北京大學

本科生毕业论文

题目：基于单样例的交互式缺陷检测和修复：语言设计与工具实现

Interactive Bug Detection and Repair Based on Single Example: Language Design and Tool Implementation

姓　　名：张石然

学　　号：1700012858

院　　系：信息科学技术学院

本科专业：计算机科学与技术

指导导师：熊英飞

二〇二一 年 六 月

北京大学本科毕业论文导师评阅表

学生姓名	张石然	学生学号	1700012858	论文成绩	
学院（系）	信息科学技术学院			学生所在专业	计算机科学与技术
导师姓名	熊英飞	导师单位/所在研究所	北京大学软件工程研究所	导师职称	副教授
论文题目（中、英文）	基于单样例的交互式缺陷检测和修复：语言设计与工具实现 Interactive Bug Detection and Repair Based on Single Example: Language Design and Tool Implementation				

导师评语

（包含对论文的性质、难度、分量、综合训练等是否符合培养目标的目的等评价）

根据已有缺陷检测和修复更多同类缺陷是代码静态检测的一个基本手段，但现有技术往往需要大量的同类已有缺陷进行学习，而在实践中对特定类别该数据往往不可得。本团队之前的工作构建了首个基于单个缺陷样例构造缺陷模式并生成自动修复工具的工作，但该工作应用到检测缺陷时精度却难以达到实用要求。在天津大学姜佳君老师的共同指导下，张石然的本科毕业论文设计了一套表示缺陷模式的语言和一套用于编辑该语言程序的图形界面工具，使得自动生成的缺陷模式可以经过人工编辑。实验表明，张石然所设计的语言和图形界面工具使得有经验的程序员只需要少量操作就能够修改缺陷模式，并且修改后的缺陷模式可以以较高的准确率检测出大量之前无法准确检测的缺陷。

总的来说，论文提升了一个已有方案的效果，具有一定的学术价值和实用价值。论文行为规范，达到了本科毕业论文的要求。

导师签名：

年　　月　　日

版权声明

摘要

现代软件开发中有很多重复性工作，现有研究提出了程序变换的方法，尝试将这些重复性工作自动化。先前工作 GenPat 提出了基于单样例的程序变换推导方法，自动化地完成了代码重复修改的工作。但受限于设计的局限，在缺陷检测和修复场景下得到了大量假阳性的结果。

本文基于先前的研究，设计了基于 Datalog 的可交互的模板表示形式，在进行抽象节点匹配的同时支持人工修改，将补丁具体语义纳入了考虑。与此同时，本文开发了便于用户进行交互操作的用户图形界面工具，并对工具效果进行了系统化的实验验证。

结果证明，本文方法在缺陷检测的场景下取得了优于 GenPat 的效果，将原先的准确率从 2.6%提升到了 78.3%，同时将人工筛选和修正补丁的时间缩短了近 7 倍。此外，本文的工具在帮助开发者进行开源项目开发场景下也表现良好，共在 5 个开源项目上提交了 71 个有效补丁，有 64.8%已经被开发者接受。

关键词：缺陷检测，缺陷修复，程序变换语言，交互式工具

Abstract

There is a lot of repetitive work in modern software development. Existing researches put forward different methods of program transformations and try to automate these repetitive work. In a previous research, the researchers proposed the tool GenPat, which involves a program transformation derivation method based on a single example that can automatically complete the repetitive code modification. However, due to the limitation of design, this method results a large number of false positive results in defect detection and repair scenarios.

Based on the previous research, this thesis designs pattern representation in Datalog, which supports manual modification and takes the specific semantics of patches into account. At the same time, this thesis develops a graphical user interface tool which is convenient for users to modify the pattern, and systematically verifies the effectiveness of the tool.

The results show that the proposed method is superior to GenPat in the scene of defect detection, which improves the original accuracy rate from 2.6% to 78.3%, and shortens the time of manual patch correction by nearly 7 times. In addition, the tool also performs well in the scenario of helping developers manage open source projects. A total of 71 effective patches have been submitted in five open source projects, and 64.8% of them have been accepted by developers.

Key Words: Bug Detection, Defect Repair, Program Transformation Language, Interactive Tools

全文目录

第一章 绪论

在当今时代，软件系统已经被大规模地使用在人类生活的方方面面，给人类生活带来了极大的便利。然而，现代软件开发中有很多重复性工作，如 M. Kim 等人指出重复的代码编辑[20]，Q. Gao 等人重复的缺陷修复[21]，B. Ray 等人重复的提交[28]等等。这些重复性工作虽然不完全相同，但非常相似。如 M. Kim 等人发现，平均而言，有 75%的成熟软件的架构变化是系统性的，这些变化的上下文有着相似的语义，如调用了相同的函数或者访问了同一字段[20]。H. A. Nguyen 等人的研究表明，17%至 45%的缺陷修复都是重复性修复，涉及到对许多函数的相似修改[25],[26]。在 API 修改的场景中，为对 API 进行统一的迁移或升级，大量的重复性修改也经常出现[1],[2],[3]。这样类似的修改还会出现在分支程序（forked projects）的补丁之间[28]。

现有工作针对这个现象提出了多种自动化的方法。最早比较通用的方法是采用搜索替换的方式进行修改，但这类方法只支持简单的文本替换，而没有考虑补丁的上下文语义。随着技术的发展，有些工具，如 Eclipse 的重构功能，在程序变换时保留了对语义的考虑，但它仅限于考虑预先定义的语义信息[27]。当前一种比较经典的策略是利用多个实例的统计信息，确定变换中哪些部分是具体的，哪些部分是抽象的，从而进行程序变换[29],[30],[31]。但在实际场景下，开发者提交的补丁具有稀疏性，大量缺陷类型通常只有少数的补丁实例。

H. A. Nguyen 等人提出了根据单个补丁修改实例检测新的编辑位置的方法，但这个方法没有自动化补丁的泛化，需要程序员通过手动编辑代码，以实现对类似代码片段的修改[25],[26]。有一些方法提供了集成程序变换语言的工具，自动化了模板泛化的过程，但却要求开发者自行确定修改操作[23],[24]。基于补丁语义进行程序变换推导的一类工具解决了这个问题，通过对比补丁修改前后的代码差异，自动化生成了编辑操作，但这个方法的问题在于具体的补丁语义的表达能力较弱，且没有将补丁的上下文信息纳入考虑[22]。M. Kim 等人提出了基于单样例的程序变换，将控制流依赖、数据依赖等上下文信息加入了模板提取的考虑中，解决了先前工作的问题，但 Sydit 依旧采用了一些预定义的规则来指导抽象泛化过程[27]。J. Jiang 等人改进了 Sydit，利用代码大数据对程序变换进行指导，但是

却无法处理特定补丁独有的语义，导致结果出现大量假阳性错误[5]。

本文基于先前的 GenPat 单样例程序变换工作，在现有工作的基础上，提出了在缺陷检测和修复场景下的程序变换语言，通过加入对单个修改样例模板的人工修复，来处理自动化方法无法考虑的补丁特有的语义信息，改善了先前工作出现大量假阳性错误的问题。我们工作的主要贡献包括：

- 设计了基于 Datalog 的可交互的模板表示形式：在进行抽象节点匹配的同时支持人工修改，能考虑补丁具体语义；
- 开发了便于用户进行交互操作的用户图形界面（graphical user interface，简称 GUI）工具，缩短了人工修正错误的缺陷检测和修复的时间，提高了开发效率；
- 对工具效果进行了系统化的实验验证，并在开源项目上提交了有效提交请求（pull request，简称 PR）被开发者接受，证明了该工具的有效性。

我们的工具在缺陷检测的场景下取得了良好的效果。实验首先分析了 GenPat 在缺陷检测和修复场景下的效果，发现结果中有大量假阳性错误。随后，实验对比了 GenPat 和本文工具在由 6 个开源项目和经过筛选的 65 个补丁实例构成的数据集上的效果，并发现本文工具在加入了一定量的人工修改之后，取得了比 GenPat 更高的准确率。此外，本文的工具在帮助开发者进行开源项目开发场景下也表现良好。实验共选取了 12 个修改实例，在 5 个开源项目上提交了 71 个有效补丁，其中有 46 个补丁已经被开发者接受。

下面简要介绍本文的结构：第二章，阐释了本文的研究动机，包括对先前的研究工作的简介和一个研究动机实例的介绍；第三章，陈述了现有的相关工作和技术；第四章，详细介绍了本文的方法设计和工具实现；第五章，介绍了本文的实验设计、实验结果及分析；第六章，总结了本文的工作，并提出未来研究可能展开的方向。

