 200* 以内成功预测数量与 *TD* 相差不大。因此，Hanabi 采用表 4.7中的配置 *2* 来实例化玲珑框架，即使用图4.5所示的扩展规则集合，XGBoost 作为学习算法，Beam 搜索为路径查找，类型约束作为约束方程。

Hanabi 使用缺陷程序的全部代码作为玲珑框架实例的训练集。由于同一个程序的相邻缺陷版本之间改动不多，为全部缺陷单独训练模型会带来不必要的开销。Hanabi 采取的策略是对于每个项目，按其开发年份划分，使用每个自然年的第一个缺陷提取训练数据，并用于修复当年的全部缺陷。由于 Math 和 Lang 等项目早期规模很小，为此 Hanabi 为其添加 JDK 7 中 `java.lang` 等相关包的源码作为补充。在数据集 Defects4J 中，缺陷编号越大的缺陷其版本年代越早。因此，用于训练的缺陷和应用的修复范围如表 4.10所示。由于 Bugs.jar 中按照项目的缺陷报告的问题 (issue)ID 编号而非按自然数编码，故不在此详细列出。

4.4.2.2　研究问题

本文通过回答以下研究问题来验证 Hanabi 修复缺陷的能力：

- **问题 4**：Hanabi 在真实缺陷上的修复表现如何？
- **问题 5**：与已有修复方法相比，Hanabi 的效果如何？
- **问题 6**: Hanabi 修复模板各自的贡献如何？

- **问题 7**：Hanabi 能否缓解程序修复的过拟合问题？

4.4.2.3　研究问题的回答

问题 4：Hanabi 的修复效果。 Hanabi 的整体修复效果如表 4.11所示。Hanabi 一共产生 42 个“疑似正确”修复。经过人工验证，其中 32 个是正确的修复，10 个是可以通过全部测试的错误修复。整体上，Hanabi 的精确率为 72%(32/42)，召回率为 12%(32/269)。在来自 Defects4J 数据集的缺陷上，Hanabi 精确率为 75.7%(28/37)，召回率为 12.5%(28/224)。在来自 Bugs.jar 数据集的缺陷上，Hanabi 精确率为 80%(4/5)，召回率为 9%(4/45)。可以看出，Hanabi 在两个数据集上的表现虽然略有不同，但是相差不到 5%，仍在合理的范围内。这表明 Hanabi 并没有过拟合在某一数据集上。

表 4.11　Hanabi 的整体修复效果

项目名称	缺陷个数	正确修复数	错误修复数	精确率	召回率
Math	106	19	6	76%	18%
Lang	65	4	0	100%	6%
Time	27	2	0	100%	7%
Chart	26	3	3	50%	12%
Accumulo	17	1	0	100%	6%
Camel	28	3	1	75%	11%
总计	269	32	10	76%	12%

精确率 = 正确修复数/(正确修复数 + 错误修复数)×100%，**召回率** = 正确修复数/缺陷个数 ×100%。

问题 5：Hanabi 与现有方法的比较。根据针对问题 4 的回答，本文与现有的缺陷修复技术进行比较，包括：SimFix[33]、CapGen[2]、SketchFix[65]、ELIXIR[86]、JAID[38]、ssFix[68]、ACS[1] 和 Nopol[43,58]。其中，ACS 和 Nopol 与 Hanabi 一样，都是专门针对条件表达式缺陷修复的方法。其余方法则是不限定修复的类型。其中一些方法 (如 JAID[38]) 会产生多个满足规约的补丁，在本次比较中仅取第一个可通过全部测试的补丁。上述方法与表 4.9中所示的缺陷仅有 Defects4J 部分是重合的，因此只比较上述方法在 Defects4J 的 4 个项目上总计 224 个缺陷上的修复效果。

本文试图从两个方面比较。首先，由于 Hanabi 是针对条件语句缺陷的修复方法，本文取出其中所有的 143 个条件语句缺陷作为子集，比较 Hanabi 和现有技术针对条件语句的修复效果。其次，在全部的 224 个缺陷中，比较 Hanabi 和其余方法的整体修复效果，观察在非条件语句错误上，Hanabi 是否会引入其他的错误修复。上述方法的修复数据直接取自原始其原始论文报告的数据。条件语句缺陷的分类参照 Sobreira 等人的已有工作[44]①。另外，该文章指出 Defects4J 中 64% 的缺陷都与条件语句有关，可见

①参见网站 http://program-repair.org/defects4j-dissection

条件语句缺陷是广泛存在的[①]。

表 4.12　在 Defects4J 的 143 个条件语句缺陷上的修复效果比较

项目名称	总计	Hanabi	ACS	Nopol	SimFix	SketchFix	CapGen	ELIXIR	JAID	ssFix
Chart	16	3 (1)	2 (0)	1 (3)	1 (3)	2 (1)	1 (0)	2 (1)	1 (2)	1 (2)
Math	61	15 (4)	8 (4)	1 (11)	3 (9)	4 (1)	4 (2)	4 (3)	0 (5)	3 (10)
Lang	46	2 (0)	1 (1)	3 (1)	7 (4)	1 (0)	0 (0)	1 (3)	1 (3)	1 (6)
Time	19	2 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (2)
总计	143	22 (5)	12 (5)	5 (15)	11 (16)	7 (2)	5 (2)	8 (7)	2 (10)	5 (20)
精确率	-	**81%**	71%	25%	41%	78%	71%	53%	17%	20%
召回率	-	**15%**	8%	3%	8%	5%	3%	6%	1%	3%

加粗数字是表现最佳的方法。**X (Y)**：X 是正确修复数，Y 是错误的“疑似正确”的修复数。

表 4.13　在 Defects4J 的全部 224 上的修复效果比较

项目名	总计	Hanabi	ACS	Nopol	SimFix	SketchFix	CapGen	ELIXIR	JAID	ssFix
Chart	26	3 (3)	2 (0)	1 (5)	4 (4)	6 (2)	4 (0)	4 (3)	2 (2)	3 (4)
Math	106	19 (6)	12 (4)	1 (20)	14 (12)	7 (1)	12 (4)	12 (7)	1 (7)	10 (16)
Lang	65	4 (0)	2 (2)	3 (4)	9 (3)	3 (1)	5 (0)	8 (4)	1 (7)	5 (7)
Time	27	2 (0)	1 (0)	0 (1)	1 (0)	0 (1)	0 (0)	2 (1)	0 (0)	0 (4)
总计	224	28 (9)	17 (6)	5 (30)	28 (19)	16 (5)	21 (4)	26 (15)	4 (16)	18 (31)
精确率	-	76%	74%	14%	60%	76%	**84%**	63%	20%	37%
召回率	-	**13%**	8%	2%	**13%**	7%	9%	12%	2%	8%

加粗数字是表现最佳的方法。**X (Y)**：X 是正确修复数，Y 是错误的“疑似正确”的修复数。

表 4.12 展示了所涉及的修复方法在条件语句缺陷上的修复效果。可以看出，Hanabi 在 143 个缺陷中成功修复了 22 个，取得了 15% 的召回率。在 27 个“疑似正确”的修复中只有 5 个是错误修复，取得了 81% 的精确率。与现有方法相比，Hanabi 在召回率和精确率上都实现了最高。在与最新的条件语句修复工作 ACS 相比，Hanabi 在不多产生错误修复的情况下，成功多修复了 10 个缺陷。在修复条件语句缺陷上，Hanabi 表现超越了现有方法。

表 4.13 展示了所涉及的方法在 Defects4J 的全部 224 个缺陷上的修复效果。可以看出，即便是在引入 Hanabi 处理范围之外的缺陷，Hanabi 依然可以实现最高的召回率。Hanabi 和 SimFix 都可以产生 28 个正确修复，但是 Hanabi 产生的错误修复更少。同时，Hanabi 实现了第二高的精确率，仅比 CapGen 低 8%。

值得注意的是有时一个非条件语句缺陷可以使用语义等价的条件语句进行修复。例如在缺陷 Math-35 中，开发者提供的补丁是调用了一个项目中已有的带边界检查的方法，而 Hanabi 产生的语义等价补丁是直接在调用处插入 `if` 边界检查。因此，在表 4.13中，Hanabi 和 ACS 产生的正确修复数量都多于表 4.12中的结果。

为了检查每个修复方法能修复多少其他方法不能修复的缺陷，本文通过文氏图展

[①]文章 [44] 分析的 Defects4J 版本为 1.0。

示相关方法的正确修复个数，如图4.10所示。图4.10 将在整体修复中实现了最高召回率的 SimFix 方法和最新的条件语句缺陷修复方法 ACS 单独列出。可以看出，在 Defects4J 上，Hanabi 能够成功修复 8 个其余方法无法修复的缺陷。这说明 Hanabi 是已有方法的良好补充。

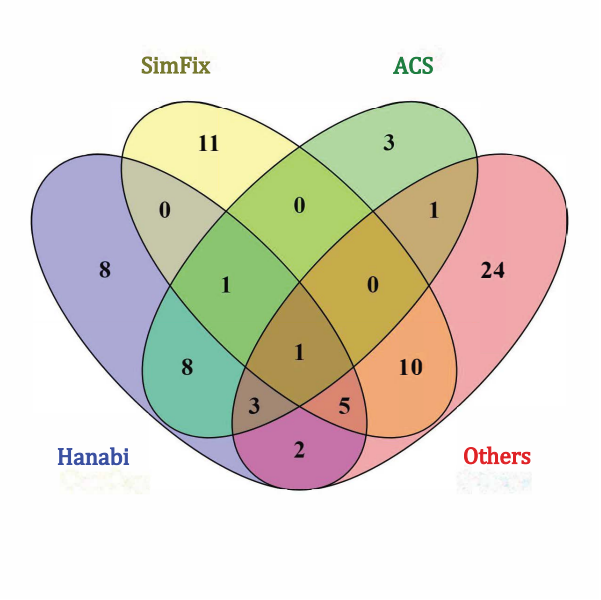

图 4.10　已有方法正确修复的重叠情况

问题 6：修复模板的贡献。表 4.14 展示了所有“疑似正确”的修复中模板 1 和模板 2 所占的比例。可以看到，模板 1 和模板 2 各占比 50%。而正确的修复中，模板 1 产生了 20 个修复，占 62.5%，模板 2 修复了 12 个修复，占 37.5%。在错误的修复中，模板 1 仅有 1 个，而模板 2 产生了 9。模板 1 的正确修复多于模板 2，而错误的修复远小于模板 2。这说明补足遗漏的边界检查修复要比替换已有的条件表达式相对容易，而且替换条件表达式更容易出错。这个结论与 ACS 的实验结果和作者分析是一致的。在 ACS 中，模板 1 正确修复了 15 个缺陷，模板 2 仅能正确修复 2 个缺陷。在 Hanabi 中，模板 2 可以产生 9 个。通过模板 2 提供的正确修复的数量，也显示出 Hanabi 的修复能力优于 ACS。

表 4.14　两个模板在修复中的表现

模板编号	Correct	Incorrect	Total
模板 1	20 (62.5%)	1 (10%)	21 (50%)
模板 2	12 (37.5%)	9 (90%)	21 (50%)
总计	32	10	42

X (Y): X 是补丁个数, Y 是其在同一列内占的百分比。

问题 7：Hanabi 能否缓解过拟合问题。 Hanabi 的研究目的是通过玲珑框架缓解缺陷自

动修复技术中的补丁过拟合问题，通过针对问题 4 、问题 5 和问题 6 的回答，我们可以看出，相比已有方法，Hanabi 在保持最高召回率的前提下，也实现了较高的精度，同时也修复了 8 个其余修复方法不能修复的缺陷。换句话说，玲珑框架可以在更复杂的补丁空间中更准确地搜索到正确的补丁。因此，可以得到结论：在将程序修复问题转化为程序估计问题后，使用高性能的玲珑框架实例，可以缓解当前程序自动修复中面临的补丁过拟合问题。

4.5 讨论

本节分析并讨论 Hanabi 的修复能力。通过分析两个复杂缺陷的修复过程，分析 Hanabi 的修复能力。可以看到，将基于测试的缺陷修复问题归约为程序估计问题，并使用玲珑框架求解是可行的。

4.5.1 缺陷 Math-15 的修复分析

```
double r0 = value;
long a0 = (long) FathMath.floor(r0);
//开发者的补丁
+if(FastMath.abs(a0)>overflow){
//Hanabi的补丁
+if(FastMath.abs(a0)>FastMath.abs(overflow)){
//缺陷
-if(a0>overflow){
  throw new FractionConversionException(value,a0,1);
}
```

Listing 4.1 缺陷 Math-26 及其补丁

列表 4.1 展示了 Defects4J 数据集编号为 Math-26 的缺陷及其补丁。目前，在已有的修复工作中，只有 Hanabi 能正确修复该缺陷。这段代码检查变量 `value` 是否超过了溢出界限 `overflow`，如果超过则抛出异常。然而，出错代码只考虑到了 `value` 为正数时的情况。开发者的补丁首先计算绝对值，然后进行比较，如此能兼顾正数和负数的情况。

这个缺陷比较复杂，因为它引入了多个变量、二元运算和方法调用。同时，在出错位置的上下文中，共有七个可访问的变量，而且可访问的 API 方法数也非常巨大。由于该补丁的复杂程度，目前没有方法能修复它。

首先分析为什么已有方法无法修复这个缺陷。对于一些限制简单补丁空间的修复方法，例如 ACS[1]、ELIXIR[86] 和 Nopol[43] 等方法，限制了 API 调用，因此无法产生正

确的补丁。而且，该补丁的缺陷报告也不提供补丁的信息①，限制了 ELIXIR 的功能。该代码并不存在相似代码，因此基于相似代码的修复方法也无法工作，例如 ssFix[68]、SimFix[33] 和 CapGen[2] 等。

接下来分析 Hanabi 的搜索空间。Hanabi 的检验空间是通过收集项目中的表达式，并将表达式中的变量位置挖空得到的。该项目有 2023 个不同的检验，平均每个表达式拥有 2.06 个变量。因此，在缺陷位置处由条件表达式组成的补丁空间大小为 $2023\times(2+4+3)^{2.06}\approx 187,000$。如此巨大的空间使得我们无法用暴力穷举方法解决，并且已有机器学习方法无法在如此多的类别进行分类。得益于实例化的玲珑框架，Hanabi 能够产生正确修复。

4.5.2　缺陷 Camel-7459 的修复分析

```
1  for (int i = 0; i < uri.length(); i++) {
2    char ch = uri.charAt(i);
3    char next;
4    //开发者的补丁
5    +if(i<=uri.length()-2){
6    //Hanabi的补丁
7    +if(i<uri.length()-1){
8    //缺陷
9    -if(i<uri.length()-2){
10       next = uri.charAt(i + 1);
```

Listing 4.2　缺陷 Camel-7459 及其补丁

列表 4.2 展示 Bugs.jar 中的一个缺陷及其补丁。参照表 4.9中的项目信息，Bugs.jar 中的项目规模显著大于 Defects4J，这使得程序的空间更加巨大。而且当前位置可访问的变量数为 11 个，更加剧了补丁空间的指数爆炸。

然而，当前代码片段提供了足够的上下文信息，例如内部是使用变量 `i` 作为循环索引，并且使用 `uri.length()` 作为边界。缺陷处代码也能提供一些信息，例如变量的使用情况和所调用的方法等。因此，在 Hanabi 的修复过程中，变量 `i` 和 `uri` 在当前检验内有了更高的概率。最终，正确的补丁在百万数量级的补丁空间中排名第 89，同时也是第一个可以通过全部测试的补丁。

4.6　小结

本章提出了将基于测试的程序修复问题转化为程序估计问题并解决的方法。针对修复条件语句缺陷，实现了将玲珑框架实例化的方法。在实例化的玲珑框架上，系统

①https://issues.apache.org/jira/browse/MATH-836

地探索了适合修复的配置，并实现了修复条件语句缺陷的方法 Hanabi 。实验结果表明，Hanabi 的修复效果超越了现有的基于测试的修复方法，证实将基于测试的修复归约为程序估计问题并求解是可行的。

第五章　基于状态同余的测试验证加速方法

5.1　引言

在2.1节中介绍的缺陷自动修复相关方法中，可以看到基于测试集的不完全规约修复方法是当前的主要研究方向。然而，正如前文所述，基于测试的修复验证补丁时依赖运行测试，而实践中往往需要验证大量补丁才会遇到能够通过测试的补丁。由公式1.2可知，验证阶段的计算复杂度很高。为了验证补丁，朴素的方法是将候选补丁依次编译并运行测试集。然而，对于大型软件，编译和运行测试集的时间，都不是可以忽略的。由于待验证补丁集合巨大，验证是缺陷自动修复工具中耗时最多的部分。例如，在缺陷自动修复方法 AE[21] 中，验证阶段执行测试的时间可以占全部修复时间的 60% 以上[32]。

在现有研究中，为了保证实验的可行性，往往对修复时间设置上限。因此，如果补丁的验证速度变慢，则在一定时间内，修复工具能探索的疑似出错位置和候选补丁都会变少，从而不利于成功修复。当前修复方法中，为了保证探索足够多的缺陷位置和补丁，往往使用较高的时间上限。例如，SimFix[33]、GenPat[34] 等方法将时间上限设为 5 小时，AE[21] 、Prophet[36] 和 Fix2Fit[35] 等方法的时间上限达到了 12 小时。长达数小时的修复时间拖慢了开发进度并且无法及时地为开发者提供反馈，降低了缺陷修复技术的实用性。

Weimer 等人很早就提出基于测试的缺陷自动修复方法本质上就是变异分析 (mutation analysis) 方法[21]。补丁 (patch) 和变异 (mutant) 都是对于原始程序的小规模改动。生成补丁的方法，可以视为是变异操作 (mutation operator)。验证补丁是否正确和检查变异是否存活都需要编译并执行已有测试集。而这两个技术都面临着在执行测试阶段计算消耗过大的问题。

变异分析的加速是被广泛研究的课题，有研究者将变异分析的加速技术应用在缺陷自动修复上。例如，AE[21] 通过等价变异体过滤技术剔除语义等价的补丁；JAID[38] 引入了变异提要 (mutant schemata)[121] 技术削减编译开销；F1X[39] 利用测试等价技术[119] 将补丁按照能通过的测试划分等价类，每个等价类只需运行一个代表的补丁即可获得结果。然而，当前基于测试的修复流程中依然存在大量冗余计算。补丁验证阶段耗时过长、验证效率过低，依然是当前基于测试的修复技术的主要瓶颈。