第二章 研究动机

本章主要介绍先前的研究工作和遇到的困难，并提供了一个动机实例。

1. 先前工作介绍

软件的演化过程中存在着各种各样重复或相似的修改。本文继续了 J. Jiang 等人[5]的工作，优化了基于单样例的程序变换方法。先前的工作采用超图提取和匹配的方式，提出了一套基于单样例的程序变换（program transformation）的框架，实现了一个可以通过上下文和代码大数据推导程序变换的工具 GenPat。结果显示，GenPat 在系统性编辑（systematic editing）和程序修复（program repair）两个场景下都获得了不错的效果。

这一部分将从修改提取、缺陷挖掘和补丁生成这三个模块简单介绍 GenPat 的设计。GenPat 工具的整体框架如图 2.1 所示。根据修改实例，GenPat 采用 GumTree 算法[16]，通过提取代码超图和代码修改的操作序列，结合小样本学习的思路，给出了该实例的程序变换模板。在待检测的项目代码上，工具通过静态分析的方法将待检测的代码片段同样转换为代码超图的表达形式，并用已经生成的模板进行匹配和缺陷挖掘。在检测到可能的缺陷之后，工具会根据模板和缺陷的上下文生成修复补丁，推荐给工具的使用者。

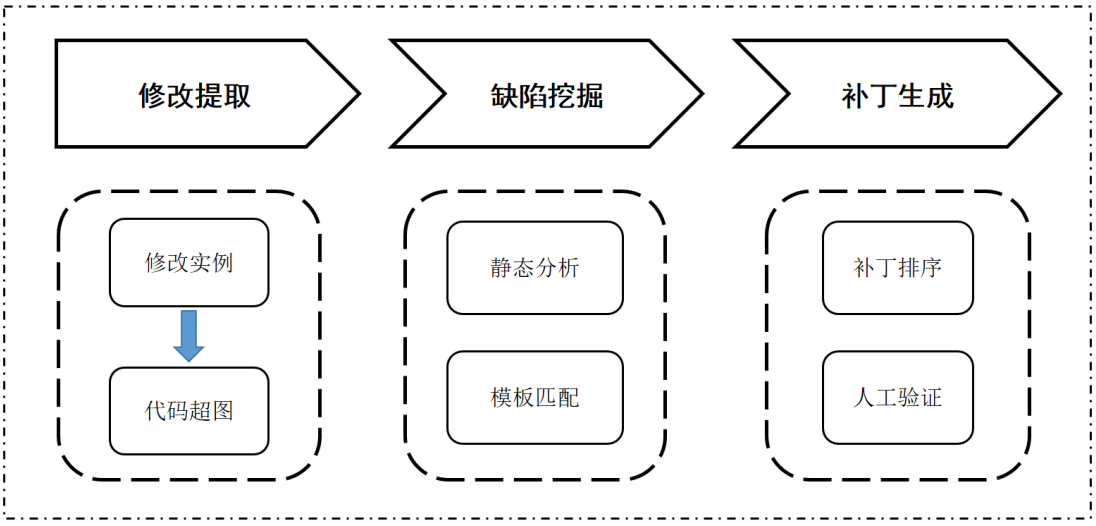

图2.1 GenPat工具框架

1.1 修改提取

在修改提取阶段，该工具通过提取修改补丁中的信息来获取每个补丁执行的修改，构造修改模板。基于已有的修改实例的输入，GenPat 将修改前后的代码抽象成两个 AST。借鉴 GumTree 算法的思想，GenPat 首先在两个 AST 之间匹配节点，然后在匹配过程中将修改提取为用超图形式表示的模板。确定两个节点匹配的条件包括：

- 它们具有相同的节点类型；
- 它们的所有祖先都匹配；
- 可以通过从目标 AST 到源 AST 插入一些节点来满足前两个条件。

1.2 缺陷挖掘

缺陷挖掘阶段的输入是一个修改实例的模板和一个待检测的项目代码文件。GenPat 工具首先通过静态分析的方法先将项目代码抽象为 AST，然后根据超图匹配的规则，用修改提取阶段生成的修改模板，对待检测项目进行检测和匹配。对于超图 A 和超图 B，其匹配规则如下：

- 根据节点属性和节点间关系，计算 B 中的每个元素与 A 中元素能够匹配的集合；
- 利用代码结构相似度和文本相似性，对多个匹配进行打分并排序，其中，相似度、代码结构相似度和文本相似度的计算方式如下：

$$\text{相似度} = \text{代码结构相似度} + \text{文本相似度}$$

$$\text{代码结构相似度} = \text{B 中匹配的节点个数} \div \text{B 中的总节点个数}$$

$$\text{文本相似度} = (\text{补丁的最长公共子序列长度} * 2) \div (\text{原文本长度} + \text{待检测文本长度})$$

1.3 补丁生成

一个修改实例只能提取一个修改模板，但一个模板最终会生成多个修复补丁。GenPat 工具会参考预定义的推荐补丁数量，根据上一模块已经按照相似度排好顺序的匹配，生成具体的补丁推荐。对于每次补丁生成，GenPat 针对每个匹配，通过利用被检测代码的文本信息，将模板的节点转换为具体的代码。

生成的补丁将通过 diff 文件的形式，作为输出直接被提供给工具的使用者，由使用者人工判断补丁的正确性。

2. 研究动机实例

J. Jiang 等人的工作验证了基于单样例的程序变换推导方法的有效性，并在与 Sydit 对比的实验场景中取得了良好的效果[5]。本文工作试图将这个方法应用在大规模的缺陷检测和修复中，但实践发现，这个方法在通用的缺陷检测的场景下准确率并不高。

本节将以 Apache Hadoop 项目中一个补丁为实例进行分析。

```
80  80      @Override
81  81      public synchronized List<LogMutation> getPendingMutations() {
82      -     return pendingMutations;
    82  +     return new LinkedList<>(pendingMutations);
83  83      }
84  84
```

图2.2 补丁实例

在该补丁中，开发者对返回值 `pendingMutations` 做了修改，创建了一个以 `pendingMutations` 为参数的链表（list）类型的匿名对象，并将其返回。这个补丁的修复避免了未决变化（pending mutations）在 LevelDB 存储中的恢复问题，是对 Hadoop 资源管理（resource manager）模块中一个功能缺陷的修复。

利用原有的 GenPat 工具，在 Apache Flink 项目中进行检测，我们发现了一些可能的程序变换。

```
1  ----Adapted Patch----
2   @Override
3   public List<NimbusInfo> getAllNimbuses() throws Exception{
4  -return Arrays.asList(leaderAddress);
5  +return new LinkedList<>(Arrays.asList(leaderAddress));
6   }
7
```

图2.3.1 GenPat指导的可能的程序变换（1）

经过人工分析，我们发现，这些检测到的可能的缺陷并不是正确的。如图2.3.1 所示，在第一个例子中，返回值已经通过添加 `Arrays.asList()`，将变量

leaderAddress 从数组转换为了链表，补丁的修复对返回值似乎没有任何帮助。

```
----Adapted Patch----
 public List<String> getTopologyActions(String topologyName){
-return topologyToActions.get(topologyName);
+return new LinkedList<>topologyToActions.get(topologyName));
 }

```

图2.3.2 GenPat指导的可能的程序变换（2）

图 2.3.2 的第二个例子中看起来是对一个接口调用的封装，在这个场景下，生成补丁为返回值创建一个匿名对象似乎毫无意义。针对图 2.3.3 的第三个例子，该方法的返回值是映射（map），而原补丁修改实例支持的函数类型却是链表，基本可以判定这个生成的补丁也不是正确的。

```
----Adapted Patch----
 @Override
 public Map<String, Object> getComponentConfiguration(){
-return null;
+return new LinkedList<>(null);
 }

```

图2.3.3 GenPat指导的一些可能的程序变换（3）

基于这个实例，本文发现，目前 GenPat 工具在应用补丁实例进行检测修复的时候，会产生很多假阳性（false positive）的结果。对比原补丁和这些假阳性的错误，本文认为，使用者如果对补丁的特性比较熟悉，那么筛选这些错误补丁是比较容易的，但人工筛选将会耗费大量时间。仅此一个实例，GenPat 工具就在 Apache Flink 上检测出了近 200 个假阳性结果，如果每个结果需要约 15 秒的判断时间，那么这些假阳性结果要耗费将近一个小时的时间和人力去进行筛选。但与此同时，要排除这些错误实际上又非常容易，因为它们与原补丁的差异比较明显，通过对返回值 pendingMutations 的一些属性特征的保留，就可以过滤掉大部分错误。