其计算冗余主要来自于编译阶段和测试运行阶段。在编译阶段，由于补丁和补丁之间绝大部分代码是相同的，这意味着补丁在编译过程中存在大量对相同代码的重复编译。在测试运行阶段，对大量补丁反复执行测试会引入大量的重复计算，例如对于同一组测试输入，所有补丁的启动阶段的计算都是重复的。

为了缓解这两个问题，学者分别提出了变异提要方法[121]和分支流计算(split-stream execution)方法[170,171]。

变异提要是将某个程序位置保留接口，而将该位置不同的实现(即不同的变异/补丁)一同编译到接口中，运行时动态选择具体的实现。变异提要技术可以将编译时的计算复杂度由 $O(n)$ 降至 $O(1)$。变异提要技术示例如图5.1所示。图中将两个变异和原始程序编译成共一个可执行文件，并通过在启动测试时设置全局变量 `MUT_ID` 的值来选择启用的代码。

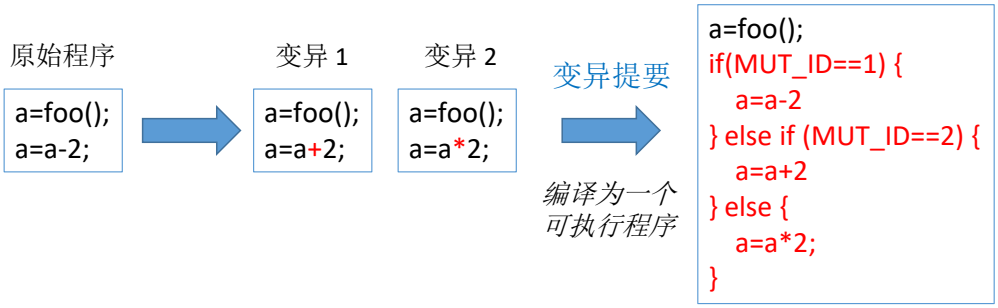

图 5.1　变异提要技术示例

分支流计算是将所有变异(补丁)在同一个进程中启动，运行到变异(修复)位置后，为每个变异(补丁)分支(fork)出一个新的子进程完成剩余的计算。分支流计算可以共享一个补丁位置之前的运算。

为了加速测试验证补丁的过程，本文提出一种动态分析方法 AccMut 来约减测试执行中的冗余计算。与分支流计算一样，AccMut 携带所有补丁启动一个进程，当遇到有补丁的位置时，分析在当前状态下补丁的运行结果，并将其结果分为等价类，为每个等价类分支出子进程完成剩余执行。若有若干个补丁在同一代码位置(位置可以是表达式、语句或基本块等)，在相同的状态下，程序在该位置执行完成后达到相同的状态，则称这些补丁为**状态同余**。换句话说，我们无法根据一个代码位置执行前后的状态来区分出状态同余的补丁。AccMut 通过合并状态同余的补丁，实现约减冗余计算，继而达到加速验证过程中的测试执行。

AccMut 使用 `POSIX fork()` 系统调用分支出新的进程。`POSIX fork()` 的写时复制(copy-on-write)机制保证了创建新进程的效率。同时，通过限制归并等价类和分支的复杂度，将代价控制在较小的范围。由于补丁验证和变异分析的等价性，AccMut 也可以用于加速变异分析的执行速度。

实验表明，AccMut 可以大幅提升补丁验证效率。在大型开源软件上，AccMut 比变异提要技术平均加速 8.95 倍，比分支流执行技术平均加速 2.56 倍。

5.2　方法概览

本节以一个启发式样例整体介绍 AccMut 。以如下代码为例。

```
1  int foo(int a){
2    int i, res;
3    a = a + 1; // P1:a << 1
4    for(i = 0; i < 2; i++){
5      a = a/2; // P2:a + 2, P3:a * 2
6      res += time_consuming(a);
7    }
8    return res;
9  }
10 void test_foo(){
11   assert(foo(1) == RESULT);
12 }
```

其中，`foo` 是被测试函数，而所示的一个测试用例是检查当输入 `a=1` 时该函数的返回结果是否为 `RESULT`。在函数 `foo` 中，会在循环内调用一个计算耗时但无副作用 (side effect) 的函数 `time_consuming`。该方法内有两个疑似出错的位置，即第 3 行和第 5 行。第 3 行上有一个待验证补丁 *P1*，将 `a = a + 1` 替换为 `a = a « 1`。第 5 行上有两个补丁 *P2* 和 *P3*，分别把 `a = a / 2` 替换为 `a = a + 2` 和 `a = a * 2`。

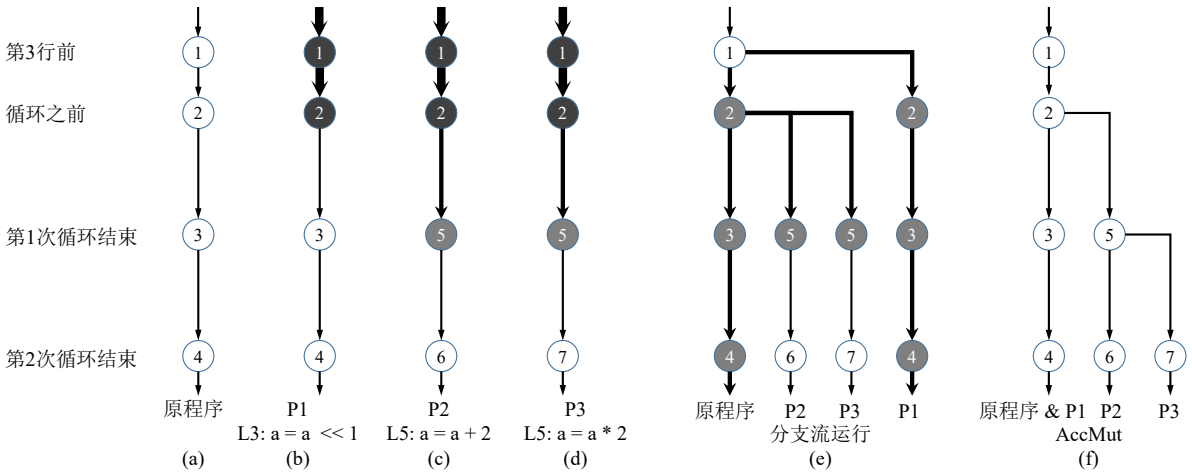

图 5.2　示例代码执行过程中的冗余计算分析

图5.2中展示验证示例代码的不同补丁的方法的执行冗余情况。其中，圆圈代表程序状态，内部用数字标记序号，数字相同的圆圈代表状态相同。圆圈的颜色代表状态的冗余程度，颜色越深代表冗余越多。箭头代表程序执行中的状态迁移过程，箭头越粗表示该执行冗余的越多。

在朴素的验证方法中，修复工具会依次从头到尾执行各个程序变体，并收集它们的运行结果。原程序和验证三个补丁的执行过程，分别如图5.2(a)、(b)、(c) 和 (d) 中所示。可以发现，在朴素的验证过程中，不同补丁的起始部分的运行都是冗余的 (如状态 1 和状态 2)。在 *P2* 和 *P3*，在执行到第一次循环结束时，都达到状态 5，二者之前的运行也是冗余的。

分支流执行的过程如图5.2(e) 所示。最初的进程携带原始程序和所有补丁开始，执行到第 3 行时，分支出新的子进程给补丁 *P1*，并在子进程中完成其剩余的计算。当第一次执行到第 5 行时，分支出两个新子进程给 *P2* 和 *P3*，分别完成各自的执行。可以看到，状态 1 的冗余完全被消除了。状态 2、状态 3、状态 4 和状态 5 的冗余程度减轻，只有两个进程重复执行到这些状态。分支流执行约减了一部分冗余的计算，但是还有提高的空间。

AccMut 的执行过程如图5.2(f) 所示。与分支流执行一样，AccMut 执行到需修复的位置，首先预执行该位置的补丁，并收集在当前状态下执行补丁的结果，将结果按等价类划分，再判断需要多少新的子进程。例如执行到第 3 行，发现 *P1* 的对状态的改变与原始程序一致，即原始程序和 *P1* 状态同余，因此继续在原始进程中携带 *P1*。当第一次执行到第 5 行时，发现 *P2* 和 *P3* 与原程序的状态不同，但是 *P2* 和 *P3* 是状态同余的，因此分支出新的进程携带这两个补丁。新进程再次执行到第 5 行时，发现 *P2* 和 *P3* 结果状态不再相同，因此再分支出子进程将二者分开。最终，我们可以发现，对于上述验证样例，AccMut 可以削减全部的冗余计算。

为了实现上述功能，在进程的每个状态，AccMut 首先预执行一系列语句，并将这些语句造成的状态改变收集起来并聚类。AccMut 将对状态造成的“改变”按等价类划分。如果等价类的大小 n 大于 1，则 AccMut 分支出 $n-1$ 个子进程。每个子进程代表等价类中的语句，然后在子进程中应用对状态的“改变”。对于原始进程，则应用原程序的改变。如此执行，直到每个进程结束。

然而，在现实中存储、比较和聚类“改变”是代价很高的操作。尤其是当一个语句产生了大规模的“改变”，例如一个函数调用语句调用了一个有大量副作用的函数，改变了很多内存单元的值。因此，保存并比较全部的状态是不现实的。

为了解决这一问题，AccMut 记录抽象改变而非具体改变。抽象改变远远小于具体状态改变，但是保留了足够的信息来描述状态改变的内容。例如，一个函数调用的抽象改变可以用其所有参数来表示，而不是执行该函数的状态迁移过程，而当需要实际改变当前状态时，再通过这些参数执行函数。通过这种方法，我们可以减少存储和比较改变的代价。

当两个抽象改变相同时，则将其应用在同样的系统状态会进入相同的新状态。而两个不同的抽象状态，产生的新状态可能相同也可能不相同。例如，以不同的参数调

用一个函数，也可能产生相同的改变。

5.3　AccMut 基础框架

本节将形式化地描述 AccMut 方法。首先给出一些必要的定义，然后在这些定义的基础上由简单到复杂地依次描述朴素的补丁验证方法、分支流执行和 AccMut 。

5.3.1　定义

为了描述 AccMut 方法，本小节给出必须的定义。按 Weimer[21] 等人所提出的，生成补丁的方法是由修复工具提取的变异算子，再通过将变异算子应用在可疑出错的代码上，即可得到该位置的候选补丁集合。而朴素基于测试的自动修复会将补丁应用在出错软件，产生大量的程序变体 (variant)，最终通过测试集验证每个变体。为了一般化描述，本章将程序视为是一个抽象的位置 (location) 集合组成。位置是可以应用变异算子的代码单位，而且一个程序的每个补丁都有唯一的标志 *ID*。

变异过程 p 是一个函数，将一个位置映射到一个变体集合。每个变体 v 由其代码 $v.code$ 和其唯一的标志 $v.I$ 组成。对于程序中的任意两个位置，其变体集合是一致的，即有 $\bigcup_{v\in p(l_1)} v.I = \bigcup_{v\in p(l_2)} v.I$。对于同一个位置的两个变体集合是不相交的，即对于任意的位置 l，有 $v_1, v_2 \in p(l) \Rightarrow v_1.I \cap v_2.I = \emptyset$。给定一个 ID 为 i 的变体，在每个满足 $i \in v.I \wedge v \in p(l)$ 的位置 l，其代码块 $v.code$ 组成一个新的程序，称其为变异 (mutant)。

例如，考虑如下程序：

```
1  a = a + 1;
2  b = b + 1;
```

假设，我们尝试替换其中的算数运算符来产生补丁，即变异算子为将 +、-、* 替换为其他二元算数操作符。第 1 行，将会产生 3 个程序变体：(a) `a = a + 1`、(b) `a = a - 1` 和 (c) `a = a * 1`。其中，(a) 为原程序而 (b) 和 (c) 为补丁 (即变异)。类似地，第 2 行也产生 3 个程序变体：(d) `b = b + 1`、(e) `b = b - 1` 和 (f) `b = b * 1`。上述 4 个补丁的 ID 编号为 1-4。如果只能应用单行补丁 (即变异分析中的一阶变异)，这两行中，最多只能启用一个变异，即当第 1 行使用 (a) 的时候，才能在第 2 行使用 (e) 或 (f)。我们可得：(a).I = {3, 4}、(b).I = {1}、(c).I = {2}、(d).I = {1, 2}、(e).I = {3} 以及 (f).I = {4}。

该模型也支持同时修复多行的补丁，即同时在程序的多处应用补丁，在其他行应用原程序的代码块。

给定程序 p，我们使用 $STATE_p$ 表示其运行中所有可能的状态。简单地讲，状态是由内存、寄存器和外设等存储位置的值的集合，而程序的执行可被看作是一个状态序

列。

我们定义函数 ϕ，映射状态到即将要执行的代码位置。如果状态为 s 的程序即将终止，则有 $\phi(s) = \perp$，即该进程将结束。`execute` 操作模拟了程序的执行。即给定状态 s 和代码块 c，$\texttt{execute}(s, c)$ 是在状态为 s 的情况下执行 c 并更新 s。`execute` 可以看做两个连续的原子操作：`try` 和 `apply` 。$\texttt{try}(s, c)$ 仅在 s 下执行 c，返回用于描述系统改变的变量 x，但并不更新 s。而 $\texttt{apply}(x, s)$ 则在状态 s 上应用 x 的改变，并更新 s。其中，必须保证 $\texttt{apply}(\texttt{try}(s, c), s)$ 的结果与 $\texttt{execute}(s, c)$ 相同。但是，当满足 $x = y \Rightarrow \texttt{apply}(x, s) = \texttt{apply}(y, s)$ 时，$x \neq y \Rightarrow \texttt{apply}(x, s) \neq \texttt{apply}(y, s)$ 不一定满足，从而可以定义*抽象改变*。

为了有效地实现 AccMut，我们需要三个额外的操作。由于这三个操作需要频繁使用，因此实现应尽可能的高效。

- `fork` 。该操作与 POSIX 系统调用 `fork()` 类似，作用是从当前进程中分支出一个子进程。
- $\texttt{filter\_variants}(V, I)$。该操作根据补丁的 ID 集合更新一个程序变体集合，只保留这些启用的补丁的变体，即 V 被更新为 $\{v \mid v \in V \wedge v.I \cap I \neq \emptyset\}$。其中，$V$ 中的变体来自同一的位置。
- $\texttt{filter\_mutants}(I, V)$。该操作根据一个变体集合更新补丁的 ID 集合，保留只有一个变体的补丁的 ID，即 I 被更新为 $\{i \mid i \in I \wedge \exists v \in V.i \in v.I\}$。其中，$V$ 中的变体来自同一的位置。

5.3.2 朴素的基于测试的补丁验证过程

Algorithm 5.1: 朴素的基于测试的补丁验证

```
   Input: p: 一个变异过程
   Data: s: 当前系统状态
1  foreach i ∈ ID do
2      s ← 初始系统状态
3      while φ(s) ≠ ⊥ do
4          {v} ← filter_variants(p(φ(s)), {i})
5          execute(v.code, s)
6      end
7      save(s, i)
8  end
```

在上一小节中的定义的基础上，本小节描述朴素的补丁验证过程，如算法5.1所示。给定系统中所有的补丁集合 ID，该算法依次验证其中的每个补丁 (第 1 行)。验证一个补丁的执行过程是一系列的状态迁移 (第 3 行)。在每次迁移，系统首先选择当前程序

位置上合适的变体 (第 4 行)，然后执行该变体 (第 5 行)。最后通过调用 save(s, i) 将运算结果保存 (第 7 行)。

值得注意的是，为了节省变异时开销，本节所描述的算法都是在应用变异提要方法的基础上。这与分别编译再执行是等价的。

5.3.3　基于分支流运行的补丁验证过程

Algorithm 5.2: 分支流运行和 AccMut 的主体循环

```
   Input: p: 变异过程
   Data: s: 当前系统状态
   Data: I: 当前进程所表示的补丁的 ID 集合
1  I ← 所有的补丁 ID
2  s ← 初始状态
3  while φ(s) ≠ ⊥ do
4  |  proceed(p(φ(s)))
5  end
6  for i ∈ I do
7  |  save(s, i)
8  end
```

算法5.2展示了分支流执行的主体循环。其与朴素的验证算法有 3 点不同：(1) 对于每个进程，维护一个集合 I 表示其携带的补丁的 ID；(2) 在运行的结束，为每个 I 中的 ID 执行 save() 操作；(3) 过程中调用了函数 proceed()，用于处理状态迁移和实施分支。

函数 proceed() 的细节见算法5.3。如果只有一个变体需要执行，则直接执行并返回 (第 2-6 行)。如果有多个变体，则为当前进程选择一个变体 (第 7 行)，并为剩余的每个变体都分支其新的子进程 (第 8-15 行)。当分支出新进程之后，该进程代表对应的变体的补丁的 ID(第 12 行)。最终，更新原始进程中的补丁 ID 集合，并根据所选择的变体执行 (第 16-17 行)。

5.3.4　基于 AccMut 的补丁验证过程

AccMut 的主体循环与分支流执行一致，都是算法5.2，但是函数 proceed() 的算法不同。AccMut 的 proceed 细节如算法5.4所示。首先与分支流执行一样，检查变体的数量 (第 2-6 行)。从第 7-10 行开始两个方法开始不同。AccMut 收集不同补丁的改变到集合 X。然后将改变按定价类划分聚类 (第 11 行)。剩余的算法部分与分支流执行类似，将每个等价类与一个变体对应并执行，只不过在分支流计算中，每个等价类只会有一个变体。首先，选择当前进程代表的等价类 (第 12 行)；然后，为每个聚类分支出新的子进程 (第 13-22 行)；最后更新补丁 ID 和每个进程各自的状态 (第 23-26 行)。

Algorithm 5.3: 分支流执行的 proceed(V) 函数的算法

```
Input: V：当前位置的变体集合
Data: s：当前系统状态
Data: I：当前进程表示的补丁的 ID 的集合
filter_variants(V, I)
if |V| = 1 then
    v ← V 中唯一的变体
    execute (v.code, s)
    return
end
v′ ← V 中任意一个变体
foreach v ∈ V, 且 v ≠ v′ do
    pid ← fork()
    if 在子进程中 then
        filter_mutants(I, {v})
        execute (v.code, s)
        return //在子进程中直接返回
    end
end
filter_mutants(I, v′)
execute (v′.code, s)
```

5.3.5 AccMut 的正确性证明

定理 5.1. *朴素的补丁验证、分支流执行和 AccMut 的算法会按照相同序列、以相同的参数执行 $save(s, i)$。*

证明概要。该定理可以由关于状态迁移数量的数学归纳法证明。假设对于这三个算法，每个补丁的状态迁移过程是相同的。当状态迁移序列的长度为 0 时，该命题成立。当长度为 k 时，可以看到对于每个补丁每个算法选择相同的变体，因此该命题对于 $k+1$ 成立。

5.4 AccMut 的实现

本节介绍在 LLVM-IR 的基础上实现 AccMut 框架的具体细节。LLVM[172] 是被广泛使用的编译基础套件，其可以支持多种主流语言作为前端，例如 C/C++、Java 和 Python 等。LLVM 将所有前端语言转换为其中间表示 (intermediate representation) LLVM-IR，再由后端针对不同的处理器架构生成相应的机器码。已有研究表明，由于源码写法多种多样，而转化到中间代码后模式数量减少，从而有利于修复，并且修复速度快于源码级别的修复[78]。因此，本文针对 LLVM-IR 实现 AccMut 并验证。AccMut 的实现已开

Algorithm 5.4: AccMut 的 proceed(V) 函数的算法

```
Input: V：当前位置的变体集合
Data: s：当前系统状态
Data: I：当前进程表示的补丁的 ID 的集合
filter_variants(V, I)
if |V| = 1 then
    v ← V 中唯一的元素
    execute (v.code, s)
    return
end
X = ∅
foreach v ∈ V do
    X ← X ∪ {try(v.code, s)}
end
𝕏 ← 将 X 中的改变按等价类划分
X_cur ← 等价类 𝕏 中任意一个元素
foreach X ∈ 𝕏 − {X_cur} do
    V ← X 中对应的变体
    pid ← fork()
    if 在子进程中 then
        filter_mutants(I, V)
        x ← 等价类 X 中的改变
        apply(x, s)
        return
    end
end
V ← X_cur 中对应的变体
filter_mutants(I, V)
x ← 等价类 X_cur 中的改变
apply(x, s)
```

源至 Github ①。

为了实现 AccMut，需要实现以下几个操作：`try`、`apply`、`fork`、`large_change`、`filter_variants` 以及 `filter_mutants`。本节将依次介绍实现情况。

5.4.1　变异算子

变异算子应用在代码上可以得到补丁。AccMut 中所采用的变异算子如表 5.1所示。这些变异算子在已有的修复工作中是较为流行的并且被证实是有效的，例如 Par[73]、Kali[25]、ASTOR[59] 和 TBar[80] 等都采用了表中的部分变异算子。

①`https://github.com/wangbo15/accmut`

表 5.1 AccMut 中用于生成补丁的变异算子

变异操作名称	描述	示例
AOR	替换算数操作符	$a + b \rightarrow a - b$
LOR	替换逻辑操作符	$a \,\&\, b \rightarrow a \mid b$
ROR	替换关系运算操作符	$a == b \rightarrow a >= b$
LVR	替换数字字面量	$T \rightarrow T + 1$
COR	替换位操作符	$a \,\&\&\, b \rightarrow a \,||\, b$
SOR	替换移位操作符	$a >> b \rightarrow a << b$
STDC	删除函数调用	$foo() \rightarrow nop$
STDS	删除内存存储	$a = 5 \rightarrow nop$
UOI	插入单元运算符	$b = a \rightarrow a++;\ b = a$
ROV	改变操作符顺序	$foo(a,b) \rightarrow foo(b,a)$
ABV	插入求绝对值操作	$foo(a,b) \rightarrow foo(abs(a),b)$

5.4.2 fork 的实现

本文直接使用 POSIX 系统调用 `fork` 来实现 `fork` 操作。POSIX `fork` 是类 Unix 系统中用于创建进程的主要方式。由于其写时复制机制 (copy-on-write mechanism)[173]，保证了分支新进程的效率。即父进程和子进程虽然在不同的虚拟内存空间中，但是最初共享同样的物理内存，若其中一者有写入操作，则将所写入的物理内存页复制。通过这种方式实现 `fork` 可以在常数时间内完成该操作。

然而直接应用 POSIX `fork` 会引入问题。虽然 POSIX `fork` 可以保证内存的复制机制，但是子进程与父进程依然共享一些资源。尤其是二者的文件指针并没有分开，即子进程写入文件后，移动了文件指针的位置，这会影响到父进程。为了解决这一问题，AccMut 实现了虚拟文件系统，即在文件打开时，就将所有内容读入内存，进程结束时再将文件内容写入按照进程标志 (pid) 编码的硬盘文件。通过该虚拟文件系统，可以同样利用 POSIX `fork` 的写时复制机制来处理文件系统。实现采用 POSIX `fork` 也会引入其他的一些限制，例如不能处理多线程程序[174] 和内核等系统底层程序[175] 等。

5.4.3 try 和 apply 的实现

实现 `try` 和 `apply` 的关键在于如何定义*抽象改变*。由于 LLVM-IR 是静态单赋值形式的三地址码，大多数指令只修改某内存地址的固定大小的内存单元。对于这些指令，*改变*很容易定义，只需记录该指令修改的内存位置以及在该位置上新存入的值即可。

但是对于函数调用 `call` 需要特殊处理，因为被调用函数可能存在大量的修改，影响很多内存单元。因此，如前文示例所述，函数调用的抽象改变被定义为传入的参数的值。因为 LLVM-IR 是类型良好定义的语言，这种抽象能保证正确性。例如对于函数 $void\ foo(int, int)$ 有如下两个补丁：ROV (交换参数位置，$foo(a,b) \rightarrow foo(b,a)$) 和 STDC(删

除该调用)。即该位置有三个程序变体，$foo(a,b)$、$foo(b,a)$ 和空操作 (nop)。当 $a=b$ 时，前两者的抽象改变是等价的，而第三个与前二者是不同的。

5.4.4 filter_variants 和 filter_mutants 的实现

由于 filter_variants 会在每个位置被调用，而 filter_mutants 会在每次分支新进程后被调用，因此二者的复杂度应尽可能的降低，最好可以降至 $O(1)$。本小节讨论如何针对只修改一处的补丁实现这两个操作。

而这两个操作都含有复杂度较高的计算，例如集合的交运算，对于大小为 n 的集合，其理论复杂度是 $O(n\log n)$。而表示补丁 ID 的集合有可能很大，会导致这两个操作代价很高。为了实现高效的操作，AccMut 利用基于变异的补丁生成方式的特点，即每个位置程序变体数有一个较小的上限 u。如果两个操作复杂度的是关于 u 的，则有 $O(u)=O(u^2)=O(\log u)=\ldots=O(1)$，即将有常数时间的复杂度。

在每个程序位置都有两种程序变体：一种是通过变异算子产生的补丁，被称为是*补丁变体*；另一种是原始程序，被称为是*原始变体*。对于其余的补丁 ID，则启用原始变体，即保证每次只有一处位置应用补丁。

在此基础上，本文对于不同的数据结构设置不同的策略。首先，对于每个进程维护一个保持补丁 ID 的集合。最初，该集合包含全部的补丁的 ID。每当分支出新的进程时，该集合在子进程中的大小不会超过 u。因此，可以使用两种不同的策略表示该集合。对于原始进程的，使用位向量 (bit vector) 存储，每一位表示一个补丁 ID。当该位置为 1 时，表明该补丁被携带，0 表示没有被携带。在位向量中，添加、删除和访问 ID 的操作的复杂度是 $O(1)$。当一个进程被分支出来，其位向量继承自其双亲进程，同时使用一个长度固定为 u 的数组表示 ID 集合。因此，在有限长度的数组上的查询等操作的复杂度也是 $O(1)$。

另外，在算法5.4中有程序变体的集合 V。同样对于每个变体都有一个有其表示的补丁的 ID 集合。对于两种不同的变体 (即补丁变体和原始变体)，可以使用不同的表示方法。AccMut 使用元组 VariantSet 存储变体集合。VariantSet 的结构是 (ori_variant, ori_included, mut_variants)。其中，ori_variant 是原始变体的代码；ori_included 是布尔变量，表示原始变体是否被包含在集合；mut_variants 是补丁变体的列表。每个补丁变体 v 都有代码块和一个补丁 ID。为了避免混淆，可以使用 $v.i$ 来表示唯一的补丁 ID。通过这种方法，可以快速的检查原始变体是否在集合中。同时，由于 mut_variants 的大小上界是 u，因此在这个集合上的操作是常数时间。

filter_variants 的实现见算法5.5。首先过滤补丁变体 (第 3-6 行)。因为其中所有的操作计算复杂度为 $O(1)$(因 u 上界较小)，而此处循环的复杂度也被限制在 u 以内，因此该部分复杂度是 $O(1)$。之后，决定原始变体是否被保留 (第 8-12 行)。因为只能应

Algorithm 5.5: `filter_variants`

Input: V：需要被过滤的变体集合
Input: I：用于过滤 V 的补丁 ID 集合

```
1  V' ← 新建 VariantSet
2  V'.ori_variant ← V.ori_variant
3  foreach v ∈ V.mut_variants do
4  |   if I.contains(v.i) then
5  |   |   V'.mut_variants.add(v)
6  |   end
7  end
8  if V'.mut_variants.size < I.size then
9  |   V'.ori_included = true
10 else
11 |   V'.ori_included = false
12 end
13 V ← V'
```

Algorithm 5.6: `filter_mutants`

Input: I: 需要被过滤的补丁 ID 集合
Input: V: 用于过滤 I 的变体集合
Data: MV: 当前位置所有的补丁变体集合

```
1  if V.ori_included then
2  |   foreach v ∈ MV − V.mut_variants do
3  |   |   I.remove(v.i)
4  |   end
5  else
6  |   I ← 新建空链表
7  |   foreach v ∈ V.mut_variants do
8  |   |   I.add(v.i)
9  |   end
10 end
```

用一处补丁变体，所以如果当前选择的变体比当前进程表示的补丁少，则必选择剩余的补丁和原始变体。类似地可得该部分的复杂度也为 $O(1)$。因此该方法整体的复杂度是 $O(1)$。

`filter_mutants` 的实现见算法5.6。如果 V 中有原始变体，则移除其他补丁变体(第 1-4 行)。否则，直接添加相应的补丁 ID(第 5-9 行)。同理，该算法的复杂度也为 $O(1)$。

5.4.5　并行控制

如果不控制并行，AccMut 将会短时间内产生大量的进程，而过多的进程会带来系统拥堵，并且引入很多进程调度的开销。为此，AccMut 限制了可并行的最大进程数。为了直观地衡量 AccMut 的性能，在实现中将子进程上限设为 1。首先，在实验中，可以避免调度开销引入的性能波动，有利于获得稳定的实验数据。其次，实现较为简单，只需要分支子进程时让双亲进程等待 (wait) 即可。AccMut 可以扩展为支持多个进程并行的实现。

5.5　实验验证

本节实验验证的目标是回答以下研究问题：**与现有方法相比，AccMut 的加速效果如何？**

5.5.1　实验对象

为了验证 AccMut 加速效果，本文选用了 11 个真实的开源软件，如表 5.2所示。其中，除了 Vim 7.4，其余十个都来自 SIR 数据集[176]。SIR (Software-artifact Infrastructure Repository)①数据集是在软件质量相关研究中被广泛使用的开源软件数据集。本次验证中所选取的软件功能比较多样化，包括词法分析、文件压缩和文本编辑器等等。另外，由于 Vim 7.4 过于庞大，本文只选取了其中两个最大的组件 *eval* 和 *spell* 作为实验对象，在表 5.2中二者大小的之和在括号中表示。这些项目共有 50 万行代码、20736 个测试、27612 尝试修复的位置。平均每个位置，尝试应用 12.2 个补丁 (见 $\bar{\boldsymbol{u}}$ 列)。在这些项目中，在一个位置上最多尝试 22 至 43 次 (见 Max $\boldsymbol{u}$ 列)。因此，算法5.5 和算法5.6 的复杂度可视为常数级。

值得说明的是，由于 Java 程序无法支持 `fork` 进程等操作，在类 Unix 系统中，`fork` 是基础的系统调用。因此 AccMut 选择在类 Unix 系统上的 C 语言程序进行验证。由于针对的编程语言不同，本章的实验对象与 Hanabi 的验证对象不同。

①`https://sir.csc.ncsu.edu/portal/index.php`

表 5.2　实验验证 AccMut 的项目

项目名称	LOC	测试数	补丁数	位置数	$\bar{u}$	Max u	描述
flex	10334	42	56916	5119	11.1	32	词法分析生成器
gzip	4331	214	37326	3058	12.2	22	文件压缩工具
grep	10102	75	58571	4373	13.4	34	文本查找工具
printtokens	475	4130	1862	199	9.4	22	简易词法分析器
printtokens2	401	4115	2501	207	12.1	22	简易词法分析器
replace	512	5542	3000	220	13.6	22	模式匹配工具
schedule	292	2650	493	55	9.0	22	优先级调度器
schedule2	297	2710	1077	121	8.9	22	优先级调度器
tcas	135	1608	937	73	12.8	35	简易碰撞模拟器
totinfo	346	1052	756	63	12.0	22	单词统计工具
vim 7.4	477257 (42073)	98	173683	14124	12.3	43	文本编辑器
总计	504482	20736	337122	27612	12.2	—	—

LOC：代码行数，由 *cloc* 收集。**位置数**：尝试应用补丁的位置数。$\bar{u}$：u 的平均值，即一个程序位置平均尝试的补丁数。**Max u**：u 的最大值。

5.5.2　实验过程

在本次实验中,本文选择两个对比方法:变异提要方法[121] 和分支流执行方法[170,171]。如前文所述，变异提要方法可以约减编译时的开销，而分支流执行是在变异提要方法的基础上约减执行时开销。AccMut 在分支流执行的基础上，尝试通过将状态同余的补丁聚类，进一步实现约减冗余计算。本文按照5.3节中的描述算法，重新基于 LLVM-IR 实现了这两个对照方法。

实验中，本文分别针对表 5.2中的项目运行 3 个技术，收集运行时间和所需进程数等实验数据。为了使数据更稳定，所有实验被运行 3 次，并取平均时间为最终结果。

5.5.3　实验结果

实验中三个方法的运行时间见表 5.3，其中我们可以发现：

- AccMut 一般可以在 20 分钟内完成，而变异提要最高需要消耗 3 小时 26 分钟。
- 在所有的项目上，AccMut 表现优于其余二者。这显示出 AccMut 方法的有效性，其约减的冗余计算的收益大于其操作的代价。
- AccMut 相对于分支流执行的加速比平均有 2.56 倍，而相对变异提要加速 8.95 倍。AccMut 可以显著提升现有方法的速度。
- 分支流计算同样显著优于变异提要技术，这符合 Tokumoto 等人的研究[171]。

为了详细分析 AccMut 的性能，本文记录了实验中三个方法各自执行的进程数，如表 5.4所示。在实验中，变异提要会验证全部的补丁，而分支流执行只会验证一个测试所覆盖到的补丁。可以看到，AccMut 所验证的补丁数显著小于另外两个方法，即存在

表 5.3　实验的运行时间结果

项目名	AccMut	SSE	MS	SSE/AccMut	MS/AccMut	MS/SSE
flex	12m58s	40m22s	1h26m17s	3.11x	6.65x	2.13x
gzip	50.4s	2m32s	55m19s	3.02x	65.85x	21.84x
grep	2m19s	7m16s	58m56s	3.13x	25.36x	8.10x
printtokens	11m55s	23m36s	2h10m54s	1.94x	10.98x	5.67x
printtokens2	11m35s	38m7s	57m24s	3.30x	4.97x	1.51x
replace	17m27s	41m22s	44m56s	2.37x	2.57x	1.09x
schedule	3m14s	6m20s	8m14s	1.96x	2.54x	1.30x
schedule2	6m40s	13m18s	17m05s	2.00x	2.56x	1.28x
tcas	7.7s	21.2s	16m31s	2.6x	128.7x	46.7x
totinfo	1m55s	4m28s	6m19s	2.33x	3.30x	1.41x
vim 7.4	1m9s	2m10s	3h26m6s	1.88x	179.2x	95.1x
总计	1h10m10s	2h59m52s	10h28m1s	2.56x	8.95x	3.49x

SSE：分支流执行 (Split-Stream Execution)；**MS**：变异提要 (Mutant Schemata)；**XXX/YYY** = YYY 关于 XXX 的加速比。

大量的补丁是状态同余的。

另外，尽管 AccMut 减少了所需要验证的补丁数量，但是为了实现其功能也引入了额外的开销，例如维护集合的操作、对状态聚类等。本文测量了三个方法 AccMut 所约减的执行指令数和其为了实现功能执行额外的指令数。为了进一步了解开销的情况，本文分析了 AccMut 在执行过程中，所运行的来自原程序的指令数和实现其功能的指令数。由于 Linux 系统信号冲突，导致无法直接测量各部分的时间消耗，因此这里只能以指令数估计。本文跟踪了 *tcas* 和 *printtokens* 在前 100 个测试的执行情况，结果如表 5.5所示。

由于在每个位置需要选择变体，导致三个方法的额外代价都比原始程序多。AccMut 的代价是最高的，额外代价是原本程序执行的 79 倍。但是 AccMut 绝对的开销更低，因为削减了更多的冗余计算。尽管存在很大的开销，但是需注意此处并没有考虑补丁在编译时的代价。在基于测试的修复方法的验证阶段中，朴素的方法是需要将补丁依次植入原程序并编译得到可执行文件后再执行测试。表 5.6 展示了在这两个项目的前 100 个测试上变异提要方法与朴素验证方法的时间对比。可以看到变异提要方法相对朴素验证方法的方式平均快 9.62 倍。