因此，我们想在补丁泛化的阶段就对某些重要的特征信息进行标记保留，这样在缺陷挖掘和修复生成的时候，可以有效避免大量的假阳性结果的出现，避免人力开发资源的不必要的浪费。

第三章 相关工作

这一部分主要介绍本文工作涉及到的领域内的现有工作和相关技术。

1. 程序变换语言

程序变换语言是将一种语言转换成另一种语言的一套规则系统。在该领域的现有工作主要分为两类，一类工作设计了较为通用的程序变换语言，提供了一套样例泛化规则和变换策略，但这一类工作通常需要开发者结合源代码语言和目标代码语言，集成程序变换语言的规则和设计思路，自行实现具体的程序变换工具。另一类工作将程序变换语言与具体的应用场景相结合，设计了一套适用于具体场景的模板提取和修改规则，如基于补丁语义的程序变换语言，但这类方法的问题在于语言的表达能力过弱，不适用于通用的程序变换场景。程序变换语言的提出降低了程序变换工具的开发难度。

1.1 通用目的的程序变换语言

这类工作，如 Stratego[32]、TXL[24]等，设计了通用场景下的程序变换语言，通常被应用在编译器、优化、程序合成、重构等场景中。Stratego 由泛化策略（generic strategies）和生成规则（transformation rules）两部分组成，为程序变换场景提供了一套通用的样例泛化规则和变换策略。TXL 是一种语言转换和快速原型构建系统，设计了分析、转换和解析输出三个阶段，能够实现对任意上下文无关文法语言的解析变换和特定结构的匹配、检索与转换功能。

这一类语言通常表达能力较强，但较为底层和抽象，需要开发者结合源代码语言和目标代码语言，集成程序变换语言的规则和设计思路，自行实现具体的程序变换工具。

1.2 特定领域的程序变换语言

这一类程序变换语言通常是针对特定领域问题而设计的，如 PATL[33]的提出

主要解决了 API 迁移领域中语句层面多对多映射的问题；Twinning[34]语言设计主要解决了 API 一对多变换的问题，但缺乏安全性检查；SWIN[35]语言基于 Twinnig 语言增加了一个类型检查系统，保证了 API 迁移过程中类型的正确性。SmPL[22]则主要集中于基于补丁语义的程序变换语言设计，在程序上下文中通过标识语句之间的依赖关系，使用模型检查的方式来进行匹配。

但是这类语言的缺点是语言设计与具体场景问题的结合较为紧密，语言表达能力较弱，不适用于通用的程序变换场景。

2. 程序变换

2.1 基于多个实例的程序变换

这类工作，如 Genesis[30]、LASE[36]等，利用多个实例的统计信息，确定变换中哪些部分是具体的，哪些部分是抽象的，从而进行程序变换。其特点是依赖多个同类型的修改去生成抽象模板。但在实际开发中，开发者提交的补丁具有稀疏性，大量缺陷类型通常只有少数的补丁实例。因此这类方法虽然较为经典，但在实际应用场景中的效果仍有很大的提升空间。

2.2 基于单样例的程序变换

当前的基于单样例进行程序变换推导的工作是针对上述基于多个实例进行程序变换推导的改进。但现有工作，如 Sydit[27]，采用了一些预定义的规则来指导样例泛化过程，同时只考虑了结构信息而没有考虑具体补丁的语义信息，使得该方法无法灵活应对复杂的开发场景和需求。GenPat 工作[5]改进了 Sydit 方法，利用代码大数据对程序变换进行指导，但是受限于全自动化的方法，依旧却无法处理特定补丁独有的语义。

3. 缺陷自动修复

在软件开发过程中，开发者通常会花费大量的时间和精力来修复软件中的缺陷。有数据表明，程序开发人员在修复代码缺陷上的时间开销可能达到了总开发

时间的一半[8]。软件缺陷自动修复（automatic software repair）技术是自动化解决软件缺陷问题的技术领域之一。自动缺陷修复工具通常以一个有缺陷的程序和一组测试集为输入，其中测试集需要包含至少一个能够复现代码缺陷的测试用例。工具最终输出一个在源程序上能够通过测试、修复缺陷的补丁。

程序缺陷自动修复技术框架主要包括三部分：缺陷定位、补丁生成、补丁验证。缺陷自动修复工具可以根据补丁生成的不同方法分为四类：基于启发式搜索、基于人工修复模板、基于语义约束、基于统计分析[12]。

现有的自动修复工具，如 ARJA[9]、DirectFix[11]等均依赖测试定位程序中的缺陷。Q. Gao 等人提出的 LeakFix[9]摆脱了对测试输入的依赖，针对内存泄露错误进行自动修复。A. Marginean 等人提出的 Sapfix[13]同样不依赖于测试输入，针对面向对象语言中的空指针缺陷。这些技术不依赖测试输入，但同时只能修复已经被发现的缺陷，不能检测其他同类缺陷。

第四章　方法介绍与工具实现

基于已有工作，本文设计了一个基于 Datalog 的程序变换语言，并开发了相关的用户图形界面工具，通过引入人工修改的方法，来提高缺陷检测的准确率和程序员的开发效率。本章将详细介绍该程序变换语言的设计和工具的实现。

1. 方法设计

这一部分简单介绍本文的顶层设计方法。

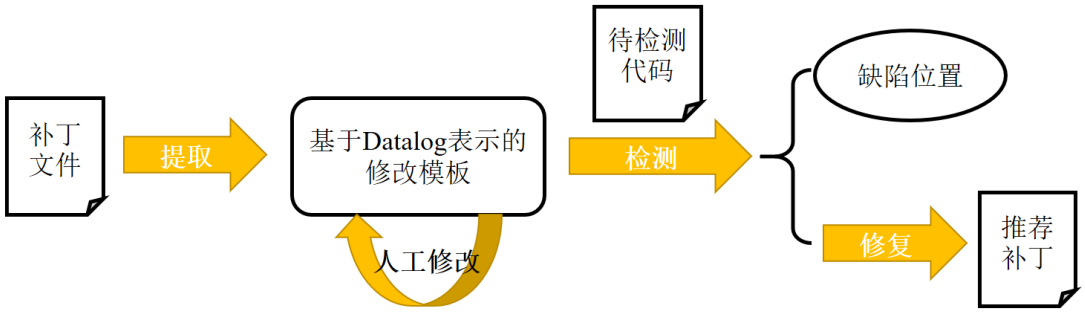

图4.1 整体流程

如上是整个基于单样例进行缺陷检测和修复的整体流程。通过使用者提供的一个补丁文件，本文的方法可以将它提取成一个由 Datalog 表示的修复模板。在这个修复模板中，每个抽象节点拥有一些节点属性，使用者可以通过对这个节点包含或不包含的选择，来决定进行程序变换时抽象和保留的元素。

一份被使用者确定的模板会保存在本地，并被本文工具用来作为新的输入去检测待检测库上的缺陷。如果能检测到结果，那么本文的方法会将缺陷的位置和推荐的修复补丁一并告诉使用者。

2. 基于 Datalog 的模板表示形式

这一部分将对本文的主要贡献之一——基于 Datalog 的修复模板表示形式的设计——给予介绍。

2.1 Datalog 简介

Datalog 是一种声明式的逻辑编程语言（declarative logic programming language），在语法上是 Prolog 的一个子集[18]。Datalog 最初用于演绎数据库（deductive database），具有与 SQL 相类似的表达能力，现在也可用于数据集成（data integration）、程序分析（program analysis）、安全（security）、云计算（cloud computing）、机器学习（machine learning）等领域[19]。

Datalog 可以实现对知识库的演绎推理，即可以从已知的事实中根据规则推理得到新的事实。但与 Prolog 不同，Datalog 对事实和规则的出现顺序不做要求，即两条规则的出现顺序对换，执行的结果仍然相同。它是一阶谓词逻辑中 Horn 子句（Horn clause）逻辑的一种受限形式，只允许变量或常量作为谓词的自变元，不允许函数作为谓词的自变元。

Datalog 的语句由事实（fact）和规则（rule）组成，其中事实是关于这个世界的某个断言（assertion），如“张三是李四的父亲”，而规则是那么允许我们从其他事实中推断事实的句子（sentence），如“如果甲是乙的父亲，乙是丙的父亲，那么甲是丙的祖父”。规则中通常包含变量，如上面例子中的“甲”“乙”“丙”就是变量。这些事实和规则都属于知识（knowledge）的一种特殊形式。[17]