5.6　小结

基于测试的修复方法的效率瓶颈在补丁验证阶段。已有研究表明，在以测试为规约的修复中，执行测试验证补丁时间占全部修复时间的绝大部分。补丁验证阶段中，需要对候选补丁进行编译并执行测试。本质上，基于测试的修复与变异分析一样，都存

表 5.4　平均每个测试所运行的进程数

Subjects	AccMut	SSE	MS	AccMut/SSE	AccMut/MS	SSE/MS
flex	5881.1	16610.7	56916	35.4%	10.3%	29.1%
gzip	434.5	1638.1	37326	26.5%	1.2%	4.4%
grep	1303.2	4014.4	58571	32.5%	2.2 %	6.9%
printtokens	413.5	1019.3	1862	40.6%	22.2%	54.7%
printtokens2	750.4	1724.4	2501	43.5%	30.0%	68.9%
replace	483.7	1484.1	3000	32.6%	16.1%	49.5%
schedule	180.0	405.9	493	44.3%	36.5%	82.3%
schedule2	384.9	844.2	1077	45.6%	35.7%	78.4%
tcas	99.2	434.1	937	22.9%	10.6%	46.3%
totinfo	220.5	566.0	756	39.0%	29.2%	74.9%
vim 7.4	601.0	1472.7	173683	40.8%	0.3%	0.8%
平均	977.5	2746.7	30647.4	35.6%	3.2%	9.0%

SSE：分支流执行 (Split-Stream Execution)；**MS**：变异提要 (Mutant Schemata)；**XXX/YYY** = XXX 占 YYY 的百分比。

表 5.5　执行的指令数

指令类型	AccMut	SSE	MS	AccMut/SSE	AccMut/MS	SSE/MS
原程序指令	1,054,174	2,404,487	37,617,433	43.8%	2.9%	6.4%
额外指令	83,148,833	96,490,502	177,988,701	86.2%	46.7%	54.2%
总计	84,203,007	98,894,989	215,606,134	85.1%	39.1%	45.9%

在对大量程序的微小变体反复执行测试的过程。这个过程中存在大量的冗余计算，降低了修复效率。

本章提出了状态同余概念，并通过分析补丁之间状态同余的性质来削减冗余计算。基于该概念本章提出了 AccMut 方法，通过使用一个进程携带全部补丁，当遇到修复位置时，将该位置上的所有补丁按状态同余关系聚类。其中每个与原始进程中不同的类别分支出新的进程完成剩余的运算。相比于分支流执行方法只能共享修复位置的计算，AccMut 可以进一步共享每个分组内的补丁的计算。

实验结果表明，与分支流执行方法相比，AccMut 加速了 2.56 倍，与变异提要方法相比，AccMut 加速了 8.95 倍。AccMut 能大幅提升验证效率，缓解基于测试的修复方法面临的修复效率较低的问题。

表 5.6　变异提要与朴素的验证方法的运行时间对比

项目名	MS	Naive	Naive/MS
tcas	50s	323s	6.46X
printtoken	36s	504s	14.00X
总计	86s	827s	9.62X

MS：变异提要 (Mutant Schemata)；**Naive**：朴素验证方法。

第六章　基于约束的修复问题的归约

6.1　引言

在前面的章节中，介绍了将基于测试的缺陷修复问题归约为程序估计问题并解决。基于约束的修复是另一类较为常见的修复方法。约束是描述正确程序应满足的属性，可以从安全属性规约或程序语义等方面提取。约束的常见形式是逻辑表达式，并且程序或补丁的语义应满足该表达式。

基于测试的修复对测试集的质量要求较高，已有研究表明修复方法的效果与测试集的覆盖率等指标高度相关[25]。然而在很多情况下并没有高质量的测试集，有时需修复部分甚至没有对应的测试，这使得基于测试缺陷修复方法难以发挥其效果。例如，当用户新发现一个崩溃错误提交缺陷报告时，仅会提供一个可以触发崩溃的输入，而软件已有测试都无法检测到该缺陷。此时，如果仅仅依赖软件原有的测试集，则极易产生过拟合的补丁。当测试不充分时，基于测试的修复方法无法发挥效果的原因主要有两方面：(1) 该类修复方法多使用基于频谱 (spectrum-based) 的缺陷定位，而该方法以来高质量的测试集提供足够的覆盖信息以计算某位置出错的可能性；(2) 使用测试集作为不完全规约对补丁进行验证，导致其返回补丁的仍然无法通过其他的测试或者其他规约[24]。

对于一些特定类型的缺陷，当缺乏高质量测试集的时候，基于约束的修复可以发挥更好的效果。本文发现在一些特定的崩溃缺陷中存在内在的约束。如果能以这些约束作为规约，则可以将修复问题*归约为程序估计问题*，即在给定上下文下生成满足规约的补丁。例如在列表6.1所示代码中，在第 6 行发生数组越界访问，此处下标 `i` 应满足约束是 $0 \leq i \leq len$，而补丁是第 5 行将 `i <= len` 改为 `i < len`。

Listing 6.1　代码约束示例

```
1  int len = 10;
2  ...
3  int array[len];
4  int i;
5  for(i = 0; i <= len; i++) // 修复位置
6    array[i] = i;           // 出错位置
```

为了将基于约束的修复归约为程序估计问题，需要解决以下问题：

- **约束提取**：给定触发缺陷的输入，如何生成约束？值得注意的是，在给定一个触发错误的输入上，如果仅针对具体值修复，则必然产生错误的补丁。在示例代

码中，我们发现当 $i = 10$ 时，`array[i]` 发生访问越界错误。但是如果直接使用程序执行时的具体值将会得到约束 $i < 10$，而非正确的约束 $i < len$。

- **缺陷定位**：如何在没有测试集辅助的情况下进行定位？
- **约束传播**：在程序出错位置提取的约束，并不能直接应用在修复位置，如何将约束正确地传播到修复位置？从修复位置到出错位置，变量可能发生一系列改变，出错约束传播到修复位置，其内容也应随之改变。
- **补丁生成**：如何根据约束生成补丁？所生成的补丁应保证能满足约束。

为了解决上述问题，本文提出一种基于约束的缺陷自动修复方法 ExtractFix 。ExtractFix 通过 Sanitizer [①] 检测特殊类型的缺陷，并将其转为崩溃 (crash)，例如使用 AddressSanitizer[177] 当检测到数组越界等内存错误时，会指出非法访问的位置并崩溃。ExtractFix 需要提供一个可以触发错误输入提供给 Sanitizer 以获得崩溃位置。当发现程序崩溃后，ExtractFix 通过模板提取出*能避免缺陷的约束* (crash-free constraint, CFC)。例如，对于缓冲区溢出错误，ExtractFix 使用的模板如下：

$$access(buffer) < base(buffer) + size(buffer)$$

给定一个可以触发崩溃的输入，ExtractFix 通过 Sanitizer 提取约束，并通过程序中的变量、表达式代入模板避免使用崩溃报告中的具体值而导致产生过拟合的约束。随后，在崩溃位置，通过*静态分析控制和数据依赖的语句和出错执行中动态切片*求交集获得修复位置的集合。接着，从崩溃位置后向 (backward) 由近及远地依次尝试修复位置。在修复位置，通过计算最弱前条件将崩溃位置的*能避免缺陷的约束*反向传播过来。为了高效地通过符号执行计算最弱前条件，本文提出了受限的符号执行，从而降低实施符号执行的代价。最后，将传播来的约束作为规约，以修复位置的程序片段为上下文，可以通过解程序估计问题生成满足约束的补丁并应用。

在实验验证中，ExtractFix 在 56 个缺陷上，成功提取了 43 个缺陷的约束，并正确修复了其中的 28 个缺陷。ExtractFix 在修复数量和质量上均优于现有方法。同时，ExtractFix 的修复效率很高，在实验中的平均修复时间仅为 9.46 分钟，最长修复时间为 41 分钟。ExtractFix 能够缓解程序修复技术面临的补丁过拟合问题和修复效率过低问题。

[①]Sanitizer 直译为除草剂、杀菌剂，在软件领域使用的是抽象含义，因此该词本文不做翻译。

6.2　方法概览

6.2.1　ExtractFix 修复流程概览

崩溃 (crash) 是系统运行时由于违反了某种属性进入到非法的状态导致程序终止的行为。属性可以是程序员提供的断言 (assert)、操作系统提供的隐式断言 (如除零异常和空指针解引用) 和 Sanitizer 提供的程序状态合法性检查。Sanitizer 会对原始程序插装，植入检测语句。例如，对于整数溢出，会在运算后判断结果是否出错，如果出错则通过 `abort` 终止程序。为了避免程序在某输入下崩溃，可以使程序状态在崩溃位置之前满足某种条件，称为*避免崩溃约束* (crash-free-constraint，CFC)。例如，对于指针 `p`，避免其因空指针解引用崩溃的 CFC 是 $p \neq 0$；对于数组访问 `a[i]`，避免发生非法访问的 CFC 是 $0 \leq i < SIZE$，其中 $SIZE$ 是数组 `a` 的元素个数。

ExtractFix 主要包括以下四个主要步骤：

1. **约束提取**：给定程序和一个让其崩溃的输入，ExtractFix 首先在二者基础上提取 CFC。CFC 应该不局限在运行时的具体值，而应该是抽象或者符号化的。ExtractFix 根据不同的崩溃类型选择不同的模板生成 CFC 公式。
2. **错误定位**：在得到崩溃位置后，在该输入所执行语句的动态切片上，以崩溃位置为起点反向遍历，添加与崩溃位置有数据依赖和控制依赖的语句。
3. **约束传播**：CFC 关于是崩溃位置的约束。对于在崩溃位置之前的修复位置，CFC 是修复位置的约束 CFC' 传播而来的。对于两个位置之间的程序 P，三者满足以下霍尔三元组 (Hoare triple)：

$$\{CFC'\}\ P\ \{CFC\} \qquad \text{(CFC 传播)}$$

   CFC' 是保证 CFC 成立的最弱前条件 (weakest precondition)[178]。在上面的公式 (CFC 传播) 中 P 和 CFC 已知。若该式可解，就可以得到 CFC'。ExtractFix 提出受限的符号执行，能高效地将约束反向传播至修复位置。
4. **补丁生成**：在得到修复位置及其 CFC' 之后，下一步是生成补丁。补丁生成是将修复位置的语句 ρ 替换为补丁 f，并使得以下霍尔三元组成立：

$$\{true\}\ [\rho \mapsto f]\ \{CFC'\}\ P\ \{CFC\} \qquad \text{(CFC 修复)}$$

   如果补丁保证 CFC' 被满足，则在崩溃位置 CFC 被满足。以 (CFC 修复) 式为规约，以修复位置周围代码为上下文，则可以通过解程序估计问题得到补丁。

6.2.2 ExtractFix 修复实例

本小节使用开源软件 GNU-Coreutils 中的一个真实缺陷为例，描述 ExtractFix 的修复过程。其代码列表6.2所示。

这段代码目的是用 `bits` 来填充缓冲区 `r`，并通过循环调用 `memcpy` 来实现将剩余的空间填充。但是该段代码有缺陷。根据用户提供的缺陷报告 ①，当 `size=13` 时，`memcpy` 的源区间和目标区间发生重叠，而这是未定义行为。未定义行为在一些系统上会带来风险。若 `size=13`，`for` 循环将会在第二次迭代终止，并且有 `i=6`。接下来，在第 7 行，由于 `r+(13-6)>r+6`(即 `r+7>r+6`)，`memcpy` 的源区间和目标区间发生重叠。Sanitizer 会检测出该未定义行为并崩溃。

Listing 6.2 GNU-Coreutils 缺陷实例

```
1 void fillp (char *r, size_t size){
2   ...
3   r[2] = bits & 255;
4   for (i = 3; i < size / 2 ; i *= 2)
5     memcpy(r + i, r, i);
6   if (i < size)
7     memcpy(r + i, r, size - i) ;
8   ...
9 }
```

首先，ExtractFix 使用触发崩溃的输入 (`size=13`) 执行程序，并在第 7 行崩溃退出。根据崩溃的类型选择生成 CFC 的模板。`memcpy(dest, src, n)` 的源区间与目标区间重叠的未定义行为的模板是 $des + n \leq src \vee src + des \leq dest$。因此，在这个例子中，$CFC$ 是：

$$(r+i)+(size-i) \leq r \vee r+(size-i) \leq (r+i) \equiv (size \leq 0 \vee size \leq 2*i)$$

由于 `size` 的类型是 `size_t`，因此 $size \leq 0$ 恒为真，我们只需要关注 $size \leq 2*i$。其次，根据执行流的切片和控制依赖，第 4 行的循环控制条件被加入修复位置集合。再次，将 CFC 沿着所有可能的路径传播至该修复位置，即沿着 `if` 语句中路径条件为 $i < size$ 的条件传播，最终 CFC' 并没有发生改变。最后，由补丁生成器使用 CFC' 生成补丁 `i<=size/2`。ExtractFix 返回的修复如下：

```
- for (i = 3; i < size/2; i *= 2)
+ for (i = 3; i <= size/2; i *= 2)
```

①https://debbugs.gnu.org/cgi/bugreport.cgi?bug=26545

该补丁与开发者提供的补丁一致。而如果使用基于测试的修复，由于测试严重不足，很容易产生过拟合的补丁。例如，Fix2Fit[35] 在这个例子上，返回的补丁为：

```
- for (i = 3; i < size/2; i *= 2)
+ for (i = 3; i < size/2 || i == 6; i *= 2)
```

很明显，这是一个过拟合的修复，例如，当 `i=7` 时，程序仍会崩溃。

6.3 ExtractFix 方法

ExtractFix 主要包括四个步骤，即约束提取、错误定位、约束传播和补丁生成。本节依次介绍这几个部分。

6.3.1 约束提取

ExtractFix 根据一个有缺陷的程序和一个引发其崩溃的输入提取*避免崩溃约束* (CFC)。ExtractFix 在第一步中得到*崩溃的位置*以及表示如何避免崩溃的*避免崩溃约束* CFC。崩溃位置 (如文件名和行号) 可以由编译器提供的调试信息获得。例如对于 clang 编译器，在编译时开启`-g` 选项，编译器就会在 debug-info [①] 中记录相应语句的源码位置。CFC 则根据出错类型由模板产生，通过在模板中填入出错的语句的元素得到。ExtractFix 当前考虑的崩溃类型主要有以下两种：

1. 开发者提供的显示断言引发的崩溃，如 `assert(C)`。
2. Sanitizer 检测程序的非法状态而导致的崩溃。

ExtractFix 所能处理的崩溃类型，以及对应类型所使用的模板见表 6.1。其中类型 1 是开发者断言引入的崩溃，而类型 2-6 都是 Sanitizer 引入的崩溃。这六种缺陷都是软件中常见的且有较大危害的缺陷。

表 6.1　ExtractFix 支持的崩溃类型及其避免崩溃约束的模板

编号	表达式	*CFC* 模板	错误描述
1	`assert(`C`)`	C	断言
2	`*p`	$p + sizeof(*p) \leq base(p) + size(p) \wedge p \geq base(p)$	内存越界
3	`a` op `b`	$MIN \leq \mathrm{a}\ op\ \mathrm{b} \leq MAX$	整数溢出
4	`memcpy(dest,src,n)`	$dest + n \leq src \vee src + n \leq dest$	未定义
5	`*p (p=null)`	$p \neq null$	空指针解引用
6	`a/b (b=0)`	$b \neq 0$	除零异常

接下来详细讨论根据崩溃提取 CFC 的方法。

[①] https://llvm.org/docs/SourceLevelDebugging.html

用户断言。用户断言直接就是布尔表达式，因此对于 `assert(C)` 语句其 $CFC = C$。

基于 Sanitizer 的约束提取。Sanitizer 是一种动态错误检测方法，一般通过插装 (instrument) 等方式改变程序，植入状态检测和退出代码。当检测到系统出现非法状态时，则报警并退出。例如，为了监测除零异常，Sanitizer 会在每次做除法前检查除数是否为 0；而检测内存访问是否合法时，Sanitizer 会在内存分配时记录该内存对象的起始地址和长度，当访问该内存对象时，根据存储的记录检查是否在合法范围内。

值得说明的是，Sanitizer 报告的出错信息只包括具体值。例如对如下程序的内存访问越界错误：

```
1 len = 4;
2 int* arr = (int*) malloc(sizeof(int)*len); // 设 arr 分配在 0x2000
3 arr[len];
```

Sanitizer 报告的信息为 0x2010 ≤ 0x2000 + 0x10 被违反，引发溢出崩溃。详细报告信息为：访问内存地址 0x2010 时发生上越界，该内存对象基址为 0x2000，长度为 16 字节。

可以看出，Sanitizer 所报告的信息是具体的，无法直接用于生成 CFC。我们真正需要的是符号化的约束，即 $arr + len < sizeof(sizeof(int) * len)$。ExtractFix 通过静态分析，并与程序的调试信息关联，可以取得符号化的 CFC。然而，在有的情况下这种方法无法工作。例如，当 `malloc` 在一个函数被调用，而其分配的对象在另一个函数内被使用时，无法通过现有代码中的变量和符号构造一般化的 CFC。为此，ExtractFix 扩展了 Sanitizer 约束语言，用于处理更一般的情况。

一些 Sanitizer 为了实现能够检测缺陷，通过添加一些扩展状态来记录足够的信息。扩展状态一般是由插装或者 Sanitizer 提供的库来实现。而 Sanitizer 的扩展状态对于被监测的程序而言是透明的。因此，Sanitizer 的断言是基于其扩展状态的，可能会引入它的库的实现。为了能处理 Sanitizer 的扩展状态，ExtractFix 不要求 CFC 中一定要使用原始程序中的变量、函数和表达式，而可以使用 Sanitizer 的扩展状态。例如，在表 6.1中，对于内存访问边界检查，ExtractFix 引入了以下两个操作：

- $base(p)$：指针 p 所指向的内存地址上的对象的基址；
- $size(p)$: 指针 p 所指向的内存地址上的对象的大小。

而对于整数溢出错误，则根据当前操作的整数的位数，提供了对应的最大值和最小值 (即表 6.1 中类型 3 的 `MAX` 和 `MIN`)。

6.3.2 基于依赖的错误定位

在 ExtractFix 得到崩溃位置和 CFC 之后，下一步需要确定修复位置集合。在基于测试的缺陷修复中，一般是利用基于频谱的方法进行缺陷定位。然而该类方法对测试

Algorithm 6.1: 基于依赖的修复位置定位

```
Input: 崩溃位置 crashLoc
Input: 过程间控制流图 ICFG
Output: 修复位置集合 fixLocs
1  fixLocs := {crashLoc};
2  repeat
3      fixLocsPrev := fixLocs;
4      foreach fixLoc ∈ fixLocsPrev, loc ∈ ICFG − FixLocsPrev do
5          if depends(loc, fixLoc) ∧ dominates(CFG, loc, crashLoc) then
6              fixLocs := fixLocs ∪ {loc};
7          end
8      end
9  until fixLocsPrev = fixLocs;
10 rFixLocs := rank(fixLocs);
11 return rFixLocs;
```

集质量要求较高，前文所述的几种崩溃缺陷很少能满足。因此，ExtractFix 提出了一种基于依赖的定位方法，如算法6.1所示。

崩溃位置一定是数据或控制依赖修复位置，否则修复位置是否改变对崩溃位置的执行毫无作用。同时，由于已有输入的执行触发崩溃，所需修复的位置一定是在所执行的语句之间。另一方面，由于过程间控制流图 *ICFG* 在实际程序中可能十分庞大，因此 ExtractFix 只选取图中被执行到语句。因此，ExtractFix 首先收集触发崩溃的过程中所执行过的语句；其次，由于依赖关系是可传递的，将崩溃位置所有依赖的语句形成的依赖闭包加入候选语句；最后，二者取交集，并按照距离崩溃位置的远近由近及远排序。

为了简化后续约束传播中的面临的挑战，在得到上述交集后，只留下其中支配 (dominate) 崩溃位置的程序位置。即程序一旦执行到崩溃位置 c，则之前必然经过了修复位置 f。例如图6.1所示，所有经过崩溃位置的路径都经过了修复位置，但是若沿着路径的 P_{other} 经过了修复位置，但是并不会经过崩溃位置。

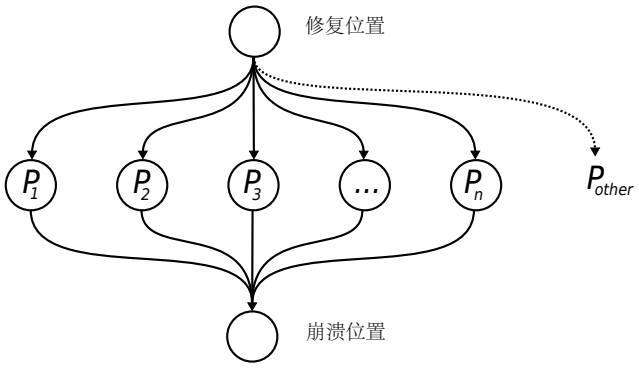

图 6.1　修复位置支配崩溃位置示例

6.3.3 约束传播

对于一个公式 φ，其最弱前条件是保证其成立并且限制最小的前条件。因此，这里考虑如何在修复位置 l 生成最弱前条件 CFC' 使得在崩溃位置 CFC 成立。Jager 等人证明了在对程序实施前向的符号的结果与最弱前条件的结果等价[179]，因此可以使用符号执行获得最弱前条件。即对于所有的确定性程序 P 和任意的期待的后条件 Q 有：$wp(P,Q)$=$fwd(P,Q)$。其中 wp 是从 P 中提取的最弱前条件以满足 Q，fwd 是在 P 上执行前向符号执行的结果，并将 Q 中使用的变量用变量的符号状态替代。

示例。考虑如下程序 P: $(x = x + x; x = x + x; x = x + x)$ 和后条件 Q: $x < 8$。保证 Q 的最弱前条件是 $wp(P,Q) = \{x < 1\}$。类似地，如果设置 x 是符号变量，并对 P 实施符号执行，可得 $8x < 8$，即 $fwd(P,Q) = \{x < 1\}$。

因此，可以使用前向符号执行来计算最弱前条件。给定一个修复位置 l、崩溃位置 c 和约束 CFC，ExtractFix 在 l 和 c 之间实施符号执行，并计算 l 上的最弱前条件 CFC'。ExtractFix 的符号执行首先按照具体输入 t 进行具体执行，直到修复位置 l。具体输入 t 可以为引发崩溃的输入，也可以为其他任意可以使程序执行到 l 的输入。在修复位置，ExtractFix 插入符号变量并启动符号执行，探索从修复位置 l 到崩溃位置 c 的全部可能的路径 Π 。

符号执行由于存在路径爆炸等局限，实施的计算代价很高。为了提升 ExtractFix 的性能，本文提出了受控的符号执行，通过限定符号化变量的个数和符号执行的范围，将实施符号执行的代价限定在可接受的范围内。

6.3.3.1 插入符号变量

在修复位置，当前已有的基于语义的修复方法将当前待修复的表达式替换为一阶或二阶的符号变量，例如 SemFix[40]、Angelix[42] 和二阶符号执行修复[96]。给定测试集 T，上述方法会在修复位置和缺陷位置之间 T 所经过的路径上收集约束，使得 T 可以全部通过。然而，ExtractFix 为了计算最弱前条件，需要覆盖 f 和 c 之间理论上所有可行的路径。为此，ExtractFix 按照如下程序变换模式引入二阶符号变量 ρ：

- 改变赋值表达式的右值：

$$x := E; \mapsto x := \rho(v_1, \ldots, v_n);$$

- 改变分支条件：

$$if(E)\{\ldots\} \mapsto if(\rho(v_1, \ldots, v_n))\{\ldots\}$$

- 添加 `if` 判断：

$$S; \mapsto if(\rho(v_1, \ldots, v_n))\{S;\}$$

- 添加 `if-return` 形式的语句：

$$: if(\rho(v_1, \ldots, v_n))\{return\ C;\}$$

其中，S 是语句，E 是表达式，C 是常量，$v_1, \ldots, v_n$ 是修复位置的活跃变量。只有在其他变换方式无法产生正确补丁时，才使用 `if-return` 变换，其中项目相关的错误代码 C 根据工具 Talos 提取。ExtractFix 同样设置 CFC 所依赖的活跃变量 V 为符号变量。ExtractFix 可以引入多个符号变量。如果修复位置的变量 v 可能影响崩溃位置的 CFC 的真值，同样将其设置为符号变量。上述引入符号变量的策略可以保证进入足够小的符号变量，同时保证修复位置和崩溃位置之间所有的路径都可以被探索到。设置了符号变量之后，就可以使用符号执行引擎探索这些路径。

6.3.3.2　符号执行区间

由于符号执行代价高昂，为了防止探索无关的路径 (例如图6.1 中的路径 P_{other})，不会经过崩溃位置的路径会被提前终止。通过控制流分析，可以判断路径是否经过某点。通过植入符号变量和提前终止无关路径，ExtractFix 可以极大地缓解了路径爆炸问题。而且，由于修复位置一般距离崩溃位置较近，符号执行中的路径数量可以被进一步限制。在实验验证中，路径爆炸问题并不十分严重，说明了上述策略的有效性。

6.3.3.3　收集约束

在使用符号执行探索路径之后，ExtractFix 对每个从 l 到 c 可能的路径 $\pi_j \in \Pi$ 收集其路径约束 pc_j。另外，沿着每个路径 π_j，所有 CFC 中使用的变量可以被符号变量表示 (即 V 和 ρ)。通过把 CFC 中的元素替换为符号变量，CFC 被重写为 CFC'_j。按这种方法，$pc_j \Rightarrow CFC'_j$ 与将 CFC 沿着路径 π_j 反向传播至修复位置的结果等价。考虑如下程序：

$$input\ x,\ i;\ \ if(i{>}0)\ y{=}x{+}1;\ else\ y{=}x{-}1;\ \ output\ y;$$

其中，假设 CFC 是 $(y > 5)$。沿着 *if-then* 分支，我们可得约束 $(i{>}0 \Rightarrow x + 1 > 5)$。

6.3.4 补丁生成

在将 CFC 传播至修复位置之后，需要生成补丁来改写修复位置的语句，并保证以下公式成立：

$$\{true\}[\rho \mapsto f]\{CFC'\}$$

ExtractFix 的补丁生成算法虽然是针对部分程序的，但是理论上也可在完整程序上工作，因为其前条件是最弱的 $true$。只需满足 $\{true\}[\rho \mapsto f]\{CFC'\}$，$\{CFC'\}$ 可以保证在任意条件下被满足。

假设 Π 是修复位置到崩溃位置之间所有可能的路径，对于每个 $\pi_j \in \Pi$，所生成的补丁 f 应在所有的输入空间下蕴含 CFC'_j。即我们需要生成函数 f，使得下面的 $\varphi_{correct}$ 被满足。

$$\varphi_{correct} \coloneqq \bigwedge_{j=1}^{|\Pi|} \left((\rho = f(V) \wedge pc_j) \Rightarrow CFC'_j \right) \tag{6.1}$$

其中，V 是 f 中使用的变量。例如，对于6.2.2小节中的例子，其 $\varphi_{correct}$ 是：

$$\varphi_{correct} = (\rho = f(size, i) \wedge \neg\rho \wedge i < size) \Rightarrow size \leq i * 2$$

因此，在给定约束 $\varphi_{correct}$ 为规约以及当前修复位置的上下文的情况下，我们可以通过求解程序估计问题来生成补丁。

6.4 实验验证

为了验证将基于约束的修复归约为程序估计问题是否有效，本节针对 ExtractFix 进行验证。主要回答以下研究问题：

- **问题 1**: ExtractFix 的修复效果如何？
- **问题 2**：ExtractFix 的修复是否高效？

6.4.1 ExtractFix 的实现

ExtractFix 的输入为一个可以导致崩溃的缺陷程序以及一个可以触发其崩溃的输入，其主要包括四部分：约束提取模块、缺陷定位模块、约束传播引擎以及补丁生成模块。其整体概览如图6.2所示。其中，灰色矩形表示其四个主要模块。

约束提取模块。约束提取模块的输入为缺陷程序以及可以导致其崩溃的输入，输出为崩溃位置和 CFC。约束提取模块是在以下几个 Sanitizer 基础上实现的：针对内存上溢/下溢错误的 Lowfat[180,181] 以及针对整数溢出、*memcpy* 未定义行为、空指针解引用和

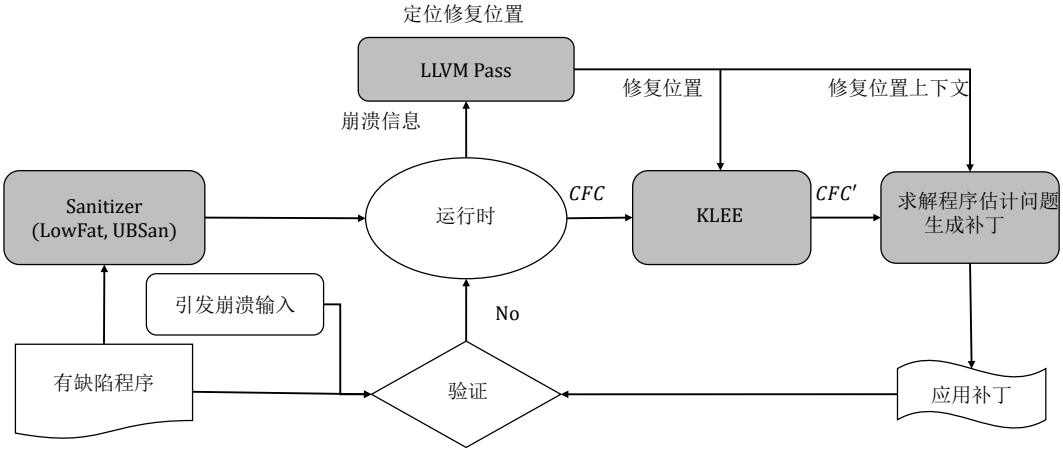

图 6.2　ExtractFix 整体概览

除零异常等错误的 UBSAN[①]。当检测到崩溃时，约束提取模块通过将 Sanitizer 报告的具体值替换为程序中的变量等代码元素，并应用相应错误类型的模板，得到符号化的 CFC。该过程需要程序以调试模式编译 (即开启 gcc 和 clang 的 `-g` 选项)。

缺陷定位模块。缺陷定位模块输入为缺陷程序和崩溃位置，返回修复位置列表。缺陷定位模块的控制和数据依赖的分析以静态分析方法实现。该部分是在 LLVM Pass [②]基础上实现的，因为 LLVM 提供了控制和数据依赖分析工具。

约束传播引擎。约束传播引擎是在 KLEE[182] 的基础上实现的。为了提取最弱前条件，ExtractFix 修改了 KLEE 的路径探索策略。首先，只在修复位置和缺陷位置之间实施符号执行。其次，终止探索无法通过崩溃位置的路径。

补丁生成模块。补丁生成模块的输入为修复位置上下文以及规约，输出为通过解程序估计问题给出的补丁序列。修复位置的上下文为修复位置上原有的语句或表达式，规约为反向传播到修复位置的约束 CFC'。ExtractFix 以二阶符号执行引擎[96] 生成程序，解决程序估计问题。

6.4.2　实验设置

本节使用两个数据集验证 ExtractFix：ManyBugs[183] 和本文从开源项目中收集的缺陷集。ManyBugs 是被广泛使用的 C 语言的缺陷集，例如已有工作 GenProg[20]、Prophet[36] 和 Angelix[42] 就使用该数据集进行验证。

验证过程中，所用的缺陷和缺陷，应符合如下条件：

1. 只考虑 ExtractFix 可支持的缺陷类型，即断言被违反、内存访问越界、空指针解引用、除零异常以及整数溢出等；

[①] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[②] LLVM Pass：http://llvm.org/docs/WritingAnLLVMPass.html

表 6.2　自建数据集的软件统计信息

程序名称	缺陷个数	Loc	描述
Libtiff	11	81K	处理 TIFF 文件的库
Binutils	2	98K	字节码文件操作工具库
Libxml2	5	299K	XML 的 C 语言解析器
Libjpeg	4	58K	C 语言的操作 JPEG 文件的库
FFmpeg	2	617K	语音视频文件操作库
Jasper	2	29K	图像处理库
Coreutil	4	78K	GNU 核心功能程序库
总计	30	—	—

2. 所选用的程序应该能被 LLVM 编译，并能用 KLEE 实施符号执行。

3. 有引发崩溃的程序输入并且崩溃能被复现。

在 ManyBugs 中，跳过无法支持 KLEE 的 python 和 fbc，并忽略不支持的错误类型，最终在 Libtiff、Lighttd 和 Php 这三个项目中选择了 26 个缺陷。

而本文在开源软件的真实缺陷中，构建了另一个数据集，如表 6.2 所示。其中包括了 16 个内存访问错误、4 个整数溢出错误、5 个除零错误和 3 个 API 断言错误。在这些软件的历史提交中，可以检索到为了修复这些缺陷开发者提供的补丁。对于该自建数据集，以开发者补丁作为正确补丁。对于每个缺陷，ExtractFix 设定的修复时间上限为 1 小时。

6.4.3　问题 1：ExtractFix 的修复效果

为了回答**问题 1**，本小节首先分析 ExtractFix 的提取 CFC 的效果、定位效果以及修复效果，之后与现有方法的修复效果进行比较。由于表 6.2中收集的被测软件没有 CFC 的真言 (oracle)，本文通过人工分析代码出错原因的方式提取正确的 CFC，并与 ExtractFix 产生的进行比较。随后检查 ExtractFix 的定位与开发者修复的位置是否一直。最后检查 ExtractFix 能否生成补丁以及生成的补丁是否与开发者提供的补丁语义等价。

ExtractFix 的整体修复结果如表 6.3所示。其中前三行为在 ManyBugs 上的修复效果，其余为在自建数据集上的修复效果。*CFC* 列表示正确生成 CFC 的数量。*FL (T1/T3)* 列表示成功定位在第一 (Top 1，T1) 的个数和在前三个以内 (Top 3, T3) 的个数。对于两个数据修复的详细数据，ManyBugs 的数据见表 6.4，自建数据集的数据见表 6.5。

在表 6.4和表 6.5中，*Sanitizer* 列表示所使用的 Sanitizer 类型；模板列表示用于产生 CFC 的模板；*FL* 列表示定位情况，L-N 表示尝试第 N 个位置时达到了正确的修复位置；修复？列表示能否产生“疑似正确的”补丁，即能否生成补丁使程序能通过引发崩溃的输入；正确？列表示补丁是否真正正确；距离列表示修复位置与崩溃位置间隔的行数；时间 *(m)* 列表示修复所用的分钟数。

表 6.3　ExtractFix 在两个数据集上的整体修复结果

程序名称	缺陷数	CFC	FL (T1/T3)	生成补丁数	正确补丁数	平均时间 (m)
Libtiff*	5	3	2 / 3	3	2	4.32
Lighttd	3	2	1 / 2	2	1	7.50
Php	18	14	6 / 10	14	9	11.11
Libtiff*	11	9	7 / 8	9	6	5.64
Binutils	2	2	1 / 1	2	1	26.28
Libxml	5	4	3 / 3	4	2	13.80
Libjpeg	4	3	1 / 2	3	2	12.01
FFmpeg	2	2	1 / 1	2	2	8.23
Jasper	2	2	1 / 1	2	1	1.07
Coreutil	4	2	1 / 2	2	2	5.17
总计	56	43	24 / 33	43	28	9.46

Manybugs 和本文收集的软件都包括 Libtiff，但是两个缺陷集并不相交。

避免崩溃约束的提取。在总共 56 个缺陷中，ExtractFix 成功提取了 43 个缺陷的 CFC，经过人工检查，发现它们都是正确的。对于一些缓冲区上溢缺陷和空指针解引用缺陷无法生成 CFC，是因为涉及到多重指针，无法通过编译器提供的调试信息分析出正确的代码元素对象。该结果说明 ExtractFix 针对这几种类型的缺陷，提取约束的策略是有效的。