Datalog 的事实和规则都可以采用 Horn 子句的一般形式来表示：

$$L_0 :- L_1, ..., L_n$$

其中，L_i 是原子（atom）$p_i(t_1, ..., t_{ki})$的一个表示，p_i 是一个谓词符号（predicate symbol），t_j 表示一个或为常量（constant）或为变量（variable）的参数（term）。一条 Datalog 语句中，“L_0”被称为规则头（head），“:-”是蕴含符号（implication），“$L_1, ..., L_n$”被称为规则体（body）。Datalog 的逻辑推导规则意味着当规则体为真时，规则头亦为真。一个规则体为空的 Horn 子句可以表示一个事实。

因此，在上面的例子中，事实“张三是李四的父亲”就可以被表示为：

父亲(李四, 张三)；

规则“如果甲是乙的父亲，乙是丙的父亲，那么甲是丙的祖父”则可以被表示为：

祖父(丙, 甲) :- 父亲(乙, 甲), 父亲(丙, 乙)。

2.2 基本概念

在修改模板的提取模块中，我们采用了基于 Datalog 的模板表示形式。这一部分主要介绍修改实例的 Datalog 表示中涉及到的一些概念。

2.2.1 节点

节点（Node）指代码的抽象语法树节点，用 Datalog 表示为 V_i，其中节点编号 i 和节点构成一一对应关系。

2.2.2 节点属性

节点属性（node attribute）共有四类，包括节点类型（node type）、值类型（value type）、字符串和函数接口。

节点类型指的是代码的抽象语法树中定义的节点类型，如返回语句（return statement）、赋值（assignment）、函数声明（method declaration）、函数调用（method invocation）等，用 Datalog 表示为 `NType(`V_i`, name)`，表示节点 V_i 的节点类型是 `name`。

值类型是指通过静态分析获得的表达式的取值类型，包括整型 int、浮点型 float、字符串 String 等。值类型的 Datalog 表示为 `VType(`V_i`, type)`，表示以节点 V_i 为根节点的表达式的取值类型为 `type`。

字符串是代码经过格式化之后的字符串表示形式，其 Datalog 的表示形式为 `Content(`V_i`, string)`，表示以节点 V_i 为根节点的语法树所表示的代码字符串为 `string`。

函数接口是单独提取出来的一类代码属性，仅针对函数调用类型相关的节点，如 MethodInvocation、SuperMethodInvocation，用 Datalog 表示为 `AttrAPI(`V_i`, api)`，表示以节点 V_i 为根节点的代码的 API 为 `api`。

2.2.3 边/节点关系

边/节点关系共包含三类，分别是父子关系、祖先关系和数据依赖关系。

父子关系表示抽象语法树中父节点与子节点之间的关系，用 Datalog 表示为

Parent(V_i, V_j)，表示节点 V_i 是节点 V_j 的父亲。

祖先关系是父子关系的传递（不包含父子关系），因为这一类关系太多，所以暂时并没有用 Datalog 进行输出表示，但在实际程序的匹配中，祖先关系也被纳入了考虑，以保证代码结构。

数据依赖关系定义为函数内的定值（definition）依赖，仅针对变量，其 Datalog 的表示形式为 Datadep(V_i, V_j)，表示节点 V_i 的数据依赖于节点 V_j，即节点 V_i 处使用的变量由节点 V_j 通过赋值语句、变量定义、参数传递等方式定义。

2.3 模板表示及提取过程

基于 Datalog，我们给出了修改模板的一些基本概念。一个修改模板由节点集合 U、边集合 R 以及在 U 中某些节点上的代码操作集合 M 组成。我们认为这些基本概念构成了补丁的属性，在用 Datalog 的模板表示形式中，我们使用“+”表示对这个属性的保留，使用“–”表示对这个属性的抽象。

修改模板的提取过程如下：

1、首先将 U、R、M 置为空集合；

2、将补丁实例修改前后的代码抽象为 AST，通过对比 AST 提取修改操作构成的修改集合 M，其中，修改操作包括插入（insertion）、删除（deletion）、替换（replacement）等；

3、根据 M 中的修改，定位被修改代码的节点，将被修改代码节点及其父节点加入到集合 U 当中；

4、对定位到的修改代码进行上下文的扩展。扩展规则如下：遍历 U 中的所有节点，将与该节点具有父子关系和数据依赖关系的相邻节点加入到节点集合 U 中。根据预定义的扩展层数，重复应用扩展规则，从内到外迭代，逐层扩展。

5、按照一定策略，对 U 中的每个节点属性进行抽象。根据上一节的定义，考虑如下四类节点属性：

- 节点类型：针对以下三类节点，该属性保留，不进行抽象：被修改节点的父节点[1]、Statement 类型节点、个别特殊类型节点[2]。其他节点该属性

[1] 保留被修改节点的父节点是为了保证代码修改可以正确适配。

[2] 保留特殊类型节点可以避免在匹配阶段出现明显错误，如类型 MType 只能匹配 MType 的节点，而不能匹配其他的表达式如变量等。

均需要被抽象。

- 值类型：根据开源代码中出现的频繁程度决定是否保留。
- 字符串：根据开源代码中出现的频繁程度决定是否保留。目前仅加入了变量出现的频繁程度的考虑，暂时没有对复杂表达式进行统计。
- 函数接口：保留所有函数接口，适配函数签名修改的场景。

6、将 U 中任意两个节点之间的边加入到集合 R 中，目前尚未对 R 中的边做进一步优化处理。

2.4 一个基于 Datalog 的修改模板实例

结合先前的动机实例，让我们查看一个基于 Datalog 的修改模板。

```
-Node
-NType(V1, METHDECL)
-API(V1, SELF.getPendingMutations(ABSTRACT))

-Node
-Content(V2, getPendingMutations)
-NType(V2, SNAME)
-VType(V2, List)

+Node
-Content(V3, {return pendingMutations;})
+NType(V3, BLOCK)

+Node
-Content(V4, return pendingMutations;)
+NType(V4, RETURN)

+Node
-Content(V5, pendingMutations)
-NType(V5, SNAME)
+VType(V5, LinkedList)

-Parent(V1, V2)
-Parent(V1, V3)
+Parent(V3, V4)
+Parent(V4, V5)
```

图4.2 基于Datalog的模板实例展示

如图 4.2 所示，这是第二章中提供的补丁抽象出来的 Datalog 模板。本文方法统一采用“+”“–”表示对某一节点或节点属性的保留或抽象。由这个模板可见，先前的动机实例的 AST 共有五个节点，其中与补丁修改相关的是 V4 节点。

与此同时，这个模板可以结合补丁的其他语义信息被抽象成一颗树，如图 4.3 所示，在这个例子中，根节点和函数体（body）节点呈现为父子关系，函数体是一个块状域对象（block statement），其中又包含着返回域（return statement），返回域下有一个表达式（expression）为返回值 `pendingMutations`。除此之外，每个节点下面还有一些属性，包括 id、include、attrs、position 等。

id 是这个节点在整棵树中的唯一标识符，如根节点的 id 为 1。

include 表明这个节点本身或节点属性是否应该在模板中被包含。如①所示，include 节点的值是 bool 类型，False 表示不包含，节点应该被抽象。这个节点用 Datalog 的模板表示形式表示为`-Node(V1)`。