修复位置的定位。在分析得到 CFC 的基础上，进一步分析错误定位的性能。在 43 个缺陷中，有 24 个缺陷的定位排在第一 (T1)，有 33 个缺陷的定位排在前三 (T3)。可以看出，ExtractFix 的定位策略是有效的。

补丁生成。在 56 个缺陷中，ExtractFix 一共产生了 43 个“疑似正确”的补丁。这些修复通过改变分支条件、修改赋值语句的右值或插入 `if` 检测语句。

例如，在 Libtiff 中的缓冲区上溢缺陷 CVE-2014-8128 上，开发者的补丁是在循环开始位置插入了 `if` 检查语句：

```
+ if (nrows == 256) break;
```

ExtractFix 通过修改 `while` 语句的退出条件实现修复，与开发者的补丁语义等价：

```
- while (err >= limit)
+ while (err >= limit && nrows < 256)
```

在实验的修复过程中，一旦生成正确的约束，ExtractFix 就能产生正确的补丁。

与现有方法比较。为了衡量 ExtractFix 的修复能力，将其与现有方法 Prophet[36]、Angelix[42] 和 Fix2Fit[35] 进行比较，结果如表 6.6所示。其中，Prophet 是基于学习的修复方法，Angelix 是最新的基于语义的修复方法，Fix2Fit 是最新的防止补丁过拟合的方法。由于工具的编译配置原因，Fix2Fit 无法运行 Libjpeg、Prophet 无法运行 Binutils 和

表 6.4 ExtractFix 在 ManyBugs 上的修复的详细结果

项目名	编号	类型	**Sanitizer**	模板	**CFC**	**FL**	修复？	正确？	距离	时间 (m)
Libtiff	207c78a	ND	UBSan	T_5	✗	-	✗	—	—	—
	0a36d7f	BO	Lowfat	T_2	✓	L-1	✓	✓	4	3.44
	ee65c74	IO	UBSan	T_3	✓	L-3	✓	✗	10	5.66
	865f7b2	BO	Lowfat	T_2	✗	-	✗	—	—	—
	565eaa2	ND	UBSan	T_5	✓	L-1	✓	✓	2	3.86
Lighttd	1914	BU	Lowfat	T_2	✗	-	✗	—	—	—
	2662	AS	Assert	T_1	✓	L-3	✓	✓	9	8.09
	2786	BO	Lowfat	T_2	✓	L-1	✓	✗	7	6.91
Php	5bb0a44e06	ND	UBSan	T_5	✓	L-4	✓	✗	10	16.23
	426f31e790	AA	APISan	T_4	✓	L-1	✓	✓	2	14.31
	2a6968e43a	BO	Lowfat	T_2	✓	L-1	✓	✓	2	12.09
	8deb11c0c3	ND	UBSan	T_5	✓	L-1	✓	✗	1	10.08
	7f2937223d	ND	UBSan	T_5	✗	-	✗	—	—	—
	2adf58cfcf	ND	UBSan	T_5	✓	L-2	✓	✓	5	9.89
	3acdca4703	ND	UBSan	T_5	✓	L-1	✓	✓	5	9.68
	c2fe893985	ND	UBSan	T_5	✗	-	✗	—	—	—
	93f65cdeac	ND	UBSan	T_5	✗	-	✗	—	—	—
	8d520d6296	ND	UBSan	T_5	✓	L-1	✓	✓	1	7.89
	cacf363957	AA	APISan	T_4	✓	F	✓	✗	2	8.96
	c1e510aea8	ND	UBSan	T_5	✓	L-2	✓	✓	4	10.23
	f330c8ab4e	ND	UBSan	T_5	✓	L-5	✓	✓	32	10.24
	1d6c98a136	ND	UBSan	T_5	✓	F	✓	✗	2	30.02
	acaf9c5227	ND	UBSan	T_5	✓	L-1	✓	✓	7	5.89
	032bbc3164	BO	Lowfat	T_2	✓	L-2	✓	✓	47	4.30
	1923ecfe25	AA	APISan	T_4	✗	-	✗	—	—	—
	cfa9c90b20	ND	UBSan	T_5	✓	L-1	✓	✗	1	5.66
总计	26	—	—	–	19	—	19	12	(avg) 8.1	(avg) 9.6

BO：缓冲区上溢 (buffer overflow)；*BU*：缓冲区下溢 (buffer underflow)；*IO*：整数溢出 (integer overflow)；*DZ*：除零 (divide-by-zero)；*AA*：AIP 断言 (API assert)；*ND*：空指针解引用 (null pointer dereference)；*AS*：开发者断言 (developer assertion)。

FFmpeg。在不同项目各自的比较上，ExtractFix 除了在 Coreutil 上生成补丁数略少于 Fix2Fit，其在其他所有的软件上都生成了最多或者并列最多的“疑似正确”补丁。考虑到 Fix2Fit 是基于模糊测试的技术，其时间上限为 12 小时，远超 ExtractFix 所用时间，可以认为 ExtractFix 在生成补丁上表现更好。另外，对于生成“疑似正确的”补丁的总数，ExtractFix 超过了其他三个方法。对于正确补丁，ExtractFix 不但在分项目还是总数上，都超越了其他三个方法，其产生 28 个正确修复远超第二位 Fix2Fit 的 11 个。因此，可以认为 ExtractFix 不但可以生成更多的补丁，也可以生成质量更高的补丁。

小结。通过对 ExtractFix 三个修复步骤的效果分析以及和已有方法的比较，可以认为 ExtractFix 的修复方法是有效的，并且 ExtractFix 能够缓解补丁过拟合问题。

表 6.5　ExtractFix 在自建数据集上的修复的详细结果

项目名	编号	类型	Sanitizer	模板	CFC	FL	修复?	正确?	距离	时间 (m)
Libtiff	CVE-2016-5321	BO	Lowfat	T_2	✓	L-1	✓	✓	2	1.68
	CVE-2014-8128	BO	Lowfat	T_2	✓	L-1	✓	✓	5	2.40
	CVE-2016-5314	BO	Lowfat	T_2	✗	-	✗	—	—	—
	Bugzilla 2633	BO	Lowfat	T_2	✓	L-5	✓	✗	12	4.03
	CVE-2016-10094	BO	Lowfat	T_2	✓	L-2	✓	✗	2	1.87
	CVE-2016-3186	AA	APISan	T_4	✓	L-1	✓	✓	2	32
	CVE-2017-7601	IO	UBSan	T_3	✓	L-1	✓	✗	3	2.38
	CVE-2016-9273	BO	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2016-3623	DZ	UBSan	T_6	✓	L-1	✓	✓	2	2.05
	CVE-2017-7595	DZ	UBSan	T_6	✓	L-1	✓	✓	2	2.20
	Bugzilla 2611	DZ	UBSan	T_6	✓	L-1	✓	✓	1	2.13
Binutils	CVE-2018-10372	BO	Lowfat	T_2	✓	F	✓	✗	2	16.57
	CVE-2017-15025	DZ	UBSan	T_6	✓	L-1	✓	✓	2	36.00
Libxml2	CVE-2016-1834	IO	UBSan	T_3	✓	F	✓	✗	12	5.97
	CVE-2016-1839	BU	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2016-1838	BO	Lowfat	T_2	✓	L-1	✓	✗	3	4.12
	CVE-2012-5134	BU	Lowfat	T_2	✓	L-1	✓	✓	2	40.83
	CVE-2017-5969	ND	UBSan	T_5	✓	L-1	✓	✓	2	4.30
Libjpeg	CVE-2018-14498	BO	Lowfat	T_2	✓	L-10	✓	✗	3	1.22
	CVE-2018-19664	BO	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2017-15232	ND	UBSan	T_5	✓	L-1	✓	✓	2	1.37
	CVE-2012-2806	BO	Lowfat	T_2	✓	L-3	✓	✓	10	33.26
FFmpeg	CVE-2017-9992	BO	Lowfat	T_2	✓	L-4	✓	✓	7	9.27
	Bugzilla-1404	IO	UBSan	T_3	✓	L-1	✓	✓	3	7.20
Jasper	CVE-2016-8691	DZ	UBSan	T_6	✓	L-1	✓	✓	5	1.08
	CVE-2016-9387	IO	UBSan	T_3	✓	F	✓	✗	5	1.05
Coreutil	Bugzilla-26545	AA	APISan	T_4	✓	L-3	✓	✓	4	6.03
	Bugzilla-25003	AA	APISan	T_4	✓	L-1	✓	✓	2	4.30
	GNUBug-25023	BO	Lowfat	T_2	✗	-	✗	—	—	—
	GNUBug-19784	BO	Lowfat	T_2	✗	-	✗	—	—	—
总计	30	—	—	—	24	—	24	16	(avg)4.0	(avg)9.3

BO：缓冲区上溢 (buffer overflow)；*BU*：缓冲区下溢 (buffer underflow)；*IO*：整数溢出 (integer overflow)；*DZ*：除零 (divide-by-zero)；*AA*：AIP 断言 (API assert)；*ND*：空指针解引用 (null pointer dereference)；*AS*：开发者断言 (developer assertion)。

6.4.4　问题 2：ExtractFix 的修复效率

由于 ExtractFix 中引入了符号执行，而符号执行的可扩展性被认为是较差的。通过回答**问题 2**，可以判断 ExtractFix 可扩展性如何。在实验中，本文自建的数据集是由真实开源项目组成的，也包含大型程序。例如 FFmpeg 的代码行数达到了 61.7 万行。在表 6.3中给出了 ExtractFix 的修复时间，可以看出修复平均仅用时 9.46 分钟，最多用时 41 分钟。这说明 ExtractFix 方法是高效的，原因主要有：(1) 受限的符号执行降低了符号执行的代价；(2) 基于程序估计问题的求解产生的补丁是高效的。因此，可以认为 ExtractFix 有较高的修复效率，能够缓解基于约束的修复的效率问题。

表 6.6　ExtractFix 与 Prophet、Angelix 和 Fix2Fit 的修复效果比较

程序名称	缺陷数	补丁总数				正确补丁数			
		Prophet	Angelix	Fix2Fit	ExtractFix	Prophet	Angelix	Fix2Fit	ExtractFix
Libtiff	5	2	**3**	**3**	**3**	1	1	1	**2**
Lighttd	3	**2**	**2**	**2**	**2**	0	0	0	**1**
Php	18	10	7	9	**14**	6	4	6	**9**
Libtiff	11	7	7	7	**9**	1	0	1	**6**
Binutils	2	-	-	1	**2**	-	-	0	**1**
Libxml2	5	3	0	**4**	**4**	0	0	1	**2**
Libjpeg	4	**3**	-	-	**3**	1	-	-	**2**
FFmpeg	2	-	-	**2**	**2**	-	-	1	**2**
Jasper	2	**2**	**2**	**2**	**2**	0	0	0	**1**
Coreutil	4	2	-	**3**	2	0	-	1	**2**
总计	56	31	21	33	**43**	9	5	11	**28**

前三行是在 ManyBugs 数据集上的修复结果，其余是在自建数据集上的修复结果。其中，产生最多补丁和最多正确补丁的方法数据被加粗。

6.5　小结

基于测试的修复对测试集的质量需求较高，然而现实中经常有缺陷缺乏足够的测试的情况。此时基于测试的修复技术的能力受到了限制。其原因主要有：(1) 缺陷定位依赖于测试；(2) 验证依赖于测试。在测试不足的情况下，基于测试的修复更容易产生过拟合的补丁。本章提出一种将基于约束的修复归约为程序估计问题并解决的方法 ExtractFix 。ExtractFix 主要解决了以下四个子问题：首先，针对特定类型的缺陷，基于崩溃信息和模板提取约束；随后，根据静态分析与动态执行信息进行错误定位；再次，使用最弱前条件将约束反向传播至修复位置；最后，以修复位置的程序为上下文，以传播来的约束为规约，通过求解程序估计问题得到补丁。

在实验验证中，ExtractFix 正确修复了 56 个缺陷中的 28 个，修复效果显著超越现有基于测试的修复方法。另外，ExtractFix 的也达到了较高的修复效率，在大型软件上最多修复时间为 41 分钟，平均仅用时 9.46 分钟。ExtractFix 证实了在程序估计问题的框架下解决基于约束的修复的可行性，并且能缓解现有修复技术的补丁过拟合问题和验证效率问题。

第七章　结论及展望

在软件大规模应用的今天，软件缺陷不可避免。然而，软件缺陷的调试与修复十分消耗时间和人力资源。为此，自动化地修复软件缺陷对于降低开发成本、增强软件安全性有着重要的意义。

自 2009 年 GenProg 被提出以来，该研究方向经过了十余年的发展。研究者提出各种修复方法，然而目前该技术与大规模实际应用仍有很大的距离。该方法普遍面临着补丁过拟合问题和修复效率不高问题。这两大问题阻碍了修复技术的实际应用。在这种背景下，本文提出了程序估计问题，并研究如何将具体的修复问题归约为该问题。本文还针对具体修复方法研究如何提升效率。

本章对全文进行总结，并展望未来工作。

7.1　本文工作总结

为了缓解缺陷自动修复中的补丁过拟合和修复效率不高两大挑战，本文围绕程序估计问题展开一系列研究，主要包括：

1. 提出了解决程序估计问题的玲珑框架；
2. 提出了将基于测试的缺陷修复归约为程序估计问题的方法；
3. 提出了通过合并状态同余补丁方法，消除以测试补丁验证中冗余计算；
4. 提出了将基于约束的缺陷修复归约为程序估计问题的方法。

程序估计问题的提出与解决。本文首先定义了程序估计问题，即在给定上下文内搜索满足给定规约且概率最大的程序。为了解决程序估计问题，本文提出玲珑框架。玲珑框架解决了定义搜索空间，如何在空间中搜索，如何赋以程序概率以及如何尽早过滤非法程序等问题。具体来说，玲珑框架主要包含如下工作：

- 扩展了上下文无关文法，提出了扩展规则，并在此基础上提出了 AST 的泛化版本扩展树。此外，本文还探索了扩展规则的性质，提出 AST 和扩展树相互转化的方法。
- 通过将程序视为顶点，将扩展规则视为边，玲珑框架将程序生成问题转化为程序空间中的路径查找问题。
- 提供计算程序概率的方法。
- 提供静态分析方法对非法程序剪枝。

前三个工作可以使得玲珑框架能够合理地定义和搜索程序空间，从而实现在给定上下文中找到概率高且满足规约的程序，进而缓解补丁过拟合问题。最后一个工作可

以加快搜索过程，有助于缓解修复效率不高的问题。

基于测试的修复归约为程序估计问题。本文提出针对 Java 条件表达式实例化玲珑框架的方法，并提出基于测试的修复方法 Hanabi ，并将修复问题规约为程序估计问题。本文实例化玲珑框架，依次设计了文法规则、扩展规则、路径搜索算法、统计学习模型以及静态分析的约束。本文针对条件表达式提供了一系列选项，并通过系统的对照试验探索不同配置的性能。之后，Hanabi 按照基于测试的修复的典型做法进行使用基于频谱的方法进行缺陷定位并使用运行测试集进行补丁验证。而补丁生成则由程序估计问题求解而得，即以修复位置的周围代码为上下文、以 Java 类型和程序规模为规约启动实例化的玲珑框架来生成条件表达式。实验结果表明，Hanabi 修复效果超越现有基于测试的修复方法，提升了修复能力并缓解了补丁过拟合问题。这说明将基于测试的修复归约为程序估计问题求解是可行的。

基于状态同余的补丁验证加速。本文提出基于状态同余的补丁验证加速方法 AccMut 。由于在实际项目中软件较为复杂，难以提取针对补丁的规约，因此往往仍然需要使用测试对补丁进行验证。基于测试的修复的性能瓶颈在补丁验证阶段。其中，在编译补丁和执行测试的过程中存在大量冗余计算。AccMut 通过分析补丁之间的状态同余关系，将其划分为等价类，并使同一类别内的补丁共享计算。实验结果表明，AccMut 的加速效果超越现有方法，缓解补丁验证效率过低的问题。

基于约束的修复归约为程序估计问题。本文提出了在程序估计问题框架下解决基于约束的修复问题的方法 ExtractFix 。针对一些类型的崩溃缺陷，ExtractFix 使用基于模板的约束提取方法。针对高质量的测试集不可访问的情形，ExtractFix 基于依赖分析和错误执行的覆盖情况确定修复位置。随后，ExtractFix 使用受限的符号执行技术，高效地将约束传播至修复位置。最终，在以修复位置的代码为上下文、以约束为规约，ExtractFix 可以将修复问题归约为程序估计问题求解。实验结果表明，ExtractFix 的修复能力超越现有基于测试的修复方法，缓解补丁过拟合问题和验证效率不足的问题。这说明将基于约束的修复归约为程序估计问题求解是可行的。

7.2 未来工作展望

本文方法的进一步提升。如前文所示，本文方法的实验结果证实了相关方法的有效性，然而目前缺陷修复问题所面临的挑战依然存在，距离实用化依然有一定距离。

本文的方法也需要进一步扩展提升，主要包括以下几个方面。

1. 对于依靠程序估计问题的解决依然存在一些挑战，例如对于小样本程序如何合适地估计其概率。这需要对如何表示程序以及如何学习概率展开研究。

2. 基于测试的修复过程中，测试验证阶段依然存在很多冗余的计算有待消除。例如，当前 AccMut 主要针对单行代码上的补丁进行优化，如果能扩展分析范围，则能发现更多可共享的计算。
3. 基于约束的修复中，针对一般性缺陷提取约束是一项挑战。已有方法主要依赖于测试等程序的正确执行信息来收集约束，如果能结合更多信息获得更精准的约束，势必能够提升基于约束的修复方法的能力。

将新的问题归约为程序估计问题。本文展示了如何将程序修复问题归约为程序估计问题。然而，程序估计问题作为一般性问题，可以用于解决其他方面的问题。凡是在需要生成程序并估计其概率的场景，都可以尝试归约为程序估计问题。例如，基于样例编程 (programming by example，PBE) 问题可以较为直接地归约为程序估计问题。

将本文的加速方法用于提升其他问题的效率。本文提出一些提升修复效率的方法，这些方法不局限在缺陷修复问题的场景上。AccMut 可以直接用于加速变异分析，而其包含的程序多个变体共享计算的思路，可以用于加速软件产品线测试等相关问题。ExtractFix 中的受限的符号执行也可以应用在测试生成等需要实施符号执行的问题上，以提高相关方法的可延展性。

参考文献

[1] Yingfei Xiong, Jie Wang, Runfa Yan *et al.* "*Precise condition synthesis for program repair*". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. **2017**: 416–426.

[2] Ming Wen, Junjie Chen, Rongxin Wu *et al.* "*Context-aware patch generation for better automated program repair*". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. **2018**: 1–11.

[3] 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**.

[4] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, **2005**.

[5] ISDW Group *et al.* "*1044-2009-IEEE Standard Classification for Software Anomalies*". *IEEE, New York*, **2010**.

[6] Jifeng Xuan, He Jiang, Zhilei Ren *et al.* "*Developer prioritization in bug repositories*". In: *2012 34th International Conference on Software Engineering (ICSE)*. **2012**: 25–35.

[7] John Anvik, Lyndon Hiew and Gail C Murphy. "*Who should fix this bug?*" In: *Proceedings of the 28th international conference on Software engineering*. **2006**: 361–370.

[8] 李斌，贺也平，马恒太。"程序自动修复: 关键问题及技术" [J]。软件学报，**2019**，*30*(2)：244–265。

[9] Martin Monperrus. "*Automatic software repair: a bibliography*". *ACM Computing Surveys (CSUR)*, **2018**, *51*(1): 1–24.

[10] Luca Gazzola, Daniela Micucci and Leonardo Mariani. "*Automatic software repair: A survey*". *IEEE Transactions on Software Engineering*, **2017**, *45*(1): 34–67.

[11] 玄跻峰，任志磊，王子元 等。"自动程序修复方法研究进展" [J]。软件学报，**2016**，*27*(4)：771–784。

[12] 王赞，郜健，陈翔 等。"自动程序修复方法研究述评" [J]。计算机学报，**2018**，*41*(3)：588–610。

[13] Fan Long and Martin Rinard. "*An analysis of the search spaces for generate and validate patch generation systems*". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. **2016**: 702–713.

[14] Johannes Bader, Andrew Scott, Michael Pradel *et al.* "*Getafix: Learning to fix bugs automatically*". *Proceedings of the ACM on Programming Languages*, **2019**, *3*(OOPSLA): 1–27.

[15] Alexandru Marginean, Johannes Bader, Satish Chandra *et al.* "*Sapfix: Automated end-to-end repair at scale*". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. **2019**: 269–278.

[16] Manish Motwani, Sandhya Sankaranarayanan, René Just *et al.* "*Do automated program repair techniques repair hard and important bugs?*" *Empirical Software Engineering*, **2018**, *23*(5): 2901–2947.

[17] 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 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. **2019**: 1–12.

[18] 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: *2019 12th IEEE conference on software testing, validation and verification (ICST)*. **2019**: 102–113.

[19] Yiling Lou, Ali Ghanbari, Xia Li *et al.* "*Can automated program repair refine fault localization? a unified debugging approach*". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. **2020**: 75–87.

[20] Westley Weimer, ThanhVu Nguyen, Claire Le Goues *et al.* "*Automatically finding patches using genetic programming*". In: *2009 IEEE 31st International Conference on Software Engineering*. **2009**: 364–374.

[21] W. Weimer, Z.P. Fry and S. Forrest. "*Leveraging program equivalence for adaptive program repair: Models and first results*". In: *ASE*. **2013**: 356–366.

[22] Yuhua Qi, Xiaoguang Mao, Yan Lei *et al.* "*The Strength of Random Search on Automated Program Repair*". In: *ICSE*. **2014**: 254–265.

[23] Yuhua Qi, Xiaoguang Mao and Yan Lei. "*Efficient automated program repair through fault-recorded testing prioritization*". In: *2013 IEEE International Conference on Software Maintenance*. **2013**: 180–189.

[24] Edward K Smith, Earl T Barr, Claire Le Goues *et al.* "*Is the cure worse than the disease? overfitting in automated program repair*". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. **2015**: 532–543.

[25] 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.

[26] Xuan Bach D Le, Ferdian Thung, David Lo *et al.* "*Overfitting in semantics-based automated program repair*". *Empirical Software Engineering*, **2018**, *23*(5): 3007–3033.

[27] Xuan-Bach D Le, David Lo and Claire Le Goues. "*Empirical study on synthesis engines for semantics-based program repair*". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. **2016**: 423–427.

[28] Qi Xin and Steven P Reiss. "*Identifying test-suite-overfitted patches through test case generation.*" In: *ISSTA*. **2017**: 226–236.

[29] 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*. **2017**: 831–841.

[30] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller *et al.* "*Semantic program repair using a reference implementation*". In: *Proceedings of the 40th International Conference on Software Engineering*. **2018**: 129–139.

[31] 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*. **2018**: 789–799.

[32] 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*. **2009**: 947–954.

[33] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang *et al.* "*Shaping program repair space with existing patches and similar code*". In: *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. **2018**: 298–309.

[34] Jiajun Jiang, Luyao Ren, Yingfei Xiong *et al.* "*Inferring program transformations from singular examples via big code*". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2019**: 255–266.

[35] Xiang Gao, Sergey Mechtaev and Abhik Roychoudhury. "*Crash-avoiding program repair*". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. **2019**: 8–18.

[36] 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.

[37] Ethan Fast, Claire Le Goues, Stephanie Forrest *et al.* "*Designing better fitness functions for automated program repair*". In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. **2010**: 965–972.

[38] Liushan Chen, Yu Pei and Carlo A Furia. "*Contract-based program repair without the contracts*". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2017**: 637–647.

[39] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan *et al.* "*Test-equivalence analysis for automatic patch generation*". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **2018**, *27*(4): 1–37.

[40] 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.

[41] Sergey Mechtaev, Jooyong Yi and Abhik Roychoudhury. "*Directfix: Looking for simple program repairs*". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. **2015**: 448–458.

[42] 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.

[43] Jifeng Xuan, Matias Martinez, Favio Demarco *et al.* "*Nopol: Automatic repair of conditional statement bugs in java programs*". *IEEE Transactions on Software Engineering*, **2016**, *43*(1): 34–55.

[44] Victor Sobreira, Thomas Durieux, Fernanda Madeiral *et al.* "*Dissection of a bug dataset: Anatomy of 395 patches from defects4j*". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. **2018**: 130–140.

[45] 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.

[46] Fan Long and Martin Rinard. "*Staged program repair with condition synthesis*". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. **2015**: 166–178.

[47] 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*. **2005**: 273–282.

[48] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo *et al.* "*A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization*". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **2013**, *22*(4): 1–40.

[49] Ripon Saha, Yingjun Lyu, Wing Lam *et al.* "*Bugs. jar: a large-scale, diverse dataset of real-world java bugs*". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. **2018**: 10–13.

[50] 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*. **2011**: 26–36.

[51] Hao Zhong and Zhendong Su. "*An empirical study on real bug fixes*". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. **2015**: 913–923.

[52] Jiajun Jiang, Yingfei Xiong and Xin Xia. "*A manual inspection of defects4j bugs and its implications for automatic program repair*". *Science China Information Sciences*, **2019**, *62*(10): 200102.

[53] Jingjing Liang, Yaozong Hou, Shurui Zhou *et al.* "*How to Explain a Patch: An Empirical Study of Patch Explanations in Open Source Projects*". In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. **2019**: 58–69.

[54] Claire Le Goues, Michael Pradel and Abhik Roychoudhury. "*Automated program repair*". *Communications of the ACM*, **2019**, *62*(12): 56–65.

[55] 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.

[56] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest *et al.* "*A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each*". In: *ICSE*. **2012**: 3–13.

[57] Claire Le Goues, Stephanie Forrest and Westley Weimer. "*Current challenges in automatic software repair*". *Software quality journal*, **2013**, *21*(3): 421–443.

[58] 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.

[59] Matias Martinez and Martin Monperrus. "*Astor: A program repair library for java*". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. **2016**: 441–444.

[60] Matias Martinez and Martin Monperrus. "*Astor: Exploring the design space of generate-and-validate program repair beyond GenProg*". *Journal of Systems and Software*, **2019**, *151*: 65–80.

[61] Yuan Yuan and Wolfgang Banzhaf. "*ARJA: Automated repair of java programs via multi-objective genetic programming*". *IEEE Transactions on Software Engineering*, **2018**.

[62] Yuan Yuan and Wolfgang Banzhaf. "*Toward Better Evolutionary Program Repair: An Integrated Approach*". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **2020**, *29*(1): 1–53.

[63] Xiao Liang Yu, Omar Al-Bataineh, David Lo *et al.* "*Smart Contract Repair*". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **2020**, *29*(4): 1–32.

[64] 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.

[65] 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*. **2018**: 12–23.

[66] Jinru Hua, Mengshi Zhang, Kaiyuan Wang *et al.* "*Sketchfix: A tool for automated program repair approach using lazy candidate generation*". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2018**: 888–891.

[67] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé *et al.* "*iFixR: Bug report driven program repair*". In: *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. **2019**: 314–325.

[68] 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.

[69] Moumita Asad, Kishan Kumar Ganguly and Kazi Sakib. "*Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair*". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. **2019**: 328–332.

[70] Seemanta Saha *et al.* "*Harnessing evolution for multi-hunk program repair*". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. **2019**: 13–24.

[71] Jindae Kim, Jeongho Kim, Eunseok Lee *et al.* "*The effectiveness of context-based change application on automatic program repair*". *Empirical Software Engineering*, **2020**, *25*(1): 719–754.

[72] Vidroha Debroy and W Eric Wong. "*Using mutation to automatically suggest fixes for faulty programs*". In: *2010 Third International Conference on Software Testing, Verification and Validation*. **2010**: 65–74.

[73] Dongsun Kim, Jaechang Nam, Jaewoo Song *et al.* "*Automatic patch generation learned from human-written patches*". In: *2013 35th International Conference on Software Engineering (ICSE)*. **2013**: 802–811.

[74] Martin Monperrus. "*A critical review of" automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair*". In: *Proceedings of the 36th International Conference on Software Engineering*. **2014**: 234–242.

[75] 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*. **2016**: 727–738.

[76] Reudismam Rolim, Gustavo Soares, Loris D'Antoni *et al.* "*Learning syntactic program transformations from examples*". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. **2017**: 404–415.

[77] Xuan-Bach D Le, Duc-Hiep Chu, David Lo *et al.* "*S3: syntax-and semantic-guided repair synthesis via programming by examples*". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. **2017**: 593–604.

[78] 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*. **2019**: 19–30.

[79] Kui Liu, Anil Koyuncu, Kisub Kim *et al.* "*LSRepair: Live search of fix ingredients for automated program repair*". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. **2018**: 658–662.

[80] 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*. **2019**: 31–42.

[81] 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: *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. **2020**.

[82] Kui Liu, Anil Koyuncu, Dongsun Kim *et al.* "*Avatar: Fixing semantic bugs with fix patterns of static analysis violations*". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. **2019**: 1–12.

[83] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé *et al.* "*Fixminer: Mining relevant fix patterns for automated program repair*". *Empirical Software Engineering*, **2020**: 1–45.

[84] 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.

[85] Mauricio Soto and Claire Le Goues. "*Using a probabilistic model to predict bug fixes*". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. **2018**: 221–231.

[86] 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.

[87] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta *et al.* "*Experience Report: How Effective is Automated Program Repair for Industrial Software?*" In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. **2020**: 612–616.

[88] Rahul Gupta, Soham Pal, Aditya Kanade *et al.* "*DeepFix: Fixing Common C Language Errors by Deep Learning.*" In: *Aaai*. **2017**: 1345–1351.

[89] 周风顺，王林章，李宣东。"*C/C++ 程序缺陷自动修复与确认方法*" [J]。软件学报，**2017**，*30*(5)：1243–1255。

[90] 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.

[91] Michele Tufano, Cody Watson, Gabriele Bavota *et al.* "*An empirical study on learning bug-fixing patches in the wild via neural machine translation*". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **2019**, *28*(4): 1–29.

[92] Yi Li, Shaohua Wang and Tien N Nguyen. "*DLFix: Context-based Code Transformation Learning for Automated Program Repair*". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. **2020**.

[93] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang *et al.* "*CoCoNuT: combining context-aware neural translation models using ensemble for program repair*". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. **2020**: 101–114.

[94] Susmit Jha, Sumit Gulwani, Sanjit A Seshia *et al.* "*Oracle-guided component-based program synthesis*". In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. **2010**: 215–224.

[95] Y. Ke, K. T. Stolee, C. L. Goues *et al.* "*Repairing Programs with Semantic Code Search (T)*". In: *ASE*. **2015**: 295–306.

[96] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti *et al.* "*Symbolic execution with existential second-order constraints*". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2018**: 389–399.

[97] Xuan-Bach D Le, Duc-Hiep Chu, David Lo *et al.* "*JFIX: semantics-based repair of Java programs via symbolic PathFinder*". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. **2017**: 376–379.

[98] Rajeev Alur, Rastislav Bodik, Garvit Juniwal *et al.* "*Syntax-guided synthesis*". In: *2013 Formal Methods in Computer-Aided Design*. **2013**: 1–8.

[99] Andrew Reynolds, Morgan Deters, Viktor Kuncak *et al.* "*Counterexample-guided quantifier instantiation for synthesis in SMT*". In: *International Conference on Computer Aided Verification*. **2015**: 198–216.

[100] Dennis Jeffrey, Neelam Gupta and Rajiv Gupta. "*Fault localization using value replacement*". In: *Proceedings of the 2008 international symposium on Software testing and analysis*. **2008**: 167–178.

[101] Qing Gao, Yingfei Xiong, Yaqing Mi *et al.* "*Safe memory-leak fixing for c programs*". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. **2015**: 459–470.

[102] Junhee Lee, Seongjoon Hong and Hakjoo Oh. "*Memfix: static analysis-based repair of memory deallocation errors for c*". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2018**: 95–106.

[103] Rijnard van Tonder and Claire Le Goues. "*Static automated program repair for heap properties*". In: *Proceedings of the 40th International Conference on Software Engineering*. **2018**: 151–162.

[104] Xuezheng Xu, Yulei Sui, Hua Yan *et al.* "*VFix: value-flow-guided precise program repair for null pointer dereferences*". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. **2019**: 512–523.

[105] Zhen Huang, David Lie, Gang Tan *et al.* "*Using safety properties to generate vulnerability patches*". In: *2019 IEEE Symposium on Security and Privacy (SP)*. **2019**: 539–554.

[106] 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.

[107] Jierui Liu, Tianyong Wu, Jun Yan *et al.* "*Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation*". In: *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. **2016**: 342–352.

[108] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux *et al.* "*Range fixes: Interactive error resolution for software configuration*". *Ieee transactions on software engineering*, **2014**, *41*(6): 603–619.

[109] Aaron Weiss, Arjun Guha and Yuriy Brun. "*Tortoise: Interactive system configuration repair*". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2017**: 625–636.

[110] 蒋炎岩，许畅，马晓星 等。"获取访存依赖: 并发程序动态分析基础技术综述" [J]。软件学报，**2017**，*28*(4)：747–763。

[111] Guoliang Jin, Linhai Song, Wei Zhang *et al.* "*Automated atomicity-violation fixing*". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. **2011**: 389–400.

[112] Peng Liu and Charles Zhang. "*Axis: Automatically fixing atomicity violations through solving control constraints*". In: *2012 34th International Conference on Software Engineering (ICSE)*. **2012**: 299–309.

[113] Peng Liu, Omer Tripp and Charles Zhang. "*Grail: context-aware fixing of concurrency bugs*". In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. **2014**: 318–329.

[114] Haopeng Liu, Yuxi Chen and Shan Lu. "*Understanding and generating high quality patches for concurrency bugs*". In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. **2016**: 715–726.

[115] Yan Cai and Lingwei Cao. "*Fixing deadlocks via lock pre-acquisitions*". In: *Proceedings of the 38th international conference on software engineering*. **2016**: 1109–1120.

[116] Guoliang Jin, Wei Zhang and Dongdong Deng. "*Automated concurrency-bug fixing*". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. **2012**: 221–236.

[117] Zhongxing Yu, Matias Martinez, Benjamin Danglot *et al.* "*Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system*". *Empirical Software Engineering*, **2019**, *24*(1): 33–67.

[118] Yang Hu, Umair Z Ahmed, Sergey Mechtaev *et al.* "*Re-factoring based program repair applied to programming assignments*". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2019**: 388–398.

[119] René Just, Michael D Ernst and Gordon Fraser. "*Efficient mutation analysis by propagating and partitioning infected execution states*". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. **2014**: 315–326.

[120] Ben Mehne, Hiroaki Yoshida, Mukul R Prasad *et al.* "*Accelerating search-based program repair*". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. **2018**: 227–238.

[121] Roland H Untch, A Jefferson Offutt and Mary Jean Harrold. "*Mutation analysis using mutant schemata*". In: *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*. **1993**: 139–148.

[122] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh *et al.* "*Program synthesis*". *Foundations and Trends in Programming Languages*, **2017**, *4*(1-2): 1–119.

[123] 王博，卢思睿，姜佳君 等。"基于动态分析的软件不变量综合技术" [J]。软件学报，**2020**，*31*(6)：1681–1702。

[124] Zohar Manna and Richard Waldinger. "*A deductive approach to program synthesis*". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **1980**, *2*(1): 90–121.

[125] Moshe Y Vardi and Pierre Wolper. "*Automata-theoretic techniques for modal logics of programs*". *Journal of Computer and System Sciences*, **1986**, *32*(2): 183–221.

[126] J Richard Buchi and Lawrence H Landweber. "*Solving sequential conditions by finite-state strategies*". In: *The Collected Works of J. Richard Büchi*. Springer, **1990**: 525–541.

[127] Zohar Manna and Richard J Waldinger. "*Toward automatic program synthesis*". *Communications of the ACM*, **1971**, *14*(3): 151–165.

[128] Zohar Manna and Richard Waldinger. "*Fundamentals of deductive program synthesis*". In: *Logic, Algebra, and Computation*. Springer, **1991**: 41–107.

[129] Emanuel Kitzelmann. "*Inductive programming: A survey of program synthesis techniques*". In: *International workshop on approaches and applications of inductive programming*. **2009**: 50–73.