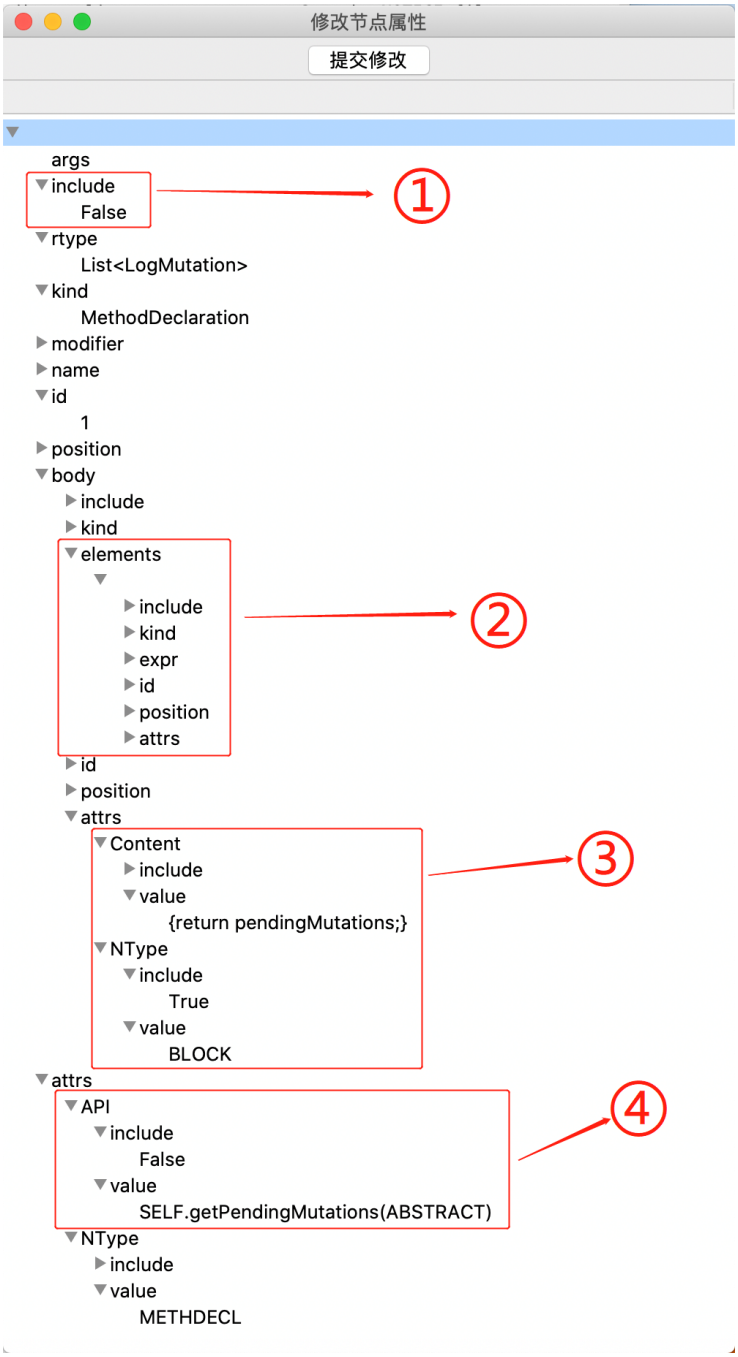

图4.3 模板实例展示

attrs 表示节点属性，共有四类，分别是 VType、Content、NType、API，如图 4.2 中③和④所示。每个节点下面均有两个子节点，分别是 include 和 value，

表示当前属性是否应该被包含和当前属性的值。

position 表示当前节点所在的位置，共有四个子节点，分别是 endLine、endColumn、startColumn 和 startLine。该设计与模板本身的表示关联不大，因此不给予详细介绍。

3. 对模板的人工修改办法

由上一节所描述的，基于 Datalog 的模板表示形式，通过“+”和“–”的表示形式，为使用者提供了人工修改模板的方法。

本文方法一共提供了两种对模板的修改办法，其中一种办法是输入通过形如图 4.2 中的 Datalog 模板表示形式，对已有的节点、节点属性和节点关系进行修改。这种方法会根据一个正则表达式对用户的输入进行匹配和解析，并将格式正确的输入添加到或覆盖掉原来的模板。用来匹配的正则表达式如下：

```
(-|\\+)\\s+\\d+(\\s+(NType|Content|VType|API))?
```

表示增加（+）或删除（–）编号为“\\d+”的节点或节点属性。节点属性包括 NType、Content、VType 和 API。

另一种办法是在模板的树状图上，对 include 值进行修改。include 属性是一个 bool 类型的变量，True 表示包含该节点，False 表示在程序变换中当前节点需要被抽象。通过对布尔类型值的改变，使用者可以进行模板的手动修改。

使用“+”和“–”的表示方式与 SmPL 的程序变换语言有些类似。但二者的不同之处在于，SmPL 中，“+”“–”表示了补丁中具体代码的修改，而我们将这种修改从具体代码迁移到了代码抽象结构的节点属性上，这样的设计使得本文方法的通用性更高，语言表达能力更强。

4. 工具实现

基于上述思路，本文以 Java 代码的缺陷检测和修复为目标，设计并实现了如下的 GUI 工具。该工具使用 Python 作为前端用户图形界面工具，复用了先前的 GenPat 技术作为补丁泛化、模式匹配和修改的后端。

4.1 GUI 工具介绍

前端用 Python 编写而成，图形界面调用了 tkinter 库。工具主要包括三个部分，分别是补丁文件的导入、模板的提取和修改、缺陷检测和修复。具体架构如下图所示。

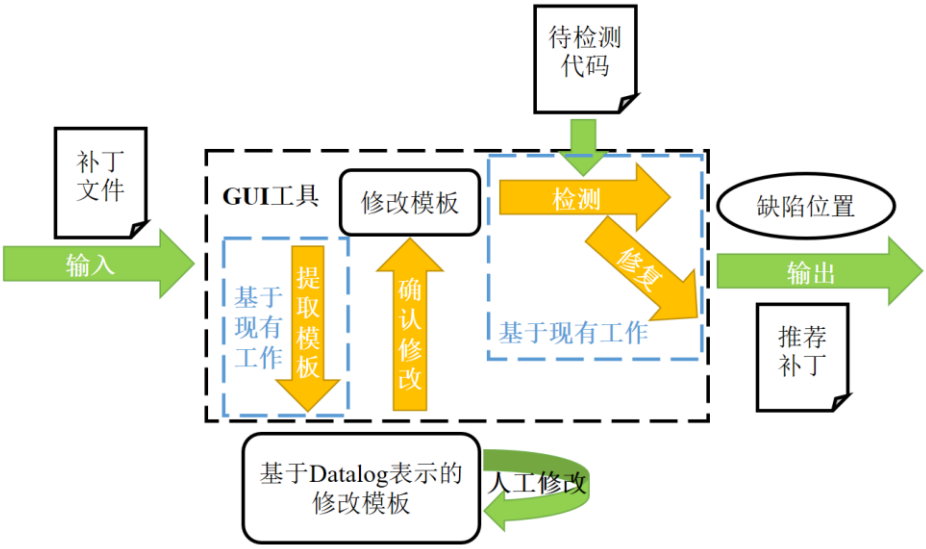

图4.4 工具架构

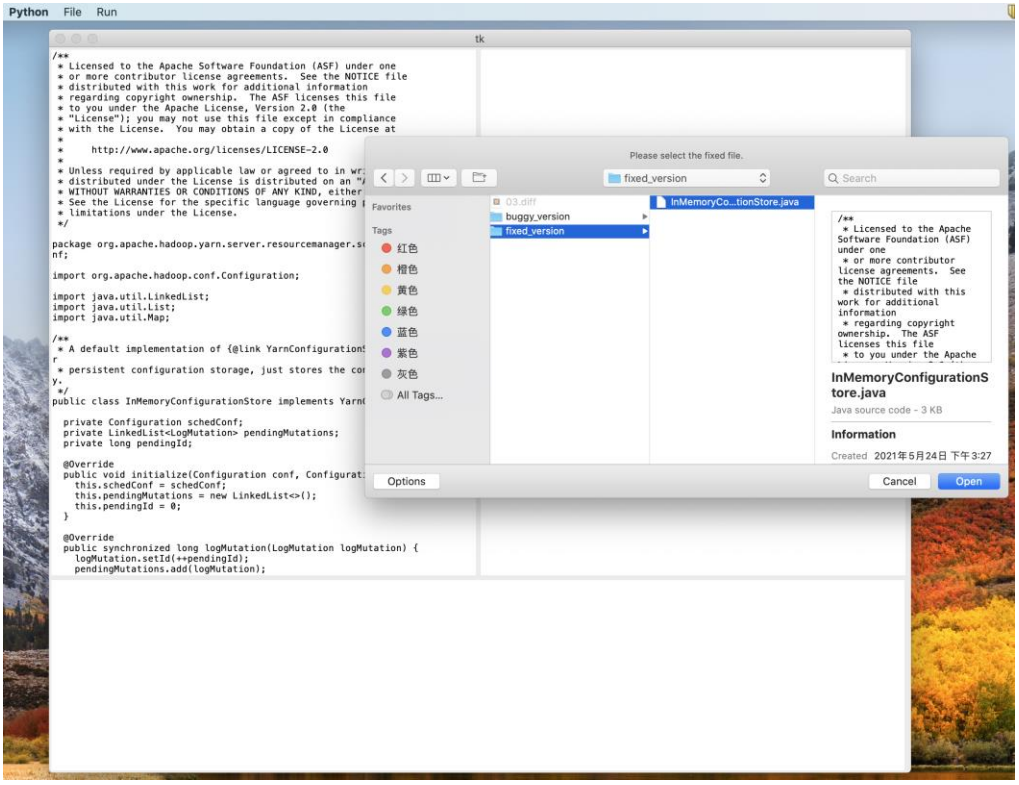

图4.5 补丁文件导入界面

补丁文件导入部分采用了 tkinter 库中的 filedialog 模块来实现，通过调用 askopenfilename()方法，分别导入有缺陷文件和修复文件，并在内存中记录两个相关文件的路径。