[130] Sumit Gulwani. "*Automating string processing in spreadsheets using input-output examples*". *ACM Sigplan Notices*, **2011**, *46*(1): 317–330.

[131] Rajeev Alur, Pavol Černỳ and Arjun Radhakrishna. "*Synthesis through unification*". In: *International Conference on Computer Aided Verification*. **2015**: 163–179.

[132] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh *et al.* "*TRANSIT: specifying protocols with concolic snippets*". *ACM SIGPLAN Notices*, **2013**, *48*(6): 287–296.

[133] Sumit Gulwani, Vijay Anand Korthikanti and Ashish Tiwari. "*Synthesizing geometry constructions*". *ACM SIGPLAN Notices*, **2011**, *46*(6): 50–61.

[134] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik *et al.* "*Scaling up superoptimization*". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. **2016**: 297–310.

[135] Rajeev Alur, Arjun Radhakrishna and Abhishek Udupa. "*Scaling enumerative program synthesis via divide and conquer*". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. **2017**: 319–336.

[136] Aditya Menon, Omer Tamuz, Sumit Gulwani *et al.* "*A machine learning framework for programming by example*". In: *International Conference on Machine Learning*. **2013**: 187–195.

[137] Percy Liang, Michael I Jordan and Dan Klein. "*Learning programs: A hierarchical Bayesian approach*". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. **2010**: 639–646.

[138] Pranav Garg, Christof Löding, P Madhusudan *et al.* "*ICE: A robust framework for learning invariants*". In: *International Conference on Computer Aided Verification*. **2014**: 69–87.

[139] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan *et al.* "*Learning invariants using decision trees and implication counterexamples*". *ACM Sigplan Notices*, **2016**, *51*(1): 499–512.

[140] 刘斌斌，董威，王戟。"智能化的程序搜索与构造方法综述" [J]。软件学报，**2018**，*29*(8)：2180–2197。

[141] 胡星，李戈，刘芳 等。"基于深度学习的程序生成与补全技术研究进展" [J]。软件学报，**2019**，*30*(5)：1206–1223。

[142] Abram Hindle, Earl T Barr, Zhendong Su *et al.* "*On the naturalness of software*". In: *2012 34th International Conference on Software Engineering (ICSE)*. **2012**: 837–847.

[143] Veselin Raychev, Martin Vechev and Eran Yahav. "*Code completion with statistical language models*". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. **2014**: 419–428.

[144] Zhaopeng Tu, Zhendong Su and Premkumar Devanbu. "*On the localness of software*". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. **2014**: 269–280.

[145] Veselin Raychev, Pavol Bielik, Martin Vechev *et al.* "*Learning programs from noisy data*". *ACM SIGPLAN Notices*, **2016**, *51*(1): 761–774.

[146] Veselin Raychev, Pavol Bielik and Martin Vechev. "*Probabilistic model for code with decision trees*". *ACM SIGPLAN Notices*, **2016**, *51*(10): 731–747.

[147] Jian Li, Yue Wang, Michael R Lyu *et al.* "*Code completion with neural attention and pointer networks*". *arXiv preprint arXiv:1711.09573*, **2017**.

[148] Vincent J Hellendoorn and Premkumar Devanbu. "*Are deep neural networks the best choice for modeling source code?*" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. **2017**: 763–773.

[149] Anshita Saxena, Aniket Saxena and Jyotirmay Patel. "*DeepCoder: An Approach to Write Programs*". In: *International Conference on Advanced Research and Innovation in Engineering*. **2017**.

[150] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang *et al.* "*Deep API learning*". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. **2016**: 631–642.

[151] Edsger W Dijkstra *et al.* "*A note on two problems in connexion with graphs*". *Numerische mathematik*, **1959**, *1*(1): 269–271.

[152] Peter E Hart, Nils J Nilsson and Bertram Raphael. "*A formal basis for the heuristic determination of minimum cost paths*". *IEEE transactions on Systems Science and Cybernetics*, **1968**, *4*(2): 100–107.

[153] Mark F. Medress, Franklin S. Cooper, James W. Forgie *et al.* "*Speech Understanding Systems*". *Artif. Intell.* **1977**, *9*(3): 307–316.

[154] Cameron B Browne, Edward Powley, Daniel Whitehouse *et al.* "*A survey of monte carlo tree search methods*". *IEEE Transactions on Computational Intelligence and AI in games*, **2012**, *4*(1): 1–43.

[155] Patrick Cousot and Radhia Cousot. "*Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. **1977**: 238–252.

[156] Alfred V. Aho, Monica S. Lam, Ravi Sethi *et al. Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., **2006**.

[157] Zhimin He, Fayola Peters, Tim Menzies *et al.* "*Learning from open-source projects: An empirical study on defect prediction*". In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. **2013**: 45–54.

[158] Song Wang, Taiyue Liu and Lin Tan. "*Automatically learning semantic features for defect prediction*". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. **2016**: 297–308.

[159] Tianqi Chen and Carlos Guestrin. "*Xgboost: A scalable tree boosting system*". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. **2016**: 785–794.

[160] Hans Peter Deutsch. *Principle Component Analysis*. Palgrave Macmillan UK, **2002**.

[161] W Eric Wong, Ruizhi Gao, Yihao Li *et al.* "*A survey on software fault localization*". *IEEE Transactions on Software Engineering*, **2016**, *42*(8): 707–740.

[162] Rui Abreu, Peter Zoeteweij and Arjan J. C. van Gemund. "*An Evaluation of Similarity Coefficients for Software Fault Localization*". In: IEEE Computer Society, **2006**: 39–46.

[163] Jifeng Xuan and Martin Monperrus. "*Learning to combine multiple ranking metrics for fault localization*". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. **2014**: 191–200.

[164] Friedrich Steimann, Marcus Frenkel and Rui Abreu. "*Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators*". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. **2013**: 314–324.

[165] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Locating faults through automated predicate switching*". In: *Proceedings of the 28th international conference on Software engineering*. **2006**: 272–281.

[166] José Campos, André Riboira, Alexandre Perez *et al.* "*Gzoltar: an eclipse plug-in for testing and debugging*". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. **2012**: 378–381.

[167] Thomas Durieux, Fernanda Madeiral, Matias Martinez *et al.* "*Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts*". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2019**: 302–313.

[168] Dinh Xuan Bach Le, Lingfeng Bao, David Lo *et al.* "*On reliability of patch correctness assessment*". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. **2019**: 524–535.

[169] Gordon Fraser and Andrea Arcuri. "*Evosuite: automatic test suite generation for object-oriented software*". In: *ESEC/FSE*. **2011**: 416–419.

[170] Kim N King and A Jefferson Offutt. "*A fortran language system for mutation-based software testing*". *Software: Practice and Experience*, **1991**, *21*(7): 685–718.

[171] Susumu Tokumoto, Hiroaki Yoshida, Kazunori Sakamoto *et al.* "*MuVM: Higher Order Mutation Analysis Virtual Machine for C*". In: *ICST*. **2016**: 320–329.

[172] Chris Lattner and Vikram Adve. "*LLVM: A compilation framework for lifelong program analysis & transformation*". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* **2004**: 75–86.

[173] Randal E Bryant, O'Hallaron David Richard and O'Hallaron David Richard. *Computer systems: a programmer's perspective*. Prentice Hall Upper Saddle River, **2003**.

[174] Milos Gligoric, Vilas Jagannath, Qingzhou Luo *et al.* "*Efficient mutation testing of multithreaded code*". *Software Testing, Verification and Reliability*, **2013**, *23*(5): 375–403.

[175] Iftekhar Ahmed, Carlos Jensen, Alex Groce *et al.* "*Applying Mutation Analysis on Kernel Test Suites: An Experience Report*". In: *ICSTW*. **2017**: 110–115.

[176] Hyunsook Do, Sebastian G. Elbaum and Gregg Rothermel. "*Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.*" *Empirical Software Engineering: An International Journal*, **2005**, *10*(4): 405–435.

[177] Konstantin Serebryany, Derek Bruening, Alexander Potapenko *et al.* "*AddressSanitizer: A fast address sanity checker*". In: *Presented as part of the 2012 USENIX Annual Technical Conference USENIX ATC 12)*. **2012**: 309–318.

[178] Satish Chandra, Stephen J Fink and Manu Sridharan. "*Snugglebug: a powerful approach to weakest preconditions*". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. **2009**: 363–374.

[179] Ivan Jager and David Brumley. "*Efficient directionless weakest preconditions*". In: *Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab*. **2010**.

[180] Gregory J. Duck and Roland H. C Yap. "*Heap Bounds Protection with Low Fat Pointers*". In: *Proceedings of the 25th International Conference on Compiler Construction*. ACM, **2016**: 132–142.

[181] Gregory J. Duck, Roland H. C. Yap and Lorenzo Cavallaro. "*Stack Bounds Protection with Low Fat Pointers.*" In: *Network and Distributed System Security Symposium (NDSS)*. **2017**.

[182] Cristian Cadar, Daniel Dunbar, Dawson R Engler *et al.* "*KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. **2008**: 209–224.

[183] Claire Le Goues, Neal Holtschulte, Edward K Smith *et al.* "*The ManyBugs and IntroClass benchmarks for automated repair of C programs*". *IEEE Transactions on Software Engineering*, **2015**, *41*(12): 1236–1256.

附录 A　Hanabi 学习算法的参数设置说明

表 A.1　Hanabi 中 XGBoost 的重要参数设置

参数名称	E	V	L	参数描述
booster	gbtree	gbtree	gbtree	所用的 booster 种类
objective	multi:softprob	binary:logistic	multi:softprob	使用输出概率的目标
max_depth	7	6	7	树的最大深度
eta	0.1	0.1	0.1	学习率，即每一步迭代的步长
gamma	0.2	0.2	0.2	节点分裂所需的最小损失函数下降值
lambda	1	1	1	权重的 L2 正则化项
subsample	0.7	0.7	0.7	每棵树随机采样的比例
col_sample_bytree	0.2	0.2	0.2	每棵随机采样特征数的占比
min_child_weight	1	1	1	孩子的实例权重所需满足的最小和
eval_metric	merror	error	merror	验证数据的评价指标
num_boost_round	1000	1000	1000	训练轮数

表中的 `E`、`V` 和 `L` 三列是这三个非终结符对应模型的参数。

如第四章中所述，Hanabi 通过实例化玲珑框架来生成 Java 程序的 `if` 语句的条件表达式。Hanabi 需要在当前上下文内预测选择哪一条扩展规则，因此 Hanabi 需要对每个非终结符训练模型。Hanabi 的实现中支持了多个学习模型，包括朴素贝叶斯、支持向量机和 XGBoost。

在朴素贝叶斯模型中，Hanabi 使用了高斯朴素贝叶斯分类器 (Gaussian Naive Bayes)[①]，不涉及参数设置。高斯朴素贝叶斯分类器假设数据符合高斯分布，并在此基础上应用朴素贝叶斯公式进行计算。

在支持向量机模型中，Hanabi 的参数 `decision_function_shape` 设置为 `ovr`。

Hanabi 最终使用 XGBoost 模型实施修复。根据4.2.2小节所述，Hanabi 把对 `E` 和 `L` 的预测建模为多分类问题，而把对 `V` 的预测建模为二元分类问题。在 Hanabi 的 XGBoost 模型中，多分类问题和二元分类问题的参数略有不同，详见表 A.1。表中的参数名称及其描述引用自 XGBoost 官方文档 [②]。

[①] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
[②] https://xgboost.readthedocs.io/en/latest/parameter.html

博士期间研究成果

发表论文

[1] Xiang Gao, **Bo Wang**, Gregory J. Duck, Ruyi Ji, Yingfei Xiong and Abhik Roychoudhury. "Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction". In: *ACM Transactions on Software Engineering and Methodology (TOSEM). 2021. To appear.*

[2] Yingfei Xiong, **Bo Wang**, Guirong Fu and Linfei Zang. "Learning to Synthesize". In: *Proceedings of the 4th International Workshop on Genetic Improvement Workshop. 2018: 37-44.*

[3] **Bo Wang**, Yingfei Xiong, Yangqingwei Shi, Lu Zhang and Dan Hao. "Faster Mutation Analysis via Equivalence Modulo States". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 2017: 295–306.* **ACM SIGSOFT 杰出论文奖.**

[4] **Bo Wang**. "Dynamic Analysis of Shared Execution in Software Product Line Testing". In: *Proceedings of the 20th International Systems and Software Product Line Conference. 2016: 340-341.*

[5] 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),*

待投论文

[1] "L2S: a Framework for Program Synthesis under a Partial Specification".

[2] "Scoped Mutation Analysis with Sparse Interpretation".

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。

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

致谢

首先感谢北京大学。北大培养了钟南山、于敏等为国为民的侠之大者，培养了屠呦呦、王选等为用科学造福人民的科学家，培养了张益唐、樊锦诗等孜孜不倦钻研学问的研究者。未名湖畔的大师凝聚成北大的精神，深深感染着每个北大学子。这里还有来自全国各地的优秀同学，与优秀的人为伍给了我莫大的鞭策和激励。在燕园中的经历，是我一生的宝贵财富。

感谢杨芙清院士和梅宏院士。两位院士为中国的软件事业打下深厚的根基，为中国软件事业做出了巨大贡献。两位院士建立的北大软件研究所，为我们提供了世界一流的研究平台。读博期间，见证了北大软件工程方向由大变强，从顶会影响力来看，北大软件所已经是软件工程方向世界第一实验室，这离不开两位院士高瞻远瞩的规划。在此软件所的平台上，我们能够与世界其他顶尖大学交流合作，开阔研究视野，并在学科的前沿开展研究。

感谢我的导师熊英飞副教授。在刚进入实验室的时候，熊老师带领我熟悉科研流程，从选题、设计实验到报告，都手把手的指导。有几次在截至日前，熊老师还和我们一起在实验室里通宵达旦修改论文。对于实验中不符合预期的现象，熊老师迅速想到可能的解释，指导下一步的探索方向。在研究陷入僵局的时候，熊老师以他的乐观与坚毅劝导并指点我。熊老师对未知世界的好奇、对已有研究的洞察、对未来发展方向敏锐的嗅觉还有对艰深理论的坚守，都示范了一个纯粹的科学家的样子。希望我今后能以熊老师为榜样前进，立志做好的研究。

感谢张路教授和郝丹副教授在对我的科研和培养过程上的指导。感谢软件所的胡振江教授、金芝教授、谢涛教授、谢冰教授、黄罡教授、曹东刚教授、陈向群教授、焦文品教授、郭耀教授、王千祥教授、刘譞哲副教授、张伟副教授、邹艳珍副教授、陈泓捷副教授、赵海燕副教授、李戈副教授等老师在我的博士期间的指导和帮助。感谢徐松青老师、荆婷婷老师等教务老师的耐心帮助。感谢北大的刘先华副教授、闵瑛美博士，我在二位老师的精彩授课中学到很多。

感谢北京化工大学的李征教授、北京理工大学的刘辉教授、中科院软件所的魏峻研究员、中科院软件所的张健研究员、中科院计算所的武成岗研究员、武汉大学的玄跻峰教授、新加坡管理大学的孙军副教授、北京科技大学的何啸副教授对我的帮助和指导。

感谢新加坡国立大学的 Abhik Roychoudhury 教授。ExtractFix 是在 Abhik 组访问期间完成的工作，感谢 Abhik Roychoudhury 教授提供的交流机会。感谢该项目的合作者

高祥和 Gregory Duck。感谢新加坡国立大学 Tsunami 小组的董震、高明远、俞小亮、陈馨惠等成员的帮助。我很怀念与你们一起讨论研究意义，赞赏你们对待研究的态度，并祝你们早日做出自己满意的工作。

感谢同门姜佳君，与你交流技术，讨论学术是十分享受的事，你的勤奋和乐观感染着我，感谢你在博士期间的帮助。感谢学弟卢思睿，你废寝忘食的调试代码，让我明白如何热爱一件事。感谢与我一起合作的史杨勋惟、臧琳飞、刘雨轩、伏贵荣、周兆平、吉如一等同学，三人行必有我师，从你们身上我学到很多。感谢我的同门李军、高庆、悦茹茹、邹达明、米亚晴、王杰、杨小东、梁晶晶、刘鑫远、曾沐颔、章嘉晨、陈逸凡、任路遥、张星、张云帆等同学，感谢你们的陪伴与帮助，与你们相处的这段时光对我十分宝贵。特别感谢梁晶晶、张星和陈逸凡同学对本文的校验。感谢陈俊洁、唐浩、涂菲菲、娄一翎、陈震鹏、孙泽宇、王冠成、张洁、曹英魁、刘兆鹏、王敏、王春晖、朱家鑫、周建祎、谭鑫、李念语、罗武、傅英杰、陈天宇等其他组的同学。

感谢我的室友夏玉堂，你的善良、正直与乐观深深地影响着我。感谢刘畅、彭佳和沈忱等好友，祝你们实现自己的人生理想。感谢我的家人，感谢父母的陪伴和支持，让我得以安心完成学业。感谢蒋燎原女士的陪伴和鼓励，你对生活的热爱和坦然的态度使我备受鼓舞。

攻读博士学位是我人生中重要的经历，纵使期间会有迷茫与怀疑，我依然很感恩当时做出读博的决定。这段经历让我变得更好，我逐渐明白了想要怎样的生活，想成为怎样的人。在漫长的摸索中我逐渐看清了自己的边界，但是依然保留着突破它的勇气。最后，以日本作家柴田翔在其获得芥川文学奖的作品《别了，我们的生活》中的一段话作为全文的结尾：将来我们老了也许青年人会问：你们年轻时怎么样？那时我们会回答，我们青年时也一样有困难。当然，时代不同，困难也不同，可是有困难这一点是相同的，并且告诉他们我们习惯于在困难中生存，一直到老了的现在。可我们中也有从时代的困难中挣脱出去，勇敢投入新生活的人。听了我们这样回答，也许会有年轻人说：这样的事你们年轻时有，我们现在也有，要是我们具备挣脱困难的勇气，到老时我们的生活也许会更有意义！

感谢一直以来的幸福人生。

今迄は幸せな人生だったと感謝している。

北京大学学位论文原创性声明和使用授权说明

原创性声明

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

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

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

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日