模板提取部分通过 jpype 库启动了载有 GenPat 工具 jar 包的 JVM，将两个文件的路径传给 GenPat 工具，执行模板提取工作。前端会将后端返回的模板的 Datalog 表示通过文本框展示给用户，同时会将后端返回 json 格式的模板可视化成一棵树。前端 GUI 工具通过遍历 json 模板，借助 tkinter 库中的 ttk.Treeview 模块将所有节点和属性可视化。效果如图 4.2 和 4.3 所示。

模板的每一个节点包括了一个 bool 类型的名为 include 的属性，这个属性意味着是否要在模板中保留这个具体内容，值为 True 时表示保留，False 表示抽象。本文工具设计了用户可以通过点击直接改变 include 属性的状态，从而实现对模板的修改。在完成对模板的修改之后，点击“提交修改”，可以将模板存储到本地的 pattern 文件中。

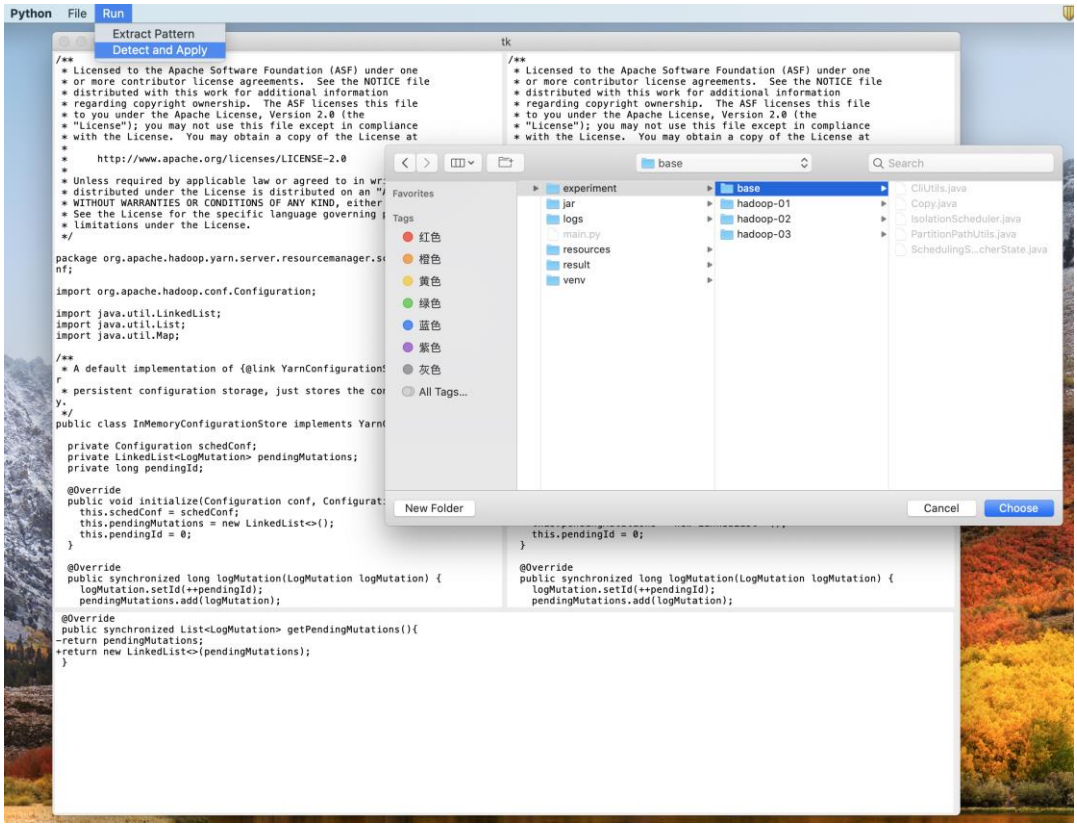

图4.6 选取待检测项目路径

在第三个部分中，用户可以通过选择菜单栏中的“Run - Detect and Apply”，在固定的开源项目路径下用上一步中修改好的模板进行检测修复。前端采用

filedialog 模块下的 askdirectory()方法询问使用者的待检测路径，将获取的路径信息通过名为 `repo_path` 的临时变量传递给后端。后端程序运行完成后，前端会通过 messagebox 模块下的 showinfo()方法通知用户检测完成，可以在对应路径下查看检测结果和推荐补丁。

4.2 GenPat 工具复用说明

先前的 GenPat 工具利用超图完成了基于单样例的程序变换推导，过程中采用代码大数据指导节点的抽象和保留。整个工作通过超图的抽象和匹配，将所有流程耦合在了一起，没有为使用者提供人工修改的空间和余地。本文对先前的 GenPat 工作进行了改造，将本来耦合在一起的各个模块进行了解耦操作，通过 maven 的打包，生成了两个 jar 包，共向前端提供了三种服务，包括：修改提取、缺陷挖掘和模板适配。

其中，修改提取部分对应 GenPat 工具下的 mfix.core.pattern.PatternExtractor 模块。后端工具首先通过 Eclipse JDT 提供的 ASTPaster 模块，将补丁修改前后的代码解析成抽象语法树，并通过遍历整个 AST 将所有节点属性补全。随后，GenPat 工具通过 GumTree 算法将两个 AST 进行对比，标记出修改的节点，并通过 mfix.core.node.ast.Node 模块中的 doAbstraction 方法将模板中不重要的属性进行抽象。后端同时会通过 diff 格式的文件，向前端提供补丁修改前后代码文本上的异同。

缺陷挖掘部分对应 mfix.core.node.match.DetectorMatcher 模块，通过第二章介绍的相似度的计算进行匹配操作。在最后模板适配部分，GenPat 工具根据不同的节点类型——如函数调用（method declaration）、返回域（return statement）等——对 mfix.core.node.ast.Node.adpatModifications 接口的实现，根据被检测代码的上下文语义信息，将抽象的语法树结构重新转换为具体的代码，生成推荐的修复补丁。

4.3 工具演示

回顾第二章的动机实例，我们发现大量的假阳性错误来源于对返回值类型的约束条件太松。

下图是 GenPat 工具提取出来的模板，未经人工修改。

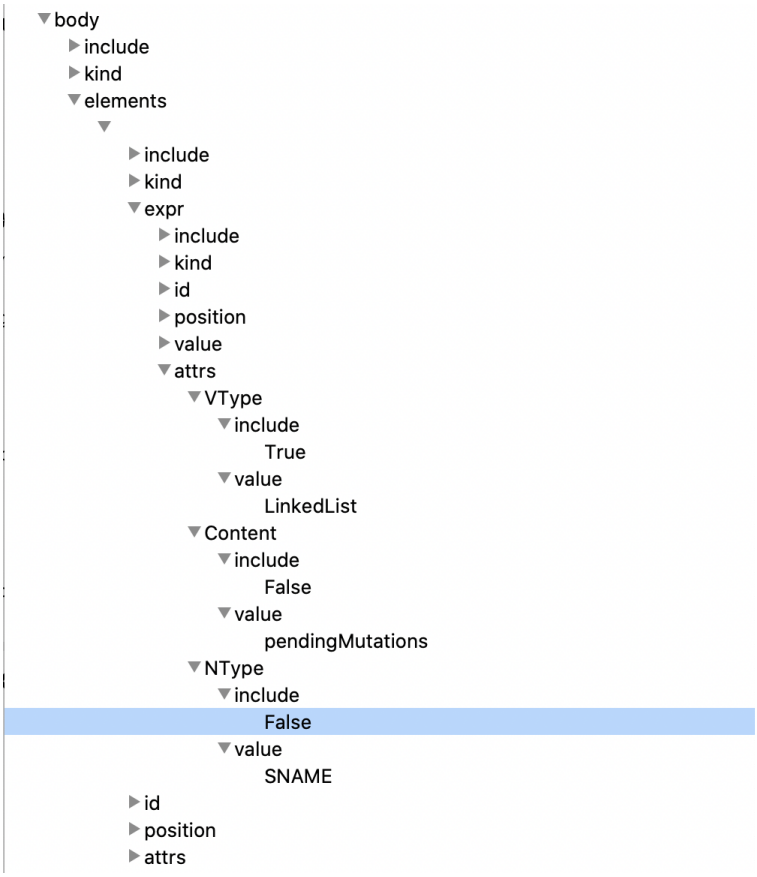

图4.7 模板节点修改示例

可以发现，返回值 `pendingMutations` 共有三个属性特征，其中，返回值类型为 LinkedList，返回值内容为 pendingMutations，返回值节点类型为 SNAME（即 simple name），而这三个属性只有链表类型被考虑进去了。回想第二章的三个假阳性结果，前两个例子的返回值类型都是链表，第三个例子的返回值是 null，也可以与链表类型正确匹配。因此，仅考虑返回值类型是不够的。根据补丁实例的特性，本文将返回值节点类型的属性也纳入了考虑，通过点击 include 下的布尔值，将 NType 属性选为保留，最终过滤掉了 97.9%的假阳性结果。

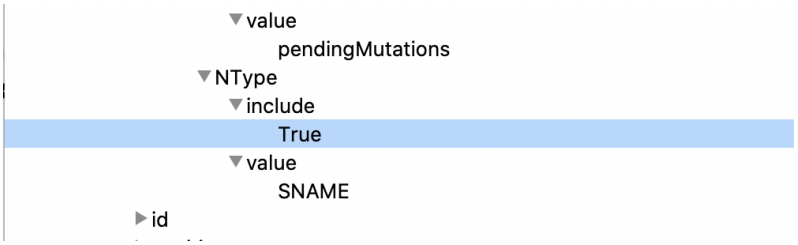

图4.8 修改节点属性值后的模板

第五章 实验设计与结果

为了验证引入人工修改之后的有效性，本文设计了如下的实验。通过收集基准数据并进行实验，本文从准确率对比和修改人力估计两个方面对上述设计方法进行了评测。与此同时，本文还利用工具，在帮助开源项目管理者进行缺陷检测和修复方面展开了实验。这一部分将详细介绍实验的设计与配置、获得的结果及其分析。

1. 实验设计与配置

首先，对原有的 GenPat 工具效果进行了实证性的评估，尝试探索 GenPat 工具在不同补丁类型上的检测效果。实验选取了 4 个最新的 Apache 开源项目：Flink、Storm、Samza 和 Hive，以及 2 个 Microsoft 的开源 Java 项目：appcenter-sdk-android 和 azure-tools-for-java。本文在开源项目中随机抽取了 146 个补丁，并手动过滤掉了 81 个不符合缺陷修复条件的补丁，构成实验的数据集。

其次，实验在上述 4 个 Apache 的开源项目上，利用筛选得到的 65 个补丁实例，对工具的有效性进行了验证。有效性的评价指标包括准确率（precision）和修改人力估计。准确率是指被正确修复的缺陷占所有被修复缺陷的比例，其中正确修复的缺陷个数由人工确定，所有被修复缺陷的个数以工具最终提供给使用者的补丁个数为准。修改人力估计是指对使用本文工具前后人力投入影响的估计，以修改时间为主。

最后，实验还验证了工具的实用性。本文一共选取了 12 个补丁修改实例，在 8 个开源项目上进行了缺陷的检测和修复。这 12 个补丁修改实例的选取一方面参考了 X. Li 等人[3]的工作，另一方面也借鉴了前两个实验的结果，选取了在实验中得到检测结果的几个补丁，本文认为这样几个补丁的通用性更高。

2. 实验结果

2.1 原有工具效果实证

这部分实验对原有的 GenPat 工具效果进行了实证性的评估。参考先前工作，本文发现 GenPat 工具的效果主要受到三个方面的影响，包括：

- 数据集自身有噪点，影响了工具的表现
- 存在一些先前工作无法支持的补丁修改类型
- 部分假阳性结果来自于实现方法有错误

本文选取了 4 个最新的 Apache 开源项目： Flink、Storm、Samza 和 Hive，以及 2 个 Microsoft 的开源 Java 项目：appcenter-sdk-android 和 azure-tools-for-java，并从这些开源项目中随机抽取了 65 个补丁。结果如下：

表5.1 使用本文工具前的准确率结果

开源项目名称	检测		修复	
	正确数	总数	正确数	总数
Flink	4	341	4	4
Storm	1	4	1	1
Samza	6	19	6	6
Hive	19	83	19	19
appcenter-sdk-android	0	162	0	0
azure-tools-for-java	0	546	0	0
总计	30	1155	30	30

2.2 有效性实验

实验在上述 4 个 Apache 的开源项目上，利用筛选得到的 65 个补丁实例，对工具的有效性进行了验证。实验结果分两方面呈现，一方面如表 5.2 呈现的，在

上述数据集上使用工具后进行检测和修复的准确率，另一方面是计算使用工具增加或减少的人力时间的估计。

目前的实验结果如下：

表5.2 使用本文工具后的准确率结果

开源项目名称	检测		修复	
	正确数	总数	正确数	总数
Flink	5	7	5	5
Storm	1	1	1	1
Samza	11	12	11	11
Hive	19	26	19	19
总计	36	46	36	36

在 65 个补丁实例中，检测到缺陷的补丁共有 14 个。假设开发者对每个生成的补丁需要花费 20s 的时间来判断它的正确性，对一个补丁实例模板的修改需要花费 3min 的时间，那么使用本文工具前，进行一次补丁正确性判断大约耗时为

$$1155 \times 20s = 385min$$

使用本文工具后，进行一次补丁正确性判断大约耗时为

$$46 \times 20s + 14 \times 3min = 57.3min$$

人力在实验数据集上节约了近 330 分钟的时间，开发速度提升了约 6.7 倍。

2.3 实用性实验

这一部分的实验筛选了 12 个较为通用的补丁修改实例，在 8 个开源项目进行了检测和修复。经人工验证，目前工具已经检测到了 71 个有效补丁并在 GitHub 上提交给了开源项目的开发者，其中 46 个补丁已经被开发者接受，接受率为 64.8%。已提交补丁的 GitHub PR 链接见附录 A。

3. 结果总结与分析

我们的工具在缺陷检测的场景下取得了良好的效果。原先的 GenPat 工作在缺陷检测和修复场景下有大量假阳性错误。本文的方法通过引入人工修改，消除了 99%的错误结果，将原先的准确率从 2.6%提升到了 78.3%。而自动化工具并没有增加更多的人工负担，反而减少了近 7 倍的补丁筛选和人工修复时间。此外，本文的工具在帮助开发者进行开源项目开发场景下也表现良好。实验共选取了 12 个修改实例进行检测，最后共在 5 个开源项目上提交了 71 个有效补丁，其中有 46 个补丁已经被开发者接受。

但本文方法在一些补丁的检测上依旧不能达到 100%的准确率，并且在实用性测试中发现了一个错误的修复补丁。经过分析，发现本文的方法存在如下几个缺陷：

- 用以检测的补丁质量通用性较差，只适用于特定项目；
- 程序变换语言表达能力有限，无法将某些补丁独有的语义加入用节点属性表达；
- 前端程序的可交互性能力有限，影响了用户对模板的修改效率和体验；
- 后端程序在实现上有一些非结构性错误，影响了最后补丁的生成质量。

在工具性能的通用测试上，本文发现该工具在 API 误用缺陷检测等特定场景下具有更高的准确率，而针对插入式修改的补丁输入，则更依赖人工修改来修正最后的结果。因为 API 误用缺陷修复补丁本身包含了更多语义信息，而插入式语句相比于前者受到更弱的上下文的约束和代码大数据的抽象指导，从而需要使用者手动将某些关键的节点信息加入模板，进行自动化匹配和修改。

第六章 总结与展望

1. 现有工作总结

本文基于先前的 GenPat 工作，设计了基于 Datalog 的可交互的模板表示形式，并开发了供使用者进行交互操作的用户图形界面工具。通过复用现有技术，本文实现了基于单个补丁实例进行缺陷检测和修复，优化了先前工作 GenPat 出现大量假阳性结果的不足，并在实际开发中取得了一定效果。

2. 未来工作展望

目前的图形界面工具只实现了一个比较简单的界面，补丁修改前后的代码呈现也处于比较苍白的阶段。同时，当前的工具无法向用户呈现一个更为清晰的可视化修改，因其还未实现抽象模板节点和对应代码之间的动态交互，即无法在用户对节点进行修改时，在代码对应位置进行高亮和提示。

除此之外，本文提出的方法，虽然在通用的缺陷检测和修复场景上对比 GenPat 有更为良好的表现效果，但依旧缺乏与更多现有技术的横向对比。在工具性能的通用测试上，本文发现该工具在 API 误用缺陷检测等特定场景下具有更高的准确率，而针对插入式修改的补丁输入，则更依赖人工修改来修正最后的结果。这一现象要求本文在未来工作中寻找新的自动化方法，解决上述问题。

参考文献

[1] S. Amann, H. A. Nguyen, S. Nadi, *et al.*, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE Transactions on Software Engineering*, 2018.

[2] M. Monperrus and M. Mezini, "Detecting Missing Method Calls As Violations of the Majority Rule," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2013, vol. 22, no. 1: pp. 1-25.

[3] X. Li, J. Jiang, S. Benton, *et al.*, "A Large-scale Study on API Misuses in the Wild," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST' 21, 2021.

[4] Y. Acar, M. Backes, S. Fahl, *et al.*, "You Get Where You're Looking For. The Impact of Information Sources on Code Security," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2016.

[5] J. Jiang, L. Ren, Y. Xiong, *et al.*, "Inferring Program Transformations From Singular Examples via Big Code," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE' 19, IEEE Computer Society Press, 2019.

[6] M. Wen, Y. Liu, R. Wu, *et al.*, "Exposing Library API Misuses via Mutation Analysis," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*, ser. ICSE' 19, 2019.

[7] H. Zhong, L. Zhang, T. Xie, *et al.*, "Inferring Resource Specifications from Natural Language API Documentation," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE' 09, IEEE Computer Society, 2009: pp. 307-318.

[8] T. Britton, L. Jeng, G. Carver, *et al.*, "Reversible Debugging Software: Quantify the Time and Cost Saved Using Reversible Debuggers," 2013.

[9] Q. Gao, Y. Xiong, Y. Mi, *et al.*, "Safe Memory-Leak Fixing for C Programs," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, ser. ICSE' 15, 2015.

[10] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Transactions on Software Engineering*, 2018, vol. 46, no. 10: pp. 1040-1067.

[11] A. Marginean, J. Bader, S. Chandra, *et al.*, "Sapfix: Automated End-to-End Repair at Scale," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP, IEEE, 2019: pp. 269-278.

[12] J. Jiang, J. Chen, and Y. Xiong, "Survey of Automatic Software Repair Techniques," *Journal of Software*, 2021 (in Chinese).

[13] S. Mechtaev, J. Yi, A. Roychoudhury, "Directfix: Looking for Simple Program Repairs," in

Proceedings of the 37th IEEE International Conference on Software Engineering, ser. ICSE, IEEE, 2015, vol. 1: pp. 448-458.

[14] F. Li, R. Fergus, and P. Perona, "A Bayesian approach to unsupervised one-shot learning of object categories," in *Proceedings of the 9th IEEE International Conference on Computer Vision*, ser. ICCV' 03, IEEE, 2003: pp. 1134-1141.

[15] K. Zhao, X. Jin, and Y. Wang, "Survey on Few-shot Learning," *Journal of Software*, 2021, vol. 32, no. 2: pp. 349-369 (in Chinese).

[16] J. Falleri, F. Morandat, X. Blanc, *et al.*, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE' 14, 2014: pp. 313-324.

[17] S. Ceri, G. Gottlob, and L. Tanca, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Transactions on Knowledge and Data Engineering*, 1989, vol. 1, no. 1: pp. 146-166.

[18] Wikipedia, "Datalog", Available: https://en.wikipedia.org/wiki/Datalog.

[19] S. Huang, T. Green, and B. Loo, "Datalog and Emerging Applications: An Interactive Tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD' 11, Association for Computing Machinery, 2011: pp. 1213-1216.

[20] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE' 09, IEEE Computer Society, 2009: pp. 309-319.

[21] Q. Gao, H. Zhang, J. Wang, *et al.*, "Fixing Recurring Crash Bugs via Analyzing Q&A Sites," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE' 15, pp: 307-318.

[22] J. Andersen and J. L. Lawall, "Generic patch inference," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE' 08, 2008: pp. 337-346.

[23] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning Development Tools with the Way Programmers Think About Code Changes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ser. CHI' 70, 2007: pp. 567-576.

[24] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, 2006, vol. 61, no. 3: pp. 190-210.

[25] H. A. Nguyen, T. T. Nguyen, G. W. Jr., *et al.*, "A Graph-based Approach to API Usage Adaptation," in *Proceedings of the 2010 ACM SIGPLN International Conference on Systems, Programming, Languages and Applications*, ser. OOPSLA' 10, 2010.

[26] T. T. Nguyen, H. A. Nguyen, N. H. Pham, *et al.*, "Recurring Bug Fixes in Object-Oriented Programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE' 10: pp. 315-324.

[27] N. Meng, M. Kim, and K. S. McKinley, “Systematic Editing: Generating Program Transformations from an Example,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI’ 11, 2011: pp. 329-342.

[28] B. Ray and M. Kim, “A Case Study of Cross-System Porting in Forked Projects,” in *Proceedings of the 9th European software engineering conference held jointly with 12nd ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE’ 12, 2012.

[29] R. Rolim, G. Soares, L. D. Antoni, *et al.*, “Learning Syntactic Program Transformations from Examples,” in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*, ser. ICSE’ 17, 2017.

[30] F. Long, P. Amidon, and M. Rinard, “Automatic Inference of Code Transforms for Patch Generation,” in *Proceedings of the 14th European software engineering conference held jointly with 17th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE’ 17, 2017.

[31] J. Bader, A. Scott, M. Pradel, *et al.*, “Getafix: Learning to Fix Bugs Automatically,” in *Proceedings of the 2019 ACM SIGPLN International Conference on Systems, Programming, Languages and Applications*, ser. OOPSLA’ 19, 2019.

[32] E. Visser, “Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5,” in *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 2051, Springer, Berlin, 2001.

[33] L. Qiu, X. Liang, and Z. Huang, “PATL: A RFID Tag Localization based on Phased Array Antenna,” *Scientific Report*, vol. 7, 2017.

[34] M. Nita and D. Notkin. “Using Twinning to Adapt Programs to Alternative APIs,” in *Proceedings of the 32nd IEEE/ACM International Conference on Software Engineering*, ser. ICSE’ 10, 2010.

[35] J. Li, C. Wang, Y. Xiong, et al., “SWIN: Towards Type-Safe Java Program Adaptation Between APIs,” in *Proceedings of the 2015 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM’ 15, 2015.

[36] N. Meng, M. Kim, and K. S. McKinley, “LASE: Locating and Applying Systematic Edits by Learning from Examples,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE’ 13, IEEE Press, 2013: pp. 502–511.

附录

以下是已提交补丁的 GitHub PR 链接：

- https://github.com/apache/flink/pull/15438
- https://github.com/apache/storm/pull/3393
- https://github.com/apache/hive/pull/2209
- https://github.com/apache/storm/pull/3388
- https://github.com/apache/hive/pull/2127
- https://github.com/apache/hadoop/pull/2803
- https://github.com/apache/hadoop/pull/2804
- https://github.com/apache/hadoop/pull/2805
- https://github.com/apache/hadoop/pull/2810

致谢

感谢熊老师三年来的指导和帮助！您极强的科研能力和严谨的学术态度，让我得以一窥软工前沿研究的门槛与魅力。而与您的多次讨论则让我受益匪浅，谢谢您解答了我诸多对于科研课题和人生选择的困惑。在熊老师的指导下，我才独立地完成了这次研究。

感谢天津大学的姜佳君老师对这篇工作的全力支持，谢谢您无条件地帮助我面对这个课题的许多困难，耐心而细心地解答了我对这个课题的诸多问题。感谢任路遥师兄，带领我展开这个领域的科研。感谢实验室的其他师兄师姐们对我友善的帮助。感谢软件工程研究所和程序设计语言实验室为我提供了一个浓厚的学术氛围！

感谢北大这个园子，给了我恣意生长的空间，在学业与学业之间，还让我有机会加入外国戏剧研究所、女子篮球校队和院队、校冰壶队和五四文学社。感谢这些社团的小伙伴们，没有你们我精彩的大学生活将黯然失色。

感谢我的舍友，这四年里我们一起成长。感谢我的高中同学刘畅、陈立玮和李天益，你们在我下坠的时候接住我，给了我最温暖的支持和帮助。感谢这四年间所有遇见的人，谢谢你们让我看到这个世界的精彩和美好！

感谢我的家人们，尤其感谢我的父母，你们承受了我青春期的叛逆和决绝，也共享了我的荣誉与忧愁。没有你们就不会有现在的我，谢谢你们给了我全部的爱！

感谢我自己，在许许多多困难的时刻咬牙坚持了下来。人生本就是一场漫长而苦难的修行，希望能再接再厉，不要辜负遇见的每一个人。